Honeywell

ASSEMBLY LANGUAGE

SERIES 60 (LEVEL 6)

GCOS/BES2

SOFTWARE

Honeywell

ASSEMBLY LANGUAGE

SERIES 60 (LEVEL 6)

GCOS/BES2

SUBJECT:

Detailed Description of Series 60 (Level 6) GCOS/Basic Executive System 2 Assembly Language

SOFTWARE SUPPORTED:

This publication supports Release 0200 of the Series 60 (Level 6) GCOS/Basic Executive System 2. When a later release of the system occurs, see the Subject Directory of the latest *Series 60 (Level 6) GCOS/BES2 Software Overview and System Conventions* manual (Order No. AU50) to ascertain whether this revision of this manual supports that release.

DATE:

July 1976

ORDER NUMBER:

AU43, Rev. 0

PREFACE

This manual describes the GCOS/Basic Executive System 2 (GCOS/BES2) assembly language, a machine-oriented language for writing programs to execute upon the Series 60 (Level 6) 6/30 Models. Unless stated otherwise herein, the term BES will be used to refer to the GCOS/BES2 software; the term Level 6 will indicate the specific models of Series 60 (Level 6) on which the described software executes.

Where appropriate, the actions performed by the BES Assembler as it processes elements of the assembly language are also discussed. Within this document, the term "assembly language" includes both Assembler control statements and assembly language instructions.

Section 1 introduces the Level 6 describing both data representation and the hardware registers. Section 2 describes the basic elements of the BES assembly language, and Section 3 describes the considerations the programmer must make when writing his source program. Sections 4 and 5 describe, in detail, the Assembler control statements and assembly language instructions, respectively. The macro facility is described in Section 6. Appendix A provides programmer reference information. Appendix B describes the hexadecimal numbering system. Appendix C contains a sample assembly language program. Appendix D describes how to debug an assembly language program. Appendix E contains a list of flags produced by the assembler to notify the user of a source code error. Appendix F describes error flags that may be issued by the Macro Preprocessor. Appendix G contains a list of reserved symbolic names.

Descriptions and examples within this manual use the following conventions:

{ }	Indicates that one of the options enclosed in the braces must be selected.
[ ]	Indicates that one or none of the enclosed options need be selected; if one of the options is underlined, it is selected as the default if you do not select any of the options enclosed in the brackets.
...	Indicates either a logical sequence (e.g., A,B...) or that the immediately preceding type of value can be repeated (e.g., a...).
a	Indicates that the character must be replaced by any valid ASCII character.
n	Indicates that the character must be replaced by any valid numeric (decimal) digit.
d	Indicates that the character must be replaced with a binary digit.
h	Indicates that the character must be replaced with a hexadecimal digit (0 through 9, A through F).
c	Indicates that the character must be replaced with a, n, or h, above.
Δ	Indicates that one or more spaces or horizontal tab characters are required.

Uppercase letters, numbers, and any of the following special characters must be coded exactly as shown:

( )	$
<	.
>	/
=	*
+	,
-	;

GCOS/BES2 Subject Directory

This subject directory is designed to assist the user in finding information about specific topics related to GCOS/BES2. Topics are listed alphabetically; each topic is accompanied by the order number of each manual in which the topic is described. At the end of the Subject Directory, all GCOS/BES2 manuals are listed according to the alphabetic/numeric sequence of their order numbers.

Subject	*Order No.*
Allocate Disk File (Utility Set 1)	AU47
Application Development (Overview)	AU50
ASCII Character Set and Conversion Tables	AU50
Assembling Programs	AU48
Assembler Diagnostic Flags	AU43
Assembly Source Language	AU43
BASIC	AU44
Bootstrap Generator	AU47
Bootstrapping and Loading	AU46
Buffer Manager	AU45
Building an Online Application	AU49
Card Loader	AU46
Clock Manager	AU45
COBOL Compilation	AU48
COBOL Source Language	AU42
COBOL Statements	AU42
COBOL Compiler Diagnostic Messages	AU42
COBOL Operating Procedures	AU46
Command Processor	AU48
Communications	AU45
Compare Disk Volumes/Files/Members (Utility Set 3)	AU47
Configuration Load Manager	AU49
Console Messages (Error and Informational)	AU46
Control Panel	AU46
Copy Disk Volume/File/Member (Utility Set 3)	AU47
Cross-Reference Program	AU48
Debugging (Offline)	AU47
Debugging (Online)	AU49
Delete Disk File/Member (Utility Set 1)	AU47
Disk Conventions	AU50
Disk Loader	AU46
Dumps	AU47
Dump Edit	AU47
Editor	AU48
Equipment Requirements	AU50
Error Reporting by Online Applications	AU46
Error Reporting by System Software	AU46
Executive Components	AU45
File Manager	AU45
File Naming Conventions	AU50
Floating-Point Simulator	AU45
FORTRAN Compilation	AU48
FORTRAN Compiler Diagnostic Messages	AS32
FORTRAN Intrinsic Functions	AS32
FORTRAN Source Language	AS32
FORTRAN Statements and Procedures	AS32
Glossary of System Terms	AU50
Hexadecimal Numbering System	AU43
Initialize Disk Volume/File (Utility Set 1)	AU47
Input/Output Drivers	AU45

Linker	AU48
List Disk Volume/File Description (Utility Set 1)	AU47
Loaders	AU46
Macro Facility Usage	AU43
Macro Preprocessor	AU48
Offline Applications	AU45
Operating Procedures	AU46
Operator Interface Manager	AU45
Overlay Loader	AU45
Paper Tape Loader	AU46
Planning an Online Application	AU49
Print Disk File/Member (Utility Set 2)	AU47
Program Development Tools	AU48
Program Naming Conventions	AU50
Program Patch	AU47
Punch Disk File/Member to Paper Tape (Utility Set 2)	AU47
Rename Disk Volume/File/Member (Utility Set 1)	AU47
Replace Memory Values (Utility Set 1)	AU47
Scientific Branch Simulator	AU45
Software Release Materials (Contents)	AU50
System Conventions	AU50
System Software and Documentation (Overview)	AU50
Task Manager	AU45
Trace Trap Handler	AU45
Transfer Input to Disk File/Member (Utility Set 2)	AU47
Trap Handling (Offline)	AU46
Trap Handling (Online)	AU45
Utility Programs	AU47

The following publications constitute the GCOS/BES2 manual set. The Subject Directory in the latest *Series 60 (Level 6) GCOS/BES2 Software Overview and System Conventions* manual lists the current revision number and addenda (if any) for each manual in the set.

Order No.	*Manual Title*
AS32	Series 60 (Level 6) GCOS/BES FORTRAN Reference Manual
AU41	Series 60 (Level 6) GCOS/BES2 COBOL Reference Manual
AU43	Series 60 (Level 6) GCOS/BES2 Assembly Language Reference Manual
AU44	Series 60 (Level 6) GCOS/BES2 BASIC Reference Manual
AU45	Series 60 (Level 6) GCOS/BES2 Executive and Input/Output
AU46	Series 60 (Level 6) GCOS/BES2 Operator's Guide
AU47	Series 60 (Level 6) GCOS/BES2 Utility Programs
AU48	Series 60 (Level 6) GCOS/BES2 Program Development Tools
AU49	Series 60 (Level 6) GCOS/BES2 Planning and Building an Online Application
AU50	Series 60 (Level 6) GCOS/BES2 Software Overview and System Conventions

In addition to the GCOS/BES2 manual set, the following manual is required by GCOS/BES users as a general hardware reference:

Order No.	*Manual Title*
AS22	Honeywell Level 6 Minicomputer Handbook

The following manual provides detailed information regarding programming for the Multiline Communications Processor:

Order No.	*Manual Title*
AT97	Series 60 (Level 6) MLCP Programmer's Reference Manual

CONTENTS

ILLUSTRATIONS

TABLES

SECTION 1

INTRODUCTION

Computer programs can be written in high-level languages or machine-oriented lower level languages. High-level languages are generally designed for specific environments (e.g., COBOL is a business-oriented language, and FORTRAN is a scientifically-oriented language). Low-level languages (i.e., assembly languages) support a wide range of application environments.

ASSEMBLY LANGUAGES

Computer logic is designed to interpret only machine (i.e., object) code. Since object code is composed of binary digits, it is difficult to interpret unless the binary representation is translated into a more convenient, readable code. As a result, assembly languages have been developed to simplify the problem of writing programs in object code. These intermediate-level assembly languages consist of assembler-controlling statements and operational instructions.

As illustrated in Figure 1-1, an assembler interprets the assembly language (i.e., source code) program and translates it into object code, which the computer executes to produce the desired results.

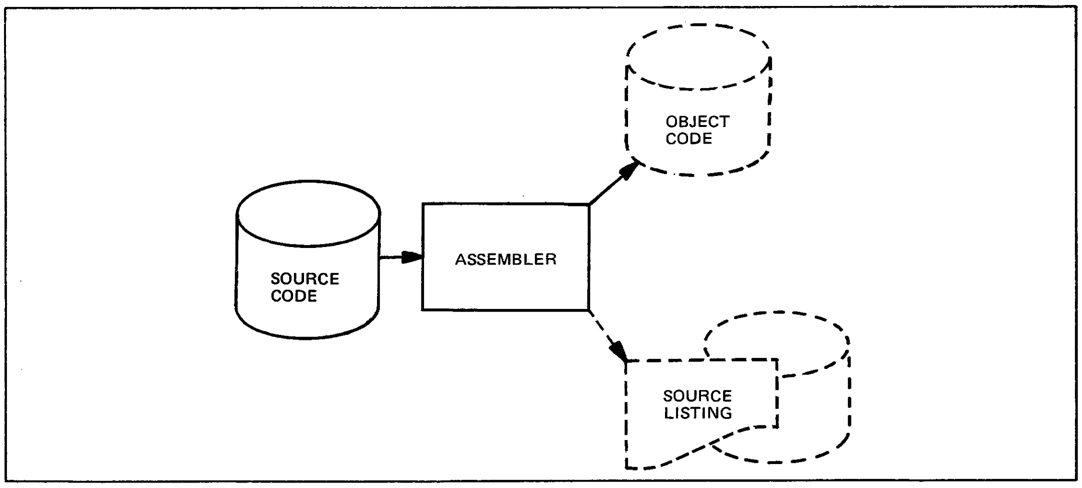


Figure 1-1. Assembler Functions

One of the primary differences between assembly languages and high-level languages is that each assembly language instruction is equivalent to a single machine-level instruction, whereas a single high-level language instruction can be translated into any number of machine-level instructions. The advantage, then, is that the assembly language gives you more control over the operations to be performed.

LEVEL 6 DATA REPRESENTATION

All data stored in main memory must be in predefined, system-recognizable formats. All data elements are based on 16-bit memory words. The format of each word is defined from left to right, with the first bit numbered 0 and the last 15. The leftmost bit (i.e., bit 0) is considered the most significant and the rightmost (i.e., bit 15) is the least significant, with each intervening bit less significant than the one to its left.

Because of this predefined format, it is possible to access data at any of the following levels:

- o Bit - 1 bit
- o Byte (half-word) - 8 bits
- o Word - 16 bits
- o Multiword - 32, 64 bits

Regardless of the size of the data item being accessed, addresses generated by the operand(s) in an instruction point to the most significant bit of the item. For example, to access a multiword data item in main memory, the address generated by the Assembler (from the operand contained in the instruction) points to the first bit (i.e., bit 0) in the first word of the item.

The system supports a maximum of 128K bytes (i.e., 64K words) of addressable memory, and each word can be accessed through a 16-bit address pointer.

Each four bits of data are represented by a single hexadecimal value in a listing or printout, although the bits are stored in memory in binary form. The hexadecimal equivalent of a binary value is derived by converting each successive four bits to the hexadecimal value as follows:

0000 = 0	1000 = 8
0001 = 1	1001 = 9
0010 = 2	1010 = A
0011 = 3	1011 = B
0100 = 4	1100 = C
0101 = 5	1101 = D
0110 = 6	1110 = E
0111 = 7	1111 = F

Thus, if a listing shows that a word at a given address contains the hexadecimal value 8FD3, it means that the system contains the stored binary value 1000111111010011.

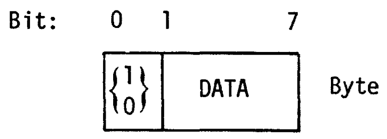
Data stored in memory can be in any of the following forms:

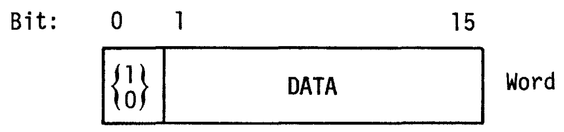
- o Signed integer
- o Unsigned integer
- o Floating-point

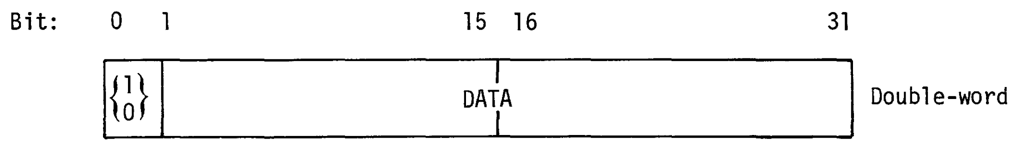
A signed or unsigned integer byte can also be stored in a hardware general register. A floating-point constant occupies two (short-precision) or four (long-precision) memory words and may also be stored in the software-simulated scientific register.

Signed Integer Data

Signed integers stored in memory contain a sign (0 = +; 1 = -) in bit 0 and the data in the remaining bits. Negative numbers appear in twos-complement form. Byte, word, and double-word formats are permitted, as follows:

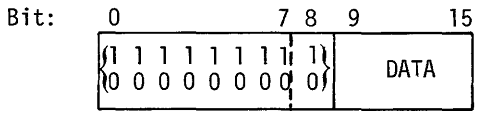






If the first digit in the hexadecimal representation of a signed integer is 0 through 7, the value is positive and is stored in memory exactly as it was coded; if the first digit is 8 through F, the value is negative and is stored in memory as the twos complement of the coded inteter. For example, if the contents of a signed integer word appearing in memory are BDA0, the decimal equivalent is -12640.

When a signed integer byte is loaded from memory into a hardware general register, the seven data bits are placed into bits 9 through 15 of the register and the sign into bit 8. The sign is then extended through bit 0 of the register, as follows:

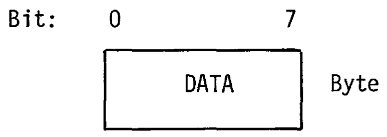


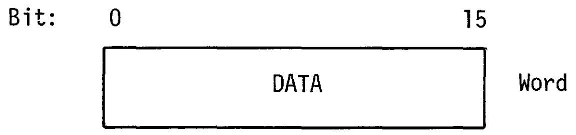
The sign of the integer byte (i.e., the first bit of the 8-bit byte), which is contained in bit 8 of the register, is extended through the first byte of the register.

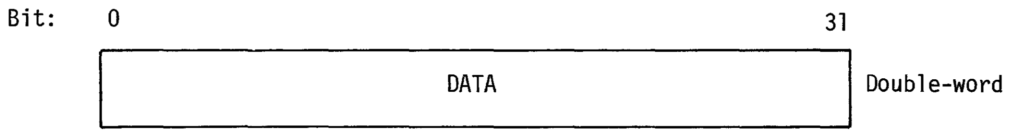
If the first byte of the register contains the hexadecimal value FF, the integer in the second byte is a negative value; if the first byte contains the hexadecimal value 00, the value of the second byte is positive.

Unsigned Data

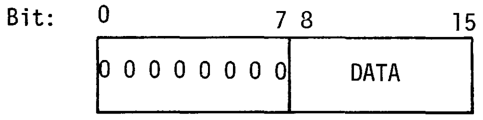
Unsigned data appears in memory in three possible formats:





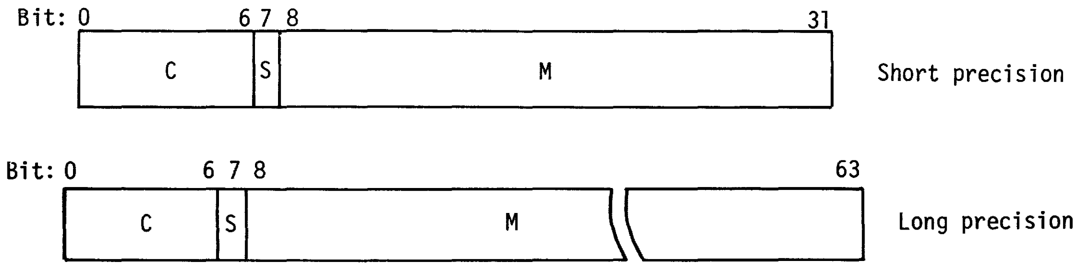


When an unsigned data byte is loaded from memory into a hardware general register, the byte is placed into register bits 8 through 15, and register bits 0 through 7 are set to 0, as follows:



Floating-Point Data

Floating-point data appears in memory either as a short-precision (32-bit) or long-precision (64-bit) constant, as follows:



C

Represents the characteristic (excess 64 power-of-16 exponent) of the number. The characteristic represents exponents with a range from -64 to +63. Since the characteristic has no sign bit, the number 64 (decimal) is effectively added to each exponent, thus allowing a characteristic range of 0 to 127 to represent exponents with a range of -64 to +63.

S

Sign bit (0 = +; 1 = -) of the mantissa.

M

Magnitude of the mantissa.

A floating-point constant in memory may be loaded into the software-simulated scientific register, described later in this section. If the floating-point constant had been specified as long-precision, the low-order (rightmost) 32 bits are ignored during the loading process.

LEVEL 6 HARDWARE CONSIDERATIONS

Hardware Registers

Level 6 provides hardware registers that can be loaded or read by various assembly language instructions. Of these registers, one is the program counter, seven are address registers, seven are general registers (of which three double as index registers), one is a mode control register, one is a system status register, and one is an indicator register.

Program Counter (P) Register

The program counter, or P-register, contains the address of the currently executing instruction. It is used by the Central Processor to generate the effective address of data based upon various operands in the assembly language instruction set (see "Addressing Techniques" in Section 5). Its content can be modified only by the JMP and branch instructions. If necessary, you can refer to the P-register for the address of the instruction that caused the system to abort a program. For this purpose, the contents of the P-register can be displayed at the control panel.

Address (Bn) Registers

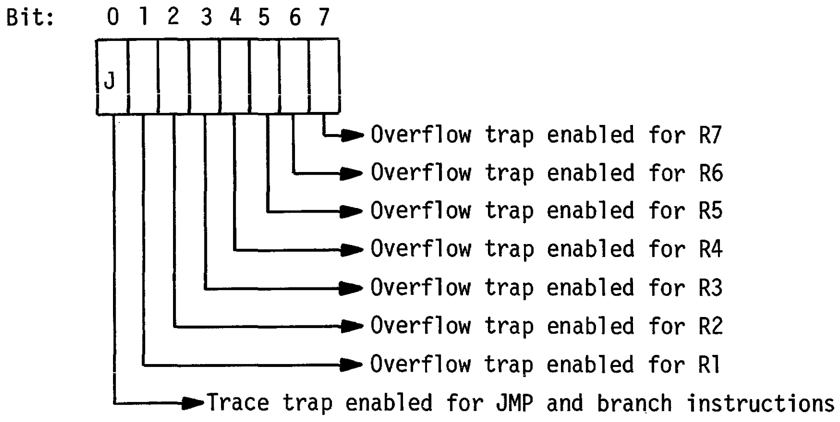
The seven address registers can be used in the formulation of addresses by pointing to any procedure, data, or location in main memory. Typically, the address registers contain addresses, pointers, or base references for use in generating effective addresses and referring to data through relative addresses (see "Addressing Techniques" in Section 5).

General (Rn) Registers

The seven general registers can be used as accumulators, and the first three (R1, R2, R3) can be used as index registers (see "Addressing Techniques" in Section 5).

Mode (M1) Register

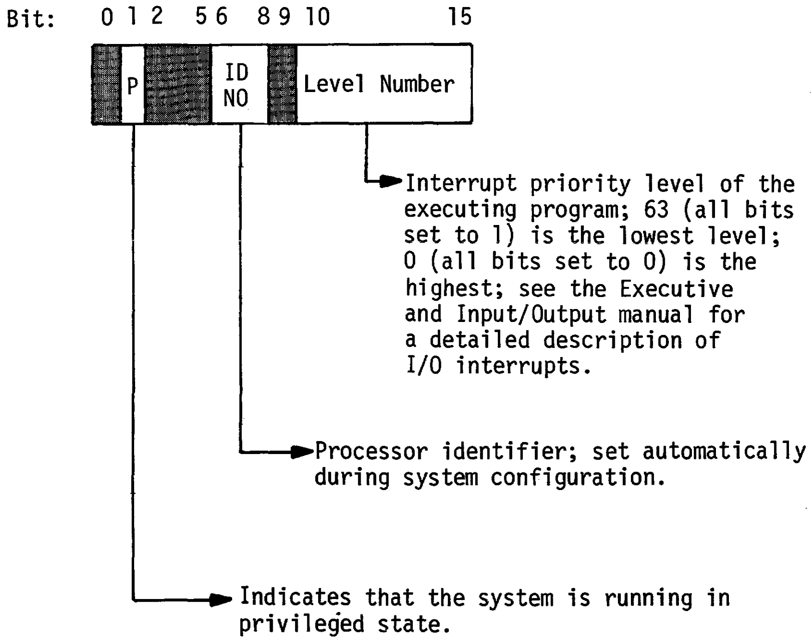
The mode, or M1, register contains the trap enable control bits. Its contents can be altered by the MTM assembly language instruction, and used by other instructions in the assembly language instruction set. The bits in the mode control register have the following meanings when set to binary 1:


Bit: 0 1 2 3 4 5 6 7
J
Overflow trap enabled for R7
Overflow trap enabled for R6
Overflow trap enabled for R5
Overflow trap enabled for R4
Overflow trap enabled for R3
Overflow trap enabled for R2
Overflow trap enabled for R1
Trace trap enabled for JMP and branch instructions

Setting one or more overflow trap bits makes it possible to enter the Trace Trap Handler by a trap-to-trap vector 6. See the Executive and Input/Output manual for a detailed description of trap handlers.

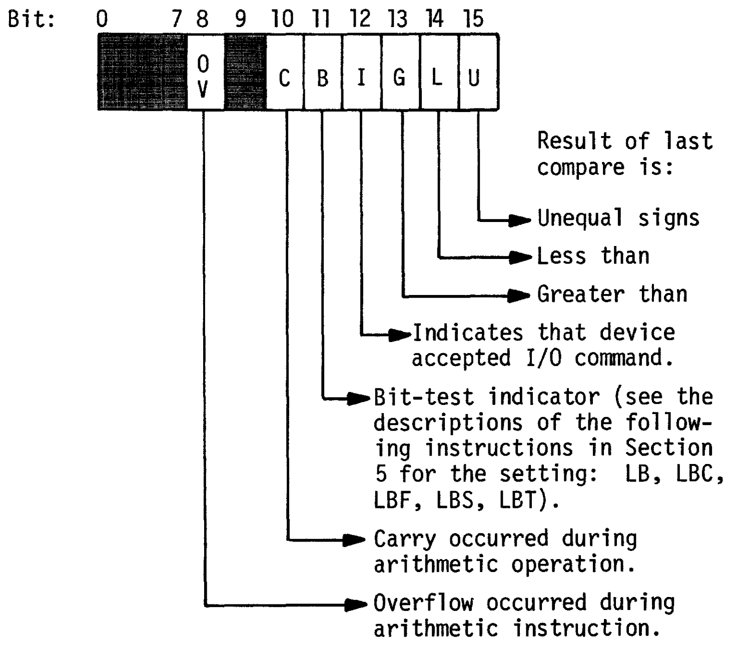
System Status (S) Register

The S-register contains the status and security bits for the system. The contents, which can be read by an executing program, have the following meaning, depending on which bits are set to binary 1:


Bit: 0 1 2 5 6 8 9 10 15
P
ID NO
Level Number
Interrupt priority level of the executing program; 63 (all bits set to 1) is the lowest level; 0 (all bits set to 0) is the highest; see the Executive and Input/Output manual for a detailed description of I/O interrupts.
Processor identifier; set automatically during system configuration.
Indicates that the system is running in privileged state.

Indicator (I) Register

The I-register contains overflow and program status indicators. When set to binary 1, the bits have the following meaning:


Bit: 0 7 8 9 10 11 12 13 14 15
OV
C
B
I
G
L
U
Result of last compare is:
Unequal signs
Less than
Greater than
Indicates that device accepted I/O command.
Bit-test indicator (see the descriptions of the following instructions in Section 5 for the setting: LB, LBC, LBF, LBS, LBT).
Carry occurred during arithmetic operation.
Overflow occurred during arithmetic instruction.

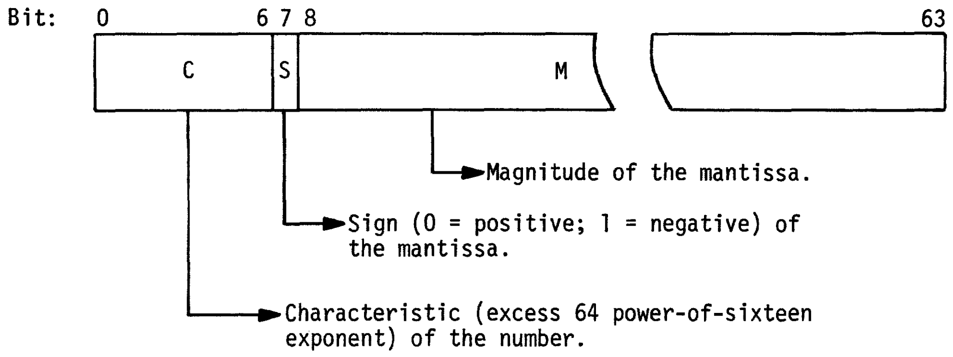
Scientific Information Processor (SIP) Registers

The Level 6 Scientific Information Processor (SIP) is an optional hardware unit containing three identical scientific accumulator registers, one scientific indicator register, one SIP mode register, and one SIP trap mask register. The SIP performs arithmetic operations on single- and double-precision floating-point data and also provides a set of scientific branch instructions.

Scientific Accumulator (Sn) Registers

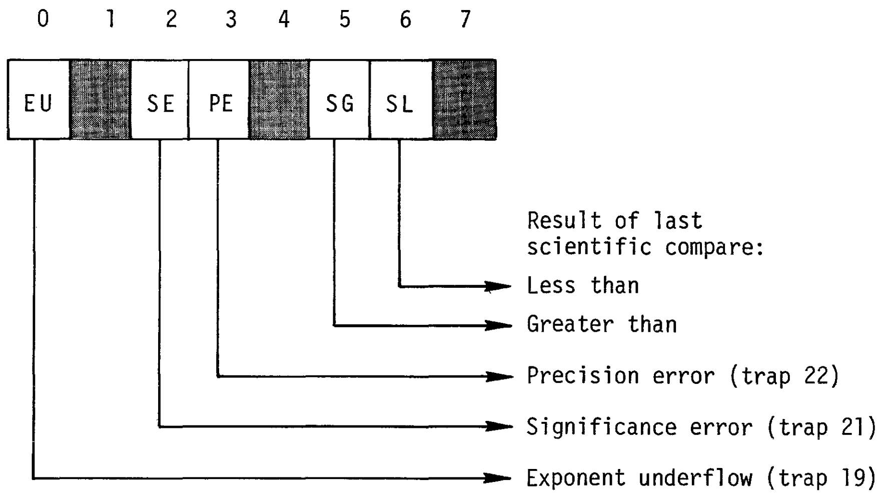
The SIP provides three 64-bit scientific accumulator registers for use in either short- or long-precision floating-point operations. When these registers are used in short-precision operations, only the high-order (leftmost) 32 bits participate.

The format of the scientific accumulator registers is shown below.


Bit: 0 6 7 8 63
C
S
M
Magnitude of the mantissa.
Sign (0 = positive; 1 = negative) of the mantissa.
Characteristic (excess 64 power-of-sixteen exponent) of the number.

Scientific Indicator (SI) Register

The 8-bit SI-register contains error and status indicators that can be tested with the scientific branch instructions. When set to binary 1, the bits have the following meanings:

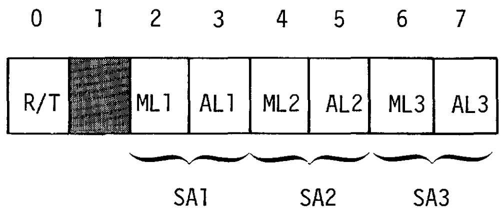

0 1 2 3 4 5 6 7
EU
SE
PE
SG
SL
Result of last scientific compare:
Less than
Greater than
Precision error (trap 22)
Significance error (trap 21)
Exponent underflow (trap 19)

Traps and trap handlers are discussed in the Executive and Input/Output manual.

SIP Mode (M4) Register

The SIP mode, or M4, register is an 8-bit control register residing in the SIP but with a copy in the CPU. Both versions are set to 0 upon CPU initialization and both may be modified with an MTM instruction (see Section 5). If only the SIP is initialized, the CPU copy of the register is cleared, and the contents of both versions must be reestablished with an MTM.

The format of the M4-register is as follows:



R/T: Round/Truncate Mode

0 – Truncate
1 – Round

ML: Memory Length (Length of main memory data field to or from which data is transferred via a scientific accumulator (SA))

0 – Two words
1 – Four words

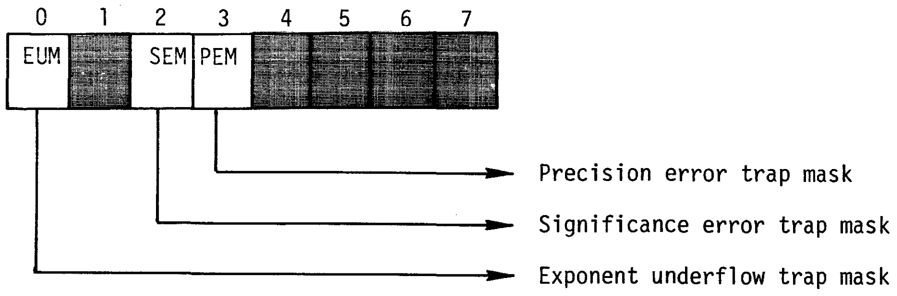
AL: Accumulator Length (Length of scientific accumulator data field to or from which data is transferred to/from main memory, a hardware register, or another SIP register)

0 – Two words
1 – Four words

SIP Trap Mask (M5) Register

The SIP Trap Mask, or M5, register is an 8-bit control register residing in the SIP but with a copy in the CPU. Both versions are set to 0 upon CPU initialization and both may be modified with an MTM instruction (see Section 5). If only the SIP is initialized, the CPU copy of the register is leared, and the contents of both versions must be reestablished with an MTM.

The format of the M5-register is as follows:


0
1
2
3
4
5
6
7
EUM
SEM
PEM
Precision error trap mask
Significance error trap mask
Exponent underflow trap mask

Software Simulation of the Scientific Information Processor

For Level 6 systems on which a Scientific Information Processor (SIP) is not installed or available, BES provides a limited equivalent of the SIP functions through software simulation. Two trap handlers, the Floating-Point Simulator, entered via trap vector 3, and the Scientific Branch Simulator, entered via trap vector 5, are available. These two simulators are described in the Executive and Input/Output manual.

The Floating-Point Simulator and Scientific Branch Simulator provide the same functions as the SIP, with the following differences:

- o Only one scientific accumulator register (S1) is supported.
- o Only short-precision floating-point operations may be performed.
- o General registers R4, R5, and R7 must be reserved for use by the simulators while they are executing.
- o Since, in the absence of an SIP, no SI-register is available, the simulators use the G, L, and U bits of the I-register for scientific compares.
- o Not all SIP instructions are simulated. See "Assembly Language Instructions" in Section 5 to determine whether or not an individual instruction is available with one of the simulators.

SECTION 2

ELEMENTS OF BES ASSEMBLY LANGUAGE

The principal elements of the BES assembly language are:

- o Mnemonic codes
- o Symbolic names
- o Constants
- o Expressions

These elements are combined to form a source program that consists of:

1. Machine instructions to be assembled, on a one-to-one basis, into their corresponding object code representations.
2. Assembler control statements which are interpreted by the Assembler to control the assembly process, allocate work and storage areas in memory, and to define constant data used by the program.

MNEMONIC CODES

Assembler control statements, which direct the Assembler in the preparation of object code, and assembly language instructions are specified by predefined mnemonic names of one to five characters in length. These mnemonic (operation) codes are described, in detail, in Sections 4 and 5.

SYMBOLIC NAMES

Locations, values, and other data pertinent to the determination of assembly language instruction or Assembler control statement operand values can be referred to by the use of reserved (predefined) and user-defined names.

Character strings can be assigned as names of memory locations, registers, values, or other objects to be referred to in the development of object code. The manner in which a symbolic name is defined depends on the attributes of the object referred to by that name.

Regardless of the manner of definition and the type of object being referred to, the symbolic name must conform to the following rules:

1. It must be from one to six characters long.
2. It must be composed of alphabetic characters (A,B,...Z), digits (0,1,...9), and/or the special characters $ and –(underscore).
3. The first character must be a $ or alphabetic character.

The following types of symbolic names can be used in Assembler control statements and assembly language instructions:

- o Identifiers - Reserved symbols designating the hardware registers and the scientific register
- o Labels - User-defined and reserved symbols designating locations in memory and values

Identifiers

Identifiers are reserved symbolic names that refer to hardware registers or to the software-simulated scientific register. In addition, names that are defined to be equivalent to identifiers (through the EQU Assembler control statement) are treated as identifiers.

The following identifiers refer to hardware registers:

- o $B1 through $B7 - Address (base) registers
- o $R1 through $R7 - General registers
- o $R1 through $R3 - Index registers
- o $M1 through $M7 - Mode control registers
- o $S1 through $S3 - Scientific accumulator registers

Labels

Labels are symbolic names that can be used to refer to locations and values. They must be defined in a manner specific to the attributes of the location or value to which they refer (i.e., each label is typed according to the location or value attributes, which also establish the context in which they can be used). The types of labels and their methods of definition are as follows:

- o Internal location label - Refers to a location allocated within the assembled program. It is defined by its occurrence in the label field of an instruction (resulting in the allocation of memory to the program). The definition of labels appearing in certain Assembler control statements that do not cause memory to be allocated (e.g., EQU statement) depend on the statement and its operands.
- o External location label - Refers to a location in another, independently assembled program. It is defined by appearing in the operand list of an XLOC statement.
- o Common location label - Refers to a location allocated to FORTRAN-compatible common blocks. It is possible to specify that the object code resulting from assembly language instructions is to be allocated to a common block area rather than to the area allocated to the program by means of the ORG statement. All labels that appear in instructions that result in the allocation of common block locations are defined as common location labels. In addition, labels specified in the COMM statement are defined as common location labels; these labels can be used to refer to locations in the common block by indicating their offset from the first word.
- o Internal value label - Refers to a value defined within the program. It is assigned by its occurrence in the label field of an EQU statement with an operand expression (see "Expressions" in this section) that yields a scalar value.
- o External value label - Refers to a value defined in another, independently assembled program. It is defined by appearing in the operand list of an XVAL statement.
- o Complex label - Refers to the label of an EQU statement that has an address expression (see "Expressions" in this section), or the label of another EQU statement that has an address expression, in the operand field.

Table 2-1 summarizes the types of labels and how they are defined.

User-Defined Labels

User-defined labels can be either permanent or temporary. Permanent labels can be defined only once in a program; they must conform to the rules listed under "Symbolic Names" in this section.

The 26 temporary labels ($A, $B, ..., $Z) may be defined as often as necessary within a single program. They may be referred to only in the operand of a hardware instruction or of a define constant (DC) assembly control statement. You must be careful, during programming, that you are referring to the desired definition of a temporary label when the label has multiple definitions within a single program.

Temporary labels must be defined as internal location labels.

Reserved Labels

Reserved labels are predefined and cannot be redefined by the user. The following reserved labels are available:

- o $ - Refers to the location to be allocated as a result of the statement in which it appears as a reference (i.e., the current location). It can be either an internal location or common location label type.
- o $AF - Refers to the address form of the system configuration. A value of 1 indicates that the system configuration is the short-address form configuration; a 2 indicates a long-address form configuration. $AF is an internal value label.
- o $IV - Refers to the address of the interrupt vector for the priority level at which the application is currently executing. A description of interrupt vectors and priority levels can be found in the Executive and Input/Output manual.

TABLE 2-1. DEFINING BES SYMBOLIC NAMES

Type	How Defined
Internal location label	Appears in label field of an assembly language instruction or Assembler control statement (except EQU or COMM statements) when the location counter type attribute (set by the ORG statement) is internal.
External location label	Appears in the operand field of an XLOC statement.
Common location label	Appears in the label field of a COMM statement; or appears in label field of an assembly language instruction or Assembler control statement (except EQU or COMM statements) when the location counter type attribute (set by the ORG statement) is common.
Internal value label	Appears in label field of an EQU statement that has an expression that yields a scalar arithmetic value in the operand field.
External value label	Appears in the operand field of an XVAL statement.
Complex label	Appears in the label field of an EQU statement that contains an address expression in the operand field; or appears in the label field of an EQU statement that contains a label identifying another EQU statement that contains an address expression in the operand field.
Same as operand	Appears in the label field of an EQU statement that contains an operand other than one of those listed above; e.g., an identifier.

CONSTANTS

Arithmetic and nonarithmetic values can be expressed in decimal, hexadecimal, character, or binary form, all of which are converted by the Assembler to the appropriate machine code format. Depending on the context, such values may be assigned as object code or be used by the Assembler in the computation of operand locations or values.

The following types of constants are supported:

- o String constants
- o Arithmetic constants

String Constants

String constants can be expressed as ASCII, hexadecimal, or bit strings. Regardless of how they are expressed, string constants have the following format:

[(n)] $\begin{Bmatrix}\underline{A}\\ Z\\ B\end{Bmatrix}$'c[c...]'

[(n)]

Specifies an optional decimal integer in the range from 1 to 255, which represents the replication factor (number of times the coded string is to be repeated.

$\begin{Bmatrix}\underline{A}\\ Z\\ B\end{Bmatrix}$

Specifies whether the string is expressed in ASCII (A; default if none of these values is specified), hexadecimal (Z), or bit (B).

'c[c...]'

Identifies the character(s) in the string; to include an apostrophe, a double apostrophe must be specified (i.e., '' is interpreted as ').

String constants are left-justified.

ASCII String Constants

An ASCII string constant is written as the letter A (optionally) followed by a string of any of the valid ASCII characters enclosed within apostrophes.

An ASCII string constant denotes the value formed by replacing all double apostrophes by a single apostrophe and removing the delimiting apostrophes.

The value of an ASCII string constant cannot be more than 255 ASCII characters (each of which is eight bits long).

The format of an ASCII string constant is as follows:

[(n)] [A]'a[a...]'

The following examples illustrate how to specify ASCII string constants:

1. 'ASCII SAMPLE1'
2. A'ASCII SAMPLE2'
3. (4)A 'DATAΔ'

The characters enclosed within the apostrophes can be any character shown in Table B-4. The examples shown above result in the following values being stored in memory, respectively:

1. ASCII SAMPLE1
2. ASCII SAMPLE2
3. DATAΔ DATAΔ DATAΔ DATAΔ

Hexadecimal String Constants

A hexadecimal string constant is written as the letter Z followed by a string of any of the valid hexadecimal digits (i.e., 0 through F) enclosed within apostrophes.

A hexadecimal string constant denotes the value formed by replacing the characters contained within the delimiting apostrophes with their binary values and removing the delimiting apostrophes.

The value of a hexadecimal string constant cannot be more than 510 hexadecimal digits (each of which is four bits long).

The format of a hexadecimal string constant is as follows:

[(n)] Z'h[h...]'

The following example illustrates how to specify a hexadecimal string constant:

```
Z'5449544C452053414D504C4531'
```

This example translates into TITLE△ SAMPLE1 (see Appendix B).

Bit String Constants

A bit string constant is written as the letter B followed by a string of binary digits (i.e., 0 and 1) enclosed within apostrophes.

A bit string constant denotes the value formed by converting the 0 and 1 characters contained within the delimiting apostrophes to 0 and 1 bits.

The value of a bit string constant cannot be more than 2040 binary digits (each of which is one bit long).

The format of a bit string constant is as follows:

```
[(n)] B'd[d...]'
```

The following example illustrates how bit string constants are expressed:

```
B'00011010'
```

This bit string provides an 8-bit mask that can be used by an assembly language instruction.

Truncation/Padding of String Constants

Various statements require a half-word (8-bit) value, whole-word (16-bit) value, or a value that is an integral number of words in length. In order to satisfy these requirements, string constants are automatically truncated or padded.

If truncation is required, low-order (i.e., the rightmost) bits are discarded, and the Assembler issues a diagnostic message.

If padding is required, low-order bits are appended to the value. ASCII string constants are padded with spaces; hexadecimal and bit strings are padded with 0's.

Table 2-2 describes how the Assembler handles the various situations that require truncation or padding.

TABLE 2-2. RULES OF TRUNCATION/PADDING STRING CONSTANTS

If a string constant appears:	It is converted to:
In a nontrivial arithmetic expression	A whole-word value.
As the only term of the operand of a short value immediate (SI) instruction	A half-word value.
As the only term of an operand of a DC Assembler control statement	A value having a length that is an integral number of words; such string constants are never truncated.
As the operand of a TEXT Assembler control statement	A string having an initial bit offset which is a multiple of 4 (for hexadecimal string constants) or a multiple of 8 (for ASCII string constants) with slack bits inserted between successive operands. A bit string constant can begin at any bit position; slack bits never precede a bit string operand.
In any context not listed above	A whole-word value.

NOTES: 1. If two or more rules apply to the same string constant, the first takes precedence.
2. Refer to specific statements identified in this table for additional information.

Arithmetic Constants

An arithmetic constant specifies the value of a real number. An arithmetic constant is either an integer constant, fixed-point constant, or a floating-point constant.

Integer Constants

Integer constants can be expressed as decimal or hexadecimal integers. They may be preceded by a plus (+) or minus (-) sign, indicating a positive or negative value, respectively, and must be within the range -32768 to +32767; if unsigned, an integer constant is assumed to be positive.

Integer constants have the following format:

```
+   n[n...]
-   X'h[h...]'
```

+
-
Specifies whether the value is positive (+; the default value) or negative(-).

n[n...]
Is a decimal integer constant as defined below.

X'h[h...]'
Is a hexadecimal integer constant as described below.

Decimal Integer Constants

Decimal integer constants are expressed as character strings composed of the decimal digits 0 through 9.

The following examples illustrate valid decimal integer constants:

1. 31764
2. +4652
3. -6781

Hexadecimal Integer Constants

A hexadecimal integer constant is written as the letter X followed by a character string composed of the hexadecimal digits 0 through 9 and A through F enclosed within apostrophes.

The following examples illustrate hexadecimal integer constants:

1. +X'2F'
2. X'7FFF'
3. -X'8000'

Using Table B-3, you can see that the decimal equivalent of the above examples is +47, +32767, and -32768, respectively.

Fixed-Point Constants

A fixed-point constant is written as a decimal number with an associated scale factor. When the resultant value is stored in memory, a fixed-point constant appears as a signed integer word with negative values in twos complement form. The scale factor (s) gives the location of the implied binary point in the stored constant. A positive scale factor means that the point is situated s bits to the left of the rightmost bit stored in memory. A negative scale factor means that the point is situated s bits to the right of the rightmost bit stored in memory. Thus, a fixed-point value can be considered to be written as the product formed by multiplying the decimal number by 2^s.

Fixed-point constants have the following format:

$$\begin{bmatrix}+\\-\end{bmatrix}\left\{\begin{matrix} i\ [.[f]] \\ [i]\ .f \end{matrix}\right\} B \begin{bmatrix}+\\-\end{bmatrix} s$$

[±]
Specifies the sign of the constant. The + sign may be omitted.

i
Specifies the integer part of the decimal number.

f
Specifies the fractional part of the decimal number.

[±]s
Specifies the value and sign of the scale factor.

The value of a fixed-point constant must fall within the range

$$2^{-s} \leqslant |R| < 2^{15-s}$$

where R is the value of the decimal number.

The following examples illustrate how to specify fixed-point constants and show the hexadecimal representations of the resultant values in memory.

Source Language	*Stored Value*
2.5B4	0028
2.5B8	0280
65536B-15	0002
65536B-7	0200
-2.5B8	FD80
-65536B-15	FFFE

Floating-Point Constants

BES assembly language provides a convenient method with which you can write a decimal number and have the Assembler convert it into floating-point format. (See Section 1 for a description of floating-point data.)

Two formats for writing floating-point constants are available:

Format 1

$$\begin{bmatrix}+\\-\end{bmatrix}\left\{\begin{matrix} i\ .[f] \\ [i]\ .f \end{matrix}\right\} \qquad \text{SHORT PRECISION}$$

Format 2

$$\begin{bmatrix}+\\-\end{bmatrix}\left\{\begin{matrix} i\ .[f] \\ [i]\ .f \end{matrix}\right\} E \begin{bmatrix}+\\-\end{bmatrix} c \qquad \text{SHORT PRECISION POWER-OF-10}$$

[±]
Specifies the sign of the constant. The + sign may be omitted if desired.

i
Specifies the integer part of a decimal number.

f
Specifies the fractional part of a decimal number.

E
Indicates that a short precision power-of-10 floating-point representation is desired.

[±] c

Expresses the power of 10 by which the coded decimal number should be multiplied to produce the value wanted. The + sign may be omitted if desired.

NOTE: If the decimal point is omitted, the number is assumed to be an integer.

The absolute value of a floating-point constant must be greater than or equal to 2^{-260} (approximately 3.3753×10^{-80}) and less than 2^{252} (approximately 4.7428×10^{80}).

Normalization

Floating-point constants are stored as normalized hexadecimal floating-point numbers with a 7-bit excess 64 power-of-16 characteristic and a 25-bit signed magnitude mantissa. A normalized floating-point number has a nonzero high-order hexadecimal fraction digit. If one or more high-order fraction digits are zero, the number is said to be unnormalized. Normalization consists of shifting the fraction left until the high-order hexadecimal digit is nonzero and reducing the characteristic by the number of hexadecimal digits shifted.

Examples

The following examples illustrate how to specify floating-point constants and show the hexadecimal representations of the resultant values in memory. You can determine sign, characteristic, and mantissa of the resulting floating-point numbers by dividing the hexadecimal representations into parts according to the patterns described in Section 1.

Source Language	*Stored Value*
0.5	8080 0000
0.5E12	9474 6A52
6.665039063E-2	8011 1000
-6.665039063E-2	8111 1000

EXPRESSIONS

Expressions are combinations of symbolic names and constants used as operands within Assembler control and assembly language (machine) instructions. Expressions can represent locations (internal or external), values, and addresses. Components of an expression can be joined by various functions and arithmetic operators, as follows:

Arithmetic Operator	*Meaning*
+	Addition (or Unary +)
-	Subtraction (or Unary -)
*	Multiplication
/	Division

Boolean Function	*Meaning*
AND	Conjunction
OR	Inclusive Disjunction
XOR	Exclusive Disjunction
NOT	Negation

Shift Functions	*Meaning*
ALS	Arithmetic Left Shift
ARS	Arithmetic Right Shift
LLS	Logical Left Shift
LRS	Logical Right Shift

Arithmetic Function	*Meaning*
MOD	Remainder after division

When a value is operated upon by an arithmetic operator or function or by an arithmetic shift function the value is considered to be a 16-bit signed (twos complement) binary integer. When a value is operated upon by a Boolean or logical shift function the value is considered to be a 16-bit string. You must ensure that the results of a Boolean or shift operation will be meaningful when subsequently interpreted as an integer value by the Assembler.

To use a function within an expression you write the function name followed by its operands, enclosed in parentheses and separated by a comma; e.g., AND (TAG1,TAG2).

Evaluating Expressions

Within an expression you may use parentheses to eliminate ambiguities or to specify the order of evaluation. Expressions within parentheses are evaluated first. Within nested parentheses, evaluation proceeds from the least inclusive set to the most inclusive. If parenthesized expressions are at the same level of inclusiveness or if parentheses are not used, the following hierarchical order of evaluation applies:

1. All functions
2. Unary plus and minus
3. Multiplication and division
4. Addition and subtraction

Once the values resulting from these operations have been computed, evaluation proceeds from left to right.

Location and Value Expressions

The Assembler permits expressions to be used to specify values and locations. An internal value expression denotes a computation to be performed by the Assembler and produces an integer scalar value.

A location expression denotes a computation of an address that can be internal to the referencing program, in a separately assembled program (i.e., external to the referencing program), or in a common memory block.

Internal Value Expressions

An internal value expression, which produces an integer scalar value, is written as a sum-of-products algebraic expression. The product portion consists of two or more factors to be multiplied or divided as indicated by the * or / operators, preceding the multiplier or divisor factor. In addition, each factor can be preceded by a unary plus (+) or minus (-) operator.

Each factor of the product portion of the expression must be an internal value expression enclosed within parentheses, an integer or string constant (see "Constants" in this section), or an internal value label (see "Labels" in this section).

The sum portion of the algebraic expression consists of two terms to be added or subtracted as indicated by the + or - operator preceding the addend or subtrahend term. In addition, each term can be preceded by a unary plus (+) or minus (-) operator.

Each sum of an internal value expression must take one of the following forms:

- o An internal value expression plus or minus an integer or string constant, an internal value label, an internal value expression enclosed within parentheses, or a product of such terms
- o The difference between two internal locations
- o The difference between two common locations within the same common block

The following examples illustrate internal value expressions. In these examples, labels of the form VALc are internal value labels, labels of the form LOCc are internal location labels, and labels of the form COMMc are common location labels.

Example 1:

X'34F0'+(VAL8-(VALB/(X'E4'*2)))

In this example, the expression is evaluated as follows:

1. The product of X'E4'*2 is calculated.
2. The value associated with VALB is divided by the product of step 1, above.
3. The quotient of step 2, above, is subtracted from the value associated with VAL8.
4. The difference calculated in step 3, above, is added to X'34F0'

Example 2:

B'11110110'+(COMM1-COMM2)/2*(54+VALF-(LOCA-LOCB))

The expression in example 2 is evaluated as follows:

1. The difference between LOCA and LOCB is calculated.
2. The difference between COMM1 and COMM2 is calculated.
3. The sum of 54 and value associated with VALF is calculated.
4. The result of step 1, above, is subtracted from the result of step 3.
5. The result of step 2, above, is divided by 2.
6. The quotient calculated in step 5 is multiplied by the result of step 4.
7. The bit string constant B'11110110' is padded to occupy a full word and added to the result of step 6.

Location Expressions

Location expressions are used to express computations to be used by the Assembler. There are three types of location expressions:

- o Internal location expressions - Refer only to values that are defined within the referencing program.
- o External location expressions - Refer to one memory address defined in an external program and may refer to elements within the referencing program.
- o Common location expressions - Refer to one or more locations within common blocks and may refer to elements within the referencing program.

Each of the above types of location expressions produces a memory address.

Internal Location Expressions

Internal location expressions, which produce a memory address based upon a computation using only internal values, must take one of the following forms:

- o An internal location expression plus or minus an integer or string constant, an internal value label, or an internal value expression enclosed within parentheses.
- o An internal value expression plus an internal location label, an internal location expression enclosed within parentheses, or a $ (which is valid only if the Assembler's location counter type attribute is internal when the expression is processed).

The following example illustrates internal location expressions. In this example, labels of the form LOCc are internal location labels.

Example:

```
(LOC3-LOCD)+X'30F2'+LOCA
```

The expression in this example is evaluated as follows:

1. The address associated with LOCD is subtracted from the address associated with LOC3 yielding an internal value.
2. X'30F2' is added to the result of step 1 yielding another internal value.
3. The address associated with LOCA is added to the result of step 2 yielding an internal location as the final result.

External Location Expressions

External location expressions, which produce a memory address based upon a computation using external location labels and internal values, must take one of the following forms:

- o An external location expression plus or minus an integer or string constant, an internal value label, or an internal value expression.
- o An internal value expression plus an external location label or an external location expression.

The following example illustrates an external location expression. In the example, labels of the form XLOCc are external location labels and labels of the form VALc are internal value labels.

Example:

```
((VAL1+VALA)+XLOC2)X'2A22'
```

This sample expression is evaluated as follows:

1. The values associated with VAL1 and VALA are added together.
2. The offset associated with XLOC2 is added to the result of step 1.
3. X'2A22' is added to the result of step 2.

Common Location Expressions

Common location expressions, which produce a memory address based upon a computation using one or more locations within a common block and internal values, must take one the following forms:

- o A common location expression plus or minus an integer or string constant, an internal value label, or an internal value expression.
- o An internal value expression plus a common location label, a common location expression, or a $ (which is valid only if the assembler's location counter type attribute is common when the expression is processed).

A memory address referring to a common block is represented by the name of the common block and an optional offset from the beginning of that common block.

The following example illustrates a common location expression. In the example COMMc is a common location label and labels of the form VALc are internal value labels.

Example:

```
((COMMA+42)-(COMMA+80))-VAL2)*2+X'1000'+COMMB
```

The expression in this example is evaluated as follows:

1. The difference between COMMA+42 and COMMA+80 is calculated.
2. The value associated with VAL2 is subtracted from the result of step 1.
3. The result of step 2 is multiplied by 2.
4. X'1000' is added to the result of the calculation in step 3.
5. The offset associated with COMMB is added to the result of step 4. This offset is then associated with the name of the common block containing COMMB to complete the evaluation of this expression.

Address Expressions

An address expression specifies the addressing form used in an instruction. It contains special character identifiers that are assembled into corresponding object code to control run-time address development processes such as indirection and indexing.

The various forms of address expressions permitted by the Assembler are described in detail in Section 5 (see "Addressing Techniques").

Function	**Examples**	
VAL1	EQU	X'100'
VAL2	EQU	X'10F'
VAL3	EQU	3
LOC1	EQU $	(at location 200 hexadecimal)

AND
```
DC <LOC1+AND(VAL13VAL2)
```
resolves to address 300 hexadecimal

OR
```
DC <LOC1+OR(VAL1VAL2)
```
resolves to address 30F hexadecimal

XOR
```
DC <LOC1+XOR(VAL1,VAL2)
```
resolves to address 20F hexadecimal

NOT
```
VAL4 EQU NOT(VAL2)
```
resolves to value FEF0 hexadecimal

ALS
```
VAL5 EQU ALS(VAL1,VAL3)
```
resolves to value 800 hexadecimal)

ARS
```
VAL6 EQU ARS(VAL1,VAL3)
```
resolves to value 20 hexadecimal

LLS
```
VAL7 EQU LLS(VAL2,12)
```
resolves to value F000 hexadecimal

LRS
```
VAL8 EQU LRS(VAL2VAL3)
```
resolves to value 21 hexadecimal

MOD
```
VAL9 EQU MOD(VAL2,VAL1)
```
resolves to value F hexadecimal

REFERENCES

References are the use of symbolic names as labels in assembly statements to refer to locations or values.

The employment of references is dependent upon two conditions:

1. The resolution of labels by the two-pass BES Assembler.
2. The position of the referencing statement within the body of the program.

A simple rule may always be applied to determine the validity of a reference: the reference to a label is legitimate if during the second assembly pass, at the point in the program where the referencing statement is positioned, the value of the label being referred to, has been defined.

References may be made either forward or backward. A forward reference is a reference to a label that is defined after the referencing statement. A backward reference is a reference to a label defined in a statement before the referencing statement.

Further, forward or backward references may be categorized as either simple or complex. A simple reference is a forward or backward reference to a label that is directly defined by the referenced statement. A complex reference is a forward or backward reference to a label defined by an equate (EQU) statement that in turn makes at least one additional reference.

Example:
References

```
A   DC     13
G   DC     7
    LDR    $R1,A    (Valid simple backward reference)
    LDB    $B1,X    (Valid simple forward reference)
W   EQU    E
B   EQU    G
    LDR    $R2,E    (Invalid complex forward reference (label E not defined at
                    this point))
    LDR    $R3,W    (Invalid complex backward reference (label W can never be
                    defined in a two-pass assembly))
E   EQU    D
    LDR    $R4,E    (Valid complex backward reference (label E has been defined
                    at this point))
    LDR    $R5,C    (Valid complex forward reference (label C has been defined
                    in the first assembly pass))
C   EQU    B
D   RESV   1
X   DC     3
```

Restrictions that apply to references are as follows:

1. All forward references to a label defined by a complex equate statement are invalid.
2. A forward reference in an origin (ORG) or common (COMM) statement is invalid.
3. A forward reference in the first operand of a reserve (RESV) or conditional assembly control IFxx statement is invalid.
4. A complex reference involving one or more intermediary equate statements making a forward reference is invalid.

SECTION 3

PROGRAMMING CONSIDERATIONS

Before writing an assembly language source program, you should take into consideration both features and constraints inherent in the design of the Assembler and the system. This section describes the considerations that should be made, as well as the various rules that must be followed, when coding your source program. These include:

- o Rules of formatting your source language statements
- o Ordering of statements in an assembly language program
- o Rules governing the calling of system services and external procedures
- o Utility programs that supplement assembly language source programs

ASSEMBLY LANGUAGE SOURCE STATEMENT FORMATS

As mentioned in Section 2, the BES assembly language consists of Assembler controlling statements and assembly language (operational) instructions. Assembly language source code must be submitted to the Assembler in a recognizable format so that it can be interpreted accurately. Therefore, when coding assembly language source statements, you must conform to the following formatting conventions:

```
Column 1─┐
        ┌Δ                ┐        ┌           ┌Δ          ┌Δ              ┐┐┐
        │label Δ          │ opcode │ Δoperand  │ ,operand  │ ,operand [...]│││ [Δcomments]
        │linenum Δ        │        │ Δ         │ ;         │ ;             │││
        └linenum-label Δ  ┘        └           └           └               ┘┘┘
```

The semicolon (;) indicates to the Assembler that the next operand is contained in the next sequential source line (i.e., the continuation statement), which has the following format:

```
Column 1 ──────┐
               ↓
         [linenum] [Δ] ┌ Δoperand ┌ ;              ┐┐     [Δcomments]
                       │          │ ,operand [...] ││
                       └ Δ        └ Δ              ┘┘
```

In addition to comments being included on individual assembly language source statements, comment statements, which have the following format, can be included in the source language program.

```
Column 1 ──────┐
              ┌*┐
              └/┘          comments
```

The asterisk (*) indicates that the comment line is to be included in the listing wherever it is included in the source language program. The slash (/) indicates that the printer is to skip to the top of the next page of the listing before printing the comment. Printing of lines can be overridden by the inclusion of a NLST Assembler control statement in the source code (see Section 4).

In the above formats, label is any user-specified tag, linenum is any user-specified line number, linenum-label indicates a line number followed by a label with no intervening spaces, opcode and operand indicate the required assembly language fields described in Sections 4 and 5, and blank (Δ) indicates that one or more blanks or horizontal tab characters must be coded. Any number of blanks and/or horizontal tab characters can follow a comma (,). A line number is an unsigned decimal integer of any length. Line numbers are ignored by the Assembler.

Except for the order in which information must be supplied, the source language format is free-form. However, it is suggested that you establish a fixed format for coding source statements (e.g., always starting opcodes in the eleventh position and operands in the twenty-first) so that you can read your listing more easily.

ORDER OF STATEMENTS IN SOURCE PROGRAM

With the following exceptions, Assembler control statements can be entered in any order:

1. The TITLE statement must be the first statement in the source program.
2. The EQU statement must define complex type labels before they are referred to within the source program.
3. The END statement must be the last statement in the source program.

However, to simplify the reading of listings, it is recommended that you group all of the memory allocating Assembler control statements (e.g., DC, TEXT, RESV) at the beginning of your program (immediately following the TITLE statement), except for the conditional assembly-control Assembler control statements (i.e., IF, NULL, FAIL).

Assembly language instructions are coded in the order in which they are to execute.

CALLING SYSTEM SERVICES

System services (e.g., the Task Manager) can be requested by coding a request sequence similar to the following:

```
SAVE <savloc,2'639C'
LNJ  $B5,<entry
```

In the above sequence, 'savloc' is the label of the context save area and "entry" is the external label of the appropriate entry point of the system service routine.

The X'639C' operand results in saving the contents of the following hardware registers: R1, R2, R6, R7, I, B3, B4, B5. It may be necessary to save their contents because the various system services use these registers and may alter their contents. Any but the I-register can be used to pass arguments to the requested system service, and any register not used by the system service (i.e., R3, R4, R5, B1, B2, B6, B7) can be made available for the service to return a value to the requesting program.

All system services execute a JMP $B5. For that reason, the LNJ instruction must identify B5 as the register containing the return address.

For additional information about calling system services, see the Software Overview and System Convention manual.

CALLING EXTERNAL PROCEDURES

Procedures that are assembled separately from the invoking procedure are designated external procedures.

The individual elements of data passed to an external procedure are known as *arguments*. The external procedure interprets these arguments as *parameters*; to the external procedure, the order of the parameters is the same as the order of the arguments passed from the invoking procedure.

Although the standard calling sequence does not allow reentrant code, an external procedure *can* be reentrant (i.e., it can be used by more than one program concurrently), provided the following rules are observed:

1. The only work space used by the external procedure comprises (1) the parameters passed to it and (2) the hardware registers.
2. All the arguments to be passed in nested external procedure invocations are constants.

In other words, an external procedure can be reentrant if the invoking program provides the required work space and formally passes it to the external procedure.

External procedures can be requested by coding request sequences such as the following:

```
LAB $B7,arglist
LNJ $B5,<entry
```

In the above sequence, 'entry' is the external label of the appropriate entry point of the called (external) procedure, and 'arglist' is the argument list to be passed to the called (external) procedure.

Alternatively, you could use a request such as the following:

```
CALL entry,arg1,arg2,...
```

This request is similar to the preceding sequence except that the CALL Assembler control statement automatically generates the argument list, loads its address into B7, and sets the return address in B5. As a result, when the external procedure completes its work, control is returned to the next sequential instruction or statement in the calling program.

For additional information about calling external procedures, see the Software Overview and System Conventions manual.

ASSEMBLER-RELATED UTILITY PROGRAM

A special Assembler-related utility program is available to assembly language programmers. The cross-reference utility program, which is described in detail in the Program Development Tools manual, provides a listing of all labels and symbols in the source module. In addition, it flags labels that are undefined or defined more than once.

The listing includes the label, the location at which it is defined and a list of all locations that refer to that label. The list of labels is alphabetical; the locations that refer to each label are listed in ascending sequence.

BES ASSEMBLER

The BES Assembler processes source statements written in BES assembly language, translates the statements into object code, and produces a listing of the source program together with its associated assembly information.

The Assembler accepts arguments that allow you to control its operation in various ways. Detailed information about the Assembler and its arguments can be found in the Program Development Tools manual.

Scientific Instruction Processor (SIP) Programming Considerations

Since the SIP and the Level 6 central processor operate asynchronously, you must ensure that they do not come into conflict by attempting to use a main memory operand concurrently. You can guarantee proper synchronization by obeying the following rules:

1. If the source field of any of the following instructions refers to a main memory location or R-register, do not modify that location or register until a scientific branch instruction or another floating-point instruction is executed:

   SAD
   SCM
   SCZD
   SCZQ
   SDV
   SLD
   SML
   SNGD
   SNGQ
   SSB
   SSW

2. If the result field of any of the following instructions refers to a main memory location, do not modify that location until a scientific branch instruction or another floating-point instruction is executed:

   SNGD
   SNGQ
   SST
   SSW

Descriptions of the above instructions can be found in Section 5 under "Assembly Language Instructions."

SECTION 4

ASSEMBLER CONTROL STATEMENTS

Every assembly language program must contain, in addition to the assembly language instructions, a set of instructions that tells the Assembler about the program. These Assembler control statements, which are not assembled into the object text, provide information to the Assembler for:

- o Controlling the assembly of the program
- o Controlling the listing of assembly language instructions and Assembler control statements
- o Defining constants to be used by the program
- o Defining main memory storage and/or work areas
- o Defining symbols
- o Linking assembly language programs
- o Conditioning the assembly of various parts of a program

Assembler control statements must be coded as described in Section 3 (see "Assembly Language Source Statement Formats"), except that some explicitly prohibit the use of labels. For that reason, each Assembler control statement described in this section identifies labels where they are required or permitted; when not shown under "Source Language Format," labels are not allowed.

ASSEMBLY-CONTROLLING STATEMENTS

Assembly-controlling statements tell the Assembler where the beginning and end of each program are; they also set the Assembler's location counter.

The following statements are the assembly-controlling subset of Assembler control statements:

- o END
- o ORG
- o TITLE

These statements are described in detail later in this section.

LIST-CONTROLLING STATEMENTS

List-controlling statements control the listing of an assembly language source program via a printer, disk, or console typewriter. The following statements are available to provide this function:

- o CLST
- o LIST
- o NLST

These statements are described in detail later in this section.

DATA-DEFINING STATEMENTS

Data-defining statements are required to define data used in the object text. The Assembler assigns this data to memory locations at the exact point at which they are defined. The following statements are the data-defining subset of the Assembler control statements:

o DC
o TEXT

These statements are described in detail later in this section.

STORAGE-ALLOCATION STATEMENTS

Storage-allocation statements direct the Assembler to make areas of memory available for use as storage and/or work space. This subset of the Assembler control statements consists of the following statements:

o COMM
o RESV

These statements are described in detail later in this section.

SYMBOL-DEFINING STATEMENTS

Symbol-defining statements assign specific meanings to given symbolic names; they also may identify symbolic names defined outside the program but used within it. The assembler control statements provided to support the symbol-defining function are:

o EQU
o XLOC
o XVAL

These statements are described in detail later in this section.

PROGRAM-LINKING STATEMENTS

Large programs are often written as several separately assembled smaller programs. At execution time, it is necessary for these separately assembled programs to establish communication links. The linking processes (see the Program Development Tools manual) use information from the following program-linking statements to assign final addresses and/or data values to be used by the separately assembled procedures (i.e., programs) common to a single assembly language program:

o CALL
o CTRL
o XDEF

The program-linking statements are described in detail later in this section.

CONDITIONAL ASSEMBLY-CONTROL STATEMENTS

Conditional assembly-control statements allow a comprehensive source program to be written to cover many situations. Then, during assembly, they can direct the Assembler to assemble or inhibit assembly of particular assembly language instructions (and/or groups of assembly language instructions) when specific conditions occur. The following statements provide the Assembler with information for conditional assembly:

o FAIL
o IF
o NULL

These statements are defined in detail later in this section.

ASSEMBLER CONTROL STATEMENTS

The remainder of this section lists and describes the Assembler control statements in alphabetical order. The descriptions include the expanded name of the statement, its source language format (including the label field, where it is permitted or required), a detailed description of what the statement does, and a description of each of its operands.

Information about the various symbolic names identified in the statements is contained in Section 2.

CALL

Instruction:
Call external procedure

Source Language Format:
[label]ΔCALLΔ[obj-mod-name.]entry[,arg1[,...,arg31]]

Description:
Initiates a transfer of control to a specified external subroutine.

The operands have the following meanings:

obj-mod-name.
If specified, it is the object text name of the external procedure; otherwise, it is assumed to have the same name as the entry point (entry).

entry
Identifies the entry point in the procedure to which control is transferred.

arg1,...arg31
If specified, provides addresses of arguments to be passed.

If the argument list is *not* included, the CALL statement is broken down by the Assembler as follows:

```
CTRL    LINK obj-mod-name
XLOC    entry
LAB     $B7,=1
LNJ     $B5,<entry
```

If the argument list *is* included, the CALL statement is broken down as follows:

```
CTRL    LINK obj-mod-name
XLOC    entry
LAB     $B7,$+$AF+3
LNJ     $B5,<entry
B       >$+n*$AF+1
DC      <arg1[,<arg2]...
```

The XLOC statement shown in the breakdowns provides a temporary label that is not entered into the Assembler's symbol table, and ceases to exist after the LNJ instruction is executed. The term n, shown in the B-instruction in the second breakdown is an internally computed constant equal to the number of arguments specified in the CALL statement; this makes it possible for the Assembler to branch around the DC statement(s).

Additional information about calling external procedures can be found in the Software Overview and System Conventions manual.

CLST

Instruction:
Conditional Listing

Source Language Format:

[label]ΔCLSTΔint-val-expression

Description:
If the internal value expression is $\geqslant 0$, the CLST statement does not appear in the assembly listing. If the internal value expression is < 0, the CLST statement appears in the assembly listing with an error flag (Z-conditional assembly error). The comment field may be used to provide additional information concerning the error. The label of a CLST statement is not entered into the Assembler's symbol table.

COMM

Instruction:
Define common block

Source Language Format:
[label]ΔCOMMΔint-val-exp

Description:
Allows you to define a common block compatible with FORTRAN common areas.

The label field and operands have the following meanings:

label
: If specified, the common area is given that name; otherwise, it is unlabeled (i.e., blank) common, and is given the symbolic name $COMM (by implication).

int-val-exp
: Specifies the size (in words) of the common area. The Linker (see the Program Development Tools manual) assigns all common blocks with the same name to the same memory area regardless of the memory location in the source program at which they are defined (i.e., the COMM statement does not alter the Assembler's location counter.)

  int-val-exp is an internal value expression (see Section 1), and must be defined prior to the occurrence of this COMM statement. It must not contain a forward reference. Elements in a common block can be referenced by the name of the common block plus the element's displacement within the block.

CTRL

Instruction:
Pass control information to Linker

Source Language Format:

ΔCTRLΔcommand-line

Description:
Provides a method of passing Linker commands from the source program to the Linker (see the Program Development Tools manual for a description of the Linker).

The operand has the following meaning:

command-line
: Specifies data to be passed verbatim to the Linker as part of the program's object text (i.e., it is not verified by the Assembler).

DC

Instruction:
Define constant(s)

Source Language Format:

$$[\text{label}]\Delta\text{DC}\Delta \left\{ \begin{array}{l} < \left\{ \begin{array}{l} \text{location-expression} \\ \left\{\begin{array}{l}+\\-\end{array}\right\} \text{temporary-label} \end{array}\right\} \\ [=]\,\text{string-constant} \\ [=]\,\text{arithmetic-constant} \\ [=]\,\text{internal-value-label} \\ [=]\,\text{external-value-label} \\ [=]\,\text{internal-value-expression} \\ [=]\,\text{complex-label} \end{array} \right\} [,\ .\ .\ .\ ]$$

Description:
Defines data to be included in the object text. The Assembler interprets the constants, converts them to the proper binary representation, and assigns them to successive memory locations at the exact point at which the DC statement appears in the source program.

The operands have the following meanings:

$$< \left\{ \begin{array}{l} \text{location-expression} \\ \left\{\begin{array}{l}+\\-\end{array}\right\} \text{temporary-label} \end{array}\right\}$$

Causes a 1- or 2-word address pointer, as appropriate, to be allocated.

[=] string-constant
: Is padded, if necessary, to make an integral number of words; the padded value is allocated to memory.

[=] arithmetic-constant
: Causes a 1- or 2-word real binary number to be allocated.

[=] internal-value-label
[=] external-value-label
[=] internal-value-expression
: Causes a 1-word binary integer to be allocated.

[=] complex-label
Processed as described above for:

$$<\left\{\begin{array}{l}\text{location-expression}\\ \left\{\begin{array}{c}+\\-\end{array}\right\}\ \text{temporary-label}\end{array}\right\}$$

external-value-label, or internal-value-expression, depending on whether the label has been equated to a direct immediate memory form of addressing or an internal value immediate operand form of addressing. No other form of addressing can be used.

Detailed descriptions of the various types of labels, constants, and expressions can be found in Section 2 (e.g., internal-value-label is described under "Labels," string-constant is described under "Constants," and location-expression is described under "Expressions").

END

Instruction:
End of program

Source Language Format:
ΔENDΔprogram-name [,internal-location-expression]

Description:
Identifies the end of the assembly language program. This Assembler control statement must be the last statement in every assembly language source program.

The operands have the following meanings:

program-name
Must be the same program name specified in the source program's TITLE statement.

internal-location-expression
If specified, it identifies the program's normal entry point. (See "Expressions" in Section 2 for a description of internal location expressions.)

EQU

Instruction:
Equate

Source Language Format:

$$\text{label}\Delta\text{EQU}\Delta\left\{\begin{array}{l}\text{location-expression}\\ \text{internal-value-expression}\\ \text{address-expression}\\ \text{complex-label}\\ \text{identifier}\\ \text{fixed-point-constant}\end{array}\right\}$$

Description:
Assigns the value identified in the operand field, together with all of its associated attributes, to the label.

The operands have the following meanings:

fixed-point-constant
location-expression
internal-value-expression
   The label is treated by the Assembler as the same type as the operand (see "Expressions" in Section 2).

address-expression
complex-label
   The label is treated as a complex type (see "Expressions" and "Labels" in Section 2).

NOTES: 1. The address expression cannot be a hexadecimal string constant as defined under "Immediate Memory Addressing" in Section 5.
2. Complex labels cannot contain external or common references.

identifier
   The label is treated as an identifier that is equivalent to this one (see "Identifiers" in Section 2).

FAIL

Instruction:
Identifies a statement that should never be assembled.

Source Language Format:
[label]ΔFAIL

Description:
If the FAIL statement is assembled, an Assembler error flag (Z-conditional assembly error) is generated.
The FAIL statement is used in conditional assemblies to ensure that the prevailing conditions are logically consistent.

If the statement is labeled, the label is *not* entered into the Assembler's symbol table; as a result, it can be referred to only by a preceding IF statement.

IF

Instruction:
Conditional skip

Source Language Format:

$$[\text{label}]\Delta\text{IF}\left\{\begin{matrix}\text{OD}\\ [\text{N}]\left\{\begin{matrix}\text{P}\\ \text{N}\\ \text{Z}\end{matrix}\right\}\\ \text{EV}\end{matrix}\right\}\Delta\text{int-val-expression,int-loc-label}$$

Description:
If the specified condition is met, the Assembler skips subsequent statements until the label is encountered; otherwise, the next sequential instruction is processed.

The opcode is interpreted as follows:

IFP
: Skip to int-loc-label if int-val-expression is positive.

IFNP
: Skip to int-loc-label if int-val-expression is not positive.

IFN
: Skip to int-loc-label if int-val-expression is negative.

IFNN
: Skip to int-loc-label if int-val-expression is not negative.

IFZ
: Skip to int-loc-label if int-val-expression is zero.

IFNZ
: Skip to int-loc-label if int-val-expression is not zero.

IFOD
: Skip to int-loc-label if int-val-expression is odd.

IFEV
: Skip to int-loc-label if int-val-expression is even.

The operands have the following meanings:

int-val-expression
: Internal value expression (see "Expressions" in Section 2); forward references are not permitted.

int-loc-label
: Internal location label (see "Labels" in Section 2) identifying the location of the next statement or instruction to be processed by the Assembler if the condition is met.

: If a label is specified, it is *not* entered in the Assembler's symbol table; as a result, it can be referred to only by a preceding IF statement.

LIST

Instruction:
List following source statements

Source Language Format:

ΔLIST

Description:
Causes subsequent assembly language instructions and Assembler control statements to be included on the assembly listing. Listing of the statements continues until the end of the program or until an NLST Assembler control statement is encountered.

NLST

Instruction:
Inhibit listing of following source statements

Source Language Format:
ΔNLST

Description:
Prevents subsequent assembly language instructions and Assembler control statements from being included in the assembly listing. Listing of the statements continues to be inhibited until the end of the program or until a LIST Assembler control statement is encountered.

This statement overrides the use of * or / comment source statements (see Section 3).

NULL

Instruction:
No effect; processing continues

Source Language Format:
[label]ΔNULL

Description:
Has no effect on the assembly process.

This Assembler control statement is commonly used to define a label referred to by an IF statement. Processing continues with the next sequential instruction.

If the statement is labeled, the label is *not* entered into the Assembler's symbol table; as a result, it can be referred to only by an IF statement.

ORG

Instruction:
Origin

Source Language Format:

$$[\text{label}]\Delta\text{ORG}\Delta\left\{\begin{matrix}\text{common-location-expression}\\ \text{internal-location-expression}\end{matrix}\right\}$$

Description:
Assigns the attributes and value of the operand to the location counter (i.e., if the operand is a common location expression, the location counter type attribute is set to common; if the operand is an internal location expression, the location counter type attribute is internal). The initial value of the Assembler's location counter is internal location 0.

The label field and operands have the following meanings:

label
    If specified, the label will be assigned the value contained in the location counter *before* the new value is stored in the location counter.

common-location-expression

Sets the location counter type attribute to common and sets the location counter value to the specified offset in the common block. Temporary labels cannot be defined while the location counter has the common attribute.

internal-location-expression

Sets the location counter type attribute to internal and sets the location counter to the specified value of the location expression (see Section 2 for a description of common location and internal location expressions). Regardless of the type attribute of the expression specified in the operand, it must not contain a forward reference.

RESV

Instruction:
Reserve main memory space

Source Language Format:
[label]ΔRESVΔint-val-expa[,int-val-expb]

Description:
Reserves space in main memory for use by the object text program (generated by the Assembler) as work or storage space.

The label field and operands have the following meanings:

label

If specified, the reserved area is given that name.

int-val-expa

This is an internal value expression (see Section 2) that specifies the size (in words) of the reserved area. It must not contain a forward reference.

int-val-expb

If specified, it is an internal value expression (see Section 2) specifying the initial value to which each word in the reserved area is initialized when the object text program is loaded. If this operand is not specified, the contents of the reserved area are undefined.

TEXT

Instruction:
Allocate space for text

Source Language Format:
[label]ΔTEXTΔstring-constant[,string-constant[,...]]

Description:
Causes the Assembler to allocate the binary representation of the successive string constants concatenated into the fewest number of words (i.e., packed). The Assembler inserts "slack bits" (0's) between successive operands, as necessary, to ensure that each ASCII string begins at a bit position that is a multiple of 8, and that each hexadecimal string constant begins at a bit position that is a multiple of 4; if the last word occupied by the concatenated string is not full, slack bits are added to fill it. (String constants are described in Section 2.)

TITLE

Instruction:
Start of program

Source Language Format:
ΔTITLEΔprogram-name [,rev-number] [Δpage-header]

Description:
Identifies the beginning of the assembly language source program. This statement is required.

The operands have the following meanings:

program-name
: Name by which the source program can be referred to. The name must conform to the following rules:

1. One through six characters (A through Z, 0 through 9, $ or – (underscore).
2. First character must be one of the following:
   a. $
   b. A, B ... Z

rev-number
: Optional operand identifying the revision number of the program. It must be an ASCII string constant of one through eight characters in length.

page-header
: Optional comment line that will appear at the top of each page in the assembly listing (together with the revision number). Up to 20 characters are permitted.

XDEF

Instruction:
External label definition

Source Language Format:

$$\Delta\text{XDEF}\Delta \left\{ \begin{array}{l} \text{label} \\ \left(\text{label}, \left\{ \begin{array}{l} \text{int-loc-exp} \\ \text{int-val-exp} \end{array} \right\}\right) \end{array} \right\} [\ .\ .\ .\ ]$$

Description:
Identifies labels to be made available to external procedures. These labels can then be referred to through XLOC and XVAL statements in the external procedures. The occurrence of a label in an XDEF statement does not define that label for use elsewhere within that program (the label is not entered into the Assembler's symbol table).

The operands have the following meanings:

label
: Identifies a label, defined elsewhere in the source program, as an internal location label or internal value label (see Section 2), that can be referred to by a separately assembled program through an XLOC or XVAL statement.

$\left(\text{label, } \left\{ \begin{array}{l} \text{int-loc-exp} \\ \text{int-val-exp} \end{array} \right\} \right)$

int-loc-exp and int-val-exp are internal location or internal value expressions, respectively, which are evaluated by the Assembler, with the resulting value and type being associated with the label. The label can be referred to by a separately assembled program through an XLOC or XVAL statement.

Regardless of which form of the operands is used, the Assembler evaluates the label and generates a type and value attribute to be associated with the label. The results of the evaluation are passed to the Linker with the object text for use during the linking process (see the Program Development Tools manual).

NOTES: 1. It is not necessary for all labels identified through the XDEF statement to be referred to by an external program.
2. If a label is not identified to an external procedure by an XDEF statement, the label can be defined at link time by the LDEF command to the Linker.

XLOC

Instruction:
Define external locations to be referenced

Source Language Format:
ΔXLOCΔlabela [,labelb] ...

Description:
Identifies labels associated with locations in programs assembled separately from this program (i.e., external procedures), but used in this program.

The external program must identify the labels in an XDEF Assembler control statement.

The operands have the following meanings:

label
 Identifies the external location label(s) (see Section 2) used in this program.

XVAL

Instruction:
Define external values to be referenced

Source Language Format:
ΔXVALΔlabela [,labelb] ...

Description:
Identifies labels associated with values in programs assembled separately from this program (i.e., external procedures), but used in this program.

The external program must identify the labels in the XDEF Assembler control statement.

The operands have the following meanings:

label

Identifies the external value label(s) (see Section 2) used in this program.

SECTION 5

ASSEMBLY LANGUAGE INSTRUCTIONS

The BES assembly language instruction set provides the means by which you can write your source programs. These assembly language instructions, which are assembled into object text, enable you to perform the following types of operations:

- o Arithmetic
- o Boolean
- o Branching
- o Comparison
- o Controlling
- o Input/Output
- o Loading
- o Modification
- o Shifting
- o Storing
- o Swapping

The following paragraphs identify which of the assembly language instructions are included in each of the above operations. However, detailed information about each of the instructions is contained in the alphabetical list of instructions later in this section. In addition to identifying the assembly language instructions by operation, they are also listed by type (e.g., double operand). The various types can be distinguished not only by their op codes, but by their formats; therefore, the valid format for each type of instruction is included in the description of each type of instruction. However, the detailed format of each instruction is not shown, since the format used must conform to that described in Section 3.

ARITHMETIC OPERATIONS

The following assembly language instructions perform arithmetic operations (Add, Subtract, Multiply, Divide):

ADD	INC
ADV	MLV
CAD	MUL
DEC	NEG
DIV	SUB

BOOLEAN OPERATIONS

Boolean operations (Inclusive OR, Exclusive OR, AND) are provided through the following assembly language instructions.

AND	OR	XOH
ANH	ORH	XOR
CPL		

BRANCH OPERATIONS

The following instructions exist to support branching operations (Branch if ..., Branch unconditionally). This subset comprises following:

B	BEVN	BLEZ
BAG	BEZ	BLZ
BAGE	BG	BNE
BAL	BGE	BNEZ
BALE	BGEZ	BNOV
BBF	BGZ	BODD
BBT	BINC	BOV
BCF	BIOF	BSE
BCT	BIOT	BSU
BDEC	BL	NOP
BE	BLE	

COMPARE OPERATIONS

The following assembly language instructions perform the comparison operation (Compare X to Y):

CMB	CMR
CMH	CMV
CMN	CMZ

CONTROL OPERATIONS

Control instructions affect the flow of an assembly language program. They provide a means of entering trap handlers, starting and stopping hardware clocks, passing control to system service routines or external procedures, and jumping. This subset comprises the following:

BRK	LNJ	WDTF
ENT	MCL	WDTN
HLT	RTCF	
JMP	RTCN	
LEV	RTT	

INPUT/OUTPUT OPERATIONS

The following assembly language instructions are provided to support the input/output operations:

IO	IOH	IOLD

LOAD OPERATIONS

Load operations are provided through the following instructions:

LAB	LBT	LDR
LB	LDB	LDV
LBC	LDH	LLH
LBS	LDI	RSTR

MODIFY OPERATIONS

Modification (Clear Memory, Increment or Decrement the Contents of a Memory Location) operations are provided by the following assembly language instructions:

CL	CLH	MTM

SCIENTIFIC INSTRUCTIONS

The following set of instructions executes on the (optional) Scientific Instruction Processor (SIP). These instructions manipulate data in floating-point format and utilize the scientific accumulator registers (Sn) and the scientific indicator register (SI) provided by the SIP.

SAD	SBLEZ	SCZD
SBE	SBLZ	SCZQ
SBEU	SBNE	SDV
SBEZ	SBNEU	SLD
SBG	SBNEZ	SML
SBGE	SBNPE	SNGD
SBGEZ	SBNSE	SNGQ
SBGZ	SBPE	SSB
SBL	SBSE	SST
SBLE	SCM	SSW

SHIFT OPERATIONS

Shift operations are achieved through the following assembly language instructions:

DAL	DOL	SCL
DAR	DOR	SCR
DCL	SAL	SOL
DCR	SAR	SOR

STORE OPERATIONS

The following assembly language instructions are available to store the contents of specific registers in main memory or other registers:

SAVE	STB	STR
SDI	STH	STS
SRM	STM	

SWAP OPERATIONS

Swapping (i.e., exchanging) is supported through the following:

SWB	SWR

ASSEMBLY LANGUAGE INSTRUCTION TYPES

In addition to identifying assembly language instructions by the operations they perform, they can be classified by type:

- o Branch-on-indicator (BI)
- o Branch-on-register (BR)
- o Double operand (DO)
- o Generic (GE)
- o Input/output (IO)
- o Shift (SHS and SHL)
- o Short-value-immediate (SI)
- o Single operand (SO)

Branch-on-Indicator (BI) Instructions

Branch-on-indicator (BI) instructions have the following source language format:

```
[label] ΔopcodeΔaddress-expression
```

The opcode identifies the I-register bit to be tested for a specific condition.

The address-expression identifies the address of the next instruction to be executed if the condition exists. It must specify one of the following addressing forms (see "Addressing Techniques" in this section):

- o Direct Immediate memory address
- o Direct P-relative
- o Short displacement

The BI instructions are included in the alphabetical list of assembly language instructions later in this section.

Branch-on-Register (BR) Instructions

Branch-on-register (BR) instructions have the following source language format:

$$[\text{label}]\Delta\text{opcode}\Delta \left\{\begin{array}{l}\text{R-register}\\ \text{integer-constant}\end{array}\right\}\text{,addr-expression}$$

The opcode identifies the R-register condition that is to be tested for the existence of a specific condition.

The first operand identifies the R-register to be tested. If an integer constant is specified, the assembler assumes that the integer is an R-register identifier.

The second operand specifies one of the following addressing forms (see "Addressing Techniques" in this section):

- o Immediate memory address (direct form only)
- o P-relative
- o Short displacement

See the alphabetical list of instructions later in this section for detailed descriptions of the BR instructions.

Double Operand (DO) Instructions

Double operand (DO) instructions have the following source language format:

$$[\text{label}]\Delta\text{opcode}\Delta \left\{\begin{array}{l}\text{R-register}\\ \text{B-register}\\ \text{M-register}\\ \text{S-register}\\ \text{integer-constant}\end{array}\right\}\text{,addr-expression [,mask]}$$

The opcode identifies the operation to be performed and the type of register that is required in the first operand.

The first operand identifies the register that contains one of the data elements to be used in the operation, as well as the register that is to contain the result. All of the registers, except for the S-register are hardware registers; the S-register is a software-simulated scientific register provided by the Floating-Point Simulator (see the Executive and Input/Output manual). If an integer constant is specified, the Assembler assumes that it refers to a register that is of the type required by the opcode.

The second operand specifies an address expression that gives the location of the other data element to be used in the operation. If an address expression is not specified, the second operand must be a complex label equated to an address expression. (See "Labels" in Section 2 for a description of complex labels, and "Addressing Techniques" in this section for a description of address expressions.)

The third operand is valid only for the Store Register Masked (SRM) instruction.

The alphabetical list of assembly language instructions later in this section provides detailed descriptions of each of the DO instructions.

Generic (GE) Instructions

Generic (GE) instructions are identifiable by the fact that they contain no operands, as follows:

[label]ΔopcodeΔ

All of the GE instructions perform controlling operations. The alphabetical list of instructions later in this section describes the GE instructions.

Input/Output (IO) Instructions

Input/output (IO) instructions have the following source language format:

[label]ΔopcodeΔaddress-expression,address-expression[,address-expression]

The opcode identifies the instruction as one of the following types:

- o Data and command I/O
- o Address and range output

The address expression in the first operand identifies the location from which a data word is transferred to the I/O bus, or the location to which a data word is transferred from the I/O bus.

The second operand address expression identifies the channel number and function code, or the location where this information can be found.

The third operand address expression is valid only for the input/output load (IOLD) instruction. It identifies the location of the word that contains the range. When this instruction is specified, the address expression in the first operand identifies the location of a byte of data to be transferred to the I/O bus.

Address expressions are described under "Addressing Techniques" in this section. The IO instructions are described in the alphabetical list later in this section.

Shift (SHS and SHL) Instructions

Shift (SHS and SHL) instructions have the following source language format:

$$\text{[label]}\Delta\text{opcode}\Delta \left\{ \begin{matrix} \text{R-register} \\ \text{integer-constant} \end{matrix} \right\} \text{,int-val-expression}$$

The opcode identifies the format, type and direction of the shift. The formats can be:

- o SHS - Shift short
- o SHL - Shift long

The valid types are:

- o Arithmetic
- o Open
- o Closed

The direction of the shift can be:

- o Right
- o Left

The first operand identifies the register (or register pair for long-precision shifts) containing the data to be shifted. For short-precision shifts, any R-register can be specified; for long-precision shifts, the R-register specified must be $R3, $R5, or $R7, with the preceding even-numbered register ($R2, $R4, or $R6, respectively) being implied. Use of an integer constant implies that the R-register with that number is specified.

The internal value expression (see Section 1) in the second operand specifies the number of bits to be shifted. For short-precision shifts, the count must be within the range 1 through 15; if 0 is specified, the system uses the value found in bits 12 through 15 of $R1. For long-precision shifts, the count must be within the range 1 through 31; if 0 is specified, the value in bits 11 through 15 of $R1 is used.

Detailed descriptions of the SHS and SHL instructions are included in the alphabetical list of instructions later in this section.

Short-Value-Immediate (SI) Instructions

Short-value-immediate (SI) instructions have the following source language format:

$$\text{[label]}\Delta\text{opcode}\Delta \left\{\begin{array}{l}\text{R-register}\\ \text{integer-constant}\end{array}\right\} \text{,[=]} \left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{int-val-expression}\\ \text{fixed-point-constant}\end{array}\right\}$$

The opcode identifies the operation to be performed.

The first operand specifies an R-register that contains one of the data elements to be operated upon and receives the result of the operation. If an integer constant is used, the corresponding R-register is assumed (i.e., X'5' implies R-register $R5).

The second operand is a 1-byte (8-bit) value. If it is a string constant (see Section 2), it is treated as a half-word string; if the length of the string is greater than 8 bits, low order (i.e., the rightmost) bits are truncated; if less than 8 bits, 0's are appended to the low order bit positions. If the second operand is not a string constant, the value is considered to be numeric within the range -128 to +127.

Integer constants, string constants, internal value labels, internal value expressions, and fixed point constants are described in Section 2. The SI instructions are described in detail in the alphabetical list later in this section.

Single Operand (SO) Instructions

Single operand (SO) instructions have the following source language format:

$$\text{[label]}\Delta\text{opcode}\Delta\text{addr-expression} \left[, \left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{external-value-label}\\ \text{int-val-expression}\\ \text{fixed-point-constant}\end{array}\right\} \right]$$

The opcode identifies the operation to be performed.

The first operand address expression (see "Addressing Techniques" in this section) identifies the location of the data element to be operated upon.

The second operand is valid only for the Save (SAVE) and Restore (RSTR) instructions. It specifies the value of a one-word mask that indicates which registers are to be saved and restored. Integer constants, string constants, internal value labels, external value labels, internal value expressions, fixed point constants are described in Section 2.

The SO instructions are described in the alphabetical list of assembly language instructions later in this section.

ADDRESSING TECHNIQUES

Many of the assembly language instructions require the use of address expressions in their operand fields. Address expressions can take any of the following forms:

- o Register addressing
- o Immediate memory addressing
- o Immediate operand addressing
- o P-relative addressing
- o B-relative addressing
- o Short displacement addressing
- o Special addressing
- o Interrupt vector addressing

Any of these addressing forms can be used to specify the location of data to be used in an operation. Furthermore, the data can be referenced directly, indirectly, via indexing, or by utilizing the push/pop feature.

Register Addressing

Register addressing is specified when a value or address is contained in a register. This form of address expression is specified as follows:

```
= $Rn   =$Bn   =$Sn
```

=$Rn and =$Bn are mutually exclusive; i.e., some instructions permit the use of =$Rn and others allow =$Bn (the descriptions of the various instructions identify which is valid for that instruction). The =$Rn form is generally used in those instructions that require some data to be contained in the register. The =$Bn form is valid for those instructions that expect to find an address in the register. The =$Sn form addresses the scientific accumulator registers.

The following examples illustrate register addressing. In the examples, assume that $B5 contains the address 3FFF, that $B3 contains the address 12A4, that $R5 contains the value 2012, and that $R7 contains the value 00ED.

Example 1:

```
ADD X'7',=$R5
```

In this example, the contents of $R5 are added to the contents of $R7, and the result (20FF) is stored in $R7. Since this instruction requires that the first operand specify an R-register, the Assembler assumes that the integer constant refers to $R7 and generates code to executes the instruction accordingly.

Example 2:

```
LDB $B5,=$B3
```

In this example, the address stored in $B3 is loaded into $B5.

Immediate Memory Addressing (IMA)

Immediate memory addressing is specified when a value or address is contained in a main memory location. This form of addressing allows you to reference a location directly, indirectly, and through indexing (direct or indirect). Depending on how you wish to reference the memory location, you can specify immediate memory addressing as follows:

$$< \left\{ \begin{matrix} \text{location-expression} \\ \left\{ \begin{matrix} + \\ - \end{matrix} \right\} \text{temporary-label} \end{matrix} \right\} \qquad \text{– Direct IMA}$$

$$< \left\{ \begin{matrix} \text{location-expression} \\ \left\{ \begin{matrix} + \\ - \end{matrix} \right\} \text{temporary-label} \end{matrix} \right\} \qquad \text{– Indirect IMA}$$

$$< \left\{ \begin{matrix} \text{location-expression} \\ \left\{ \begin{matrix} + \\ - \end{matrix} \right\} \text{temporary-label} \end{matrix} \right\} .\$R \left\{ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} \right\} \qquad \text{– Indexed Direct IMA}$$

$$< \left\{ \begin{matrix} \text{location-expression} \\ \left\{ \begin{matrix} + \\ - \end{matrix} \right\} \text{temporary-label} \end{matrix} \right\} .\$R \left\{ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} \right\} \qquad \text{– Indexed Indirect IMA}$$

When a source instruction indicating immediate memory addressing is assembled, the actual *address* of the operand is assembled into the operand field. Therefore, any internal, external, or common location expression is a valid operand. In contrast, P-relative addressing (defined later in this section) creates object code in which the *displacement* from the current instruction to the operand is assembled into the operand field.

Direct Immediate Memory Addressing

Direct immediate memory addressing makes it possible for you to specify explicitly the location of the data or address to be used in an operation.

The following example illustrates the use of this form of immediate memory addressing. In the example, assume that INTLB1 is an internal location label at location 20F4 and that location contains the address 0F0B, and that $B3 contains the address 111A.

Example:

```
    LDB $B3,<INTLB1
```

In this example, the contents (0F0B) of location 20F4 (specified by INTLB1) are loaded into $B3, replacing its current contents.

Figure 5-1 illustrates how the instruction in the example is stored in memory and how the data is found.

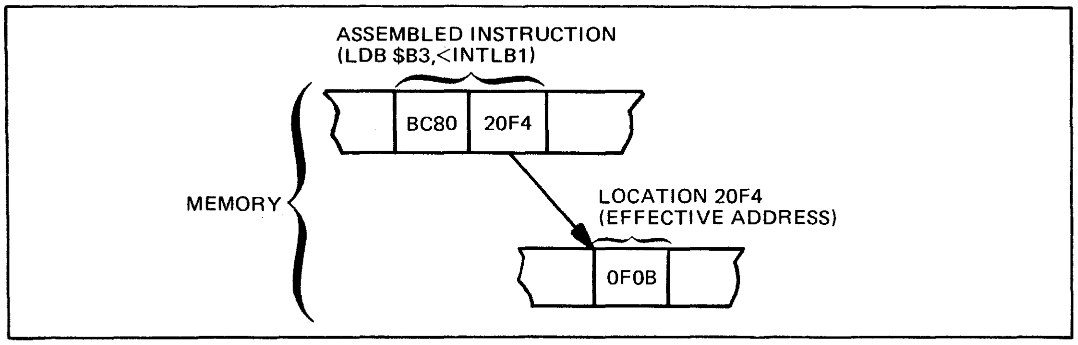


Figure 5-1. Direct Immediate Memory Addressing

Indirect Immediate Memory Addressing

This form of immediate memory addressing is available when you want to refer to a location whose address is stored in another location.

The following example illustrates the use of this form of immediate memory addressing. Assume that $C is a temporary label, whose next definition is at location 30A2 and that location contains the address 100C. Further, assume that location 100C contains the value 0F2C, and that $R6 contains the value 10D3.

Example:

```
ADD $R6,*<+$C
```

In this example, the system goes to the location specified at location 30A2 (identified by +$C, the + indicating that a forward reference is involved), which is 100C. It then adds the value found there (i.e., 0F2C) to the contents of $R6, and stores the result (1FFF) in $R6.

Figure 5-2 illustrates how the instruction appears in memory and how the data used in the instruction is found.

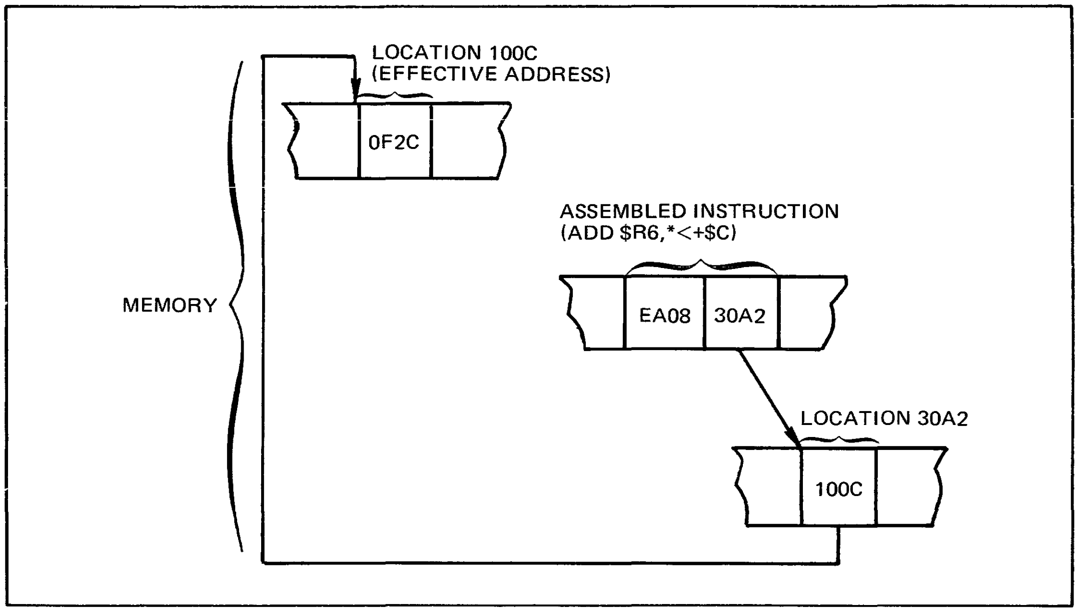


Figure 5-2. Indirect Immediate Memory Addressing

Indexed Direct Immediate Memory Addressing

Indexed direct immediate memory addressing is available when you want to refer to data or an address that has a known number of words beyond a specific location.

The following example illustrates the use of this form of immediate memory addressing. Assume that TABLE1 is an internal location label at location 2000, and that word 3 in the table is the address of an error routine. Also, assume that $R3 contains the value 0003.

Example:

```
LDB $B1,<TABLE1.$R3
```

In this example, the system adds the contents of the index register ($R3) to the address of TABLE1 (i.e., 2000). Then the contents of that location (i.e., the address of the error routine) are loaded into $B1.

Figure 5-3 illustrates how the instruction appears in memory and how it locates the effective address.

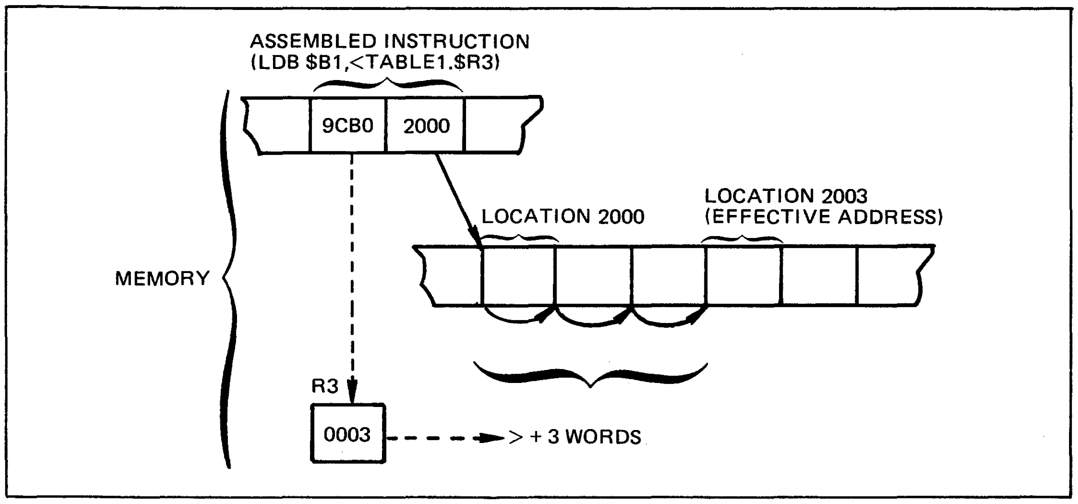


Figure 5-3. Indexed Direct Immediate Memory Addressing

Indexed Indirect Immediate Memory Addressing

This form of immediate memory addressing combines the feature of indirect immediate memory addressing with indexing to generate the location of the data or address to be used in an operation.

The following example illustrates the use of this form of immediate memory addressing. In the example, assume that TABL1A is an internal location label at location 20AA and that that location contains the address 30FF. Also assume that $R1 contains the value 0F00, that $R2 contains the value 401A, and that location 3FFF contains the value 3D91.

Example:

```
ADD $R2,*<TABL1A.$R1
```

In this example, the contents of $R1 (i.e., 0F00) are added to the contents of location 20AA (i.e., 30FF) to obtain the effective address of the data to be used in the operation. Then, the data found at location 3FFF (0F00 + 30FF) is added to the contents of $R2 as follows: 3D91 + 401A = 7DAB. The result is then stored in $R2.

Figure 5-4 illustrates how the instruction appears in memory, and how the system locates the data to be used in the operation.

Immediate Operand Addressing

Immediate operand addressing makes it possible to specify a literal value or address as the address expression. Depending on the type of instruction, this form of addressing must be specified in one of the following forms:

$= \left\{ \begin{array}{l} \text{internal-value-expression} \\ \text{location-expression} \end{array} \right\}$ (LDB, STB, SWB, CMB)

$= \left\{ \begin{array}{l} \text{hex-string-constant} \\ \text{floating-point-constant} \end{array} \right\}$ (SAD, SCM, SCZD, SDV, SLD, SML, SNGD, SSB, SST, SSW)

$= \left\{ \begin{array}{l} \text{internal-value-expression} \\ \text{external-value-label} \\ \text{fixed-point-constant} \end{array} \right\}$ (All others)

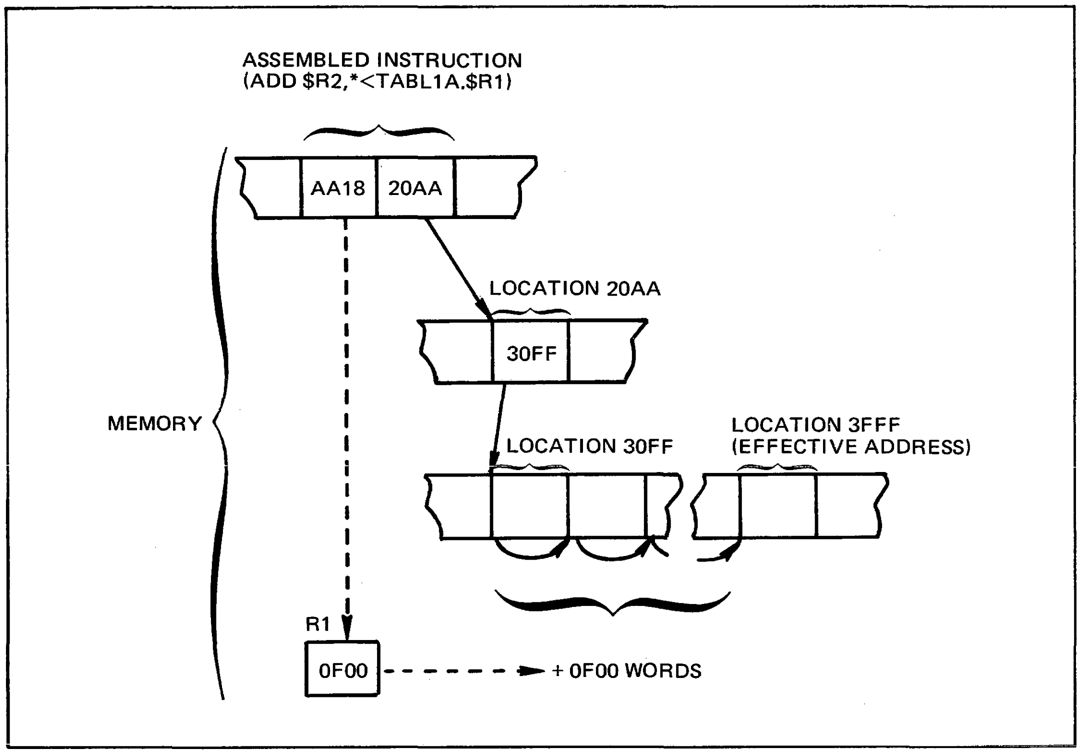
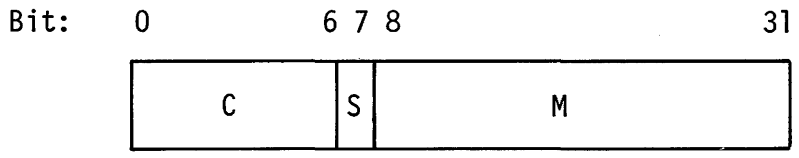


Figure 5-4. Indexed Indirect Immediate Memory Addressing

The hex-string-constant form must specify a hexadecimal string constant that provides the following information for the scientific instructions:



c Characteristic (excess power-of-16 exponent) of the mantissa.
s Sign (0 = positive; 1 = negative) of the mantissa.
m Magnitude of the mantissa.

The following examples illustrate the use of the immediate operand addressing form of addressing. Assume that $S1 is the scientific accumulator register and that it contains the value 84130000 (indicating a floating-point number with a value of 19.000000, that $R5 contains the value 300A, and that INTVAL is the label of an internal value expression that is equated to 1FF3.

Example 1:

```
SAD $S1,=Z'8280000A'
```

In this example, the floating-point value specified by the hexadecimal string constant (i.e., 8.000010_{10} is added to the floating-point value stored in $S1 (i.e., 19.00000), and the result is stored in $S1.

Figure 5-5 illustrates how the above example is stored in memory and how it determines the effective address.

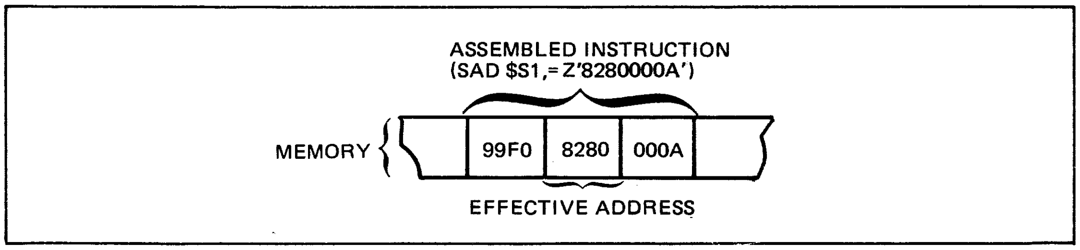


Figure 5-5. Immediate Operand Addressing-Scientific Instruction

Example 2:

```
ADD $R5,=INTVAL
```

In this example, the value equated to the internal value label INTVAL (i.e., 1FF3) is added to the value contained in $R5 (i.e., 300A), and the result (4FFD) is stored in $R5.

Figure 5-6 illustrates how the above ADD instruction is stored in memory and how it finds the effective address.

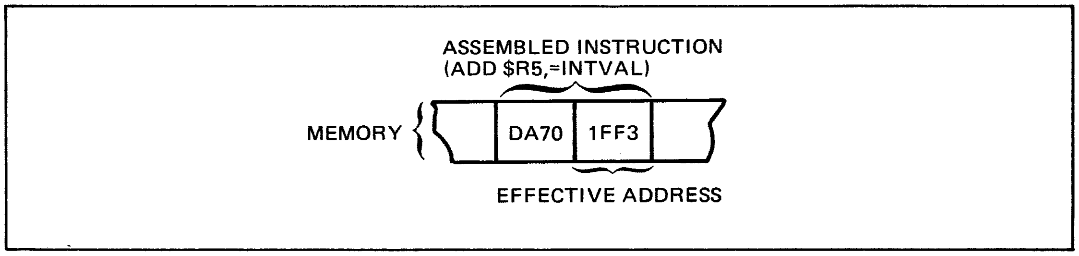


Figure 5-6. Immediate Operand Addressing

P-Relative Addressing

P-relative addressing is available for those situations in which you want to reference data or an address by indicating its (Assembler-calculated) displacement from the current location (i.e., the location of the currently executing instruction). This form of addressing allows you to reference a location directly or indirectly. Depending on which way you want to reference a location, you can specify P-relative addressing as follows:

$$\left\{\begin{array}{l}\text{location-expression}\\ \left\{\begin{array}{c}+\\ -\end{array}\right\}\text{temporary-label}\end{array}\right\} \quad \text{– Direct P-Relative Addressing}$$

$$*\left\{\begin{array}{l}\text{location-expression}\\ \left\{\begin{array}{c}+\\ -\end{array}\right\}\text{temporary-label}\end{array}\right\} \quad \text{– Indirect P-Relative Addressing}$$

Direct P-Relative Addressing

This form of addressing is available when you want to specify a location relative to the contents of the P-register (i.e., the address of the currently executing instruction) directly.

The following example illustrates this form of P-relative addressing. In the example, assume that $R5 contains the value 3F10, and that INTLOC is a location label at location 1110, which contains the value 1E10.

Example:

SUB $R5,INTLOC

In this example, the contents of the location identified by INTLOC (1E10) are subtracted from the contents of $R5, and the result (2100) is stored $R5.

Figure 5-7 illustrates the above instruction in memory, and shows how it finds the effective address.

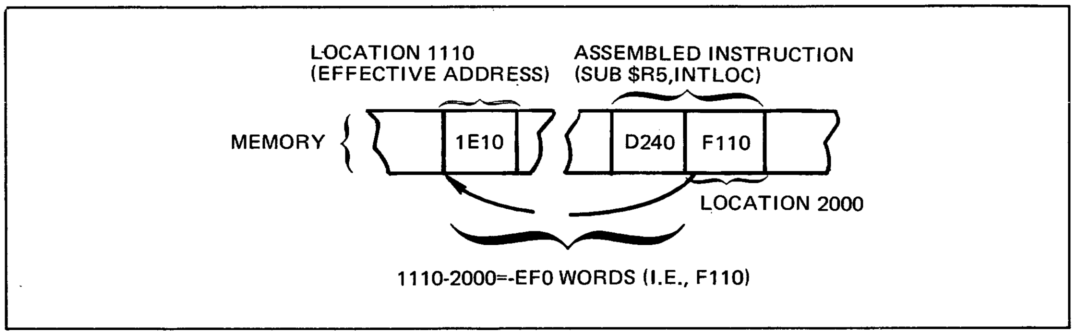


Figure 5-7. Direct P-Relative Addressing

Indirect P-Relative Addressing

Indirect P-relative addressing is similar to indirect immediate memory addressing.

The following example illustrates indirect P-relative addressing. In the example assume that $E precedes the current instruction, and that location that it identifies contains the address 3000; furthermore, assume that location 3000 contains the value 20AA, and that $R1 contains the value 4F44.

Example:

ADD $R1,*-$E

This instruction adds the contents of the location pointed to by location 3000 (i.e., 20AA) to the value contained in $R1, and stores the result (6FEE) in $R1.

Figure 5-8 shows how the instruction described above is stored in memory and how it locates the data to be used in the operation.

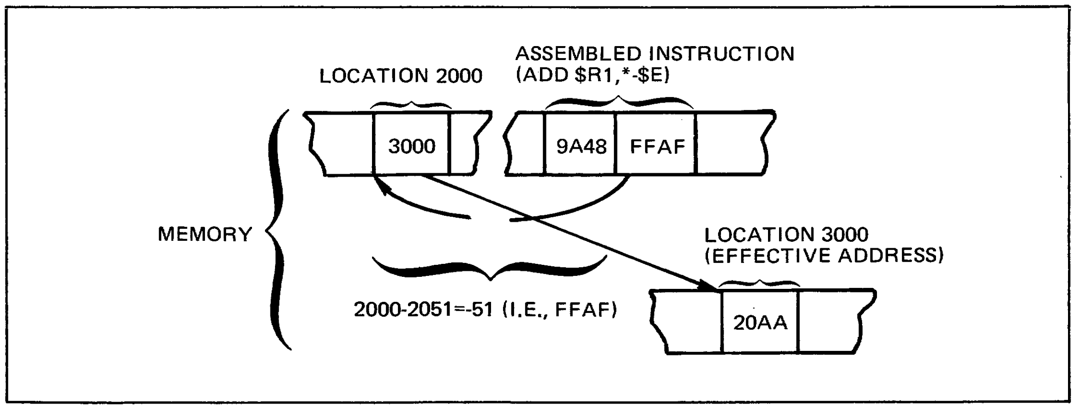


Figure 5-8. Indirect P-Relative Addressing

B-Relative Addressing

B-relative addressing is used when you want to reference through an address register (i.e., $B1, $B2,...$B7) a location that contains data or an address. This form of addressing can be used to reference a location directly, indirectly, through indexing, as a displacement, or through the push/pop feature.

The push/pop feature causes the system to automatically *decrement* the contents of the specified address or index register *before* executing the instruction, or automatically *increment* its contents *after* execution, as specified in the address expression.

Depending on the features you want to use in the address expression, B-relative addressing can take any of the following forms:

$Bn	– Direct B-relative addressing
*$Bn	– Indirect B-relative addressing
$\$Bn.\$R\begin{Bmatrix}1\\2\\3\end{Bmatrix}$	– Indexed direct B-relative addressing
$*\$Bn.\$R\begin{Bmatrix}1\\2\\3\end{Bmatrix}$	– Indexed indirect B-relative addressing
$\$Bn.\begin{Bmatrix}\text{int-val-expression}\\\text{external-val-label}\end{Bmatrix}$	– Direct B-relative plus displacement addressing
$*\$Bn.\begin{Bmatrix}\text{int-val-expression}\\\text{external-val-label}\end{Bmatrix}$	– Indirect B-relative plus displacement addressing
-$Bn	– B-relative addressing with automatic decrement before execution (Push)
+$Bn	– B-relative addressing with automatic increment after execution (Pop)
$\$Bn.\text{-}\$R\begin{Bmatrix}1\\2\\3\end{Bmatrix}$	– Indexed direct B-relative addressing with automatic decrement of index register before execution (Push)
$\$Bn.+\$R\begin{Bmatrix}1\\2\\3\end{Bmatrix}$	– Indexed direct B-relative addressing with automatic increment of index register after execution (Pop)

The first four forms of B-relative addressing are similar to their immediate memory addressing counterparts, except that the location of the data or address to be used in the operation is contained in an address register rather than being expressed as a location expression or label.

The next two are similar to the P-relative forms of addressing. The last four utilize the push/pop feature, as defined above. However, the last two forms require that you identify $B1, $B2, or $B3 as the address register although use of +$R1, +$R2, or +$R3 causes the system to specify $B5, $B6, or $B7, respectively, when the instruction is stored in memory. As a result, when reading a memory printout, you must remember that although the stored instruction indicates that the $B5, $B6, or $B7 register was specified, in reality $B1, $B2, or $B3, respectively, was coded and their contents used in the generation of the effective address of the data or address used in the operation.

Direct B-Relative Addressing

This form of addressing is available when you want to use data or an address whose location is contained in an address register.

The following example illustrates direct B-relative addressing. In the example, assume that $B7 contains the address 20F2, and that $B2 contains the address 4FFF.

Example:

```
    LDB $B2,$B7
```

In this example, the contents of the address contained in $B7 are loaded into and replace the contents of $B2.

Figure 5-9 shows how the instruction in the example is stored in memory and how the effective address is found.

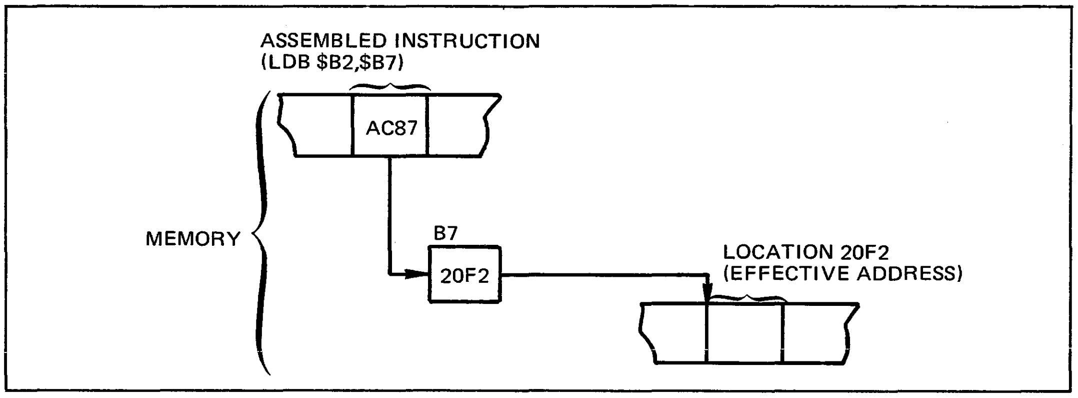


Figure 5-9. Direct B-Relative Addressing

Indirect B-Relative Addressing

Like indirect immediate memory addressing, this form of addressing is used when you want to use data or an address contained at a location whose address is pointed to by an address register.

The following example illustrates indirect B-relative addressing. In the example, $B3 contains the address 100F, address 100F contains address 302A, and address 302A contains the address 3FFF; furthermore, $B1 contains the address 1110.

Example:

```
    STB $B1,*$B3
```

In this example, the address 1110 is stored at location 302A, replacing the address that was contained there (i.e., 3FFF).

Figure 5-10 illustrates how the sample instruction is stored in memory and how it derives the effective address.

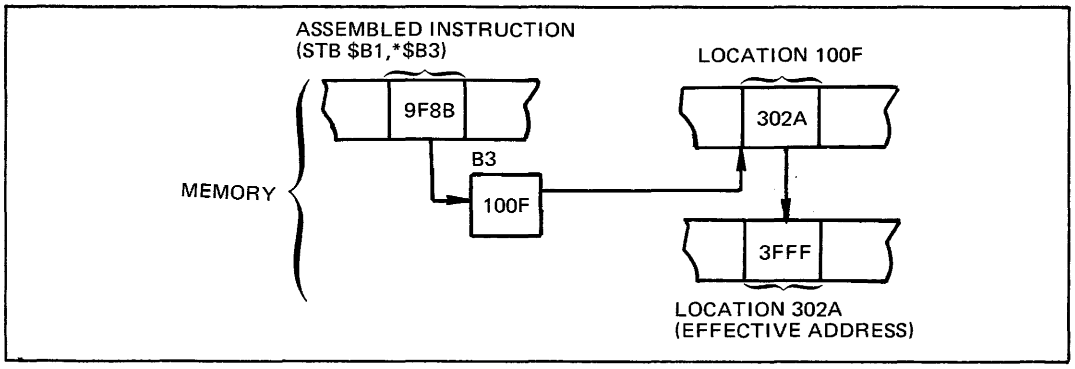


Figure 5-10. Indirect B-Relative Addressing

Indexed Direct B-Relative Addressing

This form of addressing, like indexed direct immediate memory addressing, uses an index register to compute the effective address of the data or address to be used in the operation. The contents of the index register are added to the contents of the address register to derive the location of the data or address to be included in the operation.

In the following example, which illustrates indexed direct B-relative addressing, $R3 contains the value 1110, $R1 contains the value 0002, $B5 contains the address 3FFD, and memory location 3FFF contains the value 9999.

Example:

```
ADD $R3,$B5.$R1
```

In this example, the system adds the contents of $R1 to the contents of $B5 to compute the address of the data to be used in the operation. The result is 3FFF (i.e., 3FFD + 2). The contents of location 3FFF are added to the contents of $R3, and the result (AAA9) is stored in $R3.

Figure 5-11 illustrates how the above example appears in memory.

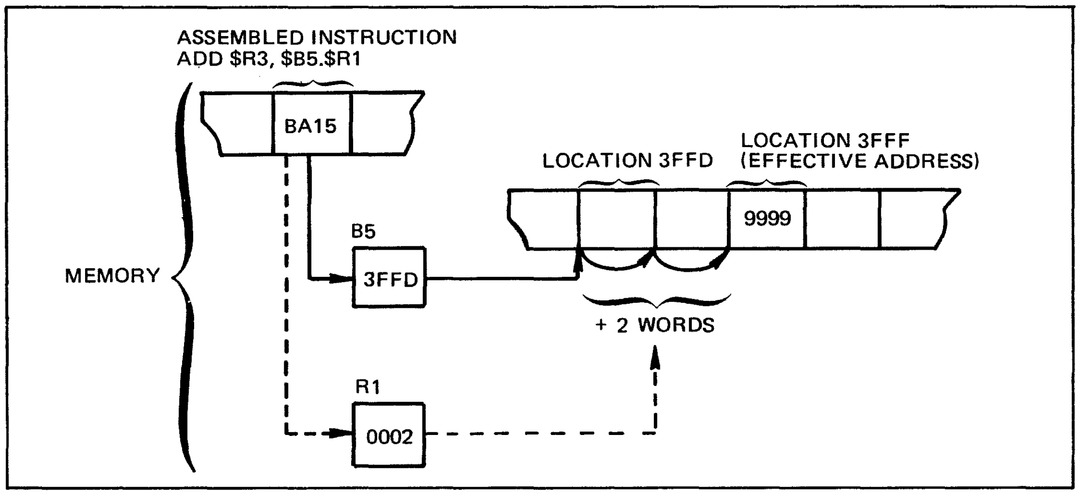


Figure 5-11. Indexed Direct B-Relative Addressing

Indexed Indirect B-Relative Addressing

This form of B-relative addressing is similar to indexed indirect immediate memory addressing. The contents of the index register are added to the contents of the location pointed to by the address register to obtain the effective address of the data to be used in the operation.

The following example illustrates this form of addressing. In the example, assume that $B5 contains the address 2022 and that that address contains the address 1000; also, assume that $R2 contains the value 40FF, that $R1 contains the value 001A, and that location 101A contains the value 1001.

Example:

```
ADD $R2,*$B5.$R1
```

In this example, the contents of $R1 (001A) are added to the contents of the location pointed to by $B5 (1000). The contents of the resulting location (101A) are added to the contents of $R2, and the result (5100) is stored in $R2.

Figure 5-12 illustrates how the sample instruction is stored in memory and how it derives the effective address.

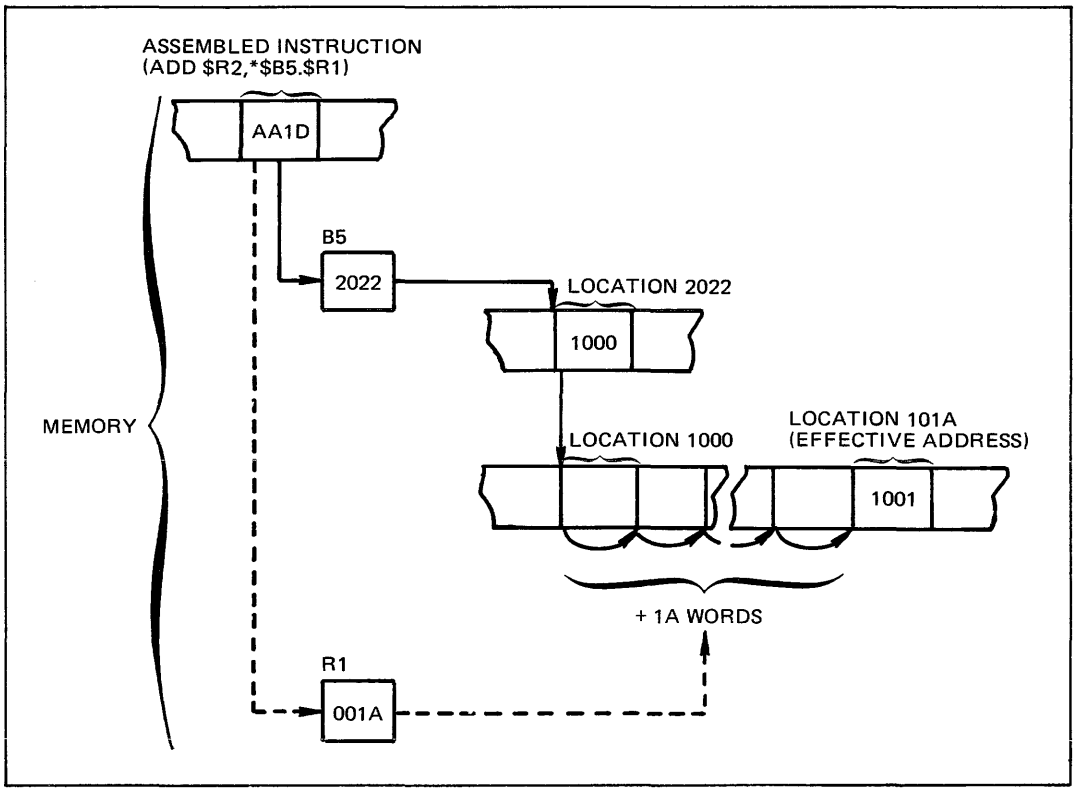


Figure 5-12. Indexed Indirect B-Relative Addressing

Direct B-Relative Plus Displacement Addressing

This form of addressing causes the system to compute the effective address by adding a specific value to the contents of an address register.

The following example illustrates this form of addressing. In the example, assume that XVAL2A is an external value label equated to the value 000A, that $B5 contains the address 2000, that memory location 200A contains the value 20ED and that $R6 contains the value 6DFE.

Example:

```
SUB $R6,$B5.XVAL2A
```

This instruction computes the effective address of the data to be used by adding 000A to the contents of $B5 (2000). It then subtracts the contents (20ED) of the effective address (200A) from the contents of $R6, and stores the result (4D11) in $R6.

Figure 5-13 shows how the above example is stored in memory and how it derives the effective address of the data.

Indirect B-Relative Plus Displacement Addressing

This form of addressing adds a displacement value to the contents of the specified address register. Then, the effective address is obtained by checking the contents of the location whose address is derived through the preceding operation.

In the following example of this form of addressing, EXP10 is an internal value expression equated to 0010, $B4 contains the address 30FF, location 310F contains the address 10FE, location 10FE contains the value 400D, and $R7 contains the value 1013.

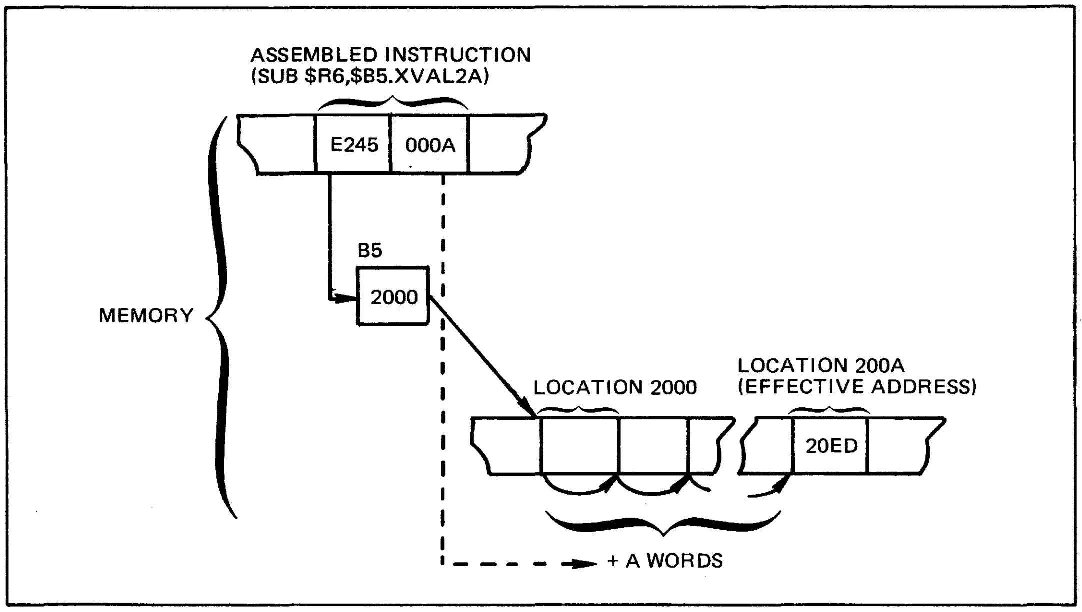


Figure 5-13. Direct B-Relative Plus Displacement Addressing

Example:

ADD $R7,$B4.EXP10

In this example, the displacement value 0010 is added to the contents of $B4 (i.e., 0010 + 30FF), producing the address 310F. Then, applying the indirection operator, the contents of the location 310F (i.e., 10FE) are used as a memory address. The value found at location 10FE (i.e., 400) is added to the contents of $R7. The result (5020) is stored in $R7.

Figure 5-14 illustrates how this form of addressing generates an effective address when stored in memory.

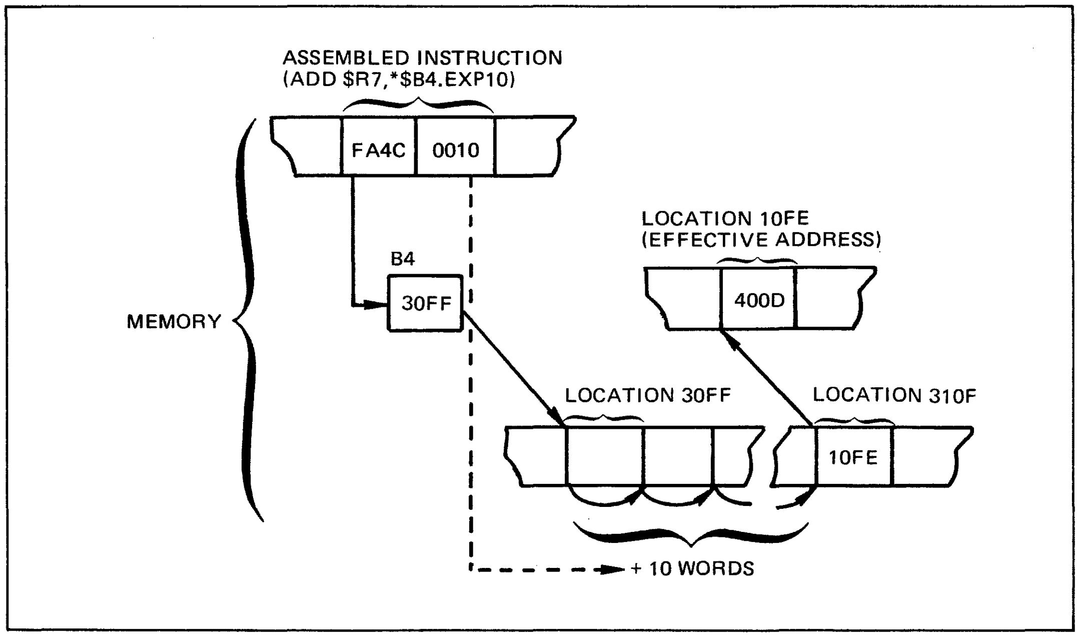


Figure 5-14. Indirect B-Relative Plus Displacement Addressing

B-Relative Push Addressing

This form of B-relative addressing causes the contents of the specified address register to be decremented by one *before* the effective address is formed. The new address in the register is the effective address of the location or data to be used in the operation.

In the following example, $R5 contains the value 30FF, $B5 contains the address 4011, and memory location 4010 contains the value 0001.

Example:

```
ADD $R5,-$B5
```

In this example, the contents of location derived by subtracting one from the address contained in $B5 are added to the contents of $R5, and the result (3100) is stored in $R5. The next time $B5 is used, it will contain the address 4010.

Figure 5-15 illustrates how the sample instruction described above is stored in memory and how it derives the effective address of the data to be used in the operation.

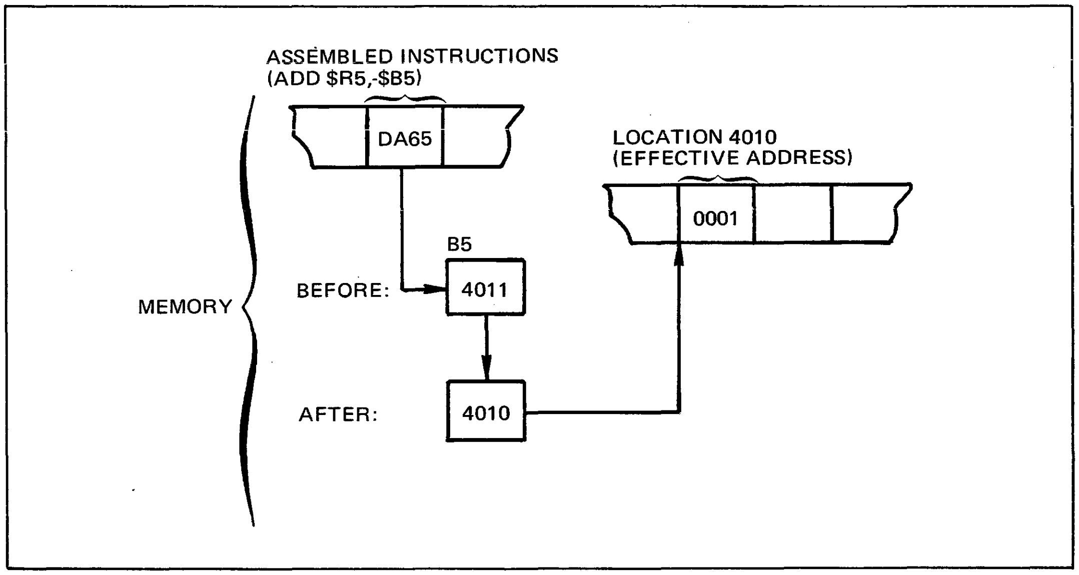


Figure 5-15. B-Relative Push Addressing

B-Relative Pop Addressing

This form of B-relative addressing causes the contents of the specified address register to be incremented by one *after* the effective address is formed.

In the following example, $R3 contains the value 222A, $B2 contains the address A000, and location A000 contains the value 0005.

Example:

```
ADD $R3,+$B2
```

In this example, the contents of location A000 are added to the contents of $R3, and the result (222F) is stored in $R3.

The address stored in $B2 is then incremented by one. The next time $B2 is used in an instruction, it will contain the address A001.

Figure 5-16 shows how the instruction above is stored in memory and how it derives an effective address.

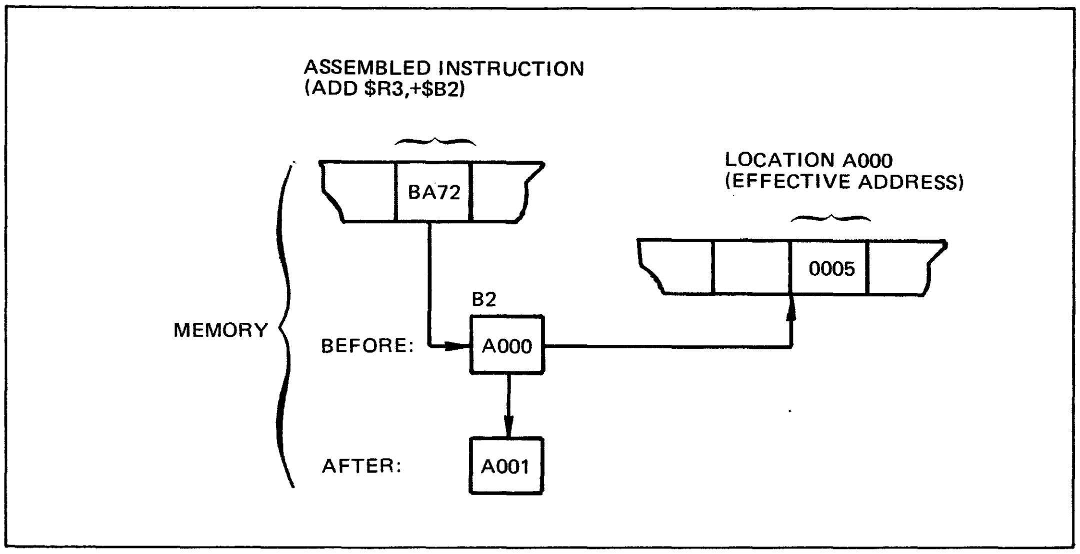


Figure 5-16. B-Relative Pop Addressing

Indexed B-Relative Push Addressing

This form of B-relative addressing decrements the contents of the specified index register by 1, then computes the effective address of the data or address to be used in the operation as described under "Indexed Direct B-Relative Addressing," above.

In the following example, $R1 contains the value 0003, $R2 contains the value 20F0, $B3 contains the address 20A0, and memory location 20A2 contains the value DF0F.

Example:

```
ADD $R2,$B3.-$R1
```

In this example, the effective address of the data to be used in the operation is derived by subtracting 1 from the contents of the index register, then adding the revised contents to the address contained in $B3. Then, the contents of the effective address are added to the contents of $R2 (i.e., 20F0 + DF0F), and the result (FFFF) is stored in $R2. The next time the index register $R1 is used, it will contain the value 0002.

When indexed B-relative push addressing is used, only address registers $B1, $B2, or $B3 can be specified in the address expression. Figure 5-17 illustrates how the sample instruction described above is stored in memory and how it derives the effective address of the data to be used in the operation.

Indexed B-Relative Pop Addressing

This form of B-relative addressing computes the effective address of the location or data to be used in the operation as described under "Indexed Direct B-Relative Addressing," in this section. After computing the effective address, the contents of the index register are incremented by 1.

In the following example of this form of B-relative addressing. $B3 contains the address 1000, $R2 contains the value 20A0, $R6 contains the value 2FFF, and location 30A0 contains the value 0001.

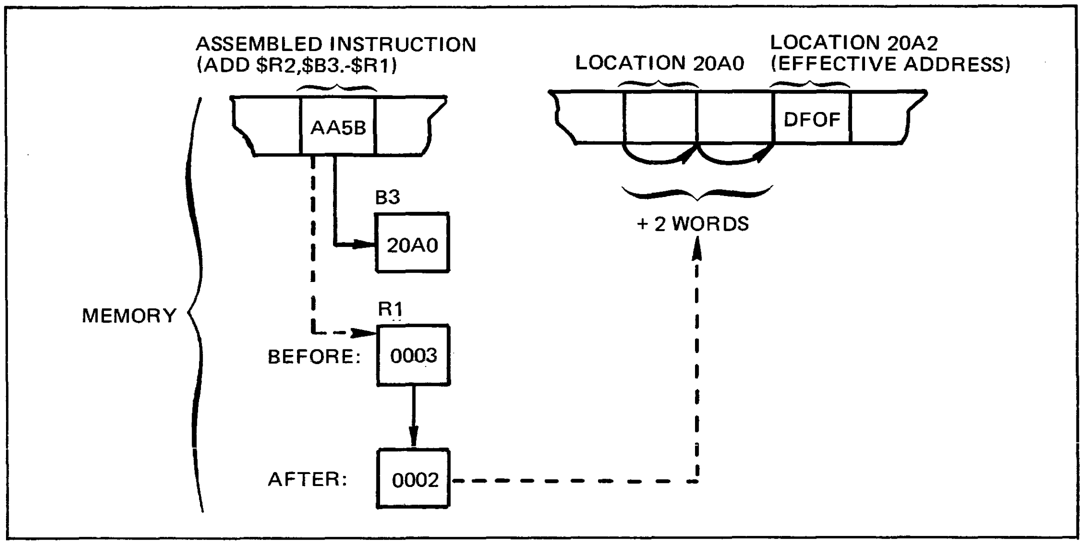


Figure 5-17. Indexed B-Relative Push Addressing

Example:

ADD $R6,$B3,+$R2

In this example, the effective address of the data to be added to the contents of $R6 is derived by adding the contents of the index register to the contents of $B3. The value found at that location (30A0) is then added to the contents of $R6, and the result (3000) is stored in $R6.

After the effective address is formed, the contents of the index register are incremented by 1. The next time the index register is used, it will contain the value 20A1.

When using B-relative pop addressing, only address registers $B1, $B2, or $B3 can be specified in the address expression. However, when stored in memory, the instruction will indicate $B5, $B6, or $B7, respectively, although the contents of the specified register are always used in the computation of the effective address.

Figure 5-18 illustrates how the sample instruction described above is stored in memory and how it derives the effective address of the data to be used in the operation.

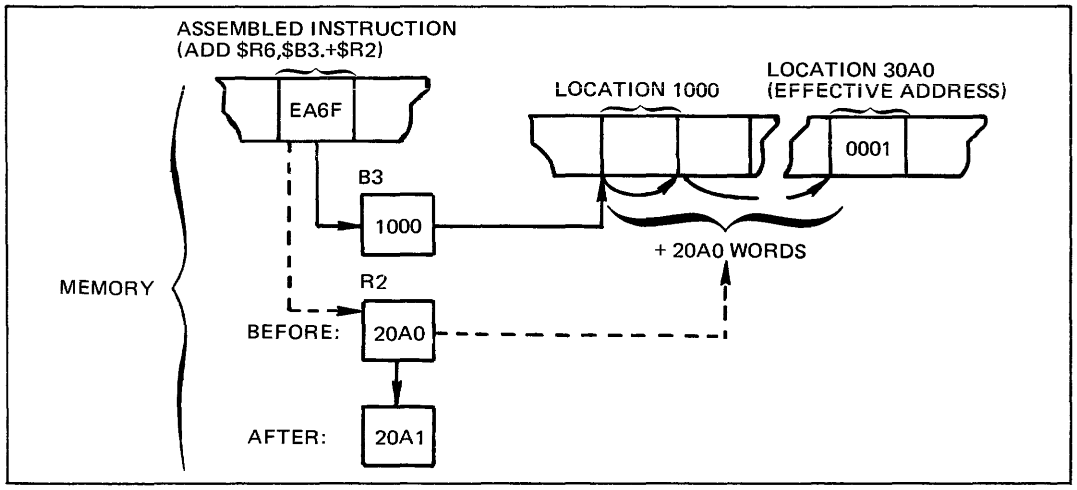


Figure 5-18. Indexed B-Relative Pop Addressing

Short Displacement Addressing

Short displacement addressing is available only for branch instructions. It is specified as follows:

$$> \left\{ \begin{array}{l} \text{internal-location-expression} \\ \left\{ \begin{array}{c} + \\ - \end{array} \right\} \text{temporary-label} \end{array} \right\}$$

When this form of addressing is used, the referenced location must be within the range -64 words to -1 word and +2 words to +63 words from the location of the instruction specifying it (i.e., it cannot reference itself or the Location following it).

The following example illustrates the use of short displacement addressing. In the example, $R3 contains the value 3033 and $F is a temporary label at a location preceding the instruction by 24 words.

Example:

```
BODD 3,>-$F
```

In this example, 3 is identified with $R3, and since its contents are an odd value, control is transferred to the instruction located at the memory address identified by $F (i.e., $F preceding the instruction illustrated in the example).

Figure 5-19 illustrates how the above example is stored in memory and how it derives the effective address of the location to be branched to.

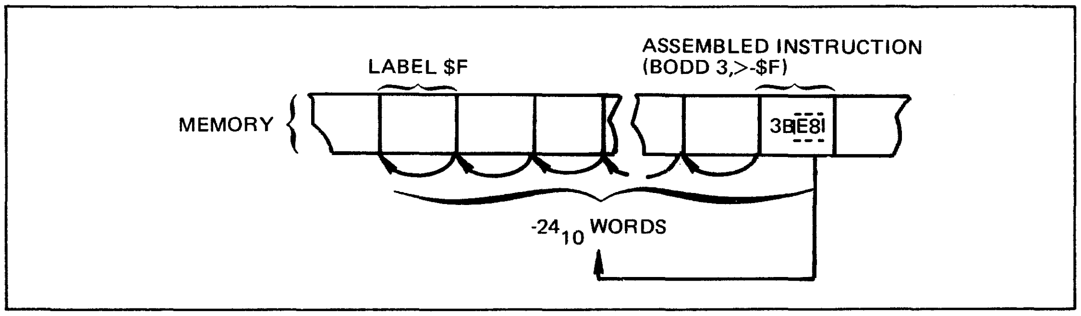


Figure 5-19. Short Displacement Addressing

Specialized Address Expressions

The following address expression is available for specifying an embedded control word in an I/O instruction. It can be used only in the second operand, and is specified as follows:

$$>= \left\{ \begin{array}{l} \text{internal-value-expression} \\ \text{external-value-label} \end{array} \right\}$$

The following example illustrates the use of this address form. In the example, $B3 contains the address 2002, which is assumed to be the address of the output control word, and it is to be output over channel 010. The value for sending the output control word over channel 010 is 0405.

Example:

```
IO $B3,>=Z'0405'
```

In the example, the output control word is extracted from location 2002, as specified by $B3, and sent over the desired channel.

Figure 5-20 illustrates how the example above is stored in memory and how it derives the effective address of the data.

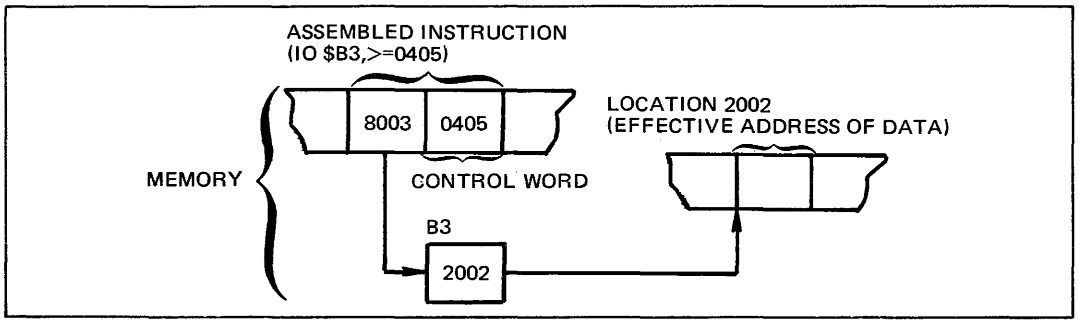


Figure 5-20. Specialized Address Expressions

Interrupt Vector Addressing

Interrupt vector addressing provides a convenient method by which you can examine the contents of the interrupt save area for the priority level at which your program is currently executing. (Priority levels and interrupt save areas are described in the Executive and Input/Output Manual.) Interrupt vector addressing is specified as follows:

$$\text{\$IV.} \begin{Bmatrix} \text{internal-value-expression} \\ \text{external-value-label} \end{Bmatrix}$$

In this form of addressing, $IV. points to the second word within the interrupt save area, and the value provides a displacement from the second word to another word within the interrupt save area. In the example below, the fifth word of the interrupt save area is loaded into R1. (Note that to address the second word of an interrupt save area, you require a displacement of 1, etc.)

Example:

```
          LDR $R1,$IV,THREE
THREE     EQU 3
```

Figure 5-21 illustrates how the above example locates the desired memory word and places it into R1.

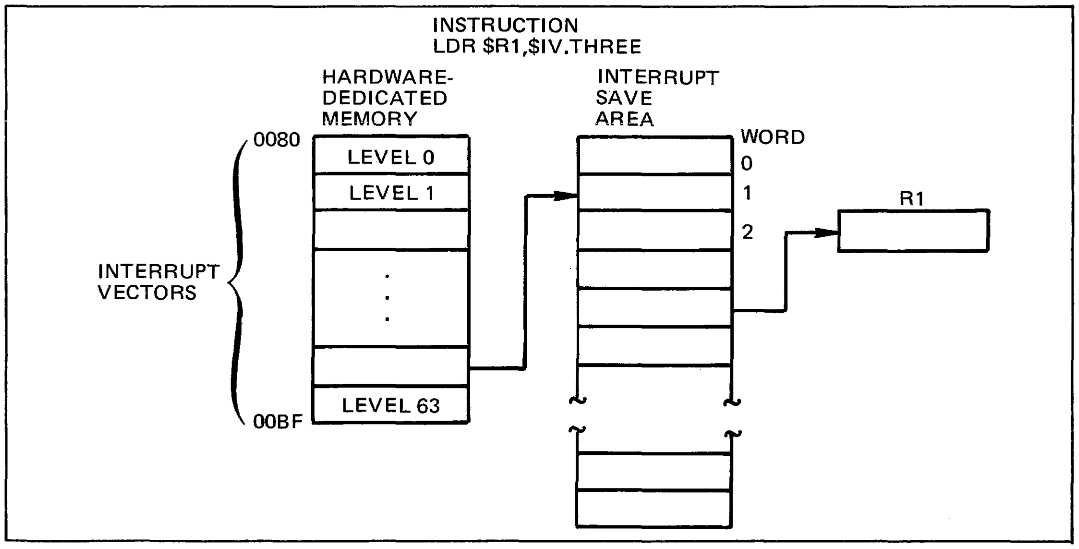


Figure 5-21. Interrupt Vector Addressing

ASSEMBLY LANGUAGE INSTRUCTIONS

The remainder of this section lists (alphabetically) and describes the assembly language instructions. The description of each instruction includes the name, type, format, and explanation of operands.

When an operand specifies a symbolic name, constant, or expression (other than an address expression), you can refer to Section 2 for a detailed description of those elements. Address expressions are defined in this section under "Addressing Techniques." Before using the following instructions you should fully understand the assembly language elements described in Section 2 and in this section.

Although not shown in the source language format, all assembly language instructions can be labeled.

ADD

Instruction:
Add Contents to R-register

Type:
DO

Source Language Format:

$$\Delta\text{ADD}\Delta \left\{\begin{array}{l}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} \text{address-expression}$$

Description:
Adds the contents of the location or R-register identified in the address expression to the contents of the R-register specified in the first operand. The result is saved in the first operand R-register.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{array}{l}=\$\text{Bn}\\ =\$\text{Sn}\end{array}\right\} \text{register addressing}$$

Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

- o If the result is more than 2^{15}-1(32767) or less th8n -2^{15}(-32768), the OV-bit is set to 1; otherwise, it is set to 0.
- o If, during the summation, a carry occurs, the C-bit is set to 1; otherwise, it is set to 0.

ADV

Instruction:
Add value to R-register

Type:
SI

Source Language Format:

$$\Delta\text{ADV}\Delta \left\{\begin{array}{l}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} ,[=] \left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{internal-value-expression}\end{array}\right\}$$

Description:
Adds the 8-bit value (with sign extended) specified in the second operand to the contents of the R-register identified in the first operand. The result is saved in R-register.

The contents of the I-register are affected as follows:

- o If the result is more than 2^{15}-1(32767) or less than -2^{15} (-32768), the OV-bit is set to 1; otherwise, it is set to 0.
- o If, during the summation, a carry occurs, tht C-bit is set to 1; otherwise, it is set to 0.

AND

Instruction:
AND contents with R-register

Type:
DO

Source Language Format:

$$\Delta\text{AND}\Delta \left\{\begin{array}{l}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} \text{,address-expression}$$

Description:
Logically AND's the contents of the R-register identified in the first operand with the contents of the location or R-register specified in the address expression. The result is saved in the first operand R-register.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{array}{l}=\$\text{Bn}\\ =\$\text{Sn}\end{array}\right\}\text{register addressing}$$
Short displacement addressing
Specialized addressing

The following chart illustrates the result of logically ANDing bits:

First operand bit:	0	0	1	1
Second operand bit:	1	0	1	0
Result:	0	0	1	0

ANH

Instruction:
Logically AND half-word (byte) with R-register

Type:
DO

Source Language Format

$$\Delta\text{ANH}\Delta \left\{ \begin{array}{l} \$\text{Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{ address-expression}$$

Description:
Logically AND's the contents of the R-register identified in the first operand with the contents of the byte specified in the address expression.

Prior to the operation, the byte operand is internally expanded to word length by extending the sign through the eight high-order bit positions. The byte selected to participate in the operation is determined by the format of the address expression, as follows:

- o Register Addressing (=$Rn): The rightmost byte of the register is selected.
- o Memory Addressing *Without* Indexing: Immediate Memory Addressing: The leftmost byte of the word at the designated memory address is selected.
- o Memory Addressing *With* Indexing: The memory address indicates a starting point. The index register contains an arithmetic value to be added to the starting point. The value specifies the number of bytes before or after the starting point needed to reach the byte selected for the operation.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left. \begin{array}{l} =\$\text{Bn} \\ =\$\text{Sn} \end{array} \right\} \text{ register addressing}$$
Short displacement addressing
Specialized addressing

The following chart illustrates the result of logically ANDing bits:

First operand bit:	0	0	1	1
Second operand bit:	1	0	1	0
Result:	0	0	1	0

B

Instruction:
Branch unconditionally

Type:
BI

Source Language Format:

$$\Delta \text{B} \Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches unconditionally to the location specified in the operand.

If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BAG

Instruction
Branch if algebraically greater than

Type:
BI

Source Language Format:

$$\Delta \text{BAG} \Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison indicates that the contents of the R-register were algebraically greater than the contents of the memory location specified in the compare instruction.

Action if Branch Occurs:
If the J-bit in the M1-register contains binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BAGE

Instruction:
Branch if algebraically greater than or equal to

Type:
BI

Source Language Format:

$$\Delta\text{BAGE}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison indicates that the contents of the R-register were either algebraically greater than or equal to the contents of the memory location specified in the compare instruction.

Action if Branch Occurs:
If the J-bit in the M1-register contains binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BAL

Instruction:
Branch if algebraically less than

Type:
BI

Source Language Format:

$$\Delta\text{BAL}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison indicates that the contents of the R-register were algebraically less than the contents of the memory location specified in the compare instruction.

Action if Branch Ocurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BALE

Instruction:
Branch if algebraically less than or equal to

Type:
BI

Source Language Format:

$$\Delta\text{BALE}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison indicates that the contents of the R-register were algebraically less than or equal to the contents of the memory location specified in the compare instruction.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BBF

Instruction:
Branch if bit-test indicator false

Type:
BI

Source Language Format:

$$\Delta\text{BBF}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the B-bit in the I-register is set to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BBT

Instruction:
Branch if bit-test indicator true

Type:
BI

Source Language Format:

$$\Delta\text{BBT}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the B-bit in the I-register is set to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BCF

Instruction:
Branch if no carry

Type:
BI

Source Language Format:

$$\Delta\text{BCF}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Branches to the location specified in the operand if the C-bit in the I-register is set to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BCT

Instruction:
Branch if carry

Type:
BI

Source Language Format:

$$\Delta\text{BCT}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the C-bit in the I-register is set to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BDEC

Instruction:
Branch and decrement

Type:
BR

Source Language Format:

$$\Delta\text{BDEC}\Delta \left\{\begin{array}{l}\$\text{Rn}\\ \text{X‘n’}\\ \text{n}\end{array}\right\}, \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Subtracts 1 from the contents of the R-register identified in the first operand; then, branches to the location specified in the second operand if the contents of the R-register are greater than or equal to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BE

Instruction:
Branch if equal

Type:
BI

Source Language Format:

$$\Delta\text{BE}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison sets both the G- and L-bits of the I-register to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BEVN

Instruction:
Branch if R-register even

Type:
BR

Source Language Format:

$$\Delta\text{BEVN}\Delta \left\{ \begin{array}{c} \$\text{Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} , \left\{ \begin{array}{l} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{array} \right\}$$

Description:
Branches to the location specified in the second operand if the R-register identified in the first operand contains an even value.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BEZ

Instruction:
Branch if R-register equal to 0

Type:
BR

Source Language Format:

$$\Delta\text{BEZ}\Delta \left\{ \begin{array}{c} \$\text{Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} , \left\{ \begin{array}{l} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{array} \right\}$$

Description:
Branches to the location specified in the second operand if the R-register identified in the first operand contains 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BG

Instruction:
Branch if greater than

Type:
BI

Source Language Format:

$$\Delta\text{BG}\Delta \begin{Bmatrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison sets the G bit of the I-register to 1.
Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BGE

Instruction:
Branch if greater than or equal to

Type:
BI

Source Language Format:

$$\Delta\text{BGE}\Delta \begin{Bmatrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison sets the L-bit of the I-register to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BGEZ

Instruction:
Branch if R-register greater than or equal to 0

Type:
BR

Source Language Format:

$$\Delta\text{BGEZ}\Delta \left\{\begin{array}{c}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} , \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the second operand if the R- register identified in the first operand contains a positive value or 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BGZ

Instruction:
Branch if R-register greater than 0

Type:
BR

Source Language Format:

$$\Delta\text{BGZ}\Delta \left\{\begin{array}{c}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} , \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the second operand if the R-register identified in the first operand contains a positive value.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BINC

Instruction:
Branch and increment

Type:
BR

Source Language Format:

$$\Delta\text{BINC}\Delta \begin{Bmatrix} \text{\$Rn} \\ \text{X'n'} \\ \text{n} \end{Bmatrix} , \begin{Bmatrix} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{Bmatrix}$$

Description:
Adds 1 to the contents of the R-register identified in the first operand; then, branches to the location specified in the second operand if the contents of the R-register is not 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BIOF

Instruction:
Branch if I/O indicator false

Type:
BI

Source Language Format:

$$\Delta\text{BIOF}\Delta \begin{Bmatrix} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the I-bit in the I-register is set to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BIOT

Instruction:
Branch if I/O indicator true

Type:
BI

Source Language Format:

$$\Delta\text{BIOT}\Delta \begin{Bmatrix} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the I-bit in the I-register is set to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BL

Instruction:
Branch if less than

Type:
BI

Source Language Format:

$$\Delta\text{BL}\Delta \begin{Bmatrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison sets the L-bit of the I-register to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BLE

Instruction:
Branch if less than or equal to

Type:
BI

Source Language Format:

$$\Delta\text{BLE}\Delta \begin{Bmatrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison sets the G-bit of the I-register to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BLEZ

Instruction:
Branch if R-register equal to or less than 0

Type:
BR

Source Language Format:

$$\Delta\text{BLEZ}\Delta \begin{Bmatrix}\text{\$Rn}\\ \text{X'n'}\\ \text{n}\end{Bmatrix}, \begin{Bmatrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the second operand if the R-register identified in the first operand contains a negative value or 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BLZ

Instruction:
Branch if R-register less than 0

Type:
BR

Source Language Format:

$$\Delta\text{BLZ}\Delta \begin{Bmatrix}\text{\$Rn}\\ \text{X'n'}\\ \text{n}\end{Bmatrix}, \begin{Bmatrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the second operand if the R-register identified in the first operand contains a negative value.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BNE

Instruction:
Branch if not equal

Type:
BI

Source Language Format:

$$\Delta\text{BNE}\Delta\left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison sets either (but not both) the G-bit or the L-bit of the I-register to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BNEZ

Instruction:
Branch if R-register not equal to 0

Type:
BR

Source Language Format:

$$\Delta\text{BNEZ}\Delta\left\{\begin{array}{l}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\},\left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the second operand if the R-register identified in the first operand contains a value other than 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BNOV

Instruction:
Branch if no R-register overflow

Type:
BI

Source Language Format:

$$\Delta\text{BNOV}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the OV-bit in the I-register is set to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BODD

Instruction:
Branch if R-register odd

Type:
BR

Source Language Format:

$$\Delta\text{BODD}\Delta \left\{\begin{array}{l}\text{\$Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\},\left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the second operand if the R-register identified in the first operand contains an odd value.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BOV

Instruction:
Branch if R-register overflow

Type:
BI

Source Language Format:

$$\Delta\text{BOV}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the OV-bit in the I-register is set to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BRK

Instruction:
Break trap

Type:
GE

Source Language Format:

ΔBRKΔ

Description:
Enters the trace procedure by a trap to trap vector 2; this instruction is used for debugging.

BSE

Instruction:
Branch if signs equal

Type:
BI

Source Language Format:

$$\Delta\text{BSE}\Delta \begin{Bmatrix} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison indicates that the sign of the value in the R-register was equal to the sign of the value in the memory location in the most recent compare instruction (i.e., the U-bit in the I-register is set to 0.)

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

BSU

Instruction:
Branch if signs unlike

Type:
BI

Source Language Format:

$$\Delta\text{BSU}\Delta \begin{Bmatrix} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent comparison indicates that the sign of the value in the R-register was unequal to the sign of the value in the memory location or R-register in the most recent compare instruction (i.e., the U-bit in the I-register is set to 1.)

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

CAD

Instruction:
Add carry bit to contents

Type:
SO

Source Language Format:
ΔCADΔaddress-expression

Description:
Adds the contents of the C-bit in the I-register to the contents of the location specified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{matrix} =\$Bn \\ =\$Sn \end{matrix}\right\} \text{register addressing}$$
Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

- o If a carry occurs during the operation, the C-bit is set to 1; otherwise, it is set to 0.
- o If the result is more than 16 bits long, the OV-bit is set to 1; otherwise, it is set to 0.

CL

Instruction:
Clear

Type:
SO

Source Language Format:

ΔCLΔaddress-expression

Description:
Stores zeros in the location or R-register specified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{matrix}\text{=\$Bn} \\ \text{=\$Sn}\end{matrix}\right\}$ register addressing
Short displacement addressing
Specialized addressing

CLH

Instruction:
Clear half-word

Type:
SO

Source Language Format:

ΔCLHΔaddress-expression

Description:
Stores 0's in the half-word (byte) location specified in the address expression.

- o If the address expression specifies =Rn, 0's are stored in the rightmost byte of the register.
- o If the operand specifies immediate memory addressing *without* indexing, or an immediate operand format 0's are stored in the leftmost byte of the word found at the specified location.
- o If the operand specifies immediate memory addressing *with* indexing, the index register is aligned to count bytes relative to the leftmost byte of the word specified. 0's are stored in the byte thus addressed.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

CMB

Instruction:
Compare contents to B-register

Type:
DO

Source Language Format:

$$\Delta\text{CMB}\Delta \left\{ \begin{array}{l} \$\text{Bn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{,address-expression}$$

Description:
Compares the contents of the B-register identified in the first operand to the contents of the location or B-register specified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left. \begin{array}{l} =\$\text{Rn} \\ =\$\text{Sn} \end{array} \right\}$ register addressing
Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

- o If the contents of the B-register are greater than the contents of the location, the G-bit is set to 1; otherwise, it is set to 0.
- o If the contents of the B-register are less than the contents of the location, the L-bit is set to 1; otherwise, it is set to 0.
- o The setting of the U-bit is undefined.

CMH

Instruction:
Compare half-word (byte) to R-register

Type:
DO

Source Language Format:

$$\Delta\text{CMH}\Delta \left\{ \begin{array}{l} \$\text{Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{,address-expression}$$

Description:
Compares the contents of the R-register identified in the first operand to the contents of the byte specified in the address expression.

Prior to the operation, the byte operand is internally expanded to word length by extending the sign through the eight high-order bit positions. The byte selected to participate in the operation is determined by the format of the address expression, as follows:

- o Register Addressing (=$Rn): The rightmost byte of the register is selected.
- o Memory Addressing *Without* Indexing: Immediate Memory Addressing: The leftmost byte of the word at the designated memory address is selected.
- o Memory Addressing *With* Indexing: The memory address indicates a starting point. The index register contains an arithmetic value to be added to the starting point. The value specifies the number of bytes before or after the starting point needed to reach the byte selected for the operation.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn } register addressing
=$Sn }
Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

- o If the contents of the R-register are greater than the contents of the created temporary word, the G-bit is set to 1; otherwise, it is set to 0.
- o If the contents of the R-register are less than the contents of the created temporary word, the L-bit is set to 1; otherwise, it is set to 0.
- o If the contents of the R-register and the contents of the created temporary word do not have like signs, the U-bit is set to 1; otherwise, it is set to 0.

CMN

Instruction:
Compare address to null

Type:
SO

Source Language Format:
    ΔCMNΔaddress-expression

Description:
Compares the contents of the location or B-register specified by the address expression to a null address (the address 0).

The contents of the I-register are affected as follows:

- o The G-bit is set to 0 if the contents of the specified location or register are equal to null; otherwise, it is set to 1.
- o The L-bit is set to 0.
- o The U-bit is affected, but its value is undefined.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Rn\\=\$Sn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

CMR

Instruction:
Compare contents to R-register

Type:
DO

Source Language Format:

$$\Delta\text{CMR}\Delta \left\{\begin{array}{l}\$Rn\\X'n'\\n\end{array}\right\} \text{,address-expression}$$

Description:
Compares the contents of the R-register identified in the first operand to the contents of the location or R-register specified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\=\$Sn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

- o If the contents of the R-register are greater than the contents of the location, the G-bit is set to 1; otherwise, it is set to 0.
- o If the contents of the R-register are less than the contents of the location,.the L-bit is set to 1; otherwise, it is set to 0.
- o If the content of bit 0 of the R-register is not equal to the content of bit 0 of the location, the U-bit is set to 1; otherwise, it is set to 0.

CMV

Instruction:
Compare value to R-register

Type:
SI

Source Language Format:

$$\Delta\text{CMV}\Delta \left\{\begin{array}{l}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} ,[=] \left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{integer-value-label}\\ \text{integer-value-expression}\\ \text{fixed-point-constant}\end{array}\right\}$$

Description:
Compares the 8-bit value (with sign extended) specified in the second operand to the contents of the R-register identified in the first operand.

Except for the string constant form all values are assumed to be numeric.

The contents of the I-register are affected as follows:

- o If the contents of the R-register are greater than the value (with sign extended), the G-bit is set to 1; otherwise, it is set to 0.
- o If the contents of the R-register are less than the value (with sign extended), the L-bit is set to 1; otherwise, it is set to 0.
- o If the sign of the R-register and the sign of the value are not equal, the U-bit is set to 1; otherwise, it is set to 0.

CMZ

Instruction:
Compare to 0

Type:
SO

Source Language Format:
  ΔCMZΔaddress-expression

Description:
Compares the contents of the location or R-register specified in the address expression to 0.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{array}{l}=\$\text{Bn}\\ =\$\text{Sn}\end{array}\right\} \text{register addressing}$$

Short displacement addressing
Specialized addressing

The $Bn.$R1, $Bn.$R2, or $Bn.$R3 form of addressing can be used by this instruction to cause a trap for the purpose of sizing main memory provided the generated effective address is less than or equal to 64K.

The contents of the I-register are affected as follows:

- o If the contents of the specified location do not equal 0, the G-bit is set to 1; otherwise, it is set to 0.

- o The L-bit is set to 0.
- o If the first bit of the specified location equals 1, the U-bit is set to 1; otherwise, it is set to 0.

CPL

Instruction:
Complement

Type:
SO

Source Language Format:

ΔCPLΔaddress-expression

Description:
One's complements the contents of the location or R-register specified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{array}{l}=\$Bn\\=\$Sn\end{array}\right\}\text{register addressing}$$

Short displacement addressing
Specialized addressing

DAL

Instruction:
Double-shift arithmetic-left

Type:
SHL

Source Language Format:

$$\Delta\text{DAL}\Delta\left\{\begin{array}{l}\$R\left\{\begin{array}{c}3\\5\\7\end{array}\right\}\\ X'\left\{\begin{array}{c}3\\5\\7\end{array}\right\}'\\ \left\{\begin{array}{c}3\\5\\7\end{array}\right\}\end{array}\right\}\text{,internal-value-expression}$$

Description:
Left shifts the contents of the even-odd R-register pair (i.e., R2 and R3, R4 and R5, R6 and R7) identified in the first operand the number of bit positions specified by the internal value expression in the second operand. The bit positions vacated by the shift are filled with binary 0's.

The internal value expression must be ⩾ 0 and ⩽ 31.
If the internal value expression equals 0, the contents are shifted left the number of bit positions derived by using the value in bits 11 through 15 of general register R1.

The contents of the I-register are affected as follows:

- o If the contents of bit 0 in the even-numbered R-register changes, the OV-bit is set to 1; otherwise, it is set to 0.

DAR

Instruction:
Double-shift arithmetic-right

Type:
SHS

Source Language Format:

$$\Delta\text{DAR}\Delta\left\{\begin{matrix} \$\text{R}\begin{Bmatrix}3\\5\\7\end{Bmatrix} \\ \text{X}'\begin{Bmatrix}3\\5\\7\end{Bmatrix}' \\ \begin{Bmatrix}3\\5\\7\end{Bmatrix}\end{matrix}\right\}\text{,internal-value-expression}$$

Description:
Shifts the contents of the even-odd R-register pair (i.e., R2 and R3, R4 and R5, R6 and R7) identified in the first operand right the number of bit positions specified by the internal value expression in the second operand. The bit positions vacated by the shift are filled with the sign value originally contained in bit 0.

The internal value expression must be ⩾ 0 and ⩽ 31.
The contents of the I-register are affected as follows:

- o C-bit contains the last binary digit shifted out of the odd-numbered R-register.

DCL

Instruction:
Double-shift closed-left

Type:
SHS

Source Language Format:

$$\Delta\text{DCL}\Delta \left\{ \begin{array}{c} \$\text{R}\left\{\begin{array}{c}3\\5\\7\end{array}\right\} \\ \text{X}'\left\{\begin{array}{c}3\\5\\7\end{array}\right\}' \\ \left\{\begin{array}{c}3\\5\\7\end{array}\right\} \end{array} \right\} \text{,internal-value-expression}$$

Description:
Shifts the contents of the even-odd R-register pair (i.e., R2 and R3, R4 and R5, R6 and R7) identified in the first operand left the number of bit positions specified by the internal value expression in the second operand. The bits shifted out of the even-numbered R-register are placed in the bit positions of the odd-numbered R-register vacated as the bits are shifting left.

The internal value expression must be $\geqslant 0$ and $\leqslant 15$.
If the internal value expression equals 0, the contents are shifted left the number derived by using the value in bits 11 through 15 of general register R1.

DCR

Instruction:
Double-shift closed-right

Type
SHS

Source Language Format:

$$\Delta\text{DCR}\Delta \left\{ \begin{array}{c} \$\text{R}\left\{\begin{array}{c}3\\5\\7\end{array}\right\} \\ \text{X}'\left\{\begin{array}{c}3\\5\\7\end{array}\right\}' \\ \left\{\begin{array}{c}3\\5\\7\end{array}\right\} \end{array} \right\} \text{,internal-value-expression}$$

Description:
Shifts the contents of the even-odd R-register pair (i.e., R2 and R3 R4 and R5, R6 and R7) identified in the first operand right the number of bit positions specified by the internal value expression in the second operand. The bits shifted out of the odd-numbered R-register are placed in the bit positions of the even-numbered R-register vacated as the bits are shifting right.

The internal value expression must be ≥ 0 and ≤ 15.
If the internal value expression equals 0, the contents are shifted right the number derived by using the value in bits 11 through 15 of general register R1.

DEC

Instruction:
Decrement

Type:
SO

Source Language Format:

ΔDECΔaddress-expression

Description:
Decrements by 1 the contents of the location or R-register specified in the address expression, then copies bit 0 of the addressed word or register into I(B).

This instruction operates in read modify write (RMW) mode, which prevents any other processor in a multiprocessor environment from accessing the location being modified until the modification is completed.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{matrix}\text{=\$Bn}\\ \text{=\$Sn}\end{matrix}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

- o If the decrementation causes a carry to occur, the C-bit is set to 1; otherwise, it is set to 0.
- o If the value being decremented was - 32768 (-2^{15}), I(OV) is set to 1; otherwise, I (OV) is cleared to 0.
- o I (B) is set as described above.

DIV

Instruction:
Divide R-register by contents of location

Type:
DO

Source Language Format

$$\Delta\text{DIV}\Delta \left\{\begin{matrix}\text{\$Rn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\} \text{,address-expression}$$

Description:
Divides the contents of the R-register identified in the first operand by the contents of the location or R-register specified in the address expression. The result is saved in the first operand R-register (except for the remainder, which is ignored).

If R7 is identified as the first operand R-register, the double integer operand contained in R6 and R7 is divided by the single integer operand identified by the address expression. The result is saved in R7 and the remainder is saved in R6.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
=$Sn
Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

1. I(OV) is set to 1 if
   a. The divisor = 0
   b. The quotient is greater than 2^{15} -1 (32767) or less than -2^{15} (-32768)

   Otherwise I(OV) is cleared to 0.

   Divide operations that cause I(OV) to be set terminate with all operands unchanged.
2. I(C) is set to 1 if the remainder is not 0, or cleared to 0 if the remainder is 0. I(C) is unchanged when the first operand is $R7. If the divisor = 0 or if the dividend is -2^{15} times the divisor, I(C) is undefined.

DOL

Instruction:
Double-shift open-left

Type:
SHL

Source Language Format:

$$\Delta\text{DOL}\Delta\left\{\begin{array}{l} \$R\left\{\begin{array}{c}3\\5\\7\end{array}\right\} \\ X'\left\{\begin{array}{c}3\\5\\7\end{array}\right\}' \\ \left\{\begin{array}{c}3\\5\\7\end{array}\right\}\end{array}\right\}\text{,internal-value-expression}$$

Description:
Shifts the contents of the even-odd R-register pair (i.e., R2 and R3, R4 and R5, R6 and R7) identified in the first operand left the number of bit positions specified by the internal value expression in the operand. The bit positions vacated by the shift are filled with binary 0's.

The internal value expression must be ⩾ 0 and ⩽ 31.
If the internal value expression equals 0, the contents are shifted left the number derived by using the value in bits 11 through 15 of general register R1.

The contents of the I-register are affected as follows:

o C-bit contains the last binary digit shifted out of the even-numbered R-register.

DOR

Instruction
Double-Shift open-right

Type:
SHL

Source Language Format:

$$\Delta\text{DOR}\Delta \left\{ \begin{array}{l} \$R \left\{\begin{array}{c}3\\5\\7\end{array}\right\} \\ X' \left\{\begin{array}{c}3\\5\\7\end{array}\right\} ' \\ \left\{\begin{array}{c}3\\5\\7\end{array}\right\} \end{array} \right\} \text{,internal-value-expression}$$

Description:
Shifts the contents of the even-odd R-register pair (i.e,. R2 and R3, R4 and R5, R6 and R7) identified in the first operand right the number of bit positions specified by the internal value expression in the operand. The bit positions vacated by the shift are filled with binary 0's.

The internal value expression must be ⩾ 0 and ⩽ 31.
If the internal value expression equals 0, the contents are shifted right the number derived by using the value in bits 11 through 15 of general register R1.

The contents of the I-register are affected as follows:

o C-bit contains the last binary digit shifted out of the odd-numbered R-register.

ENT

Instruction:
Enter

Type:
SO

Source Language Format:

$\Delta\text{ENT}\Delta \left\{\begin{array}{l}\text{immediate-memory-address}\\ \text{B-relative-addressing}\end{array}\right\}$

Description:
Jumps to the memory location specified by the operand; also, sets the P-bit in the S-register to 0 (i.e., sets the bit to indicate slave mode).

If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, or if the J-bit contains a binary 0, execution commences at the specified location.

HLT

Instruction:
Halt

Type:
GE

Source Language Format:
ΔHLTΔ

Description:
Stops program execution. HLT state is indicated on the control panel. All interrupts will be honored.

The P-bit in the S-register must be set to 1 (i.e., the central processor must be in the privileged state) for this instruction to be executed. If not, the unprivileged use of a privileged operation is signified by a trap to trap vector 13.

INC

Instruction:
Increment

Type:
SO

Source Language Format:
ΔINCΔaddress-expression

Description:
Copies bit 0 of the contents of the location or R-register specified in the address expression into I(B), then increments by 1 the contents of the location or register.

This instruction operates in read modify write (RMW) mode, which prevents any other processor in a multiprocessor environment from accessing the location being modified until the modification is completed.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

- o If the incrementation causes a carry to occur, the C-bit is set to 1; otherwise, it is set to 0.
- o If the value being incremented was 32767, I(OV) is set to 1; otherwise, it is cleared to 0.

IO

Instruction:
Input/Output (word)

Type:
IO

Source Language Format:
    ΔIOΔaddress-expression,address-expression

Description:
1. If the function code (F) is odd (indicating output): sends the command word (CH,F) specified by the second operand and the word specified by the first operand to the addressed IO channel.
2. If the function code (F) is even (indicating input): sends the command word (CH,F) specified by the second operand to the addressed channel. If the channel accepts the command, receives a word response from the channel and stores it in the word location or R-register specified by the first operand. If the channel does not accept the command, the contents of the location or register remain unchanged.

In both cases above, if the IO channel accepts the command, the I-bit in the indicator register is set to binary 1.

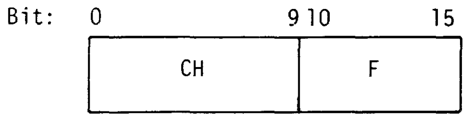
For the first operand, the address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

For the second operand, the address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing

The channel number and function code are contained in the R-register or memory word specified by the second operand. The channel number and function code occupy 16 bits formatted as follows:



CH is the channel number and F is the function code. The channel number is odd for output (memory-to-device) transfer and even for input (device-to-memory) transfer. The function code is controller-specific, subject to these constraints:

1. If F is odd, data (specified by the first operand) is transferred from the CPU to the controller.
2. If F is even, data is transferred from the controller to the CPU, which stores the data in the R-register or memory word specified by the first operand.

The following shows how the required channel number and function code are used. Assume that the status of a read operation on channel 20_{16} is to be stored into the word labeled STATUS. Also assume that the controller uses the standard function code 18_{16} for "input status register." The IO instruction to accomplish this could be coded as shown below:

```
IO STATUS,>=Z'0818'
```

or it could be coded as:

```
IO STATUS,>=X'20'*64+X'18'
```

For detailed information on the bus, refer to the Handbook.

The contents of the I-register are affected as follows:

- o If the controller accepted the command, the I-bit is set to 1; otherwise, it is cleared to 0.

IOH

Instruction:
Input/output half-word

Type:
IO

Source Language Format:
ΔIOHΔaddress-expression,address-expression

Description:
This instruction is identical to the IO instruction, except that the first operand specifies a half-word as follows:

- If it specifies =\$R, the rightmost byte of the specified R-register is sent (i.e., function code is odd) to the bus.
- If it specifies memory addressing *without* indexing, or an immediate operand addressing format, the leftmost byte of the word found at the specified location is sent (i.e:., function code is odd) to the bus.
- If it specifies memory addressing *with* indexing, the index register is aligned to count bytes relative to the leftmost byte of the word specified. The byte thus addressed is sent (i.e., function code is odd) to the bus.

For each of the above cases, if the function code is even, the first operand specifies the byte in which the response from the bus is to be stored.

See the description of the IO instruction for details regarding the coding of the operands.

IOLD

Instruction:
Input/output load

Type:
IO

Source Language Format:
    ΔIOLDΔaddress-expression,address-expression,address-expression

Description:
Sends the controller the effective address (specified by the first operand), the channel number and function code (specified in the second operand), and the range (i.e., number of bytes to be transferred) value (specified in the third operand) over the channel specified in the second operand to the bus. The address and range value are used to load the controller address and range registers.

For the first operand, the address expression can take any of the forms described earlier in this section under"Addressing Techniques" except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Rn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing
Immediate operand addressing

For the second operand, the address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Rn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing
Immediate operand addressing

The second operand of this instruction must specify the function code 09_{16}.

For the third operand, the address expression can take any of the forms described earlier in this section under "Addressing Techniques" except for the following:

$\left.\begin{array}{l}=\$Bn\\ =\$Sn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The following shows how the required channel number and function code are used. Assume that 128 bytes are to be read from the device on channel 20_{16} into the buffer labeled BUFFER. The IOLD instruction to output this information to the controller could be coded as shown below:

```
IOLD BUFFER,>=Z'0809',=128
```

For detailed information about the bus, see the Handbook.

The contents of the I-register are affected as follows:

- o If the channel accepted the command, the I-bit is set to 1; otherwise, it is set to 0.

JMP

Instruction:
Jump

Type:
SO

Source Language Format:

$$\Delta\text{JMP}\Delta\left\{\begin{array}{l}\text{immediate-memory-address}\\ \text{B-relative-addressing}\end{array}\right\}$$

Description:
Jumps to the location specified in the operand.

If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, or if the J-bit contains a binary 0, execution commences at the specified location.

LAB

Instruction:
Load effective address into B-register

Type:
DO

Source Language Format:

$$\Delta\text{LAB}\Delta\left\{\begin{array}{c}\$\text{Bn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,address-expression}$$

Description:
Loads the effective address generated by the address expression into the B-register identified in the first operand.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

- o Register addressing
- o Short displacement addressing
- o Specialized addressing

LB

Instruction:
Load bit

Type:
SO

Source Language Format:

$$\Delta\text{LB}\Delta\text{address-expression}\left[\;,\left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{external-value-label}\\ \text{internal-value-expression}\\ \text{fixed-point-constant}\end{array}\right\}\right]$$

Description:

1. If the first operand specifies indexing, the index register is aligned to count bits relative to bit 0 of the specified word. The bit thus addressed is loaded into the B-bit of the I-register.

2. If the first operand does *not* specify indexing, the value (mask) in the second operand identifies which bit(s) are to be checked (e.g., Z'8000' indicates that the first bit of the word found at the specified location is to be checked); then, if (any of) the specified bit(s) contain a binary 1, the B-bit of the I-register is set to 1; otherwise, it is set to 0.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{array}{l}=\$\text{Bn}\\ =\$\text{Sn}\end{array}\right\}\text{register addressing}$$

Short displacement addressing
Specialized addressing

LBC

Instruction:
Load bit and complement

Type:
SO

Source Language Format:

$$\Delta\text{LBC}\Delta\text{address-expression}\left[,\left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{external-value-label}\\ \text{internal-value-expression}\\ \text{fixed-point-constant}\end{array}\right\}\right]$$

Description:

1. If the first operand specifies indexing, the index register is is aligned to count bits relative to bit 0 of the specified word. The bit thus addressed is loaded into the B-bit of the I-register.
   Upon completion of the operation, the addressed bit is set to the one's complement of its value.
2. If the first operand does *not* specify indexing, the value (mask) in the second operand identifies which bit(s) are to be checked (e.g., Z'8000' indicates that the first bit of the word found at the specified location of R-register is to be checked); then, if (any of) the specified bit(s) contains a binary 1, the B-bit of I-register is set to 1; otherwise it is set to 0.

This instruction operates in read modify write (RMW) mode, which prevents any other processor in a multiprocessor environment from accessing the location being modified until the modification is completed.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\ =\$Sn\end{array}\right\}$ register addressing

Short displacement addressing

Specialized addressing

LBF

Instruction:

Load bit and set false

Type:

SO

Source Language Format:

$$\Delta\text{LBF}\Delta\text{address-expression}\left[,\left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{external-value-label}\\ \text{internal-value-expression}\\ \text{fixed-point-constant}\end{array}\right\}\right]$$

Description:

1. If the first operand specifies indexing, the index register is aligned to count bits relative to bit 0 of the specified word. The bit thus addressed is loaded into the B-bit of the I-register.
   Upon completion of the operation, the addressed bit is set to 0.

2. If the first operand does not specify indexing, the value (mask) in the second operand identifies which bit(s) are to be checked (e.g., Z'8000' indicates that the first bit of the word found at the specified location or R-register is to be checked); then, if (any of) the specified bit(s) contains a binary 1, the B-bit of the I-register is set to 1; otherwise, it is set to 0.

This instruction operates in read modify write (RMW) mode, which prevents any other processor in a multiprocessor environment from accessing the location being modified until the modification is completed.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{matrix}\text{=\$Bn}\\ \text{=\$Sn}\end{matrix}\right\}$ register addressing
Short displacement addressing
Specialized addressing
Immediate operand addressing

LBT

Instruction:
Load bit and set true

Type:
SO

Source Language Format:

$$\Delta\text{LBT}\Delta\text{address-expression}\left[,\left\{\begin{matrix}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{external-value-label}\\ \text{internal-value-expression}\\ \text{fixed-point-constant}\end{matrix}\right\}\right]$$

Description:

1. If the first operand specifies indexing, the index register is aligned to count bits relative to bit 0 of the specified word. The bit thus addressed is loaded into the B-bit of the I-register.
   Upon completion of the operation, the addressed bit is set to 1.
2. If the first operand does not specify indexing, the value (mask) in the second operand identifies which bit(s) are to be checked (e.g., Z'8000' indicates that the first bit of the word found at the specified location of R-register is to be checked); then, if (any of) the specified bit(s) contains a binary 1, the B-bit of the I-register is set to 1; otherwise, it is set to 0.
   Upon completion of the operation, the bit(s) checked in accordance with the mask is (are) set to 1.

This instruction operates in read modify write (RMW) mode, which prevents any other processor in a multiprocessor environment from accessing the location being modified until the modification is completed.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{matrix}\text{=\$Bn}\\ \text{=\$Sn}\end{matrix}\right\}$ register addressing
Short displacement addressing
Specialized addressing

LBS

Instruction:
Load bit and swap

Type:
SO

Source Language Format:

$$\Delta\text{LBS}\Delta\text{address-expression}\left[,\begin{Bmatrix}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{external-value-label}\\ \text{internal-value-expression}\\ \text{fixed-point-constant}\end{Bmatrix}\right]$$

Description:

1. If the first operand specifies indexing, the index register is aligned to count bits relative to bit 0 of the specified word. The bit thus addressed is interchanged with the B-bit of the I-register.
2. If the first operand does not specify indexing, the value (mask) in the second operand identifies which bit(s) are to be checked (e.g., Z′8000′ indicates that the first bit of the word found at the specified location or R-register is to be checked); then, if (any of) the specified bit(s) contains a binary 1, the B-bit of the I-register is set to 1; otherwise, it is set to 0.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{matrix}\text{=\$Bn}\\ \text{=\$Sn}\end{matrix}\right\}$ register addressing
Short displacement addressing
Specialized addressing

LDB

Instruction:
Load B-register

Type:
DO

Source Language Format:

$$\Delta\text{LDB}\Delta \quad \left\{\begin{matrix}\$\text{Bn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\} \quad \text{,address-expression}$$

Description:
Loads the contents of the location or B-register specified by the address expression into the B-register identified in the first operand.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{matrix}=\$\text{Rn}\\ =\$\text{Sn}\end{matrix}\right\}$ register addressing
Short displacement addressing
Specialized addressing
Immediate operand addressing with an internal value expression

LDH

Instruction:
Load half-word (byte) into R-register

Type:
DO

Source Language Format:

$$\Delta\text{LDH}\Delta \quad \left\{\begin{matrix}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\} \quad \text{,address-expression}$$

Description:
Loads the contents of the location specified in the address expression, as described below, into the R-register identified in the first operand:

- o If the address expression specifies =$Rn the rightmost byte (sign extended) of that R-register is loaded into the R-register specified by the first operand.
- o If the address expression specifies memory addressing *without* indexing, or an immediate operand addressing format, the leftmost byte (sign extended) of the word found at the specified location is loaded into the R-register.
- o If the address expression specifies memory addressing *with* indexing, the index register is aligned to count bytes relative to the leftmost byte of the word specified. The byte thus addressed is loaded (sign extended) into the R-register.

In all cases, the selected byte is loaded into the rightmost byte of the R-register, with the sign extended to the left.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{matrix}=\$\text{Bn}\\ =\$\text{Sn}\end{matrix}\right\}$ register addressing
Short displacement addressing
Specialized addressing

LDI

Instruction:
Load double-word integer

Type:
SO

Source Language Format:
ΔLDIΔaddress-expression

Description:
Loads the contents of the location specified by the address expression into register R6 and the contents of the next location into register R7.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

If =$Rn is used, only =$R3 (loads the contents of R2 and R3 into R6 and R7, respectively) or =$R5 (loads the contents of R4 and R5 into R6 and R7, respectively) may be used.

LDR

Instruction:
Load R-register

Type:
DO

Source Language Format:

$$\Delta\text{LDR}\Delta \begin{Bmatrix}\text{\$Rn}\\ \text{X'n'}\\ \text{n}\end{Bmatrix} \text{,address-expression}$$

Description:
Loads the contents of the location or R-register identified in the address expression into the R-register identified in the first operand.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

LDV

Instruction:
Load value

Type:
SI

Source Language Format:

$$\Delta\text{LDV}\Delta\left\{\begin{array}{l}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,[=]}\left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{internal-value-expression}\\ \text{fixed-point-constant}\end{array}\right\}$$

Description:
Loads the 8-bit value identified in the second operand into the right half-word of the R-register specified in the first operand. The contents of bit 8 are extended through the left half-word of the R-register.

Except for the string constant form of the second operand, all values are assumed to be numeric.

LEV

Instruction:
Level Change

Type:
SO

Source Language Format:
ΔLEVΔaddress-expression

Description:
Sets or resets level activity bits according to the contents of the location indicated by the address expression.

The following bit configurations in the indicated location produce the actions described below.

Bit:	0	1	2	3	4	5	6	7	8	9	10 15
	0	0	0	0	0	0	0	0	0	0	Level Number

Schedule Interrupt Level, Scan and Dispatch

The level activity bit for the designated level will be set. The level activity bits will be scanned and the highest active level ascertained. The context of the current level will be saved (unless the current level is the highest active level). The context of the highest active level will be restored (again, unless the current level is the highest active level).

Bit:	0	1	2	3	4	5	6	7	8	9	10 15
	0	1	0	0	0	0	0	0	0	0	Level Number

Schedule Interrupt Level, Defer Interrupt

The level activity bit for the designated level will be set. Execution will continue at the current level.

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

Inhibit

The level activity bit for priority level 3 will be set. The interrupt vector for priority level 3 will be set equal to the interrupt vector for the current level. Execution of the current task continues at priority level 3. The use of level 3 as the inhibit level is a software convention.

Bit:	0	1	2	3	4	5	6	7	8	9	10 15
	1	0	0	0	0	0	0	0	0	0	Level Number

Schedule Interrupt Level, Suspend, Scan and Dispatch

The level activity bit for the designated level will be set. The level activity bit for the current level will be reset. The level activity bits will be scanned and the highest level ascertained. The context of the current level will be saved. The context of the highest active level will be restored.

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

Suspend, Inhibit

The level activity bit for the current level will be reset. The level activity bit for priority level 3 will be set. The interrupt vector for priority level 3 will be set equal to the interrupt vector for the current level. Execution of the task continues at priority level 3. The use of level 3 as the inhibit level is a software convention.

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

Enable

Enable is used to end execution at priority level 3. The level activity bit for priority level 63 will be set. The level activity bit for priority level 3 will be reset. The level activity bits will be scanned and the highest active level ascertained. The context of the current level is saved (unless the level where the inhibit originated is now the highest active level). The context of the highest active level will be restored (again, unless the level where the inhibit originated is now the highest active level).

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\ =\$Sn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The P-bit in the S-register must be set to 1 (i.e., the central processor must be in the privileged state) for this instruction to be executed. If the P-bit is not set to 1, the unprivileged use of a privileged operation is signified by a trap to trap vector 13. (Traps and trap handling are described in the Executive and Input/Output manual.)

The contents of the S-register are affected as follows:

- o Bits 10 through 15 of the S-register will be set to indicate the priority level at which processing continues after execution of the LEV instruction.

LLH

Instruction:
Load logical half-word (byte) into R-register

Type:
DO

Source Language Format:

$$\Delta\text{LLH}\Delta\left\{\begin{array}{l}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,address-expression}$$

Description:
Loads the contents of the location specified in the address expression, as described below, into the R-register identified in the first operand.

- o If the address expression specifies =$Rn the rightmost byte of that R-register is loaded into the R-register specified by the first operand.
- o If the address expression specifies memory addressing *without* indexing, or an immediate operand addressing format, the leftmost byte of the word found at the specified location is loaded into the R-register.
- o If the address expression specifies memory addressing *with* indexing, the index register is aligned to count bytes relative to the leftmost byte of the word specified. The byte thus addressed is loaded into the R-register.

In all cases, the selected byte is loaded into the rightmost byte of the R-register, with 0's loaded into the leftmost byte.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$\text{Bn}\\=\$\text{S}\end{array}\right\}$ register addressing

Short displacement addressing

Specialized addressing

LNJ

Instruction:

Load B-register and jump

Type:

DO

Source Language Format:

$$\Delta\text{LNJ}\Delta\left\{\begin{array}{l}\$\text{Bn}\\\text{X'n'}\\\text{n}\end{array}\right\},\left\{\begin{array}{l}\text{P-relative-address}\\\text{immediate-memory-address}\\\text{B-relative-address}\end{array}\right\}$$

Description:

Loads the address of the next sequential instruction into the B-register identified in the first operand, and jumps to the location specified in the second operand.

If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the second operand is executed. The last instruction in the subroutine should be:

```
JMP $Bn
```

MCL

Instruction:

Call monitor via trap

Type:

GE

Source Language Format:

ΔMCLΔ

Description:

Calls monitor by a trap to trap vector 1.

MLV

Instruction:

Multiply by value

Type:

SI

Source Language Format:

$$\Delta\text{MLV}\Delta\begin{Bmatrix}\text{\$Rn}\\ \text{X'n'}\\ \text{n}\end{Bmatrix},[=]\begin{Bmatrix}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{internal-value-expression}\\ \text{fixed-point-constant}\end{Bmatrix}$$

Description:
Multiplies the contents of the R-register identified in the first operand by the 8-bit value (with sign extended) specified in the second operand. The result is saved in the first operand R-register.

If R7 is identified as the first operand R-register, the result (double-precision format) is saved in R6 and R7, with the most significant part in R6 and the least significant in R7.

The contents of the I-register are affected as follows:

- o If the result is more than 2^{15}-1 (32767) or less than -2^{15} (-32768) (except if R7 is specified), the OV-bit is set to 1; otherwise; it is set to 0.

MTM

Instruction:
Modify or test M-register

Type:
DO

Source Language Format:

$$\Delta\text{MTM}\Delta\begin{Bmatrix}\text{\$Mn}\\ \text{X'n'}\\ \text{n}\end{Bmatrix}\text{,address-expression}$$

Description:
Modifies or tests the contents of the M-register identified in the first operand with the contents (mask) of the location or R-register specified by the address expression.

The mask is treated as two 8-bit fields; then, depending on the content of corresponding bits in the two fields (i.e., bit 1 in the first field and bit 1 in the second; bit 2 in the first field and bit 2 in the second; etc.), the corresponding bit in the M-register (i.e , if bit 1 in the two mask fields, then bit 1 in the M-register) is altered as described below:

- o If bit n in the first mask field is 1, the corresponding bit in the M-register is loaded with the contents of the corresponding bit from the second mask field (i.e., M-register is modified).
- o If bit n in the first mask field is 0 and the same bit in the second mask field is 1, the corresponding bit in the M-register is inclusively ORed with the contents of the B-bit in the I-register. If the result of the ORing is 1, the B-bit is set to 1; otherwise, it is set to 0 (i.e., M-register is tested).

- If bit n in the first mask field is 0 and the same bit in the second mask field is 0, the corresponding bit in the M-register is neither modified nor tested.

NOTE: The assembly language instructions LEV, SAVE, and STM store the contents of the M-register in a form suitable for reloading by MTM.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\=\$Sn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

MUL

Instruction:
Multiply R-register

Type:
DO

Source Language Format:

$$\Delta\text{MUL}\Delta\left\{\begin{array}{c}\$Rn\\X'n'\\n\end{array}\right\}\text{,address-expression}$$

Description:
Multiplies the contents of the R-register identified in the first operand by the contents of the location or R-register specified in the address expression. The result is saved in the first operand R-register.

If R7 is identified as the first operand R-register, the result (double-precision format) is saved in R6 and R7, with the most significant part in R6 and the least significant in R7.

The contents of the I-register are affected as follows:

- If the product is more than 2^{15} -1 (32767) or less than -2^{15} (-32768) (except if R7 is specified), the OV-bit is set to 1; otherwise, it is set to 0.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\=\$Sn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

NEG

Instruction:
Negate

Type:
SO

Source Language Format:
ΔNEGΔaddress-expression

Description:
Two's complements the contents of the location or R-register specified in the address expression.

The contents of the I-register are affected as follows:

- o If a carry occurs during the operation, the C-bit is set to 1; otherwise, it is set to 0.
- o If the value complemented was -32768, the OV-bit is set to 1; otherwise, it is set to 0.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\\text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

NOP

Instruction:
No operation

Type:
BI

Source Language Format:

$$\Delta\text{NOP}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\\text{direct-P-relative-address}\\\text{short-displacement-address}\end{array}\right\}$$

Description:
Performs no operation.

If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion of the trace procedure or if the J-bit contains a binary 0, processing continues with the next sequential instruction in the program.

OR

Instruction:
Inclusive OR with R-register

Type:
DO

Source Language Format:

$$\Delta\text{OR}\Delta \left\{\begin{matrix}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\} \text{,address-expression}$$

Description:
Inclusively ORs the contents of the R-register identified in the first operand with the contents of the location or R-register specified in the address expression. The result is saved in the first operand R-register.
The following chart illustrates the result of inclusively ORing bits:

First operand bit:	0	0	1	1
Second operand bit:	1	0	1	0
Result:	1	0	1	1

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{matrix}=\$\text{Bn}\\ =\$\text{Sn}\end{matrix}\right\} \text{register addressing}$$

Short displacement addressing
Specialized addressing

ORH

Instruction:
Half-word (byte) inclusive OR with R-register

Type:
DO

Source Language Format:

$$\Delta\text{ORH}\Delta \left\{\begin{matrix}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\} \text{,address-expression}$$

Description:
Inclusively OR's the contents of the R-register identified in the first operand with the contents of the byte specified in the address expression.

Prior to the operation, the byte operand is internally expanded to word length by extending the sign through the eight high-order bit positions. The byte selected to participate in the operation is determined by the format of the address expression, as follows:

- o Register Addressing (=$Rn): The rightmost byte of the register is selected.
- o Memory Addressing *Withhout* Indexing: Immediate Memory Addressing: The leftmost byte of the word at the designated memory address is selected.

- o Memory Addressing *With* Indexing: The memory address indicates a starting point. The index register contains an arithmetic value to be added to the starting point. The value specifies the number of bytes before or after the starting point needed to reach the byte selected for the operation.

The following chart illustrates the result of inclusively ORing bits:

First operand bit:	0	0	1	1
Second operand bit:	1	0	1	0
Result:	1	0	1	1

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\=\$Sn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

RSTR

Instruction:
Restore context

Type:
SO

Source Language Format:

$$\Delta\text{RSTR}\Delta\left\{\begin{array}{l}\text{immediate-memory-address}\\\text{B-relative-address}\\\text{P-relative-address}\end{array}\right\},\left\{\begin{array}{l}\text{integer-constant}\\\text{string-constant}\\\text{internal-value-label}\\\text{external-value-label}\\\text{internal-value-expression}\\\text{fixed-point-constant}\end{array}\right\}$$

Description:
Restores the registers specified in the second operand mask starting from the location specified in the address expression.

The second operand is a mask that specifies which registers are to be restored. If the mask is all zeros, the contents of R1 are used as the mask.

Depending on which bits in the specified mask are set to 1, the registers that can be restored are as follows:

Bit:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	R1	R2	R3	R4	R5	R6	R7	I	B1	B2	B3	B4	B5	B6	B7

This mask should be the same as the one used to save the registers (see the SAVE instruction).

RTCF

Instruction:
Real-time clock off

Type:
GE

Source Language Format:

```
    ΔRTCFΔ
```

Description:
Disables real-time clock interrupts.

The P-bit in the S-register must be set to 1 (i.e., the central processor must be in the privileged state) for this instruction to be executed. If not, the unprivileged use of a privileged operation is signified by a trap to trap vector 13.

RTCN

Instruction:
Real-time clock on

Type:
GE

Source Language Format:

```
ΔRTCNΔ
```

Description:
Enables real-time clock interrupts, which will occur only when the real-time clock interrupt level is higher than the priority interrupt level specified in the S-register.

The P-bit in the S-register must be set to 1 (i.e., the central processor must be in the privileged state) for this instruction to be executed. If not, the unprivileged use of a privileged operation is signified by a trap to trap vector 13.

For a detailed description of traps and trap handling procedures (i.e., trap handlers), refer to the Executive and Input/Output manual.

For a detailed description of interrupts, refer to the Handbook.

RTT

Instruction:
Return from trap

Type:
GE

Source Language Format:

ΔRTTΔ

Description:

Restores the registers that were saved in the trap save area when the trap was entered; restores the central processor to the nonprivileged state if entering the trap caused the state to change from nonprivileged to privileged; returns the trap save area block to the trap save area memory pool; returns control to the next instruction to be executed (determined by the event that caused the trap and/or by the trap handler).

SAD

Instruction:

Scientific add

Type:

DO

Source Language Format;

$$\Delta\text{SAD}\Delta \left\{\begin{matrix}\$\text{Sn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\} \text{,address-expression}$$

Description:

Adds the floating-point or integer value in the location, scientific accumulator, or R-register identified in the second operand to the contents of the scientific accumulator specified in the first operand. The result is saved in the scientific accumulator.

This instruction uses the optional Scientific Instruction Processor (SIP). A Floating-Point Simulator is available to allow this instruction to be executed on systems that do not include an SIP. Information on the Floating-Point Simulator is available in the Executive and Input/Output manual.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

If register addressing is used, the value forms are:

$$=\$\text{R} \left\{\begin{matrix}4\\5\\6\\7\end{matrix}\right\}$$

If =$R7 is specified, the 32-bit value contained in the register pair formed by R6 and R7 becomes the operand.

=$Sn

=$Sn

If immediate operand addressing is used, you must provide a floating-point constant or hexadecimal string constant in suitable floating-point format.

If the second operand is =$R4, =$R5, =$R6, or =$R7, the integer value contained in the specific R-register is internally converted to floating-point format before it is added to the S-register by the first operand.

Scientific Indicator Settings:

EU: set to 1 on exponent underflow; otherwise, set to 0.
PE: set to 1 if nonzero bits are lost during right shift; otherwise, set to 0.

SAL

Instruction:
Single-shift arithmetic-left

Type:
SHS

Source Language Format:

$$\Delta\text{SAL}\Delta \left\{ \begin{array}{l} \$\text{Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{,internal-value-expression}$$

Description:
Shifts the contents of the R-register identified in the first operand left the number of bit positions specified in the internal value expression. The bit positions vacated by the shift are filled with binary 0's.

The contents of the I-register are affected as follows:

- o If the contents of bit 0 in the R-register change, the OV-bit is set to 1; otherwise, it is set to 0.

The internal value expression must be $\geqslant 0$ and $\leqslant 15$.
If the internal value expression equals 0, the contents are shifted left the number derived by using the value in bits 12 through 15 of general register R1.

SAR

Instruction:
Single-Shift arithmetic-right

Type:
SHS

Source Language Format:

$$\Delta\text{SAR}\Delta \left\{ \begin{array}{l} \$\text{Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{,internal-value-expression}$$

Description:
Shifts the contents of the R-register identified in the first operand right the number of bit positions specified in the internal value expression. The bit positions vacated by the shift are filled with the sign value originally contained in bit 0.

The contents of the I-register are affected as follows:

- o C-bit contains the last binary digit shifted out of the R-register.

The internal value expression must be $\geqslant 0$ and $\leqslant 15$.
If the internal value expression equals 0, the contents are shifted right the number derived by using the value in bits 12 through 15 of general register R1.

SAVE

Instruction:
Save context

Type:
SO

Source Language Format:

$$\Delta\text{SAVE}\Delta \left\{\begin{array}{l}\text{immediate-memory-address}\\ \text{B-relative-address}\\ \text{P-relative-address}\end{array}\right\}, \left\{\begin{array}{l}\text{integer-constant}\\ \text{string-constant}\\ \text{internal-value-label}\\ \text{external-value-label}\\ \text{internal-value-expression}\\ \text{fixed-noint-constant}\end{array}\right\}$$

Description:
Saves the registers specified in the second operand starting at the location specified in the address expression.

The second operand is a mask that specifies which registers are to be saved. Each bit in the mask represents a particular register which can be saved, as shown below:

Bit:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	R1	R2	R3	R4	R5	R6	R7	I	B1	B2	B3	B4	B5	B6	B7

If a mask bit is set to 1, the corresponding register is saved. If a mask bit is 0, the corresponding register is not saved. If the mask is all 0's, the contents of R1 are used as the mask.

The registers are saved in reverse order. For example, if the second operand specified Z'CA01' (which, when translated into binary is 1100 1010 0000 0001), indicating that registers M1, R1, R4, R6, and B7 are to be saved, the context save area will contain the registers starting with B7 and ending with M1.

SBE

Instruction:
Scientific branch on equal

Type:
BI

Source Language Format:

$$\Delta\text{SBE}\Delta \begin{Bmatrix}\text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets both the SL- and SG-bits of the SI-register to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5.

SBEU

Instruction:
Scientific branch on exponent underflow

Type:
BI

Source Language Format:

$$\Delta\text{SBEU}\Delta \begin{Bmatrix}\text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the EU-bit in the SI-register to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed. If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5.

SBEZ

Instruction:
Branch if scientific accumulator equal to 0

Type:
BR

Source Language Format;

$$\Delta\text{SBEZ}\Delta \begin{Bmatrix}\$\text{Sn}\\ \text{X'n'}\\ \text{n}\end{Bmatrix} , \begin{Bmatrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the second operand if the scientific accumulator identified in the first operand contains a floating-point value algebraically equal to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5. $S1 is the only scientific accumulator register supported by the simulator.

SBG

Instruction:
Scientific branch on greater than

Type:
BI

Source Language Format:

$$\Delta\text{SBG}\Delta \begin{Bmatrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{Bmatrix}$$

Description;
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the SG-bit in the SI-register to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5.

SBGE

Instruction:
Scientific branch on greater than or equal

Type:
BI

Source Language Format:

$$\Delta\text{SBGE}\Delta\ \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the S1-bit of the SI-register to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is enterd via trap vector 5.

SBGEZ

Instruction:
Branch if scientific accumulator equal to or greater than 0

Type:
BR

Source Language Format:

$$\Delta\text{SBGEZ}\Delta\ \left\{\begin{array}{l}\text{\$Sn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\ ,\ \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the second operand if the scientific accumulator identified in the first operand contains a nonnegative floating-point value.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. (Upon completion, the trace procedure automatically branches to the

address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5. $S1 is the only scientific accumulator register supported by the simulator.

SBGZ

Instruction:
Branch if scientific accumulator greater than 0

Type:
BR

Source Language Format:

$$\Delta\text{SBGZ}\Delta \left\{\begin{array}{l}\$\text{Sn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} , \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the second operand if the scientific accumulator identified in the first operand contains a positive floating-point value.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5. $S1 is the only scientific accumulator register supported by the simulator.

SBL

Instruction:
Scientific branch if less than

Type:
BI

Source Language Format:

$$\Delta\text{SBL}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the S1-bit of the SI-register to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instructions sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5.

SBLE

Instruction:
Scientific branch on less than or equal

Type:
BI

Source Language Format:

$$\Delta\text{SBLE}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the SG-bit in the SI-register to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5.

SBLEZ

Instruction:
Branch if scientific accumulator equal to or less than 0

Type:
BR

Source Language Format:

$$\Delta\text{SBLEZ}\Delta \left\{\begin{array}{l}\text{\$Sn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} , \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the second operand if the scientific accumulator identified in the first operand contains a floating-point value algebraically equal to or less than 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5. $S1 is the only scientific accumulator register supported by the simulator.

SBLZ

Instruction:
Branch if scientific accumulator less than 0

Type:
BR

Source Language Format:

$$\Delta\text{SBLZ}\Delta \quad \left\{\begin{matrix}\text{\$Sn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\} , \left\{\begin{matrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{matrix}\right\}$$

Description:
Branches to the location specified in the second operand if the scientific accumulator identified in the first operand contains a negative floating-point value.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5. $S1 is the only scientific accumulator register supported by the simulator.

SBNE

Instruction:
Scientific branch on not equal

Type:
BI

Source Language Format

$$\Delta\text{SBNE}\Delta \quad \left\{\begin{matrix}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{matrix}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets either the SL- or SG-bit of the SI-register to 1.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5.

SBNEU

Instruction:
Scientific branch on not exponent underflow

Type:
BI

Source Language Format:

$$\Delta\text{SBNEU}\Delta \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the EU-bit of the SI-register to 0.

Action if Branch Occurs:
If the J-bit of the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

SBNEZ

Instruction:
Branch if scientific accumulator not equal to 0

Type:
BR

Source Language Format:

$$\Delta\text{SBNEZ}\Delta \left\{\begin{array}{l}\text{\$Sn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} , \left\{\begin{array}{l}\text{direct-IMA}\\ \text{direct-P-relative-address}\\ \text{short-displacement-address}\end{array}\right\}$$

Description:
Branches to the location specified in the second operand if the scientific accumulator identified in the first operand contains a floating-point value not algebraically equal to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator is entered via trap vector 5. $S1 is the only scientific accumulator register supported by the simulator.

SBNPE

Instruction:
Scientific branch on not precision error

Type:
BI

Source Language Format:

$$\Delta\text{SBNPE}\Delta \begin{Bmatrix}\text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the PE-bit of the SI-register to 0.

Action if Branch Occurs:
If the J-bit of the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

SBNSE

Instruction:
Scientific branch on not significance error

Type:
BI

Source Language Format:

$$\Delta\text{SBNSE}\Delta \begin{Bmatrix}\text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address}\end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the SE-bit of the SI-register to 0.

Action if Branch Occurs:
If the J-bit in the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

SBPE

Instruction:
Scientific branch on precision error

Type:
BI

Source Language Format:

$$\Delta\text{SBPE}\Delta \begin{Bmatrix} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the PE-bit of the SI-register to 1.

Action if Branch Occurs:
If the J-bit of the M1-register contains a binary 1, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

SBSE

Instruction:
Scientific branch on significance error

Type:
BI

Source Language Format:

$$\Delta\text{SBSE}\Delta \begin{Bmatrix} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{Bmatrix}$$

Description:
Branches to the location specified in the operand if the result of the most recent scientific comparison sets the SE-bit of the SI-register to 1.

Action if Branch Occurs:
If the J -bit of the M1-register contains a binary 1, the trace procedure is entered via trap vector 2. Upon completion, the trace procedure automatically branches to the address specified by the operand. In this case, or if the J-bit contains a binary 0, the instruction sequence starting at the location specified by the operand is executed.

SCL:

Instruction:
Single-shift closed-left

Type:
SHS

Source Language Format:

$$\Delta\text{SCL}\Delta \left\{ \begin{array}{c} \text{\$Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{,internal-value-expression}$$

Description:
Shifts the contents of the R-register identified in the first operand left the number of bit positions specified in the internal value expression. The bits shifted out of the register are placed in the bit positions vacated by shifted bits as they are shifting.

The internal value expression must be $\geqslant 0$ and $\leqslant 15$.
If the internal value expression equals 0, the contents are shifted left the number derived by using the value in bits 12 through 15 of general register R1.

SCR

Instruction:
Single-shift closed-right

Type:
SHS

Source Language Format:

$$\Delta\text{SCR}\Delta \left\{ \begin{array}{c} \text{\$Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{,internal-value-expression}$$

Description:
Shifts the contents of the R-register identified in the first operand right the number of bit positions specified in the internal value expression. The bits shifted out of the register are placed in the bit positions vacated by shifted bits as they are shifting.

The internal value expression must be $\geqslant 0$ and $\leqslant 15$.
If the internal value expression equals 0, the contents are shifted right the number derived by using the value in bits 12 through 15 of general register R1.

SCM

Instruction
Scientific compare

Type:
DO

Source Language Format:

$$\Delta\text{SCM}\Delta \left\{\begin{array}{l}\$\text{Sn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} \text{,address-expression}$$

Description:
Compares the contents of the scientific accumulator identified in the first operand to the floating-point or integer value in the location specified in the second operand.

Scientific Indicator Settings:

SG: Set to 1 if contents of the scientific accumulator are greater than the contents of the location; otherwise, set to 0.
SL: Set to 1 if contents of the scientific accumulator are less than the contents of the location; otherwise, set to 0.
PE: Set to 1 if nonzero bits are lost during right shift for scaling before comparison; otherwise, set to 0.

If the Scientific Information Processor (SIP) is not installed on this system, the instruction causes the Floating-Point Simulator to be entered via trap vector 3. Since the SI-register is not available if the SIP is not installed, the I-register is used as a substitute. The contents of the I-register are affected as follows:

- o If the contents of the register are greater than the contents of the location, the G-bit is set to 1; otherwise, it is set to 0.
- o If the contents of the register are less than the contents of the location, the L-bit is set to 1; otherwise, it is set to 0.
- o If the content of bit 7 of the register is not equal to the content of bit 7 of the location, the U-bit is set to 1; otherwise, it is set to 0.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques" except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

If register addressing is used, the valid forms are:

$$=\$\text{R} \left\{\begin{array}{c}4\\5\\6\\7\end{array}\right\}$$

=$Sn

If =$R7 is specified, the 32-bit value contained in the register pair formed by R6 and R7 becomes the operand.

($S1 is the only scientific accumulator register supported by the simulator.)

=$Sn

If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

If the second operand is =$R4, =$R5, =$R6, or =$R7, the integer value contained in the specific R-register is internally converted to floating-point format before it is compared to the S-register specified by the first operand.

SCZD

Instruction:
Scientific compare to zero (short-precision)

Type:
SO

Source Language Format:
ΔSCZDΔaddress-expression

Description:
Compares the short-precision floating-point value in the specified location or scientific accumulator to 0.

Scientific Indicator Settings:
SL: Set to 1 if contents of the location are less than 0; otherwise, set to 0.
PE: Set to 1 if nonzero bits are lost during right shift for scaling before comparison; otherwise, set to 0.

If the Scientific Information Processor (SIP) is not installed on this system, the instruction causes the Floating-Point Simulator to be entered via trap vector 3. Since the SI-register is not available if the SIP is not installed, the I-register is used as a substitute.

The contents of the I-register are affected as follows:

- o If the contents of the specified location do not equal 0, the G-bit is set to 1; otherwise, it is set to 0.
- o The L-bit is set to 0.
- o If bit 7 of the specified location equals 1, the U-bit is set to 1; otherwise, it is set to 0.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\=\$Rn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The only valid form of register addressing is:

=$Sn ($Sn is the only scientific accumulator register supported by the simulator.)

If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

SCZQ

Instruction:
Scientific compare to 0 (long-precision)

Type:
S0

Source Language Format:
ΔSCZQΔaddress-expression

Description:
Compares the floating-point value in the specified location or scientific accumulator to 0.

Scientific Indicator Settings:
SL: Set to 1 if contents of the location are less than 0; otherwise, set to 0.
PE: Set to 1 if nonzero bits are lost during right shift for scaling before comparison; otherwise, set to 0.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn } register addressing
=$Rn }
Short displacement addressing
Specialized addressing

The only valid form of register addressing is:
=$Sn

If immediate operand addressing is used, you must provide a string constant in suitable floating-point format.

SDI

Instruction:
Store Double word integer

Type:
SO

Source Language Format:
ΔSDIΔaddress-expression

Description:
Stores the contents of register R6 into the location specified by the address expression and the contents of register R7 into the next location.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\=\$Rn\\=\$Sn\end{array}\right\}$ register addressing NOTE: =$R3 and =$R5 are legal.

Short displacement addressing
Specialized addressing

SDV

Instruction:
Scientific divide

Type:
DO

Source Language Format:

$$\Delta SDV\Delta \left\{\begin{array}{l}\$Sn\\X'n'\\n\end{array}\right\} ,address\text{-}expression$$

Description:
Divides the contents of the scientific accumulator identified in the first operand by the contents of the location, scientific accumulator, or R-register specified in the second operand. The result is saved in the scientific accumulator (except for the remainder, which is ignored).

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator is entered via trap vector 3. $S1 is the only scientific accumulator register supported by the simulator.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

If register addressing is used, the valid forms are:

$$=\$R \left\{\begin{array}{c}4\\5\\6\\7\end{array}\right\}$$

If =$R7 is specified, the 32-bit value contained in the register pair formed by R6 and R7 becomes the operand.

=$Sn

=$Sn

If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

If the second operand is =$R4, =$R5, =$R6, or =$R7, the integer value contained in the specific R-register is internally converted to floating-point format before it is divided into the S-register specified by the first operand.

Scientific Indicator Settings:
EU: Set to 1 on exponent underflow; otherwise, set to 0.
PE: Set to 1 if nonzero bits are lost during right shift; otherwise, set to 0.

SLD

Instruction:
Scientific load

Type:
DO

Source Language Format:

$$\Delta\text{SLD}\Delta \left\{\begin{array}{l}\$\text{Sn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} \text{,address-expression}$$

Description:
Loads the contents of the location, scientific accumulator, or R-register identified in the second operand into the scientific accumulator identified in the first operand.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator is entered via trap vector 3. $S1 is the only scientific accumulator register supported by the simulator.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

If register addressing is used, the valid forms are:

$$=\$\text{R} \left\{\begin{array}{c}4\\5\\6\\7\end{array}\right\}$$

If =$R7 is specified, the 32-bit value contained in the register pair formed by R6 and R7 becomes the operand.

=$Sn

If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

If the second operand is =$R4, =$R5, =$R6, or =$R7, the integer value contained in the specific R-register is internally converted to floating-point format before it is loaded to the S-register specified by the first operand.

Scientific Indicator Settings:
EU: Set to 1 on exponent overflow; otherwise, set to 0.
PE: Set to 1 if nonzero bits are lost during right shift; otherwise, set to 0.

SML

Instruction:
Scientific multiply

Type:
DO

Source Language Format:

$$\Delta\text{SML}\Delta \left\{\begin{matrix}\$\text{Sn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\} \text{,address-expression}$$

Description:
Multiplies the contents of the scientific accumulator identified in the first operand by the contents of the location, scientific accumulator, or R-register specified in the second operand. The result is saved in the scientific accumulator.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator is entered via trap vector 3. $S1 is the only scientific accumulator register supported by the simulator.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

If register addressing is used, the valid forms are:

$$=\$\text{R} \left\{\begin{matrix}4\\5\\6\\7\end{matrix}\right\}$$

If =$R7 is specified, the 32-bit value contained in the register pair formed by R6 and R7 becomes the operand.

=$Sn

=$Sn
If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

If the second operand is an R-register the integer value contained in the specific R-register is internally converted to floating-point format before it is multiplied to the S-register specified by the first operand.

Scientific Indicator Settings:
EU: Set to 1 on exponent underflow; otherwise, set to 0.
PE: Set to 1 if nonzero bits are lost during right shift; otherwise, set to 0.

SNGD

Instruction:
Scientific negate (short-precision)

Type:
SO

Source Language Format:

ΔSNGDΔaddress-expression

Description:
Negate the short-precision floating-point number at the location or scientific accumulator specified in the operand.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator is entered via trap vector 3.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\ =\$Rn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The only valid form of register addressing is:

=$Sn ($S1 is the only scientific accumulator register supported by the simulator.)

SNGQ

Instruction:
Scientific negate (long-precision)

Type:
SO

Source Language Format;
ΔSCGQΔadddress-expression

Description:
Negate the long-precision floating-point number at the location or scientific accumulator specified in the operand.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}=\$Bn\\ =\$Rn\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The only valid form of register addressing is:

=$Sn ($S1 is the only scientific accumulator register supported by the simulator.) If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

SOL

Instruction:
Single-shift open-left

Type:
SHS

Source Language Format:

$$\Delta\text{SOL}\Delta\left\{\begin{array}{c}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,internal-value-expression}$$

Description:
Shifts the contents of the R-register identified in the first operand left the number of bit positions specified in the internal value expression. The bit positions vacated by the shift are filled with binary 0's.

The contents of the I-register are affected as follows:

- o C-bit contains the last binary digit shifted out of the R-register.

The internal value expression must be $\geqslant 0$ and $\leqslant 15$.
If the internal value expression equals 0, the contents are shifted right the number derived by using the value in bits 12 through 15 of general register R1.

SOR

Instruction:
Single-shift open-right

Type:
SHS

Source Language Format:

$$\Delta\text{SOR}\Delta\left\{\begin{array}{c}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,internal-value-expression}$$

Description:
Shifts the contents of the R-register identified in the first operand right the number of bit positions specified in the internal value expression. The bit positions vacated by the shift are filled with binary 0's.

The contents of the I-register are affected as follows:

- o C-bit contains the last binary digit shifted out of the R-register.

The internal value expression must be $\geqslant 0$ and $\leqslant 15$.
If the internal value expression equals 0, the contents are shifted right the number derived by using the value in bits 12 through 15 of general register R1.

SRM

Instruction:
Store register masked

Type:
DO

Source Language Format:

$$\Delta\text{SRM}\Delta\left\{\begin{array}{l}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,address-expression,mask}$$

Description:
AND's the contents of the R-register identified in the first operand with the mask; AND's the contents of the location or R-register specified by the address expression with the complement of the mask; then inclusively OR's the values obtained from the two AND's. Then, stores the result in the second operand.

See the AND and OR instructions described in this section.

If the mask =0, the contents of R1 are used in place of the mask.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following.

$$\left.\begin{array}{l}=\$\text{Bn}\\ =\$\text{Sn}\end{array}\right\}\text{register addressing}$$

Short displacement addressing
Specialized addressing

SSB

Instruction:
Scientific subtract

Type:
DO

Source Language Format:

$$\Delta\text{SSB}\Delta\left\{\begin{array}{l}\$\text{Sn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,address-expression}$$

Description:
Subtracts the contents of the location, scientific accumulator, or R-register identified in the second operand from the contents of the scientific accumulator specified in the first operand. The result is saved in the scientific accumulator.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator is entered via trap vector 3. $S1 is the only scientific accumulator register supported by the simulator.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

If register addressing is used, the valid forms are:

$$=\$R \begin{Bmatrix} 4 \\ 5 \\ 6 \\ 7 \end{Bmatrix}$$

If =$R7 is specified, the 32-bit value contained in the register pair formed by R6 and R7 becomes the operand.

=$Sn

=$Sn

If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

If the second operand is an R-register, the integer value contained in the specific R-register is internally converted to floating-point format before it is subtracted from the S-register specified by the first operand.

Scientific Indicator Settings:
EU: Set to 1 on exponent underflow; otherwise, set to 0.
PE: Set to 1 if nonzero bits are lost during right shift; otherwise, set to 0.

SST

Instruction:
Scientific store

Type:
DO

Source Language Format:

$$\Delta\text{SST}\Delta \begin{Bmatrix} \$Sn \\ X'n' \\ n \end{Bmatrix} \text{,address-expression}$$

Description:
Stores the contents of the scientific accumulator identified in the first operand in the location, scientific accumulator, or R-register specified in the address expression.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator is entered via trap vector 3. $S1 is the only scientific accumulator register supported by the simulator.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

If register addressing is used, the valid forms are:

$$=\$R \left\{ \begin{matrix} 4 \\ 5 \\ 6 \\ 7 \end{matrix} \right\}$$

If =$R7 is specified, the 32-bit value contained in the register pair formed by R6 and R7 becomes the operand.

=$Sn

=$Sn

If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

If the second operand is an R-register, the floating-point value contained in the specific scientific accumulator is converted to integer format before it is stored into the specified R-register.

Scientific Indicator Settings:
- EU: Set to 1 on exponent underflow; otherwise, set to 0.
- SE: Set to 1 if resultant floating-point value has a zero fraction; otherwise, set to 0.
- PE: Set to 1 if nonzero bits are lost during right shift; otherwise, set to 0.

SSW

Instruction:
Scientific swap

Type:
DO

Source Language Format:

$$\Delta SSW\Delta \left\{ \begin{matrix} \$Sn \\ X'n' \\ n \end{matrix} \right\} \text{,address-expression}$$

Description:
Swaps the contents of the scientific accumulator identified in the first operand with

the contents of the location, scientific accumulator, or R-register specified in the address expression.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator is entered via trap vector 3. $S1 is the only scientific accumulator register supported by the simulator.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
Short displacement addressing
Specialized addressing

If register addressing is used, the valid forms are:

$$=\$R\left\{\begin{matrix}4\\5\\6\\7\end{matrix}\right\}$$

=$Sn

=$Sn

If immediate operand addressing is used, you must provide a floating-point constant or string constant in suitable floating-point format.

If an R-register is specified as the second operand, the value specified by the first operand is internally converted to integer format, and the value specified by the second operand is internally converted to floating-point. These converted values are then interchanged.

$\left.\begin{matrix}=\$Rn\\=\$Sn\end{matrix}\right\}$ register addressing
Short displacement addressing
Specialized addressing
Immediate operand addressing with an internal value expression

Scientific Indicator Settings:
- EU: Set to 1 on exponent underflow; otherwise, set to 0.
- SE: Set to 1 if resultant floating-point value has a zero fraction; otherwise, set to 0.
- PE: Set to 1 if nonzero bits are lost during right shift; otherwise, set to 0.

STB

Instruction:
Store B-register

Type:
DO

Source Language Format:

$$\Delta\text{STB}\Delta \left\{\begin{array}{c}\$\text{Bn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} \text{,address-expression}$$

Description:
Stores the contents of the B-register identified in the first operand in the location or B-register identified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{array}{l}=\$\text{Rn}\\ =\$\text{Sn}\end{array}\right\}\text{ register addressing}$$
Short displacement addressing
Specialized addressing
Immediate operand addressing with an internal value expression

STH

Instruction:
Store R-register halfword (byte)

Type:
DO

Source Language Format:

$$\Delta\text{STH}\Delta \left\{\begin{array}{c}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\} \text{,address-expression}$$

Description:
Stores the rightmost byte of the R-register identified in the first operand into the location specified in the address expression as follows:

- o If the address expression specifies the =$Rn addressing form, the byte is stored in the rightmost byte of the specified R-register.
- o If the address expression specifies memory addressing *without* indexing, the byte is stored in the leftmost byte of the word found at the specified location.
- o If the address expression specifies memory addressing *with* indexing, the index register is aligned to count bytes relative to the leftmost byte of the word specified. The R-register byte is thus stored in the memory byte thus addressed.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{array}{l}=\$\text{Bn}\\ =\$\text{Sn}\end{array}\right\}\text{ register addressing}$$
Short displacement addressing
Specialized addressing

STM

Instruction:
Store M-register

Type:
DO

Source Language Format:

$$\Delta\text{STM}\Delta \left\{ \begin{array}{l} \$\text{Mn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{,address-expression}$$

Description:
Stores the 8-bit M-register identified in the first operand in the right half-word of the location or R-register specified in the address expression; the left half-word of the location is filled with 1's.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left. \begin{array}{l} =\$\text{Bn} \\ =\$\text{Sn} \end{array} \right\}$ register addressing
Short displacement addressing
Specialized addressing

STR

Instruction:
Store R-register

Type:
DO

Source Language Format:

$$\Delta\text{STR}\Delta \left\{ \begin{array}{l} \$\text{Rn} \\ \text{X'n'} \\ \text{n} \end{array} \right\} \text{,address-expression}$$

Description:
Stores the contents of the R-register identified in the first operand in the location identified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left. \begin{array}{l} =\$\text{Bn} \\ =\$\text{Sn} \end{array} \right\}$ register addressing
Short displacement addressing
Specialized addressing

STS

Instructions:
Store S-register

Type:
SO

Source Language Format:

```
    ΔSTSΔaddress-expression
```

Description:
Stores the contents of the system status (s) register in the location or R-register identified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\\text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

SUB

Instruction:
Subtract from R-register

Type:
DO

Source Language Format:

$$\Delta\text{SUB}\Delta\left\{\begin{array}{l}\text{\$Rn}\\\text{X'n'}\\\text{n}\end{array}\right\}\text{,address-expression}$$

Description:
Subtracts the contents of the location or R-register identified in the address expression from the contents of the R-register specified in the first operand. The result is saved in the first operand R-register.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\\text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

The contents of the I-register are affected as follows:

- o If the result is more than 2^{15}-1 (32767) or less than -2^{15} (-32768), the OV-bit is set to 1; otherwise, it is set to 0.
- o If, during the subtraction, a carry occurs, the C-bit is set to 1; otherwise, it is set to 0.

SWB

Instruction:
Swap B-register

Type:
DO

Source Language Format:

$$\Delta\text{SWB}\Delta\left\{\begin{array}{c}\$\text{Bn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,address-expression}$$

Description:
Swaps the contents of the B-register identified in the first operand with the contents of the location or B-register specified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$$\left.\begin{array}{l}=\$\text{Rn}\\ =\$\text{Sn}\end{array}\right\}\text{ register addressing}$$
Short displacement addressing
Specialized addressing
Immediate operand addressing with an internal value expression

SWR

Instruction:
Swap R-register

Type:
DO

Source Language Format:

$$\Delta\text{SWR}\Delta\left\{\begin{array}{c}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{array}\right\}\text{,address-expression}$$

Description:
Swaps the contents of the R-register identified in the first operand with the contents of the location or R-register specified in the address expression.

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing
Immediate operand addressing

WDTF

Instruction:
Watchdog timer off

Type:
GE

Source Language Format:
    ΔWDTFΔ

Description:
Disables the watchdog timer interrupt (i.e., level 1 interrupt).

The P-bit in the S-register must be set to 1 (i.e., the central processor must be in the privileged state) for this instruction to be executed. If not, the unprivileged use of a privileged operation is signified by a trap to trap vector 13.

WDTN

Instruction:
Watchdog timer on

Type:
GE

Source Language Format:
    ΔWDTNΔ

Description:
Enables watchdog timer interrupt (i.e., level 1 interrupt).

The P-bit in the S-register must be set to 1 (i.e., the central processor must be in the privileged state) for this instruction to be executed. If not, the unprivileged use of a privileged operation is signified by a trap to trap vector 13.

XOH

Instruction:
Half-word (byte) exclusive OR with R-register

Type:
DO

Source Language Format:

$$\Delta\text{XOH}\Delta\left\{\begin{matrix}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\}\text{,address-expression}$$

Description:
Exclusively OR's the contents of the R-register identified in the first operand with the contents of the byte specified in the address expression.

Prior to the operation, the byte operand is internally expanded to word length by extending the sign through the eight high-order bit positions. The byte selected to participate in the operation is determined by the format of the address expression, as follows:

- o Register Addressing (=$Rn): The rightmost byte of the register is selected.
- o Memory Addressing *Without* Indexing; Immediate Memory Addressing: The leftmost byte of the word at the designated memory address is selected.
- o Memory Addressing *With* Indexing: The memory address indicates a starting point. The index register contains an arithmetic value to be added to the starting point. The value specifies the number of bytes before or after the starting point needed to reach the byte selected for the operation.

The following chart illustrates the result of exclusively ORing bits:

First operand bit:	0	0	1	1
Second operand bit:	1	0	1	0
Result:	1	0	0	1

The address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

=$Bn register addressing
=$Sn
Short displacement addressing
Specialized addressing

XOR

Instruction:
Exclusive OR with R-register

Type:
DO

Source Language Format:

$$\Delta\text{XOR}\Delta\left\{\begin{matrix}\$\text{Rn}\\ \text{X'n'}\\ \text{n}\end{matrix}\right\}\text{,address-expression}$$

Description:
Exclusively OR's the contents of the R-register identified in the first operand with the contents of the location or R-register specified in the address expression. The result is saved in the first operand R-register.

The following chart illustrates the result of exclusively ORing bits:

First operand bit:	0	0	1	1
Second operand bit:	1	0	1	0
Result:	1	0	0	1

The address expression can take any of the forms described earlier in this section under "Addressing Techniques" except for the following:

$\left.\begin{array}{l}\text{=\$Bn}\\ \text{=\$Sn}\end{array}\right\}$ register addressing
Short displacement addressing
Specialized addressing

SECTION 6
MACRO FACILITY

The Macro Preprocessor is a program development tool that provides a convenient method for including in a source module specified sequences of statements that are specified in a macro routine.

A macro routine is a block of source code that is written only once and can be included multiple times within a given source module. A single statement, known as a macro call, is specified in the source module each time the sequence of statements is to be included. A source module containing one or more macro calls is called an unexpanded source module. Macro routines can be at the beginning of a source module or in a macro library; those occurring within a source module are called inline macro routines.

NOTE: Honeywell provides a library of macro routines that support MLCP programming. (See the Software Overview and System Conventions, and the MLCP Programmer's Reference Manual.)

The Macro Preprocessor produces an expanded source module which is used as input to the Assembler. The expanded source module may contain an error flag for each nonfatal error.[1] (Nonfatal error flags are described in Appendix F.) If a fatal error occurs, processing terminates, an error message is issued through the system console, and control returns to the Command Processor. (Error messages issued by the Macro Preprocessor are described in the Operator's Guide.)

ORDER OF STATEMENTS WITHIN A SOURCE MODULE

Statements within a source module must be in the order listed below:

1. TITLE assembler control statement.
2. LIBM macro control statements and/or macro routines delimited by MAC and ENDM macro control statements.

   (Optional) LIST or NLST Assembler control statement.

   NOTE: LIBM statements, macro routines, and a LIST or NLST statement can be intermixed.

3. Statements that constitute the body of the source module; includes macro calls.
4. END Assembler control statement.

Macro control statements and macro calls are described in this section. Assembler control statements are described in Section 4.

MACRO ROUTINES

A macro routine can be either generalized or specialized. A generalized macro routine causes a fixed expansion in the source module. A specialized macro routine permits specified values to be included in the expanded source module.

[1] The expanded source module includes error flags only if the IC argument was specified in the load command to the Command Processor. (See "Input to Command Processor Before Macro Preprocessor is Loaded" in Section 7 of the Program Development Tools manual.)

The following information is described below.

- o Creating a macro routine
- o Specializing a macro routine
- o Including protection operators
- o Situating a macro routine

Creating a Macro Routine

A macro routine must be preceded by a MAC macro control statement and followed by an ENDM macro control statement.

MAC Macro Control Statement, Without Parameters

The MAC statement assigns a name to a macro routine; it must immediately precede every macro routine. MAC must be the last entry on the source line, or it must be immediately followed by a comma and an optional comment.

Format:

```
    macro-nameΔMAC[,comment]
```

macro-name

Name of the macro routine; must be a valid symbolic name. To include the macro routine within a source module, specify the macro name in a macro call.

NOTE: A macro routine can be specialized by including macro parameters in the MAC statement. (See "MAC Macro Control Statement, Including Parameters" later in this section.)

Contents of Macro Routine

A macro routine can include:

- o Macro control statements, excluding MAC and ENDM
- o Macro functions
- o Assembler control statements, excluding END
- o Assembly language statements

Macro control statements and macro functions are described in this section. Assembler control statements and assembly language statements are described in Sections 4 and 5, respectively.

ENDM Macro Control Statement

The ENDM statement designates the end of a macro routine, it must immediately follow each macro routine.

Format:

```
    [label]ΔENDM
```

label

Symbolic name that identifies the ENDM statement.

Specializing a Macro Routine by Parameter Substitution

In a given macro routine, up to 35 different macro parameters can be referenced. Parameters are named P1 to P9 and PA to PZ. Each parameter name must be preceded by a substitution operator (question mark) to indicate that substitution will occur; i.e., a value will be substituted.

Macro parameters can be assigned values in the MAC statement and/or in macro calls.

When a macro call is specified, each macro parameter reference in the requested macro routine is replaced with the parameter's value. If a parameter was assigned a value in the MAC statement and then assigned a different value in the macro call, the value specified in the macro call is included. If no value was specified in the MAC statement or in the macro call, the parameter is equal to a null ASCII character string; i.e., ''.

MAC Macro Control Statement, Including Parameters

The MAC statement assigns a name to a macro routine and optionally assigns values to macro parameters.

Format:

macro-nameΔMAC $\left[\Delta P_j[=v]\right]$ $\left[, P_k[=v]\right]$ ...

macro-name

Name of the macro routine being created; must be a valid symbolic name.

P_n

Macro parameter name; can be P1 to P9 or PA to PZ. Parameter names can be specified in any order.

NOTE: It may be impossible to specify all parameters on one source line. Parameters can be continued on the next line by replacing the last comma with a semicolon. (See "Assembly Language Source Statement Formats" in Section 3.)

=v

Value of macro parameter; can be any alphanumeric characters.

NOTE: To include a comma, space or horizontal tab as part of a parameter value, specify that character within apostrophes.

If a value is not specified, the corresponding parameter remains equal to a null ASCII character string.

Example:

This example illustrates an unexpanded source module that includes a MAC statement with parameters. The resulting expanded source module includes those parameter values.

Unexpanded source module:

```
        TITLE EXMPL

SAMPLE MAC P3=5;     Designates beginning of macro routine and
PB='6,'              assigns values to parameters P3 and PB
        LDV $R1=?P3)
                    > Statements to be included in source module
        LDR $R2=?PB)

FINI    ENDM         Designates end of macro routine
         :
        SAMPLE       Macro call requesting macro routine named
         :           SAMPLE
         :
```

Expanded source module:

```
TITLE EXMPL
   .
   .
   .
LDV $R1,=5       }Macro call replaced by contents of macro
LDR $R2,='6,'    }routine named SAMPLE
   .
   .
   .
```

Protection Operators

Protection operators are brackets; they enclose one or more characters that are not to be interpreted by the Macro Preprocessor. Protection operators can be included in macro routines and/or in statements that constitute the body of a source module.

NOTE: Brackets illustrated in each command's *Format* are not protection operators; they enclose optional characters.

Example:

This example illustrates an unexpanded source module, which includes protection operators, and the resulting expanded source module.

Unexpanded source module:

```
       TITLE EXMPL

SAMPLE MAC P7=3      Designates beginning of macro routine and
                     assigns value to parameter P7

       NEWA[?]P7     Substitution operator will not be inter-
                     preted by Macro Preprocessor, so no value
                     will be substituted

       NEWB ?P7      Reference to P7 will be replaced with its
                     value

       ENDM          Designates end of macro routine
        .
        .
        .
      [SAMPLE]       Not interpreted as macro call because name
        .            of macro routine is enclosed within protec-
        .            tion operators
        .
       SAMPLE        Macro call; in the expanded source module
                     will be replaced by contents of macro rou-
                     tine named SAMPLE
        .
        .
        .
```

Expanded source module:

```
TITLE EXMPL
  .
  .
  .
SAMPLE
  .
  .
  .
NEWA ?P7 }
         }    Contents of macro routine named SAMPLE
NEWB 3   }
  .
  .
  .
```

Protection operators cannot extend over operand or argument delimiters; to protect adjacent operands or arguments, enclose each one individually in brackets.

Example 1:

```
FOOΔ[AB],[CD]
```

The above macro call FOO designates that parameter P1 equals [AB] and parameter P2 equals [CD].

Example 2:

```
FOOΔ[AB,CD]
```

The above macro call FOO is *not* equivalent to the macro call illustrated in example 1. The macro call in example 2 specifies that parameter P1 equals [AB and parameter P2 equals CD].

If any part of a label or operation code is protected, the entire label or operation code is protected.

Example:

```
LAB[EL]LD[R] Δ$R1,=100
```

The above statement is considered to have label and operation code.

Protection operators do not appear in expanded source modules unless the operators are embedded in other protection operators. Embedded protection operators are removed from the expanded source module only if that module is reprocessed by the Macro Preprocessor. One level of embedded protection operators is removed each time the expanded source module is reprocessed.

Example 1:

```
NEWA[?]P7
```

The above statement would appear in the expanded source module as NEWA?P7.

Example 2:

```
DC A [BC[DEF]GH] I'
```

The above statement would appear in the expanded source module as DC'A BC[DEF]GHI '. Only the outermost protection operators are removed, unless the expanded source module is then reprocessed by the Macro Preprocessor.

Situating Macro Routines

Macro routines can be in the source module in which they are requested by macro call(s) and/or in macro libraries on a diskette volume. A macro library is a partitioned file whose members are macro routines. Each member must be a single macro routine that is referenced in a macro call by its member name. Its member name must be identical to the label of its MAC statement. There can be multiple macro libraries, but all libraries must be on the same diskette volume.

All macro routines within a source module must be at the beginning of the module. (See "Order of Statements Within a Source Module" earlier in this section.)

To place a macro routine in a macro library, use the Editor insert command or the XF command of Utility Set 2. (These commands are described in the Program Development Tools manual and the Utility Programs manual, respectively.)

If the source module to be processed by the Macro Preprocessor includes macro calls that request library-resident macro routines, before loading the Macro Preprocessor you must specify in an AT 06 command to the Command Processor the volume name of the diskette that contains macro libraries. (See "Input to Command Processor Before Macro Preprocessor is Loaded" in Section 7 of the Program Development Tools manual.) A LIBM statement must be included in the source module for each macro library that contains macro routines that will be requested in that module.

LIBM Macro Control Statement

The LIBM statement specifies the name of a macro library and indicates whether all or only specified macro routines in that library will be made available so that they can be requested in subsequent macro calls. If applicable, you must specify LIBM statement(s) at the beginning of the source module.

Format:

LIBMΔlibrary [,macro-name] ...

library

Name of the macro library that contains macro routine(s).

macro-name

Name(s) of macro routine(s) in the macro library that may be requested in macro call(s); must be a valid symbolic name. The names must be different from the names of inline macro routines. If the same name is specified, the inline macro routine is used.

Default: All macro routines in the specified macro library may be included in the expanded source module by subsequent macro calls.

MACRO CALLS

A macro call is a statement that causes a specified macro routine to be included in the source module and optionally assigns or reassigns values to parameters in that macro routine. The macro routine is included in the expanded source module at the location of the macro call.

If a parameter is assigned a value *only* in the macro call or in *both* the macro call and the MAC statement, the value in the macro call is used. If a parameter is not assigned a value in the macro call but it was assigned a value in the MAC statement, that value is used. If it was not assigned a value in either location, its default value is a null ASCII character string.

If no parameter values are included in a macro call, the macro-name must be the last entry on the source line, or it must be immediately followed by a comma and an optional comment.

Format:

[label]Δmacro-name$\left[\Delta P_1\text{-value}\left[, [P_2\text{-value}]\right] \ldots\right]$

label

Symbolic name that identifies the macro call.

macro-name

Name of the macro routine to be included in the expanded source module; this name must correspond to a name designated in a MAC macro control statement.

P_n-value

Value of macro parameter; can be any alphanumeric characters.

NOTE: To include a comma, space, or horizontal tab as part of a parameter value, specify that character within apostrophes.

In a macro call, parameters are positional; i.e., their values must be specified so that they correspond to parameters P1 to P9 and PA to PZ. A comma must be specified for each parameter whose value is not specified. All parameters beyond the last specified parameter's value are considered to be omitted.

NOTE: It may be impossible to specify all parameter values on one source line. Parameter values can be continued on the next line by replacing the last comma with a semicolon. (See "Assembly Language Source Statement Formats" in Section 3.)

Example:

This example illustrates an unexpanded source module in which parameters are assigned values only in a MAC statement, only in a macro call, and in both a MAC statement and a macro call. The resulting expanded source module illustrates the inclusion of the macro routine and the appropriate parameter values.

Unexpanded source module:

```
       TITLE MCL

SAMPLE MAC P3=1,P5=8   Designates beginning of macro routine
                       and assigns values to parameters P3
                       and P5
       DC ?P3,?P2

     * NEWB?P5

FINI   ENDM            Designates end of macro routine
         :
NUVAL  SAMPLE ,2,,,5','5  Macro call that assigns value to
                          parameter P2, and assigns different
                          value to parameter P5; i.e., P2
                          equals 2, and P5 equals 5','5
         :
```

Expanded source module:

```
  TITLE MCL
    :
  DC 1,2         First parameter value was assigned in
                 MAC statement; second parameter value
                 was assigned in macro call

* NEWB5','5      Since different values were assigned to
                 P5 in the MAC statement and in the macro
                 call, the value in the macro call is
                 used
    :
```

Nested Macro Call

A nested macro call is a macro call that occurs within a macro routine. Whenever a nested macro call is encountered, processing of the current macro routine stops; i.e., all of its macro parameters are saved, and the nested macro call is processed. The nested macro call has its own macro parameters. After the nested macro call is processed, processing of the previous macro routine resumes at the point of termination.

Macro calls may be nested to as many levels as memory permits. Each level consists of one macro routine that calls another. For example, if macro routine A contains a macro call to macro routine B, one level of nesting exists. If macro routine B contains a macro call to macro routine C, two levels of nesting exist.

Example:

This example illustrates an unexpanded source module that contains a nested macro call and the resulting expanded source module.

Unexpanded Source Module

```
       TITLE NSTD
MACRO1 MAC
       NEWA
       NEWB
       ENDM
MACRO2 MAC
       NEWX
       NEWY
       MACRO1          Nested macro call
       NEWZ
       ENDM
       .
       .
       .
       MACRO2          Macro call
       .
       .
       .
```

Expanded Source Module

```
       TITLE NSTD
       .
       .
       .
       NEWX
       NEWY
       NEWA }
       NEWB }          Contents of nested macro call
       NEWZ
       .
       .
       .
```

Recursive Macro Calls

A recursive macro call is a nested macro call that calls either the routine within which the call is located or another routine in the nest that eventually calls the original routine. A recursive macro call must be designed to reach its ENDM statement exactly once per call to it. An example of a recursive call is the case in which a macro routine processes parameter 1 and then if parameter 2 is present, calls itself with ?P2 for parameter 1, ?P3 for parameter 2, etc.; that is, each parameter has been shifted one position left. A recursive macro call is processed the same as any other nested macro call. The depth of recursion is limited only by the amount of memory available to the Macro Preprocessor.

CONTROLLING EXPANSIONS

When a macro call requests a given macro routine, it need not always result in the same expansion. Values in that routine may vary, and the statements to be included in the source module may vary. This flexibility is accomplished by including macro variables and conditional macro control statements in the macro routine.

Macro Variables

There are two types of macro variables: local and global. A local variable can be assigned a value only in the macro routine in which it is referenced. A global variable can be assigned a value anywhere in the source module; e.g., in the macro routine in which it is referenced, in any other macro routine in the source module, or in statements that constitute the body of the source module.

Variables have fixed names; only their values can be altered. Global variables are named G1 to G9 and GA to GZ. Local variables are named L1 to L9 and LA to LZ. To designate in a macro routine that substitution will occur, precede each variable name with a substitution operator (question mark); e.g., ?G1. When the macro routine is processed, the Macro Preprocessor will replace each reference to a variable with its value.

A variable can be assigned an alphanumeric or numeric value by specifying the SETA or SETN macro control statement, respectively.

SETA Macro Control Statement

The set alphanumeric macro control (SETA) statement assigns an alphanumeric value to a local or global macro variable. If you assign a value to a variable and then redefine the variable in a subsequent SETA or SETN statement, the last value specified is used.[2]

When assigning a value to a *global* macro variable, you can specify SETA anywhere within the source module. When assigning a value to a *local* macro variable, you must specify SETA in the macro routine in which the variable is referenced.

Format:

```
    variable△SETA value
```

variable

Name of the local or global macro variable that is being assigned a value; must be L1 to L9, LA to LZ, G1 to G9, or GA to GZ.

value

Must be alphanumeric. (See "Designating Alphanumeric Values" at the end of this section.)

Example:

This example illustrates an unexpanded source module in which macro variables are assigned values in SETA statements. The resulting expanded source module includes those macro variable values.

[2]When a nested macro call is encountered, values of local variables, and parameters in the current macro routine are saved and are still applicable after the nested macro call is processed.

Unexpanded source module:

```
       TITLE VALUE

SAMPLE MAC               Designates beginning of macro routine

    L4 SETA DE           Assigns value to L4
    L5 SETA BC           Assigns value to L5
    L4 SETA 'A'?L5,['2'] Assigns different value to L4
       DC ?L4
       :
       ENDM              Designates end of macro routine

       SAMPLE            Macro call
       :
```

Expanded source module:

```
       TITLE VALUE

       DC ABC,'2'        Last value specified for L4 is used;
                         apostrophes included only if they
                         were enclosed within protection
                         operators
       :
       :
```

Apostrophes Within SETA Statements

The operand of the SETA statement begins at the first character after the operation code that is neither a blank, horizontal tab, nor] (close protection) character. The operand terminates at the end of the statement or at the first blank or horizontal tab not within apostrophes after the beginning of the operand.

Unprotected apostrophes within the operand of the SETA statement are not considered part of the variable's value and are removed from the operand before the operand's value is assigned to the macro variable. For example:

G6ΔSETAΔXYZ'Δ'123	assigns the value XYZΔ123 to the global variable G6
GTΔSETAΔXYZ['Δ']123	assigns the value XYZ['Δ'] 123 to the global variable GT

SETN Macro Control Statement

The set numeric macro control (SETN) statement assigns a numeric value to a local or global macro variable. If you assign a value to a variable and then redefine the variable in a subsequent SETN or SETA statement, the last value specified is used.[3]

When assigning a value to a *global* macro variable, you can specify SETN anywhere within the source module. When assigning a value to a *local* macro variable, you must specify SETN in the macro routine in which the variable is referenced.

[3] When a nested macro call is encountered, values of local variables and parameters in the current macro routine are saved and are still applicable after the nested macro call is processed.

Format:

variable∆SETN value

variable

Name of the local or global macro variable that is being assigned a numeric value; must be L1 to L9, LA to LZ, G1 to G9, or GA to GZ.

value

Must be numeric. (See "Designating Numeric Values" at the end of this section.)

The operand of the SETN statement begins at the first character after the operation code that is neither a blank nor a horizontal table. The operand terminates at the end of the statement or at the first blank or horizontal tab not within apostrophes after the beginning of the operand. For example:

GL∆SETN∆22+8	assigns the value 30 to the global variable GL.
GA∆SETN∆6+'∆0'	assigns the value ∆6 (2036 in hexadecimal) to the global variable GA.

Example:

This example illustrates an unexpanded source module in which macro variables are assigned values in SETN statements. The resulting expanded source module includes those macro variable values.

Unexpanded source module:

```
       TITLE EXMPL
SAMPLE MAC                Designates beginning of macro routine
    L5 SETN 3             Assigns value to L5
    L6 SETN 2*(?L5*?G2)+1 Assigns value to L6
       DC ?L6
       :
  FINI ENDM               Designates end of macro routine
    G2 SETN 2             Assigns value to G2
       SAMPLE             Macro call
       :
```

Expanded source module:

```
       TITLE EXMPL
       DC 13
       :
       :
```

Conditional Macro Control Statements

Conditional macro routine that covers many situations. Depending on the conditions at a given time, only certain statements are processed.

Conditional macro control statements are listed and described below:

- o FAIL
- o GOTO
- o IF
- o NULL

FAIL Macro Control Statement

The FAIL statement is used to ensure that conditions are logically consistent; it does not affect expansions. The Macro Preprocessor issues a Z error flag for each FAIL statement.

Format:

[label]ΔFAIL

label

Symbolic name that identifies FAIL statement.

NOTE:: If an assembly control FAIL statement is desired within a macro routine, it must be protected.

GOTO Macro Control Statement

The GOTO statement causes the Macro Preprocessor to stop processing the macro routine or to resume processing at a specified statement. The statement at which processing will resume can be in any location within the macro routine; i.e., it need not be subsequent to the GOTO statement.

Format:

$$\text{[label]}\Delta\text{GOTO}\Delta \begin{Bmatrix} * \\ \text{skip-label} \end{Bmatrix}$$

label

Symbolic name that identifies the GOTO statement.

*

Causes Macro Preprocessor to stop processing the macro routine; i.e., the current line is considered an ENDM macro control statement. Processing resumes at the statement that follows the current macro call.

skip-label

Symbolic name of statement at which Macro Preprocessor should resume processing.

IF Macro Control Statement

The IF statement causes the Macro Preprocessor to evaluate characters in either one or two operands to determine if a specified condition exists. If the condition exists, the Macro Preprocessor stops processing the macro routine or resumes processing at a specified statement that is subsequent to the IF statement. If the condition does *not* exist, the next sequential instruction is processed.

Formats:

Evaluating characters in one operand:

$$[\text{label}]\Delta\text{IF}\left\{\begin{matrix}[\text{N}]\left\{\begin{matrix}\text{P}\\ \text{N}\\ \text{Z}\end{matrix}\right\}\\ \text{OD}\\ \text{EV}\end{matrix}\right\}\Delta\text{operand},\left\{\begin{matrix}*\\ \text{skip-label}\end{matrix}\right\}$$

label
: Symbolic name that identifies the IF statement.

[N]P
: (Not) positive.

[N]N
: (Not) negative.

[N]Z
: (Not) zero.

OD
: Odd.

EV
: Even.

operand
: Character(s) being evaluated; must be numeric. (See "Designating Numeric Values" at the end of this section.)

*
: If condition in IF statement is true, causes Macro Preprocessor to stop processing macro routine; i.e., the current line is considered an ENDM macro control statement. Processing resumes at the statement that follows the current macro call.

skip-label
: If condition in IF statement is true, designates symbolic name of statement at which Macro Preprocessor should resume processing.

Comparing characters in two operands:

$$[\text{label}]\Delta\text{IF }[\text{N}]\left\{\begin{matrix}\text{G}\\ \text{L}\\ \text{E}\end{matrix}\right\}\Delta\text{operand}_1,\text{operand}_2,\left\{\begin{matrix}*\\ \text{skip-label}\end{matrix}\right\}$$

label
: Symbolic name that identifies the IF statement.

[N]G
: (Not) greater than.

[N]L
: (Not) less than.

[N]E
: (Not) equal to.

operand$_1$ operand$_2$

Character strings being compared; must be alphanumeric. (See "Designating Alphanumeric Values" at the end of this section.)

Starting with the leftmost character, the Macro Preprocessor compares each character in operand$_1$ to the character in the corresponding position in operand$_2$. The characters are compared until either a pair of unequal characters is encountered, or all of the characters have been compared. If the operands are different lengths, the rightmost characters of the shorter operand are considered to be ASCII blanks. (Table 2-2 describes the hexadecimal values of ASCII characters.)

*

If condition in IF statement is true, causes Macro Preprocessor to stop processing macro routine; i.e., the current line is considered an ENDM macro control statement. Processing resumes at the statement that follows the current macro call.

skip-label

If condition in IF statement is true, designates symbolic name of statement that Macro Preprocessor should process next.

NOTE: If an assembly control IF statement is desired within a macro routine, it must be protected.

Example 1 – Evaluating characters in one operand:

Unexpanded Source Module:

```
      TITLE CONDL
  BGN MAC
      IFOD 1,TAG1
     [FAIL]
 TAG1 DC 1
      :
      IFOD 2,TAG1
     [FAIL]
 TAG1 DC1
      :
 FINI ENDM
      :
      BGN
      :
```

Expanded Source Module:

```
      TITLE CONDL
      :
 TAG1 DC 1
      :
      FAIL
```

```
TAG1 DC 1
      .
      .
      .
      .
      .
```

Example 2—Comparing characters in two operands:

Unexpanded Source Module:

```
          TITLE TWO
     INCL MAC
          .
          .
          IFE AB,AB,TAG1
         [FAIL]
     TAG1 DC 1
          .
          .
          IFE AB,CD,TAG1
         [FAIL]
     TAG1 DC 1
          .
          .
     FINI ENDM
          .
          .
          INCL
          .
          .
          .
```

Expanded Source Module:

```
          TITLE TWO
          .
          .
          .
          .
          .
     TAG1 DC 1
          .
          .
          FAIL
     TAG1 DC 1
          .
          .
          .
          .
          .
```

NULL Macro Control Statement

The NULL statement has no effect on the processing of macro routines. Processing continues with the next sequential instruction.

This statement is often used to define a label referenced by an IF or GOTO statement.

Format:

[label]ΔNULL

label
  Name of the label being defined.

NOTE: If an assembly control NULL statement is desired within a macro routine, it must be protected.

MACRO FUNCTIONS

Macro functions have the following capabilities:

- o Determine number of characters that are in a specified character string (AL function)
- o Convert a numeric value to its hexadecimal equivalent (CH function)
- o Search a character string for an embedded character string (IX function)
- o Determine which character within a character string is the first character that is the first character of another character string (SR function)
- o Specify which characters within a character string should be included in the source statement (SS function)
- o Permit parameters and variables to be referenced by their positions (V function)
- o Determine which character within a character string is the first character that is *not* in another character string (VR function)

Macro functions can be specified in any location(s) of statements in macro routines. Within one statement there can be multiple macro functions; these functions can be nested. Nested macro functions are processed from the innermost function to the outermost function.

Format of Macro Functions

Macro functions are described alphabetically on the subsequent pages. As indicated in their formats, each function is preceded by a substitution operator (question mark) and its arguments are enclosed within one set of parentheses. Most functions require that you specify either a numeric or an alphanumeric value. Methods of specifying these values are described at the end of this section under "Designating Numeric Values" and "Designating Alphanumeric Values."

Macro functions require one, two, or (optionally) three arguments.

First Argument

The first argument of a macro function begins with the first character following the open parenthesis, (, after the function name. For a macro function which accepts only a single argument, the argument is terminated with the first unprotected close parenthesis,) , not enclosed within apostrophes. For a macro function which accepts more than one argument, the first argument is terminated by the first unprotected comma not enclosed within apostrophes.

Examples:

?AL(ΔXY(5)Δ)	The argument to the AL function is ΔXY(5)Δ.
?SS(ABΔC',5',4)	The first argument to the SS function is ABΔC',5'.

Middle Argument

The middle argument of a macro function begins with the first character following the comma which terminated the previous argument and ends with the first unprotected comma not enclosed within apostrophes.

Example:

?SS(AEIOU(Y),(12-2)/5,3)	The middle argument to the SS function is (12-2)/5.

Last Argument

The last argument of a macro function begins with the first character following the comma which terminated the previous argument. The last argument is terminated by the first unprotected close parenthesis not within apostrophes following the macro function name.

Examples:

?IX(LMΔ')'P,Δ')'P)	The last argument to the IX function is Δ')'P.
?SS(A1B2C3D4,(1+3)/2)	The last argument to the SS function is (1+3)/2.
?SS(AB[C,D],[C)])	The last argument to the SS function is [C)].

Length Attribute Macro Function

The length attribute (AL) function causes the Macro Preprocessor to designate the number of characters that are in a specified character string. If a null ASCII character string is specified, the Macro Preprocessor returns a zero.

Format:

```
?AL(arg)
```

arg

Character string whose length is to be determined; must be alphanumeric.

Example:

```
?AL(?L5+?P5)
```

If variable L5 equals 2AB, and parameter P5 equals 5B, the above function will be replaced with 6.

Hexadecimal Conversion Macro Function

The hexadecimal conversion (CH) function converts a numeric integer constant to its hexadecimal equivalent.

Format:

?CH(arg$_1$,arg$_2$)

arg$_1$

Numeric value to be converted to hexadecimal.

arg$_2$

Numeric value that specifies the *format* of the hexadecimal representation, as described below:

Value of arg$_2$	*Meaning*
Not Specified	Hexadecimal integer constant with no insignificant zeros.
0	Value in arg$_1$ is converted to an unsigned hexadecimal integer. The value returned is the ASCII representation of the significant digits of the unsigned hexadecimal integer.

>0	Hexadecimal integer constant. The value of arg_2 designates the number of character positions; can be 1 to 4.
<0	The value specified in arg_2 designates the number of character positions; can be -1 to -4. The value in arg_1 is converted to an unsigned hexadecimal integer. The value returned is the ASCII representation of the specified number of characters.

Examples:

Condition	*Function Specified*	*Result*
arg_2 not specified	?CH(10)	X'A'
arg_2 =0	?CH(10,0)	A
arg_2 >0	?CH(10,1)	X'A'
arg_2	?CH(10,2)	X'0A'
	?CH(10,3)	X'00A'
	?CH(10,4)	X'000A'
arg_2 <0	?CH(10,-1)	A
	?CH(10,-2)	0A
	?CH(10,-3)	0A
	?CH(10,-4)	000A

Index Macro Function

The index (IX) function causes the Macro Preprocessor to search a specified character string for the occurrence of an embedded character string.

Format:

```
?IX(arg1,arg2)
```

arg_1
Character string being searched; must be alphanumeric.

arg_2
Embedded character string for which the Macro Preprocessor will search; must be alphanumeric.

The value returned specifies the character position within arg_1 of the first (leftmost) character of the embedded character string. If arg_2 is not contained within arg_1 or arg_2 is a null ASCII character string (e.g., ''), a zero is returned.

Example:

```
?IX(ABCDE5,CDE5)
```

The above statement causes the Macro Preprocessor to search ABCDE5 for the character string CDE5. Since the embedded character string starts in the third character position of ABCDE5, the Macro Preprocessor replaces the index function with a 3.

Search Macro Function

The search (SR) function causes the Macro Preprocessor to determine which character of a specified character string is the first (leftmost) character that is also included in another specified character string.

Format:

?SR(arg_1,arg_2)

arg_1

Character string that contains character(s) for which the Macro Preprocessor will look; must be alphanumeric.

arg_2

Character string that will be searched in order to locate a certain character; must be alphanumeric.

The Macro Preprocessor includes in the source module the character position of the leftmost character in arg_1 that is also in arg_2.

If arg_1 or arg_2 is a null ASCII character string, or if no characters in arg_1 are also in arg_2, zero is returned.

Example 1:

```
?SR(CHARSUBSTRING,STRING)
```

The above macro function causes the Macro Preprocessor to determine the leftmost character of CHARSUBSTRING that is also in STRING. Since the character R is the leftmost character of CHARSUBSTRING that is also in STRING and it is in the fourth character position of CHARSUBSTRING, the macro function is replaced with 4.

Example 2:

```
?SR(FAB2,'BCA1')
```

The above macro function causes the Macro Preprocessor to determine the leftmost character of FAB2 that is also in 'BCA1'. Since A is the leftmost character in FAB2 that is also in 'BCA1', and it is in the second character position of FAB2, the macro function is replaced with 2.

Example 3:

```
?SR(BA3,?L1)
```

The above macro function causes the Macro Preprocessor to determine the leftmost character of BA3 that is also in local variable 1. If L1 equals 23A, A is the first character that is also in L1. Since A is in the second character position of BA3, the Macro Preprocessor includes 2 in the source statement.

Substring Macro Function

The substring (SS) function causes the Macro Preprocessor to include in the source statement a specified number of characters of a specified character string, beginning with the character that is in a specified character position.

Format:

?SS(arg_1, arg_2 [,arg_3])

arg_1

Character string that contains the characters to be included in the source statement; must be alphanumeric.

arg_2

Character position of the first character in arg_1 that is to be included; must be numeric.

arg_3

Number of characters to be included.

Default: The character whose character position was specified in arg_2 ,and all subsequent characters of arg_1.

If arg_1 is a null ASCII character string, arg_3 is $\leqslant 0$, or the value specified in arg_2 is greater than the length of arg_1, a null ASCII character string is included in the source statement.

Example 1:

```
?SS(?P2,?L5,3)
```

If P2=ABCDE and L5=2, the above function designates that the source statement include *three* characters of ABCDE, starting with the character in the second character position. BCD would be included.

Example 2:

```
?SS(?P2,?L5)
```

If P2=ABCDE and L5=2, the above function designates that the source statement include *all* characters of ABCDE, starting with the character in the second character position. BCDE would be included.

Example 3:

```
G6   SETA  ?SS(ABΔC',5',4) yields
G6   SETA  C',5' , which leaves C,5 in G6.
```

Vector Orientation Macro Function

The vector orientation (V) function permits macro parameters and macro variables to be referenced by their positions rather than by their names.

Format:

$$?V\begin{Bmatrix} P \\ L \\ G \end{Bmatrix}(\text{arg})$$

P

Parameter

L
   Local variable.

G
   Global variable.

arg
   Numeric value that identifies a parameter or variable; must be from 1 to 35.

Example:

```
?SS(?VP(10),2,3)
```

The above function illustrates usage of the vector orientation function within a substring (SS) function. The function ?VP(10) identifies parameter PA. If PA= ABCDE, the above substring function is replaced with BCD.

Verify Macro Function

The verify (VR) function causes the Macro Preprocessor to specify which character in a specified character string is the first character that is *not* in another specified character string.

Format:

?VR(arg_1,arg_2)

arg_1
   Character string that will be searched; must be alphanumeric.

arg_2
   Character string that contains the characters for which the Macro Preprocessor is going to look; must be alphanumeric.

The Macro Preprocessor designates the character position of the leftmost character in arg_1 that is not found in arg_2. If arg_1 is a null ASCII character string, or if every character in arg_1 occurs in arg_2, zero is designated.

Example 1:

```
?VR(STRINGSUBSTRING,STRINGCHARSTRING)
```

The above macro function causes the Macro Preprocessor to specify the leftmost character in STRINGSUBSTRING that is *not* in STRINGCHARSTRING. Since U is the leftmost character in STRINGSUBSTRING that is not in STRINGCHARSTRING and it is in the eighth character position of STRINGSUBSTRING, the Macro Preprocessor replaces the function with 8.

Example 2:

```
?VR(?P3,?G5)
```

If parameter P3 has a value of ABC3D, and global variable G5 has a value of AD3, the first character of P3 that is not in G5 is B, the second character of P3. Therefore, the Macro Preprocessor replaces the function with a 2.

EXAMPLE ILLUSTRATING MACRO FACILITY

Figure 6-1 illustrates a sample unexpanded source module and an Assembler listing of the resulting expanded source module.

```
          TITLE     USEMAC
*
*INCLUDE IN-LINE MACRO ROUTINES.
*
POLY      MAC
*THIS MACRO GENERATES CODE TO COMPUTE
*Y=X**N + X**(N-1) + ... + X + 1.
*X IS DESIGNATED BY PARAMETER 1.
*Y IS DESIGNATED BY PARAMETER 2.
*N IS DESIGNATED BY PARAMETER 3.
*
[*]
          LDV       $R1,1
G2        SETN      ?P3       NUMBER OF FACTORS.
TESTN     IFZ       ?G2,STOREX          COMPLETE?
[*]
          FACTOR    ?P1       NO...NESTED CALL FOR ANOTHER FACTOR.
[*]
G2        SETN      ?G2-1     DECREASE FACTOR COUNTER.
          GOTO      TESTN
STOREX    STR       $R1,?P2   STORE POLYNOMIAL VALUE.
          ENDM
*
*
FACTOR    MAC
*
*THIS MACRO GENERATES CODE WHICH MULTIPLIES ($R1) BY THE
*CONTENTS OF THE LOCATION DESIGNATED BY PARAMETER 1, AND
*ADDS 1 TO THE PRODUCT.
*
          MUL       $R1,?P1
          ADV       $R1,1
          ENDM
*
*
MOVER     MAC       P4=0
*
*THIS MACRO GENERATES CODE WHICH PERFORMS A "MEMORY TO MEMORY"
*MOVE OF DATA.
*IF PARAMETER 4 IS NON-ZERO, THE CODE WILL MOVE BYTES.
*IF PARAMETER 4 IS ZERO, THE CODE WILL MOVE WORDS. (DEFAULT OPTION).
*PARAMETER 1 SPECIFIES THE SOURCE ADDRESS.
*PARAMETER 2 SPECIFIES THE DESTINATION ADDRESS.
*PARAMETER 3 SPECIFIES THE NUMBER OF UNITS (BYTES OR WORDS) TO MOVE.
*
          IFNZ      ?P4,BYTMOV          BYTES OR WORDS?
*
*MOVE WORDS...
LL        SETA      LDR       USE LOCAL VARIABLES TO DEFINE DESIRED OPCODES.
LS        SETA      STR
          GOTO      SAME
*
*MOVE BYTES...
BYTMOV    NULL
LL        SETA      LDH
LS        SETA      STH
*
SAME      NULL
[*]
[*]
*USE VECTOR FUNCTION TO SUBSTITUTE PARAMETERS.
          LAB       $B1,?VP(1)
          LAB       $B2,?VP(2)
          CL        =$R1
*NEXT STATEMENT WILL HAVE A UNIQUE LABEL.
NXT?L1    ?LL       $R3,$B1.$R1
          ?LS       $R3,$B2.+$R1
*GET UNIT COUNT AS A HEX INTEGER.
          CMR       $R1,=?CH(?VP(3))
```

Figure 6-1. Sample Unexpanded Source Module and Assembler Listing of Resulting Expanded Source Module

```
*USE DEFAULT VALUE OF GLOBAL VARIABLE, G1.
         BE        >?G1+2
         B         >NXT?L1
*THE FOLLOWING NULL IS FOR THE ASSEMBLER.
         (NULL)
[*]
[*]
         ENDM
*
*
*MAKE USE OF THE IN-LINE MACRO DEFINITIONS DEFINED ABOVE.
*
RELZRO   LDV       $R1,2
         STR       $R1,X
         POLY      X,Y,5     COMPUTE Y=X**5+X**4+X**3+X**2+X+1, FOR X=2.
*
         MOVER     A,B,11,1  MOVE 11 BYTES FROM A TO B.
*
         HLT
X        RESV      1
Y        RESV      1
A        RESV      20
B        RESV      20
         END       USEMAC,RELZRO
EOF
```

::
::ASSEMBLER LISTING OF RESULTING EXPANDED SOURCE MODULE

```
USEMAC               L6 ASSEMBLER-0110                    PAGE 0001

    000001                                 TITLE     USEMAC
    000002                        *
    000003                        *INCLUDE IN-LINE MACRO ROUTINES.
    000004                        *
    000005                        *
    000006                        *
    000007                        *
    000008                        *
    000009                        *
    000010                        *
    000011                        *MAKE USE OF THE IN-LINE MACRO DEFINITIONS DEFINED ABOVE.
    000012                        *
    000013  0000  1C02            RELZRO   LDV       $R1,2
    000014  0001  9F40 001F                STR       $R1,X
    000015                        *
    000016  0003  1C01                     LDV       $R1,1
    000017                        *
    000018  0004  9B40 001C                MUL       $R1,X
    000019  0006  1E01                     ADV       $R1,1
    000020                        *
    000021                        *
    000022  0007  9B40 0019                MUL       $R1,X
    000023  0009  1E01                     ADV       $R1,1
    000024                        *
    000025                        *
    000026  000A  9B40 0016                MUL       $R1,X
    000027  000C  1E01                     ADV       $R1,1
    000028                        *
    000029                        *
    000030  000D  9B40 0013                MUL       $R1,X
    000031  000F  1E01                     ADV       $R1,1
    000032                        *
    000033                        *
    000034  0010  9B40 0010                MUL       $R1,X
    000035  0012  1E01                     ADV       $R1,1
    000036                        *
    000037  0013  9F40 000E       STOREX   STR       $R1,Y     STORE POLYNOMIAL VALUE.
    000038                        *
    000039                        *
    000040                        *
    000041  0015  9BC0 000D                LAB       $B1,A
    000042  0017  ABC0 001F                LAB       $B2,B
    000043  0019  8751                     CL        =$R1
    000044  001A  B091            NXT007   LDH       $R3,$B1.$R1
    000045  001B  B7DE                     STH       $R3,$B2.+$R1
```

Figure 6-1 (cont). Sample Expanded Source Module and Assembler Listing of Resulting Expanded Source Module

```
   000046  001C  9970 000B                      CMR       $R1,=X'B'
   000047  001E  0902                           BE        >$+2
   000048  001F  0FFB                           B         >NXT007
   000049                                       NULL
   000050                          *
   000051                          *
   000052                          *
   000053  0020  0000                           HLT
   000054  0021                    X            RESV      1
   000055  0022                    Y            RESV      1
   000056  0023                    A            RESV      20
   000057  0037                    B            RESV      20
   000058  004B  0000                           END       USEMAC,RELZRO
 0000 ERR COUNT
```

Figure 6-1 (cont). Sample Expanded Source Module and Assembler Listing of Resulting Expanded Source Module

PROGRAMMING CONSIDERATIONS

1. In an unexpanded source module, each macro control statement and each other type of statement that contains error flag(s) can comprise up to 74 characters. Each other line can comprise up to 80 characters. Subsequent characters are truncated.
2. Input to the Macro Preprocessor may be either uppercase or lowercase characters. All lowercase characters in ASCII, hexadecimal, and bit string constants, and in hexadecimal integer constants remain lowercase characters; all other lowercase characters within the source module are converted to uppercase.
3. When an expanded source module is assembled, the Assembler issues an error flag for each statement that contains a null ASCII character string.
4. If insufficient memory exists, memory can be conserved by:
   a. Assigning some or all macro routines to macro libraries.
   b. Limiting the level of nested macro calls.
   c. Limiting the size of macro parameter and variable values.
   d. Reducing attach table size.
   e. Including the OA argument in the load command to the Command Processor. (See the Program Development Tools manual.)
   f. Specifying in LIBM macro control statement only those macro routines that will be requested in subsequent macro calls.

Initialized Values of Macro Variables

Each local macro variable is initialized to be a null ASCII character string, except for the following:

L1

Unique 3-character string. Each time there is a macro call, the value of L1 is incremented by 1; can be from 001 to ZZZ. This variable permits a statement in a macro routine to have a unique label each time the routine is requested in a macro call; e.g., if the label of a statement is SMP?L1, the label would be SMP001 the first time the routine is requested, and SMP002 the second time the routine is requested.

L2

Numeric value that designates the level of nesting in the current macro call. If the macro call does not include a nested macro call, L2 equals 0.

L3

Numeric value that designates the number of the last parameter that was assigned a value in the current macro call. If the macro call does not include any parrmetrers, L3 equals 0.

L4

Label of the current macro call. If no label is specified, L4 equals a null ASCII character string.

Global macro variable G1 is initialized to equal $. Each other global variable is a null ASCII character string. These values remain in effect unless they are reassigned in SETA or SETN macro control statements.

Designating Numeric Values

When an operand or argument requires a numeric value, the value must be from -32768 to +32767. (See "Truncation/Padding of String Constants" in Section 2 to determine how characters are truncated, if necessary.) A numeric value can be specified as follows:

- o Decimal integer constant (e.g., 31764, +4652)
- o Hexadecimal integer constant (e.g., +X'2F' , X'7000')
- o Substitution operator followed by macro variable name whose contents are the source language representation of a decimal or hexadecimal integer constant (e.g., ?G3, ?L4)
- o Substitution operator followed by macro parameter name whose contents are the source language representation of a decimal or hexadecimal integer constant (e.g., ?P2)
- o Substitution operator followed by a macro function that returns a numeric value
- o Expression that combines any of the above character strings by including arithmetic operators (e.g., 31764+(?G3))

Designating Alphanumeric Values

When an operand or argument requires an alphanumeric value, you can specify any type of alphanumeric character string, including the following:

- o Substitution operator followed by macro variable name
- o Substitution operator followed by macro parameter name
- o Substitition operator followed by macro function
- o Expression that combines any of the above character strings by specifying them adjacent to each other

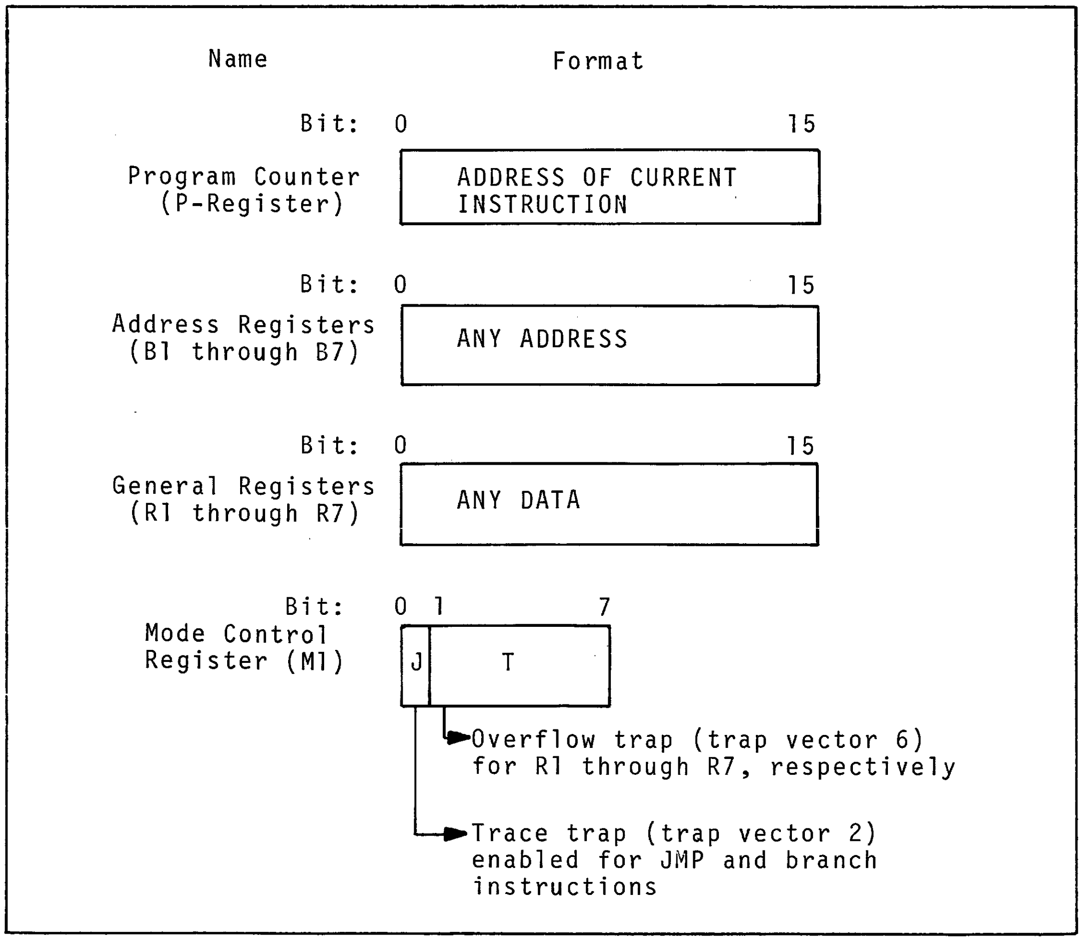
APPENDIX A

PROGRAMMER'S REFERENCE INFORMATION

This appendix provides, in a summarized form, information about the internal representation of the assembly language instructions, the operations they perform, and other useful data for coding and debugging your program.

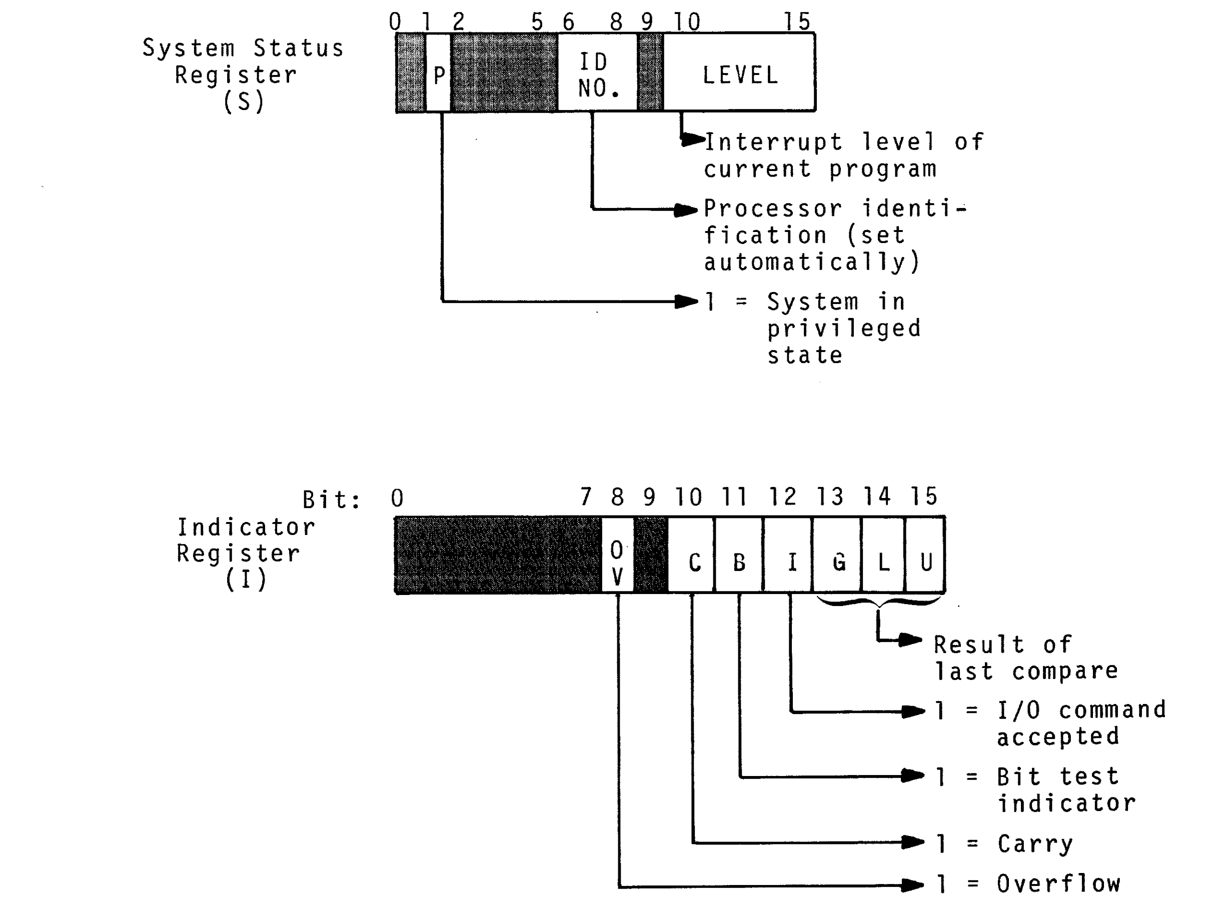
SUMMARY OF HARDWARE REGISTERS

Figure A-1 is a list of Level 6 registers and their formats. The length of each register is shown in bits.



Name
Format
Bit: 0 15
Program Counter (P-Register)
ADDRESS OF CURRENT INSTRUCTION
Bit: 0 15
Address Registers (B1 through B7)
ANY ADDRESS
Bit: 0 15
General Registers (R1 through R7)
ANY DATA
Bit: 0 1 7
Mode Control Register (M1)
J
T
Overflow trap (trap vector 6) for R1 through R7, respectively
Trace trap (trap vector 2) enabled for JMP and branch instructions

Figure A-1. Level 6 Hardware Registers


System Status Register (S)
0 1 2 5 6 8 9 10 15
P
ID NO.
LEVEL
Interrupt level of current program
Processor identification (set automatically)
1 = System in privileged state
Bit: 0 7 8 9 10 11 12 13 14 15
Indicator Register (I)
OV
C
B
I
G
L
U
Result of last compare
1 = I/O command accepted
1 = Bit test indicator
1 = Carry
1 = Overflow

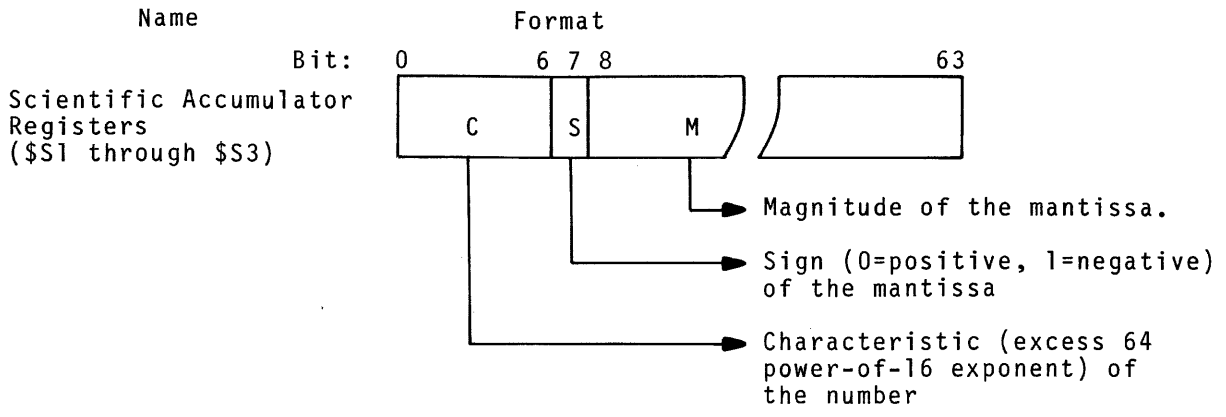
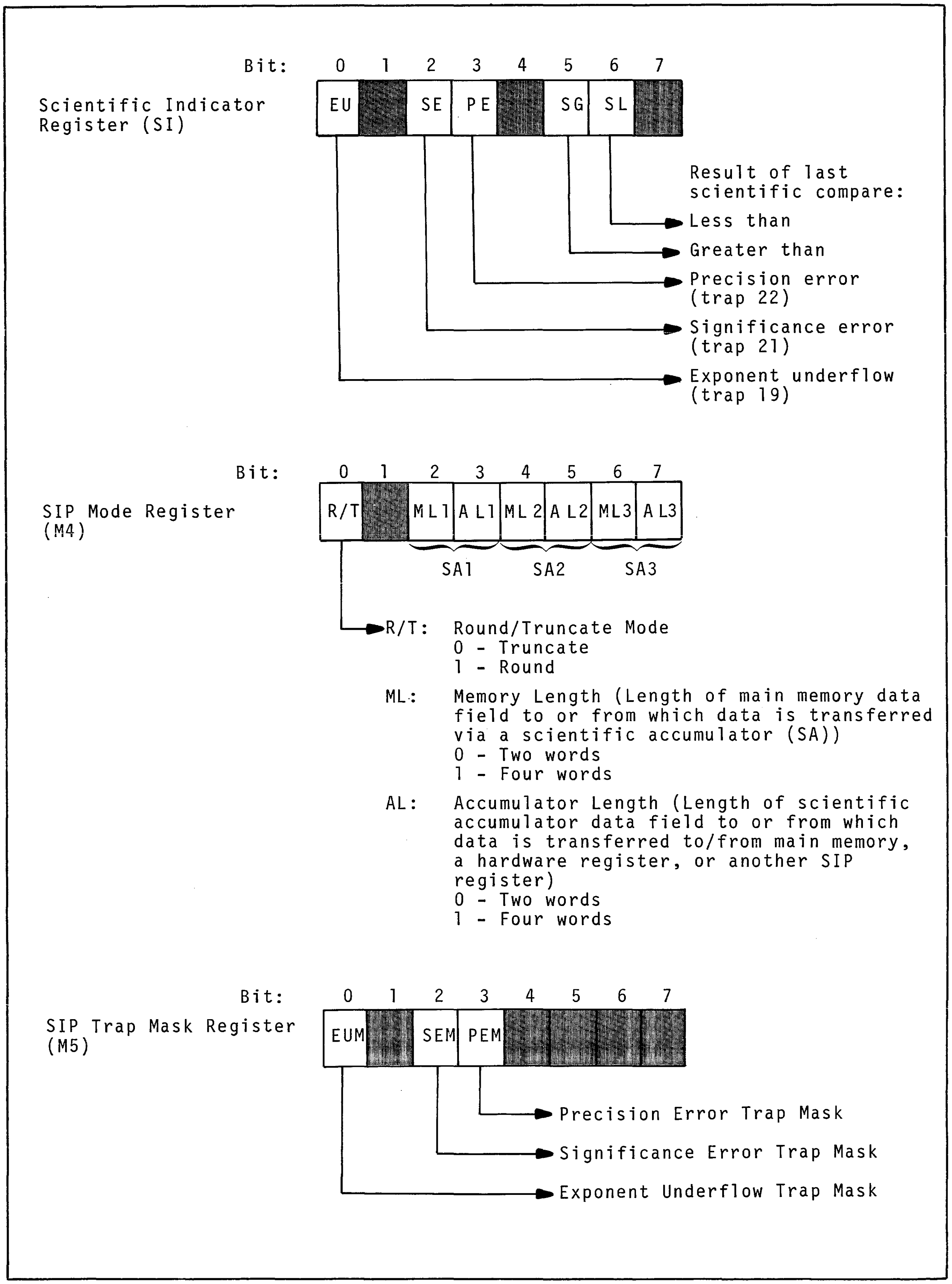

Name
Format
Bit: 0 6 7 8 63
Scientific Accumulator Registers ($S1 through $S3)
C
S
M
Magnitude of the mantissa.
Sign (0=positive, 1=negative) of the mantissa
Characteristic (excess 64 power-of-16 exponent) of the number

Figure A-1 (cont). Level 6 Hardware Registers

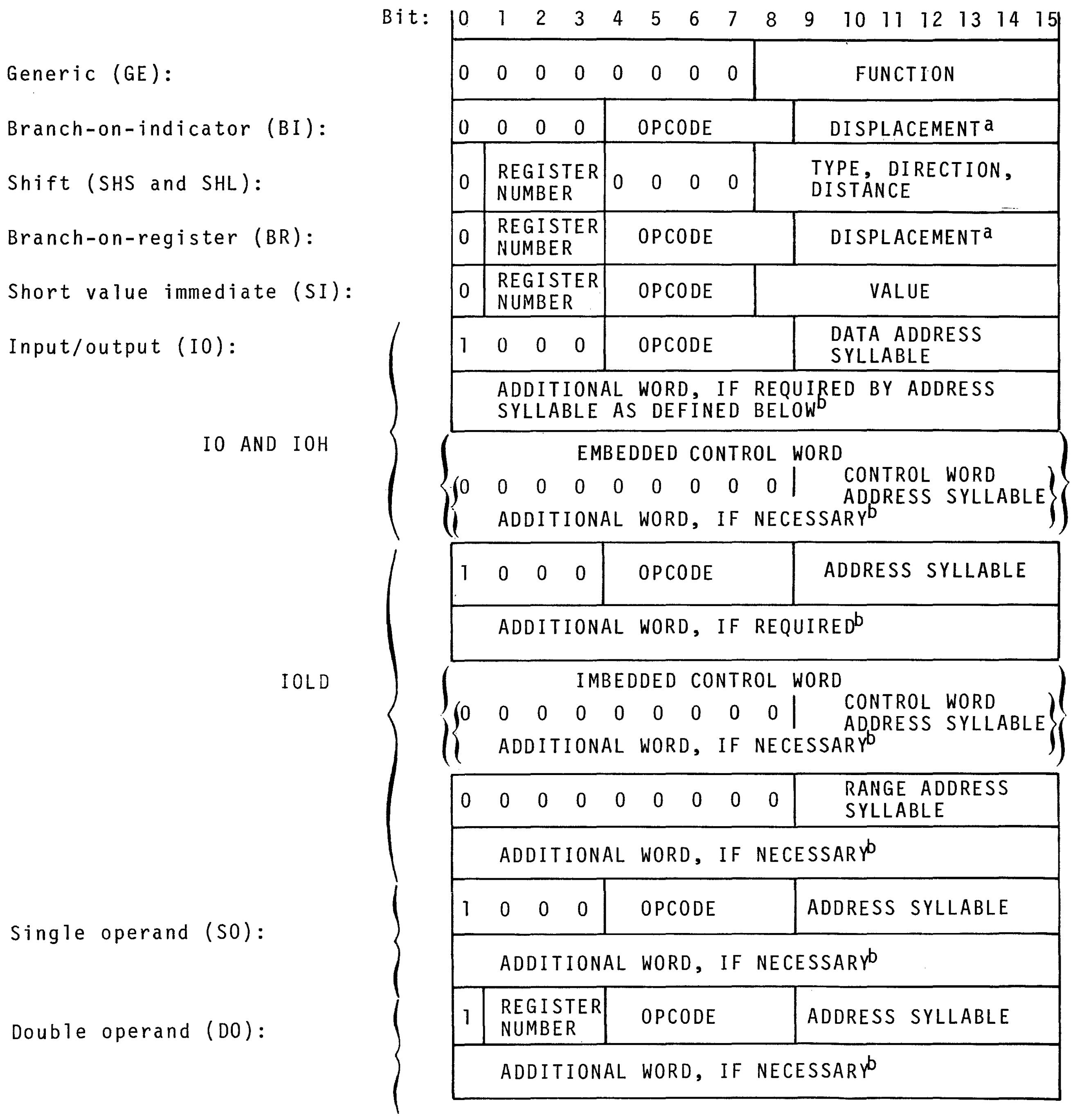

Bit: 0 1 2 3 4 5 6 7
Scientific Indicator Register (SI)
EU
SE
PE
SG
SL
Result of last scientific compare:
Less than
Greater than
Precision error (trap 22)
Significance error (trap 21)
Exponent underflow (trap 19)
Bit: 0 1 2 3 4 5 6 7
SIP Mode Register (M4)
R/T
ML1
AL1
ML2
AL2
ML3
AL3
SA1
SA2
SA3
Bit: 0 1 2 3 4 5 6 7
SIP Trap Mask Register (M5)
EUM
SEM
PEM
Precision Error Trap Mask
Significance Error Trap Mask
Exponent Underflow Trap Mask

R/T:	Round/Truncate Mode 0 - Truncate 1 - Round
ML:	Memory Length (Length of main memory data field to or from which data is transferred via a scientific accumulator (SA)) 0 - Two words 1 - Four words
AL:	Accumulator Length (Length of scientific accumulator data field to or from which data is transferred to/from main memory, a hardware register, or another SIP register) 0 - Two words 1 - Four words

Figure A-1 (cont). Level 6 Hardware Registers

ASSEMBLY LANGUAGE INTERNAL FORMATS BY TYPE

Each of the seven types (i.e., generic, branch-on-register, etc.) of assembly language instructions is stored in memory in a predefined format, as shown in Figure A-2.



Bit: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Generic (GE):
0 0 0 0 0 0 0 0 | FUNCTION
Branch-on-indicator (BI):
0 0 0 0 | OPCODE | DISPLACEMENT[a]
Shift (SHS and SHL):
0 | REGISTER NUMBER | 0 0 0 0 | TYPE, DIRECTION, DISTANCE
Branch-on-register (BR):
0 | REGISTER NUMBER | OPCODE | DISPLACEMENT[a]
Short value immediate (SI):
0 | REGISTER NUMBER | OPCODE | VALUE
Input/output (IO):
1 0 0 0 | OPCODE | DATA ADDRESS SYLLABLE
ADDITIONAL WORD, IF REQUIRED BY ADDRESS SYLLABLE AS DEFINED BELOW[b]
IO AND IOH
EMBEDDED CONTROL WORD
0 0 0 0 0 0 0 0 0 | CONTROL WORD ADDRESS SYLLABLE
ADDITIONAL WORD, IF NECESSARY[b]
1 0 0 0 | OPCODE | ADDRESS SYLLABLE
ADDITIONAL WORD, IF REQUIRED[b]
IOLD
IMBEDDED CONTROL WORD
0 0 0 0 0 0 0 0 0 | CONTROL WORD ADDRESS SYLLABLE
ADDITIONAL WORD, IF NECESSARY[b]
0 0 0 0 0 0 0 0 0 | RANGE ADDRESS SYLLABLE
ADDITIONAL WORD, IF NECESSARY[b]
Single operand (SO):
1 0 0 0 | OPCODE | ADDRESS SYLLABLE
ADDITIONAL WORD, IF NECESSARY[b]
Double operand (DO):
1 | REGISTER NUMBER | OPCODE | ADDRESS SYLLABLE
ADDITIONAL WORD, IF NECESSARY[b]

[a]If the displacement value specified is 0, the location to be branched to is specified in the next sequential word; if it is 1, the next sequential word specifies the displacement (in words) from the address of this instruction; otherwise, the displacement value specified is the displacement, in two's complement form, from the current instruction to the destination.

Figure A-2. Internal Formats of Assembly Language Instructions

[b]Depending on the form of the address expression used in the source code, the generated address syllable may occupy one or two words, as follows:

- o If the address expression was of the immediate memory, immediate operand, or P-relative address forms, the hexadecimal address of the location specified, the displacement to it, or the value of the operand itself is contained in the next sequential word or words.
- o If the address expression was of the B-relative address form, the address of the location is derived by performing the operation(s) specified in Table A-3.
- o If the address expression was of the register addressing form, the value or address is contained in the specified register.

For those instruction types that show register number in bits 1 through 3, this is the number of register specified in the first operand of the multiple-operand instruction that requires a register in the first operand.

Figure A-2 (Cont). Internal Formats of Assembly Language Instructions

HEXADECIMAL REPRESENTATION OF INSTRUCTIONS

Table A-1 illustrates the hexadecimal representation of the assembly language instructions as they appear in a printout. These representations are derived from the formats of the various types described under "Assembly Language Internal Formats by Type," (Figure A-2.

In the table, when 0+addsyl or 0+x is specified, it indicates that the last byte is a 7-bit byte preceded by a binary 0; 8+addsyl or 8+x indicates a 7-bit byte preceded by a binary 1. In either case, only the last seven bits are significant. addsyl is defined in Table A-2; x is the displacement in a branch instruction, as defined under "Assembly Language Internal Formats by Type," (Figure A-2); d is the shift displacement, in bits.

TABLE A-1. INTERNAL PRESENTATION OF ASSEMBLY LANGUAGE INSTRUCTIONS

First Hexadecimal Digit	Second Hexadecimal Digit	Third Hexadecimal Digit	Fourth Hexadecimal Digit	Instruction	Type
0	0	0	0	HLT	GE
			1	MCL	
			2	BRK	
			3	RTT	
			4	RTCN	
			5	RTCF	
			6	WDTN	
			7	WDTF	
	2	0+x	x	BL	BI
	2	8+x	x	BGE	
	3	0+x	x	BG	
	3	8+x	x	BLE	
	4	0+x	x	BOV	
	4	8+x	x	BNOV	
	5	0+x	x	BBT	
	5	8+x	x	BBF	
	6	0+x	x	BCT	

TABLE A-1 (CONT). INTERNAL REPRESENTATION OF ASSEMBLY LANGUAGE INSTRUCTIONS

First Hexidecimal Digit	Second Hexadecimal Digit	Third Hexadecimal Digit	Fourth Hexadecimal Digit	Instruction	Type
0	6	8+x	x	BCF	BI
	7	0+x	x	BIOT	
	7	8+x	x	BIOF	
	8	0+x	x	BAL	
	8	8+x	x	BAGE	
	9	0+x	x	BE	
	9	8+x	x	BNE	
	A	0+x	x	BAG	
	A	8+x	x	BALE	
	B	0+x	x	BSU	
	B	8+x	x	BSE	
	F	0+x	x	NOP	
	F	8+x	x	B	
1-3	4	0+x	x	SBLZ	
		8+x	x	SBGEZ	
	5	0+x	x	SBEZ	
		8+x	x	SBNEZ	
	6	0+x	x	SBGZ	
		8+x	x	SBLEZ	
4	4	0+x	x	SBL	
		8+x	x	SBGE	
	5	0+x	x	SBEQ	
		8+x	x	SBNE	
	6	0+x	x	SBG	
		8+x	x	SBLE	
5	4	0+x	x	SBPE	
		8+x	x	SBNPE	
6	4	0+x	x	SBSE	
		8+x	x	SBNSE	
7	4	0+x	x	SBEU	
		8+x	x	SBNEU	
1-7	0	0	d	SOL	SHS
		1	d	SCL	
		2	d	SAL	
		3	d	DCL	
		4	d	SOR	
		5	d	SCR	
		6	d	SAR	
		7	d	DCR	
		8			
		9	d	DOL	SHL
		A			
		B	d	DAL	
		C			
		D	d	DOR	
		E			
		F	d	DAR	
	7	0+x	x	BDEC	BR
	7	8+x	x	BINC	

TABLE A-1 (CONT). INTERNAL REPRESENTATION OF ASSEMBLY LANGUAGE INSTRUCTIONS

First Hexadeciaml Digit	Second Hexadecimal Digit	Third Hexadecimal Digit	Fourth Hexadecimal Digit	Instruction	Type
1-7	8	0+x	x	BLZ	BR
	8	8+x	x	BGEZ	
	9	0+x	x	BEZ	
	9	8+x	x	BNEZ	
	A	0+x	x	BGZ	
	A	8+x	x	BLEZ	
	B	0+x	x	BEVN	
	B	8+x	x	BODD	
	C	immedvalue		LDV	SI
	D	immedvalue		CMV	
	E	immedvalue		ADV	
	F	immedvalue		MLV	
8	0	0+addsyl		IO	IO
	1	0+addsyl		IOH	
	1	8+addsyl		IOLD	
	2	0+addsyl		NEG	SO
	2	8+addsyl		LB	
	3	8+addsyl		JMP	
	6	0+addsyl		CPL	
	7	0+addsyl		CL	
	7	8+addsyl		CLH	
	8	0+addsyl		LBF	
	8	8+addsyl		DEC	
	9	0+addsyl		LBT	
	9	8+addsyl		CMZ	
	A	0+addsyl		LBS	
	A	8+addsyl		INC	
	B	0+addsyl		LBC	
	B	8+addsyl		ENT	
	C	0+addsyl		STS	
	C	8+addsyl		LDI	
	D	0+addsyl		SDI	
	D	8+addsyl		CMN	
	E	0+addsyl		LEV	
	E	8+addsyl		CAD	
	F	0+addsyl		SAVE	
	F	9+addsly		RSTR	
C	C	0+addsyl		SCZQ	
	8	8+addsyl		SCZD	
	D	0+addsyl		SNGQ	
	9	8+addsyl		SNGD	
9-F	0	0+addsyl		MTM	DO
	0	8+addsyl		LDH	
	1	8+addsyl		CMH	
	2	0+addsyl		SUB	
	2	8+addsyl		LLH	
	3	0+addsyl		DIV	
	3	8+addsyl		LNJ	
	4	0+addsyl		OR	
	4	8+addsyl		ORH	

TABLE A-1 (CONT). INTERNAL REPRESENTATION OF ASSEMBLY LANGUAGE INSTRUCTIONS

First Hexadecimal Digit	Second Hexadecimal Digit	Third Hexadecimal Digit	Fourth Hexadecimal Digit	Instruction	Type
9-F	5	0+addsyl		AND	DO
	5	8+addsyl		ANH	
	6	0+addsyl		XOR	
	6	8+addsyl		XOH	
	7	0+addsyl		STM	
	7	8+addsyl		STH	
	8	0+addsyl		LDR	
9-B	8	8+addsyl		SLD	
D-F	8	8+addsyl		SCM	
9-F	9	0+addsyl		CMR	
9-B	9	8+addsyl		SAD	
D-F	9	8+addsyl		SSB	
9-F	A	0+addsyl		ADD	
	A	8+addsyl		SRM	
	B	0+addsyl		MUL	
	B	8+addsyl		LAB	
9-B	C	0+addsyl		SML	
D-F	C	0+addsyl		SDV	
9-F	C	8+addsyl		LDB	
	D	8+addsyl		CMB	
	E	0+addsyl		SWR	
	E	8+addsyl		SWB	
	F	0+addsyl		STR	
	F	8+addsyl		STB	

TABLE A-2. ADDRESS SYLLABLES

mmmm	rrr = 000		rrr = ddd		
	i = 0	i = 1	i = 0	i = 1	
000	<location	*<location	$Bn	*$Bn	
001	<location.$R1	*<location.$R1	$Bn.$R1	*$Bn.$R1	
010	<location.$R2	*<location.$R2	$Bn.$R2	*$Bn.$R2	
011	<location.$R3	*<location.$R3	$Bn.$R3	*$Bn.$R3	
100	location	*location	$Bn.value	*Bn.value	
101	reserved	reserved	{=$Rn =$Bn}	$Bk.–$R1	$Bq.+$R1
110	reserved	reserved	–$Bn	$Bk.–$R2	$Bq.+$R2
111	{=location =value}	reserved	+$Bn	$Bk.–$R3	$Bq.+$R3

NOTE: An address syllable can be represented as mmmirrr, which are the last seven bits in the word; n can be any number between 1 and 7 and is equal to rrr for rrr≠0; k is a number within the range 1 through 3 and is equal to rrr for rrr = 1, 2, 3; and q is a number within the range 1 through 3 and is equal to rrr-4 for rrr = 4, 5, 6, 7. For more imformation about these address expressions, see "Addressing Techniques" in Section 5.

VALID ADDRESS EXPRESSIONS

Table 4-3 lists all of the valid address expressions and shows graphically how each derives the effective address of the data to be used in the operation.

The various types of symbolic names, constants, and expressions (other than address expressions) are described in detail in Section 2.

TABLE A-3. SUMMARY OF VALID FORMS OF ADDRESS EXPRESSIONS

Addressing Technique		**Address Expression Form**	**Generation of Effective Address**
Register Addressing		=$Rn =$Bn =$Sn	$\underline{\text{Rn}} = \underline{\text{EA}}$ $\underline{\text{Bn}} = \underline{\text{EA}}$ $\underline{\text{Sn}} = \underline{\text{EA}}$
Immediate Memory Addressing	Direct	$<\left\{\begin{matrix}\text{locexpression}\\ \left\{\begin{matrix}+\\-\end{matrix}\right\}\text{templabel}\end{matrix}\right\}$	location = EA
	Indirect	$*<\left\{\begin{matrix}\text{locexpression}\\ \left\{\begin{matrix}+\\-\end{matrix}\right\}\text{templabel}\end{matrix}\right\}$	$\underline{\text{location}} = \text{EA}$
	Indexed Direct	$<\left\{\begin{matrix}\text{locexpression}\\ \left\{\begin{matrix}+\\-\end{matrix}\right\}\text{templabel}\end{matrix}\right\}.\$R\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}$	$\text{location} + \underline{\text{R}\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}} = \text{EA}$
	Indexed Indirect	$*<\left\{\begin{matrix}\text{locexpression}\\ \left\{\begin{matrix}+\\-\end{matrix}\right\}\text{templabel}\end{matrix}\right\}.\$R\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}$	$\underline{\text{location}} + \underline{\text{R}\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}} = \text{EA}$
Immediate Operand Addressing		=locexpression	Address of current address syllable + 1 = EA
		=hexstringconstant	
		$=\left\{\begin{matrix}\text{intvalexpression}\\ \text{extvallabel}\end{matrix}\right\}$	
P-Register Addressing	Direct	$\left\{\begin{matrix}\text{intlocexpression}\\ \left\{\begin{matrix}+\\-\end{matrix}\right\}\text{templabel}\end{matrix}\right\}$	internal location = EA
	Indirect	$*\left\{\begin{matrix}\text{intlocexpression}\\ \left\{\begin{matrix}+\\-\end{matrix}\right\}\text{templabel}\end{matrix}\right\}$	$\underline{\text{internal location}} = \text{EA}$
B-Register Addressing	Direct	$Bn	$\underline{\text{Bn}} = \text{EA}$
	Indirect	*$Bn	$\underline{\text{Bn}} = \text{location}$ $\underline{\text{location}} = \text{EA}$
	Indexed Direct	$\$\text{Bn}.\$\text{R}\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}$	$\underline{\text{Bn}} + \underline{\text{R}\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}} = \text{EA}$
	Indexed Indirect	$*\$\text{Bn}.\$\text{R}\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}$	$\underline{\text{Bn}} = \text{location}$ $\underline{\text{location}} + \underline{\text{R}\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}} = \text{EA}$
	Direct + Displacement	$\$\text{Bn}.\left\{\begin{matrix}\text{intvalexpression}\\ \text{extvallabel}\end{matrix}\right\}$	$\underline{\text{Bn}} + \text{value} = \text{EA}$

TABLE A-3 (CONT). SUMMARY OF VALID FORMS OF ADDRESS EXPRESSIONS

Addressing Technique		**Address Expression Form**	**Generation of Effective Address**
B-Register Addressing (Cont.)	Indirect + Displacement	$*\$Bn.\begin{Bmatrix}\text{intvalexpression}\\ \text{extvallabel}\end{Bmatrix}$	$\underbrace{\text{Bn}}$ + value = location $\underbrace{\text{location}}$ = EA
	Push	-$Bn	$\underbrace{\text{Bn}} \leftarrow (\underbrace{\text{Bn}} - 1)$ $\underbrace{\text{Bn}} = \text{EA}$
	Pop	+$Bn	$\underbrace{\text{Bn}} = \text{EA}$ $\underbrace{\text{Bn}} \leftarrow (\underbrace{\text{Bn}} + 1)$
	Indexed Push	$\$B\begin{Bmatrix}1\\2\\3\end{Bmatrix}.-\$R\begin{Bmatrix}1\\2\\3\end{Bmatrix}$	$\underbrace{R\begin{Bmatrix}1\\2\\3\end{Bmatrix}} \leftarrow (\underbrace{R\begin{Bmatrix}1\\2\\3\end{Bmatrix}} - 1)$ $\underbrace{B\begin{Bmatrix}1\\2\\3\end{Bmatrix}} + \underbrace{R\begin{Bmatrix}1\\2\\3\end{Bmatrix}} = \text{EA}$
	Indexed Pop	$\$B\begin{Bmatrix}1\\2\\3\end{Bmatrix}.+\$R\begin{Bmatrix}1\\2\\3\end{Bmatrix}$	$\underbrace{B\begin{Bmatrix}1\\2\\3\end{Bmatrix}} + \underbrace{R\begin{Bmatrix}1\\2\\3\end{Bmatrix}} = \text{EA}$ $\underbrace{R\begin{Bmatrix}1\\2\\3\end{Bmatrix}} \leftarrow (\underbrace{R\begin{Bmatrix}1\\2\\3\end{Bmatrix}} + 1)$
Short Displacement		$>\begin{Bmatrix}\text{intlocexpression}\\ \begin{Bmatrix}+\\-\end{Bmatrix}\text{templabel}\end{Bmatrix}$	location = EA
Special		$>=\begin{Bmatrix}\text{intvalexpression}\\ \text{extvallabel}\end{Bmatrix}$	Word following the word(s) containing op code + first operand address syllable = EA
Interrupt Vector		$\$IV.\begin{Bmatrix}\text{int-val expression}\\ \text{ext-val-label}\end{Bmatrix}$	IV + value = EA

NOTE: The symbols used in this table have the following meanings:

$\underbrace{\quad}$ -	Contents of . . .	* -	Indirection indicator
EA -	Effective Address	< -	Immediate memory addressing
← -	Replaces the . . . (the element pointed at)	> -	Short displacement addressing
		<=	Specified Addressing
		. -	Component separator (indexing and displacement)

locexpression – location expression (any type)
templabel – temporary label
hexstringconstant – hexadecimal string constant
intvalexpression – internal value expression
int-val-label-internal value label
extvallabel – external value label
intlocexpression – internal location expression

All other notations represent standard usage as defined in the preface of this manual or required Assembler-specific symbols.

APPENDIX B

HEXADECIMAL NUMBERING SYSTEM

Level 6 stores all data in memory in the form of binary digits. However, to save space in printouts, this data is always shown in its hexadecimal equivalent (unless an ASCII memory dump is requested). This appendix explains how to convert from hexadecimal to decimal and vice versa, as well as how to perform hexadecimal arithmetic operations.

Table B-1 shows the comparison between binary (i.e., base 2), decimal (i.e., base 10), and hexadecimal (i.e., base 16) symbols.

TABLE B-1. COMPARISON OF BINARY, DECIMAL, AND HEXADECIMAL SYMBOLS

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

In the course of coding your assembly language program, it is possible to define data as a decimal, hexadecimal, or binary number, or an ASCII symbol, as illustrated in Table B-4. However, in memory, all data is stored in binary.

Data that is defined as ASCII in the source program is stored as the binary equivalent of the ASCII symbol, and shown in the printout as the hexadecimal equivalent of the stored binary value.

Numeric data, on the other hand, is converted to hexadecimal, and stored as the binary equivalent of the hexadecimal digit.

Table B-2 illustrates how the value 32 is stored in memory depending on how it is defined in the source program (i.e., depending on whether it is defined as an ASCII value, binary value, decimal value, or hexadecimal value.) In addition, it shows how the stored value would appear in an ASCII or hexadecimal printout.

TABLE B-2. STORAGE AND PRINTOUT OF THE VALUE 32

Data Type	Stored in Memory	Hex Printout	ASCII Printout
A'32'	0011001100110010	3332	32
X'32'	0000000000110010	0032	.2
Z'32'	0011001000000000	3200	2.
32 (Dec)	0000000000100000	0020	.Space
B'00110010'	0011001000000000	3200	2.

As you can see in this table, hexadecimal and binary are identical. In addition, it illustrates how an ASCII symbol is expanded according to Table B-4. Finally, it shows a decimal value that is first converted to its hexadecimal (i.e., binary) equivalent and then stored in memory.

The following pages explain how to compute the conversions and how to do hexadecimal arithmetic.

DECIMAL-TO-HEXADECIMAL CONVERSION

The system automatically converts all decimal data to its binary (i.e., hexadecimal) equivalent when storing it in memory. It then operates on that binary data.

You can determine how a decimal number will be stored in memory as follows:

1. Divide the decimal number by 16. The remainder becomes the low-order (i.e., rightmost) hexadecimal digit.
2. Divide the whole number result of the last division by 16. The remainder becomes the next-highest-order hexadecimal digit.
3. Continue this process until the whole number result of a division is 0. The remainder becomes the highest-order (i.e., leftmost) hexadecimal digit.

For example, to determine the hexadecimal equivalent of the decimal number 47,401, do the following:

1. Divide 47,401 by 16.
   The result is 2962. The remainder is $\underline{9}$.
2. Divide 2962 by 16.
   The result is 185. The remainder is $\underline{2}$.
3. Divide 185 by 16.
   The result is 11. The remainder is $\underline{9}$.
4. Divide 11 by 16.
   The result is 0. The remainder is $\underline{11}$.

Using Table B-1, you can see that in hexadecimal 11 is represented by B. Thus, the hexadecimal equivalent of 47401_{10} is B929.

HEXADECIMAL-TO-DECIMAL CONVERSION

The type of conversion you will most commonly be confronted with will be from hexadecimal to decimal because, unless you specifically request an ASCII memory dump, printouts of memory will always be in hexadecimal. To identify ASCII data readily, look for repetition of the first character in a byte. For example,

$\underline{3}1\underline{3}2$ $\underline{3}3\underline{3}3$ $\underline{3}3\underline{3}5$ $\underline{3}6\underline{3}7$ xxxx xx...

is a list of ASCII numbers (i.e., 1, 2, 3, 3, 3, 5, 6, 7, in the example). In most other cases, the hexadecimal symbols will appear to be quite random. If the stored hexadecimal symbols represent numeric data, you can convert it to decimal as follows:

1. Multiply the decimal equivalent (see Table B-1) of the high-order (i.e., leftmost) hexadecimal digit by 16.
2. Add the decimal equivalent of the next-lowest-order hexadecimal to the result of step 1.
3. Multiply the result of step 2 by 16.
4. Repeat steps 2 and 3 until you reach the last hexadecimal digit.
5. Simply add the decimal equivalent of the last hexadecimal digit to the result of the last previous multiplication.

For example, to convert the hexadecimal value 1C8A to its decimal equivalent, do the following:

1. Multiply 1 by 16.
   The result is 16.
2. Add 12 (i.e., C = 12_{10}).
   The result is 28.
3. Multiply 28 by 16.
   The result is 448.
4. Add 8.
   The result is 456.
5. Multiply 456 by 16.
   The result is 7296.
6. Add 10 (i.e., A = 10_{10}).
   The result is 7306.

Thus, the decimal equivalent of $1C8A_{16}$ is 7306.

Alternatively, you may use Table B-3 to convert hexadecimal numeric data to its decimal equivalent.

TABLE B-3. HEXADECIMAL/DECIMAL CONVERSION

Word							
Byte				Byte			
H1	**Decimal**	**H2**	**Decimal**	**H3**	**Decimal**	**H4**	**Decimal**
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10

TABLE B-3 (CONT). HEXADECIMAL/DECIMAL CONVERSION

Word							
Byte				Byte			
H1	Decimal	H2	Decimal	H3	Decimal	H4	Decimal
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

NOTE: H1 is the first hexadecimal digit
H2 is the second hexadecimal digit
H3 is the third hexadecimal digit
H4 is the fourth hexadecimal digit
first and second digits of a byte

If H1 is 0 through 7, the number is positive and you compute the decimal equivalent of the given hexadecimal number by summing the decimal equivalent of H1, H2, H3, and H4.

NOTE: For a signed integer byte, use H3 and H4 only.

If H1 is 8 through F, the number is negative, and you must find the two's complement before using the table. You can compute the two's complement by subtracting the hexadecimal number from 10000 (hexadecimal) or by changing all 0's to 1 and all 1's to 0 and then adding a binary 1. You can then find the decimal equivalent directly from the table, appending a minus sign to the final result.

HEXADECIMAL-TO-ASCII CONVERSION

If the stored data is an ASCII value, it can be translated by converting the hexadecimal value in the printout to its ASCII equivalent using Table B-4.

For example, the locations that contain the start of your program should have the following hexadecimal representation:

5449 544C 4520 hhhh hh...

By pairing the digits (e.g, 54) and locating the character in the table where these two digits intersect, you can ascertain the ASCII equivalent of the stored hexadecimal value. Remembering that the first hexadecimal digit corresponds to the H1 row and that the second digit corresponds to the H2 column, the above representation translates to: TITLEΔ.

If you wish to ascertain the hexadecimal equivalent of an ASCII character, simply locate the character in the table and record the H1H2 values at the top and left of the table.

TABLE B-4. HEXADECIMAL/ASCII CONVERSION

	H1							
H2	01	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P		p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	:
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	→	n	μ
F	SI	US	/	?	O	–	o	DEL

Control Characters

NUL	Null
SOH	Start of Heading
STX	Start of Text
ETX	End of Text
EOT	End of Transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	Bell
BS	Backspace
HT	Horizontal Tab
LF	Line Feed
VT	Vertical Tab
FF	Form Feed
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC1	Device Control 1
DC2	Device Control 2
DC3	Device Control 3
DC4	Device Control 4

NAK	Negative Acknowledge
SYN	Synchronous Idle
ETB	End of Transmission Block
CAN	Cancel
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator
SP	Space
DEL	Delete

HEXADECIMAL ADDITION

Table B-5 illustrates a hexadecimal addition table. When using this table, whenever there is a result that has a 1 preceding a hexadecimal value, that 1 represents a carry. When a carry occurs in an arithmetic operation, the C-bit in the I-register is set to 1; if the carry results in the high-order digit being lost because the result field is not large enough to contain the result, the 0V-bit in the I-register is set to 1.

TABLE B-5. HEXADECIMAL ADDITION TABLE

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

The following example illustrates how the table can be used in hexadecimal addition:

```
augend     A2B5
addend    +494F
          -----
```

The result of adding F and 5 is 14. Therefore, 4 becomes the low-order digit in the sum; 1 is carried. Then, B + 4 + 1 = 10; as before, 0 becomes the next-lowest-order digit, and 1 is carried. Then, 9 + 2 + 1 = C; there is no carry. Finally, A + 4 = E, with no carry. The sum of the hexadecimal numbers shown above is EC04.

During this addition, the C-bit in the I-register was set to 1; however (assuming that the result field allowed a 1-word result), since the addition of the high-order digits did not result in a carry (which would have meant that the carried digit would have been lost), the 0V-bit was not set to 1 (i.e., it was set to 0).

HEXADECIMAL SUBTRACTION

Hexadecimal subtraction is the opposite of hexadecimal addition. Instead of carries, it is necessary to borrow. When you borrow a 1 from the next-highest-order digit of a minuend, it is the equivalent of adding 16 to the minuend of the digit you are subtracting from. The following example illustrates this concept:

minuend	3A
subtrahend	-1B

Since B is higher than A, it is necessary to borrow 1 from the 3, and adding 16 to A (i.e., 16 + 10 = 26), and subtracting B (i.e., 11) from the result, obtaining 15 (but since this is hexadecimal arithmetic, you must change the 15 to F); then, you must subtract 1 from 2 (don't forget that 1 was borrowed from the 3); the result of this operation is 1F.

HEXADECIMAL MULTIPLICATION

To do hexadecimal multiplication, you can use Table B-6. As when multiplying in any numbering system, you must record the low-order digit and add the remainder (i.e., the high-order hexadecimal digit shown in the table) to the result of the multiplication of the next-lowest-order hexadecimal digit.

TABLE B-6. HEXADECIMAL MULTIPLICATION TABLE

1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

For example, to multiply the following hexadecimal digits:

multiplicand	2A5
multiplier	x 3

Using the table, 3 x 5 = F, and there is no remainder. Then, 3 x A = 1E; E is recorded, and the remainder (i.e., 1) is saved to be added to the result of the multiplication of the next digit. So, 3 x 2 = 6, plus the remainder of 1 = 7. The result of this arithmetic operation is 7EF.

HEXADECIMAL DIVISION

Due to the complexity of this type of operation, it is suggested that you convert the hexadecimal digits to decimal, perform the division, and then convert the answer to hexadecimal.

APPENDIX C

SAMPLE ASSEMBLY LANGUAGE PROGRAM

The following sample program illustrates many of the aspects of the assembly language described in this manual. For a definition of the fields that appear in the listing, refer to the Program Development Tools manual.

```
CHKNML              L6 ASSEMBLER-0100                     PAGE 0001

    000001                                      TITLE     CHKNML
    000002                              * PROGRAM COMPARES TEST RESULTS OF TEST MODULES WHOSE ADDRESSES ARE
    000003                              * STORED IN $COMM TO THE EXPECTED TEST RESULTS AS DESCRIBED IN TABLOC
    000004                                      XVAL      TSTMAX
    000005                                      XLOC      TABLOC
    000006                                      XLOC      ZIOSOL
    000007                                      XLOC      ZIOSWR
    000008                                      XLOC      ZIOSCO
    000009        0100                          COMM      X'100'
    000010                              * GET FILENAME  AND CHANNEL NO
    000011  0000  A843 FFEF             STRT    LDR       $R2,$B3.-17
    000012  0002  BBA3                          LAB       $B3,$B3.$R2
    000013  0003  BBC3 FFEC                     LAB       $B3,$B3.-20       SET B3 TO LIST FILE AT
    000014  0005  9873                          LDR       $R1,+$B3          SET B3 TO FILENAME
    000015  0006  1D02                          CMV       $R1,2
    000016  0007  0981 007C                     BNE       ERNLST            NO LIST FILE ATTACHED
    000017  0009  9843 0007                     LDR       $R1,$B3.7         SET R1 TO CHANNEL NO.
    000018                              * OPEN LIST FILE
    000019  000B  CBC0 0079                     LAB       $B4,LSTDCB
    000020  000D  D380 0000     X               LNJ       $B5,<ZIOSOL       OPEN ROUTINE
    000021  000F  1981 006F                     BNEZ      $R1,EROPEN
    000022                              * WRITE HEADER MSG
    000023  0011  1C1E                          LDV       $R1,X'1E'         MSG LENGTH
    000024  0012  2C00                          LDV       $R2,X'0'
    000025  0013  BBC0 0081                     LAB       $B3,WBUF01        MSG ADDRESS
    000026  0015  CBC0 006F                     LAB       $B4,LSTDCB
    000027  0017  D380 0000     X               LNJ       $B5,<ZIOSWR       WRITE ROUTINE
    000028  0019  1981 0066                     BNEZ      $R1,ERHDR
    000029  001B  3CFF                          LDV       $R3,-X'1'
    000030  001C  3E01                  TLOOP   ADV       $R3,X'1'
    000031  001D  B970 0000     X               CMR       $R3,=TSTMAX       CHECKED ALL TEST RESULTS ?
    000032  001F  0301 004E                     BG        ENDTST
    000033  0021  CCB0 0000     K               LDB       $B4,<$COMM.$R3
    000034  0023  CBC4 001C                     LAB       $B4,$B4.X'1C'     CREATE STATUS BLOCK PTR
    000035  0025  F830 0000     X               LDR       $R7,<TABLOC.$R3   GET EXPECTED VALUE
    000036  0027  F944 0003                     CMR       $R7,$B4.X'3'      COMPARE TO ACTUAL STATWD
    000037  0029  0973                          BE        >TLOOP            TEST OK - CHECK NEXT TEST
    000038  002A  EBC0 007A                     LAB       $B6,WBUF2A
    000039  002C  D830 0000     K               LDR       $R5,<$COMM.$P3
    000040  002E  F3C0 0027                     LNJ       $B7,DUMPWD        CONVERT TEST ADDR TO ASCII
    000041  0030  EBC0 0077                     LAB       $B6,WBUF2B
    000042  0032  D804                          LDR       $R5,$B4
    000043  0033  F3C0 0022                     LNJ       $B7,DUMPWD        CONVERT SYML VALUE TO ASCII
    000044  0035  CBC4 0001                     LAB       $B4,$B4.X'1'
    000045  0037  EBC0 0073                     LAB       $B6,WBUF2C
    000046  0039  D804                          LDR       $R5,$B4
    000047  003A  F3C0 001B                     LNJ       $B7,DUMPWD        CONVERT TEST NUM TO ASCII
    000048  003C  CBC4 0001                     LAB       $B4,$B4.X'1'
    000049  003E  EBC0 006F                     LAB       $B6,WBUF2D
    000050  0040  D804                          LDR       $R5,$B4
    000051  0041  F3C0 0014                     LNJ       $B7,DUMPWD        CONVERT SYMV VALUE TO ASCII
    000052  0043  CBC4 0001                     LAB       $B4,$B4.X'1'
    000053  0045  EBC0 006B                     LAB       $B6,WBUF2E
    000054  0047  D804                          LDR       $R5,$B4
    000055  0048  F3C0 000D                     LNJ       $B7,DUMPWD        CONVERT STATUS WORD TO ASCII
    000056                              * WRITE VALUES
    000057  004A  1C1E                          LDV       $R1,X'1E'         MSG LENGTH
    000058  004B  2C00                          LDV       $R2,X'0'
    000059  004C  BBC0 0057                     LAB       $B3,WBUF20        MSG ADDRESS
    000060  004E  CBC0 0036                     LAB       $B4,LSTDCB
```

```
CHKNML              L6 ASSEMBLER-0100                              PAGE 0002

  000061  0050  D380 0000    X                  LNJ        $B5,<ZIOSWR         WRITE ROUTINE
  000062  0052  1981 002E                       BNEZ       $R1,ERVAL
  000063  0054  83C0 FFC7                       JMP        TLOOP
  000064                         * ROUTINE ACCEPTS A VALUE IN R5 AND PUTS ITS ASCII EQUIVALENT
  000065                         * IN THE TWO WORDS POINTED TO BY B6
  000066  0056  4CFC             DUMPWD         LDV        $R4,-X'4'           SET COUNTER
  000067  0057  CF40 0000    T                  STR        $R4,+$C
  000068  0059  7C00                            LDV        $R7,X'0'
  000069  005A  4C00             $A             LDV        $R4,X'0'
  000070  005B  5084                            DOL        $R5,4
  000071  005C  4E30                            ADV        $R4,X'30'
  000072  005D  C940 0000    T                  CMR        $R4,+$F
  000073  005F  0380         T                  BLE        >+$E
  000074  0060  4E07                            ADV        $R4,X'07'
  000075  0061  F454             $E             OR         $R7,=$R4
  000076  0062  8AC0 FFF5    T                  INC        +$C
  000077  0064  0600         T                  BCT        >+$D
  000078  0065  7088                            DOL        $R7,8
  000079  0066  0FF4         T                  B          >-$A
  000080  0067  EF46 0000        $D             STR        $R6,$B6.X'0'
  000081  0069  FF46 0001                       STR        $R7,$B6.X'1'
  000082  006B  8387                            JMP        $B7                 RETURN TO CALLER
  000083  006C  0000             $C             DC         Z'0'
  000084  006D  0039             $F             DC         Z'0039'
  000085                         * WRITE END TEST
  000086  006E  1C0A             ENDTST         LDV        $R1,X'A'            MSG LENGTH
  000087  006F  2C00                            LDV        $R2,X'0'
  000088  0070  BBC0 0043                       LAB        $B3,WBUF03          MSG ADDRESS
  000089  0072  CBC0 0012                       LAB        $B4,LSTDCB
  000090  0074  D380 0000    X                  LNJ        $B5,<ZIOSWR         WRITE ROUTINE
  000091  0076  1981 000B                       BNEZ       $R1,EREND
  000092                         * CLOSE LIST FILE
  000093  0078  CBC0 000C                       LAB        $B4,LSTDCB
  000094  007A  D380 0000    X                  LNJ        $B5,<ZIOSCO         CLOSE ROUTINE
  000095  007C  1981 0006                       BNEZ       $R1,ERCLS
  000096  007E  0000                            HLT
  000097  007F  0000             EROPEN         HLT
  000098  0080  0000             ERHDR          HLT
  000099  0081  0000             ERVAL          HLT
  000100  0082  0000             EREND          HLT
  000101  0083  0000             ERCLS          HLT
  000102  0084  0000             ERNLST         HLT
  000103  0085  0000             LSTDCB         RESV       16,0
  000104  0095  4120             WBUF01         DC         'A tloc  tsym  tnum  tval  tswd'
          0096  746C
          0097  6F63
          0098  2020
          0099  7473
          009A  796D
          009B  2020
          009C  746E
          009D  756D
          009E  2020
          009F  7476
          00A0  616C
          00A1  2020
          00A2  7473
          00A3  7764
  000105  00A4  4120             WBUF20         DC         'A '
  000106  00A5  2020             WBUF2A         DC         '      '


CHKNML              L6 ASSEMBLER-0100                              PAGE 0003

          00A6  2020
          00A7  2020
  000107  00A8  2020             WBUF2B         DC         '      '
          00A9  2020
          00AA  2020
  000108  00AB  2020             WBUF2C         DC         '      '
          00AC  2020
          00AD  2020
  000109  00AE  2020             WBUF2D         DC         '      '
          00AF  2020
          00B0  2020
  000110  00B1  2020             WBUF2E         DC         '      '
          00B2  2020
          00B3  2020
  000111  00B4  4120             WBUF03         DC         'A end test'
          00B5  656E
          00B6  6420
          00B7  7465
          00B8  7374
  000112  00B9                                  END        CHKNML
 0000 ERR COUNT
```

APPENDIX D

DEBUGGING ASSEMBLY LANGUAGE PROGRAMS

There are two ways to debug and correct programs written in assembly language. One is by using the Debugger (see the Utility Programs manual); the other is by reading and interpreting the contents of memory through a memory dump (which can be obtained with the disk/memory transfer utilities, also described in the Utility Programs manual).

DEBUGGER

This utility program is intended for use during program development phases as a tool for program testing and error detection.

The Debugger operates in interactive mode, maintaining a dialogue with the console operator. It gives him visibility of all memory locations and addressable registers, and the ability to modify the contents of either. The ability to perform memory searches is provided, as well as the ability to display memory areas in both hexadecimal and ASCII notations.

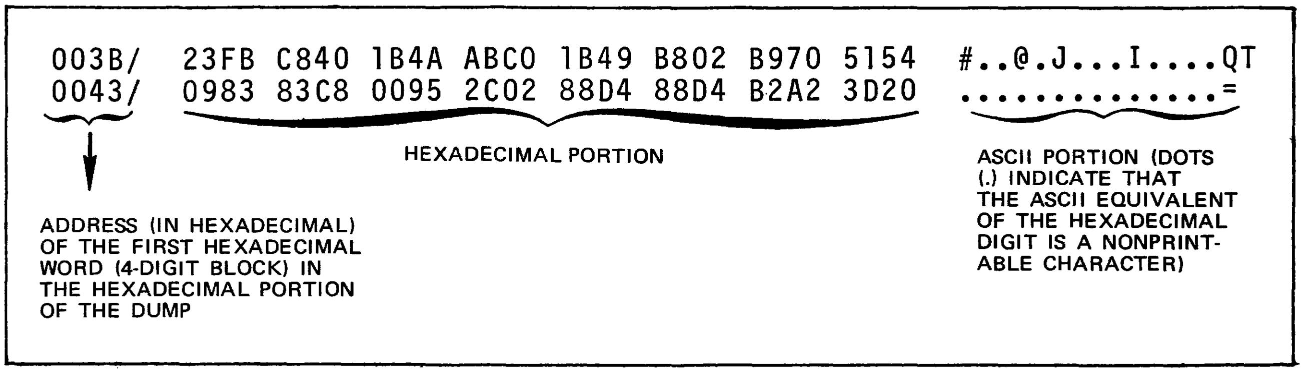
See the Utility Programs manual for a detailed description of the Debugger.

READING AND INTERPRETING MEMORY DUMPS

The remainder of this appendix describes how to read and interpret the contents of memory as they appear in a memory dump (see Figure D-1).

It is possible to interpret the hexadecimal portion of the dump illustrated in Figure D-1, as follows:

1. Since the ASCII portion of the dump shows no meaningful data, it is apparent that the assembler language program contains no string constants in the locations illustrated. Therefore, the hexadecimal digits probably represent assembly language instructions.
2. Break each word down into its binary equivalent. For example, C840 in location 003C becomes 1100 1000 0100 0000.
3. Using Table A-1, we find that C indicates that the instruction is probably a double operand (DO) instruction.
4. Continuing to use Table A-1, we find that the 8 plus a binary 0 in the eighth bit position indicates that the instruction is probably LDR.



```
003B/  23FB C840 1B4A ABC0 1B49 B802 B970 5154  #..@.J...I....QT
0043/  0983 83C8 0095 2C02 88D4 88D4 B2A2 3D20  ...............=
```

HEXADECIMAL PORTION
ADDRESS (IN HEXADECIMAL) OF THE FIRST HEXADECIMAL WORD (4-DIGIT BLOCK) IN THE HEXADECIMAL PORTION OF THE DUMP
ASCII PORTION (DOTS (.) INDICATE THAT THE ASCII EQUIVALENT OF THE HEXADECIMAL DIGIT IS A NONPRINTABLE CHARACTER)

Figure D-1. ASCII/Hexadecimal Memory Dump

5. By checking the table under "Assembly Language Internal Formats by Type" in Appendix A, it is possible to interpret the contents of the binary representation illustrated in step 2, above. That is, bits 1-3 identify the first operand register; in this case $R4 (the LDR instruction requires that the first operand register be an R-register.
6. Then, using Table A-2, it is possible to interpret the contents of the address syllable portion of the binary data shown in step 2; i.e., 100 0000. Using the table, the binary data corresponds to the columns as follows: mmmirrr. Thus, mmm = 100, i = 0, and rrr = 000. In that block, we find that the second operand is in the form of a location label.
7. We now know that the instruction is: LDR $R4,label. Thus, the address expression is the P + Displacement form of addressing.
8. Checking the description of that form of addressing in Section 5 (see "Addressing Techniques"), we see that the displacement between the address of this instruction plus 1 and the address of the label is loaded into the next consecutive word (i.e., location 003D). In this dump, the displacement is 1B4A.
9. The effective address of the data to be loaded into $R4 is in location 1B86 (i.e., (3C ± 1) ± 1B4A)).

Following is a complete list, by address, of the instructions shown in Figure D-1. You can perfect your ability to read memory dumps by interpreting the dump and comparing your results to those listed below. The procedure, until you become proficient, is basically as described above. After you have had the opportunity to read and interpret dumps several times, many of the steps can be skipped, as you will be able to interpret the data without checking all of the tables and descriptions identified above. As you can see by the nine steps described above, it is imperative that you understand the addressing techniques described in Section 5 (including how they are stored in memory), and how to interpret the address syllable.

Location	*Instruction/Meaning*
003B	Has no meaning in the context in which it appears; it is probably an address associated with the instruction in location 003A.
003C	LDR $R4,label
003D	Displacement between this location and the location containing the label identified in the LDR instruction.
003E	LAB $B2,label
003F	Displacement between this location and the location containing the label identified in the LAB instruction.
0040	LDR $R3,$B2
0041	CMR $R3,='QT'
0042	Value to be compared to the contents of $R3 in the CMR instruction.
0043	BNE $B3
0044	JMP *label
0045	Displacement between this location and the location containing the effective address (see "Indirect P-Relative Addressing" in Section 5).
0046	LDV $R2,X'20'
0047	DEC =$R4
0048	DEC =$R4
0049	LLH $R3,*B2.$R2
004A	CMV $R3,X'20'

APPENDIX E

SOURCE CODE ERROR NOTIFICATION BY ASSEMBLER

Columns 1 through 4 of the Assembler listing can contain up to four alphabetic characters (flags) which indicate possible errors in the source language statement. Columns 5-10 contain a six-digit decimal number corresponding to a sequential count of the source statements read. The error flags that can be produced by the Assembler are as follows:

FLAG	MEANING
A	Operand field format error
C	Numeric conversion error
D	Out of range short displacement
E	Illegal address expression
F	Illegal forward reference
H	Improper header
L	Label field format error
M	Multiply-defined symbol
N	No matching left parenthesis
O	Illegal operation code
P	Assembler control statement operand error
Q	Address <0 or $\geqslant 32K$
R	Illegal register reference
S	Improper statement format
T	Truncation warning constant/string constant
U	Undefined symbol
X	Expression too complex
Z	Conditional assembly error

APPENDIX F

SOURCE CODE ERROR NOTIFICATION BY MACRO PREPROCESSOR

The Macro Preprocessor issues error flags for nonfatal errors in the source code only if the IC argument was specified in the load command to the Command Processor. (See "Input to Command Processor Before Macro Preprocessor is Loaded" in the Program Development Tools manual.) If the IC argument was specified, each statement that contains a nonfatal error appears in the expanded source module as a comment and is preceded by the appropriate error flag(s).

An error flag is an alphabetic character that denotes the cause of an error. There can be up to four error flags per statement; subsequent errors are not designated. In a listing, column 1 contains an asterisk, columns 2 through 5 contain the error flag(s), column 6 is blank, and subsequent columns contain the source statement and other pertinent information. Error flags that can be produced by the Macro Preprocessor are listed below.

Error Flag	*Meaning*
A	Operand field format error
C	Numeric conversion error
E	Illegal expression
I	Invalid macro routine, MAC statement, or ENDM statement
J	Macro function error
L	Label field format error
M	Multiple inline macro routines were assigned the same name
N	No matching left parenthesis
O	Illegal operation code
S	Improper statement format
T	Truncation warning
V	Variable/parameter error in macro call or MAC statement
X	Expression too complex
Z	Conditional processing error

APPENDIX G

RESERVED SYMBOLIC NAMES

The following is an alphabetic list of all symbolic names (labels and identifiers) that have been defined within the BES Assembler and may not be redefined by the user.

Reserved Symbolic Name	**Definition**
$	Current location
$AF	Address format
$B1,$B2,..$B7	Base registers 1 through 7
$IV	Interrupt vector for current priority level
$M1,$M2,...$M7	Mode control registers 1 through 7
$R1,$R2,...$R7	General registers 1 through 7; index registers 1 through 3
$S1,$S2,$S3	Scientific registers 1 through 3

All reserved symbols added to future versions of Level 6 Assemblers will begin with a dollar sign ($). It is therefore recommended that user-defined labels not begin with $.

INDEX

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 6) GCOS/BES2 ASSEMBLY LANGUAGE	ORDER NO.	AU43, REV. 0
		DATED	JULY 1976

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. ☐

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

CUT ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

16047, 2876, Printed in U.S.A.

AU43, Rev. 0