SERIES 60 (LEVEL 6)

GCOS

ASSEMBLY LANGUAGE REFERENCE ADDENDUM B

SUBJECT

Changes and Additions to the Manual

SPECIAL INSTRUCTIONS

Insert the attached pages into the manual (Revision 1, dated June 1978) according to the Collating Instructions on the back of this cover. Except for new Appendixes L and M, change bars in the margins indicate new or changed information and asterisks denote deletions.

Note:

Insert this cover behind the manual cover to indicate that the manual has been updated with this Addendum.

SOFTWARE SUPPORTED

This update describes Release 0200 of the Assembler, which executes under the Series 60 (Level 6) GCOS 6 MOD 400 Releases 0110 and 0120 and MOD 600 (Release 0110) Executives. See the Manual Directory of the appropriate *System Concepts* manual for information as to later releases supported by this document.

ORDER NUMBER

CB07-01B

July 1979

24022
2779
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update this manual, remove old pages and insert new pages as follows:

Remove	Insert
iii, iv	iii, iv
ix through xii	ix through xii
1-9, 1-10	1-9, 1-10
2-7 through 2-12	2-7 through 2-12
—	2-12.1, blank
3-1, 3-2	3-1, blank
—	3-1.1, 3-2
4-17, 4-18	4-17, 4-18
4-23, 4-24	4-23, 4-24
5-17 through 5-20	5-17 through 5-20
5-27 through 5-30	5-27 through 5-30
5-87, 5-88	5-87, 5-88
5-103 through 5-106	5-103 through 5-106
5-123, 5-124	5-123, 5-124
5-133, 5-134	5-133, 5-134
5-143, 5-144	5-143, 5-144
6-25, 6-26	6-25, 6-26
6-53, 6-54	6-53, 6-54
6-57, 6-58	6-57, 6-58
6-61, 6-62	6-61, 6-62
8-1, 8-2	8-1, 8-2
8-5, 8-6	8-5, 8-6
—	8-26.1, blank
A-9 through A-11, blank	A-9 through A-11, blank
C-1, C-2	C-1, C-2
—	Appendix L
—	Appendix M

MANUAL DIRECTORY

The following publications constitute the GCOS 6 manual set. See the Manual Directory of the appropriate *System Concepts* manual for the current revision number, and addenda (if any) of the relevant operating system specific publications.

Order No.	*Manual Title*
CB01	*GCOS 6 Program Preparation*
CB02	*GCOS 6 Commands*
CB03	*GCOS 6 Communications Processing*
CB04	*GCOS 6 Sort/Merge*
CB05	*GCOS 6 Data File Organizations and Formats*
CB06	*GCOS 6 System Messages*
CB07	*GCOS 6 Assembly Language Reference*
CB08	*GCOS 6 System Service Macro Calls*
CB09	*GCOS 6 RPG Reference*
CB10	*GCOS 6 Intermediate COBOL Reference*
CB12	*GCOS 6 Entry-Level COBOL Reference*
CB13	*GCOS 6 FORTRAN Reference*
CB14	*GCOS 6 Advanced COBOL Reference*
CB15	*GCOS 6 Advanced COBOL Reference Guide*
CB16	*GCOS 6 I-D-S/II Reference Card*
CB20	*GCOS 6 MOD 400 System Concepts*
CB21	*GCOS 6 MOD 400 Program Execution and Checkout*
CB22	*GCOS 6 MOD 400 Programmer's Guide*
CB23	*GCOS 6 MOD 400 System Building*
CB24	*GCOS 6 MOD 400 Operator's Guide*
CB27	*GCOS 6 MOD 400 Programmer's Pocket Guide*
CB28	*GCOS 6 MOD 400 Master Index*
CB30	*Remote Batch Facility User's Guide*
CB31	*Data Entry Facility User's Guide*
CB32	*Data Entry Facility Operator's Quick Reference Guide*
CB33	*Level 6/Level 6 File Transmission Facility User's Guide*
CB34	*Level 6/Level 62 File Transmission Facility User's Guide*
CB35	*Level 6/Level 64 (Native) File Transmission Facility User's Guide*
CB36	*Level 6/Level 66 File Transmission Facility User's Guide*
CB37	*Level 6/Series 200/2000 File Transmission Facility User's Guide*
CB38	*Level 6/BSC 2780/3780 File Transmission Facility User's Guide*
CB39	*Level 6/Level 64 (Emulator) File Transmission Facility User's Guide*
CB40	*2780/3780 Workstation Facility User's Guide*
CB41	*HASP Workstation Facility User's Guide*
CB42	*Level 66 Host Resident Facility User's Guide*
CB43	*Terminal Concentration Facility User's Guide*
CB44	*Interactive Function User's Guide*
CB50	*GCOS 6 MOD 600 System Concepts*
CB51	*GCOS 6 MOD 600 Program Execution and Checkout*
CB52	*GCOS 6 MOD 600 Programmer's Guide*
CB53	*GCOS 6 MOD 600 System Building*
CB54	*GCOS 6 MOD 600 Administrator's Guide*
CB55	*GCOS 6 MOD 600 Transaction Driven System*
CB56	*I-D-S/II Data Base Administrator's Guide*
CB57	*I-D-S/II Data Base User's Guide*
CB58	*GCOS 6 MOD 600 Operator's Guide*
CB59	*GCOS 6 MOD 600 Master Index*
CD46	*Display Formatting and Control*

CD47	*GCOS 6 MOD 200 System Concepts*
CD48	*GCOS 6 MOD 200 Application Development Guide*
CD49	*GCOS 6 MOD 200 Operator's Guide*
CD50	*GCOS 6 MOD 200 HASP Workstation Facility User's Guide*
CD51	*GCOS 6 MOD 200 L6 to L6 File Transmission Facility User's Guide*
CD52	*GCOS 6 MOD 200 L6 to L66 File Transmission Facility User's Guide*
CF11	*RBF/64 User's Guide*
CG65	*GCOS 6 MOD 600 Operator's Pocket Guide*
CG66	*GCOS 6 MOD 600 Programmer's Pocket Guide*
CG71	*GCOS 6 MOD 600 System Building Memory Calculator*
CG72	*GCOS 6 MOD 600 Software and Documentation Directory*

In addition, the following publications provide supplementary information:

Order No.	*Manual Title*
AT97	*Level 6 Communications Handbook*
CC71	*Level 6 Minicomputer Systems Handbook*
FQ41	*Writable Control Store User's Guide*

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 7-3
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 7-33

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 8-4
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
Requote Macro Function 8-26.1
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
Dump Edit 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

Appendix L. Assembly Language Program Independence

Assembly Language Program Hardware Independence L-1
Self-modifying Procedures L-1
Writing Source Programs That Can Be Executed in Both SAF and LAF Configurations L-1
SAF/LAF Independence by Assembly L-3
SAF/LAF Independence by Loading L-3
Differences between SAF and LAF L-3
General Rules for Writing SLIC Programs L-4
Procedures for Writing Specific Parts of a SLIC Program L-4
Addressing Mode L-4
Data Structures Containing Pointers L-5
Data Management Structures L-5
Argument Lists and Pointer Arrays L-5
Request Blocks L-6
Individual Pointers L-7
Hardware-Defined Structures L-7
Immediate Memory Address Operands L-7
Absolute Addresses L-7

Appendix M. Reentrant Programs

Figures

1-1 Assembler Functions 1-1
1-2 Level 6 Registers 1-6
5-1 Direct Immediate Memory Addressing 5-8
5-2 Indirect Immediate Memory Addressing 5-9
5-3 Indexed Direct Immediate Memory Addressing 5-10
5-4 Indexed Indirect Immediate Memory Addressing 5-11
5-5 Immediate Operand Addressing-Scientific Instruction 5-11
5-6 Immediate Operand Addressing 5-12
5-7 Direct P-Relative Addressing 5-12
5-8 Indirect P-Relative Addressing 5-13
5-9 Direct B-Relative Addressing 5-14
5-10 Indirect B-Relative Addressing 5-15
5-11 Indexed Direct B-Relative Addressing 5-16
5-12 Indexed Indirect B-Relative Addressing 5-17
5-13 Direct B-Relative Plus Displacement Addressing 5-17
5-14 Indirect B-Relative Plus Displacement Addressing 5-18
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
6-2 Commercial Processor Indexed Direct P-relative Addressing .. 6-7
6-3 Commercial Processor Indirect P-relative Addressing 6-8
6-4 Commercial Processor Direct and Indirect B-Relative Plus Displacement Addressing 6-10
6-5 Commercial B-Relative Plus Displacement With Indexing Addressing 6-11
6-6 Commercial Processor IMO Addressing 6-12
6-7 Flow Diagram of SEF Micro Operation 6-19
6-8 Trap Context 6-21
6-9 Shift Instruction Formats 6-55
8-1 Sample Unexpanded Source Module and Assembler Listing of Resulting Expanded Source Module 8-32
A-1 Level 6 Hardware Registers A-1
A-2 Internal Formats of Assembly Language Instructions A-5
C-1 Listing of CHKNML Program C-1
C-2 Listing of Bubble Sort Program C-3
D-1 ASCII/Hexadecimal Memory Dump D-2
H-1 Internal Formats of Commercial Processor Instructions H-1
H-2 Remote Descriptor Address Generation H-4
H-3 Decimal Data Descriptor Format .. H-4
H-4 Alphanumeric Data Descriptor Format H-6
H-5 Binary Data Descriptor Format H-6
H-6 Commercial Processor Address Syllable Format H-7
H-7 Commercial Processor Hardware Test Program H-8
J-1 Queue Management J-2
K-1 Stack Structure K-1
L-1 Methods of Achieving SAF/LAF Independence L-2
L-2 Valid Combinations of Compilation Units for Linking L-2

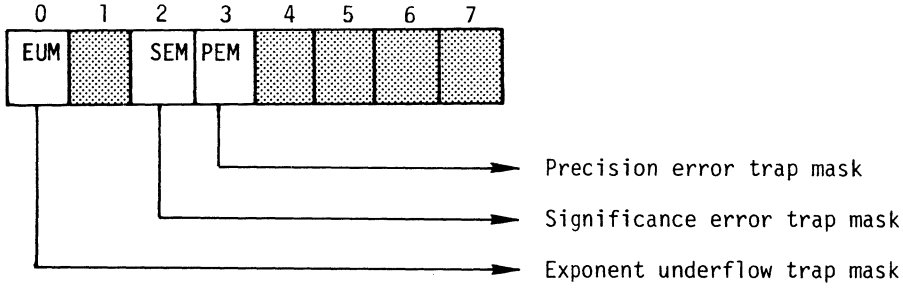
Tables

2-1 Defining Symbolic Names 2-3
2-2 Rules of Truncation/Padding String Constants 2-6
2-3 Internal Sign Convention and Range of Values for Unpacked Decimal Integers 2-8
2-4 Prefix Letter and Range of Values for Signed and Unsigned Packed Decimal Integers 2-8
5-1 Indexed Addressing Modes 5-25
6-1 Micro Operations for Edit Instructions 6-13
6-2 Edit Insertion Table at Initialization 6-13
6-3 Edit Flags for Micro Operations 6-14
6-4 Code for Replacing EIT Entries 6-15
6-5 Character Insertion by MFLS Micro Operation 6-18
6-6 Commercial Processor Trap Vectors and Events 6-21
7-1 Scientific Traps 7-2
A-1 Internal Representation of Assembly Language Instructions A-6
A-2 Address Syllables for CPU & SIP Instructions A-9
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

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 for MOD 400, or DSIP must be specified in SYSTEM macro routine for MOD 600. (See *System Building* manual.)

The DSIP simulates all functions of the SIP. The SSIP is partial simulator which is available in MOD 400 only. 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. See also the Section "Scientific Instructions" later in this manual.

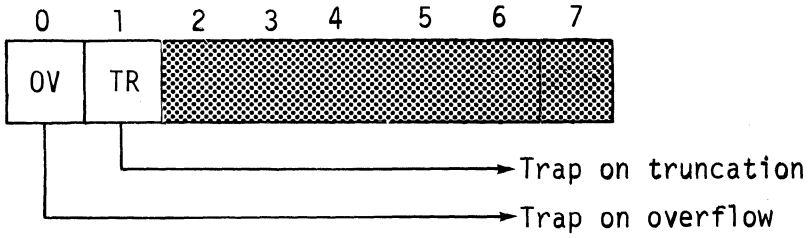
- SSIP uses registers R4, R5, and R7 to simulate a scientific register (assumed to be $S1). A task that executes scientific instructions that might be simulated by SSIP should dedicate these three registers to the use of the simulator.
- SSIP uses the CPU-1 register to store the results of a scientific 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:
  — Either CPU branch instructions or simulated SIP branch instructions may be used to test these indicators. The simulated SIP branch instructions are recommended since they are upward compatible with the DSIP and the SIP hardware.
  — Execution of scientific instructions alters the CPU I-register instead of the SIP's SI register.
- The SSIP does not support the MTM or STM instruction on Models 20 and 30.
- 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

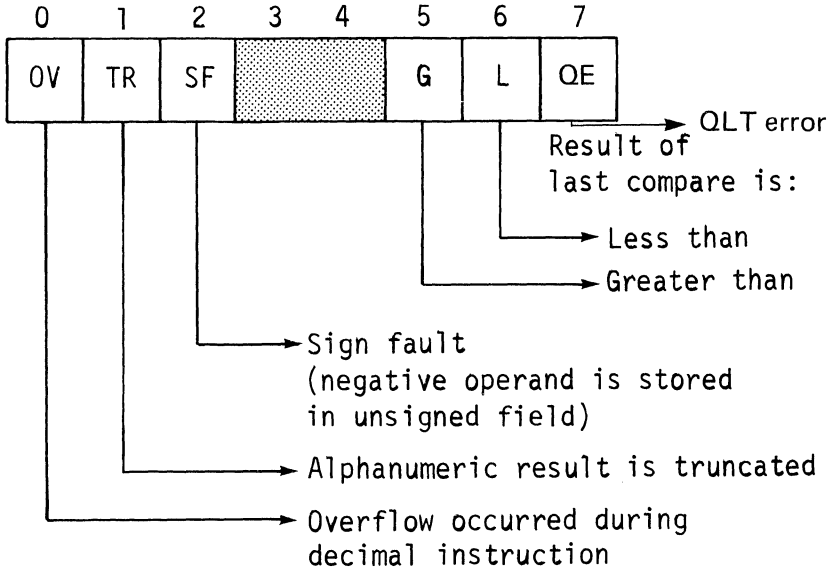
The 8-bit Commercial Processor mode register is a copy of the M3 register (in the CPU) which is provided for use with the Commercial Processor. Both are set to zero at initialization of the CPU. Both registers may be modified with an MTM instruction. If only the Commercial Processor is initialized, the 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 Commercial Instruction Processor (CIP) is not available, GCOS provides a subset of the CIP instructions through software simulation. The CIP simulator is entered via trap vector 5.

Note the following considerations with respect to the use of the CIP simulator.

- The Alphanumeric Search (SRCH) and Alphanumeric Verify (VRFY) opcodes are not supported.
- On Model 30, the CIP simulator supports the MTM, STM, LRDB, and SRDB instructions.
- On Model 20, the CIP simulator supports the MTM and STM instructions.
- Bit 7 of the Commercial Processor Mode Register must be set to zero.

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.

$$\left[\begin{matrix}+\\-\end{matrix}\right] \left\{\begin{matrix}\text{n[n...]}\\ \text{X'h[h...]'}\end{matrix}\right\}$$

$$\left[\begin{matrix}+\\-\end{matrix}\right]$$

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.

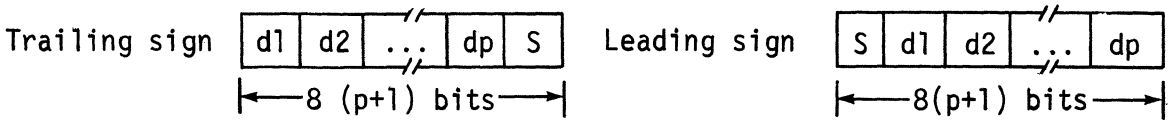
Unpacked Decimal Integers

The prefix letter designating the internal sign convention and the range of values allowed for each convention of unpacked decimal integers are shown in Table 2-3.

TABLE 2-3. INTERNAL SIGN CONVENTION AND RANGE OF VALUES FOR UNPACKED DECIMAL INTEGERS

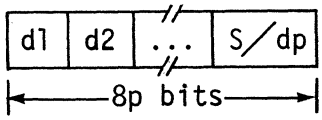
Sign Convention	*Letter*	*Range of Values*
Leading separate	L	$-10^{30}<n<+10^{30}$
Trailing separate	T	$-10^{30}<n<+10^{30}$
Trailing overpunch	O	$-10^{31}<n<+10^{31}$
Unsigned	N	$0\leq n<+10^{31}$

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).

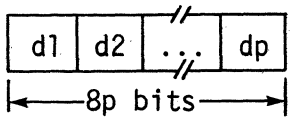
The format of an unpacked decimal integer with the sign indicated by a trailing overpunch is as follows:



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:



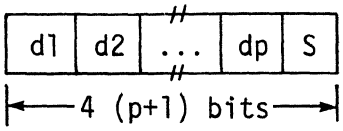
Packed Decimal Integers

The prefix letter and the range of values for signed and unsigned packed decimal integers are shown in Table 2-4.

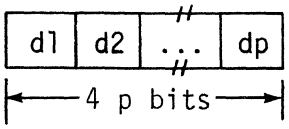
TABLE 2-4. PREFIX LETTER AND RANGE OF VALUES FOR SIGNED AND UNSIGNED PACKED DECIMAL INTEGERS

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:



Format for packed signed decimal integers



Format for packed unsigned decimal integers

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:

Source language	Stored Value *(hexadecimal)*
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 binary point is situated s bits to the left of the rightmost bit stored in memory. A negative scale factor means that the binary 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

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

Format 2

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

[±]
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, $0<r\leq 31$.

[±]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.

Source Language	*Stored Value*	
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

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

Format 2

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

Format 3

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

[±]

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 X 10^{-79}) less than 2^{252} (approximately 7.2370 X 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.

Source Language	*Stored Value*			
−.5	8180	0000		
5.	8250	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 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:

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

Boolean Function	*Meaning*
AND	Conjunction of argument1 and argument2
OR	Inclusive disjunction of argument1 and argument2
XOR	Exclusive disjunction of argument1 and argument2
NOT	Negation of argument1

Shift Function	*Meaning*
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

Arithmetic Function	*Meaning*
MOD	Remainder after division when argument1 is divided by argument2
MAX	The value of the algebraically largest argument
MIN	The value of the algebraically smallest argument

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	$0 \leq$ argument2 < 15
ARS	$0 \leq$ argument2 < 15
LLS and LRS	$0 \leq$ 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

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:

Column 1 →

{Δ / labelΔ / linenumΔ / linenum-labelΔ} opcode {Δoperand {,operand {,operand[...]}} / ;} / Δ} [Δcomments]

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

Column 1 →

[linenum][Δ]operand {;,operand[...] / Δ} [Δcomments]

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

Column 1 →

{* / / / # / @ / linenum* / linenum/ / linenum# / linenum@} comments

The asterisk (*) indicates that the comment line is to be included in the listing wherever it is included in the source language program. The slash (/) indicates that the Assembler is to cause the printer to skip to the top of the next page of the listing before printing the comment. The pound sign (#) and the at-sign (@) designate macro processor comment lines. Upon request the macro processor generates comment lines that begin with the at-sign (@). These lines are macro control statements without errors. The macro processor unconditionally generates comment lines that begin with the pound sign (#). These lines are statements that generate macro processing errors. 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 *Commands* manual.

IF

Instruction:

Conditional skip

Source Language Format:

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

Description:

If the specified condition is met, the Assembler skips (reads but does not process) 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 lie of assembly code. This is an example of varying an assembly procedure without altering the assembly language source program.

LCOMM

Instruction:

Define local common block

Source Language Format:

labelΔLCOMMΔint-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.

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, and must be ≥ 0. 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.

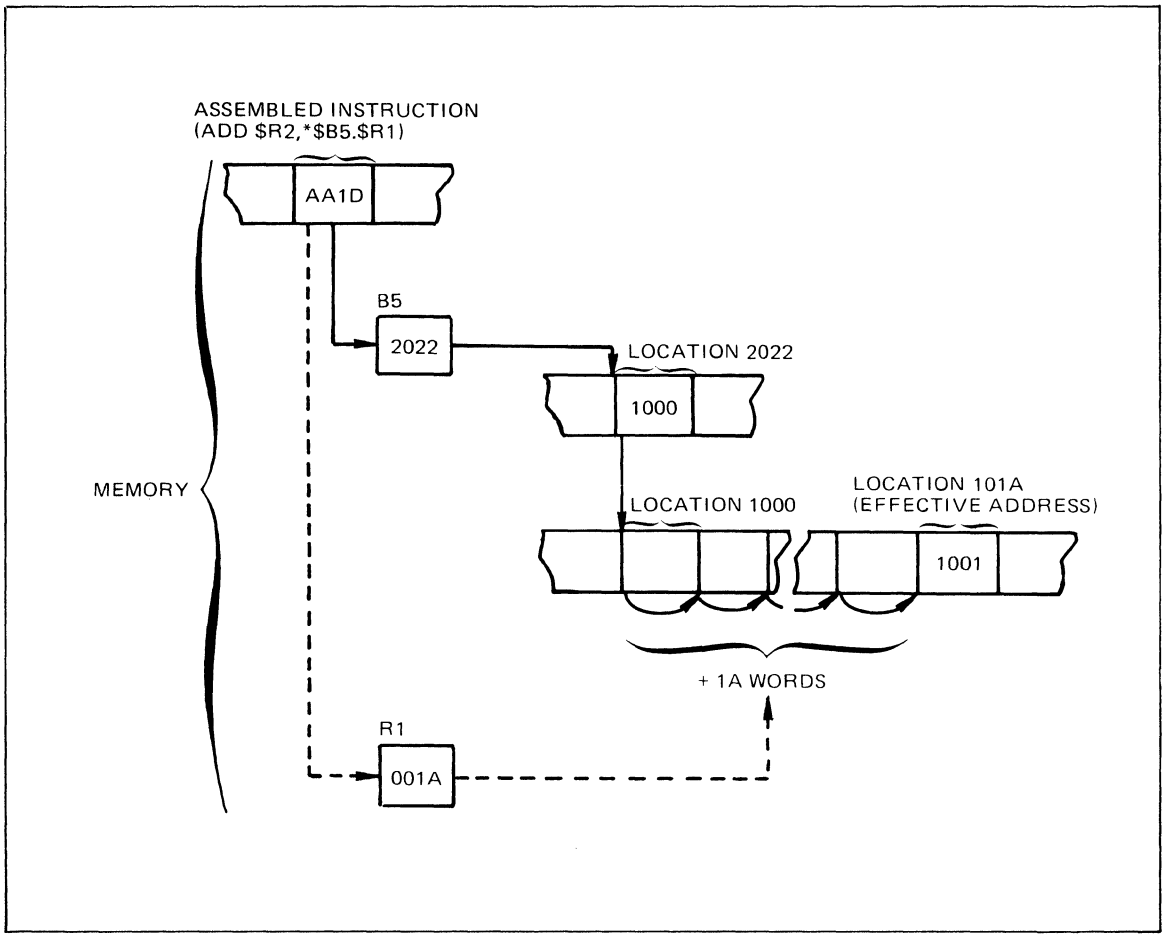


Figure 5-12. Indexed Indirect B-Relative Addressing

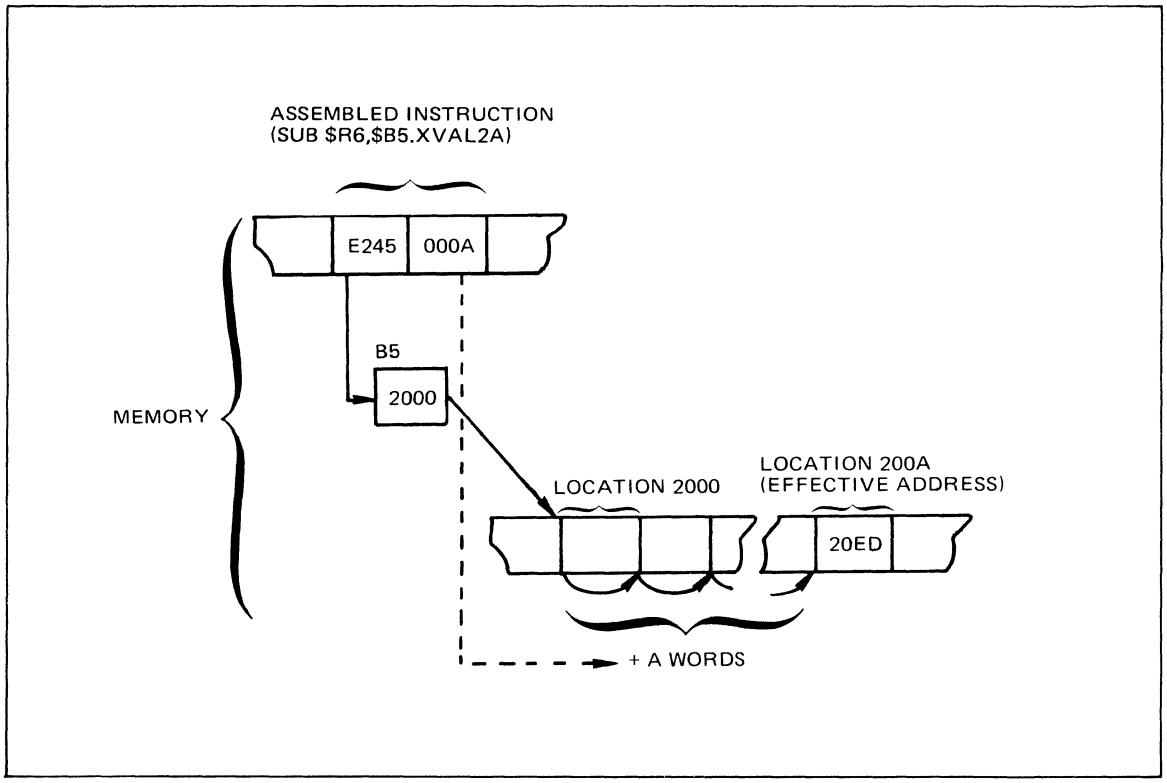


Figure 5-13. Direct B-Relative Plus Displacement Addressing

INDIRECT B-RELATIVE PLUS DISPLACEMENT ADDRESSING

This form of addressing effectively 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.

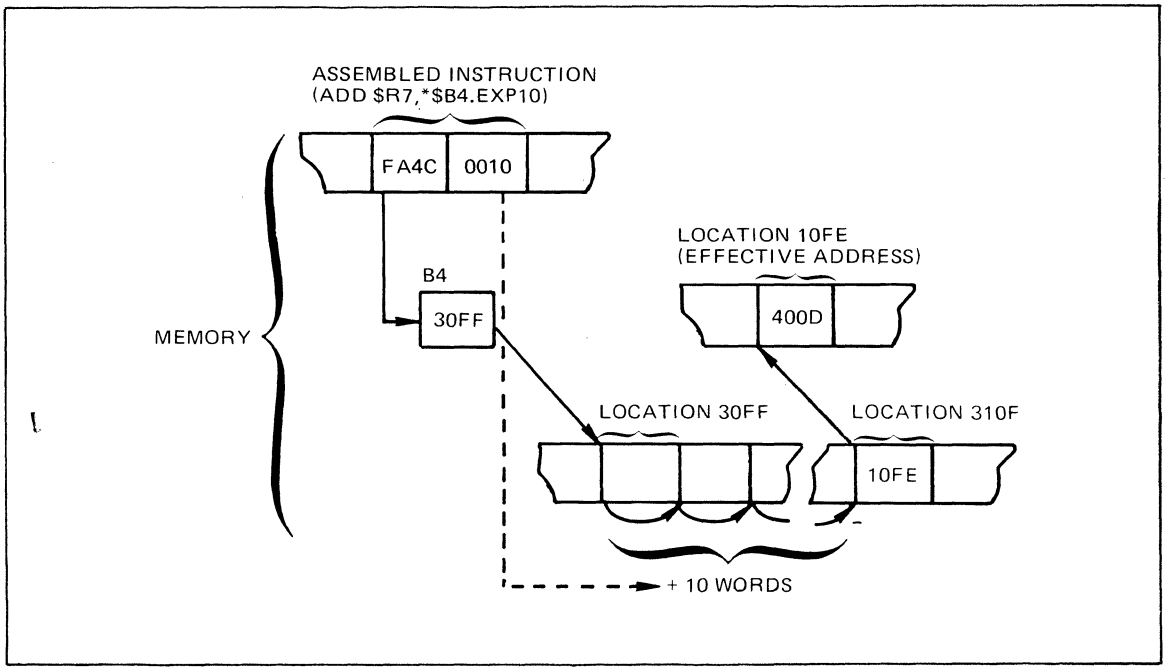


Figure 5-14. Indirect B-Relative Plus Displacement Addressing

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 Appendix M. 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.

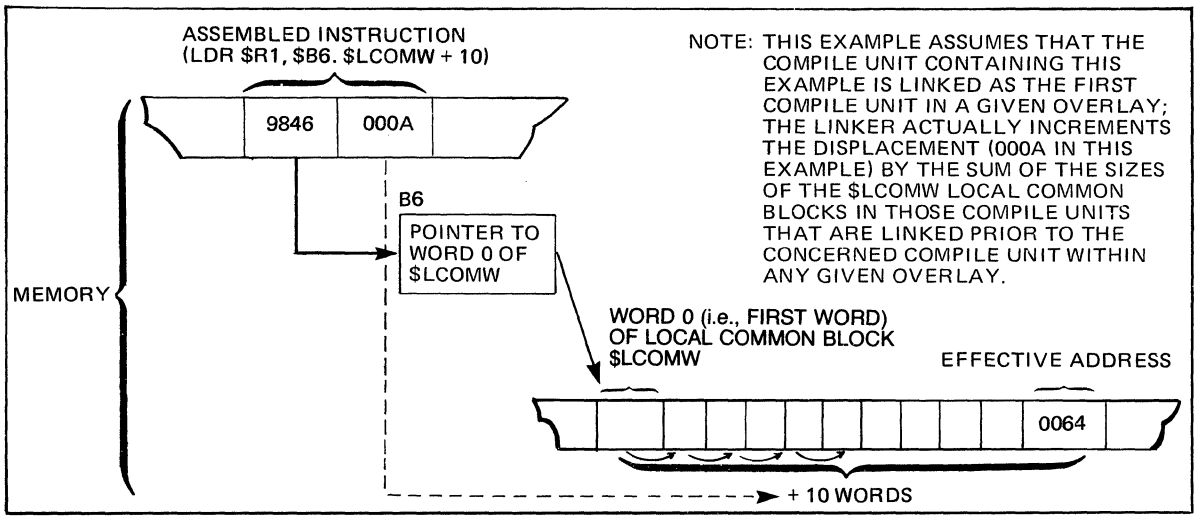


Figure 5-15. Direct B6-Relative Plus Local Common Block Plus Displacement Addressing

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. 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.

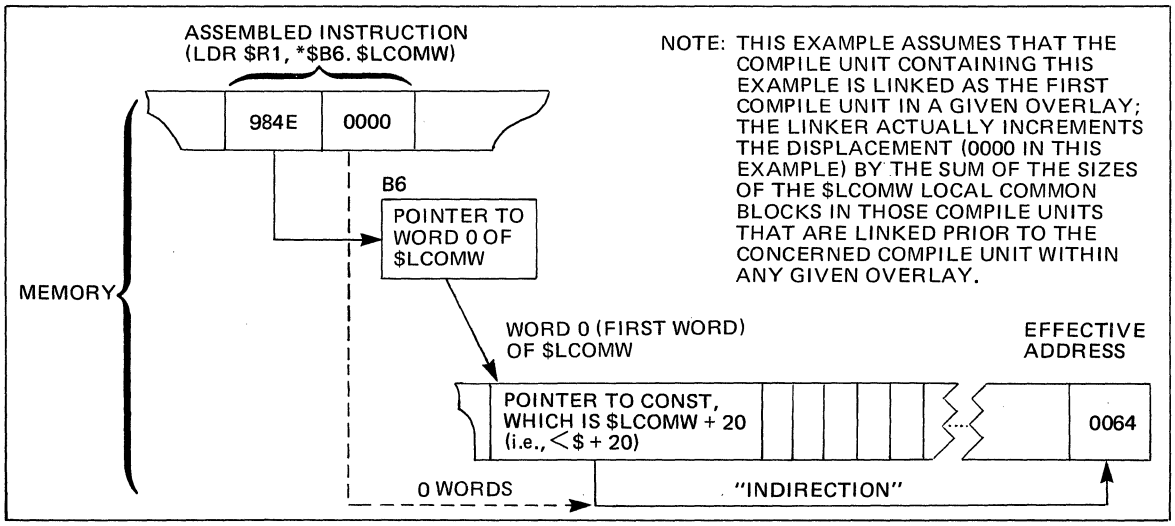


Figure 5-16. Indirect B6-Relative Plus Local Common Block Plus Displacement Addressing

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.

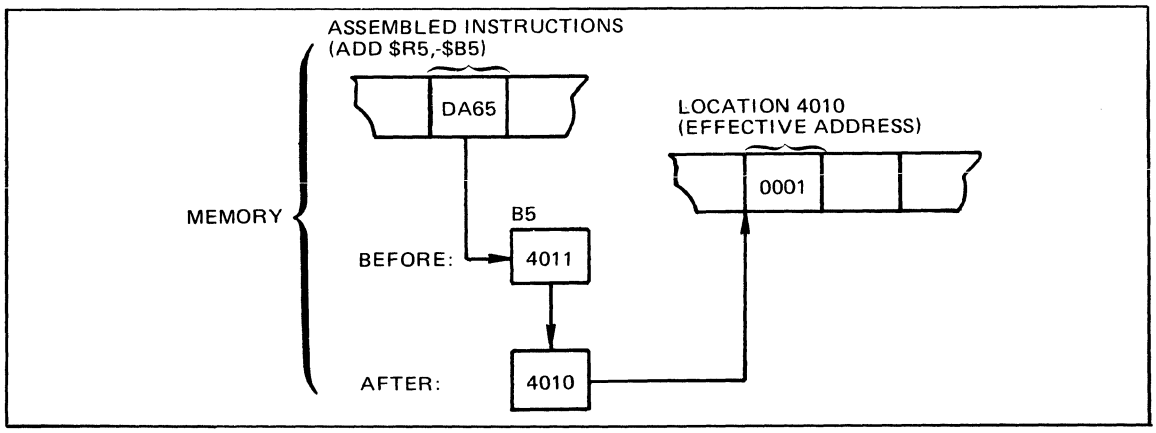


Figure 5-17. B-Relative Push Addressing

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:

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

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 (lower memory address, see Figure K-1).

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 Models 40 and 50.

ADD

Instruction:

Add Contents to R-register

Type:

DO

Source Language Format:

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

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:

=$Bn, =$Sn register addressing
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:

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

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

Instruction:

Add integer double

Type:

SO

Source Language Format:

ΔAIDΔaddress-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:

=$Bn
=$Sn } 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). In all cases, the immediate operand must be a constant that has not been assigned a symbolic name.

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 Models 40 and 50.

ENT

Instruction:

Enter

Type:

SO

Source Language Format:

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

Description:

Jumps to the memory location specified by the operand; also, sets the P-bit of the ring field in the S-register 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

Instruction:

Halt

Type:

GE

Source Language Format:

ΔHLT

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.

LDI

Instruction:

Load double-word integer

Type:

SO

Source Language Format:

ΔLDIΔaddress-expression

Description:

Loads the contents of the location specified by the address expression into register R6 and the contents of the next location into register R7.

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

=$Bn } register addressing
=$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). In all cases, the immediate operand must be a constant that has not been assigned a symbolic name.

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

LDR

Instruction:

Load R-register

Type:

DO

Source Language Format:

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

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:

$$\Delta LDT\Delta \left\{ \begin{matrix} \$Bn \\ X'n' \\ n \end{matrix} \right\}$$

Description:

Loads the T register with the address contained in $Bn (see Figure K-1).

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 Models 40 and 50.

LDV

Instruction:

Load value

Type:

SI

Source Language Format:

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

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.

RLQ

Instruction:

Relinquish stack space

Type:

GE

Source Language Format:

ΔRLQΔ$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, see Figure K-1), 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 Models 40 and 50.

RSTR

Instruction:

Restore context

Type:

SO

Source Language Format:

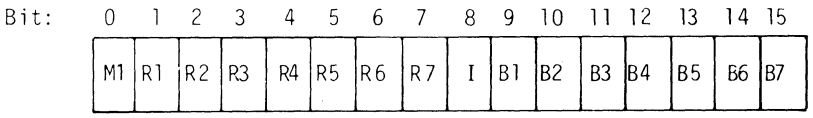
$$\Delta\text{RSTR}\Delta \begin{Bmatrix} \text{immediate-memory-address} \\ \text{B-relative-address} \\ \text{P-relative-address} \\ \text{interrupt-vector-addressing} \end{Bmatrix}, \begin{Bmatrix} \text{external-value-label} \\ \text{internal-value-expression} \\ \text{single-precision-fixed-point-constant}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

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:



This mask should be the same as the one used to save the registers (see the SAVE instruction).

SDI

Instruction:

Store Double word integer

Type:

SO

Source Language Format:

ΔSDIΔaddress-expression

Description:

Stores the contents of register R6 into the location specified by the address expression and the contents of register R7 into the next location.

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

=$Bn
=$Rn } register addressing
=$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). In all cases, the immediate operand must be a constant that has not been assigned a symbolic name.

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

Instruction:

Subtract integer double

Type:

SO

Source Language Format:

ΔSIDΔaddress-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:

=$Bn, =$Sn } register 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). In all cases, the immediate operand must be a constant that has not been assigned a symbolic name.

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 Models 40 and 50.

STS

Instructions:

Store S-register

Type:

SO

Source Language Format:

ΔSTSΔaddress-expression

Description:

Stores the contents of the system status (s) register in the location or R-register identified in the address expression.

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

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

STT

Instruction:

Store Stack Address Register

Type:

GE

Source Language Format:

ΔSTT

Description:

This stack instruction moves the address contained 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 Models 40 and 50.

ALR

Instruction:
Alphanumeric move

Type:
Character string

Source Language Format:

$$\Delta\text{ALR}\Delta \left\{\begin{matrix}\text{DESCA(description)} \\ \text{int-val-expression}\end{matrix}\right\} , \left\{\begin{matrix}\text{DESCA(description)} \\ \text{int-val-expression}\end{matrix}\right\}$$

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:

$$\Delta\text{AME}\Delta \left\{\begin{array}{l}\text{DESCA(description)}\\ \text{int-val-expression}\end{array}\right\}, \left\{\begin{array}{l}\text{DESCA(description)}\\ \text{int-val-expression}\end{array}\right\}, \left\{\begin{array}{l}\text{DESCA(description)}\\ \text{int-val-expression}\end{array}\right\}$$

Description:

The character string 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.

DRS

Instruction:

Decimal right shift

Type:

Shift

Source Language Format:

$$\Delta DRS\Delta \left\{\begin{matrix}DESCP(description)\\DESCU(description)\\int\text{-}val\text{-}expression\end{matrix}\right\} \quad [[,int\text{-}val\text{-}expression] \; [,R[OUNDED]]]$$

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:

$$\Delta\text{DSB}\Delta \begin{Bmatrix} \text{DESCP(description)} \\ \text{DESCU(description)} \\ \text{int-val-expression} \end{Bmatrix}, \begin{Bmatrix} \text{DESCP(description)} \\ \text{DESCU(description)} \\ \text{int-val-expression} \end{Bmatrix}$$

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 ngative 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.

MAT

Instruction:

Alphanumeric move and translate

Type:

Character string

Source Language Format:

$$\Delta MAT\Delta \left\{\begin{array}{l}\text{DESCA(description)}\\ \\ \text{int-val-expression}\end{array}\right\},\left\{\begin{array}{l}\text{DESCA(description)}\\ \\ \text{int-val-expression}\end{array}\right\},\left\{\begin{array}{l}\text{DESCA(description)}\\ \\ \text{int-val-expression}\end{array}\right\}$$

Description:

The character string 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:

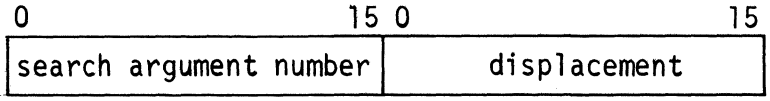
Character string

Source Language Format:

$$\Delta\text{SRCH}\Delta \left\{\begin{matrix}\text{DESCA(description)}\\ \text{int-val-expression}\end{matrix}\right\}, \left\{\begin{matrix}\text{DESCA(description)}\\ \text{int-val-expression}\end{matrix}\right\}, \left\{\begin{matrix}\text{DESCA(description)}\\ \text{int-val-expression}\end{matrix}\right\}$$

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). (This simulator does not support the Alphanumeric Search (SRCH) and Alphanumeric Verify (VRFY) opcodes.) 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.

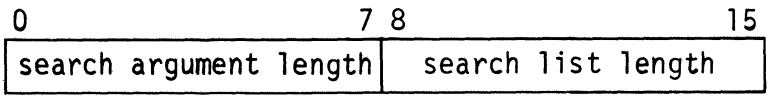


If a match is not found, the G-bit of the Commercial Processor indicator register is cleared to zero, the L-bit is set to one, and the receiving field is not changed.

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.

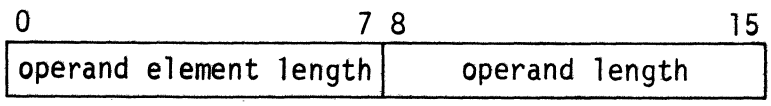


If the search argument length is equal to the search list length, the search list consists of only one argument.

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.



The results of a search instruction for this array and various search arguments are as follows.

	Commercial Processor Indicator Register		*DD2 Field*	
SA	*L-Bit*	*G-Bit*	*SA Number*	*Displacement*
ca	0	0	0	08
a	0	0	0	00
mjo	0	0	0	10
mjpo	1	0	unchanged	
acbec	0	0	0	04
eacba	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.

Displacement	*String*
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.

			Commercial Processor Indicator Register		*DD2 Field*	
SAL	*SLL*	*SA*	*L-Bit*	*G-Bit*	*SA Number*	*Displacement*
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

VRFY

VRFY

Instruction

Alphanumeric verify

Type:

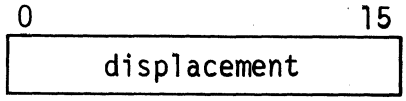
Character string

Source Language Format:

$$\Delta\text{VRFY}\Delta \left\{\begin{matrix}\text{DESCA(description)}\\ \text{int-val-expression}\end{matrix}\right\} ,\left\{\begin{matrix}\text{DESCA(description)}\\ \text{int-val-expression}\end{matrix}\right\} ,\left\{\begin{matrix}\text{DESCA(description)}\\ \text{int-val-expression}\end{matrix}\right\}$$

Description:

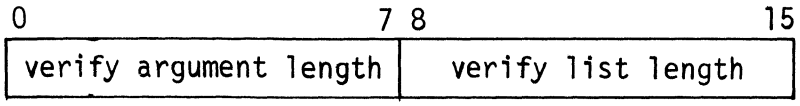
The character string or array of character strings defined by the third data descriptor (DD3) is examined. If at least one character of the string (or element of the array) does not match any one of the verify arguments, the G-bit of the Commercial Processor indicator register is cleared to zero, the L-bit is set to one, and the receiving field specified by the second data descriptor (DD2) is loaded with the displacement. (This simulator does not support the Alphanumeric Search (SRCH) and Alphanumeric Verify (VRFY) opcodes.) 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.

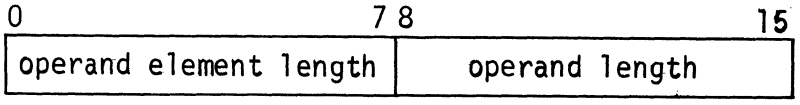


If the verify argument length is equal to the verify list length, the verify list consists of only one argument.

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.



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 *Systems Messages* manual.) The pound sign (#) and the at sign (@) designate macro processor comment lines. Upon request the macro processor generates comment lines that begin with the at sign (@). These lines are macro control statements without errors. The macro processor unconditionally generates comment lines that begin with the pound sign (#). These lines are statements with macro processing errors contained in them.

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

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.

Expanded source module:

```
TITLE EXMPL
.
.
.
LDV $R1,=5      }  Macro call replaced by contents of macro
LDR $R2,='6,'   }  routine named SAMPLE
.
.
.
```

PROTECTION OPERATORS

Protection operators are brackets; they enclose one or more characters that are not to be interpreted by the Macro Preprocessor. Protection operators can be included in macro routines and/or in statements that constitute the body of a source 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 inter-
                              preted by Macro Preprocessor, so no value
                              will be substituted
            NEWB ?P7          Reference to P7 will be replaced with its
                              value
            ENDM              Designates end of macro routine
            .
            .
            .
            [SAMPLE]          Not interpreted as macro call because name
            .                 of macro routine is enclosed within protec-
            .                 tion operators
            .
            SAMPLE            Macro call; in the expanded source module
            .                 will be replaced by contents of macro rou-
            .                 tine named SAMPLE
            .
```

Expanded source module:

```
TITLE EXMPL
.
.
.
SAMPLE
.
.
.
NEWA ?P7    }  Contents of macro routine named SAMPLE
NEWB 3      }
.
.
.
```

Protection operators cannot extend over operand or argument delimiters; to protect adjacent operands or arguments, enclose each one individually in brackets.

Example 1:

FOOΔ[AB],[CD]

The above macro call FOO designates that parameter P1 equals [AB] and parameter P2 equals [CD].

Example 2:

FOOΔ[AB,CD]

The above macro call FOO is *not* equivalent to the macro call illustrated in example 1. The macro call in example 2 specifies that parameter P1 equals the character string consisting of the following three characters: [AB, and parameter P2 equals the character string consisting of the following three characters: CD].

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 DCΔ'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.)

REQUOTE MACRO FUNCTION

The requote macro function replaces each apostrophe in an alphanumeric character string with two adjacent apostrophes, and encloses the entire resultant string within single apostrophes. All characters in the original character string that are not apostrophes remain unchanged and appear in the resultant string.

Format:

?RQ(arg)

arg

An alphanumeric character string to be requoted. (See "Designating Alphanumeric Values" in this section.)

Example 1:

?RQ(ABC) yields
'ABC'

Example 2:

?RQ('WHO') yields

"'WHO'"

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	
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	{=$Rn, =$Bn, =$Sk}	$Bk.-$R1	$Bq.+$R1
110	reserved	reserved	-$Bn	$Bk.-$R2	$Bq.+$R2
111	{=location, =value}	$IV. value	+$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		=$Rn =$Bn =$Sn	Rn = EA Bn = EA Sn = EA
Immediate Memory Addressing	Direct	< { locexpression / { + / − } templabel }	location = EA
	Indirect	*< { locexpression / { + / − } templabel }	location = EA
	Indexed Direct	< { locexpression / { + / − } templabel } .$R { 1 / 2 / 3 }	location + R { 1 / 2 / 3 } = EA
	Indexed Indirect	*< { locexpression / { + / − } templabel } .$R { 1 / 2 / 3 }	location + R { 1 / 2 / 3 } = EA
Immediate Operand Addressing		=locexpression =stringconstant = { intvalexpression / extvallabel }	Address of current address syllable + 1 = EA
P-Relative Addressing	Direct	{ intlocexpression / { + / − } templabel }	internal location = EA
	Indirect	* { intlocexpression / { + / − } templabel }	internal location = EA
B-Relative Addressing	Direct	$Bn	Bn = EA
	Indirect	*$Bn	Bn = location location = EA
	Indexed Direct	$Bn.$R { 1 / 2 / 3 }	Bn + R { 1 / 2 / 3 } = EA
	Indexed Indirect	*$Bn.$R { 1 / 2 / 3 }	Bn = location location + R { 1 / 2 / 3 } = EA
	Direct + Displacement	$Bn. { intvalexpression / extvallabel }	Bn + 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-Relative Addressing (Cont.)	Indirect + Displacement	*$Bn. {intvalexpression / extvallabel}	Bn + value = location location = EA
	B6 direct + Displacement	$B6.$LCOMW + intvalexpression	B6 + value = EA
	B6 indirect + Displacement	*$B6.$LCOMW + intvalexpression	B6 + value = location location = EA
	Push	-$Bn	Bn ← (Bn - 1) Bn = EA
	Pop	+$Bn	Bn = EA Bn ← (Bn + 1)
	Indexed Push	$B {1/2/3} .-$R {1/2/3}	R{1/2/3} ← (R{1/2/3} - 1) B{1/2/3} + R{1/2/3} = EA
	Indexed Pop	$B {1/2/3} .+$R {1/2/3}	B{1/2/3} + R{1/2/3} = EA R{1/2/3} ← (R{1/2/3} + 1)
Short Displacement		>{intlocexpression / {+/-} templabel}	location = EA
Special		>={intvalexpression / extvallabel}	Word following the word(s) containing op code + first operand address syllable = EA
Interrupt Vector		$IV.{intvalexpression / extvallabel}	IV + value = EA

NOTE: The symbols used in this table have the following meanings:

( )	Contents of . . .	*	Indirect memory addressing
EA	Effective Address	<	Immediate memory addressing
←	Replaces the element pointed at	>	Short displacement addressing
locexpression – location expression (any type)		>=	Specified Addressing Component separator (indexing and displacement)
templabel – temporary label			
stringconstant - string constant			
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 C

Sample Assembly Language Program

The following sample programs illustrate many of the aspects of the assembly language described in this 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          0100                            COMM      X'100'
000010                                  * GET FILENAME  AND CHANNEL NO
000011  0000    A843 FFEF               START     LDR       $R2,$B3.-17
000012  0002    BBA3                              LAB       $B5,$B3.$R2
000013  0003    BBC3 FFEC                         LAB       $B3,$B3.-20         SET B3 TO LIST FILE AT
000014  0005    9873                              LDR       $R1,+$B3            SET B3 TO FILENAME
000015  0006    1D02                              CMV       $R1,2
000016  0007    0981 007C                         BNE       ERNLST              NO LIST FILE ATTACHED
000017  0009    9843 0007                         LDR       $R1,$B3.7           SET R1 TO CHANNEL NO.
000018                                  * OPEN LIST FILE
000019  000B    CBC0 0079                         LAB       $B4,LSTDCB
000020  000D    D380 0000       X                 LNJ       $B5,<ZIOSOL         OPEN ROUTINE
000021  000F    1981 006F                         BNEZ      $R1,EROPEN
000022                                  * WRITE HEADER MSG
000023  0011    1C1E                              LDV       $R1,X'1E'           MSG LENGTH
000024  0012    2C00                              LDV       $R2,X'0'
000025  0013    BBC0 0081                         LAB       $B3,MBUF01          MSG ADDRESS
000026  0015    CBC0 006F                         LAB       $B4,LSTDCB
000027  0017    D380 0000       X                 LNJ       $B5,<ZIOSWR         WRITE ROUTINE
000028  0019    1981 0066                         BNEZ      $R1,ERWDR
000029  001B    3CFF                              LDV       $R3,-X'1'
000030  001C    3E01                    TLOOP     ADV       $R3,X'1'
000031  001D    B970 0000       X                 CMR       $R3,=TSTMAX         CHECKED ALL TEST RESULTS ?
000032  001F    0301 004E                         BG        ENDTST
000033  0021    CCB0 0000       R                 LDB       $B4,<$COMM.$R3
000034  0023    CBC4 001C                         LAB       $B4,$B4.X'1C'       CREATE STATUS BLOCK PTR
000035  0025    F830 0000       X                 LDR       $R7,<TABLOC.$R3     GET EXPECTED VALUE
000036  0027    F944 0003                         CMR       $R7,$B4.X'3'        COMPARE TO ACTUAL STATWD
000037  0029    0973                              BE        >TLOOP              TEST OK - CHECK NEXT TEST
000038  002A    EBC0 007A                         LAB       $B6,MBUF2A
000039  002C    D830 0000       R                 LDR       $R5,<$COMM.$P3
000040  002E    F3C0 0027                         LNJ       $B7,DUMPWD          CONVERT TEST ADDR TO ASCII
000041  0030    EBC0 0077                         LAB       $B6,MBUF2B
000042  0032    D804                              LDR       $R5,$B4
000043  0033    F3C0 0022                         LNJ       $B7,DUMPWD          CONVERT SYML VALUE TO ASCII
000044  0035    CBC4 0001                         LAB       $B4,$B4.X'1'
000045  0037    EBC0 0073                         LAB       $B6,MBUF2C
000046  0039    D804                              LDR       $R5,$B4
000047  003A    F3C0 001B                         LNJ       $B7,DUMPWD          CONVERT TEST NUM TO ASCII
000048  003C    CBC4 0001                         LAB       $B4,$B4.X'1'
000049  003E    EBC0 006F                         LAB       $B6,MBUF2D
000050  0040    D804                              LDR       $R5,$B4
000051  0041    F3C0 0014                         LNJ       $B7,DUMPWD          CONVERT SYMV VALUE TO ASCII
000052  0043    CBC4 0001                         LAB       $B4,$B4.X'1'
000053  0045    EBC0 006B                         LAB       $B6,MBUF2E
000054  0047    D804                              LDR       $R5,$B4
000055  0048    F3C0 000D                         LNJ       $B7,DUMPWD          CONVERT STATUS WORD TO ASCII
000056                                  * WRITE VALUES
000057  004A    1C1E                              LDV       $R1,X'1E'           MSG LENGTH
000058  004B    2C00                              LDV       $R2,X'0'
000059  004C    BBC0 0057                         LAB       $B3,MBUF20          MSG ADDRESS
000060  004E    CBC0 0036                         LAB       $B4,LSTDCB
```

Figure C-1. Listing of CHKNML Program

```
CHKNML —SAF 1977/11/21 0940 : 05.6 ASSEMBLER—0100—11/09/1223 GCOS6 MD0400—S100—11/17/0634 PAGE 0002

000061  0050  D380 0000      X                LNJ       $B5,<ZIOSWR          WRITE ROUTINE
000062  0052  1981 002E                       BNEZ      $R1,ERVAL
000063  0054  83C0 FFC7                       JMP       TLOOP
000064                                * ROUTINE ACCEPTS A VALUE IN R5 AND PUTS ITS ASCII EQUIVALENT
000065                                * IN THE TWO WORDS POINTED TO BY R6
000066  0056  4CFC                    DUMPWD   LDV       $R4,-X'4'            SET COUNTER
000067  0057  CF40 0000      T                STR       $R4,+$C
000068  0059  7C00                             LDV       $R7,X'0'
000069  005A  4C00                    $A       LDV       $R4,X'0'
000070  005B  5084                             DOL       $R5,4
000071  005C  4E30                             ADV       $R4,X'30'
000072  005D  C940 0000      T                CMR       $R4,+$F
000073  005F  0380           T                BLE       >+$E
000074  0060  4E07                             ADV       $R4,X'07'
000075  0061  F454                    $E       OR        $R7,=$R4
000076  0062  8AC0 FFF5      T                INC       +$L
000077  0064  0600            T                BCT       >+$D
000078  0065  7088                             DOL       $R7,8
000079  0066  0FF4            T                B         >-$A
000080  0067  EF46 0000                 $D       STR       $R6,$B6.X'0'
000081  0069  FF46 0001                          STR       $R7,$B6.X'1'
000082  006B  8387                              JMP       $B7                  RETURN TO CALLER
000083  006C  0000                    $C       DC        Z'0'
000084  006D  0039                    $F       DC        Z'0039'
000085                                * WRITE END TEST
000086  006E  1C0A                    ENDTST   LDV       $R1,X'A'             MSG LENGTH
000087  006F  2C00                             LDV       $R2,X'0'
000088  0070  BBC0 0043                         LAB       $B3,WBUF03           MSG ADDRESS
000089  0072  CBC0 0012                         LAB       $B4,LSTDCB
000090  0074  D380 0000      X                LNJ       $B5,<ZIOSWR          WRITE ROUTINE
000091  0076  1981 000B                       BNEZ      $R1,EREND
000092                                * CLOSE LIST FILE
000093  0078  CBC0 000C                       LAB       $B4,LSTDCB
000094  007A  D380 0000      X                LNJ       $B5,<ZIOSCO          CLOSE ROUTINE
000095  007C  1981 0006                       BNEZ      $R1,ERCLS
000096  007E  0000                             HLT
000097  007F  0000                    EROPEN   HLT
000098  0080  0000                    ERMDR    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

Appendix L
Assembly Language Program Independence

ASSEMBLY LANGUAGE PROGRAM HARDWARE INDEPENDENCE

If an assembly language program written for a Series 6/20 or 6/30 is to be used on a Series 6/40 or 6/50, the program must be written to be program independent of the hardware model. The additional features in the larger Series 6/40 and 6/50 that must be considered are instruction prefetching that affects self-modifying procedures and long address form (LAF). The GCOS 6 MOD 400 Linker produces SAF, LAF, or SAF-LAF Independent Code (SLIC) bound units.

SELF-MODIFYING PROCEDURES

Use of a self-modifying procedure should be carefully considered for two reasons: (1) a self-modifying procedure cannot be made reentrant, and (2) the instruction, as modified, might not be executed because of the instruction prefetching feature of the Series 6/40 and 6/50. With instruction prefetching, an arbitrary number of words are prefetched in parallel with the execution of the current instruction. The prefetch buffer is emptied only when a transfer of control occurs. If an instruction is stored in a word that previously was prefetched, the prefetch buffer is *not* cleared and the prefetched instruction will be executed as it was prior to modification.

However, if a self-modifying procedure must be used, the program must contain code to remove the prefetched instruction from the prefetch buffer after modification is complete but before the modified code is executed. This can be done by executing an unconditional branch of the form:

```
B  $+2  FLUSH THE PREFETCH
```

WRITING SOURCE PROGRAMS THAT CAN BE EXECUTED IN BOTH SAF AND LAF CONFIGURATIONS

There are two methods for writing a source program so that it can be executed in both SAF and LAF configurations: SAF/LAF independence by assembly, which produces a program that is assembled differently for each type of configuration, and SAF/LAF independence by loading, which produces a program that is assembled and linked in the same way but is loaded differently. For the second method, SAF/LAF Independent Code (SLIC) is used to create the source program.

A SLIC program consists entirely of Assembler control statements, assembly instructions, and macro calls, all of which are described in the *Assembly Language Reference* manual. These items must be selected and combined according to the rules and restrictions described in the following text. SLIC is the code that results from this procedure.

As shown by Figure L-1, a program can run on a SAF and LAF configuration, if all the compilation units are SLIC compilation units and linking is done by a GCOS 6 MOD 400 Linker. When requested, the Assembler produces SLIC compilation units. However, the Assembler does not check that the code conform to the SLIC rules and restrictions. If the code does not conform, the results of the program are unpredictable.

The valid ways in which SAF and SLIC compilation units and LAF and SLIC compilation units can be linked into bound units are shown in Figure L-2.

The following system service macro calls should not be used in a program written in SAF/LAF independent code (SLIC):

$CRB	$PRBLK	$TRB	$MGCRB	$MGCRT
$CRBD	$RBD	$TRBD	$MGIRB	$MGIRT
$IORB	$SRB	$WAITL	$MGRRB	$MGRRT
$IORBD	$SRBD	$WLIST		

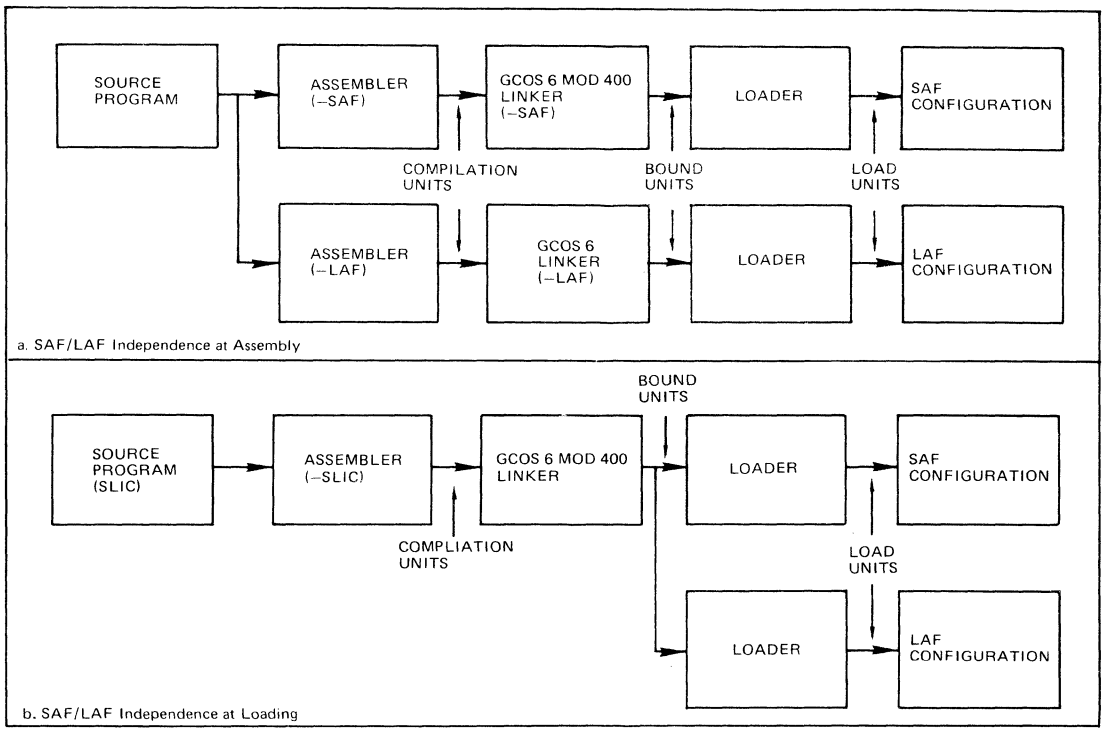


Figure L-1. Methods of Achieving SAF/LAF Independence

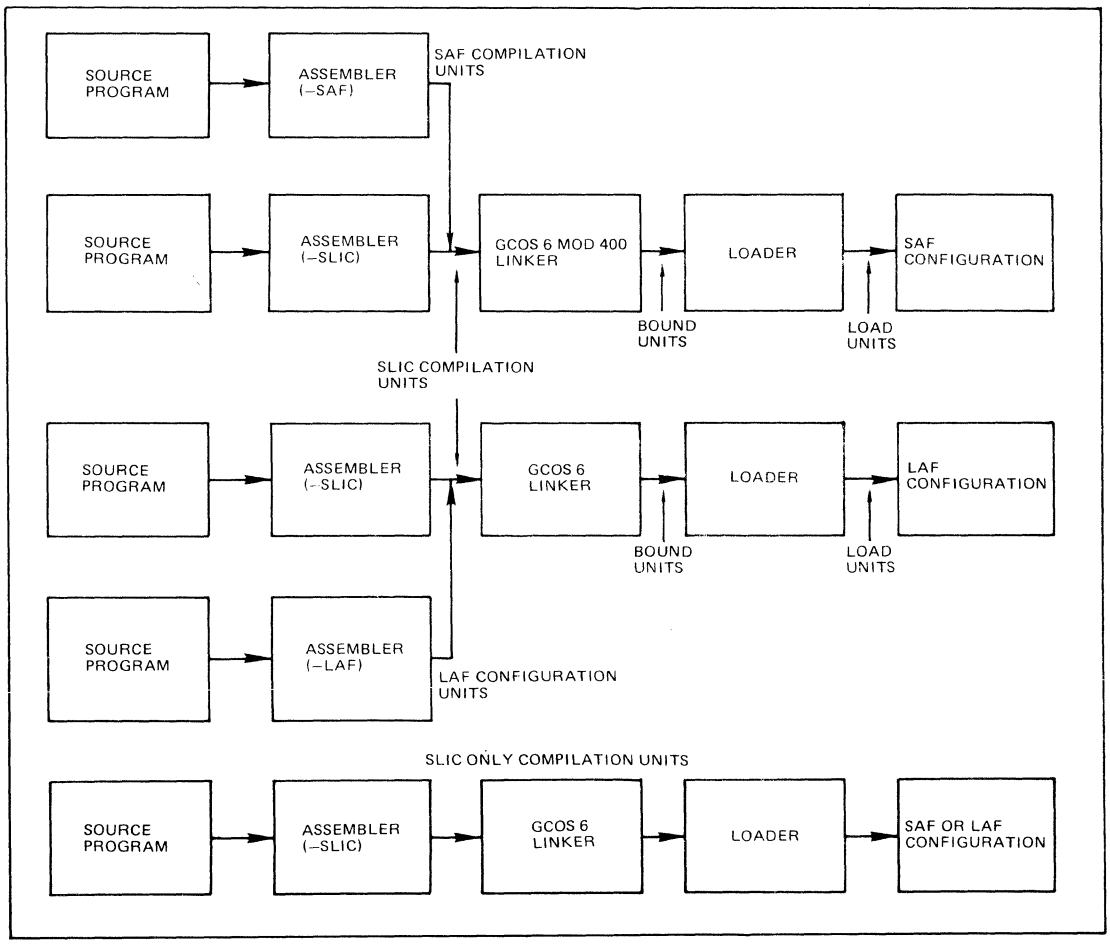


Figure L-2. Valid Combinations of Compilation Units for Linking

SAF/LAF INDEPENDENCE BY ASSEMBLY

An assembly language program assembled to execute under a SAF system can be converted to execute under a LAF system by simply reassembling the program for execution on the LAF system. Reassembly is usually possible *provided that the following rules are observed when the program is written.*

1. The program must be written so it will assemble without errors in either configuration; e.g., a short displacement branch must satisfy the conditons $-64 \leq d \leq 1$ or $2 \leq d \leq 63$ words on the LAF configuration as well as on the SAF configuration.
2. All memory locations should be referenced by their symbolic names. The assembly language label $AF can also be used in expressions to correctly reference the desired memory location; however, the $AF reference should be used with care since its use requires a good understanding of how the hardware operates.
3. Offsets to elements of a data structure containing pointers must be defined symbolically. When the data structure actually exists in another program, the assembly language label $AF can be used in an equate statement to provide the proper template.
4. All constants used in index computation to reference arrays of structures containing pointers must be symbolically defined.

   For example: if the span of an array element is "a" words plus "b" addresses, then the constant should be defined by the expression a+b*$AF. This constant can then be used to compute an index register value which is in turn used in a LAB instruction to set a base register to the beginning of the desired occurrence of the array element.
5. All fields that are to contain pointers must be defined as address constants or a reserve of $AF words. Such fields must be referenced by their symbolic names.
6. All external procedure calling sequences that modify their argument list must be designed to operate correctly, through the use of $AF, whether assembled for a SAF or LAF configuration.
7. The size of a common block that contains pointers must be specified by an expression involving the label $AF to give the correct size, whether the program is assembled for a SAF or LAF configuration.
8. All address manipulation must be performed using base registers (B1—B7). The LAB instruction with base plus displacement or base plus index addressing is useful for address manipulation.

SAF/LAF INDEPENDENCE BY LOADING

This section contains rules for writing assembly language programs that can be executed (without reassembly or relinking, but with suitable modifications by the loader) in either a SAF or LAF configuration. That is, the source language program can be assembled and linked into a bound unit. This bound unit can then be loaded and executed on either a SAF or LAF configuration.

DIFFERENCES BETWEEN SAF AND LAF

Memory is allocated and most memory addresses are determined by the Assembler or a compiler. SAF and LAF differ in their definition and use of memory addresses. This difference affects the following items:

1. Instructions or data whose size (space allocated) depends on the addressing mode; that is,
   a. Instructions that use IMA operands (and base register instructions that use IMO operands).
   b. Declarations of memory addresses as data; that is, address constants or address variables.
2. Data whose location in memory depends on the addressing mode; in particular, data structures whose address or format is determined by hardware specifications, such as interrupt and trap vector and save areas (IV, TV, ISA, TSA).

3. References to such instructions or data. The significant instances of this are:
   a. References to (sequences of) instructions using IMA operands.
   b. References to data structures containing pointers.
   c. References to data structures defined by hardware.

   Such references are resolved by (1) the Assembler or compiler (for most internal or common references), (2) the Linker (for some internal references, some common references, and external references), or (3) the loader (for some external references and for relocation).
4. Memory addresses, whether in instructions or in data declarations, that contain values prior to the start of execution. The significant instances of this are:
   a. IMA operands and IMO operands in base register instructions.
   b. Declarations of pointers with initial values; that is, address constants (DC <location-expression).

   These memory addresses must be examined because the value of the memory address must be resolved in a single word for SAF and in two words for LAF.
5. Addressing formats and instructions whose execution is different in the two addressing modes. Specifically, the addressing formats for indexing with or without pre-decrement or post-increment (the .$R, .+$R, .−$R types) and for push and pop (+$B and −$B) operate differently when used with the five base register instructions:
   LDB, STB, CMB, SWB, CMN

GENERAL RULES FOR WRITING SLIC PROGRAMS

1. Allocate two words for all memory addresses, whether they are instruction operands or data declarations. That is, generate or assemble essentially in LAF. This ensures that sufficient space is allocated to execute in LAF. (The Assembler will set $AF equal to 2 when invoked with the -SLIC control argument.)
2. When loading a SLIC program for execution in SAF, the loader will:
   a. Replace a sequence of (two word) pointers in an argument list or a pointer array by a sequence of one-word pointers followed by an equal number of one word NOPs. That is, the sequence is compressed into consecutive words. Adjustment of references to such argument lists and pointer arrays is *not* performed. In the case of an argument list, the control word is also adjusted appropriately.
   b. Replace an individual (two word) memory address, whether an instruction operand or a data item, by a single-word memory address followed by a one-word NOP. That is, the value is moved into the first of the two words. References to the leftmost of the two words work for both SAF and LAF execution.

PROCEDURES FOR WRITING SPECIFIC PARTS OF A SLIC PROGRAM

The following procedures for writing specific parts of a SLIC program are derived from the general rules described previously. Methods for handling data structures, pointers, argument lists, and other commonly used items are described.

ADDRESSING MODE

Invoke the language processor with the -SLIC argument. For the Assembler, this sets $AF equal to 2. Assembly language programs should use the ARGLST and PTRAY Assembler control statements to define argument lists and pointer arrays, respectively. The CALL statement will also generate an appropriately identified argument list. Individual pointers should be defined as address constants or by a RESV statement with the reserved label $AF.

DATA STRUCTURES CONTAINING POINTERS

The techniques used for declaring, allocating, and referencing data structures containing pointers differ somewhat, depending on the kind of data structure. The most commonly used data structures containing pointers are classified as follows:

- Data management structures (FIBs).
- Argument lists (in calls) and pointer arrays.
- Request blocks (RBs).
- Individual pointers.
- Hardware defined structures.

For FIBs, two words are allocated for each pointer whether execution is to be in SAF or LAF. For argument lists, pointer arrays, and request blocks, one word is allocated for each pointer when execution is to be in SAF or two words are allocated when execution is to be in LAF.

For argument lists and pointer arrays in a SLIC program, the loader compresses the sequence of two-word pointers into consecutive single-word pointers for execution in SAF. For request blocks, the loader does not compress the structure.

With this approach, software — including the operating system — has to deal with only one form for a given system data structure. For FIBs (and individual pointers), there is only one form, regardless of the addressing mode in which the program is executing. For argument lists, pointer arrays, request blocks, and hardware defined structures, a program executing in a given addressing mode receives only the form corresponding to that address mode.

An individual pointer must always be addressed by its first (or only) word. This is how the hardware works, and is why the loader moves the value into the first word when loading for execution in SAF. (Elements of an argument list or a pointer array, other than the first, cannot be referenced symbolically, as noted later.)

References to a pointer should be with instructions that explicitly operate on addresses; e.g., LDB. Other instructions, such as those that always operate upon two-word items, should be used carefully in a SLIC program. For example, arithmetic operations cannot be performed because when they are executed in SAF, the value of a pointer appears in the high-order position (first of the two words), not in the low-order position (second of the two words) appropriate for arithmetic.

DATA MANAGEMENT STRUCTURES (FIBS)

Data management structures (FIBs) must be allocated with two words for each pointer in them. A FIB is not compressed when loaded for execution in SAF; but the loader does move the value of the pointer from the second word into the first.

This kind of structure can be declared symbolically. Honeywell supplies the declaration as a macro for use in assembly language programs.

Data items, including pointers, can be referred to symbolically via the declaration. Refer to a data item by the label assigned to it or by an expression not using $AF.

When referring to pointers with base register instructions, do *not* use the indexed, push, or pop addressing formats. These addressing formats will not work with this kind of structure, because they index, increment, or decrement by one-word units when executing SAF and two-word units when executing in LAF.

An initial value can be declared for any data item, including pointers.

ARGUMENT LISTS AND POINTER ARRAYS

When argument lists and pointer arrays are used as system data structures (e.g., in inter-program communication), a standard form is required. Argument lists and pointer arrays use one-word (consecutive) pointers when they are executed in SAF. They must be allocated with two-word pointers in a SLIC program, so that it can be executed in LAF. However, they are compressed by the loader when they are loaded for execution in SAF. This permits them to be declared with initial values — in particular, it minimizes the need to assign values to arguments at execution time.

Thus, when a SLIC program executes in SAF, the pointers in an argument list or a pointer array occupy consecutive words, and the remainder of the structure is initialized to a sequence of one-word NOPs.

Although this kind of structure can be declared with initial values (because it will be altered appropriately by the loader for execution in SAF), it should be referenced only by base register instructions because the addresses of the pointers in it depend on the addressing mode used at execution time.

Assembly language programs should define argument lists via the CALL statement or through use of the ARGLST Assembler control statement. Pointer arrays should be defined by the PTRAY Assembler control statement.

The first pointer of an argument list or pointer array and the control word of an argument list can be referred to symbolically. When a SLIC program is loaded for execution in SAF, the pointers are compressed from a sequence of two-word values into a sequence of one-word (consecutive) values. As a result, references to other data items in this kind of structure must be computed at execution time.

Refer to a pointer in this kind of structure only with base register instructions with indexed, push, or pop addressing formats. These addressing formats will work because they index, increment, or decrement by one-word units when executing in SAF and two-word units when executing in LAF. For example, suppose there are n elements (arguments or elements of a pointer array), and the location named N contains the desired element number in the range 1 to n. Let register B7 point either to the argument list's control word or directly to the first word of the pointer array. Then, a convenient way of referring to the desired element is:

For argument lists		*For arrays*	
LAB	$B1, $B7.1	LAB	$B1, $B7
LDR	$R1,N	LDR	$R1, N
LDB	$B2, $B1.-$R1	LDB	$B2, $B1.-$R1

If the element number is known at assembly time, rather than being a variable as assumed in the code sequences above, then the references to N can be replaced by an immediate memory operand (=N) or the LDR may be replaced by an LDV if N≤127. Do *not* use the base register plus displacement addressing format (as in LDB $B2,$B1.N-1), because that addressing format does not adjust for addressing mode.

REQUEST BLOCKS (RBS)

A request block must be allocated with two words for each pointer in it when it is executed in LAF, but only one word for each pointer when it is executed in SAF. In a SLIC program, the two-word allocation is *not* compressed by the loader for execution in SAF.

A request block *cannot* be declared symbolically in a SLIC program. Since it has one-word pointers when it is executed in SAF and two-word pointers when it is executed in LAF, the same declaration cannot be used for execution in both addressing modes. This kind of structure must be constructed (have values placed in it) at execution time.

Data items (including pointers) in request blocks *cannot*, in general, be referred to symbolically. Since pointers occupy a different number of words in the two addressing modes, addresses within the structure are not known at assembly time. References to data items in a request block must be computed at execution time.

A convenient technique for constructing a request block is to step through it item by item, using the automatic incrementation addressing formats. When pointers are referenced, base register instructions can be used with the indexed, push, or pop addressing formats. These instructions work on either addressing mode because they use one-word units when executed in SAF and two-word units when executed in LAF. Do *not* use the LAB instruction with indexing, incrementation, or decrementation, since the LAB uses one-word units in both addressing modes.

An initial value *cannot* be declared for a data item in a request block.

INDIVIDUAL POINTERS

An individual pointer ("DC <location-expression" in assembly language) can be declared symbolically. Each pointer must be declared as an individual data item.

An individual pointer should be referred to by its label. However, as is the case in SAF/LAF independence by assembly, an individual pointer can also be referred to by a location expression involving the use of $AF.

When referring to an individual pointer (with a base register instruction), do not use the indexed, push, or pop addressing formats.

An initial value can be declared for an individual pointer. Thus, a SLIC program can contain individual address constants (as well as address constants in FIBs, argument lists, and pointer arrays).

HARDWARE-DEFINED STRUCTURES

Certain data structures are defined by the hardware. These structures have one-word pointers when executing in SAF, and two-word pointers when executing in LAF. Structures of this kind are:

- Base register areas used with SAVE and RSTR instructions. The same mask should be used to restore registers as was used to save them, and the save area must have two words reserved for each base register to be saved.
- Trap and interrupt vectors and save areas:

  User programs must reference trap save areas in the same way that request blocks are referenced; i.e., the addresses needed must be computed at execution time. Only the operating system is allowed to access the trap vectors, interrupt vectors, and interrupt save areas.
- Queue frames and stack headers:

  Queue frames and stack headers are treated the same as request blocks for the purpose of creating a SLIC program.

IMMEDIATE MEMORY ADDRESS OPERANDS

An immediate memory address (IMA) operand ("< location-expression" in assembly language) cannot be followed by other fields of the instruction because the loader would not move those other fields when loading for execution in SAF. The loader places the value of the IMA operand only into the first word, and sets the second word to a NOP.

This constraint applies to the following instructions:

- Input/output instructions — IO, IOH, and IOLD.
- Bit instructions — LB, LBC, LBF, LBS, and LBT; these instructions cannot be masked, but can be indexed if they use the IMA operand field.
- SAVE, RSTR, SRM.

Other instructions either do not allow IMA operands or have only one possible address operand and do not have control fields following, so they can be used without restriction.

ABSOLUTE ADDRESSES

Only the operating system and certain system programs such as Debug need to reference absolute memory locations. If any programs that need absolute addressing are to be written as SLIC programs, all absolute addresses must be generated at execution time.

Appendix M
Reentrant Programs

A program is defined as reentrant if a single copy of its code can be simultaneously executed by several tasks; the tasks may be in the same task group or in different task groups.

There are two categories of reentrant assembly language programs:

1. "Code only" programs that use no statically (permanently) allocated data storage (except the hardware registers). Data storage required by such programs is dynamically allocated. Normally, only system programs and small service subroutines (e.g.; binary to decimal conversion) are written this way.
2. Programs in which the code and statically allocated data are separated by the use of common blocks, with the allocation of static data storage being managed by the system.

Programs belonging to the second category are discussed below. It is assumed that the reentrant programs must operate in both MOD 400 and MOD 600. The use of dynamically allocated data storage is not discussed in this appendix.

A reentrant program defines the following three address spaces and their initial content:

- Pure code section
- Local data section
- Nonlocal data section

Pure code consists of pure procedures and constants. A pure procedure is one that never modifies any part of itself during execution. One copy of the pure code is shared by all users of the reentrant program.

Local data is data that has a scope of identification no greater than the source unit in which it is declared. All other data is considered to be nonlocal.

In an assembly language program, these three sections are identified as follows:

1. Anything that does not have its origin defined as any kind of a common block is part of the pure code section.
2. Anything that has its origin defined in the local common block named $LCOMW is part of the local data section.
3. Anything that has its origin defined in a local common block other than $LCOMW or in a nonlocal common block is part of the nonlocal data section.

The distinction of local data from nonlocal data is strictly for addressing purposes; the Linker combines them into a single load element.

The use of pointer data is restricted as follows:

1. A pointer, including IMA operands and IMO operands of the five base register instructions, in a pure code section may refer only to objects in a pure code section.
2. A pointer in a data section may refer only to objects in a data section; it may not refer to a pure code section.

All references made by executable code to local data must use B6 relative addressing. The first word of local data is referenced by $B6.$LCOMW, the second by $B6.$LCOMW+1, etc.

The program has no direct access to nonlocal data. Instead, the program must use indirect addressing through the local data to reference nonlocal data. Normally, this is done as follows:

1. Allocate a pointer in the local data section initialized with the address of the common block (or some location within it) to be referenced,
2. Load that pointer into a base register, other than B6, using B6 relative addressing, and

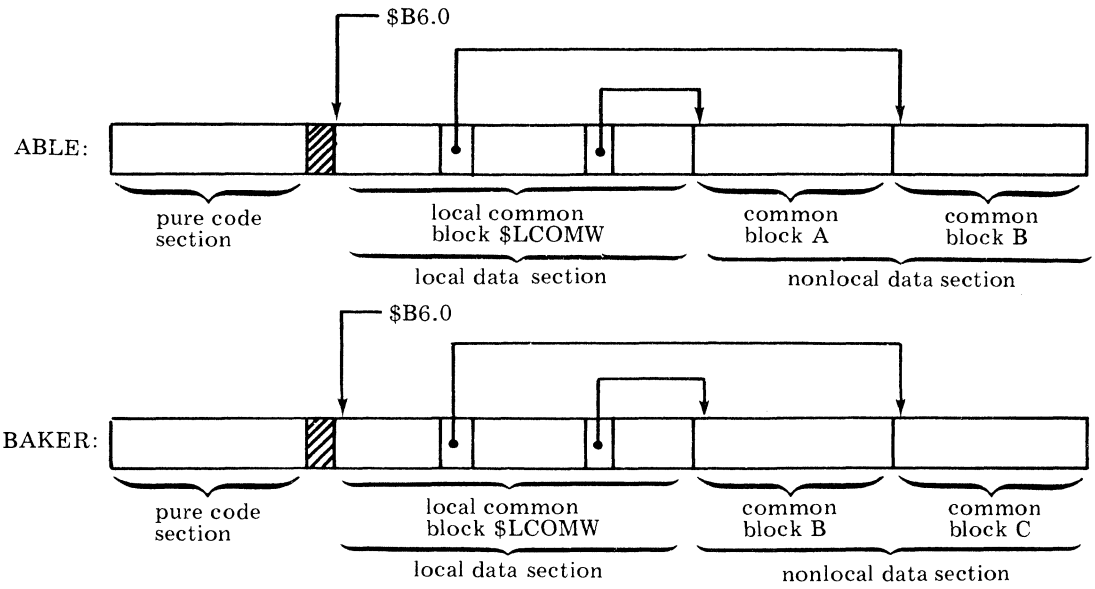
3. Access the nonlocal data using base relative addressing with the base register loaded in step 2.

The second step given above may be omitted when a base register is known to contain a pointer to the desired common block.

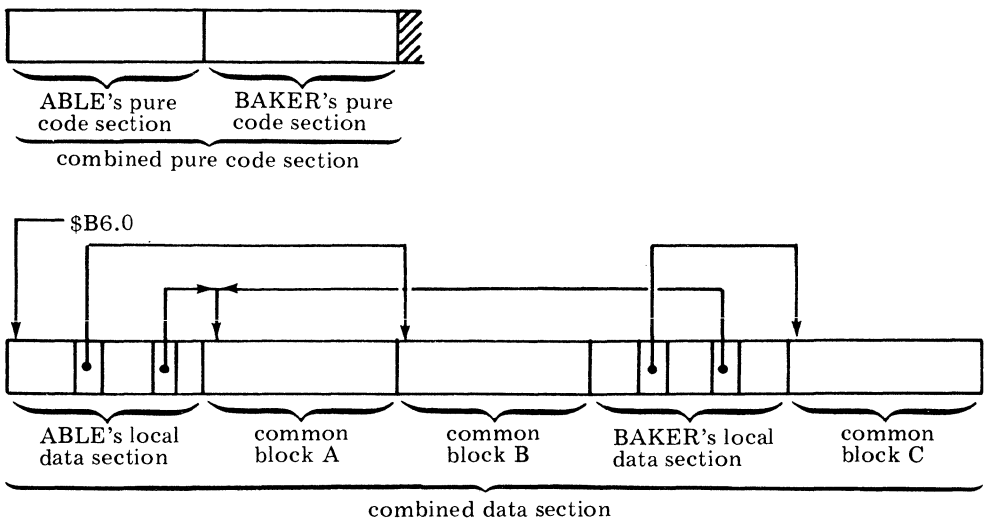
When a set of programs, written as described above, is linked in a "Link Separate environment", the Linker maintains the separation of code from data by placing the code and data in different load elements. The Linker also adjusts all B6 relative displacements, referring to local data, found in the code to reflect the positioning of that local data in the data load element. At execution time, the loading of a data load element causes B6 to be set to provide addressability to that data.

Example:

Assume there are two programs: ABLE and BAKER. Assume that ABLE declares and references common blocks A and B, and that BAKER declares and references common blocks B and C. The programmer will "see" ABLE and BAKER as shown below.



After ABLE and BAKER are linked as a bound unit, the executing code sees the following:



In the above illustration, the Linker has adjusted the displacement in all of BAKER's B6 relative references to its local data by the size of ABLE's local data plus the size of common block

A plus the size of common block B. This assumes that the bound unit was linked for MOD 400 and the size of the combined data section is less than 32K words. If the bound unit was linked for MOD 600 or the size of the combined data section is greter than 32K words, B6 would point to a location 32K words, B6 would point to a location 32K words further to the right, and the displacement in all of ABLE's and BAKER's B6 relative references to their local data would have an additional adjustment of −32768 words.

The preceding example only considered programs linked into the root of a bound unit. When a reentrant program has overlays, some formal call/return mechanism, such as the Call/Cancel/Exit Controller, must be used to save the calling overlay's B6 and set the called overlay's B6 on the call and to restore the calling overlay's B6 on the return.

The use of B6 relative addressing to reference local data places some limitations on the scope of data in a reentrant bound unit having overlays when compared to non-reentrant bound units. Data that may be referenced from a particular overlay of a bound unit linked in a "Link Separate environment" consists of:

1. That overlay's local data. This data may be referenced directly using B6 relative addressing.
2. That overlay's nonlocal data. This data may be referenced indirectly via a pointer contained in the overlay's local data.
3. When an overlay is formally called by another overlay, it may access any data received as a formal parameter using standard methods for accessing parameters.

The following is a summary of the rules for writing reentrant programs with statically allocated data.

1. Data must be separated from code through the use of common blocks and local common blocks. All local data must be placed in the local common block named $LCOMW (i.e., $LCOMW must be declared by the Assembler control statement "$LCOMW LCOMM int-val-expression").
2. In the executable code, all references to local data must be through B6 relative addressing; e.g., $B6.$LCOMW+int-val-expression.
3. In the executable code, all references to nonlocal data must be made via pointers (to the nonlocal data) contained in the local data.
4. The program must be linked in a "Link Separate environment". A "Link Separate environment" is specified by the -R control argument of the LINKER command in MOD 400 or through the use of LINKS Linker commands in MOD 600.

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

CUT ALONG LINE

TITLE: SERIES 60 (LEVEL 6) GCOS ASSEMBLY LANGUAGE REFERENCE ADDENDUM B

ORDER NO. CB07-01B

DATED JULY 1979

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here ☐ and furnish complete mailing address below.

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

CUT ALONG LINE

FOLD ALONG LINE

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell