

Honeywell



LEVEL 6

SOFTWARE

**GCOS 6
ASSEMBLY
LANGUAGE
REFERENCE**

SERIES 60 (LEVEL 6)
GCOS 6 ASSEMBLY LANGUAGE
REFERENCE

SUBJECT

Detailed description of Series 60 (Level 6) GCOS 6 Assembly Language including:

- Central Processor Unit (CPU) Instructions
- Scientific Instruction Processor (SIP) Instructions
- Commercial Processor Instructions
- Assembler Control Statements
- Macro Control Statements and Macro Calls

SPECIAL INSTRUCTIONS

This manual supersedes CB07, Rev. 0 dated January 1978. Change bars indicate new and changed information; asterisks denote deletions.

SOFTWARE SUPPORTED

This publication supports Release 0110 of the Series 60 (Level 6) GCOS 6 MOD 400 Operating Systems; see the Manual Directory of the latest GCOS 6 MOD 400 *System Concepts* manual (Order No. CB20) for information as to later releases supported by this manual.

ORDER NUMBER

CB07, Rev. 1

June 1978

Honeywell

Preface

This manual describes the GCOS 6 assembly language, a machine-oriented language for writing programs to execute on the Series 60 (Level 6) models. In this manual, unless stated otherwise, the term GCOS refers to the GCOS 6 software; the term Level 6 refers to the Series 60 (Level 6) on which the described software is executed.

Where appropriate, the actions performed by the GCOS Assembler as it processes elements of the assembly language are also discussed. In this manual, the term assembly language includes both Assembler control statements and assembly language instructions.

Section 1 describes the data representation and the hardware registers. Section 2 describes the basic elements of the GCOS assembly language, and Section 3 describes the considerations the programmer must make when writing a source program. Sections 4 and 5 describe, in detail, the Assembler control statements and assembly language instructions, respectively. Section 6 and Section 7 consist of detailed descriptions of the commercial instructions and the scientific instructions. The macro facility is described in Section 8. 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 lists the flags that may be issued by the Assembler. Appendix F lists the error flags that may be issued by the Macro Preprocessor. Appendix G contains a list of reserved symbolic names. Appendix H provides reference information for Commercial Processor operation. Appendix J and Appendix K provide reference information for queue instructions and stack instructions, respectively.

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; the letters a through f are considered equivalent to the corresponding uppercase letters).
- 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 (lowercase letters that represent keywords, however, are considered equivalent to the corresponding uppercase letters):

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

Users of the Writable Control Store feature should refer to the Writable Control Store User's Guide for information about the WCS instructions and the WCS Assembler, and to the GCOS 6 MOD 400 Operator's Guide for command information to invoke the WCS Loader.

MANUAL DIRECTORY

The following publications comprise the GCOS 6 manual set. The Manual Directory in the latest *GCOS 6 MOD 400 Systems Concepts* manual (Order No. CB20) lists the current revision number and addenda (if any) for each manual in the set.

<i>Order No.</i>	<i>Manual Title</i>
— CB01	<i>GCOS 6 Program Preparation</i>
— CB02	<i>GCOS 6 Commands</i>
— CB03	<i>GCOS 6 Communications Processing</i>
— CB04	<i>GCOS 6 Sort/Merge</i>
— CB05	<i>GCOS 6 Data File Organizations and Formats</i>
— CB06	<i>GCOS 6 System Messages</i>
— CB07	<i>GCOS 6 Assembly Language Reference</i>
— CB08	<i>GCOS 6 System Service Macro Calls</i>
CB09	<i>GCOS 6 RPG Reference</i>
CB10	<i>GCOS 6 Intermediate COBOL Reference</i>
— CB20	<i>GCOS 6 MOD 400 System Concepts</i> ✓
CB21	<i>GCOS 6 MOD 400 Program Execution and Checkout</i>
— CB22	<i>GCOS 6 MOD 400 Programmer's Guide</i>
— CB23	<i>GCOS 6 MOD 400 System Building</i>
CB24	<i>GCOS 6 MOD 400 Operator's Guide</i>
CB25	<i>GCOS 6 MOD 400 FORTRAN Reference</i>
CB26	<i>GCOS 6 MOD 400 Entry-Level COBOL Reference</i>
— CB27	<i>GCOS 6 MOD 400 Programmer's Pocket Guide</i>
CB28	<i>GCOS 6 MOD 400 Master Index</i>
CB30	<i>Remote Batch Facility User's Guide</i>
CB31	<i>Data Entry Facility User's Guide</i>
CB32	<i>Data Entry Facility Operator's Quick Reference Guide</i>
— CB33	<i>Level 6/Level 6 File Transmission Facility User's Guide</i>
CB34	<i>Level 6/Level 62 File Transmission Facility User's Guide</i>
CB35	<i>Level 6/Level 64 (Native) File Transmission Facility User's Guide</i>
CB36	<i>Level 6/Level 66 File Transmission Facility User's Guide</i>
CB37	<i>Level 6/Series 200/2000 File Transmission Facility User's Guide</i>
CB38	<i>Level 6/BSC 2780/3780 File Transmission Facility User's Guide</i>
CB39	<i>Level 6/Level 64 (Emulator) File Transmission Facility User's Guide</i>
CB40	<i>IBM 2780/3780 Workstation Facility User's Guide</i>
CB41	<i>HASP Workstation Facility User's Guide</i>
CB42	<i>Level 66 Host Resident Facility User's Guide</i>
CB43	<i>Terminal Concentration Facility User's Guide</i>

In addition, the following documents provide general hardware information:

<i>Order No.</i>	<i>Manual Title</i>
AS22	<i>Honeywell Level 6 Minicomputer Handbook</i>
AT04	<i>Level 6 System and Peripherals Operation Manual</i>
AT97	<i>MLCP Programmer's Reference Manual</i>
FQ41	<i>Writable Control Store User's Guide</i>

Contents

Section 1. Introduction

Assembly Languages	1-1
Level 6 Data Representation	1-1
Signed Integer Data	1-2
Unsigned Data	1-3
Floating-Point Data	1-4
Hardware Registers	1-4
Address Registers	1-4
Base Address (Bn) Registers	1-4
Program Counter (P-Register)	1-4
Remote Descriptor Base Register (RDBR)	1-4
Stack Register (T)	1-5
General (Rn) Registers	1-5
Mode (M) Registers	1-5
System Status (S) Register	1-5
Indicator (I) Register	1-7
Scientific Information Processor (SIP) Registers	1-7
Scientific Accumulator (Sn) Registers	1-7
Scientific Indicator (SI) Register	1-8
SIP Mode (M4) Register	1-8
SIP Trap Mask (M5) Register	1-9
Software Simulation of the Scientific Instruction Processor	1-9
Commercial Processor Registers	1-9
Commercial Processor Mode Register	1-9
Commercial Processor Indicator Register	1-10
Software Simulation of the Commercial Processor	1-10
Initialization and Modification of M-Registers	1-11

Section 2. Elements of Assembly Language

Mnemonic Codes	2-1
Symbolic Names	2-1
Identifiers	2-2
Labels	2-2
User-Defined Labels	2-2
Reserved Labels	2-3
Constants	2-4
String Constants	2-5
ASCII String Constants	2-5
Hexadecimal String Constants	2-5
Bit String Constants	2-6
Truncation/Padding of String Constants	2-6
Arithmetic Constants	2-7
Binary Integer Constants	2-7
Binary Integer Constants in Decimal Notation	2-7

Binary Integer Constants in Hexadecimal Notation	2-7
Decimal Integer Constants	2-7
Unpacked Decimal Integers	2-8
Packed Decimal Integers	2-8
Examples of Decimal Integers	2-9
Fixed-Point Constants	2-9
Floating-Point Constants	2-10
Normalization	2-11
Expressions	2-11
Evaluating Expressions	2-13
Location and Value Expressions	2-13
Value Expressions	2-13
Internal Value Expressions	2-13
External Value Expressions	2-14
Location Expressions	2-15
Internal Location Expressions	2-15
External Location Expressions	2-15
Common Location Expressions	2-16
Address Expressions	2-17
References	2-17

Section 3. Programming Considerations

Assembly Language Source Statement Formats	3-1
Order of Statements in Source Program	3-2
Calling System Services	3-2
Calling External Procedures	3-2
Alternate Method of Handling Input/Output and File Manipulation	3-2
Assembler	3-2
Cross-Reference Listing	3-3
SAF/LAF Considerations	3-3
Reentrancy Considerations	3-3

Section 4. Assembler Control Statements

Assembly-Controlling Statements	4-1
List-Controlling Statements	4-1
Data-Defining Statements	4-1
Storage-Allocation Statements	4-2
Symbol-Defining Statements	4-2
Program-Linking Statements	4-2
Conditional Assembly Control Statements	4-2
Operation Code-Defining Statement	4-2
Assembler Control Statements	4-2
ARGLST	4-3
BORG	4-4
BTEXT	4-5
CALL	4-6

CALL2	4-7
CLST	4-8
COMM	4-9
CTRL	4-10
DC	4-11
DEFGEN	4-12
EDEF	4-13
END	4-14
EQU	4-15
FAIL	4-16
IF	4-17
LCOMM	4-18
LIST	4-19
ΔLIST	4-19
NLST	4-20
NULL	4-21
ORG	4-22
PTRAY	4-23
RESV	4-24
TEXT	4-25
TITLE	4-26
XDEF	4-27
XLOC	4-28
XVAL	4-29

Section 5. Assembly Language Instructions

Arithmetic Operations	5-1
Boolean Operations	5-2
Branch Operations	5-2
Compare Operations	5-2
Control Operations	5-2
Input/Output Operations	5-2
Load Operations	5-2
Memory Management Operations	5-3
Modify Operations	5-3
Move Operations	5-3
Queue Operations	5-3
Shift Operations	5-3
Stack Operations	5-3
Store Operations	5-3
Swap Operations	5-3
Assembly Language Instruction Types	5-4
Branch-on-Indicator (BI) Instructions	5-4
Branch-on-Register (BR) Instructions	5-4
Double Operand (DO) Instructions	5-4
Generic (GE) Instructions	5-5
Input/Output (IO) Instructions	5-5
Shift (SHS and SHL) Instructions	5-5
Short-Value-Immediate (SI) Instructions	5-6
Single Operand (SO) Instructions	5-6
Addressing Techniques	5-7
Register Addressing	5-7
Immediate Memory Addressing (IMA)	5-8
Direct Immediate Memory Addressing	5-8
Indirect Immediate Memory Addressing	5-9
Indexed Direct Immediate Memory Addressing	5-9
Indexed Indirect Immediate Memory Addressing	5-10

Immediate Operand Addressing	5-10
P-Relative Addressing	5-12
Direct P-Relative Addressing	5-12
Indirect P-Relative Addressing	5-13
B-Relative Addressing	5-13
Direct B-Relative Addressing	5-14
Indirect B-Relative Addressing	5-15
Indexed Direct B-Relative Addressing	5-15
Indexed Indirect B-Relative Addressing	5-16
Direct B-Relative Plus Displacement Addressing	5-16
Indirect B-Relative Plus Displacement Addressing	5-18
Direct B6-Relative Plus Local Common Block Plus Displacement Addressing	5-18
Indirect B6-Relative Plus Local Common Block Plus Displacement Addressing	5-19
B-Relative Push Addressing	5-20
B-Relative Pop Addressing	5-21
Indexed B-Relative Push Addressing	5-21
Indexed B-Relative Pop Addressing	5-22
Short Displacement Addressing	5-23
Specialized Address Expression	5-24
Interrupt Vector Addressing	5-24
Indexed Addressing Considerations	5-25
Establishing a Multiplication Factor	5-26
AID, SID, LDI, and SDI Instructions	5-26
B-Register Instructions in LAF Configuration	5-26
Scientific Instructions	5-26
Bit/Byte Manipulating Instructions	5-26
Assembly Language Instructions	5-27
ACQ	5-27
ADD	5-28
ADV	5-29
AID	5-30
AND	5-31
ANH	5-32
ASD	5-33
B	5-34
BAG	5-35
BAGE	5-36
BAL	5-37
BALE	5-38
BBF	5-39
BBT	5-40
BCF	5-41
BCT	5-42
BDEC	5-43
BE	5-44
BEVN	5-45
BEZ	5-46
BG	5-47
BGE	5-48
BGEZ	5-49

Indirect P-Relative Addressing	6-8
Commercial Processor B-Relative Addressing	6-9
Commercial Processor Direct B-Relative Plus Displacement Addressing	6-9
Commercial Processor Indirect B-Relative Plus Displacement Addressing	6-9
Commercial Processor Direct B-Relative Plus Displacement With Indexing Addressing	6-10
Commercial Processor Indirect B-Relative Plus Displacement With Indexing Addressing	6-10
Immediate Operand (IMO) Addressing ..	6-11
Micro Edit Functions	6-12
Edit Insertion Table	6-13
Edit Flags	6-14
Change Edit Insertion Table (CHT) Micro Operation	6-14
End Floating Suppression (ENF) Micro Operation	6-15
Ignore Source Character (IGN) Micro Operation	6-16
Insert Asterisk on Suppress (INSA) Micro Operation	6-16
Insert Blank on Suppress (INSB) Micro Operation	6-16
Insert Multiple Characters (INSM) Micro Operation	6-16
Insert Character on Negative (INSN) Micro Operation	6-16
Insert Character on Positive (INSP) Micro Operation	6-17
Move with Float Currency Symbol Insertion (MFLC) Micro Operation	6-17
Move with Float Sign Insertion (MFLS) Micro Operation	6-17
Move Source Character (MVC) Micro Operation	6-18
Move with Zero Suppression and Asterisk Replacement (MVZA) Micro Operation	6-18
Move with Zero Suppression and Blank Replacement (MVZB) Micro Operation	6-18
Set Edit Flags (SEF) Micro Operation ..	6-19
Commercial Processor Traps	6-20
Trap 23 Unavailable Resource (UR)	6-21
Trap 24 Bus or Memory Error (BE)	6-21
Trap 25 Divide by Zero (DZ)	6-21
Trap 26 Illegal Specification (IS)	6-22
Trap 27 Illegal Character (IC)	6-22
Trap 28 Truncation (TR)	6-22
Trap 29 Overflow (OV)	6-22
Trap 30 Quality Logic Test (QLT) Error (QE)	6-22
Execution Details for Commercial Instructions	6-22
Detailed Descriptions of Commercial Instructions	6-23
ACM	6-24
ALR	6-25
AME	6-26
CBD	6-27

CBE	6-28
CBG	6-29
CBGE	6-30
CBL	6-31
CBLE	6-32
CBNE	6-33
CBNOV	6-34
CBNSF	6-35
CBNTR	6-36
CBOV	6-37
CBSF	6-38
CBTR	6-39
CDB	6-40
CSNCB	6-41
CSYNC	6-42
DAD	6-43
DCM	6-44
DDV	6-45
DLS	6-46
DMC	6-47
DME	6-48
DML	6-52
DRS	6-53
DSB	6-54
DSH	6-55
MAT	6-57
SRCH	6-58
VRFY	6-62

Section 7. Scientific Instructions

Scientific Traps	7-1
Scientific Instruction Processor (SIP) Programming Considerations	7-2
Detailed Descriptions of Scientific Instructions	7-2
SAD	7-2
SBE	7-4
SBEU	7-5
SBEZ	7-6
SBG	7-7
SBGE	7-8
SBGEZ	7-9
SBGZ	7-10
SBL	7-11
SBLE	7-12
SBLEZ	7-13
SBLZ	7-14
SBNE	7-15
SBNEU	7-16
SBNEZ	7-17
SBNPE	7-18
SBNSE	7-19
SBPE	7-20
SBSE	7-21
SCM	7-22
SCZD	7-23
SCZQ	7-24
SDV	7-25
SLD	7-26
SML	7-27
SNGD	7-28
SNGQ	7-29
SSB	7-30
SST	7-31
SSW	7-32

Section 8. Macro Facility

Order of Statements within a Source
 Program 8-1

Macro Routines 8-1

 Creating a Macro Routine 8-2

 MAC Macro Control Statement,
 without Parameters 8-2

 Contents of Macro Routine 8-2

 ENDM Macro Control Statement 8-3

 Specializing a Macro Routine by
 Parameter Substitution 8-3

 MAC Macro Control Statement,
 Including Parameters 8-4

 Protection Operators 8-5

 Situating Macro Routines 8-6

 LIBM Macro Control Statement 8-7

 INCLUDE Macro Control Statement 8-9

Macro Calls 8-11

 Nested Macro Call 8-12

 Recursive Macro Calls 8-13

Controlling Expansions 8-13

 Macro Variables 8-13

 Macro Substitution 8-14

 SETA Macro Control Statement 8-15

 SETN Macro Control Statement 8-16

 Conditional Macro Control Statements .. 8-17

 FAIL Macro Control Statement 8-17

 GOTO Macro Control Statement 8-18

 IF Macro Control Statement 8-19

 NULL Macro Control Statement 8-22

Macro Functions 8-23

 Format of Macro Functions 8-23

 Length Attribute Macro Function 8-23

 Type Attribute Macro Function 8-24

Hexadecimal Conversion Macro
 Function 8-25

 Index Macro Function 8-26

 Search Macro Function 8-27

 Substring Macro Function 8-28

 Translate Macro Function 8-29

 Vector Orientation Macro Function 8-30

 Verify Macro Function 8-31

Example Illustrating Macro Facility 8-31

Programming Considerations 8-34

 Initialized Values of Macro Variables 8-34

 Designating Numeric Values 8-35

 Designating Alphanumeric Values 8-35

 Alphanumeric Value Conventions 8-36

 Balanced Apostrophes 8-36

 Balanced Parentheses 8-36

 Commas and Semicolons 8-37

 Spaces and Horizontal Tabs 8-37

Appendix A. Programmer's Reference Information

Summary of Hardware Registers A-1

Assembly Language Internal Formats by
 Type A-4

Hexadecimal Representation of
 Instructions A-6

Valid Address Expressions A-10

Appendix B. Hexadecimal Numbering System

Decimal-to-Hexadecimal Conversion B-2

Hexadecimal-to-Decimal Conversion B-2

Hexadecimal-to-ASCII Conversion B-4

Hexadecimal Addition B-5

Hexadecimal Subtraction B-5

Hexadecimal Multiplication B-6

Hexadecimal Division B-6

Appendix C. Sample Assembly Language Program

Appendix D. Debugging Assembly Language Programs

Debug D-1

DumpEdit D-1

Reading and Interpreting Memory
 Dumps D-1

Appendix E. Notification Flags Issued by Assembler

Source Code Error Flags E-1

Statement Reference Flags E-1

Appendix F. Source Code Error Notification by Macro Preprocessor

Appendix G. Reserved Symbolic Names

Appendix H. Programmer's Reference Information for Commercial Processor Operation

Internal Formats of Commercial
 Processor Instructions H-1

Internal Format of Data Descriptors H-4

 Decimal Data Descriptors H-4

 Unpacked Decimals H-4

 Packed Decimals H-5

 Alphanumeric Data Descriptor H-6

 Binary Data Descriptor H-6

 Address Syllable H-7

Appendix J. Programmer's Reference Information for Queue Instructions

Appendix K. Programmer's Reference Information for Stack Instructions

Stack Frame K-1

Stack Instruction Formats K-2

 Load Stack Address Register (LDT) K-2

 Store Stack Address Register (STT) K-2

 Acquire Stack Frame (ACQ) K-2

 Relinquish Stack Frame (RLQ) K-2

Figures

1-1	Assembler Functions	1-1		6-9	Shift Instruction Formats	6-55
1-2	Level 6 Registers	1-6		8-1	Sample Unexpanded Source Module and Assembler Listing of Resulting Expanded Source Module	8-32
5-1	Direct Immediate Memory Addressing	5-8		A-1	Level 6 Hardware Registers	A-1
5-2	Indirect Immediate Memory Addressing	5-9		A-2	Internal Formats of Assembly Language Instructions	A-5
5-3	Indexed Direct Immediate Memory Addressing	5-10		C-1	Listing of CHKNML Program	C-1
5-4	Indexed Indirect Immediate Memory Addressing	5-11		C-2	Listing of Bubble Sort Program	C-3
5-5	Immediate Operand Addressing-Scientific Instruction	5-11		D-1	ASCII/Hexadecimal Memory Dump	D-2
5-6	Immediate Operand Addressing	5-12		H-1	Internal Formats of Commercial Processor Instructions	H-1
5-7	Direct P-Relative Addressing	5-12		H-2	Remote Descriptor Address Generation	H-4
5-8	Indirect P-Relative Addressing	5-13		H-3	Decimal Data Descriptor Format ..	H-4
5-9	Direct B-Relative Addressing	5-14		H-4	Alphanumeric Data Descriptor Format	H-6
5-10	Indirect B-Relative Addressing	5-15		H-5	Binary Data Descriptor Format	H-6
5-11	Indexed Direct B-Relative Addressing	5-16		H-6	Commercial Processor Address Syllable Format	H-7
5-12	Indexed Indirect B-Relative Addressing	5-17		H-7	Commercial Processor Hardware Test Program	H-8
5-13	Direct B-Relative Plus Displacement Addressing	5-17		J-1	Queue Management	J-2
5-14	Indirect B-Relative Plus Displacement Addressing	5-18		K-1	Stack Structure	K-1
5-15	Direct B6-Relative Plus Local Common Block Plus Displacement Addressing	5-19				
5-16	Indirect B6-Relative Plus Local Common Block Plus Displacement Addressing	5-20				
5-17	B-Relative Push Addressing	5-20				
5-18	B-Relative Pop Addressing	5-21				
5-19	Indexed B-Relative Push Addressing	5-22				
5-20	Indexed B-Relative Pop Addressing	5-23				
5-21	Short Displacement Addressing	5-23				
5-22	Specialized Address Expressions	5-24				
5-23	Interrupt Vector Addressing	5-25				
5-24	VLD Instruction Operations	5-149				
6-1	Commercial Processor Direct P-relative Addressing	6-6		2-1	Defining Symbolic Names	2-3
6-2	Commercial Processor Indexed Direct P-relative Addressing ..	6-7		2-2	Rules of Truncation/Padding String Constants	2-6
6-3	Commercial Processor Indirect P-relative Addressing	6-8		5-1	Indexed Addressing Modes	5-25
6-4	Commercial Processor Direct and Indirect B-Relative Plus Displacement Addressing	6-10		6-1	Micro Operations for Edit Instructions	6-13
6-5	Commercial B-Relative Plus Displacement With Indexing Addressing	6-11		6-2	Edit Insertion Table at Initialization	6-13
6-6	Commercial Processor IMO Addressing	6-12		6-3	Edit Flags for Micro Operations	6-14
6-7	Flow Diagram of SEF Micro Operation	6-19		6-4	Code for Replacing EIT Entries	6-15
6-8	Trap Context	6-21		6-5	Character Insertion by MFLS Micro Operation	6-18
				6-6	Commercial Processor Trap Vectors and Events	6-21
				7-1	Trap Vectors and Events	7-1
				A-1	Internal Representation of Assembly Language Instructions	A-6
				A-2	Address Syllables for CPU & SIP Instructions	A-9

Tables

A-3	Summary of Valid Forms of Address Expressions for CPU and SIP Instructions	A-10
B-1	Comparison of Binary, Decimal, and Hexadecimal Symbols	B-1
B-2	Storage and Printout of Value 32	B-2
B-3	Hexadecimal/Decimal Conversion	B-3
B-4	Hexadecimal/ASCII Conversion	B-4
B-5	Hexadecimal Addition Table	B-5
B-6	Hexadecimal Multiplication Table	B-6
H-1	Commercial Instruction Summary	H-2
H-2	Commercial Processor Address Syllables	H-7

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 interprets only machine (i.e., object) code. Since object code is composed of binary digits, it is difficult to understand 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.

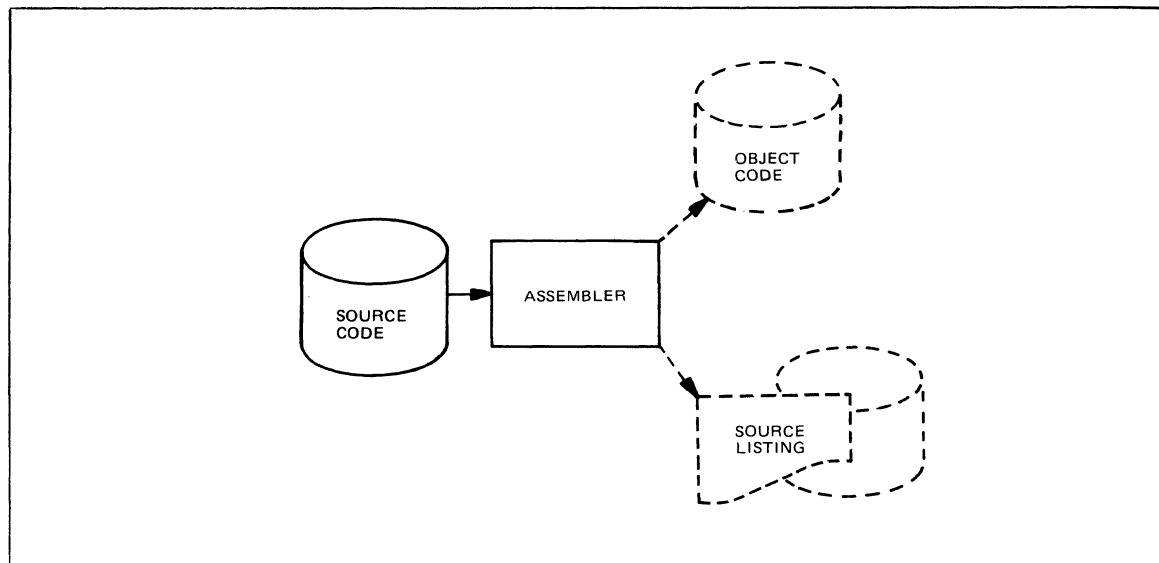


Figure 1-1. Assembler Functions

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:

- Bit — 1 bit
- 4 — bit digit
- Byte (half-word) — 8 bits
- Word — 16 bits
- 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.

*

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.

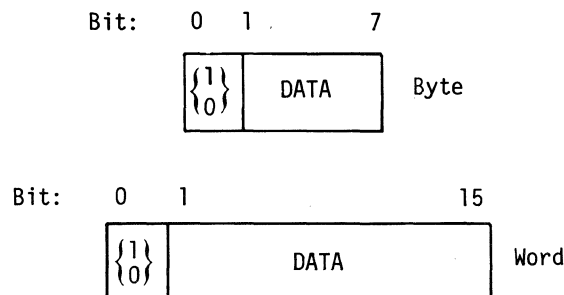
Data stored in memory can be in any of the following forms:

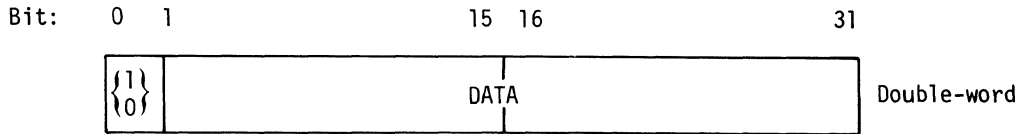
- Signed integer
- Unsigned integer
- Floating-point

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 scientific registers.

SIGNED INTEGER DATA

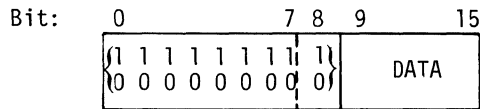
Signed integers stored in memory contain a sign (0 = +; 1 = -) in bit 0 and the data in the remaining bits. Negative numbers appear in twos-complement form. Byte, word, and double-word formats are permitted, as follows:





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 two's complement of the coded integer. For example, if the contents of a signed integer word appearing in memory are BDA0, the decimal equivalent is -16992.

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:

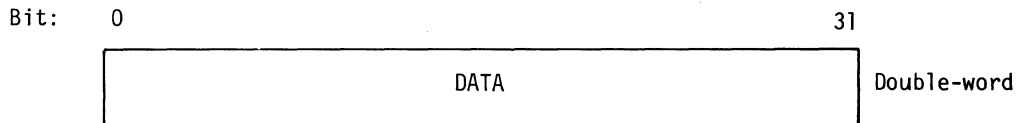
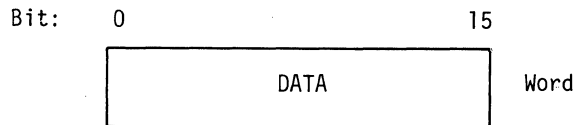
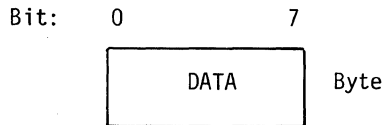


The sign of the integer byte (i.e., the first bit of the 8-bit byte), which is contained in bit 8 of the register, is extended through the first byte of the register.

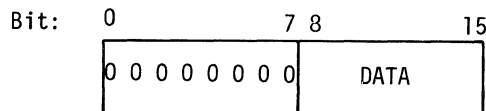
If the first byte of the register contains the hexadecimal value FF, the integer in the second byte is a negative value; if the first byte contains the hexadecimal value 00, the value of the second byte is positive.

UNSIGNED DATA

Unsigned data appears in memory in three possible formats:

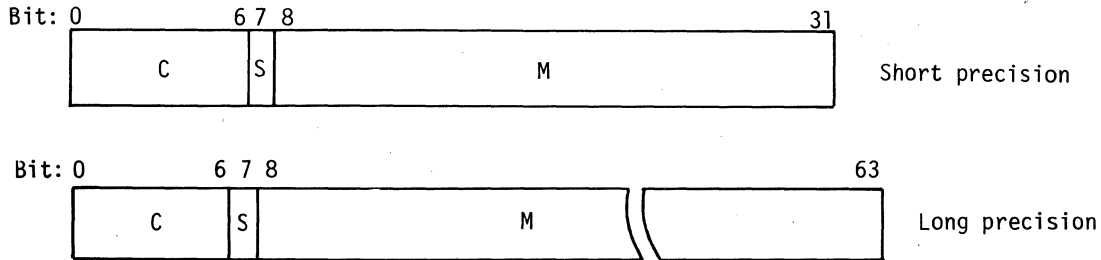


When an unsigned data byte is loaded from memory into a hardware general register, the byte is placed into register bits 8 through 15, and register bits 0 through 7 are set to 0, as follows:



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

Mantissa — a normalized hexadecimal fraction.

A floating-point constant in memory may be loaded into a scientific register or a software-simulated scientific register, described later in this section.

HARDWARE REGISTERS

Level 6 hardware registers (Figure 1-2) consist of word operand registers, address registers, control registers, and mode registers.

ADDRESS REGISTERS

The length of the address registers is 16 bits for 6/30 models and 20 bits for 6/40 and 6/50 models. (The 12 leftmost bits of the 32-bit LAF address in memory must be zero.)

BASE (B_n) REGISTERS

The seven base registers are used in the formulation of addresses by pointing to any procedure, data, or location in main memory. Typically, the base 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).

PROGRAM COUNTER (P-REGISTER)

The program counter (P-register) is used by the central processor to generate the effective address of data based on various operands in the assembly language instruction set. (See "Addressing Techniques" in Section 5.) The P-register contains the address of the next instruction only during execution of the current instruction. The address of the next instruction is the address of the current instruction plus the length of the current instruction. However, JMP, ENT, LNJ, and branch instructions modify the contents of the P-register to a jump or branch address.

REMOTE DESCRIPTOR BASE REGISTER (RDBR)

This register is used with Commercial Processor instructions and is available only on 6/40 and 6/50 models. See Appendix H "Programmer's Reference Information for Commercial Processor Operation."

STACK REGISTER (T)

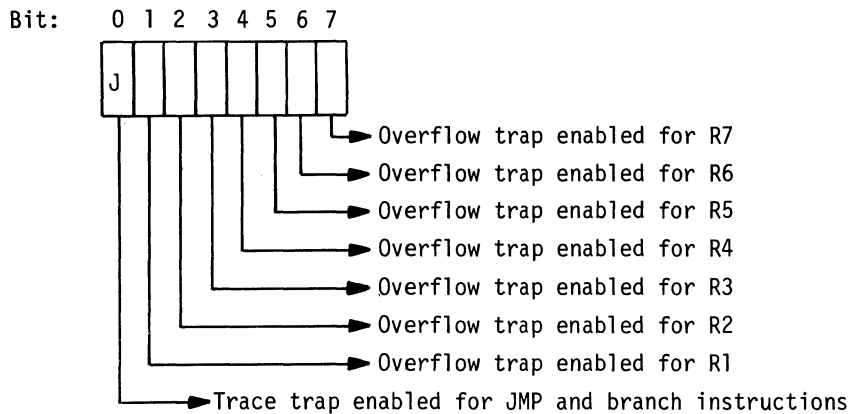
This register is used with stack instructions and is available only on 6/40 and 6/50 models. See Appendix K "Programmer's Reference Information for Stack Instructions."

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 (M) REGISTERS

Register M1 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 M1 register have the following meanings when set to binary 1:



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 *System Services Macro Calls* manual for a detailed description of trap handlers.

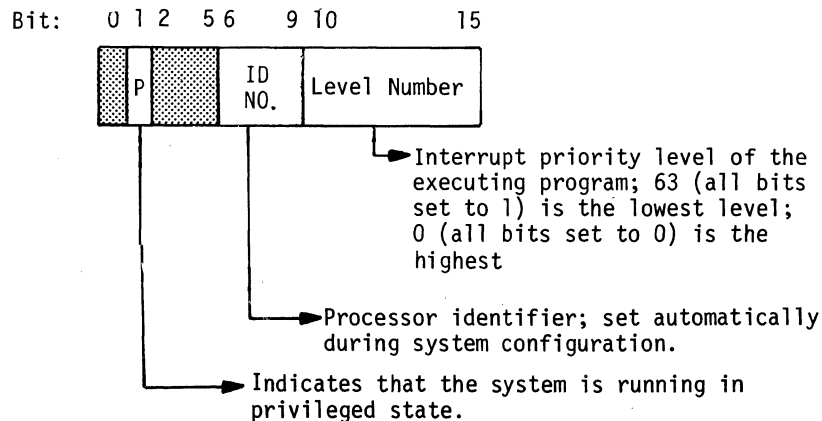
Registers M2, M6, and M7 are reserved for future use.

The format of the Commercial Processor control register M3 is the same as that of the Commercial Processor mode register which is described later in this section.

The formats of registers M4 and M5 are the same as those of the SIP mode register and the SIP trap mask register respectively. These registers are described later in this section.

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:



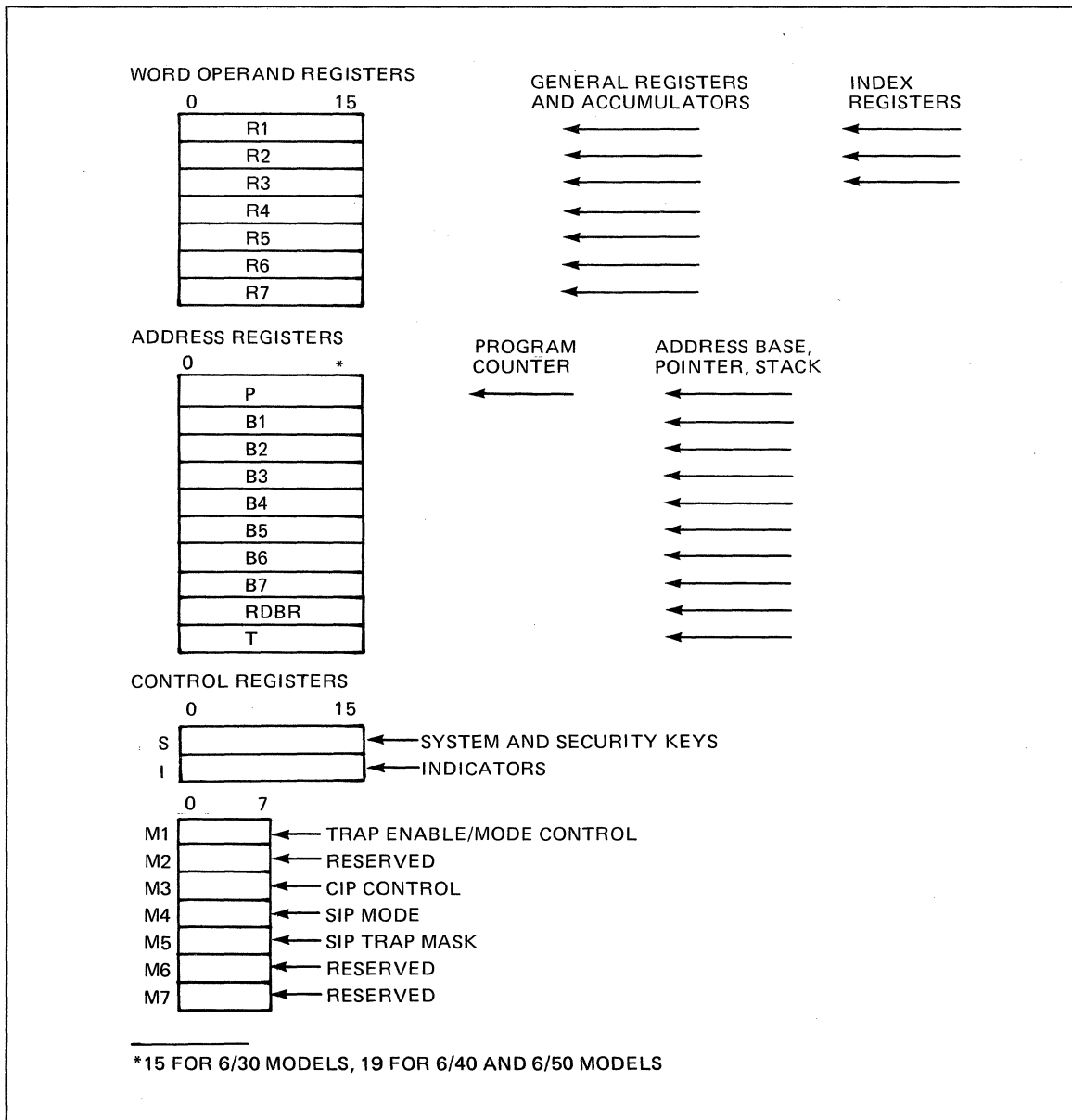
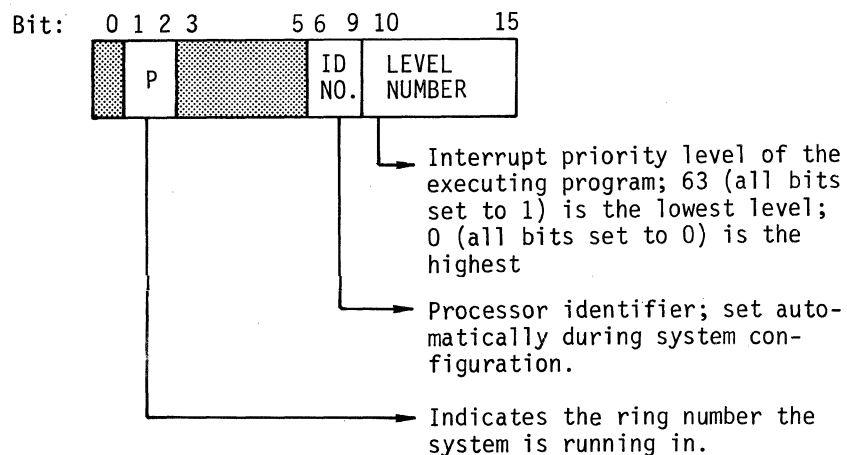


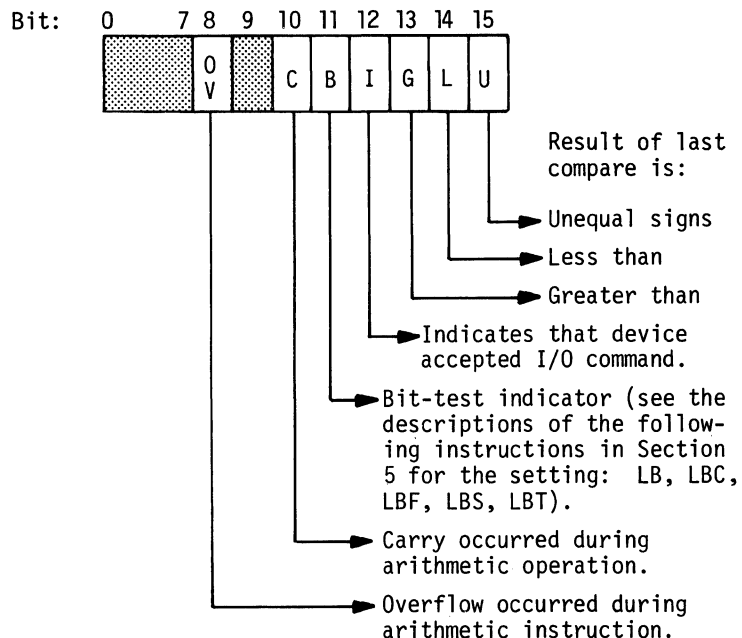
Figure 1-2. Level 6 Registers

If the hardware configuration includes a Memory Management Unit, the contents of the S-register have the following meaning:



INDICATOR (I) REGISTER

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



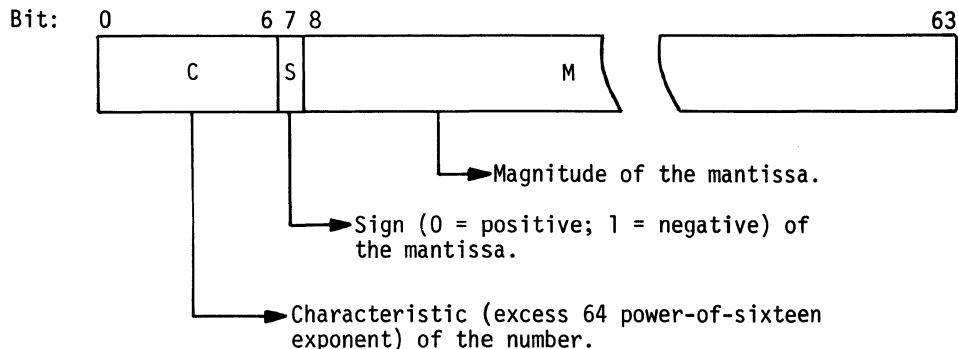
SCIENTIFIC INSTRUCTION PROCESSOR (SIP) REGISTERS

The Level 6 Scientific Instruction 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 (S_n) REGISTERS

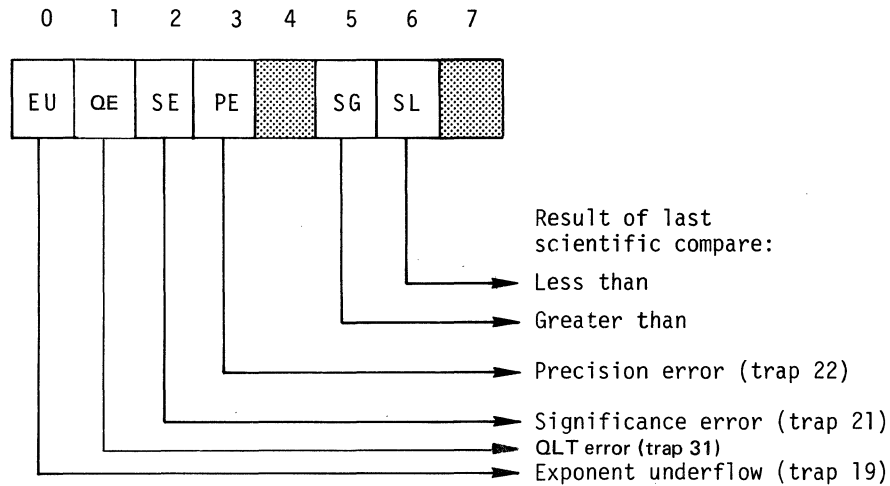
The SIP provides three 64-bit scientific accumulator registers for use in either short- or long-precision floating-point operations. When these registers are used in short-precision operations, only the high-order (leftmost) 32 bits participate.

The format of the scientific accumulator registers is shown below.



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:

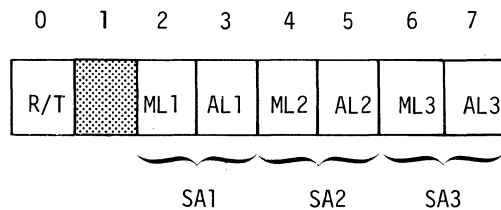


Traps and trap handlers are discussed in the *System Services Macro Calls* manual.

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 CP. Both versions are set to 0 upon CP initialization and both may be modified with an MTM instruction (see Section 5). If only the SIP is initialized, the CP copy of the register is not 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)

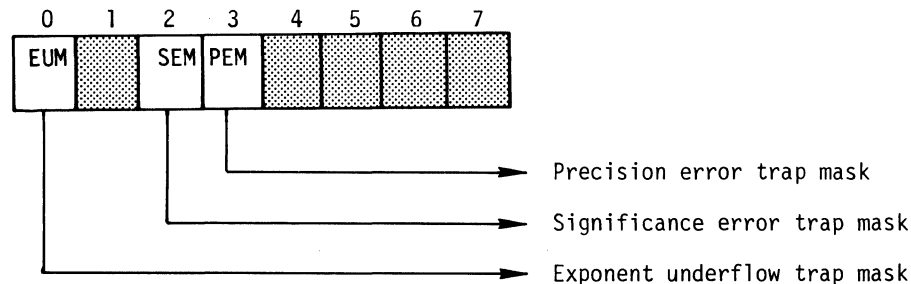
0 — Two words

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 CP. Both versions are set to 0 upon CP initialization and both may be modified with an MTM instruction (see Section 5). If only the SIP is initialized, the CP copy of the register is not cleared, and the contents of both versions must be reestablished with an MTM.

The format of the M5-register is as follows:



SOFTWARE SIMULATION OF THE SCIENTIFIC INSTRUCTION PROCESSOR

For systems on which a Scientific Instruction Processor (SIP) is not available, GCOS provides the equivalent SIP functions through software simulation. Two simulators are available: the Single-Precision SIP Simulator (SSIP) and the Double-Precision SIP Simulator (DSIP). If a configuration is to support scientific instructions when a SIP is not present, SSIP or DSIP must be specified in the CLM directive SYS. (See *System Building* manual.)

The DSIP simulates all functions of the SIP. The SSIP is a partial simulator. The simulators are entered via trap vector 3 (for scientific floating-point instructions) or trap vector 5 (for scientific branch instructions).

Note the following considerations with respect to the use of the SSIP.

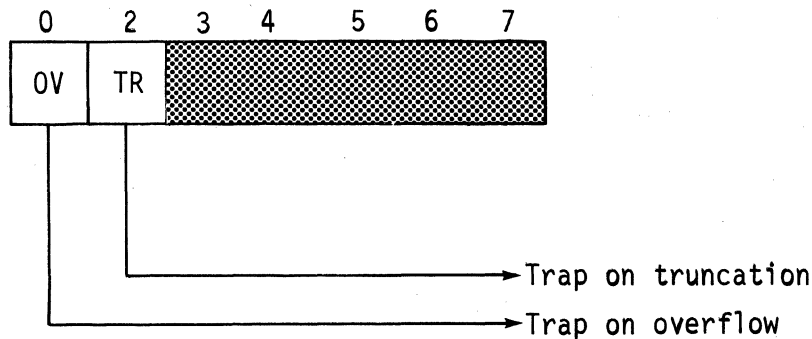
- SSIP uses registers R4, R5, and R7 to simulate a scientific register (assumed to be SA1). A task that executes scientific instructions that might be simulated by SSIP should dedicate these three registers to the use of the simulator.
- SSIP uses the CPU I-register to store the results of a scientific compare instead of simulating the scientific indicator register. Thus, if scientific compare instructions are to be simulated by SSIP (as opposed to being simulated by DSIP or executed by the SIP), then:
 - CPU branch instructions must be used to test the result of a scientific compare instead of the normal scientific branch instructions.
 - Execution of scientific instructions alter the CPU I-register instead of the SIP's SI register.
- On 6/30 systems, the SSIP does not support the MTM or STM instruction.
- SSIP rounds results when appropriate; DSIP truncates results unless otherwise instructed. Thus, results produced by the SSIP may not agree exactly with those produced by the DSIP.

COMMERCIAL PROCESSOR REGISTERS

The Commercial Processor, an optional hardware unit, contains two registers: the Commercial Processor mode register, and the Commercial Processor indicator register.

COMMERCIAL PROCESSOR MODE REGISTER

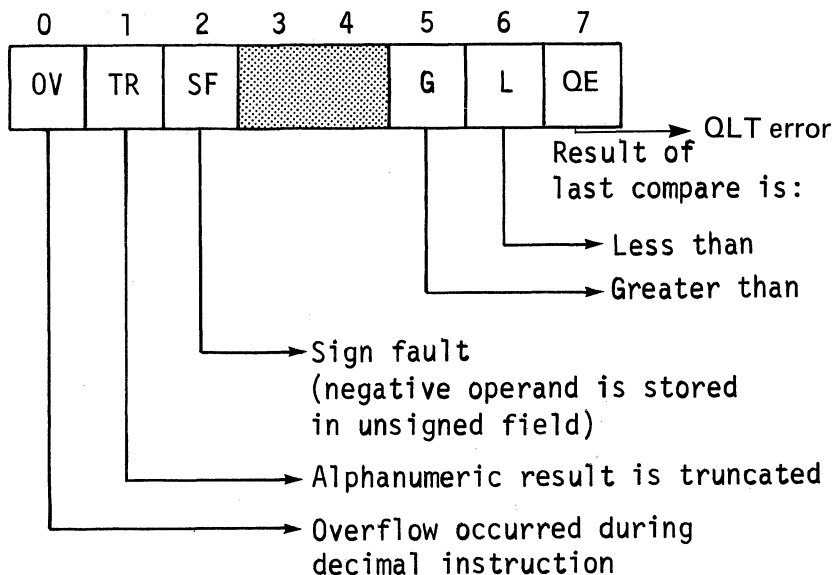
The 8-bit Commercial Processor mode register is a copy of the M3 register (in the CPU) which is provided for use with the Commercial Processor. Both are set to zero at initialization of the CPU. Both registers may be modified with an MTM instruction. If only the Commercial Processor is initialized, the M3 register is not cleared, and the contents of both registers must be established with an MTM instruction. The format of the Commercial Processor mode register and the M3 register is shown below. When set to binary 1, the bits have the following meanings:



Note that, although the contents of the Commercial Processor mode register is not saved, the equivalent information in the M3 register is saved or restored as a function of the mask bits in the interrupt save area. When a restore is done, the restored value is sent to the Commercial Processor by the CPU.

COMMERCIAL PROCESSOR INDICATOR REGISTER

The 8-bit Commercial Processor indicator register is cleared at initialization. During the execution of an instruction that affects the register, only the bits pertinent to the instruction are preset (set or reset). All other bits remain unchanged. During the execution of a branch instruction, all bits including the one being tested are left unchanged. When set to binary 1, the bits have the following meaning:



The contents of the Commercial Processor indicator register will be saved or restored as a function of the mask bits in the interrupt save area.

SOFTWARE SIMULATION OF THE COMMERCIAL PROCESSOR

For systems on which a Scientific Instruction Processor (SIP) is not available, GCOS provides the equivalent SIP functions through software simulation. Two simulators are available; the Single-Precision SIP Simulator (SSIP) and the Double-Precision SIP Simulator. If a configuration is to support scientific instructions when a SIP is not present, SSIP or DSIP must be specified in the CLM directive SYS. (See *System Building* manual.)

INITIALIZATION AND MODIFICATION OF M-REGISTERS

When each task starts, the operating system establishes the following default values for registers M1, M3, M4, and M5.

- M1 = 00 Trace trap and all R-register overflow traps disabled.
- M3 = 00 Commercial Processor overflow trap and truncation trap disabled; Commercial Processor is under direct CPU firmware control (i.e., not in software test mode).
- M4 = 03 Truncation mode is in effect, Scientific accumulators \$S1 and \$S2 and associated memory operands are two words long; \$S3 and associated memory operands are four words long.
- M5 = 20 Significance error trap enabled; exponent underflow and precision error traps disabled.

The contents of these registers can be modified by the assembly language instruction MTM.

Section 2

Elements of Assembly Language

The principal elements of Level 6 assembly language are:

- Mnemonic codes
- Symbolic names
- Constants
- 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 define constant data used by the program.
3. Macro call statements, which are interpreted by the Macro Processor to further define the source 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 six 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.
4. The lowercase alphabetic characters are considered equivalent to the corresponding uppercase characters.

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

- Identifiers — Reserved symbols designating the hardware registers, the scientific accumulators, and certain address syllables.
- 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 registers. 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:

- \$B1 through \$B7 — Base registers
- \$R1 through \$R7 — General registers
- \$R1 through \$R3 — Index registers
- \$M1 through \$M7 — Mode control registers
- \$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:

- 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.
- External location label — Refers to a location in another independently assembled or compiled program. It is defined by appearing in the operand list of an XLOC statement.
- 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 internal area normally allocated to the program. 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 and LCOMM statements 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.
- 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” later in this section) that yields a dimensionless value.
- 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.
- 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.

TABLE 2-1. DEFINING SYMBOLIC NAMES

Type	How Defined
Internal location label	Appears in label field of an assembly language instruction or Assembler control statement (except EQU, COMM, or LCOMM 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 or LCOMM statement; or appears in label field of an assembly language instruction or Assembler control statement (except EQU, COMM, or LCOMM 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 dimensionless 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.

RESERVED LABELS

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

- \$ — The Assembler maintains a location counter which contains the address of the next available object memory location. The symbol \$ represents this address and has the attribute of internal location or common location, depending on whether the program is currently "originated" in a non-common area or in a common block, respectively. The initial value of the location counter is location zero.

The Assembler also maintains a byte indicator which indicates whether the next available byte of object memory is the even (i.e., high order or left) or odd (i.e., low order or right) byte of the word whose address is contained in the location counter. The only statement that causes memory to be allocated that is permitted when the byte indicator is set to indicate the odd byte is the Byte Text (BTEXT) Assembler control statement. The byte indicator is initially set to indicate the even byte.

Normally, the location counter is incremented by the number of words required to store the object code resulting from a given statement *after* the statement has been processed. Normally, the byte indicator is not altered. Exceptions to this general rule are as follows:

- Assembler control statements, such as Equate and Common, that do not cause any memory to be allocated have no affect on either the location counter or the byte indicator.
- The Origin Assembler control statement which sets the location counter to a specified value and sets the byte indicator to indicate the even byte.
- The Byte Origin Assembler control statement which sets the location counter to a specified value and sets the byte indicator to indicate the odd byte.
- The Define Constants (DC) Assembler control statement which does not affect the byte indicator, but increments the location counter *after each* operand is processed. Thus, a \$ appearing as a label in an operand of a DC statement will always refer to the *first*, or only, word of memory allocated for that operand. Because of this, the DC statement:

```
DC 1,2,<$-2
```

will produce the same object code as the following DC statements:

```
DC 1
DC 2
DC <$-2
```

- The Byte Text Assembler control statement which increments the location counter and

byte indicator, concatenated to form a byte address, by the number of bytes allocated. Either, or both, the location counter and byte indicator may be altered.

—The Pointer Array (PTRAY) Assembler control statement which does not affect the byte indicator and increments the location counter *after each* operand is processed. Thus, a \$ appearing as a label in an operand of a PTRAY statement will always refer to the first, or only, word of memory allocated for that operand.

—The Argument List (ARGLST) Assembler control statement which does not affect the byte indicator and increments the location counter as follows:

—First the location counter is incremented by 1 after the control word is allocated, but *before* the first operand is processed.

—Then the location counter is incremented *after each* operand is processed.

Thus a \$ appearing as a label in an operand of an ARGLST statement will always refer to the first, or only, word of memory allocated for that operand.

—The Call (CALL and CALL2) Assembler control statements, which do not affect the byte indicator, but increment the location counter at various times, as appropriate for the breakdowns shown for these statements in Section 4.

—The Input/Output statements, which do not affect the byte indicator, but increment the location counter *after each* operand is processed. Thus a \$ appearing as a label in an operand of an Input/Output statement will always refer to the word of memory containing that operand's address syllable.

—The Commercial Processor nonbranch statements, which do not affect the byte indicator, but increment the location counter as follows:

—First the location counter is incremented by 1 after the op code word is allocated, but *before* the first operand is processed.

—Then the location counter is incremented *after each* operand is processed.

Thus a \$ appearing as a label in an operand of a nonbranch Commercial Processor instruction will always refer to the first, or only, word of memory allocated for that operand.

- \$AF — This label refers to the address mode requested by the user. \$AF is 1 for SAF, and 2 for LAF or SLIC.
- \$IV — Refers to the content 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 *System Services Macro Calls* manual.
- \$RZERO — Refers to relocatable location zero of the program. \$RZERO is an internal location label.
- \$SW — Refers to the current status of the external switches. The Assembler requests the value of the external switches from the operating system; \$SW is then defined to be this 16-bit value which corresponds to External Switch 0 through External Switch 15. The high order bit is switch 0. \$SW is an internal value label. Use of \$SW with conditional Assembler control statements provides a method of varying the assembly procedure without altering the assembly language source program. (See the *Commands* manual for a discussion of the Modify External Switches Command.)

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:

- String constants
- 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:

$$[(n)] \left[\left\{ \begin{array}{c} A \\ Z \\ B \end{array} \right\} \right] 'c...'$$

[(n)]

Specifies an optional decimal integer in the range from 1 to 255, which represents the replication factor (the string is concatenated to itself n-1 times).

$$\left[\left\{ \begin{array}{c} A \\ Z \\ B \end{array} \right\} \right]$$

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

'c...']

Identifies the character(s) in the string.

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; to include an apostrophe, a double apostrophe must be specified (i.e., "is interpreted as").

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:

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

The following examples illustrate how to specify ASCII string constants:

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

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. DATAA DATAA DATAA DATAA

HEXADECIMAL STRING CONSTANTS

A hexadecimal string constant is written as the letter Z followed by a string of characters representing any of the valid hexadecimal digits (i.e., 0 through 9 and A through F) enclosed within apostrophes.¹

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).

¹The lowercase letters a through f are considered equivalent to the corresponding uppercase letters.

The format of a hexadecimal string constant is as follows:

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

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

`Z'5449544C452053414D504C4531'`

This example translates into `TITLEA SAMPLE1` (see Appendix B).

BIT STRING CONSTANTS

A bit string constant is written as the letter B followed by a string of characters representing the 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'[b...]`

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 (i.e., string constants are left-justified). 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.
3. Double integer instructions (AID, LDI, SDI, and SID) require string constants or double precision fixed-point constants to fully define 32 bits (i.e., 2 words).

ARITHMETIC CONSTANTS

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

BINARY INTEGER CONSTANTS

Binary integer constants can be represented in decimal or hexadecimal notation. 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, a binary integer constant is assumed to be positive.

$$\begin{array}{l} \left[\begin{array}{l} + \\ - \end{array} \right] \left\{ \begin{array}{l} n[n\dots] \\ X'h[h\dots]' \end{array} \right\} \\ \left[\begin{array}{l} + \\ - \end{array} \right] \end{array}$$

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

n[n...]

Specify decimal digits.

h[h...]

Specify hexadecimal digits

Binary Integer Constants in Decimal Notation

A binary integer constant expressed in decimal notation is written as a character string composed of the decimal digits 0 through 9. The following examples illustrate valid binary integer constants in decimal notation.

1. 31764
2. +4652
3. -6781

Binary Integer Constants in Hexadecimal Notation

A binary integer constant expressed in hexadecimal notation is written as the letter X followed by a character string composed of the hexadecimal digits 0 through 9 and A through F (the lowercase letters a through f are considered equivalent to the corresponding uppercase letters) within apostrophes. The following examples illustrate binary integer constants in hexadecimal notation.

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

The decimal equivalent of these examples is +47, +32767 and -32768 respectively as can be determined by reference to Table B-3.

Decimal Integer Constants

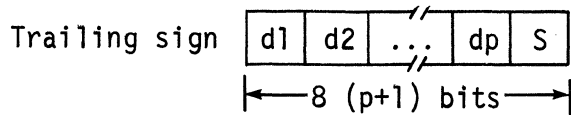
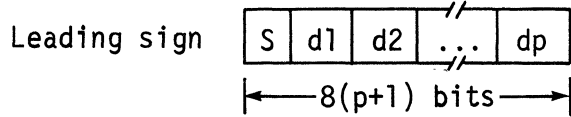
Decimal integer constants are represented by a letter from the set L,T,O,N,P,U followed by a character string enclosed in apostrophes. In general, they may be preceded by a plus (+) or minus (-) sign indicating a positive or negative value. The letter indicates whether the value is internally represented as a packed or unpacked number and designates the internal sign convention. The character string is composed of the digits 0 through 9. Decimal integer constants begin at a word boundary and occupy an integral number of words, possibly including trailing digits which may be unused.

Unpacked Decimal Integers

The prefix letter designating the internal sign convention and the range of values allowed for each convention of unpacked decimal integers are as follows:

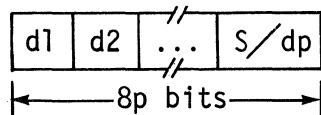
Sign Convention	Letter	Range of Values
Leading separate	L	$-10^{30} < n < +10^{30}$
Trailing separate	T	$-10^{30} < n < +10^{30}$
Trailing overpunch	O	$-10^{31} < n < +10^{31}$
Unsigned	N	$0 \leq n < 10^{31}$

The storage formats for separate signed unpacked decimal integers are as follows:



In these formats, dn is the ASCII representation of a decimal digit, S indicates the sign, and p indicates the precision, which must be greater than zero and less than 32. The plus sign is represented by the ASCII character + (hexadecimal 2B) the minus sign by the ASCII character - (hexadecimal 2D).

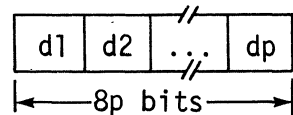
The format of an unpacked decimal integer with the sign indicated by a trailing overpunch is as follows:



The rightmost character in storage depends on the least significant digit of the integer and on whether the integer is positive or negative as shown below.

		Least Significant Digit									
		0	1	2	3	4	5	6	7	8	9
Positive	ASCII graphic	{	A	B	C	D	E	F	G	H	I
	Hexadecimal code	7B	41	42	43	44	45	46	47	48	49
Negative	ASCII graphic	}	J	K	L	M	N	O	P	Q	R
	Hexadecimal code	7D	4A	4B	4C	4D	4E	4F	50	51	52

The format of an unsigned unpacked decimal integer is as follows:

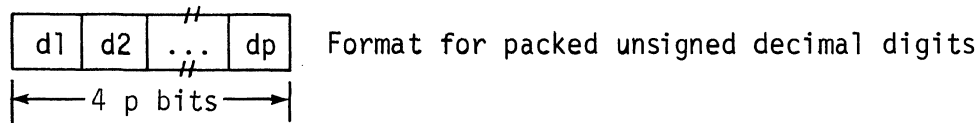
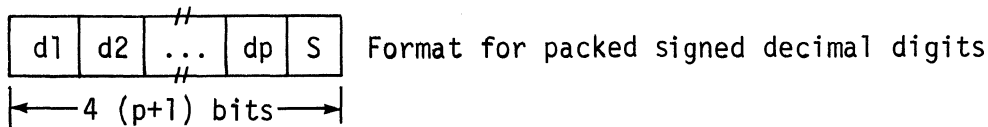


Packed Decimal Integers

The prefix letter and the range of values for signed and unsigned packed decimal integers are as follows:

Prefix Letter	Type	Range
P	Signed	$-10^{30} < n < +10^{30}$
U	Unsigned	$0 \leq n < 10^{31}$

The formats of packed decimal integers are as follows:



Examples of Decimal Integers

The source language and the associated stored value for the various types of decimal integers are given in the following examples:

<i>Source language</i>	<i>Stored Value (hexadecimal)</i>
P'125'	125B
-P'99436'	9943 6D00
U'125'	1250
U'99436'	9943 6000
L'125'	2B31 3235
-L'99436'	2D39 3934 3336
T'125'	3132 352B
-T'99436'	3939 3433 362D
O'125'	3132 4530
-O'99436'	3939 3433 4F30
O'20'	327B
-O'20'	327D
N'125'	3132 3530

FIXED-POINT CONSTANTS

A fixed-point constant is written as a decimal number with an associated scale factor and an optional precision field. When the resultant value is stored in memory, a fixed-point constant appears as a signed integer with negative values in two's 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, the true value of a fixed point binary number may be calculated by multiplying its integer representation by 2^{-s} .

The two formats for writing fixed-point constants are, as follows:

Format 1

$$\left[\begin{array}{c} + \\ - \end{array} \right] \left\{ \begin{array}{l} i.[f] \\ [i].f \end{array} \right\} B \left[\begin{array}{c} + \\ - \end{array} \right] s \text{ SINGLE PRECISION}$$

Format 2

$$\left[\begin{array}{c} + \\ - \end{array} \right] \left\{ \begin{array}{l} i.[f] \\ [i].f \end{array} \right\} B \left(r, \left[\begin{array}{c} + \\ - \end{array} \right] s \right) \text{ SINGLE OR DOUBLE PRECISION}$$

[±]

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.
- r Specifies the precision of the constant, $0 < r \leq 31$.
- $[\pm]s$ Specifies the value and sign of the scale factor.

Format 1 has an implied precision of 15 bits. The value of a fixed-point constant must fall within the range

$$2^{-s} \leq |R| < 2^{31-s}$$

where R is the value of the decimal number.

Fixed-point constants are stored as aligned signed two's complement binary numbers; that is they occupy one word if they are single precision and two words if they are double precision. The assumed binary point is located s bits to the left of the rightmost bit if the scale factor is positive, and -s bits to the right of the rightmost bit when the scale factor is negative.

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

<i>Source Language</i>	<i>Stored Value</i>	
2.5B4	0028	
2.5B8	0280	
65536B-15	0002	
65536B-7	0200	
-2.5B8	FD80	
-65536B-15	FFFE	
262144B(20,0)	0004	0000
262144B(20,-7)	0000	0800
262144B(15,-7)	0800	
-262144B(20,0)	FFFC	0000
-262144B(20,-7)	FFFF	F800

FLOATING-POINT CONSTANTS

The 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.)

There are three formats for floating-point constants:

Format 1

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

Format 2

$$\begin{bmatrix} + \\ - \end{bmatrix} \left\{ \begin{array}{l} i.[.f] \\ [i].f \end{array} \right\} E \begin{bmatrix} + \\ - \end{bmatrix} c \text{ SHORT PRECISION}$$

Format 3

$$\begin{bmatrix} + \\ - \end{bmatrix} \left\{ \begin{array}{l} i.[.f] \\ [i].f \end{array} \right\} D \begin{bmatrix} + \\ - \end{bmatrix} c \text{ DOUBLE PRECISION}$$

$[\pm]$ 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 floating-point representation is desired.
- D Indicates that a double-precision 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 5.3976×10^{-79}) and less than 2^{-252} (approximately 7.2370×10^{75}).

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 or 57-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.

<i>Source Language</i>	<i>Stored Value</i>
-5	8080 0000
0.5E12	9474 6A52
0.5D12	9474 6A52 8800 0000
-0.5D12	9574 6A52 8800 0000
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, external, or common), values, and addresses. Components of an expression can be joined by various functions and arithmetic operators, as follows:

<i>Arithmetic Operator</i>	<i>Meaning</i>
+	Addition (or Unary +)
-	Subtraction (or Unary -)
*	Multiplication
/	Division
<i>Boolean Function</i>	<i>Meaning</i>
AND	Conjunction of argument1 and argument2
OR	Inclusive disjunction of argument1 and argument2
XOR	Exclusive disjunction of argument1 and argument2
NOT	Negation of argument1

<i>Shift Function</i>	<i>Meaning</i>
ALS	Arithmetic left shift of argument1 by argument2 bits
ARS	Arithmetic right shift of argument1 by argument2 bits
LLS	Logical left shift of argument1 by argument2 bits
LRS	Logical right shift of argument1 by argument2 bits
<i>Arithmetic Function</i>	<i>Meaning</i>
MOD	Remainder after division when argument1 is divided by argument2

General Format of a Function:

function-name (argument 1, argument 2)

NOTE: The Boolean NOT function has only one argument.

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 (two's 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 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. The results of each computation must be within the allowable range of integer dimensionless values. The range is from -32768 to +32767.

The shift functions must satisfy the conditions specified below or else the function will not be performed and the operation will be flagged as an error condition.

ALS	$0 \leq \text{argument2} < 15$
ARS	$0 \leq \text{argument2} < 15$
LLS and LRS	$0 \leq \text{argument2} < 15$

Argument2 in the arithmetic function MOD must not equal 0. If this condition is not satisfied, an error condition is flagged and the function is performed as if argument2 is equal 1.

The arguments in all arithmetic operations and functions must be binary integers.

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).

Below are examples of functions:

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

AND

DC <LOC1+AND(VAL1,VAL2)
resolves to address 300 hexadecimal

OR

DC <LOC1+OR(VAL1,VAL2)
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(VAL2,VAL3)
resolves to value 21 hexadecimal

MOD

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

EVALUATING EXPRESSIONS

Within an expression, evaluation proceeds from left to right on a same level of inclusiveness until a higher level is reached. The levels of hierarchy are:

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

Parentheses can be used to change the evaluation order. Each lesser inclusive set of parentheses is a higher hierarchy level.

LOCATION AND VALUE EXPRESSIONS

The Assembler permits expressions to be used to specify values and locations. Internal and external value expressions denote a computation to be performed by the Assembler and produce integer dimensionless values.

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.

VALUE EXPRESSIONS

Value expressions are used to express computations to be done by the Assembler. There are two types of value expressions:

- Internal value expressions — Refer only to values that are defined within the referencing program.
- External value expressions — Refer to one value defined in an external program and may refer to elements within the referencing program.

Internal Value Expressions

An internal value expression, which produces an integer dimensionless value, may be written as a single factor or 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 expression must be one of the following items:

(int-val-expression)
binary integer
string constant
int-val-label
assembler function
Commercial Processor edit function

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:

$$\text{int-val-exp} \left\{ \begin{array}{l} + \\ - \end{array} \right\} \left\{ \begin{array}{l} \text{binary-integer} \\ \text{string-constant} \\ \text{assembler-function} \\ \text{Commercial-Processor-edit-function} \\ \text{int-val-label} \\ \text{product (or quotient) of these terms} \end{array} \right\}$$

The difference between two internal locations.

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:

$$X'34F0' + (\text{VAL8} - (\text{VALB} / (\text{X}'E4' * 2)))$$

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:

$$B'11110110' + (\text{COMM1} - \text{COMM2}) / 2 * (54 + \text{VALF} - (\text{LOCA} - \text{LOCB}))$$

The expression in example 2 is evaluated as follows:

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

External Value Expressions

An external value expression references one value defined in another program (declared by an XVAL statement in the referencing program) and may reference additional elements defined in the referencing program. An external value expression must take one of the following forms:

$$[+] \left\{ \begin{array}{l} \text{ext-val-label} \\ \text{ext-val-exp} \\ (\text{ext-val-exp}) \end{array} \right\} \left[\left\{ \pm \right\} \left\{ \begin{array}{l} \text{binary-integer} \\ \text{string-constant} \\ \text{int-val-label} \\ \text{assembler-function} \\ \text{Commercial-Processor-edit-function} \\ (\text{int-val-exp}) \end{array} \right\} \right]$$

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

The following example illustrates external value expressions. In this example, VALZ is an internal value and VALEX is an external value.

Example:

18 + VALEX + VALZ

1. The two internal values are added together, i.e., 18 + VALZ
2. The value 18 + VALZ is the offset associated with the external value, VALEX.

LOCATION EXPRESSIONS

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

- Internal location expressions — Refer only to locations that are defined within the referencing program.
- External location expressions — Refer only to locations defined in an external program and may refer to elements within the referencing program.
- Common location expressions — Refer only to locations within common blocks and may refer to other 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 elements, must take one of the following forms:

$$\text{int-loc-exp}\{\pm\} \left\{ \begin{array}{l} \text{binary integer} \\ \text{string constant} \\ \text{assembler function} \\ \text{Commercial-Processor-edit-function} \\ \text{int-val-label} \\ \text{(int-val-exp)} \end{array} \right\}$$
$$\text{int-val-exp} + \left\{ \begin{array}{l} \text{int-loc-label} \\ \text{(int-loc-exp)} \\ \$ \end{array} \right\}$$

In the previous form, the \$ 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:

$$\text{ext-loc-exp } \{ \pm \} \left\{ \begin{array}{l} \text{binary integer} \\ \text{string constant} \\ \text{assembler function} \\ \text{Commercial-Processor-edit-function} \\ \text{int-val-label} \\ \text{(int-val-exp)} \end{array} \right\}$$

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

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:

$$\text{common-loc-exp } \{ \pm \} \left\{ \begin{array}{l} \text{binary integer} \\ \text{string constant} \\ \text{assembler function} \\ \text{Commercial-Processor-edit-function} \\ \text{int-val-label} \\ \text{(int-val-exp)} \end{array} \right\}$$

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

In the previous form the \$ 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").

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 Assembler.²
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) common (COMM) or a local common (LCOMM) 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.

²An assembly pass is a complete read of the source program.

Section 3

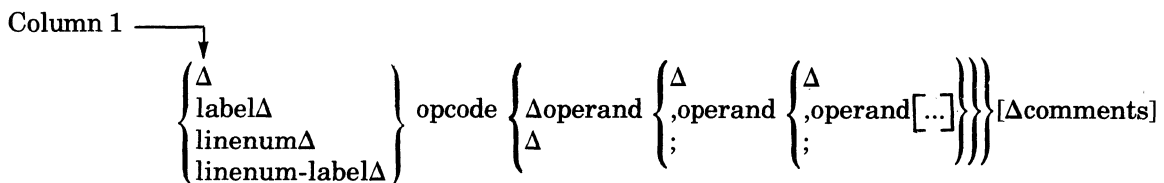
Programming Considerations

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:

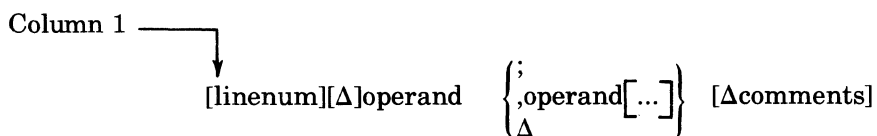
- Rules of formatting your source language statements
- Ordering of statements in an assembly language program
- Rules governing the calling of system services and external procedures
- Utility programs that supplement assembly language source programs

ASSEMBLY LANGUAGE SOURCE STATEMENT FORMATS

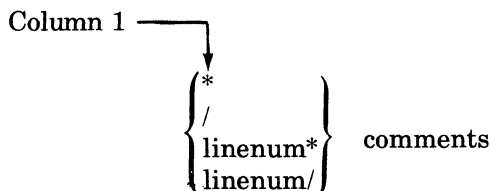
As mentioned in Section 2, the assembly language consists of Assembler controlling statements and assembly language (operational) instructions. Assembly language source code must be submitted to the Assembler in a recognizable format so that it can be interpreted accurately. Therefore, when coding assembly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



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:



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.



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 Assembler is to cause 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 an 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 through 7, 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 a free-form. However, it is suggested that you establish a fixed format for coding source statements (e.g., always starting op codes 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 END statement must be the last statement in the source program.

CALLING SYSTEM SERVICES

System services (e.g., the Task Manager) can be requested through the use of monitor service calls and macro calls. For information concerning requests for system services see the *System Services Macro Calls* 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.

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.

ALTERNATE METHOD OF HANDLING INPUT/OUTPUT AND FILE MANIPULATION

Input/output and file manipulation can be accomplished by writing Assembler routines or by using monitor service requests. Details concerning monitor service requests are contained in the *System Service Macro Calls* manual.

ASSEMBLER

The Assembler processes source statements written in 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 Preparation* manual.

CROSS-REFERENCE LISTING

When a source program is assembled, the Assembler produces a cross-reference listing, if the proper option is specified in the command invoking the Assembler. This list itemizes all labels and symbols in the source module and flags labels that are undefined or defined more than once.

SAF/LAF CONSIDERATIONS

For execution in a Mod 400 systems, assembly language programs may be in either the long address form (LAF) or the short address form (SAF). In some circumstances it may be desirable to create an assembly language program that can be executed in both SAF and LAF configurations. For instructions on writing such a program, see *Program Preparation* manual.

REENTRANCY CONSIDERATIONS

A program is defined as reentrant, if a single copy of the code portion of a bound unit can be simultaneously executed by several tasks, which may be in the same task group or different task groups. GCOS 6 software is designed to facilitate the writing of reentrant programs. See *Program Preparation* manual.

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, most of which are not assembled into the object text, provide information to the Assembler for:

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

- BORG
- END
- ORG
- TITLE

LIST-CONTROLLING STATEMENTS

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

- CLST
- LIST
- NLST

DATA-DEFINING STATEMENTS

Data-defining statements are required to define data used in the program. 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:

- ARGLST
- BTEXT
- DC
- PTRAY
- TEXT

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:

- COMM
- LCOMM
- RESV

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:

- EQU
- XLOC
- XVAL

PROGRAM-LINKING STATEMENTS

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

- CALL
- CALL2
- CTRL
- EDEF
- XDEF

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:

- FAIL
- IF
- NULL

OPERATION CODE-DEFINING STATEMENT

- DEFGEN

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.

ARGLST

Instruction:

Create argument list

Source Language Format:

[label]ΔARGLSTΔarg1 [,arg2] . . .

Description:

Creates a word containing an arbitrary constant whose bits 9 through 15 specify the value $(M*\$AF+1)$, where M is the number of arguments. This arbitrary constant is followed by the relocatable address of each argument in the statement. The relocatable addresses are the same as though the following DC statement was coded after the arbitrary constant.

[label]ΔDCΔ<argument₁ [, <argument₂] . . .

If the Assembler is invoked with the SLIC argument, this instruction will also identify the resulting object text as being an argument list. The SLIC (SAF/LAF Independent Code) argument must be used if compilation units produced by the Assembler is to run in both SAF and LAF configurations.

The maximum number of arguments that can be specified in an ARGLST instruction is 31. The ARGLST instruction facilitates the writing of reentrant programs by allowing the arguments to be separated (i.e., placed in a different segmented address space) from the calling statement. (See the Program Preparation manual for instructions on writing reentrant programs.)

BORG

BORG

Instruction:

Byte Origin

Source Language Format:

$$[\text{label}]\Delta\text{ORGA} \left\{ \begin{array}{l} \text{common-location-expression} \\ \text{internal-location-expression} \end{array} \right\}$$

Description:

Sets the byte indicator to the odd (i.e., right) byte and 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 initial value of the Assembler's byte indicator is the even (i.e., left) byte.

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 assigned to 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.

BTEXT

Instruction:

Allocate space for text

Source Language Format:

$$[\text{label}]\Delta \text{BTEXT} \Delta \left\{ \begin{array}{l} \text{string-constant} \\ \text{Commercial-Processor-} \\ \text{edit-function} \end{array} \right\} \left[\left\{ \begin{array}{l} \text{string-constant} \\ \text{Commercial-Processor-} \\ \text{edit-function} \end{array} \right\} \dots \right]$$

Description:

Causes the Assembler to allocate the binary representation of the successive string constants, and/or edit functions concatenated into the fewest number of bytes (i.e., packed). The Assembler inserts "slack bits" (0's) between successive operands as necessary. Each ASCII string constant or edit function begins at a bit position that is a multiple of 8; each hexadecimal string constant begins at a bit position that is a multiple of 4; bit string constants have no slack bits inserted.

If the last byte occupied by the concatenated string is not exactly full, the remaining bits are zero-filled. The first byte of the concatenated string is allocated at the object memory byte location indicated by the Assembler's location counter and byte indicator. Following processing of the Byte Text statement, the values of the location counter and byte indicator indicate the object memory byte location of the first byte following the concatenated string allocated by the BTEXT statement.

NOTE: When the byte indicator indicates the odd (right) byte, BTEXT is the only memory-allocating assembly language statement allowed.

CALL

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 and causes that subroutine to be linked with the calling procedure.

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. The maximum number of arguments is 31.

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 entry name in the XLOC statement shown in the breakdowns 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).

If the assembler is invoked with the "SLIC" control argument, it will identify the object unit text resulting from the branch and DC statements in the second breakdown as being an argument list. Additional information about calling external procedures may be found in the *Program Preparation* manual.

CALL2

Instruction:

Call external procedure.

Source Language Format:

```
[label]ΔCALL2Δ[obj-mod-name.]entry,address-expression
```

Description:

Initiates a transfer of control to a specified external subroutine, causes that subroutine to be linked with the calling procedure, and generates the address of a list of arguments to be made available to the called subroutine. The argument list is normally generated by the ARGLST statement which is previously described in this section.

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.

address-expression

Generates the address of the argument list. The address expression must be one of the forms valid in a LAB instruction.

The CALL2 statement is broken down as follows:

CTRL	LINK obj-mod-name
XLOC	entry
LAB	\$B7,address-expression
LNJ	\$B5,<entry

The entry name in the XLOC statement shown in the breakdown is not entered into the Assembler's symbol table and ceases to exist after the LNJ instruction is assembled.

CLST

CLST

Instruction:

Conditional Listing

Source Language Format:

[label]ΔCLSTΔint-val-expression

Description:

If the internal value expression is ≥ 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).

NOTE: Temporary label definition is prohibited when ORG has been performed in a common block.

int-val-exp

Specifies the size (in words) of the common area. The Linker (see the *Program Execution and Checkout* 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 2), 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

CTRL

Instruction:

Pass control information to Linker

Source Language Format:

ΔCTRLΔlinker-directive(s)

Description:

Provides a method of passing Linker directives from the source program to the Linker (see the *Program Execution and Checkout* manual for a description of the Linker).

The operand has the following meaning:

linker-directive(s)

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 Constants

Source Language Format:

[label]ΔDCΔoperand₁[,operand₂]...

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 of the DC statement must conform to one of the following formats:

Format 1:

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

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

Format 2:

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

Causes a 1-word displacement from the current location to the specified location to be allocated.

Format 3:

(e.g., int-loc-expression produces the same results as int-loc-expression — \$)

$$[=] \left\{ \begin{array}{l} \text{string-constant} \\ \text{decimal-integer-constant} \end{array} \right\}$$

Constants are padded if necessary to make an integral number of words; the padded value is allocated to memory. ASCII string constants are padded by appending a space. Hexadecimal and bit string constants and decimal integer constants are padded by a sufficient number of zero bits.

Format 4:

$$[=] \left\{ \begin{array}{l} \text{int-val-expression} \\ \text{ext-val-expression} \end{array} \right\}$$

Causes a 1-word binary integer to be allocated.

Format 5:

[=] fixed-point-constant

Causes a 1- or 2-word fixed point value to be allocated. If the precision of the constant is from 1 through 15, one word is allocated; if the precision is from 16 through 31, two words are allocated.

Format 6:

[=] floating-point-constant

Single precision floating-point constants cause a 2-word value to be allocated.

Double precision floating-point constants cause a 4-word value to be allocated.

Floating-point constants are described in Section 2.

Format 7:

[=] Commercial-Processor-data-descriptor

Causes the allocation of a 2-word data descriptor.

DEFGEN

Format 8:

complex-label

Processed as described above for a format 1, 4, or 5 operand, depending on whether the label has been equated to a direct IMA address expression, an external value IMO address expression, an internal value IMO address expression, a short fixed point constant IMO address expression, or a Commercial Processor data descriptor funtion, respectively. A complex type label equated to any other address expression is illegal.

The number of operands is limited only in that the address space of a single program is restricted to relative locations 0 through 32767.

DEFGEN

Instruction:

Define generic operation code

Source Language Format:

label Δ DEFGEN Δ int-val-expression

Description:

Allows the user to create operation codes that specify generic instructions

The name (specified by label) is declared to be a one-word generic instruction (specified by int-val-expression).

The DEFGEN statement is designed for use with the Writeable Control Store. However, the presence of the Writeable Control Store is not essential for its use.

The DEFGEN statement must precede any instruction that references the label.

A Honeywell mnemonic for an assembly instruction may be specified as a label. (See example below.) In this case, the user defined instruction is executed whenever it subsequently occurs (rather than the instruction specified by the Honeywell mnemonic).

Example:

The assembly language instruction NOP is a one-or two-word instruction that requires an address expression. For a source program that uses one-word NOP's in the same way throughout, the NOP may be redefined to give the appearance of a generic instruction. Assume a source program that repeatedly uses the following instruction: NOP Δ >\$+2. This is a one-word instruction whose assembled value is 0F02. NOP can be redefined as follows by the DEFGEN statement and thereafter used without including an address expression.

```
NOP DEFGEN Z'0F02'
```

```

.
.
.
NOP
```

EDEF

Instruction:

External label definition — For use with Linker and Loader

Source Language Format:

$$\Delta EDEF \Delta \left\{ \begin{array}{l} \text{label-1} \\ \left(\text{label-2}, \left\{ \begin{array}{l} \text{int-loc-exp} \\ \text{int-val-exp} \end{array} \right\} \right) \end{array} \right\} \quad [\dots]$$

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 EDEF 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-1

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 the same name in a separately assembled program through an XLOC or XVAL statement.

label-2

$$\left(\text{label-2}, \left\{ \begin{array}{l} \text{int-loc-exp} \\ \text{int-val-exp} \end{array} \right\} \right)$$

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 this evaluation are passed to the Linker with the object text for use during the linking process and subsequently passed to the Loader to be included as part of the bound unit being created (see *Program Execution and Checkout* manual).

Notes:

1. It is not necessary for all labels identified through the EDEF statement to be referred to by an external program.
2. If a label is not identified to an external procedure by an EDEF statement, the label can be identified in the bound unit at link time by the EDEF directive to the Linker, provided it has been defined via the XDEF statement.
3. If a label is not identified to the Linker by either the EDEF or XDEF statement, it may be identified to the Linker via the LDEF or VDEF Linker directive as appropriate. It then may be subsequently identified to the bound unit with the EDEF Linker directive.

END

END

Instruction:

End of program

Source Language Format:

Δ END Δ program-name[,internal-location-expression]

Description:

Identifies the end of the assembly language program. Statements subsequent to this statement will be ignored by the Assembler. If this statement is missing, the Assembler will generate an END statement.

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:

label Δ EQU Δ	}	location-expression value-expression address-expression complex-label identifier single precision fixed-point-constant
-----------------------------	---	---

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:

single-precision-fixed-point-constant

location-expression

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).

Note:

Complex labels cannot contain Commercial Processor address expressions.

Complex labels cannot contain direct or indirect B6 relative plus local common block plus displacement addressing (see "B-relative addressing" in Section 5).

identifier

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

FAIL

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:

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

Description:

If the specified condition is met, the Assembler skips subsequent statements until the label is encountered; otherwise, the next sequential instruction is processed. (0 is neither positive nor negative.)

The opcode is interpreted as follows:

IFP

Skip to label if int-val-expression is positive (i.e. > 0).

IFNP

Skip to label if int-val-expression is not positive (i.e. ≤ 0).

IFN

Skip to label if int-val-expression is negative (i.e. < 0).

IFNN

Skip to label if int-val-expression is not negative (i.e. ≥ 0).

IFZ

Skip to label if int-val-expression is zero.

IFNZ

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

IFOD

Skip to label if int-val-expression is odd.

IFEV

Skip to 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.

label

Label (see "Labels" in Section 2) identifying 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.

Example:

```
IFNZ AND($SW,Z'4000'),SKIPIT
```

External Switch 1 is checked. If it is set the Assembler skips the subsequent statements until the label SKIPIT is encountered. If External Switch 1 is not set, the Assembler goes to the next line of assembly code. This is an example of varying an assembly procedure without altering the assembly language source program.

LCOMM

LCOMM

Instruction:

Define local common block

Source Language Format:

label Δ LCOMM Δ int-val-exp

Description:

Provides a way for a block of data local to a program to be allocated not by the Assembler, but by the Linker using standard linking procedures for allocating common blocks. The data allocated by use of the LCOMM statement is not shared.

The label field and operands have the following meanings

label

The name of the common area.

NOTE: LCOMM does not allow a temporary label to be specified.

int-val-exp

Specifies the size (in words) of the common area. The Linker (see the *Program Execution and Checkout* 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 LCOMM statement does not alter the Assembler's location counter). In the case of a local common block, the Linker removes the name of the local common block from its symbol table after it has linked the program which defined the local common block.

int-val-exp is an internal value expression (see Section 1), and must be defined prior to the occurrence of this LCOMM 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.

LIST

Instruction:

List following source statements

Source Language Format:

Δ LIST

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

NLST

Instruction:

Inhibit listing of following source statements

Source Language Format:

Δ NLST

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

ORG

Instruction:

Origin

Source Language Format:

$$[\text{label}]\Delta\text{ORGA} \left\{ \begin{array}{l} \text{common-location-expression} \\ \text{internal-location-expression} \end{array} \right\}$$

Description:

Sets the byte indicator to the even (i.e., left) byte and 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 0. The initial value of the Assembler's byte indicator is the even (i.e., left) byte.

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 assigned to 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.

PTRAY

Instruction:

Create pointer array

Source Language Format:

```
[label]ΔPTRAYΔ location-exp1[,location-exp2] . . .
```

Description:

Creates an array of pointers. The address of a memory word is referred to as a pointer. Pointers may occur at the level of machine language both as direct addresses and as indirect addresses.

The Assembler generates the object unit code as if the statement were transformed into the following DC statement.

```
[label]ΔDCΔ <location-exp1 [, <location-exp2] . . .
```

If the Assembler is invoked with the SLIC argument, it will also identify the object unit text resulting from the PTRAY statement as being a pointer array. This is necessary so that in loading a SLIC program, the Loader will compress addresses if executing in SAF mode.

RESV

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 bound unit as work or storage space.

The label field and operands have the following meanings:

label

If specified, the first word of 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 bound unit 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:

$$[\text{label}]\Delta\text{TEXT}\Delta \left\{ \begin{array}{l} \text{String-constant} \\ \text{Commercial Processor-edit-function} \end{array} \right\}$$

$$\left[\left\{ \begin{array}{l} \text{,string-constant} \\ \text{,Commercial Processor-edit-function} \end{array} \right\} [\dots] \right]$$

Description:

Causes the Assembler to allocate the binary representation of the successive string constants, and/or edit functions concatenated into the fewest number of words (i.e., packed). The Assembler inserts "slack bits" (0's) between successive operands as necessary. Each ASCII string constant or edit function begins at a bit position that is a multiple of 8; each hexadecimal string constant begins at a bit position that is a multiple of 4; bit string constants have no slack bits inserted. If the last word occupied by the concatenated string is not exactly full, the remaining bits are zero-filled.

TITLE

TITLE

Instruction:

Start of program

Source Language Format:

Δ TITLE Δ program-name[,rev-number] [Δ page-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
3. The lowercase letters are considered to be equivalent to the corresponding uppercase letters.

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 — For use with Linker only.

Source Language Format:

$$\Delta XDEF \Delta \left\{ \begin{array}{l} \text{label-1} \\ \left(\text{label-2}, \left\{ \begin{array}{l} \text{int-loc-exp} \\ \text{int-val-exp} \end{array} \right\} \right) \end{array} \right\} \left[\dots \right]$$
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-1

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 the same name in a separately assembled program through an XLOC or XVAL statement.

$$\left(\text{label-2}, \left\{ \begin{array}{l} \text{int-loc-exp} \\ \text{int-val-exp} \end{array} \right\} \right)$$

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 Execution and Checkout* manual). The results of the evaluation are not, however, passed to the Loader.

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 or VDEF command to the Linker.
3. If the label definition is to be passed to the Loader, use EDEF in lieu of XDEF.

XLOC

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 same external location may be defined by XLOC statements more than once in an assembly language program.

The external program must identify the labels in an XDEF or EDEF Assembler control statement or by an equivalent means in other programming languages.

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:

Δ XVAL Δ labela[,labelb] . . .

Description:

Identifies labels associated with values in programs assembled separately from this program (i.e., external procedures), but used in this program. The same external value may be defined by XVAL statements more than once in an assembly language program.

The external program must identify the labels in an XDEF or EDEF Assembler control statement or by an equivalent means in other programming languages.

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 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:

- Arithmetic
- Boolean
- Branching
- Comparison
- Controlling
- Input/Output
- Loading
- Memory Management
- Modification
- Move
- Queue management
- Shifting
- Stack management
- Storing
- Swapping

The following assembly language operations can execute only if the central processor is in the privileged state:

ASD	IOH	RTCN
CNFG	IOLD	WDTF
HLT	LEV	WDTN
IO	RTCF	

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
AID ¹	MUL
CAD	NEG
DEC	SID ¹
DIV	SUB

¹Instruction is executable only on the 6/40 and 6/50 models.

BOOLEAN OPERATIONS

Boolean operations (Inclusive OR, Exclusive OR, AND, and NOT) 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	LEV	RTT
CNFG ²	LNJ	WDTF
ENT	MCL	WDTN
HLT	RTCF	
JMP	RTCN	

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	LRDB ²
		RSTR

²Instruction is executable only on the 6/40 and 6/50 models.

MEMORY MANAGEMENT OPERATIONS

The following assembly language instructions are provided to support the memory management operations:³

ASD VLD

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

MOVE OPERATION

The following instruction performs a move operation:

MMM⁴

QUEUE OPERATIONS

Programmer's reference information for queue instructions are in Appendix J.

The following assembly language instructions are provided to support queue operations:⁵

DQA QOH
DQH QOT

SHIFT OPERATIONS

Shift operations are achieved through the following assembly language instructions:

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

STACK OPERATIONS

Programmer's reference information for stack instructions are in Appendix K.

The following assembly language instructions are provided to support stack operations:⁶

ACQ RLQ
LDT STT

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
SRDB STM
SRM

SWAP OPERATIONS

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

SWB SWR

³Memory management operations require a Memory Management Unit which is available only on the 6/40 and 6/50 models.

⁴Instruction is executable only on the 6/40 and 6/50 models.

⁵The hardware required for queue operations is available only with the 6/40 models.

⁶The hardware required for stack operations is available only with the 6/40 models.

ASSEMBLY LANGUAGE INSTRUCTION TYPES

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

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

- Direct Immediate memory address
- Direct P-relative
- 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:

[label]ΔopcodeΔ $\left. \begin{array}{l} \text{R-register} \\ \text{binary-integer-constant} \end{array} \right\} \text{,address-expression}$

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 a binary 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):

- Direct immediate memory address
- Direct P-relative
- 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:

[label]ΔopcodeΔ $\left. \begin{array}{l} \text{R-register} \\ \text{B-register} \\ \text{M-register} \\ \text{S-register} \\ \text{binary-integer-constant} \end{array} \right\} \text{,address-expression } [\text{,mask}]$

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. If the Scientific Instruction Processor is present, the S-registers are hardware registers; otherwise, they may be provided as software-simulated scientific registers maintained by the Floating-Point Simulator. In the 6/40 and 6/50 models the M-registers are hardware registers. In the 6/30 models only M1 is a hardware register. In the 6/30 models, M2 through M7 may be provided as software-simulated mode registers. The Floating-Point Simulator maintains the M4 and M5 registers and the Commercial Processor simulator maintains the M3 register. There is no Honeywell-supplied software to maintain the simulated M2, M6, and M7 registers.

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, as defined in assembly language or by the user (with a DEFGEN statement) have the following format:

[label]Δopcode

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:

- Data and command I/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 the first byte of a buffer of data to be transferred to or from the device addressed by the channel number specified by the second operand.

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:

[label]ΔopcodeΔ $\left\{ \begin{array}{l} \text{R-register} \\ \text{integer-constant} \end{array} \right\}$,int-val-expression

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

- SHS — Shift short
- SHL — Shift long

The valid types are:

- Arithmetic
- Open
- Closed

The direction of the shift can be:

- Right
- 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:

$$[\text{label}]\Delta\text{opcode}\Delta \left\{ \begin{array}{l} \text{R-register} \\ \text{binary-integer-constant} \end{array} \right\} , [=] \left\{ \begin{array}{l} \text{Commercial-Processor- edit-function} \\ \text{Assembler-function} \\ \text{binary-integer-constant} \\ \text{string-constant} \\ \text{internal-value-label} \\ \text{int-val-expression} \end{array} \right\}$$

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 a binary 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.

Binary 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:

$$[\text{label}]\Delta\text{opcode}\Delta\text{addr-expression} \left[\left\{ \begin{array}{l} \text{Commercial-Processor- edit-function} \\ \text{Assembler-function} \\ \text{binary-integer-constant} \\ \text{string-constant} \\ \text{internal-value-label} \\ \text{external-value-label} \\ \text{int-val-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right\} \right]$$

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), Restore (RSTR) and bit handling instructions. It specifies the value of a one-word mask that indicates which registers are to be saved and restored or which bits are to be manipulated. Binary 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:

- Register addressing
- Immediate memory addressing
- Immediate operand addressing
- P-relative addressing
- B-relative addressing
- Short displacement addressing
- Special addressing
- 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:

$=\$R_n = \$B_n = \$S_n$

$=\$R_n$, $=\$B_n$, and $=\$S_n$ are mutually exclusive; i.e., some instructions permit the use of $=\$R_n$ and others allow $=\$B_n$ (the descriptions of the various instructions identify which is valid for that instruction). The $=\$R_n$ form is generally used in those instructions that require some data to be contained in the register. The $=\$B_n$ form is valid for those instructions that expect to find an address in the register. The $=\$S_n$ 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 execute 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 an 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:

$$\begin{aligned}
 &< \left\{ \begin{array}{l} \text{location-expression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \\ \text{temporary-label} \end{array} \right\} && \text{Direct IMA} \\
 \\
 &*\left\{ \begin{array}{l} \text{location-expression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \\ \text{temporary-label} \end{array} \right\} && \text{Indirect IMA} \\
 \\
 &< \left\{ \begin{array}{l} \text{location-expression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \\ \text{temporary-label} \end{array} \right\} \cdot \$R \begin{array}{l} \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\} \end{array} && \text{Indexed Direct IMA} \\
 \\
 &*\left\{ \begin{array}{l} \text{location-expression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \\ \text{temporary-label} \end{array} \right\} \cdot \$R \begin{array}{l} \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\} \end{array} && \text{Indexed Indirect IMA}
 \end{aligned}$$

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 displacement word 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 INTLBI 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,<INTLBI

In this example, the contents (0F0B) of location 20F4 (specified by the INTLBI) 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.

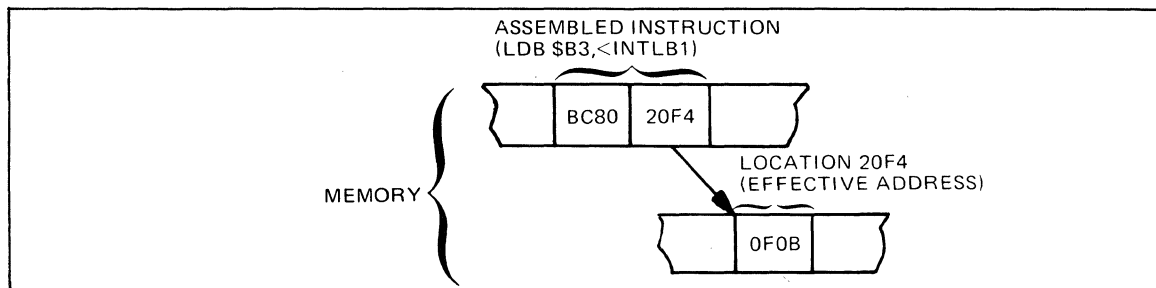


Figure 5-1. Direct Immediate Memory Addressing

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.

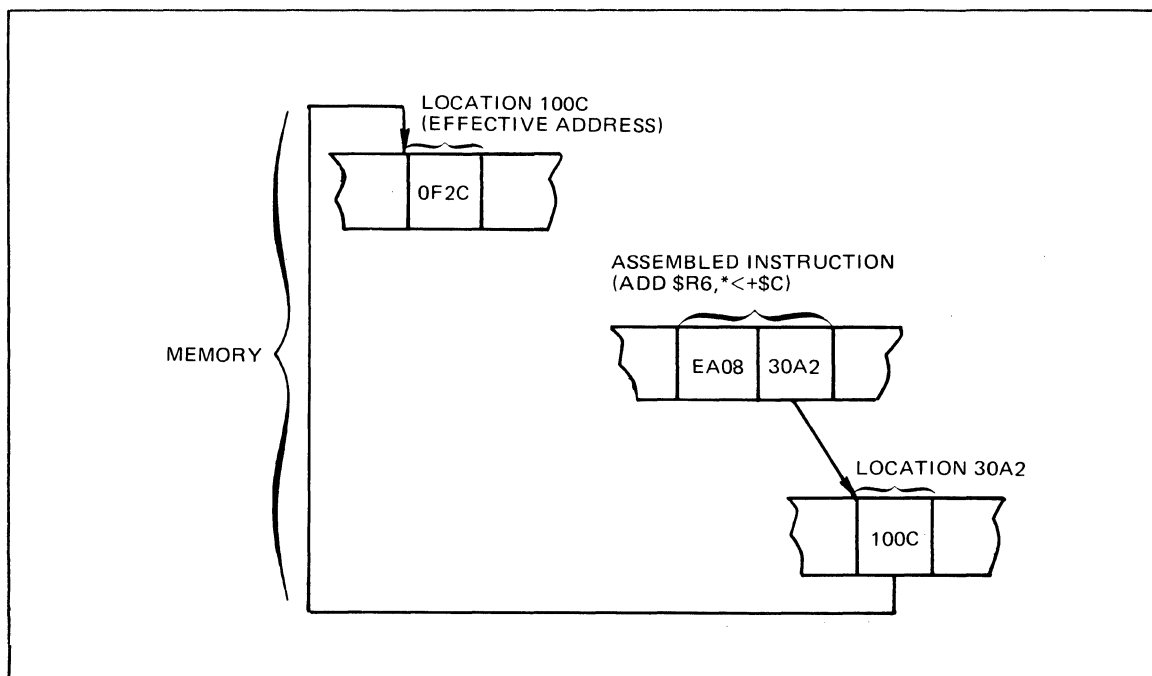


Figure 5-2. Indirect Immediate Memory Addressing

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 displacement 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, for a SAF configuration, 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.

For a LAF configuration, the system effectively multiplies the contents of the index register by 2 because the instruction is a base register instruction, and then adds this product to the address of TABLE1. (After this instruction is performed, the contents of \$R3 remain unchanged.)

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

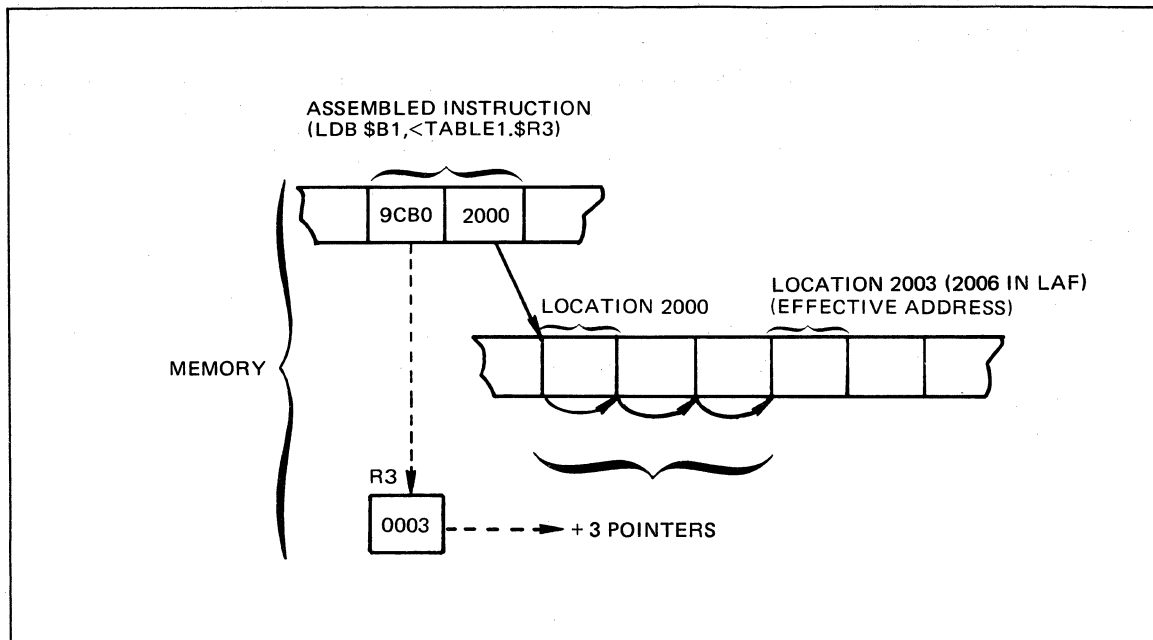


Figure 5-3. Indexed Direct Immediate Memory Addressing

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:

- = location-expression (LDB, STB, SWB, CMB, CMN)
- = { string-constant } (SAD, SCM, SCZD, SDV, SLD,
floating-point-constant } SML, SNGD, SSB, SST, SSW)
- = { internal-value-expression } (All other CP instructions)
external-value-label
fixed-point-constant
external-value-expression

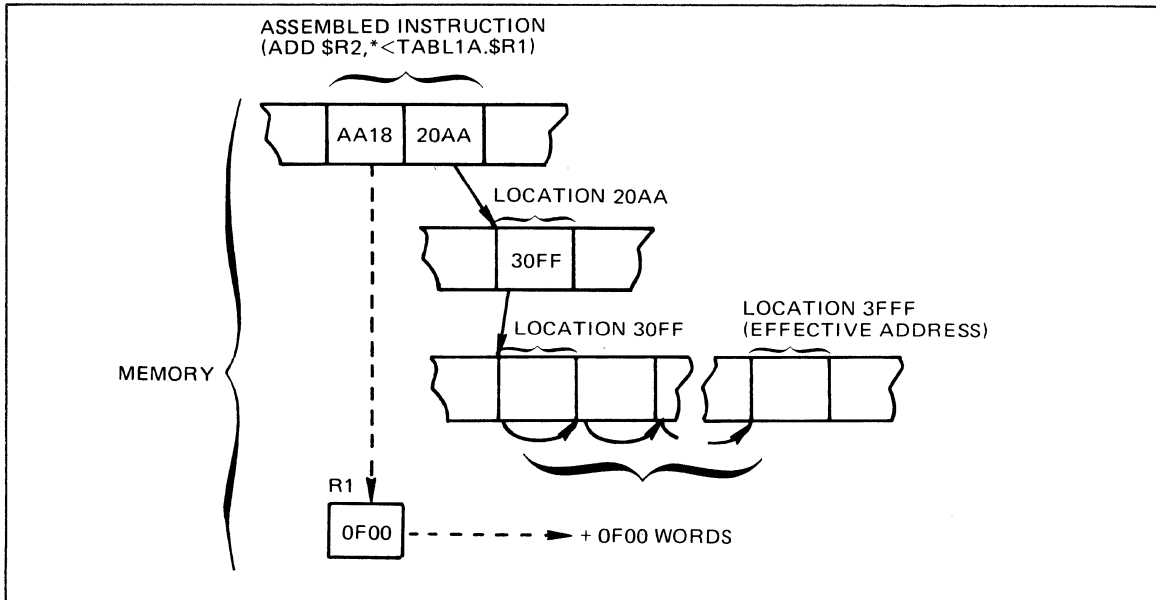
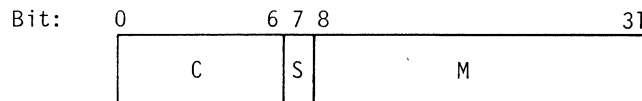


Figure 5-4. Indexed Indirect Immediate Memory Addressing

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



- c Characteristic (excess 64 power-of-16 exponent) of the mantissa.
- s Sign (0 = positive; 1 = negative) on 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), 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), is added to the floating-point value stored in \$S1 (i.e., 19), and the result is stored in \$S1.

The following code:

SAD \$S1,= 8.000010

which uses a floating-point constant value, will produce the same object code as example 1, above.

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

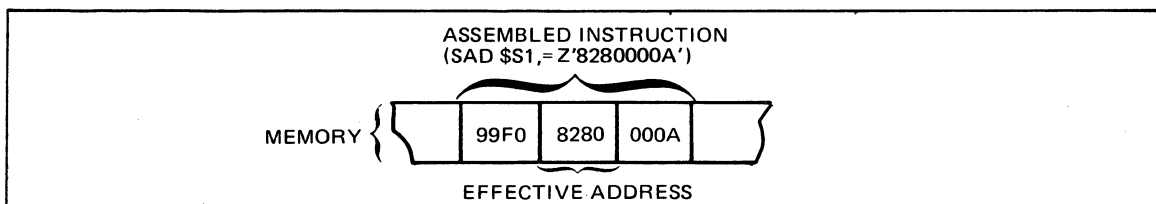


Figure 5-5. Immediate Operand Addressing-Scientific Instruction

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.

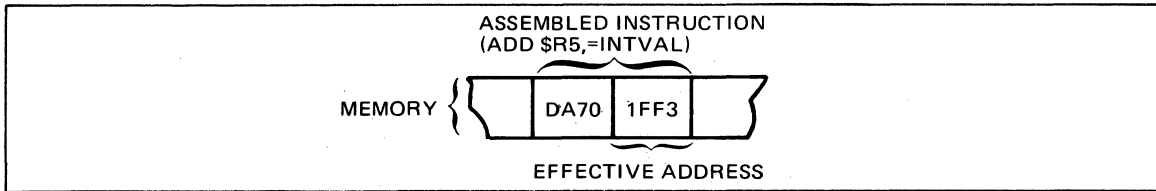
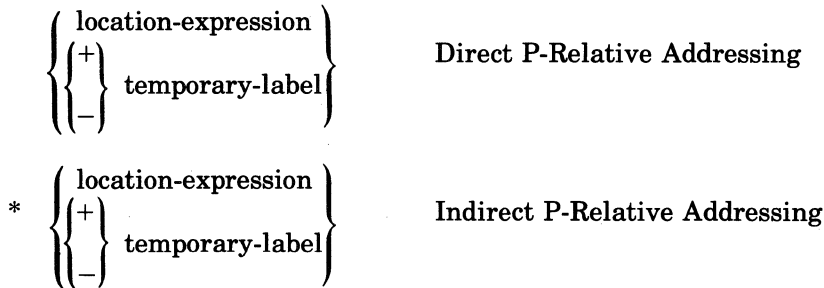


Figure 5-6. Immediate Operand Addressing

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:



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 an internal 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 in \$R5.

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

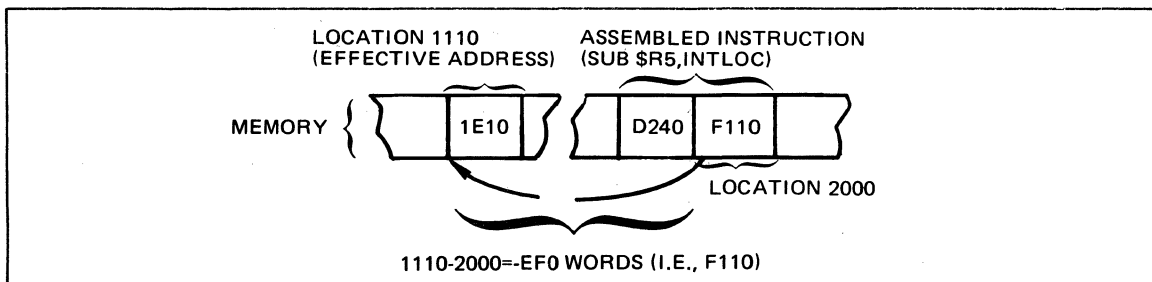


Figure 5-7. Direct P-Relative Addressing

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.

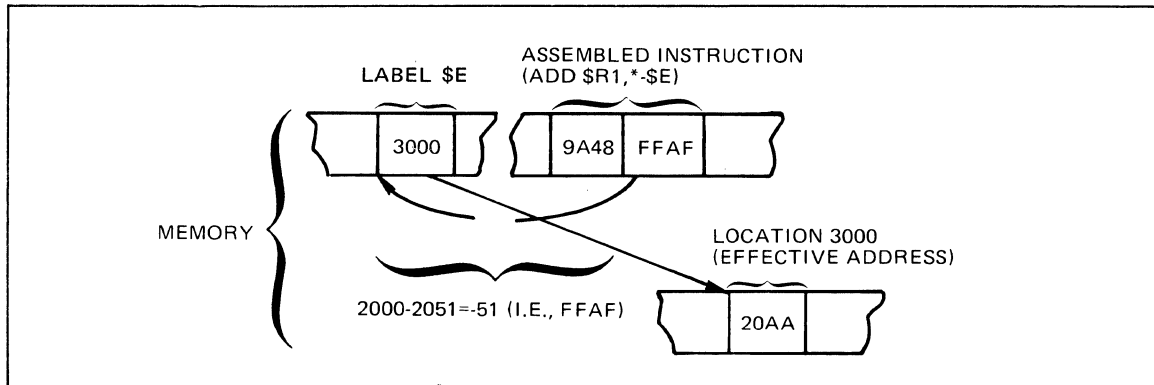


Figure 5-8. Indirect P-Relative Addressing

B-RELATIVE ADDRESSING

In B-relative addressing, a base register (i.e., \$B1, \$B2, ... \$B7) is used to reference 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 feature causes the hardware to automatically *decrement* the contents of the specified base or index register *before* executing the instruction. The pop feature causes the hardware to automatically *increment* its contents *after* execution.

B-relative addressing can take any of the formats given below. The first four forms 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 a base register rather than being expressed as a location expression or label. The next two are similar to the P-relative forms of addressing. The \$B6.\$LCOMW forms are provided to facilitate the writing of reentrant programs. Guidelines for writing reentrant programs are given in the *Program Preparation* manual.

\$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

$\$B_n$	$\left\{ \begin{array}{l} \text{int-val-expression} \\ \text{external-val-label} \end{array} \right\}$	— Direct B-relative plus displacement addressing
$*\$B_n$	$\left\{ \begin{array}{l} \text{int-val-expression} \\ \text{external-val-label} \end{array} \right\}$	— Indirect B-relative plus displacement addressing
$\$B6.\$LCOMW + \text{int-val-expression}$		— Direct B6 relative plus local common block plus displacement addressing
$*\$B6.\$LCOMW + \text{int-val-expression}$		— Indirect B6 relative plus local common block plus displacement addressing
$-\$B_n$		— B-relative addressing with automatic decrement before execution (Push)
$+\$B_n$		— B-relative addressing with automatic increment after execution (Pop)
$\$B \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}, -\$R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}$		— Indexed direct B-relative addressing with automatic decrement of index register before execution (Push)
$\$B \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}, +\$R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}$		— Indexed direct B-relative addressing with automatic increment of index register after execution (Pop)

DIRECT B-RELATIVE ADDRESSING

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

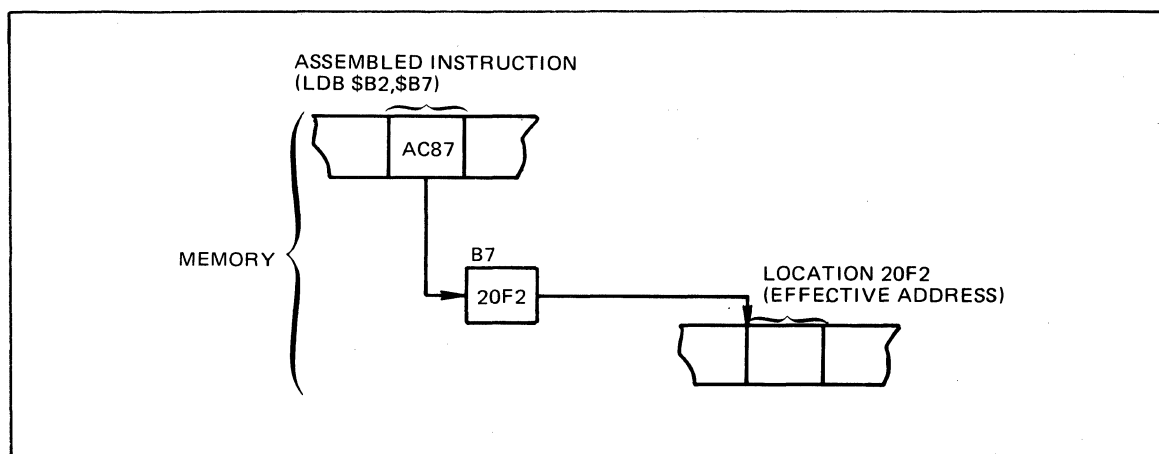


Figure 5-9. Direct B-Relative Addressing

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 a base 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.

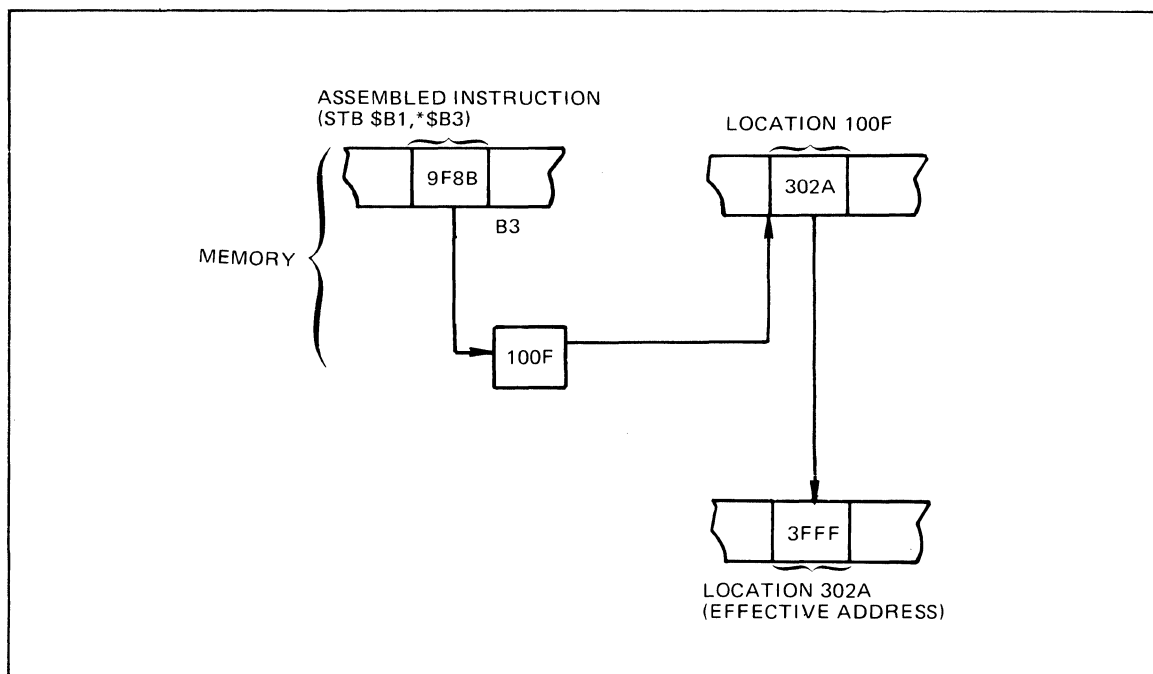


Figure 5-10. Indirect B-Relative Addressing

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 effectively multiplied by a factor (the factor is determined by the instruction) and added to the contents of the base 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 (effectively multiplied by one) 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.

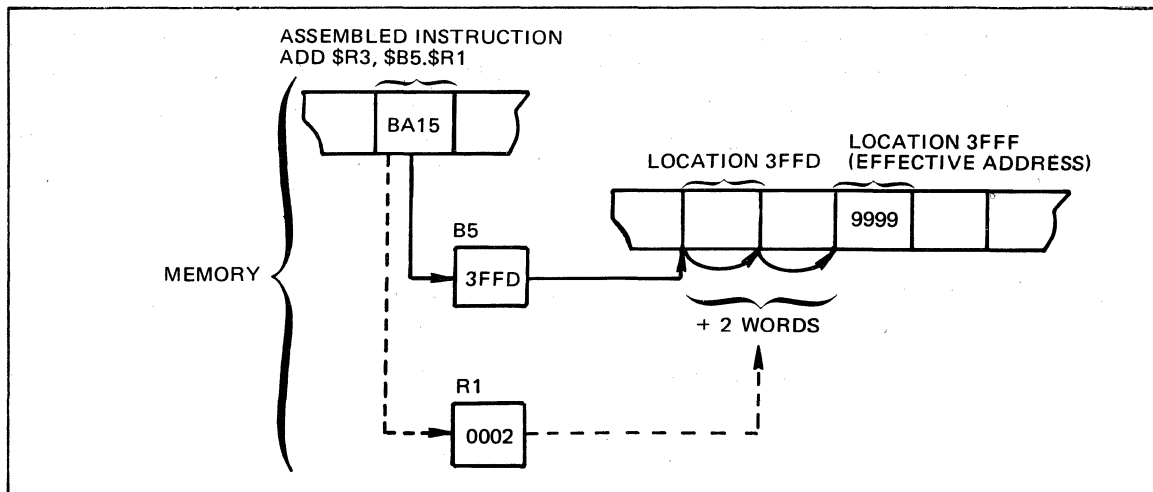


Figure 5-11. Indexed Direct B-Relative Addressing

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 effectively multiplied by a factor (the factor is determined by the instruction) and added to the contents of the location pointed to by the base 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.

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 a base 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.

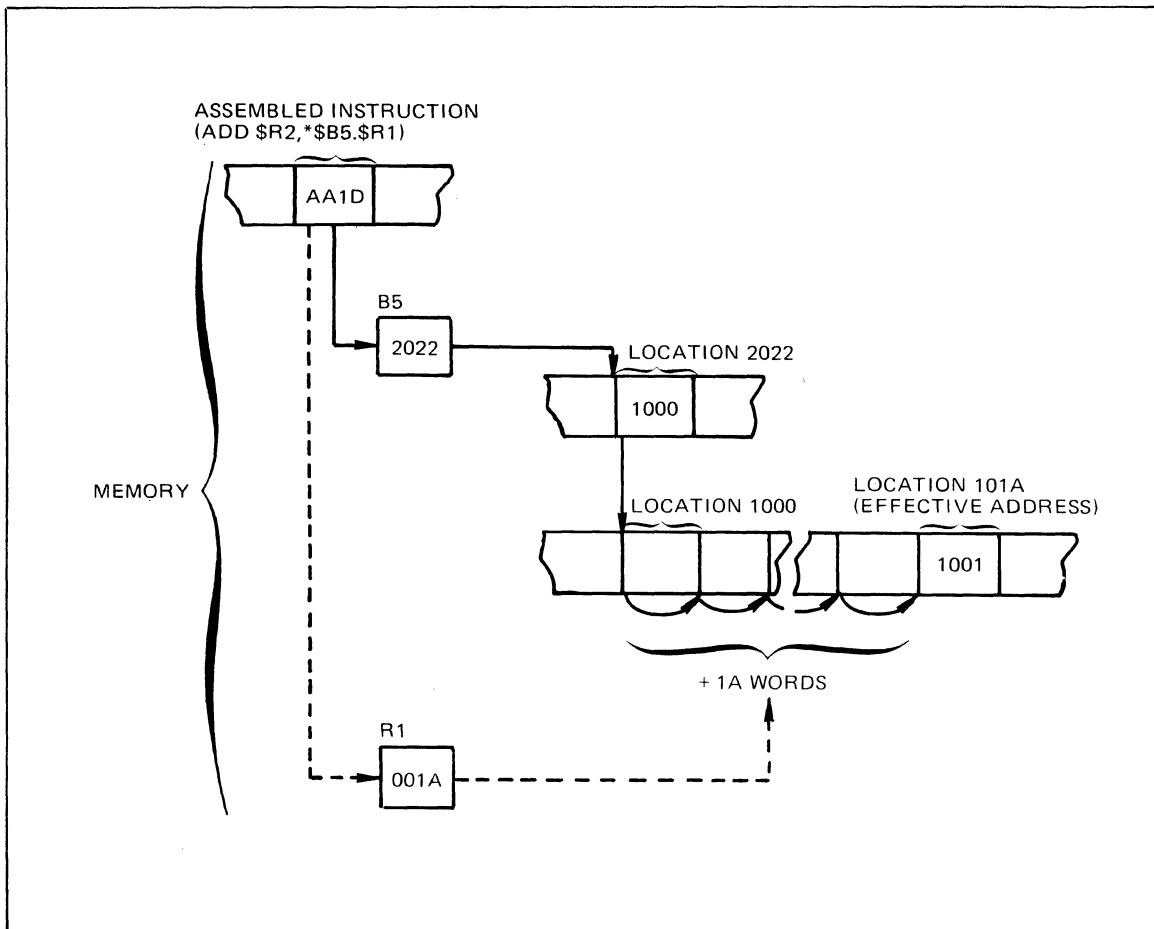


Figure 5-12. Indexed Indirect B-Relative Addressing

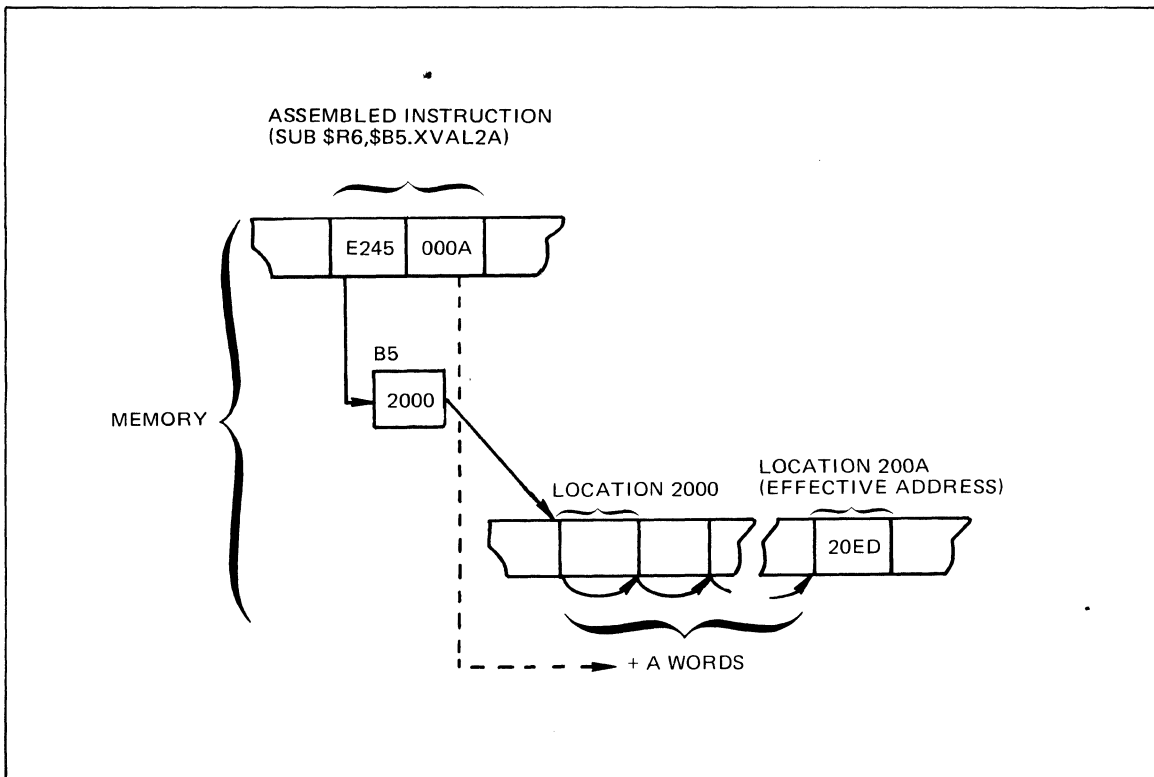


Figure 5-13. Direct B-Relative Plus Displacement Addressing

INDIRECT B-RELATIVE PLUS DISPLACEMENT ADDRESSING

This form of effectively addressing adds a displacement value to the contents of the specified base register. Then, the effective address is the contents of the location whose address is derived through this 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.

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., 400D) 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.

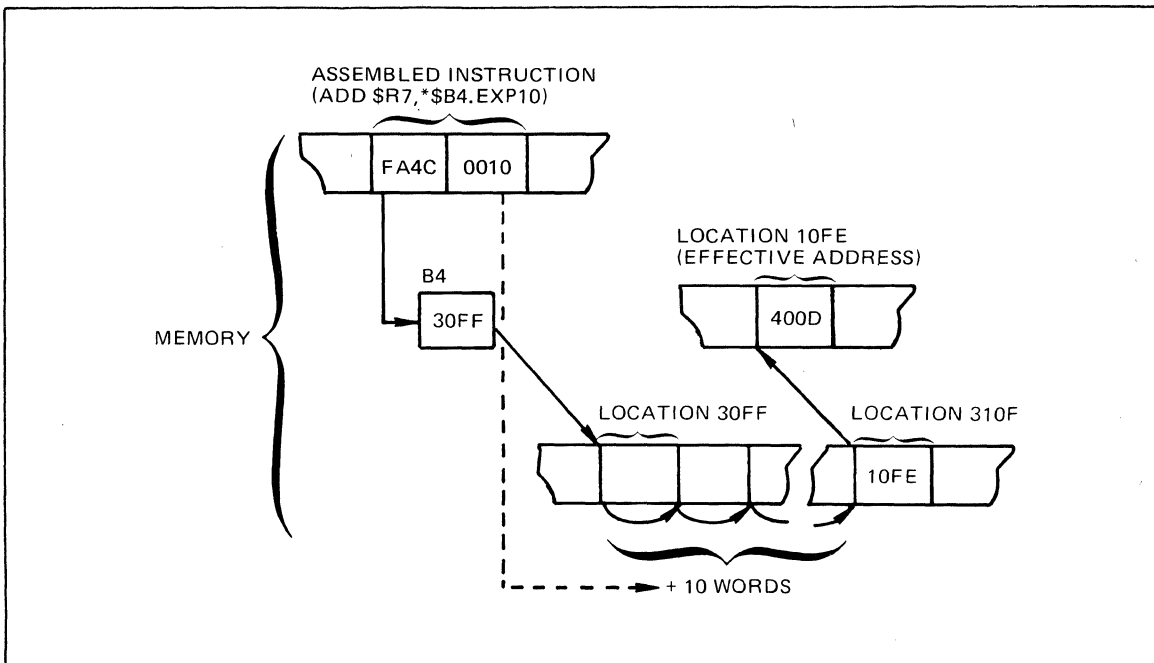


Figure 5-14. Indirect B-Relative Plus Displacement Addressing

DIRECT B6-RELATIVE PLUS LOCAL COMMON BLOCK PLUS DISPLACEMENT ADDRESSING

In this form of addressing, the effective address is computed by adding a specified value to the contents of base register \$B6. This addressing form assumes that \$B6 contains the address of the combined \$LCOMW local common blocks. For information on the loading of \$B6, see the *Program Execution and Checkout* manual. The value that is added to the contents of \$B6 is assumed to be an offset value (before adjustment by the Linker) into the local common block, \$LCOMW.

Example:

```
TEN      EQU      10
$LCOMW  LCOMM    300
          ORG     $LCOMW+10
          DC     100
```

In this example, suppose that the constant 100 which is contained in the eleventh word of the local common block, \$LCOMW, is to be loaded into data register \$R1. If at execution time, \$B6 contains the address of the combined \$LCOMW local common blocks, then either of the following instructions will accomplish the desired result.

```
LDR $R1,$B6.$LCOMW+TEN
LDR $R1,$B6.$LCOMW+10
```

Figure 5-15 illustrates how this form of addressing generates an effective address when stored in memory.

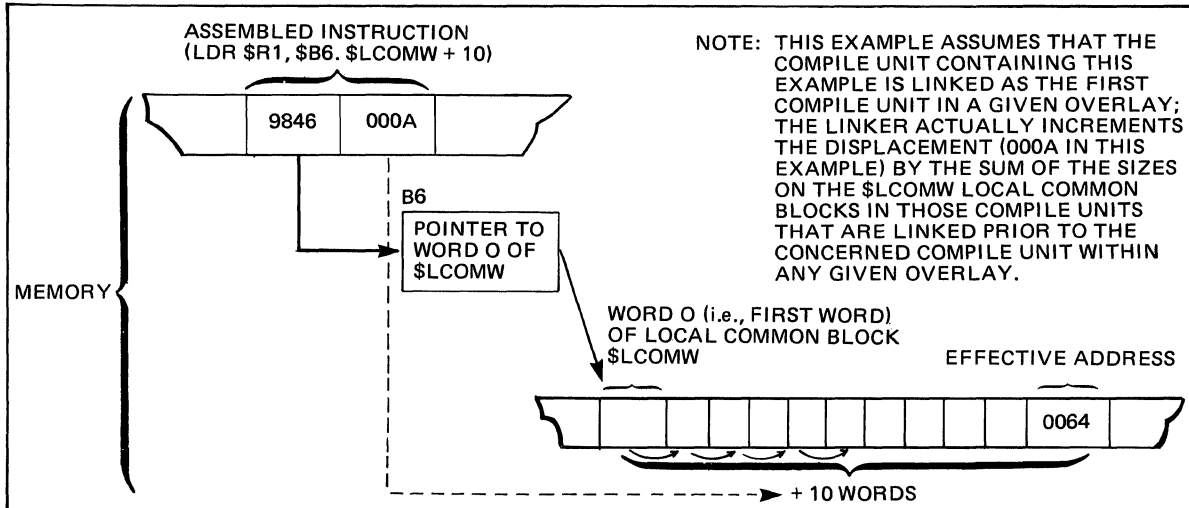


Figure 5-15. Direct B6-Relative Plus Local Common Block Plus Displacement Addressing

INDIRECT B6 - RELATIVE PLUS LOCAL COMMON BLOCK PLUS DISPLACEMENT ADDRESSING

In this form of addressing, the effective address is specified by the contents of the location computed by effectively adding a value to the contents of base register \$B6. This addressing form assumes that \$B6 contains the address of the combined \$LCOMW local common blocks. For information on the loading of \$B6, see the *Program Execution and Checkout* manual. The value that is added to the contents of \$B6 is assumed to be an offset value (before adjustment by the Linker) into the local common block, \$LCOMW.

Example:

```
$LCOMW  LCOMM  300
          ORG    $LCOMW
          DC    <CONST
          ORG    $LCOMW+20
CONST    DC    100
```

In this example, assume that the constant 100 which is contained in the 21st word of the local common block, \$LCOMW, is to be loaded into data register \$R1, and that the address of the constant is known to be in word zero of the local common block. If at execution time, \$B6 contains the address of the local common block, then the following instruction will accomplish the desired result.

```
LDR $R1,*$B6.$LCOMW
```

Figure 5-16 illustrates how this form of addressing generates an effective address when stored in memory.

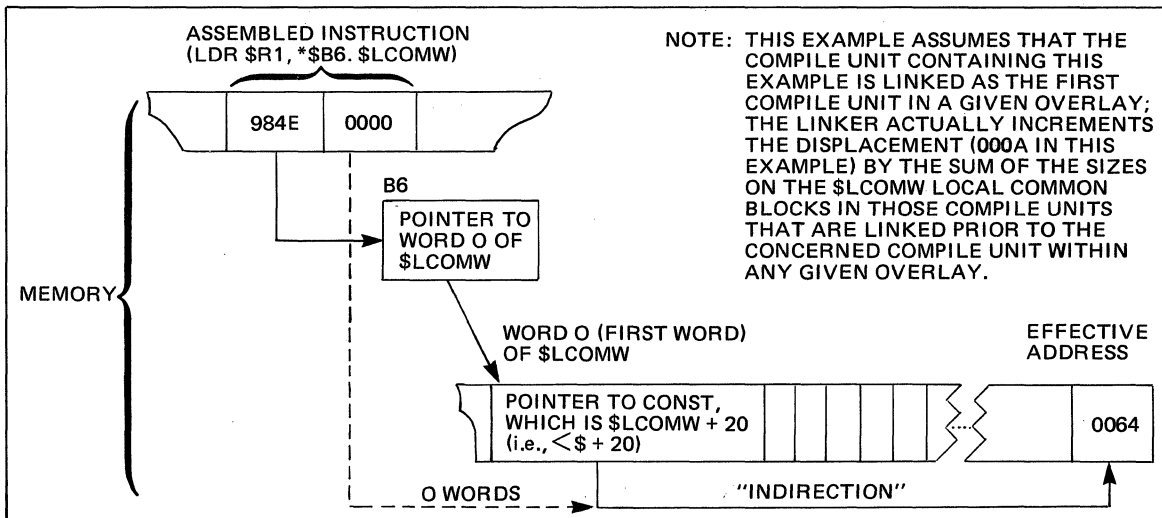


Figure 5-16. Indirect B6-Relative Plus Local Common Block Plus Displacement Addressing

B-RELATIVE PUSH ADDRESSING

This form of B-relative addressing causes the contents of the specified base register to be decremented 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. The B register is decremented by:

- One for all instructions accessing one-bit, one-byte, or one-word operands.
- Two for all instructions accessing double-word operands.
- Four for all instructions accessing quadruple-word operands.
- One for SAF configurations or two for LAF configurations for the LDB, STB, SWB, CMB, and CMN instructions.

Note:

LAB is an instruction accessing a one-word operand.

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 5-17 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.

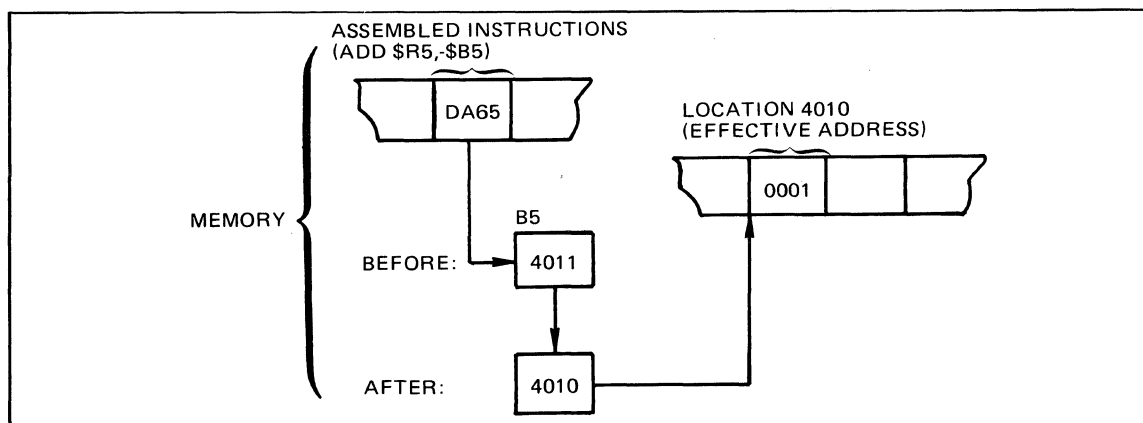


Figure 5-17. B-Relative Push Addressing

B-RELATIVE POP ADDRESSING

This form of B-relative addressing causes the contents of the specified base register to be incremented after the effective address is formed. The old address that was in the register is the effective address of the location or data to be used in the operation. The B register is incremented by:

- One for all instructions accessing one-bit, one-byte, or one-word operands.
- Two for all instructions accessing double-word operands.
- Four for all instructions accessing quadruple-word operands.
- One for SAF configurations or two for LAF configurations for the LDB, STB, SWB, CMB, and CMN instructions.

Note:

LAB is an instruction accessing a one-word operand.

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 5-18 shows how the instruction above is stored in memory and how it derives an effective address.

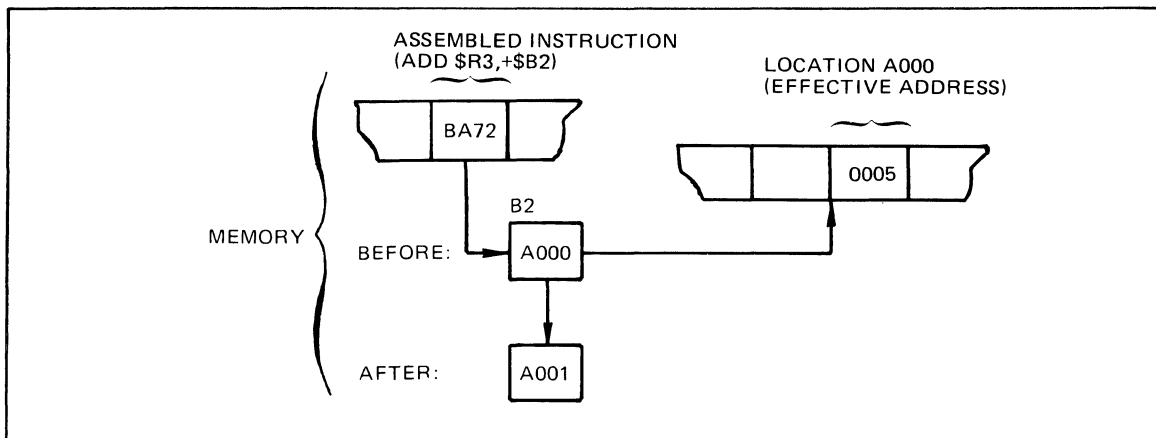


Figure 5-18. B-Relative Pop Addressing

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 effectively 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 base registers \$B1, \$B2, or \$B3 can be specified in the address expression. Figure 5-19 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.

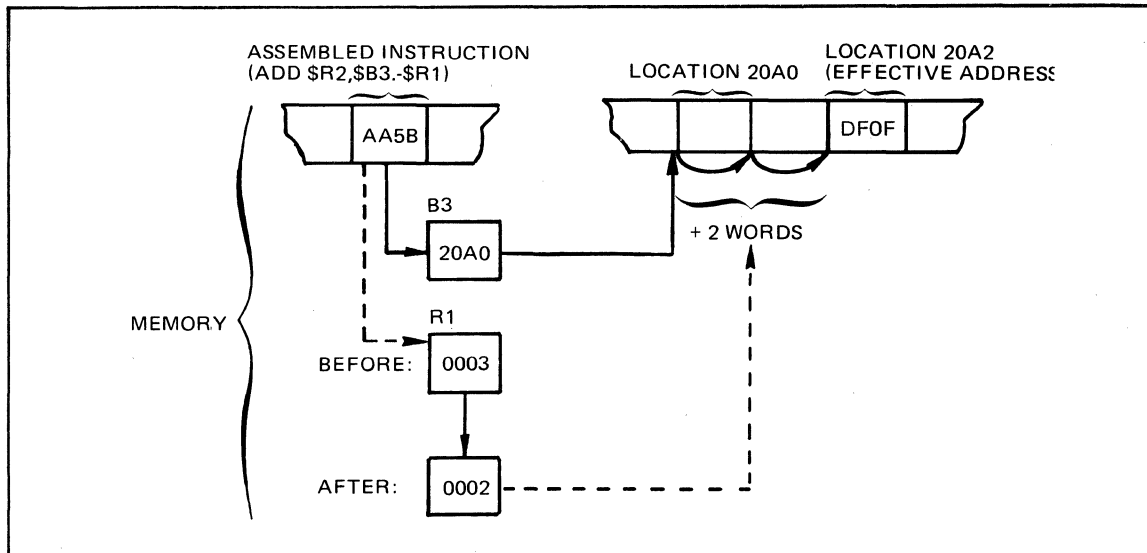


Figure 5-19. Indexed B-Relative Push Addressing

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.

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 effectively 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 base 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 5-20 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.

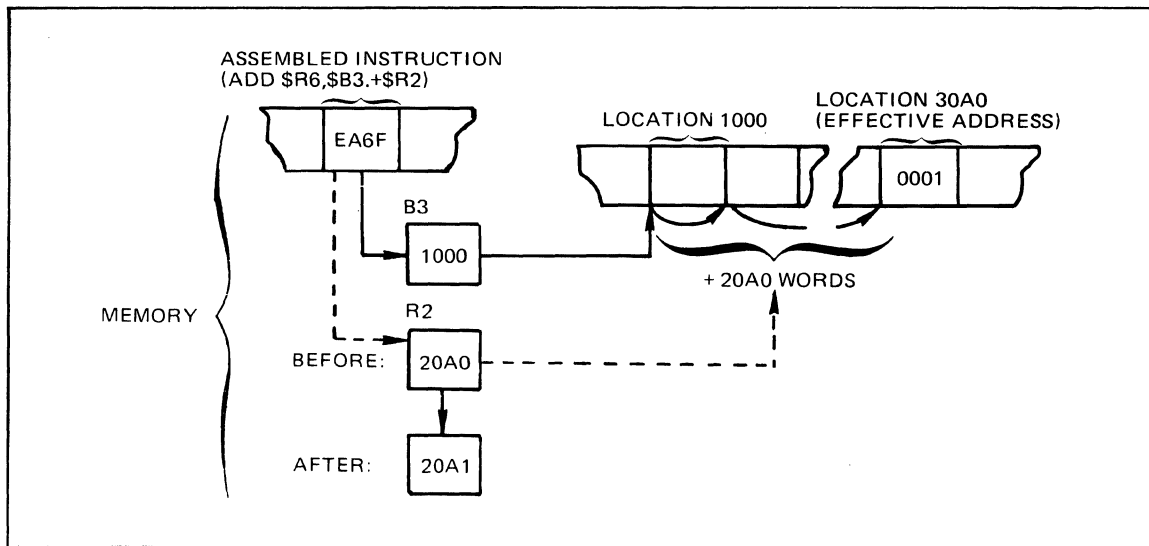


Figure 5-20. Indexed B-Relative Pop Addressing

SHORT DISPLACEMENT ADDRESSING

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

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

When this form of addressing is used, the referenced location must be within one of the ranges -64 words to -1 word or $+2$ 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 5-21 illustrates how the above example is stored in memory and how it derives the effective address of the location to be branched to.

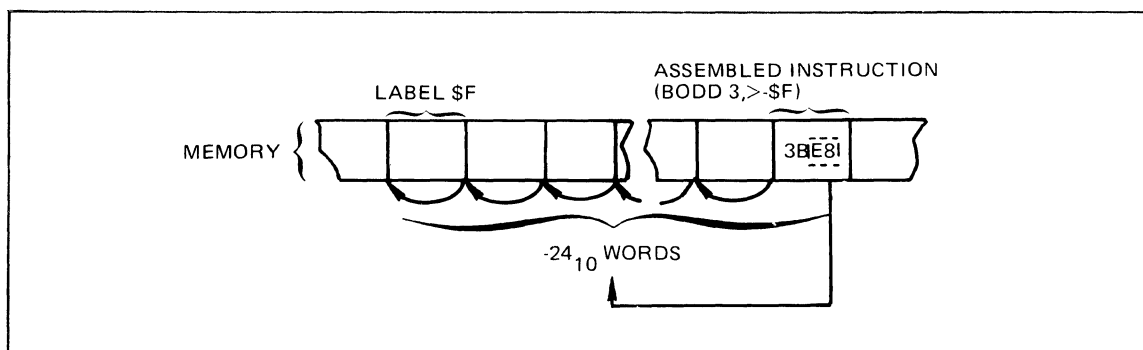


Figure 5-21. Short Displacement Addressing

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:

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

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:

```
IOΔΔ$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 5-22 illustrates how the example above is stored in memory and how it derives the effective address of the data.

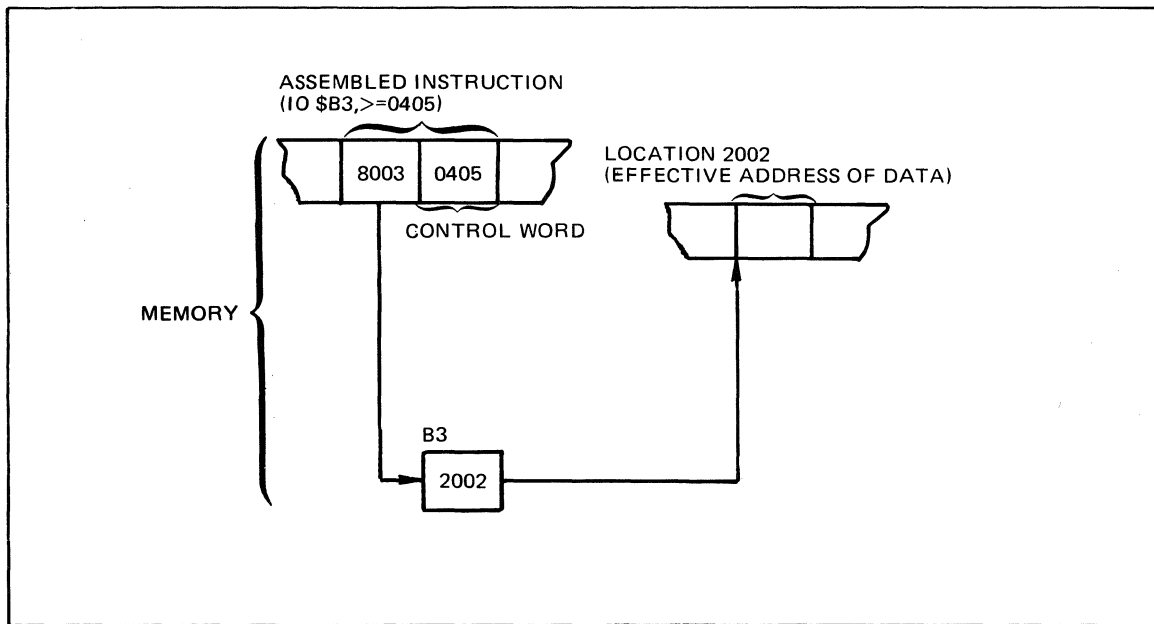


Figure 5-22. Specialized Address Expressions

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 *GCOS 6 MOD 400 System Concepts* manual.) Interrupt vector addressing is specified as follows:

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

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 0, etc.)

Example 1:

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

Figure 5-23 illustrates how example 1 above locates the desired memory word and places it into R1.

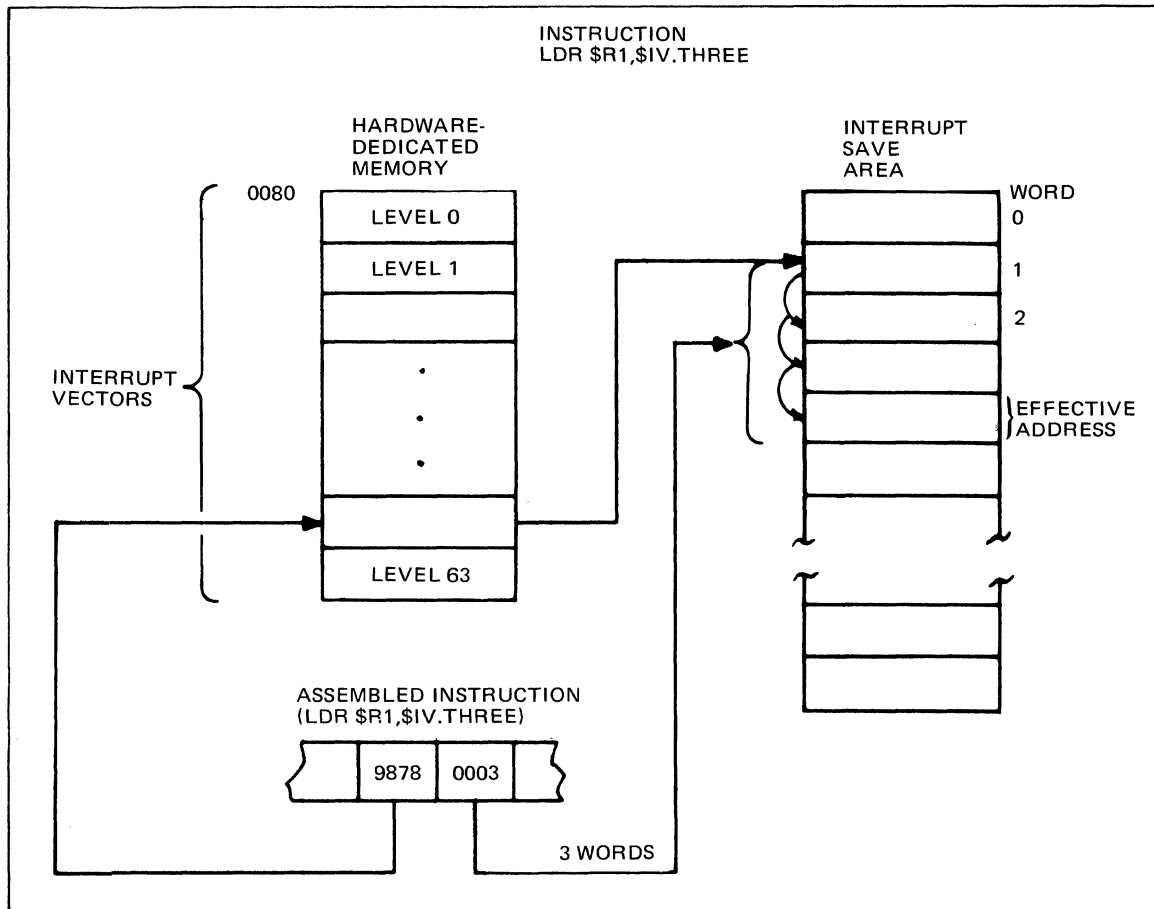


Figure 5-23. Interrupt Vector Addressing

Example 2:

`LDB $B1,$IV.-$AF`

In Example 2, the address of the first trap save area is loaded into address register `$B1`.

INDEXED ADDRESSING CONSIDERATIONS

Table 5-1 shows available modes of indexed addressing.

TABLE 5-1. INDEXED ADDRESSING MODES

Mode	Format
Indexed direct IMA	<label>.\$Rx
Indexed indirect IMA	* <label>.\$Rx
Indexed direct B-relative addressing	\$Bn.\$Rx
Indexed indirect B-relative addressing	*\$Bn.\$Rx
Indirect B-relative push addressing	\$Bk.-\$Rx
Indexed B-relative pop addressing	\$Bk.+\$Rx

\$Bn is any B-register, B1 through B7
 \$Bk is a B-register, B1 through B3
 \$Rx is an R-register, R1 through R3

In simple cases for all indexing modes, the contents of index register `$Rx` are used unchanged in computing the effective address. In other cases certain instructions and/or machine operating modes may cause the value in the index register to be multiplied by 2 or 4 before being used to

compute the effective address. The contents of the index register is not changed after execution of the instruction.

The following subsections describe how to establish this multiplication factor and which instructions and machine modes use a factor other than 1.

All instructions not specifically discussed below have an indexing multiplier of 1.

ESTABLISHING A MULTIPLICATION FACTOR

Not all operands occupy only one word in memory. For example, LAF-configuration addresses occupy two words; LDI and SDI instructions manipulate 2-word operands; and scientific instructions may manipulate 2- or 4-word operands.

Assume a table in memory consists of LAF addresses. The first entry in the table (entry 0) starts at word 0, but the third entry in the table (entry 2) starts at word 4. In general, entry n starts at word $2n$ for LAF mode addresses. With 4-word scientific operands, entry n starts at word $4n$.

Level 6 hardware accepts the entry number n as an indexing quantity. It multiplies that by the factor (1, 2, or 4) appropriate to the operand being operated on, then uses the result to find the word in a table where the operand begins. Thus, the general rule is to (1) determine the size of the operand that the instruction will operate on *when it executes* (i.e., one, two, or four words in memory), and (2) use that size as the multiplication factor for the index register's contents for this instruction in this mode.

AID, SID, LDI, AND SDI INSTRUCTIONS

Regardless of operating modes, the AID, SID, LDI, and SDI instructions operate on 2-word operands; thus their indexing multiplier is 2.

B-REGISTER INSTRUCTIONS IN LAF CONFIGURATION

On a 6/40 or 6/50 model in LAF configuration, addresses occupy two words in memory. Thus the indexing multiplier for the following B-register manipulating instructions in LAF configuration is 2: LDB, STB, SWB, CMB, and CMN. In SAF configuration the multiplier is 1. (Note that the LAB instruction is not in this category; its indexing multiplier is always 1.)

SCIENTIFIC INSTRUCTIONS

On a 6/40 or 6/50 model with the SIP hardware option or any model with the SIP double-precision simulator, the hardware register M4 contains Scientific Instruction mode information, for each of the three scientific accumulators, that includes the size of its operands in memory, i.e., either two or four words. This size is also the indexing multiplier for any Scientific Instruction that references memory.

Register M4 defines operand size separately for each accumulator. Thus the same scientific instruction (e.g., SLD) may have a different operand size and indexing multiplier in different usages within the same program. For example, the multiplier might be 2 when dealing with SA1, and 4 when dealing with SA3.

Register M4, together with the operand size and indexing multiplier, can be dynamically changed.

BIT/BYTE MANIPULATING INSTRUCTIONS

The preceding discussions apply also to bit (e.g., LB) and byte (e.g., LDH) manipulating instructions. In these cases indexing multipliers of $1/16$ or $1/2$, complete with possible remainders, are conceivable. For example an index value of 37 in an LBF instruction would compute to word 2, bit 5 from the base, and be correct. However, it is more practical to consider the index value as simply representing the actual number of bits or bytes to be offset from the base. As a result, an alternative to the general rule given under "Establishing a Multiplication Factor" above is: In indexed addressing, measure the index value in units of the size of the operand with which it is operating, whether the operand unit is one bit or four words.

ASSEMBLY LANGUAGE INSTRUCTIONS

The remainder of this section lists (alphabetically) and describes the assembly language instructions for the Central Processing Unit (CPU). Assembly language instructions for the Commercial Processor and the Scientific Instruction Processor (SIP) are given in Sections 6 and 7 respectively. 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), 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 formats, all assembly language instructions can be labeled.

ACQ

Instruction:

Acquire stack space

Type:

GE

Source Language Format:

$$\Delta ACQ \Delta \left\{ \begin{array}{l} \$Bn \\ X'n' \\ n \end{array} \right\}, \$Rn$$

Description:

This stack instruction acquires an additional frame, of the size specified by the contents of $\$Rn$, from the currently available stack space. $\$Bn$ is set to point to this newly acquired frame.

If the size specified by Rn is such that the currently available stack space is exceeded, a trap to trap vector 10 occurs.

Stack instructions are double-word instructions with the following characteristics.

- A common first word.
- Bits 0 through 8 and bit 12 of the second word contain zeros.

If bits 0 through 8 and bit 12 of the second word are not zero, the result is a trap to trap vector 16.

Bits 9 through 11 of the ACQ instruction specify the register $\$Rn$ bits 13 through 15 specify register $\$Bn$.

This instruction is executable only on 6/40 and 6/50 models.

ADD

ADD

Instruction:

Add Contents to R-register

Type:

DO

Source Language Format:

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

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:

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

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

- 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.
- 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:

$$\Delta\text{ADV}\Delta \left\{ \begin{array}{l} \$R_n \\ X'_n \\ n \end{array} \right\} , [=] \left\{ \begin{array}{l} \text{internal-value-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right\}$$

Description:

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

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

- 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.
- If, during the summation, a carry occurs, the C-bit is set to 1; otherwise, it is set to 0.

AID

AID

Instruction:

Add integer double

Type:

SO

Source Language Format:

Δ AID Δ address-expression

Description:

Adds the value of the double-word integer specified by the address expression to the value in the register pair \$R6, \$R7. The result is saved in \$R6 and \$R7, with the most significant part in \$R6 and the least significant part in \$R7.

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

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

If the address expression specifies memory addressing with indexing, the index register is aligned to count double-words relative to the word specified.

If Immediate Operand Addressing is specified, the immediate operand may only use a binary integer constant (which is sign extended to 32 bits by the Assembler), a double precision fixed-point constant, or a string constant of exactly two words (i.e., four bytes or 32 bits).

If $=\$Rn$ is used, only $=\$R3$ (adds the contents of R2 and R3 into R6 and R7 respectively), $=\$R5$ (adds the contents of R4 and R5 into R6 and R7, respectively), or $=\$R7$ (doubles the value contained in R6 and R7) may be used.

If a carry occurs, the C-bit of the I-register is set to 1, else it is set to 0.

If overflow occurs, the OV-bit if the I-register is set to 1, else it is set to 0.

This instruction is executable only on 6/40 and 6/50 models.

AND

Instruction:

AND contents with R-register

Type

DO

Source Language Format:

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

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:

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

The following chart illustrates the result of performing a logical AND operation on bits:

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

ANH

ANH

Instruction:

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

Type:

DO

Source Language Format:

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

Description:

A logical AND operation is performed on 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:

- Register Addressing ($=\$Rn$): The rightmost byte of the register is selected.
- Memory Addressing *Without* Indexing or Immediate Operand Addressing: The leftmost byte of the word at the designated memory address is selected.
- 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 following chart illustrates the result of performing a logical AND operation on bits:

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

ASD

Instruction:

Activate Segment Descriptor

Type:

GE

Source Language Format:

Δ ASD

Description:

The MMU segment descriptor whose first and second word, respectively, are contained in \$R6 and \$R7 is moved into the MMU segment descriptor specified by the Effective Address contained in \$B5.

The ASD instruction is privileged.

Segment descriptors are entries in physical memory allocation tables which relate physical memory allocations to specific process usages. Segment descriptors are used to specify the resources available to a particular process under the memory protection subsystem.

This instruction is available only with systems that have a Memory Management Unit.

B

B

Instruction:

Branch unconditionally

Type:

BI

Source Language Format:

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

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 sequences 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 either, but not both, of the G- or U- bits of the I-register equals 1.

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

BAGE

Instruction:

Branch if algebraically greater than or equal to

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the U- and L-bits of the I-register are *both* 0 or *both* 1.

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 either, but not both, of the U- or L-bits of the I-register equals 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.

BALE

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 if the G- and U-bits of the I-register are *both* 0 or *both* 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.

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

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:

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

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

BCT

Instruction:

Branch if carry

Type:

BI

Source Language Format:

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

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:

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

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 not equal 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.

BE

BE

Instruction:

Branch if equal

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if both the G- and L-bits of the I-register are 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.

BEVN

Instruction

Branch if R-register even

Type:

BR

Source Language Format:

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

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

BEZ

Instruction:

Branch if R-register equal to 0

Type:

BR

Source Language Format:

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

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:

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

Description:

Branches to the location specified in the operand if the G-bit of 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.

BGE

BGE

Instruction:

Branch if greater than or equal to

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the L-bit of 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.

BGEZ

Instruction:

Branch if R-register greater than or equal to 0

Type:

BR

Source Language Format:

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

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

BGZ

Instruction:

Branch if R-register greater than 0

Type:

BR

Source Language Format:

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

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:

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

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

BIOF

Instruction:

Branch if I/O indicator false

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the 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:

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

Description:

Branches to the location specified in the operand if the 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

BL

Instruction:

Branch if less than

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the L-bit of 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.

BLE

Instruction:

Branch if less than or equal to

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the G-bit of 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.

BLEZ

BLEZ

Instruction:

Branch if R-register equal to or less than 0

Type:

BR

Source Language Format:

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

Description:

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

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

Description:

Branches to the location specified in the second operand if the R-register identified in the first operand contains a 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

BNE

Instruction:

Branch if not equal

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if either, but not both, the G-bit or the L-bits of the I-register are 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.

BNEZ

Instruction:

Branch if R-register not equal to 0

Type:

BR

Source Language Format:

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

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

BNOV

Instruction:

Branch if no R-register overflow

Type:

BI

Source Language Format:

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

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:

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

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

BOV

Instruction:

Branch if R-register overflow

Type:

BI

Source Language Format:

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

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:

Δ BRK Δ

Description:

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

BSE

BSE

Instruction:

Branch if signs equal

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the U-bit in the I-register is 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.

BSU

Instruction:

Branch if signs unlike

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the U-bit in the I-register is equal 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

CAD

Instruction:

Add carry bit to contents

Type:

SO

Source Language Format:

Δ CAD Δ address-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:

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

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

- If a carry occurs during the operation, the C-bit is set to 1; otherwise, it is set to 0.
- 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.

CL

Instruction:

Clear

Type:

SO

Source Language Format:

 $\Delta CL \Delta$ address-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:

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

CLH

CLH

Instruction:

Clear half-word

Type:

SO

Source Language Format:

Δ CLH Δ address-expression

Description:

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

The byte to be cleared is determined by the format of the address expression, as follows:

- If the address expression specifies Register Addressing ($=\$R_n$), 0's are stored in the rightmost byte of the register.
- If the operand specifies Memory Addressing *without* indexing, or Immediate Operand Addressing 0's are stored in the leftmost byte of the word found at the specified location.
- If the operand specifies 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:

- = $\$B_n$ register addressing
- Short displacement addressing
- Specialized addressing

CMB

Instruction:

Compare contents to B-register

Type:

DO

Source Language Format:

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

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:

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

Immediate operand addressing with a value expression.

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

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

CMH

CMH

Instruction:

Compare half-word (byte) to R-register

Type:

DO

Source Language Format:

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

Description:

Compares the contents of the R-register identified in the first operand to the contents of the 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:

- Register Addressing ($=\$R_n$): The rightmost byte of the register is selected.
- Memory Addressing *Without* Indexing or Immediate Operand Addressing: The leftmost byte of the word at the designated memory address is selected.
- 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:

$=\$B_n$ } register addressing
 $=\$S_n$ }
Short displacement addressing
Specialized addressing

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

- 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.
- 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.
- 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:

Δ CMN Δ address-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:

- 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.
- The L-bit is set to 0.
- 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:

=\$Rn } register addressing
=\$Sn }

Short displacement addressing

Specialized addressing

Immediate operand addressing with a value expression.

CMR

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:

- 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.
- 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.
- 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:

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

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.

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

- 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.
- 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.
- 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

CMZ

Instruction:

Compare to 0

Type:

SO

Source Language Format:

Δ CMZ Δ address-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:

$=\$B_n$ } register addressing
 $=\$S_n$ }
Short displacement addressing
Specialized addressing

Note that, the $\$B_n.\$R1$, $\$B_n.\$R2$, or $\$B_n.\$R3$ form of addressing can be used by this instruction to cause a trap for the purpose of sizing main memory.

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

- If the contents of the specified location do not equal 0, the G-bit is set to 1; otherwise, it is set to 0.
- The L-bit is set to 0.
- If the first bit of the specified location equals 1, the U-bit is set to 1; otherwise, it is set to 0.

CNFG

Instruction:

Configure

Type:

GE

Description:

Causes the CPU to perform an input/output operation and a scan.

The effect of the input/output operation is equivalent to that produced by the execution of the following instruction:

$$IO = \$R7, = \$R6$$

Register R6 contains the command word (CH,F) that specifies a channel number and a function code. (See the IO instruction, for format of the command word.) The function code must designate an output function. The command word and the word contained in register R7 are sent to the addressed IO channel.

A predetermined list of channel numbers is scanned to determine the presence or absence of optional processors. The results of the scan are used to update internal CPU firmware/hardware flags that direct instruction execution; i.e., trap or execute by an optional processor. An identical scan is performed automatically when the system is powered up or initialized.

This privileged instruction, which is executable only on 6/40 and 6/50 models, is normally used to control the configuration of option boards; e.g., the Commercial Processor and the SIP.

CPL

CPL

Instruction:

Complement

Type:

SO

Source Language Format:

Δ CPL Δ address-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:

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

The following chart illustrates the result of logically one's complementing bits:

Operand bit:	1	0
Result:	0	1

DAL

Instruction:

Double-shift arithmetic-left

Type:

SHL

Source Language Format:

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

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.

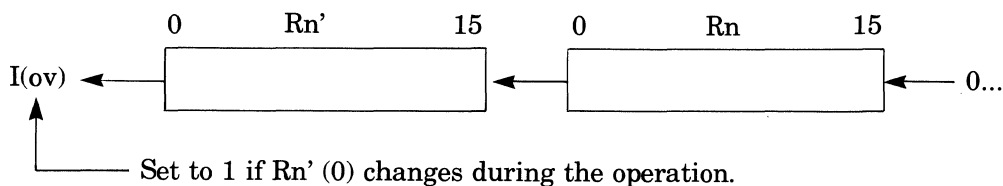
The internal value expression must be ≥ 0 and ≤ 31 .

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:

- If the contents of bit 0 in the even-numbered R-register changes at any time during the operation, the OV-bit is set to 1; otherwise, it is set to 0.

The following illustrates the operation of the DAL instruction:



DAR

DAR

Instruction:

Double-shift arithmetic-right

Type:

SHL

Source Language Format:

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

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 bit positions vacated by the shift are filled with the sign value originally contained in bit 0.

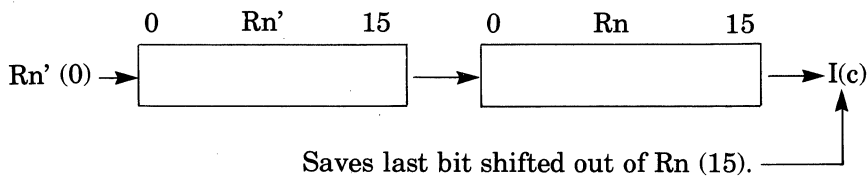
The internal value expression must be ≥ 0 and ≤ 31 .

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:

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

The following illustrates the operation of the DAR instruction:



DCL

Instruction:

Double-shift closed-left

Type:

SHS

Source Language Format:

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

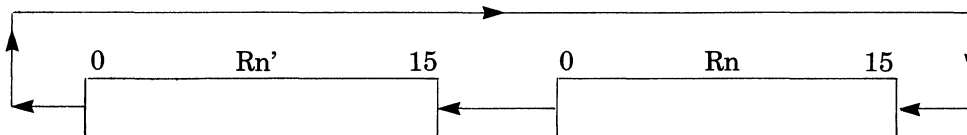
Description:

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 ≥ 0 and ≤ 15 .

Note that the DCL instruction is short shift.

The following illustrates the operation of the DCL instruction:



DCR

DCR

Instruction:

Double-shift closed-right

Type:

SHS

Source Language Format:

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

Description:

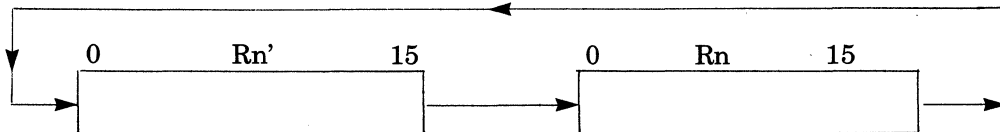
Shifts the contents of the even-odd R-register pair (i.e., R2 and R3 R4 and R5, R6 and R7) identified in the first operand right the number of bit positions specified by the internal value expression in the second operand. The bits shifted out of the odd-numbered R-register are placed in the bit positions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

Note that the DCR instruction is short shift.

The following illustrates the operation of the DCR instruction:



DEC

Instruction:

Decrement

Type:

SO

Source Language Format:

 Δ DEC Δ address-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:

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

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

- If the decrementation causes a carry to occur (i.e., the value being decremented was not zero), the C-bit is set to 1; otherwise, it is set to 0.
- If the value being decremented was -32768 (-2^{15}), I(OV) is set to 1; otherwise, I(OV) is cleared to 0.
- I(B) is set as described above.

DIV

DIV

Instruction:

Divide R-register by contents of location

Type:

DO

Source Language Format:

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

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 quotient is saved in the first operand R-register. The remainder 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 quotient 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 will 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:

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

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.

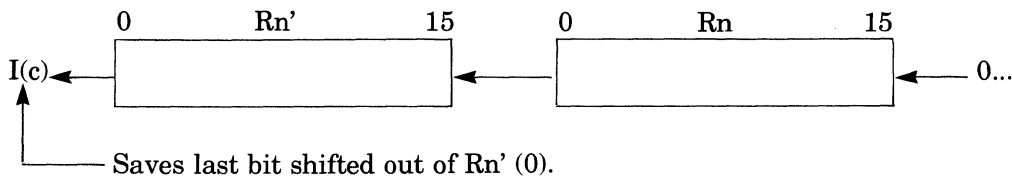
The internal value expression must be ≥ 0 and ≤ 31 .

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:

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

The following illustrates the operation of the DOL instruction:



DOR

DOR

Instruction:

Double-Shift open-right

Type:

SHL

Source Language Format:

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

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.

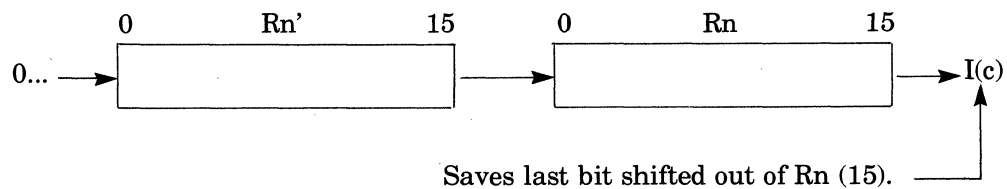
The internal value expression must be ≥ 0 and ≤ 31 .

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:

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

The following illustrates the operation of the DOR instruction:



DQA

Instruction:

Dequeue on address

Type:

GE

Source Language Format:

 Δ DQA

Description:

This instruction unlinks a frame whose PRIORITY word address exactly matches the address contained in register B1. Register B2 points to the LOCK word.

If the instruction is not successfully completed, the carry bit of the indicator register is cleared to zero; otherwise it is set to one and the G- and L-bits indicate the results as follows:

Indicator Register Bit

<i>G</i>	<i>L</i>	<i>Result</i>
0	0	Frame was unlinked
0	1	No match found

Queue instructions can be executed only on model 6/40 and 6/50 systems.

DQH

DQH

Instruction:

Dequeue from head

Type:

GE

Source Language Format:

Δ DQH

Description:

This instruction unlinks the first frame whose priority is equal to or numerically greater than that specified by register R5. Register B2 points to the LOCK word. If the instruction is not completed, the carry bit of the indicator register is cleared to zero. If the instruction is completed:

- The carry bit of the indicator register is set to one
- The pointer to the PRIORITY word of the unlinked frame, if any, is loaded into register B1
- The G- and L- bits of the indicator register indicate the results as follows:

Indicator Register Bits

<i>G</i>	<i>L</i>
0	0
1	0
0	1

Results

Unlinked frame was first whose priority equalled that specified by register R5.

Unlinked frame was first whose priority number was greater than that specified by register R5.

No frame was unlinked; register B1 is unchanged; no frame found whose priority number was equal to or greater than that specified by register R5.

Queue instructions can be executed only on model 6/40 and 6/50 systems.

ENT

Instruction:

Enter

Type:

SO

Source Language Format:

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

Description:

Jumps to the memory location specified by the operand; also, sets the P-bit or the high order bit of the ring field in the S-register, as appropriate, to 0 (i.e., sets the bit to indicate the unprivileged state).

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

HLT

Instruction:

Halt

Type:

GE

Source Language Format:

Δ HLT

Description:

Stops program execution. HLT state is indicated on the control panel. All interrupts are honored.

The P-bit of the S-register must be set to 1, or the ring field of the S-register must be set to 1x, whichever is appropriate; 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 results in a trap to trap vector 13.

A halt instruction on a user level may prevent a lower priority user level from completing a Monitor service operation. The Monitor may be interrupted in a way that causes a system interlock. If user level halts are used during program development, the level specified should be the lowest priority in the system.

INC

Instruction:

Increment

Type:

SO

Source Language Format:

 Δ INC Δ address-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:

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

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

- If the incrementation causes a carry to occur (i.e., the value being incremented was -1), the C-bit is set to 1; otherwise, it is set to 0.
- If the value being incremented was 32767, I(OV) is set to 1; otherwise, it is cleared to 0.
- I(B) is set on as described above.

IO

IO

Instruction:

Input/Output (word)

Type:

IO

Source Language Format:

Δ IO Δ address-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.

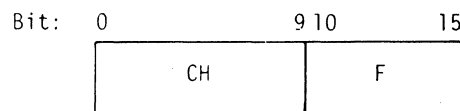
For the first operand, the address expression can take any of the forms described earlier in this section under "Addressing Techniques," except for the following:

= $\$B_n$ } register addressing
= $\$S_n$ }
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:

= $\$B_n$ } register addressing
= $\$S_n$ }
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 *Honeywell Level 6 Minicomputer Handbook*.

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

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

The IO instruction is privileged.

IOH

IOH

Instruction:

Input/output half-word

Type:

IO

Source Language Format:

Δ IOH Δ address-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 $=\$Rn$, 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.

The IOH instruction is privileged.

IOLD

Instruction:

Input/output load

Type:

IO

Source Language Format:

 Δ IOLD Δ address-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:

= \$Bn	} register addressing
= \$Rn	
= \$Sn	
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:

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

The second operand of this instruction must specify the function code 09₁₆ for most controllers.

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:

= \$Bn	} register addressing
= \$Sn	
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 20₁₆ 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 *Honeywell Level 6 Minicomputer Handbook*.

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

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

The IOLD instruction is privileged.

JMP

JMP

Instruction:

Jump

Type:

SO

Source Language Format:

$$\Delta\text{JMP}\Delta \left\{ \begin{array}{l} \text{immediate-memory-address} \\ \text{B-relative-addressing} \\ \text{P-relative-addressing} \\ \text{interrupt-vector-addressing} \end{array} \right\}$$

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:

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

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:

- Register addressing
- Short displacement addressing
- Specialized addressing

NOTE: The Level 6 hardware does not consider LAB to be a base register instruction.

LB

LB

Instruction:

Load bit

Type:

SO

Source Language Format:

$$\Delta LB \Delta \text{address-expression} \left[, \left\{ \begin{array}{l} \text{internal-value-expression} \\ \text{external-value-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right\} \right]$$

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. The second operand must be omitted when the first operand specifies indexing.
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. If the value of the second operand is zero, the contents of \$R1 are used as a mask.

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

LBC

Instruction:

Load bit and complement

Type:

SO

Source Language Format:

$$\Delta LBC \Delta \text{address-expression} \left[\begin{array}{l} \text{internal-value-expression} \\ \text{external-value-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right]$$

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 the one's complement of its value. The second operand must be omitted when the first operand specifies indexing.
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. If the value of the second operand is zero, the contents of \$R1 are used as a mask. Upon completion of the instruction, each bit of the first operand which was checked is set to the one's complement of its original value.

LBF

LBF

Instruction:

Load bit and set false

Type:

SO

Source Language Format:

$$\Delta\text{LBF}\Delta\text{address-expression} \left[\begin{array}{l} \text{internal-value-expression} \\ \text{external-value-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right]$$

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. The second operand must be omitted when the first operand specifies indexing.
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. If the value of the second operand is zero, the contents of \$R1 are used as a mask. Upon completion of the instruction, each bit of the first operand which was checked 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:

= \$Bn } register addressing
= \$Sn }
Short displacement addressing
Specialized addressing
Immediate operand addressing

LBS

Instruction:

Load bit and swap

Type:

SO

Source Language Format:

$$\Delta\text{LBS}\Delta\text{address-expression} \left[, \left\{ \begin{array}{l} \text{internal-value-expression} \\ \text{external-value-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right\} \right]$$

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 interchanged with the B-bit of the I-register. The second operand must be omitted when the first operand specifies indexing.
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. If the value of the second operand is zero, the contents of \$R1 are used as a mask. Upon completion of the instruction, each bit of the first operand that was checked is set equal to the original value of the B-bit of the I-register.

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

LBT

LBT

Instruction:

Load bit and set true

Type:

SO

Source Language Format:

$$\Delta\text{LBT}\Delta\text{address-expression} \left[, \left\{ \begin{array}{l} \text{internal-value-expression} \\ \text{external-value-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right\} \right]$$

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 1. The second operand must be omitted when the first operand specifies indexing.
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. If the value of the second operand is zero, the contents of \$R1 are used as a mask. Upon completion of the operation, the bit(s) checked in accordance with the mask is (are) set to 1.

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:

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

LDB

Instruction:

Load B-register

Type:

DO

Source Language Format:

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

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:

= $\$R_n$ } register addressing
 = $\$S_n$ }

Short displacement addressing

Specialized addressing

Immediate operand addressing with a value expression.

LDH

LDH

Instruction:

Load half-word (byte) into R-register

Type:

DO

Source Language Format:

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

Description:

Loads the contents of the location specified in the address expression, as described below, into the R-register identified in the first operand:

- 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.
- 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.
- 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:

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

LDI

Instruction:

Load double-word integer

Type:

SO

Source Language Format:

 Δ LDI Δ address-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:

= $\$B_n$	} register addressing
= $\$S_n$	
	Short displacement addressing
	Specialized addressing

If the address expression specifies memory addressing with indexing, the index register is aligned to count double-words relative to the word specified.

If Immediate Operand Addressing is specified, the immediate operand may only use a binary integer constant (which is sign extended to 32 bits by the Assembler), a double precision fixed-point constant, or a string constant of exactly two words (i.e., four bytes or 32 bits).

If = $\$R_n$ is used, only = $\$R_3$ (loads the contents of R2 and R3 into R6 and R7, respectively) or = $\$R_5$ (loads the contents of R4 and R5 into R6 and R7, respectively) and = $\$R_7$ may be used.

LDR

LDR

Instruction:

Load R-register

Type:

DO

Source Language Format:

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

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:

- =\$Bn } register addressing
- =\$Sn }

Short displacement addressing
Specialized addressing

LDT

Instruction:

Load stack address register

Type:

GE

Source Language Format:

$$\Delta LDT \Delta \begin{pmatrix} \$Bn \\ X'n' \\ n \end{pmatrix}$$

Description:

Loads the T register with the address contained in \$Bn.

Stack instructions are double-word instructions with the following characteristics:

- A common first word.
- Bits 0 through 8 and bit 12 of the second word contain zeros.

If bits 0 through 8 and bit 12 of the second word are not zero, the result is a trap to trap vector 16. Register \$Bn is specified in bits 13 through 15 of the second word of the LDT instruction.

Stack instructions can be executed only on model 6/40 and 6/50 systems.

*

LDV

LDV

Instruction:

Load value

Type:

SI

Source Language Format:

$$\Delta LDV \Delta \left\{ \begin{array}{l} \$R_n \\ X'n' \\ n \end{array} \right\}, [=] \text{internal-value-expression}$$

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

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:

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

The P-bit in the S-register must be set to 1 or the ring field in the S-register must be 1x, as appropriate, (i.e., the central processor must be in the privileged state) for this instruction to be executed. The unprivileged use of a privileged operation is signified by a trap to trap vector 13. (Traps and trap handling are described in the *System Service Macro Calls* manual.)

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

- 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:

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

Description:

Loads the contents of the location specified in the address expression, as described below, into the R-register identified in the first operand.

- If the address expression specifies $=\$Rn$, the rightmost byte of that R-register is loaded into the R-register specified by the first operand.
- 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.
- 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:

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

LNJ

LNJ

Instruction:

Load B-register and jump

Type:

DO

Source Language Format:

$$\Delta\text{LNJA} \left\{ \begin{array}{l} \$Bn \\ X'n' \\ n \end{array} \right\}, \left\{ \begin{array}{l} \text{P-relative-address} \\ \text{immediate-memory-address} \\ \text{B-relative-address} \\ \text{interrupt-vector-addressing} \end{array} \right\}$$

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

LRDB

Instruction:

Load Remote Descriptor Base Register

Type:

GE

Source Language Format:

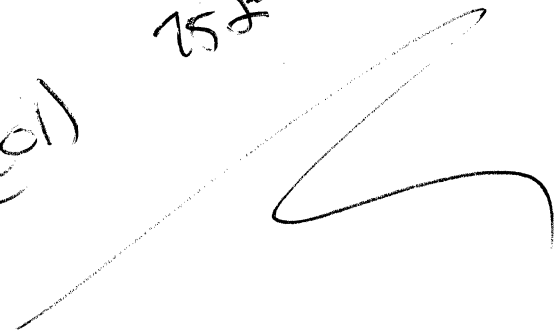
ΔLRDB

Description:

Loads the contents of register B3 into the Remote Descriptor Base Register.

This instruction is executable only on 6/40 and 6/50 models.

*(2011) Mary Anne called
757-8432*



MCL

MCL

Instruction:

Call monitor via trap

Type:

GE

Source Language Format:

ΔMCL

Description:

Calls monitor by a trap to trap vector 1.

MLV

Instruction:

Multiply by value

Type:

SI

Source Language Format:

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

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:

- 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 and the operation is not performed (the first operand R-register is unchanged); otherwise, it is set to 0.

MMM

MMM

Instruction:

Memory to Memory Move

Type:

GE

Source Language Format:

Δ MMM

Description:

The number of bytes of memory specified in $\$R6$ are moved from one location to another. The address of the first byte to be moved is identified by $\$B2 + \$R2$, where $\$R2$ contains a signed byte displacement from $\$B2$. The address of the first byte of the receiving field is identified by $\$B3 + \$R3$ where $\$R3$ contains a signed byte displacement from $\$B3$.

If the sending and receiving fields overlap, the operation is only valid if the receiving field's effective address is less than the effective address of the sending field (i.e., a left shift of n bytes is permitted, while a right shift of overlapping fields is undefined).

The values in the registers are subject to the following requirements:

$\$R6 \geq 0$

$-32768 \leq \$R2 + \$R6 \leq 32767$

$-32768 \leq \$R3 + \$R6 \leq 32767$

Values in the registers which do not meet the requirements specified above are signified by traps. Trap vector 16 signifies that the value in $\$R6$ is less than zero. Trap vector 15 signifies that either $\$R2 + \$R6$ or $\$R3 + \$R6$ is greater than 32767 or less than -32768 . The values in the registers are not altered.

MMM is interruptable.

Successful completion results in the values in registers $\$B2$ and $\$B3$ remaining unchanged, while the values in registers $\$R2$ and $\$R3$ have each been incremented by the number of bytes specified in $\$R6$, and $\$R6$ equals zero.

The results of abnormal termination are as follows:

- The values in registers $\$B2$ and $\$B3$ are unchanged.
- The values in registers $\$R2$ and $\$R3$ are incremented by the value in $\$R6$ minus the number of bytes not moved.
- The value in register $\$R6$ equals the number of bytes not moved.

This instruction is executable only on 6/40 and 6/50 models.

MTM

Instruction:

Modify or test M-register

Type:

DO

Source Language Format:

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

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:

- 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).
- 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 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.

At completion of the instruction, the B-bit in the I-register is set to 1 if (any of) the tested bit(s) is set to 1; otherwise (or if no bits were tested) the B-bit in the I-register is set to 0.

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:

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

MUL

MUL

Instruction:

Multiply R-register

Type:

DO

Source Language Format:

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

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:

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

NEG

Instruction:

Negate

Type:

SO

Source Language Format:

 Δ NEG Δ Address-expression

Description:

Twos complements the contents of the location or R-register specified in the address expression.

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

- If a carry occurs during the operation (i.e., the number complemented is zero), the C-bit is set to 1; otherwise, it is set to 0.
- 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:

=\$Bn } register addressing
=\$Sn }

Short displacement addressing

Specialized addressing

NOP

NOP

Instruction:

No operation

Type:

BI

Source Language Format:

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

Description:

Performs no operation.

OR

Instruction:

Inclusive OR with R-register

Type:

DO

Source Language Format:

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

Description:

Performs an inclusive OR operation on 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 performing inclusive OR operation on 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:

=\$Bn } register addressing

=\$Sn }

Short displacement addressing

Specialized addressing

ORH

ORH

Instruction:

Half-word (byte) inclusive OR with R-register

Type:

DO

Source Language Format:

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

Description:

An inclusive OR operation is performed on the contents of the R-register identified in the first operand with the contents of the byte specified in the address expression. The result is saved in the first operand R-register.

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:

- Register Addressing ($=\$R_n$): The rightmost byte of the register is selected.
- Memory Addressing *Without* Indexing or Immediate Operand Addressing: The leftmost byte of the word at the designated memory address is selected.
- 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 performing an inclusive OR operation on 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:

$=\$B_n$ } register addressing
 $=\$S_n$ }

Short displacement addressing

Specialized addressing

QOH

Instruction:

Queue on head

Type:

GE

Source Language Format:

 Δ QOH

Description:

This instruction links a new frame into a list before the first frame that has the same priority number or the first frame that has a numerically higher priority number. If no frames in the list have an equal or higher priority number, the new frame becomes the last frame in the list.

- Register B1 points to the PRIORITY word of the frame to be added.
- Register B2 points to the lock word of the list.
- Register R5 contains the priority to be assigned to the new frame. The PRIORITY word will be loaded with the contents of R5.

If the instruction is not successfully completed, the carry bit of the indicator register is cleared to zero; otherwise it is set to one and the G- and L-bits indicate the position at which the frame is linked as shown below.

Indicator Register Bit Position in List

G L

0	0	Before frame with same priority
1	0	Before frame with higher priority number, or as last frame

Attempts to multiply enqueue a frame will cause unspecified results.

Queue instructions can be executed only on Model 6/40 and 6/50 systems.

QOT

QOT

Instruction:

Queue on tail

Type:

GE

Source Language Format:

QOT

Description:

This instruction links a new frame into a list after the last frame that has the same priority number or before the first frame that has a numerically higher priority number. If no frames in the list have an equal or higher priority number, the new frame becomes the last frame in the list.

- Register B1 points to the PRIORITY word of the frame to be added.
- Register B2 points to the lock word of the list
- Register R5 contains the priority to be assigned to the new frame. The PRIORITY word will be loaded with the contents of R5.

If the instruction is not successfully completed, the carry bit of the indicator register is cleared to zero; otherwise it is set to one and the G- and L-bits indicate the position at which the frame is linked as shown below.

Indicator Register Bit Position in List

G L

0	0	After frame with same priority
0	1	Before frame with higher priority number, or as last frame

Attempts to multiply enqueue a frame will cause unspecified results.

Queue instructions can be executed only on Model 6/40 and 6/50 systems.

RLQ

Instruction:

Relinquish stack space

Type:

GE

Source Language Format:

Δ RLQ Δ \$Bn

Description:

This stack instruction releases the most recently acquired stack frame. If the stack is emptied by this instruction, the result is a trap to trap vector 9. If the stack is not emptied, the current length of the stack is adjusted and the base register specified, \$Bn (bits 13 through 15 of the second word of the instruction), is set to point to the new top frame.

Stack instructions are double-word instructions with the following characteristics:

- A common first word.
- Bits 0 through 8 and bit 12 of the second word contain zeros.

If bits 0 through 8 and bit 12 of the second word are not zero, the result is a trap to trap vector 16.

Stack instructions can be executed only on model 6/40 and 6/50 systems.

RSTR

RSTR

Instruction:

Restore context

Type:

SO

Source Language Format:

$$\Delta RSTR \Delta \left\{ \begin{array}{l} \text{immediate-memory-address} \\ \text{B-relative-address} \\ \text{P-relative-address} \\ \text{interrupt-vector-addressing} \end{array} \right\}, \left\{ \begin{array}{l} \text{external-value-label} \\ \text{internal-value-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right\}$$

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:

Δ RTCF

Description:

Disables real-time clock interrupts.

The P-bit in the S-register must be set to 1 or the ring field in the S-register must be set to 1x, as appropriate (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

RTCN

Instruction:

Real-time clock on

Type:

GE

Source Language Format:

Δ RTCN

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 or the ring field in the S-register must be set to 1x, as appropriate, (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 *System Service Macro Calls* manual.

For a detailed description of interrupts, refer to the *Honeywell Level 6 Minicomputer Handbook*.

RTT

Instruction:

Return from trap

Type:

GE

Source Language Format:

Δ RTT

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).

SAL

SAL

Instruction:

Single-shift arithmetic-left

Type:

SHS

Source Language Format:

$$\Delta\text{SAL}\Delta \left\{ \begin{array}{l} \$Rn \\ X'n' \\ n \end{array} \right\}, \text{internal-value-expression}$$

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 0s.

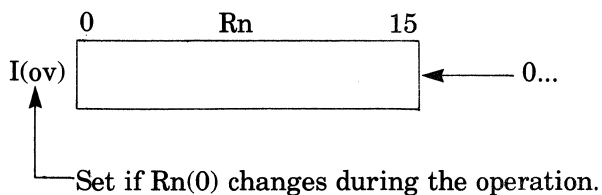
The contents of the I-register are affected as follows:

- If the contents of bit 0 in the R-register change at any time during the operation, the OV-bit is set to 1; otherwise, it is set to 0.

The internal value expression must be ≥ 0 and ≤ 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.

The following illustrates the operation of the SAL instruction:



SAR

Instruction:

Single-Shift arithmetic-right

Type:

SHS

Source Language Format:

$$\Delta\text{SAR}\Delta \left\{ \begin{array}{l} \$Rn \\ X'n' \\ n \end{array} \right\}, \text{internal-value-expression}$$

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.

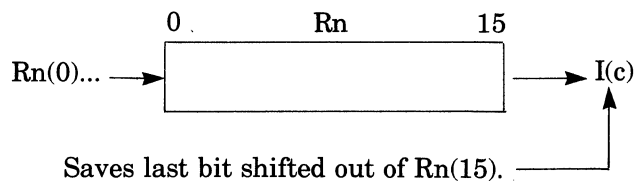
The contents of the I-register are affected as follows:

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

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 12 through 15 of general register R1.

The following illustrates the operation of the SAR instruction:



*

SAVE

SAVE

Instruction:

Save context

Type:

SO

Source Language Format:

$$\Delta\text{SAVE}\Delta \left\{ \begin{array}{l} \text{immediate-memory-address} \\ \text{B-relative-address} \\ \text{P-relative-address} \\ \text{interrupt-vector-addressing} \end{array} \right\}, \left\{ \begin{array}{l} \text{internal-value-expression} \\ \text{external-value-expression} \\ \text{single-precision-fixed-point-constant} \end{array} \right\}$$

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
	M1	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 0, 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. If the 8-bit M1-register is to be saved, the contents are stored in the right half word of the location, and the left half-word is filled with 1s.

SCL

Instruction:

Single-shift closed-left

Type:

SHS

Source Language Format:

$$\Delta SCL \Delta \left\{ \begin{array}{l} \$Rn \\ X'n' \\ n \end{array} \right\}, \text{internal-value-expression}$$

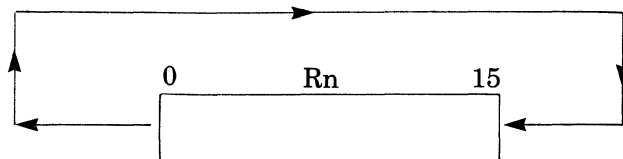
Description:

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 ≥ 0 and ≤ 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.

The following illustrates the operation of the SCL instruction:



SCR

SCR

Instruction:

Single-shift closed-right

Type:

SHS

Source Language Format:

$$\Delta\text{SCRA} \left\{ \begin{array}{l} \$Rn \\ X'n' \\ n \end{array} \right\}, \text{internal-value-expression}$$

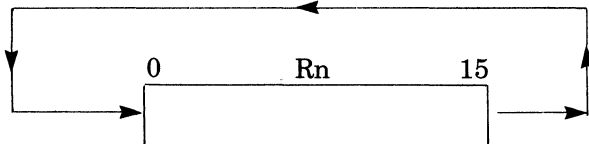
Description:

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 ≥ 0 and ≤ 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.

The following illustrates the operation of the SCR instruction:



SDI

Instruction:

Store Double word integer

Type:

SO

Source Language Format:

 Δ SDI Δ address-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:

= \$Bn	} register addressing
= \$Rn	
= \$Sn	

If the address expression specifies memory addressing with indexing, the index register is aligned to count double-words relative to the word specified.

If Immediate Operand Addressing is specified, the immediate operand may only use a binary integer constant (which is sign extended to 32 bits by the Assembler), a double precision fixed-point constant, or a string constant of exactly two words (i.e., four bytes or 32 bits).

Note:

= \$R3, = \$R5, and = \$R7 are permitted and refer to register pairs \$R2, \$R3; \$R4, \$R5, and \$R6, \$R7, respectively.

Short displacement addressing

Specialized addressing

*

SID

SID

Instruction:

Subtract integer double

Type:

SO

Source Language Format:

Δ SID Δ address-expression

Description:

Subtracts the value of the double-word integer specified by the address expression from the value in the register pair \$R6, \$R7. The result is saved in \$R6 and \$R7, with the most significant part in \$R6 and the least significant part in \$R7.

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

If the address expression specifies memory addressing with indexing, the index register is aligned to count double-words relative to the word specified.

If Immediate Operand Addressing is specified, the immediate operand may only use a binary integer constant (which is sign extended to 32 bits by the Assembler), a double precision fixed-point constant, or a string constant of exactly two words (i.e., four bytes or 32 bits).

If $=\$Rn$ is used, only $=\$R3$ (subtracts the contents of R2 and R3 from R6 and R7 respectively), or $=\$R5$ (subtracts the contents of R4 and R5 from R6 and R7 respectively), or $=\$R7$ (clears R6 and R7) may be used.

If a borrow is required during the subtraction, the C-bit of the I-register is set to 0; otherwise it is set to 1.

If overflow occurs, the OV-bit of the I-register is set to 1, otherwise it is set to 0.

* This instruction is executable only on 6/40 and 6/50 models.

SOL

Instruction:

Single-shift open-left

Type:

SHS

Source Language Format:

$$\Delta\text{SOLA} \left\{ \begin{array}{l} \$R_n \\ X'n' \\ n \end{array} \right\}, \text{internal-value-expression}$$

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 0s.

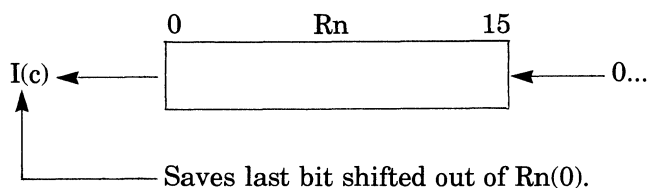
The contents of the I-register are affected as follows:

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

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 12 through 15 of general register, R1.

The following illustrates the operation of the SOL instruction:



SOR

SOR

Instruction:

Single-shift open-right

Type:

SHS

Source Language Format:

$$\Delta\text{SOR}\Delta \left\{ \begin{array}{l} \$R_n \\ X'n' \\ n \end{array} \right\}, \text{internal-value-expression}$$

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 0s.

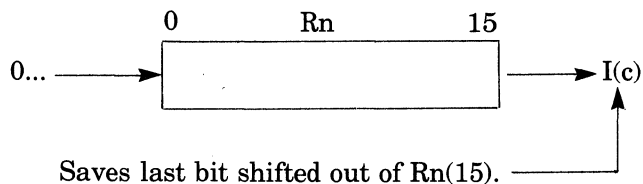
The contents of the I-register are affected as follows:

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

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 12 through 15 of general register \$R1.

The following illustrates the operation of the SOR instruction:



SRDB

Instruction:

Store Remote Descriptor Base Register

Type:

GE

Source Language Format:

ΔSRDB

Description:

Stores the contents of the Remote Descriptor Base Register in register B3.

This instruction is executable only on 6/40 and 6/50 models.

SRM

SRM

Instruction:

Store register masked

Type:

DO

Source Language Format:

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

Description:

Each bit in the mask is individually examined. If the mask bit is 1, the corresponding bit in the operand identified by the effective address is changed to the same value as the corresponding bit in the R-register. If the mask bit is 0, the value of the bit in the operand identified by the effective address is not changed.

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:

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

Example:

Assume that \$R1 contains the value Z'F300' and that \$R2 contains the value Z'3DA5'. Also assume that the value of bits 4, 6, and 7 of \$R1 (i.e., 0, 1 and 1, respectively) are to be put into the corresponding bits of \$R2, leaving the value of the other bits of \$R2 unchanged. Then the following SRM would specify this operation:

SRM \$R1,=\$R2,Z'0B00'

and the resultant value in \$R2 is Z'37A5'.

STB

Instruction:

Store B-register

Type:

DO

Source Language Format:

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

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:

= $\$R_n$ } register addressing
 = $\$S_n$ }

Short displacement addressing

Specialized addressing

Immediate operand addressing with a value expression.

STH

STH

Instruction:

Store R-register halfword (byte)

Type:

DO

Source Language Format:

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

Description:

Stores the rightmost byte of the R-register identified in the first operand into the location specified in the address expression as follows:

- If the address expression specifies the $=\$Rn$ addressing form, the byte is stored in the rightmost byte of the specified R-register.
- If the address expression specifies Memory Addressing *without* indexing or Immediate Operand Addressing, the byte is stored in the leftmost byte of the word found at the specified location.
- 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 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:

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

STM

Instruction:

Store M-register

Type:

DO

Source Language Format:

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

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 1s.

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

STR

STR

Instruction:

Store R-register

Type:

DO

Source Language Format:

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

Description:

Stores the contents of the R-register identified in the first operand 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:

= $\$Bn$ } register addressing

= $\$Sn$ }

Short displacement addressing

Specialized addressing

STS

Instructions:

Store S-register

Type:

SO

Source Language Format:

Δ STS Δ address-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:

= B_n } register addressing
= S_n }
Short displacement addressing
Specialized addressing

STT

STT

Instruction:

Store Stack Address Register

Type:

GE

Source Language Format:

Δ STT

Description:

This stack instruction moves the address in the T register to register \$B7.

Stack instructions are double-word instructions with the following characteristics:

- A common first word.
- Bits 0 through 8 and bit 12 of the second word contain zeros.

If bits 0 through 8 and bit 12 of the second word are not zero, the result is a trap to trap vector 16.

Stack instructions can be executed only on model 6/40 and 6/50 systems.

SUB

Instruction:

Subtract from R-register

Type:

DO

Source Language Format:

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

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:

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

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

- 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.
- If a borrow is required during the subtraction, the C-bit is set to 0, otherwise it is set to 1.

SWB

SWB

Instruction:

Swap B-register

Type:

DO

Source Language Format:

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

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:

= $\$Rn$ } register addressing
= $\$Sn$ }

Short displacement addressing

Specialized addressing

Immediate operand addressing with a value expression.

SWR

Instruction:

Swap R-register

Type:

DO

Source Language Format:

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

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:

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

*

VLD

VLD

Instruction:

Validate

Type:

GE

Source Language Format:

VLD

Description:

Determines the access rights to the data whose beginning (virtual) address is contained in \$B5 and whose length, in bytes, is contained in \$R3. The access rights are determined with respect to the effective ring value contained in bits 1 and 2 of \$R5. VLD indicates the accessibility of the data by storing a value in \$R3.

The possible values to be stored in \$R3 after execution of a VLD instruction are as follows:

<i>\$R3 Value</i>	<i>Meaning</i>
-1	Invalid segment
0	Read access permitted, write access not permitted
+2	Read/Write access permitted
-2	No access permitted

Figure 5-24 illustrates the VLD instruction operations.

This instruction is available only with systems that have a Memory Management Unit.

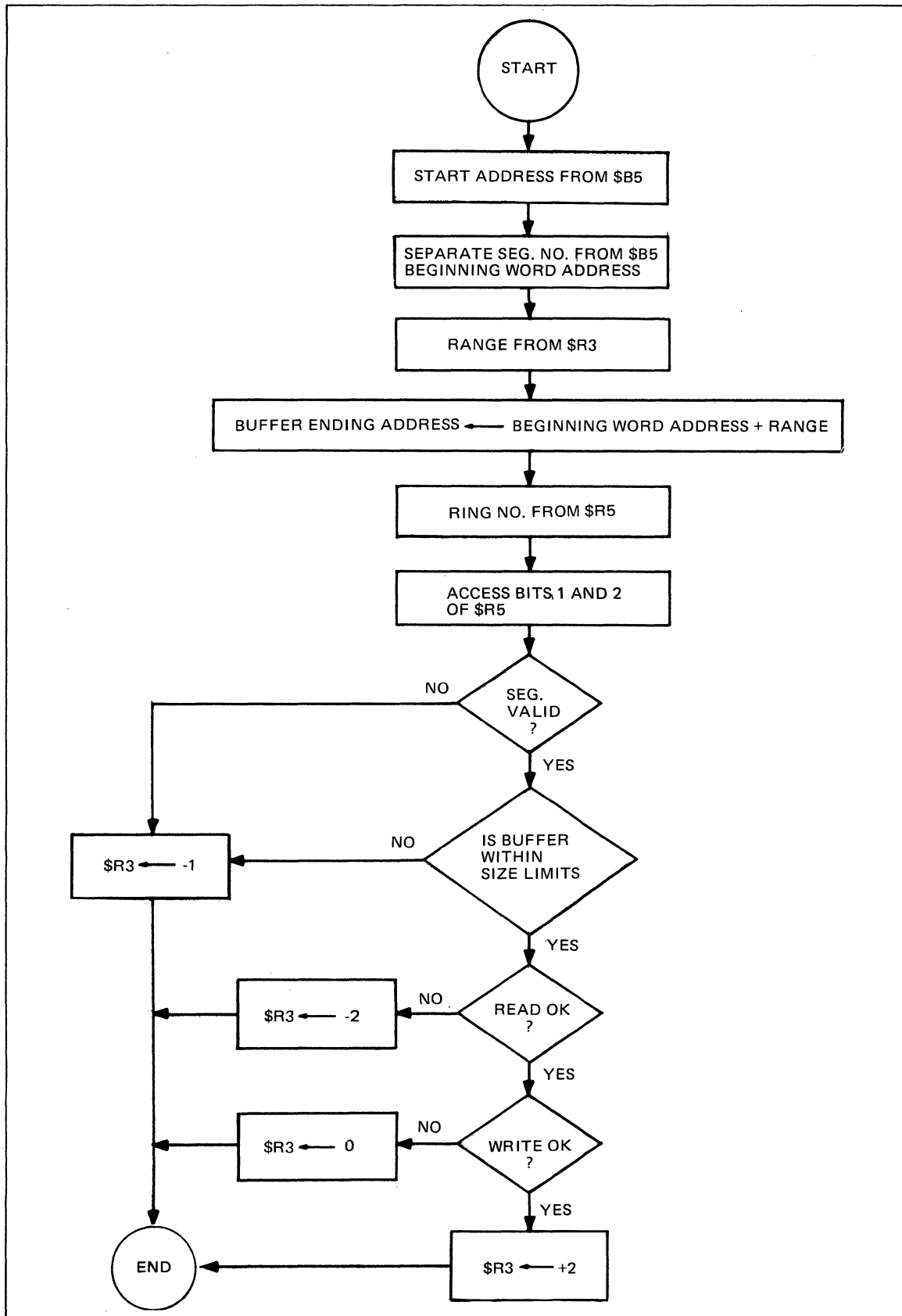


Figure 5-24. VLD Instruction Operations

WDTF

WDTF

Instruction:

Watchdog timer off

Type:

GE

Source Language Format:

Δ WDTF

Description:

Disables the watchdog timer interrupt (i.e., level 1 interrupt).

The P-bit in the S-register must be set to 1 or the ring field in the S-register must be set to 1x, as appropriate, (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:

Δ WDTN

Description:

Enables watchdog timer interrupt (i.e., level 1 interrupt).

The P-bit in the S-register must be set to 1 or the ring field in the S-register must be set to 1x, as appropriate, (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

XOH

Instruction:

Half-word (byte) exclusive OR with R-register

Type:

DO

Source Language Format:

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

Description:

An exclusive OR operation is performed on the contents of the R-register identified in the first operand with the contents of the byte specified in the address expression, and the result is stored in the register identified in the first operand.

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:

- Register Addressing ($=\$Rn$): The rightmost byte of the register is selected.
- Memory Addressing *Without* Indexing or Immediate Operand Addressing: The leftmost byte of the word at the designated memory address is selected.
- 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 performing an exclusive OR operation on 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:

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

Description:

An exclusive OR operation is performed on 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 performing an exclusive OR operation on 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



Section 6

Commercial Instructions

The commercial instructions are executed by the Commercial Processor, an optional hardware item, or by the Commercial Processor simulator, a software item that provides the same functionality.

The commercial instruction set includes the capabilities for processing bytes, packed decimal data and unpacked decimal data. Thus the speed with which business-oriented applications are handled is greatly increased.

BASIC FEATURES

The basic features of the commercial instruction set are as follows:

- Decimal Arithmetic
 - Packed decimal and unpacked decimal data types may be intermixed.
 - Decimal arithmetic operands may be 1 through 31 digits in length.
 - Decimal arithmetic instructions are add, subtract, multiply, and divide.
 - All decimal arithmetic instructions provide a hardware rounding option.
- Data Manipulation Capabilities
 - There are three data modes — ASCII, packed decimal, and unpacked decimal.
- Data Movement
 - Alphanumeric movement from left or right with character-fill.
 - Numeric move with fill and/or rounding.
 - Radix conversion and transliteration instructions.
- Data Comparison
 - Alphanumeric comparison with fill
 - Numeric comparisons between fields of the same or different format and character type.
 - Scan to determine if one or more strings consisting of one or more characters is contained in a source string.
 - Scan to determine if one or more strings consisting of one or more characters is not contained in a source string.
- Editing Capabilities
 - Editing capabilities with micro operations are provided.
 - Micro operations provide alphanumeric and numeric edited move instructions with the capability of editing character and numeric strings on a character-by-character or digit-by-digit basis, or in a concatenated series of characters and digits.
 - Micro operations are not altered by their execution; therefore, a sequence of micro operations can be set to describe a data field and then can be used repeatedly by the edit instructions.
 - A single instruction can perform a complicated edit function with great speed.

The commercial instructions are divided into six categories, as described in the subsequent text. The internal formats of the commercial instructions are in Appendix H.

COMMERCIAL PROCESSOR PROGRAMMING CONSIDERATIONS.

During the execution of a program that includes two adjacent Commercial Processor instructions, the CPU must wait for the first to be completed before initiating the second. However, if a Commercial Processor instruction is followed by a CPU or SIP instruction, execution of the CPU or SIP instruction begins before the Commercial Processor instruction is completed. Under such conditions, the CPU or SIP must not access any of the operands specified by the Commercial Processor instruction. The possibility of such a conflict is eliminated by inserting a commercial

synchronizing (CSYNC) instruction between the Commercial Processor and the CPU or SIP instruction. CSYNC performs no operation except to make the CPU wait for completion of the preceding Commercial Processor instruction.

COMMERCIAL INSTRUCTION CATEGORIES

DECIMAL ARITHMETIC INSTRUCTIONS

These instructions perform the basic arithmetic functions (add, subtract, multiply, and divide) on packed and unpacked decimal data, and allow the comparison of decimal data.

DAD	decimal add
DCM	decimal compare
DDV	decimal divide
DML	decimal multiply
DSM	decimal subtract

RADIX AND MODE CONVERSION INSTRUCTIONS

These instructions allow

- Decimal and binary data to be converted from one type to the other.
- Decimal data of one type to be moved and converted to decimal data of a different type (e.g., packed to unpacked).

CBD	convert binary to decimal
CDB	convert decimal to binary
DMC	decimal move and convert

SHIFT INSTRUCTIONS

Shift instructions allow decimal data to be shifted a specified number of digits to the right or to the left. The sign character, if any, is not affected by the shift operation (i.e., it does not get shifted).

DLS	decimal left shift
DRS	decimal right shift
DSH	decimal shift

EDIT INSTRUCTIONS

These instructions provide for a combined move and edit of decimal or alphanumeric data.

AME	alphanumeric move and edit
DME	decimal move and edit

CHARACTER STRING INSTRUCTIONS

These instructions provide for moving, comparing, and translating alphanumeric strings.

ACM	alphanumeric compare
ALR	alphanumeric move left to right
MAT	alphanumeric move and translate
SRCH	search
VERFY	verify

BRANCH INSTRUCTIONS

The Commercial Processor branch instructions are similar to the CPU branch instructions described in Section 5. Execution of the Commercial Processor branch instructions takes place mainly in the CPU. The Commercial Processor provides information about the state of the indicators. The CPU uses this information to decide whether or not a branch is required.

The commercial branch instructions are as follows:

CBE	commercial branch on equal
-----	----------------------------

CBG	commercial branch on greater
CBGE	commercial branch on greater than or equal
CBL	commercial branch on less than
CBLE	commercial branch on less than or equal
CBNE	commercial branch on not equal
CBNOV	commercial branch on no overflow
CBNTR	commercial branch on no truncation
CBNSF	commercial branch on no sign fault
CBOV	commercial branch on overflow
CBSF	commercial branch on sign fault
CBTR	commercial branch on truncation
CSNCB	commercial synchronize and branch (i.e., B)
CSYNC	commercial synchronize (i.e., NOP)

COMMERCIAL PROCESSOR INSTRUCTION FORMAT

The basic format of Commercial Processor instructions, other than branch instructions, are as follows:

$$\Delta \text{op code} \Delta \left\{ \begin{array}{l} \text{data-descriptor} \\ \text{int-val-expression} \end{array} \right\} \left[\left\{ \begin{array}{l} \text{data-descriptor} \\ \text{int-val-expression} \end{array} \right\} \right] \left[\left\{ \begin{array}{l} \text{data-descriptor} \\ \text{int-val-expression} \end{array} \right\} \right]$$

One, two, or three operands are required depending on the Commercial Processor instruction. All Commercial Processor instructions, except Commercial Processor branch instructions, require at least one data descriptor. A data descriptor specifies the type of data on which the instruction is to operate and the location of the data. Each data descriptor includes a Commercial Processor address expression. An internal value expression designates a remote descriptor and is used with the remote descriptor base register (RDBR). Address expressions are explained later in this section.

COMMERCIAL PROCESSOR DATA DESCRIPTORS

Data descriptors can be in line (entered as part of the instruction) or they can be remote (placed in a remote descriptor array). If remote descriptors are used, the starting address of the remote descriptor array must be loaded into the remote descriptor base register of the CPU. (This can be accomplished by use of the CPU instruction LRDB.) The desired remote descriptor is referenced by an internal value expression in the Commercial Processor instruction. The internal value expression is effectively multiplied by two to obtain the offset of the desired remote descriptor from the starting address of the array of descriptors whose starting address is contained in the remote descriptor base register (RDBR). (For further information on the use of the RDBR, see Appendix H.) The data descriptor names and the type of data they specify are as follows:

<i>Name</i>	<i>Data Type</i>
DESCA	Alphanumeric
DESCP	Packed decimal
DESCU	Unpacked decimal
DESCB	Binary

ALPHANUMERIC DATA DESCRIPTOR

An alphanumeric data descriptor describes a field composed of 8-bit (byte) characters, normally in ASCII format.

Format:

DESCA (Commercial Processor address expression

$$[\text{byte_offset} \text{ } [\text{byte_length} \left[\left\{ \begin{array}{l} \text{FILL} \\ \text{NO_FILL} \end{array} \right\} \right]]])$$

byte_offset

An internal value expression having a value equal to 0 to 1, that specifies an 8-bit byte offset within the addressed word.

Default = 0 (See individual instructions for meaning of 0)

byte_length

An internal value expression specifying the length of operand in bytes. Valid range is 0 through 31.

Default = 0 (See individual instructions for meaning of 0)

FILL/NO_FILL

Specifies whether the FILL option is to be used in case of unequal length fields.

Default = FILL (shorter field with specified character)

PACKED-DECIMAL DATA DESCRIPTOR

A packed decimal data descriptor describes a number whose sign and each digit is internally represented by 4 bits.

Format:

DESCP (Commercial Processor address expression

$$\left[, \text{digit_offset} \left[, \text{digit_length} \left[, \left\{ \begin{array}{l} \text{U[NSIGNED]} \\ \text{T[RAILING]} \end{array} \right\} \right] \right] \right])$$

digit_offset

An internal value expression, having a value in the range 0 to 3, that specifies an offset in 4-bit digits within the addressed word.

Default = 0.

digit_length

An internal value expression that specifies the length of the number in digits.

Default = 0 (See individual instruction for meaning of 0)

Valid range is from 0 through 31.

U[NSIGNED]

Specifies the number is unsigned.

T[RAILING]

Specifies the number has a trailing sign.

Default = TRAILING

UNPACKED-DECIMAL DATA DESCRIPTOR

An unpacked-decimal data descriptor describes a number whose sign and each digit are represented by the corresponding 8-bit (byte) ASCII character.

Format:

DESCU (Commercial Processor address expression

$$\left[, \text{byte_offset} \left[, \text{byte_length} \left[, \left\{ \begin{array}{l} \text{U[NSIGNED]} \\ \text{O[VERPUNCHED]} \\ \text{L[EADING]} \\ \text{T[RAILING]} \end{array} \right\} \right] \right] \right])$$

byte_offset

An internal value expression equal to 0 or 1 that specifies on 8-bit (byte) offset within the addressed word.

Default = 0

byte_length

An internal value expression that specifies the length of the number in bytes.

Default = 0

Valid range is from 0 through 31.

U[NSIGNED]

Specifies an unsigned number.

O[VERPUNCHED]

Specifies a number whose sign is indicated by a trailing overpunch.

L[EADING]

Specifies a number whose sign is indicated by the leading byte.

T[RAILING]

Specifies a number whose sign is indicated by the trailing byte.

Default = OVERPUNCHED

BINARY DATA DESCRIPTOR

A binary data descriptor describes a binary integer internally represented by 16 or 32 bits, word-aligned, and in two's complement format.

Format:

DESCB (Commercial Processor address expression[,byte length])

byte_length

An internal value expression, having a value of 0, 2, or 4, that specifies the length of the binary integer in bytes.

Default = 0 (See individual instruction for meaning of 0)

Note that a binary data descriptor is actually an alphanumeric data descriptor specifying no fill and a length, after possible escape to Rn(11-15), of either 2 bytes or 4 bytes.

ADDRESSING TECHNIQUES FOR COMMERCIAL PROCESSOR INSTRUCTIONS

A Commercial Processor address expression defines the addressing technique used to reference data. It can take any of the following forms:

- P-relative addressing
- B-relative addressing
- Immediate operand addressing
- Use of remote descriptor base register (For details, see "Commercial Processor Data Descriptors," above, and Appendix H)

P-RELATIVE ADDRESSING

P-relative addressing designates data by indicating the (Assembler-calculated) displacement from the current location (i.e., the location of the second word of the data descriptor). P-relative addressing references a location directly (with or without indexing) or indirectly as shown by the following formats.

$$\left\{ \begin{array}{l} \text{location-expression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \\ \text{temporary-label} \end{array} \right\} \text{ Direct P-relative addressing}$$
$$\left\{ \begin{array}{l} \text{location-expression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \\ \text{temporary-label} \end{array} \right\} \cdot \$R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\} \text{ Indexed direct P-relative addressing}$$
$$* \left\{ \begin{array}{l} \text{location-expression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \\ \text{temporary-label} \end{array} \right\} \text{ Indirect P-relative addressing}$$

DIRECT P-RELATIVE ADDRESSING

This form of addressing allows you to specify a location relative to the P-register (i.e., the address of the data descriptor). The following example illustrates this form of addressing. The decimal number whose characteristics and address are specified by the first operand is subtracted from the number whose characteristics and address are specified by the second operand. The result is stored at the address specified by the second operand.

Example:

```
ONEK DC +P'1024'           Packed decimal, 4 digits plus sign
FOURK DC N'4096'          Unpacked, unsigned decimal, 4 digits
```

```
DSB DESC(ONEK,0,5,TRAILING);
DESCU(FOURK,0,4,UNSIGNED)
```

At the completion of the instruction, the unpacked unsigned decimal N'3072' is stored at symbolic location FOURK.

Figure 6-1 illustrates the above example. The following assumptions are made in this figure:

- Symbolic location ONEK is at location 3000
- Symbolic location FOURK is at location 3D50
- The decimal subtract (DSB) instruction is assembled at location 1000

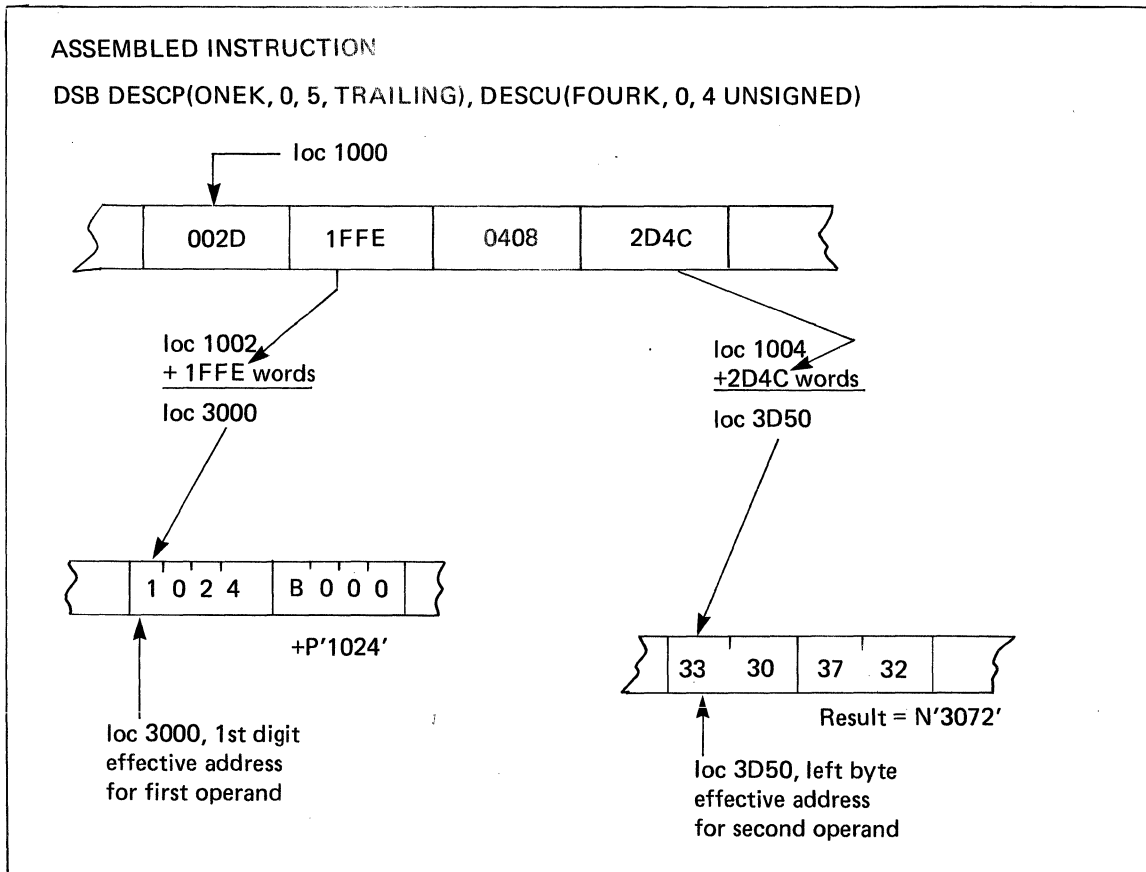


Figure 6-1. Commercial Processor Direct P-relative Addressing

Indexed Direct P-Relative Addressing

In this type of addressing, the contents of the specified index register, aligned to count digits or bytes, are added to the displacement to derive the location of the data to be included in the operation.

Example:

Convert binary number -300 to unpacked decimal with leading sign; and offset by one byte in receiving field. Assume that index register R1 contains 4 and index register R2 contains 2.

```

BININ    DC 0
          DC 0
          DC -300    (FED4 in hexadecimal)
RESULT   RESV 4,'ΔΔ' (20202020202020 in hexadecimal)
  
```

```

CBD      DESCB(BININ.$R1,2);
          DESCU(RESULT.$R2,1,4,LEADING)
  
```

Since the sending field is symbolic location BININ+2, which contains -300, and receiving field is symbolic location RESULT+1, at completion of the instruction, the RESULT string will contain:

```

20 20 20 2D 33 30 30 20
Δ  Δ  Δ  —  3  0  0  Δ
  
```

Figure 6-2 illustrates the above example. The following assumptions are made in this figure:

- Symbolic location BININ is at location 3000
- Symbolic location RESULT is at location 3D50
- The convert from binary to decimal (CBD) instruction is assembled at location 1000

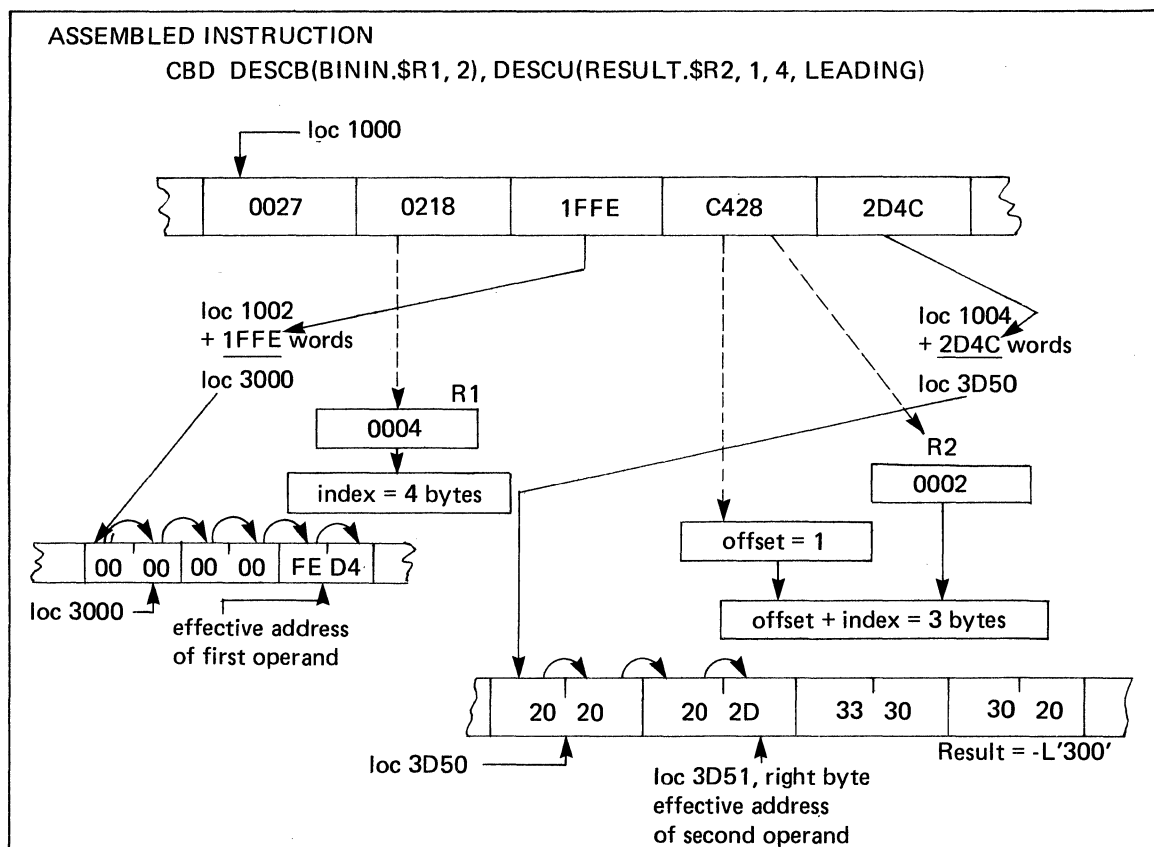


Figure 6-2. Commercial Processor Indexed Direct P-relative Addressing

Indirect P-Relative Addressing

In this type of addressing, a location relative to the contents of the P-register contains the location of the data to be used by the instruction. In the following example, a binary number is converted to an unpacked decimal number. The first data descriptor specifies the binary number by referencing the location (BINLOC) that contains its address. The second data descriptor references the location (DECLOC) that contains the location (RESULT) of the receiving field.

Example:

Convert binary number -300 to unpacked decimal with leading sign and offset by one byte in the receiving field.

```

BINLOC DC <BININ
DECLOC DC <RESULT
BININ  DC -300          (FED4, hexadecimal)
RESULT RESV 3, '      ' (2020202020, hexadecimal)

```

```

CBD  DESCB(*BINLOC,2);
     DESCU(*DECLOC,1,4,LEADING)

```

At completion of the instruction, the receiving field will contain the following data.

```

202D33303020, hexadecimal
Δ - 3 0 0 Δ, decimal

```

Figure 6-3 illustrates the above example. The following assumptions are made in this figure:

- Symbolic location BINLOC is at location 3000
- Symbolic location DECLOC is at location 3D50
- Symbolic location BININ is at location C08F
- Symbolic location RESULT is at location D317
- The convert binary to decimal (CBD) instruction is assembled at location 1000

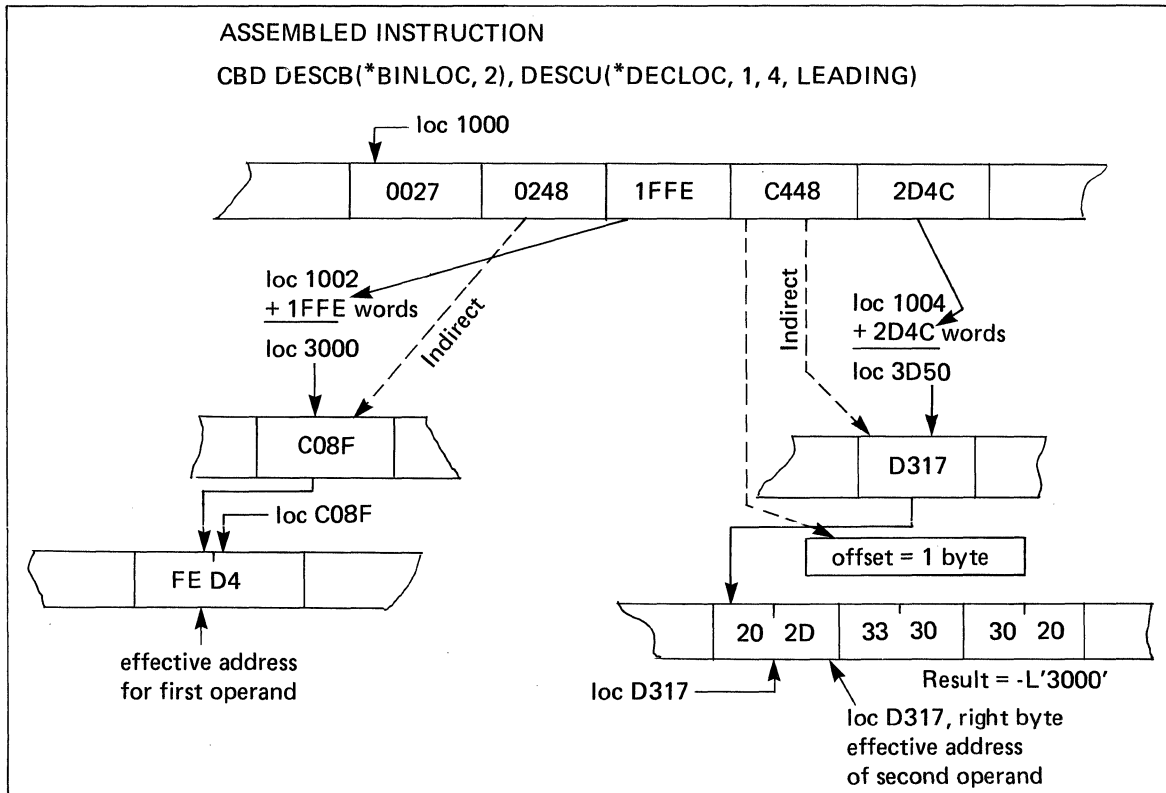


Figure 6-3. Commercial Processor Indirect P-relative Addressing

COMMERCIAL PROCESSOR B-RELATIVE ADDRESSING

In this form of addressing, a base register (i.e., \$B1, ..., \$B7) is used to reference a location that contains data or an address. This form of Commercial Processor addressing can be used to reference a location directly or indirectly as a displacement, or as a displacement with indexing.

B-relative addressing for Commercial Processor instructions can be represented by any one of the following expressions:

$$\$Bn. \left\{ \begin{array}{l} \text{int-val-expression} \\ \text{ext-val-expression} \end{array} \right\} \quad \text{Direct B-relative plus displacement}$$
$$*\$Bn. \left\{ \begin{array}{l} \text{int-val-expression} \\ \text{ext-val-expression} \end{array} \right\} \quad \text{Indirect B-relative plus displacement}$$
$$\$Bn. \left\{ \begin{array}{l} \text{int-val-expression} \\ \text{ext-val-expression} \end{array} \right\} .Rn \quad \text{Direct B-relative plus displacement with indexing}$$
$$*\$Bn. \left\{ \begin{array}{l} \text{int-val-expression} \\ \text{ext-val-expression} \end{array} \right\} .Rn \quad \text{Indirect B-relative plus displacement with indexing.}$$

COMMERCIAL PROCESSOR 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 a base register. An example of direct B-relative plus displacement addressing in combination with indirect B-relative plus displacement addressing follows the explanation of the indirect form of addressing.

COMMERCIAL PROCESSOR INDIRECT B-RELATIVE PLUS DISPLACEMENT ADDRESSING

This addressing form effectively adds a displacement value to the contents of the specified base register to produce a new address. The contents of this subsequent location are now used as the effective memory address. An example of indirect B-relative plus displacement addressing in combination with direct B-relative plus displacement addressing is given in the following example.

Example of Commercial Processor Direct and Indirect B-Relative Plus Displacement Addressing

ADDR1 DC N'1234' Unpacked, unsigned decimal, 4 digits

ADDR2 DC +P'2092' Packed decimal, 4 digit plus sign

•
•
•

DSB DESCU(\$B5.3,0,4,U),DESCP(★\$B1.-2,0,5)

At the completion of the instruction, the packed decimal with trailing sign +P'858' is stored at symbolic location ADDR2.

Figure 6-4 illustrates the above example. The following assumptions are made in this figure:

- Symbolic location ADDR1 is at location 3000
- Symbolic location ADDR2 is at location 3D50
- The decimal subtract (DSB) instruction is assembled at location 1000
- \$B5 contains 2FFD

- \$B1 contains 3FE2
- Location 3FE0 contains a pointer to location 3D50

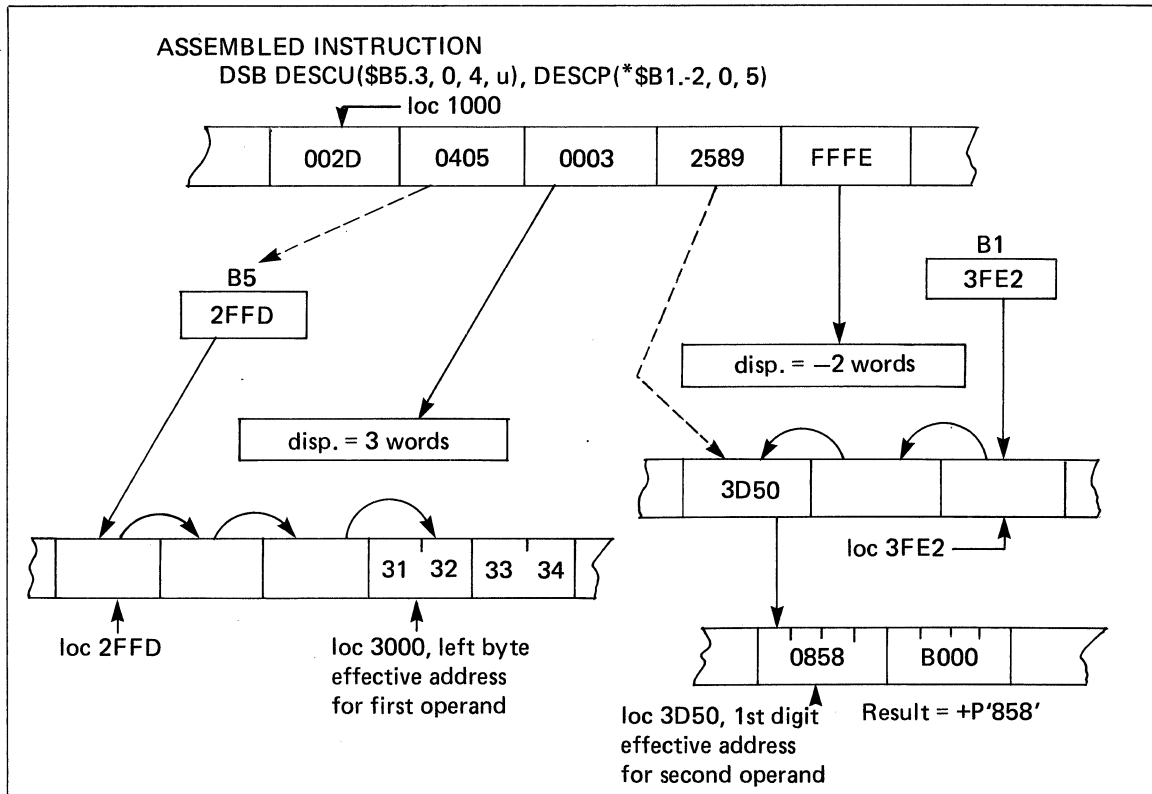


Figure 6-4. Commercial Processor Direct and Indirect B-relative Plus Displacement Addressing

COMMERCIAL PROCESSOR DIRECT B-RELATIVE PLUS DISPLACEMENT WITH INDEXING ADDRESSING

In this form of addressing, the effective address is computed by first determining the B-relative plus displacement address; the content of an index register is then added to this result to determine the effective address.

An example of this form of addressing, in combination with indirect B-relative plus displacement with indexing addressing follows the explanation of the indirect form of addressing.

COMMERCIAL PROCESSOR INDIRECT B-RELATIVE PLUS DISPLACEMENT WITH INDEXING ADDRESSING

To determine the effective address, the B-relative plus displacement address is first computed. The contents of the index register are then added to the contents of the location pointed to by the result of the first computation to obtain the effective address.

Example of Commercial Processor Direct and Indirect B-Relative Plus Displacement With Indexing Addressing:

This example is built upon the preceding example, dealing with direct and indirect B-relative with displacement addressing. Figure 6-5 illustrates these forms of addressing. The assembled instruction for this example is assumed to be:

DSB DESCU(\$B5.3.\$R2,0,4,U),DESCP(★\$B1.-2.\$R3,0,5)

All other assumptions for the preceding example remain unchanged. In addition, \$R2 is assumed to contain +6 and \$R3 is assumed to contain +7.

Starting with loc 3000 for the first operand in the preceding example, we find the effective address with indexing to become loc 3003 (i.e., loc 3000 + 0006 bytes).

For the second operand of the previous example, we find the value 3D50 at the location 3FE0, which is the B-relative plus displacement address. The contents of \$R3 are added to this value to give the effective address for the second operand; i.e., 3D50 + 0007 = 3D57.

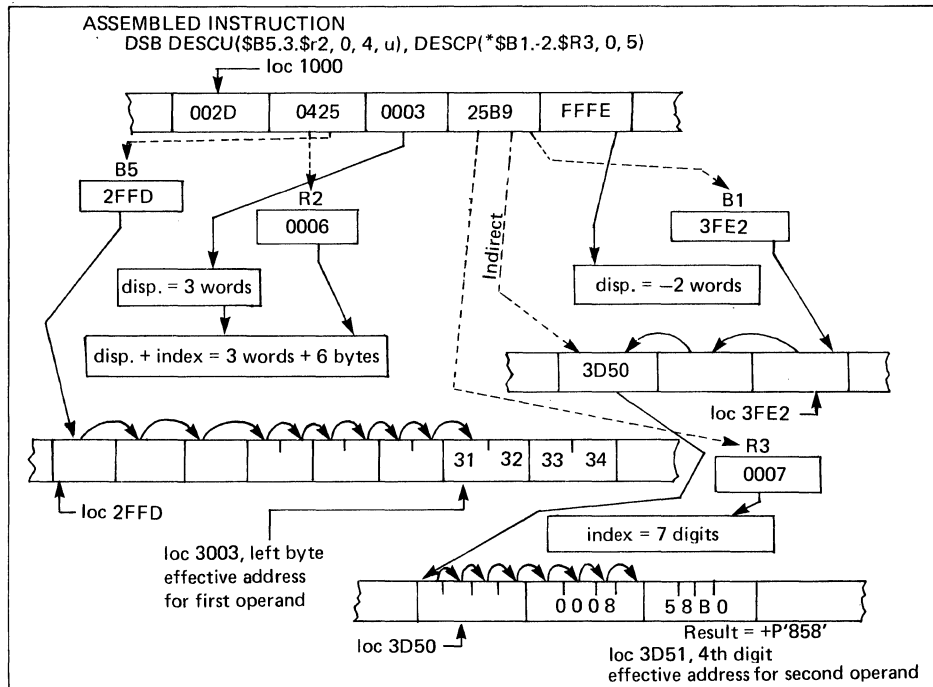


Figure 6-5. Commercial Processor Direct and Indirect B-relative Plus Displacement with Indexing Addressing

IMMEDIATE OPERAND (IMO) ADDRESSING

Immediate operand addressing makes it possible to specify a decimal or string constant as the address expression. An IMO operand always occupies one word of memory. If the expressed operand is more than one word long, it is truncated on the right to a single word. If the operand is less than one word long, it is stored left justified with low order bits zero filled. For numeric Commercial Processor instructions, the format of the IMO address expression is:

= decimal-integer-constant

For alphanumeric Commercial Processor instructions, the format of the IMO address expression is:

= string constant

An IMO address expression must not be specified by the second data descriptor of any Commercial Processor instruction, except for the compare instructions ACM and DCM.

An IMO may not be a binary integer constant.

An IMO address expression must not be specified by any Commercial Processor edit instruction.

The following example illustrates Commercial Processor IMO addressing.

Example:

Assume we wish to convert the packed decimal value of -997 to an unpacked form.

DMC DESCU(=-P'997',0,4),DESCU(RESET,1,7,LEADING)

RESULT RESV 5,Z'2020'

The following assumptions are made:

- Symbolic location RESULT is at location 9003
- The decimal-move-and-convert (DMC) instruction is assembled at location 7000.

Figure 6-6 illustrates the above example.

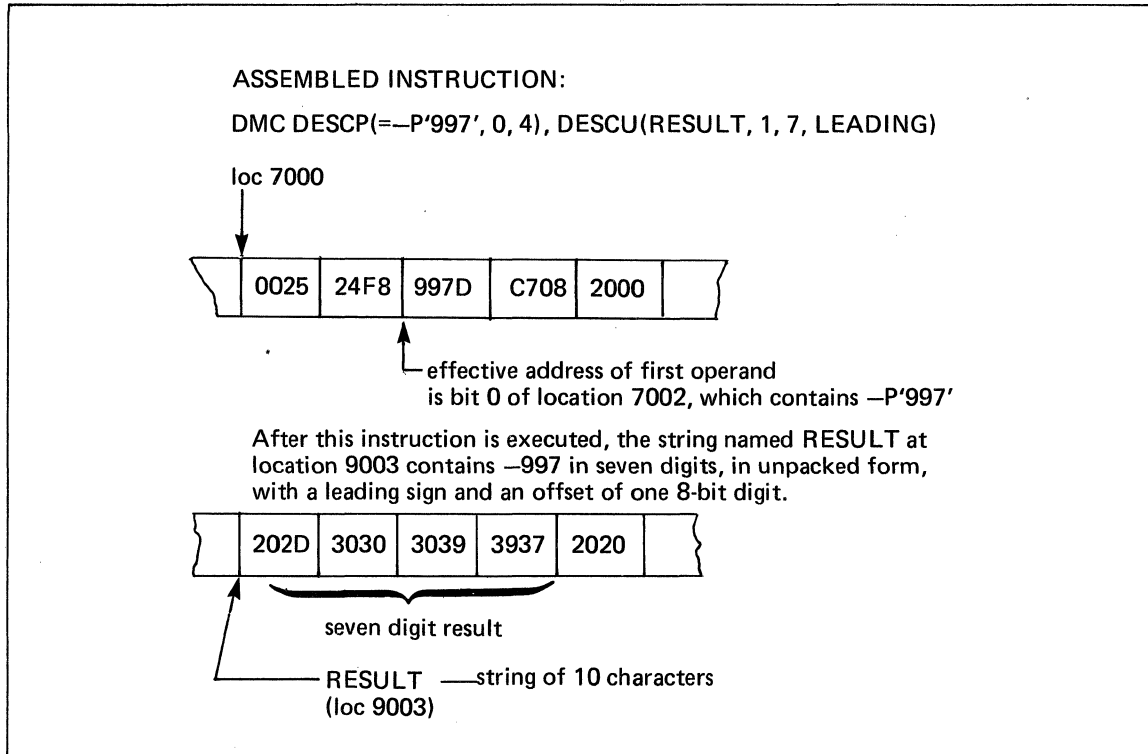
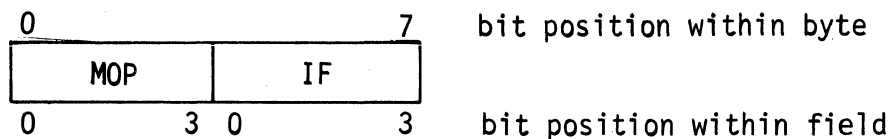


Figure 6-6. Commercial Processor IMO Addressing

MICRO EDIT OPERATIONS

The two move and edit instructions — AME and DME — require micro operations. The sequence of micro operations to be executed must be placed in storage before the edit instruction is executed. The sequence is referenced by the third data descriptor of the AME or DME instruction. Some micro operations insert characters in the string of characters being manipulated; for example, the currency symbol or the decimal point. Other micro operations replace characters in the source string. For example, leading zeros in the source string may be replaced with asterisks or blanks in the receiving string. The characters used for replacement or insertion are contained in a table called the Edit Insertion Table (EIT) which is described later in this section.

The third data descriptor of a DME or AME instruction points to an array of micro operations. Each micro operation is represented by eight bits as shown by the following format.



MOP — Micro operation code field
IF — Information field

The format for specifying a Commercial Processor micro operation is as follows:
Commercial Processor-micro-operation(int-val-expression)
where, Commercial Processor-micro-operation is one of the mnemonics in Table 6-1, and

(int-val-expression) must resolve to an integer in the range 0 to 16. Not all integers in this range are valid for all micro operations. See description of micro operations for restrictions.

Examples:

MVZB(4)
SEF(0)

Before an edit instruction is executed, you must set up a string by use of the TEXT assembler control statement. This string contains the desired micro operations in the sequence in which they are performed. For example, the statement

MOP TEXT SEF(0),MVZA(8)

sets up a string named MOP that contains the micro operations SEF and MVZA.

The fourteen micro operations that are available for use with the edit instructions are listed in Table 6-1.

The function of the information field (IF) depends on the micro operation. For example, the IF field may specify:

- The number (1 through 16) of the source digits to be manipulated. (An IF entry of zero is used to represent 16 digits.)
- The character in the Edit Insertion Table to be used.
- The state in which edit flags are to be placed.

TABLE 6-1. MICRO OPERATIONS FOR EDIT INSTRUCTIONS

Code Hexadecimal	Bit	Mnemonic	Function
0	0000	CHT	Change Edit Insertion Table
1	0001	ENF	End floating suppression
2	0010	IGN	Ignore specified source characters
3	0011	INSA	Insert asterisk on suppress
4	0100	INSB	Insert blank on suppress
5	0101	INSM	Insert multiple EIT entry 1 characters
6	0110	INSN	Insert on negative
7	0111	INSP	Insert on positive
8	1000	MFLC	Move with float currency symbol insertion
9	1001	MFLS	Move with float sign insertion
A	1010	unassigned	
B	1011	unassigned	
C	1100	MVC	Move source character
D	1101	MVZA	Move with zero suppression and asterisk replacement
E	1110	MVZB	Move with zero suppression and blank replacement
F	1111	SEF	Set edit flags

EDIT INSERTION TABLE

During the execution of an edit instruction, a hardware table of eight 8-bit characters holds insertion information. At the start of each edit instruction, this table is initialized as shown in Table 6-2. Any one or all of the table entries can be changed by the Change Table (CHT) micro operation.

TABLE 6-2. EDIT INSERTION TABLE AT INITIALIZATION

	Entry number							
	1	2	3	4	5	6	7	8
Character	b	*	+	-	\$,	.	0
Hexadecimal code	20	2A	2B	2D	24	2C	2E	30

EDIT FLAGS

Six edit flags are provided for use with the edit micro operations. Some edit flags record conditions that occur during execution of the edit instruction. Others allow the user to establish conditions for control of the editing. The editing performed by certain micro operations depends on the status of specified edit flags. Editing may occur in two stages: (1) while an item is moved to the receiving field and (2) at the end of the sequence of micro operations (called post edit). The edit flags and their functions are given in Table 6-3.

TABLE 6-3. EDIT FLAGS FOR MICRO OPERATIONS

Name	Function
End Suppression(ES)	Initial status is OFF. Set on when a non-zero digit from a numeric sending field is moved to the receiving field. It can be interrogated and its status changed by certain micro operations.
Sign (SN)	Initial status is OFF for alphanumeric sending fields, and for numeric sending fields that are positive, or unsigned, or equal to plus or minus zero. Initial status is ON for numeric sending fields that are negative. It can be interrogated by certain micro operations, but, once initialized, it cannot be changed by any micro operation.
Zero(Z)	Initial status is ON. Changed to OFF when a non-zero digit is moved from a numeric sending field to the receiving field. Once the Z flag is changed to OFF, it remains OFF for the duration of the edit instruction. The Z flag can be neither interrogated nor altered by any micro operation. At post edit, the Z flag is interrogated and the editing then performed depends on its status.
Blank-When-Zero(BZ) ^a	Initial status is OFF. It can be set ON by an ENF or a SEF micro operation. Once it is set to ON, it cannot be changed to OFF by any micro operation. At the end of a micro operation sequence, if <i>both</i> the Z and BZ flags are ON, the entire receiving field will be force-filled with the character in EIT(1) — normally a space character. Note that this post edit operation completely blanks out whatever character string has been moved into the receiving field.
Asterisk-When-Zero(AZ) ^a	Initial status is OFF. It can be set ON by the SEF micro operation. Once set ON, it cannot be changed to OFF by any micro operation. At the end of a micro operation sequence, if the Z flag is ON, the BZ flag is OFF, and the AZ flag is ON, the following editing is performed. The entire receiving field, except for any characters corresponding to EIT(7) — normally a decimal point — will be filled with the character specified by EIT(2) — normally an asterisk.
Plus/Minus(PM)	Initial status is OFF. It can be set ON by the SEF micro operation. Once set ON it cannot be changed to OFF by any micro operation. The status of the PM flag affects the operation of the following micro operations: MFLS, ENF, INSP, INSN. (See detailed description of micro operations later in this section.)

^aThe blank-when-zero and the asterisk-when-zero operations are mutually exclusive. In case of conflict, the blank-when-zero operation takes precedence; i.e., the BZ flag is interrogated before the AZ flag.

CHANGE EDIT INSERTION TABLE (CHT) MICRO OPERATION

This micro operation changes characters in the Edit Insertion Table as specified by the IF entry in the micro operation. If a single character is to be changed, the replacement character is specified in the 8-bit byte immediately following the micro operation. If all characters are to be changed, the replacement characters are specified in the eight 8-bit bytes immediately following the micro operation. To change a given character, specify its entry number as shown by Table 6-4. Note that an entry in the range of 9 to F will cause an illegal specification (IS) trap.

TABLE 6-4. CODE FOR REPLACING EIT ENTRIES

IF CODE	OPERATION
0	Replace all eight EIT entries
1	Replace EIT Entry 1
2	Replace EIT Entry 2
3	Replace EIT Entry 3
4	Replace EIT Entry 4
5	Replace EIT Entry 5
6	Replace EIT Entry 6
7	Replace EIT Entry 7
8	Replace EIT Entry 8
9 — F	Trap 26 Illegal specification

In the following example, the string of eight characters beginning at the address specified by A8 is moved to the receiving field of eight characters beginning at the address specified by WK8K. In the move, leading zeros, are replaced by the pound sign (#). Normally, the zeros would be replaced by asterisks but the CHT micro operation has changed the asterisk (EIT entry 2) to the pound sign. Note that the sending field count and the receiving field count are not changed by the CHT micro operation.

Example:

```

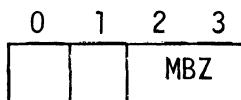
MOP   TEXT   CHT(2),'#',MVZA(8)
A8    DC     '00$81.05'
WK8K  RESV   4,'
.
.
.
AME   DESCA(A8,0,8);
      DESCA(WK8K,0,8);
      DESCA(MOP,0,3)

```

After the instruction is executed, the contents of the receiving field will be ##\$81.05.

END FLOATING SUPPRESSION (ENF) MICRO OPERATION

This micro operation terminates zero suppression or floating insertion and forces the insertion of the appropriate sign or the appropriate currency symbol, and, optionally, sets the BZ flag ON. Only bits 0 and 1 of the IF are used; bits 2 and 3 must be zero.



The functions performed for valid arguments (IF entries) are as follows. An invalid argument will generate trap 26, Illegal Specification.

<i>Argument</i>	<i>IF</i>	<i>Function</i>
0	0000	Insert appropriate sign; do not change BZ
4	0100	Insert appropriate sign; set BZ ON
8	1000	Insert appropriate currency symbol; do not change BZ
12	1100	Insert appropriate currency symbol; set BZ ON

The sign or currency symbol inserted depends on the status of various edit flags as shown below. The entry x under an edit flag column indicates that the flag is not interrogated; i.e., its status makes no difference for the specified insertion. The insertion characters shown are those normally in the EIT.

Edit Flag Status

<i>ES</i>	<i>PM</i>	<i>SN</i>	<i>IFO</i>	<i>Insertion Character</i>
ON	x	x	x	None
OFF	OFF	OFF	0	EIT(1) b

OFF	x	ON	0	EIT(4)	-
OFF	x	x	1	EIT(5)	\$
OFF	ON	OFF	0	EIT(3)	+

At completion of the micro operation, the sending field count is not changed; the receiving field count is decremented by one if an insertion occurred, otherwise it is not changed.

IGNORE SOURCE CHARACTER (IGN) MICRO OPERATION

This micro operation causes a specified number of bytes in the sending field to be skipped. The sending field count is reduced accordingly and no change is made in the receiving field count. The number of bytes to be skipped is specified by the IF entry. The Assembler stores zero in the IF field to represent 16.

INSERT ASTERISK ON SUPPRESS (INSA) MICRO OPERATION

This micro operation causes a character to be inserted in the receiving field. The character that is inserted depends on the state of the end suppression (ES) flag and the IF entry in the micro operation. If the ES flag is OFF, the IF entry is ignored and the character specified by EIT(2) (normally an asterisk) is inserted. If the ES flag is ON, the insertion character is specified by the IF entry as follows:

<i>IF Entry</i>	<i>Character</i>
1 through 8	EIT(1) through EIT(8)
0	Character specified in byte that follows the micro operation.

An IF entry of 9 through 15 causes an invalid specification (IS) trap.

At completion, the receiving field count is decremented by one; the sending field count is not changed.

INSERT BLANK ON SUPPRESS (INSB) MICRO OPERATION

This micro operation causes a character to be inserted in the receiving field. The character that is inserted depends on the state of the end suppression (ES) flag and the IF entry in the micro operation. If the ES flag is OFF, the IF entry is ignored and the character specified by EIT(1) (normally a blank) is inserted. If the ES flag is ON, the character is specified by the IF entry as follows:

<i>IF Entry</i>	<i>Character</i>
1 through 8	EIT(1) through EIT(8)
0	Character specified in byte that follows the micro operation.

An IF entry of 9 through 15 causes an invalid specification (IS) trap.

At completion, the receiving field count is decremented by one; the sending field count is not changed.

INSERT MULTIPLE CHARACTERS (INSM) MICRO OPERATION

This micro operation causes 1 to 16 EIT(1) characters — normally blanks — to be inserted into the receiving field. The receiving field count is decremented by the number of characters inserted; the sending field count is not changed. The number of characters inserted corresponds to the IF entry. The Assembler stores zero in the IF field to represent 16.

INSERT CHARACTER ON NEGATIVE (INSN) MICRO OPERATION

This micro operation causes a character to be inserted into the receiving field. The receiving field count is decremented by 1; the sending field count is not changed. The character that is inserted depends on the state of the SN flag and the IF entry. If the SN flag is OFF, the IF entry is ignored and the character specified by EIT(1) — normally a blank — is inserted. If the SN flag is ON, the insertion character is specified by the IF entry as follows:

<i>IF Entry</i>	<i>Character</i>
1 through 8	EIT(1) through EIT(8)
0	Character specified in byte that follows micro operation.

An IF entry of 9 through 15 causes an invalid specification (IS) trap.

At completion, the receiving field count is decremented by one; the sending field count is not changed.

INSERT CHARACTER ON POSITIVE (INSP) MICRO OPERATION

This micro operation causes a character to be inserted in the receiving field. The receiving field count is decremented by 1; the sending field count is not changed. The character that is inserted depends on the state of the PM flag, the state of the SN flag, and the IF entry.

If both the PM and SN flags are OFF, the character is specified by the IF entry. For IF entries 1 through 8, the character is specified by EIT(1) through EIT(8). For an IF entry of 0, the character is specified by the byte that follows the micro operation. The character inserted for other states of the flags is shown below.

<i>PM Flag</i>	<i>SN Flag</i>	<i>Character</i>
OFF	OFF	Specified by IF entry
OFF	ON	EIT(1) b
ON	OFF	EIT(3) +
ON	ON	EIT(4) -

MOVE WITH FLOAT CURRENCY SYMBOL INSERTION (MFLC) MICRO OPERATION

This micro operation floats the appropriate currency symbol over a specified number of sending field units as a function of the ES flag. (In this context, a sending field unit is 4 bits for packed decimal data; otherwise it is one byte.) The number of sending field units upon which the operation is to be performed corresponds to the IF entry. The Assembler stores zero in the IF field to represent 16.

Let the IF entry specify n sending field units. Then when the micro operation is executed, the next n units are fetched one at a time and the following actions occur.

- If the ES flag is OFF and the sending field unit is zero, the character specified by EIT(1) (normally a blank) is moved to the receiving field.
- If the ES flag is OFF and the sending field unit is not zero, the character specified by EIT(5) (normally a \$) is moved to the receiving field and then the non-zero unit is moved to the receiving field, and the ES flag is turned ON.
- If the ES flag is OFF, and all units in the sending field are zeros, then n EIT(1) characters are moved to the receiving field. The ES flag remains OFF.
- If the ES flag is ON, the next sending field unit is moved to the receiving field. If the ES flag is ON at initiation of the micro operation, then n sending field units are moved to the receiving field.
- If the sending field contains at least one leading zero unit and one non-zero unit, then n+1 units are moved to the receiving field and the ES flag is set ON.
- At completion of the micro operation, the receiving field count is decremented by either n or n+1; the sending field count is decremented by n.

MOVE WITH FLOAT SIGN INSERTION (MFLS) MICRO OPERATION

This micro operation floats the appropriate sign character over the specified number of sending field units as a function of the ES flag, the SN flag, and the PM flag, and whether the sending field unit is zero. (In this context, a sending field unit is 4 bits for packed decimal data; otherwise it is one byte.) The number of sending field units upon which the operation is to be performed corresponds to the IF entry. The Assembler stores zero in the IF field to represent 16.

Let the IF entry specify n sending field units. Then, when the micro operation is executed, the next n units are fetched one at a time and the following actions occur.

- If the ES flag is OFF, a character is moved to the receiving field as specified by Table 6-5. An EIT(1) character replaces a zero digit or zero character. An EIT(1) character or a sign character is inserted ahead of a nonzero digit or character in the receiving field; the nonzero unit is then moved to the receiving field, and the ES flag is set ON.
- If the ES flag is OFF, and all units in the sending field are zeros, then n EIT(1) characters are moved to the receiving field. The ES flag remains OFF.
- If the ES flag is ON, the next sending field unit is moved to the receiving field. If the ES flag is ON at initiation of the micro operation, then n sending field units are moved to the receiving field.
- If the sending field contains at least one leading zero unit and one non-zero unit, then n+1 units are moved to the receiving field and the ES flag is set ON.
- At completion of the micro operation, the receiving field count is decremented by either n or n+1; the sending field count is decremented by n.

TABLE 6-5. CHARACTER INSERTION BY MFLS MICRO OPERATION

ES Flag	PM Flag	SN Flag	Sending Field Unit	Character
OFF	x	x	= 0	EIT(1) b
OFF	OFF	OFF	≠ 0	EIT(1) b
OFF	x	ON	≠ 0	EIT(4) -
OFF	ON	OFF	≠ 0	EIT(3) +

MOVE SOURCE CHARACTER (MVC) MICRO OPERATION

This micro operation moves the specified number of sending field units to the receiving field. (In this context, a unit is 4 bits for packed decimal data; otherwise it is one byte.) The number of sending field units upon which the operation is to be performed corresponds to the IF entry. The Assembler stores zero in the IF field to represent 16. At completion of the operation, the sending field count and the receiving field count are decremented by the specified number of sending field units.

MOVE WITH ZERO SUPPRESSION AND ASTERISK REPLACEMENT (MVZA) MICRO OPERATION

This micro operation replaces sending field units that are zeros with asterisks as a function of the ES flag. (In this context, a unit is 4 bits for packed decimal data; otherwise it is one byte.) The number of sending field units upon which the operation is performed corresponds to the IF entry. The Assembler stores zero in the IF field to represent 16. Let the IF entry specify n sending field units. Then, when the operation is executed, the next n units are fetched one at a time and the following actions occur.

- If the ES flag is OFF and the sending field unit is zero, the character designated by EIT(2) (normally an asterisk) is moved to the receiving field.
- If the ES flag is OFF and the sending field unit is not zero, the sending field unit is moved to the receiving field and the ES flag is set ON.
- If the ES flag is ON, the sending field unit is moved to the receiving field. (If the ES flag is on at initiation of the micro operation, n sending field units are moved to the receiving field.)
- At completion of the micro operation, the sending field count and the receiving field count are decremented by n.

MOVE WITH ZERO SUPPRESSION AND BLANK REPLACEMENT (MVZB) MICRO OPERATION

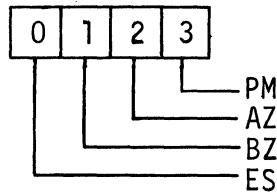
This micro operation replaces sending field units that are zeros with blanks as a function of the ES flag. (In this context, a unit is 4 bits for packed decimal data; otherwise it is one byte.) The number of sending field units upon which the operation is performed corresponds to the IF entry.

The Assembler stores zero in the IF field to represent 16. Let the IF entry specify n sending field units. Then, when the operation is executed, the next n units are fetched one at a time and the following actions occur.

- If the ES flag is OFF and the sending field unit is zero, the character designated by EIT(1) (normally a blank) is moved to the receiving field.
- If the ES flag is OFF and the sending field unit is not zero, the sending field unit is moved to the receiving field and the ES flag is set ON.
- If the ES flag is ON, the sending field unit is moved to the receiving field. (If the ES flag is on at initiation of the micro operation, n sending field units are moved to the receiving field.)
- At completion of the micro operation, the sending field count and the receiving field count are decremented by n.

SET EDIT FLAGS (SEF) MICRO OPERATION

This micro operation is used to control four edit flags. The IF field represents a 4-bit binary mask. Each bit is associated with an edit flag as shown by the following diagram.



The ES flag can be set ON or cleared OFF. The other edit flags can be set ON but cannot be cleared OFF by this micro operation. The sequence of events that occur during this micro operation are summarized below and illustrated in; Figure 6-7.

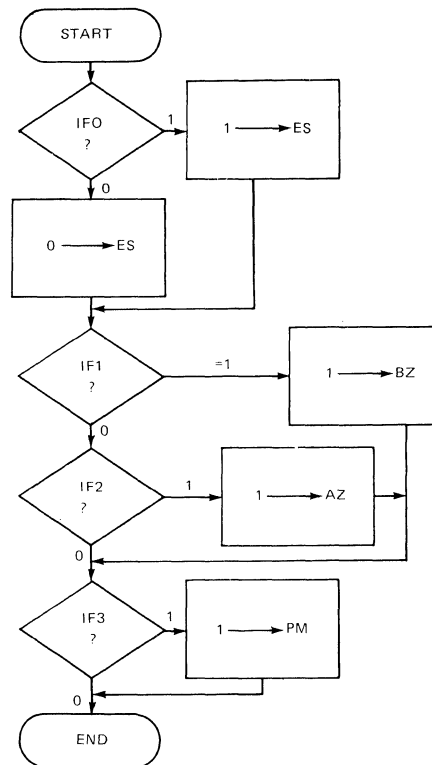


Figure 6-7. Flow Diagram of SEF Micro Operation

- If IF(0) = 1, the ES flag is set ON.
- If IF(0) = 0, the ES flag is cleared OFF.
- If IF(1) = 1, the BZ flag is set ON, and IF(2) is ignored.
- If IF(1) = 0, the BZ flag is not changed and IF(2) is interrogated.
- If IF(2) = 1 and IF(1) = 0, the AZ flag is set ON.
- If IF(2) = 0, the AZ flag is not changed.
- If IF(3) = 1, the PM flag is set ON.
- If IF(3) = 0, the PM flag is not changed.

This micro operation has no effect on the sending or receiving field count.

COMMERCIAL PROCESSOR TRAPS

The Commercial Processor trap facility monitors the execution of all Commercial Processor instructions and sends trap requests to the CPU whenever certain events occur. These traps and the events that cause them are shown in Table 6-6. The results of a Commercial Processor trap are as follows:

- The trapped Commercial Processor instruction is aborted.
- The operands of the Commercial Processor instruction usually remain unchanged.
- The bit (if any) of the Commercial Processor indicator that reflects the trap condition is changed.
- The bits of the Commercial Processor indicator register used by the trapped instruction are left in an unspecified state.
- All other bits of the Commercial Processor indicator remain unchanged.
- The CPU, upon completion of the current CPU instruction,
 - Interrogates the Commercial Processor for the trap vector code.
 - Saves CPU trap context in a trap save area.
 - Branches to the trap handling procedure specified by the trap vector.

If more than one trap condition exists, the Commercial Processor sends the condition with the highest priority to the CPU. The other condition(s) are lost. Trap conditions are usually detected in the order listed in Table 6-6.

Because Commercial Processor instructions are executed in parallel with CPU instructions, the following information about the Commercial Processor instruction being executed is stored when a trap occurs.

- At initiation of the Commercial Processor instruction:
 - CPU stores the address of the Commercial Processor instruction.
 - Commercial Processor stores the effective address(es) of the operand(s).
- At trap time:
 - CPU stores the trap context of the machine.
 - The A-word of the Trap Save Area will contain the address of the Commercial Processor instruction that caused the trap.
 - The P-word of the Trap Save Area will contain the address of the next sequential instruction.
 - A T&V routine's trap handler (or the trap handler of any other task having exclusive use of the Commercial Processor) can obtain the effective address(es) of the data descriptors of the trapped instruction by use of a read IOLD instruction directed to the Commercial Processor. The format of the trap context is shown in Figure 6-8. The Commercial Processor always uses 8 words of context regardless of the range information. The number of valid addresses in the trap context depends on the number of data descriptors required for the Commercial Processor instruction that trapped.

TABLE 6-6. COMMERCIAL PROCESSOR TRAP VECTORS AND EVENTS

Trap Vector Mnemonic and Number	Trap Event	C/U ^a	Commercial Indicator	Commercial Mask
TV (UR) #23	Reference to Unavailable Resource	U	—	—
TV (BE) #24	NML Bus or Memory Error	U	—	—
TV (IS) #26	Illegal Specification	U	—	—
TV (DZ) #25	Divide by Zero	U	—	—
TV (IC) #27	Illegal Character	U	—	—
TV (TR) #28	Truncation	C	CI (TR)	CM (TR)
TV (OV) #29	Overflow	C	CI (OV)	CM (OV)
TV #30	QLT error	U	—	—

^aC = conditional Trap
 U = unconditional Trap

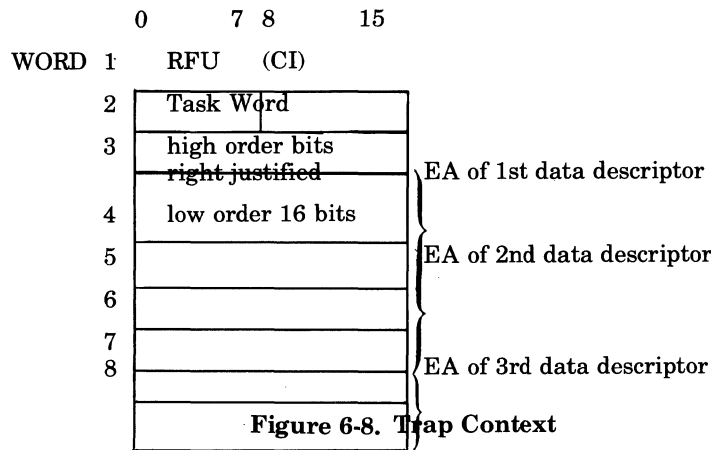


Figure 6-8. Trap Context

The following descriptions of Commercial Processor traps include a list of the conditions that cause each trap. During the execution of some Commercial Processor instructions, traps may be generated by conditions that are peculiar to the instruction. Where applicable, such conditions are included in the description of the Commercial Processor instruction.

TRAP 23 UNAVAILABLE RESOURCE (UR)

Condition causing trap:

The Commercial Processor has referenced an unavailable resource.

TRAP 24 BUS OR MEMORY ERROR (BE)

Conditions causing trap:

- The Commercial Processor has detected a bus or memory error.
- The Commercial Processor has detected a bus command that is out of sequence.
- The Commercial Processor has detected a Commercial Processor hardware error.

TRAP 25 DIVIDE BY ZERO (DZ)

This trap is unconditionally generated whenever the divisor of a decimal divide instruction is equal to zero.

TRAP 26 ILLEGAL SPECIFICATION (IS)

Conditions causing trap:

- An undefined Commercial Processor operation code is detected
- An operand has a zero length
- A separate signed operand consists of only a sign
- The second data descriptor contains an IMO address expression

TRAP 27 ILLEGAL CHARACTER (IC)

Conditions causing trap:

- An illegal decimal digit is detected (i.e., the low order four bits of a digit is not 0 through 9)
- An illegal sign digit is detected (i.e., a digit that is not one of the recognized signs)
- An illegal overpunch is detected.

TRAP 28 TRUNCATION (TR)

Conditions causing trap:

- For this trap to be generated, the TR-bit of the Commercial Processor mode register must be set to 1. (The contents of the Commercial Processor mode register can be changed by the CPU instruction MTM.) The trap is then generated if the receiving field cannot contain all characters of the result.

If truncation occurs, the TR-bit of the Commercial Processor indicator register is set to 1 and the leftmost part of the result is stored in the receiving field.

TRAP 29 OVERFLOW (OV)

Conditions causing trap:

- For this trap to be generated, the OV-bit of the Commercial Processor mode register must be set to 1. (The contents of the Commercial Processor mode register can be changed by the CPU instruction MTM.) The trap is then generated if the receiving field cannot contain all significant digits of the result (leading zeros are not considered significant).
If a trap occurs, the OV-bit of the Commercial Processor indicator register is set to 1, the instruction is aborted, and the original operands are not changed.
- If overflow occurs but no trap is generated (OV-bit of Commercial Processor mode register is 0), the receiving field will not contain the most significant digit(s) of the result. The OV-bit of the Commercial Processor indicator register will be set to 1.

TRAP 30 QUALITY LOGIC TEST (QLT) ERROR (QE)

Conditions causing trap:

- Any malfunction detected during a Commercial Processor QLT (mini or maxi) generates this trap.
Normally, the operating system will handle this trap. For further information, refer to the *Level 6 Minicomputer Handbook*.

EXECUTION DETAILS FOR COMMERCIAL INSTRUCTIONS

Unless indicated otherwise in the description of the individual instruction, the details given below apply to all instructions in the following categories:

Decimal arithmetic
Radix and mode conversion
Shift

- Data descriptors may be generated in line as part of the instruction or they may be referenced by internal value expressions

- If remote descriptor references are generated, the internal value expression specifies the offset from the address stored in the remote descriptor base register (in units of double words) to the desired descriptor. The value of the internal value expression can range from 0 through 4095.
- The effective address developed from a descriptor points to the leftmost digit of the operand.
- The G and L bits of the Commercial Processor indicator register indicate the value of the result relative to zero.
- If the result is shorter than the receiving field, the receiving field will be zero filled to the left.
- Plus (+) and minus (−) zero are allowed on input but are assumed to be plus zero during instruction execution. All zero operands generated by the Commercial Processor are plus zeros.
- The SF-bit of the Commercial Processor indicator register is set to 1 when a negative operand is to be stored in an unsigned field.
- Operands having different sign conventions are allowed.
- Unpacked decimal zero is represented by 30 (hexadecimal); packed decimal zero is represented by 0 (hexadecimal).
- All signed results will have the hardware generated signs, regardless of the sign convention used by the operands at execution time.
- Zone bits (leftmost four bits of an unpacked decimal number) are ignored on input. The zone bit of all results is set to 3 (hexadecimal).
- Identical overlapping of operands is allowed. If operands overlap in any other way, the results are unspecified.

Unless otherwise indicated, the following details apply to all character string instructions.

- Data descriptors may be generated in line as part of the instruction or referenced by internal value expressions.
- If remote descriptor references are generated, the internal value expression specifies the offset from the address stored in the remote descriptor base register (in units of double words) to the desired descriptor. The value of the internal value expression can range from 0 through 4095.
- Identical overlapping of operands is allowed. If operands overlap in any other way, the results are unspecified.
- Trap 26, illegal specification, is generated if the second operand specifies an IMO. (Exception is alphanumeric compare ACM instruction.)
- Trap 26, illegal specification, is generated if the third operand specifies an IMO.
- The truncation bit (TR bit) of the Commercial Processor indicator register is set to 1, if the receiving field cannot accept all characters of the result.
- Only the leftmost portion of the result will be in the receiving field, if the receiving field cannot accept all the characters of the result.

DETAILED DESCRIPTIONS OF COMMERCIAL INSTRUCTIONS

The remainder of this section contains detailed descriptions of each commercial instruction. The descriptions are arranged in alphabetical sequence by instruction mnemonics. Each description includes the name, type, format, and operands, and a functional explanation. Commercial Processor data descriptors and Commercial Processor address expressions are previously described in this section. Symbolic names, constants, and expressions (other than address expressions) are described in Section 2.

All commercial instructions can be labeled, although labels are not shown in the source language formats of the descriptions that follow.

A summary of the commercial instructions is given in Table H-1 of Appendix H.

ACM

ACM

Instruction:

Alphanumeric compare

Type:

Character string

Source Language Format:

$$\Delta ACM \Delta \left\{ \begin{array}{l} \text{DESCA (description)} \\ \text{int-val-expression} \end{array} \right\} , \left\{ \begin{array}{l} \text{DESCA (description)} \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The character string specified by the first operand is compared with the character string specified by the second operand. The comparison proceeds character-by-character from left to right. When a mismatch occurs, the binary value of the two unlike characters determine whether the first character string is greater or less than the second, and the G or L bit of the Commercial Processor indicator register is set accordingly.

If the length of the strings are unequal, the shorter string is unconditionally extended to the right with the fill character specified by the second data descriptor. Note that the extension takes place internally in the Commercial Processor and not in main memory.

If the length of both strings are zero, the strings are considered to be equal.

Both operands may specify IMO's.

If the value of the byte length specified by the first data descriptor is zero, the length is contained in the right byte of register R4 and can be from 0 through 255 bytes. If the value of the byte length specified in the first data descriptor is not zero, that value, which can be from 1 through 31, is the length.

If the value of the byte length specified by the second data descriptor is zero, register R5 contains the fill character (in the left byte) and the length (in the right byte). When escape to register R5 occurs, the length can be from 1 through 255 characters. If the value of the byte length specified in the second data descriptor is not zero, that value is the length and the fill character is an ASCII blank (20 hexadecimal). In this case, the length can be from 1 through 31 bytes.

Applicable Traps

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal Specification

The contents of the Commercial Processor indicator register are affected as follows:

- If the value of the first operand string is less than the value of the second operand string, the L-bit is set to 1; otherwise, it is set to 0.
- If the value of the first operand string is greater than the value of the second operand string, the G-bit is set to 1; otherwise, it is set to 0.

ALR

Instruction:
Alphanumeric move

Type:
Character string

Source Language Format:

$$\Delta ALR \Delta \left\{ \begin{array}{l} \text{DESCA(description)} \\ \text{int-val-expression} \end{array} \right\} \left\{ \begin{array}{l} \text{DESCA(description)} \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The character string is moved from the address specified by the first operand (sending field) to the address specified by the second operand. If the length of the receiving field is zero, the TR-bit (truncation bit) of the Commercial Processor indicator register is set to 1, and the instruction is aborted. Trap 28, truncation, may then be generated as described previously under "Commercial Processor Traps."

If the length of the sending field is zero, the receiving field is filled or not as specified by the second data descriptor.

If the value of the byte length specified by the first data descriptor is zero, the length is contained in the right byte of register R4 and can be from 0 through 255 bytes. If the value of the byte length specified in the first data descriptor is not zero, that value, which can be from 1 through 31, is the length.

If the value of the byte length specified by the second data descriptor is zero, register R5 contains the fill character (in the left byte) and the length (in the right byte). When escape to register R5 occurs, the length can be from 1 through 255 characters. If the value of the byte length specified in the second data descriptor is not zero, that value is the length, and the fill character is an ASCII blank (20 hexadecimal). In this case, the length can be from 1 through 31 bytes.

Applicable Traps:

- Trap 23 Reference to unavailable resource
- Trap 24 Bus or memory error
- Trap 26 Illegal specification
- Trap 28 Truncation

The contents of the Commercial Processor indicator register are affected as follows:

- If the length of the first operand string is greater than the length of the second operand string, the TR-bit is set to 1; otherwise, it is set to 0.

AME

AME

Instruction:

Alphanumeric move and edit

Type:

Edit

Source Language Format:

$$\Delta\text{AME}\Delta \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The character string in the sending field specified by the first data descriptor (DD1) is edited in accordance with the micro operations in the field specified by the third data descriptor (DD3), and moved to the receiving field specified by the second data descriptor (DD2).

The number of edited characters stored in the receiving field can be either more or less than those in the sending field. The receiving field may have more characters when micro operations specify one or more characters are to be inserted. The receiving field may have less characters when a micro operation specifies that one or more characters of the sending field are to be skipped.

The instruction terminates normally when the receiving field is filled. Normal termination occurs even though the sending field or the string of micro operations have not been exhausted.

An illegal specification trap (Trap 26) is generated if either the sending field or the string of micro operations are exhausted before the receiving field is filled.

Execution details are as follows:

- The effective address developed from a data descriptor points to the leftmost character of the operand.
- All operations take place from left to right.
- The valid length of the sending field, the receiving field, and the string of micro operations ranges from 1 through 255. Lengths from 32 through 255 are specified via escape to an R register. (See Appendix H.)
- During execution of the instruction, the sending field count indicates the current number of characters remaining to be processed. The count is decremented every time a character is moved out or skipped over.
- During execution of the instruction, the receiving field count indicates the current number of positions that remain to be filled. The count is decremented every time a character is moved into the receiving field.
- The Edit Insertion Table (EIT) is always initialized when the edit instruction is initiated.
- The edit flags are always initialized when the edit instruction is initiated.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal Specification

Conditions causing trap:

- The sending field or the string of micro operations is exhausted before the receiving field is filled.
- The length of the sending field, or the receiving field, or the string of micro operations is zero.

CBD

Instruction:

Convert binary to decimal

Type:

Radix and mode conversion

Source Language Format:

$$\Delta\text{CBDA}\Delta \left\{ \begin{array}{l} \text{DESCB}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The binary value specified by the first operand is converted to the decimal data type specified by the second operand and stored, right justified, at the address specified by the second operand.

Execution Details:

- If the length of the receiving field is greater than the number of significant digits in the decimal equivalent of the binary value, the receiving field is zero filled to the left.
- The high order bit of the binary operand is a sign bit (i.e., the binary value is stored in two's complement notation).
- If the first data descriptor specifies and IMO, the results are unspecified.

Applicable Traps:

During execution of this instruction, Traps 23, 24, 26, and 29 may be generated in the same way as during execution of the decimal add instruction DAD.

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal specification

Trap 29 Overflow

The contents of the Commercial Processor indicator register are affected as follows:

- If the number of significant digits in the converted result is greater than the number of digit positions available in the receiving field, the OV-bit is set to 1; otherwise, it is set to 0.
- If the value to be converted is negative and the receiving field is described as unsigned, the SF-bit is set to 1; otherwise, it is set to 0.

CBE

CBE

Instruction:

Commercial Branch if equal

Source Language Format:

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

Description:

Branches to the location specified by the operand if the G-bit and the L-bit of the CIP indicator register are both 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.

CBG

Instruction:

Commercial Branch if greater

Source Language Format:

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

Description:

Branches to the location specified by the operand if the G-bit of the Commercial Processor indicator register is 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.

CBGE

CBGE

Instruction:

Commercial Branch if greater than or equal

Source Language Format:

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

Description:

Branches to the location specified by the operand if the L-bit of the Commercial Processor indicator register is 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.

CBL

Instruction:

Commercial Branch if less than

Source Language Format:

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

Description:

Branches to the location specified by the operand if the L-bit of the Commercial Processor indicator register is 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.

CBLE

CBLE

Instruction:

Commercial Branch if less than or equal

Source Language Format:

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

Description:

Branches to the location specified by the operand if the G-bit of the Commercial Processor indicator register is 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.

CBNE

Instruction:

Commercial Branch if not equal

Source Language Format:

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

Description:

Branches to the location specified by the operand if either (but not both) the G-bit or the L-bit of the Commercial Processor indicator register is 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.

CBNOV

CBNOV

Instruction:

Commercial Branch if no overflow

Source Language Format:

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

Description:

Branches to the location specified by the operand if the OV-bit of the Commercial Processor indicator register is 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.

CBNSF

Instruction:

Commercial Branch if no sign fault

Source Language Format:

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

Description:

Branches to the location specified by the operand if the SF-bit of the Commercial Processor indicator register is 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.

CBNTR

CBNTR

Instruction:

Commercial Branch if no truncation

Source Language Format:

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

Description:

Branches to the location specified by the operand if the TR-bit of the Commercial Processor indicator register is 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.

CBOV

Instruction:

Commercial Branch on overflow

Source Language Format:

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

Description:

Branches to the location specified by the operand if the OV-bit of the Commercial Processor indicator register is 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.

CBSF

CBSF

Instruction:

Commercial Branch on sign fault

Source Language Format:

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

Description:

Branches to the location specified by the operand if the SF-bit of the Commercial Processor indicator register is 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.

CBTR

Instruction:

Commercial Branch on truncation

Source Language Format:

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

Description:

Branches to the location specified by the operand if the TR-bit of the Commercial Processor indicator register is 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.

CDB

CDB

Instruction:

Convert decimal to binary

Type:

Radix and mode conversion

Source Language Format:

$$\Delta\text{CDB}\Delta \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCB}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The decimal value specified by the first operand is converted to binary and stored in two's complement notation at the address specified by the second operand.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal specification

Trap 27 Illegal character

Trap 29 Overflow

The contents of the Commercial Processor indicator register are affected as follows:

- If the number of significant bits in the converted result is greater than the number of bit positions available in the receiving field (i.e., 15 bits for 2 bytes or 31 bits for 4 bytes), the OV-bit is set to 1; otherwise it is set to 0.

CSNCB

Instruction:

Commercial Synchronize and branch

Source Language Format:

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

Description:

Branches to the location specified by the operand after the previous Commercial Processor instruction has been completed.

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.

CSYNC

CSYNC

Instruction:

Commercial Synchronize

Type:

Branch

Source Language Format:

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

Description:

Prevents the CPU from going to the next instruction until the previous Commercial Processor instruction has been completed. Performs no operation.

DAD

Instruction:

Decimal add

Type:

Decimal arithmetic

Source Language Format:

$$\Delta DADA \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

Adds the decimal value at the address specified by the first operand to the decimal value at the address specified by the second operand and stores the result at the address of the second operand.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal Specification

Trap 27 Illegal Character

Trap 29 Overflow

Whenever Trap 23 or Trap 24 occurs, the preservation of the original operands cannot be guaranteed. If any other trap occurs, the operands remain unchanged.

The contents of the Commercial Processor indicator register are affected as follows:

- If the number of significant digits in the result is greater than the number of digit positions available in the receiving field, the OV-bit is set to 1; otherwise, it is set to 0.
- If the result is negative and the receiving field is described as unsigned, the SF-bit is set to 1; otherwise, it is set to 0.
- If the result is less than zero, the L-bit is set to 1; otherwise, it is set to 0.
- If the result is greater than zero, the G-bit is set to 1; otherwise, it is set to 0.

DCM

DCM

Instruction:

Decimal compare

Type:

Decimal arithmetic

Source Language Format:

$$\Delta\text{DCM}\Delta \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The decimal value specified by the first operand is compared algebraically with the decimal value specified by the second operand.

Execution Details:

- If the number of digits in the decimal values are not the same, the shorter one is supplied with leading zeros.
- Plus and minus zero are treated as equal.
- The data descriptors may specify immediate memory operands (IMO's).

*

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal Specification

Trap 27 Illegal character

The contents of the Commercial Processor indicator register are affected as follows:

- If the value specified by the first operand is less than the value specified by the second operand, the L-bit is set to 1; otherwise, it is set to 0.
- If the value specified by the first operand is greater than the value specified by the second operand, the G-bit is set to 1; otherwise, it is set to 0.

DDV

Instruction:

Decimal divide

Type:

Decimal arithmetic

Source Language Format:

$$\Delta DDV\Delta \left\{ \begin{array}{l} \text{DESCP(description)} \\ \text{DESCU(description)} \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCP(description)} \\ \text{DESCU(description)} \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCP(description)} \\ \text{DESCU(description)} \\ \text{int-val-expression} \end{array} \right\}$$

Description:

Divides the decimal value (the dividend) at the address specified by the second operand by the decimal value (the divisor) at the address specified by the first operand. Places the quotient at the address specified by the third operand. Places the remainder at the address specified by the second operand. If the absolute value of the divisor is greater than that of the dividend, the quotient is zero and the dividend and the divisor remain unchanged.

- If the sign of DD1 is the same as that of DD2, the sign of the quotient is +.
- If the sign of DD1 is not the same as that of DD2, the sign of the quotient is -.
- The sign of the remainder is always the same as that of the dividend (DD2) unless the remainder is zero.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 25 Divide by zero

Trap 26 Illegal specification

Trap 27 Illegal character

Trap 29 Overflow

The contents of the Commercial Processor indicator register are affected as follows:

- If the number of significant digits in the quotient is greater than the number of digit positions available in the receiving field, the OV-bit is set to 1; otherwise, it is set to 0.
- If the quotient is negative and the receiving field is described as unsigned, the SF-bit is set to 1; otherwise, it is set to 0.
- If the quotient is less than zero, the L-bit is set to 1; otherwise, it is set to 0.
- If the quotient is greater than zero, the G-bit is set to 1; otherwise, it is set to 0.

*

DLS

DLS

Instruction:

Decimal left shift

Type:

Shift

Source Language Format:

$$\Delta DLSA \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\} [,\text{int-val-expression}]$$

Description:

The decimal value specified by the first operand is shifted left. The vacated digit positions are zero filled. The second operand, if present, specifies the distance (number of digits shifted) and it must be an integer from 0 through 31.

When the second operand is present, the assembler:

- Sets shift control word 1 (SCW1) to 0178 (hexadecimal)
- Clears bit 0 of SCW2 to 0 (i.e., left shift)
- Loads the value specified by the second operand in bits 3 through 7 of SCW2
- Clears bit 8 of SCW2 to 0 (i.e., no rounding)

When the second operand is omitted, the assembler generates the shift control words as it does for the DSH instruction when the second operand is omitted. The shift direction and the distance must then be obtained from register R5. For an explanation of shift control words, see Decimal Shift instruction DSH.

Applicable Traps:

The traps that may be generated during execution of this instruction are the same as those for the DSH instruction.

Note that only one shift instruction, decimal shift (DSH), is available in the hardware. The decimal left shift (DLS) and the decimal right shift (DRS) instruction are provided by the Assembler for the programmer's convenience.

DMC

Instruction:

Decimal move and convert

Type:

Radix and mode conversion

Source Language Format:

$$\Delta\text{DMCA} \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The decimal value of the data type specified by the first operand is converted to the data type specified by the second operand and stored, right justified, at the address specified by the second operand.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal specification

Trap 27 Illegal character

Trap 29 Overflow

The contents of the Commercial Processor indicator register are affected as follows:

- If the number of significant digits in the sending field is greater than the number of digit positions available in the receiving field, the OV-bit is set to 1; otherwise, it is set to 0.
- If the value contained in the sending field is negative and the receiving field is described as unsigned, the SF-bit is set to 1; otherwise, it is set to 0.
- If the value being operated on is less than zero, the L-bit is set to 1; otherwise, it is set to 0.
- If the value being operated on is greater than zero, the G-bit is set to 1; otherwise, it is set to 0.

DME

DME

Instruction:

Decimal move and edit

Type:

Edit

Source Language Format:

$$\Delta DME \Delta \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The decimal digits in the sending field specified by the first data descriptor (DD1) are edited in accordance with the micro operations in the field specified by the third data descriptor (DD3), and moved to the receiving field specified by the second data descriptor. If the sending field contains packed decimals they are converted to unpacked decimals before they are stored in the receiving field.

The number of edited characters stored in the receiving field can be either more or less than the number of digits in the sending field. The receiving field may have more characters when micro operations specify one or more characters are to be inserted. The receiving field may have less characters when a micro operation specifies that one or more digits are to be skipped.

The instruction terminates normally when the receiving field is filled. Normal termination occurs even though the sending field or the string of micro operations have not been exhausted.

An illegal specification trap (Trap 26) is generated if either the sending field or the string of micro operations are exhausted before the receiving field is filled.

Execution details are as follows:

- The effective address developed from a data descriptor points to the leftmost character of the operand.
- All operations take place from left to right.
- The valid length of the sending field ranges from 1 through 31 digits.
- The valid length of the receiving field and the string of micro operations ranges from 1 through 255 characters. Lengths from 32 through 255 can be specified via escape to an R register. (See Appendix H.)
- During execution of the instruction, the sending field count indicates the current number of digits remaining to be processed. The count is decremented every time a digit is moved out or skipped over.
- During execution of the instruction, the receiving field count indicates the current number of positions that remain to be filled. The count is decremented every time a character is moved into the receiving field.
- The Edit Insertion Table (EIT) is always initialized when the edit instruction is initiated.
- The edit flags are always initialized when the edit instruction is initiated.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal Specification

Conditions causing trap:

- The sending field or the string of micro operations is exhausted before the receiving field is filled.

- The length of the sending field, or the receiving field, or the string of micro operations is zero.

Trap 27 Illegal character

EXAMPLES OF DME (DECIMAL MOVE AND EDIT) INSTRUCTION

Example 1: Full Zero Suppression

Up to six digits are moved with leading zeros suppressed.

```

MOP1      TEXT      MVZB(6)
.
.
.
DME       DESCN(NUMBER,0,6,UNSIGNED);
          DESCA(OUTPUT,0,6);
          DESCA(MOP,0,1)
    
```

The results of this instruction for various inputs are as follows:

NUMBER	OUTPUT
012345	12345
000123	123
000000	123456

Example 2: Partial Zero Suppression

The last three digits are moved whether they are leading zeros or not.

```

MOP2      TEXT      MVZA(3),MVC(3)
.
.
.
DME       DESCU(NUMBER,0,6,UNSIGNED);
          DESCA(OUTPUT,0,6);
          DESCA(MOP2,0,2)
    
```

The results of this instruction for various inputs are as follows:

NUMBER	OUTPUT
012345	*12345
000123	***123
000000	***000

Example 3: Floating Sign Insertion

The floating sign (+ or -) is supplied in front of the first nonzero digit. If entire sending field is zeros, no sign is supplied.

```

MOP3      TEXT      SEF(1),MFLS(6),INSM(1)
.
.
.
DME       DESCU(NUMBER,0,7,TRAILING);
          DESCA(OUTPUT,0,7);
          DESCA(MOP3,0,3)
    
```

The results of this instruction for various inputs are as follows:

NUMBER	OUTPUT
123456+	+123456
001234-	1234-
000000+	123456

DME

Note that the INSM micro operation is required for an input of all zeros. In this case, no sign is supplied and, without the INSM, the micro operation field would be exhausted before the receiving field is filled resulting in a trap 26. The INSM is not executed if NUMBER contains a nonzero digit.

Example 4: Negative Sign Only Insertion

The negative sign (-) is supplied if the input is negative; a space is supplied if the input is positive. Same as Example 3, except that a space is inserted if the sending field is positive.

```
MOP4      TEXT      MFLS(6),INSM(1)
.
.
.
DME       DESCU(NUMBER,0,7,LEADING);
          DESCA(OUTPUT,0,7);
          DESCA(MOP4,0,4)
```

The results of this instruction for various inputs are as follows:

NUMBER	OUTPUT
+123456	123456
-000123	-123
+000000	000000

Example 5: Floating Currency Symbol Insertion

In this example the sending field NUMBER has two assumed decimal places. The currency symbol is forced to the left of the decimal point. The first micro operation applies to the first five digits of the data and causes a \$ to be placed in front of the first nonzero digit. If the first five digits are all zero, the ENF(8) micro operation forces the insertion of the \$ and turns off further suppression. The third micro operation forces the sixth digit to be moved as is, and the last two micro operations moves a decimal point and two more digits to the receiving field.

```
MOP5      TEXT      MFLC(5),ENF(8),MVC(1),INSB(7),MVC(2)
.
.
.
DME       DESCU(NUMBER,0,8,UNSIGNED);
          DESCA(OUTPUT,0,10);
          DESCA(MOP5,5)
```

The results of this instruction for various inputs are as follows:

NUMBER	OUTPUT
12345678	\$123456.78
00012345	\$123.45
00000123	\$1.23
00000012	\$0.12
00000000	\$0.00

Example 6: Separation of Data

In this example, the sending field NUMBER consists of eight digits. The four leftmost digits represent quantity. The four rightmost digits represent price, which has two assumed decimal places. The micro operations specified by MOP6 function as follows:

MVZB(4)	Moves four digits and suppresses leading zeros
INSB(1)	Inserts one space
INSB(0),'@'	Inserts at sign
INSB(1)	Inserts one space
INSB(5)	Inserts dollar sign
SEF(0)	Turns ES flag OFF
MVZA(2)	Moves next two digits, replacing leading zeros with asterisks

INSB(7) Inserts decimal point
 MVC(2) Moves last two digits
 MOP6 TEXT MVZB(4),INSB(1),INSB(0),'@',INSB(1),INSB(5);
 SEF(0),MVZA(2),INSB(7),MVC(2)
 .
 .
 DME DESCU(NUMBER,0,8,UNSIGNED);
 DESCA(OUTPUT,0,12);
 DESCA(MOP6,0,10)

The results of this instruction for various inputs are as follows:

<i>NUMBER</i>	<i>OUTPUT</i>
00340010	34 @ \$** .10
09900110	990 @ \$*1.10
70810025	7081 @ \$** .25
00505000	50 @ \$50.00

DML

DML

Instruction:

Decimal multiply

Type:

Numeric

Source Language Format:

$$\Delta DML \Delta \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

Multiplies the decimal value (the multiplier) at the address specified by the first operand by the decimal value (the multiplicand) at the address specified by the second operand and stores the result (the product) at the address specified by the second operand.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal specification

Trap 27 Illegal character

Trap 29 Overflow

The contents of the Commercial Processor indicator register are affected as follows:

- If the number of significant digits in the product is greater than the number of digit positions available in the receiving field, the OV-bit is set to 1; otherwise, it is set to 0.
- If the product is negative and the receiving field is described as unsigned, the SF-bit is set to 1; otherwise, it is set to 0.
- If the product is less than zero, the L-bit is set to 1; otherwise, it is set to 0.
- If the product is greater than zero, the G-bit is set to 1; otherwise, it is set to 0.

DRS

Instruction:

Decimal right shift

Type:

Shift

Source Language Format:

$$\Delta\text{DRSA} \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\} \quad [,\text{int-val-expression} \quad [,\text{R}[\text{OUNDED}]]]$$

Description:

The decimal value specified by the first operand is shifted right. The vacated digit positions are zero filled. The second operand, if present, specifies the distance (number of digits shifted) and must be an integer from 0 through 31.

When the second operand is present, the assembler:

- Sets shift control word 1 (SCW1) to 0178 (hexadecimal).
- Sets bit 0 of SCW2 to 1 (i.e., right shift).
- Loads the value specified by the second operand in bits 3 through 7 of SCW2.
- Sets bit 8 of SCW2 to 1, if the third operand is present (i.e., rounding).
- Clears bit 8 of SCW2 to 0, if the third operand is absent (i.e., no rounding).

When the second and third operands are omitted, the assembler generates the shift control words as it does for the DSH instruction when the second operand is omitted. The shift direction, the distance, and the rounding control must then be obtained from register R5. For an explanation of shift control words, see Decimal Shift instruction DSH.

Applicable Traps:

The traps that may be generated during execution of this instruction are the same as those for the DSH instruction.

Note that only one shift instruction, decimal shift (DSH), is available in the hardware. The decimal left shift (DLS) and the decimal right shift (DRS) instruction are provided by the Assembler for the programmer's convenience.

DSB

DSB

Instruction:

Decimal subtract

Type:

Decimal arithmetic

Source Language Format:

$$\Delta\text{DSBA} \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

Subtracts the decimal value (the subtrahend) at the address specified by the first operand from the decimal value (the minuend) at the address specified by the second operand and stores the result (the difference) at the address specified by the second operand.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal specification

Trap 27 Illegal Character

Trap 29 Overflow

The contents of the Commercial Processor indicator register are affected as follows:

- If the number of significant digits in the difference is greater than the number of digit positions available in the receiving field, the OV-bit is set to 1; otherwise, it is set to 0.
- If the difference is negative and the receiving field is described as unsigned, the SF-bit is set to 1; otherwise, it is set to 0.
- If the difference is less than zero, the L-bit is set to 1; otherwise, it is set to 0.
- If the difference is greater than zero, the G-bit is set to 1; otherwise, it is set to 0.

DSH

Instruction:

Decimal shift

Type:

Shift

Source Language Format:

$$\Delta\text{DSHA} \left\{ \begin{array}{l} \text{DESCP}(\text{description}) \\ \text{DESCU}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{int-val-expression} \\ \text{ext-val-expression} \end{array} \right\}$$

Description:

The decimal value designated by the first operand is shifted the distance (number of digits) and in the direction specified by the shift control words. The digit positions that are vacated are zero filled. Rounding may be specified for a right shift. If rounding is specified and the value of the last digit shifted out is from 5 through 9, the absolute value of the operand is increased by one; e.g., +12 becomes +13; -12 becomes -13.

The decimal shift instructions use two shift control words SCW1 and SCW2 as shown in Figure 6-9.

The second operand, if present, specifies the value of SCW2, and the assembler generates 0178 (hexadecimal) as the value of SCW1. The shift control information is taken from SCW2.

If the second operand is omitted, the Assembler generates 0078 (hexadecimal) and 0000 (hexadecimal) as the values of SCW1 and SCW2, respectively, and the shift control information is taken from register R5. The sign character, if any, is not affected by the shift operation (i.e., it does not get shifted).

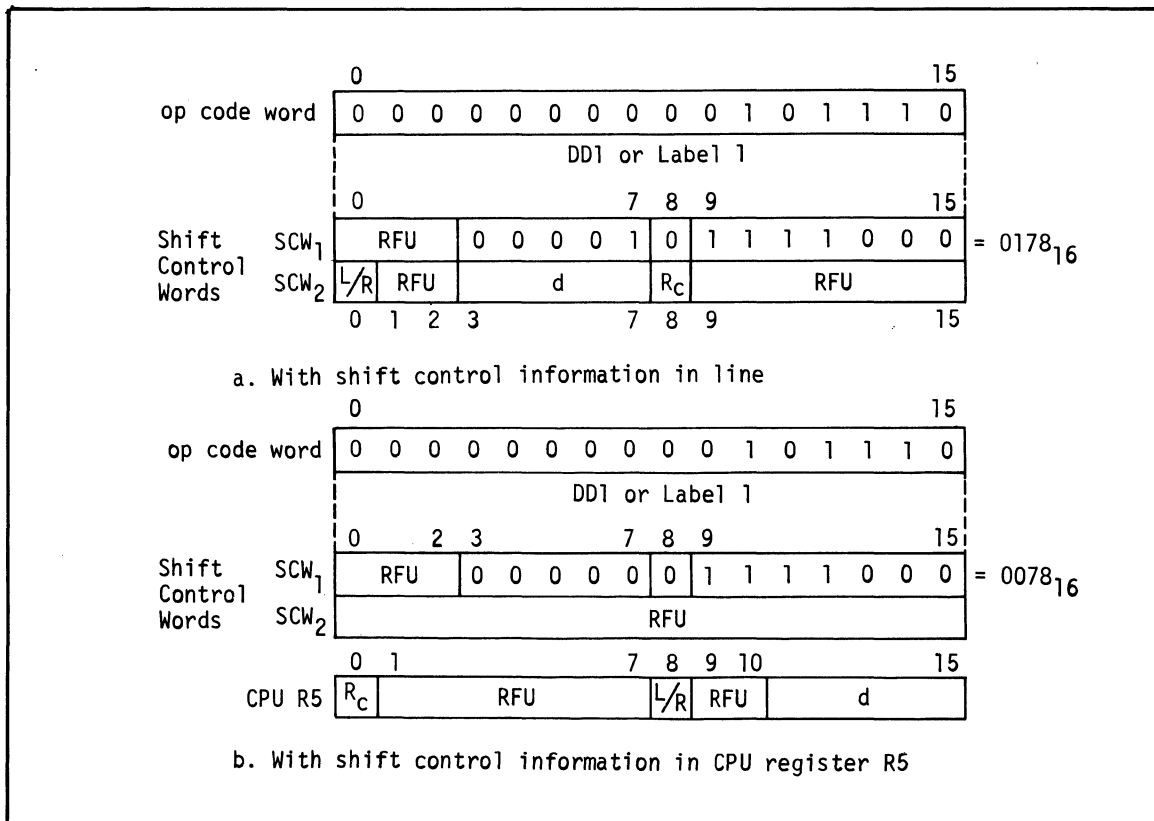


Figure 6-9. Shift Instruction Formats

DSH

Note that the formats of SCW2 and R5 are reversed; i.e., the information that is assigned to the first byte of SCW2 is assigned to the second byte of R5, and that assigned to the second byte of SCW2 is assigned to the first byte of R5. When the information comes from R5, the left and right bytes must be reversed on the megabus for the Commercial Processor to execute the instruction successfully.

Shift control information is specified by SCW2 or by register R5 as follows.

- Direction of shift: bit 0 of SCW2; bit 8 of R5
 - bit = 0 left shift
 - bit = 1 right shift
- Distance of shift in digits: bits 3 through 7 of SCW2; bits 11 through 15 of R5
 - range = 0 through 31
- Rounding: bit 8 of SCW2; bit 0 of R5
 - bit = 0 do not round
 - bit = 1 round after shifting right (ignored for left shift)

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal specification

During execution of this instruction, this trap is generated by the conditions previously listed under the heading "Commercial Processor Traps." For this instruction, this trap is also generated if the first operand is an IMO.

Trap 27 Illegal character

The conditions that generate this trap are the same as those previously listed under the heading "Commercial Processor Traps." However, checks for illegal characters are performed only upon completion of the shift. Thus, any illegal characters shifted out will not be checked.

Trap 29 Overflow

The OV-bit of the Commercial Processor indicator register is set to 1, if a non-zero digit is shifted out, Trap 29 may then be generated as previously described under the heading "Commercial Processor Traps."

The contents of the Commercial Processor indicator register are affected as follows:

- If a left shift specifies a shift distance greater than the number of leading zeros in the value to be shifted (i.e., one or more significant digits are lost) or a right shift specifies a shift distance greater than the number of trailing zeros in the value to be shifted, the OV-bit is set to 1; otherwise, it is set to 0.
- If the shifted value is less than zero, the L-bit is set to 1; otherwise, it is set to 0.
- If the shifted value is greater than zero, the G-bit is set to 1; otherwise, it is set to 0.

MAT

Instruction:

Alphanumeric move and translate

Type:

Character string

Source Language Format:

$$\Delta\text{MAT}\Delta \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\} \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\} \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The character string in the sending field (specified by the first data descriptor) is translated and moved to the receiving field (specified by the second data descriptor). The third data descriptor specifies a 256-byte translation table. Each character in the sending field is used as a displacement from the base of the table and the corresponding character from the table is stored in the receiving field.

If the byte length specified by the first data descriptor is zero, the length is contained in the right byte of register R4 and can be from 0 through 255 bytes. If the byte length specified in the first data descriptor is not zero, that value, which can be from 1 through 31, is the length. If the length of the sending field specified by register R4 is zero, the receiving field is filled or not filled as specified by the second data descriptor. Fill characters, if specified, are ASCII blanks and are not translated.

If the byte length specified in the second data descriptor is not zero, that value, which can be from 1 through 31, is the length. If the byte length specified by the second data descriptor is zero, the length is contained in register R5 and can be from 0 through 255 bytes. If the length of the receiving field specified by register R5 is zero, the instruction is aborted and the truncation bit (TR bit) of the Commercial Processor indicator register is set to 1. Trap 28 (truncation) may then be generated as previously described under "Commercial Processor Traps."

The length field of the third data descriptor is ignored by the hardware.

The contents of the Commercial Processor indicator register are affected as follows:

- If the number of characters in the sending field is greater than the number of character positions in the receiving field, the TR-bit is set to 1; otherwise, it is set to 0.

Example:

IN	DC	=Z'00020409'
TR	DC	= 'abcdefg\$.!''
OUT	RESV	4, ' '
	.	
	.	
	.	
MAT		DESCA(IN,0,4,NO_FILL);
		DESCA(OUT,0,4,NO_FILL);
		DESCA(TR,0,11,NO_FILL)

After execution of the MAT instruction the receiving field OUT will contain the following string: ace!

SRCH

SRCH

Instruction:

Alphanumeric search

Type:

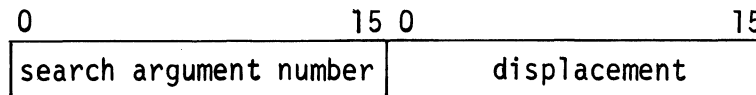
Character string

Source Language Format:

$$\Delta\text{SRCH}\Delta \left\{ \begin{array}{l} \text{DESCA(description)} \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCA(description)} \\ \text{int-val-expression} \end{array} \right\}, \left\{ \begin{array}{l} \text{DESCA(description)} \\ \text{int-val-expression} \end{array} \right\}$$

Description:

The character string or array of character strings defined by the third data descriptor (DD3) is searched to see if it contains any of the search arguments (one or more) in the search list defined by the first data descriptor (DD1). If a match is found, the G and L bits of the Commercial Processor indicator register are cleared to zero, and the displacement and search argument number are loaded into the receiving field defined by the second data descriptor (DD2). The receiving field must be four bytes long and word aligned, otherwise the results are unspecified. The displacement is the distance in bytes between the origin of the string (or array) to be searched and the position at which the first match occurs. The search argument number designates the one that caused the match. The first argument in the list is identified as 0, the second as 1, etc. The format of the receiving field is shown below.

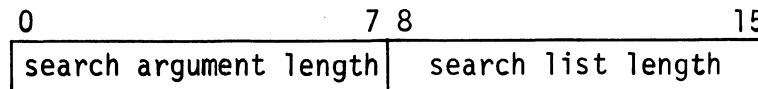


If a match is not found, the G-bit of the Commercial Processor indicator register is cleared to zero, the L-bit is set to one, and the receiving field is not changed.

The search argument list can contain one or more search arguments each consisting of one or more characters. If multiple arguments are specified, each must be the same length.

If the length field of DD1 is not equal to zero, the search argument list contains only one search argument whose length (1 to 31 bytes) is specified by the length field.

If the length field of DD1 is equal to zero, the search argument list is specified by register R4. The format of register R4 is shown below.

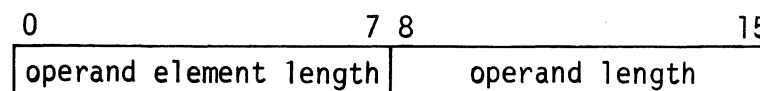


If the search argument length is equal to the search list length, the search list consists of only one argument.

If the ratio of the search list length to the search argument length is an integer, that integer designates the number of search arguments.

If the ratio of the search list length to the search argument length is not an integer, the ratio is truncated to the integer value and that integer designates the number of search arguments.

The character string (or array) to be searched is specified by DD3. If the length field of DD3 is not equal to zero, the operand is a character string whose length (1 through 31) is specified by the length field. If the length field is equal to zero, the operand to be searched is specified by register R6. The format of register R6 is shown below.



If the operand element length is one, the operand is a character string whose length (specified by the low order byte of R6) must be from 1 through 255. In this case the search argument length must be less than or equal to the operand length, otherwise a not found indication will result.

If the operand element length is not equal to one, it specifies the length of each element of an array. The length of the array is the largest multiple of the operand element length that is less than or equal to the operand length (specified by the low order byte of R6). In this case, if the search argument length is greater than the operand element length, each search will overflow into the next entry except when the last operand element is searched. Thus the last comparison will result in a mismatch.

Applicable Traps:

Trap 23 Reference to unavailable resource

Trap 24 Bus or memory error

Trap 26 Illegal character

Conditions causing trap:

- Length of search list is less than length of search argument
- Length of operand is less than length of operand element
- Length of search argument is zero
- Length of operand element is zero
- The length of one or more of the three operands is zero

The contents of the Commercial Processor indicator register are affected as follows:

- The G- and L-bits are set as described above.
- All other bits are unchanged.

Examples of Search Instruction:

The examples given below are divided into four categories as follows:

- Search String Single (i.e., search a string to determine if it contains the single search argument given in the search list)
- Search String Multiple (i.e., search a string to determine if it contains any one of the multiple search arguments given in the search list)
- Search Array Single (i.e., search an array to determine if it contains the single search argument given in the search list)
- Search Array Multiple (i.e., search an array to determine if it contains any one of the multiple search arguments given in the search list)

Example 1: Search String — Single Search Argument

The search list defined by DD1 contains one search argument of one or more characters.

If a match is found, the receiving field specified by DD2 is loaded with a zero (the search argument number) and the displacement. If a match is not found, DD2 is not changed.

Assume that DD3 defines the following string.

Displacement:	0	1	2	3	4	5	6	7	8	9	A	B	C
String:	a	b	c	d	e	f	g	h	i	j	d	e	k

The results of a search instruction for this string and various search arguments are as follows.

SA	<i>Commercial Processor</i>		<i>DD2 Field</i>	
	<i>Indicator Register</i>		<i>SA Number</i>	<i>Displacement</i>
	<i>L-Bit</i>	<i>G-Bit</i>		
d	0	0	0	3
f g	0	0	0	5

SRCH

f h	1	0		unchanged	
d e k	0	0	0		A
l m	1	0		unchanged	
a	0	0	0		0

Example 2: Search String — Multiple Search Arguments

The search list defined by DD1 contains multiple search arguments. Each search argument can consist of one or more characters but all search arguments must be the same length. The search argument length (SAL) and the search list length (SLL) is specified by register R4.

If a match is found, the search argument number and the displacement is loaded into the receiving field specified by DD2. If a match is not found, DD2 is not changed.

Assume that DD3 defines the following string,

Displacement:	0	1	2	3	4	5	6	7	8	9	A	B	C
String:	a	b	c	d	e	f	g	h	i	j	d	e	k

The results of a search instruction for this string and various search arguments are as follows.

<i>Commercial Processor</i>					<i>DD2 Field</i>	
<i>Indicator Register</i>					<i>SA Number</i>	<i>Displacement</i>
<i>SAL</i>	<i>SLL</i>	<i>SA</i>	<i>L-Bit</i>	<i>G-Bit</i>		
1	3	f,e,j	0	0	1	4
1	2	r,s	1	0	unchanged	
2	6	de,hi,er	0	0	0	3
3	6	cdf,hij	0	0	1	7
2	4	cb,ka	1	0	unchanged	

As an example of the sequence of comparisons that occur in seeking a match, consider the third search in the above list. This search specifies three search arguments of two characters; namely, de,hi,er. The comparisons are as follows:

```

ab de
ab he
ab er
bc de
bc hi
bc er
cd de
cd hi
cd er
de de Match of search argument 0 at displacement of 3

```

Example 3: Search Array — Single Search Argument

The search list defined by DD1 contains one search argument of one or more characters.

If a match is found, the receiving field specified by DD2 is loaded with a zero (the search argument number) and the displacement. If a match is not found, DD2 is not changed.

Assume that DD3 defines the following array for which R6 specifies the length of each element (OEL) as 4, and the length of the operand (OL) as 24.

<i>Displacement</i>	<i>String</i>
00	a b d f
04	a c b e
08	c a d e
0C	d e f g
10	m j o p
14	e a c b

The results of a search instruction for this array and various search arguments are as follows.

<i>Commercial Processor</i>			<i>DD2 Field</i>	
<i>Indicator Register</i>				
<i>SA</i>	<i>L-Bit</i>	<i>G-Bit</i>	<i>SA Number</i>	<i>Displacement</i>
ca	0	0	0	08
a	0	0	0	00
mjo	0	0	0	10
mjpo	1	0	unchanged	
acbec	0	0	0	04
eacha	1	0	unchanged	
bac	1	0	unchanged	
cade	0	0	0	08

Example 4: Search Array — Multiple Search Arguments

The search list defined by DD1 contains multiple search arguments. Each search argument can consist of one or more characters but all search arguments must be the same length. The search argument length (SAL) and the search list length (SLL) is specified by register R4.

If a match is found, the search argument number and the displacement are stored in the receiving field specified by DD2. If a match is not found, DD2 is not changed.

Assume that DD3 defines the following array for which register R6 specifies the length of each element (OEL) as 4 and the operand length (OL) as 24.

<i>Displacement</i>	<i>String</i>
00	a b d f
04	a c b e
08	c a d e
0C	d e f g
10	m j o p
14	e a c b

The results of a search instruction for this array and various search arguments are as follows.

<i>Commercial Processor</i>			<i>DD2 Field</i>			
<i>Indicator Register</i>						
<i>SAL</i>	<i>SLL</i>	<i>SA</i>	<i>L-Bit</i>	<i>G-Bit</i>	<i>SA Number</i>	<i>Displacement</i>
3	6	acb,acd	0	0	0	04
1	3	c,a,d	0	0	1	00
4	8	defg,abcd	0	0	0	0C
2	6	ad,ea,mj	0	0	2	10
3	9	aab,abb,eac	0	0	2	14
5	10	abdfb,mjope	0	0	1	10

VERFY

VERFY

Instruction

Alphanumeric verify

Type:

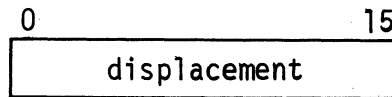
Character string

Source Language Format:

$$\Delta\text{VERFY}\Delta \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\} , \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\} , \left\{ \begin{array}{l} \text{DESCA}(\text{description}) \\ \text{int-val-expression} \end{array} \right\}$$

Description:

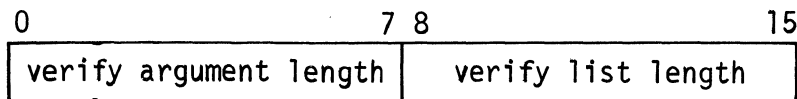
The character string or array of character strings defined by the third data descriptor (DD3) is examined. If at least one character of the string (or element of the array) does not match any one of the verify arguments, the G-bit of the Commercial Processor indicator register is cleared to zero, the L-bit is set to one, and the receiving field specified by the second data descriptor (DD2) is loaded with the displacement. The displacement is the distance in bytes between the origin of the string (or array) and the place where the first mismatch is found. The format of the receiving field is shown below.



If each of the characters of the string (or elements of the array) is equal to any one of the verify arguments, the G- and L-bits of the CIP indicator register are cleared to zero and the receiving field is not changed.

If the length field of DD1 is not equal to zero, the verify argument list contains only one search argument whose length (1 through 31 bytes) is specified by the length field.

If the length field of DD1 is equal to zero, the verify argument list is specified by register R4. The format of register R4 is shown below.

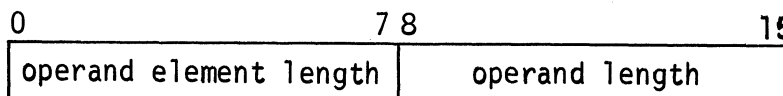


If the verify argument length is equal to the verify list length, the verify list consists of only one argument.

If the ratio of the verify list length to the verify argument length is an integer, that integer designates the number of verify arguments.

If the ratio of the verify list length to the verify argument length is not an integer, the ratio is truncated to the integer value and that integer designates the number of verify arguments.

The character string (or array) to be verified is specified by DD3. If the length field of DD3 is not equal to zero, the operand is a character string whose length (1 through 31) is specified by the length field. If the length field is equal to zero, the operand to be searched is specified by register R6. The format of register R6 is shown below.



If the operand element length is one, the operand is a character string whose length (specified by the low order byte of R6) must be from 1 through 255. If the operand is a character string, the length of the verify argument must be one otherwise the results are unspecified.

If the operand element length is not equal to one, it specifies the length of each element of an array. The length of the array is the largest multiple of the operand element length that is less than or equal to the operand length (specified by the low order byte of R6). In this case, if the verify argument length is greater than the operand element length, each verify will overflow into the next entry except when the last operand is verified. Thus the last comparison will result in a mismatch.

Applicable Traps:

- Trap 23 Reference to unavailable resource
- Trap 24 Bus or memory error
- Trap 26 Illegal Character

Conditions causing trap:

- Length of verify list is less than length of verify argument
- Length of operand is less than length of operand element
- Length of verify argument is zero
- Length of operand element is zero
- The length of one or more of the three operands is zero

The contents of the Commercial Processor indicator register are affected as follows:

- The L- and G-bits are set as described above.
- All other bits are unchanged.

Examples of Verify Instruction:

The examples given below are divided into four categories as follows.

- Verify String Single (i.e., verify a string to determine if its characters are all equal to the single verify argument given in the verify list)
- Verify String Multiple (i.e., verify a string to determine if its characters are each equal to one of the multiple verify arguments given in the verify list)
- Verify Array Single (i.e., verify an array to determine if its elements are all equal to the single verify argument given in the verify list)
- Verify Array Multiple (i.e., verify an array to determine if its elements are each equal to one of the multiple verify arguments given in the verify list)

Example 1: Verify String — Single Verify Argument

The verify list defined by DD1 contains one verify argument consisting of a single character.

The receiving field specified by DD2 is loaded with the displacement of the first character (in the string being verified) that is not the same as that of the verify argument. If all characters are the same, the receiving field is not changed.

Assume that DD3 defines the following strings. The results of verify instructions for the specified verify arguments are as follows.

Displacement:	0	1	2	3	4	5	6
First string:	a	a	a	b	a	a	c
VA	<i>CIP Indicator Register</i>						
	<i>L-Bit</i>	<i>G-Bit</i>	<i>Displacement</i>				
a	1	0	3				
b	1	0	0				

VERFY

Displacement:	0	1	2	3	4	5	6	
Second string:	b	b	b	b	b	b	b	
b	0		0					unchanged

Example 2: Verify String — Multiple Verify Arguments

The verify list defined by DD1 contains multiple verify arguments of one character each. The verify argument length (VAL) and the verify list length (VLL) is specified by register R4.

The receiving field specified by DD2 is loaded with the displacement of the first character in the string being verified that is not the same as any of the verify arguments. If there is no character that is different, the receiving field is not changed.

Assume that DD3 defines the following string.

Displacement:	0	1	2	3	4	5	6	7	8	9
String:	a	b	c	b	b	a	d	b	c	c

The results of the verify instructions with this string and various verify arguments are as follows.

<i>Commercial Processor</i>			
<i>Indicator Register</i>			
VA	L-Bit	G-Bit	Displacement
a,b	1	0	2
a,b,c,e	1	0	6
a,b,c,d	0	0	unchanged

Example 3: Verify Array — Single Verify Argument

The verify list defined by DD1 contains one verify argument consisting of one or more characters.

The receiving field defined by DD2 is loaded with the displacement of the first element of the array that does not contain the verify argument. If all elements of the array contain the verify argument, the receiving field is not changed.

Assume that DD3 defines the following array for which register R6 specifies the length of each element as 3 and the length of the operand as 12.

<i>Displacement</i>	<i>String</i>
0	a b a
3	a b c
6	a b d
9	a c b

The results of the verify instruction for this array and various search arguments are as follows.

<i>Commercial Processor</i>			
<i>Indicator Register</i>			
VA	L-Bit	G-Bit	Displacement
ab	1	0	9
abc	1	0	0
a	0	0	unchanged
abaa	1	0	3

Example 4. Verify Array — Multiple Search Arguments

The verify list defined by DD1 contains multiple search arguments. Each verify argument can consist of one or more characters, but all search arguments for a given instruction must be the same length. The verify argument length and the verify list length are specified by register R4.

The receiving field specified by DD2 is loaded with the displacement of the first element of the array that does not contain any of the verify arguments. If all elements of the array contain at least one of the verify arguments, the receiving field is not changed.

Assume that DD3 defines the following array for which register R6 specifies the length of each element as 3 and the length of the operand as 12.

<i>Displacement</i>	<i>String</i>
0	a b c d
4	a c d b
8	b c a d
C	a c b d

The results of the verify instruction for this array and various verify arguments are as follows.

<i>VA</i>	<i>Commercial Processor Indicator Register</i>		<i>Displacement</i>
	<i>L-Bit</i>	<i>G-Bit</i>	
ab,ac	1	0	8
ab,ac,bc	0	0	unchanged
abc,acd,acb	1	0	8
abcd,acbd	1	0	4



Section 7

Scientific Instructions

The scientific instructions are executed by the Scientific Instruction Processor (SIP), an optional hardware item, or by the SIP Simulator, a software item that provides the same functionality.

The SIP operates on a powerful set of scientific instructions that are particularly useful for FORTRAN applications. The instruction set includes arithmetic operations on single- and double-precision floating-point operands, and on single- and double-word integer operands.

The SIP accepts and processes only one command at a time. However, following the extraction of most scientific instructions from memory and the transfer of the instruction to the SIP, the CPU can proceed with the extraction of the next instruction while the scientific instruction is being executed by the SIP. Note that the execution of scientific instructions may involve the reading or writing of main memory. Software that makes use of the SIP should avoid premature reading or writing of memory locations specified by an SIP instruction.

The SIP includes three variable accumulators which may contain floating-point values of two or four words. Associated with each accumulator are control bits that specify the accumulator length and the length of the memory operand directed to a given accumulator. (See *SIP Registers* in Section 1.)

In the SIP, all operands are stored in floating-point format. Operands directed to the SIP from main memory are in floating-point format; operands from the CPU are in integer format, but are converted to floating-point values before they are entered into scientific calculations.

SCIENTIFIC TRAPS

The SIP trap facility monitors the execution of all SIP instructions and sends trap requests to the CPU whenever certain conditions occur. These traps and the conditions that cause them are shown in Table 7-1. Trap 23, reference to unavailable resource, and Trap 24, bus parity or uncorrected main memory error, are functions of the megabus. Trap 20, program error, identifies program errors detected by the SIP. Note that program errors detected by the CPU activate Trap 16. If more than one trap condition exists, the SIP sends the trap with the highest priority to the CPU. The other conditions are lost. The priority of the traps is indicated in Table 5-1 by their location; the trap at the top (Trap 31) has the highest priority.

TABLE 7-1. TRAP VECTORS AND EVENTS

Trap Vector Number	Trap Event	C/U ^a	Scientific Indicator	Scientific Mask	SIP-CPU Trap Code
TV31	QLT Failure (Mini or Maxi)	u	—	—	21
TV23	Reference to Unavailable Resources	u	—	—	29
TV24	Megabus/Memory Error	u	—	—	28
TV20	Program Error (SIP)	u	—	—	2C
TV7	Divide by Zero	u	—	—	39
TV8	Exponent Overflow	u	—	—	38
TV21	Significance Error	c	SI(SE)	M5(SE)	2B
TV19	Exponent Underflow	c	SI(EUF)	M5(EUF)	2D
TV22	Precision Error	c	SI(PE)	M5(PE)	2A
TV00	No Trap Event Set	—	—	—	00

^aC — Conditioned Trap on Indicator and Mask
 U — Unconditioned Trap

SAD

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:

If the operand 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:

SAD	SML
SCM	SNGD
SCZD	SNGQ
SCZQ	SSB
SDV	SST
SLD	SSW

The instruction:

SST \$S1, = \$S1

can be used to force the SIP into synchronization with the CPU.

DETAILED DESCRIPTIONS OF SCIENTIFIC INSTRUCTIONS

The remainder of this section contains detailed descriptions of the scientific instructions. The descriptions are arranged in alphabetical sequence by instruction mnemonics. Each description includes the name, type, format, and operands. Symbolic names, constants and expressions are described in Section 2.

SAD

Instruction:

Scientific add

Type:

DO

Source Language Format:

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

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.

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:

$$=\$R \left\{ \begin{array}{l} 4 \\ 5 \\ 6 \\ 7 \end{array} \right\} \quad \text{If } =\$R7 \text{ is specified, the 32-bit value contained in the register pair formed by R6 and R7 becomes the operand.}$$

=\$Sn

If immediate operand addressing is used, you must provide a floating-point constant or 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 contents of 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.

If the SIP is not installed, the SIP simulator, if present, is entered via trap vector 3.

SBE

SBE

Instruction:

Scientific branch on equal

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if both the SL- and SG-bits of the SI-register are 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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBEU

Instruction:

Scientific branch on exponent underflow

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the EU-bit in the SI-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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBEZ

SBEZ

Instruction:

Branch if scientific accumulator equal to 0

Type:

BR

Source Language Format:

$$\Delta\text{SBEZ}\Delta \begin{pmatrix} \$S_n \\ X'n' \\ n \end{pmatrix}, \begin{pmatrix} \text{direct-IMA} \\ \text{direct-P-relative-address} \\ \text{short-displacement-address} \end{pmatrix}$$

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, if present, is entered via trap vector 5.

SBG

Instruction:

Scientific branch on greater than

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the SG-bit in the SI-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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBGE

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 SL-bit of the SI-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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBGEZ

Instruction:

Branch if scientific accumulator equal to or greater than 0.

Type:

BR

Source Language Format:

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

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, if present, is entered via trap vector 5.

SBGZ

SBGZ

Instruction:

Branch if scientific accumulator greater than 0

Type:

BR

Source Language Format:

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

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 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, if present, is entered via trap vector 5.

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 SL-bit of the S1-register is set 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, if present, is entered via trap vector 5.

SBLE

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 SG-bit in the SI-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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBLEZ

Instruction:

Branch if scientific accumulator equal to or less than 0

Type:

BR

Source Language Format:

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

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, if present, is entered via trap vector 5.

SBLZ

SBLZ

Instruction:

Branch if scientific accumulator less than 0

Type:

BR

Source Language Format:

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

Description:

Branches to the location specified in the second operand if the scientific accumulator identified in the first operand contains a 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, if present, is entered via trap vector 5.

SBNE

Instruction:

Scientific branch on not equal

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if either the SL- or SG-bit of the SI-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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBNEU

SBNEU

Instruction:

Scientific branch on not exponent underflow

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the EU-bit of the SI-register is set 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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBNEZ

Instruction:

Branch if scientific accumulator not equal to 0

Type:

BR

Source Language Format:

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

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, if present, is entered via trap vector 5.

SBNPE

SBNPE

Instruction:

Scientific branch on not precision error

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the PE-bit of the SI-register is set 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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBNSE

Instruction:

Scientific branch on not significance error

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the SE-bit of the SI-register is set to 0.

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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBPE

SBPE

Instruction:

Scientific branch on precision error

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the PE-bit of the SI-register is set to 1.

Action if Branch Occurs:

If the J-bit of the 1-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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SBSE

Instruction:

Scientific branch on significance error

Type:

BI

Source Language Format:

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

Description:

Branches to the location specified in the operand if the SE-bit of the SI-register is set 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.

If the Scientific Information Processor (SIP) is not installed on this system, the Scientific Branch Simulator, if present, is entered via trap vector 5.

SCM

SCM

Instruction:

Scientific compare

Type:

DO

Source Language Format:

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

Description:

Compares the contents of the scientific accumulator identified in the first operand to the floating-point or integer value in the location, scientific accumulator, or R-register specified in the second operand.

Scientific Indicator Settings:

SG: Set to 1 if contents of the scientific accumulator identified by the first operand are greater than the value specified by the second operand location; otherwise, set to 0.

SL: Set to 1 if contents of the scientific accumulator identified by the first operand are less than the value specified by the second operand 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, if present, to be 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:

- = B_n register addressing
- Short displacement addressing
- Specialized addressing

If register addressing is used, the valid forms are:

$$=\$R \left\{ \begin{array}{l} 4 \\ 5 \\ 6 \\ 7 \end{array} \right\} \quad \text{If } =\$R7 \text{ is specified, the 32-bit value contained in the} \\ \text{register pair formed by R6 and R7 becomes the operand.}$$

= S_n

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 = R_4 , = R_5 , = R_6 , or = R_7 , the integer value contained in the specified 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:

Δ SCZD Δ address-expression

Description:

Compares the short-precision floating-point value in the specified location to 0.

Scientific Indicator Settings:

SG: Set to 1 if the contents of the location are greater than 0; otherwise, set to 0.

SL: Set to 1 if the 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, if present, to be 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:

=\$Bn	}	register addressing
=\$Rn		
=\$Sn		

Short displacement addressing
Specialized addressing

SCZQ

SCZQ

Instruction:

Scientific compare to 0 (long-precision)

Type:

SO

Source Language Format:

Δ SNZQ Δ address-expression

Description:

Compares the floating-point value in the specified location to 0.

Scientific Indicator Settings:

SG: Set to 1 if the contents of the location are greater than 0; otherwise, set to 0.

SL: Set to 1 if the 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:

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

Short displacement addressing

Specialized addressing

If immediate operand addressing is used, you must provide a string constant in suitable floating-point format.

If the Scientific Information Processor (SIP) is not installed on this system, the Floating-Point Simulator, if present, is entered via trap vector 3.

SDV

Instruction:

Scientific divide

Type:

DO

Source Language Format:

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

Description:

Divides the contents of the scientific accumulator identified by 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 identified by the first operand (except for the remainder, which is ignored).

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator, if present, 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:

- = \$Bn register addressing
- Short displacement addressing
- Specialized addressing

If register addressing is used, the valid forms are:

$$= \$R \left\{ \begin{array}{l} 4 \\ 5 \\ 6 \\ 7 \end{array} \right\} \begin{array}{l} \text{If } = \$R7 \text{ is specified, the 32-bit value contained in the} \\ \text{register pair formed by R6 and R7 becomes the operand.} \end{array}$$

$$= \$S_n$$

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 contents of the S-register specified by the first operand.

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 added to the contents of 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

SLD

Instruction:

Scientific load

Type:

DO

Source Language Format:

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

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, if present, 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:

- = B_n register addressing
- Short displacement addressing
- Specialized addressing

If register addressing is used, the valid forms are:

$$=\$R \left\{ \begin{array}{l} 4 \\ 5 \\ 6 \\ 7 \end{array} \right\} \quad \text{If } =\$R7 \text{ is specified, the 32-bit value contained in the}$$

register pair formed by R6 and R7 becomes the operand.

$$=\$S_n$$

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 = R_4 , = R_5 , = R_6 , or = R_7 , the integer value contained in the specific R-register is internally converted to floating-point format before it is added 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:

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

Description:

Multiplies the contents of the scientific accumulator identified by 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 identified by the first operand.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator, if present, 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:

- = \$Bn register addressing
- Short displacement addressing
- Specialized addressing

If register addressing is used, the valid forms are:

$$= \$R \left\{ \begin{array}{l} 4 \\ 5 \\ 6 \\ 7 \end{array} \right\} \text{ If } = \$R7 \text{ is specified, the 32-bit value contained in the}$$

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 an R-register the integer value contained in the specific R-register is internally converted to floating-point format before it is multiplied by the contents of 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

SNGD

Instruction:

Scientific negate (short-precision)

Type:

SO

Source Language Format:

Δ SNGD Δ address-expression

Description:

Negate the short precision floating-point number at the location or in the scientific accumulator specified by the operand.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator, if present, 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:

$=\$Sn$

SNGQ**Instruction:**

Scientific negate (long-precision)

Type:

SO

Source Language Format:

Δ SCGQ Δ address-expression

Description:

Negate the long-precision floating-point number at the location or in the scientific accumulator specified by the operand.

If the SIP is not installed on this system, the Floating-Point Simulator, if present, 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:

$=\$Sn$

SSB

SSB

Instruction:

Scientific subtract

Type:

DO

Source Language Format:

$$\Delta\text{SSB}\Delta \begin{Bmatrix} \$S_n \\ X'n' \\ n \end{Bmatrix}, \text{address-expression}$$

Description:

Subtracts the contents of the location, scientific accumulator, or R-register identified by the second operand from the contents of the scientific accumulator specified by 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, if present, 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:

- = \$Bn register addressing
- Short displacement addressing
- Specialized addressing

If register addressing is used, the valid forms are:

$$=\$R \begin{Bmatrix} 4 \\ 5 \\ 6 \\ 7 \end{Bmatrix} \left\{ \begin{array}{l} \text{If } =\$R7 \text{ is specified, the 32-bit value contained in the} \\ \text{register pair formed by R6 and R7 becomes the operand.} \end{array} \right.$$

= \$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 contents of 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:

$$\Delta S S T \Delta \left\{ \begin{array}{l} \$S_n \\ X'n' \\ n \end{array} \right\}, \text{address-expression}$$

Description:

Stores the contents of the scientific accumulator identified by the first operand in the location, scientific accumulator, or R-register specified by the address expression.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator, if present, 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:

- = \$Bn register addressing
- Short displacement addressing
- Specialized addressing

If register addressing is used, the valid forms are:

$$= \$R \left\{ \begin{array}{l} 4 \\ 5 \\ 6 \\ 7 \end{array} \right\} \begin{array}{l} \text{If } = \$R7 \text{ is specified, the 32-bit value contained in the} \\ \text{register pair formed by R6 and R7 becomes the operand.} \end{array}$$

= \$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

SSW

Instruction:

Scientific swap

Type:

DO

Source Language Format:

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

Description:

Swaps the contents of the scientific accumulator identified by the first operand with the contents of the location, scientific accumulator, or R-register specified by the address expression.

If the Scientific Instruction Processor (SIP) is not installed on this system, the Floating-Point Simulator, if present, 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:

- = \$Bn register addressing
- Short displacement addressing
- Specialized addressing

If register addressing is used, the valid forms are:

$$= \$R \left\{ \begin{array}{l} 4 \\ 5 \\ 6 \\ 7 \end{array} \right\} \begin{array}{l} \text{If } = \$R7 \text{ is specified, the 32-bit value contained in the} \\ \text{register pair formed by R6 and R7 becomes the operand.} \end{array}$$

= \$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.

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

- = \$Rn } register addressing
- = \$Sn }
- Short displacement addressing
- Specialized addressing

*

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.

Section 8

Macro Facility

The Macro Preprocessor is a program development tool that provides a convenient method for including in a source module 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 program. A single statement, known as a macro call, is specified in the source program each time the sequence of statements is to be included. A source program containing one or more macro calls is called an unexpanded source program. Macro routines can be at the beginning of a source program or in a macro library; those occurring with a source program are called inline macro routines.

The Macro Preprocessor produces an expanded source program which is used as input to the Assembler. The expanded source program may contain an error flag for each nonfatal error. Each statement that contains a nonfatal error flag appears in the expanded source module as a comment statement with the appropriate error. (Nonfatal error flags are described in Appendix F.) If a fatal error occurs, processing terminates, an error message is issued to the error-out stream, and control returns to the Command Processor. (Error messages issued by the Macro Preprocessor are described in the *System Messages* manual.)

NOTE:

Honeywell provides a library of macro routines that support MLCP programming.
(See the *MLCP Programmer's Reference Manual*.)

ORDER OF STATEMENTS WITHIN A SOURCE PROGRAM

Statements within a source program 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
(Optional) comment statements

Note:

LIBM statements, macro routines, comment statements, 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. Identifies the end of the assembly language program. Statements subsequent to this statement will be ignored by the Assembler. If this statement is missing, both the Assembler and the Macro Preprocessor will generate an END 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.

MAC WITHOUT PARAMETERS / ENDM

The following information is described below.

- Creating a macro routine
- Specializing a macro routine
- Including protection operators
- 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:

- Macro control statements, excluding MAC and ENDM
- Macro functions
- Assembler control statements, excluding END
- 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 through 7, respectively.

ENDM / MAC WITH PARAMETERS

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 85 different macro parameters can be referenced. Parameters are named P1 to P9 and PA to PZ. (The lowercase letters are considered to be equivalent to the corresponding uppercase letters.) 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 the value of the parameter. If no value was specified in the MAC statement or in the macro call, the parameter is equal to a null character string.

MAC WITH PARAMETERS

MAC MACRO CONTROL STATEMENT, INCLUDING PARAMETERS

The MAC statement assigns a name to a macro routine and optionally assigns default value macro parameters.

Format:

macro-name ΔMAC ΔP_j[=v], P_k[=v] ...

macro-name

Name of the macro routine being created; must be a valid symbolic name.

P_n

Macro parameter name; can be from the set P1 through P9 or from the set PA through 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. (See "Designating Alphanumeric Values" at the end of this section.)

*

If a value is not specified, the corresponding parameter remains equal to a null 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,'	LDV \$R1=?P3	assigns values to parameters P3 and PB
	LDR \$R2=?PB	Statements to be included in source module
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,'	
	.	
	.	
	.	

MAC WITH PARAMETERS

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 program.

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 interpreted 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 protection operators
.	
.	
SAMPLE	Macro call; in the expanded source module will be replaced by contents of macro routine 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 designs 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]

MAC WITH PARAMETERS

If any part of a label or operation code is protected, the entire label or operation code is protected.

Example:

```
LAB[EL]ΔLD[R]Δ$R1,=100
```

The above statement is considered to have no label and no operation code.

Protection operators do not appear in expanded source modules unless the operators are embedded in other protection operators.

Example 1:

```
NEWA[?]P7
```

The above statement appears in the expanded source module as NEWA?P7.

Example 2:

```
DC 'A[BC[DEF]GH]I'
```

The above statement appears in the expanded source module as DCA'ABC[DEF]GHI'. Only the outermost protection operators are removed, unless the expanded source module is then reprocessed by the Macro Preprocessor.

Protected comment statements appear in the expanded source module with the protection operators removed. If protected comment statements appear in a macro routine, they are substituted in the expanded source module as described previously. Unprotected comment statements which appear in a macro routine are considered to document the macro routine itself; thus they are not substituted into, the expanded source module.

Example:

```
ABC MAC  
  HLT  
*COMNT1  
[*]COMNT2  
  ENDM
```

In the above example COMNT2 is considered a macro routine comment and will appear in the expanded source module as

```
*COMNT2
```

COMNT1 is not considered a macro routine comment and will not appear in the expanded source module.

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 mass storage volume. A macro library is a directory whose files are macro routines. Each file must be a single macro routine that is referenced in a macro call by its file name. Its file name must be identical to the label of its MAC statement.

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.)

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 program.

Format:

$$\Delta\text{LIBM}\Delta \left\{ \begin{array}{l} \text{symbolic-name} \\ [\text{A}]'c[\text{c...}]' \end{array} \right\} [\text{,macro-name}]...$$

symbolic-name

Name of a macro library. A macro library is a directory containing a collection of files. Each file in a macro library consists of a single macro routine. The following search rules are used to locate the macro library referenced by a symbolic-name.

1. The current working directory is searched for a subdirectory of the name, symbolic-name. If this directory is found, it is the directory used as the macro library.
2. The directory >UDD>account>MACRO, if it exists, is searched for a subdirectory of the name, symbolic-name. If this directory is found, it is the directory used as the macro library.
3. The directory >LDD>MACRO, if it exists, is searched for a subdirectory of the name, symbolic-name. If this directory is found, it is the directory used as the macro library. The volume(s) containing the system directories SYSLIB1 and SYSLIB2 are searched to find the >LDD>MACRO directory. It may require the Change System Directory command to identify the pathname of SYSLIB1 and/or SYSLIB2.

[A]'c[...]

Name of a macro library in the form of an ASCII string constant. If the value of the string, constant is a simple name, it is processed as a symbolic-name.

If the value of the string constant is not a simple name, it is expanded to an absolute pathname, which is assumed to be the name of the desired macro library.

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 else 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.

Example of Search Rules for Locating a Macro Library

This example of the search rules is based on the following assumptions:

- The current working directory is ^ VOLUSR>CURRENT
- Directory SYSLIB1 is contained on volume VOL1
- Directory SYSLIB2 is contained on volume VOL2
- The system root volume is volume ZSYS51
- The account of the task group using the macro preprocessor is PROJL6

Given these assumptions, the statement

```
LIBM EXEC_LIB
```

initiates a search for the following *directories* in the order indicated.

LIBM

Search Rule Number 1

1. Search is for: ^VOLUSR>CURRENT>EXEC_LIB

Search Rule Number 2

2. Search is for: ^ZSYS51>UDD>PROJL6>MACRO>EXEC_LIB

Search Rules Number 3

3. Search is for: ^VOL1>LDD>MACRO>EXEC_LIB

4. Search is for: ^VOL2>LDD>MACRO>EXEC_LIB

INCLUDE

INCLUDE MACRO CONTROL STATEMENT

The **INCLUDE** statement specifies that the contents of the named file be processed by the Macro Preprocessor as part of the unexpanded source input program.

The include file may contain any source statements normally appearing within the unexpanded source input program, with the following exceptions:

1. The include file may not contain any **%INCLUDE** macro control statements.
2. The include file may contain macro call statements only if they are within inline macro routines.

Format:

$$[\Delta]\%INCLUDE\Delta \left\{ \begin{array}{l} \text{symbolic-name} \\ [A]'c[...]' \end{array} \right\};$$

symbolic-name

Name of a file. The following search rules are employed to locate the include file referenced by the symbolic-name:

1. The current working directory is searched for a file of the name, symbolic-name.IN.A. If this file is found, that file is the file referenced in the **INCLUDE** statement.
2. The directory **>UDD>account>INCLUDE**, if it exists, is searched for the file symbolic-name.IN.A. If this file is found, this is the file referenced in the **INCLUDE** statement.
3. The directory **>LDD>INCLUDE**, if it exists, is searched for the file symbolic-name.IN.A. If this file is found, this is the file referenced in the **INCLUDE** statement. The volume(s) containing the system directories **SYSLIB1** and **SYSLIB2** are searched to find the **>LDD>MACRO** directory. It may require the **Change System Directory** command to identify the pathname of **SYSLIB1** and/or **SYSLIB2**.

[A]'c[...]'

Name of a file in the form of an ASCII string constant. If the value of the string constant is a simple name, then it is processed as a symbolic-name.

If the value of the string constant is not a simple name, **\Delta.IN.A** is appended to the string; then it is expanded to an absolute pathname, which is assumed to be the name of the desired include file.

The semicolon shown in the above format is part of the **INCLUDE** statement's syntax, it does not indicate.

The **%INCLUDE** statement is also used to insert macro routines from external files into the unexpanded source program as inline macro routines.

Example of Search Rules for Locating an Include File

This example of the search rules is based on the following assumptions.

- The current working directory is **^VOLUSR>CURRENT**
- Directory **SYSLIB1** is contained on volume **VOL1**
- Directory **SYSLIB2** is contained on volume **VOL2**
- The system root volume is volume **ZSYS51**
- The account of the task group using the macro preprocessor is **PROJL6**

Given these assumptions, the statement

```
%INCLUDE SOURCES;
```

initiates a search for the following *files* within the file system in the order indicated.

INCLUDE

Search Rule Number 1

1. Search is for: ^VOLUSR>CURRENT>SOURCES.IN.A

Search Rule Number 2

2. Search is for: ^ZSYS51>UDD>PROJL6>INCLUDE>SOURCES.IN.A

Search Rule Number 3

3. Search is for: ^VOL1>LDD>INCLUDE>SOURCES.IN.A
4. Search is for: ^VOL2>LDD>INCLUDE>SOURCES.IN.A

MACRO CALLS

A macro call is a statement that causes a specified macro routine to be included in the source program and optionally assigns or reassigns values to parameters in that macro routine. The macro routine is included in the expanded source program 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 character string. (See "Initialized Values of Macro Variables" at the end of this section.)

If no parameter values are present 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:

$$[\text{label}]\Delta\text{macro-name} \left[\Delta[\text{P}_1\text{-value}] \left[[\text{P}_2\text{-value}] \dots \right] \right]$$

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 (see "Designating Alphanumeric Values" at the end of this section).

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 program 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 program illustrates the inclusion of the macro routine and the appropriate parameter values. This example also illustrates the use of a parameter whose value is a null character string.

*

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
*      NEWB 5,'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 and the macro routine named in the nested call are 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

```
MACRO1  TITLE NSTD
        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      Contents of nested macro call
        NEWB
        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. Each time an ENDM statement is processed, the innermost level of recursion is terminated. 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 program 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 a macro routine. A global variable can be assigned a value anywhere in the source module; e.g., in a 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. (The lowercase letters are considered to be equivalent to the corresponding uppercase letters.) To designate in a macro routine that substitution of a macro variable 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.

SETA

A variable can be assigned an alphanumeric or numeric value by specifying the SETA or SETN macro control statement, respectively.

If a variable is never assigned a value, its initial value is used as the default. (See "Initialized Values of Macro Variables" at the end of this section.)

The values assigned to global variables are available for use by all macro routines. The values assigned to local variables are available for use only by the macro routine which assigned them. In the case of nested or recursive macro calls, each time a macro routine is called, it has a new set of local variables to use. When the nested or recursive macro routine is complete, the previous values assigned by the outer macro are restored to the local variables.

MACRO SUBSTITUTION

Macro substitution allows a character string contained in the macro prototype to be replaced by another character string that is derived from the contents of the macro parameters and/or macro variables. By this substitution, a generalized macro prototype can be specialized to produce different expansions depending on the arguments present in the various calls.

Substitution of macro parameters and variables, and evaluation and substitution of macro functions are the first operations performed upon a macro prototype during a macro expansion. Thus, macro control statements can be generalized in the same manner as any other statement in the macro prototype.

The substitution operator, "?", is used in the macro prototype to indicate where substitutions are to occur when the macro is expanded. The substitution operator must be followed by the name of a macro parameter, the name of a macro variable, or a macro function reference.

When the macro processor encounters a substitution operator followed by a macro parameter name or a macro variable name, the substitution operator and the name following are replaced by the value of the referenced parameter or variable. When a substitution operator followed by a macro function reference is encountered, the function is evaluated and then the substitution operator and function reference following it are replaced by the value returned by the function.

After substitution is completed, the replacement value is rescanned.

The occurrence of a function reference may be represented in whole or in part by a "nested" substitution operation. In this case the inner substitution operation is performed before the outer substitution operation. Substitution operations may be nested to any depth.

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:

variableΔSETAΔvalue

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 (The lower case letters are considered to be equivalent to the corresponding uppercase letters.).

value

Must be alphanumeric. (See "Designating Alphanumeric Values" at the end of this section.)

Example:

This example illustrates an unexpanded source program in which macro variables are assigned values in SETA statements. The resulting expanded source program includes those macro variable values.

Unexpanded source program:

```

SETA EXAMPLE
      TITLE VALUE
EXAMPLE MAC      Designates beginning of macro routine
L4   SETA DE      Assigns alphanumeric value to L4
L5   SETA "'X'"    Assigns alphanumeric value to L5
L4   SETA (5+6*2) Assigns different alphanumeric value to L4
G6   SETA ','      Assigns alphanumeric value to G6
      .
      .
      DC  NOT?L4
      TEXT ?L5
      DC  1?G62?G63
      .
      .
      ENDM          Designates end of macro routine
      SAMPLE       Macro Call
    
```

Expanded Source Module:

```

      TITLE VALUE
      .
      .
      DC  NOT (5+6*2)Least value for L4 is used; note that numeric
           expression is not evaluated
      TEXT 'X'      Embedded double apostrophes become single apostrophes
      DC  1,2,3     Comma inside quote strings are data
      .
      .
    
```

²When 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.

SETN

SETN MACRO CONTROL STATEMENT

The set numeric macro control (SETN) statement assigns a numeric value to a local or global macro variable. The assigned value is the ASCII representation of the decimal equivalent of the specified numeric value. 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 program. When assigning a value to a *local* macro variable, you must specify SETN in a macro routine.

Format:

variableΔSETNΔvalue

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 (The lowercase letters are considered to be equivalent to the corresponding uppercase letters.)

value

Must be numeric. (See "Designating Numeric Values" at the end of this section.)
Corresponds to internal value expression which is defined in Section 2.

The operand of the SETN statement begins at the first character after the operation code that is neither a blank nor a horizontal tab. 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:

GLΔSETNΔ22+8 assigns the value 30 in unpacked decimal representation (3330 in hexadecimal) to the global variable GL.
GAΔSETNΔ6+'Δ0' assigns the value Δ6 (2036 in hexadecimal) to the global variable GA.
G6ΔSETNΔ-X'F' assigns the value -15 (2D3135 in hexadecimal) to the global variable G6

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 numeric value to L5
L6      SETN 2*(?L5*?G2)+1 Assigns numeric value to L6; expression is evaluated
        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
.
.
.
```

³When 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.

CONDITIONAL MACRO CONTROL STATEMENTS

These statements allow the subsequent processing to be varied according to the conditions that exist when the statement is executed.

Conditional macro control statements are listed and described below:

- FAIL
- GOTO
- IF
- 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

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:

$$[\text{label}]\Delta\text{GOTO}\Delta \left\{ \begin{array}{l} * \\ \text{skip-label} \end{array} \right\}$$

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 within the macro routine at which Macro Preprocessor should resume processing. If a macro routine contains more than one statement whose symbolic name is skip-label, processing resumes at the first occurrence of such a statement after the MAC statement.

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 statement is processed.

Format 1.

Evaluating characters in one numeric operand:

$$[\text{label}]\Delta\text{IF} \left\{ \begin{array}{l} [\text{N}] \left\{ \begin{array}{l} \text{P} \\ \text{N} \\ \text{Z} \end{array} \right\} \\ \text{OD} \\ \text{EV} \end{array} \right\} \Delta\text{operand}, \left\{ \begin{array}{l} * \\ \text{skip-label} \end{array} \right\}$$

label

Symbolic name that identifies the IF statement.

[N]P

(Not positive (i.e., positive is > 0, not positive is ≤ 0))

[N]N

(Not negative (i.e., negative is > 0, not negative is ≥ 0))

[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.) Corresponds to internal value expression, which is defined in Section 2.

*

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. If a macro routine contains more than one statement subsequent to the IF statement, whose symbolic name is skip-label, processing resumes at the first occurrence of such a statement after the IF statement.

Format 2.

Comparing characters in two alphanumeric operands:

$$[\text{label}]\Delta\text{IF} [\text{N}] \left\{ \begin{array}{l} \text{G} \\ \text{L} \\ \text{E} \end{array} \right\} \Delta\text{operand}_1, \text{operand}_2, \left\{ \begin{array}{l} * \\ \text{skip-label} \end{array} \right\}$$

label

Symbolic name that identifies the IF statement.

IF

[N]G

(Not) greater than.

[N]L

(Not) less than.

[N]E

(Not) equal to.

operand₁, operand₂

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 operand₁ to the character in the corresponding position in operand₂. 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.)

The unequal characters are compared according to the ASCII collating sequence to determine the algebraic relationship between the two operands.

*

If the condition specified in the 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 is true, designates symbolic name of statement at which Macro Preprocessor should resume processing.

If a macro routine contains more than one statement subsequent to the IF statement whose symbolic name is skip-label, processing resumes at the first occurrence of such a statement after the IF statement.

NOTE:

If an assembly control IF statement is desired within a macro routine, it must be protected.

Example 1. Evaluating characters in one numeric operand:

Unexpanded Source Module:

```
TITLE CONDL
G5      SETN 1
BGN     MAC
        IFOD?G5, TAG1  Conditionalize the macro expansion via value of G5
        [FAIL]
TAG1    DC 1
        .
        .
        .
        IFEV ?G5, TAG2  Conditionalize the macro expansion via value of G5
        [FAIL]
TAG2    DC 1
        .
        .
        .
FINI    ENDM
        .
        .
        .
        BGN             Macro call
        .
        .
        .
```

Expanded Source Module:

```

        TITLE CONDL
        .
TAG1    DC 1          FAIL statement is not produced
        .
        .
TAG2    FAIL DC 1    FAIL statement for Assembler is produced
        .
        .
    
```

Example 2. Comparing characters in two alphanumeric operands:

Unexpanded Source Module:

```

        TITLE TWO
INCL   MAC
        .
        .
TAG1   IFE AB,AB,TAG1  Conditionalize the macro expansion
        [FAIL]
        DC 1
        .
        .
TAG1   IFE AB,CD,TAG1
        [FAIL]
        DC 1
        .
        .
FINI   ENDM
        .
        .
        INCL
        .
        .
    
```

Expanded Source Module:

```

        TITLE TWO
        .
        .
        .          FAIL statement is not produced
TAG1    DC 1
        .
        .
TAG1    FAIL DC 1    FAIL statement for Assembler is produced
        .
        .
        .
    
```

NULL

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:

- Determine number of characters that are in a specified character string (AL function)
- Convert a numeric value to its hexadecimal equivalent (CH function)
- Search a character string for an embedded character string (IX function)
- Determine which character within a character string is the first character that is the first character of another character string (SR function)
- Specify which characters within a character string should be included in the source statement (SS function)
- Permit parameters and variables to be referenced by their ordinal positions (V function)
- Determine which character within a character string is the first character that is *not* in another character string (VR function)
- Determine what type of constant makes up a character string (AT)
- Translate a character string from one code set to another code set (TR)

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. 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."

Substitution of macro parameters and variables, and evaluation and substitution of macro functions are the first operations performed upon a macro prototype during a macro expansion. Thus, macro control statements can be generalized in the same manner as any other statement in the macro prototype.

*

Macro functions require one, two, or (optionally) three arguments.

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 character string is specified, the AL function evaluates to zero.

Format:

?AL(arg)

arg

Character string whose length is to be determined; must be alphanumeric. (See "Designating Alphanumeric Values," at the end of this section.)

Example:

?AL(?L5+?P5)

If variable L5 equals 2AB, and parameter P5 equals 5B, the above function will be replaced with 6.

AT

TYPE ATTRIBUTE MACRO FUNCTION

The type attribute macro function inspects a character string and returns a code letter that indicates the type of data that makes up the inspected string. The code letters and the associated data types are as follows.

<i>Code Letter</i>	<i>Data Type</i>
A	ASCII string constant
B	Bit string constant
D	Double precision floating point constant
E	Single precision floating point constant
F	Short fixed point constant
I	Binary integer constant in decimal notation
L	Long fixed point constant
N	Null string
P	Packed decimal constant
U	Unpacked decimal constant
X	Binary integer constant in hexadecimal notation
Z	Hexadecimal string constant
O	Other than the above

Format:

?AT(arg)

arg

Character string to be inspected.

HEXADECIMAL CONVERSION MACRO FUNCTION

The hexadecimal conversion (CH) function converts a numeric argument to its hexadecimal equivalent.

Format:

?CH(arg₁[,arg₂])

arg₁

Value to be converted to hexadecimal; must be numeric. (See "Designating Numeric Values," at the end of this section.)

The numeric expression defining the value of arg₁ is allowed to define a 32-bit signed integer; i.e., $-2^{31} \leq \text{arg}_1 < 2^{31}$ must be satisfied.

arg₂

Value that specifies the *format* of the hexadecimal representation, as described below; must be numeric. (See "Designating Numeric Values," at the end of this section.)

<i>Value of arg₂</i>	<i>Meaning</i>
Not Specified	The value returned is a character string containing the source language representation of a binary integer constant in hexadecimal notation, with no insignificant zeros, having the same value as arg ₁ .
0	Arg ₁ is converted to a 32-bit signed twos-complement binary integer. This binary integer is then treated as if it were an 8-digit unsigned hexadecimal integer. The value returned is the character string representation of the significant digits of the unsigned hexadecimal integer.
>0	The value returned is a character string containing the source language representation of a binary integer constant in hexadecimal notation, specifying min(arg ₂ ,4) digits, having the same value as arg ₁ . When this format is used, the conditions $-2\min(4*\text{arg}_2 - 1, 15) \leq \text{arg}_1 \leq 2\min(4*\text{arg}_2 - 1, 15)$ must be satisfied.
<0	The value returned is the same as described for arg ₂ equal to zero, except that it is returned with min(-arg ₂ ,8) digits.

Examples:

<i>Condition</i>	<i>Function Specified</i>	<i>Result</i>
arg ₂ not specified	?CH(10)	X'A'
arg ₂ = 0	?CH(10,0)	A
arg ₂ > 0	?CH(10,1)	X'A'
	?CH(10,2)	X'0A'
	?CH(10,3)	X'00A'
	?CH(10,4)	X'000A'
	?CH(10,6)	X'000A'
arg ₂ < 0	?CH(10,-1)	A
	?CH(10,-2)	0A
	?CH(10,-3)	0A
	?CH(10,-4)	000A
	?CH(10,-6)	00000A

IX

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)

arg₁

Character string being searched, must be alphanumeric. (See "Designating Alphanumeric Values" at the end of this section.)

arg₂

Embedded character string for which the Macro Preprocessor will search; must be alphanumeric.

The value returned specifies the character position within arg₁ of the first (leftmost) character of the embedded character string. If arg₂ is not contained within arg₁ or arg₂ is a null character string (e.g.,"), a zero is returned.

Example:

?IX(ABCDE5,CDE5)

The above function reference 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 reference 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 in another specified character string.

Format:

?SR(arg₁,arg₂)

arg₁

String of characters from which one character at a time is selected from left to right, and compared to every character in arg₂, from left to right, until a match is found; must be alphanumeric. (See "Designating Alphanumeric Values" at the end of this section.)

arg₂

String of characters controlling the search; i.e., this argument specifies those characters that, if matched by a character of arg₁, satisfy the search; must be alphanumeric. (See "Designating Alphanumeric Values" at the end of this section.)

The search proceeds as follows. The leftmost character in arg₁ is compared with each character in arg₂ proceeding from left to right. If a match is found, the value of the function is 1, indicating that the leftmost character of arg₁ is also in arg₂. If no match is found, the second character is compared, etc. In general, the value of the function is the ordinal position of the leftmost character of arg₁ that is also a character within arg₂.

If none of the characters in arg₁ are found in arg₂, or if arg₁ or arg₂ is a null string, the value of the function is 0.

Example 1:

?SR(CHARSUBSTRING,STRING)

The above macro function reference 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 reference 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 reference is replaced with 2.

Example 3:

?SR(BA3,?L1)

The above macro function reference 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 function reference is replaced with 2.

SS

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(arg₁, arg₂ [,arg₃])

arg₁

Character string that contains the characters to be included in the source statement; must be alphanumeric. (See "Designating Alphanumeric Values," at the end of this section.)

arg₂

Character position of the first character is arg₁ that is to be included; must be numeric. (See "Designating Alphanumeric Values," at the end of this section.)

arg₃

Number of characters to be included; must be numeric. (See "Designating Alphanumeric Values," at the end of this section.)

Default: The character whose character position was specified in arg₂, and all subsequent characters of arg₁.

If arg₁ is a null character string, or if arg₂ is ≤0, or if the value specified in arg₂ is greater than the length of arg₁, a null character string is included in the source statement.

Example 1:

?SS(?P2,?L5,3)

If P2=ABCDE and L5=2, the above function reference 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 reference 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 defines G6 to be the character string C?5.

TRANSLATE MACRO FUNCTION

The translate macro function translates an alphanumeric character string from one code set to another. The translation proceeds as follows. The string specified by arg_1 is processed from left to right character by character. The arg_1 character is compared with the characters in the string specified by arg_3 . The first time a match occurs, the ordinal position in the arg_3 string is noted, and the character in the same ordinal position in arg_2 is moved to the result. If no match occurs, the arg_1 character is moved to the result; i.e., no translation. Thus, the result is an alphanumeric string of the same length as that of arg_1 .

Format:

?TR(arg_1 , arg_2 , [arg_3])

arg_1

The input string; i.e., the string to be translated. Must be alphanumeric; (See "Designating Alphanumeric Values" at the end of this section.)

arg_2

The string consisting of the code set from which the characters of the result are selected (if a translation occurs). Must be alphanumeric; (See "Designating Alphanumeric Values" at the end of this section.) If the character string specified by arg_2 is shorter than that of the character string specified by arg_3 , arg_2 is padded on the right with spaces until it is the same length as arg_3 .

arg_3

The string consisting of the code set to which the characters of the input string are matched. Must be alphanumeric; (See "Designating Alphanumeric Values" at the end of this section.) If arg_3 is omitted, the ASCII code set (See Figure B-4) is used as arg_3 .

Example:

```
LXΔSETAΔ?TR(?P1,'ABCDEFGHJKLMNOPQRSTUVWXYZ';
           'abcdefghijklmnopqrstuvwxyZ')
```

The macro control statement SETA assigns an alphanumeric value to local macro variable X. The value assigned (the result of the Translate function) is an alphanumeric string that is the same as the alphanumeric string specified by P1 except that each lowercase letter is changed to the corresponding uppercase letter. The results of the translation for various strings specified by P1 are shown below.

P1

axes	AXES
Man2	MAN2
3-a12	3-A12
New York	NEW YORK

V

VECTOR ORIENTATION MACRO FUNCTION

The vector orientation (V) functions permits macro parameters and macro variables to be referenced by their ordinal 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

Value that identifies a parameter or variable; must be from 1 to 35; 1 identifies the first parameter or variable, 2 the second, etc.; must be numeric. (See "Designating Numeric Values" at the end of this section.)

Example:

?SS(?VP(10),2,3)

The above function illustrates usage of the vector orientation function within a substring (SS) function. The function reference ?VP(10) identifies parameter PA. If PA = ABCDE, the above substring function reference is replaced with BCD.

VERIFY MACRO FUNCTION

The verify (VR) function causes the Macro Preprocessor to determine which character of a specified character string is the first (leftmost) character that is not in another specified character string.

Format:

?VR(arg₁,arg₂)

arg₁

String of characters from which one character at a time is selected, from left to right, and compared to every character in arg₂, from left to right, until no match is found; must be alphanumeric. (See "Designating Alphanumeric Values" at the end of this section.)

arg₂

String of characters controlling the search; i.e., this argument specifies those characters that, if unmatched by a character of arg₁, satisfy the search; must be alphanumeric. (See "Designating Alphanumeric Values" at the end of this section.)

The verification proceeds as follows. The leftmost character in arg₁ is compared with each character in arg₂ proceeding from left to right. If no match is found, the value of the function is 1, indicating that the leftmost character of arg₁ is not also in arg₂. If a match is found, the second character is compared, etc. In general, the value of the function is the ordinal position of the leftmost character of arg₁ that is not a character within arg₂.

If all the characters in arg₁ are found in arg₂, or if arg₁ is a null string, the value of the function reference is 0.

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 reference 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 reference with a 2.

EXAMPLE ILLUSTRATING MACRO FACILITY

Figure 8-1 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 MACRU 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     $R1,?P2      STORE POLYNOMIAL VALUE.
      ENDM
*
*
FACTOR   MAC
*
*THIS MACRU GENERATES CODE WHICH MULTIPLIES ($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 MACRU 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))
*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.

```

Figure 8-1. Sample Unexpanded Source Module and Assembler Listing of Resulting Expanded Source Module

```

*
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
EUF

```

```

::
**ASSEMBLER LISTING OF RESULTING EXPANDED SOURCE MODULE

```

```

USEMAC          ASSEMBLER-0110-05/01/1433          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          STM      $R3,$B2.+$R1
000046 001C 9970 000B          CMK      $R1,=X'B'
000047 001E 0902          BE      >$+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 8-1 (cont). Sample Expanded Source Module and Assembler Listing of Resulting Expanded Source Module

PROGRAMMING CONSIDERATIONS

1. If an expanded source program, each macro control statement and each other type of statement that contains error flag(s) can comprise up to 249 characters. Each other line can comprise up to 255 characters. Subsequent characters are truncated.
2. Input to the Macro Preprocessor may be either uppercase or lowercase characters. All lowercase characters in ASCII and hexadecimal string constants, and in hexadecimal integer constants remain lowercase characters; all other lowercase characters within the source module are converted to uppercase.
3. 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. Specifying in LIBM macro control statements only those macro routines that will actually 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 (i.e., 001,002,...,009,00A,...,00Y,00Z,010,011,...,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 ?G1?L1SM, the label would be \$001SM the first time the routine is requested, and \$002SM the second time the routine is requested if no other routines were requested in between the two requests and also assuming the initial value of G1 has not been altered.

It is recommended that labels generated by macro routines be of the form ?G1?L1xx where xx are any two characters that are unique within a given expansion of a given macro routine. It is further assumed that the global macro variable G1 contains a single character that will not be used as the first character of any label, except for those labels generated in a macro expansion.

L2

Numeric value that designates the current level of macro call nesting in the current macro call. If the macro call is not a nested macro call, L2 equals 0.

L3

Numeric value that designates the ordinal number of the last parameter that was assigned a value in the current macro call. If the macro call does not include any parameters, L3 equals 0.

L4

Label of the current macro call. If no label is specified, L4 equals a null character string.

Global macro variable G1 is initialized to equal \$. Global macro variable G2 is initialized to a 16 character string whose value reflects the contents of the External Switch Word at the time the macro processor was initiated. The first character of G2 will be the character "0" if external switch 0 was off, or the character 1 if external switch 0 was on; the second character of G2 will be a 0 or 1 depending on whether external switch 1 was off or on respectively; etc. Each other global variable is a null character string. These values remain in effect unless they are reassigned in SETA or SETN macro control statements.

G1

Global macro variable, initialized to equal \$.

G2

Global macro variable, initialized to a 16-character string. This value reflects the contents of the External Switch Word at the time the macro processor was initiated. The first character of G2 will be the character "0" if external switch 0 was off, or the character 1 if external switch 0 was on; the second character of G2 will be a 0 or 1, depending on whether external switch 1 was off or on, respectively; etc.

G3-G9, GA-GZ

Each of these global variables is a null character string.

NOTE: The above global variable values remain in effect until 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 +32767¹. (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:

- Binary integer constant in decimal notation (e.g., 31764, +4652)
- Binary integer constant in hexadecimal notation (e.g., +X'2F', X'7000')
- Substitution operator followed by macro variable name whose contents are the source language representation of a binary integer constant in decimal or hexadecimal notation (e.g., ?G3, ?L4)
- The hexadecimal conversion function, CH, allows the numeric expression which defines the first argument to evaluate a 2-word signed integer value.
- Substitution operator followed by macro parameter name whose contents are the source language representation of a decimal or hexadecimal integer constant (e.g., ?P2)
- Substitution operator followed by a macro function that returns a numeric value
- Expression that combines any of the above character strings by including arithmetic operators (e.g., 31764+(?G3)) (See "Expressions" in Section 2.)
- Assembler functions (See "Expressions" in Section 2.)

DESIGNATING ALPHANUMERIC VALUES

When an operand or argument requires an alphanumeric value, you can specify any type of alphanumeric character string, including the following:

- A character string that does *not* contain a space, horizontal tab, comma, semi-colon, apostrophe, left or right parenthesis
- An ASCII string constant (which must not be preceded by the optional letter 'A')
- Substitution operator followed by macro variable name
- Substitution operator followed by macro parameter name
- Substitution operator followed by macro function
- Expression that combines any of the above character strings by specifying them adjacent to each other or encloses an individual or concatenated string within balanced parentheses.

An alphanumeric value is terminated by a space, a horizontal tab, comma, semi-colon, unbalanced right parenthesis, or end of line, according to the system rules of the statement or function in which it appears.

Each ASCII string constant in an alphanumeric value is converted to the length delimited internal form by first replacing each unprotected substitution operator and following macro parameter or variable name by the content of that parameter or variable and replacing each unprotected substitution operator and following macro function reference by the value returned by that function. After all substitutions have been completed, the delimiting apostrophes are removed and any embedded apostrophe representations are reduced to a single character each.

Example:

```
?SS('ABC"DEF"GH',4,5)
```

In this example, the first argument of the substring function evaluates to the character string ABC'DEF'GH. The resultant substring (i.e., five characters, beginning at the fourth character) is 'DEF'.

NOTE: The arguments of a macro call statement and the arguments of a MAC statement do not have ASCII string constants converted to their length delimited form at the time the macro call statement or the MAC statement is processed. This conversion occurs when the parameter values are subsequently used as alphanumeric values within the macro routine.

Any combination of characters may be used as an argument of a macro call or a MAC statement provided that the rules described below are obeyed.

The following conventions have been defined for alphanumeric values.

ALPHANUMERIC VALUE CONVENTIONS

Balanced Apostrophes

An alphanumeric value may contain one or more quoted strings. A quoted string is any sequence of characters that begins and ends with an apostrophe. Any apostrophe to be included as part of the quoted string must be entered as a pair of apostrophes (i.e., two consecutive apostrophes). An apostrophe cannot also be paired with a third apostrophe within the quoted string.

The first quoted string starts with the first apostrophe in the argument. A quoted string ends with the first "even-numbered" apostrophe that is not immediately followed by another apostrophe. Subsequent quoted strings start with the first apostrophe after the apostrophe that ends the previous quoted string.

The first and last apostrophes of a quoted string are called "balanced" apostrophes.

The following example contains two quoted strings. The first and fourth and the fifth and sixth apostrophes are balanced sets of apostrophes (i.e., E is not a quoted string).

```
'ABC"D'E'F'
```

NOTE: Commas, semicolons, parentheses, spaces, and horizontal tabs within a quoted string do *not* terminate an alphanumeric value.

Balanced Parentheses

If an alphanumeric value or a macro call argument or a MAC statement argument contains parentheses, there must be an equal number of left and right parentheses. The nth right parenthesis must appear to the right of the nth left parenthesis.

A "set" of parentheses is balanced when a right parenthesis follows the left parenthesis. Any intervening parentheses must be members of an included set of balanced parentheses. The included set must be entirely between the including set. In the following example, the first and fourth, the second and third, and the fifth and sixth parentheses are sets of balanced parentheses:

```
(ABC(D)E)(F)
```

A parenthesis that appears between balanced apostrophes (i.e., is part of a quoted string) is not considered in determining balanced parentheses. In the following example, the middle parenthesis is not considered:

```
(')')
```

NOTE: Commas, semicolons, spaces, and horizontal tabs between balanced parentheses do *not* terminate an argument of a macro call.

Commas and Semicolons

A comma or semicolon which is not between balanced apostrophes or balanced parentheses indicates the end of an alphanumeric value, the end of an argument of a macro call, or the end of an argument of a MAC statement. The following example shows one argument containing the two commas:

(ABC,DE)F'G,H'

NOTE: The comma or semicolon that terminates an argument of a macro call is not considered to be a part of that argument.

A comma is used to terminate an argument if the next argument begins on the same source line. The next argument starts with the first character following the comma that is neither a space nor a horizontal tab.

A semicolon is used to terminate an argument if the next argument begins on the next source line. In this case, the next argument starts with the first character that is neither a space nor a horizontal tab (following the line number, if present) on the next source line.

Spaces and Horizontal Tabs

A space or horizontal tab indicates the end of an alphanumeric value, the end of the last argument of a macro call, or the end of the last argument on a MAC statement unless the space or tab appears:

- Between balanced apostrophes.
- Following a comma.
- Following the line number of a continuation line with no intervening characters, other than additional spaces and horizontal tabs.



Appendix A

Programmer's Reference Information

This appendix summarizes 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.

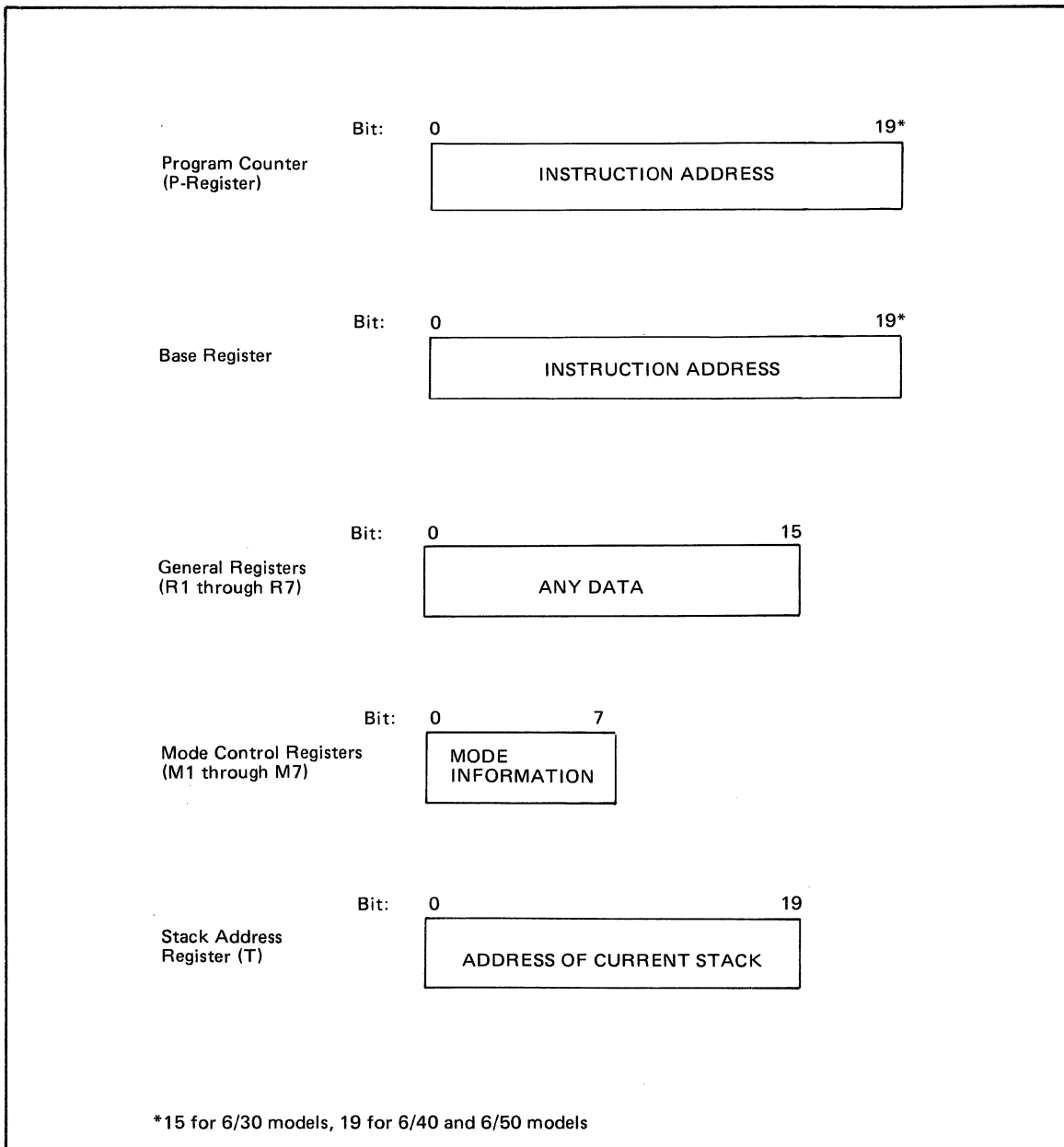


Figure A-1. Level 6 Hardware Registers

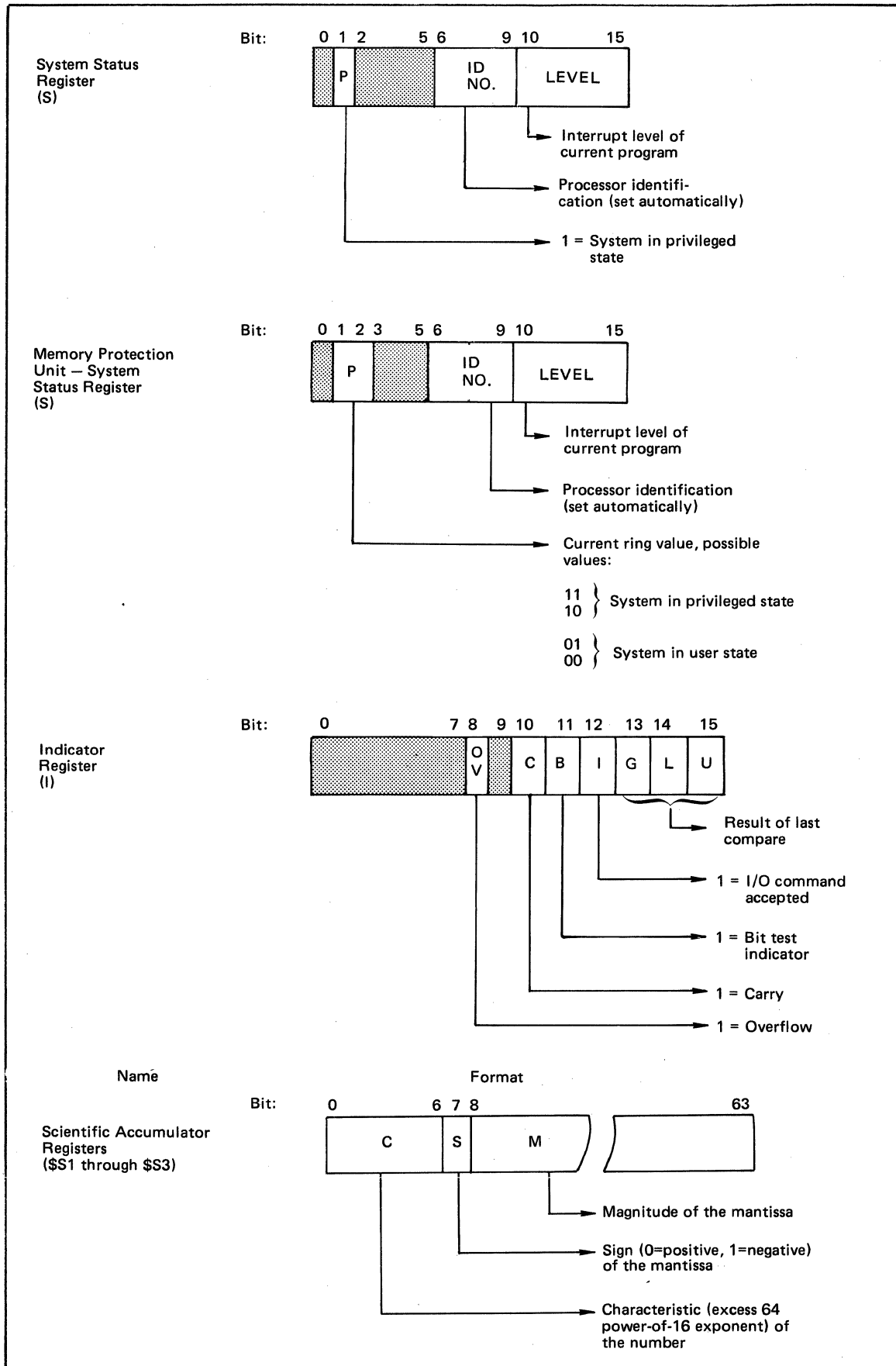


Figure A-1 (cont). Level 6 Hardware Registers

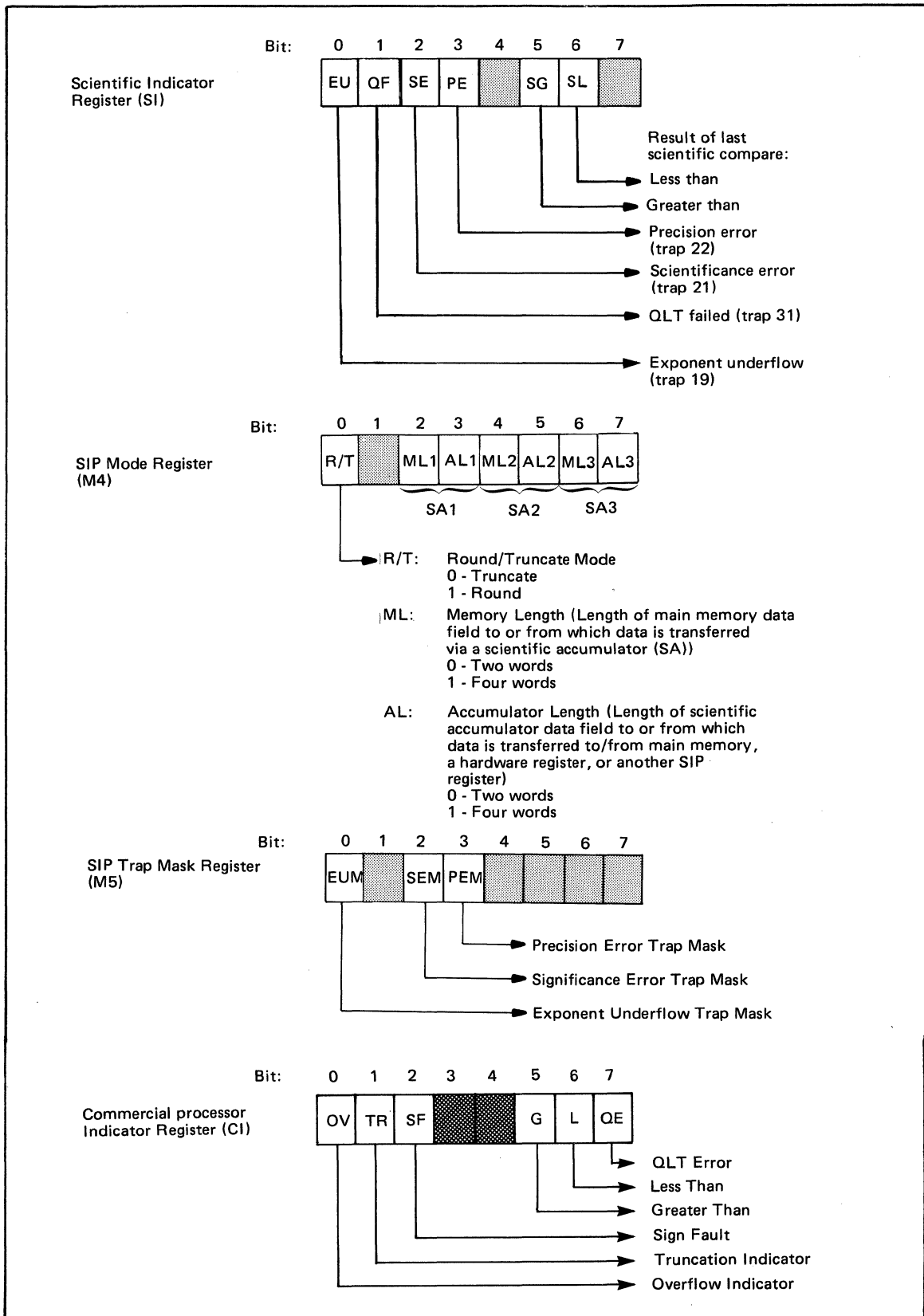


Figure A-1 (cont). Level 6 Hardware Registers

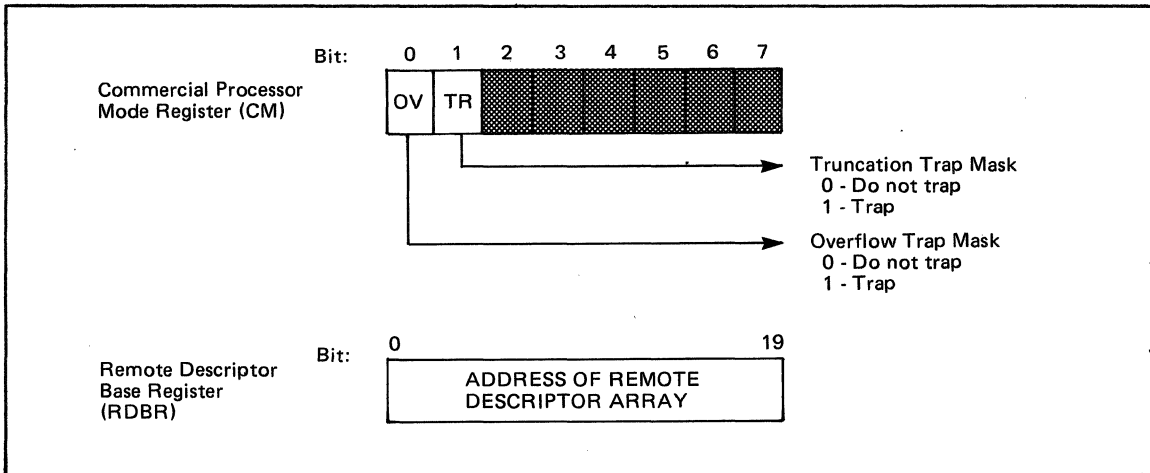


Figure A-1 (cont). Level 6 Hardware Registers

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.

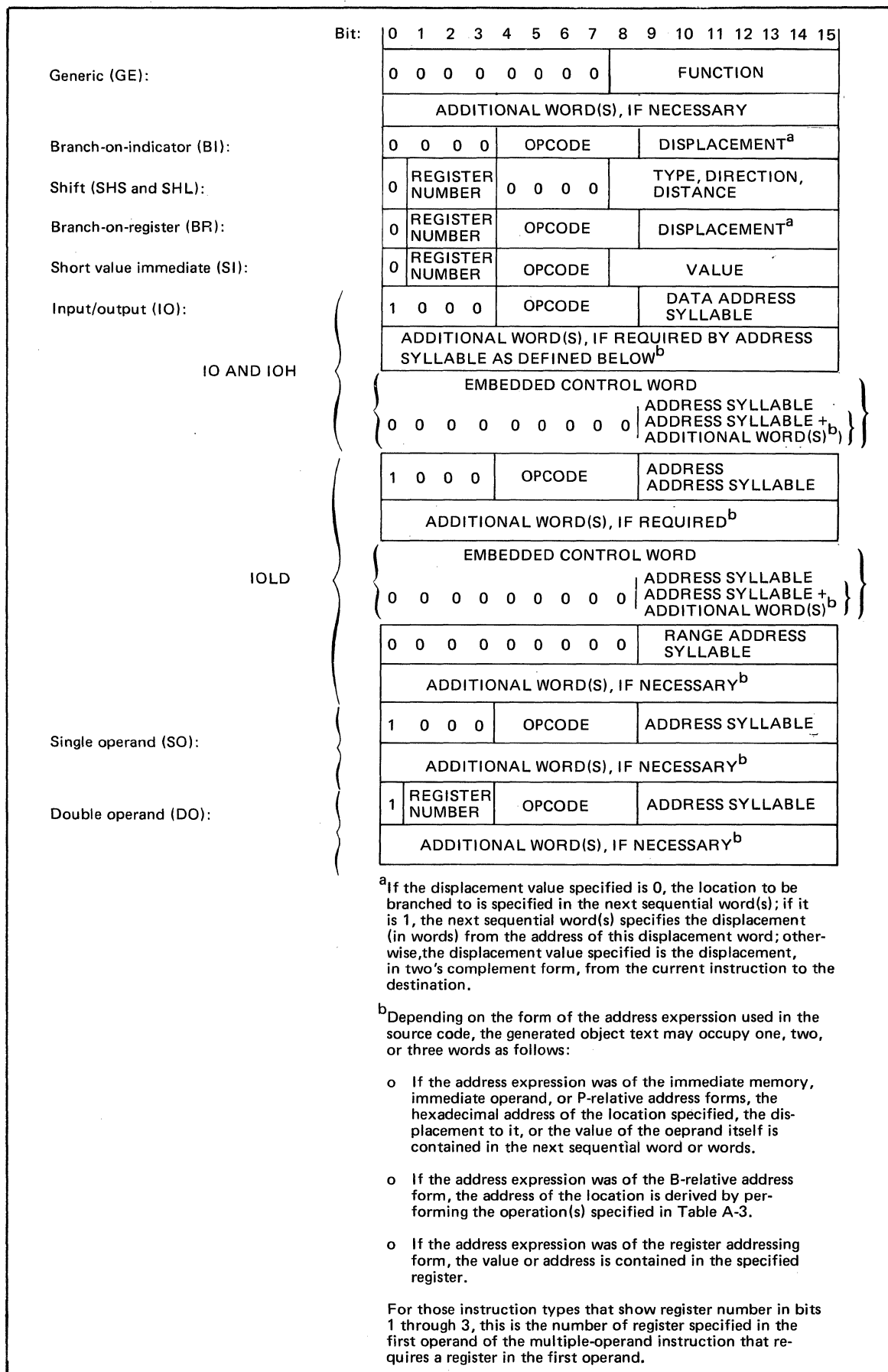


Figure A-2. Internal Formats of Assembly Language Instructions

HEXADECIMAL REPRESENTATION OF INSTRUCTIONS

Table A-1 illustrates the hexadecimal representation of the CPU and SIP 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. See Appendix H for hexadecimal representation of Commercial Processor instructions.

TABLE A-1. INTERNAL REPRESENTATION 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			
		0	0	8		MMM
		0	0	A		ASD
		0	0	B		VLD
		0	0	C		LRDB
		0	0	D		SRDB
		1	1	1		CNFG
		6	6	0		DQA
		6	6	1		QOT
6	6	2	DQH			
6	6	3	QOH			
2	2	0+x	x	BL	BI	
			x	BGE		
			x	BG		
			x	BLE		
			x	BOV		
			x	BNOV		
5	5	0+x	x	BBT		

TABLE A-1 (CONT). INTERNAL REPRESENTATION OF ASSEMBLY LANGUAGE INSTRUCTIONS

First Hexadecimal Digit	Second Hexadecimal Digit	Third Hexadecimal Digit	Fourth Hexadecimal Digit	Instruction	Type	
0	5	8+x	x	BBF	BI	
	6	0+x	x	BCT		
	6	8+x	x	BCF		
	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	BI	
	5	8+x	x	SBGEZ		
		0+x	x	SBEZ		
		8+x	x	SBNEZ		
		6	0+x	x		SBGZ
6	8+x	x	SBLEZ			
4	4	0+x	x	SBL	BI	
	5	8+x	x	SBGE		
		0+x	x	SBEQ		
		8+x	x	SBNE		
		6	0+x	x		SBG
6	8+x	x	SBLE			
5	4	0+x	x	SBPE	BI	
6	8+x	x	SBNPE			
	4	0+x	x	SBSE		
	8+x	x	SBNSE			
	7	4	0+x	x		SBEU
7	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	d			
		9	d	DOL		
		A	d	DAL		SHL
		B	d			
		C	d			
D	d	DOR				
E	d					
F	d	DAR				

TABLE A-1 (CONT). INTERNAL REPRESENTATION OF ASSEMBLY LANGUAGE INSTRUCTIONS

First Hexadecimal Digit	Second Hexadecimal Digit	Third Hexadecimal Digit	Fourth Hexadecimal Digit	Instruction	Type		
1-7	7	0+x	x	BDEC	BR		
	7	8+x	x	BINC			
	8	0+x	x	BLZ			
	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			
	4		0+addsyl	AID			
	4		8+addsyl	SID			
	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+addsyl	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			

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	3		0+addsyl	DIV	DO
	3		8+addsyl	LNJ	
	4		0+addsyl	OR	
	4		8+addsyl	ORH	
	5		0+addsyl	AND	
	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	DO
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 FOR CPU & SIP INSTRUCTIONS

mmm	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	$\begin{Bmatrix} =\$Rn \\ =\$Bn \\ =\$Sk \end{Bmatrix}$	\$Bk.-\$R1	\$Bq.+\$R1
110	reserved	reserved	-\$Bn	\$Bk.-\$R2	\$Bq.+\$R2
111	$\begin{Bmatrix} =\text{location} \\ =\text{value} \end{Bmatrix}$	\$IV. value	+\$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 = 5, 6, 7. For more information about these address expressions, see "Addressing Techniques" in Section 5.

VALID ADDRESS EXPRESSIONS

Table A-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 FOR CPU AND SIP INSTRUCTIONS

Addressing Technique		Address Expression Form	Generation of Effective Address
Register Addressing		$=\$R_n$ $=\$B_n$ $=\$S_n$	$\underline{R_n} = \underline{EA}$ $\underline{B_n} = \underline{EA}$ $\underline{S_n} = \underline{EA}$
Immediate Memory Addressing	Direct	$\left\langle \begin{array}{l} \text{locexpression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{templabel} \end{array} \right\rangle$	location = EA
	Indirect	$*\left\langle \begin{array}{l} \text{locexpression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{templabel} \end{array} \right\rangle$	<u>location</u> = EA
	Indexed Direct	$\left\langle \begin{array}{l} \text{locexpression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{templabel} \end{array} \right\rangle .SR \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$	location + $R \begin{array}{l} (1) \\ (2) \\ (3) \end{array} = EA$
	Indexed Indirect	$*\left\langle \begin{array}{l} \text{locexpression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{templabel} \end{array} \right\rangle .SR \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$	<u>location</u> + $R \begin{array}{l} (1) \\ (2) \\ (3) \end{array} = EA$
Immediate Operand Addressing		=locexpression	Address of current address syllable + 1 = EA
		=stringconstant	
		$= \left\{ \begin{array}{l} \text{intvalexpression} \\ \text{extvallabel} \end{array} \right\}$	
P-Register Addressing	Direct	$\left\{ \begin{array}{l} \text{intlocexpression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{templabel} \end{array} \right\}$	internal location = EA
	Indirect	$* \left\{ \begin{array}{l} \text{intlocexpression} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{templabel} \end{array} \right\}$	<u>internal location</u> = EA
B-Register Addressing	Direct	$\$B_n$	$\underline{B_n} = EA$
	Indirect	$*\$B_n$	$\underline{B_n} = \text{location}$ $\underline{\text{location}} = EA$
	Indexed Direct	$\$B_n .SR \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$	$\underline{B_n} + R \begin{array}{l} (1) \\ (2) \\ (3) \end{array} = EA$
	Indexed Indirect	$*\$B_n .SR \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$	$\underline{B_n} = \text{location}$ $\underline{\text{location}} + R \begin{array}{l} (1) \\ (2) \\ (3) \end{array} = EA$
	Direct + Displacement	$\$B_n \cdot \left\{ \begin{array}{l} \text{intvalexpression} \\ \text{extvallabel} \end{array} \right\}$	$\underline{B_n} + \text{value} = EA$

TABLE A-3 (CONT). SUMMARY OF VALID FORMS OF ADDRESS EXPRESSIONS FOR CPU

Addressing Technique	Address Expression Form	Generation of Effective Address
B-Register Addressing (Cont.)	Indirect + Displacement $*\$B_n \cdot \left\{ \begin{array}{l} \text{intvalexpression} \\ \text{extvallabel} \end{array} \right\}$	$\underline{B_n} + \text{value} = \text{location}$ $\underline{\text{location}} = \text{EA}$
	B6 direct + Displacement $\$B6.\$L\text{COMW} + \text{intvalexpression}$	$\underline{B6} + \text{value} = \text{EA}$
	B6 indirect + Displacement $*\$B6.\$L\text{COMW} + \text{intvalexpression}$	$\underline{B6} + \text{value} = \text{location}$ $\underline{\text{location}} = \text{EA}$
	Push $-\$B_n$	$\underline{B_n} \leftarrow (\underline{B_n} - 1)$ $\underline{B_n} = \text{EA}$
	Pop $+\$B_n$	$\underline{B_n} = \text{EA}$ $\underline{B_n} \leftarrow (\underline{B_n} + 1)$
	Indexed Push $\$B \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\} \cdot -\$R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}$	$\underline{R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}} \leftarrow (\underline{R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}} - 1)$ $\underline{B \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}} + \underline{R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}} = \text{EA}$
Indexed Pop $\$B \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\} \cdot +\$R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}$	$\underline{B \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}} + \underline{R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}} = \text{EA}$ $\underline{R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}} \leftarrow (\underline{R \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right\}} + 1)$	
Short Displacement $> \left\{ \begin{array}{l} \text{intlocexpression} \\ + \\ - \\ \text{templabel} \end{array} \right\}$	$\text{location} = \text{EA}$	
Special $> = \left\{ \begin{array}{l} \text{intvalexpression} \\ \text{extvallabel} \end{array} \right\}$	Word following the word(s) containing op code + first operand address syllable = EA	
Interrupt Vector $\$IV \cdot \left\{ \begin{array}{l} \text{intvalexpression} \\ \text{extvallabel} \end{array} \right\}$	$\underline{IV} + \text{value} = \text{EA}$	

NOTE: The symbols used in this table have the following meanings:

- | | |
|---|-----------------------------------|
| $\underline{\quad}$ - Contents of . . . | * - Indirect memory addressing |
| EA - Effective Address | < - Immediate memory addressing |
| \leftarrow - Replaces the element pointed at | > - Short displacement addressing |
| locexpression - location expression (any type) | >= Specified Addressing |
| templabel - temporary label | .- Component separator |
| stringconstant - string constant | (indexing and displacement) |
| intvalexpression - internal value expression | |
| 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 as 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 is stored as binary, and is shown in the printout as the hexadecimal equivalent of the stored binary value.

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.

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.

TABLE B-2. STORAGE AND PRINTOUT OF 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)	0000000000100 000	0020	.Space
B'00110010'	0011001000000000	3200	2.

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 27,401, do the following:

1. Divide 27,401 by 16.
The result is 1712. The remainder is 9.
2. Divide 1712 by 16.
The result is 107. The remainder is 0.
3. Divide 107 by 16.
The result is 6. The remainder is 11.
4. Divide 6 by 16.
The result is 0. The remainder is 6.

Using Table B-1, you can see that in hexadecimal 11 is represented by B. Thus, the hexadecimal equivalent of 27401₁₀ is 6B09.

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,

3132 3333 3335 3637 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 = 12₁₀).
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 = 10₁₀).
The result is 7306.

Thus, the decimal equivalent of 1C8A₁₆ 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
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.

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 twos complement before using the table. You can compute the twos complement by subtracting the hexadecimal number from 10000 (hexadecimal) or by changing all 0's to 1 and then adding a binary 1. The twos complement can also be computed by performing the ones complement of all of the bits that are to the left of the rightmost 1. (e.g., the twos complement of 0011 1000 1100 0000 is computed by "flipping" the leftmost 9 bits giving 11000111 0100 0000.) This assumes that the twos complement of zero is zero. 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							
		0	1	2	3	4	5	6	7
H2	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	~	
F	SI	US	/	?	O	_	o	DEL	

NUL Null
 SOH Start of Header
 STX Start of Text
 ETX End of Text
 EOT End of Transmission
 ENQ Enquiry
 ACK Acknowledge
 BEL Bell
 BS Backspace
 HT Horizontal Tab

- 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

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:

$$\begin{array}{r}
 \text{multiplicand} \quad \quad \quad 2A5 \\
 \text{multiplier} \quad \quad \quad \times 3 \\
 \hline
 \text{product} \quad \quad \quad 7EF
 \end{array}$$

Using the table, $3 \times 5 = F$, and there is no remainder. Then, $3 \times A = 1E$, 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, $3 \times 2 = 6$, 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 programs illustrate many of the aspects of the assembly language described in this manual. For a definition of the fields that appear in the listings, refer to the *Program Execution and Checkout* manual.

```

CHKNML  -SAF  1977/11/21 0940:05.6 ASSEMBLER-0100-11/09/1223 GCOS6 MOD0400-S100-11/17/0634 PAGE 0001

000001          TITLE      CHKNML
000002          * PROGRAM COMPARES TEST RESULTS OF TEST MODULES WHOSE ADDRESSES ARE
000003          * STORED IN $COMM TO THE EXPECTED TEST RESULTS AS DESCRIBED IN TABLOC
000004          XVAL      TSTMAX
000005          XLOC      TABLOC
000006          XLOC      ZIOSOL
000007          XLOC      ZIOSWR
000008          XLOC      ZIOSCO
000009          COMM     X'100'
000010          * GET FILENAME AND CHANNEL NO
000011  0000  AR43 FFEF  STRT  LDR  SR2,$B3,-17
000012  0002  BBA3      LAB  SR3,$R3,$R2
000013  0003  BBC3 FFEC  LAB  >B3,$B3,-20      SET B3 TO LIST FILE AT
000014  0005  9A73      LDR  SR1,+$R3      SET B3 TO FILENAME
000015  0006  1D02      CMV  SR1,2
000016  0007  09R1 007C  BNE  ERNLST      NO LIST FILE ATTACHED
000017  0009  9843 0007  LDR  SR1,$R3,7      SET R1 TO CHANNEL NO.
000018          * OPEN LIST FILE
000019  000B  CBC0 0079  LAB  SR4,LSTDCB
000020  000D  D380 0000  X    LNJ  SR5,<ZIOSOL      OPEN ROUTINE
000021  000F  1981 006F  BNEZ  SR1,EROPFN
000022          * WRITE HEADER MSG
000023  0011  1C1E      LDV  SR1,X'1E'      MSG LENGTH
000024  0012  2C00      LDV  SR2,X'0'
000025  0013  BHC0 0081  LAB  SR3,WBUF01      MSG ADDRESS
000026  0015  CBC0 006F  LAB  SR4,LSTDCB
000027  0017  D380 0000  X    LNJ  SR5,<ZIOSWR      WRITE ROUTINE
000028  0019  1981 0066  BNEZ  SR1,FRHDR
000029  001B  3CFF      LDV  SR3,-X'1'
000030  001C  3E01      ADV  SR3,X'1'
000031  001D  B970 0000  X    CMR  SR3,=TSTMAX      CHECKED ALL TEST RESULTS ?
000032  001F  0301 004E  RG  ENUTST
000033  0021  CC60 0000  K    LDB  SR4,<$COMM,$R3
000034  0023  CBC4 001C  LAB  SR4,$B4,X'1C'      CREATE STATUS BLOCK PTR
000035  0025  F830 0000  X    LDR  SR7,<TABLCC,$R3      GET EXPECTED VALUE
000036  0027  F944 0003  CMK  SR7,$B4,X'3'      COMPARE TO ACTUAL STATWD
000037  0029  0973      BE  >TLOOP      TEST OK - CHECK NEXT TEST
000038  002A  EBC0 007A  LAB  SR6,WBUF2A
000039  002C  D830 0000  K    LDR  SR5,<$COMM,$R3
000040  002E  F3C0 0027  LNJ  SR7,DUMPWD      CONVERT TEST ADDR TO ASCII
000041  0030  EBC0 0077  LAB  SR6,WBUF2B
000042  0032  D804      LDR  SR5,$B4
000043  0033  F3C0 0022  LNJ  SR7,DUMPWD      CONVERT SYML VALUE TO ASCII
000044  0035  CBC4 0001  LAB  SR4,$B4,X'1'
000045  0037  EBC0 0073  LAB  SR6,WBUF2C
000046  0039  D804      LDR  SR5,$B4
000047  003A  F3C0 001H  LNJ  SR7,DUMPWD      CONVERT TEST NUM TO ASCII
000048  003C  CBC4 0001  LAB  SR4,$B4,X'1'
000049  003E  EBC0 006F  LAB  SR6,WBUF2D
000050  0040  D804      LDR  SR5,$B4
000051  0041  F3C0 0014  LNJ  SR7,DUMPWD      CONVERT SYMV VALUE TO ASCII
000052  0043  CBC4 0001  LAB  SR4,$B4,X'1'
000053  0045  EBC0 006B  LAB  SR6,WBUF2E
000054  0047  D804      LDR  SR5,$B4
000055  0048  F3C0 000D  LNJ  SR7,DUMPWD      CONVERT STATUS WORD TO ASCII
000056          * WRITE VALUES
000057  004A  1C1E      LDV  SR1,X'1E'      MSG LENGTH
000058  004B  2C00      LDV  SR2,X'0'
000059  004C  BBC0 0057  LAB  SR3,WBUF20      MSG ADDRESS
000060  004E  CBC0 0036  LAB  SR4,LSTDCB

```

Figure C-1. Listing of CHKNML Program

```

000061 0050 D380 0000 X LNJ SR5,<ZIOSWK WRITE ROUTINE
000062 0052 1981 002E HNEZ SR1,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 R6
000066 0056 4CFC DUMPWD LDV SR4,-X'J' SET COUNTER
000067 0057 C940 0000 T STR SR4,+SC
000068 0059 7C00 LDV SR7,X'0'
000069 005A 4C00 SA LDV SR4,X'0'
000070 005B 5084 DDL SR5,4
000071 005C 4E30 ADV SR4,X'30'
000072 005D C940 0000 T CMR SR4,+SF
000073 005F 0380 T RLF >+SE
000074 0060 4E07 ADV SR4,X'07'
000075 0061 F454 SE OR SR7,=SR4
000076 0062 8AC0 FFF5 T INC +SL
000077 0064 0600 T BCT >+SD
000078 0065 7088 DDL SR7,8
000079 0066 0FF4 T B >+SA
000080 0067 EF46 0000 SD STR SR6,SR6,X'0'
000081 0069 FF46 0001 STR SR7,SR6,X'1'
000082 006B 8387 JMP RETURN TO CALLER
000083 006C 0000 SC DC Z'0'
000084 006D 0039 SF DC Z'0039'
000085 * WRITE END TEST
000086 006E 1C0A ENDTST LDV SR1,X'A' MSG LENGTH
000087 006F 2C00 LDV SR2,X'0'
000088 0070 88C0 0043 LAB SR3,WBUF03 MSG ADDRESS
000089 0072 C8C0 0012 LAB SR4,LSTDCB
000090 0074 D380 0000 X LNJ SR5,<ZIOSWK WRITE ROUTINE
000091 0076 1981 0008 HNEZ SR1,EREND
000092 * CLOSE LIST FILE
000093 0078 C8C0 0000 LAB SR4,LSTDCB
000094 007A D380 0000 X LNJ SR5,<ZIOSWK CLOSE ROUTINE
000095 007C 1981 0006 HNEZ SR1,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

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

```

Figure C-1 (cont). Listing of CHKNML Program

```

RURBLE 101577      BURBLE SORT      -SLIC 1977/12/07 1021:20.8 ASSEMBLER-0100-11/17/1346  GC056 MOD400-5100-12/01/1413  PAGE 0001

000001              TITLE      BURBLE,'101577' BURBLE SORT
000002              *THIS SUBROUTINE DOES A SIMPLE BURBLE SORT OF WHATEVER
000003              *SINGLE PRECISION BINARY INTEGERS ARE IN COMMON BLOCK, DATA.
000004              *THE SORT LEAVES THE DATA IN THE COMMON BLOCK IN ASCENDING
000005              *NUMERICAL SEQUENCE.
000006              *
000007              *USAGE IS  LMT $B5,BURBLE
000008              *
000009              *THIS PROGRAM WILL EXECUTE IN BOTH SAF AND LAF ADDRESS MODES,
000010              *HENCE ASSEMBLED IN SLIC MODF.
000011              *
000012              0000      K      LODATA      EQU      DATA
000013              0009      K      HIDATA      EQU      DATA+9
000014              0000 98C0 0009      P      BURBLE      LAB      $R1,HIDATA
000015              0002 ABC0 0000      P      LINE2      LAB      $R2,LODATA
000016              0004 1C00              LDV      $P1,0
000017              0005 B872              LINE4      LDR      $R3,+$R2
000018              0006 B902              CMK      $P3,$R2
000019              0007 0385              BLE      >LINE10
000020              0008 1C01              LDV      $P1,1
000021              0009 BE02              SWP      $P3,$P2
000022              000A BF42 FFFF              STR      $R3,$R2,-1
000023              000C AD01              LINE10     CMR      $R2,=$B1
000024              000D 0278              BL       >LINE4
000025              000E B9E1              CMZ      =*B1
000026              000F 19F3              RNEZ     $R1,>LINE2
000027              0010 B385              JMP
000028              *
000029              000A      DATA      COMM      10
000030              0000      K      ORG      DATA
000031              0000 1234              DC      X'1234',99H(15,0),Z'A3DE',100+'A',AND(Z'2FF',X'444'),-853
000032              0001 0063
000033              0002 A3DE
000034              0003 0184
000035              0004 0440
000036              0005 FCAB
000037              0006 0000              DC      0,MAX(-55,X'22'),4003,-X'R000'
000038              0007 0022
000039              0008 0FA3
000040              0009 R000
000041              000A      END      BURBLE
0000 ERR COUNT
00152 WORD SYMROL TABLE

```

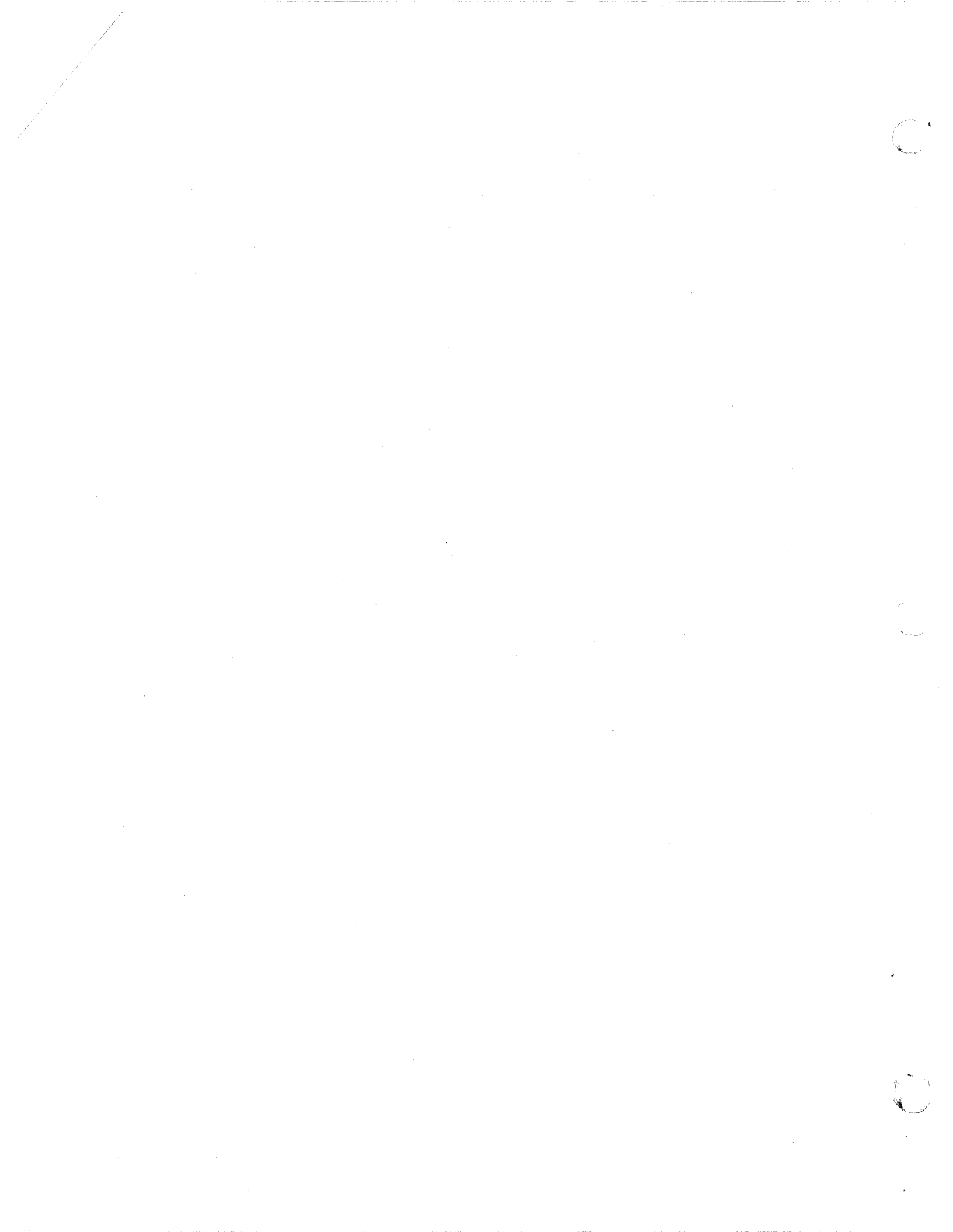
```

RURBLE 101577      BURBLE SORT      -SLIC 1977/12/07 1021:20.8 ASSEMBLER-0100-11/17/1346  GC056 MOD400-5100-12/01/1413  PAGE 0002

$B1      ****      14      23      25
$B2      ****      15      17      18      21      22      23
$B5      ****      27
$R1      ****      16      20      26
$R3      ****      17      18      21      22
N BURBLE      14
DATA          29      12      13      30
HIDATA        13      14
LINE10        23      19
LINE2         15      26
LINE4         17      24
LODATA        12      15
7 LAREIS
25 REFERENCEFS
33 RECORDS
0 U FLAGS
0 M FLAGS
1 N FLAGS

```

Figure C-2. Listing of Bubble Sort Program



Appendix D

Debugging Assembly Language Programs

There are two ways to debug and correct programs written in assembly language. One is by using the Debug program (see the *Program Execution and Checkout* manual); the other is by reading and interpreting the contents of memory through a memory dump (which can be obtained by using the Dump Edit utility also described in the *Program Execution and Checkout* manual).

DEBUG

This program is intended for use during development phases as a tool for program testing and error detection. Debug operates in interactive mode, maintaining a dialog with the operator's terminal. Debug makes visible all memory locations and addressable registers. Using Debug it is possible to modify the contents of either the memory locations or the addressable registers. Debug also makes it possible to perform memory searches and to display memory areas in both hexadecimal and ASCII notations.

See the *Program Execution and Checkout* manual for a detailed description of Debug.

DUMP EDIT

Dump Edit produces on the user file a logical and a physical dump of the contents of a dump file. Dump Edit generates, in an edited format, information such as the location and contents of hardware dedicated memory locations, the system control block, the group control blocks, and the work space blocks for each group control block.

See the *Program Execution and Checkout* manual for a detailed description of the Dump Edit.

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 readable data, it is apparent that the assembly language program contains no string constants in the locations illustrated. The hexadecimal digits could 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 LDR.
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., 1000000. Using the table, the binary data corresponds to the columns as follows: mmmirrr. Thus, mmm = 100, i = 0, and rrr = 000. In that block, the second operand is in the form of a location label.

7. Now you 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"), you 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)).

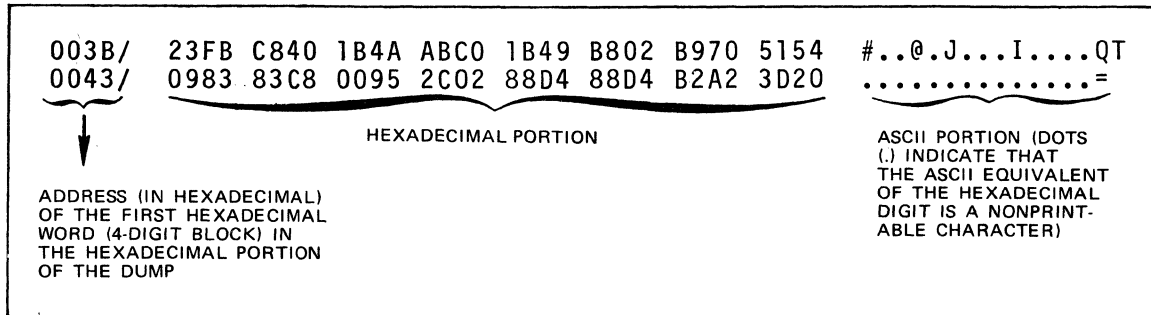


Figure D-1. ASCII/Hexadecimal Memory Dump

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 that you understand how to interpret the address syllable.

<i>Location</i>	<i>Instruction/Meaning</i>
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 >\$+3
0044	JMP *label
0045	Displacement between this location containing the effective address (see "Indirect P-Relative Addressing" in Section 5).
0046	LDV \$R2, 2
0047	DEC =\$R4
0048	DEC =\$R4
0049	LLH \$R3,*B2.\$R2
004A	CMV \$R3,X'20'

Appendix E

Notification Flags Issued By Assembler

SOURCE CODE ERROR FLAGS

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:

<i>Flag</i>	<i>Meaning</i>
A	Operand field format error
B	Byte allocation error
C	Numeric conversion error
D	Out of range short displacement
E	Illegal address expression
F	Illegal forward reference
H	Improper header
J	Function Error
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 ≥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

STATEMENT REFERENCE FLAGS

<i>Flag</i>	<i>Meaning</i>
K	Statement contains a reference to a common location
P	Statement contains a P-relative reference to an external symbol
T	Statement contains a reference to a temporary label
X	Statement contains a reference (other than P-relative) to an external symbol



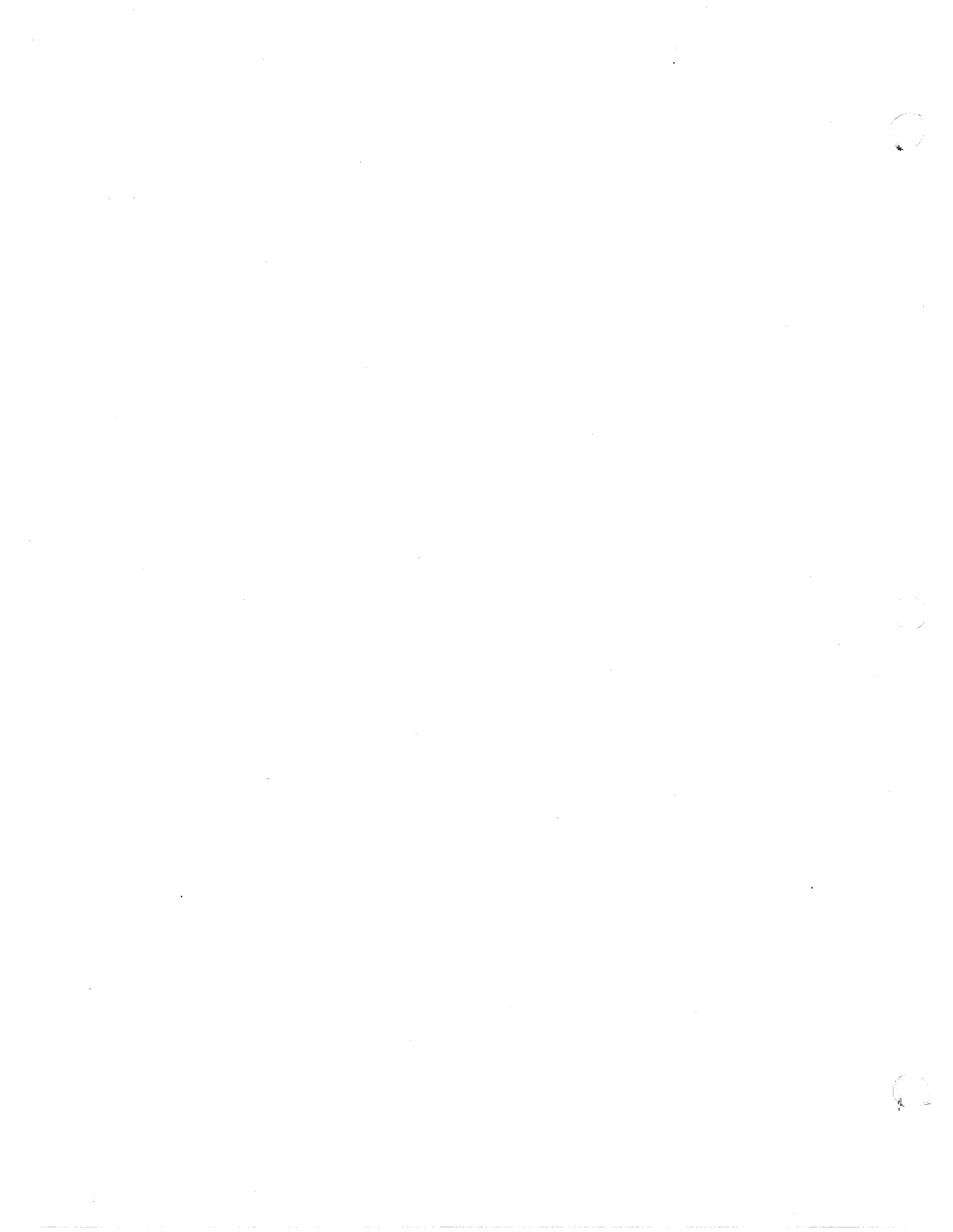
Appendix F

Source Code Error Notification By Macro Preprocessor

The Macro Preprocessor issues error flags for nonfatal errors in the source code. Each statement that contains a nonfatal error appears in the expanded source module as a comment statement with 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.

<i>Error Flag</i>	<i>Meaning</i>
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
Y	Directory or file specified cannot be found
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 Assembler and may not be redefined.

<i>Reserved Symbolic Name</i>	<i>Definition</i>
\$	Current location
\$A...\$Z	Temporary labels
\$AF	Address format
\$B1,\$B2,...\$B7	Base registers 1 through 7
\$COMM ¹	Name of unlabelled common
\$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
\$RZERO	Relocatable address zero
\$S1,\$S2,\$S3	Scientific registers 1 through 3
\$SW	External switch status

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 \$.

¹This symbol is only reserved if the program contains an unlabelled COMM statement.



TABLE H-1. COMMERCIAL INSTRUCTION SUMMARY

MNEMONIC	NAME	OP-CODE BITS 0-15	NO. OF OPCODE-RANDS	TYPE	INDICATORS					TRAPS						INSTRUCTION OPERATION	COMMENT	
					OV	TR	SF	G	L	UR	BE	IS	IC	DZ	TR			OV
DAD	Decimal Add	002C	2 (NN) a	N	X		X	X	X	X	X	X	X			X	[DD2]+[DD1] → [DD2]	
DSB	Decimal Subtract	002D	2 (NN)	N	X		X	X	X	X	X	X	X			X	[DD2]-[DD1] → [DD2]	
DML	Decimal Multiply	0029	2 (NN)	N	X		X	X	X	X	X	X	X			X	[DD2]x[DD1] → [DD2]	
DDV	Decimal Divide	002B	3 (NNN)	N	X		X	X	X	X	X	X	X	X		X	[DD2] → [DD3]; R [DD1] → [DD2]	
DCM	Decimal Compare	002F	2 (NN)	N				X	X	X	X	X	X				[DD1]::[DD2] → IND	Is DD1 G or L than DD2? Zero fill to the left supplied for smaller operand.
DMC	Decimal Move and Convert	0025	2 (NN)	N	X		X	X	X	X	X	X	X			X	[DD1] converted → [DD2]	Combinations are S → S, S → P, P → S, P → P
CBD	Convert Binary to Decimal	0027	2 (BN)	N	X		X			X	X	X				X	[DD1] converted → [DD2]	Binary operand is in 2's complement form.
CDB	Convert Decimal to Binary	002A	2 (NB)	N	X					X	X	X	X			X	[DD1] converted → [DD2]	Binary result is a 2's complement number
DSH	Decimal Shift	002E	1 (N)	N	X			X	X	X	X		X			X	Shift [DD1] Left "d" Shift [DD1] Right "d"	"d"= shift distance, OV indicator and Trap, valid only for shift left. Optional Rounding during a shift right.
ALR	Alphanumeric Move	0021	2 (AA)	A		X				X	X	X				X	[DD1] → [DD2]	Fill or do not fill to the right defined by DD2.
ACM	Alphanumeric Compare	0022	2 (AA)	A				X	X	X	X	X					[DD1]::[DD2] → IND	If DD1(L) ≠ DD2(L) then extended shorter operand as specified by DD2
MAT	Alphanumeric Move and Translate	0023	3 (AAA)	A		X				X	X	X				X	[DD1] Translate → [DD2] DD3 specifies 256 byte Translate Table	Fill character defined by DD2 is used directly
SRH	Alphanumeric Search	0028	3 (AAA)	A				X	X	X	X	X					[DD3] is searched using [DD1] as SL. Result → G, L indicators and displacement, SA number → [DD2]	
VRF	Alphanumeric Verify	0020	3 (AAA)	A				X	X	X	X	X					[DD3] is verified using [DD1] as VL. Result → G, L indicators and displacement → [DD2]	
DME	Decimal Move and Edit	0026	3 (NAA)	E						X	X	X	X				[DD1] Edited → [DD2] DD3 specifies Micro-ops	DD1=Decimal Descriptor DD2=Alphanumeric Descriptor DD3=Alphanumeric Descriptor.
AME	Alphanumeric Move and Edit	0024	3 (AAA)	E						X	X	X					[DD1] Edited → [DD2] DD3 specifies Micro-ops	DD1=DD2=DD3=Alphanumeric Descriptors

NOTE: In this table, A means alphanumeric operand, B means binary operand, and N means numeric operand.

TABLE H-1 (CONT). COMMERCIAL INSTRUCTION SUMMARY

MNEMONIC	NAME	OP-CODE	NO. OF OPE-RANDS	TYPE	INDICATORS					TRAPS						INSTRUCTION OPERATION	COMMENT	
					OV	TR	SF	G	L	UR	BE	IS	IC	DZ	TR			OV
CBOV	Branch on Overflow	130+xx	0	B						X	X						Branch to EA if CI(0) is set	
CBNOV	Branch If No Overflow	138+xx	0	B						X	X						Branch to EA if CI(0) is not set	
CBTR	Branch on Truncation	230+xx	0	B						X	X						Branch to EA if CI(1) is set	
CBNTR	Branch if No Truncation	238+xx	0	B						X	X						Branch to EA if CI(1) is set	
CBSF	Branch on Sign Fault	330+xx	0	B						X	X						Branch to EA if CI(2) is set	
CBNSF	Branch If No Sign Fault	338+xx	0	B						X	X						Branch to EA if CI(2) is not set	
CSYNC	Sync	430+xx	0	B						X	X						Wait for completion of previous instruction (if necessary) before going to the next instruction. No operation is performed.	
CSNCB	Sync and Branch	438+xx	0	B						X	X						Wait for completion of previous instruction (if necessary) then unconditionally branch to EA.	
CBE	Branch If Equal	536+xx	0	B						X	X						Branch to EA if CI(5) = CI(6) = 0	
CBNE	Branch If Not Equal	530+xx	0	B						X	X						Branch to EA if either CI(5) or CI(6) = 1	
CBG	Branch if Greater	630+xx	0	B						X	X						Branch to EA if CI(5) = 1	
CBLE	Branch if Less Than or Equal	63b+xx	0	B						X	X						Branch to EA if CI(5) = 0	
CBL	Branch If Less Than	730+xx	0	B						X	X						Branch to EA if CI(6) = 1	
CBGE	Branch If Greater Than or Equal	738+xx	0	B						X	X						Branch to EA if CI(6) = 0	

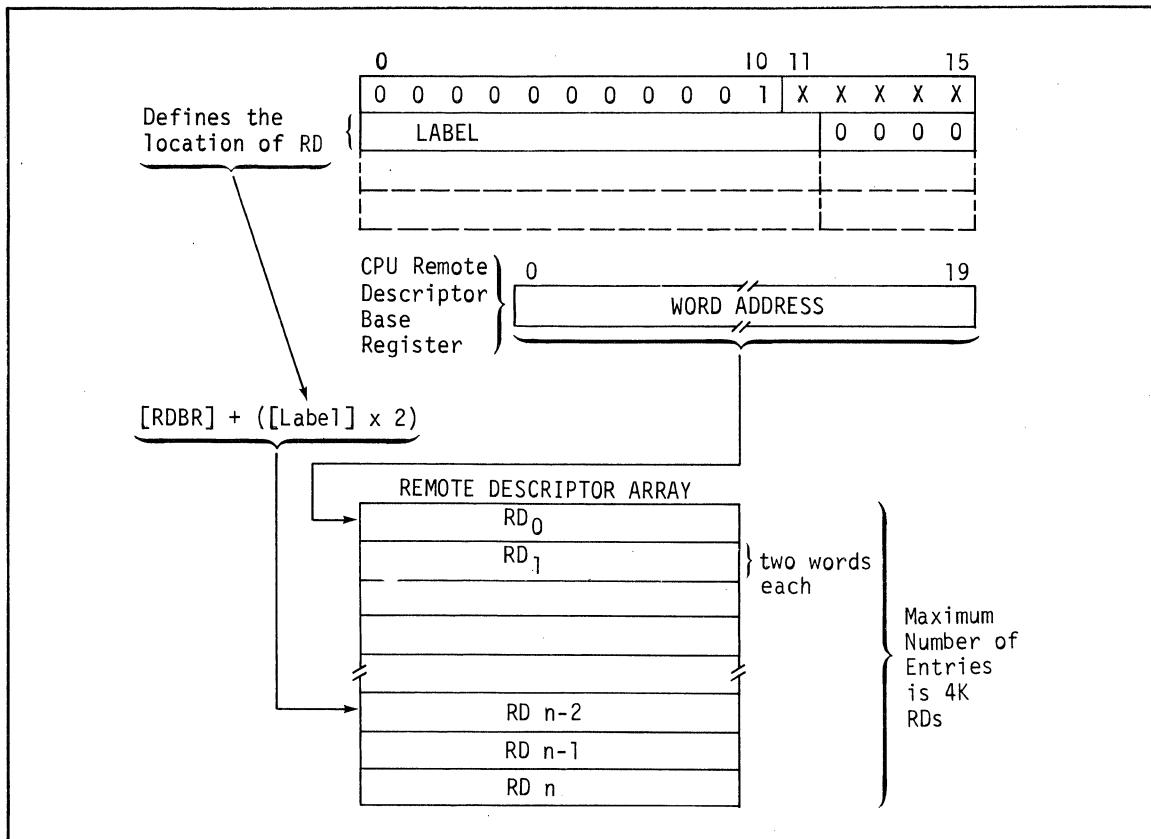


Figure H-2. Remote Descriptor Address Generation

INTERNAL FORMAT OF DATA DESCRIPTORS

DECIMAL DATA DESCRIPTORS

Decimal data descriptors can specify either unpacked decimal or packed decimal data. An unpacked decimal digit occupies one byte (8 bits); a packed decimal digit occupies 4 bits. A decimal data descriptor consists of two words as shown by Figure H-3. The contents of Word 2 is either a displacement or an immediate memory operand (IMO) and is specified by the address syllable (AS) or Word 1.

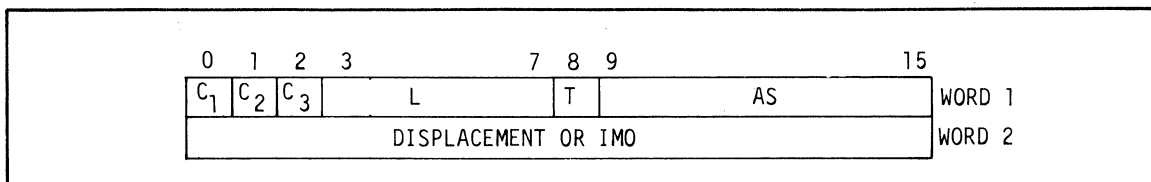


Figure H-3. Decimal Data Descriptor Format

UNPACKED DECIMALS

For unpacked decimals, the meaning of the Word 1 fields is as follows.

C1 specifies the byte offset

- When no indexing is specified, C1 specifies the offset within the addressed word.
 - If C1 is zero, the operand starts in the leftmost byte of the addressed word.
 - If C1 is one, the operand starts in the rightmost byte of the addressed word.
- When indexing is specified, the contents of C1 are added to the index value and the sum is used in calculating the effective address.

C2 and C3 specify the sign convention as shown below.

C ₂	C ₃	SIGN CONVENTION
0	0	Unsigned (assumed to be positive)
0	1	Trailing Overpunch
1	0	Leading Separate Sign
1	1	Trailing Separate Sign

L specifies the length of the operand in bytes. For unsigned or sign overpunched operands, all bytes contain digits. For separate signed operands, one byte contains the sign designation; the remaining bytes contain digits. If the contents of the L field is zero, the length is specified by bits 11 through 15 of an R register. The R register used depends on the data descriptor as follows:

- R4 for DD1
- R5 for DD2
- R6 for DD3

When the length is specified by an R register, bits 8 through 10 of the register must be zero; otherwise, the results are unspecified. The length in bytes can be from 1 through 31. An illegal specification (IS) trap is generated:

- If an operand has a length of zero, or
- If a separate signed operand has a length of one byte (i.e., if the operand consists of only a sign).

T specifies the type of decimal. The T bit must be zero for unpacked decimals.

AS specifies the address syllable prescribed by the instruction. See Figure H-6 for the address syllable format.

PACKED DECIMALS

For packed decimals, the meaning of Word 1 fields is as follows.

C1 and C2 specify the digit (4-bit) offset to the first digit.

- When no indexing is specified, C1 and C2 specify the offset within the addressed word as shown below.

C1	C2	No. of digits offset	Bit position within addressed word
0	0	0	0:3
0	1	1	4:7
1	0	2	8:11
1	1	3	12:15

- When indexing is specified, the offset value contained in C1 and C2 is added to the index value and the sum is used in calculating the effective address.

C3 specifies the sign convention.

- If C3 is zero, the operand is unsigned. (It is assumed to be positive.)
- If C3 is one, the operand has a trailing sign.

L specifies the length of the operand in 4-bit digits. The length is specified directly in the field, or indirectly in an R register in the same way as for string decimals.

T specifies the type of decimal. The T-bit must be one for packed decimals.

AS specifies the address syllable prescribed by the instruction. See Figure H-6 for the address syllable format.

ALPHANUMERIC DATA DESCRIPTOR

The format of an alphanumeric data descriptor is given in Figure H-4. The contents of Word 2 is either a displacement or an immediate memory operand (IMO) and is specified by the address syllable of Word 1.

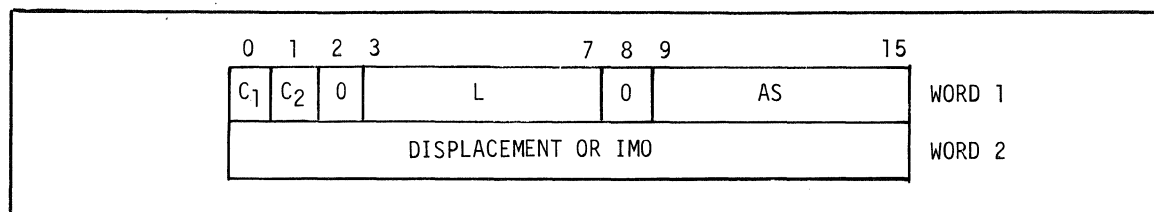


Figure H-4. Alphanumeric Data Descriptor Format

The meaning of the Word 1 fields is as follows.

C1 specifies the byte offset.

- When no indexing is specified, C1 specifies the offset within the addressed word.
 - If C1 is zero, the operand starts in the leftmost byte of the addressed word.
 - If C1 is one, the operand starts in the rightmost byte of the addressed word.
- When indexing is specified, the contents of C1 are added to the index value and the sum is used in calculating the effective address.

C2 is used for controlling fill operations. C2 is meaningful only when specified by DD2 of two instructions: alphanumeric move (ALR) and alphanumeric move and translate (MAT). See detailed descriptions of these instructions.

Bit 2 must be zero.

L specifies the length of the operand in bytes and the fill character when one is required. If the L field is not zero, its contents specify the length and the fill character is an ASCII blank (hexadecimal 20). If the L field is zero, the length is specified by bits 11 through 15 of an R register. The R register used depends on the data descriptor as follows:

R4 for DD1
R5 for DD2
R6 for DD3

When specified directly the maximum length is 31 bytes. When specified by an R register, the maximum length is 255 bytes. When the L field of a DD2 is zero, the fill character, if required, is specified by bits 0 through 7 of register R5.

Bit 8 must be zero; otherwise an illegal specification (IS) trap occurs.

AS specifies the address syllable prescribed by the instruction. See Figure H-6 for the address syllable format.

BINARY DATA DESCRIPTOR

The format of a binary data descriptor is given in Figure H-5. The contents of Word 2 is either a displacement or an immediate memory operand (IMO) and is specified by the address syllable of Word 1.

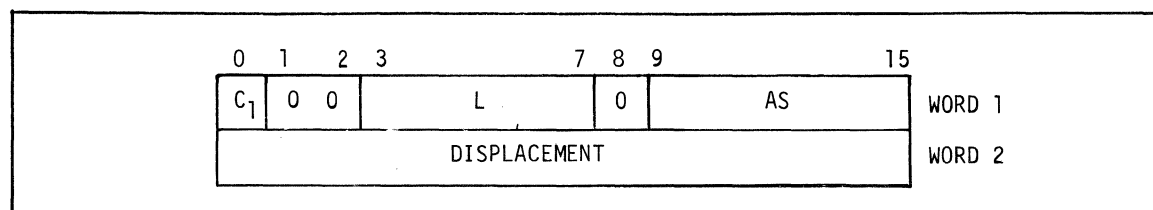


Figure H-5. Binary Data Descriptor Format

The meaning of the Word 1 fields is as follows.

C1 specifies the byte offset.

- When no indexing is specified, C1 specifies the offset within the addressed word.
 - If C1 is zero, the operand starts in the leftmost byte of the addressed word.
 - If C1 is one, the operand starts in the rightmost byte of the addressed word.
- When indexing is specified, the contents of C1 are added to the index value and the sum is used in calculating the effective address.

Bits 1 and 2 must be zero.

L specifies the binary precision which must be 16 or 32 bits.

- If L is not zero, it must be two (for a precision of 16 bits) or four (for a precision of 32) bits.
- If L is zero, the precision is specified by bits 11 through 15 of register R4 for DD1 and R5 for DD2. When these registers are used, bits 11 through 15 must contain two or four.

Bit 8 must be zero; otherwise, an illegal specification (IS) trap occurs.

AS specifies the address syllable prescribed by the instruction. See Figure H-6 for the address syllable format.

Note that a binary data descriptor is actually an alphanumeric data descriptor specifying no fill with a length, after possible escape to Rn, of either 2 bytes or 4 bytes.

ADDRESS SYLLABLE

The Commercial Processor address syllable occupies bits 9 through 15 of a data descriptor as shown by Figure H-6.

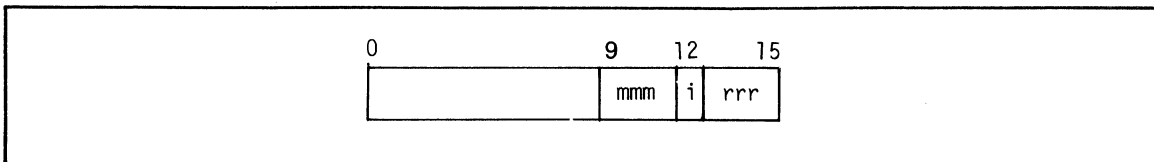


Figure H-6. Commercial Processor Address Syllable Format

Table H-2 lists the address expressions that can be specified by the Commercial Processor address syllable. In this table, n can be any; number from 1 through 7 and is equal to rrr, except when rrr = 000. The last four bits of an address syllable cannot be zeros. If the last four bits of the word are zeros, the word is a label that designates a remote descriptor.

Table H-2. COMMERCIAL PROCESSOR ADDRESS SYLLABLES

mmm	rrr = 000		rrr = 001 through 111	
	i = 0	i = 1	i = 0	i = 1
000	Remote description usage	location	\$Bn.value	*\$Bn.value
001		location.\$R1	\$Bn.value.\$R1	*\$Bn.value.\$R1
010		location.\$R2	\$Bn.value.\$R2	*\$Bn.value.\$R2
011		location.\$R3	\$Bn.value.\$R3	*\$Bn.value.\$R3
100		* location	\$Bn.value.\$R4	*\$Bn.value.\$R4
101		reserved	\$Bn.value.\$R5	*\$Bn.value.\$R5
110		reserved	\$Bn.value.\$R6	*\$Bn.value.\$R6
111		$\left. \begin{array}{l} = \text{string_constant} \\ = \text{decimal_constant} \end{array} \right\}^1$	\$Bn.value.\$R7	*\$Bn.value.\$R7

¹Not valid for use in second or third data descriptors

The Commercial Processor program (Figure H-7) provides a comprehensive example of the Commercial Processor instructions.

```

000001          TITLE      CIPPER,'770610' CIP HARDWARE TEST
000002          *
000003 0000 0043          CIPMSG  TEXT      Z'00','CIP HARDWARE TEST STARTING'
          0001 4950
          0002 2048
          0003 4152
          0004 4457
          0005 4152
          0006 4520
          0007 5445
          0008 5354
          0009 2053
          000A 5441
          000B 5254
          000C 494E
          000D 4700
000004 000E 0054          PASMSG  TFXT      Z'00','TEST '
          000F 4553
          0010 5420
000005 0011 3031          PASNO   DC        '01', ' '
          0012 2020
000006 0013 4641          PASSWD  DC        'PASSED'
          0014 5353
          0015 4544
000007          *
000008 0016 0054          FALMSG  TFXT      Z'00','TFST '
          0017 4553
          0018 5420
000009 0019 3031          FALNO   DC        '01', ' '
          001A 2020
000010 001B 4641          FAILWD  DC        'FAILED'
          001C 494C
          001D 4544
000011 001E 0010          FALPTR  DC        <FAILWD+2
000012 001F          MATTRL  RESV      32,' '
          2020          DC        ' ED '
000013 003F 2045
          0040 4420
000014 0041 4142          AB      DC        'AB'
          2020          MATEND  RFSV      3,' '
000015 0042          I1K     DC        0,0,0
000016 0045 0000
          0046 0000
          0047 0000
000017 0048 0000          PP1K   DC        0,0,0
          0049 0000
          004A 0000
000018 004B 0400          R1K    DC        1024
000019 004C 1024          P1K    DC        P'1024'
          004D 0000
000020 004E 2048          P2K    DC        P'2048'
          004F 0000
000021 0050 3072          P3K    DC        P'3072'
          0051 0000
000022 0052 6144          P6K    DC        P'6144'
          0053 0000
000023 0054 3430          N4K    DC        N'4095'
          0055 3935
000024 0056 3831          A8     DC        '8191'
          0057 3931
000025 0058 3030          N8K    DC        N'00008191'
          0059 3030
    
```

PROGRAMMER'S REFERENCE INFORMATION
FOR COMMERCIAL PROCESSOR OPERATION

H-8

CB07

Figure H-7. Commercial Processor Hardware Test Program

```

005A 3831
005B 3931
000026 005C F0D8 MOP TEXT SFF(0),MV7A(8)
000027 005D 3831 A8K DC 'A191****'
005E 3931
005F 2A2A
00A0 2A2A
000028 0061 2A2A AST8K DC '****A191'
0062 2A2A
0063 3831
0064 3931
000029 0065 3831 A8K0 DC 'A1910000'
0066 3931
0067 3030
0068 3030
000030 0069 2020 WK8K RFSV 4, ' '
000031 006D 3930 N9 DC N'9'
000032 *
000033 006E 4808 FFE7 DESTAB DC DESCA(A8,0,A),DESCA(WK8K,0,A);
0070 4808 FFF8
000034 DFSCA(MOP;
000035 0;
000036 0072 4208 FFE9 2);
000037 DFSCA(WK8K;
000038 0;
000039 0074 4808 FFF4 8);
000040 DFSCA(A8K0;
000041 0;
000042 0076 4808 FFEF 8)
000043 0078 0000 RETURN RFSV $AF,0
000044 0079 0001 TSTNO DC 1

```

Figure H-7 (cont). Commercial Processor Hardware Test Program

```

000045 /
000046 007A CBC0 FF93 PASS LAB $R4,PASMSG
000047 007C 7C00 LDV $R7,0
000048 007D 6C10 LDV $R6,16
000049 * $FROUT
000050 007E 0001 MCL
000051 007F 0803 DC Z'0803' WRITE TO ERROR OUT FILE
000052 0080 0F87 B >FAIL1
000053 *
000054 *
000055 0081 CBC0 FF94 FAIL LAB $R4,FALMSG
000056 0083 7C00 LDV $R7,0
000057 0084 6C10 LDV $R6,16
000058 * $FROUT
000059 0085 0001 MCL
000060 0086 0803 DC Z'0803' WRITE TO ERROR OUT FILE
000061 *
000062 0087 B8C0 FF89 FAIL1 LAB $R3,PASNO
000063 0089 ABC0 FF8F LAB $R2,FALNO
000064 008B 8AC0 FFED INC TSTNO
000065 008D F840 FFEB , LDR $R7,TSTNO
000066 008F 1C02 LDV $R1,2
000067 0090 8756 FAIL2 CI =$R6
000068 0091 F370 000A DIV $R7,=10
000069 0093 6E30 ADV $R6,Z'30'
000070 0094 F7DB STH $R6,$R3.-$R1
000071 0095 F792 STH $R6,$R2.$R1
000072 0096 79FA BNEZ $P7,>FAIL2
000073 *
000074 0097 8381 FAIL3 JMP $R1
    
```

Figure H-7 (cont). Commercial Processor Hardware Test Program

```

000075 /
000076 * OUTPUT STARTING MESSAGE
000077 XVAL ZFG0
000078 0098 0000 X START DC ZFG0
000079 0099 0FC0 FFDF STB $B5,RETURN
000080 009H C8C0 FF64 LAB $B4,CIPMSG
000081 009D 7C00 LDV $R7,0
000082 009E 6C1H LDV $R6,27
000083 *
000084 009F 0001 MCL
000085 00A0 0803 DC Z'0803' WRITE TO ERROR OUT FILE
000086 *
000087 * TEST 01 - ALR
000088 ALR DESCA(='PA',0,2,FILL);
000089 00A1 0021
000090 00A2 4278 5041
000091 00A4 4208 FF6F DESCA(PASSWD,0,2,FILL)
000092 00A6 4301 0001 $+2 DUMMY FOR WAIT
000093 00A8 93C0 FFD1 LNJ $B1,PASS
000094 00AA 8CF0 5041 5353 LDI ='PASS'
000095 00AD 8D40 FF65 SDI PASSWD
000096 *
000097 * TEST 02 - ACM
000098 00AF 1C03 LDV $R1,3
000099 00B0 0022 ACM DESCA(PASSWD.$R1,1,2,NOFILL);
000100 00B1 8218 FF61
000101 00B3 0248 FF6A DESCA(*FALPTR,0,2,NOFILL)
000102 00B5 5302 CRNF >TST202
000103 00B6 5384 CRE >TST201
000104 00B7 93C0 FFC9 TST202 LNJ $B1,FAIL
000105 00B9 0F83 B >TST3
000106 00BA 93C0 FFBF TST201 LNJ $B1,PASS
000107 *
000108 * TEST 03 - MAT
000109 TST3 LAB $B5,AR
000110 LAB $R6,MATEND
000111 LDV $R6,1
000112 00C1 F8C0 FF5D LAB $R7,MATBL
000113 00C3 7C02 LDV $R7,2
000114 00C4 0023 MAT DESCA($R5.0,0,2,NOFILL);
000115 00C5 0205 0000
000116 00C7 8266 0001 DESCA($R6.1.$R6,1,2,NOFILL);
000117 00C9 4277 FFFF DFSCA($R7.-1.$R7,0,2,FILL)
000118 00CB 4301 0001 CSYNC $+2 DUMMY FOR WAIT
000119 00CD E870 4544 LDR $R6,='ED'
000120 00CF E940 FF74 CMR $R6,MATEND+2
000121 00D1 0904 BF >TST301
000122 00D2 93C0 FFAF LNJ $R1,FAIL
000123 00D4 0F83 B >TST4
000124 00D5 93C0 FFA4 TST301 LNJ $B1,PASS
000125 *
000126 * TEST 04 - SRCH
000127 TST4 LDV $R6,3
000128 00D7 6C03 SRCH DESCA($R6.1.$R6,1,2,NOFILL);
000129 00D8 0028
000130 00D9 8266 0001
000131 00DB 0205 0000 DESCA($R5.0,0,2,NOFILL);
000132 00DD 4277 FFFF DESCA($R7.-1.$R7,0,2,FILL)
000133 00DF 4301 0001 CSYNC $+2 DUMMY FOR WAIT
000134 00F1 F840 FF5F LDR $R6,AR
000135 00F3 E970 4142 CMR $R6,='AR'
000136 00E5 0904 BF >TST401

```

PROGRAMMER'S REFERENCE INFORMATION
FOR COMMERCIAL PROCESSOR OPERATION H-11

CB07

Figure H-7 (cont). Commercial Processor Hardware Test Program


```

000131 00F6 93C0 FF9A          LNJ      $R1,FAIL
000132 00F8 0F83          B        >TST5
000133 00F9 93C0 FF90          TST401  LNJ      $R1,PASS
000134 *
000135 * TEST 05 -CRD, DMC, CDB
000136 00F8 0027          TSTS    CRD      DESC(B1K,2);
          00FC 0208 FF5F
000137 00FE 4608 FF56          DMC      DESC(U1K,0,6,LEADING)
000138 00F0 0025          DMC      DESC(U1K,0,6,LEADING);
          00F1 4608 FF53
000139 00F3 0608 FF54          DMC      DESC(P1K,0,6,UNSIGNED)
000140 00F5 002A          CDB      DESC(P1K,0,6,UNSIGNED);
          00F6 0608 FF51
000141 00F8 0208 FF52          DMC      DESC(B1K,2)
000142 00FA 4301 0001          CSYNC   $*2          DUMMY FOR WAIT
000143 00FC F840 FF4E          LDR     $R7,B1K
000144 00FE F970 0400          CMR     $R7,=1024
000145 0100 0904          BF      >TST501
000146 0101 93C0 FF7F          LNJ     $R1,FAIL
000147 0103 0F83          B        >TST6
000148 0104 93C0 FF75          TST501  LNJ     $R1,PASS
000149 *
000150 * TEST 06 - DAD
000151 0106 002C          TST6    DAD      DFSCP(P1K,0,5,T);
          0107 2588 FF44
000152 0109 2588 FF44          DCM      DFSCP(P2K,0,5,T);
000153 0108 002F          DCM      DFSCP(P2K,0,5,T);
          010C 2588 FF41
000154 010E 2588 FF41          DCM      DESC(P3K,0,5,T)
000155 0110 1388          CRNOV   >TST601
000156 0111 7304          CRL     >TST602
000157 0112 6303          CRG     >TST602
000158 0113 1302          CROV   >TST602
000159 0114 0F84          B        >TST601
000160 0115 93C0 FF68          TST602  LNJ     $R1,FAIL
000161 0117 0F83          B        >TST7
000162 0118 93C0 FF61          TST601  LNJ     $R1,PASS
000163 *
000164 * TEST 07 - DSB
000165 011A 002D          TST7    DSB      DFSCP(P2K,0,5,T);
          011B 2588 FF32
000166 011D 2588 FF32          DCM      DFSCP(P3K,0,5,T)
000167 011F 002F          DCM      DFSCP(=P'0',0,2,T);
          0120 22F8 0R00
000168 0122 2588 FF2D          DCM      DESC(P3K,0,5,T)
000169 0124 5384          CRE     >TST701
000170 0125 93C0 FF5B          LNJ     $R1,FAIL
000171 0127 0F83          B        >TST8
000172 0128 93C0 FF51          TST701  LNJ     $R1,PASS
000173 *
000174 * TEST 08 - DML
000175 012A 0029          TST8    DML      DFSCP(=P'2',0,2,T);
          012B 22F8 2R00
000176 012D 2588 FF20          DCM      DFSCP(P2K,0,5,T)
000177 012F 002F          DCM      DFSCP(P2K,0,5,T);
          0130 2588 FF1D
000178 0132 2588 FF1F          DCM      DESC(P6K,0,5,T)
000179 0134 5304          CRNE   >TST801
000180 0135 93C0 FF44          LNJ     $R1,PASS
000181 0137 0F83          B        >TST9

```

Figure H-7 (cont). Commercial Processor Hardware Test Program

PROGRAMMER'S REFERENCE INFORMATION
FOR COMMERCIAL PROCESSOR OPERATION H-12

CB07

```

000182 0138 93C0 FF48 TST801 LNJ $R1,FAIL
000183 *
000184 * TEST 09 - DDV
000185 013A 002B TST9 DDV DESC(P6K,0,5,T);
0138 2588 FF16
000186 013D 2588 FF14 DESC(P6K,0,5,T);
000187 013F 2588 FF0C DESC(P1K,0,5,T)
000188 0141 002F DCM DESC(P'1',0,2,T);
0142 22F8 1A00
000189 0144 2588 FF07 DESC(P1K,0,5,T)
000190 0146 1305 CROV >TST901
000191 0147 5304 CRNF >TST901
000192 0148 93C0 FF31 LNJ $R1,PASS
000193 014A 0FA3 B >TST10
000194 014B 93C0 FF35 TST901 LNJ $R1,FAIL
000195 *
000196 * TEST 10 - DSH
000197 014D 002E TST10 DSH DESCU(N4K,0,4,U),1
014E 0408 FF05
0150 0178 0001
000198 0152 002E DLS DESCU(N4K,0,4,U),1
0153 0408 FF00
0155 0178 0100
000199 0157 002E DRS DESCU(N4K,0,4,U),3,R
0158 0408 FEFR
015A 0178 8300
000200 015C 002F DCM DESCU(N9,0,1,U);
015D 0108 FF0F
000201 015F 0408 FFF4 DESCU(N4K,0,4,U)
000202 0161 53A4 CRE >TST101
000203 0162 93C0 FF1E LNJ $R1,FAIL
000204 0164 0FA3 B >TST11
000205 0165 93C0 FF14 TST101 LNJ $R1,PASS
000206 *
000207 * TEST 11 -DME
000208 0167 0026 TST11 DME DESCU(N8K,0,8,U);
0168 0808 FFEF
000209 016A 4808 FEFF DESC(AWK8K,0,8);
000210 016C 4208 FEEF DESC(MOP,0,2)
000211 016E 0022 ACM DESC(AWK8K,0,8);
016F 4808 FFF9
000212 0171 4808 FEEF DESC(ASTRK,0,8)
000213 0173 2303 CRTR >TST112
000214 0174 3302 CRSF >TST112
000215 0175 5384 CRE >TST111
000216 0176 93C0 FF0A TST112 LNJ $R1,FAIL
000217 0178 0FA3 B >TST12
000218 0179 93C0 FF00 TST111 LNJ $R1,PASS
000219 *
000220 * TEST 12 -AME
000221 017B 0024 TST12 AME DESC(A8,0,8);
017C 4808 FED9
000222 017E 4808 FEFA DESC(AWK8K,0,8);
000223 0180 4208 FEDR DFSCA(MOP,0,2)
000224 0182 0022 ACM DFSCA(WK8K,0,8);
0183 4808 FEE5
000225 0185 4808 FEDF DESC(A8K0,0,8)
000226 0187 5384 CRE >TST121
000227 0188 93C0 FEFA LNJ $R1,FAIL
000228 018A 0FA3 B >TST13
    
```

Figure H-7 (cont). Commercial Processor Hardware Test Program

```

000229
000230 018B 93C0 FFE0      *
000231                    TST121  LNJ      $B1,PASS
000232                    *
000233                    * TEST 13 - REMOTE DESCRIPTORS
000233 018D RBC0 FFE0      TST13   LAB      $R3,DESTAR
000234 018F CBC0 FEDE          LAB      $R4,DESTAR
000235 0191 000C          LRDR
000236 0192 0024          AME      0,1,2
          0193 0000
          0194 0010
          0195 0020
000237 0196 0022          ACM      3,4
          0197 0030
          0198 0040
000238 0199 53A5          CRE      >TST131
000239 019A 93C0 FFE6          LNJ      $R1,FAIL
000240 019C 83C8 FED8          JMP      *RETURN
000241                    *
000242 019E 93C0 FEDB      TST131  LNJ      $R1,PASS
000243 01A0 83C8 FED7          JMP      *RETURN
000244                    *
000245 01A2      009A          END      CIPPER,START
0001 ERR COUNT
0036R WORD SYMBOL TABLE
    
```

PROGRAMMER'S REFERENCE INFORMATION
FOR COMMERCIAL PROCESSOR OPERATION

H-14

CB07

Figure H-7 (cont). Commercial Processor Hardware Test Program

\$	****	90	114	127	142																
SAF	****	43																			
\$B1	****	74	91	101	103	118	120	131	133	146	148	160	162	170	172	180	182	192	194	203	
		205	216	218	227	230	239	242													
\$B2	****	63	71																		
\$B3	****	62	70	233																	
\$B4	****	46	55	80	234																
\$B5	****	79	106	111	125																
\$B6	****	107	112	124																	
\$B7	****	109	113	126																	
\$R1	****	66	70	71	96	97															
\$R6	****	48	57	67	69	70	71	82	108	112	115	116	123	124	128	129					
\$R7	****	47	56	65	68	72	81	110	113	126	143	144									
A8	24	33	221																		
N AAK	27																				
AAK0	29	40	225																		
AB	14	106	128																		
AST8K	28	212																			
B1K	18	136	141	143																	
CIPMSG	3	80																			
DESTAR	33	233	234																		
FAIL	55	101	118	131	146	160	170	182	194	203	216	227	239								
FAIL1	62	52																			
FAIL2	67	72																			
N FAIL3	74																				
FAILWD	10	11																			
FALMSG	8	55																			
FALNO	9	63																			
FALPTR	11	98																			
MATEND	15	107	116																		
MATTRL	12	109																			
MOP	26	34	210	223																	
N4K	23	197	198	199	201																
N8K	25	208																			
N9	31	200																			
P1K	19	151	187	189																	
P2K	20	152	153	165	176	177															
P3K	21	154	166	168																	
P6K	22	178	185	186																	
PASMSG	4	46																			
PASNO	5	62																			
PASS	46	91	103	120	133	148	162	172	180	192	205	218	230	242							
PASSWD	6	89	93	97																	
PP1K	17	139	140																		
RETURN	43	79	240	243																	
START	78	245																			
TST10	197	193																			
TST101	205	202																			
TST11	208	204																			
TST111	218	215																			
TST112	216	213	214																		
TST12	221	217																			
TST121	230	226																			
TST13	233	228																			
TST131	242	238																			
TST201	103	100																			
TST202	101	99																			
TST3	106	102																			
TST301	120	117																			
TST4	123	119																			

Figure H-7 (cont). Commercial Processor Hardware Test Program

TST401	133	130					
TST5	136	132					
TST501	14A	145					
TST6	151	147					
TST601	162	155	159				
TST602	160	156	157	158			
TST7	165	161					
TST701	172	169					
TST8	175	171					
TST801	182	179					
TST9	185	181					
TST901	194	190	191				
TSTNO	44	64	65				
UIK	16	137	13A				
WKRK	30	33	37	209	211	222	224
ZFGD	77	7A					

63 LABELS
 204 REFERENCES
 245 RECORDS
 0 H FLAGS
 0 M FLAGS
 2 N FLAGS

PROGRAMMER'S REFERENCE INFORMATION
 FOR COMMERCIAL PROCESSOR OPERATION H-16

CB07

Figure H-7 (cont). Commercial Processor Hardware Test Program

Appendix J

Programmer's Reference

Information For Queue Instructions

The queue instructions¹ allow easy maintenance of ordered lists of "frames." A frame contains a frame priority number, a next frame pointer, and an associated data structure. Each list is identified by a lock frame that contains a lock word and pointers to the head and tail of the list. See Figure J-1.

Four generic instructions are provided to enqueue or dequeue frames from the list:

- Queue on head (QOH)
- Queue on tail (QOT)
- Dequeue from head (DQH)
- Dequeue by address (DQA)

The lock word ensures that only one CPU accesses a particular queue at a time. Each queue instruction causes a fetch of the lock word with a Read-Modify-Write (RMW) cycle. If the low order bit of the lock word is set:

- The RMW cycle is completed without changing the lock.
- The carry bit of the indicator register is cleared.
- The next instruction is fetched.

If the low order bit of the lock word is cleared:

- The RMW cycle is completed and ones are written into the lock word.
- Execution of the queue or dequeue instruction is initiated.

Each queue or dequeue instruction causes a scan of the frames from the head toward the tail. When the conditions specified by the instruction are met, or the last frame is reached, the CPU:

- Links or unlinks the frame from the list
- Leaves the G- and L-bits of the indicator register in a known state
- Initiates another RMW cycle
- Writes zeros into the lock word
- Sets the carry bit of the indicator register to one
- Fetches the next instruction

If an interrupt occurs during a scan, the CPU:

- Stops the scan
- Initiates an RMW cycle
- Writes zeros into the lock word
- Clears the carry bit of the indicator register to zero
- Leaves the G- and L-bits of the indicator register undefined
- Sets the program counter to point to the queue or dequeue instruction being interrupted

After performing these actions, the CPU services the interrupt.

During the scan, the effective ring number is moved outward (i.e., becomes less privileged), if the frame being scanned is in a lower privilege ring. If the frame is in a higher privilege ring, standard protection procedures are carried out by the Memory Management Unit.

Software must build the lock frame of each list to be used. A list with no entries is a lock frame in which the first and last frame pointers point to LOCK. (See Figure J-1) The CPU will leave the lock frame in this condition when a frame is unlinked from a list having a single frame.

¹These instructions are only available on the 6/40 and 6/50 models.

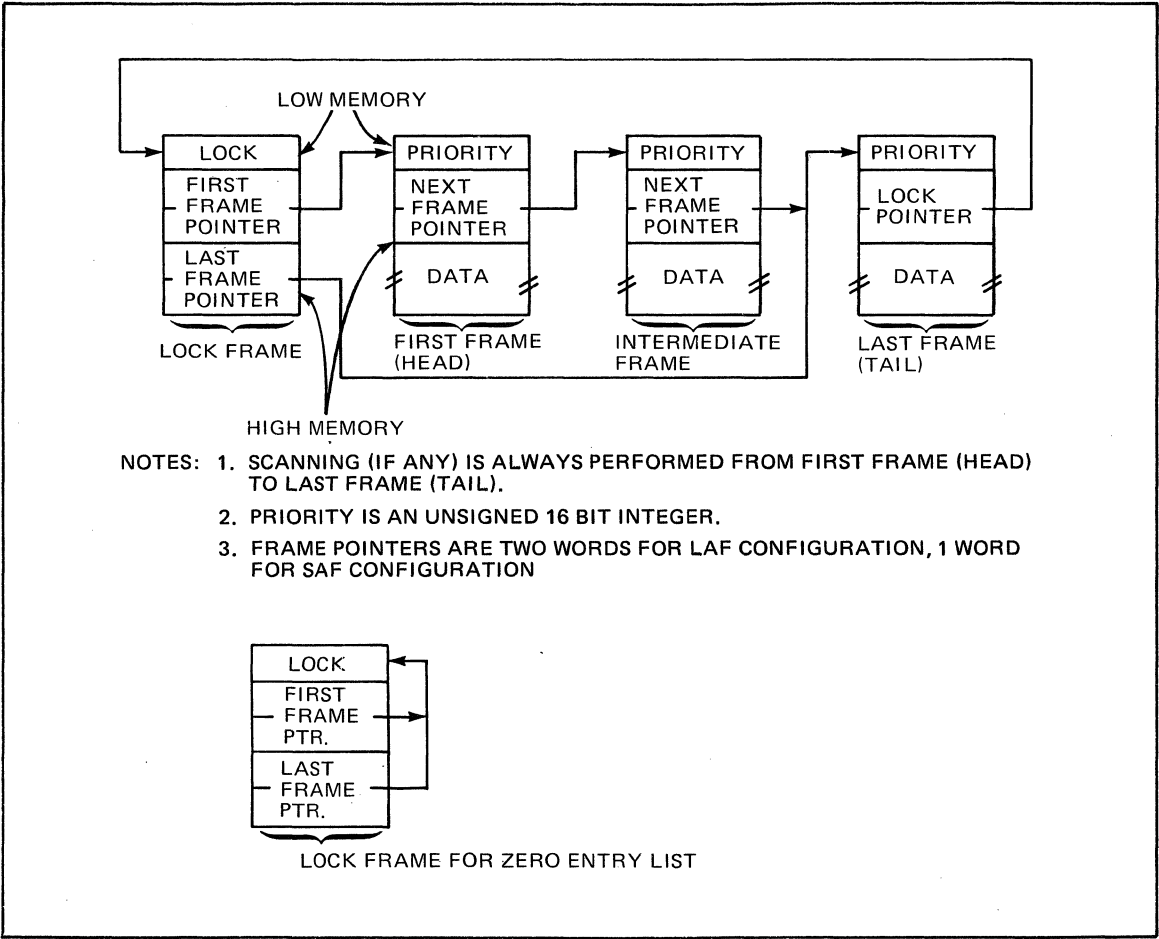


Figure J-1. Queue Management

The following DC statement will create an empty queue.

```

QUEUE DC 0;          LOCK WORD
      <QUEUE; FIRST FRAME POINTER
      <QUEUE LAST FRAME POINTER
  
```

Appendix K

Programmer's Reference Information For Stack Instructions

The Model 6/40 and 6/50 systems provide a single stack capability for each interrupt level. The stack address register, T, contains the address of the first word of the stack header. The stack header is shown in the diagram of the stack structure, Figure K-1.

Four generic instructions are provided for managing the stacks. These instructions are two-word instructions each of which has the same first word. During execution of the instructions, checks are made for stack overflow and stack underflow.

STACK FRAME

The stack header contains four entries, two of which must be null pointers as shown by Figure K-1. The number of words allocated to a stack, MW, is written by the software when the header is created. It is referenced, but not altered, by the hardware. The number of words currently consumed in a stack, CW, is written by the software when the header is created. Thereafter, the value of CW is maintained by the hardware.

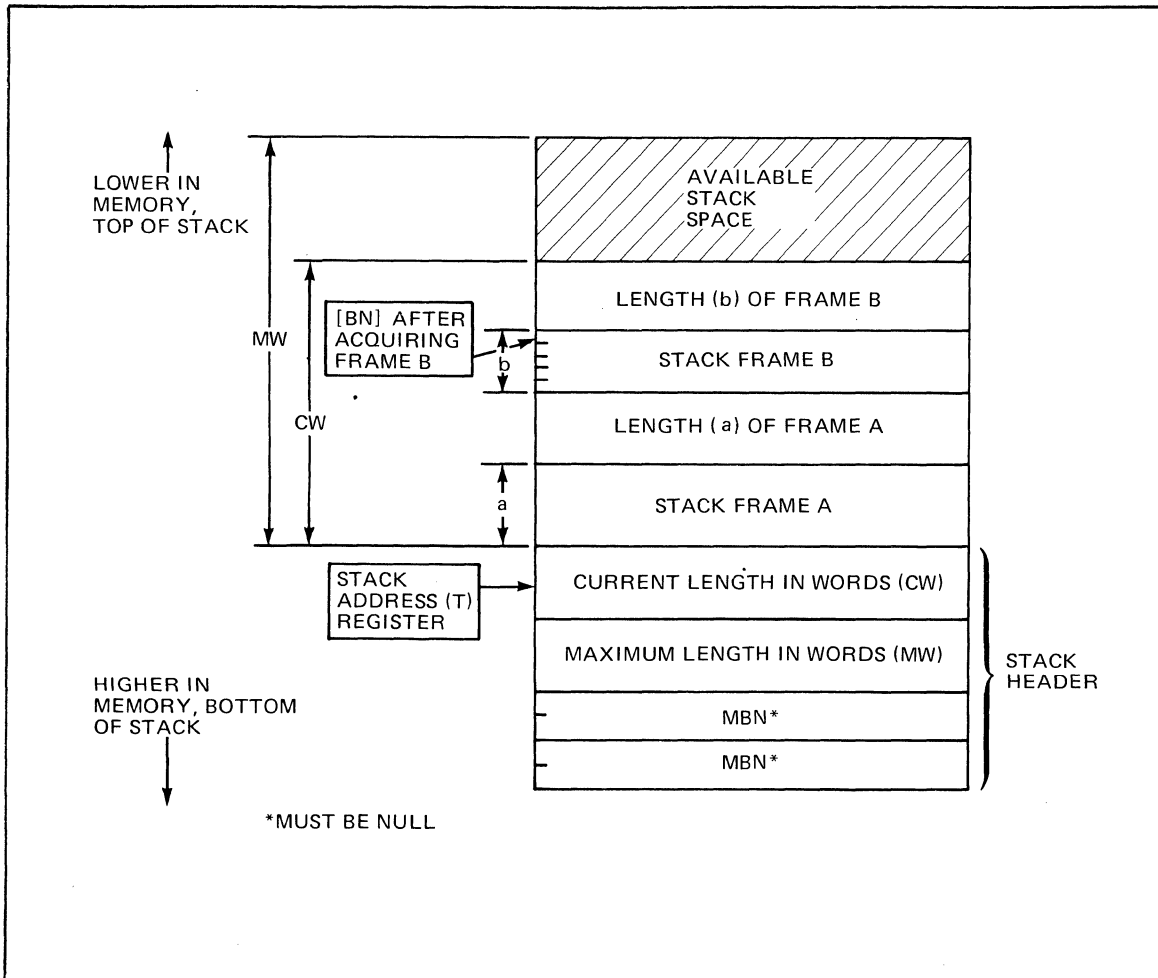
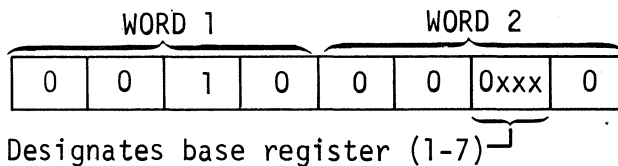


Figure K-1. Stack Structure

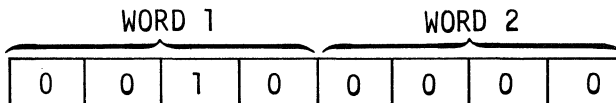
STACK INSTRUCTION FORMATS

LOAD STACK ADDRESS REGISTER (LDT)



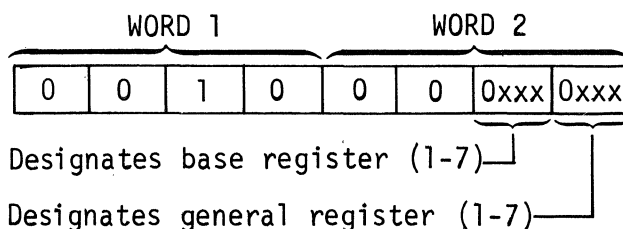
The LDT instruction loads the T register with the address contained in the specified base register.

STORE STACK ADDRESS REGISTER (STT)



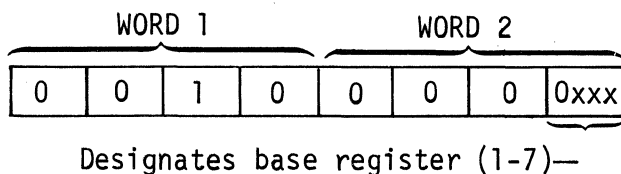
The STT instruction loads base register 7 with the address contained in the T register.

ACQUIRE STACK FRAME (ACQ)



The ACQ instruction loads the leftmost address of the newly acquired frame into the designated base register B_n, writes the length of the new frame into the previous location, and updates the current stack length accordingly. The designated general register R_n contains the number of words of stack space to be acquired. A trap to Trap 10 (stack overflow) occurs if the number of words to be acquired exceeds the available stack space.

RELINQUISH STACK FRAME (RLQ)



The RLQ instruction converts the top (most recently acquired) stack frame into available space by updating the current length (CW) in the stack header. A trap to Trap 9 (stack underflow) occurs if the stack is emptied; i.e., if CW becomes zero. If the stack is not emptied, the leftmost address of the new top frame is loaded into the designated base register.

- ACQUIRE
ACQUIRE STACK FRAME (ACQ), K-2
- ADDITION
HEXADECIMAL ADDITION, B-5
- ADDRESS
ADDRESS EXPRESSIONS, 2-17
ADDRESS REGISTERS, 1-4
ADDRESS SYLLABLE, H-7
ADDRESS SYLLABLES FOR CPU & SIP INSTRUCTIONS (TBL), A-9
COMMERCIAL PROCESSOR ADDRESS SYLLABLES (TBL), H-7
LOAD STACK ADDRESS REGISTER (LDT), K-2
REMOTE DESCRIPTOR ADDRESS GENERATION (FIG), H-4
SPECIALIZED ADDRESS EXPRESSION, 5-24
STORE STACK ADDRESS REGISTER (STT), K-2
VALID ADDRESS EXPRESSIONS, A-10
- ADDRESSING
ADDRESSING TECHNIQUES, 5-7
ADDRESSING TECHNIQUES FOR COMMERCIAL PROCESSOR INSTRUCTIONS, 6-5
B-RELATIVE ADDRESSING, 5-13
B-RELATIVE POP ADDRESSING, 5-21
B-RELATIVE PUSH ADDRESSING, 5-20
COMMERCIAL PROCESSOR B-RELATIVE ADDRESSING, 6-9
DIRECT B-RELATIVE ADDRESSING, 5-14
DIRECT B-RELATIVE PLUS DISPLACEMENT ADDRESSING, 5-16
DIRECT IMMEDIATE MEMORY ADDRESSING, 5-8
DIRECT P-RELATIVE ADDRESSING, 5-12, 6-6
IMMEDIATE MEMORY ADDRESSING (IMA), 5-8
IMMEDIATE OPERAND ADDRESSING, 5-10
IMMEDIATE OPERAND (IMO) ADDRESSING, 6-11
INDEXED ADDRESSING CONSIDERATIONS, 5-25
INDEXED ADDRESSING MODES (TBL), 5-25
INDEXED B-RELATIVE POP ADDRESSING, 5-22
INDEXED B-RELATIVE PUSH ADDRESSING, 5-21
INDEXED DIRECT B-RELATIVE ADDRESSING, 5-15
INDEXED DIRECT IMMEDIATE MEMORY ADDRESSING, 5-9
INDEXED DIRECT P-RELATIVE ADDRESSING, 6-7
INDEXED INDIRECT B-RELATIVE ADDRESSING, 5-16
INDEXED INDIRECT IMMEDIATE MEMORY ADDRESSING, 5-10
INDIRECT B-RELATIVE ADDRESSING, 5-15
- ADDRESSING (CONT)
INDIRECT B-RELATIVE PLUS DISPLACEMENT ADDRESSING, 5-18
INDIRECT IMMEDIATE MEMORY ADDRESSING, 5-9
INDIRECT P-RELATIVE ADDRESSING, 5-13, 6-8
INTERRUPT VECTOR ADDRESSING, 5-24
P-RELATIVE ADDRESSING, 5-12, 6-5
REGISTER ADDRESSING, 5-7
SHORT DISPLACEMENT ADDRESSING, 5-23
- ALPHANUMERIC
ALPHANUMERIC DATA DESCRIPTOR, 6-3, H-6
ALPHANUMERIC VALUE CONVENTIONS, 8-36
- APOSTROPHES
BALANCED APOSTROPHES, 8-36
- ARITHMETIC
ARITHMETIC CONSTANTS, 2-7
ARITHMETIC OPERATIONS, 5-1
DECIMAL ARITHMETIC INSTRUCTIONS, 6-2
- ASCII
ASCII STRING CONSTANTS, 2-5
- ASCII/HEXADECIMAL
ASCII/HEXADECIMAL MEMORY DUMP (FIG), D-2
- ASSEMBLER
ASSEMBLER, 3-2
ASSEMBLER CONTROL STATEMENTS, 4-1, 4-2
ASSEMBLER FUNCTIONS (FIG), 1-1
NOTIFICATION FLAGS ISSUED BY ASSEMBLER, E-1
- ASSEMBLER CONTROL STATEMENTS, LIST OF
ARGLST, 4-3
BORG, 4-4
BTEXT, 4-5
CALL, 4-6
CALL2, 4-7
CLST, 4-8
COMM, 4-9
CTRL, 4-10
DC, 4-11
DEFGEN, 4-12
EDEF, 4-13
END, 4-14
EQU, 4-15
FAIL, 4-16
IF, 4-17
LCOMM, 4-18
LIST, 4-19
ALIST, 4-19
NLST, 4-20
NULL, 4-21
ORG, 4-22
PTRAY, 4-23
RESV, 4-24

INDEX

ASSEMBLER CONTROL STATEMENTS, LIST OF
(CONT)

TEXT, 4-25
TITLE, 4-26
XDEF, 4-27
XLOC, 4-28
XVAL, 4-29

ASSEMBLY

ASSEMBLY LANGUAGE INSTRUCTION TYPES,
5-4
ASSEMBLY LANGUAGE INSTRUCTIONS, 5-1,
5-27
ASSEMBLY LANGUAGE INTERNAL FORMATS,
A-4
ASSEMBLY LANGUAGE SOURCE STATEMENT
FORMATS, 3-1
ASSEMBLY LANGUAGES, 1-1
CONDITIONAL ASSEMBLY CONTROL
STATEMENTS, 4-2
DEBUGGING ASSEMBLY LANGUAGE
PROGRAMS, D-1
ELEMENTS OF ASSEMBLY LANGUAGE, 2-1

ASSEMBLY CONTROLLING

ASSEMBLY CONTROLLING STATEMENTS,
4-1

ASSEMBLY LANGUAGE INSTRUCTIONS

ACQ, 5-27
ADD, 5-28
ADV, 5-29
AID, 5-30
AND, 5-31
ANH, 5-32
ASD, 5-33
B, 5-34
BAG, 5-35
BAGE, 5-36
BAL, 5-37
BALE, 5-38
BBF, 5-39
BBT, 5-40
BCF, 5-41
BCT, 5-42
BDEC, 5-43
BE, 5-44
BEVN, 5-45
BEZ, 5-46
BG, 5-47
BGE, 5-48
BGEZ, 5-49
BGZ, 5-50
BINC, 5-51
BIOF, 5-52
BIOT, 5-53
BL, 5-54
BLE, 5-55
BLEZ, 5-56
BLZ, 5-57
BNE, 5-58
BNEZ, 5-59
BNOV, 5-60
BODD, 5-61
BOV, 5-62
BRK, 5-63

ASSEMBLY LANGUAGE INSTRUCTIONS (CONT)

BSE, 5-64
BSU, 5-65
CAD, 5-66
CL, 5-67
CLH, 5-68
CMB, 5-69
CMH, 5-70
CMN, 5-71
CMR, 5-72
CMV, 5-73
CMZ, 5-74
CNFG, 5-75
CPL, 5-76
DAL, 5-77
DAR, 5-78
DCL, 5-79
DCR, 5-80
DEC, 5-81
DIV, 5-82
DOL, 5-83
DOR, 5-84
DQA, 5-85
DQH, 5-86
ENT, 5-87
HLT, 5-88
INC, 5-89
IO, 5-90
IOH, 5-92
IOLD, 5-93
JMP, 5-94
LAB, 5-95
LB, 5-96
LBC, 5-97
LBF, 5-98
LBS, 5-99
LBT, 5-100
LDB, 5-101
LDH, 5-102
LDI, 5-103
LDR, 5-104
LDT, 5-105
LDV, 5-106
LEV, 5-107
LLH, 5-109
LNJ, 5-110
LRDB, 5-111
MCL, 5-112
MLV, 5-113
MMM, 5-114
MTM, 5-115
MUL, 5-116
NEG, 5-117
NOP, 5-118
OR, 5-119
ORH, 5-120
QOH, 5-121
QOT, 5-122
RLQ, 5-123
RSTR, 5-124
RTCF, 5-125
RTCN, 5-126
RTT, 5-127
SAL, 5-128
SAR, 5-129
SAVE, 5-130

INDEX

ASSEMBLY LANGUAGE INSTRUCTIONS (CONT)

SCL, 5-131
 SCR, 5-132
 SDI, 5-133
 SID, 5-134
 SOL, 5-135
 SOR, 5-136
 SRDB, 5-137
 SRM, 5-138
 STB, 5-139
 STH, 5-140
 STM, 5-141
 STR, 5-142
 STS, 5-143
 STT, 5-144
 SUB, 5-145
 SWB, 5-146
 SWR, 5-147
 VLD, 5-148
 WDTF, 5-150
 WDTN, 5-151
 XOH, 5-152
 XOR, 5-153

ATTRIBUTE, LENGTH
 LENGTH ATTRIBUTE MACRO FUNCTION,
 B-23

BALANCED
 BALANCED APOSTROPHES, 8-36
 BALANCED PARENTHESES, 8-36

BASE
 BASE ADDRESS (BN) REGISTERS, 1-4
 REMOTE DESCRIPTOR BASE REGISTER
 (RDBR), 1-4

BI INSTRUCTIONS
 BRANCH-ON-INDICATOR (BI)
 INSTRUCTIONS, 5-4

BINARY
 BINARY DATA DESCRIPTOR, 6-5, H-6
 BINARY INTEGER CONSTANTS, 2-7
 COMPARISON BINARY, DECIMAL, AND
 HEXADECIMAL SYMBOLS (TBL), B-1

BIT
 BIT STRING CONSTANTS, 2-6

BIT/BYTE
 BIT/BYTE MANIPULATING INSTRUCTIONS,
 5-26

B-REGISTER
 B-REGISTER INSTRUCTION IN LAF
 CONFIGURATION, 5-26

B-RELATIVE
 B-RELATIVE ADDRESSING, 5-13
 B-RELATIVE POP ADDRESSING, 5-21
 B-RELATIVE PUSH ADDRESSING, 5-20
 COMMERCIAL PROCESSOR B-RELATIVE
 ADDRESSING, 6-9
 DIRECT B-RELATIVE ADDRESSING, 5-14

B-RELATIVE (CONT)

DIRECT B-RELATIVE PLUS DISPLACEMENT
 ADDRESSING, 5-16
 INDEXED B-RELATIVE POP ADDRESSING,
 5-22
 INDEXED B-RELATIVE PUSH ADDRESSING,
 5-21
 INDEXED DIRECT B-RELATIVE ADDRESSING,
 5-15
 INDEXED INDIRECT B-RELATIVE
 ADDRESSING, 5-16
 INDIRECT B-RELATIVE ADDRESSING, 5-15
 INDIRECT B-RELATIVE PLUS
 DISPLACEMENT ADDRESSING, 5-18

BLANK
 INSERT BLANK ON SUPPRESS (INSB)
 MICRO OPERATION, 6-16

BOOLEAN
 BOOLEAN OPERATIONS, 5-1

BR INSTRUCTIONS
 BRANCH-ON-REGISTER (BR) INSTRUCTIONS,
 5-4

BRANCH
 BRANCH INSTRUCTIONS, 6-2
 BRANCH OPERATIONS, 5-2

BRANCH-ON-INDICATOR
 BRANCH-ON-INDICATOR (BI)
 INSTRUCTIONS, 5-4

BRANCH-ON-REGISTER
 BRANCH-ON-REGISTER (BR)
 INSTRUCTIONS, 5-4

BUBBLE SORT LISTING
 LISTING OF BUBBLE SORT PROGRAM (FIG),
 C-3

CALL STATEMENT
 CALL, 4-6

CALLING
 CALLING EXTERNAL PROCEDURES, 3-2
 CALLING SYSTEM SERVICES, 3-2

CALLS, MACRO
 MACRO CALLS, 8-11
 RECURSIVE MACRO CALLS, 8-13

CHARACTER
 CHARACTER INSERTION FOR MFLS MICRO
 OPERATION (TBL), 6-18
 CHARACTER STRING INSTRUCTIONS, 6-2
 IGNORE SOURCE CHARACTER (IGN) MICRO
 OPERATION, 6-16
 INSERT CHARACTER ON NEGATIVE (INSN)
 MICRO OPERATION, 6-16
 INSERT CHARACTER ON POSITIVE (INSP)
 MICRO OPERATION, 6-17
 INSERT MULTIPLE CHARACTERS (INSM)
 MICRO OPERATION, 6-16

INDEX

CHARACTER (CONT)
 MOVE SOURCE CHARACTER (MVC) MICRO OPERATION, 6-18
 TRAP 27 ILLEGAL CHARACTER (IC), 6-22

CODE
 CODE FOR REPLACING EDIT ENTRIES (TBL), 6-15

CODES, MNEMONIC
 MNEMONIC CODES, 2-1

COMMAS
 COMMAS AND SEMI-COLONS, 8-37

COMMERCIAL
 ADDRESSING FOR COMMERCIAL PROCESSOR INSTRUCTIONS, 6-5
 COMMERCIAL INSTRUCTION CATEGORIES, 6-2
 COMMERCIAL INSTRUCTION SUMMARY (TBL), H-2
 COMMERCIAL INSTRUCTIONS, 6-1
 COMMERCIAL PROCESSOR ADDRESS SYLLABLE FORMAT (FIG), H-7
 COMMERCIAL PROCESSOR B-RELATIVE ADDRESSING, 6-9
 COMMERCIAL PROCESSOR DATA DESCRIPTORS, 6-3
 COMMERCIAL PROCESSOR HARDWARE TEST PROGRAM (FIG), H-8
 COMMERCIAL PROCESSOR IMO ADDRESSING (FIG), 6-12
 COMMERCIAL PROCESSOR INDICATOR REGISTER, 1-10
 COMMERCIAL PROCESSOR INSTRUCTION FORMAT, 6-3
 COMMERCIAL PROCESSOR REGISTER, 1-9
 COMMERCIAL PROCESSOR TRAPS, 6-20
 COMMERCIAL PROCESSOR (CP) PROGRAMMING, 6-1
 EXECUTION DETAILS FOR COMMERCIAL INSTRUCTIONS, 6-22
 INTERNAL FORMATS OF COMMERCIAL PROCESSOR INSTRUCTIONS, H-1
 SOFTWARE SIMULATION COMMERCIAL PROCESSOR, 1-10

COMMERCIAL INSTRUCTIONS
 ACM, 6-24
 ALR, 6-25
 AME, 6-26
 CBD, 6-27
 CBE, 6-28
 CBG, 6-29
 CBGE, 6-30
 CBL, 6-31
 CBLE, 6-32
 CBNE, 6-33
 CBNOV, 6-34
 CBNSF, 6-35
 CBNTR, 6-36
 CBOV, 6-37
 CBSF, 6-38
 CBTR, 6-39

COMMERCIAL INSTRUCTIONS (CONT)
 CDB, 6-40
 CSNCB, 6-41
 CSYNC, 6-42
 DAD, 6-43
 DCM, 6-44
 DDV, 6-45
 DLS, 6-46
 DMC, 6-47
 DME, 6-48
 DML, 6-52
 DRS, 6-53
 DSB, 6-54
 DSH, 6-55
 MAT, 6-57
 SRCH, 6-58
 VRF, 6-62

COMMON
 COMMON LOCATION EXPRESSIONS, 2-16

COMPARE
 COMPARE OPERATIONS, 5-2

CONDITIONAL STATEMENTS
 CONDITIONAL ASSEMBLY CONTROL STATEMENTS, 4-2
 CONDITIONAL MACRO CONTROL STATEMENTS, 8-17

CONSTANTS
 ARITHMETIC CONSTANTS, 2-7
 ASCII STRING CONSTANTS, 2-5
 BINARY INTEGER CONSTANTS, 2-7
 BIT STRING CONSTANTS, 2-6
 CONSTANTS, 2-4
 DECIMAL INTEGER CONSTANTS, 2-8
 FIXED-POINT CONSTANTS, 2-9
 FLOATING-POINT CONSTANTS, 2-10
 HEXADECIMAL STRING CONSTANTS, 2-5
 RULES OF TRUNCATION/PADDING STRING CONSTANTS (TBL), 2-6
 STRING CONSTANTS, 2-5
 TRUNCATION/PADDING STRING CONSTANTS, 2-6

CONTROL STATEMENTS
 ASSEMBLER CONTROL STATEMENTS, 4-1, 4-2
 CONDITIONAL ASSEMBLY CONTROL STATEMENTS, 4-2
 CONDITIONAL MACRO CONTROL STATEMENTS, 8-17
 CONTROL OPERATIONS, 5-2
 ENDM MACRO CONTROL STATEMENT, 8-3
 FAIL MACRO CONTROL STATEMENT, 8-17
 GOTO MACRO CONTROL STATEMENT, 8-18
 IF MACRO CONTROL STATEMENT, 8-19
 INCLUDE MACRO CONTROL STATEMENT, 8-9
 LIBM MACRO CONTROL STATEMENT, 8-7
 MAC MACRO CONTROL STATEMENTS, 8-2, 8-4
 NULL MACRO CONTROL STATEMENT, 8-22
 SETA MACRO CONTROL STATEMENT, 8-15
 SETN MACRO CONTROL STATEMENT, 8-16

INDEX

- CONVERSION
 - DECIMAL-TO-HEXADECIMAL CONVERSION, B-2
 - HEXADECIMAL CONVERSION MACRO FUNCTION, B-25
 - HEXADECIMAL-TO-ASCII CONVERSION, B-4
 - HEXADECIMAL-TO-DECIMAL CONVERSION, B-2
 - RADIX AND MODE CONVERSION INSTRUCTIONS, 6-2
- COUNTER, PROGRAM
 - PROGRAM COUNTER (P-REGISTER), 1-4
- CROSS-REFERENCE
 - CROSS-REFERENCE LISTING, 3-3
- DATA
 - ALPHANUMERIC DATA DESCRIPTORS, 6-3, H-6
 - BINARY DATA DESCRIPTOR, 6-5, H-6
 - COMMERCIAL PROCESSOR DATA DESCRIPTORS, 6-3
 - DECIMAL DATA DESCRIPTORS, H-4
 - FLOATING-POINT DATA, 1-4
 - INTERNAL FORMAT OF DATA DESCRIPTORS, H-4
 - LEVEL 6 DATA REPRESENTATION, 1-1
 - PACKED-DECIMAL DATA DESCRIPTOR, 6-4
 - SIGNED INTEGER DATA, 1-2
 - UNPACKED-DECIMAL DATA DESCRIPTOR, 6-4
 - UNSIGNED DATA, 1-3
- DATA-DEFINING STATEMENTS
 - DATA-DEFINING STATEMENTS, 4-1
- DC INSTRUCTION
 - DC, 4-11
- DEBUGGING
 - DEBUGGING ASSEMBLY LANGUAGE PROGRAMS, D-1
- DECIMAL
 - BINARY INTEGER CONSTANTS IN DECIMAL NOTATION, 2-8
 - COMPARISON OF BINARY, DECIMAL, AND HEXADECIMAL SYMBOLS (TBL), B-1
 - DECIMAL ARITHMETIC INSTRUCTIONS, 6-2
 - DECIMAL DATA DESCRIPTORS, H-4
 - DECIMAL INTEGER CONSTANTS, 2-8
 - PACKED DECIMAL INTEGERS, 2-8
 - UNPACKED DECIMAL INTEGER, 2-8
- DECIMAL-TO-HEXADECIMAL
 - DECIMAL-TO-HEXADECIMAL CONVERSION, B-2
- DECIMALS, PACKED AND UNPACKED
 - PACKED DECIMALS, H-5
 - UNPACKED DECIMALS, H-4
- DEFINED LABELS
 - USER-DEFINED LABELS, 2-2
- DISPLACEMENT
 - DIRECT B-RELATIVE PLUS DISPLACEMENT ADDRESSING, 5-16
 - INDIRECT B-RELATIVE PLUS DISPLACEMENT ADDRESSING, 5-18
 - SHORT DISPLACEMENT ADDRESSING, 5-23
- DIVISION, HEXADECIMAL
 - HEXADECIMAL DIVISION, B-6
- DOUBLE OPERAND INSTRUCTIONS
 - DOUBLE OPERAND (DO) INSTRUCTIONS, 5-4
- DUMP
 - ASCII/HEXADECIMAL MEMORY DUMP (FIG), D-2
 - DUMP EDIT, D-1
 - READING AND INTERPRETING MEMORY DUMPS, D-1
- EDEF STATEMENT
 - EDEF, 4-13
- EDIT
 - CHANGE EDIT INSERTION TABLE (CHT) MICRO OPERATION, 6-14
 - CODE FOR REPLACING EDIT ENTRIES (TBL), 6-15
 - DUMP EDIT, D-1
 - EDIT FLAGS, 6-14
 - EDIT INSERTION TABLE, 6-13
 - EDIT INSTRUCTIONS, 6-2
 - MICRO EDIT FUNCTIONS, 6-12
 - MICRO OPERATIONS FOR EDIT INSTRUCTIONS (TBL), 6-13
 - SET EDIT FLAGS (SEF) MICRO OPERATION, 6-19
- END
 - END FLOATING SUPPRESSION (ENF) MICRO OPERATION, 6-15
- ENDM
 - ENDM MACRO CONTROL STATEMENT, 8-3
- ERROR
 - SOURCE CODE ERROR FLAGS, E-1
 - SOURCE CODE ERROR NOTIFICATION BY MACRO PREPROCESSOR, 8-1
- EXPRESSIONS
 - ADDRESS EXPRESSIONS, 2-17
 - COMMON LOCATION EXPRESSIONS, 2-16
 - EVALUATING EXPRESSIONS, 2-13
 - EXPRESSIONS, 2-11
 - EXTERNAL LOCATION EXPRESSIONS, 2-15
 - EXTERNAL VALUE EXPRESSIONS, 2-14
 - INTERNAL LOCATION EXPRESSIONS, 2-15
 - INTERNAL VALUE EXPRESSIONS, 2-13
 - LOCATION AND VALUE EXPRESSIONS, 2-13
 - LOCATION EXPRESSIONS, 2-15
 - SPECIALIZED ADDRESS EXPRESSIONS (FIG), 5-24
 - VALID ADDRESS EXPRESSIONS, A-10
 - VALUE EXPRESSIONS, 2-13

INDEX

EXTERNAL PROCEDURES
 CALLING EXTERNAL PROCEDURES, 3-2

FAIL
 FAIL MACRO CONTROL STATEMENT, 8-17

FILE
 ALTERNATE METHOD OF HANDLING INPUT/
 OUTPUT + FILE MANIPULATION, 3-2

FIXED-POINT CONSTANTS
 FIXED-POINT CONSTANTS, 2-9

FLAGS
 EDIT FLAGS, 6-14
 EDIT FLAGS FOR MICRO OPERATIONS
 (TBL), 6-14
 NOTIFICATION FLAGS ISSUED BY
 ASSEMBLER, E-1
 SET EDIT FLAGS (SEF) MICRO
 OPERATION, 6-19
 SOURCE CODE ERROR FLAGS, E-1
 STATEMENT REFERENCE FLAGS, E-1

FLOAT
 MOVE WITH FLOAT SIGN INSERTION
 (MFLS) MICRO OPERATION, 6-17

FLOATING-POINT
 FLOATING-POINT CONSTANTS, 2-10
 FLOATING-POINT DATA, 1-4

FORMAT
 ALPHANUMERIC DATA DESCRIPTOR FORMAT
 (FIG), H-6
 BINARY DATA DESCRIPTOR FORMAT (FIG),
 H-6
 COMMERCIAL PROCESSOR ADDRESS
 SYLLABLE FORMAT (FIG), H-7
 COMMERCIAL PROCESSOR INSTRUCTION
 FORMAT, 6-3
 DECIMAL DATA DESCRIPTOR FORMAT
 (FIG), H-4
 FORMAT OF MACRO FUNCTIONS, 8-23
 INTERNAL FORMAT OF DATA
 DESCRIPTORS, H-4

FORMATS
 ASSEMBLY LANGUAGE INTERNAL FORMATS,
 A-4
 ASSEMBLY LANGUAGE SOURCE STATEMENT
 FORMATS, 3-1
 INTERNAL FORMATS OF COMMERCIAL
 PROCESSOR INSTRUCTIONS, H-1
 SHIFT INSTRUCTION FORMATS (FIG),
 6-51
 STACK INSTRUCTION FORMATS, K-2

FRAME, STACK
 ACQUIRE STACK FRAME (ACQ), K-2
 RELINQUISH STACK FRAME (RLQ), K-2
 STACK FRAME, K-1

FUNCTION, MACRO
 HEXADECIMAL CONVERSION MACRO
 FUNCTION, 8-25

FUNCTION, MACRO (CONT)
 INDEX MACRO FUNCTION, 8-26
 LENGTH ATTRIBUTE MACRO FUNCTION,
 8-23
 SEARCH MACRO FUNCTION, 8-27
 SUBSTRING MACRO FUNCTION, 8-28
 TRANSLATE MACRO FUNCTION, 8-29
 VECTOR ORIENTATION MACRO FUNCTION,
 8-30
 VERIFY MACRO FUNCTION, 8-31

FUNCTIONS
 ASSEMBLER FUNCTIONS (FIG), 1-1
 MACRO FUNCTIONS, 8-23
 MICRO EDIT FUNCTIONS, 6-12

GENERAL REGISTERS
 GENERAL (RN) REGISTERS, 1-5

GENERATION, ADDRESS
 REMOTE DESCRIPTOR ADDRESS
 GENERATION (FIG), H-4

GENERIC INSTRUCTIONS
 GENERIC (GE) INSTRUCTIONS, 5-5

GOTO
 GOTO MACRO CONTROL STATEMENT, 8-18

HARDWARE
 COMMERCIAL PROCESSOR HARDWARE TEST
 PROGRAM (FIG), H-8
 HARDWARE REGISTERS, 1-4
 SUMMARY OF HARDWARE REGISTERS, A-1

HEXADECIMAL
 BINARY INTEGER CONSTANTS IN
 HEXADECIMAL NOTATION, 2-8
 COMPARISON OF BINARY, DECIMAL, AND
 HEXADECIMAL SYMBOLS (TBL), B-1
 HEXADECIMAL ADDITION, B-5
 HEXADECIMAL CONVERSION MACRO
 FUNCTION, B-25
 HEXADECIMAL DIVISION, B-6
 HEXADECIMAL MULTIPLICATION, B-6
 HEXADECIMAL NUMBERING SYSTEM, B-2
 HEXADECIMAL REPRESENTATION OF
 INSTRUCTIONS, A-6
 HEXADECIMAL STRING CONSTANTS, 2-5
 HEXADECIMAL SUBTRACTION, B-5

HEXADECIMAL-TO-ASCII
 HEXADECIMAL-TO-ASCII CONVERSION,
 B-4

HEXADECIMAL-TO-DECIMAL
 HEXADECIMAL-TO-DECIMAL CONVERSION,
 B-2

HORIZONTAL TABS
 SPACES AND HORIZONTAL TABS, 8-37

IDENTIFIERS
 IDENTIFIERS, 2-2

INDEX

- IF
 - IF MACRO CONTROL STATEMENT, 8-19
- IGNORE
 - IGNORE SOURCE CHARACTER (IGN) MICRO OPERATION, 6-16
- IMMEDIATE MEMORY ADDRESSING
 - IMMEDIATE MEMORY ADDRESSING (IMA), 5-8
- IMMEDIATE OPERAND ADDRESSING
 - IMMEDIATE OPERAND (IMO) ADDRESSING, 6-11
- INCLUDE
 - INCLUDE MACRO CONTROL STATEMENT, 8-9
- INDEX MACRO
 - INDEX MACRO FUNCTIONS, 8-26
- INDEXED
 - INDEXED ADDRESSING, 5-25
 - INDEXED ADDRESSING MODES (TBL), 5-25
 - INDEXED B-RELATIVE POP ADDRESSING, 5-22
 - INDEXED B-RELATIVE PUSH ADDRESSING, 5-21
 - INDEXED DIRECT B-RELATIVE ADDRESSING, 5-15
 - INDEXED DIRECT IMMEDIATE MEMORY ADDRESSING, 5-9
 - INDEXED DIRECT P-RELATIVE ADDRESSING, 6-7
 - INDEXED INDIRECT B-RELATIVE ADDRESSING, 5-16
 - INDEXED INDIRECT IMMEDIATE MEMORY ADDRESSING, 5-10
- INDICATOR REGISTERS
 - COMMERCIAL PROCESSOR INDICATOR REGISTER, 1-10
 - INDICATOR (I) REGISTER, 1-7
 - SCIENTIFIC INDICATOR (SI) REGISTER, 1-8
- INDIRECT
 - INDIRECT B-RELATIVE ADDRESSING, 5-15
 - INDIRECT B-RELATIVE PLUS DISPLACEMENT ADDRESSING, 5-18
 - INDIRECT IMMEDIATE MEMORY ADDRESSING, 5-9
 - INDIRECT P-RELATIVE ADDRESSING, 5-13, 6-8
- INITIALIZATION
 - EDIT INSERTION TABLE AT INITIALIZATION (TBL), 6-13
 - INITIALIZATION AND MODIFICATION OF M-REGISTERS, 1-11
- INPUT/OUTPUT
 - ALTERNATE METHOD OF HANDLING INPUT/OUTPUT AND FILE MANIPULATION, 3-2
- INPUT/OUTPUT (CONT)
 - INPUT/OUTPUT OPERATIONS, 5-2
 - INPUT/OUTPUT (IO) INSTRUCTIONS, 5-5
- INSERT MICRO OPERATIONS
 - INSERT ASTERISK ON SUPPRESS (INSA) MICRO OPERATION, 6-16
 - INSERT BLANK ON SUPPRESS (INSB) MICRO OPERATION, 6-16
 - INSERT CHARACTER ON NEGATIVE (INSN) MICRO OPERATION, 6-16
 - INSERT CHARACTER ON POSITIVE (INSP) MICRO OPERATION, 6-17
 - INSERT MULTIPLE CHARACTERS (INSM) MICRO OPERATION, 6-16
- INSERTION
 - CHANGE EDIT INSERTION TABLE (CHT) MICRO OPERATION 6-14
 - CHARACTER INSERTION FOR MFLS MICRO OPERATION (TBL), 6-18
 - EDIT INSERTION TABLE, 6-13
- INSTRUCTION
 - ASSEMBLY LANGUAGE INSTRUCTION TYPES, 5-4
 - COMMERCIAL INSTRUCTION CATEGORIES, 6-2
 - COMMERCIAL INSTRUCTION SUMMARY (TBL), H-2
 - COMMERCIAL PROCESSOR INSTRUCTION FORMAT, 6-3
 - IMMEDIATE OPERAND ADDRESSING-SCIENTIFIC INSTRUCTION (FIG), 5-11
 - SHIFT INSTRUCTION FORMATS (FIG), 6-51
 - STACK INSTRUCTION FORMATS, K-2
 - VLD INSTRUCTION OPERATIONS (FIG), 5-149
- INSTRUCTIONS
 - ADDRESSING TECHNIQUES FOR COMMERCIAL PROCESSOR INSTRUCTIONS, 6-5
 - ASSEMBLY LANGUAGE INSTRUCTIONS, 5-1, 5-27
 - BIT/BYTE MANIPULATING INSTRUCTIONS, 5-26
 - B-REGISTER INSTRUCTIONS IN LAF, 5-26
 - BRANCH INSTRUCTIONS, 6-2
 - BRANCH-ON-INDICATOR (BI) INSTRUCTIONS, 5-4
 - BRANCH-ON-REGISTER INSTRUCTIONS, 5-4
 - CHARACTER STRING INSTRUCTIONS, 6-2
 - COMMERCIAL INSTRUCTIONS, 6-1
 - DECIMAL ARITHMETIC INSTRUCTIONS, 6-2
 - DETAILED COMMERCIAL INSTRUCTIONS, 6-23
 - DETAILED SCIENTIFIC INSTRUCTIONS, 7-2
 - DOUBLE OPERAND (DD) INSTRUCTIONS, 5-4
 - EDIT INSTRUCTIONS, 6-2
 - GENERIC (GE) INSTRUCTIONS, 5-5

INDEX

INSTRUCTIONS (CONT)
 INPUT/OUTPUT (IO) INSTRUCTIONS, 5-5
 PROGRAMMER'S INFORMATION FOR QUEUE INSTRUCTIONS, J-1
 PROGRAMMER'S INFORMATION FOR STACK INSTRUCTIONS, K-1
 RADIX AND MODE CONVERSION INSTRUCTIONS, 6-2
 SCIENTIFIC INSTRUCTIONS, 7-1
 SCIENTIFIC INSTRUCTIONS (SIP) ON 6/40 MODEL, 5-26
 SHIFT INSTRUCTIONS, 6-2
 SHIFT (SHS AND SHL) INSTRUCTIONS, 5-5
 SHORT-VALUE-IMMEDIATE (SI) INSTRUCTIONS, 5-6
 SINGLE OPERAND (SO) INSTRUCTIONS, 5-6

INSTRUCTIONS, ASSEMBLY LANGUAGE
 (SEE "ASSEMBLY LANGUAGE INSTRUCTIONS")

INSTRUCTIONS, COMMERCIAL
 (SEE "COMMERCIAL INSTRUCTIONS")

INSTRUCTIONS, SCIENTIFIC
 (SEE "SCIENTIFIC INSTRUCTIONS")

INTEGER
 BINARY INTEGER CONSTANTS, 2-7
 BINARY INTEGER CONSTANTS IN DECIMAL NOTATION, 2-8
 BINARY INTEGER CONSTANTS IN HEXADECIMAL NOTATION, 2-8
 DECIMAL INTEGER CONSTANTS, 2-8
 PACKED DECIMAL INTEGERS, 2-8
 SIGNED INTEGER DATA, 1-2
 UNPACKED DECIMAL INTEGER, 2-8

INTERNAL
 ASSEMBLY LANGUAGE INTERNAL FORMATS BY TYPE, A-4
 INTERNAL FORMAT OF DATA DESCRIPTORS, H-4
 INTERNAL FORMATS OF COMMERCIAL PROCESSOR INSTRUCTIONS, H-1
 INTERNAL LOCATION EXPRESSIONS, 2-15
 INTERNAL REPRESENTATION OF ASSEMBLY LANGUAGE INSTRUCTIONS (TBL), A-7
 INTERNAL VALUE EXPRESSIONS, 2-13

INTERRUPT
 INTERRUPT VECTOR ADDRESSING, 5-24

IO INSTRUCTIONS
 INPUT/OUTPUT (IO) INSTRUCTIONS, 5-5

LABELS
 LABELS, 2-2
 RESERVED LABELS, 2-3
 USER DEFINED LABELS, 2-2

LANGUAGE, ASSEMBLY
 ASSEMBLY LANGUAGE INSTRUCTION TYPES, 5-4

LANGUAGE, ASSEMBLY (CONT)
 ASSEMBLY LANGUAGE INSTRUCTIONS, 5-1, 5-27
 ASSEMBLY LANGUAGE INTERNAL FORMATS BY TYPE, A-4
 ASSEMBLY LANGUAGE SOURCE STATEMENT FORMATS, 3-1
 DEBUGGING ASSEMBLY LANGUAGE PROGRAMS, D-1
 ELEMENTS OF ASSEMBLY LANGUAGE, 2-1
 INTERNAL FORMATS OF ASSEMBLY LANGUAGE INSTRUCTIONS (FIG), A-5
 INTERNAL REPRESENTATION OF ASSEMBLY LANGUAGE INSTRUCTIONS (TBL), A-7

LIBM
 LIBM MACRO CONTROL STATEMENT, 8-7

LISTING
 CROSS-REFERENCE LISTING, 3-3
 LISTING OF BUBBLE SORT PROGRAM (FIG), C-3
 LISTING OF CHKNML SAMPLE PROGRAM (FIG), C-1

LIST-CONTROLLING STATEMENTS
 LIST-CONTROLLING STATEMENTS, 4-1

LOAD
 LOAD OPERATIONS, 5-2
 LOAD STACK ADDRESS REGISTER (LDT), K-2

LOCATION EXPRESSIONS
 COMMON LOCATION EXPRESSIONS, 2-16
 EXTERNAL LOCATION EXPRESSIONS, 2-15
 INTERNAL LOCATION EXPRESSIONS, 2-15
 LOCATION AND VALUE EXPRESSIONS, 2-13
 LOCATION EXPRESSIONS, 2-15

LOGIC
 TRAP 30 QUALITY LOGIC TEST (QLT) ERROR (QE), 6-22

MACRO
 CONDITIONAL MACRO CONTROL STATEMENTS, B-17
 CONTENTS OF MACRO ROUTINE, 8-2
 CREATING A MACRO ROUTINE, 8-2
 ENDM MACRO CONTROL STATEMENT, 8-3
 FAIL MACRO CONTROL STATEMENT, 8-17
 FORMAT OF MACRO FUNCTIONS, 8-23
 GOTO MACRO CONTROL STATEMENT, 8-18
 HEXADECIMAL CONVERSION MACRO, 8-25
 IF MACRO CONTROL STATEMENT, 8-19
 INCLUDE MACRO CONTROL STATEMENT, 8-9
 INDEX MACRO, 8-26
 INITIALIZED VALUES OF MACRO VARIABLES, 8-34
 LENGTH ATTRIBUTE MACRO, 8-23
 LIBM MACRO CONTROL STATEMENT, 8-7
 MAC MACRO CONTROL STATEMENT WITH PARAMETERS, 8-4
 MAC MACRO CONTROL STATEMENT WITHOUT PARAMETERS, 8-2

INDEX

MACRO (CONT)

MACRO CALLS, 8-11
 MACRO FACILITY, 8-1
 MACRO FUNCTIONS, 8-23
 MACRO ROUTINES, 8-1
 MACRO SUBSTITUTION, 8-14
 MACRO VARIABLES, 8-13
 NESTED MACRO CALL, 8-12
 NULL MACRO CONTROL STATEMENT, 8-22
 RECURSIVE MACRO CALLS, 8-13
 SEARCH MACRO FUNCTION, 8-27
 SETA MACRO CONTROL STATEMENT, 8-15
 SETN MACRO CONTROL STATEMENT, 8-16
 SITUATING MACRO ROUTINES, 8-6
 SOURCE CODE ERROR NOTIFICATION BY
 MACRO PREPROCESSOR, 8-1
 SPECIALIZING A MACRO ROUTINE BY
 PARAMETER SUBSTITUTION, 8-3
 SUBSTRING MACRO FUNCTION, 8-28
 TRANSLATE MACRO FUNCTION, 8-29
 VECTOR ORIENTATION MACRO FUNCTION,
 8-30
 VERIFY MACRO FUNCTION, 8-31

MASK REGISTER

SIP TRAP MASK (M5) REGISTER, 1-9

MEMORY

ASCII/HEXADECIMAL MEMORY DUMP
 (FIG), D-2
 DIRECT IMMEDIATE MEMORY ADDRESSING,
 5-8
 IMMEDIATE MEMORY ADDRESSING (IMA),
 5-8
 INDEXED DIRECT IMMEDIATE MEMORY
 ADDRESSING, 5-9
 INDEXED INDIRECT IMMEDIATE MEMORY
 ADDRESSING, 5-10
 INDIRECT IMMEDIATE MEMORY ADDRESSING
 5-9
 MEMORY MANAGEMENT OPERATIONS, 5-3
 READING AND INTERPRETING MEMORY
 DUMPS, D-1

MICRO OPERATION

CHANGE EDIT INSERTION TABLE (CHT)
 MICRO OPERATION, 6-14
 CHARACTER INSERTION FOR MFLS MICRO
 OPERATION (TBL), 6-18
 EDIT FLAGS FOR MICRO OPERATIONS
 (TBL), 6-14
 END FLOATING SUPPRESSION (ENF)
 MICRO OPERATION, 6-15
 FLOW DIAGRAM FOR SEF MICRO
 OPERATION (FIG), 6-19
 IGNORE SOURCE CHARACTER (IGN) MICRO
 OPERATION, 6-16
 INSERT ASTERISK ON SUPPRESS (INSA)
 MICRO OPERATION, 6-16
 INSERT BLANK ON SUPPRESS (INSB)
 MICRO OPERATION, 6-16
 INSERT CHARACTER ON NEGATIVE (INSN)
 MICRO OPERATION, 6-16
 INSERT CHARACTER ON POSITIVE (INSP)
 MICRO OPERATION, 6-17

MICRO OPERATION (CONT)

INSERT MULTIPLE CHARACTERS (INSM)
 MICRO OPERATION, 6-16
 MICRO EDIT FUNCTIONS, 6-12
 MICRO OPERATIONS FOR EDIT
 INSTRUCTIONS (TBL), 6-13
 MOVE SOURCE CHARACTER (MVC) MICRO
 OPERATION, 6-18
 MOVE WITH FLOAT SIGN INSERTION
 (MFLS) MICRO OPERATION, 6-16
 SET EDIT FLAGS (SEF) MICRO
 OPERATION, 6-19

M-REGISTERS

INITIALIZATION AND MODIFICATION OF
 M-REGISTERS, 1-11

MNEMONIC

MNEMONIC CODES, 2-1

MODE

COMMERCIAL PROCESSOR MODE REGISTER,
 1-9
 MODE (M) REGISTER, 1-5
 RADIX AND MODE CONVERSION
 INSTRUCTIONS, 6-2
 SIP MODE (M4) REGISTER, 1-8

MODIFY

MODIFY OPERATIONS, 5-3

MOVE

MOVE OPERATIONS, 5-3
 MOVE SOURCE CHARACTER (MVC) MICRO
 OPERATION, 6-18
 MOVE WITH FLOAT SIGN INSERTION
 (MFLS) MICRO OPERATION, 6-17

MULTIPLICATION

ESTABLISHING MULTIPLICATION FACTOR,
 5-26
 HEXADECIMAL MULTIPLICATION, B-6

NAMES, SYMBOLIC

DEFINING SYMBOLIC NAMES (TBL), 2-3
 RESERVED SYMBOLIC NAMES, G-1
 SYMBOLIC NAMES, 2-1

NESTED MACRO CALL

NESTED MACRO CALL, 8-12

NOP INSTRUCTION

NOP, 5-118

NULL

NULL MACRO CONTROL STATEMENT, 8-22

OPERAND

DOUBLE OPERAND (DO) INSTRUCTION, 5-4
 IMMEDIATE OPERAND ADDRESSING, 5-10
 IMMEDIATE OPERAND ADDRESSING-
 SCIENTIFIC INSTRUCTION (FIG), 5-11
 IMMEDIATE OPERAND (IMO) ADDRESSING,
 6-11
 SINGLE OPERAND (SO) INSTRUCTIONS,
 5-6

INDEX

- OPERATION CODE
 - OPERATION CODE DEFINING STATEMENT, 4-2
- OPERATIONS
 - ARITHMETIC OPERATIONS, 5-1
 - BOOLEAN OPERATIONS, 5-1
 - BRANCH OPERATIONS, 5-2
 - COMPARE OPERATIONS, 5-2
 - CONTROL OPERATIONS, 5-2
 - INPUT/OUTPUT OPERATIONS, 5-2
 - LOAD OPERATIONS, 5-2
 - MEMORY MANAGEMENT OPERATIONS, 5-3
 - MICRO OPERATIONS FOR EDIT INSTRUCTIONS (TBL), 6-13
 - MODIFY OPERATIONS, 5-3
 - MOVE OPERATIONS, 5-3
 - QUEUE OPERATIONS, 5-3
 - SHIFT OPERATIONS, 5-3
 - STACK OPERATIONS, 5-3
 - STORE OPERATIONS, 5-3
 - SWAP OPERATIONS, 5-3
 - VLD INSTRUCTION OPERATIONS (FIG), 5-149
- ORDER OF STATEMENTS
 - ORDER OF STATEMENTS IN SOURCE PROGRAM, 3-2
 - ORDER OF STATEMENTS WITHIN A SOURCE MODULE PROGRAM, 8-1
- ORG STATEMENT
 - ORG, 4-22
- PACKED
 - PACKED DECIMAL DATA DESCRIPTORS, 6-4
 - PACKED DECIMAL INTEGERS, 2-8
 - PACKED DECIMALS, H-5
- PARAMETER SUBSTITUTION
 - SPECIALIZING A MACRO ROUTINE BY PARAMETER SUBSTITUTION, 8-3
- PARENTHESES
 - BALANCED PARENTHESES, 8-36
- P-REGISTER
 - PROGRAM COUNTER (P-REGISTER), 1-4
- P-RELATIVE ADDRESSING
 - DIRECT P-RELATIVE ADDRESSING, 5-12, 6-6
 - INDEXED DIRECT P-RELATIVE ADDRESSING, 6-7
 - INDIRECT P-RELATIVE ADDRESSING, 5-13, 6-8
 - P-RELATIVE ADDRESSING, 5-12, 6-5
- POP ADDRESSING
 - B-RELATIVE POP ADDRESSING, 5-21
 - INDEXED B-RELATIVE POP ADDRESSING, 5-22
- PREPROCESSOR, MACRO
 - SOURCE CODE ERROR NOTIFICATION BY MACRO PREPROCESSOR, 8-1
- PROCEDURES, EXTERNAL
 - CALLING EXTERNAL PROCEDURES, 3-2
- PROCESSOR
 - SCIENTIFIC INFORMATION PROCESSOR (SIP) REGISTERS, 1-7
 - SOFTWARE SIMULATION OF THE SCIENTIFIC INFORMATION PROCESSOR, 1-9
- PROCESSOR, COMMERCIAL
 - (SEE "COMMERCIAL PROCESSOR")
- PROGRAM
 - LISTING OF BUBBLE SORT PROGRAM (FIG), C-3
 - ORDER OF STATEMENTS IN SOURCE PROGRAM, 3-2
 - ORDER OF STATEMENTS IN SOURCE MODULE PROGRAM, 8-1
 - PROGRAM COUNTER (P-REGISTER), 1-4
 - PROGRAM-LINKING STATEMENTS, 4-2
- PROGRAMMER'S INFORMATION
 - PROGRAMMER'S INFORMATION, A-1
 - PROGRAMMER'S INFORMATION FOR QUEUE INSTRUCTIONS, J-1
 - PROGRAMMER'S INFORMATION FOR STACK INSTRUCTIONS, K-1
- PROGRAMMING
 - COMMERCIAL PROCESSOR (CP) PROGRAMMING, 6-1
 - PROGRAMMING CONSIDERATIONS, 3-1, 8-34
- PROTECTION
 - PROTECTION OPERATORS, 8-5
- PUSH ADDRESSING
 - B-RELATIVE PUSH ADDRESSING, 5-20
 - INDEXED B-RELATIVE PUSH ADDRESSING, 5-21
- QUEUE
 - PROGRAMMER'S INFORMATION FOR QUEUE INSTRUCTIONS, J-1
 - QUEUE MANAGEMENT (FIG), J-2
 - QUEUE OPERATIONS, 5-3
- RADIX
 - RADIX AND MODE CONVERSION INSTRUCTIONS, 6-2
- REENTRANCY
 - REENTRANCY, 3-3
- REGISTER
 - INDICATOR (I) REGISTER, 1-7
 - LOAD STACK ADDRESS REGISTER (LDT), K-2
 - REGISTER ADDRESSING, 5-7
 - REMOTE DESCRIPTOR BASE REGISTER (RDPR), 1-4
 - SCIENTIFIC INDICATOR (SI) REGISTER, 1-8

INDEX

REGISTER (CONT)

SIP MODE (M4) REGISTER, 1-8
 SIP TRAP MASK (M5) REGISTER, 1-9
 STACK REGISTER (T), 1-5
 STORE STACK ADDRESS REGISTER (STT),
 K-2
 SYSTEM STATUS (S) REGISTER, 1-5

REGISTERS

ADDRESS REGISTERS, 1-4
 BASE ADDRESS (BN) REGISTERS, 1-4
 COMMERCIAL PROCESSOR REGISTERS,
 1-9
 GENERAL (RN) REGISTERS, 1-5
 HARDWARE REGISTERS, 1-4
 LEVEL 6 HARDWARE REGISTERS (FIG),
 A-1
 LEVEL 6 REGISTERS (FIG), 1-6
 MODE (M) REGISTERS, 1-5
 SCIENTIFIC ACCUMULATOR (SN)
 REGISTERS, 1-7
 SCIENTIFIC INFORMATION PROCESSOR
 (SIP) REGISTERS, 1-7
 SUMMARY OF HARDWARE REGISTERS, A-1

RELINQUISH

RELINQUISH STACK FRAME (RLQ), K-2

REMOTE

REMOTE DESCRIPTOR ADDRESS
 GENERATION (FIG), H-4
 REMOTE DESCRIPTOR BASE REGISTER
 (RDBR), 1-4

REPRESENTATION

HEXADECIMAL REPRESENTATION OF
 INSTRUCTIONS, A-6
 INTERNAL REPRESENTATION OF ASSEMBLY
 LANGUAGE INSTRUCTIONS (TBL), A-7
 LEVEL 6 DATA REPRESENTATION, 1-1

RESERVED

RESERVED LABELS, 2-3
 RESERVED SYMBOLIC NAMES, G-1

RESV STATEMENT

RESV, 4-24

ROUTINE, MACRO

CONTENTS OF MACRO ROUTINE, 8-2
 CREATING A MACRO ROUTINE, 8-2
 MACRO ROUTINES, 8-1
 SPECIALIZING A MACRO ROUTINE BY
 PARAMETER SUBSTITUTION, 8-3

SAF/LAF

SAF/LAF CONSIDERATIONS, 3-3

SCIENTIFIC

DESCRIPTIONS SCIENTIFIC
 INSTRUCTIONS, 7-2
 SCIENTIFIC ACCUMULATOR (SN)
 REGISTERS, 1-7
 SCIENTIFIC INDICATOR (SI)
 REGISTER, 1-8

SCIENTIFIC (CONT)

SCIENTIFIC INFORMATION PROCESSOR
 (SIP) REGISTERS, 1-7
 SCIENTIFIC INSTRUCTIONS, 7-1
 SCIENTIFIC INSTRUCTIONS (SIP) ON
 6/40 MODEL, 5-26
 SCIENTIFIC TRAPS, 7-1
 SOFTWARE SIMULATION SCIENTIFIC
 INFORMATION PROCESSOR, 1-9

SCIENTIFIC INSTRUCTIONS

SAD, 7-2
 SBE, 7-4
 SBEU, 7-5
 SBEZ, 7-6
 SBG, 7-7
 SBGE, 7-8
 SBGEZ, 7-9
 SBGZ, 7-10
 SBL, 7-11
 SBLE, 7-12
 SBLEZ, 7-13
 SBLZ, 7-14
 SBNE, 7-15
 SBNEU, 7-16
 SBNEZ, 7-17
 SBNPE, 7-18
 SBNSE, 7-19
 SBPE, 7-20
 SBSE, 7-21
 SCM, 7-22
 SCZD, 7-23
 SCZQ, 7-24
 SDV, 7-25
 SLD, 7-26
 SML, 7-27
 SNGD, 7-28
 SNGQ, 7-29
 SSB, 7-30
 SST, 7-31
 SSW, 7-32

SEARCH

SEARCH MACRO FUNCTION, 8-27

SEMI-COLONS

COMMAS AND SEMI-COLONS, 8-37

SERVICES, SYSTEM

CALLING SYSTEM SERVICES, 3-2

SET

SET EDIT FLAGS (SEF) MICRO
 OPERATION, 6-19

SHIFT

SHIFT INSTRUCTION FORMATS (FIG),
 6-55
 SHIFT INSTRUCTIONS, 6-2
 SHIFT OPERATIONS, 5-3
 SHIFT (SHS AND SHL) INSTRUCTIONS,
 5-5

SHORT DISPLACEMENT ADDRESSING

SHORT DISPLACEMENT ADDRESSING, 5-23

INDEX

SHORT-VALUE-IMMEDIATE INSTRUCTIONS
 SHORT-VALUE-IMMEDIATE (SI)
 INSTRUCTIONS, 5-6

SIGNED
 SIGNED INTEGER DATA, 1-2

SIMULATION
 SOFTWARE SIMULATION OF COMMERCIAL
 PROCESSOR, 1-10
 SOFTWARE SIMULATION OF SCIENTIFIC
 INFORMATION PROCESSOR, 1-9

SIP
 ADDRESS SYLLABLES FOR CPU & SIP
 INSTRUCTIONS (TBL), A-9
 SCIENTIFIC INFORMATION PROCESSOR
 (SIP) REGISTERS, 1-7
 SCIENTIFIC INSTRUCTIONS (SIP) ON
 6/40 MODEL, 5-26
 SIP MODE (M4) REGISTER, 1-8
 SIP TRAP MASK (M5) REGISTER, 1-9

SOFTWARE SIMULATION
 SOFTWARE SIMULATION OF COMMERCIAL
 PROCESSOR, 1-10
 SOFTWARE SIMULATION OF SCIENTIFIC
 INFORMATION PROCESSOR, 1-9

SORT, BUBBLE
 LISTING OF BUBBLE SORT PROGRAM
 (FIG), C-3

SOURCE
 ASSEMBLY LANGUAGE SOURCE STATEMENT
 FORMATS, 3-1
 IGNORE SOURCE CHARACTER (IGN) MICRO
 OPERATION, 6-16
 MOVE SOURCE CHARACTER (MVC) MICRO
 OPERATION, 6-18
 ORDER OF STATEMENTS IN SOURCE
 PROGRAM, 3-2
 ORDER OF STATEMENTS IN A SOURCE
 MODULE PROGRAM, 8-1
 SOURCE CODE ERROR FLAGS, E-1
 SOURCE CODE ERROR NOTIFICATION BY
 MACRO PREPROCESSOR, 8-1

SPACES
 SPACES AND HORIZONTAL TABS, 8-37

SPECIALIZED ADDRESS
 SPECIALIZED ADDRESS EXPRESSION, 5-24

SPECIALIZING MACRO ROUTINE
 SPECIALIZING A MACRO ROUTINE BY
 PARAMETER SUBSTITUTION, 8-3

STACK
 ACQUIRE STACK FRAME (ACQ), K-2
 LOAD STACK ADDRESS REGISTER (LDT),
 K-2
 PROGRAMMER'S INFORMATION FOR STACK
 INSTRUCTIONS, K-1
 RELINQUISH STACK FRAME (RLQ), K-2

STACK (CONT)
 STACK FRAME, K-1
 STACK INSTRUCTION FORMATS, K-2
 STACK OPERATIONS, 5-3
 STACK REGISTER (T), 1-5
 STACK STRUCTURE (FIG), K-1
 STORE STACK ADDRESS REGISTER (STT),
 K-2

STATEMENTS, ORDER OF
 SOURCE PROGRAM STATEMENTS, 3-2
 STATEMENTS IN SOURCE MODULE
 PROGRAM, 8-1

STATUS, SYSTEM
 SYSTEM STATUS (S) REGISTER, 1-5

STORAGE-ALLOCATION STATEMENTS
 STORAGE-ALLOCATION STATEMENTS, 4-2

STORE
 STORE OPERATIONS, 5-3
 STORE STACK ADDRESS REGISTER (STT),
 K-2

STRING
 BIT STRING CONSTANTS, 2-6
 CHARACTER STRING INSTRUCTIONS, 6-2
 HEXADECIMAL STRING CONSTANTS, 2-5
 STRING CONSTANTS, 2-5
 TRUNCATION/PADDING STRING
 CONSTANTS, 2-6

SUBSTITUTION, MACRO
 MACRO SUBSTITUTION, 8-14
 SPECIALIZING A MACRO ROUTINE BY
 PARAMETER SUBSTITUTION, 8-3

SUBSTRING, MACRO
 SUBSTRING MACRO FUNCTION, 8-28

SUBTRACTION, HEXADECIMAL
 HEXADECIMAL SUBTRACTION, B-5

SWAP OPERATIONS
 SWAP OPERATIONS, 5-3

SYLLABLE, ADDRESS
 ADDRESS SYLLABLE, H-7
 ADDRESS SYLLABLES FOR CPU & SIP
 INSTRUCTIONS (TBL), A-9
 COMMERCIAL PROCESSOR ADDRESS
 SYLLABLE, H-7

SYMBOLIC NAMES
 RESERVED SYMBOLIC NAMES, G-1
 SYMBOLIC NAMES, 2-1

SYMBOL-DEFINING STATEMENTS
 SYMBOL-DEFINING STATEMENTS, 4-2

SYSTEM
 CALLING SYSTEM SERVICES, 3-2
 HEXADECIMAL NUMBERING SYSTEM, B-2
 SYSTEM STATUS (S) REGISTER, 1-5

INDEX

TABS
 SPACES AND HORIZONTAL TABS, 8-37

TEST
 COMMERCIAL PROCESSOR HARDWARE
 TEST PROGRAM (FIG), H-8
 TRAP 30 QUALITY LOGIC TEST (QLT)
 ERROR (QE), 6-22

TEXT STATEMENT
 TEXT, 4-25

TITLE STATEMENT
 TITLE, 4-26

TRANSLATE MACRO
 TRANSLATE MACRO FUNCTION, 8-29

TRAP
 COMMERCIAL PROCESSOR TRAPS, 6-20
 SCIENTIFIC TRAPS, 7-1
 SIP TRAP MASK (M5) REGISTER, 1-9
 TRAP 23 UNAVAILABLE RESOURCE (UR),
 6-21
 TRAP 24 BUS OR MEMORY ERROR (BE),
 6-21
 TRAP 25 DIVIDE BY ZERO (DZ), 6-21
 TRAP 26 ILLEGAL SPECIFICATION (IS),
 6-22
 TRAP 27 ILLEGAL CHARACTER (IC),
 6-22
 TRAP 28 TRUNCATION (TR), 6-22
 TRAP 29 OVERFLOW (OV), 6-22
 TRAP 30 QUALITY LOGIC TEST (QLT)
 ERROR (QE), 6-22
 TRAP CONTENT (FIG), 6-55
 TRAP VECTORS AND EVENTS (TBL), 7-1

TRUNCATION/PADDING CONSTANTS
 TRUNCATION/PADDING STRING
 CONSTANTS, 2-6

UNPACKED DECIMAL
 UNPACKED DECIMAL INTEGER, 2-8
 UNPACKED DECIMALS, H-4
 UNPACKED-DECIMAL DATA
 DESCRIPTOR, 6-4

UNSIGNED DATA
 UNSIGNED DATA, 1-3

VALID ADDRESS
 VALID ADDRESS EXPRESSIONS, A-10

VALUE
 ALPHANUMERIC VALUES, 8-35
 EXTERNAL VALUE EXPRESSIONS, 2-14
 INTERNAL VALUE EXPRESSIONS, 2-13
 LOCATION AND VALUE EXPRESSIONS,
 2-13
 NUMERIC VALUES, 8-35
 VALUE EXPRESSIONS, 2-13

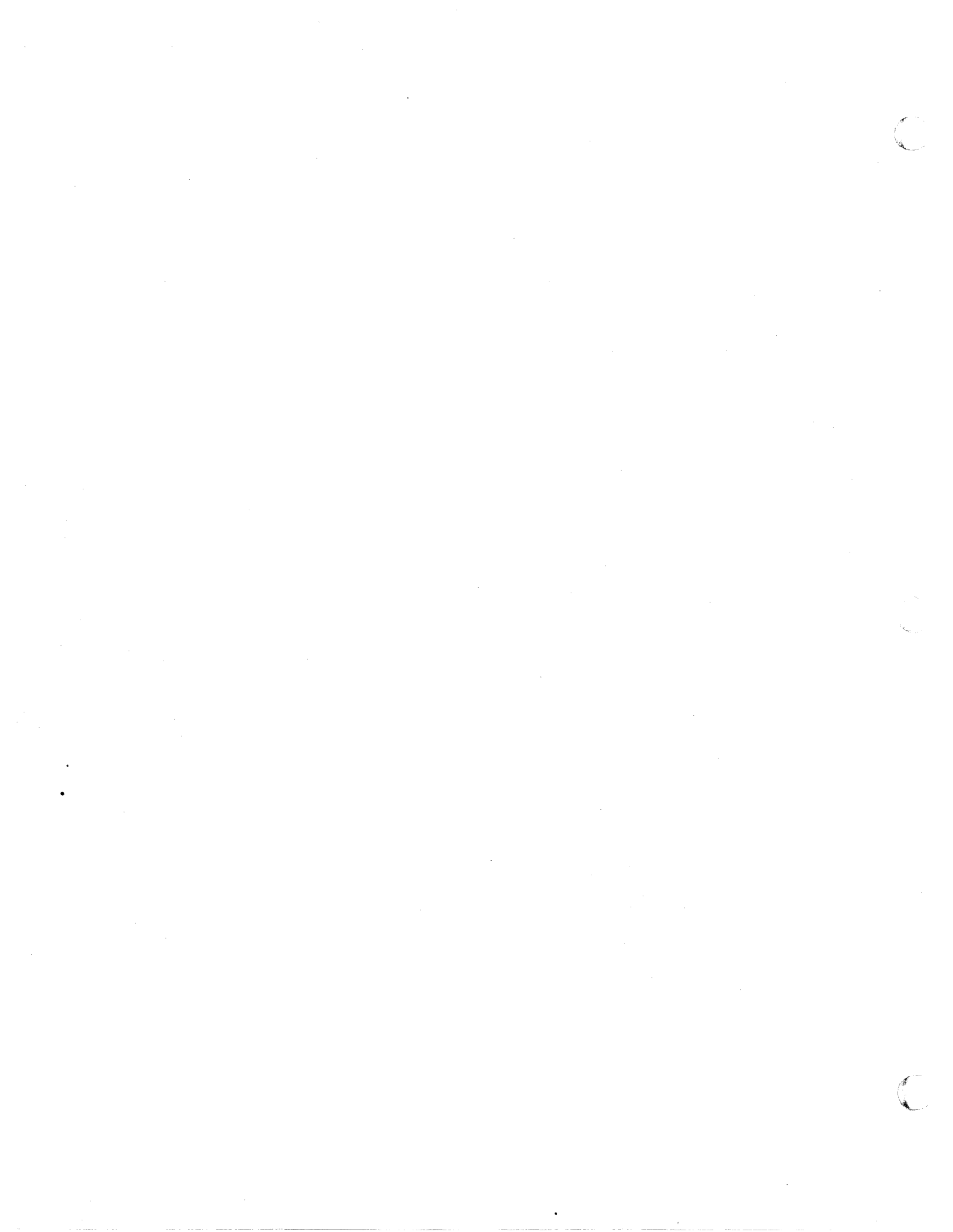
VARIABLES, MACRO
 INITIALIZED VALUES OF MACRO
 VARIABLES, 8-34

VARIABLES, MACRO (CONT)
 MACRO VARIABLES, 8-13

VECTOR
 INTERRUPT VECTOR ADDRESSING, 5-24
 VECTOR ORIENTATION MACRO, 8-30

VECTORS
 COMMERCIAL PROCESSOR TRAP VECTORS
 AND EVENTS (TBL), 6-21
 TRAP VECTORS AND EVENTS (TBL), 7-1

VERIFY
 VERIFY MACRO FUNCTION, 8-31



HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

CUT ALONG LINE

TITLE

SERIES 60 (LEVEL 6) GCOS 6
ASSEMBLY LANGUAGE REFERENCE

ORDER NO.

CB07, REV. 1

DATED

JUNE 1978

ERRORS IN PUBLICATION

[Empty box for errors in publication]

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

[Empty box for 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 _____

PLEASE FOLD AND TAPE —

NOTE: U. S. Postal Service will not deliver stapled forms

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

CUT ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE



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.

20845, 5778, Printed in U.S.A.

CB07, Rev. 1