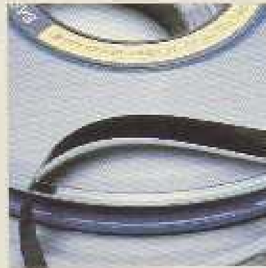
Prime Computer, Inc.

PR1ME

FDR3059-101B

Assembly Language Programmer's Guide

Rev. 16.3



The Assembly Language Programmer's Guide

The Assembly Language Programmer's Guide

by Rosemary Shields

Published by Prime Computer, Incorporated
Technical Publications Department
145 Pennsylvania Avenue, Framingham, MA 01701

Copyright © 1981 by Prime Computer, Inc.
Third Printing January 1981

All rights reserved.

The information contained in this document is subject to change without notice and should not be construed as a commitment by Prime Computer, Incorporated. Prime Computer assumes no responsibility for any errors that may appear in this document.

This document reflects the software as of Master Disk Revision Level 16.

PRIMOS® is a trademark of Prime Computer, Inc.

Credits.

Concept and Production

William I. Agush

Typesetting.

Allied Systems

Covers.

Mark-Burton

Text.

Eastern Graphics

1 INTRODUCTION

Introduction 1-1
Organization and usage 1-1
Related documents 1-1

2 CONVENTIONS

Prime conventions 2-1
Instruction description conventions 2-1
Function group definitions 2-2
Table 2-1. Function definitions 2-3
Format definitions 2-3
Table 2-2. Format definitions 2-3
General data structure 2-3
Table 2-3. Data structures 2-4
Processor characteristic 2-5
Table 2-4. Processor characteristics 2-5
Terminal session example 2-5

3 ASSEMBLING

Invoking the Prime Macro Assembler (PMA) 3-1
File usage 3-1
Assembler messages 3-2
Listing format 3-2
Figure 3-1. A-Register details 3-3
Figure 3-2. Example of assembly listing 3-4

4 LOADING R-MODE PROGRAMS

Introduction 4-1
Using the loader under PRIMOS 4-1
Normal loading 4-1
Load maps 4-3
Loader concepts 4-4
Figure 4-1. Examples of load maps 4-7
Command summary 4-8

5 LOADING SEGMENTED PROGRAMS

Introduction 5-1
Using SEG under PRIMOS 5-1
Normal loading 5-2
Load maps 5-3
Figure 5-1. Example of load map 5-5
Advanced SEG features 5-7
Command summary 5-8
SEG-level commands 5-9
LOAD subprocessor commands 5-10
MODIFY subprocessor commands 5-13

6 EXECUTING

Execution of unsegmented runfiles 6-1
Execution of segmented runfiles 6-2
Installation in the command UFD (CMDNC0) 6-2
Run-time error messages 6-5

7 DEBUGGING

Tools 7-1
Advanced debugging techniques 7-2
Debugging-PRIMOS severe errors 7-2
Memory overflow errors 7-3

8 INTERFACING WITH THE SYSTEM LIBRARIES

Table 8-1. System libraries 8-1
Figure 8-1. SR Subroutine CALL conventions 8-2
Figure 8-2. VI subroutine CALL conventions 8-3

9 DATA AND INSTRUCTION FORMATS—SRVI

Data structures 9-1
Processor characteristics 9-8
Instruction formats—I-mode 9-16
Table 9-1. Address formation special case selection 9-18

10 MEMORY REFERENCE CONCEPTS (SRV)

Background concepts 10-1
Table 10-1. Memory reference instruction format 10-2
Memory reference instruction formats 10-4
Table 10-2. V-mode two word memory reference 10-9
Addressing mode summaries and flow charts 10-10
Figure 10-1. 16S address calculation 10-11
Figure 10-2. 32S address calculation 10-13
Figure 10-3. 32R address calculation (1 of 5) 10-16
Figure 10-4. 32R address calculation (2 of 5) 10-17
Figure 10-5. 32R address calculation (3 of 5) 10-18
Figure 10-6. 32R address calculation (4 of 5) 10-19
Figure 10-7. 32R address calculation (5 of 5) 10-20
Figure 10-8. 64R address calculation (1 of 5) 10-23
Figure 10-9. 64R address calculation (2 of 5) 10-24
Figure 10-10. 64R address calculation (3 of 5) 10-25
Figure 10-11. 64R address calculation (4 of 5) 10-26
Figure 10-12. 64R address calculation (5 of 5) 10-27
Figure 10-13. 64V address calculation (1 of 3) 10-31
Figure 10-14. 64V address calculation (2 of 3) 10-32
Figure 10-15. 64V address calculation (3 of 3) 10-33

Field operations—FIELD 11–15
Floating point arithmetic—FLPT 11–16
Table 11–3. Floating point exception codes 11–16
Table 11–4. Floating point mantissa and exponent ranges 11–17

11 INSTRUCTION DEFINITIONS

Addressing mode—ADMOD 11–1
Branch—BRAN 11–2
Character string operations—CHAR 11–5
Clear register—CLEAR 11–7
Decimal arithmetic—DECI 11–8
Table 11–1. Decimal data type 11–9
Table 11–2. Edit sub-operations 11–14
Key Manipulation—KEYS 11–31
Logical operations—LOGIC 11–32
Logical test and set—LTSTS 11–33
Machine control—MCTL 11–34
Move data—MOVE 11–39
Program control and jump—PCTLJ 11–43
Process exchange—PRCEX 11–49
Queue management—QUEUE 11–49
Shift group—SHIFT 11–50
Skip conditional—SKIP 11–53
Table 11–5. Combination skip group 11–55

12 I-MODE INSTRUCTIONS

Addressing mode—ADMOD 12–1
Branch—BRAN 12–1
Character operations—CHAR 12–3
Clear register and memory—CLEAR 12–14
Decimal arithmetic—DECI 12–5
Field Operations—FIELD 12–5
Floating point arithmetic—FLPT 12–6
Integer arithmetic—INT 12–9
Integrity check for hardware—INTGY 12–14
Input/output—I/O 12–14
Key manipulation—KEYS 12–14
Logical operations—LOGIC 12–15
Logical test and set—LTSTS 12–16
Machine control—MCTL 12–17
Move data—MOVE 12–17
Program control and jump—PCTLJ 12–19
Process exchange—PRCEX 12–10
Queue management—QUEUE 12–10
Shift—SHIFT 12–21

13 INSTRUCTION SUMMARY CHART

Instruction summary 13–1

14 LANGUAGE STRUCTURE

Introduction 14-1
Lines 14-1
Statements 14-1
Figure 14-1. PMA statements 14-2
Figure 14-2. PMA line format 14-3
Memory reference instruction format 14-5
Instruction formats—I-Mode 14-5
Table 14-1. Assembler formats 14-7
How to write V or I mode code in PMA 14-8

15 DATA DEFINITION

Table 15-1. Numeric constants 15-2
Terms 15-5
Figure 15-1. Floating point data formats 15-6
Table 15-2. Modes 15-8
Expressions 15-8
Literals 15-10
Assembler attributes 15-11

16 PSEUDO OPERATIONS

Introduction 16-1
Table 16-1. Pseudo-operation summary 16-2
Figure 16-1. Pseudo-operations 16-4
Assembly control psuedo-operations (AC) 16-5
Address definition pseudo-operations (AD) 16-7
Conditional assembly pseudo-operations (CA) 16-8
Data defining pseudo-operations (DD) 16-10
Listing control pseudo-operations (LC) 16-11
Literal control pseudo-operations (LT) 16-12
Loader control pseudo-operations (LO) 16-13
Macro definition pseudo-operations (MD) 16-16
Program linking pseudo-operations (PL) 16-18
Storage allocation pseudo-operations (SA) 16-20
Symbol defining pseudo-operations (SD) 16-20

17 MACRO FACILITY

Introduction 17-1
Macro definition 17-2
Macro calls 17-3
Nesting macros 17-5
Conditional assembly 17-6
Macro listing 17-6

18 INTRODUCTION TO TAP, PSD, VPSD

Using TAP 18-1
  Table 18-1. Debugging command summary 18-1
Using PSD 18-3
  Table 18-2. PSD/VPSD versions 18-3
Using VPSD 18-3
Command line format 18-4
  Table 18-3. Input/output formats (PSD and VPSD) 18-5

19 TAP COMMAND SUMMARY

TAP command summary 19-1
  Table 19-1. TAP terminators 19-1
  Table 19-2. Keys 19-3

20 PSD COMMMAND SUMMARY

PSD command summary 20-1
  Table 20-1. PSD terminators 20-1
  Table 20-2. Key values: R and S mode 20-4

21 VPSD COMMAND SUMMARY

  Table 21-1. VPSD terminators 21-1
  Table 21-2. Key values: R and S modes 21-4
  Table 21-3. Key values: V mode 21-5

A ASSEMBLER ATTRIBUTES

B ASCII CHARACTER SET

C ERROR MESSAGES

1 OVERVIEW

1 Introduction

INTRODUCTION

This document is a comprehensive user guide for the Prime Macro Assembler (PMA) programmer. In this one document you will find almost everything you will need to know to write, assemble, load, debug and execute an assembly language program. We assume the following background: you are an experienced assembly language programmer although you may be unfamiliar with Prime's PMA; and you have been introduced to Prime's PRIMOS operating system and its major utilities through the use of a high-level language such as FORTRAN and COBOL. (If not, we recommend you read one of our other language user guides before undertaking a PMA project.)

ORGANIZATION AND USAGE

This document is organized into five major parts:

Part 1. Overview and conventions (Sections 1 and 2)

Part 2. PMA Usage (Sections 3 through 8)

Part 3. Machine Formats and Instructions (Sections 9 through 13)

Part 4. PMA Reference (Sections 14 through 17)

Part 5. Debugging Utilities Reference (Sections 18 through 21)

In addition to a tutorial section for the new PMA programmer (Part 2), it contains complete descriptions of:

Machine instructions
Data structures
Assembler pseudo operations
Assembler macro facilities
TAP, PSD and VPSD debugging facilities

RELATED DOCUMENTS

The FORTRAN Programmer's Guide

Reference Guide, System Architecture

The New User's Guide to Editor and Runoff

PRIMOS Commands Reference Guide

Reference Guide, PRIMOS Subroutines

PRIMOS Programmer's Companion

FORTRAN Programmer's Companion

Assembly Language Programmer's Companion

Reference Guide, LOAD and SEG

2

Conventions

PRIME CONVENTIONS

Symbols, abbreviations, special characters and conventions frequently used in this document are defined below.

Prime filename conventions

Filename	Function
B_filename	Binary (object) file
L_filename	Listing file
C_filename	Command file
filename	Source file
*filename	Saved (executable) file
M_filename	Map file
#filename	SEG runfile

Note

Filenames may be a maximum of 32 characters.

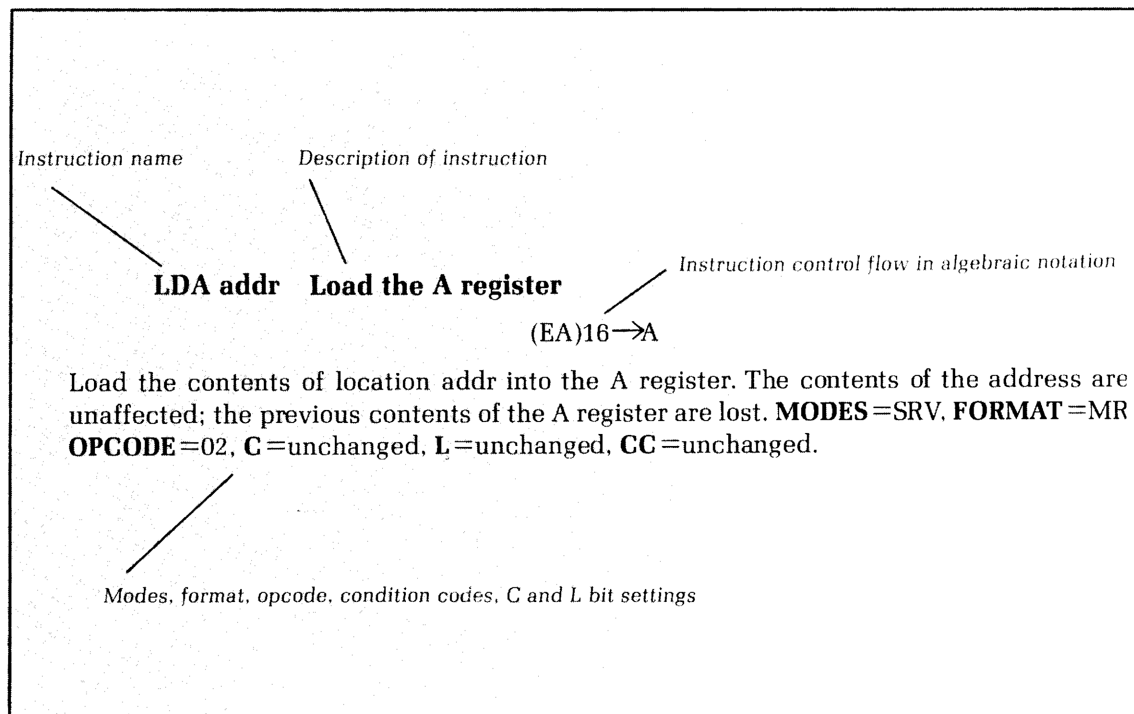
Text conventions

An item in all capital letters must be included verbatim. Rust colored letters indicate acceptable abbreviations. In TAP, PSD and VPSD commands enter *only* the rust colored letters. A quote mark (apostrophe) preceding a number means octal.

INSTRUCTION DESCRIPTION CONVENTIONS

This section describes each of the instructions in the context of the mode where they are first used. To avoid duplicate descriptions while facilitating retrieval, each instruction is described once, but listed in I-mode if appropriate.

Format illustration:



Instruction summary and description conventions

A	A Register (16-bits)
B	B Register (16 bits)
L	L Register (A|B)
E	E Register (32-bits)
F	Floating Point Register
H	Half Register (16-bits, I Mode)
R	Full register (32-bits, I Mode)
C	C-Bit in the keys
L-bit	L-Bit in the Keys
CC	Condition Codes
LB	Link Base Register
SB	Stack Base Register
PB	Procedure Base Register
XB	Temporary Base Register
S	S-Mode
R	R-Mode
V	V-Mode
I	I-Mode
FAR	Field Address Register
FLR	Field Length Register
→	Replaces

FUNCTION GROUP DEFINITIONS

The instruction definitions are grouped by primary function, such as integer arithmetic. Table 2-1 below contains the definitions for all the function groups and modes. If you wish to find a particular instruction, Section 13 contains an alphabetic list.

Table 2-1. Function Definitions

Definition	S	R	V	I
Addressing Mode	X	X	X	X
Branch			X	X
Character			X	X
Clear field	X	X	X	
Decimal Arithmetic			X	X
Field Register			X	X
Floating Point Arithmetic		X	X	X
Integer Arithmetic	X	X	X	X
Integrity	X	X	X	X
Input/Output	X	X	X	X
Keys	X	X	X	X
Logical Operations	X	X	X	X
Logical Test and Set	X	X	X	X
Machine Control	X	X	X	X
Move	X	X	X	X
Program Control and Jump	X	X	X	X
Process Exchange			X	X
Queue Control			X	X
Shift	X	X	X	X
Skip	X	X	X	X

FORMAT DEFINITIONS

Each instruction has a format. The formats and their meaning are summarized in Table 2-2. The specific bit definitions are defined in Section 9—Data and Instruction Formats and Section 10—Memory Reference Concepts.

Table 2-2. Format Definitions

Mnemonic	Definition	S	R	V	I
GEN	Generic	X	X	X	X
AP	Address Pointer			X	X
BRAN	Branch			X	
IBRN	Branch I-mode				X
CHAR	Character			X	X
DECI	Generic Decimal			X	X
PIO	Programmed I/O	X	X	X	X
SHFT	Shift	X	X	X	X
MR	Memory Reference - Non I-mode	X	X	X	
MRFR	Memory Reference - Floating Register				X
MRNR	Memory Reference Non Register				X
RGEN	Register Generic				X

GENERAL DATA STRUCTURES

Table 2-3. Data Structures—summarizes all the data structures manipulated by instructions.

Table 2-3. Data Structures

Class	S	R	V	I
Integer (Unsigned)				
16-bit	X	X	X	X
32-bit			X	X
Integer (Signed)				
16-bit	X	X	X	X
31-bit	X	X		
32-bit			X	X
Floating Point				
32-bit		X	X	X
64-bit		X	X	X
Decimal			X	X
Character String			X	X
Word				
16-bit	X	X	X	
32-bit				X
Halfword - 16 bit				X
Byte	X	X	X	X
Indirect Pointer (IP)				
16-bit	X	X	X	X
32-bit			X	X
48-bit			X	X
Address Pointer (AP)			X	X
Stacks				
Segment Header			X	X
Frame Header			X	X
Argument Template			X	X
Entry Control Block			X	X
Queue Control Block			X	X

PROCESSOR CHARACTERISTICS

Table 2-4, Processor Characteristics, lists the program visible portions of the hardware.

Table 2-4. Processor Characteristics

Class	S	R	V	I
Registers				
S, R mode	X	X		
V, I mode			X	X
Field Registers			X	X
Floating Registers		X	X	X
Keys				
S, R mode	X	X		
V, I mode			X	X
C-Bit	X	X	X	X
L-Bit			X	X
Condition Codes			X	X
Modals			X	X

SAMPLE TERMINAL SESSION

```
***** FIRST, CREATE THE FILE *****
OK, ED
GO
INPUT

EDIT
TAB 1Ø 15 25

INPUT
\SEG
\RLIT
STR\LDA\='456
\STA\BUFF1
\STA\BUFF2
\PRTN
***** DATA AREA *****
\DYNM\STCK(1)
\LINK
BUFF1\BSS\1
BUFF2\BSS\1
\ECB\STR
\END

EDIT
FILE TTY

*****
***** ASSEMBLE THE FILE *****
*****
```

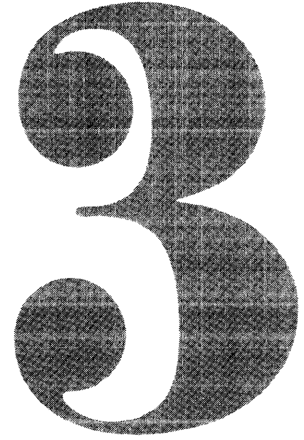
```
OK, PMA TTY
GO
0000 ERRORS (PMA-REV 15.0)
*****
***** PMA LISTING *****
*****

OK, SLIST L_TTY
GO
          SEG
                              (0001)            SEG
                              (0002)            RLIT
  000000:      02.000005      (0003) STR        LDA  ='456
  000001:      04.000400L     (0004)            STA  BUFF1
  000002:      02.000012S     (0005)            LDA  STCK
  000003:      04.000401L     (0006)            STA  BUFF2
  000004:        000611       (0007)            PRTN
                              (0008) ***** DATA AREA *****
                 000012       (0009)            DYNM STCK(1)
                              (0010)            LINK
  000400>                     (0011) BUFF1      BSS  1
  000401>                     (0012) BUFF2      BSS  1
  000402>        000000       (0013)            ECB  STR
                 000014
                 000011
                 000000
                 177400
                 014000

             000422           (0014)            END

  000005:      00.000456A


  TEXT SIZE:    PROC 000006    LINK 000022    STACK 000013

  BUFF1      000400L 0004  0011
  BUFF2      000401L 0006  0012
  STCK       000012S 0005  0009
  STR        000000  0003  0013


 0000 ERRORS (PMA-REV 15.0)

 *****
 ***** LOAD THE FILE *****
 *****
 OK, SEG
 GO
 # LOAD
 SAVE FILE TREE NAME: $TTY1
 $ LOAD B_TTY
 LOAD COMPLETE
 $ SAVE
```

```
$ MAP
*START  000000  000000  *STACK  004001  001006  *SYM    000003

SEG. #   TYPE       LOW      HIGH     TOP
004001   PROC##    001000   001005   001005
004002   DATA      000002   000021   000021


ROUTINE        ECB          PROCEDURE      ST. SIZE  LINK FR.
####      4002  000002    4001  001000    000014  000022  4002  177400
DIRECT ENTRY LINKS

COMMON BLOCKS

OTHER SYMBOLS

*****
***** EXECUTE THE PROGRAM *****
*****
$ EXECUTE
ACCESS VIOLATION               ***** ERROR - CALL VPSD *****
ER! VPSD
GO

$SN 4001

$A 1000:S
4001/ 1000 LDA#  1005
4001/ 1001 STA# LB%+ 400
4001/ 1002 LDA# SB%+ 12
4001/ 1003 STA# LB%+ 401
4001/ 1004 PRTN
4001/ 1005 DAC   456
4001/ 1006 HLT
$B 1000

$R 1000
4001/ 1000: LDA#  1005   A=0 B=0 X=0 K=14000 R=0 Y=12614

$B 1004

$PR
4001/ 1004: PRTN  A=0 B=0 X=0 K=14000 R=0 Y=12614

$PR
ACCESS VIOLATION
ER! ED TTY
*****
***** PROBLEM WAS INCORRECT ECB - NOTE THAT LOAD MAP *START
***** ENTRY HAD A ZERO VALUE.  MAIN PROGRAM NEEDS END OPERAND
***** REFERENCING THE ECB LABEL.
*****
GO
EDIT
```

```
TAB 1Ø 15 25
L END
          END
R \END\ECB1
N-1
          ECB  STR
C /     /ECB1/
ECB1      ECB  STR
FILE

OK, PMA TTY
GO
ØØØØ ERRORS (PMA-REV 15.Ø)
SLIST L_TTY
GO
         SEG
                              (ØØØ1)           SEG
                              (ØØØ2)           RLIT
  ØØØØØØ:      Ø2.ØØØØØ5     (ØØØ3) STR       LDA   ='456
  ØØØØØ1:      Ø4.ØØØ4ØØL    (ØØØ4)           STA   BUFF1
  ØØØØØ2:      Ø2.ØØØØ12S    (ØØØ5)           LDA   STCK
  ØØØØØ3:      Ø4.ØØØ4Ø1L    (ØØØ6)           STA   BUFF2
  ØØØØØ4:        ØØØ611      (ØØØ7)           PRTN
                              (ØØØ8) ***** DATA AREA *****
                 ØØØØ12      (ØØØ9)           DYNM  STCK(1)
                              (ØØ1Ø)           LINK
  ØØØ4ØØ>                    (ØØ11) BUFF1     BSS   1
  ØØØ4Ø1>                    (ØØ12) BUFF2     BSS   1
  ØØØ4Ø2>        ØØØØØØ      (ØØ13) ECB1      ECB   STR
                 ØØØØ14
                 ØØØØ11
                 ØØØØØØ
                 1774ØØ
                 Ø14ØØØ

             ØØØ422          (ØØ14)           END   ECB1

  ØØØØØ5:      ØØ.ØØØ456A


  TEXT SIZE:   PROC ØØØØØ6   LINK ØØØØ22   STACK ØØØØ13

  BUFF1        ØØØ4ØØL ØØØ4  ØØ11
  BUFF2        ØØØ4Ø1L ØØØ6  ØØ12
  ECB1         ØØØ4Ø2L ØØ13  ØØ14
  STCK         ØØØØ12S ØØØ5  ØØØ9
  STR          ØØØØØØ  ØØØ3  ØØ13


 ØØØØ ERRORS (PMA-REV 15.Ø)


OK, SEG
GO
```

```
# DELETE $TTY
# LOAD
SAVE FILE TREE NAME: $TTY
$ LOAD B_TTY
LOAD COMPLETE
$ SAVE
$ MAP
*START  004002  000002  *STACK  004001  001006  *SYM    000003


SEG. #    TYPE       LOW      HIGH      TOP
004001    PROC##    001000    001005    001005
004002    DATA      000002    000021    000021


ROUTINE        ECB            PROCEDURE       ST. SIZE  LINK FR.
####      4002  000002    4001  001000    000014  000022  4002  177400

DIRECT ENTRY LINKS

COMMON BLOCKS

OTHER SYMBOLS

$ EXECUTE
*****
***** PROGRAM NOW WORKS.  NOTE THAT LOAD MAP *START ENTRY HAS
***** ADDRESS OF SEGMENT 4002 WORD 2, THE LOCATION TO START
***** EXECUTION.
*****


OK,
```

2 PMA USAGE



Assembling

The Prime Macro Assembler (PMA) is a two pass assembler (three pass in SEG or SEGR mode). The first pass generates a symbol table and identifies external references; the second pass generates object code blocks for input to the loader and, optionally, creates a listing. The three pass assembly in SEG or SEGR mode permits optimization of stack and link frame references.

INVOKING THE PRIME MACRO ASSEMBLER (PMA)

PMA is invoked by the command:

PMA pathname [-option-1] [-option-2]. . .[-option-n]

where **pathname** is the pathname of the PMA source file and **option-1, option-2,** etc. are the mnemonics for one of the options described below. All options must be preceded by a dash "–".

For example, the command: PMA ALPHA>BETA -ERRLIST means assemble the file ALPHA located in UFD BETA and list only the errors, while PMA ALPHA means, assemble the file ALPHA located in the home UFD and produce whatever listing the program specifies (see listing pseudo-operations in Section 16 – Pseudo-Operations). The listing name (if any) will be L_ALPHA. PMA ALPHA -LISTING BETA means, assemble the file ALPHA located in the current UFD, generate a binary file, and produce whatever listing the program specifies. The listing filename will be BETA.

Option	Meaning
-INPUT treename	Input treename
-LISTING treename	Listing treename
-BINARY treename	Object file treename
-EXPLIST	Generates full assembly listing (overrides the pseudo-operation NLIST) and forces listing file generation
-ERRLIST	Generates errors-only listing, and forces listing file generation
-XREFL	Generates complete cross reference
-XREFS	Omits from the listing symbols which have been defined but not used

FILE USAGE

Three files may be involved during an assembly:

File Type	PRIMOS File unit
Source	1
Listing	2
Object	3

PMA automatically opens files for listing and object output. B_source-filename is the default object name; L_source-filename is the default listing name. Use the -BINARY option to charge the object default and -LISTING option to change the listing default.

The PRIMOS commands LISTING and BINARY permit you to concatenate files, since they remain open when the assembler returns control to PRIMOS.

ASSEMBLER MESSAGES

When the assembler reads the END statement of the input file on the second pass, it prints a message, terminates assembly, and returns control to PRIMOS command level. The message contains a decimal error count and the version number of the assembler, as in:

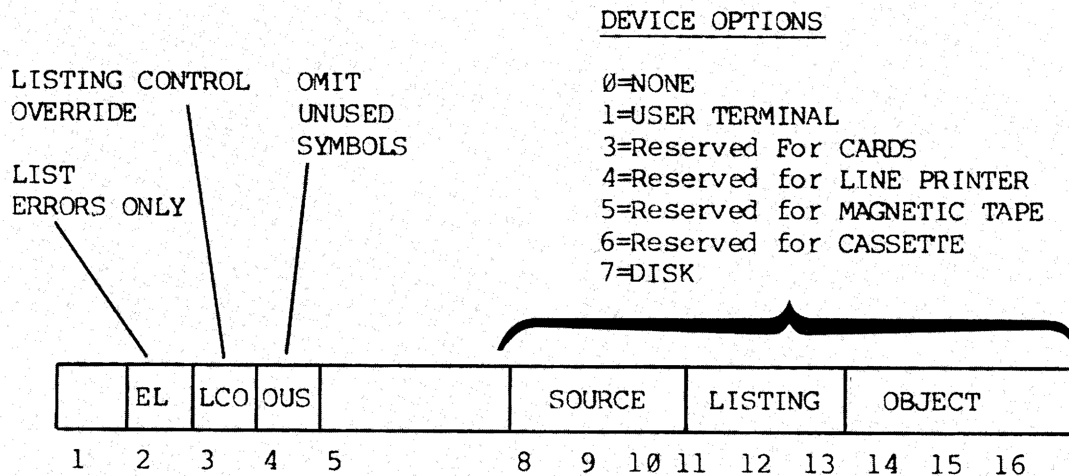
```
ØØØ1 ERRORS (PMA-REV 16.2)
```

LISTING FORMAT

Figure 3-2 shows a section of a typical assembly listing and illustrates the main features.

Each page begins with a header and a sequential page number. The first statement in a program is used as the initial page header. If column 1 of any source statement contains an apostrophe ('), columns 10-72 of that statement become the header for all pages that follow, until a new title is specified.

User-generated messages may be inserted into the listing output by using the SAY pseudo-operation in the source program. Such messages can be used to document the progress of a complex conditional assembly operation.



Error listing (bit 2): If this bit is set, only the lines containing errors are listed. Otherwise, listing is controlled by pseudo-operations in the source program.

Listing control override (bit 3): If this bit is set, the assembler overrides any listing control pseudo-operations in the source program and lists all statements, including lines within macro expansions and lines that would be skipped by conditional assembly. Otherwise, listing is controlled by pseudo-operations in the source program.

Omit unused symbols (bit 4): If this bit is set, symbols which have been defined but not referenced are omitted from the cross-reference.

Device options (bits 8-16): The last three octal digits of the A-register select source, listing and object devices respectively.

Figure 3-1. A-Register Details

```
                                (0001)              SEG
                                (0002)              RLIT
000000:     02.000005           (0003) STR          LDA   ='456
000001:     04.000400L          (0004)              STA   BUFF1
000002:     02.000012S          (0005)              LDA   STCK
000003:     04.000401L          (0006)              STA   BUFF2
000004:       000611            (0007)              PRTN
                                (0008) ***** DATA AREA *****
              000012            (0009)              DYNM  STCK(1)
                                (0010)              LINK
000400>                         (0011) BUFF1        BSS   1
000401>                         (0012) BUFF2        BSS   1
000402>       000000            (0013) ECB1         ECB   STR
              000014
              000011
              000000
              177400
              014000

            000422              (0014)              END   ECB1

000005:     00.000456A


TEXT SIZE:   PROC 000006   LINK 000022   STACK 000013
BUFF1        000400L 0004   0011

BUFF2        000401L 0006   0012

ECB1         000402L 0013   0014

STCK         000012S 0005   0009




STR          000000  0003   0013


0000 ERRORS (PMA-REV 16.2)
```

Figure 3-2. Example of Assembly Listing

Cross-reference listing (concordance)

At the end of the assembly listing appears a cross-reference listing of each symbol's name (in alphabetical order), the symbol's location or address value, and a list of all references to the symbol. The location and address values are in octal unless the PCVH pseudo-operation specifies hexadecimal listing. Each reference is identified by a four-digit line number. The NLST pseudo-operation suppresses the cross-reference listing; the option -XREFS suppresses symbols which have been defined but not used.

PBRK	Program Break. Resume loading at a new location.
CH,SS,SY,XP	Symbol control commands.
EN	ENtire save; saves copy of load session for building of program overlays.
ER	Controls action taken by loader following errors.
SZ	Controls use of Sector 0.

COMMAND SUMMARY

Following is a summary of all LOAD commands, in alphabetical order. All file and directory names may be specified by pathnames, except in the LIBRARY command. All numerical values must be octal.

ATTACH [pathname]

Attaches to specified directory.

AUTOMATIC base-length

Inserts base area of specified **length** at end of routine if >'300 locations loaded since last base area.

CHECK [symbol-name] [offset-1]. . .[offset-9]

Checks value of current PBRK against symbol or number. **symbol-name** is a 6 character symbol defined in the symbol table. **offset-1** through **9** are summed to form an address or offset from symbol name. Numbers preceded by "-" are negative.

COMMON address

Moves top/starting COMMON location to **address.**

DC [END]

Defers definition of COMMON block until SAVE command is given. (Low end of COMMON follows top of load.) **END** turns off DC.

ENTIRE pathname

Saves entire state of loader as runfile, along with temporary file, for building overlays.

ERROR n

Determines action taken in case of load errors.

n	Meaning
0	SZ errors treated as multiple indirect, others act as n=1.
1	Display multiple indirects on terminal but continue LOAD; abort load of file for all other errors.
2	Abort to PRIMOS

EXECUTE [a] [b] [x]

Starts execution with specified register values.

F/ { FORCELOAD / LIBRARY / LOAD } [pathname] [parameters]

Forceloads all modules in specified object file. See LOAD for parameters.

HARDWARE definition

Specifies expected level of instruction execution.

CPU	Definition
P450 and up	100
P350,P400	57
P300/FP	17 FP=Floating Point
P300	3
P200/HSA	1 HSA=High-speed arithmetic
P100/HSA	1
P200	0
P100	0

HARDWARE, if given, *must* precede loading of UII library.

INITIALIZE [pathname] [parameters]

Initializes LOADER and, optionally, does a LOAD. See LOAD for parameters.

LIBRARY [filename] [loadpoint]

Attaches to LIB UFD, loads specified library file (FTNLIB is default), and re-attaches to home directory.

LOAD [pathname] [parameters]

Loads the specified object module. The parameters may be entered in three formats:

1. **loadpoint [setbase-1]. . .[setbase-8]**
2. *** [setbase-1]. . .[setbase-9]**
3. **symbol [setbase-1]. . .[setbase-9]**

In form 1, **loadpoint** is the starting location of the load. In form 2, the load starts at the current PBRK location (*). In form 3, the load address can be stated symbolically (**symbol**). The remaining numeric parameters (**setbase-1**, etc.) specify the size of linkage areas to be inserted before and after modules during loading. If the last parameter is '177777, the loader requests more setbase values.

MAP [pathname] [option]

Generates load-state map on terminal, or in a file, if **pathname** is specified.

Option	Meaning
0	Load state, base area, symbol storage map; symbols sorted by address (default)
1	Load state only
2	Load state and base area
3	Unsatisfied references only
4	Same as 0
5	System Programmer map
6	Undefined symbols sorted alphabetically
7	All symbols sorted alphabetically
10	Special symbol map for PSD (in a file)

MODE { **D32R** | **D64R** | **D16S** | **D32S** }

Specifies address resolution mode for next load module (32K Relative, D32R, is default). If used, MODE must precede other LOAD commands.

P/ { **FORCELOAD** | **LIBRARY** | **LOAD** } **[pathname] [parameters]**

Begins loading at next page boundary. See LOAD for **parameters.**

PAUSE

Leaves loader to execute internal PRIMOS command. Return via START.

PBRK { **[symbol-name] [offset-1]. . .[offset-9]** | *** offset-1 [offset-2]. . .[offset-9]** }

Sets a program break to value of **symbol** plus offset or a number. * treats sum of numbers as offset from current PBRK. Offsets may be negative.

QUIT

Deletes temporary file, closes map file (if loader opened it), and returns to PRIMOS.

SAVE pathname

Writes a memory image of the loaded runfile to the disk.

SETBASE { **[base-start] [base-range]** | *** [base-range]** }

Defines starting location and size of base area. * is current value of PBRK.

SS symbol-name

Save symbol. Exempts specified **symbol** from action of XPUNGE.

SYMBOL { **symbol-name [offset-1] . . . [offset-6]** | *** offset-1 [offset-2] . . . [offset-6]** }

Establishes locations in memory map for common blocks, relocation load points, or to satisfy references. * is current value of PBRK. **Offsets** are summed and may be negative.

SZ { **YES** | **NO** }

Permits/prohibits links in sector zero.

VIRTUALBASE base-start to-sector

Copies base sector from **base-start** to corresponding locations in **to-sector**. Used for building RTOS modules.

XPUNGE dsymbols dbase

Deletes COMMON symbols, other defined symbols, and base areas.

4
Loading R-Mode programs

INTRODUCTION

The PRIMOS LOAD utility converts object modules (such as those generated by PMA) into runfiles that execute in the 32R or 64R addressing modes. (Runfiles to execute in the 64V mode *must* be loaded using the segmentation utility, SEG.)

The following description emphasizes the loader commands and functions that are of most use to the PMA programmer. For a complete description of all loader commands, including those for advanced system-level programming, refer to Reference Guide, LOAD and SEG.

USING THE LOADER UNDER PRIMOS

The PRIMOS command:

LOAD

transfers control to the R-mode loader, which prints a $ prompt character and awaits a loader subcommand. After executing a command successfully, the loader repeats the $ prompt character.

If an error occurs during an operation, the loader prints an error message, then the $ prompt charcter. Loader error messges and suggested handling techniques are discussed elsewhere in this section and in Appendix C. Most of the errors encountered are caused by large programs where the user is not making full use of the loader capabilities.

When a system error (FILE IN USE, ILLEGAL NAME, NO RIGHT, etc.) is encountered, the loader prints this system error and returns its prompt symbol, $.

The loader remains in control until a QUIT or PAUSE subcommand returns control to PRIMOS, or an EXECUTE subcommand starts execution of the loaded program.

Load subcommands can be used in command files, but comment lines result in a CM (command error) message.

NORMAL LOADING

Loading is normally a simple operation with only a few straightforward commands needed. The loader also has many additional features to optimize runfile size or speed, perform difficult loads, and deal with possible complications. The most frequently used load commands and operations are presented first; this enables immediate use of the loader. Advanced features are then described followed by a summary of all loader commands.

The following commands (shown in abbreviated form) accomplish most loading functions:

PRIMOS-Level commands:

FILMEM	Initializes user space in preparation for load.

LOAD	Invokes loader for entry of subcommands.
RESUME	Starts execution of a loaded, SAVEd runfile.

LOAD subcommands:

MODE option	Sets runfile addressing mode as D32R (default) or D64R.
LOAD pathname	loads specified object file.
LIBRARY [filename]	Loads library object files from UFD LIB. (Default is FTNLIB.)
MAP [option]	Prints loadmap. Option 3 shows unresolved references.
INITIALIZE	Returns loader to starting condition in case of command errors or faulty load.
SAVE pathname	Saves loaded memory image as runfile.
QUIT or PAUSE	Return to PRIMOS.

Most loads can be accomplished by the following basic procedure:

1. Use the PRIMOS FILMEM command to initialize memory to binary zeroes.
2. Invoke LOAD.
3. Use the MODE command to set the addressing mode, if necessary (The default is 32R mode.)
4. Use loader's LOAD subcommand to load the object file (B_pathname) and any separately assembled subroutines.
5. Use loader's LIBRARY subcommand to load subroutines called from libraries (the default is FTNLIB in the UFD LIB). Other libraries, such as SRTLIB or APPLIB, must be named explicitly.
6. If you do not have a LOAD COMPLETE, do a MAP 3 to identify the unsatisfied references, and load them.
7. SAVE the runfile under an appropriate name.

If these commands produce a LOAD COMPLETE message, then loading was accomplished. If there is a problem, it will become apparent by the absence of a LOAD COMPLETE message or by some other loader error message. (See Appendix C for a complete list of all loader error messages and their probable cause and correction.)

After a successful load, start runfile execution from LOAD command level, or quit from the loader and start execution through the PRIMOS RESUME command. An example of such a load is:

```
OK, LOAD
GO
$ MO D64R
$ DC
$ LO B_ARRAY
$ LI
```

Order of loading

The following loading order is recommended:

1. Main program.
2. Separately assembled user-generated subroutines (preferably in order of frequency of use).

3. Other Prime libraries (LI filename).
4. Standard FORTRAN library (LI).

Loading library subroutines

Standard FORTRAN mathematical and input/output functions are implemented by subroutines in the library file FTNLIB in the LIB UFD. The appropriate subroutines from the file are loaded by the LIBRARY command given without a filename argument. If subroutines from other libraries are used, such as MATHLB, SRTLIB, or APPLIB, additional LIBRARY commands are required which include the desired library as an argument.

LOAD MAPS

During loading the loader collects information about the results of the load process, which can be printed at the terminal (or written to a file) by the MAP command:

MAP [pathname] [option]

The information in the map can be consulted to diagnose problems in loading, or to optimize placement of modules, linkage areas and COMMON in complex loads.

Load information is printed in four sections, as shown in Figure 4-1. The amount of information printed is controlled by MAP option codes such as:

Option	Load Map Information
None, 0 or 4	Load state, base area, and symbol storage; symbols sorted by address
1	Load state only
2	Load state and base areas
3	Unsatisfied references only
6	Undefined symbols, sorted in alphabetical order
7	All symbols, sorted in alphabetic order

Load state

The load state area shows where the program has been loaded, the start-of-execution location, the area occupied by COMMON, the size of the symbol table, and the UII status. All locations are octal numbers.

***START:** The location at which execution of the loaded program will begin. The default is '1000.

***LOW:** The *lowest* memory image location occupied by the program. Executable code normally starts at '1000, but sector 0 address links (if any) begin at '200.

***HIGH:** The *highest* memory image location occupied by the program (excluding any area reserved for COMMON).

***PBRK:** "Program Break": The next available location for loading. It normally is the location following the last loaded module, but can be moved by PBRK or the LOAD family of commands.

***CMLOW:** The low end of COMMON.

***CMHGH:** The top of COMMON.

***SYM:** The number of symbols in the loader's symbol table. This is usually of not concern unless the symbol space crowds out the last remaining runfile buffer area. (There is room for about 4000 symbols before this is a risk.)

***UII:** A code representing the hardware required to execute the instructions in loaded modules. Codes and other information are described later in this section.

Base areas

The base area map includes the lowest, highest and next available locations for all defined base areas. Each line contains four addresses as follows:

***BASE XXXXXX YYYYYY ZZZZZZ WWWWWW**

XXXXXX	Lowest location defined for this area
YYYYYY	Next available location if starting up from XXXXXX
ZZZZZZ	Next available location if starting down from WWWWWW
WWWWWW	Highest location defined for this area

Symbol storage

The symbol storage listing consists of every defined label or external reference name printed four per line in the following format:

namexx NNNNNN

or

****namexx** NNNNNN

NNNNNN is a six-digit octal address. The ** flag means the reference is unsatisfied (i.e., has not been loaded).

Symbols are listed by ascending address (default) or in alphabetical order (MA 6 or MA 7). The list may be restricted to unsatisfied references only (MA 3 or MA 6).

COMMON blocks

The low end and size of each COMMON area are listed, along with the name (if any). Every map includes a reference to the special COMMON block LIST, defined as starting at location 1.

LOADER CONCEPTS

When standard loading goes well, the user can ignore most of the loader's advanced features. However, situations can arise where some detailed knowledge of the loader's tasks, can optimize size or performance of a runfile, or even make a critical load possible. From that viewpoint, the main tasks of the loader are:

- Convert block-format object code into a run-time version of the program (executable machine instructions, binary data and data blocks).
- Resolve address linkages (translate symbolic names of variables, subroutine entry points, data items etc. into appropriate binary address values).
- Perform address resolution (discussed later).
- Detect and flag errors such as unresolved external references, memory overflow, etc.
- Build (and, on request, print) a load map. The map may also be written to a file.
- Reserve COMMON areas as specified by object modules.
- Keep track of runfile's hardware execution requirements and make user aware of need to load subroutines from UII library.

Virtual loading

The loader occupies the upper 32K words of the user's 64K-word virtual address space. Programs up to 32K words are loaded directly into the memory locations from which they execute. Programs loaded in this manner can be started by the loader's EXECUTE command without being saved. For larger 64R-mode programs, the loader uses the available memory as buffer space and transfers loaded pages of memory to a temporary file that accomodates a full 64K-word memory image. When loading is complete, the file must be assigned a name by the loader's SAVE command; it can then be executed either through the loader's EXECUTE command or the PRIMOS RESUME command.

The loader remains attached to the working directory throughout loading, for access to the temporary file. Files in other directories can be loaded by giving a pathname in a LOAD command.

Use of pathnames

Pathnames can be used to specify object files in all commands except LIBRARY, which accepts only a simple filename of a file within the LIB UFD.

Object code

Inputs to the loader are in the form of object code—a symbolic, block-format file generated by all of Prime's language translators. Prime's standard library files consist of subroutines in this format.

The loader combines the user's main program object file with the object files of all referenced subroutines (either those in the library, or those generated and separately compiled by the user) into a single runfile. The runfile is then ready for execution, either directly through the loader's EXECUTE command or through the PRIMOS RESUME command.

Runfiles

A runfile consists of a header block followed by the runfile text in memory image format. The header contains information that enables the runfile to be brought into memory by the PRIMOS RESTORE or RESUME command. Contents of the header can be examined after a RESTORE by the PM command. (See PRIMOS Commands Reference Guide.)

Selecting the addressing mode

The 32R addressing mode is retained as the loader's default for compatibility with existing command files. The only significant difference between 32R and 64R for small programs is that 32R permits multiple indirect links, while 64R allows only one level of indirection. In certain situations such as processing of multi-dimensional arrays, 32R mode may enable the programmer to write a program that is somewhat more compact or runs slightly faster. However, for programs that approach the 32K word boundary, 64R mode ensures successful loading with no significant penalties of size or speed. Thus MODE D64R is recommended for most applications.

Base areas

"Base Area" is discussed in Section 10—Memory Reference Concepts. When one of the messages is printed:

BASE SECTOR 0 FULL

symbolname XXXXXX NEED SECTOR 0 LINK

This condition, usually encountered only when loading large programs, can be avoided in several ways:

- Give the AUTOMATIC command to enable the loader to assign local linkage areas before and after individual subroutines.
- Use setbase parameters with a LOAD or LIBRARY command to insert local linkage areas where they are needed.
- Use the SETBASE command to designate a base areas where it is required.
- During assembly, use the SETB pseudo-operation.

UII handling

The loader can keep track of the CPU hardware required to execute the instructions generated by the modules already loaded. This is shown in the UII entry in the load state section of a load map. The codes are:

UII Value	CPU Required
100	Prime 450 and up
57	Prime 350 or 400
17	Prime 300 with FP Hardware
3	Prime 300
1	Prime 100 with HSA or 200 with HSA
0	Prime 100 or 200

If the UII code on the load map is greater than the value for the target CPU, then it will be necessary to load part of the UII library to make execution possible. When a CPU encounters an instruction not implemented by hardware, a UII (Unimplemented Instruction Interrupt) occurs and control is transferred to the appropriate UII routine. This routine simulates the missing hardware with software routines.

However, the UII routine *must* be loaded by the command LI UII, which should be the last LOAD command before the program is saved. The appropriate routines will be selected from this library to satisfy the additional hardware requirements of the program.

To make sure that only the required subroutines are loaded, the user can "subtract" hardware features that are present in the CPU by entering a HARDWARE command. For example, assume:

- A load session produces a load map UII value of 57.
- The target CPU is a Prime 300 with floating point (UII value 17).

The command:

HA 17

reduces the load state UII value to 40 (i.e., '57-'17) and ensures that the floating point subroutines do not occupy space in the runfile.

If, after a HARDWARE command, the load state UII value is 0, the UII library need not be loaded.

System programming features

The following commands are primarily of interest to systems programmers. They are described in more detail in the Reference Guide, LOAD and SEG:

F/	Prefix to LOAD and LIBRARY which forceloads unreferenced modules.
P/	Prefix to LOAD and LIBRARY which starts loading on next page boundary. (Can reduce paging time.)

```
*START  ØØ1ØØØ    *LOW    ØØØ2ØØ    *HIGH   ØØ7775    *PBRK   1Ø6376
*CMLOW  Ø77777    *CMHGH  Ø77777    *SYM    ØØØØ7Ø    *UII    ØØØØØ1

*BASE   ØØØ2ØØ  ØØØ225  ØØØ777  ØØØ777
*BASE   ØØ1534  ØØ16ØØ  ØØ16Ø5  ØØ16Ø5
*BASE   ØØ2576  ØØ266Ø  ØØ2661  ØØ2661
*BASE   ØØ3624  ØØ3663  ØØ3665  ØØ3665
*BASE   ØØ4664  ØØ47Ø6  ØØ47Ø7  ØØ47Ø7

**GHOST   ØØ1Ø25    F$WA      ØØ1Ø31    F$WX      ØØ1Ø37    F$IO      ØØ1113
  F$A1    ØØ16Ø6    F$A3      ØØ16Ø6    F$A5      ØØ1613    F$A2      ØØ1621
  F$A6    ØØ1627    F$A7      ØØ1645    F$CB      ØØ2326    F$IOBF    ØØ54Ø5
  WRASC   ØØ55Ø7    IOCS$     ØØ5514    IOCS$T    ØØ5613    WATBL     ØØ5625
  LUTBL   ØØ5644    PUTBL     ØØ57Ø1    RSTBL     ØØ5736    O$ADØ7    ØØ5773
  O$ADØ8  ØØ6136    O$AAØ1    ØØ62ØØ    PRSPE$    ØØ6235    OERRT$    ØØ6427
  ERRPR$  ØØ7433    PRWF$$    ØØ7436    WTLIN$    ØØ7441    ERRSET    ØØ7444
  F$ER    ØØ7447    F$HT      ØØ7454    EXIT      ØØ7534    AC1       ØØ7537
  AC2     ØØ754Ø    AC3       ØØ7541    AC4       ØØ7542    AC5       ØØ7543
  TNOU    ØØ7544    TONL      ØØ7634    T1OU      ØØ7641    T1IB      ØØ7661
  T1OB    ØØ7666    F$AT      ØØ7673    F$AT1     ØØ7675    GCHAR     ØØ774Ø
  SCHAR   ØØ7755

COMMON BLOCKS

  LIST    ØØØØØ1                        ØØ7776  Ø764ØØ
```

A. Full Map [MAP]

```
*START  ØØ1ØØØ    *LOW    ØØØ2ØØ    *HIGH   ØØ7775    *PBRK   1Ø6376
*CMLOW  Ø77777    *CMHGH  Ø77777    *SYM    ØØØØ7Ø    *UII    ØØØØØ1

*BASE   ØØØ2ØØ  ØØØ225  ØØØ777  ØØØ777
*BASE   ØØ1534  ØØ16ØØ  ØØ16Ø5  ØØ16Ø5
*BASE   ØØ2576  ØØ266Ø  ØØ2661  ØØ2661


*BASE   ØØ3624  ØØ3663  ØØ3665  ØØ3665
*BASE   ØØ4664  ØØ47Ø6  ØØ47Ø7  ØØ47Ø7

  AC1     ØØ7537    AC2       ØØ754Ø    AC3       ØØ7541    AC4       ØØ7542
  AC5     ØØ7543    ERRPR$    ØØ7433    ERRSET    ØØ7444    EXIT      ØØ7534
  F$A1    ØØ16Ø6    F$A2      ØØ1621    F$A3      ØØ16Ø6    F$A5      ØØ1613
  F$A6    ØØ1627    F$A7      ØØ1645    F$AT      ØØ7673    F$AT1     ØØ7675
  F$CB    ØØ2326    F$ER      ØØ7447    F$HT      ØØ7454    F$IO      ØØ1113
  F$IOBF  ØØ54Ø5    F$WA      ØØ1Ø31    F$WX      ØØ1Ø37    GCHAR     ØØ774Ø
**GHOST   ØØ1Ø25    IOCS$     ØØ5514    IOCS$T    ØØ5613    LUTBL     ØØ5644
  O$AAØ1  ØØ62ØØ    O$ADØ7    ØØ5773    O$ADØ8    ØØ6136    OERRT$    ØØ6427
  PRSPE$  ØØ6235    PRWF$$    ØØ7436    PUTBL     ØØ57Ø1    RSTBL     ØØ5736
  SCHAR   ØØ7755    T1IB      ØØ7661    T1OB      ØØ7666    T1OU      ØØ7641
  TNOU    ØØ7544    TONL      ØØ7634    WATBL     ØØ5625    WRASC     ØØ55Ø7
  WTLIN$  ØØ7441

COMMON BLOCKS
          ØØ7776  Ø764ØØ    LIST      ØØØØØ1
```

B. Symbols Sorted Alphabetically [MA 7]

Figure 4-1. Examples of load maps

5
Loading segmented programs

INTRODUCTION

The PRIMOS SEG utility converts object modules (such as those generated by the PMA) into segmented runfiles that execute in either 64V or 32I addressing mode and take full advantage of the architecture and instruction set of the Prime 350 and up. Segmented runfiles offer the following advantages:

- Much larger programs: up to 256 segments per user program (32 Megabytes).
- Access to V-mode and I-mode instructions and architecture (Prime 350 and up) for faster execution.
- Ability to install shared code: single copy of a procedure can service many users, significantly reducing paging time.
- Reentrant procedures permitted: procedure and data segments can be kept separate.

The following description emphasizes the commands and functions that are of most use to the PMA programmer. For a complete description of all SEG commands, including those for advanced system-level programming, refer to the Reference Guide, LOAD and SEG.

USING SEG UNDER PRIMOS

SEG is invoked by PRIMOS command:

SEG [pathname]

A **pathname** is given only when an existing SEG runfile is to be executed. Otherwise, the command transfers control to SEG command level, which prints a "#" prompt character and awaits a subcommand. After executing a subcommand successfully, the loader repeats the prompt character. SEG employs two subprocessors, LOAD and MODIFY, which accept further subcommands. The subprocessors use the "$" prompt character.

If an error occurs during an operation, SEG prints an error message, then the prompt character. Error messages and suggested handling techniques are discussed elsewhere in this section and in Appendix C.

When a system error (FILE IN USE, ILLEGAL NAME, NO RIGHT, etc.) is encountered, SEG prints the system error and returns the prompt symbol. SEG remains in control until a QUIT subcommand returns control to PRIMOS, or an EXECUTE subcommand starts execution of the loaded program.

SEG subcommands can be used in command files, but comment lines are accepted only within the LOAD subprocessor.

NORMAL LOADING

Loading is normally a simple operation with only a few straightforward commands needed. SEG also has many additional features to optimize runfile size or speed, perform difficult loads, load for shared procedures, and deal with possible complications. To facilitate immediate use of SEG, the most frequently used commands and operations are described first. Advanced features are then described, followed by a summary of all SEG commands.

The following commands (shown in abbreviated form) accomplish most loading functions:

SEG-Level commands:

DELETE	Deletes segmented runfile.
HELP	Prints a list of SEG commands at terminal.
LOAD	Invokes loader subprocessor for entry of subcommands.

LOAD subcommands:

LOAD pathname	Loads specified object file.
LIBRARY [filename]	Loads library object files from UFD LIB. (Default is PFTNLIB and IFTNLB, in that order.)
MAP [option]	Prints loadmap. Option 3 shows unresolved references.
INITIALIZE	Returns loader to starting condition in case of command errors or faulty load.
SAVE	Saves loaded memory image as runfile.
RETURN	Returns to SEG command level.
QUIT	Return to PRIMOS.

Most loads can be accomplished by the following basic procedure:

- Invoke SEG from PRIMOS level.
- Enter the LOAD command to start the LOAD subprocessor ($ prompt)
- Use the load subprocessor's LOAD subcommand to load the object file (B_filename) and any separately assembled subroutines.
- Use load subprocessor's LIBRARY subcommand to load subroutines called from libraries (the default is PFTNLB and IFTNLB in the UFD LIB). Other libraries, such as VSRTLB or VAPPLB, must be named explicitly.
- If you do not have a LOAD COMPLETE, do a MAP 3 to identify the unsatisfied references, and load them.
- SAVE the runfile.

If these commands produce a LOAD COMPLETE message, then loading was accomplished. If there is a problem, it will become apparent by the absence of a LOAD COMPLETE message or some other SEG error message. (See Appendix C for a complete list of all SEG error messages and their probable cause and correction.)

After a successful load, start runfile execution from loader command level, or quit from the loader and start execution through the PRIMOS RESUME command. An example of such a load is:

```
OK, SEG
GO
# LOAD
SAVE FILE TREE NAME: #ARRAY
$ LO B_ARRAY
$ LI
$ SA
$ MA M_ARRAY
$ QU
```

Order of loading

The following loading order is recommended:

1. Main program.
2. Separately assembled user-generated subroutines (preferably in order of frequency of use).
3. Other Prime Libraries (LI filename).
4. Standard FORTRAN library (LI).

Loading library subroutines

Standard FORTRAN mathematical and input/output functions are implemented by subroutines in the library file FTNLIB in the LIB UFD. The appropriate subroutines from this file are loaded by the LIBRARY command given without a filename argument. If subroutines from other libraries are used, such as VSRTLB or VAPPLB, additional LIBRARY commands are required which include the desired library as an argument.

LOAD MAPS

During loading, SEG collects (and stores, as part of the segmented runfile) information about the results of the load process. This can be printed at the terminal (or written to a file) by the load subprocessor's MAP command:

MAP [pathname] [option]

The information in the map can be consulted to diagnose problems in loading, or to optimize placement of modules, linkage areas and COMMON in complex loads. If a file **pathname** is given, the map is written to a file instead of being printed at the terminal. The loadmap is particularly useful for:

- Location where program halted (Link Base (LB) address after a crash).
- Modules not loaded (MA 3 or MA 6).
- Reason for stack overflow (Stack Base (SB) address after a crash).

When a map file is specified, it is opened on PRIMOS Unit 13 and remains open until the load session is completed. Any additional MAP commands specifying output to a file will use the one already opened; exiting from the Loader (via EXECUTE, QUIT, or RETURN) closes the map file. If the user has opened a file on PRIMOS Unit 13 prior to invoking SEG's loader, then this file will be used for the map. In this case, leaving the Loader does *not* close the file.

The full SEG load map consists of seven sections, not all of which may be present in any load. (See Figure 5-1) In particular, Section III may not be present in small SEG loads. The amount of information printed is controlled by MAP option codes:

Option	Load Map Information
None, 0 or 4	Extent, segment assignments, base areas, symbol storage (symbols sorted by address), direct entry links, common blocks, and other symbols.
1	Extent and segment assignments only
2	Extent, segment assignments and base areas
3	Undefined symbols, sorted by address
6	Undefined symbols, sorted alphabetically
7	Full map, symbols, sorted in alphabetic order
10	Symbols, sorted by ascending address
11	Symbols, sorted alphabetically

Section I—Extent

The extent area shows where the program has been loaded, the start-of-execution location, and the size of the symbol table. All locations are octal numbers.

***START:** The segment number and word location for the start-of-execution. At the beginning of a load, the start address is initialized to 000000 000000. SEG fills in *START for the first segmented procedure encountered (usually the main program).

***STACK:** Segment number and word location of the start of the stack; initialized to 177777 000000 at the start of a load. This value is not changed until a loader SAVE or EXECUTE command is invoked. The default stack is in the first procedure segment with 6000 (octal) free locations at the top of memory.

***SYM:** Address of the bottom of the symbol table (one word only as it is a 64R mode address). Indicates to the user how much space is left for the symbol table. To determine the location of the top of the symbol table, generate a map prior to loading; the top and bottom of the symbol table will be identical and *SYM will also be the location of the top.

Section II—Segment assignments

Each segment is labeled as procedure (PROC) or data (DATA); the segment chosen for the stack is identified by ## following the segment type. The list is sorted in order of segment assignment.

LOW: Lowest *loaded* location in the segment. (Not necessarily the lowest *assigned* location.) Initialized to '177777 (-1) at segment creation; if the segment is used only for uninitialized COMMON areas, LOW is not changed.

HIGH: Highest *loaded* location in the segment. (Not necessarily the highest *assigned* location.) Initialized to '000000 at segment creation; if the segment is used only for uninitialized COMMON areas, HIGH is not changed.

```
*START  004002  000003  *STACK  004001  011720  *SYM    000146

SEG. #     TYPE       LOW      HIGH      TOP
004001     PROC##     000100   011723    011717
004002     DATA       000001   100462    100553

*BASE   004001  000100  000177  000777  000777

ROUTINE        ECB           PROCEDURE      ST. SIZE  LINK FR.
####      4002  000003    4001  001000    000012  000035  4002  177400
F$WB      4001  005371    4001  001067    000060  000107  4002  076035
F$RB      4001  005331    4001  001072    000060  000107  4002  076035
F$DE      4001  005411    4001  001100    000060  000107  4002  076035
F$EN      4001  005431    4001  001103    000060  000107  4002  076035
F$WA      4001  005351    4001  001123    000060  000107  4002  076035
F$RA      4001  005311    4001  001126    000060  000107  4002  076035

F$A1      4001  005451    4001  001520    000060  000107  4002  076035
F$A2      4001  005471    4001  001523    000060  000107  4002  076035
F$A5      4001  005511    4001  001526    000060  000107  4002  076035
F$A6      4001  005531    4001  001533    000060  000107  4002  076035
F$A7      4001  005551    4001  001536    000060  000107  4002  076035
F$CB      4001  005571    4001  002230    000060  000107  4002  076035
RDASC     4001  005736    4001  005611    000026  000006  4002  076344
RDBIN     4001  005756    4001  005656    000026  000006  4002  076344
WRASC     4001  005776    4001  005676    000026  000006  4002  076344
WRBIN     4001  006016    4001  005716    000026  000006  4002  076344
IOCS$     4001  006136    4001  006044    000040  000004  4002  076352
IOCS$T    4001  006707    4001  006174    000010  000152  4002  076356
ATTDEV    4001  006727    4001  006250    000014  000152  4002  076356
IOC$RA    4001  006747    4001  006313    000006  000152  4002  076356
I$BD07    4001  007117    4001  007004    000036  000002  4002  076530
O$BD07    4001  007236    4001  007142    000034  000002  4002  076532
O$AD08    4001  007350    4001  007261    000034  000002  4002  076534
I$AA12    4001  007616    4001  007373    000052  000013  4002  076536
O$AA01    4001  007701    4001  007652    000030  000002  4002  076551
F$IOER    4001  007752    4001  007722    000014  000006  4002  076553
I$AD07    4002  077170    4001  010015    000030  000056  4002  076561
O$AD07    4002  077246    4001  010131    000030  000056  4002  076637
PRWFIL    4002  077323    4001  010245    000046  000055  4002  076715
PRSPE$    4002  077377    4001  010444    000044  000054  4002  076772
SEARCH    4002  077460    4001  010605    000034  000121  4002  077046
OERRT$    4002  077574    4001  011014    000052  000662  4002  077167
F$ERX     4001  011200    4001  011154    000020  000006  4002  100051
TONL      4001  011230    4001  011221    000012  000002  4002  100057
GCHAR     4001  011251    4001  011271    000020  000000  4002  100061
SCHAR     4001  011310    4001  011330    000024  000000  4002  100061
T1OB      4001  011374    4001  011355    000046  000002  4002  100061
GETERR    4001  011460    4001  011415    000040  000000  4002  100063
ERRSET    4001  011631    4001  011500    000034  000071  4002  100063

DIRECT ENTRY LINKS
  CNIN$     4001  011652    ERKL$$    4001  011656    ERRPR$    4001  011662
  EXIT      4001  011666    PRWF$$    4001  011671    RDLIN$    4001  011675
  SRCH$$    4001  011701    TNOU      4001  011705    TNOUA     4001  011710
  WTLIN$    4001  011714

COMMON BLOCKS
            4002  000035  076400

OTHER SYMBOLS
  F$A3      4001  005451    **GHOST   4002  000033    F$IOBF    4002  076544
  PUTBL     4002  077012    RSTBL     4002  077046
```

Figure 5-1. Example of Load Map

TOP: Highest *assigned* location in the segment. Top should not be lower than HIGH. If it is, the user may have specified incorrect load addresses. When not using default values, the user is responsible for loading into correct areas. TOP is initialized to '177777 (-1) at segment creation. When space is reserved for large COMMON blocks, the loader will only set TOP to a maximum of '177776 even though the entire segment to '177777 is reserved.

The reason for this is: a LOW, HIGH, and TOP of 177777 000000 177777 labels an *empty* segment.

Section III—Base areas

***BASE VVVVVV WWWWW XXXXX YYYYYY ZZZZZZ**

VVVVVV	Segment number
WWWWWW	Lowest location for base area
XXXXXX	Next available location if starting up from lowest location
YYYYYY	Next available location if starting down from highest location
ZZZZZZ	Highest location for base area

The lowest default location for the sector zero base area is '100.

There may be a sector zero base area in each procedure segment; there must be none in data segments. Base areas other than sector zero ones may be generated by PMA modules.

Section IV—Symbols

A main program or subroutine compiled in 64V or 32I mode is called a procedure. For a complete discussion, see Sections 9 and 10 in this manual; also the Reference Guide, System Architecture. A procedure is composed of a procedure frame (the executable code), an ECB (the entry control block which points to the procedure frame), a link frame (static storage, constants, transfer vectors) and a stack frame (dynamically allocated storage which is assigned when the routine is called and released upon return from the routine). This section of the map describes these items. The ECB is normally part of the link frame although the programmer may place it in the procedure frame. The procedure frame will be located in a segment reserved for procedure frames. Link frames and COMMON blocks will be located in segments reserved for data.

The first pair of numbers in this section of the map is the segment and word address for the ECB; the second pair is the segment and word address for the procedure.

ST. SIZE: is the size of the stack frame (working area) created whenever the routine is called. Its segment (and location therein) are assigned at execution time.

LINK FR: is the size of the link frame.

The last two columns are the link frame segment and offset. Note that the offset is '400 locations lower than the actual position, for compatibility with the information printed by the PRIMOS PM command. The segment number is usually that for the ECB.

Procedures with no names, specifically a main program, are identified by #### in the name field.

Section V—Direct entry links

PRIMOS supports direct entry calls to the supervisor for certain routines. These are created as fault pointers in the SEG runfile. Where references are satisfied by these fault pointers, they will appear in the DIRECT ENTRY LINKS section of the map.

Section VI—COMMON blocks

Lists each COMMON block, its segment number, starting word address in the segment, and size.

Section VII—Other symbols (including undefined symbols)

Lists the symbol, its segment, and word address in that segment. As in Section VI, the format is three symbols per line. Unsatisfied references are preceded by **

The numbers for unsatisfied references (segment and word address) locate the last request for the routine processed by the loader. This allows the routines calling missing routines to be identified.

ADVANCED SEG FEATURES

When standard loading goes well, the user can ignore most of the SEG's advanced features. However, situations can arise where some detailed knowledge of SEG and segmented runfile organization can optimize size or performance of a runfile, or even make a critical load possible. The following topics are particularly valuable.

Segment usage

A segment is a 64K word block of user's virtual address space. Segment '4000 is the segment that SEG and other external commands occupy when invoked. Segment '4000 is the lowest-valued non-shared segment in the PRIMOS system. SEG creates a runfile of up to 256 segments.

PRIMOS assigns memory segments to a user as they are accessed. These are not re-assigned until logout. Since only a fixed number of segments are available for all users, extra segments should not be invoked unless the user is actually executing or examining a segmented program. Most of the functions of SEG use only one segment; only those options which restore a runfile use extra segments, i.e., RESTORE, RESUME, and EXECUTE.

Segmented runfiles

A segmented runfile consists of segment subfiles in a segment directory. For this reason, you *cannot* delete a SEG runfile with a PRIMOS-level DELETE command. Instead, use the DELETE command in SEG. (The TREDEL command in FUTIL also works but is slower than SEG's DELETE.)

Note

It is good practice to use the PRIMOS DELSEG command to release segments assigned by SEG during a load session. Otherwise those segments remain assigned to the user until logout, precluding their use by anyone else.

Each segment of the runfile consists of 32 ('40) subfiles of '4000 words each. Subfile 0 of the runfile is used for startup information, the load map, and the memory image subfile map. Memory image subfiles begin in segment subfile 1. Only the subfiles actually required for the runfile are stored on the disk.

SEG's loader

SEG has a virtual loader (i.e., it loads to a file rather than to memory) which requires the name of the runfile before anything is loaded. The runfile may be new or may be a previously used SEG runfile, and may be in any directory. A runfile compiled and loaded in 32R or 64R mode may not be used.

As the symbol table is always available, SEG's loader may be used to add modules to an existing runfile. Similarly, a partial load may be saved with the SEG SAVE command and the load completed later. In addition, selected modules may be replaced in a SEG runfile.

Object files

Object files of the program modules must have been created using the SEG or SEGR pseudo-operation. Modules written in other languages may also be loaded, if they have been compiled or assembled in 64V or 32I mode.

Code and data are loaded in separate segments to support re-entrant procedures. Data includes all COMMON blocks and link frames. The loader assigns code and data segments. The first segment ('4001) is used for code. Usually segment '4002 will be used for data. The loader loads data and code into appropriate segments and opens new segments as required. It is possible to put both code and data in the same segment to save space, using the MIXUP subcommand of the LOAD subprocessor.

The stack

The loader assigns a stack (a dynamic work area) when SAVE or EXECUTE is invoked. The stack is usually assigned as the next free location in the first procedure segment with '6000 free words. If no such segment exists, a new data segment is assigned with the first location in the stack set to 4; locations 0 to 3 are used for internal SEG information. The user may force the location of the stack and/or may change its size.

Use of pathnames

Pathnames can be used to specify object files in all commands except LIBRARY, which accepts only a simple filename of a file within the LIB UFD.

Base areas

Base areas normally present no problem unless the following message is printed:

SECTOR 0 BASE AREA FULL

This condition, which is extremely *unlikely* to occur, can be avoided by using the SETBASE command or the SETB pseudo-operation to designate a base area where it is required.

Locating COMMON

SEG makes sure there is no overlap of procedures and COMMON. The user has the option of moving COMMON by a COMMON or SYM command, but he takes on the responsibility of making sure it doesn't run into the stack.

COMMAND SUMMARY

Following is a summary of all SEG commands, in alphabetical order within three groups:

1. SEG-level commands
2. LOAD-subprocessor
3. MODIFY subprocessor

Files and directory names may be specified by pathnames, except in the LIBRARY commands. All numerical values must be octal. The following conventions are followed for parameters.

addr	Word address within a segment.
segno	Segment number.
psegno	Procedure segment number.
lsegno	Linkage segment number.
[a][b][x]	Values for A, B, and X registers.

Note

Segment numbers may be absolute or relative.

SEG-LEVEL COMMANDS

Commands at SEG level are entered in response to the "#" prompt.

DELETE [pathname]

Deletes a saved SEG runfiles.

HELP

Prints abbreviated list of SEG commands at terminal.

[V]LOAD[pathname]

Defines runfile name and invokes virtual loader for creation of new runfile (if name did not exist) or appending to existing runfile (if name exists). If **pathname** is omitted, SEG requests one.

MAP pathname-1 [pathname-2] [map-option]

Prints a loadmap of runfile (**pathname-1** or current loadfile (*)) at terminal or optional file **(pathname-2)**.

Option	Load Map information
0	Full map (default)
1	Extent map only
2	Extent map and base areas
3	Undefined symbols only
4	Full map (identical to 0)
5	System programmer's map
6	Undefined symbols, alphabetical order
7	Full map, sorted alphabetically
10	Symbols by ascending address
11	Symbols alphabetically

MODIFY [filename]

Invokes MODIFY subprocessor to create a new runfile or modify an existing runfile.

PARAMS [filename]

Displays the parameters of a SEG runfile.

PSD

Invokes VPSD debugging utility.

QUIT

Returns to PRIMOS command level and closes all open files.

RESTORE [pathname]

Restores a SEG runfile to memory for examination with VPSD.

RESUME [pathname]

Restores runfile and begins execution.

SAVE [pathname]

Synonym for **MODIFY**.

SHARE [pathname]

Converts portions of SEG runfile corresponding to segments below '4001 into R-mode-like runfiles.

SINGLE [pathname] segno

Creates an R-mode-like runfile for any segment.

TIME [pathname]

Prints time and date of last runfile modification.

VERSION

Displays SEG version number.

VLOAD

See **LOAD**.

LOAD SUBPROCESSOR COMMANDS

ATTACH [ufd-name] [password] [ldisk] [key]

Attaches to directory.

A/SYMBOL symbolname [segtype] segno size

Defines a symbol in memory and reserves space for it using absolute segment numbers.

COMMON { [ABS] | REL } segno

Relocates COMMON using absolute or relative segment numbers.

D/ { IL | LOAD | LIBRARY | FORCELOAD | PL *or* RL }

Continues a load using parameters of previous load command.

Note

D/ and F/ may be combined, as in D/F/LI.

EXECUTE [a] [b] [x]

Performs SAVE and executes program.

F/ { IL | LOAD | LIBRARY | FORCELOAD | PL | RL } [pathname][addr psegno lsegno]

Forceloads all routines in object file.

IL [addr psegno lsegno]

Loads impure FORTRAN library IFTNLB

INITIALIZE [pathname]

Initializes and restarts load subprocessor.

LIBRARY [filename] [addr psegno lsegno]

Loads library file (PFTNLB and IFTNLB if no filename specified).

LOAD [pathname] [addr psegno lsegno]

Loads object file.

MAP [pathname] option

Generates load map (see SEG-level MAP command).

MIXUP { [ON] | OFF }

Mixes procedure and data in segments and permits loading of linkage and common areas in procedure segments. *Not* reset by INITIALIZE.

MV [start-symbol move-block desegno]

Moves portion of loaded file (for libraries). If options are omitted, information is requested.

OPERATOR option

Enables or removes system privileges 0=enable, 1=remove. *Caution: this command is intended only for knowledgeable creators of specialized software.*

PL [addr psegno [segno]

Loads pure FORTRAN library, PFTNLB.

P/ { IL | LOAD | LIBRARY | FORCELOAD | PL | RL } [pathname] option [psegno] [lsegno]

Loads on a page boundary. The **options** are: PR=procedure only, DA=link frames only, none =both procedure and link frames.

QUIT

Performs SAVE and returns to PRIMOS command level.

RETURN

Performs SAVE and returns to SEG command level.

RL pathname [addr psegno lsegno]

Replaces a binary module in an established runfile.

R/SYMBOL symbol-name [segtype] segno size

Defines a symbol in memory and reserves space for it using relative segment assignment. (Default=data segment).

SAVE [a] [b] [x]

Saves the results of a load on disk.

SETBASE segno length

Creates base area for desectorization.

SPLIT { **segno addr** | **addr** | **addr segno addr lsegno** }

Splits segment into data and procedure portions. Formats 2 and 3 allow R mode execution if all loaded information is in segment 4000.

SS symbol-name

Saves symbol; prevents XPUNGE from deleting symbol-name.

STACK size

Sets minimum stack size.

SYMBOL [symbol-name] segno addr

Defines a symbol at specific location in a segment.

S/ { **LIBRARY** | **FORCELOAD** | **PL or IL** | **RL or LOAD** } **[pathname] [addr psegno lsegno]**

Loads an object file in specified absolute segments.

XP dsymbol dbase

Expunges symbol from symbol table and deletes base information.

symbol	Action
0	Delete all defined symbols—including COMMON area.
1	Delete only entry points, leaving COMMON areas.

dbase	Action
0	Retain all base information.
1	Retain only sector zero information.
2	Delete all base information.

MODIFY SUBPROCESSOR COMMANDS

NEW pathname

Writes a new copy of SEG runfile to disk.

PATCH segno baddr taddr

Adds a patch (loaded between **baddr** and **taddr**) to an existing runfile and saves it on disk.

RETURN

Returns to SEG command level.

SK { **ssize** | **segno addr** | **ssize 0 esegno** | **ssegno addr esegno** }

Specifies stack size (**ssize**) and location. **esegno** specifies an extension stack segment.

START segno addr

Changes program execution starting address.

WRITE

Writes all segments above '4000 of current runfile to disk.

6
Executing

This section treats the following topics:

- Execution of program memory images saved by the linking loader.
- Execution of segmented runfiles saved by SEG's loader.
- Installation of programs in the command UFD (CMDNC0).
- Use of run time.

EXECUTION OF UNSEGMENTED RUNFILES

Use the PRIMOS RESUME command to execute an unsegmented runfile:

RESUME pathname

where **pathname** is an R-mode runfile in the current UFD.

Programs which are resident in the user's memory may be executed by a START command:

START

RESUME

RESUME brings the memory-image program **pathname** from the disk into the user's memory, loads the initial register settings, and begins execution of the program. Its format is:

RESUME pathname

Example:

```
OK, R *TEST              User requests program
GO                       Execution begins
THIS IS A TEST           Output of program

OK,                      PRIMOS requests next command
```

RESUME should *not* be used for segmented (64V or 32I mode) programs; use the SEG command (discussed later) instead.

START

Once a program is resident in memory (e.g., by a previous RESUME command) you can use START to initialize the registers and begin execution. Its format is:

START [start-address]

Upon completion of the program, control returns to PRIMOS command level.

EXECUTION OF SEGMENTED RUNFILES

Use the SEG command to begin execution of a segmented program; e.g. **SEG pathname** where **pathname** is a SEG runfile. SEG loads the runfile into segmented memory and starts execution. SEG should be used for runfiles created by SEG's Loader; it should not be used for program memory images created by the LOAD utility.

Example:

```
OK, SEG #TEST              user requests program
GO                         execution begins
THIS IS A TEST             output of program

OK,                        PRIMOS requests next command
```

Upon completion of program execution, control returns to the PRIMOS command level.

You may restart a SEG runfile by the command: S 1000, provided both the SEG runfile and the copy of SEG used to invoke it are in memory.

INSTALLATION IN THE COMMAND UFD (CMDNC0)

Run-time programs in the command UFD (CMDNC0) can be invoked by keying in the program name alone. This feature of PRIMOS is useful if a number of users invoke this program. Only one copy of the program need reside on the disk in UFD CMDNC0.

Even more space is saved during execution by multiple users if the program uses shared code (64V and 32I mode only).

Program memory images saved by LOAD

Installation in the command UFD is extremely simple, providing you have access to the password. The runtime version of the program is copied into UFD CMDNC0 using PRIMOS' FUTIL file handling utility.

Example: Assume you have written a utility program called FARLEY. This utility acts as a "tickler" for dates. Using FARLEY, each user builds a file with important dates. The FARLEY utility program, upon request, prints out upcoming events or occasions of interest to the user.

Note

This utility does not necessarily actually exist; it is used as a plausible example.

First, assemble the program.

```
OK, PMA FARLEY -64R        Assemble in 64R mode
GO
0000 ERRORS (PMA-REV 16.2)ASSEMBLER MESSAGE
OK, LOAD                   Invoke the Loader
GO
$LO B_FARLEY               Load the object file; the default
                             name is used
$                          Load other required modules
 .
 .
 .
```

```
$LI                       Load the FORTRAN library
LOAD COMPLETE             Load is complete
$SA *FARLEY               Save the memory image
$QU                       Return to PRIMOS

OK, FUTIL                 Invoke the file utility
GO
>TO CMDNCØ ORDER          Defines the TO UFD as CMDNCØ;
                          password is ORDER
>COPY *FARLEY FARLEY      Copies the runtime program
                          *FARLEY into UFD=CMDNCØ
                          under the name of FARLEY
>QUIT                     Return to PRIMOS Command level

OK,
```

It was not necessary to define a FROM UFD: the default was used. Any user can now invoke this program:

```
OK, FARLEY                Invoke program
GO                        Execution beings

HOW FAR:                  Asks for future time period

etc.
```

Segmented runfiles saved by SEG's loader

A segmented program cannot be run directly from UFD CMDNC0 because PRIMOS' command processor cannot directly handle the SEG runfiles. The segmented program may be invoked by means of a non-segmented interlude program in CMDNC0.

The procedure for creating an interlude is:

1. Create the desired SEG runfile.
2. Attach to UFD SEG.
3. Run the command file CMDSEG; it will ask for a runfile name—this name is the new SEG runfile name used in step four. This command file will create the interlude program under the name *TEST.
4. Make a copy of the SEG runfile in UFD SEG using FUTIL's TRECPY command. The name of the new SEG runfile should be the name used in step three.
5. A copy of *TEST should be placed in UFD CMDNC0 using FUTIL's COPY command. The file name should be that by which the program will be invoked.

Note

If a pathname is given in step three, the runfile need not reside in UFD SEG (step four can be skipped).

Example:

1. Extensions to the FARLEY utility described above make it desirable to assemble and load it as a segmented program.

```
OK, PMA FARLEY -64V      Assemble in 64V mode
GO
0000 ERRORS (PMA-REV 16.2)

OK, SEG                  Invoke SEG utility
GO
# LOAD #FARLEY           Establish runfile name
$ LO B_FARLEY            Load object file
$ .
  .
  .
$ LI                     Load 64V mode FORTRAN library
$ SA                     Save the file
$ QU                     Return to PRIMOS
OK,
```

2. Attach to UFD SEG.

```
OK, A SEG
OK,
```

3. The command file CMDSEG creates the interlude program.

```
OK, CO CMDSEG

OK, * CMDSEG,SEG,CEH. 04/05/78
OK, * COMMAND.FILE.TO.CREATE.'CMDNC0'.SEG.RUNFILES
OK, R *CMDMA
GO
RUN FILE NAME: FARLEY
OK, PMA S$$SEG 1/5707
GO
0000 ERRORS (PMA-REV 16.2)
OK, FILMEM
OK, LOAD
$SZ
$ER 2
$MO D64R
$CO 173400
$LO B_S$$SEG 173400
$AU 2
$LO CMDLIB * 12 14 14 0 0 12 0 0 12
$AU 0
$LI
$MA 2


$SAVE *TEST
$AT
```

```
$QU
OK, DELETE S$$SEG
OK, DELETE B_S$$SEG
OK, CO TTY

OK,
```

4. UFD SEG contains the SEG runfiles which are actually executed by the interlude programs. The SEG runfile is copied here from the UFD in which it was SAVEd.

```
OK, FUTIL                  Invoke FUTIL
GO
>FROM MYUFD                FROM UFD is user's old home UFD
>TRECPY #FARLEY FARLEY     Make a copy under the invocation
```

Note

No TO UFD is defined since the default (home) is being used.

5. The interlude program *TEST is copied into the command UFD under the name by which it will be invoked.

```
>FROM *                    New FROM UFD - the current home
>TO CMDNCØ ORDER           TO UFD=CMDNCØ; password here
                             is assumed to be ORDER
>COPY *TEST FARLEY         Copy the interlude
>QUIT                      Return to PRIMOS command level

OK,
```

When FARLEY is entered at the user terminal, the FARLEY interlude program in CMDNC0 is executed. This program attaches to the SEG UFD, restores the segmented runfile FARLEY, re-attaches to the user's home UFD and begins execution of the SEG runfile.

If the SEG runfile requires only one segment of loaded information (procedure, link frames, and initialized common) in user space (segment '4000 and above) it is possible to include the interlude in the SEG runfile.

RUN-TIME ERROR MESSAGES

Appendix C contains a list of error messages which you may receive during execution, along with their meaning and origin.

7 Debugging

Debugging is really an art and an attitude, rather than a set of techniques. In accord with this, we will present some tools and we hope you will use them, develop your own ideas, and tell us about them.

TOOLS

You have a variety of tools from which to choose. Which you select is extremely context sensitive. The partial list below is intended to be an initial guide.

Tool	Where Described
PM command	Described below
COMOUTPUT command	Described below
PMA Error Messages	Appendix C of this document
Load Maps	Sections 4 and 5 of this document
Debugging Utilities	Sections 18–21 of this document

RVEC parameters

The commands RESTORE, RESUME, SAVE, PM, and START process a group of optional parameters associated with the PRIMOS RVEC vector. These parameters are stored on disk along with a starting address (SA) and ending address (EA), for every program saved by the SAVE command.

Initial values for the RVEC parameters are usually specified in the PRIMOS SAVE command, or by the Loader's SAVE command that stored the program on disk.

Each parameter is a 16-digit processor word represented by up to six octal digits.

PM command

The PM (Post Mortem) command prints the contents of the RVEC vector. PRIMOS first prints labels for the items in RVEC, then prints the values on the line in the same order. PM is an internal command and does not overlay user memory.

The Prime 350 and above contain additional registers which PM displays: the procedure base register (PB), the stack base register (SB), the link base register (LB), and the temporary base register (XB). These 32-bit registers are displayed at the user terminal on a text line separate from the other registers. Each of the Prime 400 registers is displayed as two 16-bit octal numbers separated by a slash (/) character.

Example:

```
OK, PM
SA,EA,P,A,B,X,K=
100 11763 5517 120240 20061 23534 14000

PB,SB,LB,XB:
62000/5517 64000/74012 4000/3400 11/15041
```

The above example of PM under PRIMOS IV shows a PB of 64000/3043, which indicates: ring 3 (See the System Architecture Guide for a discussion of rings), segment '4000. The word number portion of PB indicates the same number as the P parameter of PM. This number specified the location within the segment to execute the next instruction upon possible receipt of a START command. The other base registers shown in the example contain a 0, indicating that they have not been used since LOGIN. Programs that run in one of the Prime 300 addressing modes use segment 4000 ring 3, and give values as a result of invoking PM in the form shown by the example.

COMOUTPUT files

PRIMOS has a very useful tool for anyone who is debugging — or who wishes to record a particular situation. You may direct all the interactive terminal dialog to a file. This means that you can have a complete trace of a debugging session without a hard copy terminal. In addition, you can edit this file, print it out and delete it as you chose. The command is:

COMOUTPUT pathname

where **pathname** is the output file. To stop the COMO file creation process, type:

COMO -END

ADVANCED DEBUGGING TECHNIQUES

Section 9—Instruction and Data Formats, contains useful debugging data structures, such as ECB and stack frame layouts

DEBUGGING—PRIMOS SEVERE ERRORS

The following list describes several severe error conditions. In all cases the errors are fatal.

NO VECTOR — A fault occured and there is no user vector in segment '4000 to process it. There are several possibilities as to why this condition occurs:

1. Stack Overflow—Usually there is no vector because the stack is too small. Do a PM. If SB is close to top of its segment use SEG to move stack and increase its size, or create a stack extension. With the dynamic variables features of FTN, stack overflow may become a problem.
2. FLEX (Floating Point Exception) If this occurs, your program has wiped out segment '4000. Check to see that you are using '4000 properly. If so, your program is sick. To ascertain this, examine location '74 with PSD or TAP. If 0 then it is wiped out.
3. Others. The implication is that your program has gone off into random memory and has wiped things out.

POINTER FAULT — Either there is a missing argument for some subroutine call, you did not get a load complete when loading, or the program has written over its link frame and/or procedure segment. Use VPSD to examine the current stack frame and SEG to get a MAP 3 or MAP 6.

ILLEGAL SEGNO — Probably some pointer in a link or stack frame has been wiped out (i.e., referencing an array with subscripts out-of-range). Use VPSD to examine the current and past link and stack frames.

ACCESS VIOLATION — An attempt was made to access a segment for which the user does not have proper access rights. Cause usually the same as an illegal segno

Note

Most frequent cause of ILLEGAL SEGNO and ACCESS VIOLATION is an improperly dimensioned local or common array. Try getting common away from link frames by reloading.

MEMORY OVERFLOW ERRORS (MO)

As user programs become larger, MO (memory overflow) errors will become more common. Several causes and solutions to these errors follow:

When an MO error occurs, do a MA 2 and examine the resulting map for any of the following situations:

1. The address of the bottom of the symbol table (*SYM) is at or close to PBRK. This indicates that there is insufficient room below the loader for the whole program. Using HILOAD will probably solve the problem, unless of course the user is already using HILOAD. If this is the case there are only two alternatives; redesign the entire program or make hardware changes.
2. The sector zero base area is full. The next free location is '1000. The size of the sector zero base area may be increased by using a SETB 100 command at the beginning of the load (if locations 100 to 200 are free). An AU xx command may be used to insert base areas throughout the load, where xx is a small octal number which sets the size of the base area to be inserted.
3. *CMLOW is higher than *CMHGH. The total size of all the common blocks is too large causing common to wrap around through zero to high memory. Possibly common may be moved to the top of memory, if not already done. If there is more than 64K of common, and this cannot be reduced, the program cannot be run in "R"mode. A segmented program is required.
4. Others. The program requires initialized common. Common is usually defaulted to overwrite the space used by the Loader. The locations between the bottom of the symbol table and the top of the loader cannot be initialized. This would destroy the loader. Use a COMMON command to move common out of the way of the loader. HILOAD can be used to permit common to utilize locations normally used by LOAD.

8
Interfacing with the system libraries

Most of the commonly used subroutines — I/O, math functions and EXIT, are either embedded in the operating system or are in one of the FORTRAN libraries. LOAD and SEG automatically load the appropriate library when you type the command LI during a loading sequence. Other libraries, such as APPLIB and MATHLIB require the specification of their name following LI — e.g. LI APPLIB causes the application library to be searched for unresolved references.

Table 8-1 lists the commonly available system libraries. See the Reference Guide, PRIMOS Subroutines for complete descriptions of the system subroutines.

All routines, regardless of mode, should use the CALL pseudo-operation to call subroutines. S and R-mode arguments use DAC pointers; V, and I-mode arguments use AP pointers (see Section 16 for the DAC and AP pseudo-operation formats). Figure 8-1 illustrates the SR calling sequences and associated subroutine code; Figure 8-2 illustrates VI calling sequences and associated subroutine code.

Table 8-1. System Libraries

Name	Description	Mode
FTNLIB	FORTRAN Library	R
PFTNLB	FORTRAN Library pure procedures	V
IFTNLB	FORTRAN Library impure procedures	V
APPLIB	Application Library	R
VAPPLB	Application Library	V
SRTLIB	Sort Library-Files	R
VSRTLI	Sort Library-Files	V
MSORTS	Sort Library-Memory	R
MATHLB	Matrix Routines	R

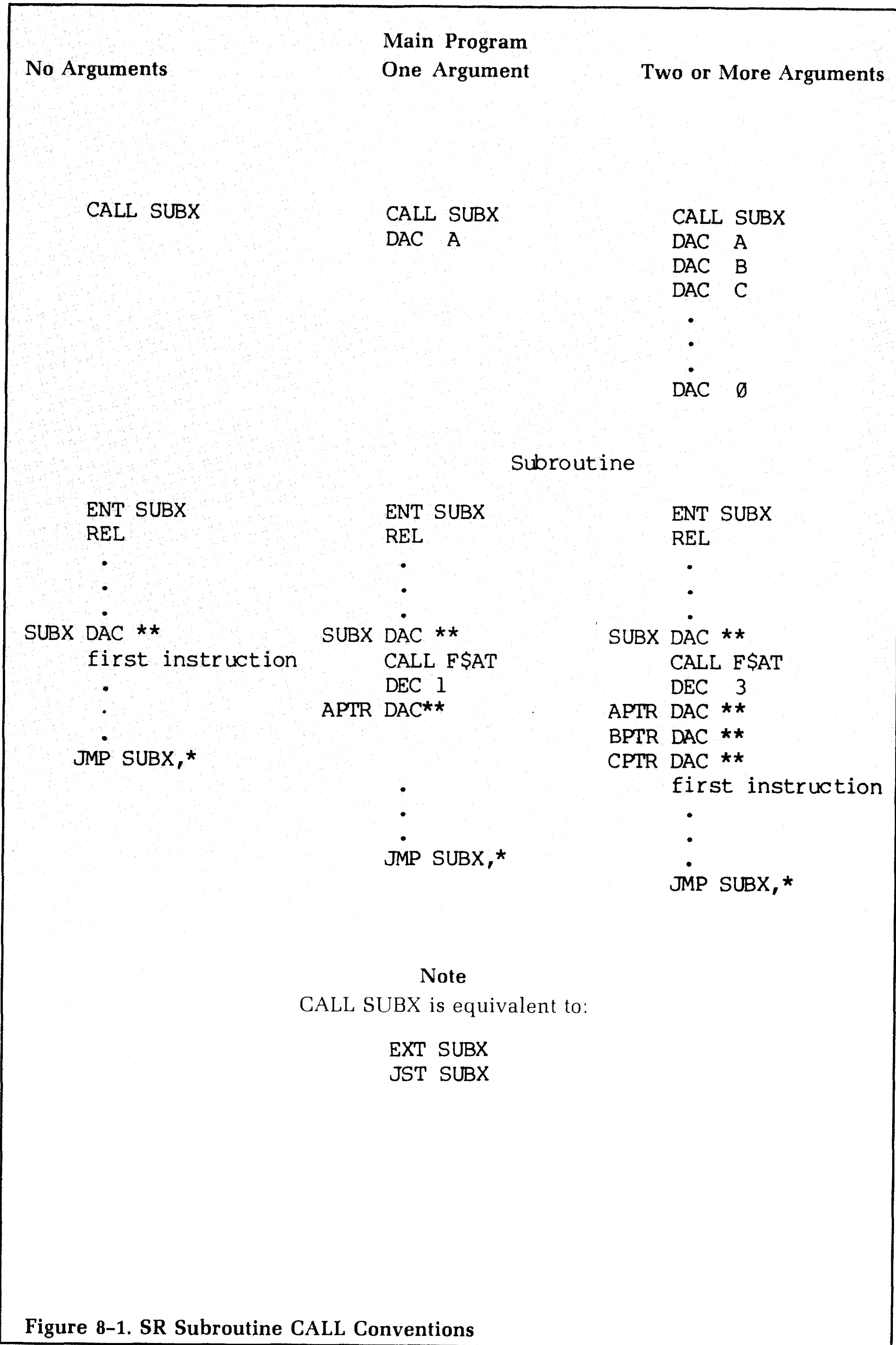


Figure 8-1. SR Subroutine CALL Conventions

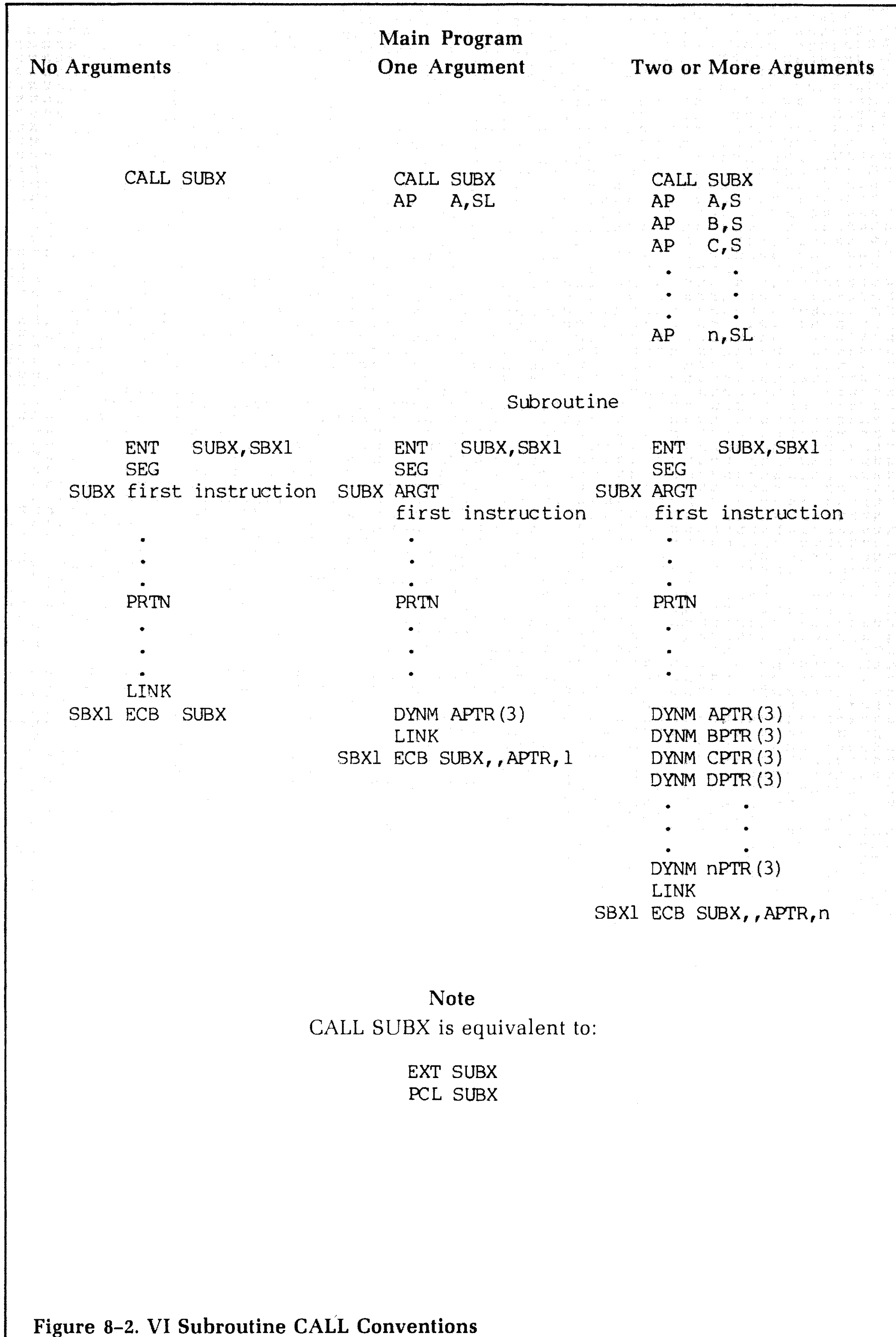


Figure 8-2. VI Subroutine CALL Conventions

3 MACHINE FORMATS AND INSTRUCTIONS

9

Data and instruction formats - SRVI

DATA STRUCTURES

Word length

- 16 bits (*SRV*)
- 32 bits (*I*)

Byte length

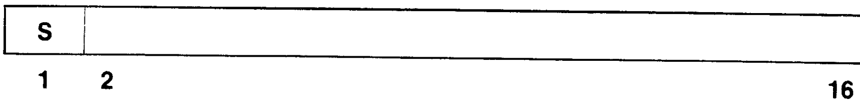
- 8 bits (*SRVI*)

Character strings

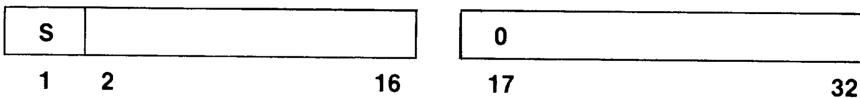
- Variable length collection of bytes from 1 to 2**17–1.

Numbers

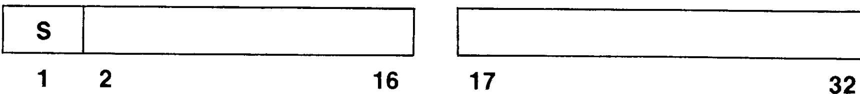
- Unsigned 16 bit integers (*SRV*)
- Unsigned 32 bit integers (*SRVI*)
- Unsigned 64 bit integers (*I*)
- Signed 16-bit integers (*SRVI*)



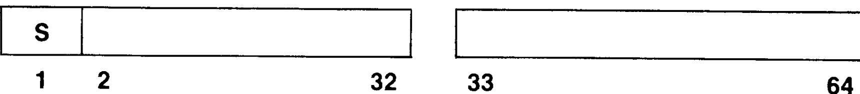
- Signed 31-bit integers (*SR*)



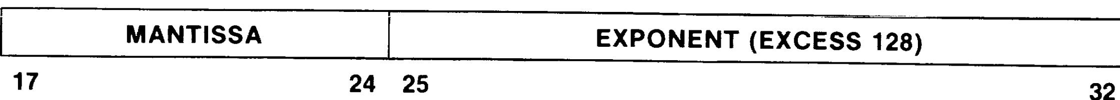
- Signed 32-bit integers (*VI*)



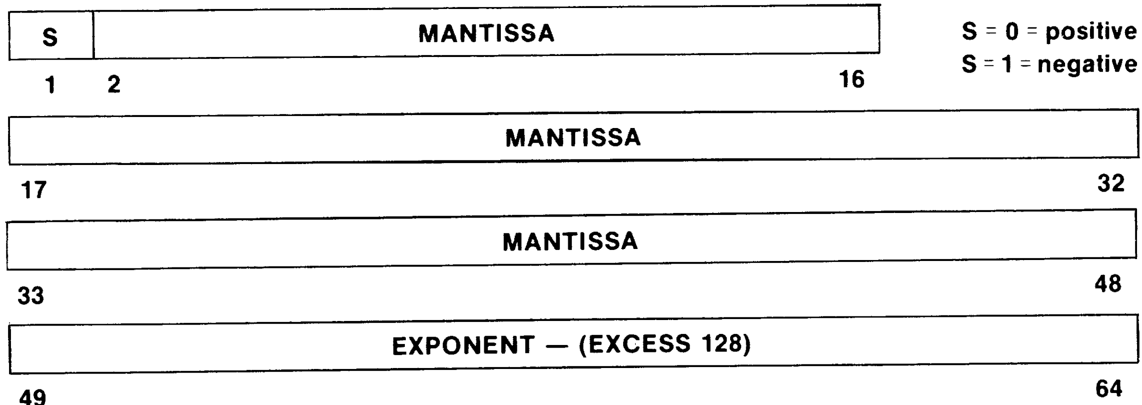
- Signed 64-bit integers (*VI*)



- Floating Point – Single Precision 32 bits (*RVI*)



- Floating Point - Double Precision 64 bits (*RVI*)

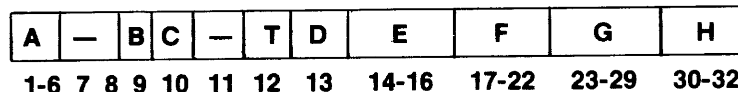


- Decimal - one to 63 digits in five forms (*VI*)

Decimal control word format (VI)

To specify the characteristics of the operation to be performed, most decimal arithmetic instructions require a control word to be loaded in the L register (general register 2 in I-Mode).

The general format is as follows:



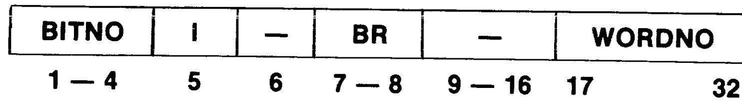
A (Bits 1–6)	Field 1, number of digits
E (Bits 14–16)	Field 1, decimal data type
B (Bit 9)	If set, sign of field 1 is treated as opposite of its actual value.
C (Bit 10)	If set, sign of field 2 is treated as opposite of its actual value. *XAD, XMP, XDV, XCM only*)
D (Bit 13)	Round flag (*XMV only*)
F (Bits 17–22)	Field 2, number of digits
H (Bits 30–32)	Field 2, decimal data type
G (Bits 23–29)	Scale differential (XAD, XMV, XCM and number of multiplier digits in XMP)
T (Bit 12)	Generate positive results always
	Unused, must be zero

The fields used by each instruction are listed in the instruction descriptions. Fields not used by an instruction must be zero.

The scale differential specifies the difference in decimal point alignment between the operator and fields for some instructions. This field is treated as a signed 7 bit two's complement number. Its value is specified as Fx=F1-F2, where Fx is the number of fractional digits in Field x. A positive value indicates a right shifting of Field 1 with respect to Field 2, and a negative value indicates a left shifting.

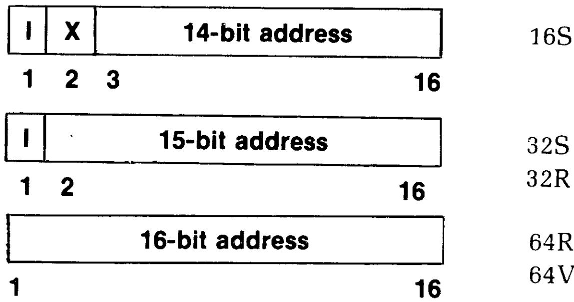
Address pointer (AP) (VI)

Two word pointer which follows AP instructions.



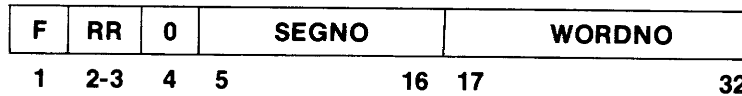
BITNO (Bits 1-4)	Bit number
I (Bit 5)	Indirect bit
BR (Bits 7-8)	Base register
	00 Procedure Base (PB)
	01 Stack Base (SB)
	10 Link Base (LB)
WORDNO (Bit 17-32)	Word number offset from base register contents

Indirect word - one word memory reference (SRV)



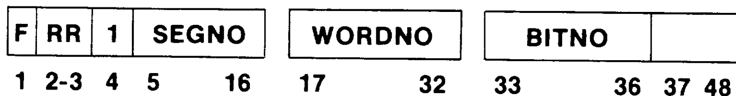
I (Bit 1)	Indirect bit
X (Bit 2)	Index bit

Indirect pointer - two word memory reference (IP) (VI)



F (Bit 1)	Generate pointer fault if set. In the fault case, the entire first word (bits 1-16) forms a fault code, and no other bits are inspected.
RR (Bits 2-3)	Ring of privilege - controls access rights
Bit 4 = 0	No third word. Bit number portion of effective address is zero.
SEGNO (Bits 5-16)	The segment number portion of the effective address
WORDNO (Bit 17-32)	The word number portion of the effective address.

Indirect pointer - three word memory reference (IP) (VI)



F (Bit 1)	Generate pointer fault if set. In the fault case, the entire first word (bits 1-16) forms a fault code, and no other bits are inspected.
RR (Bits 2-3)	Ring of privilege - controls access rights.
Bit 4 = 1	The third word is present and gives the bit number portion of the effective address.

SEGNO (Bits 5-16) The segment number portion of the effective address.
WORDNO (Bit 17-32) The word number portion of the effective address.
BITNO (Bits 33-48) The bit number portion of the effective address.

Stack segment header (VI)

Word	Contents
0 1	FREE POINTER
2 3	EXTENSION SEGMENT POINTER

Word	Meaning
0,1	Free pointer - segment number/word number of available location at which to build next frame. Must be even.
2,3	Extension segment pointer - segment number/word number of locations at which to build next frame when current segment overflows. If zero, a stack overflow fault occurs when current segment overflows.

PCL stack frame header (VI)

Word	Contents
0	0 - 0
1	STACK ROOT SEGMENT NUMBER
2 3	RETURN POINTER
4 5	CALLER'S SAVED STACK BASE REGISTER
6 7	CALLER'S SAVED LINK BASE REGISTER
8	CALLER'S SAVED KEYS
9	LOCATION FOLLOWING CALL

Word	Meaning
0	Flag bits - set to zero by PCL when frame is created
1	Stack root segment number - for locating free pointer
2,3	Return pointer - segment number/word number of return location
4,5	Caller's saved stack base register
6,7	Caller's saved link base register
8	Caller's saved keys
9	Word number of location following call - beginning of argument transfer templates, if any

CALF stack frame header (VI)

Word	Field
0	FLAG BITS
1	STACK ROOT SEGMENT NUMBER
2 3	RETURN POINTER
4 5	CALLER'S SAVED STACK BASE REGISTER
6 7	CALLER'S SAVED LINK BASE REGISTER
8	CALLER'S SAVED KEYS
9	LOCATION FOLLOWING CALL
10	FAULT CODE
11 12	FAULT ADDRESS
13 14 15	RESERVED

Word	Meaning
0	Flag bits – set to one by CALF fault
1	Stack root segment number – for locating free pointer
2,3	Return pointer – segment number/word number of return location
4,5	Caller's saved stack base register
6,7	Caller's saved link base register
8	Caller's saved keys
9	Word number of location following call - beginning of argument transfer templates, if any
10	Fault code
11,12	Fault address
13–15	Reserved

Entry control block (ECB) (VI)

Word	Field
0 1	POINTER TO CALLED PROCEDURE
2	STACK FRAME SIZE
3	STACK ROOT SEGMENT NUMBER
4	ARGUMENT LIST DISPLACEMENT
5	NUMBER OF ARGUMENTS
6 7	LINK BASE REGISTER OF CALLED PROCEDURE
8	KEYS
9 10 11 12 13 14 15	RESERVED

Word	Meaning
0,1	Pointer (ring, segment, word number) to the first executable instruction of the called procedure.
2	Stack frame size to create (in words). Must be even.
3	Stack root segment number. If zero, keep same stack.
4	Displacement in new frame of where to build argument list.
5	Number of arguments expected.
6,7	Called procedure's link base (location of called procedure's linkage frame less '400).
8	CPU keys desired by called procedure.
9–15	Reserved, must be zero.

Entry control blocks which are gates must begin on a 0 modulo 16 boundary, and must specify a new stack root.

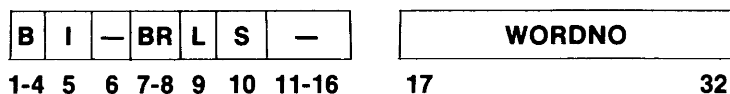
Queue control block (VI)

1	TOP POINTER		16
17	BOTTOM POINTER		32
V	000	HIGH ORDER ADDRESS	
33	34 36	37	48
49	SIZE MASK		64

Bits	Meaning
1–16	Top pointer-read
17–32	Bottom pointer-write
33 (V)	Virtual/physical control bit 0 physical queue 1 virtual queue
34–36	Reserved - must be zero
37–48	Queue data block address Segment number if virtual queue High order physical address bits if physical queue
49–64	Mask - value 2**K-1

Queue control blocks must start on word boundaries which are divisible by four, if used for DMQ. If not, a performance penalty is imposed, but the queue will work.

Argument transfer template (AP) (VI)



B (Bits 1–4)	Bit number
I (Bit 5)	Indirect bit
BR (Bits 7–8)	Base register 00 Procedure base (PB) 01 Stack base (SB) 10 Link base (LB)
L (Bit 9)	Last template for this call
S (Bit 10)	Store argument address. Last template for this argument.
WORDNO (Bits 17–32)	Word number offset from base register

PROCESSOR CHARACTERISTICS

Registers (S)

Prime 100, 200 and 300 registers are 16 bits wide. All the program visible registers are physically located in high speed memory and are addressed as memory locations 0-37. In restricted mode (normal user operation) *only* 0-7 are accessable.

Memory Address	Register Designation	Function
0	X	Index Register
1	A	Arithmetic Register
2	B	Extension Arithmetic Register
3		
4		
5		
6	VSC	Visible Shift Count
7	P	Program Counter
10	PMAR (Prime 300 only)	Page Map Address Register
11	FCODE	Fault Code
12	FAR	Fault Address Register
13-17	Reserved	
20-37	DMA '20, '22, . . . '36 (8 total)	Word Pairs for DMA channels (address and word counts)

Registers (R)

Prime 100, 200 and 300 registers are 16 bits wide. All the program visible registers are physically located in high speed memory and are addressed as memory locations 0-37. In restricted mode (normal user operation) *only* 0-7 are accessable.

Memory Address	Register Designation	Function
0	X	Index Register
1	A	Arithmetic Register
2	B	Extension Arithmetic Register
3	S	Stack Register
4	FLTH	Floating Point Accumulator - High
5	FLTL	Floating Point Accumulator - Low
6	FEXP	Floating Point Exponent
7	P	Program Counter
10	PMAR (Prime 300 only)	Page Map Address Register
11	FCODE	Fault code
12	PFAR	Page Fault Address Register
13-17	Reserved for microprogram	
20-37	DMA '20, '22, . . . '36 (8 total)	Word Pairs for DMA channels (address and word counts)

Registers (VI)

Prime 350 and above registers are 32 bits wide. Short form instructions reference the same registers as in R-mode.

Register addresses used in LDLR and STLR instructions are doubleword addresses. The notation "2 H" means the high, or left 16 bits of register address 2, while "2 L" means the low, or right 16 bits.

The following registers should *not* be written into by STLR instructions, or anomalous behavior will result.

PB The procedure base should be changed only via LPSW or programmed transfers of control.

keys The keys should be changed only via LPSW or the various mode control operations.

modals The modals should be changed only via LPSW or the various mode control operations. In no case should an LPSW ever attempt to change the current register set bits of the modals.

VI-mode register description

	Definitions
TR	Temporary Registers TR7 - Saved return pointer on a halt (automatic save)
RDMX	Register DMX RDMX1 - Used by DMC, buffer start pointer RDMX2 - REA at time of DMX trap RDMX3 - Save RD during DMQ RDMX4 - Used as working register
RATMPL	Read Address Trap Map to RP Low
RSGT	Register Segmentation Trap RSGT1 - SDW2 / address of Page Map RSGT2 - contents of Page Map / SDW2
REOIV	Register End of Instruction Vector
ZERO/ONE	Constants
PBSAVE	Procedure Base Save saved return pointer when return pointer used elsewhere
C377	Constant
PSWPB	Processor Status Word Procedure Base return pointer for interrupt return (also used for Prime 300 compatibility)
PSWKEYS	Processor Status Word Keys KEYS for interrupt return (also used for Prime 300 compatibility)
PPA	Pointer to Process A
PLA	Pointer to Level A
PCBA	Process Control Block A
PPB	Pointer to Process B
PLB	Pointer to Level B
PCBB	Process Control Block B
DSWRMA	Diagnostic Status Word RMA RMA at last Check Trap
DSWSTAT	Diagnostic Status Word Status

DSWPB Diagnostic Status Word Procedure Base
Return pointer or PBSAVE at last check
RSAVPTR Register Save Pointer
Location of Register Save Area after Halt
GR General Register
FAR0 Field Address Register 0
FLR0 Field Length Register 0
FAR1 Field Address Register 1
FLR1 Field Length Register 1
PB Procedure Base
PBH - RPH
PBL - 0
SB Stack Base
LB Link Base
XB Temporary (auxiliary) base
DTAR Descriptor Table address registers
KEYS See below
MODALS See below
OWNER Pointer to PCB of process owning this register set
FCODE Fault Code
FADDR Fault Address
TIMER 1-millisecond process timer (used for time-slice)

MICROCODE SCRATCH			DMX			CURRENT REGISTER SET (CRS)					
RS0			RS1			RS2	RS3	PRIME 300		PRIME 400	PRIME 500
ADR	HIGH	LOW	ADR	HIGH	LOW	ADR	ADR	HIGH	LOW		
0	TR0	—	40	—	—	100	140	—	—	—	GR0
1	TR1	—	41	—	—	101	141	—	—	—	GR1
2	TR2	—	42	—	—	102	142	1(A)	2(B)	L	GR2
3	TR3	—	43	—	—	103	143	—	—	E	GR3
4	TR4	—	44	—	—	104	144	—	—	—	GR4
5	TR5	—	45	—	—	105	145	3(S)	—	Y	GR5
6	TR6	—	46	—	—	106	146	—	—	—	GR6
7	TR7	—	47	—	—	107	147	0(X)	—	X	GR7
10	RDMX1	—	50	—	—	110	150	13	—	FALR0	FALR0 (FAC0)
11	RDMX2	—	51	—	—	111	151	—	—	FALR0	FALR0 (FAC0)
12	—	RATMPL	52	—	—	112	152	4(FAC)	5(FAC)	FALR1 (FAC)	FALR1 (FAC1)
13	RSGT1	—	53	—	—	113	153	6(FAC)	FAC	FALR1 (FAC)	FALR1 (FAC1)
14	RSGT2	—	54	—	—	114	154	—	—	PB	—
15	RECC1	—	55	—	—	115	155	14	15	SB	—
16	RECC2	—	56	—	—	116	156	16	17	LB	—
17	—	REOIV	57	—	—	117	157	—	—	XB	—
20	ZERO	ONE	60	(20)	(21)	120	160	10	—	DTAR3	—
21	PBSAVE	—	61	—	—	121	161	—	—	DTAR2	—
22	RDMX3	—	62	(22)	(23)	122	162	—	—	DTAR1	—
23	RDMX4	—	63	—	—	123	163	—	—	DTAR0	—
24	C377	C377	64	(24)	(25)	124	164	—	—	KEYS/MODALS	KEYS/MODALS
25	—	—	65	—	—	125	165	—	—	OWNER	—
26	—	—	66	(26)	(27)	126	166	11(FCODE)	—	FCODE	—
27	—	—	67	—	—	127	167	—	12(FADDR)	FADDR	—
30	PSWPB	—	70	(30)	(31)	130	170	—	—	TIMER	—
31	PSWKEYS	—	71	—	—	131	171	—	—	—	—
32	PPA:PLA	PCBA	72	(32)	(33)	132	172	—	—	—	—
33	PPB:PLB	PCBB	73	—	—	133	173	—	—	—	—
34	DSWRMA	—	74	(34)	(35)	134	174	—	—	—	—
35	DSWSTAT	—	75	—	—	135	175	—	—	—	—
36	DSWPB	—	76	(36)	(37)	136	176	—	—	—	—
37	RSAVPTR	—	77	—	—	137	177	—	—	—	—

General registers- 32 bits (I)

The eight general registers are numbered from 0–7. 1–7 may be used for index registers. All are used as fixed point and logical accumulators in register to memory and register to register operations.

Floating point register – single precision (RVI) (2 registers in I-mode)

Register			Contents
Prime 300	Prime 400	Prime 500	
'04	12H	10H	S (1) / MANTISSA (2–16)
'05	12L	10L	MANTISSA (17–32)
'06	13H	11H	EXPONENT (EXCESS 128) (33–48)

Floating point register - double precision (RVI) (2 registers in I-mode)

Register			Contents
Prime 300	Prime 400	Prime 500	
'04	12H	10H	S (1) | MANTISSA (2–16)
'05	12L	10L	MANTISSA (17–32)
'02	13L	11L	MANTISSA (33–48)
'06	13H	11H	EXPONENT (EXCESS 128) (49–64)

Floating point registers - 64 bits (I)

The two floating point registers are numbered 0 and 1. They are used as single and double precision accumulators in register to memory and register to register operations. The two floating point registers overlap the two field length address registers on the Prime 500 and care must be used in moving between floating point and field registers.

Base registers (VI -Mode)

The four base registers:

Procedure Base Register	PB
Stack Base Register	SB
Link Base Register	LB
Temporary Base Register	XB

have the following format:

0	RING	0	SEGNO	WORDNO
1	2–3	4	5–16	17–32

RING (Bits 2–3)	Ring Number
SEGNO (Bits 5–16)	Segment Number
WORDNO (Bits 17–32)	Word Number

Field address and length registers (VI)

There are two address registers and two length registers for the manipulation of variable length fields. They overlap the floating point accumulator.

0	RING	0	SEGNO	WORDNO	LENGTH	BITNO	0	LENGTH
1	2-3	4	5-16	17-32	33-48	49-52	53-59	60-64

RING (Bits 2-3)	Ring Number
SEGNO (Bits 5-16)	Segment Number
WORDNO (Bits 17-32)	Word Number
LENGTH (Bits 33-48, 60-64)	Length
BITNO (Bits 49-52)	Bit Number

The meaning of the value in the field length field depends on the data type being used. For a discussion of the available data types see the decimal and character instruction descriptions.

Keys (SR)

Process status information is available in a word called the keys, which can be read or set by the program. Its format is as follows:

C	DBL	—	Mode	0	Bits 9-16 of location 6
1	2	3	4-6	7-8	9 — 16

C (Bit 1)	Set by arithmetic error conditions
DBL (Bit 2)	0 - Single Precision, 1 - Double Precision.
MODE (Bits 4-6)	The current addressing mode as follows:

000	16S
001	32S
011	32R
010	64R
110	64V
100	32I

C-bit (SR): Bit 1 in the keys. Set by arithmetic error conditions and shifts (Bit 1).

Keys (VI)

Process status information is available in a 16-bit register known as the keys. It may be referenced by the LPSW, TKA, and TAK instructions.

C	0	L	MODE	F	X	LT	EQ	DEX	0	I	S
1	2	3	4-6	7	8	9	10	11	12 - 14	15	16

C (Bit 1)	C-Bit
L (Bit 3)	L-Bit
MODE (Bits 4-6)	Addressing Mode:

000	16S
001	32S
011	32R
010	64R
110	64V
100	32I

F (Bit 7)	Floating point exception disable: 0 take fault 1 set C-bit
X (Bit 8)	Integer Exception enable 0 set C-bit 1 take fault
LT (Bit 9) **EQ** (Bit 10)	Condition code bits: LT negative EQ positive
DEX (Bit 11)	Decimal exception enable 0 set C-bit 1 take fault
I (Bit 15)	In dispatcher - set/cleared only by process exchange
S (Bit 16)	Save done - set/cleared only by process exchange

C-bit (VI): Set by error conditions in arithmetic operations and by shifts.

L-bit (VI): Set by an arithmetic or shift operation except IRS, IRX, DRX. Equal to carry out of the most significant bit (bit 1) of an arithmetic operation. It is valuable for simulating multiple - precision operations and for performing unsigned comparisons following a CAS or a SUB.

Condition code bits (VI): The two condition-code bits are designated "EQ" and "LT". EQ is set if and only if the result is zero; if overflow occurs, EQ reflects the state of the result after truncation rather than before. LT reflects the extended sign of the result (before truncation, if overflow), and is set if the result is negative.

Modals (VI)

Processor status is available in another 16-bit register known as the "modals".

E	V	0	CURREG	MIO	P	S	MCK
1	2	3-8	9-11	12	13	14	15-16

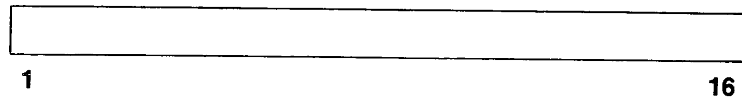
E (Bit 1)	Interrupts enabled
V (Bit 2)	Vectored-interrupt mode
CURREG (Bits 9-11)	Current register set (set/cleared only by process exchange)
MIO (Bit 12)	Mapped I/O mode
P (Bit 13)	Process-exchange mode
S (Bit 14)	Segmentation mode
MCK (Bits 15-16)	Machine-check mode

Note

Never attempt to write into the keys or the modals with the STLR instruction. The only valid way to change either the keys or the modals is to use the LPSW instruction, the keys operations OTK and TAK, or the various special-case instructions designed to manipulate specific bits of the status. Furthermore, even LPSW should not be used to alter the in-dispatcher and save-done bits of the keys or the register-set bits of the modals.

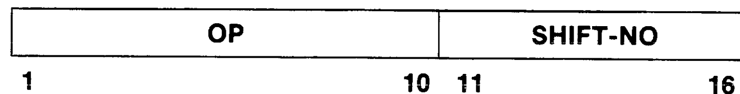
INSTRUCTION FORMATS

GENERIC (SRVI)



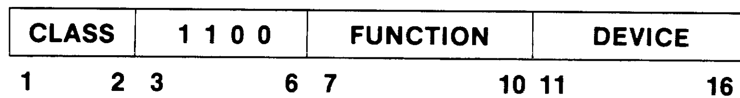
The entire instruction word is an opcode. Bits 3-6 are always zero

SHIFT (SR)



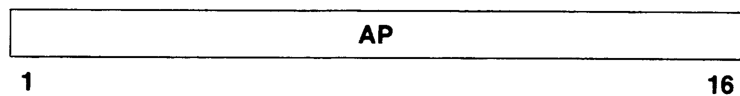
OP (Bits 1-10)	Opcode - Bits 3-6 are always zero
SHIFT-NO (Bits 11-16)	Two's complement of the number of places to be shifted. Zero means shift 63 places

I/O (SR)



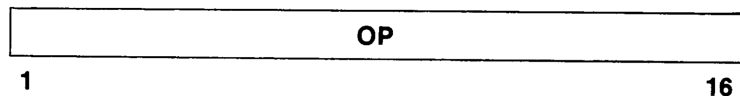
CLASS (Bits 1-2)	Type of I/O instruction 00 Control 01 Sense 10 Input 11 Output
Bits 3-6	1100
FUNCTION (Bits 7-10)	Subdivision of class. Device dependent
DEVICE (Bits 11-16)	Device type

DECIMAL (VI)



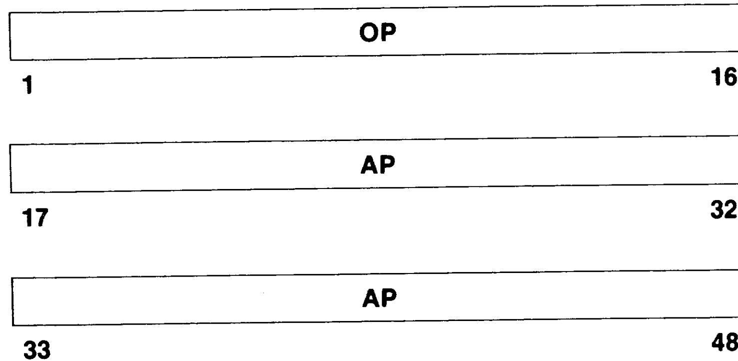
OP (Bits 1-16)	Opcode. This instruction uses previously set up field registers and a previously set up control word in register L (general register 2 in I-Mode). See decimal control word in Data Structures.

CHARACTER (VI)



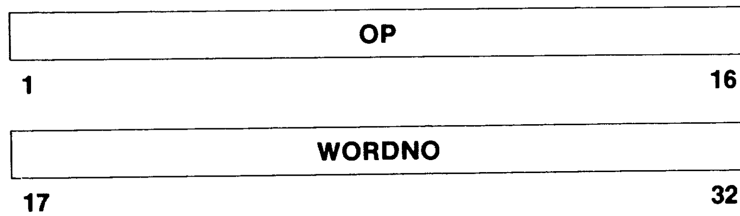
OP (Bits 1-16)	Opcode. This instruction uses previously set up field registers.

GENERIC AP (VI)



OP (Bits 1–16) Opcode
AP Bits (17–48) Address Pointer - see AP in Data Structures.

BRANCH (V)



OP (Bits 1–16) Opcode
WORDNO (Bits 17–32) Word number offset from procedure base register.

Memory reference instruction format (SRV)

See Effective Addressing Formation in Section 10 – Memory Reference Concepts.

INSTRUCTION FORMATS — I-MODE

Purpose of I-Mode

The I-Mode instruction formats provide a 32-bit general register environment, particularly useful for:

- Heavy floating point calculations.
- Heavy long integer calculations.
- Extensive complex computations with intermediate results.

Features

- V-mode data types are a subset of I-mode data types, so no conversion is needed.
- The user visible V-mode register set is a subset of the I mode registers, so data can be passed in a common subset.
- The procedure call instruction automatically switches the addressing mode on a subroutine basis so the programmer can organize programs to use the best of V or I mode.
- The generic format instructions have the same opcode and same function in V and I mode.
- The memory reference format permits convenient specification of target registers in addition to the base, index, and displacement fields.

- All forms of indexing and indirection are supported.
- The same memory reference instruction can include register to register, register to memory and immediate data forms-special instructions are not required.
- The 16-bit format (register and non register generic) is included for additional efficiency. In addition, the register to register and floating register-source addressing formats do not use the second 16-bit part (bits 17-32) of the instruction word.

FORMATS

Non-register generic

These instructions are a subset of the V-mode generics and are processed the same way.

Register generic

These instructions operate on the specified register, which may be general, field, or floating register. This class includes the branch instructions, where the branch address, in the second word, is a 16-bit procedure base displacement.

Memory reference

There are three types of memory reference instructions:

MRNR-memory reference non register:

OP	R	AD	S	B	D
1-6	7-9	10-11	12-14	15-16	17-32

Data types — Integer, unsigned and logical
Location of 2nd operand — Memory

MRGR-memory reference general register:

OP	110	OP	AD	S	B	D
1-3	4-6	7-9	10-11	12-14	15-16	17-32

Data types — Integer, unsigned and logical
Location of 2nd operand — Immediate, register memory

MRFR-memory reference floating register:

OP	110	OP	FR	OP	AD	S	B	D
1-3	4-6	7	8	9	10-11	12-14	15-16	17-32

Data type — Floating point
Location of 2nd operand — Immediate, register and memory

Index registers: General registers 1 to 7 may be used as index register; 0 means no indexing.

Register to register: No indexing or indirection may be specified and the address field insert may be an absolute value:

1. 0 or 1 if the instruction format is MRFR, or
2. 0 – 7 if the instruction format is MRGR

Table 9-1. Address Formation Special Case Selection

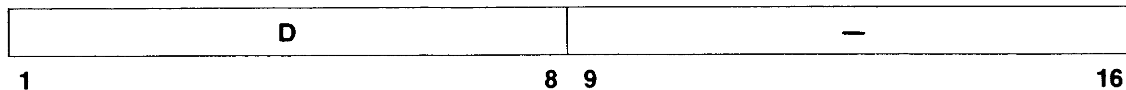
AD	S	B	Effective Address/Instruction Type
3	>0	—	(D+B)*+S *(indirect, post-index)*
3	0	—	(D+B)* *(indirect)*
2	>0	—	(D+B+S)* *(pre-index, indirect)*
2	0	—	(D+B)**(indirect)*
1	>0	—	D+B+S *(indexed)*
1	0	—	D+B *(direct)*
0	⩾0	0	REG-REG (S specifies source register)
0	0	1	Immediate Type 1
0	>0	1	Immediate Type 2
0	0	2	Immediate Type 3
0	1	2	Floating Reg Source (FRO)
0	2	2	Undefined *(will not generate UII)*
0	3	2	Floating Reg source (FR1)
0	4-7	2	Undefined *(will not generated UII)*
0	—	3	Undefined *(will not generate UII)*

Field Mnemonics:

OP	Opcode
R	Destination register
AD	Address computation code
S	Source register
B	Base register
FR	Floating register

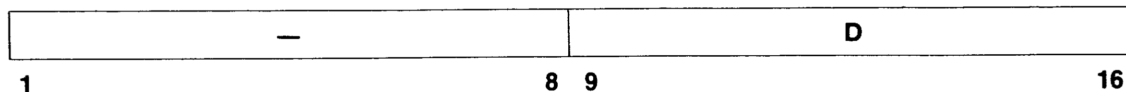
Immediate: There are three immediate data formats:

Immediate type 1



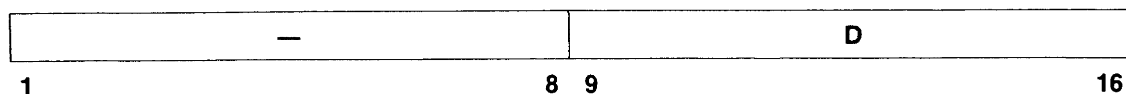
Require a 16-bit literal (no L suffix)

Immediate type 2



Sign extend full word general register instruction. Requires a 32-bit literal (with L suffix). Bit 17, the low order sign bit, is extended through the high order 16 bits.

Immediate type 3



Floating point register instruction (both single and double precision) requires floating point literal. The fractional part is truncated to eight bits, stored in the instruction.

10
Memory reference concepts - SRV

BACKGROUND CONCEPTS

Memory is addressed as a set of continuous word locations. The number of words that can be addressed by an instruction, and the way in which the address is calculated depends on the current addressing mode of the machine and the location of the address relative to the instruction.

In turn, the addressing modes of the machine differ in the size of the instruction word, the number of bits allotted to the provisional address displacement, and the number and meaning of the bits allotted to the operation code.

To reduce the number of memory references, designers wish to do as much as possible in one word. For example, in the S and R addressing modes, a one word memory reference instruction has nine bits (512 words) of direct addressability, four bits for operation codes, one bit for indirection, one bit for indexing, and one bit to control out-of-range addresses.

Within each addressing mode, there are the following tradeoffs:

1. Size of program address space
2. Levels of indirection
3. Levels of indexing
4. Whether indexing is performed before or after indirection
5. Number of operation codes available

Through the discussion of the S, R, and V addressing modes, we shall show how these variables are defined.

Memory organization

Sectors: (S-Mode and R-Mode when S=0). A sector is a contiguous group of 512 words. S-Mode memory reference instructions have nine bits (D field) of addressability to any location in a sector and one bit, the S-bit, to specify Sector 0 (S=0) or the current sector (S=1). D and S together give 10 bits, or 1024 words, of direct addressability.

Relative reach: (R-Mode and V-Mode when S=1). When S=1 the D field is interpreted as a signed number in the range −255 to +256. When $D \leqslant 240$ (R-Mode), or $D \leqslant 224$ (V-Mode), the number is treated as a code, not as a displacement. When $-240 < D < 256$ (R-Mode) or $-224 < D < 256$ (V-Mode), the address is relative to the program counter.

Segmentation: (V-Mode and I-Mode). See the System Architecture Reference Guide for a discussion of segmentation.

Effective address formation

Each memory reference instruction calculates an effective address. This calculation and its results vary depending on addressing mode and instruction format; variables include pre- and post-indexing, indirection, and base registers. For maximum clarity, we discuss the

classes by format types *and* present addressing mode flowcharts. Both the format discussions and the addressing mode flowcharts are cross referenced to each other. Table 10-1 summarizes the format classes and gives the addressing modes where they are used.

Indexing: In general, if the X-bit of the instruction is set, the contents of the index register are added to the D-field. If the indirect bit is set, the address mode and D-field determine whether indexing occurs before or after indirection. The result is truncated to the number of bits permitted by the addressing modes, and the high order bits are cleared. In V-Mode, there are two index registers, X and Y. The displacement field determines which to use and how to use it.

Note

The index register may be preset by the program to any value between -32768 and +32767.

Indirection: In general, if the I-bit is set, the D-field plus index, if any, is an intermediate address. The indirect address word at that location may, depending on the address mode, also contain X and I bits. The specific addressing mode discussion gives the details.

Address truncation (S R): After effective address formation is complete, the resulting address is truncated to the number of bits appropriate to the addressing mode in effect:

Mode	Addressing Bits	Size of Addressable Memory
16S	14	16K
32S	15	32K
32R	15	32K
64R	16	64K

Since the higher order bits of the address are zeroes, an address *cannot* be formed that addresses a memory location beyond the range of the current addressing mode. However, it is possible for an executing program to increment the program counter out of the current range (instead of overflowing to zero).

Table 10-1. Memory Reference Instruction Format

Type	No. Words	S[1]	D[2]	CB[3]	Mode
Basic	1	0	0 - '777	—	SR
Sector Relative	1	1	0 - '777	—	S
Procedure Relative	1	1	− 241 to + 255	—	R
			− 224 to + 255	—	V
Stack Postincrement/ Predecrement	1	1	− 256 to − 241	2,3	R
Base Register Relative	1	0	0 - '777	—	V
Long Reach	2	1	− 256 to − 241	0,2	R
Stack Relative	2	1	− 256 to − 241	1,3	R
Base Registers	2	1	− 256 to − 224	—	V

1. **Sector bit (S)**. Bit 7 in both one- and two-word memory reference instructions. The meaning varies, depending on the addressing mode, but in general is used to control out-of-range addresses.
2. **Displacement field (D)**. Bits 8-16 in the instruction word. Bit 8 is sign bit except in Basic, Sector Relative, and Base Register types of instruction.
3. **Class bits (CB)**. Bits 15 and 16 of the R mode two-word instructions distinguish between Long Reach and Stack Relative instruction types.

Instruction range

The range that an instruction can *directly* address is called its addressing range. The assembler and the loader analyze the assembler statement and set up both in-range and out-of-range addresses. In the discussion below we shall examine the sectored and relative address ranges and how they are set up prior to execution. The segmentation concepts and address ranges are discussed in the System Architecture Reference Guide.

Sectored: In S-mode, the memory reference instructions can address any location in sector 0 or in the sector of the instruction. When S=1, the nine bit displacement field is a location in the current sector. When S=0, the nine bit displacement is in sector 0.

The software uses the S-bit to control out-of-range addresses in the following manner: the assembler does a preliminary analysis of the relation of the displacement field (expression or symbol) to the instruction location, and passes this information to the loader, which sets up the final instruction for execution. The loader puts the object code received from the assembler together with any other required routines (such as subroutines), resolves external linkages and sets up sector 0, the communication and linkage area.

Sector 0 can also be directly addressed by the program, a useful feature for handling common data fields.

Relative: In R-mode when S=1, the D field is interpreted as a signed number in the range -226 to +255. When the two high order bits are one (D ⩽ 240) the number is treated as a code, not as a displacement. When -240<D<255 the address is relative to the program counter.

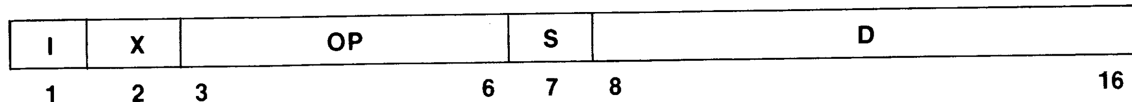
The loader analyzes the displacement field and if the effective address will be out of relative range (-240 to +255) sets S=0, I=1, and the displacement field to point to the address word in sector zero. Thus, in 64R, if the address is out of range, no indirection is possible because the loader uses the instruction word indirect bit.

Assembler Notation	Location of ADDR: Sector 0	Same Sector	Other
LDA ADDR	S=0 I=0 D=location in sector 0.	S=1 I=0 D=displacement in same sector.	S=0 I=1 D=first available link in sector 0. At that location an indirect word is constructed with I=0, pointing at ADDR with a full 14 (16S) or 15 (32S, 32R) or 16 (64R) bit indirect address.
LDA ADDR,*	S=0 I=1 D=Location in sector 0 which contains a pointer defined by the program	S=1 I=1 D=Location in same sector. It *must* contain pointer defined by program.	S=0 I=1 D=first available link in sector 0. At that location an indirect word is constructed with I=1 and a full 14 (16S) or 15 (32S, 32SR) bit indirectpointer to ADDR. *Not permitted in 64R.*

MEMORY REFERENCE INSTRUCTION FORMATS

BASIC (one word, S-bit=0)

16S, 32S, 32R, 64R



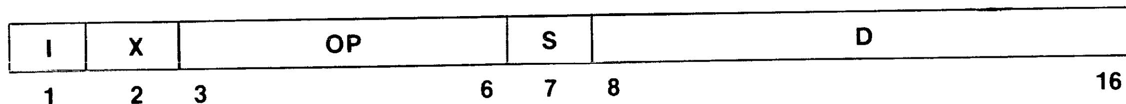
I (Bit 1)	Indirect Bit
X (Bit 2)	Index Bit
OP (Bits 3-6)	Opcode
S (Bit 7)	Sector Bit = 0
D (Bits 8-16)	Displacement in sector 0

The D-field is a displacement in sector 0. The effective address is equal to bits 8–16 of the instruction, with bits 0–7 equal to zero. Indexing and indirection are a function of the I and X bits and the addressing mode.

Addressing Mode	I	X	S	D	EA	Type
16S	0	0	0	0 to '777	0|D	*Direct*
	0	1	0	0 to '777	0|D+X	*Indexed*
	1	0	0	0 to '777	I(0|D)	*Indirect*
	1	1	0	0 to '777	I(0|D+X)	*Indirect, preindexed*
32S, 32R, 64R	0	0	0	0 to '777	0|D	*Direct*
	0	1	0	0 to '777	0|D+X	*Indexed*
	1	0	0	0 to '777	I(0|D)	*Indirect*
	1	1	0	0 to '77	I(0|D+X)	*Indirect, preindexed*
	1	1	0	'100 to '777	I(0|D)+X	*Indirect, postindexed*

SECTOR RELATIVE (One word, S-bit=1)

16S, 32S



I (Bit 1)	Indirect Bit
X (Bit 2)	Index Bit
OP (Bits 3-6)	Opcode
S (Bit 7)	Sector Bit = 1
D (Bits 8-16)	Displacement within current sector

The D-field is a displacement in the current sector. The effective address is formed by concatinating the D-field bits with the higher order bits of the program counter (P). Indexing and indirection are a function of the I and X bits and the addressing mode. Bits 1 and 2 (16S) or 1 (32S) of the final effective address are cleared. In effect, the program counter gives the sector number and the D-field, the location within the sector.

Addressing Mode	I	X	S	D	EA	Type
16S	0	0	1	0 to '777	P|D	*Direct*
	0	1	1	0 to '777	P|D+X	*Indexed*
	1	0	1	0 to '777	I(P|D)	*Indirect*
	1	1	1	0 to '777	I(P|D+X)	*Indirect, preindexed*
32S	0	0	1	0 to '777	P|D	*Direct*
	0	1	1	0 to '777	P|D+X	*Indexed*
	1	0	1	0 to '777	I(P|D)	*Indirect*
	1	1	1	0 to '777	I(P|D)+X	*Indirect, postindexed*

PROCEDURE RELATIVE (One word, S-bit=1) 32R, 64R, 64V

I	X	OP	S	D
1	2	3 – 6	7	8 – 16

I (Bit 1) Indirect Bit
X (Bit 2) Index Bit
OP (Bits 3-6) Opcode
S (Bit 7) Sector Bit=1
D (Bits 8-16) Location relative to the program counter
64V= – 224 to +255
64R= – 240 to +255

Addressing is relative to the current program counter value, which is the current instruction location plus 1. The effective address is formed by adding the value of the D-field to the updated program counter value (P). Indirection and indexing are a function of the I and X bits and the addressing mode.

Addressing Mode	I	X	S	D	EA	Type
32R, 64R	0	0	1	– 240 to +255	P+D	*Direct*
	0	1	1	– 240 to +255	P+D+X	*Indexed*
	1	0	1	– 240 to +255	I(P+D)	*Indirect*
	1	1	1	– 240 to +255	I(P+D)+X	*Indirect, postindexed*
64V	0	0	1	– 224 to +255	P+D	*Direct*
	0	1	1	– 224 to +255	P+D+X	*Indexed*
	1	0	1	– 224 to +255	I(P+D)	*Indirect*
	1	1	1	– 224 to +255	I(P+D)+X	*Indirect, postindexed*

STACK PREDECREMENT, POSTINCREMENT (One word, S-bit=1) 32R, 64R

I	X	OP	11000	XX	CB
1	2	3 … 6	7 … 12	13 14	15 16

I (Bit 1)	Indirect Bit
X (Bit 2)	Index Bit
OP (Bits 3-6)	Opcode
Bits 7–12	110000
XX (Bits 13–14)	Opcode extension
CB (Bits 15–16)	Class Bits

These classes use the stack pointer (SP) as the address displacement, and perform an auxiliary postincrement or predecrement of the pointer. Instructions using these address methods are always one-word instructions.

Addressing Mode	I	X	S	CB	EA	Type
32R, 64R	0	0	1	2	SP	*Postincrement*
	0	1	1	2	I(SP)+X	*Postincrement, indirect, post-indexed*
	1	0	1	2	I(SP)	*Postincrement, indirect*
	0	0	1	3	SP-1	*Predecrement*
	0	1	1	3	I(SP-1)+X	*Predecrement, indirect, post-indexed*
	1	0	1	3	I(SP-1)	*Predecrement, indirect*

Note

If a fault occurs during the execution of these classes, anomalous behavior can result.

BASE REGISTER RELATIVE (One word, S-bit=0) 64V

I	X	OP	S	D
1	2	3 … 6	7	8 … 16

I (Bit 1)	Indirect Bit
X (Bit 2)	Index Bit
OP (Bits 3-6)	Opcode
S (Bit 7)	Sector Bit=0
D (Bits 8-16)	Location relative to selected base register

This format provides 64V with one word based memory reference instructions, using the D-field to encode both base and displacement.

All indirection will be through 16-bit pointers in the procedure segment and the final effective address of indirect instructions will be in the procedure segment.

The effective address calculation is:

I	X	S	D	Address	Type
0	0	0	0–'7	register location	*Direct*
			'10–'377	SB+D	
			'400–'777	LB+D	
0	1	0	0–'377	if D+X<'10 then EA=register location else SB+D+X	*Indexed*
			'400–'777	LB+D+X	
1	0	0	0–'7	I(REG)	*Indirect*
			'10–'777	I(PB|D)	
1	1	0	0–'77	I[PB|D+X]	*Indirect preindexed*
1	1	0	'100–'777	I[PB|D]+X	*Indirect postindexed*

PB	Procedure base register
LB	Link base register
SB	Stack base register
X	Index register
D	Displacement field
REG	R-Mode registers, i.e., A,B,X, etc.

LONG REACH (Two word, S-bit=1) 32R, 64R

I	X	OP	11000	XX	CB
1	2	3 6	7 12	13 14	15 16

A
17 32

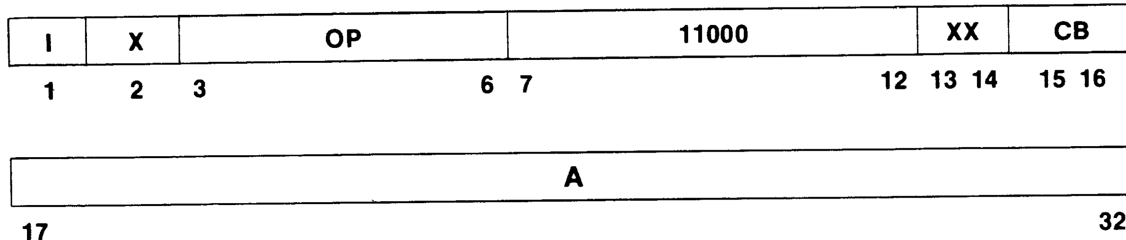
I (Bit 1)	Indirect Bit
X (Bit 2)	Index Bit
OP (Bits 3–6)	Opcode
Bits 7–12	110000
XX (Bits 13–14)	Opcode extension
CB (Bits 15–16)	Class Bits
A (Bits 17–32)	Address word

The 16-bit address word in the location following the instruction plus the I and X bits in the instruction combine in effective address calculation. The direct instruction reach is extended to 32K words (32R) or 64K words (64R), since the address is in the word following the instruction. In 32R, bit 1 is zero. In 64R, all 16 bits are used.

Addressing Mode	I	X	S	CB	EA	Type
32R, 64R	0	0	1	0	A	*Direct*
	0	1	1	0	A+X	*Indexed*
	1	0	1	0	I(A)	*Indirect*
	1	1	1	0	I(A+X)	*Indirect, preindexed*
	1	1	1	2	I(A)+X	*Indirect, postindexed*

STACK RELATIVE (Two Word, S-bit = 1)

32R,64R



I (Bit 1)	Indirect Bit
X (Bit 2)	Index Bit
OP (Bits 3-6)	Opcode
Bits 7-12	110000
XX (Bits 13-14)	Opcode extension
CB (Bits 15-16)	Class Bits
A (Bits 17-32)	Address word

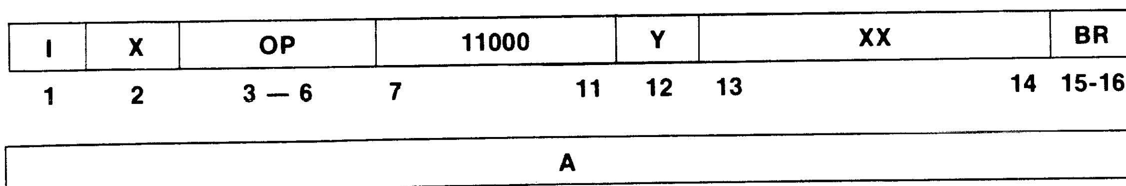
This class is identical to two-word long reach except that the contents of the stack pointer (SP) are added to the address word following the instruction word during the initial effective address calculation.

Indexing and indirection take place under control of the I and X bits and the addressing mode.

Addressing Mode	I	X	S	CB	EA	Type
32R, 64R	0	0	1	1	A+SP	*Direct*
	0	1	1	1	A+SP+X	*Indexed*
	1	0	1	1	I(A+SP)	*Indirect*
	1	1	1	1	I(A+SP+X)	*Indirect, preindexed*
	1	1	1	3	I(A+SP)+X	*Indirect, postindexed*

TWO WORD MEMORY REFERENCE

64V



I (Bit 1)	Indirect bit
X (Bit 2)	X bit
OP (Bit 3-6)	Opcode
Bits 7-12	110000
Y (Bit 12)	Y bit
XX (Bits 13-14)	Opcode extension
BR (Bits 15-16)	Base register: 00=PB, 01=SB, 10=LB, 11=XB
A (Bits 17-32)	16-bit word displacement relative to the base selected by the BR bits

I, X, Y and BR combine to give all 32 possible address combinations:

- Direct
- Indexed by X
- Indexed by Y
- Indirect
- Preindexed by X
- Preindexed by Y
- Postindexed by X
- Postindexed by Y

All indirect words are either 32 or 48 bit format and the final effective address is *always* a memory address (*never* a register). Table 10-3 shows all possible combinations.

Table 10-2. V-Mode Two Word Memory Reference

I	X	Y	BR	Effective Address	Meaning
0	0	0	0	PB|D	
			1	SB+D	*Direct*
			2	LB+D	
			3	XB+D	
0	0	1	0	PB|D+Y	
			1	SB+D+Y	*Indexed by Y*
			2	LB+D+Y	
			3	XB+D+Y	
0	1	0	0	PB|D+X	
			1	SB+D+X	*Indexed by X*
			2	LB+D+X	
			3	XB+D+X	
0	1	1	0	I(PB|D)	
			1	I(SB+D)	*Indirect*
			2	I(LB+D)	
			3	I(XB+D)	
1	0	0	0	I(PB|D+Y)	
			1	I(SB+D+Y)	*Pre-indexed by Y*
			2	I(LB+D+Y)	
			3	I(XB+D+Y)	
1	0	1	0	I(PB|D)+Y	
			1	I(SB+D)+Y	*Post-indexed by Y*
			2	I(LB+D)+Y	
			3	I(XB+D)+Y	
1	1	0	0	I(PB|D+X)	
			1	I(SB+D+X)	*Pre-indexed by X*
			2	I(LB+D+X)	
			3	I(XB+D+X)	
1	1	1	0	I(PB|D)+X	
			1	I(SB+D)+X	*Post-indexed by X*
			2	I(LB+D)+X	
			3	I(XB+D)+X	

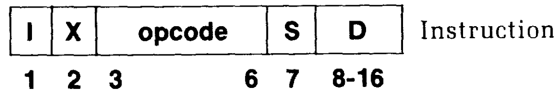
LDX and STX instructions may *only* be direct or indirect.

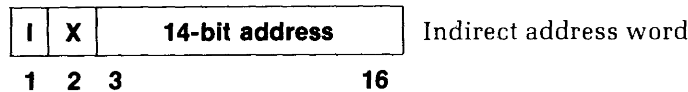
ADDRESSING MODE SUMMARIES AND FLOW CHARTS

16S summary

Address length: 14 bits; 16K word address space

Format:





Indexing: Multiple levels. In an indirect word, the index calculation is done before the indirection.

Indirection: Multiple levels.

I	X	S	D	EA	Assembler Notation	Type
0	0	0	0 to '777	0|D	LDA ADDR	*Direct*
0	1	0	0 to '777	0|D+X	LDA ADDR,1	*Indexed*
1	0	0	0 to '777	I(0|D)	LDA ADDR,*	*Indirect*
1	1	0	0 to '777	I(0|D+X)	LDA ADDR,1*	*Indirect, preindexed*
0	0	1	0 to '777	P|D	LDA ADDR	*Direct*
0	1	1	0 to '777	P|D+X	LDA ADDR,1	*Indexed*
1	0	1	0 to '777	I(P|D)	LDA ADDR,*	*Indirect*
1	1	1	0 to '777	I(P|D+X)	LDA ADDR,1*	*Indirect, preindexed*

P	Contents of program counter prior to instruction fetch (pointing at instruction).
0|D	Displacement into sector 0. Sector bits of effective address (bits 3-8) are zero.
P|D	Displacement in current sector formed by concatenation of sector bits from program counter with displacement field in instruction word.
X	Contents of index register.
I(expression)	Treat the effective address as indirect address.
ADDR	Location addressed by the LDA.

Note

If D is 0–'7 and S=0, the effective address is a register.

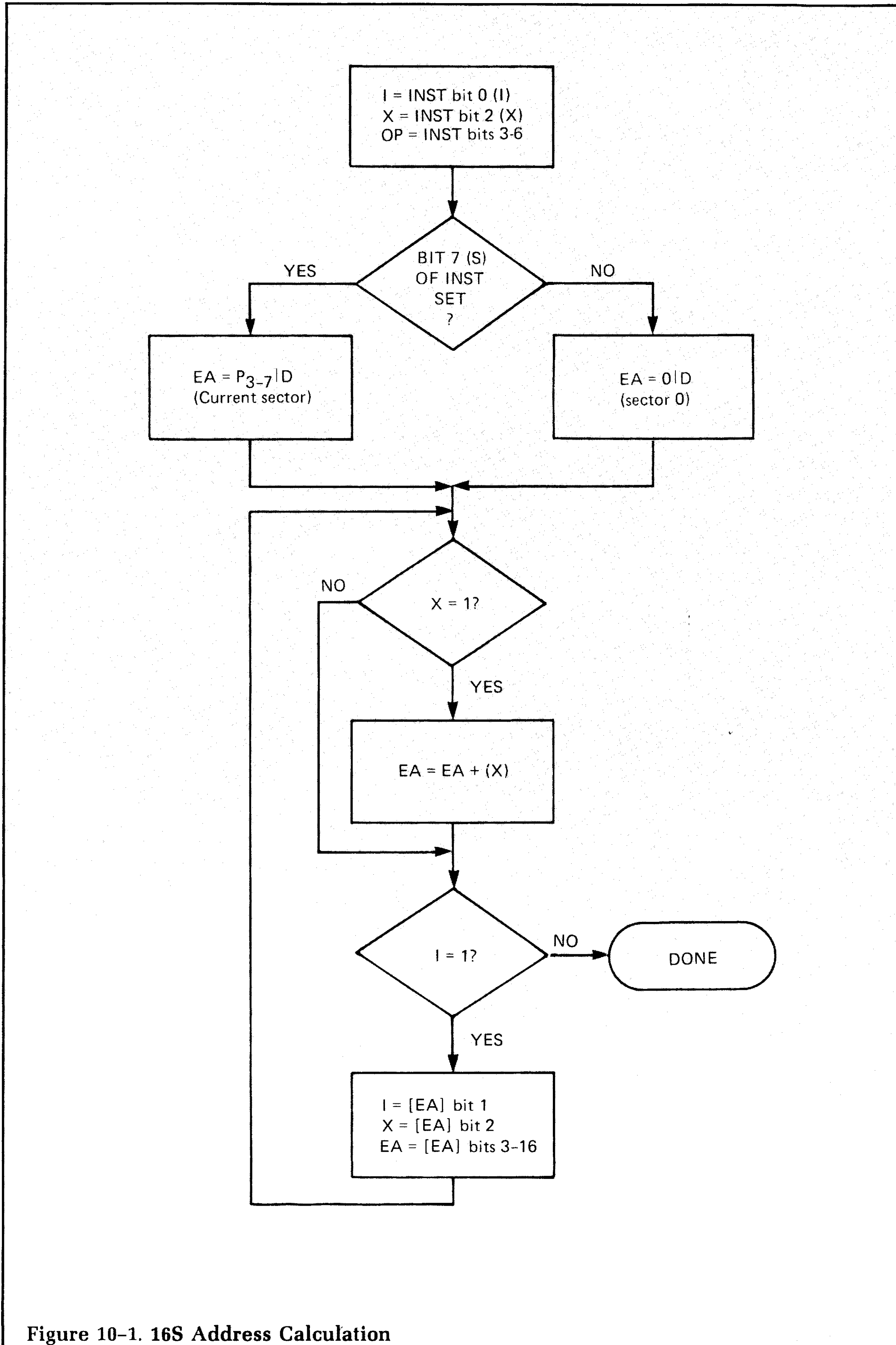
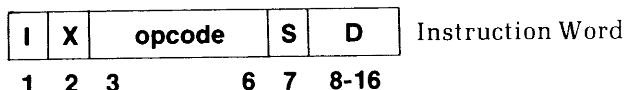


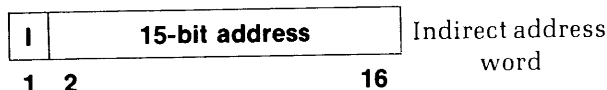
Figure 10-1. 16S Address Calculation

32S (Includes 32R when S=0) summary

Address length: 15 bits, 32K word address space

Format:





Indexing: One level. The 15-bit indirect address word eliminates the X bit. Done after all indirection is complete, except for the special case shown in the table below.

Indirection: Multiple levels.

I	X	S	D	EA	Assembler Notation	Type
0	0	0	0 to '777	0|D	LDA ADDR	*Direct*
0	1	0	0 to '777	0|D+X	LDA ADDR,1	*Indexed*
1	0	0	0 to '777	I(0|D)	LDA ADDR,*	*Indirect*
1	1	0	0 to '77	I(0|D+X)	LDA ADDR,1*	*Indirect, preindexed*
1	1	0	100 to '777	I(0|D)+X	LDA ADDR,*1	*Indirect postindexed*
0	0	1	0 to '777	P|D	LDA ADDR	*Direct*
0	1	1	0 to '777	P|D+X	LDA ADDR,1	*Indexed*
1	0	1	0 to '777	I(P|D)	LDA ADDR,*	*Indirect*
1	1	1	0 to '777	I(P|D)+X	LDA ADDR,1*	*Indirect postindexed*

P	Contents of program counter prior to instruction fetch (pointing at instruction).
0|D	Displacement into sector 0. Sector bits of effective address (bits 3-8) are zero.
P|D	Displacement in current sector formed by concatenation of sector bits from program counter with displacement field in instruction word.
X	Contents of index register.
I(expression)	Treat the effective address as indirect address.
ADDR	Location addressed by the LDA.

Note

If D is 0–'7 and S=0, the effective address is a register.

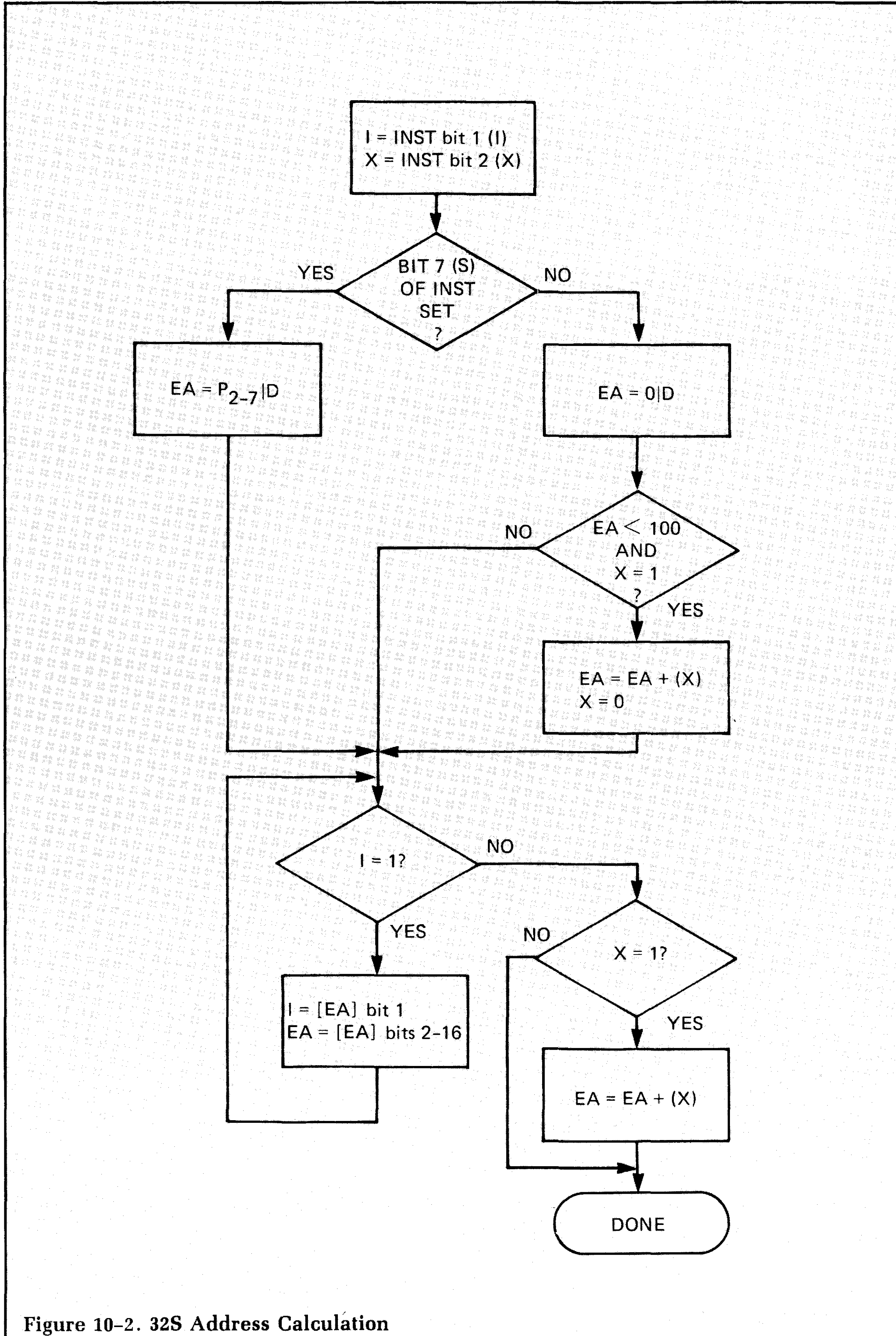
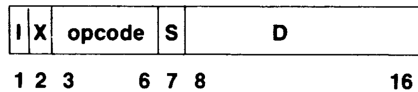


Figure 10-2. 32S Address Calculation

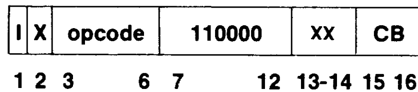
32R summary

Address length: 15 bits; 32K word address space

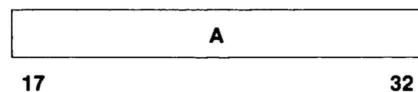
Format:



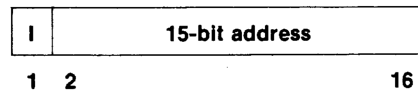
Instruction Word:
S=0 or S=1
D≥-240



Instruction Word:
S=1
D<-240



Address Word:
Long Reach and
Stack Relative



Indirect Address
Word

Indexing: One level.

Indirection: Multiple levels.

I	X	S	CB	D	EA	Assembler Notation	Type
0	0	0	—	0 to '777	0|D	LDA ADDR	*Direct*
0	1	0	—	0 to '777	0|D+X	LDA ADDR,1	*Indexed*
1	0	0	—	0 to '777	I(0|D)	LDA ADDR,*	*Indirect*
1	1	0	—	0 to '77	I(0|D+X)	LDA ADDR,1*	*Indirect, preindexed*
1	1	0	—	'100 to '777	I(0|D)+X	LDA ADDR,*1	*Indirect, postindexed*
0	0	1	—	-240 to +255	P+D	LDA ADDR	*Direct*
0	1	1	—	-240 to +255	P+D+X	LDA ADDR,1	*Indexed*
1	0	1	—	-240 to +255	I(P+D)	LDA ADDR,*	*Indirect*
1	1	1	—	-240 to +255	I(P+D)+X	LDA ADDR,*1	*Indirect, postindexed*
0	0	1	2	—	SP	LDA @+	*Postincrement*
0	1	1	2	—	I(SP)+X	LDA @+,*1	*Postincrement, indirect, postindexed*
1	0	1	2	—	I(SP)	LDA @+,*	*Postincrement, indirect*
0	0	1	3	—	SP-1	LDA -@	*Predecrement*
0	1	1	3	—	I(SP-1)+X	LDA -@,*1	*Predecrement, indirect, postindexed*
1	0	1	3	—	I(SP-1)	LDA -@,*	*Predecrement, indirect*

0	0	1	0	—	A	LDA% ADDR	*Direct, long reach*
0	1	1	0	—	A+X	LDA% ADDR,X	*Indexed, long reach*
1	0	1	0	—	I(A)	LDA% ADDR,*	*Indirect, long reach*
1	1	1	2	—	I(A+X)	LDA% ADDR,X*	*Indirect, preindexed long reach*
1	1	1	2	—	I(A)+X	LDA% ADDR,*X	*Indirect, postindexed long reach*
0	0	1	1	—	A+SP	LDA @+ADDR	*Direct, stack relative*
0	1	1	1	—	A+SP+X	LDA @+ADDR,X	*Indexed, stack relative*
1	0	1	1	—	I(A+SP)	LDA @+ADDR,*	*Indirect, stack relative*
1	1	1	1	—	I(A+SP+X)	LDA @+ADDR,X*	*Indirect, preindexed, stack relative*
1	1	1	3	—	I(A+SP)+X	LDA @+ADDR,*X	*Indirect, postindexed, relative*

P	Contents of program counter prior to instruction fetch (pointing at instruction).
0|D	Displacement into sector 0. Sector bits of effective address (bits 3-8) are zero.
X	Contents of index register.
I(expression)	Treat the effective address as indirect address.
ADDR	Location addressed by the LDA.

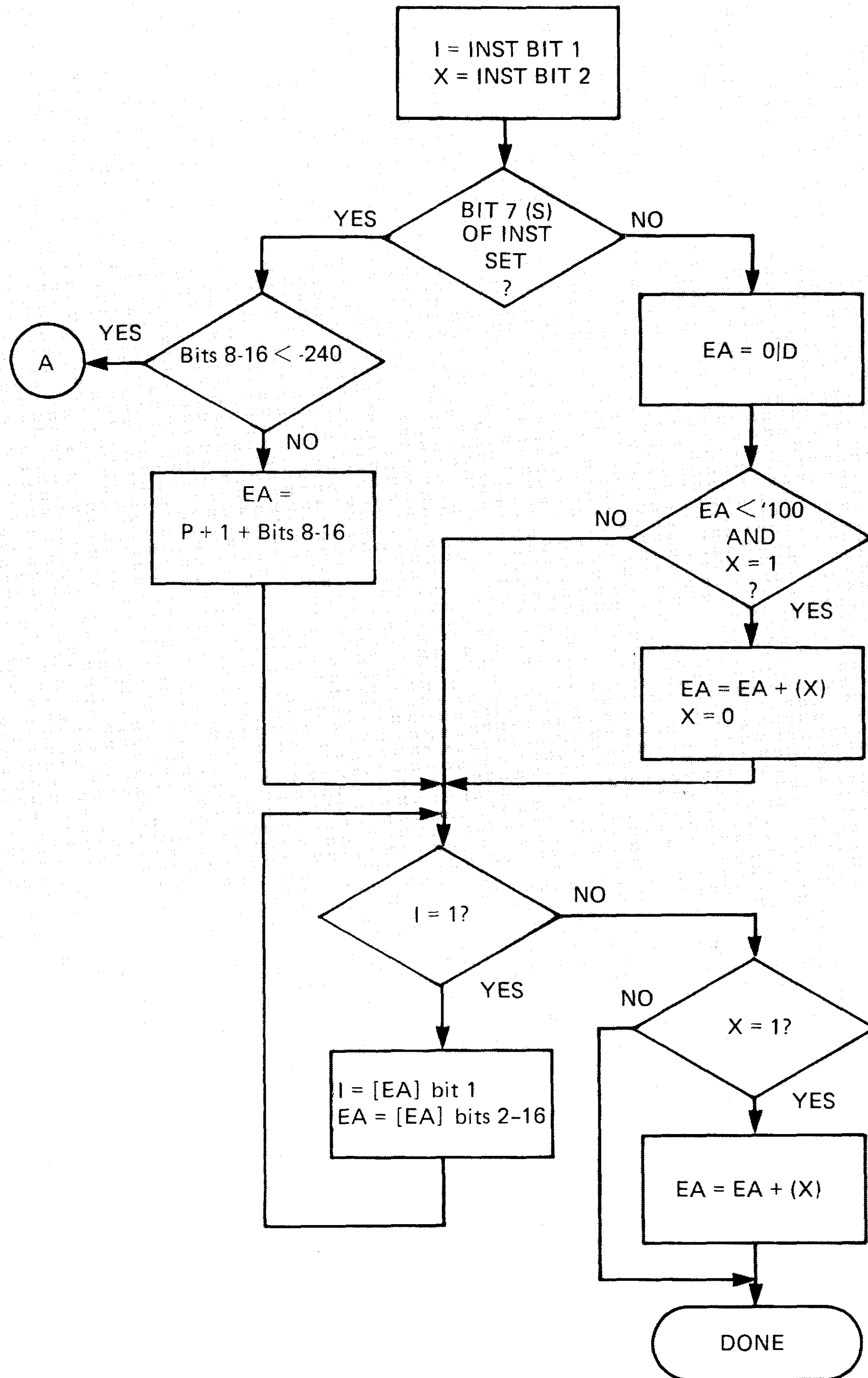


Figure 10-3. 32R Address Calculation (1 of 5)

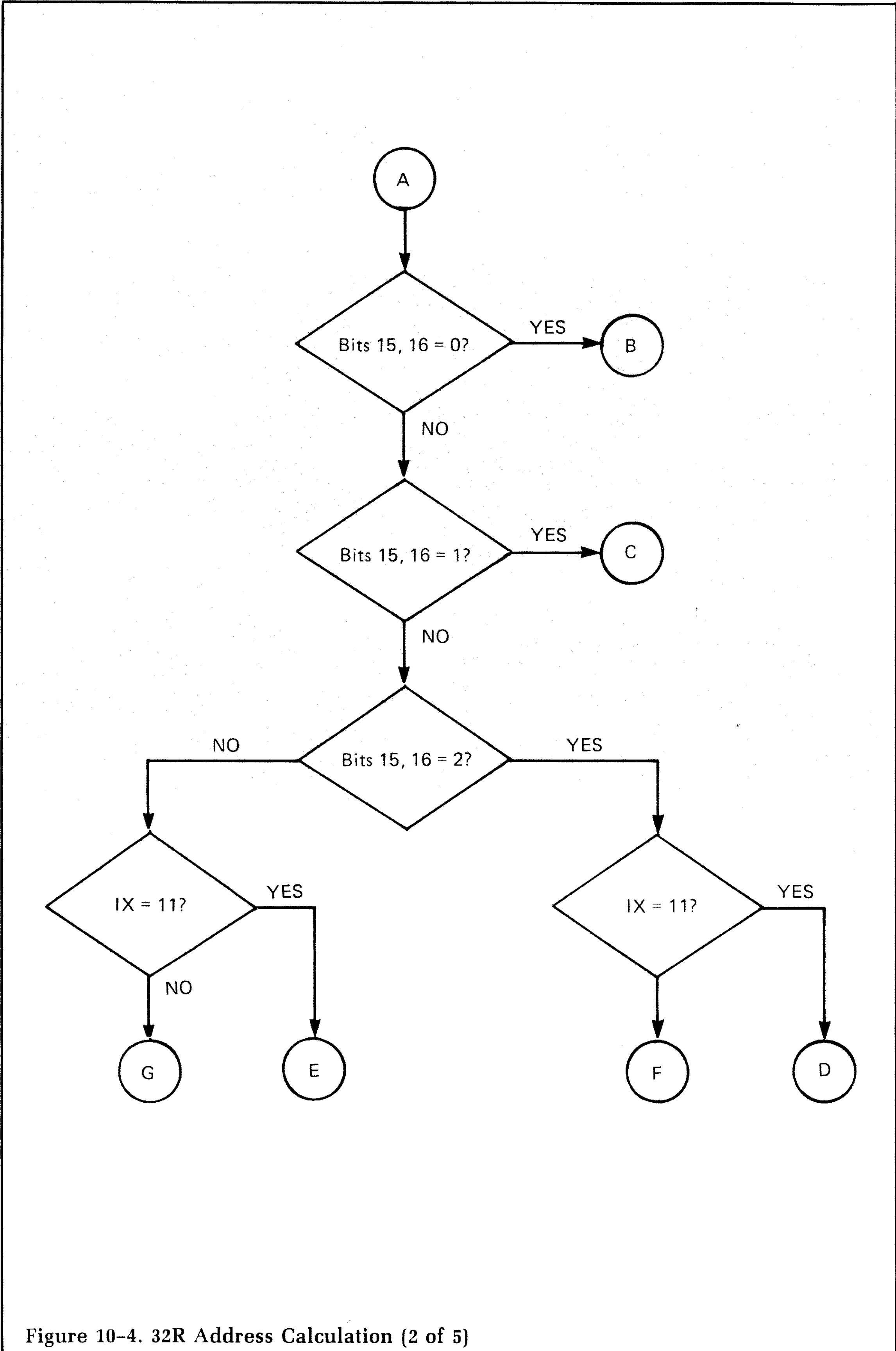


Figure 10-4. 32R Address Calculation (2 of 5)

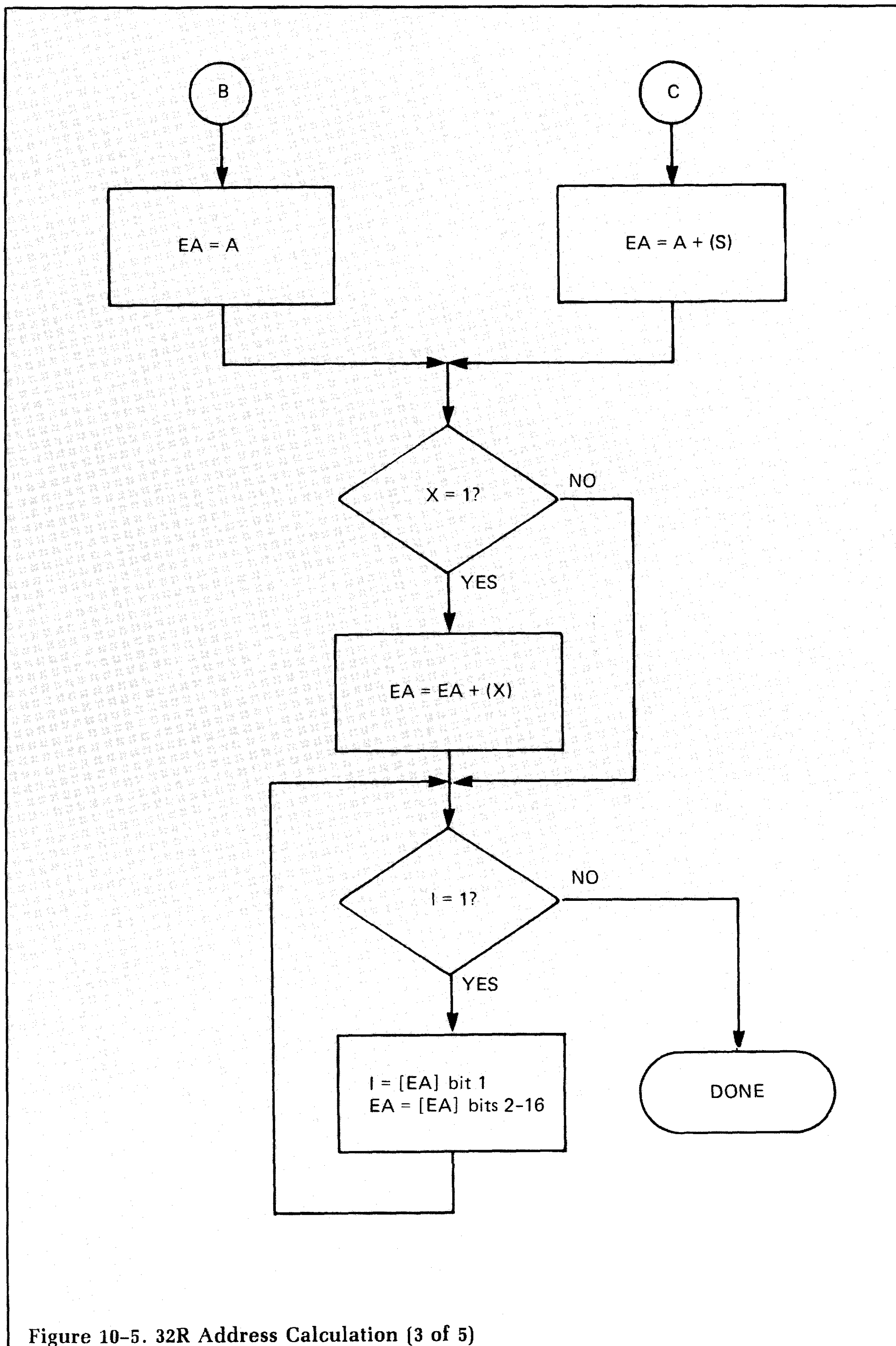


Figure 10-5. 32R Address Calculation (3 of 5)

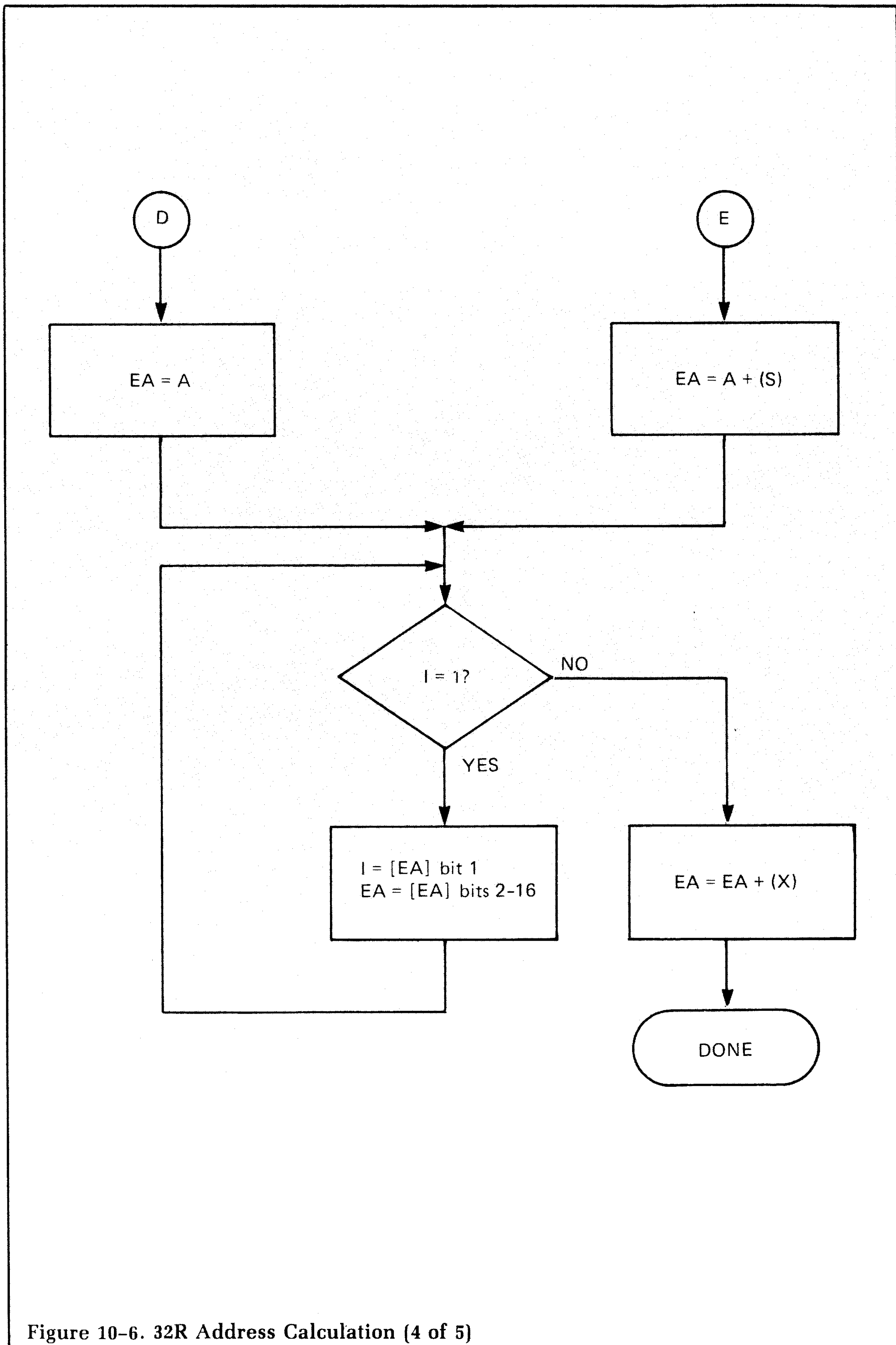


Figure 10-6. 32R Address Calculation (4 of 5)

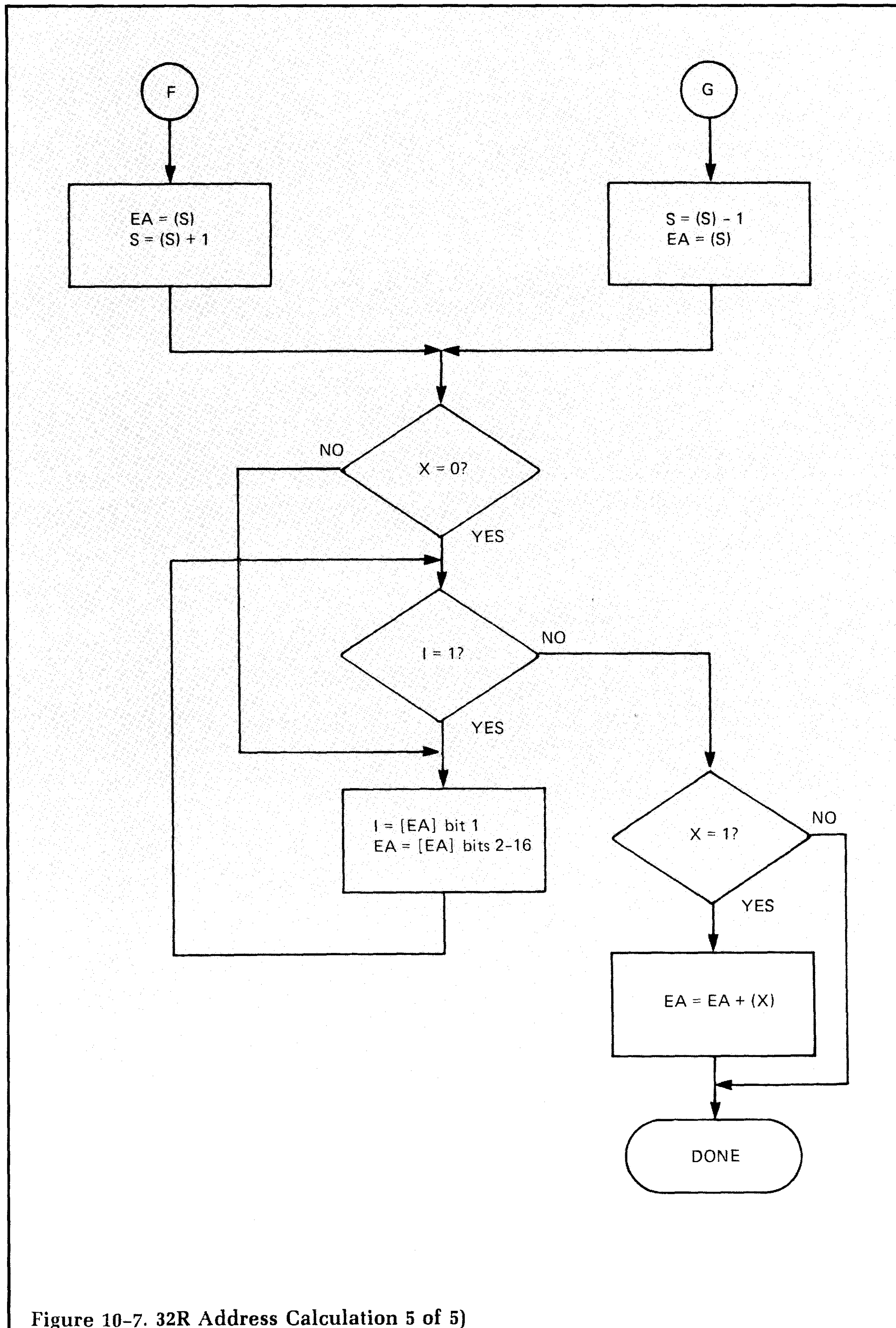
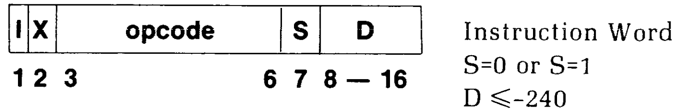


Figure 10-7. 32R Address Calculation 5 of 5)

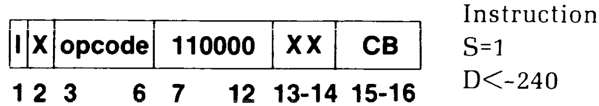
64R summary

Address length: 16 bits; 64K word address space

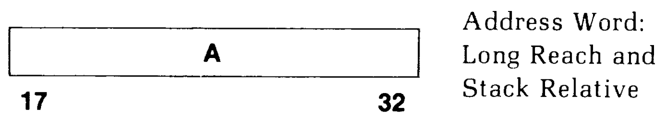
Format:



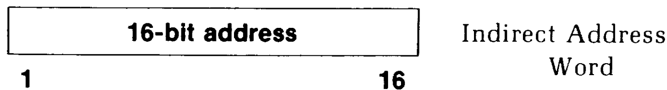
Instruction Word
S=0 or S=1
D ⩽-240



Instruction
S=1
D<-240



Address Word:
Long Reach and
Stack Relative



Indirect Address
Word

Indexing: One level.

Indirection: One level.

I	X	S	CB	D	EA	Assembler Notation	Type
0	0	0	—	0 to '777	0|D	LDA ADDR	*Direct*
0	1	0	—	0 to '777	0|D+X	LDA ADDR,1	*Indexed*
1	0	0	—	0 to '777	I(0|D)	LDA ADDR,*	*Indirect*
1	1	0	—	0 to '77	I(0|D+X)	LDA ADDR,1*	*Indirect, preindexed*
1	1	0	—	'100 to '777	I(0|D)+X	LDA ADDR*1	*Indirect, postindexed*
0	0	1	—	-240 to +255	P+D	LDA ADDR	*Direct*
0	1	1	—	-240 to +255	P+D+X	LDA ADDR,1	*Indexed*
1	0	1	—	-240 to +255	I(P+D)	LDA ADDR,*	*Indirect*
1	1	1	—	-240 to +255	I(P+D)+X	LDA ADDR,*1	*Indirect, postindexed*
0	0	1	2	—	SP	LDA @+	*Postincrement*
0	1	1	2	—	I(SP)+X	LDA @+,*1	*Postincrement, indirect, postindexed*
1	0	1	2	—	I(SP)	LDA @+,*	*Postincrement, indirect*
0	0	1	3	—	SP-1	LDA -@	*Predecrement*
0	1	1	3	—	I(SP-1)+X	LDA -@,*1	*Predecrement indirect, postindexed*

1	0	1	3	—	I(SP-1)	LDA -@,*	*Predecrement, indirect*
0	0	1	0	—	A	LDA% ADDR	*Direct, long reach*
0	1	1	0	—	A+X	LDA% ADDR,X	*Indexed, long reach*
1	0	1	0	—	I(A)	LDA% ADDR,*	*Indirect, long reach*
1	1	1	0	—	I(A+X)	LDA% ADDR,X*	*Indirect, preindexed long reach*
1	1	1	0	—	I(A)+X	LDA% ADDR,*X	*Indirect, postindexed long reach*
0	0	1	1	—	A+SP	LDA @+ADDR	*Direct, stack relative*
0	1	1	1	—	A+SP+X	LDA @+ADDR,X	*Indexed, stack relative*
1	0	1	1	—	I(A+SP)	LDA @+ADDR,*	*Indirect, stack relative*
1	1	1	1	—	I(A+SP+X)	LDA @+ADDR,X*	*Indirect, preindexed, stack relative*
1	1	1	3	—	I(A+SP)+X	LDA @+ADDR,*X	*Indirect, postindexed, stack relative*

P	Contents of program counter after instruction fetch (pointing at instruction plus 1).
0|D	Displacement into sector 0. Sector bits of effective address (bits 3-8) are zero.
X	Contents of index register.
I(expression)	Treat effective address as indirect address.
SP	Stack pointer.
ADDR	Location addressed by the LDA.

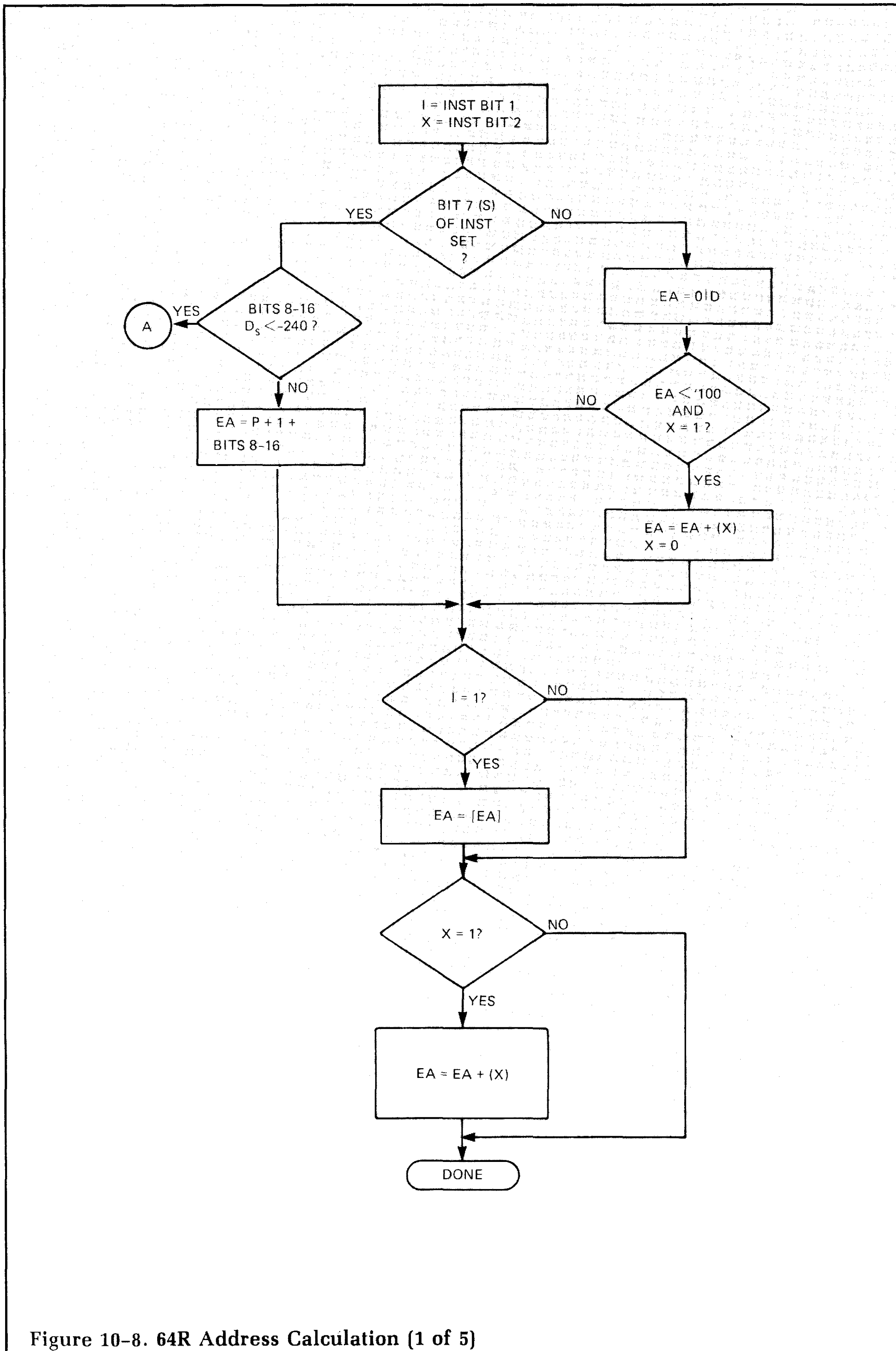


Figure 10-8. 64R Address Calculation (1 of 5)

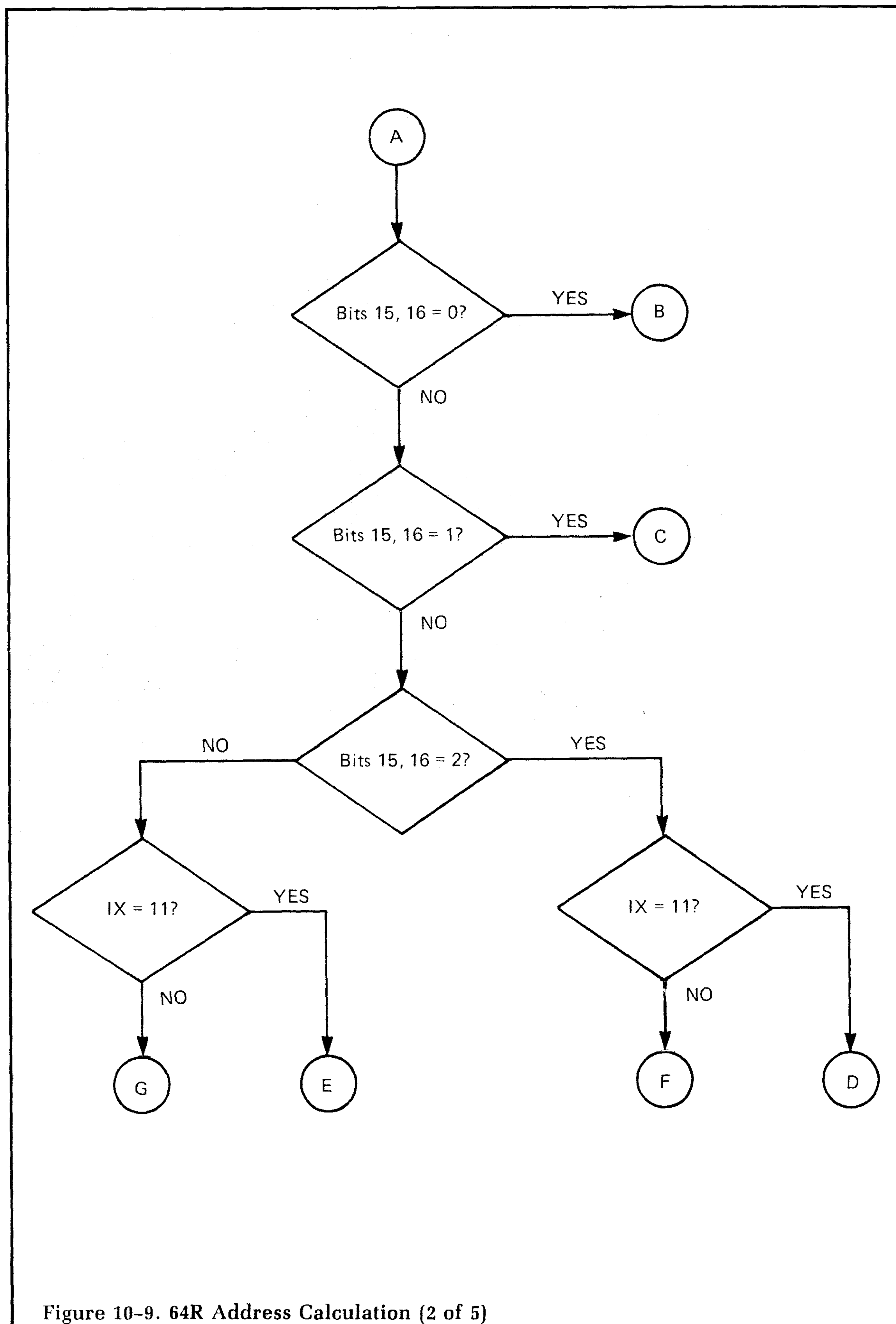


Figure 10-9. 64R Address Calculation (2 of 5)

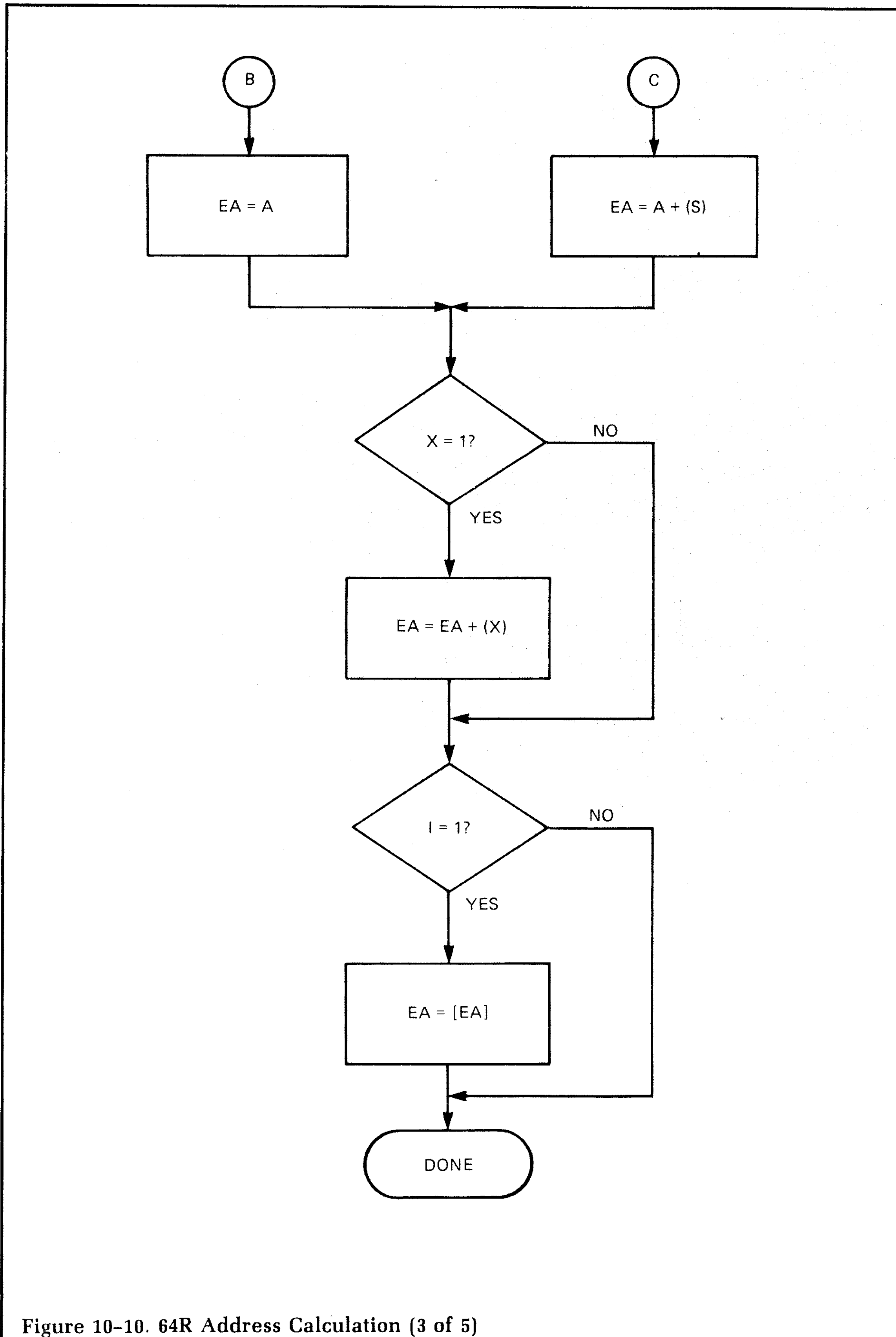


Figure 10-10. 64R Address Calculation (3 of 5)

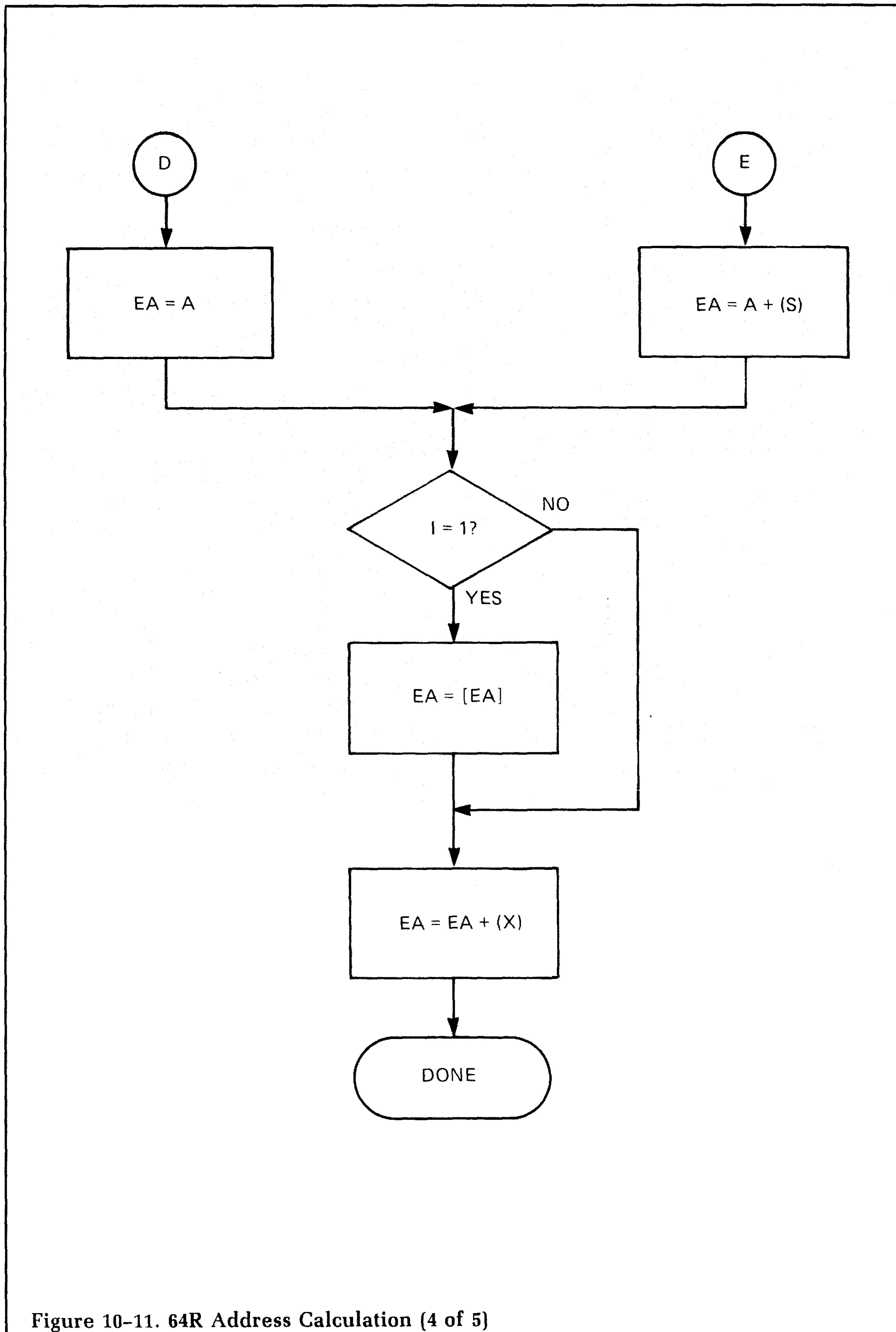


Figure 10-11. 64R Address Calculation (4 of 5)

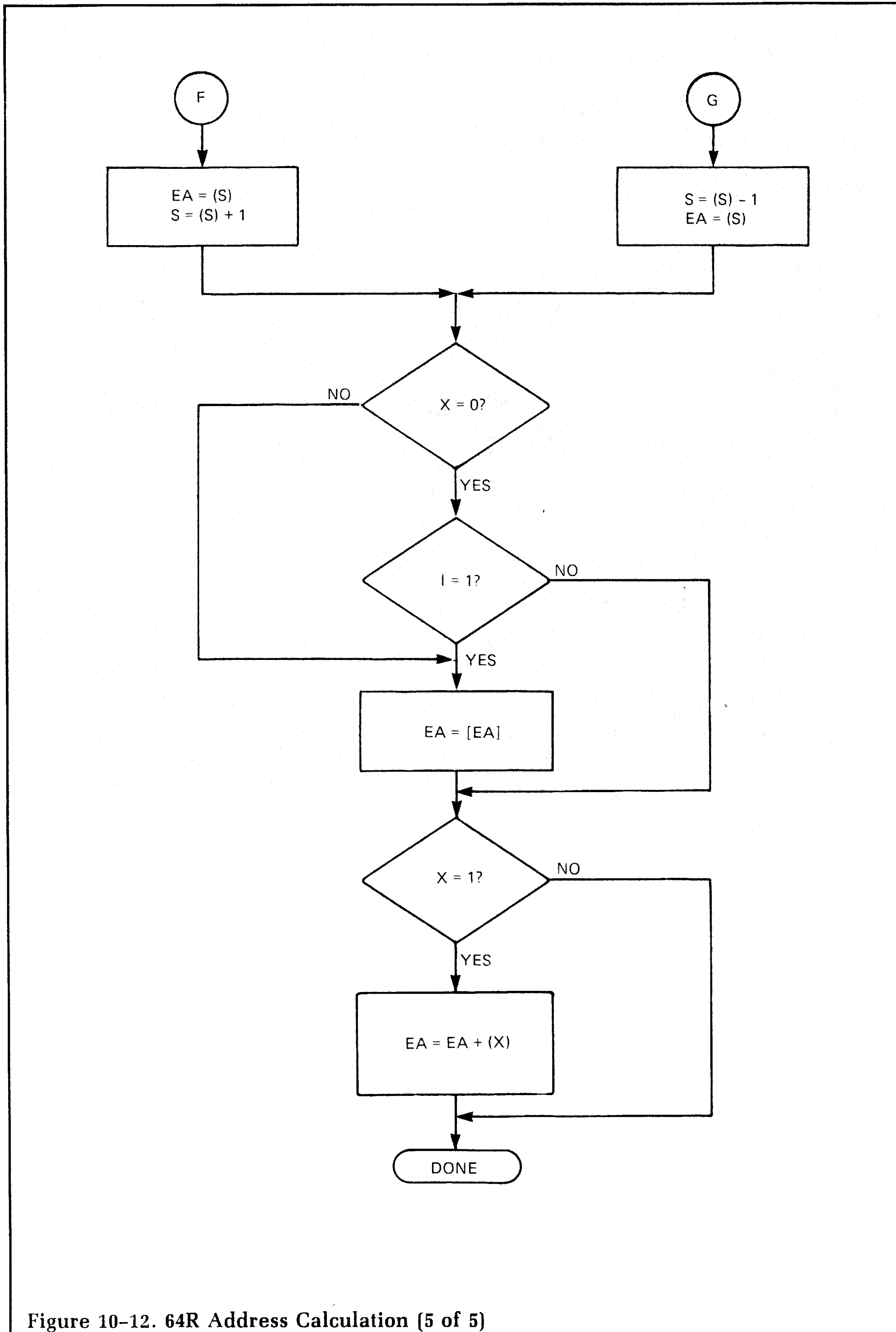
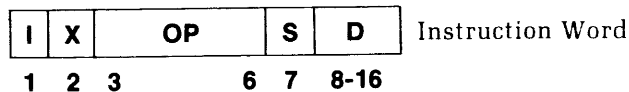


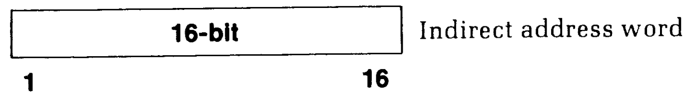
Figure 10-12. 64R Address Calculation (5 of 5)

64V PROCEDURE RELATIVE (One Word, S=1)

Address length: 16 bits; 64K word address space

Format:





Indexing: One level

Indirection: One level

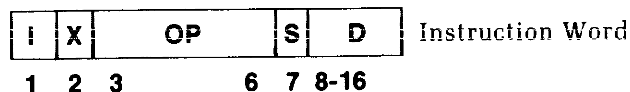
I	X	S	D	EA	Type
0	0	1	− 224 to + 255	P+D	*Direct*
0	1	1	− 224 to + 255	P+D+X	*Indexed*
1	0	1	− 224 to + 255	I(P+D)	*Indirect*
1	1	1	− 224 to + 255	I(P+D)+X	*Indirect, postindexed*

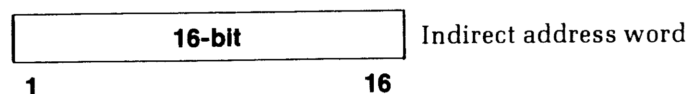
P	Contents of program counter after instruction fetch (pointing at instruction plus one).
D	Procedure segment displacement.
X	Contents of X register.
I[expression]	Treat effective address as indirect address.

64V BASE REGISTER RELATIVE (One Word, S=0)

Address Length: 3 64K segments

Format:





Indexing: One level

Indirection: One level

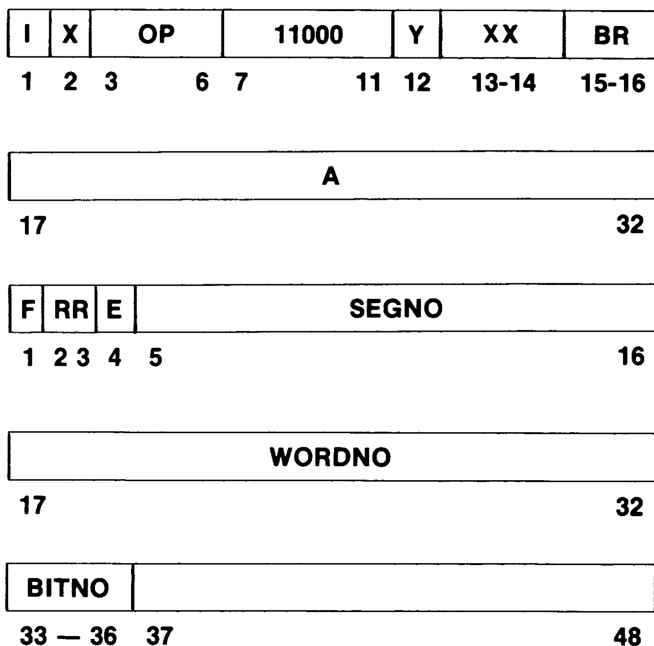
I	X	S	D	EA	Type
0	0	0	0–'7	register location	*Direct*
			'10–'777	SB+D	
			'400–'777	LB+D	
0	1	0	0–'377	if D+X<'10 then EA = register location * else SB+D+X	*Indexed*
			'400–'777	LB+D+X	
1	0	0	0–'7	I(REG)	*Indirect*
			'10–'777	I(PB|D)	
1	1	0	0–'77	I(PB|D+X)	*Indirect, preindexed*
1	1	0	'100–'777	I(PB|D)+X	*Indirect, postindexed*

REG	R-mode registers, i.e., A, B, X, etc.
PB	Procedure base register.
LB	Link base register.
SB	Stack base register.
X	Index register.
D	Displacement field.
I(expression)	Treat effective address as indirect address.
*	This is called an address trap.

64V TWO WORD MEMORY REFERENCE

Address length: 28 bits; 4096 64K segments

Format:



Indexing: X and Y

Indirection: 48 bit word

I	X	Y	BR	Effective Address	Meaning
0	0	0	0	PB|D	
			1	SB+D	*Direct*
			2	LB+D	
			3	XB+D	
0	0	1	0	PB|D+Y	
			1	SB+D+Y	*Indexed by Y*
			2	LB+D+Y	
			3	XB+D+Y	
0	1	0	0	PB|D+X	
			1	SB+D+X	
			2	LB+D+X	*Indexed by X*
			3	XB+D+X	
0	1	1	0	I(PB|D)	
			1	I(SB+D)	*Indirect*
			2	I(LB+D)	
			3	I(XB+D)	
1	0	0	0	I(PB|D+Y)	
			1	I(SB+D+Y)	*Preindexed by Y*
			2	I(LB+D+Y)	
			3	I(XB+D+Y)	
1	0	1	0	I(PB|D)+Y	
			1	I(SB+D)+Y	*Postindexed by Y*
			2	I(LB+D)+Y	
			3	I(XB+D)+Y	
1	1	0	0	I(PB|D+X)	
			1	I(SB+D+X)	*Preindexed by X*
			2	I(LB+D+X)	
			3	I(XB+D+X)	
1	1	1	0	I(PB|D)+X	
			1	I(SB+D)+X	*Postindexed by X*
			2	I(LB+D)+X	
			3	I(XB+D)+X	

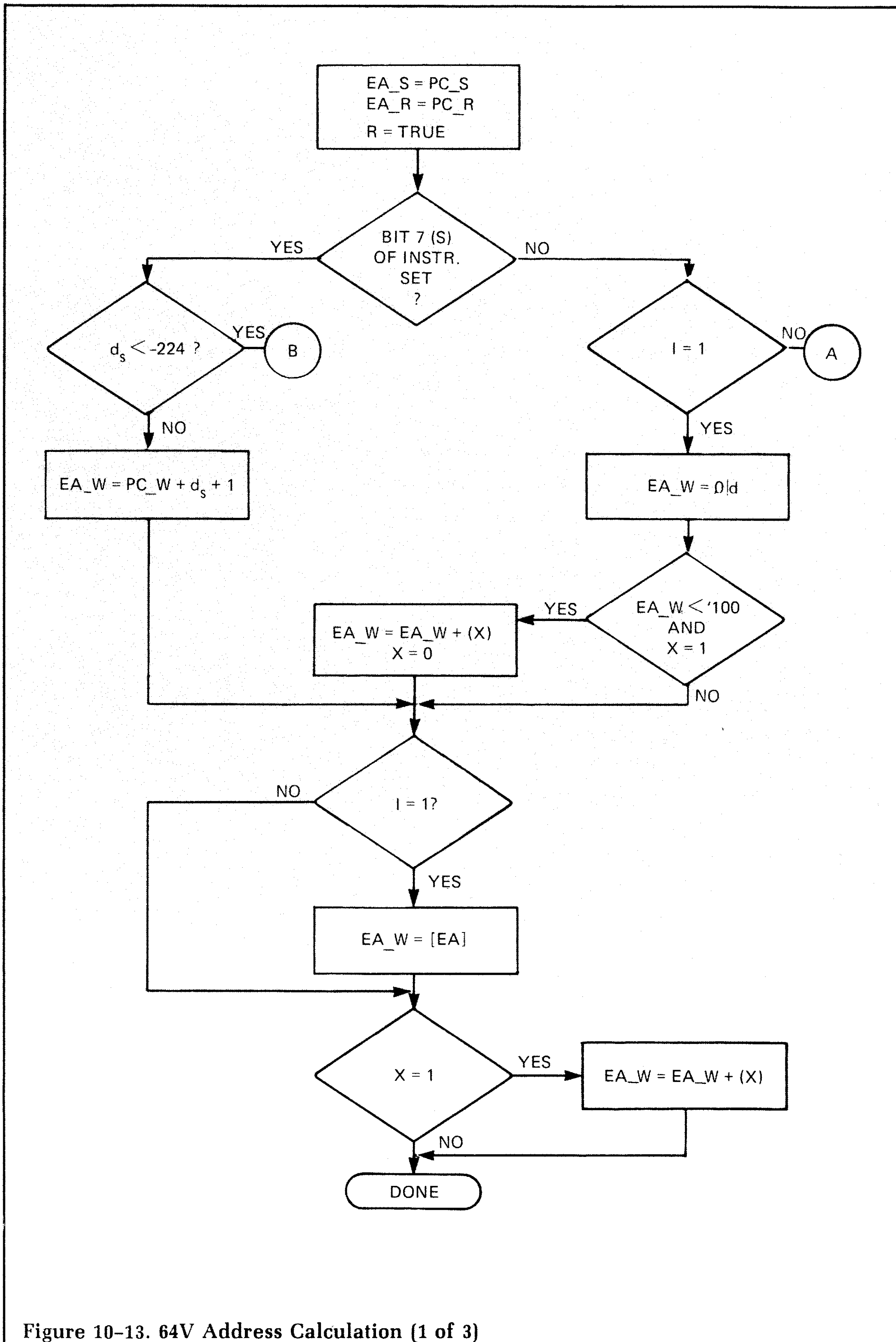


Figure 10-13. 64V Address Calculation (1 of 3)

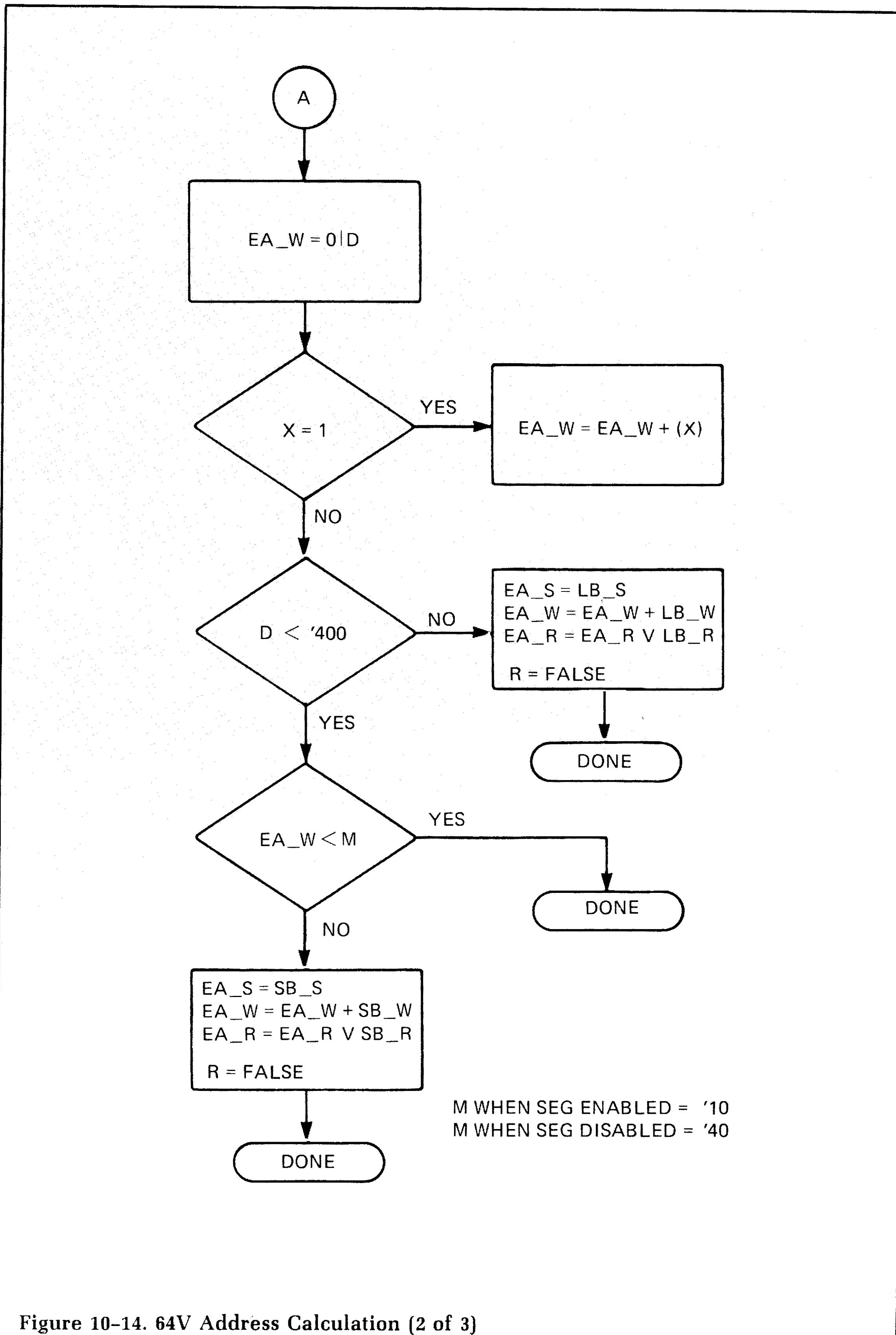


Figure 10-14. 64V Address Calculation (2 of 3)

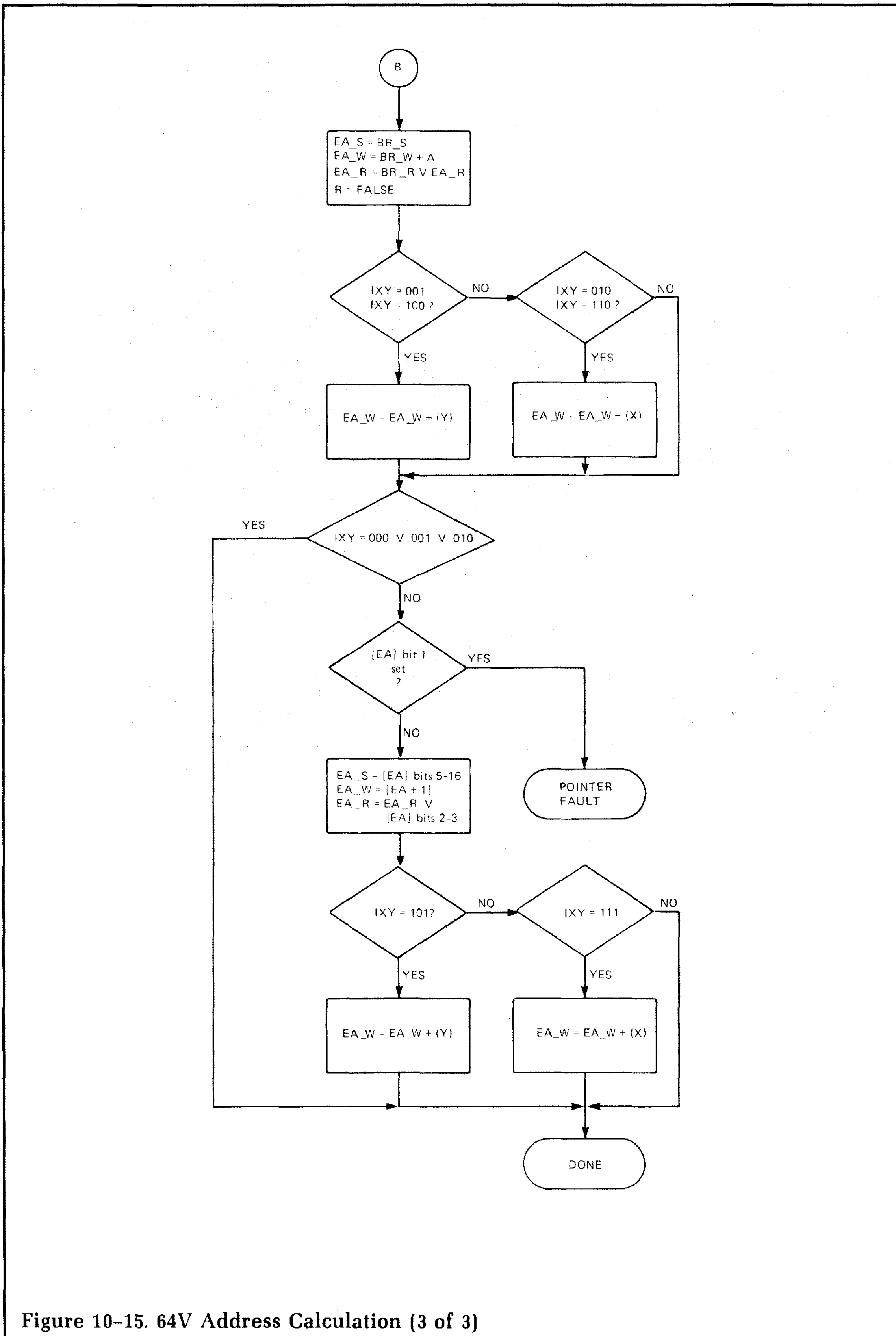


Figure 10-15. 64V Address Calculation (3 of 3)

11 Instruction definitions - SRV

ADDRESSING MODE—ADMOD

Set the addressing mode of the machine.

▶ E16S Enter 16S mode

Use 16S address calculations to form subsequent effective addresses and enable S-mode interpretation of instruction. See section on address resolution for details. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000011, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ E32S Enter 32S mode

Use 32S address calculations to form subsequent effective addresses and enable S-mode interpretation of instructions. See section on address resolution for details. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000013, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ E32R Enter 32R mode

Use 32R address calculations to form subsequent effective addresses and enable R-mode interpretation of instructions. See section on address resolution for details. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=001013, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ E64R Enter 64R mode

Use 64R address calculations to form subsequent effective addresses and enable R-mode interpretation of instructions. See section on address resolution for details. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=001011, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ E64V Enter 64V mode

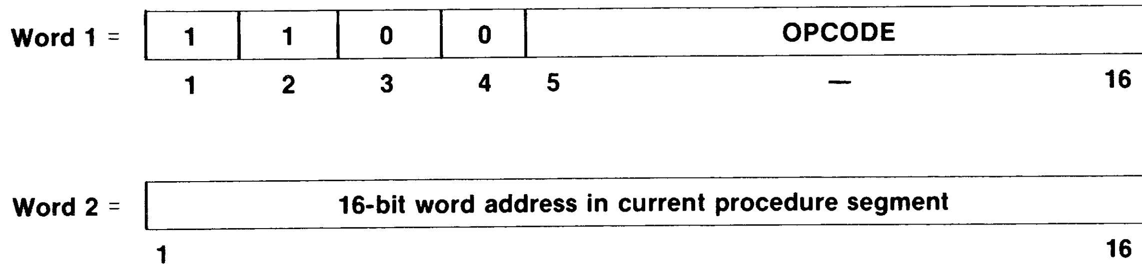
Use 64V address calculations to form subsequent effective addresses and enable 64V-mode interpretation of instructions. See section on address resolution for details. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000010, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ E32I Enter 32I mode

Use 32I address calculations to form subsequent effective addresses and enable 32I-mode interpretation of instructions. See section on address resolution for details. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=001010, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

BRANCH—BRAN

The branch instructions are two word generics which test the contents of a register or the result of a previous ARITHMETIC or COMPARE operation, as indicated by the condition codes (CC), the C-bit, and the L-bit. The bit layout is:



Condition code branches test six conditions based on the LT bit, the EQ bit, and the opcode.

Condition	Meaning
$<$	Branch if LT bit set and EQ bit cleared
$\leqslant$	Branch if LT bit set or EQ bit set
$=$	Branch if EQ bit set
$\neq$	Branch if EQ bit cleared
$\geqslant$	Branch if LT bit cleared or EQ bit set
$>$	Branch if LT bit cleared and EQ bit cleared

MODES=V, **FORMAT**=BRAN, **OPCODE**=see charts below, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ Test condition code and branch

These instructions have the following format:

Branch if condition code {LT, LE, EQ, NE, GE, GT}

For example: BCLT addr means Branch to addr if the condition code is less than zero (LT bit set and EQ bit cleared).

Mnemonic	Function	Opcode
BCLT addr	If CC$<$, then addr→PC	141604
BCLE addr	If CC$\leqslant$, then addr→PC	141600
BCEQ addr	If CC$=$, then addr→PC	141602
BCNE addr	If CC$\neq$, then addr→PC	141603
BCGE addr	If CC$\geqslant$, then addr→PC	141605
BCGT addr	If CC$>$, then addr→PC	141601

▶ Test magnitude condition and branch

These instructions have the following format:

Branch to addr if L=1 and condition code { **LT** / **LE** / **EQ** / **NE** / **GE** / **GT** }

For example: BMLT addr means Branch to addr if the L-bit is set and condition code is less than 0 (LT bit set and EQ bit cleared).

Mnemonic	**Function**	**Opcode**
BMLT addr	If L=1 and CC<, then addr→PC	141707
BMLE addr	If L=1 and CC⩽, then addr→PC	141711
BMEQ addr	If L=1 and CC=, then addr→PC	141602
BMNE addr	If L≠1 and CC≠, then addr→PC	141603
BMGE addr	If L=1 and CC⩾, then addr→PC	141606
BMGT addr	If L=1 and CC>, then addr→PC	141710

▶ Test C-bit and branch

Branch if C-bit { 0 / 1 }

BCR addr Branch if C-bit reset (equals zero): If C-bit=0, then addr→PC. **OPCODE**=141705.

BCS addr Branch if C-bit set (equals one): If C-bit=1, then addr→PC. **OPCODE**=141704.

Test L-bit

▶ Test L-bit { 0 / 1 }

BLR addr Branch if L-bit reset (equals zero): If L-bit=0, then addr →PC. **OPCODE**=141707.

BLS addr Branch if L-bit set (equals one): If L-bit=1, then addr→PC.

▶ Branch on register

These instructions have the following format:

Branch if { **A-Register (blank)** / **L-Register (L)** / **Floating-Register (F)** } { **LT** / **LE** / **EQ** / **NE** / **GE** / **GT** } **0**

For example: BLT addr means Branch to addr if the contents of the A register is less than zero (LT bit is set and EQ bit is cleared). **MODES**=V, **FORMAT**=BRAN, **OPCODES**=see chart below, **C**=unchanged, **L**=unchanged, **CC**=result.

Mnemonic	Function	Opcode
BLT addr	If A<0, then addr→PC	140614
BLE addr	If A⩽0, then addr→PC	140610
BEQ addr	If A=0, then addr→PC	140612
BNE addr	If A≠0, then addr→PC	140613
BGE addr	If A⩾0, then addr→PC	140615
BGT addr	If A>0, then addr→PC	140611
BLLT addr	If L<0, then addr→PC	140614
BLLE addr	If L⩽0, then addr→PC	140700
BLEQ addr	If L=0, then addr→PC	140702
BLNE addr	If L≠0, then addr→PC	140703
BLGE addr	If L⩾0, then addr→PC	140615
BLGT addr	If L>0, then addr→PC	140701
BFLT addr	If F<0, then addr→PC	141614
BFLE addr	If F⩽0, then addr→PC	141610
BFEQ addr	If F=0, then addr→PC	141612
BFNE addr	If F≠0, then addr→PC	141613
BFGE addr	If F⩾0, then addr→PC	141615
BFGT addr	If F>0, then addr→PC	141611

▶ Increment or decrement X or Y and branch

$$\left\{ \begin{matrix} \textbf{Increment} \\ \textbf{Decrement} \end{matrix} \right\} \left\{ \begin{matrix} X \\ Y \end{matrix} \right\} \textbf{ by 1 then branch to addr if result} \neq 0$$

MODES=V, **FORMAT**=BRAN, **OPCODE**=see chart below, **C**=unchanged, **L**=unchanged, **CC** =unchanged.

Mnemonic	Function	Opcode
BIX addr	X+1→X; if X≠0 then addr→PC	141334
BIY addr	Y+1→Y;if Y≠0 then addr→PC	141324
BDX addr	X-1→X; if X≠0 then addr→PC	140734
BDY addr	Y-1→Y; if Y≠0 then addr→PC	140724

▶ CGT Computed GOTO

If 1⩽A<, then [PC+A]→PC else PC+n→PC

Instruction word followed by n further words: word 1 contains integer n and words 2-n contain branch addresses within the current procedure segment.

If the contents of register A is less than n and greater than or equal to 1, then control passes to the address in PC+A; otherwise no branch is taken and control passes to PC+n. **MODES** =V, **FORMAT**=GEN, **OPCODE**=001314, **C**=unspecified, **L**=unspecified, **CC**=unspecified.

CHARACTER STRING OPERATIONS—CHAR

These instructions use the field address and length registers (FALR) which have been set up by field operation instructions prior to the use of these instructions. Character string operations perform memory to memory operations on variable length character fields. The FAR is used as a byte pointer and the bit offset (low order 3 bits) is ignored.

Data type: Characters are 8-bit bytes. The format is unspecified and may be determined by programmer, e.g., ASCII, EBCDIC, etc. The translate instruction (ZTRN), for example uses a table set up by the programmer to translate one character code into another.

▶ LDC FALR Load character

If field length register FLR is nonzero, load the single character pointed to by field address register FAR into A register bits 9-16. A register bits 1-8 are cleared. The field address register is advanced 8 bits to the next character, and the field length register is decremented by 1. Set condition code NE (clear EQ). If the specified field length register is zero, then set the condition code EQ. **MODES**=V, **FORMAT**=CHAR, **FALR 0 OPCODE**=001302, **FALR 1 OPCODE**=001312, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ STC FALR Store character

Store bits 9-16 of the A register into the character pointed to by field address register. The field address register is advanced 8 bits to the next character, and the field length register is decremented by 1. Set the condition code NE. If the field length register is zero, set the condition code EQ and do not store. **MODES**=V, **FORMAT**=CHAR, **FALR 0 OPCODE**=001322, **FALR 1 OPCODE**=001332, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ ZCM Compare character field

Compare field 0 to field 1 and set condition codes based on the results. If the fields are not of equal length, the shorter field is logically padded with ASCII blanks ('240).

Setup:

FAR 0	Field 0 address (byte aligned).
FLR 0	Length of field 0 in characters.
FAR 1	Field 1 address (byte aligned).
FLR 1	Length of field 1 in characters.

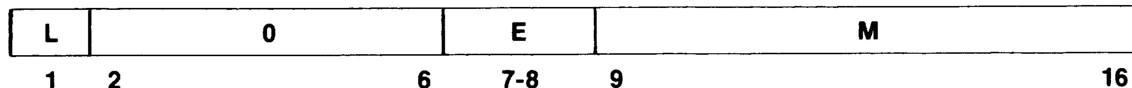
Condition code	**Result**
EQ	Field 0=field 1
LT	Field 0<field 1
GT ((LT and EQ))	Field 0>field 1

MODES=V, **FORMAT**=CHAR, **OPCODE**=001117, **C**=unchanged, **L**=unchanged, **CC**=results.

▶ ZED Edit character field

Move characters from field 0 into field 1 under the control of an edit program pointed to by XB. Movement stops when the source field is exhausted or when the end of the edit program is reached.

Edit Program Word:



L	Last entry if set.
0	Must be zero.
E	Edit opcode.
M	Edit modifier.

Opcode (E)	Mnemonic	Definition
0	CPC	Copy M characters from source to destination.
1	INL	Insert literal character M.
2	SKC	Skip M characters.
3	BLK	Supply M blanks (ASCII '240).

Setup:

FAR 0	Address of source field (byte aligned).
FAR 1	Address of destination field (byte aligned).
FLR 1	Number of characters to move and edit.
XB	Address of edit program.

MODES=V, **FORMAT**=CHAR, **OPCODE**=001111, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ZFIL Fill field

Store the character contained in bits 9–16 of the A register into each character of field 1.

Setup:

A(9–16)	Character to fill.
FAR 1	Destination field address (byte aligned).
FLR 1	Destination field length in bytes.

MODES=V, **FORMAT**=CHAR, **OPCODE**=001116, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ZMV Move character field

Move characters from field 0 to field 1, going from left to right. If the source field is shorter than the destination field, the destination field is padded with ASCII blanks ('240). If the source field is longer than the destination field, the remainder of the source field is not moved. The field address and length registers are left in an undefined state by this operation.

Setup:

FAR 0	Source field address (byte aligned).
FLR 0	Source field length in bytes.
FAR 1	Destination field address length (byte aligned).
FLR 1	Destination field length in bytes.

MODES=V, **FORMAT**=CHAR, **OPCODE**=001114, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ZMVD Move equal length fields

Move characters from field 0 to field 1. There is no padding or truncation since only the number of characters to be moved is specified.

Setup:

FAR 0	Source field address (byte aligned).
FAR 1	Destination field address (byte aligned).
FLR 1	Number of characters to move.

MODES=V, **FORMAT**=CHAR, **OPCODE**=001115, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ZTRN Translate character field

Use each character in field 0 as an index into the 256 byte table addressed by the XB register. Store each selected table character in the successive characters of field 1. Source and destination length are the same, specified by field length register 1.

Setup:

FAR 0	Source field address (byte aligned).
FAR 1	Destination field address (byte aligned).
FLR 1	Number of characters to translate and move.
XB	Address of 256-byte translate table.

For example: the source field contains a character A. The ASCII code is '301. Thus, the translate table location '301, which contains a $, is accessed. This $ is put into the destination field.

MODES=V, **FORMAT**=CHAR, **OPCODE**=001110, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

CLEAR REGISTER—CLEAR

▶ CAL Clear A left byte

0→A(1–8)

Clear bits 1–8 of register A without affecting bits 9–16. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=141050, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CAR Clear A right byte

0→A(9–16)

Clear bits 9–16 of register A without affecting bits 1–8. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=141044, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CRA Clear the A register

0→A

Reset the contents of register A to zero. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140040, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CRB Clear the B register

0→B

Reset the contents of register B to zero. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140015, **C** =unchanged, **L**=unchanged, **CC**=unchanged.

▶ CRE Clear E

0→E

Reset the contents of register E to zero. **MODES**=V, **FORMAT**=GEN, **OPCODE**=141404, **C**= unchanged, **L**=unchanged, **CC**=unchanged.

▶ CRL Clear long

0→L

Reset the contents of register L to zero. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140010, **C** =unchanged, **L**=unchanged, **CC**=unchanged.

▶ CRLE Clear L and E

0→L, 0→E

Reset the contents of registers L and E to zero. **MODES**=V, **FORMAT**=GEN, **OPCODE**= 141410, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

DECIMAL ARITHMETIC—DECI

These instructions use the field address and length registers which have been set up by field operation instructions prior to the use of the decimal arithmetic instruction. The general setup is:

EAFA 0	Source field address.
EAFA 1	Destination field address.
LDL	Control word (described below) decimal operation.

Variations on this pattern are discussed in the appropriate instructions.

Decimal data types

The decimal instruction set operates on five types of decimal data. Table 11-1 summarizes the characteristics of each type.

Table 11-1. Decimal Data Type.

Type	Code	Size of Decimal Digit	Comments
Leading Separate Sign	0	8	A plus sign (+) or a space represents a positive number. Operations generate +. A minus sign (-) represents negative number.
Trailing Separate Sign	1	8	
Packed Decimal	3	4	Use 4-bit nibble to represent each digit, followed by sign nibble. Requires odd number of digits and must start on byte boundary.
Leading Embedded Sign	4	8	A single character represents a digit and the sign of the field. When more than one character is listed, all will be recognized, but only first will be given in result field.
Trailing Embedded Sign	5	8	Embedded sign characters are as follows:

Digit	Positive	Negative
0	0,+{	-,{
1	1 A	J
2	2 B	K
3	3 C	L
4	4 D	M
5	5 E	N
6	6 F	O
7	7 G	P
8	8 H	Q
9	9 I	R

Arithmetic instruction register usage (I-mode only)

All arithmetic instructions use general registers GR0, GR1, GR3, GR4, and GR6, FLR0, FLR1 as scratch registers. These registers are not guaranteed to remain the same if an arithmetic instruction is executed.

Control word format

To specify the characteristics of the operation to be performed, most decimal arithmetic instructions require a control word to be loaded in the L register (general register 2 in I-mode).

The general format is as follows:

A	—	B	C	—	T	D	E	F	G	H
1-6	7 8	9	10	11	12	13	14-16	17-22	23-29	30-32

Where:

A	Field 1, number of digits.
E	Field 1, decimal data type (see Table 11-1).
B	If set, sign of field 1 is treated as negation of its actual value.
C	If set, sign of field 2 is treated as negation of its actual value (*XAD, XMP, XDV, XCM only*).
D	If set, then round (XMV only).
F	Field 2, number of digits.
H	Field 2, decimal data type.
G	Scale differential (*XAD, XMV, XCM only*).
T	Generate positive results always.
-	Unused, must be zero.

The fields used by each instruction are listed in the instruction descriptions. Fields not used by an instruction must be zero.

The scale differential specifies the difference in decimal point alignment between the operator and fields for some instructions. This field is treated as a signed 7 bit two's complement number, where a positive value indicates a right shifting of field 1 with respect to field 2, and a negative value indicates a left shifting.

Decimal exception (DEX)

There are two ways that an exception is handled. If the program is running in decimal exception mode, the a directed fault (similar to floating exception) is taken with the following fault codes:

	DEX Type (High)	Sub Code (Low)
Overflow	7	0
Divide by zero	7	1
Conversion	7	2

When not in decimal exception mode, the C bit is set and execution continues with the next instruction.

▶ XAD Decimal add

A	—	B	C		E	F	G	H
1-6	7 8	9	10	11 — 13	14-16	17-22	23-29	30-32

Add the source field to the destination field and place the results in the destination field. The control word determines:

1. The operation—addition or subtraction.
2. The scaling of the results.

Operations: The B and C fields control whether the operation is an add or subtract.

B	C	Operation
0	0	+ Source + Destination
0	1	+ Source – Destination
1	0	– Source + Destination
1	1	– Source – Destination

Scaling: G Field. The scale differential field in the control word is used to adjust field 1 in relation to field 2. If the scale differential is greater than zero, low order digits in field 1 will only affect the initial borrow from the low order digit of field 2. If the scale differential is less than zero, field 1 is considered to be logically extended with low order zeros when applied to field 2. **MODES**=V, **FORMAT**=DECI, **OPCODE**=001100, **C**=overflow, **L**=unchanged, **CC**=result.

▶ XBTD Binary to decimal conversion

A		E		H
1-6	7 13	14-16	17 29	30-32

Converts a 16, 32 or 64 bit signed binary number to decimal. The H field in the control word specifies the length and location of the binary source as follows:

0	16 Bits, located in EH
1	32 Bits, located in E
2	64 Bits, located in F

The condition codes are undefined for this operation. A conversion error exception is taken on overflow - see decimal exception.

This instruction converts the binary field present in EH, E or F (depending on field type) into a decimal field. Unlike the rest of the decimal arithmetic instructions, XBTD returns the decimal field in what elsewhere is known as the "source" field address register. **MODES**= V, **FORMAT**=DECI, **OPCODE**=001145, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ XCM Decimal compare

A	—	B	C		E	F	G	H
1-6	7 8	9	10	11 13	14-16	17-22	23-29	30-32

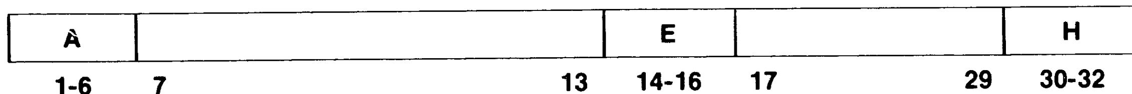
Sets the condition codes to reflect the comparison Field 2 :: Field 1 The scale difference applies as in XAD.

The condition codes are set as follows:

GT = Field 2 > Field 1
EQ = Field 2 = Field 1
LT = Field 2 < Field 1

MODES=V, **FORMAT**=DECI, **OPCODE**=001102, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ XDTB Decimal to binary conversion



Converts the decimal field to binary. The length of the binary field is specified in the H field of the control word as follows:

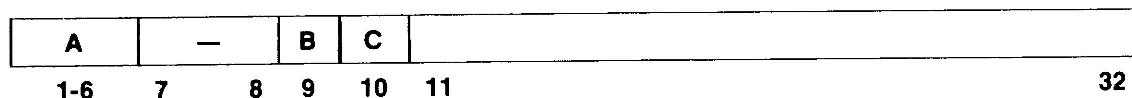
0	16 Bits, returned in A.
1	32 Bits, returned in L.
2	64 Bits, returned in L/E.

A conversion error exception is taken on overflow. The condition codes are undefined for this operation.

Field address register 1 is not used by this instruction and can be used as an accumulator for indexed pointers.

This instruction returns a 16, 32 or 64 bit integer in either the A, L, or L/E registers, depending on the destination field type. **MODES**=V, **FORMAT**=DECI, **OPCODE**=001146, **C** =unchanged, **L**=unchanged, **CC**=unspecified.

▶ XDV Decimal divide



Divide destination field by source field, placing both the quotient and remainder in the destination field.

The data type must be trailing sign embedded. To allow room for both quotient and remainder the destination field must contain the same number of leading zeros as the length of the source field.

After divide the destination field contains quotient of length (destination length—source length) followed by remainder of source length. A decimal exception (DEX) occurs if the source =0, the sign is not trailing embedded, or the destination is ⩽ source. **MODES**=V, **FORMAT**=DECI, **OPCODE**=001107, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ XED Numeric edit

Processes an edit sub-program addressed by the temporary base register (XB) to control the editing of the source field into the destination field. The source field must have leading separate sign, and must have the same number of digits and the same decimal point alignment as called for by the edit sub-program. Normal setup for the instruction would consist of a decimal move to correct the type, length, and alignment of the number to be edited. The A register must equal one if the source field is zero; otherwise the A register must be zero.

The edit sub-program consists of a list of words formatted as follows:

L	0	E	M
1	2-4	5-8	9 … 16

Where:

- L Last entry if set.
- E Edit opcode.
- M Edit modifier.

The XED instruction maintains several internal variables during its processing which are used to control the operation. These variables are:

- Zero suppress character—initial value is blank (ASCII '240).
- Floating edit character—initially not defined.
- Sign of the source field—established by fetching the first character of the source field.
- Significance flag—records the end of zero suppression.

MODES=V, **FORMAT**=DECI, **OPCODE**=001112, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ XMP Decimal multiply

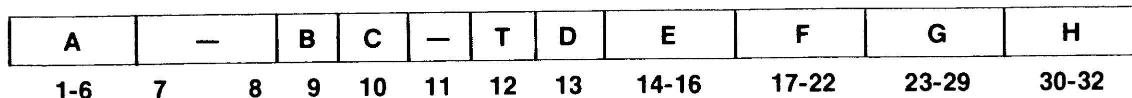
A	—	B	C	—	T	—	E	F	G	H
1-6	7-8	9	10	11	12	13	14-16	17-22	23-29	30-32

Multiply the multiplicand, in the source field, by the multiplier, in the destination field. The product is right justified in the destination field. To avoid overflow the destination field length must be greater than or equal to the number of significant digits in the multiplier plus the number of significant digits in the multiplicand. For example, to multiply 1234 by 567 set A=4, F=7, G=3. Note that the temporary base register (XB) is used by the instruction and may change. **MODES**=V, **FORMAT**=DECI, **OPCODE**=001104, **C**=overflow, **L**=unchanged, **CC**=result.

Table 11-2. Edit Sub-operations

Opcode	Mnemonic	Definition
00	**ZS**	Zero suppress next M digits. Digits are consecutively fetched from the source field and the significance flag is checked. If the significance flag is set, the digit is copied to the destination field. If the significance flag is clear and the digit is non-zero, the significance flag is set, the floating character inserted (if it is currently defined), and the digit is copied. Otherwise the zero suppress character is substituted for the zero digit in the destination field.
01	**IL**	Insert literal M in destination field.
02	**SS**	Set zero suppress character to M.
03	**ICS**	Insert literal M if the significance flag set; otherwise insert zero suppress character.
04	**ID**	Insert M digits. If significance flag is clear, it is set and the floating edit character inserted (if currently defined). Then copy M digits into the destination field.
05	**ICM**	Insert M if sign is minus; otherwise insert zero suppress character.
06	**ICP**	Insert M if sign is plus; otherwise insert zero suppress character.
07	**SFC**	Set floating character to M.
10	**SFP**	Set floating character to M if sign plus; otherwise set floating edit character to zero suppress character.
11	**SFM**	Set floating character to M if sign minus; otherwise set floating edit character to zero suppress character.
12	**SFS**	Set floating character to sign.
13	**JZ**	Jump M+1 locations ahead in edit sub program if source field equals zero.
14	**FS**	Fill next M characters with zero suppress character.
15	**SF**	Set significance flag.
16	**IS**	Insert sign.

▶ XMV **Decimal move**



Moves source to destination, changing the sign if the B bit in the control word is set, and rounding if the D bit is set and G, the scale differential, is greater than zero. If the scale differential is negative then zeros are supplied before field 1 is used for a source. The condition codes are set to reflect the state of the destination after the move. **MODES**=V, **FORMAT**=DECI, **OPCODE**=001101, **C**=unchanged, **L**=unchanged, **CC**=result.

FIELD OPERATIONS—FIELD

These instructions set up and manipulate the field address and length registers, which are used by both the decimal and character string instructions. The interpretation of the value in the field length registers depends on the data type and instruction using them.

▶ ALFA FAR Add L to field address

L+FAR→FAR

Add the 32-bit integer in register L, which represents an offset in bits, to the 26-bit unsigned word and bit number fields of the field address register. The low-order 26 bits of the sum replace the word and bit number fields of the field address register. All but the low order 20 bits of the sum must be zero. Example: to advance FAR 0 by 3 bytes, place 24 into the L register and execute ALFA 0. **MODES**=V, **FORMAT**=GEN, **FAR 0 OPCODE**=001301, **FAR 1 OPCODE**=001311, **C**=unspecified, **L**=unspecified, **CC**=unchanged.

▶ EAFA FAR, addr Effective address to field address register

[EA]48→FAR

Place the complete effective address, including the bit portion, in field address register FAR. The associated field length register is unchanged. **MODES**=V, **FORMAT**=AP, **FAR 0 OPCODE**=001300, **FAR 1 OPCODE**=001310, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LFLI FLR,DATA Load field length register immediate

DATA→FLR

Place the 16-bit unsigned integer in the second word of the instruction into field length register FLR. Clear the high order bits. This instruction loads the field length register with a constant which is 65535 or less. The associated field address register is unchanged. **MODES** =V, **FORMAT**=BRAN, **FLR 0 OPCODE**=001303, **FLR 1 OPCODE**=001313, **C**=unchanged, **L**= unchanged, **CC**=unchanged.

▶ STFA FAR,addr Store field address register

FAR→[EA]32 or [EA]48

Store contents of field address register FAR into addr as a hardware indirect pointer. If bit number field of the field address register is zero, store the first two words of the pointer and clear the pointer's extend bit; if bit number field is non-zero, store all three words of the pointer and set the pointer's extend bit. **MODES**=V, **FORMAT**=AP, **FAR 0 OPCODE**=001320, **FAR 1 OPCODE**=001330, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TFLL FLR Transfer field length register to L

FLR→L

Transfer the contents of field length register FLR to the L register as an unsigned 32-bit integer. Clear the high order 11 bits of L. **MODES**=V, **FORMAT**=GEN, **FLR 0 OPCODE**= 001323, **FLR 1 OPCODE**=001333, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TLFL FLR Transfer L-register to field length register

L→FLR

Transfer the 32-bit unsigned integer in the L register into field length register FLR. The high order 11 bits of L must be zero to make the high order 6 bits of the field length register equal to zero. This instruction is used to load the field length register with a value computed at

execution time. The maximum allowable field length is 2**20 (21 bits) - the number of bits in a 64K segment. **MODES**=V, **FORMAT**=GEN, **FLR 0 OPCODE**=001321, **FLR 1 OPCODE**=001331, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

FLOATING POINT ARITHMETIC—FLPT

See Section 9 for a description of the processor dependent register formats and the floating point data structures.

Normalization

The result of every floating point calculation is normalized. In normal form, the most significant digit of the mantissa follows the binary point. If an operation produces a mantissa that is smaller than normal, the mantissa is shifted left until the most significant bit differs from the sign bit, and the exponent is decreased by one for each shift. Bits vacated at the right are filled by zeros. If the result of an operation overflows the mantissa, it is shifted right one place, the overflow bit is made the most significant bit, and the exponent is increased by 1.

Floating point exceptions

In the basic arithmetic operations, increasing the exponent in the floating point register beyond 32639 is an overflow; decreasing it below −32896 is an underflow.

An attempt to store a single-precision number with an exponent greater than 127 or less than −128 in the two-word memory format results in a different type of exception - see Table 11-2. The number in the floating point register is not altered by the FST operation and so can be recovered if necessary.

Other detected exceptions are an attempt to divide by zero or to form an integer exceeding ±30 bits or about ±1 billion decimal.

On the Prime 350 and up, the floating point exception is a fault rather than an interrupt and is controlled by the floating point exception bit in the keys - see Section 9 - Data Formats.

Table 11-3. Floating Point Exception Codes

Register 11 (Precision)		Register 12	Type of Exception
Single	**Double**		
$100	$200	—	Overflow/Underflow (Exponent exceeds approx. 10 ±9800)
$101	$201	—	Division by zero
$102	—	(EA)	Attempt to store single precision exponent exceeding 8-bit memory format (>127, <−128)
$103	—	—	Attempt to form integer exceeding capacity. INT: A|B (30 bits) INTA: A (15 bits) INTL: L (31 bits)

Note

$ indicates hexadecimal codes

Table 11-4. Floating Point Mantissa and Exponent Ranges

Field	Single Precision-Memory	Single Precision-Register	Double Precision
Mantissa (two's complement)			
Bits	23 + Sign	31 + Sign	47 + Sign
Precision	±8,388,607	±2,147,483,647	±140,737,488,355,327
Exponent			
Bits	8	16	16
Range	-128 to +127	-32896 to +32639	-32896 to +32639
	$(10^{\pm 38})$	$(10^{+9823,-9902})$	$(10^{+9823,-9902})$

▶ DFAD addr Double precision floating add

F+[EA]64→F

Add the double precision number starting at addr to the double precision number in the floating point register and leave the result in the floating point register. (Same procedure as FAD except a 47-bit mantissa is produced.) **MODES**=RV, **FORMAT**=MR, **OPCODE**=06 02, **C** =overflow, **L**=unspecified, **CC**=unspecified.

▶ DFCM Double precision floating complement

-F→F

Two's complement the precision mantissa in floating point register and normalize if necessary. **MODES**=RV, **FORMAT**=GEN, **OPCODE**=140574, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DFCS addr Double precision floating point compare and skip

If F>[EA]64 then PC→PC
If F=[EA]64 then PC+)1→PC
If F<[EA]64 then PC+2→PC

If the contents of the floating point register is greater than the contents of addr, execute the next instruction.

If the contents of the floating point register equals the contents of addr, skip the next location in instruction sequence and execute the instruction at second location following.

If the contents of the floating point register is less than the contents of addr, skip next two locations in instruction sequence and execute the instruction at third location following. **MODES**=RV, **FORMAT**=MR, **OPCODE**=11 02, **C**=unspecified, **L**=unspecified, **CC**=unspecified.

▶ DFDV addr Double precision floating divide

F/[EA]64→F

Divide the contents of the floating point register by the number in addr and place the quotient in the floating point register with the mantissa normalized. **MODES**=RV, **FORMAT** =MR, **OPCODE**=17 02, **C**=overflow division by zero, **L**=unspecified, **CC**=unspecified.

▶ DFLD addr Double precision floating load

[EA]64→F

Load the double precision floating point number contained in the four memory words at addr into the floating point register. **MODES**=RV, **FORMAT**=MR, **OPCODE**=02 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ DFLX addr Double precision floating load index

[EA]16*4→X

Quadruple the contents of the effective address and load the result into the index register X. This instruction is useful for addressing arrays or tables of element size four words. **MODES**=V, **FORMAT**=MR, **OPCODE**=15 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ DFMP addr Double precision floating multiply

F*[EA]64→F

Multiply the contents of the floating point register by the contents of addr and place the products in the floating point register with the mantissa normalized. **MODES**=RV, **FORMAT**=MR, **OPCODE**=16 02, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DFSB addr Double precision floating subtract

F-[EA]64→F

Subtract the double precision floating point number starting at addr from the double precision floating point number in the floating point register. (Same procedure as FSB except a 47-bit mantissa is produced.) **MODES**=RV, **FORMAT**=MR, **OPCODE**=07 02, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DFST addr Double precision floating store

F→[EA]64

Store the double precision floating point number contained in the floating point register into the location specified by addr. Exponent and mantissa bit capacities are the same so that no floating point exceptions are possible. **MODES**=RV, **FORMAT**=MR, **OPCODE**=04 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FAD addr Floating add

F+[EA]32→F

Add the floating point number at addr to the contents of the floating point register and leave the resulting floating point number in the floating point register. Addition of floating point numbers is accomplished by right shifting the smaller number by the difference in the exponents. After alignment, the mantissas are added.

If there is an overflow from the most significant bit (not the sign), the sum mantissa is shifted right one place, the exponent is incremented by one and the overflow bit becomes the high-order bit in the normalized mantissa. If the result is otherwise not in normal form (as when numbers with unlike signs are added), the result is normalized. Overflow cannot occur. The C-Bit is cleared. **MODES**=RV, **FORMAT**=MR, **OPCODE**=06 01, **C**=cleared, **L**=unspecified, **CC**=unspecified.

▶ FCM Complement

-F→F

Two's complement the double precision mantissa in floating point register and normalize if necessary. **MODES**=RV, **FORMAT**=GEN, **OPCODE**=140574, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FCS addr Floating compare and skip

If F>[EA]32, then PC→PC
If F=[EA]32, then PC+1→PC
If F<[EA]32, then PC+2→PC

If the contents of the floating point register is greater than the contents of addr, execute the next instruction.

If the contents of the floating point register equals the contents of addr, skip the next location in instruction sequence and execute the instruction at second location following.

If the contents of the floating point register is less than the contents of addr, skip next two locations in instruction sequence and execute the instruction at third location following. **MODES**=RV, **FORMAT**=MR, **OPCODE**=11 01, **C**=unspecified, **L**=unspecified, **CC**=unspecified.

▶ FDBL Convert single to double float

F→F

Convert the single precision floating point number in the floating point register to a double precision precision floating point number in the floating point register. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140016, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FDV addr Floating divide

F/[EA]32→F

Divide the contents of the floating point register by the number in addr and place the quotient, with the mantissa normalized, in the floating point register. **MODES**=RV, **FORMAT**=MR, **OPCODE**=17 01, **C**=overflow division by zero, **L**=unspecified, **CC**=unspecified.

▶ FLD addr Floating load

[EA]32→F

Load the double precision number contained in the two successive words at addr into the floating point register. **MODES**=RV, **FORMAT**=MR, **OPCODE**=02 01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FLOT Convert 31-bit integer to float

Float(A|B)→F

Take the 31-bit integer in the combined A|B register and convert it into a normalized floating point number in the floating point register. **MODES**=R, **FORMAT**=GEN, **OPCODE**=140550, **C**=unspecified, **L**=unspecified, **CC**=unspecified.

▶ FLTA Convert integer to float

FLOT(A)→F

Convert the 16 bit integer in register A to a single precision floating point number in the floating point register. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140532, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FLTL Convert long integer to float

FLOT(L)→F

Convert the 32 bit integer in register L to a single precision floating point number in the floating point register. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140535, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FLX addr Floating load index

[EA]16*2→X

Double the contents of the effective address and load the result into the index register X. This instruction facilitates indexing sequences that involve double-word memory reference operations. It works directly for two-word indexing, e.g., 31-bit or 32-bit integer or floating point. **MODES**=RV, **FORMAT**=MR, **OPCODE**=15 01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FMP addr Floating multiply

F*[EA]32→F

Multiply the contents of the floating point register by the contents of addr and place the product in the floating point register, with the mantissa normalized. **MODES**=RV, **FORMAT**=MR, **OPCODE**=16 01, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FRN Round up

If bit 25 of the mantissa in the floating point register is 1, add 1 to bit 24 and clear 25. **MODES**=RV, **FORMAT**=GEN, **OPCODE**=140534, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FSB addr Floating subtract

F-[EA]32→F

Subtract the contents of addr from the floating point register by aligning exponents, and proceding as in FAD except that the [EA]32 is subtracted from the floating point register. **MODES**=RV, **FORMAT**=MR, **OPCODE**=07 01, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FSGT Floating skip if greater than zero

If floating point register is greater than zero, skip next location. **MODES**=RV, **FORMAT**=GEN, **OPCODE**=140515, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FSLE Floating skip if less than or equal to zero

If floating point register is less than or equal to zero, skip next location. **MODES**=RV, **FORMAT**=GEN, **OPCODE**=140514, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FSMI Floating skip if minus

If the floating point register is less than 0, skip next location. **MODES**=RV, **FORMAT**=GEN, **OPCODE**=140512, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FSNZ Floating skip if not zero

If the floating point register is not equal to zero, skip next location. If the floating point register is less than 0, skip next location. **MODES**=RV, **FORMAT**=GEN, **OPCODE**=140511, **C** =unchanged, **L**=unchanged, **CC**=unchanged.

▶ FSPL Floating skip if plus

If the floating point register is greater than 0, skip next location. **MODES**=RV, **FORMAT**= GEN, **OPCODE**=140513, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FST addr Floating store

F→[EA]32

Store the single precision floating point number contained in the floating point register in two memory words starting at addr. Bits 24-31 of the 31 bit mantissa are truncated when written into the 23-bit capacity memory storage. However, the mantissa may be rounded to bit 24 by a FRN instruction which adds 1 to bit 24 if bit 25 is 1. **MODES**=RV, **FORMAT**=MR, **OPCODE**=04 01, **C**=overflow, **L**=carry, **CC**=unchanged.

▶ FSZE Floating skip if zero

If the floating point register is equal to zero, skip next location. **MODES**=RV, **FORMAT**= GEN, **OPCODE**=140510, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ INT Convert float to integer

Int(F)→A|B

Convert the single precision floating point number in the floating point register into a 32 bit integer in register L. The fractional part of the floating point register is lost. If the value in the floating point register is less than -(2**31) or greater than 2**31-1, set the C-bit or initiate a floating exception. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140533, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ INTA Convert float to integer

INT(F)→A

Convert the single precision floating point number in the floating point register into a 16 bit in integer in register A. The fractional part of the floating point register is lost. Overflow occurs if the value in the floating point register is less than -(2**15) or greater than 2**15-1, and sets the C-bit or initiates a floating exception. **MODES**=V, **FORMAT**=GEN, **OPCODE**= 140531, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ INTL Convert float to long integer

INT(F)→L

Convert the single precision floating point number in the floating point register into a 32 bit integer in register L. The fractional part of FAC is lost. If the value in the floating point register is less than -(2**31) or greater than 2**31-1, set the C-bit or initiate a floating exception. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140533, **C**=overflow, **L**=unspecified, **CC**= unspecified.

INTEGER ARITHMETIC—INT

These instructions operate on 16, 31-bit and 32-bit signed integers. See Section 9 for a description of the data formats.

▶ A1A Add one to A

A+1→A

Add 1 to the 16-bit integer in register A and put the result into A. If the number incremented is 2 **15-1, set C and give a result of -2**15; otherwise clear C. **MODES**=SRV, **FORMAT**= GEN, **OPCODE**=141206, **C**=overflow, **L**=carry, **CC**=result.

▶ A2A Add two to A

A+2→A

Add 2 to the 16-bit integer in register A and put the result into A. If the number incremented is 2**15-2 or 2**15-1, set C and give a result of -2**15 or-(2**15-1); otherwise clear C. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140304, **C**=overflow, **L**=carry, **CC**=result.

▶ ACA Add C-bit to A

A+C-bit→A

Add the C-bit to the 16-bit integer in register A and put the result into A (C is treated as same order of magnitude as bit 16 of A). If the number originally in A is 2**15-1 and C set, set C and give a result of -2**15; otherwise clear C. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**= 141216, **C**=overflow, **L**=carry, **CC**=result.

▶ ADD addr Add

A+[EA]16→A

Add the 16-bit integer at addr to the 16-bit integer in register A and put the result into register A. If the sum is greater than 2**15 or less than or equal to -2**15, set C; otherwise, clear C. In the first overflow case, tne result has a minus sign, but a magnitude in positive form equal to the sum minus 2**15; in the second, the result has a plus sign, but a magnitude in negative form equal to the sum plus 2**15. **MODES**=SRV, **FORMAT**=MR, **OPCODE**=06, **C**=overflow, **L**=carry, **CC**=result.

▶ ADL addr Add long

L+[EA]32→L

Add the 32-bit integer at addr to the 32-bit integer in register L and put the result into L. If the sum is greater than 2**31 or less than -2**31, set C; otherwise, clear C. In the first overflow case, the result has a minus sign, but a magnitude in positive form equal to the sum minus 2**31; in the second, the result has a plus sign, but a magnitude in negative form equal to the sum plus 2**31. **MODES**=V, **FORMAT**=MR, **OPCODE**=06 03, **C**=overflow, **L**=carry, **CC**=result.

► ADLL Add L bit to L

L+keys(L)→L

Add the link bit (L-bit in the keys) to the contents of the L register and put the result into the L register. Overflow may be set.

This instruction is useful in implementing multiple precision arithmetic. **MODES**=V, **FORMAT**=GEN, **OPCODE**=141000, **C**=overflow, **L**=carry, **CC**=result.

► CAS addr Compare A and skip

If A>[EA]16 then PC=PC
If A=[EA]16 then PC+1→PC
If A<[EA]16 then PC+2→PC

If the contents of the A register is greater than the contents of addr, execute the next instruction.

If the contents of the A register equals the contents of addr, skip the next location in instruction sequence and execute the instruction at the second location following.

If the contents of the A register is less than the contents of addr, skip the next two locations in instruction sequence and execute the instruction at the third location following. **MODES** =SRV, **FORMAT**=MR, **OPCODE**=11, **C**=unchanged, **L**=carry, **CC**=result.

► CAZ Compare A with zero

If A>0 then PC=PC
If A=0 then PC+1→)PC
If A<0 then PC+2→PC

If the contents of the A register is greater than zero, execute the next instruction.

If the contents of the A register is equal to zero, skip the next location in instruction sequence and execute the instruction at second location following.

If the contents of the A register is less than zero, skip the next location in instruction sequence and execute the instruction at third location following. **MODES**=SRV, **FORMAT**= GEN, **OPCODE**=140214, **C**=unchanged, **L**=carry, **CC**=result.

► CHS Change sign

-A(1)→A(1)

Complement bit 1 of register A without affecting the rest of the register. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140024, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

► CLS addr Compare

If L>[EA]32 then PB+1→PB
If L=[EA]32 then PB+2→PB
If L<[EA]32 then PB+3→PB

If the contents of the L register is greater than the contents of addr, execute the next instruction.

If the contents of the L register equals the contents of addr, skip the next location in instruction sequence and execute the instruction at second location following.

If the contents of the L register is less than the contents of addr, skip next two locations in instruction sequence and execute the instruction at third location following. **MODES**=V, **FORMAT**=MR, **OPCODE**=11 03, **C**=unchanged, **L**=carry, **CC**=result.

▶ CSA Copy sign of A

A(1)→C-bit;0→A(1)

Make C equal to bit 1 of register A and clear bit 1 of A without affecting the rest of the register. Used when using single precision arithmetic to do double precision work. **MODES** =SRV, **FORMAT**=GEN, **OPCODE**=140320, **C**=result, **L**=unspecified, **CC**=unchanged.

▶ DAD addr Double add

A|B+[EA]31→A|B

Add the 31-bit integer at addr and addr+1 to the 31-bit integer in registers A|B, and put the result into A|B. If the sum is ⩾2**30 or <-2**30, set C; otherwise, clear C. In the first overflow case, the result has a minus sign but a magnitude in positive form equal to the sum minus 2**30; in the second, the result has a plus sign but a magnitude in negative form equal to the sum plus 2**30.

By definition, bit 1 of the low order word of a 31-bit integer must be 0. The instruction executes only in double precision mode. **MODES**=SR, **FORMAT**=MR, **OPCODE**=06, **C**= overflow, **L**=carry, **CC**=result.

▶ DBL Enter double precision mode

Enter double precision mode. Subsequent LDA, STA ADD and SUB instructions handle 31-bit integers. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000007, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ DIV addr Divide

A|B/[EA]16→A;REM→B

Divide the 31-bit integer in register A|B by the 16-bit integer at addr and put the quotient into A, and the remainder into B. Barring overflow, the results are defined such that A*[addr]+ B equals the original A|B and the remainder in B has the same sign as the dividend. Hence, -42 divided by 5 gives A=-8 and B=-2. Overflow occurs (and the C-bit is set) whenever the quotient is less than -(2**15)or greater than 2**15-1; The A|B register is unchanged. **MODES** =SR, **FORMAT**=MR, **OPCODE**=17, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DIV addr Divide

L/[EA]16→A;REM→B

Divide the 32-bit integer in register L by the 16-bit integer at addr and put the quotient into A, and the remainder into B. Barring overflow, the results are defined such that A*[addr]+ B equals the original L and the remainder in B has the same sign as the dividend. Hence, -42 divided by 5 gives A=-8 and B=-2.

Overflow occurs (and the C-bit is set) whenever the quotient is less than -(2**15) or greater than 2**15-1. The PIDA instruction is useful for placing 16-bit dividends into L. **MODES**=V, **FORMAT**=MR, **OPCODE**=17, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DSB addr Double subtract

A|B-[EA]31→A|B

Subtract the 31-bit integer at addr and addr+1 from the 31-bit integer in registers A|B, and place the result into A|B. If the difference is ⩾2**30 or <-2**30, set C; otherwise, clear C. In the first overflow case, the result has a minus sign but a magnitude in positive form equal to the difference minus 2**30; in the second, the result has a plus sign but a magnitude in negative form equal to the difference plus 2**30.

Bit 1 of the low order word of a 31-bit integer must be 0. The instruction executes only in double precision mode. To negate one 31-bit integer, simply subtract it from zero. **MODES** =SR, **FORMAT**=MR, **OPCODE**=07, **C**=overflow, **L**=carry, **CC**=result.

▶ DVL addr Divide long

L|E/[EA]32→L;REM→E

Divide the 64-bit integer in registers L|E by the 32-bit integer at addr and put the quotient into L, and the remainder into E. Barring overflow, the results are defined such that L*[addr]+ E equals the original L|E and the remainder in E has the same sign as the dividend. Hence, +42 divided by -5 gives L=-8 and E=+2.

Overflow occurs (and the C-bit is set) whenever the quotient is less than -(2**31) or greater than 2**31-1. **MODES**=V, **FORMAT**=MR, **OPCODE**=17 03, **C**=overflow, **L**=unspecified, **CC** =unspecified.

▶ MPL addr Multiply long

L*[EA]32→L|E

Multiply the 32-bit integer in register L by the 32-bit integer at addr, and put the 64-bit integer result into L|E. This operation never overflows because there is always room for the product. **MODES**=V, **FORMAT**=MR, **OPCODE**=16 03, **C**=cleared, **L**=unspecified, **CC**= unchanged.

▶ MPY addr Multiply

A*[EA]16→L

Multiply the 16-bit integer in register A by the 16-bit integer at addr, and put the 32-bit integer result into L. This operation never overflows because there is always room for the product. **MODES**=V, **FORMAT**=MR, **OPCODE**=16, **C**=cleared, **L**=unspecified, **CC**=unchanged.

▶ MPY addr Multiply

A*[EA]16→A|B

Multiply the 16-bit integer in register A by the 16-bit integer at addr, and put the 31-bit integer result into registers A and B. If both the multiplier and multiplicand are -2**15 then set C; otherwise clear C. **MODES**=SR, **FORMAT**=MR, **OPCODE**=16, **C**=cleared, **L**=unspecified, **CC**=unchanged.

▶ NRM Normalize

A1 A2. . .A16 B1 B2. . .B16

Shift the 31-bit integer in registers A and B left arithmetically, bringing zeros into bit 16 of B, bypassing bit 1 of B, leaving bit 1 of register A unaffected, and dropping bits out of bit 2 of register A until bit 2 of register A is in the state opposite that bit 1 of register A. Since the only data shifted out of bit 2 of register A is equal to the sign, no information is lost. Place the number of shifts performed in bits 9-16 of the keys. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000101, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PID Position for integer divide

A(2–16)→B(2–16);0→B(1);A(1)→A(2–16)

Convert the 16-bit integer in register A to a 31-bit integer in A|B by moving the contents of bits 2–16 of register A to bits 2–16 of register B, clearing bit 1 of register B and extending the sign in bit-1 of A through bits 2–16 of A. Intended to allow division of 16-bit A by 16-bit [addr] resulting in two 16-bit integers. 16 in A to 31 in A|B simplifies integer arithmetic. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000211, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PIDA Position for integer divide

A(1–16)→L(17–32);A(1)→A(2–16)

Convert the 16-bit integer in register A to a 32-bit integer in register L by moving bits 1–16 of A to bits 17–32 of L and extending the sign in bit 1 of A through bits 2–16 of A. **MODES**=V, **FORMAT**=GEN, **OPCODE**=000115, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PIDL Position for integer divide-long

L→E;L(1)→L(2–32)

Convert the 32-bit integer in register L to a 64-bit integer in registers L and E by moving the contents of L to E and extending the sign in bit 1 of L through bits 2–32 of L. PIDL is useful for placing 32 bit operands in L|E. **MODES**=V, **FORMAT**=GEN, **OPCODE**=000305, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PIM Position following integer multiply

B(2–16)→A(2–16)

Convert the 31-bit integer in registers A|B to a 16-bit integer in A by moving bits 2–16 of B into bits 2–16 of A. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000205, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PIMA Position following integer multiply

L(17–31)→A(1–16)

Convert the 32-bit integer in L to a 16-bit integer in register A by moving bits 17–32 of L into bits 1–16 of A. Overflow if a loss of precision would result. **MODES**=V, **FORMAT**=GEN, **OPCODE**=000015, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ PIML Position following integer multiply-long

L|E(33–64)→L(1–32)

Convert the 64-bit integer in registers L|E to a 32-bit integer in L by moving bits 33–64 of register L|E into bits 1–32 of register L. Overflow if a loss of precision would result. **MODES**=V, **FORMAT**=GEN, **OPCODE**=000301, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ S1A Subtract one from A

A–1→A

Subtract 1 from the 16-bit integer in register A and put the result into A. If the number decremented is -2**15, set C and give a result of 2**15–1; otherwise clear C. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140110, **C**=overflow, **L**=carry, **CC**=result.

▶ S2A Subtract two from A

A-2→A

Subtract 2 from the 16-bit integer in register A and put the result into A. If the number decremented is -(2**15-1) or -2**15, set C and give a result of 2**15-1; otherwise clear C. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140310, **C**=overflow, **L**=carry, **CC**=result.

▶ SBL addr Subtract long

L-[EA]32→L

Subtract the 32-bit integer at addr from the 32-bit integer in register L and put the result into the L register. If the difference is greater than +2**31 or less than -2**31, set C; otherwise clear C. In the first overflow case, the result has a minus sign but a magnitude in positive form equal to the difference minus 2**31; in the second, the result has a plus sign but a magnitude in negative form equal to the difference plus 2**31. **MODES**=V, **FORMAT**=MR, **OPCODE**=07 03, **C**=overflow, **L**=carry, **CC**=result.

▶ SCA Load shift count into A

keys(9-16)→A(9-16);0→A(1-8)

Load the contents of bits 9-16 of the keys into bits 9-16 of register A and clear bits 1-8 of register A. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000041, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ SGL Enter single precision mode

Return to single precision mode. Subsequent LDA, STA, ADD and SUB instructions handle 16-bit integers. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000005, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ SSM Set sign minus

1→A(1)

Set bit 1 of register A to one without affecting the rest of the register. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140500, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ SSP Set sign plus

0→A(1)

Clear bit 1 of register A without affecting the rest of the register. **MODES**=SRV, **FORMAT** =GEN, **OPCODE**=140100, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ SUB addr Subtract

A-[EA]16→A

Subtract the 16-bit integer at addr from the 16-bit integer in register A and put the result into register A. If the difference is >2**15 or <-2**15, set C; otherwise clear C. In the first overflow case, the result has a minus sign but a magnitude in positive form equal to the difference minus 2**15; in the second, the result has a plus sign but a magnitude in negative form equal to the difference plus 2**15. **MODES**=SRV, **FORMAT**=MR, **OPCODE**=07, **C**=overflow, **L**=carry, **CC**=result.

▶ TCA Two's complement A

-A→A

Form the two's complement of the contents of register A and put the result into register A. If the number is -2**15, set C and give a result of -2**15; otherwise clear C. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140407, **C**=overflow, **L**=carry, **CC**=result.

▶ TCL Two's complement long

-L→L

Form the two's complement of the contents of register L and put the result into L. If the result is -2**31, set C and give a result of -2**31; otherwise clear C. **MODES**=V, **FORMAT**=GEN, **OPCODE**=141210, **C**=overflow, **L**=carry, **CC**=result.

INTEGRITY CHECK FOR HARDWARE—INTGY

▶ EMCM Enter machine check mode

In machine check mode the microprogram responds to a machine parity error by causing a machine check interrupt if there is a non-zero vector in the interrupt location. If this location is zero the machine halts. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000503, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ LMCM Leave machine check mode

A machine parity error sets the machine check flag, but does not cause a check (V-mode) or generate an interrupt (SR-mode). **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000501, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ MDEI Memory diagnostic enable interleave

Enable the memory interleave capability. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001304, **C** =unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ MDII Memory diagnostic inhibit interleave

Inhibit the memory diagnostic interleave capability. **MODES**=V, **FORMAT**=GEN, **OPCODE** =001305, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ MDIW Memory diagnostic write interleaved

Write interleaved memory. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001324, **C**=unchanged, **L** =unchanged, **CC**=unchanged. *Restricted instruction.*

▶ MDRS Memory diagnostic read syndrome bits

Read memory syndrome bits. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001306, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ MDWC Memory diagnostic write control register

Write memory control register. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001307, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ RMC Clear machine check

Clear the machine check flag. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000021, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ SMCR Skip on machine check reset

If the machine check flag is zero (indicating no machine detected parity error), skip the next instruction in sequence. (When the processor is in machine check mode, this instruction has no meaning and executes as a skip). **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=100200, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ SMCS Skip on machine check set

If the machine check flag is set (indicating a machine detected parity error), skip the next instruction in sequence. (When the processor is in machine check mode, this instruction has no meaning and executes as a NOP). **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=101200, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ VIRY Verify

Execute the verification routine, and if there is a failure of any kind, go on to the next instruction with the number of the test that failed in register A. If there are no errors, skip the next instruction in sequence.

If the processor does not have the verification routine, this instruction executes as no-op. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000311, **C**=unspecified, **L**=unspecified, **CC**=unspecified. *Restricted instruction.*

▶ XVRY Verify the XIS board (Prime 500)

Executes a Prime 500 microcode diagnostic routine that checks the integrity of the XIS board. If the XIS board is not functional, the machine will not skip the next instruction and the A register will hold the failed micro-diagnostic test number. If the machine passes the verify instruction, the next instruction is skipped. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001113, **C**=unspecified, **L**=unspecified, **CC**=unspecified. *Restricted instruction.*

The codes and tests are:

'72	Data Move Test—Load and Unload XIS Board
'73	Normalize Test—Adjust Test
'74	Binary Multiply
'75	Binary Divide
'76	Decimal Arithmetic

INPUT/OUTPUT—I/O

▶ CAI Clear active interrupt

Terminate the presently active interrupt so that the processor can recognize interrupt requests from devices of lower priority (in higher slots) than the device for which the current interrupt is being held. This instruction is effective only in vectored interrupt mode. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000411, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ EIO addr Execute I/O

Perform the I/O instruction represented by the effective address, e.g., X='04 EIO '131000,X will execute an INA with FUNC = '10 and DEV = '04. EQ = successful INA, OTA or SKS, NE = unsuccessful INA, OTA or SKS. OCP always successful, sets NE. **MODES**=V, **FORMAT**=MR, **OPCODE**=14 01, **C**=unchanged, **L**=unchanged, **CC**=result. *Restricted instruction.*

▶ ENB Enable interrupt

Enable the external interrupt system so the processor will respond to interrupt requests over the I/O bus. This instruction takes effect following execution of the next sequential instruction. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000401, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ ESIM Enter standard interrupt mode

Enter standard interrupt mode so that all interrupts are made through location '63. **MODES** =SRV, **FORMAT**=GEN, **OPCODE**=000415, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ EVIM Enter vectored interrupt mode

Enter vectored interrupt mode so that the interrupt priority of a device is determined by its position on the I/O bus (with lower devices having higher priority) and each interrupt is made through the location specified by the interrupting device. **MODES**=SRV, **FORMAT**= GEN, **OPCODE**=000417, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ INA FUNC|DEV Input to A

Input data from device DEV into register A. FUNC determines the type of data.If the device does not respond ready, then do not perform the transfer, but execute the next instruction in sequence. If the device responds ready, then perform the transfer specified by FUNC and skip the next instruction in sequence. To perform the function specified by FUNC, the processor reads the information from DEV into register A and performs whatever control operations are appropriate to the function and the device. Depending on FUNC, the information read may be data, status, an address, a word count, or anything else.

The number of bits brought into register A depends on the type of information, the size of the device register, the mode of operation, etc.

INA instructions for any device except device '20 use a ready test and skip the next instruction if the device was ready. **MODES**=SR, **FORMAT**=PIO, **OPCODE**=54, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ INH Inhibit interrupts

Inhibit the external interrupt system so the processor will not respond to interrupt requests over the I/O bus. This instruction takes effect immediately. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=001001, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ OCP FUNC|DEV Output control pulse

Send a control pulse for the function specified by FUNC (bits 7–10) to the device specified by DEV (bits 11–16). This instruction never skips and is used for such functions as initializing a disk controller, or starting a transfer. **MODES**=SR, **FORMAT**=PIO, **OPCODE**=14, **C**= unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ OTA FUNC|DEV Out from A

Transfer data from register A to DEV. FUNC tells the device which operation to perform. If the device does not respond ready, then do not perform the transfer but instead execute the next instruction in sequence. If the device responds ready, then perform the transfer and skip the next instruction in sequence. The processor sends the contents of register A to DEV which performs whatever control operations are appropriate to the function and the device.

The number of bits actually accepted by the device depends on the type of information, the size of the device register, the mode of operation, etc. The contents of register A are unaffected.

An OTA instruction for any device except device '20 uses a ready test and the skipping procedure as stated in the description of the instruction. An OTA to device '20 makes no test and does not skip. **MODES**=SR, **FORMAT**=PIO, **OPCODE**=74, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restriction instruction.*

▶ SKS FUNC|DEV Skip if satisfied

FUNC (bits 7–10) defines a condition to be tested by the SKS. When the condition is satisfied, the device specified by DEV (bits 11–16) responds ready, and the next instruction in sequence is skipped. **MODES**=SR, **FORMAT**=PIO, **OPCODE**=34, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restriction instruction.*

KEY MANIPULATION—KEYS

See Section 9 for the format of the keys.

▶ INK Input keys

Read the keys into register A. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000043, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ OTK Output keys

A→keys

Set up the keys from the contents of register A. Each bit position in register A corresponds to the bit position in the keys, e.g., bit 1 of register A becomes the C-bit in the keys. **MODES** =SR, **FORMAT**=GEN, **OPCODE**=000405, **C**=loaded by instruction, **L**=loaded by instruction, **CC**=loaded by instruction.

▶ RCB Reset C-bit

0→C

Clear the C-bit in the keys. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140200, **C**=cleared, **L** =unspecified, **CC**=unchanged.

▶ SCB Set C-bit

1→C

Set the C-bit in the keys. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140600, **C**=set, **L**= unspecified, **CC**=unchanged.

▶ TAK Transfer A to keys

A→keys

Transfer the contents of register A to the keys register. If the new value of the keys specifies a different addressing mode, note that the new mode takes effect on the next instruction. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001015, **C**=loaded by instruction, **L**=loaded by instruction, **CC**=loaded by instruction.

▶ TKA Transfer keys to A

keys→A

Transfer the contents of the keys register to register A. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001005, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

LOGICAL OPERATIONS—LOGIC

▶ ANA addr AND to A

A.AND.[EA]16→A

AND the contents of location addr with the contents of register A and place the result in register A. A given bit of the result is 1 if the corresponding bits of both operands are 1; otherwise the resulting bit is 0.

A BIT	Memory Bit	Resulting Bit
0	0	0
0	1	0
1	0	0
1	1	1

MODES=SRV, **FORMAT**=MR, **OPCODE**=03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ANL addr AND long

L.AND.[EA]32→L

AND the contents of register L with the 32-bit quantity at addr, putting the result in L. **MODES**=V, **FORMAT**=MR, **OPCODE**=03 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CMA Complement A

NOT. A→A

Form the ones complement of the contents of register A and put the result in register A. Each one becomes a zero; each zero becomes a one. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**= 140401, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ERA addr Exclusive OR to A

A.XOR.[EA]16→A

EXCLUSIVE OR the contents of location addr with the contents of register A and place the result in register A. A given bit of the result is 1 if the corresponding bits of the operands differ; otherwise the resulting bit is 0.

A BIT	Memory Bit	Resulting Bit
0	0	0
0	1	1
1	0	1
1	1	0

MODES=SRV, **FORMAT**=MR, **OPCODE**=05, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ERL addr Exclusive OR long

L.XOR.[EA]32→L

EXCLUSIVE OR the contents of register L with the 32-bit quantity at addr, putting the result in L. **MODES**=V, **FORMAT**=MR, **OPCODE**=05 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ORA Inclusive OR

A.OR.[EA]16→)A

INCLUSIVE OR the contents of register A with the 16-bit quantity at addr, putting the result in A. **MODES**=V, **FORMAT**=MR, **OPCODE**=03 02, **C**=unchanged, **L**=unchanged, **CC**= unchanged.

LOGICAL TEST AND SET—LTSTS

If the test is satisfied, then set the A register equal to 1. If the test is not satisfied, then set the register equal to 0. These instructions simplify the analysis of complex logical expressions.

If { A Register (Blank) / Condition Code (C) / L Register (L) / Floating Point (F) Register } { LT / LE / EQ / NE / GE / GT } 0, then 1→A; else 0→A

▶ A-Register test

Mnemonic	Function	Opcode
LLT	If A < 0, then 1→A; else 0→A	140410
LLE	If A ⩽ 0, then 1→A; else 0→A	140411
LEQ	If A = 0, then 1→A; else 0→A	140413
LNE	If A ≠ 0, then 1→A; else 0→A	140412
LGE	If A ⩾ 0, then 1→A; else 0→A	140414
LGT	If A > 0, then 1→A; else 0→A	140415

MODES=SRV, **FORMAT**=GEN, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ Condition code test

Mnemonic	Function	Opcode
LCLT	If CC < 0, then 1→A; else 0→A	141500
LCLE	If CC ⩽ 0, then 1→A; else 0→A	141501
LCEQ	If CC = 0, then 1→A; else 0→A	141503
LCNE	If CC ≠ 0, then 1→A; else 0→A	141502
LCGE	If CC ⩾ 0, then 1→A; else 0→A	141504
LCGT	If CC > 0, then 1→A; else 0→A	141505

MODES=V, **FORMAT**=GEN, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ L register test

Mnemonic	Function	Opcode
LLLT	If L < 0, then 1→A; else 0→A	140410
LLLE	If L ⩽ 0, then 1→A; else 0→A	141511
LLEQ	If L = 0, then 1→A; else 0→A	141513
LLNE	If L ≠ 0, then 1→A; else 0→A	141512
LLGE	If L ⩾ 0, then 1→A; else 0→A	140414
LLGT	If L > 0, then 1→A; else 0→A	141515

MODES=V, **FORMAT**=GEN, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ **Floating register test**

Mnemonic	**Function**	**Opcode**
LFLT	If F < 0, then 1→A; else 0→A	141110
LFLE	If F ⩽ 0, then 1→A; else 0→A	141111
LFEQ	If F = 0, then 1→A; else 0→A	141113
LFNE	If F ≠ 0, then 1→A; else 0→A	141112
LFGE	If F ⩾ 0, then 1→A; else 0→A	141114
LFGT	If F > 0, then 1→A; else 0→A	141115

MODES=V, **FORMAT**=GEN, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ **LT Logic set A true**

1→A

Set A equal to one. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140417, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ **LF Logic set A false**

0→A

Set A equal to zero. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140416, **C**=unchanged, **L**=unchanged, **CC**=result.

MACHINE CONTROL—MCTL

▶ **CXCS Control extended control store**

Move the A register to control register on writable control store board. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001714, **C**=unspecified, **L**=unspecified, **CC**=unspecified. *Restricted instruction.*

▶ **EPMJ addr Enter paging mode and jump (Prime 300)**

EA→PC

EPMJ is a two-word instruction. The first word is the opcode; the second word contains a 16-bit address pointing to the final effective address which is transferred to the program counter; the associative memory registers are cleared, and paging mode is enabled. **MODES** =SR, **FORMAT**=MR, **OPCODE**=000217, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ **EPMX addr Enter paging mode and jump to XCS (Prime 300)**

EA→PC

EPMX is a two-word instruction. The first word is the opcode; the second word contains a 16-bit pointer to the location of the micro-instruction. Paging is enabled. **MODES**=SR, **FORMAT**=MR, **OPCODE**=000237, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ **ERMJ addr Enter restricted execution mode and jump (Prime 300)**

EA→PC

ERMJ is a two-word instruction. The first word is the opcode; the second word contains a 16-bit address pointing to the final effective address which is transferred to the program

counter; restricted execution mode is enabled, and interrupts are enabled. **MODES**=SR, **FORMAT**=MR, **OPCODE**=000701, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ ERMX addr Enter restricted execution mode and jump to XCS (Prime 300)

EA→PC

ERMX is a two-word instruction. The first is the opcode. The second word contains a 16-bit pointer to the location of the micro-instruction. Restricted execution mode and interrupts are enabled. **MODES**=SR, **FORMAT**=MR, **OPCODE**=000721, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ EVMJ addr Enter virtual mode and jump (Prime 300)

EA→PC

EVMJ is a two-word instruction. The first word, which has the effect of an EPMJ and ERMJ combined, is the opcode; the second word contains a 16-bit address pointing to the final effective address which is transferred to the program counter. Paging, interrupts, and restricted execution mode are enabled. **MODES**=SR, **FORMAT**=MR, **OPCODE**=000703, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ EVMX addr Enter virtual mode and jump (Prime 300)

EA→PC

EVMX is a two-word instruction. The first word, which has the effect of an EPMX and ERMX combined, is the opcode; the second word contains a 16-bit pointer to the location of the micro-instruction. Paging, interrupts, and restricted execution mode are enabled. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000000, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ HLT Halt

Halt the processor with the STOP indicator lit on the control panel and the program counter pointing to the next instruction in sequence (the instruction that would have been executed had the HLT been replaced by a no-op). The data lights display the next instruction. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000000, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ ITLB Invalidate STLB entry

Invalidate the Segmentation Translation Lookaside Buffer (STLB) entry whose address is in L. This instruction must be executed whenever the page table entry for the given address is changed.

If a Segment Descriptor Word (SDW) or a Descriptor Table Address Register (DTAR) is changed, usually the entire STLB must be invalidated. This can be done by executing ITLB once for each page of any single segment (except segment 0).

If the segment number portion of L is zero, the I/O TLB entry corresponding to address L is invalidated. **MODES**=V, **FORMAT**=GEN, **OPCODE**=000615, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ LIOT addr Load I/O TLB

Load the I/O Translation Lookaside Buffer with the following information:

1. Virtual address (VA) in segment 0. This is provided by the effective

address computed from the address pointer, addr.

2. Physical address (PA) which is the translation of the virtual address. This is obtained by the processor from segment 0. If the fault bit is set, a page fault will be generated.
3. Target virtual address (TVA) which is the segment number and page number of the virtual address that will be used by procedures accessing this information. This will be used to help invalidate the proper locations in the cache. This is provided in the L register as a virtual address. The low order 10 bits (word number in page) and the segment number are ignored.

Summary:

Information	**Source**
VA	AP
PA	Segment number page table
TVA	L

MODES=V, **FORMAT**=AP, **OPCODE**=000044, **C**=unspecified, **L**=unspecified, **CC**=unspecified. *Restricted instruction.*

▶ LPID Load process ID

A→RPID

Load the process id register from bits 1-12 of Register A. **MODES**=V, **FORMAT**=GEN, **OPCODE**=000617, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ LPMJ addr Leave paging mode and jump (Prime 300)

EA→PC

LPMJ is a two-word instruction. The first word is the opcode; the second word contains a 16-bit address pointing to the final effective address which is transferred to the program counter. Paging mode is disabled. **MODES**=SR, **FORMAT**=MR, **OPCODE**=000215, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ LPMX addr Leave paging mode and jump to XCS (Prime 300)

EA→PC

LPMX is a two-word instruction. The first word is the opcode. The second word contains a 16-bit pointer to the location of the micro-instruction. Paging is disabled. **MODES**=SR, **FORMAT**=MR, **OPCODE**=000235, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LPSW addr Load program status word

Load Program Status Word is a restricted operation which can change the status of the processor. It can be executed only in ring zero. The instruction addresses a four-word block at location addr containing a program counter (ring, segment, and word numbers) in the first two words, keys in the third word and modals in the fourth. The program counter and keys of the running process are loaded from the first three words, then the processor modals are loaded from the fourth. If the new keys have the in-dispatcher bit (bit 16) off, the current process continues in execution but at a location defined by the new program counter. If the new keys have the in-dispatcher bit on, the dispatcher is entered to dispatch the highest priority ready process. Whenever the current process again becomes the highest priority ready process, it will then resume execution at the point defined by its new program counter. The modals are associated with the processor and not the process, so in either case, the new modals are effective immediately.

This instruction is used to load the four words of the register set which cannot be correctly loaded with the STLR instruction: the program counter (ring, segment, and word number), the keys, and the modals. The STLR instruction should not be used to set these words, as it does not update the separate hardware registers in which the processor maintains duplicate information to achieve higher performance.

The LPSW instruction must never attempt to change the current-register-set bits of the modals (bits 9–11). This implies that, unless for some reason the current register set in effect for the execution of the program is known with certainty, any program wishing to execute an LPSW must inhibit interrupts (to prevent an unexpected process and register exchange), read the register set currently in effect from the present modals (as with an LDLR '24), mask those register-set bits into the modals to be loaded, and then finally execute the LPSW. Fortunately, in both usual applications of LPSW the needed register-set bits are predictable: when LPSW is first used after Master Clear to turn on process-exchange mode, the current-register-set bits should be 010 (the processor is always initialized to register set 2); and when LPSW is used to return from a fault, check, or interrupt handled by inhibited code, whatever register-set bits were stored away by the fault, check, or interrupt are still correct and can simply be reloaded.

Similarly, except to load status correctly stored on a fault, check, or interrrupt, and LPSW should never attempt to set either the save-done bit (bit 15) or the in-dispatcher bit (bit 16) of the keys. The initial LPSW following a Master Clear should have both these bits off. **MODES**=V, **FORMAT**=AP, **OPCODE**=000711, **C**=loaded by instruction, **L**=loaded by instruction, **CC**=loaded by instruction. *Restricted instruction.*

▶ LWCS Load writable control store

Load writable control store portion of extended control store board from the memory block pointed to by XB. The control register loaded by CXCS modifies this instruction. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001710, **C**=unspecified, **L**=unspecified, **CC**=unspecified. *Restricted instruction.*

▶ MIA addr Microcode indirect A

Microcode entrance. **MODES**=V, **FORMAT**=MR, **OPCODE**=12 01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ MIB addr Microcode indirect B

Microcode entrance. **MODES**=V, **FORMAT**=MR, **OPCODE**=13 01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ NOP No operation

$$PC+1 \rightarrow PC$$

Do nothing, but go on to the next instruction. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000001, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PTLB Purge TLB

Purge either the entire non I/O Translation Lookaside Buffer (TLB) or a specified physical page. The physcial page number is provided right-justified in the L register. The high-order bit of L is set to indicate a complete purge. **MODES**=V, **FORMAT**=GEN, **OPCODE**=000064, **C**=unspecified, **L**=unspecified, **CC**=unspecified. *Restricted instruction.*

▶ RRST addr Restore registers

Restore the general, floating and XB registers from the save area starting at location addr. The format of the save area is as for RSAV below. **MODES**=V, **FORMAT**=AP, **OPCODE**= 000717, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ RSAV addr Save registers

Save the general, floating and XB registers in the save area starting at location addr. Only those general and floating point registers which are not zero are saved. A save mask is generated which identifies the registers which are not zero. With the exception of XB, which is always saved, registers which are zero are not stored into the save area; their location remains untouched.

The format of the RSAV area is:

Word	Contents
1	Save Mask
2–5	FALR1 (FAC)
6–9	FALR 0
10	X
11–13	—
14	Y,S
15–17	—
18–19	E
20	A,LH
21	B,LL
22–25	—
26–27	XB

Save Mask:

MUST BE ZERO	FALR 1, FAC	FALR 0	X		Y, S		E	L, B, A	
1 — 4	5 — 6	7 — 8	9	10	11	12	13	14	15-16

The size of the RSAV area is 27 words. **MODES**=V, **FORMAT**=AP, **OPCODE**=000715, **C**= unchanged, **L**=unchanged, **CC**=unchanged.

▶ STPM Store processor model number

Store the CPU model number and microcode revision number in an eight-word field pointed to by the temporary base register (XB).

The first 32-bit field will define the processor model number. This field will not be modified on any P400 or P500 since STPM executes as a SGL which is similar to a NOP. Thus the program is required to put a zero into this field prior to executing the STPM. The following long-integer codes are assigned:

0L	P400, P500
3L	P350
4L	P450
5L	P550
6L	P650
7L	P750

The second 32-bit field will define the microcode revision. It will be a unique number that changes at each revision. The P400 and P500 do not implement this. The remaining 64-bits are reserved for future expansion.

The recommended sequence for the STPM is:

EAXB	Memory Buffer
CRL	
STL	XB%
STPM	

MODES=V, **FORMAT**=GEN, **OPCODE**=000024, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ SVC Supervisor call

An addressing mode independent method of making an operating system request. It is also independent of operating system. The call protocol is such that an operation code (request) followed by argument pointers (the 16-bit word number—on the Prime 400/500, segment number is the segment in which the SVC resides) is made available to the operating system. PRIMOS has defined a uniform set of operation codes to provide operating system independent services.

Note

On the Prime 100-300 (and on the segmented CPU's in non process exchange mode), the SVC is treated as an interrupt. On the segmented CPU's, the SVC is treated as a fault with offset '14.

MODES=SRV, **FORMAT**=GEN, **OPCODE**=000505, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ WCS Writable control store

Reserved set of 64 op codes to serve as microcode entrances.
MODES=RV, **FORMAT**=GEN, **OPCODE**=0016xx, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

MOVE DATA—MOVE

▶ DLD addr Double load

$$[EA]32 \rightarrow A|B$$

Load the contents of location addr into register A and the contents of location addr+1 into register B. This instruction executes only in double precision mode. **MODES**=SR, **FORMAT** =MR, **OPCODE**=02, **C**=unchanged, **L**=unchanged. **CC**=unchanged.

▶ DST addr Double store

$$A|B \rightarrow [EA]32$$

Store the contents of register A in location addr and the contents of register B in location addr+1. This instruction executes only in double precision mode. **MODES**=SR, **FORMAT**= MR, **OPCODE**=04, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ IAB Interchange the A and B registers

A⇔B

Move the contents of register A to register B and the contents of register B to register A. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=000201, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ICA Interchange characters in A

A(1–8) ⇔ A(9–16)

Move the contents of register A bits 1–8 to bits 9–16 and the contents of bits 9–16 to bits 1–8. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=141340, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ICL Interchange and clear left

A(1–8)→A(9–16);0→A(1–8)

Move the contents of register A bits 1–8 to bits 9–16 and then clear the left byte (bits 1–8). **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=141140, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ICR Interchange and clear right

A(9–16)→A(1–8);0→A(9–16)

Move the contents of register A bits 9–16 to bits 1–8 and clear the right byte (bits 9–16). The original contents of bits 1–8 are lost. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=141240, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ILE Interchange L and E

L⇔E

Move the contents of register L to register E and the contents of register E to register L. **MODES**=V, **FORMAT**=GEN, **OPCODE**=141414, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ IMA addr Interchange memory and the A register

[EA]16⇔A

Store the contents of the A register in location addr and load the original contents of location addr into the A register. **MODES**=SRV, **FORMAT**=MR, **OPCODE**=13, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LDA addr Load the A register

[EA]16→A

Load the contents of location addr into the A register. **MODES**=SRV, **FORMAT**=MR, **OPCODE**=02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LDL addr Load long

[EA]32→L

Move the 32-bit quantity at location addr to register L. **MODES**=V, **FORMAT**=MR, **OPCODE**=02 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LDLR addr Load L from addressed register

register(EA)W→L

Copy the contents of the register specified by the word number portion of addr into L. There are three cases of this instruction which are summarized below. Only the word portion of the effective address, (EA)W, is used.

Bit 2 of (EA)W = 1; Ignore Bits 1 and 3-9: (EA)W(10-16) - Absolute register number from 0-'177. Restricted instruction.

Bit 2 of (EA)W = 0: (EA)W(13-16) - Register 20-'37 in the current register set. Restricted instruction.

Bit 12 of (EA)W = 0: (EA)W(13-16) - Register 0-'17 in the current register set.

MODES=V, **FORMAT**=MR, **OPCODE**=05 01, **C**=unchanged. **L**=unchanged, **CC**=unchanged.

▶ LDX addr Load X

[EA]16→X

Load the contents of location addr into the X register. The contents of addr are unaffected, the previous contents of the X register are lost. This instruction cannot itself specify indexing, although an address word retrieved in the effective address calculation may do so in 16S mode. **MODES**=SRV, **FORMAT**=MR, **OPCODE**=35, **C**=unchanged, **L**=unchanged, **CC** =unchanged.

▶ LDY addr Load Y

[EA]32→Y

Move the 16 bit quantity at location addr to register Y. Cannot be indexed. **MODES**=V, **FORMAT**=MR, **OPCODE**=35 01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ STA addr Store the A register

A→[EA]16

Store the contents of the A register in location addr. The contents of the A register are unaffected; the previous contents of addr are lost. **MODES**=SRV, **FORMAT**=MR, **OPCODE** =04, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ STAC addr Store A conditionally

If [EA]16=B then A→[EA]16

Store the contents of A into location addr, if and only if, the contents of location addr equals the contents of B.

The comparison and store are guaranteed not to be separated by the execution of any other instructions. That is, it is not possible for any other instruction to change the contents of the addressed memory word after the comparison has been made but before the store takes place. The condition-code bits are set "equal" if the store takes place, otherwise "unequal". **MODES**=V, **FORMAT**=AP, **OPCODE**=001200, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ STL addr Store long

L→[EA]32

Store the contents of register L into the 32-bit long word at location addr. **MODES**=V, **FORMAT**=MR, **OPCODE**=04 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ STLC addr Store L conditionally

If [EA]32=E then L→[EA]32

Store the contents of L into the 32-bit location at addr if and only if the contents of location addr equals the contents of E.

STLC and STAC are provided to aid cooperating sequential processes in the manipulation of shared data. They often permit removal of mutually exclusive critical sections, hence possibly indefinite delays, from algorithms which would otherwise have required them.

Both of these instructions are interlocked against direct-memory input /output. Hence, these instructions may be used to interlock a process with a DMA, DMC or DMQ channel, or to interlock a memory location possibly being accessed by I/O. **MODES**=V, **FORMAT**=AP, **OPCODE**=001204, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ STLR addr Store L into addressed register

L→register (EA)W

Store the contents of L into the register location specified by addr. There are three cases of this instruction which are summarized under LDLR. Only the word portion of the effective address, ((EA)W), is used. **MODES**=V, **FORMAT**=MR, **OPCODE**=03 01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ STX addr Store X register

X→[EA]16

Store the contents of the X register in location addr. The contents of the X register are unaffected and the previous contents of addr are lost. This instruction cannot itself specify indexing, although an address word retrieved in the effective address calculation may do so in 16S mode. **MODES**=SRV, **FORMAT**=MR, **OPCODE**=15, **C**=unchanged, **L**=unchanged, **CC** =unchanged.

▶ STY addr Store Y

Y→[EA]32

Store the contents of Y into the location specified by addr. Cannot be indexed. **MODES**=V, **FORMAT**=MR, **OPCODE**=35 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TAB Transfer A to B

A→B

Move the contents of A to B. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140314, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TAX Transfer A to X

A→X

Move the contents of A to X. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140504, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TAY Transfer A to Y

A→Y

Move the contents of A to Y. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140505, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TBA Transfer B to A

B→A

Move the contents of B to A. **MODES**=V, **FORMAT**=GEN, **OPCODE**=140604, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TXA Transfer X to A

X→A

Move the contents of X to A. **MODES**=V, **FORMAT**=GEN, **OPCODE**=141034, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TYA Transfer Y to A

Y→A

Move the contents of Y to A. **MODES**=V, **FORMAT**=GEN, **OPCODE**=141124, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ XCA Exchange and clear the A register

A→B;0→A

Exchange (swap) the A and B registers; then clear A. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140104, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ XCB Exchange and clear the B register

B→A;0→B

Exchange (swap) the B and A registers; then clear register B. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140204, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

PROGRAM CONTROL AND JUMP—PCTLJ

▶ ARGT Argument transfer

The Argument Transfer operation must be the first executable instruction of any procedure which takes arguments. It serves as a holding point for the program counter while argument transfer is taking place into the new frame. The program counter is advanced past it when argument transfer is complete. Procedures which specify zero arguments in their entry control blocks must not begin with an ARGT.

The list of argument transfer templates following the caller's PCL instruction is evaluated to generate a list of actual argument pointers in the new frame. The format of each argument transfer template is shown in Section 9. Each argument pointer may require one or more templates for its generation. The last template for each argument has its S (store) bit set. The last template for the last argument in the list has its L (last) bit set to terminate the argument transfer.

Each template specifies the calculation of an address by specifying a base register, a word and bit displacement from that register, and an optional indirection. If further offsets or indirections are required to generate the final argument address, the template will not have its store bit set, and the address calculated so far will be placed in the temporary base (XB) register (ring, segment, word numbers) and X-register (bit number) for access by the next template. Only one level of indirection can be specified by each AP.

Each time a template with its store bit set is encountered, the calculated address is stored in the next argument pointer position in the new stack frame. The first argument pointer

position is specified in the procedure's ECB. If the address has a zero bit offset, the address is stored in the two-word indirect format (with the E-bit clear) and the third word is not modified. Otherwise it is stored in the three-word format (E-bit set). In either case, three words are allocated to each pointer in the argument list.

If the caller's template list generates fewer arguments than are expected by the callee (as specified in the entry control block), argument pointers containing the pointer-fault bit set and all other bits reset (pointer-fault code 100000, "omitted argument") are stored for the missing arguments. The second and third words are not modified. On the other hand, if the caller's list generates more arguments than are specified by the callee, the surplus arguments are ignored. If the called procedure attempts to reference an omitted argument, other than to simply pass it on in another call, it will experience a pointer fault. If it passes on an omitted argument in another call, the argument will appear omitted to the newly called procedure.

If a call intends to omit all expected arguments, it may be followed by an argument transfer template with its last bit set but with its store bit cleared.

MODES=V, **FORMAT**=GEN, **OPCODE**=000605, **C**=unspecified, **L**=unspecified, **CC**=unspecified.

▶ CEA Compute effective address

Interpret the contents of the A register as a 16-bit indirect address word in the current addressing mode, calculate the effective address, and place the final effective address back in the A register. **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000111, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CREP addr Call recursive entry procedure

$$(P)+ \rightarrow [(S)+1]$$
$$EA \rightarrow (P)$$

Increment the program counter, P, and load P+1 into the location following the one specified by the current R-mode stack pointer. Load addr into the program counter and continue execution from that location.

The CREP instruction performs subroutine linkage for recursive or reentrant procedures. CREP stores the return address in the second word of a stack frame created by the ENTR instruction, rather than in the destination address as in a JST.

MODES=R, **FORMAT**=MR, **OPCODE**=10 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ EAA addr Effective address to A register

$$EA \rightarrow A$$

Calculate the effective address and load it into register A. The contents of addr are unaffected and the original contents of register A are overwritten and lost. **MODES**=R, **FORMAT**=MR, **OPCODE**=01 01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ EAL addr Effective address to L

$$EA \rightarrow L$$

Calculate the effective address and put it into the L register. **MODES**=V, **FORMAT**=MR, **OPCODE**=01 01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ EALB addr Effective address to LB

EA→LB

Calculate the effective address and put it into the link base, LB. **MODES**=V, **FORMAT**=MR, **OPCODE**=13 02, **C**=unchanged, **L**= unchanged, **CC**=unchanged.

▶ EAXB addr Effective address to XB

EA→XB

Calculate the effective address and put it into the temporary base,XB. **MODES**=V, **FORMAT** =MR, **OPCODE**=12 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ENTR n Enter R-mode recursive procedure stack

(S)1 → [(S)1-n]
(S)1-n → (S)2

Alter the R-Mode stack pointer by subtracting the value of N and store the previous value of S in the new location.

The ENTR instruction allocates a block of memory as a stack frame containing N locations:

The frame is created by subtracting N from the stack pointer contents, (S)1, to form (S)2, and then storing (S)1 at that address. Thus, the first word of the frame points to the previous frame. N may be negative or positive. **MODES**=R, **FORMAT**=MR, **OPCODE**=01 03, **C**= unchanged, **L**=unchanged, **CC**=unchanged.

▶ JDX addr Jump and decrement X

X=X-1;if X=0, then EA→**PC;else PC**=PC+1

Decrement the contents of the X register by one; then, if the contents of X are not equal to zero, load addr into the program counter and continue sequential operation from that location. Otherwise, execute the next sequential instruction. **MODES**=R, **FORMAT**=MR, **OPCODE**=15 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JEQ addr Jump if equal to zero

If A=0, then EA→PC

If the contents of the A register are equal to zero, then load addr into the program counter and continue sequential operation from that location. **MODES**=R, **FORMAT**=MR, **OPCODE** =02 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JGE addr Jump if greater than or equal to zero

If A⩾0, then EA→PC

If the contents of the A register are greater than or equal to zero, then load addr into the program counter and continue sequential operation from that location. **MODES**=R, **FORMAT**=MR, **OPCODE**=07 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JGT addr Jump if greater than zero

If A>0, then EA→PC

If the contents of the A register are greater than zero, then load addr into the program counter and continue sequential operation from that location. **MODES**=R, **FORMAT**=MR, **OPCODE**=05 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JIX addr Jump and increment X

X=X+1;if X=0, then EA→PC; else PC=PC+1

Increment the contents of the X register by one; then, if the contents of X are not equal to zero, load addr into the program counter and continue sequential operation from that location. Otherwise, execute the next sequential instruction. **MODES**=R, **FORMAT**=MR, **OPCODE**=15 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JLE addr Jump if less than or equal to zero

If A⩽0, then EA→PC

If the contents of the A register are less than or equal to zero, then load addr into the program counter and continue sequential operation from that location. **MODES**=R, **FORMAT**=MR, **OPCODE**=04 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JLT addr Jump if less than zero

If A<0, then EA→PC

If the contents of the A register are less than zero, then load addr into the program counter and continue sequential operation from that location. **MODES**=R, **FORMAT**=MR, **OPCODE** =06 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JMP addr Jump

EA→PC

Transfer control to location addr by loading addr into the program counter and continue sequential operation from that location. **MODES**=SRV, **FORMAT**=MR, **OPCODE**=01, **C**= unchanged, **L**=unchanged, **CC**=unchanged.

▶ JNE addr Jump if not equal to zero

If A≠0, then EA→PC

If the contents of the A register are not equal to zero, then load addr into the program counter and continue sequential operation from that location. **MODES**=R, **FORMAT**=MR, **OPCODE**=03 30, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JST addr Jump and store

PC→[EA]16;EA+1→PC

Call a subroutine by storing the contents of the program counter (which points to the next location after the JST instruction) in location addr. Continue execution at location addr+1. In non-restricted mode, interrupts are inhibited for one instruction cycle following a JST.

The return address is truncated according to the addressing mode before it is stored, and higher-order bits of the memory location are not altered. It is thus possible to preset the I or X bits of such locations:

Mode	Preset Allowed
16S	I, X
32S, 32R	I
64, 64V	-

Note

Cannot be used in shared code.

MODES=SRV, **FORMAT**=MR, **OPCODE**=10, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JSX addr Jump and store return in X

(PC)W→X;EA→PC

Increment the program counter by one and load into the X register. Load addr into the program counter and continue sequential operation from that location. **MODES**=RV, **FORMAT**=MR, **OPCODE**=35 03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JSXB addr Jump and set XB

PC→XB;EA→PB

Save the 32-bit contents of the program counter in XB, and transfer control to location addr. JSXB may be used to make both intersegment and intrasegment subroutine calls. **MODES**= V, **FORMAT**=MR, **OPCODE**=14 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JSY addr Jump and set Y

(PC)W→Y;EA→PB

Save the word number of the program counter in the Y register and transfer control to location addr. Only the word number portion of the return address is saved, JSY may (usually) only be used to call subroutines that reside in the same procedure segment. **MODES**=V, **FORMAT**=MR, **OPCODE**=14, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PCL addr Procedure call

Call procedure whose ECB is at addr.

Step 1. **Calculation of target ring number**

1. If the caller has Read access to the segment (segment number of addr) containing the ECB, new ring=current ring.
2. If the caller has gate access to the segment containing the ECB, new ring=ring field of ECB(PB). The ECB must start on a modulo-16 boundary in this case.

If neither 1. nor 2. holds, an access violation results.

Step 2. **Stack frame allocation**

1. If ECB(stack root)=0, then stack root=ECB(stack root), else stack root = current stack segment.
2. Fetch the free pointer at location 0 of segment (stack root). If there is sufficient room remaining (size needed given by ECB(SFSIZE), allocate frame here and update free pointer in segment stack root.
3. If no room in this segment, fetch the extension pointer at location 2 of the segment pointed to by free pointer. If 0, generate stack overflow fault. Else, use extension pointer as a new free pointer and go to step 2.

Step 3. **New frame header setup**

1. The flag word (word 0) is set to 0.
2. The caller's PB, SB, LB and keys are saved (X,Y and XB are lost) in the frame header. The ring field of PB properly reflects the ring of

execution of the caller. The saved PB at this moment points to the word following the PCL instruction. It will be updated when argument transfer (if any) is complete to point beyond the argument templates. Word '11 of the stack frame is set to the word number of this initial value of saved PB (i.e., points to PCL+2).

Step 4. **Callee state load**

1. The callee's PB, LB, and keys are loaded from the entry control block, except that the ring field of PB has no effect if the ECB is not a gate. The SB register is set to point at the new stack frame.

Step 5. **Argument transfer**

1. If ECB(NARGS) is 0, this step is skipped. Otherwise, the one or more AP's (argument templates) following the PCL instruction are processed to load argument pointers into the callee's stack frame. At least one AP must follow PCL if the callee expects arguments; no AP may follow if the callee expects no arguments. The saved PB in callee's stack frame is updated to point beyond the AP's when argument transfer is done. See the ARGT instruction for a description of argument transfer.

MODES=V, **FORMAT**=MR, **OPCODE**=10 02, **C**=unspecified, **L**=unspecified, **CC**=unspecified

▶ PRTN Procedure return

Deallocates the current stack frame and returns to the environment of the procedure that called it. The stack frame is deallocated by storing the current stack base register into the free pointer. The caller's state is restored by loading his program counter, stack base register, linkage base register, and keys from the frame being left. The ring number in the program counter is weakened with the current ring number. The current stack frame consists of the frame created upon entry to the current procedure plus all extensions created during the execution of the current procedure. **MODES**=V, **FORMAT**=GEN, **OPCODE**=000611, **C**=loaded by instruction, **L**=loaded by instruction, **CC**=loaded by instruction.

▶ RTN Return from R-mode recursive procedure

[(S)+1]→P[(S)]→S

Fetch the return address from word 2 of the previous stack frame and load the result in the program counter; then transfer word 1 (the pointer to the preceding stack frame) to the S register.

If the return address is 0, (S) is unchanged and a PSU (Procedure Stack Underflow) fault is taken (interrupt through location '75 in physical memory is taken on the Prime 300). **MODES**=SR, **FORMAT**=GEN, **OPCODE**=000105, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ STEX Stack extend

Obtains additional space in the procedure stack for automatic variables. Such space is automatically deallocated and reclaimed for other uses when the procedure returns, just like the original frame created when the procedure was entered. The L register specifies the desired contiguous size of the extension in words. The size is rounded up to an even number of words. The address of the extension is returned as a segment number/word number pointer in the L register. It is possible that the extension may not be contiguous with the initial frame (there may have been insufficient room left in the same segment). Any number of extensions may be made. This instruction can cause a stack overflow fault. **MODES**=V, **FORMAT**=GEN, **OPCODE**=001315, **C**=unspecified, **L**=unspecified, **CC**=unspecified.

▶ XEC addr Execute

Execute the instruction at location addr, but do not transfer control to that location. Not all instructions can be executed by the instruction.

No multi-word instructions can be executed properly. All one-word instructions can be executed properly except JMP, JST, and address-mode changing generics. Instructions which skip do so relative to the XEC instruction. On any fault or interrupt, the saved program counter is relative to the XEC instruction. **MODES**=RV, **FORMAT**=MR, **OPCODE** =01 02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

PROCESS EXCHANGE (RESTRICTED)—PRCEX

There are seven process exchange instructions:

Mnemonic	Opcode	C	L	CC
INBC	001217	unspecified	unspecified	unspecified
INBN	001215	unspecified	unspecified	unspecified
INEC	001216	unspecified	unspecified	unspecified
INEN	001214	unspecified	unspecified	unspecified
NFYB	001211	unspecified	unspecified	unspecified
NFYE	001210	unspecified	unspecified	unspecified
WAIT	000315	unchanged	unchanged	unchanged

See the System Architecture Manual for a complete discussion of the process exchange mechanism. All process exchange instructions are restricted.

QUEUE MANAGEMENT INSTRUCTIONS—QUEUE

The instructions provided for queue manipulation are of the generic-AP class, in which a following AP-pointer provides the address to the queue control block. Section 9 contains the queue control block description.

Data is to or from register A and the results of the operation are given in the condition code bits for later testing.

addr refers to a control block in virtual space. The virtual queue control block differs from the physical in that a segment number is provided instead of a physical address. Ring zero privilege is required to manipulate physical queues; any non-ring zero attempt to access physical queues will result in a restrict mode violation fault. Also the ring number determines the privilege of access into both the control block and the data block.

▶ ABQ addr Add to bottom of queue

Add the contents of the A-register to the bottom of the queue defined by the QCB at addr. The condition codes are set EQ if the queue is full, e.g., the word could not be added. **MODES**=V, **FORMAT**=AP, **OPCODE**=141716, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ ATQ addr Add to top of queue

Add the contents of the A-register to the top of the queue defined by the QCB (Queue Control Block) at addr. The condition codes are set EQ if the queue is full, e.g., the word could not be added. **MODES**=V, **FORMAT**=AP, **OPCODE**=141717, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ RBQ addr Remove from bottom of queue

Remove a single word from the bottom of the queue defined by the QCB at addr, and place it in the A-register. If the queue is empty, set A=0 and condition codes EQ. **MODES**=V, **FORMAT**=AP, **OPCODE**=141715, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ RTQ addr Remove from top of queue

Remove a single word from the top of the queue defined by the QCB at addr, and place it in the A-register. If the queue is empty, set A=0 and condition codes EQ. **MODES**=V, **FORMAT**=AP, **OPCODE**=141714, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ TSTQ addr Test queue

Set the A-register to the number of items in the queue defined by the QCB at addr. If the queue is empty, set condition codes EQ. **MODES**=V, **FORMAT**=AP, **OPCODE**=141757, **C**=unchanged, **L**=unchanged, **CC**=result.

SHIFT GROUP—SHIFT

Shifting is the movement of the contents of a register bit-to-bit. The instructions in this group shift or rotate right or left the contents of A or the contents of A and B treated as a single register with A on the left. Although these instructions are similar in format and operation, functionally some are logical and others arithmetic.

A shift is logical or arithmetic simply in terms of the way the data word is interpreted: a logical shift treats it as a string of bits whereas an arithmetic shift treats it as a signed number.

Rotation is a cyclic logical shift such that information rotated out at one end is put back in at the other. The last bit rotated in at the right or left is also saved in C.

In a logical right or left shift, the contents of the register or registers are moved bit-to-bit with 0's brought in at the end being vacated. Information shifted out at the other end is lost. The last bit shifted out goes to C.

A right arithmetic shift fills the vacated left positions with the sign bit. The C-bit reflects the last bit shifted out on the right.

A left arithmetic shift includes the sign, but interprets a sign change as overflow. It fills the vacated right positions with 0's and sets the C-bit on overflow.

Hence, arithmetic shifting is equivalent to multiplying or dividing the number by a power of 2, provided no information is lost. These operations also use the C-bit to detect the loss of any bit of significance in a left arithmetic shift, and in all other cases to save the last bit shifted out.

In a shift instruction word, bits 3–6 are all 0's and the group is indicated by 01 in bits 1 and 2. Bits 7–10 indicate the particular type of shift, and bits 11–16 specify the twos complement of the number of places to be shifted. Mnemonics are available for the individual types, so the opcode may be regarded as the left four digits of the instruction word, with the word completed by adding the right two digits for the number of places. Note that the mnemonics are constructed using "logical" to mean a logical shift and "shift" to mean specifically an arithmetic shift.

▶ **ALL n A left logical**

$$C \leftarrow A_1—A_{16} \leftarrow 0$$

Shift the contents of register A left n places, bringing zeros into bit 16; data shifted out of bit 1 are lost, except that the last bit shifted out is saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0414xx, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ **ALR n A left rotate**

$$\hookleftarrow A_1—A_{16} \Leftrightarrow C$$

Shift the contents of register A left n places, rotating bit 1 into bit 16. The last bit rotated back in at the right is also saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0416xx, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ **ALS n A left shift**

$$A_1 \leftarrow A_2 — A_{16} \leftarrow 0$$

Shift the contents of register A left arithmetically n places, bringing zeros into bit 16. Data shifted out of bit 1 are lost. The C-bit is initially cleared. If the sign (bit 1) changes state, set C. A sign change indicates that a bit of significance (a one in a positive number, a zero in a negative) has been shifted out of the magnitude part. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0415xx, **C**=overflow, **L**=unspecified, **CC**=unchanged.

▶ **ARL n A right logical**

$$0 \rightarrow A_1—A_{16} \rightarrow C$$

Shift the contents of register A right n places, bringing zeros into bit 1; data shifted out of bit 16 are lost, except that the last bit shifted out is saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0404xx, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ **ARR n A right rotate**

$$C \Leftrightarrow A_1\text{--}A_{16} \lrcorner$$

Shift the contents of register A right n places, rotating bit 16 into bit 1. The last bit rotated back in at the left is also saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0406xx, **C** =shift extension, **L**=unspecified, **CC**=unchanged.

▶ **ARS n A right shift**

$$A_1 \rightarrow A_2—A_{16} \rightarrow C$$

Shift the contents of register A right arithmetically n places, leaving the sign (bit 1) unaffected, but shifting it into the magnitude part, zeros in a positive number, ones in a negative. Data shifted out of bit 16 are lost, except that the last bit shifted out is saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0405xx, **C**=shift extension, **L**=unspecified, **CC**= unchanged.

▶ **LLL n Long left logical**

$$C \leftarrow A_1—A_{16} \leftarrow B_1—B_{16} \leftarrow 0$$

Shift the contents of registers A and B left n places, bringing zeros into bit 16 of register B. Bit 1 of register B is shifted into bit 16 of register A; data shifted out of bit 1 of register A are lost, except that the last bit shifted out is saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0410xx, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

LLR n Long left rotate

$$\leftarrow A_1 - A_{16} \leftarrow B_1 - B_{16} \Leftrightarrow C$$

Shift the contents of registers A and B left n places, rotating bit 1 of register A into bit 16 of register B. Bit 1 of register B shifts into bit 16 of register A. The last bit rotated from register A back to B is also saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0412xx, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

LLS n Long left shift

$$A_1 \leftarrow A_2 - A_{16}\ B_1\ B_2 - B_{16} \leftarrow 0$$

Shift the contents of the 31-bit integer in register A|B left arithmetically n places, bringing zeros into bit 16 of register B, bypassing bit 1 of register B; Bit 2 of register B is shifted into bit 16 of register A. Data shifted out of bit 1 of register A are lost. If the sign (bit 1 of register A) changes state, set C; otherwise, clear C. **MODES**=SR, **FORMAT**=SHIFT, **OPCODE**= 0411xx, **C**=overflow, **L**=unspecified, **CC**=unchanged.

LLS n Long left shift

$$C \leftarrow L_1 \leftarrow L_2 - L_{32} \leftarrow 0$$

Shift the contents of the 32-bit integer in the L register left arithmetically n places, bringing zeros into bit 32. Data shifted out of bit 1 are lost. If the sign (bit 1) changes state, set C; otherwise clear C. **MODES**=V, **FORMAT**=SHIFT, **OPCODE**=0411xx, **C**=overflow, **L**=unspecified, **CC**=unchanged.

LRL n Long right logical

$$0 \rightarrow A_1 - A_{16} \rightarrow B_1 - B_{16} \rightarrow C$$

Shift the contents of register A and B right n places, bringing zeros into bit 1 of register A. Bit 16 of register A is shifted into bit 1 of register B. Data shifted out of bit 16 of register B are lost, except that the last bit shifted out is saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE**=0400xx, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

LRR n Long right rotate

$$C \Leftrightarrow A_1 - A_{16} \rightarrow B_1 - B_{16} \rightarrow$$

Shift the contents of register A and B right n places, rotating bit 16 of register B into bit 1 of register A. Bit 16 of register A is shifted into bit 1 of register B. The last bit rotated from register B back to register A is also saved in C. **MODES**=SRV, **FORMAT**=SHIFT, **OPCODE** =0402xx, **C**=shift extension, **L**=unspecified. **CC**=unchanged.

LRS n Long right shift

$$A_1 \rightarrow A_2 - A_{16}\ B_1\ B_2 - B_{16} \rightarrow C$$

Shift the contents of the 31-bit integer in register A|B right arithmetically n places, leaving bit 1 of register A unaffected, bypassing bit 1 of register B, and shifting the sign (bit 1 of register A) into the magnitude part (zeros in a positive number, ones in a negative). Bit 16 of register A is shifted into bit 2 of register B; data shifted out of B bit 16 are lost, except that the last bit shifted out is saved in C. **MODES**=SR, **FORMAT**=SHIFT, **OPCODE**=0401xx, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ **LRS n Long right shift**

$$L_1 \rightarrow L_2 — L_{32} \rightarrow C$$

Shift the contents of the 32-bit integer in the L register right arithmetically n places, leaving bit 1 unaffected. Data shifted out of bit 32 are lost, except that the last bit shifted out is saved in C. **MODES**=V, **FORMAT**=SHIFT, **OPCODE**=0401xx, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

SKIP CONDITIONAL—SKIP

▶ **DRX Decrement and replace X**

X−1→X;if X=0 then PC+1→PC

Subtract 1 from the contents of the X register and place the result back in that register. Skip the next word in sequence if the result is zero. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**= 140210, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ **IRS addr Increment memory, replace, and skip**

[EA]16+1→[EA]16; if [EA]16=0 then PC+1→PC

Add 1 to the contents of location addr and place the result back in addr. Skip the next word in sequence if the result is zero. **MODES**=SRV, **FORMAT**=MR, **OPCODE**=12, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ **IRX Increment and replace X**

X+1→X; if X=0 then PC+1→PC

Add 1 to the contents of the X register and place the result back in that register. Skip the next word in sequence if the result is zero. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=140114, **C** =unchanged, **L**=unchanged, **CC**=unchanged.

▶ **SAR n Skip on A-bit reset**

If A(n)=0 then PC+1→PC

If bit n in the A register is 0, skip the next word in sequence.

Note

The assembler will convert n to octal equivalent of the bit number minus one.

MODES=SRV, **FORMAT**=GEN, **OPCODE**=10026x, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ **SAS n Skip on A-bit Set**

If A(n)=1 then PC+1→PC

If bit n in register A is 1, skip the next instruction in sequence.

Note

The assembler will convert n to the octal equivalent of the bit number minus one.

MODES=SRV, **FORMAT**=GEN, **OPCODE**=10126x, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ SGT Skip if A greater than zero

If A>0 then PC+1→PC

If the contents of register A is greater than zero, skip the next word in sequence. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=100220, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ SKP n Skip group

Skip conditions are selected by individual bits or combinations of them.

- Bits 1-6 are always 100000.
- Bit 7=1 means if true, skip the next instruction.
- Bit 7=0 means if false, skip the next instruction.
- Bit 9=0 means test a combination of bits.

The various conditions, the bits that select them and the mnemonics and opcodes for them are given in Table 11-5. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=100000, **C**=unchanged, **L** =unchanged, **CC**=unchanged.

▶ SLE Skip if A less than or equal to zero

If A⩽0 then PC+1→PC

If the number contained in A is less than or equal to zero, skip the next word in sequence. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=101220, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ SNR n Skip on sense switch reset

If sense switch n=0 then PC+1→PC

If sense switch n is off (not up), skip the next word in sequence. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=10024x, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

▶ SNS n Skip on sense switch set

If sense switch n=1 then PC+1→PC

If sense switch n is on (up), skip the next word in sequence. **MODES**=SRV, **FORMAT**=GEN, **OPCODE**=10124x, **C**=unchanged, **L**=unchanged, **CC**=unchanged. *Restricted instruction.*

Table 11-5. Combination Skip Group

	Mnemonic	Selector Bits	Bit 7	Skip on Condition	Op Code
	NOP		1	None (no-op)	'101000
	SKP		0	Skip unconditionally	'100000
	SMI	8	1	A Minus (A(1) = 1)	'101400
	SPL	8	0	A Plus (A(1) = 0)	'100400
	SLN	10	1	LSB Nonzero (A(16) = 1)	'101100
	SLZ	10	0	LSB Zero (A(16) = 0)	'100100
	SNZ	11	1	A Nonzero	'101040
	SZE	11	0	A Zero	'100040
R	**SS1**	12	1	Sense Switch 1 Set	'101020
R	**SR1**	12	0	Sense Switch 1 Reset	'100020
R	**SS2**	13	1	Sense Switch 2 Set	'101010
R	**SR2**	13	0	Sense Switch 2 Reset	'100010
R	**SS3**	14	1	Sense Switch 3 Set	'101004
R	**SR3**	14	0	Sense Switch 3 Reset	'100004
R	**SS4**	15	1	Sense Switch 4 Set	'101002
R	**SR4**	15	0	Sense Switch 4 Reset	'100002
R	**SSS**	12–15	1	All Sense Switches 1–4 Set	'101036
R	**SSR**	12–15	0	Any of Sense Switches 1–4 Reset	'100036
	SSC	16	1	Set C	'101001
	SRC	16	0	Clear C	'100001

Skip conditions can be combined using SKP and giving the bit 7–16 configuration for the combination in the address field. All conditions combined must agree on bit 7. If bit 7 is set then the skip will take place if all conditions are true. If bit 7 is clear then the skip will take place if any of the conditions is true.

12
Instruction definitions-I

ADDRESSING MODE—ADMOD

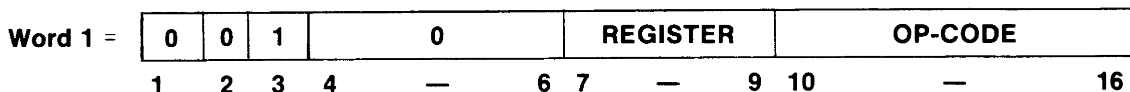
Defined in Section 11.

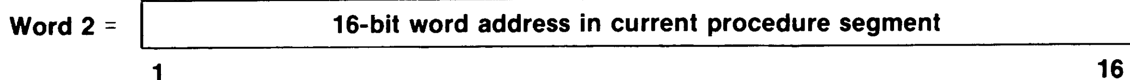
E16S	Enter 16S Mode
E32R	Enter 32R Mode
E32S	Enter 32S Mode
E64R	Enter 64R Mode
E64V	Enter 64V Mode
E32I	Enter 32I Mode

BRANCH—BRAN

The branch instructions are two word register generics which test the contents of a register or the result of a previous ARITHMETIC or COMPARE operation as indicated by the condition codes (CC), the C-bit, and the L-Bit.

The bit layout is:





Condition code branches test six conditions based on the LT bit, the EQ bit, and the opcode.

Condition	Meaning
$<$	Branch if LT bit set and EQ bit cleared
$\leqslant$	Branch if LT bit set or EQ bit set
$=$	Branch if EQ bit set
$\neq$	Branch if EQ bit cleared
$\geqslant$	Branch if LT bit cleared or EQ bit set
$>$	Branch if LT bit cleared and EQ bit cleared

▶ Test Relation to 0 and branch if true

These instructions have the following format:

Branch if {Register (R) / Half-Register (H) / Floating-Register (F)} {LT / LE / EQ / NE / GE / GT} 0

For example: BRLT R,addr means Branch to addr if Register Less Than zero. **FORMAT**=IBRN, **OPCODES**=See chart below, **C**=unchanged, **L**=unchanged, **CC**=result.

Mnemonic	Function	Opcode
BRLT R,addr	If R<0, then addr→PC	104
BRLE R,addr	If R⩽0, then addr→PC	100
BREQ R,addr	If R=0, then addr→PC	102
BRNE R,addr	If R≠0, then addr→PC	103
BRGE R,addr	If R⩾0, then addr→PC	105
BRGT R,addr	If R>0, then addr→PC	101
BHLT RH,addr	If RH<0, then addr→PC	104
BHLE RH,addr	If RH⩽0, then addr→PC	110
BHEQ RH,addr	If RH=0, then addr→PC	112
BHNE RH,addr	If RH≠0, then addr→PC	113
BHGE RH,addr	If RH⩾0, then addr→PC	105
BHGT RH,addr	If RH>0, then addr→PC	111
BFLT F,addr	If F<0, then addr→PC	124
BFLE F,addr	If F⩽0, then addr→PC	120
BFEQ F,addr	If F=0, then addr→PC	122
BFNE F,addr	If F≠0, then addr→PC	123
BFGE F,addr	If F⩾0, then addr→PC	125
BFGT F,addr	If F>0, then addr→PC	121

Test register bit and branch

BRBR R,BITNO,addr Branch if register bit reset (equals zero): If R(BITNO)=0, then addr →PC. **FORMAT**=IBRN, **OPCODE**=040–077, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

BRBS R,BITNO,addr Branch if register bit set (equals one): If R(BITNO)=1, then addr→PC. **OPCODE**=000–037, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

Branch on incremented or decremented register

These instructions have the following format:

{Increment / Decrement} {Register / Half Register} by {1 / 2 / 4} then branch if result≠0

For example: BRI1 R,addr means Increment the contents of the Register by 1 and then Branch to addr if the result is not equal to zero. **FORMAT**=IBRN, **OPCODES**-See chart below, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

Mnemonic	Function	Opcode
BRI1 R,addr	R+1→R; if R≠0, then addr→PC	130
BRI2 R,addr	R+2→R; if R≠0, then addr→PC	131
BRI4 R,addr	R+4→R; if R≠0, then addr→PC	132
BHI1 RH,addr	RH+1→RH; if RH≠0, then addr→PC	140
BHI2 RH,addr	RH+2→RH; if RH≠0, then addr→PC	141
BHI4 RH,addr	RH+4→RH; if RH≠0, then addr→PC	142
BRD1 R,addr	R-1→R; if R≠0, then addr→PC	134
BRD2 R,addr	R-2→R; if R≠0, then addr→PC	135
BRD4 R,addr	R-4→R; if R≠0, then addr→PC	136
BHD1 RH,addr	RH-1→RH; if RH≠0, then addr→PC	144
BHD2 RH,addr	RH-2→RH; if RH≠0, then addr→PC	145
BHD4 RH,addr	RH-4→RH; if RH≠0, then addr→PC	146

▶ CGT R,n Computed GOTO

If 1<R<n, then |PC+R|→PC, else PC+n→PC

Instruction word followed by n further words: word 1 contains integer n and words 2-n contain branch addresses within the current procedure segment.

If the contents of register R is less than n and greater than or equal to 1, then control passes to the address in PC+R; otherwise no branch is taken and control passes to PC+n. **FORMAT** =IBRN, **OPCODE**=026, **C**=unspecified, **L**=unspecified, **CC**=unspecified.

Defined in Section 11:

BCEQ	Branch on condition code equal
BCGE	Branch on condition code greater than or equal
BCGT	Branch on condition code greater than
BCLE	Branch on condition code less than or equal
BCLT	Branch on condition code less than
BCNE	Branch on condition code not equal
BCR	Branch if C-Bit=0
BCS	Branch if C-Bit=1
BLR	Branch if L-Bit=0
BLS	Branch if L-Bit=1
BMEQ	Branch if magnitude equal 0
BMGE	Branch if magnitude greater than or equal 0
BMGT	Branch if magnitude greater than 0
BMLE	Branch if magnitude less than or equal 0
BMLT	Branch if magnitude less than 0
BMNE	Branch if magnitude not equal 0

CHARACTER OPERATIONS—CHAR

These instructions use the field address and length registers (FALR) which have been set up by field operation instructions prior to the use of these instructions. Character string operations perform memory to memory operations on variable length character fields. The FAR is used as a byte pointer and the bit offset (low order 3 bits) is ignored.

Data type: Characters are 8-bit bytes. The format is unspecified and may be determined by programmer, e.g., ASCII, EBCDIC, etc. The translate instruction (ZTRN), for example, uses a table set up by the programmer to translate one character code into another.

▶ LDC FALR,R Load Character

If the field length register FLR is nonzero, load the single character pointed to by field address register FAR into register R, bits 9–16. Register R bits 1–8 are cleared. The low order 3 bits of the bit offset in the field address register are ignored, implying that the character must be byte aligned. The field address register is advanced 8 bits to the next character, and the field length register is decremented by 1. Set condition code NE (clear EQ). If the field length register is zero, then set the condition code EQ. **FORMAT**=RGEN, **FALR0 OPCODE** =162, **FALR1 OPCODE**=172, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ STC FALR,R Store Character

Store bits 9–16 of register R into the character pointed to by field address register FAR. The low order 3 bits of the bit offset of the field address register are ignored, implying that the character must be byte aligned. The field address register is advanced 8 bits to the next character, and the field length register is decremented by 1. Set the condition code NE (clear EQ). If the field length register is zero, set the condition code EQ and do not store. **FORMAT** =RGEN, **FALR0 OPCODE**=166, **FALR1 OPCODE**=176, **C**=unchanged, **L**=unchanged, **CC**= unchanged.

Summary of instructions defined in section 11

ZCM	Compare Character Field
ZED	Character Edit
ZFIL	Fill Character Field
ZMV	Move Character Field
ZMVD	Move Equal Length Fields
ZTRN	Translate Character Fields

CLEAR REGISTER AND MEMORY—CLEAR

▶ CR R Clear Register

$$0 \rightarrow R$$

Fill R with zeros. **FORMAT**=RGEN, **OPCODE**=056, **C**=unchanged, **L**=unchanged, **CC**= unchanged.

▶ CRBL R Clear High Byte 1 Left

$$0 \rightarrow RH(1\text{–}8)$$

Fill bits 1–8 of R with zeros. **FORMAT**=RGEN, **OPCODE**=062, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CRBR R Clear High Byte 2 Right

$$0 \rightarrow RH(9\text{–}16)$$

Fill bits 9–16 of R with zeros. **FORMAT**=RGEN, **OPCODE**=063,, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CRHL R Clear Left Halfword

0→RH

Fill bits 1–16 of R with zeros. **FORMAT**=RGEN, **OPCODE**=054, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CRHR R Clear Right Halfword

0→RL

Fill bits 17–32 of R with zeros. **FORMAT**=RGEN, **OPCODE**=055, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ZM addr Zero Memory Fullword

0→|EA|32

Fill contents of addr with zeros. **FORMAT**=MRNR, **OPCODE**=43, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ZMH addr Zero Memory Halfword

0→|EA|16

Fill contents of addr with zeros. **FORMAT**=MRNR, **OPCODE**=53, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

DECIMAL ARITHMETIC—DECI

Defined in Section 11:

XAD	Decimal Add
XBTD	Binary to Decimal Conversion
XCM	Decimal Compare
XDTB	Decimal to Binary Conversion
XDV	Decimal Divide
XED	Numeric Edit
XMP	Decimal Multiply
XMV	Decimal Move

FIELD OPERATIONS—FIELD

These instructions set up and manipulate the field address and length registers, which are used by both the decimal and character string instructions. The interpretation of the value in the field length registers depends on the data type and instruction using them.

▶ ARFA FAR,R Add Register to Field Address Register

R+FAR→FAR

Add the contents of R to field address register FAR, putting the result in the field address register. **FORMAT**=RGEN, **FAR0 OPCODE**=161, **FAR1 OPCODE**=171, **C**=unspecified, **L**=unspecified, **CC**=unchanged.

▶ TFLR FLR,R Transfer Field Length to Register

FLR→R

Move the contents of field length register FLR to R. **FORMAT**=RGEN, **FLR0 OPCODE**=163, **FLR1 OPCODE**=173, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ TRFL FLR,R Transfer Register to Field Length Register

R→FLR

Move the content of R to field length register FLR. **FORMAT**=RGEN, **FLR0 OPCODE**=165, **FLR1 OPCODE**=175, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

Summary of instructions defined in section 11

EAFA 0	Load Field Address Register 0
EAFA 1	Load Field Address Register 1
LFLI 0	Load Field Length Register Immediate 0
LFLI 1	Load Field Length Register Immediate 1
STFA 0	Store Field Address Register
STFA 1	Store Field Address Register

FLOATING POINT ARITHMETIC—FLPT

See Section 9 for a description of the processor dependent register formats and the floating point data stuctures.

Normalization

The result of every floating point calculation is normalized. In normal form, the most signficant digit of the mantissa follows the binary point. If an operation produced a mantissa that is smaller than normal, the mantissa is shifted left until the most significant bit differs from the sign bit, and the exponent is decreased by one for each shift. Bits vacated at the right are filled by zeros. If the result of an operation overflows the mantissa, it is shifted right one place, the overflow bit is made the most significant bit, and the exponent is increased by 1.

Floating point exceptions

In the basic arithmetic operations, increasing the exponent in the floating point register beyond 32639 is an overflow; decreasing it below -32896 is an underflow.

An attempt to store a single-precision number with an exponent greater than 127 or less than -128 in the two-word memory format results in a different type of exception—see Table 11-2. The number in the floating point register is not altered by the FST operation and so can be recovered if necessary.

Other detected exceptions are an attempt to divide by zero or to form an integer exceeding ±30 bits or about ±1 billion decimal.

On the Prime 350 and up, the floating point exception is a fault rather than an interrupt and is controlled by the FLEX bit in the keys—see Section 9—Data Formats.

Single Precision—32 bits

▶ FA FR,addr Floating Add

FR1|EA|32→FR

Add the floating point number at addr to the contents of the floating point number in floating point register FR, and leave the resulting floating point number in the floating point register. Addition of floating point numbers requires that their exponents be the same power of two. This is accomplished by right shifting the smaller number by the difference in the exponents. After alignment, the mantissas are added. **FORMAT**=MRFR, **FR0 OPCODE**=14, **FR1 OPCODE**=16, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FC FR,addr Floating Compare

FR::|EA|32

Compare the contents of floating point register FR with the contents of addr and set the condition codes accordingly. **FORMAT**=MRFR, **FR0 OPCODE**=04, **FR1 OPCODE**=06, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ FCM FR Floating Complement

-FR→FR

Two's complement the mantissa of floating point register FR and normalize if necessary. **FORMAT**=RGEN, **FR0 OPCODE**=100, **FR1 OPCODE**=110, **C**=overflow, **L**=unspecified, **CC**=unspecified..

▶ FD FR,addr Floating Divide

FR/|EA|32→FR

Divide the contents of floating point register FR by the number in addr and leave the normalized quotient in the floating point register.l
FORMAT=MRFR, **FR0 OPCODE**=30, **FR1 OPCODE**=32, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FL FR,addr Floating Load

|EA|32→FR

Load the floating point number contained in addr into floating point register FR. **FORMAT**=MRFR, **FR0 OPCODE**=00, **FR1 OPCODE**=02, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ FLT FR,R Convert Integer to Floating Point

Float (R)→FR

Convert the integer in R to a normalized floating point number in floating point register FR. **FORMAT**=RGEN, **FR0 OPCODE**=105, **FR1 OPCODE**=115, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FLTH FR,R Convert Halfword Integer to Floating Point

FLOAT(RH)→FR

Convert the halfword integer in RH to a normalized floating point number in floating point register FR. **FORMAT**=RGEN, **FR0 OPCODE**=102, **FR1 OPCODE**=112, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FM FR,addr Floating Multiply

FR*|EA|32→FR

Multiply the contents of floating point register FR by the contents of addr and place the product in the floating point register with the mantissa normalized. **FORMAT**=MRFR, **FR0 OPCODE**=24, **FR1 OPCODE**=26, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FRN FR Floating Round

If bit 25 of the mantissa in floating point register FR is 1, add 1 to bit 24 and clear 25. **FORMAT**=RGEN, **OPCODE**=107, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FS FR,addr Floating Subtract

FR-[EA]32→FR

Subtract the contents of addr from floating point register FR by aligning exponents, and proceding as in FA except that the contents of addr are subtracted from floating point register. **FORMAT**=MRFR, **FR0 OPCODE**=20, **FR1 OPCODE**=22, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ FST FR,addr Floating Store

FR→[EA]32

Store the single precision floating point number contained in floating point register FR in addr. Bits 24–31 of the 31 bit mantissa are truncated when written into the 23-bit capacity memory storage. However, the mantissa may be rounded to bit 24 by a FRN instruction which adds 1 to bit 24 if bit 25 is 1. **FORMAT**=MRFR, **FR0 OPCODE**=10, **FR1 OPCODE**=12, **C**=overflow, **L**=unspecified, **CC**=unchanged.

▶ INT FR,R Convert Floating Point to Integer

Int(FR)→R

Convert the floating point number in floating point register FR to an integer in R. **FORMAT** =RGEN, **FR0 OPCODE** 103, **FR1 OPCODE** 113, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ INTH FR,R Convert Floating Point to Halfword Integer

Int(FR)→RH

Convert the floating point number in floating point register FR to a halfword integer in RH. **FORMAT**=RGEN, **FR0 OPCODE**=101, **FR1 OPCODE**=111, **C**=overflow, **L**=unspecified, **CC**= unspecified.

Double Precision—64 Bits

▶ DBLE FR Convert Single to Double

FR→FR

Convert single precision floating point number in floating point register FR to double precision floating point number in the floating point register. **FORMAT**=RGEN, **FR0 OPCODE**=106, **FR1 OPCODE**=116, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ DFA FR,addr Double Floating Add

FR+[EA]64→FR

Add the contents of addr to the contents of floating point register FR and put the result in the floating point register. **FORMAT**=MRFR, **FR0 OPCODE**=15, **FR1 OPCODE**=17, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DFC FR,addr Double Floating Compare

FR::[EA]64

Compare the contents of addr with the contents of floating point register FR and set the condition codes accordingly. **FORMAT**=MRFR, **FR0 OPCODE**=05, **FR1 OPCODE**=07, **C**= unchanged, **L**=unchanged, **CC**=result.

▶ DFCM FR Double Floating Complement

-FR→FR

Two's complement the double precision mantissa in floating point register FR and normalize if necessary. **FORMAT**=RGEN, **FR0 OPCODE**=144, **FR1 OPCODE**=154, **C**=overflow, **L**= unspecified, **CC**=unspecified.

▶ DFD FR,addr Double Floating Divide

FR/|EA|64→FR

Divide the double precision floating point number in floating point register FR by the double precision floating point number starting at addr and leave the result in the floating point register. Exponents are subtracted, and after the divisor mantissa is divided into the dividend mantissa, the quotient is normalized. **FORMAT**=MRFR, **FR0 OPCODE**=31, **FR1 OPCODE**=33, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DFL FR,addr Double Floating Load

|EA|64→FR

Load the double precision number contained in the four memory words at addr into floating point register FR. **FORMAT**=MRFR, **FR0 OPCODE**=01, **FR1 OPCODE**=03, **C**=unchanged, **L**= unchanged, **CC**=unchanged.

▶ DFM FR,addr Double Floating Multiply

FR+|EA|64→FR

Multiply the double precision floating point number in floating point register FR by the double precision floating point number starting at addr and leave the result in the floating point register. Exponents are added and, after mantissas are multiplied, the product is normalized. **FORMAT**=MRFR, **FR0 OPCODE**=25, **FR1 OPCODE**=27, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DFS FR,addr Double Floating Subtract

FR -|EA|64→FR

Subtract the contents of addr from the contents of floating point register FR and put the result in the floating point register. **FORMAT**=MRFR, **OPCODE**=21, **FR0 OPCODE**=21, **FR1 OPCODE**=23, **C**=overflow, **L**=unspecified, **CC**=unspecified.

▶ DFST FR,addr Double Floating Store

FR→|EA|64

Store the contents of floating point register FR into the four memory words at addr. **FORMAT**=MRFR, **FR0 OPCODE**=11, **FR1 OPCODE**=13, **C**=unchanged, **L**=unchanged, **CC**= unchanged.

INTEGER ARITHMETIC—INT

I-mode integer arithmetic instructions operate on 16 and 32 bit integers. See Section 9 for a description of the data formats.

▶ A R,addr Add Fullword

R+|EA|32→R

Add the 32-bit integer at addr to the 32-bit integer in register R, and put the result into R. **FORMAT**=MRGR, **OPCODE**=02, **C**=overflow, **L**=carry, **CC**=result.

▶ ADLR R Add Link to Register

if keys (L)=1 then R+11→R

If the L bit is set in the keys then add 1 to the contents of register R. **FORMAT**=RGEN. **OPCODE**=014, **C**=overflow, **L**=carry, **CC**=result.

▶ AH R,addr Add Halfword

RH+|EA|16→RH

Add the 16-bit integer at addr to the 16-bit integer in bits 1-16 of register R and put the result into bits 1-16 of R. **FORMAT**=MRGR, **OPCODE**=12, **C**=overflow, **L**=carry, **CC**=result.

▶ C R,addr Compare Fullword

R::|EA|32; set CC.

Arithmetically compare the 32-bit integer in R with the 32-bit integer at addr and set the condition codes to reflect the results. **FORMAT**=MRGR, **OPCODE**=61, **C**=unchanged, **L**=carry, **CC**=result.

▶ CH R,addr Compare Halfword

RH::|EA|16; set CC.

Arithmetically compare bits 1-16 of register R with the 16-bit integer at addr and set the condition codes to reflect the results. **FORMAT**=MRGR, **OPCODE**=71, **C**=unchanged, **L**=carry, **CC**=result.

▶ CHS R Change Sign

-R(1)→R(1)

Change bit 1 of register R to its opposite. **FORMAT**=RGEN, **OPCODE**=040, **C**=unchanged, **L** =unchanged, **CC**=unchanged.

▶ CSR R Copy Sign

R(1)→C; 0→R(1)

Copy the sign bit of register R, (bit 1), into C and zero R(1). **FORMAT**=RGEN, **OPCODE**=041, **C**=R(1), **L**=unchanged, **CC**=unchanged

▶ D R,addr Divide Fullword

R|R+1/|EA|32→R; Remainder→R+1

Divide the 64-bit integer in registers R and R+1 by the 32-bit integer at addr, and put the result in R and the remainder in R+1. The least significant bit of the dividend is in bit 64. Overflow if the quotient is less than -(2**31) or greater than 2**31-1. R must be an even register. **FORMAT**=MRGR, **OPCODE**=62, **C**=overflow/div by 0, **L**=unspecified, **CC**=unspecified.

▶ DH R,addr Divide Halfword

R/|EA|16→RH; Remainder→RH

Divide the 32-bit integer in register R by the 16-bit integer at addr, and put the quotient into bits 1–16 of R and the remainder into bits 17–32 of R. The least significant bit of the dividend is in bit 32. Overflow if the quotient is less than -(2**15) or greater than 2**15-1. **FORMAT** =MRGR, **OPCODE**=72, **C**=overflow/div by 0, **L**=unspecified, **CC**=unspecified.

▶ DH1 R Decrement Half Register by 1

RH-1→RH

Subtract one from RH and put the results into RH. **FORMAT**=RGEN, **OPCODE**=130, **C**= overflow, **L**=carry, **CC**=result.

▶ DH2 R Decrement Half Register by 2

RH-2→RH

Subtract two from RH and put the result into RH. **FORMAT**=RGEN, **OPCODE**=131, **C**= overflow, **L**=carry, **CC**=result.

▶ DM addr Decrement Memory Fullword

|EA|32-1→|EA|32

Subtract one from the 32-bit integer at addr and put the result into addr. **FORMAT**=MRNR, **OPCODE**=60, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ DMH addr Decrement Memory Halfword

|EA|16-1→|EA|16

Subtract one from the 16-bit integer at addr and put the result into addr. **FORMAT**=MRNR, **OPCODE**=70, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ DR1 R Decrement Register by 1

R-1→R

Subtract one from the contents of R and put the result into R. **FORMAT**=RGEN. **OPCODE**= 124, **C**=overflow, **L**=carry, **CC**=result.

▶ DR2 R Decrement Register by 2

R-2→R

Subtract two from the contents of R and put the result into R. **FORMAT**=RGEN. **OPCODE**= 125, **C**=overflow, **L**=carry, **CC**=result.

▶ IH1 Increment Half Register by 1

RH+1→RH

Add one to the contents of RH and put the result into RH. **FORMAT**=RGEN, **OPCODE**=126, **C**=overflow, **L**=carry, **CC**=result.

▶ IH2 R Increment Half Register by 2

RH+2→RH

Add two to the contents of RH and put the result into RH. **FORMAT**=RGEN, **OPCODE**=127, **C**=overflow, **L**=carry, **CC**=result.

▶ IM addr Increment Memory Fullword

[EA]32+1→[EA]32

Add one to the 32-bit integer at addr and put the result into addr. **FORMAT**=MRNR, **OPCODE**=40, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ IMH addr Increment Memory Halfword

[EA]16+1→[EA]16

Add one to the 16-bit integer at addr and put the result into addr. **FORMAT**=MRNR, **OPCODE**=50, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ IR1 R Increment Register by 1

R+1→R

Add one to the contents of register R and put the result in R. **FORMAT**=RGEN, **OPCODE**=122, **C**=overflow, **L**=carry, **CC**=result.

▶ IR2 R Increment Register by 2

R+2→R

Add two to the contents of register R and put the result in R. **FORMAT**=RGEN, **OPCODE**=123, **C**=overflow, **L**=carry, **CC**=result.

▶ M R,addr Multiply Fullword

R*[EA]32→R|R+1

Multiply the 32-bit integer in register R by the 32-bit integer at addr and put the 64-bit result into R and R+1. The least significant bit is in bit position 64. R must be an even register. **FORMAT**=MRGR, **OPCODE**=42, **C**=overflow, **L**=unspecified, **CC**=unchanged.

▶ MH R,addr Multiply Halfword

RH*[EA]16→R

Multiply the 16-bit integer in bits 1–16 of register R by the 16-bit integer at addr and put the 32-bit result into R. The least significant bit is in bit position 32. **FORMAT**=MRGR, **OPCODE** =52, **C**=overflow, **L**=unspecified, **CC**=unchanged.

▶ PID R Position For Integer Divide

R→R+1; R(1)→R(2–32)

Convert the 32-bit integer in register R to a 64 integer in registers R and R+1 by moving the contents of R to R+1, and extending the sign in bit 1 of R through bits 2–32 of R. **FORMAT**=RGEN, **OPCODE**=052, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PIDH R Position Half Register For Integer Divide

RH→RL; R(1)→R(2–16)

Convert the 16-bit integer in RH to 32-bit integer in R by moving the contents of RH to RL, and extending the sign in bit 1 through bits 2–16 of R. **FORMAT**=RGEN, **OPCODE**=053, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ PIM R Position After Multiply

R+1→R

Convert the 64-bit integer in registers R and R+1 to a 32-bit integer in R by moving the contents of R+1 to R. Overflow if a loss of precision would result. (i.e., bit 1 of R+1 is not the same as all the bits of R). **FORMAT**=RGEN, **OPCODE**=50, **C**=overflow, **L**=unspecified, **CC** =unspecified.

▶ PIMH R Position Half Register After Multiply

RL→RH

Convert the 32-bit integer in register R to a 16-bit integer in RH by moving the contents of RL to RH. Overflow if a loss of precision would result. **FORMAT**=RGEN, **OPCODE**=51, **C**= overflow, **L**=unspecified, **CC**=unspecified.

▶ S R,addr Subtract Fullword

R-[EA]32→R

Subtract the 32-bit integer at addr from 32-bit integer in register R, and put the result into R. **FORMAT**=MRGR, **OPCODE**=22, **C**=overflow, **L**=carry, **CC**=result.

▶ SH R,addr Subtract Halfword

RH-[EA]16→RH

Subtract the 16-bit integer at addr from the 16-bit integer in bits 1-16 of register R and put the result into bits 1-16 of R. **FORMAT**=MRGR, **OPCODE**=32, **C**=overflow, **L**=carry, **CC**= results.

▶ SSM R Set Sign Minus

1→R(1)

Set the sign bit of register R, (bit 1), equal to one. **FORMAT**=RGEN, OPCODE=042, **C**= unchanged, **L**=unchanged, **CC**=unchanged.

▶ SSP R Set Sign Plus

0→R(1)

Set the sign bit of register R, (bit 1), equal to zero. **FORMAT**=RGEN, **OPCODE**=043, **C**= unchanged, **L**=unchanged, **CC**=unchanged.

▶ TC R Two's Complement Register

-R+1→R

Replace the contents of register R by its two's complement. **FORMAT**=RGEN, **OPCODE**= 046, **C**=overflow, **L**=carry, **CC**=result.

▶ TCH R Two's Complement Half Register

-RH+1→RH

Replace the contents of RH by its two's complement. **FORMAT**=RGEN, **OPCODE**=047, **C**= overflow, **L**=carry, **CC**=result.

▶ **TM addr Test Memory Fullword**

[EA]32::0; set CC

Test the contents of addr and set condition code accordingly. **FORMAT**=MRNR, **OPCODE**= 44, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ **TMH addr Test Memory Halfword**

[EA]16:0;set CC

Test the contents of addr and set condition code accordingly. **FORMAT**=MRNR, **OPCODE**= 54, **C**=unchanged, **L**=unchanged, **CC**=result.

INTEGRITY CHECK FOR HARDWARE—INTGY

Defined in Section 11.

EMCM	Enter Machine Check Mode
LMCM	Leave Machine Check Mode
MDEI	Memory Diagnostic Enable Interleaved
MDII	Memory Diagnostic Inhibit Interleaved
MDIW	Memory Diagnostic Write Interleaved
MDRS	Memory Diagnostic Read Syndrome Bits
MDWC	Load Write Control Register
RMC	Clear Machine Check
VIRY	Verify
XVRY	Verify xis

INPUT/OUTPUT—I/O

▶ **EIO addr Execute I/O**

Interpret the low order 16 bits of addr as a Prime 400 PIO instruction. Set EQ on successful INA, OTA, SKS; OCP always sets NE. **FORMAT**=MRNR, **OPCODE**=34, **C**=unchanged, **L**= unchanged, **CC**=result.

Summary of instructions from section 11

CAI	Clear Active Interrupt
ENB	Enable Interrupts
ESIM	Enter Standard Interrupt Mode
EVIM	Enter Vectored Interrupt Mode
INH	Inhibit Interrupts
IRTC	Interrupt Return
IRTN	Interrupt Return

KEY MANIPULATION—KEYS

Moves keys to and from registers. See Section 9 for the format of the keys.

▶ **INK R Input Keys**

keys→RH

Save contents of keys in RH. **FORMAT**=RGEN, **OPCODE**=070, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ OTK R Output Keys

RH→keys

Restore keys from RH. **FORMAT**=RGEN, **OPCODE**=071, **C**=loaded by instruction, **L**=loaded by instruction, **CC**=loaded by instruction.

Defined in Section 11:

RCB	Reset C-Bit (Clear)
SCB	Set C-Bit

LOGICAL OPERATIONS—LOGIC

▶ CMH RH Complement Half Register

NOT.RH→RH

Ones complement the contents of RH. **FORMAT**=RGEN, **OPCODE**=045, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ CMR R Complement Register

NOT.R→R

Ones complement the contents of R. **FORMAT**=RGEN, **OPCODE**=044, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ N R,addr AND Fullword

R.AND.|EA|32→R

AND the contents of R and addr and put the result into R. **FORMAT**=MRGR, **OPCODE**=03, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ NH R,addr AND Halfword

RH.AND.|EA|16→RH

AND the contents of RH and addr and put the result into RH. **FORMAT**=MRGR, **OPCODE**=13, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ O R,addr OR Fullword

R.OR.|EA|32→R

OR the contents of R and addr and put the result into R. **FORMAT**=MRGR, **OPCODE**=23, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ OH R,addr OR Halfword

RH.OR.|EA|16→RH

OR the contents of RH and addr and put the result into RH. **FORMAT**=MRGR, **OPCODE**=33, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ X R,addr Exclusive OR Fullword

R.XOR.|EA|32→R

Exclusive OR the contents of R and addr and put the result into R. **FORMAT**=MRGR, **OPCODE**=43, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ XH R,addr Exclusive OR Halfword

RH.XOR.|EA|16→RH

Exclusive OR the contents of RH and addr and put the result into RH. **FORMAT**=MRGR, **OPCODE**=53, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

LOGICAL TEST AND SET—LTSTS

Logical Test and Set (Logicize)

If the test is satisfied, then set the register equal to 1. If the test is not satisfied, then set the register equal to 0. These instructions simplify the analysis of complex logical expressions. The general format is:

If { **Condition Codes (C)** / **Register (blank)** / **Half Register (H)** / **Floating-Point [F] Register** } { **LT** / **LE** / **EQ** / **NE** / **GE** / **GT** } **0, then 1→R; else 0→R**

For example: LCLT R means, if the condition code is less than zero then set R equal to one, else set R equal to zero.

Mnemonic	Function	Opcode
LCLT R	If CC<0, then 1 → R; else 0 → R	150
LCLE R	If CC ⩽ 0, then 1 → R; else 0 → R	151
LCEQ R	If CC=0, then 1 → R; else 0 → R	153
LCNE R	If CC≠0, then 1 → R; else 0 → R	152
LCGE R	If CC ⩾ 0, then 1 → R; else 0 → R	154
LCGT R	If CC>0, then 1 → R; else 0 → R	155

FORMAT=RGEN, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

Mnemonic	Function	Opcode
LLT R	If R<0, then 1 → R; else 0 → R	000
LLE R	If R ⩽ 0, then 1 → R; else 0 → R	001
LEQ R	If R=0, then 1 → R; else 0 → R	003
LNE R	If R≠0, then 1 → R; else 0 → R	002
LGE R	If R ⩾ 0, then 1 → R; else 0 → R	004
LGT R	If R>0, then 1 → R; else 0 → R	005
LHLT R	If RH<0, then 1 → R; else 0 → R	000
LHLE R	If RH ⩽ 0, then 1 → R; else 0 → R	011
LHEQ R	If RH=0, then 1 → R; else 0 → R	013
LHNE R	If RH≠0, then 1 → R; else 0 → R	012
LHGE R	If RH ⩾ 0, then 1 → R; else 0 → R	004
LHGT R	If RH>0, then 1 → R; else 0 → R	015
LFLT R	If F<0, then 1 → R; else 0 → R	020,030
LFLE R	If F ⩽ 0, then 1 → R; else 0 → R	021,031
LFEQ R	If F=0, then 1 → R; else 0 → R	023,033
LFNE R	If F≠0, then 1 → R; else 0 → R	022,032
LFGE R	If F ⩾ 0, then 1 → R; else 0 → R	024,034
LFGT R	If F>0, then 1 → R; else 0 → R	025,035

FORMAT=RGEN, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ LF R Logic set False

Set R equal to zero. **FORMAT**=RGEN, **OPCODE**=016, **C**=unchanged, **L**=unchanged, **CC**= result.

▶ LT R Logic set True

Set R equal to one. **FORMAT**=RGEN, OPCODE=017, **C**=unchanged, **L**=unchanged, **CC**= result.

MACHINE CONTROL—MCTL

Defined in Section 11.

CXCS	Control Extended Control Store
HLT	Halt
ITLB	Invalidate STLB entry
LIOT	Load TLB
LPID	Load Process ID
LPSW	Load Program Status Word
LWCS	Load Writable Control Store
NOP	No Operation
PTLB	Purge TLB
RRST	Restore Registers
RSAV	Register Save
STPM	Store Process Model Number
WCS	Writable Control Store
XVRY	Verify xis

MOVE DATA—MOVE

These instructions move data from one location to another.

▶ I R,addr Interchange Register and Memory—Fullword

R⇔|EA|32

Swap the contents of R and addr. **FORMAT**=MRGR, **OPCODE**=41, **C**=unchanged, **L**= unchanged, **CC**=unchanged.

▶ ICBL R Interchange Bytes and Clear Left

RH(1–8)⇔RH(9–16); 0→|RH(1–8)|

Swap bits 1–8 and bits 9–16 of RH. Then set bits 1–8=0. **FORMAT**=RGEN, **OPCODE**=065, **C** =unchanged, **L**=unchanged, **CC**=unchanged.

▶ ICBR R Interchange Bytes and Clear Right

RH(9–16)⇔RH(1–8);0→RH(9–16)

Swap bits 9–16 and bits 1–8 of RH. Then set bits 9–16=0. **FORMAT**=RGEN, **OPCODE**=066, **C** =unchanged, **L**=unchanged, **CC**=unchanged.

▶ ICHL R Interchange Halfwords and Clear Left

RH⇔RL;0→RH

Swap halves of R and set RH=0. **FORMAT**=RGEN, **OPCODE**=060, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ICHR R Interchange Register Halfwords and Clear Right

RH⇔RL;0→RL

Swap halves of R and set RL=0. **FORMAT**=RGEN, **OPCODE**=061, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ IH R,addr Interchange Register and Memory—Halfword

RH⇔[EA]16

Swap the contents of RH and addr. **FORMAT**=MRGR, **OPCODE**=51, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ IRB R Interchange Register Bytes

RH(1-8)⇔RH(9-16)

Swap bits 1-8 of RH with bits 9-16 of RH. **FORMAT**=RGEN, **OPCODE**=064, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ IRH R Interchange Register Halves

RH⇔RL

Swap halves of R. **FORMAT**=RGEN, **OPCODE**=057, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ L R,addr Load Fullword

[EA]32→R

Load the contents of addr into R. **FORMAT**=MRGR, **OPCODE**=01, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LDAR R,addr Load Addressed Register

Stores the contents of R into the register specified by addr. There are three special cases of this instruction which are summarized in Section 11 under LDLR. **FORMAT**=MRGR, **OPCODE**=44, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LH R,addr Load Halfword

[EA]16→RH

Load the contents of addr into RH. **FORMAT**=MRGR, **OPCODE**=11, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LHL1 R,addr Load Halfword Left Shifted by 1

[EA]16.LS.1→RH

Left shift the contents of addr by 1 and put the result into RH. **FORMAT**=MRGR, **OPCODE**=04, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ LHL2 R,addr Load Halfword Left Shifted by 2

[EA]16.LS.2→RH

Left shift the contents of addr by 2 and put the result into RH. **FORMAT**=MRGR, **OPCODE** =14, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ ST R,addr Store Fullword

R→[EA]32

Store the contents of R into addr. **FORMAT**=MRGR, **OPCODE**=21, **C**=unchanged, **L**= unchanged, **CC**=unchanged.

▶ STAR R,addr Store Addressed Register

Stores the contents of the register specified by the contents of addr into R. There are three special cases of this instruction which are summarized in Section 11 under LDLR. **FORMAT** =MRGR, **OPCODE**=54, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ STCD R,addr Store Conditional Fullword

If R+1=[EA]32 then R→[EA]32

If the contents of R+1 equals the contents of addr, then store the contents of R into addr. **FORMAT**=MRGR, **OPCODE**=137, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ STCH R,addr Store Conditional Halfword

If RL=[EA]16 then RH→[EA]16

If the contents of RL equal the contents of addr, then store the contents of RH into addr. **FORMAT**=MRGR, **OPCODE**=136, **C**=unchanged, **L**=unchanged, **CC**=results.

▶ STH R,addr Store Halfword

RH→[EA]16

Store the contents of RH into addr. **FORMAT**=MRGR, **OPCODE**=31, **C**=unchanged, **L**= unchanged, **CC**=unchanged.

PROGRAM CONTROL AND JUMP—PCTLJ

These instructions either transfer control to a different location or manipulate effective addresses. They differ from branch instructions in the ability to move across segments. They differ among themselves in the complexity of operations performed and in the handling of the return address.

▶ EALB addr Effective Address to Link Base

EA→LB

Store the effective address of addr inthe link base register. **FORMAT**=MRNR, **OPCODE**=42, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ EAR R,addr Effective Address to Register

EA→R

Store the effective address of addr in R. **FORMAT**=MRGR, **OPCODE**=63, **C**=unchanged, **L** =unchanged, **CC**=unchanged.

▶ EAXB addr Effective Address to Temporary Base

EA→XB

Store the effective address of addr in the temporary base register. **FORMAT**=MRNR, **OPCODE**=52, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JMP addr Jump

EA→PC

Jump to addr. **FORMAT**=MRNR, **OPCODE**=51, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JSR R,addr Jump to Subroutine

PC(16–32)→RH;EA→PC

Jump to addr and save the 16-bit word number position of the return address in RH. **FORMAT**=MRGR, **OPCODE**=73, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

▶ JSXB addr Jump and Set XB

PC→XB;EA→PC

Jump to addr and save the full 32-bit return address in XB. **FORMAT**=MRNR, **OPCODE**=61, **C**=unchanged, **L**=unchanged, **CC**=unchanged.

Summary of instructions defined in section 11

ARGT	Argument Transfer
CALF	Call Fault Handler
PCL	Procedure Call
PRTN	Procedure Return
STEX	Stack Extend
SVC	Supervisor Call

PROCESS EXCHANGE—PRCEX

Defined in Section 11.

INBC	Interrupt Notify
INBN	Interrupt Notify
INEC	Interrupt Notify
INEN	Interrupt Notify
NFYB	Notify
NFYE	Notify
WAIT	Wait

QUEUE MANAGEMENT—QUEUE

The instructions provided for queue manipulation are register generics with AP-pointer providing the address to the queue control block. See Section 9 for a description of the queue control block.

Data is to or from general register 2 and the results of the operation are given in the condition code bits for later testing.

addr refers to a control block in virtual space. The virtual queue control block differs from the physical in that a segment number is provided instead of a physical address. Ring zero

privilege is required to manipulate physical queues; any non-ring zero attempt to access physical queues will result in a restrict mode violation fault. Also, the ring number determines the privilege of access into both the control block and the data block.

▶ ABQ addr Add to Bottom of Queue

Add the contents of general register 2 to the bottom of the queue defined by the QCB (Queue Control Block) at addr. The condition codes are set EQ if the queue is full, e.g., the word could not be added. **FORMAT**=RGEN, **OPCODE**=134, **C**=unchanged, **L**=unchanged, **CC**= result.

▶ ATQ addr Add to Top of Queue

Add the contents of general register 2 to the top of the queue defined by the QCB at addr. The condition codes are set EQ if the queue is full,. e.g., the word could not be added. **FORMAT**=RGEN, **OPCODE**=135, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ RBQ addr Remove from Bottom of Queue

Remove a single word from the bottom of the queue defined by the QCB at addr, and place it in general register 2. But, if the queue is empty, set general register 2=0 and condition codes EQ. **FORMAT**=RGEN, **OPCODE**=133, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ RTQ addr Remove from Top of Queue

Remove a single word from the top of the queue defined by the QCB at addr, and place it in general register 2. But if the queue is empty, set general register 2=0 and condition codes EQ. **FORMAT**=RGEN, **OPCODE**=132, **C**=unchanged, **L**=unchanged, **CC**=result.

▶ TSTQ addr Test Queue

Set general register 2 to the number of items in the queue defined by the QCB at addr. If the queue is empty, set condition codes EQ. **FORMAT**=RGEN, **OPCODE**=104, **C**=unchanged, **L** =unchanged, **CC**=result.

SHIFT—SHIFT DATA

Register Shifts

▶ ROT R,addr Rotate

Rotate the bits in R. The low order 16 bits of addr tell how many bits to shift, in what direction and whether full or halfword.

Bit 1=0=left

Bit 1=1=right

Bit 2=0=word (32)

Bit 2=1=halfword

Bits 3-10=unused

Bits 11-16=two's complement of number of bits to shift

FORMAT=MRGR, **OPCODE**=24, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ SHA R,addr Shift Arithmetic

Shift R arithmetically. The low order 16 bits of addr tell how many bits to shift, in what direction and whether full or halfword.

Bit 1=0=left

Bit 1=1=right

Bit 2=0=word (32)

Bit 2=1=halfword

Bits 3-10=unused

Bits 11-16=two's complement of number of bits to shift

FORMAT=MRGR, **OPCODE**=15, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ SHL R,addr Shift Logical

Shift R logically. The low order 16 bits of addr tell how many bits to shift, in what direction and whether full or halfword.

Bit 1=0=left

Bit 1=1=right

Bit 2=0=word (32)

Bit 2=1=halfword

Bits 3-10=unused

Bits 11-16=two's complement of number of bits to shift

FORMAT=MRGR, **OPCODE**=05, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ SL1 R Shift Register Left 1

Shift R left one bit logically. **FORMAT**=RGEN, **OPCODE**=072, **C**=shift extension, **L**= unspecified, **CC**=unchanged.

▶ SL2 R Shift Register Left 2

Shift R left two bits logically. **FORMAT**=RGEN, **OPCODE**=073, **C**=shift extension, **L**= unspecified, **CC**=unchanged.

▶ SR1 R Shift Register Right 1

Shift R right one bit logically. **FORMAT**=RGEN, **OPCODE**=074, **C**=shift extension, **L**= unspecified, **CC**=unchanged.

▶ SR2 R Shift Register Right 2

Shift R right two bits logically. **FORMAT**=RGEN, **OPCODE**=075, **C**=shift extension, **L**= unspecified, **CC**=unchanged.

Half Register Shifts

▶ SHL1 R Shift Half Register Left 1

Shift RH left one bit logically. **FORMAT**=RGEN, **OPCODE**=076, **C**=shift extension, **L**= unspecified, **CC**=unchanged.

▶ SHL2 R Shift Half Register Left 2

Shift RH left two bits logically. **FORMAT**=RGEN, **OPCODE**=077, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ SHR1 R Shift Half Register Right 1

Shift RH right one bit logically. **FORMAT**=RGEN, **OPCODE**=120, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

▶ SHR2 R Shift Half Register Right 2

Shift RH right two bits logically. **FORMAT**=RGEN, **OPCODE**=121, **C**=shift extension, **L**=unspecified, **CC**=unchanged.

13 Instruction summary chart

INSTRUCTION SUMMARY

This chart contains a complete list of instructions for the Prime 100 through 500. Each instruction is followed by its octal code, format, function information on addressing mode and hardware availability, and a one line description of the instruction.

The columns in the list are as follows:

R — RESTRICTIONS

blank	regular instruction
R	instruction causes a restricted mode fault if executed in other than right 0
P	instruction may cause a fault depending on address
W	writable control store instruction, may be programmed in wcs to cause a fault
M	Machine specific—use only on specified CPU. Usually an instruction reserved for operating system, such as EPMJ.

MNEM — a mnemonic name recognized by the assembler PMA.

OPCODE — Octal operation code of the instruction. The codes are indented so that I/O instructions are isolated from generics, and the memory reference and register instructions of the P500 are sorted apart from the MR instructions of the P100-400.

RI — Register (R) and Immediate (I) forms available (P500 memory reference instructions only); Y = YES, N = NO.

FORM — Format of instruction:

MNEMONIC	DEFINITION
GEN	Generic
AP	Address Pointer
BRAN	Branch
IBRN	I-mode Branch
CHAR	Character
DECI	Decimal
PIO	Programmed I/O
SHFT	Shift
MR	Memory Reference—non I-mode
MRFR	Memory Reference—Floating Register
MRNR	Memory Reference—Non Register
RGEN	Register Generic

FUNC — Function of instruction

MNEMONIC	DEFINITION
ADMOD	Addressing Mode
BRAN	Branch
CHAR	Character
CLEAR	Clear field
DECI	Decimal Arithmetic
FIELD	Field Register
FLOAT	Floating Point Arithmetic
INT	Integer
INTGY	Integrity
IO	Input/Output
KEYS	Keys
LOGIC	Logical Operations
LTSTS	Logical Test and Set
MCTL	Machine Control
MOVE	Move
PCTLJ	Program Control and Jump
PRCEX	Process Exchange
QUEUE	Queue Control
SHIFT	Register Shift
SKIP	Skip

MODE — Addressing modes in which instruction functions as defined:

S Sectored
R Relative
V 64V (P400–P500)
I 32I (P500)

1 2 3 Column — How instruction is implemented

1 = Prime 100, 200, 300 series
2 = Prime 400 series
3 = Prime 500 series

Codes are:

- \- Not implemented. Do not use this mnemonic on this CPU.
- H Implemented by standard hardware.
- O Implemented by hardware option or UII library if option is not present.
- U Implemented by UII library
- A UII on 100, 200, hardware on 300
- B Optional on 100, 200, hardware on 300
- C Not implemented on 100, optional on 200, 300
- D UII on 100, optional on 200, 300
- E Not implemented on 100, hardware on 200, 300
- F Not implemented on 100, 200, hardware on 300
- G Not implemented on 100, optional on 200, hardware on 300

C — How instruction affects C and L bits, codes are:

- \- C and L are unchanged
- 1 C = unchanged, L = carry
- 2 C = overflow status, L = carry
- 3 C = overflow status, L = unspecified
- 4 C = status extension, L = unspecified
- 5 C = result, L = unspecified

6 C = unspecified, L = unspecified
7 C = loaded by instruction, L = loaded by instruction

CC — How instruction affects condition codes, codes are:
- – condition codes are not altered
- 1 condition codes are set to reflect the result of arithmetic operation or compare
- 4 condition codes are set to reflect result of branch, compare or logicize operand state.
- 5 condition codes are indeterminant
- 6 condition codes are loaded by instruction
- 7 special results are shown in condition codes for this instruction

DESCRIPTION — a brief description of the instruction

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
	A	02	YY	MRGR	INT	**I**	—	—	H	2	1	**Add Fullword**
	A1A	141206		GEN	INT	**SRV**	H	H	H	2	1	**Add One to A**
	A2A	140304		GEN	INT	**SRV**	H	H	H	2	1	**Add Two to A**
	ABQ	141716		AP	QUEUE	**V**	—	H	H	—	7	**Add to Bottom of Queue**
	ABQ	134		AP	QUEUE	**I**	—	—	H	—	7	**Add to Bottom of Queue**
	ACA	141216		GEN	INT	**SRV**	H	H	H	2	1	**Add C-Bit to A**
	ADD	06		MR	INT	**SRV**	H	H	H	2	1	**Add**
	ADL	06 03		MR	INT	**V**	—	H	H	2	1	**Add Long**
	ADLL	141000		GEN	INT	**V**	—	H	H	2	1	**Add Link Bit to L**
	ADLR	014		RGEN	INT	**I**	—	—	H	2	1	**Add Link to R**
	AH	12	YY	MRGR	INT	**I**	—	—	H	2	1	**Add Halfword**
	ALFA 0	001301		GEN	FIELD	**V**	—	H	H	6	5	**Add L to Field Address**
	ALFA 1	001311		GEN	FIELD	**V**	—	H	H	6	5	**Add L to Field Address**
	ALL	0414XX		SHFT	SHIFT	**SRV**	H	H	H	4	—	**A Left Logical**
	ALR	0416XX		SHFT	SHIFT	**SRV**	H	H	H	4	—	**A Left Rotate**
	ALS	0415XX		SHFT	SHIFT	**SRV**	H	H	H	2	—	**A Left Shift**
	ANA	03		MR	LOGIC	**SRV**	H	H	H	—	—	**AND**
	ANL	03 03		MR	LOGIC	**V**	—	H	H	—	—	**AND Long**
	ARFA 0	161		RGEN	FIELD	**I**	—	—	H	6	—	**Add R to Field Address**
	ARFA 1	171		RGEN	FIELD	**I**	—	—	H	6	—	**Add R to Field Address**
	ARGT	000605		GEN	PCTLJ	**VI**	—	H	H	6	5	**Argument Transfer**
	ARL	0404XX		SHFT	SHIFT	**SRV**	H	H	H	4	—	**A Right Logical**
	ARR	0406XX		SHFT	SHIFT	**SRV**	H	H	H	4	—	**A Right Rotate**
	ARS	0405XX		SHFT	SHIFT	**SRV**	H	H	H	4	—	**A Right Shift**
	ATQ	141717		AP	QUEUE	**V**	—	H	H	—	7	**Add to Top of Queue**
	ATQ	135		AP	QUEUE	**I**	—	—	H	—	7	**Add to Top of Queue**
	BCEQ	141602		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if CC = 0**
	BCGE	141605		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if CC ⩾ 0**
	BCGT	141601		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if CC > 0**
	BCLE	141600		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if CC ⩽ 0**
	BCLT	141604		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if CC < 0**
	BCNE	141603		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if CC •NE• 0**
	BCR	141705		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if C-Bit = 0**
	BCS	141704		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if C-Bit = 1**
	BDX	140734		BRAN	BRAN	**V**	—	H	H	—	—	**Decrement X and branch if X •NE• 0**
	BDY	140724		BRAN	BRAN	**V**	—	H	H	—	—	**Decrement Y and branch if Y •NE• 0**
	BEQ	140612		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if A = 0**
	BFEQ	141612		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if F = 0**
	BFEQ	122		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if F = 0**
	BFGE	141615		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if F ⩾ 0**
	BFGE	125		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if F ⩾ 0**
	BFGT	141611		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if F > 0**
	BFGT	121		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if F > 0**
	BFLE	141610		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if F ⩽ 0**
	BFLE	120		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if F ⩽ 0**

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
	BFLT	141614		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if F < 0**
	BFLT	124		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if F < 0**
	BFNE	141613		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if F •NE• 0**
	BFNE	123		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if F •NE• 0**
	BGE	140615		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if A ≥ 0**
	BGT	140611		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if A > 0**
	BHD1	144		IBRN	BRAN	**I**	—	—	H	—	—	**Decrement H by One; Branch if H •NE• 0**
	BHD2	145		IBRN	BRAN	**I**	—	—	H	—	—	**Decrement H by Two; Branch if H •NE• 0**
	BHD4	146		IBRN	BRAN	**I**	—	—	H	—	—	**Decrement H by Four; Branch if H •NE• 0**
	BHEQ	112		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if H = 0**
	BHGE	105		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if H ≥ 0**
	BHGT	111		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if H > 0**
	BHI1	140		IBRN	BRAN	**I**	—	—	H	—	—	**Increment H by One; Branch if H •NE• 0**
	BHI2	141		IBRN	BRAN	**I**	—	—	H	—	—	**Increment H by Two; Branch if H •NE• 0**
	BHI4	142		IBRN	BRAN	**I**	—	—	H	—	—	**Increment H by One; Branch if H •NE• 0**
	BHLE	110		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if H ≤ 0**
	BHLT	104		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if H < 0**
	BHNE	113		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if H is not equal to 0**
	BIX	141334		BRAN	BRAN	**V**	—	H	H	—	—	**Increment X and Branch if X •NE• 0**
	BIY	141324		BRAN	BRAN	**V**	—	H	H	—	—	**Increment Y and Branch if Y •NE• 0**
	BLE	140610		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if A ≤ 0**
	BLEQ	140702		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if L = 0**
	BLGE	140615		BRAN	BRAN	**V**	—	H	H	—	4	**Branch is L ≥ 0**
	BLGT	140701		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if L > 0**
	BLLE	140700		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if L ≤ 0**
	BLLT	140614		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if L < 0**
	BLNE	140703		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if L •NE• 0**
	BLR	141707		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if L-Bit = 0**
	BLS	141706		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if L-Bit = 1 (Set)**
	BLT	140614		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if A < 0**
	BMEQ	141602		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if Magnitude = 0**
	BMGE	141706		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if Magnitude is ≥ 0**
	BMGT	141710		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if Magnitude is > 0**
	BMLE	141711		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if Magnitude is ≤ 0**
	BMLT	141707		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if Magnitude is < 0**
	BMNE	141603		BRAN	BRAN	**VI**	—	H	H	—	—	**Branch if Magnitude is •NE• 0**
	BNE	140613		BRAN	BRAN	**V**	—	H	H	—	4	**Branch if A •NE• 0**
	BRBR	040-077		IBRN	BRAN	**I**	—	—	H	—	—	**Branch if R bit n = 0**
	BRBS	000-037		IBRN	BRAN	**I**	—	—	H	—	—	**Branch if R bit n = 1**
	BRD1	134		IBRN	BRAN	**I**	—	—	H	—	—	**Decrement R by One; Branch if R •NE• 0**
	BRD2	135		IBRN	BRAN	**I**	—	—	H	—	—	**Decrement R by Two; Branch if R •NE• 0**
	BRD4	136		IBRN	BRAN	**I**	—	—	H	—	—	**Decrement R by Four; Branch if R •NE• 0**
	BREQ	102		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if R = 0**
	BRGE	105		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if R ≥ 0**
	BRGT	101		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if R > 0**
	BRI1	130		IBRN	BRAN	**I**	—	—	H	—	—	**Increment R by one and branch if •NE• 0**
	BRI2	131		IBRN	BRAN	**I**	—	—	H	—	—	**Increment R by 2 and branch if •NE• 0**
	BRI4	132		IBRN	BRAN	**I**	—	—	H	—	—	**Increment R by 4 and branch if •NE• 0**
	BRLE	100		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if R ≤ 0**
	BRLT	104		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if R < 0**
	BRNE	103		IBRN	BRAN	**I**	—	—	H	—	4	**Branch if R •NE• 0**
	C	61	YY	MRGR	INT	**I**	—	—	H	1	1	**Compare Fullword**
R	**CAI**	000411		GEN	IO	**SRVI**	H	H	H	—	—	**Clear Active Interrupt**
	CAL	141050		GEN	CLEAR	**SRV**	H	H	H	—	—	**Clear A Left**
	CALF	000705		AP	PCTLJ	**VI**	—	H	H	6	5	**Call Fault Handler**
	CAR	141044		GEN	CLEAR	**SRV**	H	H	H	—	—	**Clear A Right Byte**
	CAS	11		MR	SKIP	**SRV**	H	H	H	1	1	**Compare A and Skip**
	CAZ	140214		GEN	SKIP	**SRV**	H	H	H	1	1	**Compare A with Zero**
	CEA	000111		GEN	PCTLJ	**SR**	H	H	H	—	—	**Compute Effective Address**
	CGT	001314		GEN	BRAN	**V**	—	H	H	6	5	**Computed GOTO**
	CGT	026		RGEN	BRAN	**I**	—	—	H	—	7	**Computed GOTO**
	CH	71	YY	MRGR	INT	**I**	—	—	H	1	1	**Compare Halfword**
	CHS	140024		GEN	INT	**SRV**	H	H	H	—	—	**Change Sign**

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
	CHS	040		RGEN	INT	**I**	—	—	H	—	—	**Change Sign**
	CLS	11 03		MR	LOGIC	**V**	—	H	H	1	1	**Compare L and Skip**
	CMA	140401		GEN	LOGIC	**SRV**	H	H	H	—	—	**Complement A**
	CMH	045		RGEN	LOGIC	**I**	—	—	H	—	—	**Complement H**
	CMR	44		RGEN	LOGIC	**I**	—	—	H	—	—	**Complement R**
	CR	056		RGEN	CLEAR	**I**	—	—	H	—	—	**Clear**
	CRA	140040		GEN	CLEAR	**SRV**	H	H	H	—	—	**Clear A**
	CRB	140015		GEN	CLEAR	**SRV**	H	H	H	—	—	**Clear B**
	CRBL	062		RGEN	CLEAR	**I**	—	—	H	—	—	**Clear High Byte 1 Left**
	CRBR	063		RGEN	CLEAR	**I**	—	—	H	—	—	**Clear High Byte 2 Right**
	CRE	141404		GEN	CLEAR	**V**	—	H	H	—	—	**Clear E**
	CREP	10 02		MR	PCTLJ	**R**	A	H	H	—	—	**Call Recursive Entry Procedure**
	CRHL	054		RGEN	CLEAR	**I**	—	—	H	—	—	**Clear Left Half Register**
	CRHR	055		RGEN	CLEAR	**I**	—	—	H	—	—	**Clear Right Half Register**
	CRL	140010		GEN	CLEAR	**SRV**	H	H	H	—	—	**Clear L**
	CRLE	141410		GEN	CLEAR	**V**	—	H	H	—	—	**Clear L and E**
	CSA	140320		GEN	MOVE	**SRV**	H	H	H	5	—	**Copy Sign of A**
	CSR	041		RGEN	MOVE	**I**	—	—	H	5	—	**Copy Sign of R**
R	**CXCS**	001714		GEN	MCTL	**VI**	—	H	H	6	5	**Control Extended Control Store**
	D	62	YY	MRGR	INT	**I**	—	—	H	3	5	**Divide Fullword**
	DAD	06		MR	INT	**SR**	B	H	H	2	1	**Double Add**
	DBL	000007		GEN	INT	**SR**	H	H	H	—	—	**Enter Double Precision Mode**
	DBLE	106		RGEN	FLPT	**I**	—	—	H	—	—	**Convert Single to Double Float**
	DFA	15,17	YY	MRFR	FLPT	**I**	—	—	H	3	5	**Double Floating Add**
	DFAD	06 02		MR	FLPT	**RV**	A	H	H	3	5	**Double Floating Add**
	DFC	05,07	YY	MRFR	FLPT	**I**	—	—	H	—	1	**Double Floating Compare**
	DFCM	140574		GEN	FLPT	**RV**	C	H	H	3	5	**Double Floating Complement**
	DFCM	144		RGEN	FLPT	**I**	—	—	H	3	5	**Double Floating Complement**
	DFCS	11 02		MR	FLPT	**RV**	A	H	H	6	5	**Double Floating Compare and Skip**
	DFD	31,33	YY	MRFR	FLPT	**I**	—	—	H	3	5	**Double Floating Divide**
	DFDV	17 02		MR	FLPT	**RV**	D	H	H	3	5	**Double Floating Divide**
	DFL	01,03	YY	MRFR	FLPT	**I**	—	—	H	—	—	**Double Floating Load**
	DFLD	02 02		MR	FLPT	**RV**	A	H	H	—	—	**Double Floating Load**
	DFLX	15 02		MR	FLPT	**V**	—	H	H	—	—	**Load Double Floating Index**
	DFM	25,27	YY	MRFR	FLPT	**I**	—	—	H	3	5	**Double Floating Multiply**
	DFMP	16 02		MR	FLPT	**RV**	D	H	H	3	5	**Double Floating Multiply**
	DFS	21,23	YY	MRFR	FLPT	**I**	—	—	H	3	5	**Double Floating Subtract**
	DFSB	07 02		MR	FLPT	**RV**	A	H	H	3	5	**Double Floating Subtract**
	DFST	11,13	NN	MRFR	FLPT	**I**	—	—	H	—	—	**Double Floating Store**
	DFST	04 02		MR	FLPT	**RV**	A	H	H	—	—	**Double Floating Store**
	DH	72	YY	MRGR	INT	**I**	—	—	H	3	5	**Divide Halfword**
	DH1	130		RGEN	INT	**I**	—	—	H	2	1	**Decrement H by 1**
	DH2	131		RGEN	INT	**I**	—	—	H	2	1	**Decrement H by 2**
	DIV	17		MR	INT	**V**	—	H	H	3	5	**Divide**
	DIV	17		MR	INT	**SR**	B	H	H	3	5	**Divide**
	DLD	02		MR	MOVE	**SR**	B	H	H	—	—	**Double Load**
	DM	60	NN	MRNR	INT	**I**	—	—	H	—	1	**Decrement Fullword**
	DMH	70	NN	MRNR	INT	**I**	—	—	H	—	1	**Decrement Halfword**
	DR1	124		RGEN	INT	**I**	—	—	H	2	1	**Decrement R by One**
	DR2	125		RGEN	INT	**I**	—	—	H	2	1	**Decrement R by Two**
	DRX	140210		GEN	SKIP	**SRV**	H	H	H	—	—	**Decrement and Replace X**
	DSB	07		MR	INT	**SR**	B	H	H	2	1	**Double Subtract**
	DST	04		MR	MOVE	**SR**	B	H	H	—	—	**Double Store**
	DVL	17 03		MR	INT	**V**	—	H	H	3	5	**Divide Long**
	E16S	000011		GEN	ADMOD	**SRVI**	H	H	H	—	—	**Enter 16S Mode**
	E32I	001010		GEN	ADMOD	**SRVI**	—	—	H	—	—	**Enter 32I Mode**
	E32R	001013		GEN	ADMOD	**SRVI**	H	H	H	—	—	**Enter 32R Mode**
	E32S	000013		GEN	ADMOD	**SRVI**	H	H	H	—	—	**Enter 32S Mode**
	E64R	001011		GEN	ADMOD	**SRVI**	H	H	H	—	—	**Enter 64R Mode**
	E64V	000010		GEN	ADMOD	**SRVI**	—	H	H	—	—	**Enter 64V Mode**
	EAA	01 01		MR	MOVE	**R**	A	H	H	—	—	**Effective Address to A**
	EAFA 0	001300		AP	FIELD	**VI**	—	H	H	—	—	**Effective Address to Field Address Register 0**

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
	EAFA 1	001310		AP	FIELD	**VI**	—	H	H	—	—	**Effective Address to Field Address Register 1**
	EAL	01 01		MR	PCTLJ	**V**	—	H	H	—	—	**Effective Address to L**
	EALB	42	NN	MRNR	PCTLJ	**I**	—	—	H	—	—	**Effective Address to LB**
	EALB	13 02		MR	PCTLJ	**V**	—	H	H	—	—	**Effective Address to LB**
	EAR	63	NN	MRGR	PCTLJ	**I**	—	—	H	—	—	**Effective Address to R**
	EAXB	52	NN	MRNR	PCTLJ	**I**	—	—	H	—	—	**Effective Address to XB**
	EAXB	12 02		MR	PCTLJ	**V**	—	H	H	—	—	**Effective Address to XB**
R	**EIO**	34	NN	MRGR	IO	**I**	—	—	H	—	7	**Execute I/O**
R	**EIO**	14 01		MR	IO	**V**	—	H	H	—	7	**Execute I/O**
R	**EMCM**	000503		GEN	INTGY	**SRVI**	E	H	H	—	—	**Enter Machine Check Mode**
R	**ENB**	000401		GEN	IO	**SRVI**	H	H	H	—	—	**Enable Interrupts**
	ENTR	01 03		MR	PCTLJ	**R**	A	H	H	—	—	**Enter Recursive Procedure Stack**
	EPMJ	000217		MR	MCTL	**SR**	H	—	—	—	—	**Enter Paging Mode and Jump**
R	**EPMX**	000237		MR	MCTL	**SR**	H	—	—	—	—	**Enter Paging Mode and Jump to XCS**
	ERA	05		MR	LOGIC	**SRV**	H	H	H	—	—	**Exclusive OR to A**
	ERL	05 03		MR	LOGIC	**V**	—	H	H	—	—	**Exclusive OR to L**
R	**ERMJ**	000701		MR	MCTL	**SR**	H	—	—	—	—	**Enter Restricted Execution Mode and Jump**
R	**ERMX**	000721		MR	MCTL	**SR**	H	—	—	—	—	**Enter Restricted Execution Mode and Jump to WCS**
R	**ESIM**	000415		GEN	IO	**SRVI**	H	H	H	—	—	**Enter Standard Interrupt Mode**
R	**EVIM**	000417		GEN	IO	**SRVI**	H	H	H	—	—	**Enter Vectored Interrupt Mode**
R	**EVMJ**	000703		MR	MCTL	**SR**	F	—	—	—	—	**Enter Vectored Mode and Jump**
R	**EVMX**	000723		MR	MCTL	**SR**	F	—	—	—	—	**Enter Virtual Mode and Jump to WCS**
	FA	14,16	YY	MRFR	FLPT	**I**	—	—	H	3	5	**Floating Add**
	FAD	06 01		MR	FLPT	**RV**	A	H	H	3	5	**Floating Add**
	FC	04,06	YY	MRFR	FLPT	**I**	—	—	H	—	1	**Floating Compare**
	FCM	140530		GEN	FLPT	**RV**	C	H	H	3	5	**Floating Complement**
	FCM	100		RGEN	FLPT	**I**	—	—	H	3	5	**Floating Complement**
	FCS	11 01		MR	FLPT	**RV**	A	H	H	6	5	**Floating Compare and Skip**
	FD	30,32	YY	MRFR	FLPT	**I**	—	—	H	3	5	**Floating Divide**
	FDBL	140016		GEN	FLPT	**V**	—	H	H	—	—	**Convert Single to Double Float**
	FDV	17 01		MR	FLPT	**RV**	D	H	H	3	5	**Floating Divide**
	FL	00,02	YY	MRFR	FLPT	**I**	—	—	H	—	—	**Floating Load**
	FLD	02 01		MR	FLPT	**RV**	A	H	H	—	—	**Floating Load**
	FLOT	140550		GEN	FLPT	**R**	C	H	H	3	5	**Convert 31-Bit Integer to Float**
	FLT	105,115		RGEN	FLPT	**I**	—	—	H	3	5	**Convert Integer to Floating**
	FLTA	140532		GEN	FLPT	**V**	—	H	H	3	5	**Convert Integer to Floating**
	FLTH	102,112		RGEN	FLPT	**I**	—	—	H	3	5	**Convert Halfword to Floating**
	FLTL	140535		GEN	FLPT	**V**	—	H	H	3	5	**Convert Long Integer to Floating**
	FLX	15 01		MR	FLPT	**RV**	A	H	H	—	—	**Load Double Word Index**
	FM	24,26	YY	MRFR	FLPT	**I**	—	—	H	3	5	**Floating Multiply**
	FMP	16 01		MR	FLPT	**RV**	D	H	H	3	5	**Floating Multiply**
	FRN	140534		GEN	FLPT	**RV**	D	H	H	3	5	**Floating Round**
	FRN	107		RGEN	FLPT	**I**	—	—	H	3	5	**Floating Round**
	FS	20,22	YY	MRFR	FLPT	**I**	—	—	H	3	5	**Floating Subtract**
	FSB	07 01		MR	FLPT	**RV**	A	H	H	3	5	**Floating Subtract**
	FSGT	140515		GEN	FLPT	**RV**	C	H	H	—	—	**Floating Skip if > 0**
	FSLE	140514		GEN	FLPT	**RV**	C	H	H	—	—	**Floating Skip ≤ 0**
	FSMI	140512		GEN	FLPT	**RV**	C	H	H	—	—	**Floating Skip if Minus**
	FSNZ	140511		GEN	FLPT	**RV**	C	H	H	—	—	**Floating Skip if Not Zero**
	FSPL	140513		GEN	FLPT	**RV**	C	H	H	—	—	**Floating Skip if Plus**
	FST	10,12	NN	MRFR	FLPT	**I**	—	—	H	3	—	**Floating Store**
	FST	04 01		MR	FLPT	**RV**	A	H	H	3	—	**Floating Store**
	FSZE	140510		GEN	FLPT	**RV**	C	H	H	—	—	**Floating Skip if Zero**
R	**HLT**	000000		GEN	MCTL	**SRVI**	H	H	H	—	—	**Halt**
	I	41	YN	MRGR	MOVE	**I**	—	—	H	—	—	**Interchange Register and Memory-Fullword**
	IAB	000201		GEN	MOVE	**SRV**	H	H	H	—	—	**Interchange A and B**
	ICA	141340		GEN	MOVE	**SRV**	H	H	H	—	—	**Interchange Characters in A**
	ICBL	065		RGEN	MOVE	**I**	—	—	H	—	—	**Interchange Bytes and Clear Left**
	ICBR	066		RGEN	MOVE	**I**	—	—	H	—	—	**Interchange Bytes and Clear Right**

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
	ICHL	060		RGEN	MOVE	I	—	—	H	—	—	Interchange Halves and Clear Left
	ICHR	061		RGEN	MOVE	I	—	—	H	—	—	Interchange Halves and Clear Right
	ICL	141140		GEN	MOVE	SRV	H	H	H	—	—	Interchange and Clear Left
	ICR	141240		GEN	MOVE	SRV	H	H	H	—	—	Interchange and Clear Right
	IH	51	YN	MRGR	MOVE	I	—	—	H	—	—	Interchange Register and Memory Halfword
	IH1	126		RGEN	INT	I	—	—	H	2	1	Increment by One
	IH2	127		RGEN	INT	I	—	—	H	2	1	Increment by Two
	ILE	141414		GEN	MOVE	V	—	H	H	—	—	Interchange L and E
	IM	40	NN	MRNR	INT	I	—	—	H	—	1	Increment Fullword
	IMA	13		MR	MOVE	SRV	H	H	H	—	—	Interchange Memory and A
	IMH	50	NN	MRNR	INT	I	—	—	H	—	1	Increment Halfword
R	INA	54		PIO	IO	SR	H	H	H	—	—	Input to A
R	INBC	001217		AP	PRCEX	VI	—	H	H	6	5	Interrupt Notify
R	INBN	001215		AP	PRCEX	VI	—	H	H	6	5	Interrupt Notify
R	INEC	001216		AP	PRCEX	VI	—	H	H	6	5	Interrupt Notify
R	INEN	001214		AP	PRCEX	VI	—	H	H	6	5	Interrupt Notify
R	INH	001001		GEN	IO	SRVI	H	H	H	—	—	Inhibit Interrupts
	INK	000043		GEN	KEYS	SR	H	H	H	—	—	Input Keys
	INK	070		RGEN	KEYS	I	—	—	H	—	—	Save Keys
	INT	140554		GEN	FLPT	R	C	H	H	3	5	Convert Floating to Integer
	INT	103,113		RGEN	FLPT	I	—	—	H	3	5	Convert Floating to Integer
	INTA	140531		GEN	FLPT	V	—	H	H	3	5	Convert Floating to Integer
	INTH	101,111		RGEN	FLPT	I	—	—	H	3	5	Convert Floating to Halfword Integer
	INTL	140533		GEN	FLPT	V	—	H	H	3	5	Convert Floating to Integer Long
	IR1	122		RGEN	INT	I	—	—	H	2	1	Increment R by One
	IR2	123		RGEN	INT	I	—	—	H	2	1	Increment R by Two
	IRB	064		RGEN	MOVE	I	—	—	H	—	—	Interchange Bytes
	IRH	057		RGEN	MOVE	I	—	—	H	—	—	Interchange Halves
	IRS	12		MR	SKIP	SRV	H	H	H	—	—	Increment Memory Replace and Skip
R	IRTC	000603		GEN	IO	VI	—	H	H	7	6	Interrupt Return
R	IRTN	000601		GEN	IO	VI	—	H	H	7	6	Interrupt Return
	IRX	140114		GEN	SKIP	SRV	H	H	H	—	—	Increment and Replace X
R	ITLB	000615		GEN	MCTL	VI	—	H	H	—	—	Invalidate STLB entry
	JDX	15 02		MR	PCTLJ	R	A	H	H	—	—	Jump and Decrement X
	JEQ	02 03		MR	PCTLJ	R	A	H	H	—	—	Jump if = 0
	JGE	07 03		MR	PCTLJ	R	A	H	H	—	—	Jump if ⩾ 0
	JGT	05 03		MR	PCTLJ	R	A	H	H	—	—	Jump if > 0
	JIX	15 03		MR	PCTLJ	R	A	H	H	—	—	Jump and Increment X
	JLE	04 03		MR	PCTLJ	R	A	H	H	—	—	Jump if ⩽ 0
	JLT	06 03		MR	PCTLJ	R	A	H	H	—	—	Jump if < 0
	JMP	51	NN	MRNR	PCTLJ	I	—	—	H	—	—	Jump
	JMP	01		MR	PCTLJ	SRV	H	H	H	—	—	Jump
	JNE	03 03		MR	PCTLJ	R	A	H	H	—	—	Jump if •NE• 0
	JSR	73	NN	MRGR	PCTLJ	I	—	—	H	—	—	Jump to Subroutine
	JST	10		MR	PCTLJ	SRV	H	H	H	—	—	Jump and Store PC
	JSX	35 03		MR	PCTLJ	RV	H	H	H	—	—	Jump and Store Return in X
	JSXB	61	NN	MRNR	PCTLJ	I	—	—	H	—	—	Jump and Store Return in XB
	JSXB	14 02		MR	PCTLJ	V	—	H	H	—	—	Jump and Store Return in XB
	JSY	14		MR	PCTLJ	V	—	H	H	—	—	Jump and Store Return in Y
	L	01	YY	MRGR	MOVE	I	—	—	H	—	—	Load
	LCEQ	141503		GEN	LTSTS	V	—	H	H	—	—	Test CC Equal to 0 and Set A
	LCEQ	153		RGEN	LTSTS	I	—	—	H	—	—	Test CC = 0 and Set R
	LCGE	141504		GEN	LTSTS	V	—	H	H	—	—	Test CC ⩾ 0 and Set A
	LCGE	154		RGEN	LTSTS	I	—	—	H	—	—	Test CC ⩾ 0 and Set R
	LCGT	141505		GEN	LTSTS	V	—	H	H	—	—	Test CC > 0 and Set A
	LCGT	155		RGEN	LTSTS	I	—	—	H	—	—	Test CC > 0 and Set R
	LCLE	141501		GEN	LTSTS	V	—	H	H	—	—	Test CC ⩽ 0 and Set A
	LCLE	151		RGEN	LTSTS	I	—	—	H	—	—	Test CC ⩽ 0 and Set R
	LCLT	141500		GEN	LTSTS	V	—	H	H	—	—	Test CC < 0 and Set A
	LCLT	150		RGEN	LTSTS	I	—	—	H	—	—	Test CC < 0 and Set R
	LCNE	141502		GEN	LTSTS	V	—	H	H	—	—	Test CC •NE• 0 and Set A
	LCNE	152		RGEN	LTSTS	I	—	—	H	—	—	Test CC •NE• 0 and Set R

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
	LDA	02		MR	MOVE	SRV	H	H	H	—	—	Load A
	LDAR	44	NN	MRGR	MOVE	I	—	—	H	—	—	Load From Addressed Register
	LDC 0	162		RGEN	CHAR	I	—	—	H	—	7	Load Character
	LDC 1	172		RGEN	CHAR	I	—	—	H	—	7	Load Character
	LDC 0	001302		CHAR	CHAR	V	—	H	H	—	7	Load Character
	LDC 1	001312		CHAR	CHAR	V	—	H	H	—	7	Load Character
	LDL	02 03		MR	MOVE	V	—	H	H	—	—	Load Long
P	LDLR	05 01		MR	MOVE	V	—	H	H	—	—	Load From Addressed Register
	LDX	35		MR	MOVE	SRV	H	H	H	—	—	Load X
	LDY	35 01		MR	MOVE	V	—	H	H	—	—	Load Y
	LEQ	140413		GEN	LTSTS	SRV	H	H	H	—	4	Test A = 0; Set A
	LEQ	003		RGEN	LTSTS	I	—	—	H	—	4	Test R = 0; Set R
	LF	140416		GEN	LTSTS	SRV	H	H	H	—	4	Logic Set A False
	LF	016		RGEN	LTSTS	I	—	—	H	—	4	Logic Set R False
	LFEQ	141113		GEN	LTSTS	V	—	H	H	—	4	Test F = 0; Set A
	LFEQ	023,033		RGEN	LTSTS	I	—	—	H	—	4	Test F = 0; Set R
	LFGE	141114		GEN	LTSTS	V	—	H	H	—	4	Test F ⩾ 0; Set A
	LFGE	024,034		RGEN	LTSTS	I	—	—	H	—	4	Test F ⩾ 0; Set R
	LFGT	141115		GEN	LTSTS	V	—	H	H	—	4	Test F > 0; Set A
	LFGT	025,035		RGEN	LTSTS	I	—	—	H	—	4	Test F > 0; Set R
	LFLE	141111		GEN	LTSTS	V	—	H	H	—	4	Test F ⩽ 0; Set A
	LFLE	021,031		RGEN	LTSTS	I	—	—	H	—	4	Test F ⩽ 0; Set R
	LFLI 0	001303		BRAN	FIELD	VI	—	H	H	—	—	Load Field Length Register 0
	LFLI 1	001313		BRAN	FIELD	VI	—	H	H	—	—	Load Field Length Register 1
	LFLT	141110		GEN	LTSTS	V	—	H	H	—	4	Test F < 0; Set A
	LFLT	020,030		RGEN	LTSTS	I	—	—	H	—	4	Test F < 0; Set R
	LFNE	141112		GEN	LTSTS	V	—	H	H	—	4	Test F •NE• 0; Set A
	LFNE	022,032		RGEN	LTSTS	I	—	—	H	—	4	Test F •NE• 0; Set R
	LGE	140414		GEN	LTSTS	SRV	H	H	H	—	4	Test A ⩾ 0; Set A
	LGE	004		RGEN	LTSTS	I	—	—	H	—	4	Test R ⩾ 0; Set R
	LGT	140415		GEN	LTSTS	SRV	H	H	H	—	4	Test A > 0; Set A
	LGT	005		RGEN	LTSTS	I	—	—	H	—	4	Test R > 0; Set R
	LH	11	YY	MRGR	MOVE	I	—	—	H	—	—	Load Halfword
	LHEQ	013		RGEN	LTSTS	I	—	—	H	—	4	Test H = 0; Set H
	LHGE	004		RGEN	LTSTS	I	—	—	H	—	4	Test H ⩾ 0; Set H
	LHGT	015		RGEN	LTSTS	I	—	—	H	—	4	Test H > 0; Set H
	LHL1	04	YN	MRGR	MOVE	I	—	—	H	—	—	Load Halfword Left Shifted by 1
	LHL2	14	YN	MRGR	MOVE	I	—	—	H	—	—	Load Halfword Left Shifted by 2
	LHLE	011		RGEN	LTSTS	I	—	—	H	—	4	Test H ⩽ 0; Set H
	LHLT	000		RGEN	LTSTS	I	—	—	H	—	4	Test H < 0; Set H
	LHNE	012		RGEN	LTSTS	I	—	—	H	—	4	Test H •NE• 0; Set H
R	LIOT	000044		AP	MCTL	VI	—	—	H	6	5	Load I/O TLB (Prime 750 only)
	LLE	140411		GEN	LTSTS	SRV	H	H	H	—	4	Test A ⩽ 0; Set A
	LLE	001		RGEN	LTSTS	I	—	—	H	—	4	Test R ⩽ 0; Set R
	LLEQ	141513		GEN	LTSTS	V	—	H	H	—	4	Test L = 0; Set A
	LLGE	140414		GEN	LTSTS	V	—	H	H	—	4	Test L ⩾ 0; Set A
	LLGT	141515		GEN	LTSTS	V	—	H	H	—	4	Test L > 0; Set A
	LLL	0410XX		SHFT	SHIFT	SRV	H	H	H	4	—	Long Left Logical
	LLLE	141511		GEN	LTSTS	V	—	H	H	—	4	Test L ⩽ 0; Set A
	LLLT	140410		GEN	LTSTS	V	—	H	H	—	4	Test L < 0; Set A
	LLNE	141512		GEN	LTSTS	V	—	H	H	—	4	Test L •NE• 0; Set A
	LLR	0412XX		SHFT	SHIFT	SRV	H	H	H	4	—	Long Left Rotate
	LLS	0411XX		SHFT	SHIFT	SRV	H	H	H	2	—	Long Left Shift
	LLT	140410		GEN	LTSTS	SRV	H	H	H	—	4	Test A < 0; Set A
	LLT	000		RGEN	LTSTS	I	—	—	H	—	4	Test R < 0; Set R
R	LMCM	000501		GEN	INTGY	SRVI	E	H	H	—	—	Leave Machine Check Mode
	LNE	140412		GEN	LTSTS	SRV	H	H	H	—	4	Test A •NE• 0; Set A
	LNE	002		RGEN	LTSTS	I	—	—	H	—	4	Test R •NE• 0; Set R
R	LPID	000617		GEN	MCTL	VI	—	H	H	—	—	Load Process ID
R	LPMJ	000215		MR	MCTL	SR	F	—	—	—	—	Leave Paging Mode and Jump
R	LPMX	000235		MR	MCTL	SR	F	—	—	—	—	Leave Paging Mode and Jump to XCS
R	LPSW	000711		AP	MCTL	VI	—	H	H	7	6	Load Program Status Word

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
	LRL	0400XX		SHFT	SHIFT	SRV	H	H	H	4	—	Long Right Logical
	LRR	0402XX		SHFT	SHIFT	SRV	H	H	H	4	—	Long Right Rotate
	LRS	0401XX		SHFT	SHIFT	SRV	H	H	H	4	—	Long Right Shift
	LT	140417		GEN	LTSTS	SRV	H	H	H	—	4	Set A = 1
	LT	017		RGEN	LTSTS	I	—	—	H	—	4	Set R = 1
R	LWCS	001710		GEN	MCTL	VI	—	H	H	6	5	Load Writable Control Store
	M	42	YY	MRGR	INT	I	—	—	H	3	—	Multiply Fullword
R	MDEI	001304		GEN	INTGY	VI	—	H	H	6	5	Memory Diagnostic Enable Interleave
R	MDII	001305		GEN	INTGY	VI	—	H	H	6	5	Inhibit Interleaved
R	MDIW	001324		GEN	INTGY	VI	—	H	H	6	5	Write Interleaved
R	MDRS	001306		GEN	INTGY	VI	—	H	H	6	5	Read Syndrome Bits
R	MDWC	001307		GEN	INTGY	VI	—	H	H	6	5	Load Write Control Register
	MH	52	YY	MRGR	INT	I	—	—	H	3	5	Multiply Halfword
	MIA	64	NN	MRGR	MCTL	I	—	—	H	—	—	Microcode Entrance
	MIA	12 01		MR	MCTL	V	—	H	H	—	—	Microcode Entrance
	MIB	74	NN	MRGR	MCTL	I	—	—	H	—	—	Microcode Entrance
	MIB	13 01		MR	MCTL	V	—	H	H	—	—	Microcode Entrance
	MPL	16 03		MR	INT	V	—	H	H	3	—	Multiply Long
	MPY	16		MR	INT	V	—	H	H	3	—	Multiply
	MPY	16		MR	INT	SR	B	H	H	3	—	Multiply
	N	03	YY	MRGR	LOGIC	I	—	—	H	—	—	AND Fullword
R	NFYB	001211		AP	PRCEX	VI	—	H	H	6	5	Notify
R	NFYE	001210		AP	PRCEX	VI	—	H	H	6	5	Notify
	NH	13	YY	MRGR	LOGIC	I	—	—	H	—	—	AND Halfword
	NOP	000001		GEN	MCTL	SRVI	H	H	H	—	—	No Operation
	NRM	000101		GEN	INT	SR	H	H	H	—	—	Normalize
	O	23	YY	MRGR	LOGIC	I	—	—	H	—	—	OR Fullword
R	OCP	14		PIO	IO	SR	H	H	H	—	—	Output Control Pulse
	OH	33	YY	MRGR	LOGIC	I	—	—	H	—	—	OR Halfword
	ORA	03 02		MR	LOGIC	V	—	H	H	—	—	Inclusive OR
R	OTA	74		PIO	IO	SR	H	H	H	—	—	Output from A
	OTK	000405		GEN	KEYS	SR	H	H	H	7	6	Restore Keys
	OTK	071		RGEN	KEYS	I	—	—	H	7	6	Restore Keys
	PCL	41	NN	MRNR	PCTLJ	I	—	—	H	6	5	Procedure Call
	PCL	10 02		MR	PCTLJ	V	—	H	H	6	5	Procedure Call
	PID	000211		GEN	INT	SR	B	H	H	—	—	Position for Integer Divide
	PID	052		RGEN	INT	I	—	—	H	—	—	Position for Integer Divide
	PIDA	000115		GEN	INT	V	—	H	H	—	—	Position for Integer Divide
	PIDH	053		RGEN	INT	I	—	—	H	—	—	Position for Integer Divide
	PIDL	000305		GEN	INT	V	—	H	H	—	—	Position Long for Integer Divide
	PIM	000205		GEN	INT	SR	B	H	H	—	—	Position After Multiply
	PIM	50		RGEN	INT	I	—	—	H	3	5	Position After Multiply
	PIMA	000015		GEN	INT	V	—	H	H	3	5	Position After Multiply
	PIMH	51		RGEN	INT	I	—	—	H	3	5	Position After Multiply
	PIML	000301		GEN	INT	V	—	H	H	3	5	Position After Multiply Long
	PRTN	000611		GEN	PCTLJ	VI	—	H	H	7	6	Procedure Return
R	PTLB	000064		GEN	MCTL	VI	—	—	H	6	5	Purge TLB (Prime 750 only)
	RBQ	141715		AP	QUEUE	V	—	H	H	—	7	Remove From Bottom of Queue
	RBQ	133		AP	RGEN	I	—	—	H	—	7	Remove From Bottom of Queue
	RCB	140200		GEN	KEYS	SRVI	H	H	H	5	—	Clear C-Bit (Reset)
R	RMC	000021		GEN	INTGY	SRVI	E	H	H	—	—	Clear Machine Check
	ROT	24	NN	MRGR	SHIFT	I	—	—	H	4	—	Rotate
	RRST	000717		AP	MCTL	VI	—	H	H	—	—	Register Restore
	RSAV	000715		AP	MCTL	VI	—	H	H	—	—	Register Save
	RTN	000105		GEN	PCTLJ	SR	H	H	H	—	—	Return
	RTQ	141714		AP	QUEUE	V	—	H	H	—	7	Remove From Top of Queue
	RTQ	132		RGEN	QUEUE	I	—	—	H	—	7	Remove From Top of Queue
	S	22	YY	MRGR	INT	I	—	—	H	2	1	Subtract Fullword
	S1A	140110		GEN	INT	SRV	H	H	H	2	1	Subtract One from A
	S2A	140310		GEN	INT	SRV	H	H	H	2	1	Subtract Two from A
	SAR	10026X		GEN	SKIP	SRV	H	H	H	—	—	Skip on A Bit Clear
	SAS	10126X		GEN	SKIP	SRV	H	H	H	—	—	Skip on A Bit Set

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
	SBL	07 03		MR	INT	V	—	H	H	2	1	Subtract Long
	SCA	000041		GEN	INT	SR	H	H	H	—	—	Load Shift Count into A
	SCB	1406000		GEN	KEYS	SRVI	H	H	H	5	—	Set C-Bit in Keys
	SGL	000005		GEN	INT	SR	H	H	H	—	—	Enter Single Precision Mode
	SGT	100220		GEN	SKIP	SRV	H	H	H	—	—	Skip if A Greater Than Zero
	SH	32	YY	MRGR	INT	I	—	—	H	2	1	Subtract Halfword
	SHA	15	NN	MRGR	SHIFT	I	—	—	H	4	—	Shift Arithmetic
	SHL	05	NN	MRGR	SHIFT	I	—	—	H	4	—	Shift Logical
	SHL1	076		RGEN	SHIFT	I	—	—	H	4	—	Shift H Left One
	SHL2	077		RGEN	SHIFT	I	—	—	H	4	—	Shift H Left Two
	SHR1	120		RGEN	SHIFT	I	—	—	H	4	—	Shift H Right One
	SHR2	121		RGEN	SHIFT	I	—	—	H	4	—	Shift H Right Two
	SKP	100000		GEN	SKIP	SRV	H	H	H	—	—	Skip
R	SKS	34		PIO	IO	SR	H	H	H	—	—	Skip if Satisfied
	SL1	072		RGEN	SHIFT	I	—	—	H	4	—	Shift R Left One
	SL2	073		RGEN	SHIFT	I	—	—	H	4	—	Shift R Left Two
	SLE	101220		GEN	SKIP	SRV	H	H	H	—	—	Skip if A Less Than or Equal to Zero
	SLN	101100		GEN	SKIP	SRV	H	H	H	—	—	Skip if LSB Nonzero (A(16)=1)
	SLZ	100100		GEN	SKIP	SRV	H	H	H	—	—	Skip if LSB Zero (A(16)=0)
	SMCR	100200		GEN	INTGY	SRV	E	H	H	—	—	Skip on Machine Check Reset
	SMCS	101200		GEN	INTGY	SRV	E	H	H	—	—	Skip on Machine Check Set
	SMI	101400		GEN	SKIP	SRV	H	H	H	—	—	Skip if A Minus
R	SNR	10024X		GEN	SKIP	SRV	H	H	H	—	—	Skip on Sense Switch Clear
R	SNS	10124X		GEN	SKIP	SRV	H	H	H	—	—	Skip on Sense Switch Set
	SNZ	101040		GEN	SKIP	SRV	H	H	H	—	—	Skip if A Non-Zero
	SPL	100400		GEN	SKIP	SRV	H	H	H	—	—	Skip if A Plus
R	SR1	100020		GEN	SKIP	SRV	H	H	H	—	—	Skip if Sense Switch 1 Clear
	SR1	074		RGEN	SHIFT	I	—	—	H	4	—	Shift R Right One
R	SR2	100010		GEN	SKIP	SRV	H	H	H	—	—	Skip if Sense Switch 2 Clear
	SR2	075		RGEN	SHIFT	I	—	—	H	4	—	Shift R Right Two
R	SR3	100004		GEN	SKIP	SRV	H	H	H	—	—	Skip if Sense Switch 3 Clear
R	SR4	100002		GEN	SKIP	SRV	H	H	H	—	—	Skip if Sense Switch 4 Clear
	SRC	100001		GEN	SKIP	SRV	H	H	H	—	—	Skip if C-Bit is Clear
R	SS1	101020		GEN	SKIP	SRV	H	H	H	—	—	Skip if Sense Switch 1 Clear
R	SS2	101010		GEN	SKIP	SRV	H	H	H	—	—	Skip if Sense Switch 2 Clear
R	SS3	101004		GEN	SKIP	SRV	H	H	H	—	—	Skip if Sense Switch 3 Clear
R	SS4	101002		GEN	SKIP	SRV	H	H	H	—	—	Skip if Sense Switch 4 Clear
	SSC	101001		GEN	SKIP	SRV	H	H	H	—	—	Skip if C-Bit is Set
	SSM	140500		GEN	INT	SRV	H	H	H	—	—	Set Sign Minus
	SSM	042		RGEN	INT	I	—	—	H	—	—	Set Sign Minus
	SSP	140100		GEN	INT	SRV	H	H	H	—	—	Set Sign Plus
	SSP	043		RGEN	INT	I	—	—	H	—	—	Set Sign Plus
R	SSR	100036		GEN	SKIP	SRV	H	H	H	—	—	Skip if Any Sense Switch is Clear
R	SSS	101036		GEN	SKIP	SRV	H	H	H	—	—	Skip if All Sense Switches are Set
	ST	21	NN	MRGR	MOVE	I	—	—	H	—	—	Store Fullword
	STA	04		MR	MOVE	SRV	H	H	H	—	—	Store A
	STAC	001200		AP	MOVE	V	—	H	H	—	7	Store A Conditionally
	STAR	54	NN	MRGR	MOVE	I	—	—	H	—	—	Store into Addressed Register
	STC 0	166		RGEN	CHAR	I	—	—	H	—	7	Store Character
	STC 1	176		RGEN	CHAR	I	—	—	H	—	7	Store Character
	STC 0	001322		CHAR	CHAR	V	—	H	H	—	7	Store Character
	STC 1	001332		CHAR	CHAR	V	—	H	H	—	7	Store Character
	STCD	137		AP	MOVE	I	—	—	H	—	7	Store Conditional Fullword
	STCH	136		AP	MOVE	I	—	—	H	—	7	Store Conditional Halfword
	STEX	001315		GEN	PCTLJ	V	—	H	H	6	5	Stack Extend
	STEX	027		RGEN	PCTLJ	I	—	—	H	6	5	Stack Extend
	STFA 0	001320		AP	FIELD	VI	—	H	H	—	—	Store Field Address Register
	STFA 1	001330		AP	FIELD	VI	—	H	H	—	—	Store Field Address Register
	STH	31	NN	MRGR	MOVE	I	—	—	H	—	—	Store Halfword
	STL	04 03		MR	MOVE	V	—	H	H	—	—	Store Long
	STLC	001204		AP	MOVE	V	—	H	H	—	7	Store L Conditionally
P	STLR	03 01		MR	MOVE	V	—	H	H	—	—	Store L into Addressed Register

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	C	CC	DESCRIPTION
R	STPM	000024		GEN	MCTL	VI	—	H	H	—	—	Store Processor Model Number
	STX	15		MR	MOVE	SRV	H	H	H	—	—	Store X
	STY	35 02		MR	MOVE	V	—	H	H	—	—	Store Y
	SUB	07		MR	INT	SRV	H	H	H	2	1	Subtract
	SVC	000505		GEN	PCTLJ	SRVI	H	H	H	—	—	Supervisor Call
	SZE	100040		GEN	SKIP	SRV	H	H	H	—	—	Skip if A Zero
	TAB	140314		GEN	MOVE	V	—	H	H	—	—	Transfer A to B
	TAK	001015		GEN	KEYS	V	—	H	H	7	6	Move A to Keys
	TAX	140504		GEN	MOVE	V	—	H	H	—	—	Transfer A to X
	TAY	140505		GEN	MOVE	V	—	H	H	—	—	Transfer A to Y
	TBA	140604		GEN	MOVE	V	—	H	H	—	—	Transfer B to A
	TC	046		RGEN	INT	I	—	—	H	2	1	Two's Complement R
	TCA	140407		GEN	INT	SRV	H	H	H	2	1	Two's Complement A
	TCH	047		RGEN	INT	I	—	—	H	2	1	Two's Complement H
	TCL	141210		GEN	INT	V	—	H	H	2	1	Two's Complement Long
	TFLL 0	001323		GEN	FIELD	V	—	H	H	—	—	Transfer Field Length to L
	TFLL 1	001333		GEN	FIELD	V	—	H	H	—	—	Transfer Field Length to L
	TFLR 0	163		RGEN	FIELD	I	—	—	H	—	—	Move Field Length to R
	TFLR 1	173		RGEN	FIELD	I	—	—	H	—	—	Move Field Length to R
	TKA	001005		GEN	KEYS	V	—	H	H	—	—	Move Keys to A
	TLFL	001321		GEN	FIELD	V	—	H	H	—	—	Transfer L to Field Length Register
	TLFL	001331		GEN	FIELD	V	—	H	H	—	—	Transfer L to Field Length Register
	TM	44	NN	MRNR	MCTL	I	—	—	H	—	1	Test Memory Fullword
	TMH	54	NN	MRNR	INT	I	—	—	H	—	1	Test Memory Halfword
	TRFL 0	165		RGEN	FIELD	I	—	—	H	—	—	Transfer R to Field Length Register
	TRFL 1	175		RGEN	FIELD	I	—	—	H	—	—	Transfer R to Field Length Register
	TSTQ	141757		AP	QUEUE	V	—	H	H	—	7	Test Queue
	TSTQ	104		RGEN	QUEUE	I	—	—	H	—	7	Test Queue
	TXA	141034		GEN	MOVE	V	—	H	H	—	—	Transfer X to A
	TYA	141124		GEN	MOVE	V	—	H	H	—	—	Transfer Y to A
R	VIRY	000311		GEN	INTGY	SRVI	G	H	H	6	5	Verify
R	WAIT	000315		AP	PRCEX	VI	—	H	H	—	—	Wait
	WCS	0016XX		GEN	MCTL	RVI	—	O	O	—	—	Writeable Control Store
	X	43	YY	MRGR	LOGIC	I	—	—	H	—	—	Exclusive OR Fullword
	XAD	001100		DECI	DECI	VI	—	U	H	3	1	Decimal Add
	XBTD	001145		DECI	DECI	VI	—	U	H	—	—	Binary to Decimal Conversion
	XCA	140104		GEN	MOVE	SRV	H	H	H	—	—	Exchange and Clear A
	XCB	140204		GEN	MOVE	SRV	H	H	H	—	—	Exchange and Clear B
	XCM	001102		DECI	DECI	VI	—	U	H	—	1	Decimal Compare
	XDTB	001146		DECI	DECI	VI	—	U	H	—	5	Decimal to Binary Conversion
	XDV	001107		DECI	DECI	VI	—	U	H	—	1	Decimal Divide
	XEC	01 02		MR	PCTLJ	RV	F	H	H	—	—	Execute
	XED	001112		DECI	DECI	VI	—	—	H	—	—	Numeric Edit
	XH	53	YY	MRGR	LOGIC	I	—	—	H	—	—	Exclusive OR Halfword
	XMP	001104		DECI	DECI	VI	—	U	H	3	1	Decimal Multiply
	XMV	001101		DECI	DECI	VI	—	U	H	—	1	Decimal Move
R	XVRY	001113		MCTL	GEN	VI	—	U	H	6	5	Verify XIS
	ZCM	001117		CHAR	CHAR	VI	—	U	H	—	1	Compare Character Field
	ZED	001111		CHAR	CHAR	VI	—	—	H	—	—	Character Edit
	ZFIL	001116		CHAR	CHAR	VI	—	U	H	—	—	Fill Character Field
	ZM	43	NN	MRNR	CLEAR	I	—	—	H	—	—	Clear Fullword
	ZMH	53	NN	MRNR	CLEAR	I	—	—	H	—	—	Clear Halfword
	ZMV	001114		CHAR	CHAR	VI	—	U	H	—	—	Move Character Field
	ZMVD	001115		CHAR	CHAR	VI	—	U	H	—	—	Move Equal Length Fields
	ZTRN	001110		CHAR	CHAR	VI	—	U	H	—	—	Translate Character Fields

4 PMA REFERENCE

14
Language structure

INTRODUCTION

The Prime Macro Assembler's language structure is both flexible and simple. For example, here is a program which includes three pseudo-operations, a machine instruction and a literal.

```
REL             PSEUDO-OPERATION - USE RELOCATABLE ADDRESSING
LDA  ='123      MACHINE INSTRUCTION - LITERAL
CALL EXIT       PSEUDO-OPERATION - SUBROUTINE CALL
END             PSEUDO-OPERATION - END OF SOURCE CODE
```

This section describes the structure and function of PMA language statements, and the elements, constants, symbols and expressions which comprise them.

LINES

Input to the assembler consists of instruction statements and comments (See Figure 14-1). The basic unit of information is the line (See Figure 14-2). Fields, statements and comments within a line can be delimited by either spaces, commas or colons, depending on the construction.

There are three basic line formats:

Comment Line	Column 1 contains an asterisk (*). The entire line is treated as a comment.
Change Page Heading Lines	Column 1 contains an apostrophe ('). The rest of the line is used as a page title for subsequent pages.
Statements	See below.

STATEMENTS

Types

Statements may be:

- Mnemonic representations of machine instructions.
- Assembler pseudo-operations.
- Macro definitions and calls.

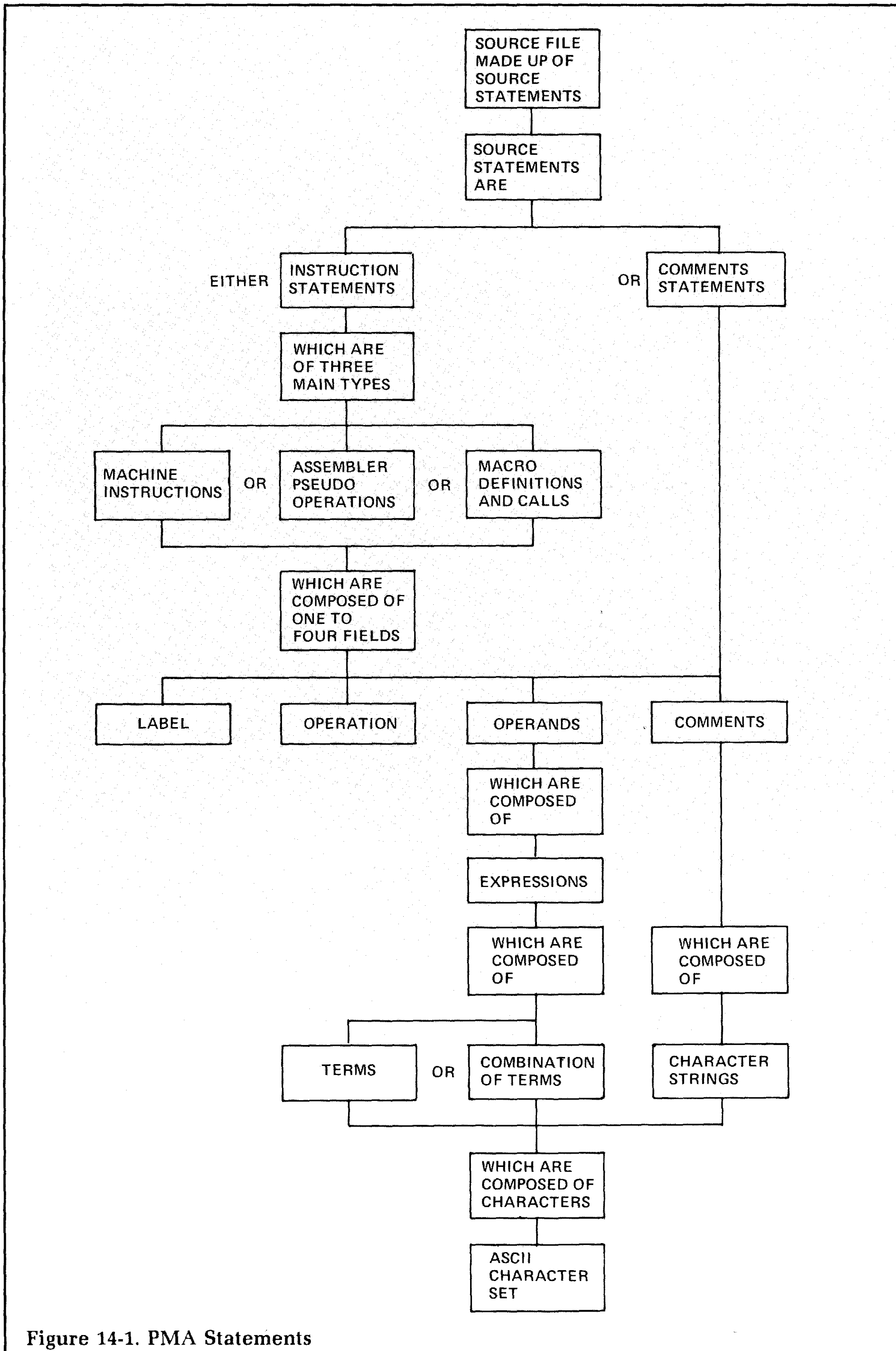


Figure 14-1. PMA Statements

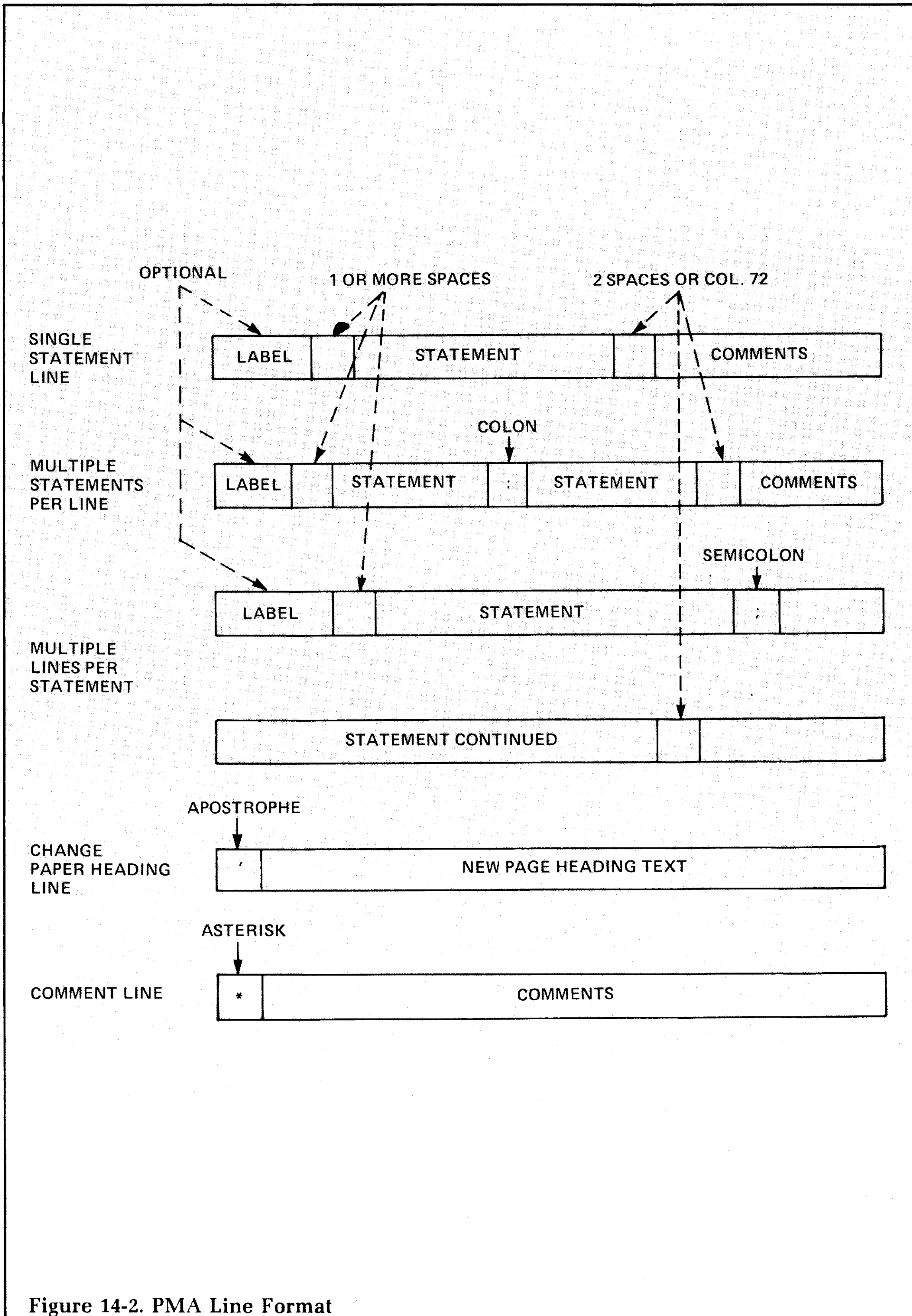


Figure 14-2. PMA Line Format

Syntax

Statements may have four possible fields, delimited by spaces:

[label] operation [operand] . . . [comment]

Label: Assigns a name to a program location, such as a subroutine entry point, the address of a constant, or a storage field.

The first character of a label *must* be in column 1 of a line. If a statement does not have a label, the first column must be blank. Labels must be legal symbols.

Operation: Defines the action taken at assembly time (pseudo-operations, macro definitions and calls) or at execution time (machine instructions). The operation is the only field required in all types of instructions and must be entered exactly as shown in the individual statement definition.

Operand: Contains information to be acted upon by the operation code. The number of operands and their meaning is operation-specific. Some statements do not require an operand; others require several.

Comments: Ignored by PMA except for printing in the listing. Comments document the meaning of the operation. All text following either column 72 or two spaces after the last operand (ten spaces with macro calls) is treated as a comment.

Elements

Statement elements — labels, operation codes and operands — are composed of constants, symbols and expressions. These are a subset of the printing ASCII characters. The entire ASCII character set, printing and non-printing, may be used in comments, macro instruction operands and within literal text fields.

Constants: Constants are explicit data values. A constant may be any of the following data types.

- Decimal
- Binary
- Hexadecimal
- Octal
- Character (ASCII)
- Address

Constants may be used in expressions to represent bit configurations, absolute addresses, displacements and data. Section 15, Data Definition, contains a full discussion of data types and formats.

Symbols: Symbols are alphanumeric strings which represent locations or data. They may be from 1 to 32 characters in length. The first character *must* be a letter (A-Z), and the remaining characters may be letters, numerals (0-9), the dollar sign ($), or underscore (_). Symbols containing more than 32 characters are allowed in the source code, but only the first 32 characters are examined by the assembler.

Expressions: Expressions contain one or more constants or symbols, called terms, which have single precision integer values. Multiple terms are joined by operators, which may be arithmetic, relational, logical or shift. At assembly time, PMA evaluates an expression by performing the indicated operations, if any, thus producing a single precision integer result.

Example	Comment
A + 3	Arithmetic expression consisting of a variable "A" and a constant "3"

A.LS.(ALPA/5)	Shift expression consisting of a variable (A) and a sub-expression (ALPA/5).
BETA.GE.A+$FF	Logic expression consisting of a variable (BETA) and a sub-expression (A+$FF).

You may use expressions as:

- Instruction operands
- Literals
- Part of macro definitions and calls
- Symbol-defining pseudo-operation operands

Multiple statements per line: Statements may be packed two or more per line. Each statement is separated from the one following by a colon (:). The PMA assembler processes the first non-space character following the colon as the operation of a new statement. The last statement in the line is terminated by two spaces or column 73, and the rest of the line treated as a comment. If the line begins with a label, the label is assigned to the first statement on that line. Since labels must begin in column 1, there can only be one label per line, as in:

```
LABEL1 LDA = 123:LDA = 456 COMMENT2
```

Multiple lines per statement: Any statement may be interrupted by a semicolon (;) and continued on the next line. The rest of the line following the semicolon is treated as a comment. Processing of the statement continues with the first non-space character in the following line. Semicolons appearing within comments are not interpreted as continuation requests.

MEMORY REFERENCE INSTRUCTION FORMAT (SRV)

Operation Field

Mnemonic: The operation field must include one of the memory reference instruction mnemonics.

Triple asterisk (dummy instruction): A triple asterisk in place of an instruction mnemonic is a pseudo-operation code that causes the assembler to form a memory reference instruction with an op-code of zero. Another asterisk may be added to specify indirect addressing. The variable field of such a statement is treated like any other memory reference instruction.

Percent sign (%): A % following the mnemonic tells the assembler to process this instruction in two word (long-reach) format.

Pound sign (#): A # following the mnemonic tells the assembler to process this instruction in one word format.

Operand Field

The operand field of a memory reference instruction contains an address expression, which may be modified by indirection, indexing, and by case register references.

Symbolic addresses: Addresses can be specified by any constant, symbol, literal, or expression which can be evaluated as a 16-bit number.

Indexing: Indexing is optional and is specified by a "1", "X", or, in V or I mode, a "Y", following the address expression. The form "0" is interpreted as non-indexing.

Indirection: An asterisk (*) tells the assembler that this address is a pointer to another address.

Stack: An at sign (@) used alone in the address tells the assembler that this instruction references the stack. This notation is not legal in 64V or 32I; stack base—SB%—notation is used in these modes.

Base registers (V or I mode): A percent sign (%) following SB, LB, PB or XB tells the assembler that the operand is relative to a particular base register.

Asterisk (current location): An asterisk (*) in the operand field represents the current value of the assembler location counter.

Double asterisk (initial zero): A double asterisk (**) in the operand field causes the assembler to load zeroes in the 9-bit address field and sector bit. (Indexing and indirect addressing conventions are unchanged.) This convention is used when the desired location is to be developed or modified by other instructions or is not known at the time of assembly.

Equals sign (literals): A literal is a constant preceded by an equals sign, as in:

```
LDA ='100
```

The assembler associates the numerical value of each literal with the symbol used ('100 in this case) and reserves a storage location for a constant of the value. The value in the operand field will be the address of the literal.

Special Cases

The assembler will generate a long instruction if any of the following cases apply:

1. Indexing by Y is specified.
2. A percent (%) opcode modifier is used.
3. The opcode does not have a short form (zero opcode extension).
4. If the mode of the address is external of common (SEG/SEGR mode only).
5. The address is linkage relative and not in the range '400 to '777.
6. The address is stack relative and not in the range '10 to '377.
7. The address is temporary base relative.
8. Indirection is specified and a # opcode modifier is not used (SEG or SEGR mode only).

If any of the cases apply and a # opcode suffix was used, an error will be generated.

In SEG or SEGR mode, if the expression mode is absolute, it must be in the range 0 to 7 to specify a compatible register address. If a long instruction is generated, an error will be reported since register set addressing is only available with short instructions. Because of rule 8 above, an instruction that specified indirection through a register must also have a # opcode suffix.

INSTRUCTION FORMATS - I-MODE

The assmbler formats for I-Mode instructions are listed in Table 14-1. See Section 9, Data Structures, for the I-Mode machine formats.

Table 14-1. Assembler Formats (I-Mode)

Instruction Type	Assembler
MRGR	op r,addr
MRFR	op f,addr
MRNR	op addr
IBRN	op {r, h, f} ,word
* IBRN bit test	op r,bitno, word
RGEN	op r
GEN	op
AP	op ap
** RGEN field register	op falr,r
** BRAN field register	op falr,data
** AP field register	op falr,ap

* bitno selects a specific opcode
** FALR selects a specific opcode

Abbreviations

WORD	word number address field, no indirection or indexing
ADDR	full segmented address field, optional indirection/indexing by general registers 1-7
AP	argument pointer, optional indirection
BIT	bit number (1-32)
DATA	16 bit data word
FALR	field register number (0-1)
F	floating register number (0-1)
OPCODE	instruction operation code
R	general register number (0-7)

```
                      (0001) ****** I-MODE ADDRESSING ILLUSTRATION *******
                      (0002) *
                      (0003) *
                      (0004)          SEGR
              000004  (0005) IX       EQU  4
                      (0006) *
000000: 002240.000032 (0007) STRT     L    1,DATA      DIRECT
000002: 002540.000030 (0008)          L    2,PTR,*     INDIRECT
000004: 002674.000030 (0009)          L    3,PTR,7     INDEXED
000006: 003174.000030 (0010)          L    4,PTR,*7    POST-INDEXED, INDIRECT
000010: 003334.000030 (0011)          L    5,PTR,7*    PRE-INDEXED, INDIRECT
000012: 003560.000030 (0012)          L    6,PTR,*(IX) POST-INDEXED, INDIRECT
000014: 003730.000030 (0013)          L    7,PTR,(IX+2)* PRE-INDEXED, INDIRECT
                      (0014) *
000016: 002205.000100A (0015)         L    1,='100L  GENERAL REGISTER IMMEDIATE
000020: 003042.000400L (0016)         L    4,='1234567L GENERAL REGISTER LITERAL
000022: 014002.040201A (0017)         FL   0,=1.0      FLOATING REGISTER IMMEDIATE
000024: 014042.000402L (0018)         FL   0,=3.14159 FLOATING REGISTER LITERAL
                      (0019) *
000026:        002214 (0020)          L    1,3         REGISTER TO REGISTER
000027:        014216 (0021)          DFL  0,1         FLOATING REGISTER TO REGISTER
                      (0022) *
                      (0023) *
000030:               (0024) PTR      BSS  2
000032:               (0025) DATA     BSS  2
                      (0026) *
                      (0027) *
           000034     (0028)          END
```

HOW TO WRITE V OR I MODE CODE IN PMA

In order to take advantage of the PMA facilities, the structure of a V or I Mode program should reflect the system architecture design for the separation of code and data (see Reference Guide, System Architecture).

The recommended structure is:

Prologue

SEG/SEGR	Sets up segmented address space
RLIT	Puts literals in the procedure area
ENT	Entry point declarations

Code

Executable code

Data Area

DYNM	Stack variable declarations
LINK	defines linkage area containing static variables
ECB	entry control block

End

References ECB name.

PMA makes using the segmented architecture easy. Thus the programmer can write straightforward code, such as LDA ADDR. The assembler, depending on the definition of ADDR, may generate a one word or two word instruction, and may reference either the stack area, the linkage area, the procedure area or a temporary area. This is possible because symbols carry a great deal of state information with them.

Key Points

ECB: The ECB (entry control block) describes the environment the program runs in. It includes the location to start execution, the name of the first argument, if any, and the number of arguments. The ECB is the link used by the system to run the module. It may be located anywhere, but normally it goes in the linkage data area in order to produce pure code.

Stack: The default stack size includes a stack frame header. Additionally, all the stack variables defined by the DYNM pseudo-operation are added to the stack frame size and the size is automatically put into the ECB definition. Stack variables are defined sequentially and may be any size. For example DYNM STAR(1) generates one word, while argument pointers, which are three word indirect pointers to the first word of an argument, would be defined as, for example, DYNM STPTR(3). Definition of stack variables by DYNM allows the assembler to build the appropriate addressing forms automatically e.g. DYNM STAR(1); LDA STAR will cause the assembler to generate the address form explicitly shown by LDA# SB%+STAR.

Code: The assembler automatically places code in the procedure segment; the PROC pseudo-operation is not necessary unless you want to put some code after the LINK pseudo-operation which defines the linkage area.

Note

Since the assembler picks the instruction length for you, be careful about using skips and compares—they assume a one word instruction following them. Also, be aware that, by default, all pointers are long.

Linkage area: The LINK pseudo-operation tells the assembler to tag the variables which are defined after it (using BSS, ECB, DATA, etc.) as linkage base relative. These are the static, impure data and variables required for pure procedures.

Literals: The RLIT pseudo-operation will cause literals to be generated in the procedure area.

Examples

The series of annotated examples beginning on page 14-10 below show a subroutine as a programmer might write it in V-Mode, I-Mode, and R-Mode. Argument transfer and referencing are also shown in Section 8—Interfacing with System Libraries.

ADDARY, jrw, 01/11/79   -32I MODE-

```
                     (0001) *   ADDARY, jrw, 01/11/79   -32I MODE-
                     (0002) *
                     (0003) *   Add the elements of a 10 dimensional array of 16-bit values, producing
                     (0004) *   a 16-bit result and returning a count of the number of members of the
                     (0005) *   array which were zero.  The result is printed on the user terminal.
                     (0006) *
                     (0007) *
                     (0008) *
                     (0009) *       Calling sequence (Fortran):
                     (0010) *
                     (0011) *           INTEGER ARY(10), NZERO, ADDARY, RESULT
                     (0012) *             ...
                     (0013) *           RESULT = ADDARY (ARY, NZERO)
                     (0014) *
                     (0015) *
                     (0016) *       Calling sequence (PMA):
                     (0017) *
                     (0018) *           CALL    ADDARY
                     (0019) *           AP      ARY,S               ARRAY WHOSE ELEMENTS ARE TO BE SUMMED
                     (0020) *           AP      NZERO,SL            RETURNED # ELEMENTS = 0
                     (0021) *           ...                         RESULT RETURNED IN (A)  (GR2H)
                     (0022) *
                     (0023) *
                     (0024) *
            000400   (0025)        ENT     ADDARY,ADDECB   ENTRY DECLARATION
                     (0026) *
                     (0027)        SEGR                    32I-MODE SEGMENTED ADDRESSING
                     (0028)        RLIT                    PLACE LITERALS IN PROCEDURE FRAME
                     (0029) *
                     (0030) *
            000000   (0031) ADDARY EQU     *               ECB CAUSES CONTROL TO BE PASSED HERE ON CALL
000000:     000605   (0032)        ARGT                    TRANSFER POINTERS TO ARGUMENTS
                     (0033) *
000001: 022201.177766A (0034)      LH      1,=-10          # ELEMENTS TO ADD
000003: 134741.000015S (0035)      ZMH     NZERO,*         INITIALIZE # ELEMENTS WHICH = 0
000005: 022601.000000A (0036)      LH      3,=0            INITIALIZE ACCUMULATOR
                     (0037) *
                     (0038) *---LOOP TO PERFORM ACTUAL ADDITION:
            000007   (0039) ADDLP  EQU     *
000007: 022541.000012S (0040)      LH      2,ARY,*         PICK UP NEXT ARRAY ELEMENT
000011: 020513.000015  (0041)      BHNE    2,LP10          TEST ZERO
000013: 134141.000015S (0042)      IMH     NZERO,*         VALUE IS ZERO, BUMP COUNTER
000015:     024610   (0043) LP10   AH      3,2             COMPUTE NEW SUM (RESULT => GR3H)
000016: 134041.000013S (0044)      IMH     ARY+1           UPDATE WORD# OF POINTER TO NEXT ARRAY ELEMENT
000020: 020340.000007  (0045)      BHI1    1,ADDLP         BRANCH IF NOT DONE, PROCESS NEXT ELEMENT
                     (0046) *
                     (0047) *---PRINT RESULT ON USER TERMINAL...
000022: 062641.000020S (0048)      STH     3,SUM           SO WE CAN PRINT SUM
000024: 114342.000420L (0049)      CALL    TNOUA
000026: 000100.000043  (0050)      AP      =C'RESULT IS ',S
000030: 000300.000050  (0051)      AP      =10,SL
000032: 114342.000422L (0052)      CALL    TODEC           PRINT DECIMAL RESULT
000034: 000700.000020S (0053)      AP      SUM,SL
000036: 114342.000424L (0054)      CALL    TONL            PRINT NEW-LINE
                     (0055) *
                     (0056) *---THRU HERE WHEN DONE - RETURN WITH SUM IN THE GR2H (A-REGISTER).
000040: 022441.000020S (0057)      LH      2,SUM
000042:     000611   (0058)        PRTN                    BACK TO CALLER
                     (0059) *
                     (0060) *
                     (0061) *---DATA DEFINITION:
                     (0062) *
            000012   (0063)        DYNM    ARY(3),NZERO(3),SUM
            000015
            000020
```

```
                           (0064) *
                           (0065) *
                           (0066)         LINK
                           (0067) *
 000400>         000000    (0068) ADDECB ECB      ADDARY,,ARY,2
                 000022
                 000012
                 000002
                 177400
                 010000

                           (0069) *
                           (0070) *
                           (0071) *
           000420          (0072)         END

 000043:      00.151305A
 000044:      00.151725A
 000045:      00.146324A
 000046:      00.120311A
 000047:      00.151640A
 000050:      00.000012A

 000420> 000000.000000E
 000422> 000000.000000E
 000424> 000000.000000E

 TEXT SIZE:   PROC 000051   LINK 000026   STACK 000021

 ADDARY     000000  0031   0068
 ADDECB     000400L 0068
 ADDLP      000007  0039   0045
 ARY        000012S 0040   0044   0063   0068
 LP10       000015  0041   0043
 NZERO      000015S 0035   0042   0063
 SUM        000020S 0048   0053   0057   0063
 TNOUA      000000E 0049
 TODEC      000000E 0052
 TONL       000000E 0054


0000 ERRORS (PMA-REV 16.2)
BOTTOM
qu
```

ADDARY, jrw, 01/11/79   -64R MODE-

```
                    (0001) *   ADDARY, jrw, 01/11/79   -64R MODE-
                    (0002) *
                    (0003) *   Add the elements of a 10 dimensional array of 16-bit values, producing
                    (0004) *   a 16-bit result and returning a count of the number of members of the
                    (0005) *   array which were zero.  The result is printed on the user terminal.
                    (0006) *
                    (0007) *
                    (0008) *
                    (0009) *      Calling sequence (Fortran):
                    (0010) *
                    (0011) *         INTEGER ARY(10), NZERO, ADDARY, RESULT
                    (0012) *          ...
                    (0013) *         RESULT = ADDARY (ARY, NZERO)
                    (0014) *
                    (0015) *
                    (0016) *      Calling sequence (PMA):
                    (0017) *
                    (0018) *         CALL    ADDARY
                    (0019) *         DAC     ARY                 ARRAY WHOSE ELEMENTS ARE TO BE SUMMED
                    (0020) *         DAC     NZERO               RETURNED # ELEMENTS = 0
                    (0021) *         DEC     0                   (TO TERMINATE ARG LIST)
                    (0022) *         STA     RESULT              RESULT RETURNED IN (A)
                    (0023) *
                    (0024) *
                    (0025) *
           000000   (0026)       ENT     ADDARY              ENTRY DECLARATION
                    (0027) *
                    (0028)       C64R                        CHECK 64R MODE ADDRESSING VIOLATIONS
                    (0029)       REL                         RELATIVE MODE ASSEMBLY
                    (0030) *
                    (0031) *
000000:   00.000000A (0032) ADDARY DAC  **                   RETURN ADDRESS SAVED HERE BY 'JST' INSTRUCTION
000001:   10.000000E (0033)       CALL    F$AT               TRANSFER ARGUMENTS' ADDRESSES
000002:     000002   (0034)       DATA    2                  (2 ARGUMENTS)
000003:   00.000000A (0035) ARY   DAC     **                 PTR TO ARRAY
000004:   00.000000A (0036) NZERO DAC     **                 PTR TO COUNTER FOR ELEMENTS W/ VALUE = 0
                    (0037) *
000005:   35.000033  (0038)       LDX     =-10               # ELEMENTS TO ADD
000006:     140040   (0039)       CRA                        INITIALIZE COUNTERS
000007:   04.000032  (0040)       STA     SUM
000010:   44.000004  (0041)       STA     NZERO,*            # ELEMENTS WHICH = 0
                    (0042) *
                    (0043) *---LOOP TO PERFORM ACTUAL ADDITION:
            000011   (0044) ADDLP EQU     *
000011:   42.000003  (0045)       LDA     ARY,*              PICK UP NEXT ARRAY ELEMENT
000012:     101040   (0046)       SNZ                        TEST ZERO
000013:   52.000004  (0047)       IRS     NZERO,*            VALUE IS ZERO, BUMP COUNTER
000014:   06.000032  (0048)       ADD     SUM                COMPUTE NEW SUM
000015:   04.000032  (0049)       STA     SUM
000016:   12.000003  (0050)       IRS     ARY                BUMP POINTER TO NEXT ARRAY ELEMENT
000017:     140114   (0051)       IRX                        TEST DONE
000020:   01.000011  (0052)       JMP     ADDLP              NOT DONE, PROCESS NEXT ELEMENT
                    (0053) *
                    (0054) *---PRINT RESULT ON USER TERMINAL...
000021:   10.000000E (0055)       CALL    TNOUA
000022:   00.000034  (0056)       DAC     =C'RESULT IS '     TEXT TO BE PRINTED
000023:   00.000041  (0057)       DAC     =10                # CHARACTERS
000024:     000000   (0058)       DEC     0
000025:   10.000000E (0059)       CALL    TODEC              PRINT DECIMAL RESULT
000026:   00.000032  (0060)       DAC     SUM
000027:   10.000000E (0061)       CALL    TONL               PRINT NEW-LINE
                    (0062) *
                    (0063) *---THRU HERE WHEN DONE - RETURN WITH SUM IN THE A-REGISTER.
000030:   02.000032  (0064)       LDA     SUM
000031:   41.000000  (0065)       JMP     ADDARY,*           BACK TO CALLER
                    (0066) *
                    (0067) *
                    (0068) *---DATA DECLARATION:
                    (0069) *
000032:              (0070) SUM   BSS     1                  TEMPORARY SUM
                    (0071) *
                    (0072) *
                    (0073) *
          000033     (0074)       END

000033:   00.177766A
000034:   00.151305A
000035:   00.151725A
```

```
  000036:      00.146324A
  000037:      00.120311A
  000040:      00.151640A
  000041:      00.000012A

  TEXT SIZE:  000042 WORDS
^214^014
  ADDARY     000000  0032  0065
  ADDLP      000011  0044  0052
  ARY        000003  0035  0045  0050
  F$AT       000000E 0033
  NZERO      000004  0036  0041  0047
  SUM        000032  0040  0048  0049  0060  0064  0070
  TNOUA      000000E 0055
  TODEC      000000E 0059
  TONL       000000E 0061


0000 ERRORS (PMA-REV 16.2)
BOTTOM
```

```
ADDARY, jrw, 01/11/79   -64V MODE-


                        (0001) *   ADDARY, jrw, 01/11/79   -64V MODE-
                        (0002) *
                        (0003) *   Add the elements of a 10 dimensional array of 16-bit values, producing
                        (0004) *   a 16-bit result and returning a count of the number of members of the
                        (0005) *   array which were zero.  The result is printed on the user terminal.
                        (0006) *
                        (0007) *
                        (0008) *
                        (0009) *      Calling sequence (Fortran):
                        (0010) *
                        (0011) *         INTEGER ARY(10), NZERO, ADDARY, RESULT
                        (0012) *           ...
                        (0013) *         RESULT = ADDARY (ARY, NZERO)
                        (0014) *
                        (0015) *
                        (0016) *      Calling sequence (PMA):
                        (0017) *
                        (0018) *         CALL    ADDARY
                        (0019) *         AP      ARY,S              ARRAY WHOSE ELEMENTS ARE TO BE SUMMED
                        (0020) *         AP      NZERO,SL           RETURNED # ELEMENTS = 0
                        (0021) *         ...                        RESULT RETURNED IN (A) (GR2H)
                        (0022) *
                        (0023) *
                        (0024) *
              000400    (0025)        ENT     ADDARY,ADDECB   ENTRY DECLARATION
                        (0026) *
                        (0027)        SEG                     64V-MODE SEGMENTED ADDRESSING
                        (0028)        RLIT                    PLACE LITERALS IN PROCEDURE FRAME
                        (0029) *
                        (0030) *
              000000    (0031) ADDARY EQU     *               ECB CAUSES CONTROL TO BE PASSED HERE ON CALL
000000:       000605    (0032)        ARGT                    TRANSFER POINTERS TO ARGUMENTS
                        (0033) *
000001:    35.000037    (0034)        LDX     =-10            # ELEMENTS TO ADD
000002:       140040    (0035)        CRA                     INITIALIZE COUNTERS
000003:    04.000020S   (0036)        STA     SUM
000004: 051421.000015S  (0037)        STA     NZERO,*         # ELEMENTS WHICH = 0
                        (0038) *
                        (0039) *---LOOP TO PERFORM ACTUAL ADDITION:
              000006    (0040) ADDLP  EQU     *
000006: 045421.000012S  (0041)        LDA     ARY,*           PICK UP NEXT ARRAY ELEMENT
000010: 140613.000014   (0042)        BNE     LP10            TEST ZERO
000012: 065421.000015S  (0043)        IRS     NZERO,*         VALUE IS ZERO, BUMP COUNTER
000014:    06.000020S   (0044) LP10   ADD     SUM             COMPUTE NEW SUM
000015:    04.000020S   (0045)        STA     SUM
000016:    12.000013S   (0046)        IRS     ARY+1           UPDATE WORD# OF POINTER TO NEXT ARRAY ELEMENT
000017: 141334.000006   (0047)        BIX     ADDLP           BRANCH IF NOT DONE, PROCESS NEXT ELEMENT
                        (0048) *
                        (0049) *---PRINT RESULT ON USER TERMINAL...
000021: 061432.000420L  (0050)        CALL    TNOUA
000023: 000100.000040   (0051)        AP      =C'RESULT IS ',S
000025: 000300.000045   (0052)        AP      =10,SL
000027: 061432.000422L  (0053)        CALL    TODEC           PRINT DECIMAL RESULT
000031: 000700.000020S  (0054)        AP      SUM,SL
000033: 061432.000424L  (0055)        CALL    TONL            PRINT NEW-LINE
                        (0056) *
                        (0057) *---THRU HERE WHEN DONE - RETURN WITH SUM IN THE A-REGISTER.
000035:    02.000020S   (0058)        LDA     SUM
000036:       000611    (0059)        PRTN                    BACK TO CALLER
                        (0060) *
                        (0061) *
                        (0062) *---DATA DEFINITION:
                        (0063) *
              000012    (0064)        DYNM    ARY(3),NZERO(3),SUM
              000015
              000020
                        (0065) *
                        (0066) *
                        (0067)        LINK
                        (0068) *
000400>       000000    (0069) ADDECB ECB     ADDARY,,ARY,2
              000022
              000012
              000002
              177400
              014000
```

```
                        (0070) *
                        (0071) *
                        (0072) *
         000420         (0073)          END

 000037:      00.177766A
 000040:      00.151305A
 000041:      00.151725A
 000042:      00.146324A
 000043:      00.120311A
 000044:      00.151640A
 000045:      00.000012A

 000420> 000000.000000E
 000422> 000000.000000E
 000424> 000000.000000E

 TEXT SIZE:   PROC 000046   LINK 000026   STACK 000021



 ADDARY     000000  0031   0069
 ADDECB     000400L 0069
 ADDLP      000006  0040   0047
 ARY        000012S 0041   0046   0064   0069
 LP10       000014  0042   0044
 NZERO      000015S 0037   0043   0064
 SUM        000020S 0036   0044   0045   0054   0058   0064
 TNOUA      000000E 0050
 TODEC      000000E 0053
 TONL       000000E 0055

0000 ERRORS (PMA-REV 16.2)
BOTTOM
```

15 Data definition

This section discusses all aspects of the definition and usage of data constants within a program.

CONSTANTS

Constants are divided into two major categories: numeric and character. They may be explicitly defined by pseudo-operations, such as OCT and DEC, or implicitly defined by usage within expressions and literals.

Constants are used in expressions, literals, and DATA statements. In expressions, each constant must be one 16-bit word. See Data Defining Pseudo-Operations for a full discussion of all data type pseudo-operations referenced in this section.

The format of a constant determines how PMA will process it. Table 15-1 shows the data types and formats of all legal numeric constants. Normally, you would use DATA or DEC to define stand-alone constants, and the form defined by data type symbol to express constants in an expression or literal.

Integer constants

All integer constants are signed whole-number quantities and may be single or double precision. Single precision is the default: double precision is expressed by appending the letter L to the constant. The sign, if present, follows the data type symbol.

Precision	Address Mode		Range
Single	SRVI	*From:*	−32,768 (−2**15)
		To:	+32,767 (2**15-1)
Double	SR	*From:*	−1,073,741,824 (−2**30)
		To:	+1,073,741,823 (2**30-1)
Double	VI	*From:*	-2,147,483,648 [-2**31]
		To:	+2,147,483,647 (2**30-1)

Decimal: Whole number, base 10 quantities

Data Type Symbol: none

Precision	Constant	Listing Representation
Single Precision	DATA 123	000173
	DEC 123	000173
	123	000173
Double Precision	DATA 123L	000000 000173
	DEC 123L	000000 000173
	123L	000000 000173

Table 15-1. Numeric Constants

Class	Source	Symbol	Pseudo-Op	Binary Scale	Notes	Exponent	Notes	Precision	Symbol	Expression	Example
I	D	—	DEC	—	1	—	—	Single	—	YES	123
								Double	L[2]	NO	1234L
	O	'	OCT	—	—	—	—	Single	—	YES	'123
		O '	OCT	—	—	—	—	Double	L[3]	NO	'1234L
	H	$	HEX	—	—	—	—	Single	—	YES	$1A8
		X' '	HEX	—	—	—	—	Double	L	NO	$1A8L
	B	%	—	—	—	—	—	Single	—	—	—
		B' '	—	—	—	—	—	Double	—	—	—
FX	D	—	DEC	B	Req.	E	Opt.	Single	—	YES	12.5B2
				BB	Req.	E	Opt.	Double	—	NO	12.5BB
				BBB	Req.	E	Opt.	Triple	—	NO	12.5BBB
				BBBB	Req.	E	Opt	Quadruple	—	NO	12.5BBBB
FP	D	—	—	—	4	E	Opt.	Single	—	NO	1.23E-2
				—	4	D	Req.	Double	—	NO	1.23D-6

B Binary
D Decimal
FP Floating point
FX Fixed Point
0 Octal
1 Decimal integers have no decimal point, binary scaling or exponent.
2 Generates 32-bit long integers without holes.
3 Octal digits should leave hole in high order bit of second word if in non SEG mode.
4 Must be absent.

Octal: Whole number, base 8 quantities

Data Type Symbol: Apostrophe ('), letter O plus single quotes (O' ').

Precision	Constant	Listing Representation
Single Precision	DATA '123	000123
	DATA O'123'	000123
	OCT 123	000123
	'123	000123
	O'123'	000123
Double Precision	DATA '123L	000000 000123
	OCT 123L	000000 000123
	'123L	000000 000123

Hexadecimal: Whole number, base 16 quantities. The hexadecimal digit values are:

Hexadecimal	Decimal
0 - 9	0 - 9
A	10
B	11
C	12
D	13
E	14
F	15

Data Type Symbol: Dollar Sign ($), letter X plus single quotes (' ').

Precision		Constant	Listing Representation
Single Precision	DATA	$30BF'	030277
	DATA	X'30BF	030277
	HEX	30BF	030277
		X'30BF'	030277
Double Precision	DATA	$30BFL	000000 030277
	HEX	30BFL	000000 030277
		$30BFL	000000 030277

The hexadecimal and octal are bit representations, not base conversions, so if you wish to represent a 31-bit number (bit 17=0) you must explicitly specify the zero.

Binary: Whole number, base 2 quantities

Unlike the other integer data types (decimal, octal and hexadecimal), there are no special binary pseudo-operations. The general data defining pseudo-operation, DATA, may be used with the binary designator to define binary strings.

Data Type Symbol: Percent Sign (%), letter B plus single quotes (B' ').

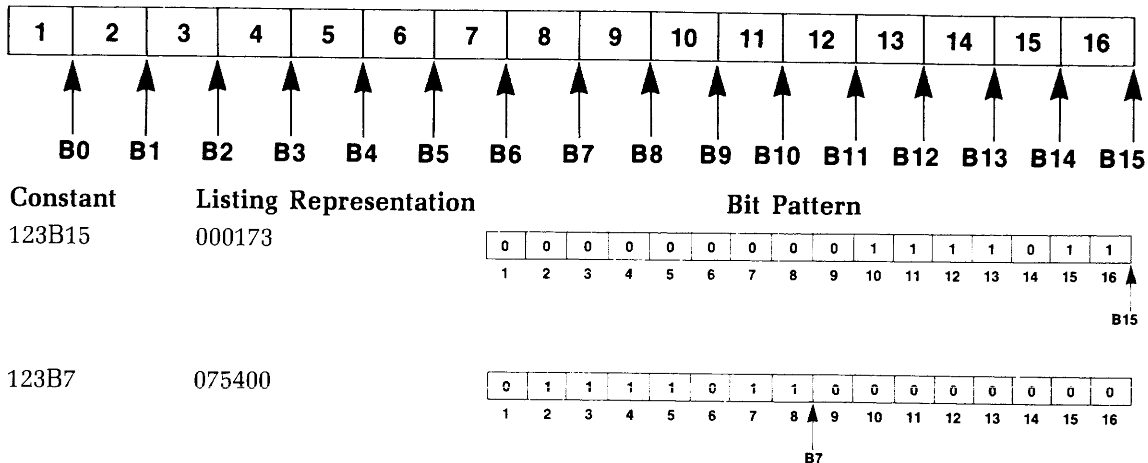
Precision		Constant	Listing Representation
Single Precision	DATA	%11100101	000345
	DATA	B'11100101'	000345
		%'11100101'	000345
		B'11100101'	000345
Double Precision	DATA	%11100101L	000000 000345

Fixed point decimal constants

Fixed point decimal constants must have an explicit binary point, expressed as a binary scale factor (see discussion below). These constants may include a decimal point and/or an exponent (see discussion below). The precision may be single, double, triple or quadruple and is indicated by the number of Bs in the binary scale factor, e.g.,

B=single
BB=double
BBB=triple
BBBB=quadruple

Binary scaling: Binary scaling, which is valid only for fixed point decimal constants, determines where the binary point will be. The figure below shows the single precision binary point positions. Bit 1 is the sign bit.



123B6 gives an assembly error because there is not enough room to the left of the binary point to contain the whole number representation of 123. This is true for all addressing modes—see Assembly Control Pseudo-Operations—and all precisions. Negative scaling is, however, permitted.

Constant	Listing Representation	Bit Pattern
123B18	000017	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 (bits 1–16)

The binary digits that extend to the right of the word are truncated without error. If this were double precision, these bits would continue into the second word. Normally, you will probably use B15 for single precision fixed point numbers.

Constant	Listing Representation	Bit Pattern
123.5B7	075600	0 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 (bits 1–16; B7 binary point between bits 8 and 9)

Note the handling of the fractional portion of the number relative to the binary point.

Precision: As stated above, there are four levels of precision for fixed point decimal numbers:

B=single—one 16-bit word

BB=double—two 16-bit words

BBB=triple—three 16-bit words

BBBB=quadruple—four 16-bit words

The format varies between SEG and non-SEG modes — see Assembly Control Pseudo-operations. In SEG mode, the sign bit (bit 1) of each subsequent word following the first is included in the binary count. Thus, in the truncation example above, bits 1 and 2=0, 3 and 4=1. In non-SEG modes, the sign bit of all subsequent words is always 0, and binary point counting continues from bit 2 of each word. The truncation example would have bit 1=0 and bits 2 and 3=1 and 4=0.

To generalize—PMA converts a constant entered as K_{10} B_n to K_2 (2**-n), where K_{10} is the decimal constant, K_2 is the same constant expressed in binary, and n is the number following the letter "B", "BB", "BBB", or "BBBB".

Powers of 10 (E): If an E code is present, it must precede the required B code. The decimal value of the constant is multiplied by the power of 10, specified by the integer following the E, before it is converted to binary. The exponent may be positive or negative.

Again, be careful to ensure that there is enough room to the left of the binary point to hold all the digits. If not, an error will occur.

SEG examples:

Precision	Constant	Listing Representation
Double Precision	123E1BB17	000463 100000
Single Precision	123E0B15	000173
	123E1B15	002316

Non-SEG examples:

Precision	Constant	Listing Representation
Double Precision	123E1BB17	000463 040000
Single Precision	123E0B15	000173
	123E1B15	002316

Floating point decimal constants

Binary Scaling must not be used in floating point constants.

Single precision floating point: Single-precision floating point quantities are expressed by a decimal fraction, with or without a decimal exponent (Emm).

Constant	Listing Representation
1.28E2	040000 000210
1.28	050753 102601
–11.28	122702 107604
1.28E–14	071512 145122

The assembler converts the specified values to an 8 bit binary exponent and a 23 bit binary fraction in two successive words, as shown in Figure 15-1. The exponent is represented in excess-128 notation, and can range from 2**–127 to 2**+127 (roughly 10**–38 to 10**+38). An error message is generated if the exponent exceeds this range. The assembler automatically generates a normalized fraction of the largest possible value less than 1. Numbers specified in this format have significant decimal digits.

Negative numbers are formed by generating a positive number of the specified magnitude and then forming the two's complement of both data words, excluding the exponent. The number zero is assembled as two consecutive all-zero data words.

Double precision floating point: Double precision floating point quantities are expressed by a decimal integer or fraction with a decimal exponent (Dmm).

The assembler converts the specified value to a 16-bit binary exponent and 47-bit binary fraction, in four successive words, as shown in Figure 15-1. The exponent is represented in the same excess-128 notation as single-precision floating point. The assembler automatically generates a normalized fraction of the largest possible value less than 1.

Negative numbers are formed by generating a positive number of the specified magnitude and then taking the two's complement of all three fraction words, excluding the exponent. The number zero is assembled as consecutive all-zero data words.

Character (ASCII) constants

ASCII character strings are specified by the letter C followed by the string enclosed in apostrophes, and are packed two per 16-bit word. Colons (:) and semicolons (;) may be encoded by preceding them with the PMA escape character (an exclamation point).

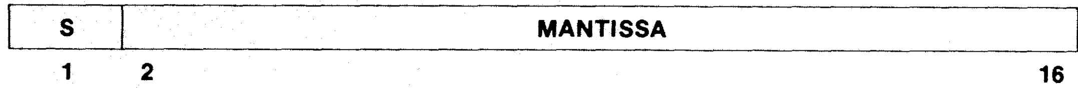
Constant	Listing Representation
C'AB'	140702
C'A'	140640

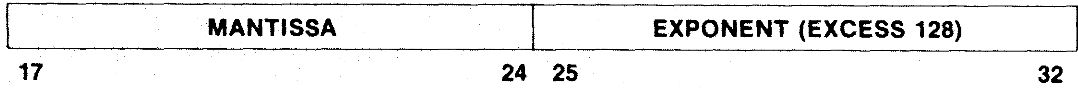
Single characters defined by C'' are left-justified with the right half of the word filled with a blank (ASCII representation '240). Single characters defined by R'' are right-justified with the left half of the word filled with zeros.

Constant	Listing Representation
C'A'	140640
R'A'	000301

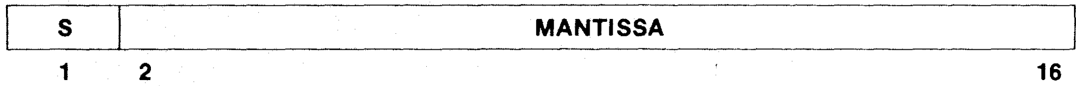
TERMS

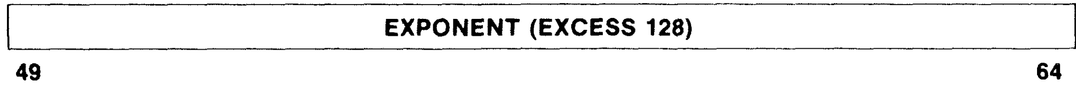
A term is the smallest element of PMA which represents a separate and distinct value. It has a single precision integer value (fits in a 16-bit word) and may be a constant or a symbol. Terms may be used alone or in combination with other terms to form expressions.





A. Single Precision Floating Point





B. Double Precision Floating Point

Figure 15-1. Floating Point Data Formats

Every term, whether used alone or in an expression, has both a value and a mode which are either defined by the assembler or inherent in the term itself. Symbols defined by the EQU, SET, and XSET pseudo-operations receive both the mode and the value of the term or evaluated expression; labels take the current mode and value of the program counter (see Origin Control pseudo-operations for a discussion of how the mode of the program counter is set). Examples include:

'123	Octal constant.
C'A'	ASCII constant.
ALPHA	Symbol.
1.23E2	Invalid because it is a floating point number; it does not have a single precision integer value.
C'ABC'	Invalid because the value is too large for a 16-bit word.

Value

The value of a term is the numeric representation which is assembled into the object code. It can be a location or data.

Symbol	Usage	Explanation
LABSYM	LABSYM LDA LOC	LABSYM is a label symbol whose value is a location (program counter value) of the instruction LDA LOC.
DATSYM	DATSYM DATA '10	DATSYM is a label symbol whose value is the location (program counter value) of the constant '10.
ADSYM	ADSYM DAC LOC	ADSYM is a label symbol whose value is the location (program counter value) of the address constant LOC.
ABSSYM	ABSSYM EQU '10	ABSSYM is a symbol whose value is '10.
CHRSYM	CHRSYM EQU C'A'	CHRSYM is a symbol whose octal value is 140640.

Mode

The mode defines whether the value associated with a symbol is absolute or relative. Table 15-2 summarizes the use of the modes defined below.

Absolute: The value of the symbol does not change upon program relocation. Symbols equated to constants and the results of expression operations other than addition and subtraction have a mode of absolute.

Stack relative: The symbol is defined relative to the start of the stack area. Variables defined by the DYNM pseudo-operation or by a + value (non-segmented addressing modes) or by SB%+value (segmented addressing modes) have a mode of stack relative.

External: The symbol is defined in a separately assembled module and is identified by an EXT pseudo-operation.

Procedure absolute (SEG or SEGR only): The symbol is defined relative to the start of the procedure segment and is identified by PB%+value.

Linkage base relative (SEG or SEGR only): The symbol is defined relative to the start of the link frame and is identified by LB%+value, or * if the origin is LINK frame.

Temporary base relative (SEG or SEGR only): The symbol is defined relative to the contents of the temporary base register and is identified by XB%+value.

Relative (Procedure relative (SEG or SEGR): The symbol is defined relative to the start of the module.

Common: The symbol is defined relative to a common data area which has been defined by a COMM pseudo-operation. This common data area may be shared by several independently assembled routines.

Table 15-2. Modes

Mode	Generated By	Symbol	Usage	Value	Listing Representation
Absolute	Labels[1] Constants **	ABSSYM	ABSSYM EQU '123	'123	000123A
	Expression Operations	EXPSYM	EXPSYM EQU '3+'4	'7	000007A
Relative (non-SEG) Procedure (SEG) Relative	Labels	LABSYM	LABSYM LDA LOC	Current PC	Current PC
Common	COMM Pseudo-Operation	B	COMM A, B, C(3), D	'1[2]	000001C
Stack Relative	DYNM Pseudo-Operation	STKSYM	DYNM ('3)	'3	000003S
	@(Non-SEG mode)	SKSYM2	SKSYM2 EQU @+'6	'6	000006S
	SB% (SEG modes)	SKSYM3	SKSYM3 EQU SB%+'7	'7	000007S
External	EXT Pseudo-Operation	EXTSYM	EXT EXTSYM	0	000000E
Procedure Absolute	PB%	PBSYM	PBSYM EQU PB%+'2	'2	000002P
Linkage Base Relative	LB%	LBSYM	LBSYM EQU LB%+'5	'5	000005L
	LINK	LBSYML	LINK		
			LBSYM DAC 5	'5	000005L
Temporary Base Relative	XB%	XBSYM	XBSYM EQU XB%+'3	'3	000003T

Notes

1 - If PB% or ABS

2 - Offset from start of COMMON.

EXPRESSIONS

As described in Section 14 - Language Structure, expressions contain one or more terms (constants or symbols) joined by operators.

Operators

Expressions may contain arithmetic, logical, relational and shift operators.

Arithmetic operators: Perform addition, subtraction, multiplication, and division operations:

Operator	Meaning	Example	Result (Octal)
+	Addition	'3+'4	000007
-	Subtraction	'10-'3	000005
*	Multiplication	'20*'10	000200
/	Division	'20/'10	000002

Logical operators: Perform a logical operation on two 16-bit operands:

Operator	Meaning	Example	Result (Octal)
.OR.	Logical OR	'123.OR.'456	000577
.XOR.	Logical Exclusive OR	'123.XOR.'456	000575
.AND.	Logical AND	'123.AND.'456	000002

Relational operators: Perform a comparison of two 16-bit operands with a result of 0 if false and 1 if true.

Operator	Relation	Example	Result (Octal)
.EQ.	Equal	'123.EQ.'123	000001
		'123.EQ.'456	000000
.NE.	Not equal	'123.NE.'123	000000
		'123.NE.'456	000001
.GT.	Greater than	'123.GT.'123	000000
		'456.GT.'123	000001
.GE.	Greater than or equal	'123.GE.'123	000001
		'123.GE.'456	000000
.LE.	Less than or equal	'123.LE.'123	000001
		'123.LE.'456	000001
.LT.	Less than	'123.LT.'456	000001
		'456.LT.'123	000000

Shift operators: Perform logical right or left shift of an expression, using the syntax:

argument-expression {.LS. | .RS.} shift-count-expression

Operator	Meaning	Example	Result (Octal)
.LS.	Left shift	'123.LS.'3	001230
.RS.	Right shift	'123.RS.'3	000012

Usage

Space conventions: Operators may be preceded and/or followed by a single space (more than one space causes PMA to treat the rest of the line as a comment).

Sign conventions: The operands for arithmetic operators may be signed.

Operator priority: In expressions with more than one operator, the operator with the highest priority is performed first. In cases of equal priority, the evaluation proceeds from left to right. You may use parentheses to alter the natural order of evaluation.

Priority	Operator
Highest	* /
	+ -
	.RS. .LS.
	.GT. .GE. .EQ. .NE. .LE. .LT.
	.AND.
	.OR.
Lowest	.XOR.

Resultant mode: For all operations other than addition and subtraction, the mode of both operands must be absolute and the result is absolute.

When an addition operator is used, at least one of the operands must be absolute, and the result mode is the mode of the other operand.

When a subtraction operator is used, at least the second operand must be absolute, and the result mode is the mode of the first operand.

LITERALS

A literal is an expression preceded by an equal-sign (=), as in:

```
= '37
= *+'37
```

You can use literals as operands in order to introduce data into your program. You cannot, however, use a literal as a term in an expression.

The assembler places the data which you specified in a literal into a "literal pool". It then assembles the address of this literal into the object code of the instruction that contains the literal specification. Thus the assembler saves you a programming step by storing your literal data for you.

Literals may be one or more words in length and may contain any legal data item or expression. Note, however, that if you use colon (:) or semicolon (;) in a character constant construction, you *must* precede it by an exclamation mark (!) escape character.

Non-SEG usage

When RLIT is used after a FIN statement, literals which have been already collected in a literal pool by the FIN will not be redefined. For example:

```
         REL
                              (0001)         REL
                              (0002)         RLIT
  000000:       02.000001     (0003)         LDA   ='123      '123 IS AN OCTAL LITERAL
  000001:       00.000123A    (0004)         FIN

  000002:       02.000001     (0005)         LDA   ='123      '123 IS IN THE SAME LITERAL
*                                                               POOL AS '123 ABOVE
  000003:       04.000005     (0006)         STA   BUFF
  000004:       02.000006     (0007)         LDA   ='456      '456 WILL BE IN BUFFER POOL
*                                                               AFTER END
  000005:       00.000010A    (0008) BUFF    DAC   '10        DEFINE BUFFER AREA
               000006         (0009)         END              END OF SOURCE CODE

  000006:       00.000456A
```

SEG usage

Literals may be placed in either the procedure segment or in the linkage segment. If an RLIT pseudo-op is used, literals will be generated in the same way as in a non-SEG assembly with an RLIT. If an RLIT pseudo-op is not used, the literals will be placed in the linkage frame.

The FIN pseudo-op may still be used to control the placement of literals, but the assembly origin at the time a FIN occurs can affect the literal placement.

If RLIT mode is specified and a FIN occurs while in linkage origin, the FIN will act as:

```
HERE    EQU  *
        PROC
        FIN
        ORG  HERE
```

Correspondingly, if not in RLIT mode and a FIN occurs while in precedure origin, the FIN will have the effect of:

```
HERE    EQU  *
        LINK
        FIN
        ORG  HERE
```

ASSEMBLER ATTRIBUTES

Assembler attributes can be specified by a number proceded by the pound character (#). The attribute number may be a variable, or an expression within parentheses, as long as such variables have been previously defined as absolute integer values. Attribute references are evaluated as absolute integer values, and may be used in both macro definitions and macro calls. See Appendix A for a complete list.

16
Pseudo-operations

INTRODUCTION

Pseudo-operation statements are commands to the assembler, rather than executable machine instructions. Pseudo-operation functions include:

- Assembly control (AC)
- Address definition (AD)
- Conditional assembly (CA)
- Data definition (DD)
- Listing control (LC)
- Literal (LT)
- Loader control (LO)
- Macro definition (MD)
- Program linking (PL)
- Storage allocation (SA)
- Symbol definition (SD)

Table 16-1 contains an alphabetical listing of all the pseudo-operations, their functional class and their restrictions, if any.

Pseudo-operations have an operation field and an operand field separated by spaces. Labels are usually optional, but some pseudo-operations either require a label to be present, or prohibit it (see Figure 16-1).

The operation field contains the mnemonic name that identifies the pseudo-operation.

The operand field may contain one or more terms separated by single spaces or commas. Terms may be constants, symbols, or expressions as defined in Section 15. In certain operations, such as BCI, terms may also consist of ASCII character strings.

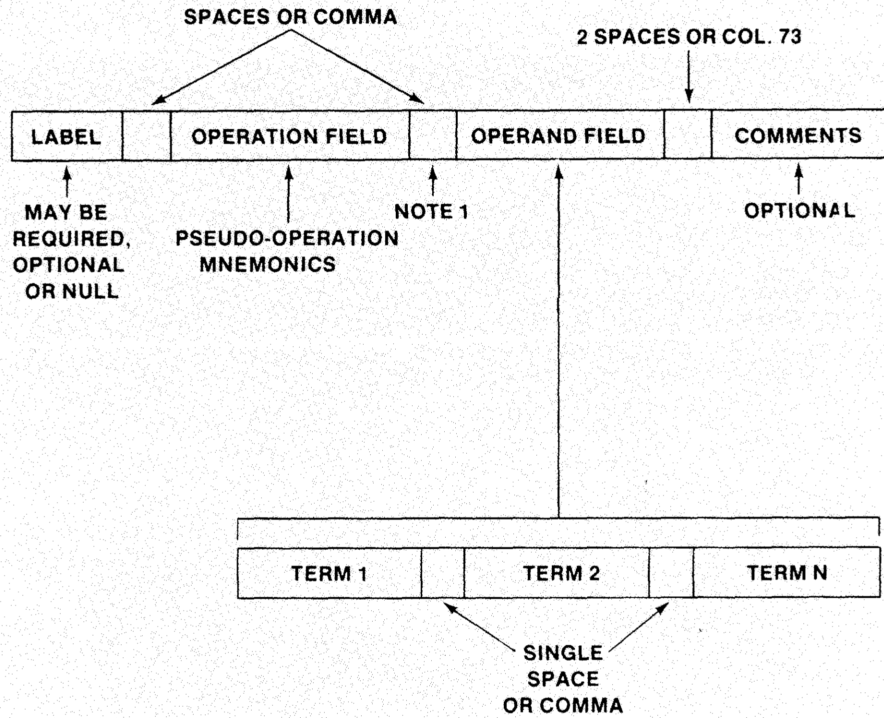
Symbols used in the operand field must be previously defined, unless otherwise stated in the pseudo-operation definition.

Address expressions are evaluated as 16-bit integer values and used as a 16-bit memory address, unless otherwise stated. Certain statements (DAC and XAC) accept indirect address (*) and indexing (,1) symbols. These are interpreted according to the addressing mode which is in effect.

Table 16-1. Pseudo-Operation Summary

Name	Function	Class	Restriction
ABS	Set mode to absolute	AC	*Not in SEG/SEGR MODE*
AP	Argument pointer	AD	*SEG/SEGR mode*
BACK	Loop back	CA	*Macros only*
BCI	Define ASCII string	DD	
BES	Allocate block ending with symbol	SA	
BSS	Allocate block starting with symbol	SA	
BSZ	Allocate block set to zeros	SA	
C64R	Check 64R	AC	
CALL	External subroutine reference	PL	
CENT	Conditional entry	LO	
COMM	FORTRAN compatible COMMON	SA	
D16S	Use 16S address mode	LO	
D32I	Use 32I address mode	LO	
D32R	Use 32R address mode	LO	
D32S	Use 32S address mode	LO	
D64R	Use 64R address mode	LO	
D64V	Use 64V address mode	LO	
DAC	Address definition (Prime 100-300)	AD	
DATA	Define data constant	DD	
DDM	Use default mode	LO	
DEC	Define decimal integer constant	DD	
DFTB	Define table block	CA	
DFVT	Define value table	CA	
DUII	Define UII	LO	
DYMN	Define stack relative symbol	SD	
DYNT	Direct entry definition	PL	
ECB	Entry control block	PL	*SEG/SEGR mode*
EJCT	Eject page	LC	
ELM	Enter loader mode	LO	
ELSE	Reverse conditional assembly	CA	
END	End of source statements	AC	
ENDC	End conditional assembly area	CA	
ENDM	End macro definition	MD	
ENT	Define entry point	PL	
EVEN	Outputs NOP if current address not even	AC	
EQU	Fixed symbol definition	SD	
EXT	External reference	PL	
FAIL	Force error message	CA	
FIN	Insert literals	LT	
GO	Forward reference	CA	
HEX	Define hexadecimal integer constant	DD	
IFTT	If table true	CA	
IFTF	If table false	CA	
IFVT	If value true	CA	
IFVF	If value false	CA	
IF	If true	CA	
IFx	Arithmetic conditional if	CA	
IP	Indirect pointer	AD	*SEG/SEGR mode*
LINK	Put code in linkage segment	AC	*SEG/SEGR mode*

LIR	Load if required	LO	
LIST	Enable listing	LC	
LSMD	List macro expansions data only	LC	
LSTM	List macro expansions	LC	
MAC	Begin macro definition	MD	
N64R	Not 64R	LO	
NLSM	Don't list macro expansions	LC	
NLST	Inhibit listing	LC	
OCT	Define octal integer constant	DD	
ORG	Define origin location	AC	
PCVH	Print cross reference values in HEX	LC	
PROC	Put code in procedure segment	AC	*SEG/SEGR mode*
REL	Set mode to relocatable	AC	*Not in SEG/SEGR MODE*
RLIT	Optimize literals	LT	
SAY	Print message	MD	
SCT	Select code within macro	MD	
SCTL	Select code from macro list	MD	
SDM	Set default mode	LO	
SEG	Segmentation assembly - 64V	AC	*Must be first statement in listing*
SEGR	Segmentation assembly - 32I	AC	*Must be first statement in listing*
SET	Changeable symbol definition	SD	
SETB	Set base sector	LO	
SUBR	Define entry point	PL	
SYML	Allows long external names	PL	*SEG/SEGR mode*
VFD	Define variable fields	DD	
XAC	External address definition	AD	
XSET	Changeable symbol definition	SD	



NOTE:
IF MORE THAN 10 SPACES FOLLOW THE OPERATION FIELD, THE ASSEMBLER ASSUMES THERE IS NO OPERAND FIELD AND TREATS THE REST OF THE LINE AS COMMENTS.

Figure 16-1. Pseudo-operations.

ASSEMBLY CONTROL PSEUDO-OPERATIONS (AC)

Affect the actions of PMA during the assembly process.

Name	Function	Restriction
ABS	Set mode to absolute	*Not in SEG/SEGR mode*
C64R	Check 64R	
END	End of source statements	
EVEN	Outputs NOP if current address not even.	
LINK	Put code in linkage segment	*SEG/SEGR mode*
ORG	Define origin location	
PROC	Put code in procedure segment	*SEG/SEGR mode*
REL	Set mode to relocatable	*Not in SEG/SEGR mode*
SEG	Segmentation assembly - 64V	*Must appear before executable code*
SEGR	Segmentation assembly -32I	*Must appear before executable code*

▶ ABS

Sets the assembly and loading mode to absolute, within the program address space. ABS may be terminated by REL and vice versa. ABS mode is the default assembly mode.

▶ C64R

Directs the assembler to flag any instructions and/or memory reference not compatible with 64R addressing mode. The following cases are detected:

1. An indirect or indexed DAC
2. An indirect, single-word memory reference instruction with an address that is neither in sector zero nor within the relative reach of the instruction.

▶ [label] END [address-expression]

Terminates assembly of the source program. All literals accumulated since either the start of the program, or the last FIN statement, are assigned locations starting at the current location count. If the program is segmented and literals are in the linkage segment, refer to Literal Control Pseudo-Operations for further information.

If an **address-expression** is not specified, the first location of the first module becomes the start address; otherwise, the address-expression defines the start address.

▶ [label] EVEN

Outputs a NOP if the current address is not even. Forces even alignment only for the instruction or data immediately following the EVEN.

▶ LINK

Places subsequent code in the linkage frame. The program counter value is set to one more than the highest value previously used in the linkage area. The mode is set to link. LINK may be terminated by PROC and vice versa.

▶ [label] ORG address-expression

Sets the assembler location count equal to the value of the **address-expression**.

In *non-SEG* mode, the mode of the address-expression may be absolute, relative or common. The value of the program counter is set to the value of the expression. If the mode of the address-expression is common, then the mode of the program counter is set to common. If the mode of the *program counter* is common, then it is set to the mode of the address-expression. Otherwise, the mode of the program counter remains unchanged.

In *SEG-mode*, the mode of the address-expression may be absolute, procedure relative, linkage or common. The value of the program counter is set to the value of the address-expression. If the mode of the address-expression is absolute, then the mode of the program counter remains unchanged. In all other cases whether relative, linkage, or common both the mode and the value of the program counter are set equal to that of the address-expression.

▶ PROC

Places subsequent code in the procedure segment. The value of the program counter is set to one more than the highest value previously used in the procedure segment. The mode is set to procedure relative. PROC may be terminated by LINK or vice versa.

▶ REL

Sets the assembly and loading mode to relocatable. REL may be terminated by ABS and vice versa.

▶ SEG

Directs the assembler to create a 64V segmented mode assembly module. SEG *must* appear before any instructions, pseudo-operations or macro calls which generate instructions, as well as before any DYNM pseudo-operations. D64V *(q.v.)* only governs instruction formats; it does not create a segmented module. Modules assembled in SEG mode must be loaded by SEG — the PRIMOS segmented loader utility.
SEG has the following effects:

- Sets the assembler into three pass assembly mode to optimize stack and link frame references.
- Removes restrictions on placement of DYNM pseudo-operations.
- Sets the instruction and address resolution mode to D64V.
- Initializes the assembler program counter to procedure relative zero.

▶ SEGR

Directs the assembler to create a 32I segmented mode assembly module. SEGR *must* appear before any instructions, pseudo-operations or macro calls which generate instructions, as well as before any DYNM pseudo-operation. D32I (*q.v.*) only governs instruction formats; it does not create a segmented module. Modules assembled in SEGR mode must be loaded by SEG — the PRIMOS segmented loader utility.

SEGR has the following effects:

- Sets the assembler into three pass assembly mode to optimize stack and link frame references.
- Removes restrictions on placement of DYNM pseudo-operations.
- Sets the instruction and address resolution mode to D32I.
- Initializes the assembler program counter to procedure relative zero.

ADDRESS DEFINITION PSEUDO-OPERATIONS (AD)

Create address constants which may be referenced by instruction statements.

Name	Function	Restriction
AP	Argument pointer	*SEG/SEGR mode*
DAC	Local address definition	
IP	Indirect pointer	*SEG/SEGR mode*
XAC	External address definition	

▶ [label] AP address-expression [,modifier]

Generates an argument pointer in the form used by the 64 V/32I Procedure Call instruction (PCL). **address-expression** is an argument variable, written in memory reference address format. All 64V address forms may be used - except indexing. **modifier** controls the storage of address-expression as follows:

- **S** Set argument store bit.
- **SL** Set argument store bit. Last argument.
- ***S** Set argument store bit. Argument is indirect.
- ***SL** Set argument store bit. Argument is indirect and last.
- * Intermediate indirect argument. Do not store.

See Section 9 - Data Formats - for the argument pointer machine format.

▶ [label] DAC address-expression

Generates a 16-bit pointer in SR addressing format. **address-expression** is written in SR addressing format, with indirect addressing and indexing specified by * and ,1 respectively. If REL mode (see Assembly Control Pseudo-Operations) is in effect, the loader performs relocation during loading.

The assembler generates a 16-bit constant which is acted upon by the loader as follows:

Addressing mode	Address Length	Index	Indirection
16S	14	YES	YES
32S	15	NO	YES
32R	15	NO	YES
64R	16	NO	NO
64V	16	NO	NO

[label] DAC ** is a convention used to indicate a subroutine entry point (see Program Linking Pseudo-Operations).

▶ [label] IP address-expression

Generates a 32-bit 64V/32I indirect pointer. **address-expression** must be one of the following: procedure relative, linkage relative, common or external. SEG sets up the pointer at load time.

▶ [label] XAC symbol

Generates a 16-bit pointer to the external **symbol**. The symbol name may be the same as a local symbol without conflict. XAC is like a DAC except that it references external symbols. The address of the external symbol is filled in at load time.

CONDITIONAL ASSEMBLY PSEUDO-OPERATIONS (CA)

Enable the programmer to selectively assemble portions of a source file.

Name	Function	Restriction
BACK, BACK TO	Loop back	*Macros only*
DFTB	Define table block	
DFVT	Define value table	
ELSE	Reverse conditional assembly	
ENDC	End conditional assembly area	
FAIL	Force error message	
GO	Forward reference	
IF	Conditional	
IFx	Conditional	
IFTF	If table false	
IFTT	If table true	
IFVF	If value false	
IFVT	If value true	

▶ **[label-1] {BACK | BACK TO} label-2;**

Directs the assembler to repeat source statements that have already been assembled, beginning with the statement specified by **label-2**. Such backward references are permitted only within a macro definition. Both the BACK, BACK TO and label-2 must lie within the same MAC-ENDM range.

▶ **label DFTB (symbol, absolute-expression–1),. . .**

Creates a programmer symbol table. **label** is a table name, **symbol** is an argument whose value is **absolute-expression–1.** The symbols defined have no conflict with existing symbols.

For example:

```
A       EQU  5
X       DFTB (A,1),(B,2)
X       EQU  6
```

There are no conflicts since the two X's and A's are different types.

If a DFTB is defined with the same name as a previously defined table, the contents are appended to the previous table.

▶ **label DFVT (absolute-expression–1, absolute-expression–2),. . .**

Creates a programmer value table. **absolute-expression–1** is the locator value and **absolute-expression–2** is the value to be substituted.

▶ **ELSE**

Reverses the condition set up by an IFx statement until the matching ENDC statement is reached. If the IFx condition inhibited assembly, the ELSE statement enables assembly, and vice versa. ELSE statements that lie within the bounds of other IFx-ENDC pairs nested within the conditional assembly area are ignored.

▶ ENDC

Defines the end of a conditional assembly area started by an IFx statement. Every IFx statement *must* have a matching ENDC.

▶ FAIL

Generates an F error in the listing.

▶ [label] IF absolute-expression, statement

Provides the ability to selectively assemble code based on the results of a test. The operand consists of an **absolute-expression** followed by a **statement**. If the expression is true (has a non-zero result) the statement is assembled; otherwise the statement is ignored and the next line is processed. The operand of the IF statement cannot be continued onto the following line, because the skip-if-false condition proceeds to the next physical, rather than logical, line.

▶ [label] { IFM | IFP | IFZ | IFN } absolute-expression

Sets specific tests to control code assembly. The **absolute-expression** is evaluated, and if the result corresponds to the IF condition, assembly proceeds normally. Otherwise, the assembler ignores all subsequent statements until an ENDC or ELSE statement is reached.

For every IFx statement there *must* be a matching ENDC statement. IFx and ENDC pairs may be nested within each other. The nesting depth count is checked even in sections of code that are being skipped by a previous IFx statement.

▶ label IFTF symbol

Searches the table, whose name is **label,** for **symbol.** If symbol is not found, puts its value in assembler attribute 124 and assembles code to the matching ELSE or ENDC. If the symbol is found, skip to the line following the matching ELSE or ENDC.

▶ label IFTT symbol

Searches the table, whose name is **label,** for **symbol.** If symbol is found puts its value in assembler attribute 124 and assembles code to the matching ELSE or ENDC. If the symbol is not found, skip to the line following the matching ELSE or ENDC.

▶ label IFVF symbol

Obtains a value in a value table. If **symbol** does not equal a locator value in the value table whose name is **label,** put its value in assembler attribute 124 and assemble the code to the matching ELSE or ENDC. If the locator value is found, skip to the line following the matching ELSE or ENDC.

▶ label IFVT symbol

Obtains value in a value table. If **symbol** equals a locator value in the value table whose name is **label,** puts its value is assembler attribute 124 and assembles the code to the matching ELSE or ENDC. If the locator value is not found, skip to the line following the matching ELSE or ENDC.

▶ **{GO | GO TO} label**

Causes suspension of assembly of all subsequent statements until a statement having the specified **label** is found. The GO or GO TO statement *must* point forward to a label that is not yet defined. An error condition exists if the assembler reaches an END, MAC, or ENDM statement before finding the specified label.

DATA DEFINING PSEUDO-OPERATIONS (DD)

Initialize memory locations to known starting values. For coding convenience, data and address constants may be specified in a variety of formats. Simple coding conventions allow the programmer to use decimal, octal, binary and hexadecimal integers, decimal floating point, and ASCII character constants. The assembler interprets the notation and automatically generates one, two or more data words in the proper internal binary format.

Name	Function	Restriction
BCI	Define ASCII string	
DATA	Define data constant	
DEC	Define decimal integer constant	
HEX	Define hexadecimal integer constant	
OCT	Define octal integer constant	
VFD	Define variable fields	

▶ **[label] BCI {'string' | n, string}**

Loads ASCII character strings by packing the specified ASCII characters two per word, starting with the most significant 8 bits. Assembled words are loaded starting at the current location count.

In the *first* format, the **string** is delimited by any character other than a digit. If an odd number of characters is specified, the least significant half of the last word is filled with zeroes.

In the *second* format, the **string** is preceded by a word count, **n**, which is the number of characters divided by two and rounded up.

▶ **[label] DATA [(absolute-expression-1)] absolute-expression-2, . . .**

Defines **absolute-expression-2** **absolute-expression-1** times. Absolute-expression-1 is assumed to be 1 if omitted.

The operand expression(s) are assembled into the current location. The operand may contain any number of subfields, separated by commas. Subfields are assembled in consecutive locations starting with the left-most subfield. If an expression requires more than one location (e.g. floating point), consecutive locations are used. See Section 15 - Data Definition - for a full discussion of allowable formats.

▶ **[label] DEC decimal-integer-constant, . . .**

Defines decimal integers. Each **decimal-constant** in the operand is evaluated as a decimal constant, converted into one or more binary words, and loaded starting at the current location. All numeric formats accepted by the DATA statement may be used with DEC.

▶ **[label] HEX hexadecimal-integer-constant, . . .**

Defines hexadecimal integers by converting the **hexadecimal-constants** within the operand to 16-bit integer values and loading them in consecutive locations starting at the current location.

▶ **[label] OCT octal-integer-constant, . . .**

Defines octal integers. Each **octal-constant** in the operand is loaded at the current location.

▶ **[label] VFD absolute-expression-1, absolute-expression-2, . . .**

Permits 16-bit data words to be formed with subfields of varying length. In the pairs of constants, **absolute-expression-1** gives the subfield size, **absolute-expression-2** gives the value. The first pair is the most significant subfield; subsequent field size value pairs load less significant subfields of the 16-bit word. For any pair, if a value exceeds the specified field size, the more significant overflow bits are exclusive-OR'ed with the subfield to the left. No error message is generated. If the entire word is not specified, the least significant end is filled with zeroes.

An error message is printed if the assembler attempts to load more than 16 bits.

LISTING CONTROL PSEUDO-OPERATIONS (LC)

Format the assembler listing.

Name	Function	Restriction
EJCT	Eject page	
LIST	Enable listing	
LSMD	List macro expansions data only	
LSTM	List macro expansions	
NLSM	Don't list macro expansions	
NLST	Inhibit listing	
PCVH	Print cross reference values in hexadecimal	

▶ **EJCT**

Causes the listing device to eject the page (execute a form feed), print the current page title and page number, and feed three blank lines before resuming the listing. This function is operable *only* with devices having a mechanical form feed capability, such as a line printer.

▶ **LIST**

Lists all statements except those generated by macro expansions. Since this is the assembler's default mode, a LIST statement is not required unless a NLST statement has previously inhibited listing.

▶ **LSMD**

Lists macro calls plus any data generated by macros.

▶ **LSTM**

Lists macro call statements plus all lines generated by the macro expansion including code and data values.

▶ NLSM

Inhibits listing of statements generated by macro expansion. Only the macro call is listed. Ignored if -EXPLIST command line option is specified.

▶ NLST

Inhibits listing of all subsequent statements until a LIST statement is encountered. LIST and NLST may be used together in source text for selective control over the sections to be listed. The LSTM, LSMD, and NLSM statements provide control of listing for macro definitions. Ignored if -EXPLIST command line option is specified.

▶ PCVH

Prints symbol values in the cross reference in hexadecimal instead of octal.

LITERAL CONTROL PSEUDO-OPERATIONS (LT)

Govern the placement of literals. Also see END, which is described under Assembly Control Pseudo-Operations.

Name	Function	Restriction
FIN	Insert literals	
RLIT	Optimize literals	

▶ [label] FIN

Controls the placement of literal pools. All literals defined since an RLIT statement, the start of the program, or the last FIN statement, are assembled into a literal pool starting at the current location. Processing of subsequent statements begins at the first location following the literals. By using FIN, the programmer can distribute literals throughout the program, and possibly reduce the number of out-of-range indirect address links that must be formed by the loader to access literals.

▶ [label] RLIT

Directs the assembler to optimize literal allocation for relative addressing modes (32R, 64R, 64V, 32I modes). Normally (i.e., without RLIT), literals are assigned locations following a FIN or END statement. If a defined literal is referenced following a FIN, it is assigned another location following the next FIN or END statement. However, in a program that contains an initial RLIT, a literal that has already been defined and is still within the relative or multiword reach (see Section 10 - Memory Reference) is referenced directly, without allocating a new location.

Note

RLIT must precede executable code.

Normally in SEG or SEGR mode, literals would be placed in the linkage segment; issuance of RLIT causes placement of literals in the procedure segment.

Usage (non-SEG or SEGR): When RLIT is used with a FIN statement, literals which have been already collected in a literal pool by a FIN will not be redefined. For example:

```
_         REL
                                (0001)          REL
                                (0002)          RLIT
  000000:       02.000001       (0003)          LDA  ='123      '123 IS AN OCTAL LITERAL
  000001:       00.000123A      (0004)          FIN

  000002:       02.000001       (0005)          LDA  ='123      '123 IS IN THE SAME LITERAL
*                                                                POOL AS '123 ABOVE
  000003:       04.000005       (0006)          STA  BUFF
  000004:       02.000006       (0007)          LDA  ='456      '456 WILL BE IN BUFFER POOL
*                                                                  AFTER END
  000005:       00.000010A      (0008) BUFF     DAC  '10        DEFINE BUFFER AREA
              000006            (0009)          END             END OF SOURCE CODE

  000006:       00.000456A
```

Usage (SEG/SEGR): The FIN pseudo-operation may still be used to control the placement of literals, but the assembly origin at the time a FIN occurs and the use of RLIT can affect the literal placement.

If RLIT is specified and a FIN occurs while in linkage origin, the FIN will act as:

```
HERE     EQU  *
         PROC
         FIN
         ORG  HERE
```

Correspondingly, if RLIT is not specified and a FIN occurs while in procedure origin, the FIN will have the effect of:

```
HERE     EQU  *
         LINK
         FIN
         ORG  HERE
```

LOADER CONTROL PSEUDO-OPERATIONS (LO)

Provide control information for the loader. Addressing mode control pseudo-operations (D16S, D32S, D64R, D64V, D32I) control the assembler memory reference instruction processing as well as loader address resolution mode. Mode commands entered during loading set the loader's current mode only, and are overridden by mode control pseudo-operations in the program.

Incompatible instructions (e.g., a 64V instruction in 32R mode), are flagged by the assembler. The default mode of the assembler is relative, unless a SEG pseudo-operation is used, in which case 64V mode is the default.

Note that DUII, LIR and CENT simplify the preparation of library packages that automatically load the modules appropriate to the machine in which the code is to be executed.

Name	Function	Restriction
CENT	Conditional entry	
D16S	Use 16S addressing mode	
D32S	Use 32S addressing mode	
D32R	Use 32S addressing mode	
D64R	Use 64R addressing mode	
D64V	Use 64V addressing mode	
D32I	Use 32I addressing mode	
DDM	Use default mode	
DUII	Define UII	
ELM	Enter loader mode	
LIR	Load if required	
N64R	Not 64R	
SDM	Set default mode	*Not 64V or 32I*
SETB	Set base sector	

▶ CENT symbol

Provides a conditional ENT capability. The loader will load a module containing a CENT only if something else in the module — such as an LIR — tells it to load the module. This is true even if the CENT **symbol** matches an unresolved external reference.

Typically, a module containing a CENT will be part of a library.

▶ D16S

Directs the assembler and the loader to use 16R address resolution.

▶ D32R

Directs the assembler and the loader to use 32R address resolution.

▶ D32S

Directs the assembler and the loader to use 32S address resolution.

▶ D64R

Directs the assembler and the loader to use 64R address resolution.

▶ D64V

Directs the assembler and the loader to use 64V address resolution.

▶ D32I

Directs the assembler and the loader to use 32I address resolution.

▶ DDM

Directs the assembler and the loader to use the default addressing mode. The default mode is initially set at the start of a load and is only changed by a SDM pseudo-operation.

▶ DUII absolute-expression-1, absolute-expression-2

Triggers the loading of a UII package. **absolute expression-1** is a bit mask, defining instruction sets that the UII package emulates, and **absolute-expression-2** is a bit mask, defining hardware instruction sets that must be present to execute the UII package.

Bit number	Meaning
1–9	Must be 0
10	Prime 450 and up
11	Prime 300, 400
12	Undefined
13	Double Precision Floating Point
14	Single Precision Floating Point
15	Prime 300 Only
16	High Speed Arithmetic

▶ ELM

Causes the loader to generate an enter addressing mode instruction in the current loader addressing mode at the current counter.

▶ LIR absolute-expression

Controls library program loading. The program will be loaded if any of the instruction groups specified have been used in previously loaded code. **absolute-expression** is a bit mask, defining instruction groups that are to trigger loading. Bit assignments are the same as for DUII.

▶ [label] N64R

Informs the loader that the program is not to be loaded in the 64R addressing mode. If such a program is loaded in the 64R addressing mode, the loader will report a 'N6' error.

▶ SDM absolute-expression

Directs the loader to set its default addressing mode to **absolute-expression**. Legal values of the expression are:

0	16S Mode
1	32S Mode
2	64R Mode
3	32R Mode

SDM does not change the current addressing mode and *cannot* be used in 64V or 32I modes.

▶ [label] SETB address-expression-1, absolute-expression-2

Specifies the starting-address **address-expression-1** and the size **absolute-expression-2** of a base area for out-of-range indirect address links.

Normally, the loader generates address links starting at location '200 of sector zero. This statement permits the loader to generate address links in the same sector as the instruction which refers to them. Memory locations to be used for this purpose must be reserved by the program.

The first SETB for a given base area determines the location at which the indirect word table will begin in that sector. The table then grows upward (increasing addresses). Other SETB

pseudo-operations referencing the same sector do not redefine the table for that sector — table filling resumes where it left off.

At the end of each module, the base sector reverts to sector zero. The loader maintains a record of the last location used in each base sector. When the base sector reverts to zero, no indirect words are lost.

MACRO DEFINITION PSEUDO-OPERATIONS (MD)

Create macros. See Conditional Assembly and Listing Control Pseudo-Operations for other MACRO specific pseudo-operations.

Name	Description
ENDM	End macro definition
MAC	Begin macro definition
SAY	Print message
SCT	Select code within macro
SCTL	Select code from list within macro

▶ ENDM

Terminates assembly of a macro definition. ENDM must be the last statement in a macro definition — just as END is the last statement in an assembler program.

▶ label MAC { dummy-words, . . . | argument-values, . . . | argument-identifiers, . . . }

Begins the definition of the macro named by the label field. The name is formed in the same way as any variable or label. Following MAC are statements that make up the **macro definition**; for example:

```
TRANSFER MAC
         LDA   <1>
         STA   <2>
```

The integers enclosed in angle brackets are **argument references**. During assembly they are replaced by **argument values** specified in a **macro call**. Optional **dummy words** ("noise words") and argument identifiers ("positional noise words") are described in Section 17 — Macro Facility.

Macro definitions may contain macro calls to any depth, but macro definitions themselves cannot be nested.

▶ [label] SAY ASCII-expression

Defines a message which is printed starting in columnm 1 of the listing. Normally, the SAY message is used within a macro to generate error comments or other messages. Macro argument references, enclosed by angle brackets are replaced by their equivalent character string before output.

If a listing device is assigned, SAY statements generate output regardless of the status of the listing options.

▶ [label] SCT absolute-expression

Assembles selected code groups based on **absolute-expression**. The expression must be a constant or an expression that can be evaluated as a single-precision number. The argument value may be positive or negative, with a range to ± 4000. This value determines which code groups are assembled.

Code groups: Code using SCT must be in groups delimited by marker lines, which have a percent symbol (%) in column 1 followed by a numeric argument:

Marker	Meaning
%	Ordinary marker line.
%1	If the preceding section of code was assembled, continue assembly from this marker to the next marker.
%2	If no other sections of code have been assembled, assemble from this line to the next marker.
%/	End of control range.

The %2 marker is useful to identifiy a section of code that is to be assembled if the argument value of the SCT statement is out of range.

Argument Value	Assembly Condition
0	Assemble from the SCT statement to the first % marker; then skip to the %/ line.
1	Skip to the first % marker; assemble from there to the second % marker; then skip to the %/ marker.
n	Skip to the n'th % line marker, if any; assemble from there to marker n +1; then skip to the %/ marker. If there is no n'th % marker, proceed as for -n.
-n	Skip to a %2 line marker, if any, and assemble from there to the next % marker; then skip to the %/ line. If there is no %2 marker, skip to the % line.

No other SCT statements may appear within the control range; SCT statements *cannot* be nested. It is possible, however, to call another macro containing an SCT from within an SCT area.

▶ [label] SCTL absolute-expression-1, absolute-expression-2, . . .

Assembles selected code groups. The results of a comparison between the first **absolute-expression-1** and the rest of the argument list controls the selection of the code group. Absolute-expression-1 and each expression in the argument list must be a constant or an expression that can be evaluated as a single-precision number. The argument value may be positive or negative, with a range to ± 4000. This value determines which code groups are assembled.

Code groups: Code using SCTL must be in groups delimited by marker lines, which have a percent symbol (%) in column 1 followed by a numeric argument:

Marker	Meaning
%	Ordinary marker line.
%1	If the preceding section of code was assembled, continue assembly from this marker to the next marker.
%2	If no other sections of code have been assembled, assemble from this line to the next marker.
%/	End of control range.

The %2 marker is useful to identify a section of code that is to be assembled if the argument value of the SCTL statement is out of range.

Argument Value	Assembly Condition
0	Assemble from the SCTL statement to the first % marker; then skip to the %/ line.
1	Skip to the first % marker; assemble from there to the second % marker; then skip to the %/ marker.
n	Skip to the n'th % line marker, if any; assemble from there to marker n +1; then skip to the %/ marker. If there is no n'th % marker, proceed as for -n.
-n	Skip to a %2 line marker, if any, and assemble from there to the next % marker; then skip to the %/ line. If there is no %2 marker, skip to the % line.

Expression comparison: The position on the argument list of the expression which equals absolute-expression-1 determines the result:

Expression Position	Meaning
absolute-expression-1=absolute-expression-2	SCT 0
absolute-expression-1=absolute-expression-n	SCT n
no match	SCT -n

No other SCTL statements may appear within the control range; SCTL statements cannot be nested. It is possible, however, to call another macro containing an SCTL from within an SCTL area.

PROGRAM LINKING PSEUDO-OPERATIONS (PL)

Coordinate the interaction of the assembler and loader in resolving address references between main programs and external subroutines.

Name	Function	Restriction
CALL	External subroutine reference	
DYNT	Direct entry call	
ECB	Define entry control block	*SEG/SEGR*
EXT	Flag external reference	
SUBR,		
ENT	Define entry point	

▶ [label] CALL symbol

In non-64V modes, CALL generates a JST to **symbol**, which is defined by the assembler as

external. Unlike EXT, there is no conflict between a local variable and a CALL operand with the same symbol.

The operand must contain a single symbol (not an expression) of up to 6 characters, of which the first must be alphabetic. A ,1 for indexing and * for indirect addressing is optional.

In 64V and 32I modes, CALL generates a PCL instruction to an external symbol.

▶ DYNT address-expression

Defines a direct entry point into the operating system. System libraries only.

▶ [label] ECB entry-point, [link base], displacement, n-arguments [, stack-size] [, keys]

Generates an entry control block to define a procedure entry. It must go in the linkage frame with the subroutine entry point pointing to the ECB.

Parameter	Meaning
entry-point	Procedure relative value; entry point for subroutine.
link-base	Link base register value.
displacement	Displacement in stack frame for argument list. May be stack relative or absolute expression.
n-arguments	Number of arguments expected; default is zero.
stack-size	Initial stack frame size. Default is maximum area specified in DYNM statements.
keys	CPU keys for procedure. Default is 64V addressing mode, all other keys zero.

For example:

```
         ENT   ECBNAM
LAB1     LDA   ='123
         LINK
ECBNAM   ECB   LAB1
         END
```

If the default value for a parameter is desired, the parameter may be omitted, leaving only its delimiting comma. Any string of trailing commas may be omitted.

Note

The main program — that which you call PRIMOS level using SEG#Name — is a subroutine to SEG and must, therefore, have an ECB and the ECB name on the END statement. It need not have an ENT because SEG will give a dummy entry point name to a routine called at this level.

▶ [label] EXT symbol

Identifies external variables. The names appearing in the operand of this statement are flagged as external references. Whenever other statements in the main program reference one of these names, a special block of object text is generated that notifies the loader to fill in the address properly. The assembler fills the address fields with zeroes.

Names defined by the EXT pseudo-operations are unique only in the first 6 characters (loader restriction) and should not appear in a label field internal to the program.

▶ **[label] SUBR symbol-1** *or* **ENT symbol-1 [, symbol-2]**

Link subroutine entry points to external names used in CALL, XAC or EXT statements in calling programs. SUBR and ENT are identical in effect. **symbol-1** is the external name used by calling program, whereas, **symbol-2** is the entry name used in subroutine, if different from symbol-1.

▶ **SYML** *SEG/SEGR*

Allows long external names up to eight characters to be generated by the assembler. Must follow SEG or SEGR but precede any generated code.

STORAGE ALLOCATION PSEUDO-OPERATIONS (SA)

Control the allocation of storage within the program address space.

Name	Function	Restriction
BSS	Allocate block starting with symbol	
BES	Allocate block ending with symbol	
BSZ	Allocate block set to zeroes	
COMM	FORTRAN compatible COMMON	

▶ **[label] {BSS | BES | BSZ} absolute-expression**

Allocates a block of words of the size specified in the **absolute-expression** starting at the current location count. If there is a label, it is assigned to the first word of the block (BSS and BSZ) or to the last word of the block (BES). For BSZ, all words within the block are set to zeroes.

▶ **[label] COMM symbol [(absolute-expression)]**

Defines FORTRAN-compatible named COMMON areas. These areas are allocated by the loader. The **label** assigns a name to the block as a whole, and the operand field specifies named variables or arrays within the block. Additional COMM statements with the same block name are treated as continuations. **symbol** alone reserves a single location; the optional **absolute-expression** reserves locations equal to its value. In SEG mode, the loader sets up a 32-bit indirect pointer in the linkage segment which points to the common area.

SYMBOL DEFINING PSEUDO-OPERATIONS (SD)

Variables used as address symbols are usually defined when they appear in the label field of an instruction or pseudo-operation statement. Symbols so defined are given the relocation mode and value of the program counter at that location. The EQU, SET and XSET statements make it possible to equate symbols to any numerical value, including ones that lie outside the range of addresses in a program.

Name	Function	Restrictions
DYNM	Declare stack relative	*R and V only*
EQU	Symbol definition	
SET	Symbol definition	
XSET	Symbol definition	

▶ **DYNM [absolute-expression-1,] [symbol] [(absolute-expression-2)] [=absolute-expression-3], . . .]**

Declares stack relative symbols. Since references to stack relative symbols generate two-word instructions, stack relative symbols must be declared before they are used (*REL mode only*).

expression-1 is the stack size, **symbol** is the name, **expression-2** is the number of words to allocate for that symbol and **expression-3** is the stack offset.

In the following format descriptions, these abbreviations are used:

sc Current stack allocation count (initially=2)

Note

Initially '12 in SEG mode. Note also that subroutine argument addresses require three words and are specified: ARGn(3).

sm	Maximum allocation count
symbol	Symbol to be assigned stack relative offset
exp1	Expression defining number of words for symbol
exp2	Expression defining stack offset

Format Description

1. symbol
   - symbol is assigned offset=sc
   - sc=sc+1
   - if (sc .GT. sm) sm=sc
2. symbol (exp1)
   - symbol is assigned offset=sc
   - sc=sc+exp1
   - if (sc .GT. sm) sm=sc
3. symbol=exp2
   - symbol is assigned offset=exp2
   - if (exp2+1 .GT. sm) sm=sc
4. symbol (exp1)=exp2
   - symbol is assigned offset=exp2
   - if (exp2+exp1 .GT. sm) sm=exp2+exp1
5. = exp2
   - sc=exp2

▶ **symbol { EQU | SET | XSET } absolute-expression [,symbol=absolute-expression]** *Format 1*

▶ **{ EQU | SET | XSET } symbol=absolute-expression, . . .** *Format 2*

In *format 1*, the **symbol** in the field label is equated to the **absolute-expression**, which may be any expression which is legal in the current addressing mode. Any symbols used in the expression must already be defined.

In *format 2*, symbols are assigned numerical values by equality expressions in the operand field. One or more equality expressions can be used, separated by commas.

Note that format 1 can be extended by **symbol=value** expressions.

EQU, SET and XSET perform the same functions; however, a symbol defined by EQU may *not* be redefined, while a symbol once defined by SET or XSET may be redefined by subsequent SET or XSET statements without causing an error message.

Symbols defined by XSET will not appear in the cross reference listing.

17
Macro facility

INTRODUCTION

The macro facility enables the programmer to define functions using simple English, or other language phrases. For example, the macro:

```
TRANSFER DATA TO SAVE
```

replaces the simple but cryptic assembly coding:

```
LDA   DATA
STA   SAVE
```

Once a macro function has been defined, it can be called any number of times within a program. Different argument values (DATA and SAVE in the above example) can be supplied with each call. Dummy words, such as TO or FROM, can be added to increase intelligibility. These words must be identified during macro definition so that they will not be treated as additional arguments in a macro call.

After a system-level programmer has defined a set of macros, a specialist in an application field can formulate macro calls to solve his particular problems. The application specialist gains the advantage of macro's capabilities without becoming involved in the details of assembly language programming.

The example below illustrates a simple macro definition and call. The discussion which follows describes each element.

```
          REL
                              (0001)            REL
                              (0002)            LSTM
                              (0003) TRANSFER   MAC
                              (0004)            LDA   <1>
                              (0005)            STA   <2>
                              (0006)            ENDM
                              (0007)            TRANSFER DATA SAVE
000000:      02.000002        (ML01)            LDA   DATA
000001:      04.000003        (ML01)            STA   SAVE
                              (ML01)            ENDM
000002:         000005        (0008) DATA       OCT   5
000003:                       (0009) SAVE       BSS   2
          000005              (0010)            END
```

MACRO DEFINITION

Each macro definition must begin with a MAC pseudo-operation. The MAC statement must have a label ('TRANSFER') and may have optional dummy words ('FROM' 'TO') and argument identifiers in the variable field. Statements which make up the macro definition follow, terminated by an ENDM pseudo-operation.

Argument references

Argument references are expressions enclosed within angle brackets. Any field of a statement within a macro definition may contain an argument reference. The expression may contain variables as well as absolute integers, provided the variable has been previously defined as an absolute integer. For example:

```
         REL
         LSTM
TRANSFER MAC
         LDA  <J>
         STA  <K>
         ENDM
J        EQU  1
K        EQU  2
         ENDM
```

is the same as the previous transfer macro example. The label field of the macro call is not automatically assigned; it replaces argument <0>, if any, during assembly.

Assembler attribute references

Certain useful attributes of a macro can be specified by a number preceded by the pound character (#). The attribute number may be a variable, or an expression within parentheses, as long as such variables have been previously defined as absolute integer values. Attribute references are evaluated as absolute integer values, and may be used in both macro definitions and macro calls. See Appendix A for a complete list.

Local labels within macros

Local labels, which do not conflict with labels outside of the macro, can be assigned within a macro definition by using the ampersand character (&) as the first character of the label. The ampersand is replaced by a 4-digit macro call number, thereby assuring uniqueness of the label regardless of the macro's environment. Use of the ampersand outside of a macro will result in the substitution of 4 zeros.

Examples:

Assigned Local Label	Evaluation As	In Macro Call
ABC	002ABC	0002
X3A	1739X3A	1739

MACRO CALLS

A macro call is a special type of statement that uses the name of a defined macro in the operation field:

[label] macro-name { **arguments, . . .** / **dummy words, . . .** / **argument identifiers, . . .** }

For each macro call, the assembler enters the in-line code of the defined macro starting at the current location. Argument references are replaced by argument values from the variable field.

User defined macros must be defined in source statements preceding the macro call.

Argument values

The variable field of a macro call usually contains one or more argument value expressions. An argument value expression begins with the first non-space character of the variable field and continues until either a terminating comma or space appears. The comma or space is not considered to be part of the argument expression.

Argument values in parentheses: Enclose argument value expressions in parentheses when commas, spaces, or string delimiters within a single argument are desired. The outside parentheses are not considered as part of the argument expression. A typical use of this feature is in forming sub-lists of arguments for macro calls nested within a given macro definition. See NESTING MACROS, below.

Argument substitution

During assembly of a macro call, the assembler substitutes the argument values in the macro call variable field for the argument references in the macro definition. Argument expressions are matched to argument references in numerical order from left to right. The first expression in the macro call is assigned as argument 1, the second as argument 2, and so on.

Variable Field	Argument <1>	Argument <2>	Argument <3>
A	A	0	0
A+3	A+3	0	0
X,Y-1,Z*A-1	X	Y-1	Z*A-1
X,B-C (Z3X2)	X	B-C	Z3X2
(A, B-1), C	A, B-1	C	
(X, Y, (Z1+Z2),3)	X,Y,(Z1+Z2),3	0	0

In the following call to the TRANSFER macro,

```
TRANSFER ARG1 '177Ø
```

the variable ARG1 is argument 1 and the constant '1770 is argument 2. Thus, the TRANSFER macro shown would be assembled as:

```
LDA   ARG1
STA   '177Ø
```

Arguments that are not assigned values in a macro call are set to zero by the assembler.

Self documentation of macros

An ordinary macro call like:

```
TRANSFER ARG1 ARG2
```

although complete, provides only a vague description of its function. Using additional words in the variable field of a macro call, the programmer can communicate the exact nature of the function. Macro calls are made self-documenting by a combination of meaningful argument symbols, such as DATA, MESSAGE, and PRINTER, dummy words, such as TO and FROM, and argument identifiers. Dummy words are for descriptive purposes only and are ignored by the assembler, while argument identifiers act as argument keywords.

Dummy words: Dummy words applicable to a given macro are defined in the variable field of the MAC statement that starts the macro definition. For example:

```
TRANSFER MAC  FROM TO
         LDA  <1>
         STA  <2>
         ENDM
```

In the above example, FROM and TO are defined as dummy words. In any subsequent call to this macro, the assembler ignores the words FROM and TO; all other expressions in the variable field are interpreted as argument values, proceeding in numerical argument order from left to right. These values are substituted for the argument references in the macro definition statements, e.g., when the TRANSFER macro is called by TRANSFER FROM ALPHA TO '7770, the assembler ignores the FROM and TO, and assembles the macro as if the call statement were TRANSFER ALPHA, '7770.

A dummy word string can be any combination and number of letters, numerals, periods and $ signs, terminated by a comma. Any number of dummy word strings may be used. If the first character of a dummy word string is an open parentheses, all characters, including spaces and commas, up to the closing parentheses are considered part of the same string. The parentheses are not considered part of the string.

Argument identifiers: While the self-documenting effect of dummy words improves the description of macro calls, the programmer must still be careful to enter values for arguments in the proper order. Argument identifiers increase the format flexibility of macro calls by associating a particular argument number with a specific dummy word, regardless of written order. In the TRANSFER macro, for example, identifiers can be defined so that argument 2 follows the dummy word TO, and argument 1 follows FROM, regardless of the order in which TO and FROM appear in a macro call.

Argument identifiers, like dummy words, are assigned in the variable field of a MAC statement that starts a macro definition. To define an argument identifier, set a dummy word, in parentheses, equal to the desired argument number:

```
TRANSFER MAC  (FROM)=1  (TO)=2
         LDA  <1>
         STA  <2>
         ENDM
```

When a call to the macro uses a defined argument identifier in its variable field, the first non-dummy expression immediately following the identifier is taken as the value of the argument:

```
              REL
                           (0001)                REL
                           (0002)  TRANSFER MAC  (FROM)=1, (TO)=2
                           (0003)           LDA  <1>
                           (0004)           STA  <2>
                           (0005)           ENDM
                           (0006)           TRANSFER FROM ALPHA TO BETA
000000:        02.000004   (ML01)           LDA  ALPHA
000001:        04.000006   (ML01)           STA  BETA
                           (0007)           TRANSFER TO BETA FROM ALPHA

000002:        02.000004   (ML01)           LDA  ALPHA
000003:        04.000006   (ML01)           STA  BETA
000004:                    (0008)  ALPHA    BSS  2
000006:                    (0009)  BETA     BSS  2
            000010         (0010)           END
```

Both of these calls have the same effect. The expression following the dummy word FROM is taken as argument <1>, and the expression following TO is taken as argument <2>. Argument identifiers and dummy words may be used together in the same macro. Ordinary dummy words are ignored, as usual.

Arguments that are not associated with identifier words receive values in the usual positional priority - the first non-dummy word is taken as the value for the first unspecified argument, and so on. For example, the macro defined by:

```
MASK      MAC  (BY)=2,(TO)=3,MOVE,AND
          LDA  <1>
          ANA  <2>
          LDA  <3>
          ENDM
```

can be called by,

```
                           (0001)              REL
                           (0002) MASK         MAC  (BY)=2,(TO)=3,MOVE,AND
                           (0003)              LDA  <1>
                           (0004)              ANA  <2>
                           (0005)              LDA  <3>
                           (0006)              ENDM
                           (0007)              MASK INPUT BY 7 AND MOVE TO BUFFER
000000:     02.000003      (ML01)              LDA  INPUT
000001:     03.000007A     (ML01)              ANA  7
000002:     02.000004      (ML01)              LDA  BUFFER
000003:        000456      (0008) INPUT        OCT  456
000004:                    (0009) BUFFER       BSS  1
          000005           (0010)              END
```

Using the identifier words BY and TO, argument 2 is given a value of 7 and argument 3 is equated to the label BUFF1. The only remaining variable in the call is INPUT, so it is assigned as to the first unspecified argument, 1.

NESTING MACROS

Macro definitions may contain nested calls, as in the following example:

The WAIT1 macro, which calls another macro, TRANSFER, is defined by:

```
          REL
TRANSFER  MAC
          LDA  <1>
          STA  <2>
          ENDM
WAIT      MAC
          IRS  <1>
          JMP  *-1
          TRANSFER <2>




          ENDM
```

is called by,

```
         REL
TRANSFER MAC
         LDA  <1>
         STA  <2>
         ENDM
WAIT     MAC
         IRS  <1>
         JMP  *-1
         TRANSFER <2>
         ENDM
         WAIT 1ØØ,(INPUT,SAVE)
INPUT    BSS  2
SAVE     BSS  2
         END
```

is assembled as:

```
WAIT 1ØØ,(INPUT,SAVE)
```

Macro definitions may *not*, however, contain nested macro definitions.

CONDITIONAL ASSEMBLY

There are a number of pseudo-operations which allow the programmer to control the assembly of his macro. These pseudo-operations are discussed in the Conditional Assembly Pseudo-operation section.

MACRO LISTING

Three levels of listing detail for macro calls are available.

LSTM	Lists macro statements and all lines generated by expansion of the macro, including code or data values.
LSTMD	Lists macro call statements and any lines which generate code.
NLSM	Inhibits the list of macro expansions. Only the call is listed.

The default condition is NLSM, which causes only the macro call statement to be listed, with no expansion. These pseudo-operations remain in effect until a new macro listing control pseudo-operation is specified.

Table 21-3. Key Values: V MODE

C	D	L	ADR	F	X	N	Z	0
1	2	3	4 — 6	7	8	9	10	11 — 16

Bit	Name	Meaning
1	C	Carry Bit
2	D	Precision; 0=Single; 1=Double
3	L	Carry out of most significant bit
4–6	ADR	Addressing Mode 00=16S 01=32S 011=32R 010=64R 110=64V
7	F	0=Floating point exception faults
8	X	1=Integer exception faults
9	N	Negative result
10	Z	Zero result
11–16		Must be zero

▶ PROCEED [address] [a-reg] [b-reg] [x-reg] [keys]

Continue execution from breakpoint. Removes the current breakpoint if there is one, optionally sets a new breakpoint at **address**, and does a RUN command to the current program counter address. **a-reg**, **b-reg**, **x-reg** and **keys** have the same meaning as in the RUN command.

▶ QUIT

Returns to the PRIMOS operating system. In SEG's VPSD returns to SEG command level.

▶ RELOCATE value

Sets a new **value** for the access-mode relocation counter.

▶ RUN [start-add] [a-reg] [b-reg] [x-reg] [keys]

Runs the executable program starting at **start-add** location. Prior to program entry, **a-reg, b-reg, x-reg**, and **keys** are optionally loaded. Control does not return to the debugging utility unless a breakpoint is encountered.

In VPSD, use SN to specify the segment in which to run; start-add is just the 16-bit word number.

▶ SB seg-no word-no

Loads the stack base register with a segment number (**seg-no**) and a word number (**word-no**).

▶ SEARCH block-start block-end match-word [mask]

Searches memory from **block-start** to **block-end** for words equal to **match-word** under an optional 16-bit **mask**.

5 DEBUGGING UTILITIES

18
Introduction to TAP, PSD and VPSD

Prime supplies three interactive debugging programs:

- TAP (Trace And Patch) - for sectored addressing modes
- PSD (Prime Symbolic Debugger) - for sectored and relative addressing modes
- VPSD (Virtual Symbolic Debugger) - for sectored, relative and virtual addressing modes.

TAP is a small (one sector), octal format routine that examines, dumps and patches user programs. It has a breakpoint capability and can trace 16S and 32S instruction execution.

PSD is a symbolic routine that can handle all of the PRIME-300 addressing modes. In addition to the functions provided by TAP (except EXECUTE and PATCH) it has enhanced functionality and additional input/output formats.

VPSD is a symbolic routine that can handle the segmented addressing modes, as well as all of the PRIME-300 addressing modes. The functionality (except for instruction tracing) is essentially the same as PSD.

Table 18-1 gives a complete alphabetical listing of all debugging commands and the programs which use them.

USING TAP

Load the object program, using the PRIMOS commands LOAD or RESTORE, and then enter the command TAP. Since TAP is in user memory along with your program, be sure TAP has not overlaid part of your program. When ready, TAP prints the $ prompt character and waits for you to type in TAP command strings.

Terminating long operations: To terminate long operations such as DUMP, type CTRL P to return to PRIMOS command level.

Restarting: Restart at 'XX000 where XX is the sector occupied by TAP. To determine this value, RESTORE TAP and do a PM command to print the starting location (see Section 7 for a complete description of the PM command).

Table 18-1. Debugging Command Summary (*input rust colored letters in upper-case only*).

Command	Meaning	TAP	PSD	VPSD
ACCESS	Access and print or alter contents of memory word	YES	YES	YES
BREAKPOINT	Insert up to 10 breakpoints in program (TAP permits only 1 breakpoint)	YES	YES	YES
BR	Display contents of base registers	NO	NO	YES
COPY	Copy block to block	YES	YES	YES
DEFINE	Define local symbols	NO	YES	NO
DUMP	Print contents of block (or, in PSD and VPSD, write contents to optional file)	YES	YES	YES
EFFECTIVE	Search for effective address under mask	NO	YES	YES
EXECUTE	Execute a subroutine	YES	NO	NO
EXECUTE	Execute segmented program	NO	NO	YES
FILL	Fill block with constant	YES	YES	YES
GO	Proceed from breakpoint	NO	YES	NO
JUMPTRACE	Trace JMP, JST, HLT instructions	YES	YES	NO
KEYS	Update CPU status	NO	YES	YES
LB	Load link base register	NO	NO	YES
LIST	Print contents of address	YES	YES	YES
LS	Load external symbols from map file and enter symbolic address mode	NO	YES	NO
MAP	Type out all symbols with their values	NO	YES	NO
MODE	Address mode selection	NO	YES	YES
MONITOR	Execute program, reporting any reference to the specified effective address	YES	YES	NO
NOT-EQUAL	Not-equal search for constant under mask	YES	YES	YES
OPEN	Open file for memory dump or symbols	NO	YES	YES
PATCH	Patch object program	YES	NO	NO
PRINT	Print parameters	NO	YES	YES
PROCEED	Remove breakpoint and proceed	NO	YES	YES
QUIT	Return to PRIMOS (or SEG if SEG's VPSD)	NO	YES	YES
RELOCATE	Alter relocation constant	NO	YES	YES
RUN	Run object program	YES	YES	YES
SB	Load stack base register	NO	NO	YES
SEARCH	Search memory block for constant under mask	YES	YES	YES
SN	Set segment number	NO	NO	YES
SYMBOL	Enable/disable use of symbols in address typeout	NO	YES	NO
TRACE	Trace object program	YES	YES	NO
UPDATE	Update memory word	YES	YES	YES
VERIFY	Compare contents of one block of memory with another	YES	YES	YES
VERSION	Print PSD version and restart location	NO	YES	YES
WHERE	List location and remaining repeat counts for all breakpoints	NO	YES	YES
XB	Load temporary base register	NO	NO	YES
XR	Load X register	NO	YES	YES
YR	Load Y register	NO	NO	YES
ZERO	Zero breakpoint location	NO	YES	YES

USING PSD

Load the object program, using the PRIMOS commands LOAD or RESTORE, and then decide which of the three versions of PSD you need to use. Since PSD is resident in user memory with your program, you must take precautions to prevent your program from being overlaid. See Table 18-2 for the name, starting location and suggested usage of each version. None of the three versions is relocatable. When ready, PSD prints the $ prompt character and waits for you to type in PSD command strings.

Terminating long operations: To terminate long operations such as DUMP, type CTRL P to return to PRIMOS command level.

Restarting: Restart at 'XXXXX where 'XXXXX is PSD's starting address. To determine this value, type a VERSION command to print the starting location.

Table 18-2. PSD/VPSD Versions

Name	Location	Comments
PSD20	'20000	Use when your system is small, i.e., 16k
HPSD	'150000	Use when your program is so large that it overlays PSD
PSD	'60000	Normal use
VPSD	'60000 of segment '4000	Normal use
VPSD16	'160000	Use when your program is so large that it overlays VPSD

USING VPSD

There are two versions of VPSD: stand-alone VPSD and SEG's VPSD. Both reside in segment '4000. There are three ways to enter VPSD, each of which has slightly different consequences for debugging:

Action	Usage/Consequence
1. Load the object file using SEG's loader. Then return to "#" level with the "RE" command and issue the SEG command PSD. Obtain the starting address of SEG's VPSD with the VERSION command. Memory may now be examined and breakpoints set. Type "EX" to start the program. If it crashes, issue the PRIMOS level PM command to obtain the data at crash time. Then issue the PRIMOS command START using SEG's VPSD starting address.	Used when no runfile exists. When EXECUTE is given, the registers are as SEG initialized them. Preserves the entire program contents exactly as it was at the time of the crash, except for the program counter whose value you obtain via the PM command.
2. Load the runfile and enter VPSD via the "SEG filename 1/1" command.	Used when runfile exists. When EXECUTE is given, the registers are as SEG initialized them.
3. Load and execute the runfile using SEG. When the program crashes, use the PRIMOS command VPSD to call the stand-alone version of VPSD.	Use only if SEG's VPSD has been destroyed. The registers are not preserved.

COMMAND LINE FORMAT

Each command is a one or two letter operation followed by one or more operands. Separators may be spaces or commas, and values may be omitted by including extra commas. Commands may be terminated by a carriage return or a semicolon.

The ACCESS command differs from the others in that it remains in control and allows you to examine and/or alter more than one location without returning to command mode (signalled by the prompt character). The next location to be accessed is selected by the terminator used. (See ACCESS for details.)

A question mark (?) may be used to abort a command string and return to command mode.

If more than five octal digits are entered, only the last 16 bits are used.

In TAP, if the wrong function code letter is entered, simply follow it with the correct character. (Only the last input letter of the command field is interpreted.) To cancel an incorrect parameter, type an asterisk (*).

Effective address formation (PSD and VPSD only)

PSD processes input and output in all Prime-300 addressing modes; VPSD, in all Prime 350 and up addressing modes the mode is set by the MODE command.

When the index register is needed, the current value of the X register is used; VPSD may use the Y register where appropriate and so specified.

When either VPSD or PSD prints an address, it applies the same address formation process as the hardware, using the current values of the registers. For relative addresses, the access-mode current location counter is used as the value of the P register.

Relocation constant (PSD and VPSD only)

PSD can process addresses in a relocatable mode (equivalent to assembler REL) by maintaining a relocation constant which points to the start of a module. All addresses that are preceded by > are relative to this relocation constant. For a relocation constant of '3121, both $A>0 and $A '3121 would open location '3121.

The relocation constant is set by the RELOCATE command. Setting the relocation constant to 0 disables this mode.

For all output, any address which is larger than the relocation address is printed as > n, where n is the address minus the relocation address. Setting relocation constant=0 disables symbolic I/O as does SY 0.

Input/output formats (PSD and VPSD only)

While the default command line scan is octal, PSD and VPSD can accept input parameters and print output values in several different formats. The format is established by typing a colon followed by a single format letter. All input to the right of that format specifier is interpreted in that format until you type a new format specifier or a terminator. Format specifiers control the input for just the current line but have a global effect on output until you type a new format specifier. Table 18-3 describes the format specifiers. The following example illustrates their effects.

Fill and Dump Example:

`F 100 200 :HAFAF`	Fills octal locations 100 to 200 with hexadecimal digits AFAF
`D 120 130`	The typeout on the terminal will be in hexadecimal.

Symbolic instruction format: enables user to use standard PMA symbolic instruction format for output and access mode input. The only restrictions are:

1. Expressions - only + and - operations
2. No literals
3. Symbol use - global symbols if the LS procedure has been used and any symbols defned within PSD
4. Input is only legal in access mode, i.e., "S 100 200 :SAIA" is not legal
5. The suffixes "+ 1C" and "+ nB" may be used to indicate character and bit offsets in VPSD

Table 18-3. Input/Output Formats (PSD and VPSD)

Format	Code	Input	Output
ASCII	**:A**	Two characters accepted first may not be:>= @ % , .NL / ? + - : * () or blank Second is required and may not be: / ? .NL. Note—to input ASCII characters in any format use 'CC (single quote followed by two characters)	Two characters are printed. An @ is substituted for a nonprinting character
Binary	**:B**	Takes a sequence of 16 1's and 0's	Prints a sequence of sixteen 1's and 0's
Decimal	**:D**	Accepts up to five decimal (0-9) digits	Prints decimal digits
Hexadecimal	**:H**	Accepts up to four hexadecimal (o-9, A, B, D, C, E, F) digits	Prints hexadecimal digits
Octal	**:O**	Accepts up to six octal (o-7) digits	Prints octal digits
Symbolic	**:S**	Symbolic instructions (See below)	Symbolic instructions
AP *	**:P**	Symbolic instructions	Prints address pointers
Long *	**:L**	Accepts 32 bit octal integers	Prints 32 bit octal integers

*-AP and Long are VPSD only.
Constants entered in :S mode are octal

Command line operands

These may be constants, constant expressions, or symbols. The format of a constant is:

[: format] [>] { **±digits** | **ASCII-constant** } **[:format]**

where:

format	format specifier (see Table 18-3)
>	relocatable mode
ASCII-constant	two letter constant in format described in Table 18-3
digit	decimal, octal, binary or hexadecimal, depending on which format is in control

The format of a constant expression is:

constant [± constant]

Current location pointer (PSD and VPSD)

In access mode, a current location pointer is maintained, starting with the value of the start-address parameter of the ACCESS command. The location pointer determines the next location to be accessed.

During each access operation, PSD replaces the value in the open location with the new value (if specified) and uses the line terminator to compute the next value of the current location pointer. For the comma or CR line terminators, the pointer is incremented after each access. Other line terminators provide different options.

19
TAP command summary

TAP COMMAND SUMMARY

Enter rust colored letters in upper-case only.

▶ ACCESS address

Accesses a word in memory. The debugging program types the address and its contents and then waits for keyboard input in the following form: **[value] terminator**, where **value** is an octal number which replaces the contents of the accessed location, and **terminator** is one of the characters shown in Table 19-1.

Table 19-1. TAP Terminators

Terminator	Function
CR	Alters contents of current location (if a value is given), moves to current location +1, and prints its contents.
^ or ↑	Alters contents of current location (if a value is given), moves to current location -1, and prints its contents.
/ or ?	Exits from access mode. Does not close current location.

▶ BREAKPOINT location

Sets a breakpoint at the specified **location**. If the program is later executed and control reaches the breakpoint location, the debugging program prints CPU status and awaits further commands. One breakpoint is permitted. The actual breakpoint jump is placed in the object program only at execution time, and is removed after each use, however, the breakpoint address is retained for reuse. To remove the breakpoint completely, key in B'17(CR).

▶ COPY source-start source-end target

Copies a block of memory from **source-start** to **source-end** into a new block of memory starting at **target**. If the target location lies between source start and source end, the non-overlapped portion is propagated through the target area. The size of the target area is always equal to the size of the source. If source-end equals source-start, the contents of location source-start will be copied into target.

▶ DUMP block-start block-end

Prints the contents of the block of memory at locations **block-start** through **block-end** on the user terminal. The output format is eight octal words per line, preceded by the octal address of the first word on the line. Repetitious words and lines are suppressed as follows:

1. If the remainder of the current line is identical to the word last printed, the line is terminated.
2. If one or more subsequent lines are identical to the word last printed, the terminal skips one line.

▶ EXECUTE sub-name [a-reg] [b-reg] [x-reg] [keys]

Executes a subroutine by branching to location **sub-name**. The A, B, and X registers and the keys (see Table 19-2) may be changed prior to executing the subroutine. The subroutine return should be via an indirect jump through its entry point, incremented by 0, 1 or 2, depending on the number of arguments, if any.

Upon return from the subroutine, the 'TRACE' program prints the register contents as noted under RUN except that one or two meaningless words may precede the specified format to indicate that the subroutine has incremented its return link by 1 or 2.

▶ FILL block-start block-end constant

Fills the memory block from **block-start** to **block-end** with the specified octal **constant**.

If block-end does not exceed block-start, only the first location is filled. FILL is useful to test data area usage by pre-filling it with a visual pattern.

▶ JUMPTRACE [start-add] [a-reg] [b-reg]

Traces the execution of the object program starting with **start-add** (default is current location). You may set the A register and B registers. JUMPTRACE, which is very useful for control-flow tracing, produces a diagnostic printout in the following format prior to the execution of any JMP, JST or HLT instruction:

Location: instruction A= B= X= K= R=

Any typed character will stop the trace. SVC's are not included in the trace.

▶ LIST address

Prints the contents of **address**. Unlike ACCESS, LIST does not transfer the pointer to that location, a useful feature when you wish to examine a location without going there.

▶ MONITOR [start-add] [a-reg] [b-reg] address

Traces the object program from **start-add** (default is current location) to **address**. You may set the A and B registers.

If a memory reference instruction whose effective address equals address is encountered, data in trace format is printed on the terminal prior to the execution of that instruction. MONITOR answers the question "Where is the address being clobbered?"

Table 19-2. Keys

C	D	—	ADR	—	SHIFT COUNT
1	2	3 4	5 — 6	7 — 8	9 — 16

Bit	Name	Meaning
1	C	Carry Bit
2	D	Precision; 0=Single; 1=Double
3	-	Not used
4	-	Not used
5-6	ADR	Addressing Mode 00=16S 01=32S 11=32R 10=64R
7	-	Not used
8	-	Not used
9-16	SHIFT COUNT	Shift count - low order 8 bits of the floating point accumulator exponent register

Typing any character will stop the trace after several more lines. The character typed is considered part of the next command, so a space is the usual choice.

▶ NOT-EQUAL block-start block-end n-match [mask]

Searches memory between **block-start** and **block-end** for words *not* equal to **n-match** under an optional 16-bit **mask**.

The masking function is a 16-bit logical AND. If no mask is specified, the entire word is tested. When a non-match is found, the address and its contents are typed out and the search continues to block-end.

▶ PATCH patch-loc branch-loc

Inserts a patch in the object program. The instruction at **branch-loc** is replaced by a jump to **patch-loc**. The previous branch-loc instruction is inserted at patch-loc and the ACCESS sub-processor is entered with the current location set to patch-loc. You may now enter the patch, including a suitable return. Patch-loc must either be in the same sector as branch-loc or in Sector 0.

▶ RUN [start-add] [a-reg] [b-reg] [x-reg] [keys]

Runs the executable program starting at **start-add** location.

Prior to program entry, **a-reg**, **b-reg**, **x-reg** and **keys** are optionally loaded. Control does not return to the debugging utility unless a breakpoint is encountered.

▶ SEARCH block-start block-end match-word [mask]

Searches memory from **block-start** to **block-end** for words equal to **match-word** under an optional 16-bit **mask**.

If a mask is not specified, the entire word is tested. When a match is found, the address and its contents are typed out, and the search continues until location block-end has been tested.

▶ TRACE [start-add] [a-reg] [b-reg] { P-val [0] | -1 interval }

Dynamically traces executable program starting at **start-add** with **a-reg** and **b-reg** optionally preset and prints a diagnostic printout prior to the interpretive execution of each object instruction. The printout, defaults, and halt mechanism are described in the JUMPTRACE command.

When **p-val** is specified, the printout occurs only when the program counter equals p-val. If p-val is followed by 0, printout occurs the first time program counter equals p-val and every instruction thereafter.

When **-1 interval** is specified, printout occurs every interval instructions.

HLT instructions always cause a printout followed by a return to command mode.

▶ UPDATE location contents

Puts **contents** into **location** and prints the old and new contents of location.

▶ VERIFY source-start source-end copy-start

Verify memory block at locations **source-start** to **source-end** against a copy starting at **copy-start.**

The program types the address and content of each location in the block which does not match the corresponding word in the copy.

20
PSD command summary

PSD COMMAND SUMMARY

Enter rust colored letters in upper-case only.

▶ ACCESS address

Accesses a word in memory. The debugging program types the **address** and its contents and then waits for keyboard input in the following form:

[:format-symbol] [value] [:new-format] terminator

where **:format-symbol** is one of the optional input/output format symbols (see Table 18-3). The new format takes effect immediately. For example :HAF enters the hexadecimal value AF, regardless of the previous input/output mode. **value** replaces the contents of the accessed location. The format is the current input/output format. The **:new-format** symbol is one of the optional input/output format symbols (see Table 18-3). The new format takes effect *immediately* upon all subsequent output until a new format symbol is entered. **terminates** is one of the characters shown in Table 20-1.

Long instructions are input and printed in the assembler format, e.g., LDA% 2000.

Table 20-1. PSD Terminators

Terminator	Function
CR	Alters contents of current location (if a value is given), moves to current location +1 and prints its contents.
	Alters contents of current location (if a value is given), moves to current location -1 and prints its contents.
/ or ?	Exits from access mode. Does not close current location.
.n(CR)	Moves to current location +n and prints its contents (n is octal).
.-n(CR)	Moves to current location -n and prints its contents (n is octal).
@	For memory reference instructions of the form "INST* location" only. Saves a return address (current location +1), moves to the effective address location, and prints its contents. Subsequent accesses (terminated by CR, comma, ,. or . -) are relative to the effective address. A\ returns to the return address.
(	Goes to effective address without indirection, but saves current location as return address.
\	Returns to the return address saved by the last @.
)	Returns to the return address saved by the last (.
=	For memory reference instructions only; calculates and prints the effective address and its contents. No change is made to the current location or its contents.
!	Close a location, setting it to a new value if one was supplied, and return to command mode.

▶ BREAKPOINT location

Sets a breakpoint at the specified **location**. If the program is later executed and control reaches the breakpoint location, the debugging program prints CPU status and awaits further commands. Up to ten breakpoints may be inserted.

The GO command allows you to continue, leaving the breakpoint set.

▶ COPY source-start source-end target

Copies the block of memory from **source-start** to **source-end** into a new block of memory at **target**. If the target location lies between source start and source end, the non overlapped portion is propagated through the target area. The size of the target area is always equal to the size of the source.

Example:

```
F 1000   1010   :HFFFF   Fill locations 1000-1010 with hex FFFF
F 1011   1020   :HAAAA   Fill locations 1011-1020 with hex AAAA
D 1000   1020            Display locations 1000-1020
C 1010   1016   1012     Propagate alternate words of FFFF and AAAA
D 1000   1020            Display locations 1000-1020
```

▶ DEFINE symbol value

Defines a **symbol**. The **value** may be a constant or a constant expression. If the symbol has already been defined, it is given the new value.

Examples:

```
DE FOO 1ØØØ          FOO = OCTAL 1ØØØ
RE 1ØØØ              SET RELOCATION COUNTER
DE FOO >3            FOO = OCTAL 1ØØ3
DE FOO :AXX          FOO = 'XX'
DE FOO`: HF-A        FOO = 5
DE FOO :D>1Ø         FOO = OCTAL 1Ø12
```

Not allowed:

```
DE BAR FOO           SYMBOLS NOT PERMITTED AS VALUES
DE FOO (1+>3)        NO PARENTHESES
DE FOO >:HF          BAD SYNTAX.  SHOULD BE :H>F
DE FOO :AX           MUST HAVE TWO CHARACTERS AFTER :A
DE FOO :A X          FIRST CHARACTER AFTER :A MUST BE Ø-9, A-Z
DE FOO :SLT          :S IS AN OUTPUT SPECIFIER ONLY
```

▶ DUMP block-start block-end [words-per-line]

Prints the contents of the block of memory at locations **block-start** through **Block-end** on the use terminal or, optionall, in an external file. **Words-per-line** is number of words to be printed per line. The defaults is eight.

You *must* open a file before dumping to it. If there are several files open, DUMP will use the last one opened. Close the dump file before ending your session. If you have used PSD to open a file for a program use and you wish to dump to a terminal, issue an OPEN command with no parameters prior to issuing the DUMP command.

The default output format is eight octal words per line, preceded by the octal address of the first word on the line. Repetitious words are suppressed unless the number of words-per-line parameter only is specified.

Example:

```
O DMPFIL 1 2
D 1ØØØ 2ØØØ
Ø Ø 1 4
```

▶ EFFECTIVE block-start block-end address [mask]

Searches for an instruction with the specified effective **address** in the block from **block-start** to **block-end**, under an optional 16-bit **mask.**

If no mask is specified, the entire address is tested. When a match is found, the instruction and its address are printed at the user terminal. The search continues until location block-end has been tested.

Mask is a 16-bit word which may be expressed in any of the legal formats.

EFFECTIVE is useful in finding locations where a particular location is referenced.

The current value of the X register is used in the calculation. Instructions are interpreted in the current address/instruction mode as set by the MODE command and shown in the keys by the PRINT command.

▶ FILL block-start block-end constant :format

Fills the block of memory locations **block-start** through **block-end** with the specified **constant.** If block-end does not exceed block-start only the first location is filled. **:format** must be specified if you do not want the octal default. Specifying a format changes subsequent output formats. FILL is useful to test data area usage bypre-filling it with a visual pattern.

Example:

```
F 1ØØØ 1ØØ7 :HFFFF
D 1ØØØ 1ØØ7
1ØØØ FFFF
```

▶ GO [count] [a-reg] [b-reg] [x-reg] [keys]

Proceed from the current breakpoint, first executing the instruction at the breakpoint location. **count** is number of times to execute instruction at breakpoint location before breakpoint is taken. Default is one. The A, B and X registers and the keys may be preset (see Table 20-3).

A count may be overidden by resetting a breakpoint.

▶ JUMPTRACE [start-add] [a-reg] [b-reg]

Traces the execution of the object program from **start-add**. The default is current location. The A and B registers may be present; the default is old value.

Table 20-2. R and S Modes

C	D	—	ADR	—	SHIFT COUNT
1	2	3 4	5 — 6	7 — 8	9 — 16

Bit	Name	Meaning
1	C	Carry bit
2	D	Precision; 0=Single; 1=Double
3	—	Not used
4	—	Not used
5-6	ADR	Addressing Mode 00=16S 01=32S 11=32R 10=64R
7	—	Not used
8	—	Not used
9-16	SHIFT COUNT	Shift count—low order 8 bits of the floating point accumulator exponent register

JUMPTRACE, which is very useful for control-flow tracing, produces a diagnostic printout in the following format prior to the execution of any JMP, JST or HLT instruction:

Location: instruction A= B=X= K= R=

Any typed character will stop the trace. SVC's are not included in the trace.

▶ KEYS value

Sets CPU status keys to the specified octal **value.** The bit assignments vary depending on which mode you are in.

▶ LIST address

Prints the contents of **address** in the current output format.

Unlike ACCESS, LIST does not transfer the pointer to that location, a useful feature when you wish to examine a location without going there.

▶ LS

Enables the usage of external symbolic references during instruction typein and typeout.

To use the load map symbols:

1. Load the program and make a symbol file.
2. Restore the user program, invoke PSD and load the converted file.

The LS command puts PSD into symbolic mode. All addresses are typed as an offset from the nearest external symbol.

Once the load map is prepared in this manner, you can enable or disable symbol interpretation with the SYMBOL command.

Example:

```
OK, LOAD                 CALL THE LOADER
GO
$ LOAD B_PROG            LOAD THE PMA BINARY OBJECT PROGRAM
LOAD COMPLETE
$ SAVE $PROG             SAVE THE RUNFILE
$ MAP LSYM 10            CREATE A SYMBOL FILE LSYM
$ QUIT
OK, RESTORE $PROG        GET PROGRAM
OK, PSD                  GET PSD
GO
$ O LSYM 1 1             OPEN SYMBOL FILE ON FUNIT 1 FOR READING
$ LS                     LOAD SYMBOLS
$ O 014                  CLOSE FUNIT 1
```

▶ MA

Types the symbols and their definitions.

▶ MO { D16S | D32S | D32R | D64R }

D16S means use 16S address mode; **D32S,** use 32S address mode; **D32R,** use 32R address mode; and **D64R,** use 64R address mode.

Controls how effective addresses are interpreted by setting the address mode bits of the CPU status keys. See KEYS for a full discussion of the CPU status keys. Other status bits are unaffected. MODE is a fast symbolic way of setting just the address mode, when you don't care about the other CPU status key bits.

▶ MONITOR [start-add] [a-reg] [b-reg] address

Traces the object program from **start-add** (the default is the current location) looking for **address.** You may print the A and B registers.

If a memory reference instruction whose effective address equals address is encountered, data in trace format is printed on the terminal prior to the execution of that instruction. MONITOR answers the question "Where is the address being clobbered?"

Typing any character will stop the trace after several more lines. The character typed is considered part of the next command, so a space is the usual choice.

▶ NOT-EQUAL block-start block-end n-match [mask]

Searches memory between **block-start** and **block-end** for words *not* equal to **n-match** under an optional 16-bit **mask.**

The masking function is a 16-bit logical AND. If no mask is specified, the entire word is tested. When a non-match is found, the address and its contents are typed out and the search continues to block-end.

▶ OPEN file name file-unit key

Opens **file name** on **file-unit** to be used either as a DUMP output file or symbol table input file. **key** may be 1 (open for reading), 2 (open for writing), 3 (open for reading and writing), 4 (close).

The parameters are the same as for the PRIMOS OPEN command.

▶ PRINT

Prints CPU/PSD parameters in octal as follows:

prgctr:	breakpoint a-reg b-reg x-reg keys relcon
prgctr	Program counter at the time of breakpoint
relcon	Current value of the access mode relocation constant

▶ PROCEED [address] [a-reg] [b-reg] [x-reg] [keys]

Continue execution from breakpoint. Removes the current breakpoint if there is one, optionally sets a new breakpoint at **address,** and does a RUN command to the current program counter address. **a-reg, b-reg, x-reg** and **keys** have the same meaning as in the RUN command.

▶ QUIT

Returns to the PRIMOS operating system.

▶ RELOCATE value

Sets a new **value** for the access-mode relocation counter.

▶ RUN [start-add] [a-reg] [b-reg] [x-reg[[keys]

Runs the executable program starting at **start-add** location. Prior to program entry, **a-reg, b-reg, x-reg,** and **keys** are optionally loaded. Control does not return to the debugging utility unless a breakpoint is encountered.

▶ SEARCH block-start block-end match-word [mask]

Searches memory from **block-start** to **block-end** for words equal to **match-word** under an optional 16-bit **mask.**

If a mask is not specified, the entire word is tested. When a match is found, the address and its contents are typed out, and the search continues until location block-end has been tested.

▶ SYMBOL {1 | 0}

Controls the use of symbols in address typeout: **1** means turn on symbol typeout; **0**, turn off symbol typeout.

▶ TRACE [start-add] [a-reg] [b-reg] {P-val [0] | -1 interval}

Dynamically traces executable program starting at **start-add** with **a-reg** and **b-reg** optionally preset.

A diagnostic printout is produced prior to the interpretive execution of each object instruction. The printout, defaults, and halt mechanism are described in the JUMPTRACE command.

When **p-val** is specified, the printout occurs only when the program counter equals p-val. If p-val is followed by 0, printout occurs the first time program counter equals p-val and every instruction thereafter.

When **-1 interval** is specified, printout occurs every interval instructions.

HLT instructions always cause a printout followed by a return to command mode.

▶ UPDATE location contents

Puts **contents** into **location** and prints the old and new contents of a location.

▶ VERIFY block-start block-end copy

Verifies memory from **block-start** through **block-end** against a **copy** starting at copy.

The program types the address and content of each location which does not match the corresponding word in copy.

The format of a VERIFY printout is:

location block-contents copy-contents

▶ VERSION

Prints the version number and restart address of the PSD you are using. If your program gets into a loop or dies after a RUN command, you can issue a PR or GO command, starting at this restart address. This causes pseudo breakpoint, saving the registers and entering PSD. Only the program counter register value will be lost, and even this may be found by issuing a PRIMOS P command prior to restarting PSD.

▶ WHERE

Lists all currently installed breakpoints and their remaining proceed counts. A proceed count of one is not listed.

▶ XREGISTER value

Loads the X register with **value**—for example, before executing a RUN command or doing an effective address calculation.

▶ ZERO [location]

Removes the breakpoint at the specified **location**.

If location is omitted, Z removes the breakpoint at the current program counter location. (P will show the current location.)

21
VPSD command summary

Enter rust colored letters in upper case only.

▶ ACCESS address

Accesses a word in memory. The debugging program types the **address** and its contents and then waits for the keyboard input in the following form:

[:format-symbol] [value] [:new-format] terminator

where **:format-symbol** is one of the optional input/output format symbols (see Table 18-3). The new format takes effect immediately. For example, :HAF enters the hexadecimal value AF, regardless of the previous input/output mode. **value** replaces the contents of the access location. The format is the current input/output format. The **:new-format** symbol is one of the optional input/output format symbols (see Table 18-3). The new format takes effect *immediately* upon all subsequent output until a new format symbol is entered. **terminator** is one of the characters shown in Table 21-1.

Long instructions are input and printed in the same way as the assembler, e.g., LDA% 2000.

Table 21-1. VPSD Terminators

Terminator	Function
CR	Alters contents of current location (if a value is given), moves to current location +1 and prints its contents.
	Alters contents of current location (if a value is given), moves to current location -1 and prints its contents.
/ or ?	Exits from access mode. Does not close current location.
.n(CR)	Moves to current location +n and prints its contents (n is octal).
.-n(CR)	Moves to current location -n and prints its contents (n is octal).
@	For memory reference instructions of the form "INST* location" only. Saves a return address (current location +1), moves to the effective address location, and prints its contents. Subsequent accesses (terminated by CR, comma, ,. or . -) are relative to the effective address. A \ returns to the return address.
(	Goes to effective address without indirection, but saves current location as return address.
\	Returns to the return address saved by the last @.
)	Returns to the return address saved by the last (.
=	For memory reference instructions only; calculates and prints the effective address and its contents. No change is made to the current location or its contents. If the instruction references a register, the contents of the register are printed.

▶ BREAKPOINT location

Sets a breakpoint at the specified **location.** If the program is later executed and control reaches the breakpoint location, the debugging program prints CPU status and awaits further commands. Up to ten breakpoints may be inserted.

▶ BREGISTER

Prints the contents of the procedue base, stack base, link base and temporary base registers.

▶ COPY source-start source-end target

Copies the block of memory from **source-start** to **source-end** into a new block of memory at **target**. If the target location lies between source start and source end, the non-overlapped portion is propagated through the target area. The size of the target area is always equal to the size of the source.

Example:

F 1000	1010	:HFFFF		Fill locations 1000–1010 with HEX FFFF
F 1011	1020	:HAAAA		Fill locations 1011–1020 with HEX AAAA
D 1000	1020			Display locations 1000–1020
C 1010	1016	1012		Propagate alternate words of FFFF and AAAA
D 1000	1020			Display locations 1000–1020

▶ DUMP block-start block-end [words-per-line]

Prints the contents of the block of memory at locations **block-start** through **block-end** on the user terminal or optionally in an external file. **words-per-line** number of words to be printed per line. The default is eight.

You *must* open a file before dumping to it. If there are several files open, DUMP will use the last one opened. Close the dump file before ending your session. If you have used VPSD to open a file for program use and you wish to dump to a terminal, issue an OPEN command with no parameters prior to issuing the DUMP command.

The default output format is eight octal words per line, preceded by the octal address of the first word on the line. Repetitious words are suppressed unless the number of words-per-line parameter is specified.

Example:

```
O DMPFIL 1 2
D 1ØØØ 2ØØØ
Ø Ø 1 4
```

▶ EFFECTIVE block-start block-end address [mask]

Searches for an instruction with the specified effective **address** in the block from **block-start** to **block-end**, under an optional 16-bit **mask.**

If no mask is specified, the entire address is tested. When a match is found, the instruction and its address are printed at the user terminal. The search continues until location block-end has been tested.

Mask is a 16-bit word which may be expressed in any of the legal formats.

EFFECTIVE is useful in finding locations where a particular location is referenced.

The current values of the X and Y registers are used in the calculation. Instructions are interpreted in the current address/instruction mode as set by the MODE command and shown in the keys by the PRINT command.

▶ EXECUTE

Begins execution of a segmented program by passing control to SEG. SEG sets the initial register values; any other value at the time EX is issued is lost.

▶ FILL block-start block-end constant :format

Fills the block of memory locations **block-start** through **block-end** with the specified **constant**. If block-end does not exceed block-start only the first location is filled. **:format** must be specified if you do not want the octal default. Specifying a format changes subsequent output formats. FILL is useful to test data area usage by pre-filling it with a visual pattern.

Example:

```
F 1000 1007 :HFFFF
D 1000 1007
4001/1000 FFFF
```

▶ FA regno

Accesses field address register **regno.** New values may be entered to replace old ones. Carriage return advances to the "next" register, and "
"(" will switch to access mode and display the location referenced by the field address register in ASCII. A ")" will return to "FA" mode.

▶ FL regno

Accesses field length register **regno**. New values may be entered to replace old ones. Carriage return advances to the "next" register and "

▶ KEYS value

Sets CPU status keys to the specified octal **value**. The assignments vary depending on which mode you are in. See tables 21-2 and 21-3.

▶ LB seg-no word-no

Loads the link base register with a segment number (**seg-no**) and word number (**word-no**).

▶ LIST address

Prints the contents of **address** in the current output format. Unlike ACCESS, LIST does not transfer the pointer to that location, a very useful feature when you wish to examine a location without going there.

▶ MO { D16S | D32S | D32R | D64R | D64V }

D16S Means use 16S address mode; **D32S**, use 32S address mode; **D32R**, use 32R address mode; **D64R**, use 64R address mode; and **D64V**, use 64V address mode.

Controls how effective addresses are interpreted by setting the address mode bits of the CPU status keys. See KEYS for a full discussion of the CPU status keys. Other status bits are unaffected. MODE is a fast symbolic way of setting just the address mode, when you don't care about the other CPU status key bits.

D64V prints the segment and word number for all addresses (initial segment number is '4000) and interprets instructions as the Prime 400 hardware does. Base register references for all long instructions are printed as PB%, SB%, LB%, or XB%. Short instructions which reference SB or LB print SB or LB as part of the address.

▶ NOT-EQUAL block-start block-end n-match [mask]

Searches memory between **block-start** and **block-end** for words *not* equal to **n-match** under an optional 16-bit **mask**.

The masking function is a 16-bit logical AND. If no mask is specified, the entire word is tested. When a non-match is found, the address and its contents are typed out and the search continues to block-end.

▶ OPEN file name file-unit key

Opens **file name** on **file-unit** to be used either as a DUMP output file or symbol table input file. **Key** may be: 1 (open for reading), 2 (open for writing), 3 (open for reading and writing) or 4 (close).

The key parameters are the same as for the PRIMOS OPEN command.

▶ PRINT

Prints CPU/PSD parameters in octal as follows:

prgctr	breakpoint a-reg b-reg x-reg keys relcon [y-reg is VPSD]
prgctr	The program counter at the time of breakpoint
relcon	The current value of the access mode relocation

Table 21-2. Key Values: R and S Modes

C	D	—	ADR	—	SHIFT COUNT
1	2	3 4	5 — 6	7 — 8	9 — 16

Bit	Name	Meaning
1	C	Carry Bit
2	D	Precision; 0=Single; 1=Double
3	-	Not used
4	-	Not used
5-6	ADR	Addressing Mode 00=16S 01=32S 11=32R 10=64R
7	-	Not used
8	-	Not used
9-16	SHIFT COUNT	Shift count—low order 8 bits of the floating point accumulator exponent register.

If a mask is not specified, the entire word is tested. When a match is found, the address and its contents are typed out, and the search continues until location block-end has been tested.

▶ SN seg-no

Use **seg-no** as the segment number for all commands where only a word number is entered, such as UPDATE, DUMP, etc.

▶ UPDATE location contents

Puts **contents** into **location** and prints the old and new contents of location.

▶ VERIFY block-start block-end copy

Verifies memory from **block-start** through **block-end** against a copy starting at **copy**. The program types the address and content of each location which does not match the corresponding word in copy.

The format of a VERIFY printout is:

location block-contents copy-contents

▶ VERSION

Prints the version number and restart address of the VPSD you are using. If your program goes into a loop or dies after a RUN command, you can issue a PR or GO command, starting at this restart address. This causes a pseudo-breakpoint, saving the registers and entering VPSD. Only the program counter register value will be lost, and even this may be found by issuing a PRIMOS P command prior to restarting VPSD.

▶ WHERE

Lists all currently installed breakpoints and their remaining proceed counts. A proceed count of one is not listed.

▶ XB seg-no word-no

Loads temporary base register with a segment number (**seg-no**) and word number (**word-no**.

▶ XREGISTER value

Loads the X register with **value**—for example, before executing a RUN command or doing an effective address calculation.

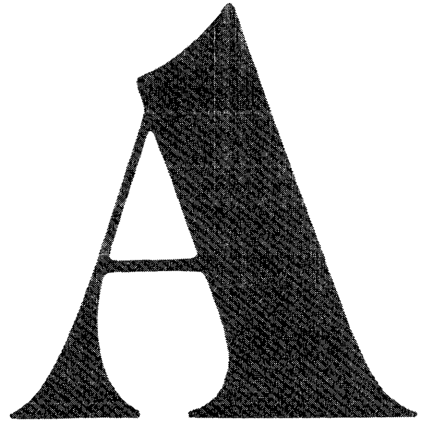
▶ YREGISTER value

Loads **value** into the Y index register.

▶ ZERO [location]

Removes the breakpoint at the specified **location**.

If location is omitted, **Z** removes the breakpoint at the current program-counter location. (P will show the current location.)



Assembler attribute table

APPENDICES

ASSEMBLER ATTRIBUTES

A list of the current assembler attributes follows. For a complete discussion of the use and function of assembler attributes see section 17, Macro Facility.

Label	Number	Description
	0	Number of arguments in current macro call
	1	Current macro call number
	100	A-register
	101	B-register
	102	X-register
CC	103	Current character pointer
CCM	104	Character count max of source line
	105	Used by dynm (must precede cdyn)
CDYN	106	Current dynamic storage pointer
MDYN	107	Maximum dynamic stack space used
	108	(spare)
MCLS	109	Macro list control
MCRC	110	Current extent of macro call number
	111	(spare)
MCRN	112	Current macro nest number
MODE	113	Current mode of assembler
NCRD	114	Current record number (card number)
NERR	115	Number of lines in program with errors
NMFL	116	No-macro-search flag (0=search)
PASS	117	Pass 1=0, pass 2=1
RPL	118	Current program counter value
STAK	119	Current temporary store stack limit
TC	120	Last character fetched
TCHB	121	TC held back flag
TCNT	122	TC repeat count
IFLG	123	Indirect operator flag (0=indirect)
DFVL	124	Table search value
SEG	125	Seg mode flag (0, 1, -1)
ABM	126	Current abstract machine 0=S,R and 1=V,I
PMB	127	Procedure size max
LBM	128	Link size max

B
ASCII character set

The standard character set used by Prime is the ANSI, ASCII 7-bit set.

PRIME USAGE

Prime hardware and software uses standard ASCII for communications with devices. The following points are particularly important to Prime usage.

- Output Parity is normally transmitted as a zero (space) unless the device requires otherwise, in which case software will compute transmitted parity. Some controllers (e.g., MLC) may have hardware to assist in parity generations.
- Input Parity is ignored by hardware and by standard software. Input drivers are responsible for making the parity bit suit the host software requirements. Some controllers (e.g., MLC) may assist in parity error detection.
- The Prime internal standard for the parity bit is one, i.e., '200 is added to the octal value.

KEYBOARD INPUT

Non-printing characters may be entered into text with the logical escape character ^ and the octal value. The character is interpreted by output devices according to their hardware.

Example: Typing ^207 will enter *one* character into the text.

CTRL-P	('220)	is interpreted as a .BREAK.
.CR.	('215)	is interpreted as a newline (.NL.)
"	('242)	is interpreted as a character erase
?	('277)	is interpreted as line kill
\	('334)	is interpreted as a logical tab (Editor)

Table B-1. ASCII Character Set (Non-Printing)

Octal Value	ASCII Char	Comments/Prime Usage	Control Char
200	NULL	Null character - filler	^@
201	SOH	Start of header (communications)	^A
202	STX	Start of text (communications)	^B
203	ETX	End of text (communications)	^C
204	EOT	End of transmission (communications)	^D
205	ENQ	End of I.D. (communications)	^E
206	ACK	Acknowledge affirmative (communications)	^F
207	BEL	Audible alarm (bell)	^G
210	BS	Back space on position (carriage control)	^H
211	HT	Physical horizontal tab	^I
212	LF	Line feed; ignored as terminal input	^J
213	VT	Physical vertical tab (carriage control)	^K
214	FF	Form feed (carriage control)	^L
215	CR	Carriage return (carriage control) *(1)*	^M
216	SO	RRS-red ribbon shift	^N
217	SI	BRS-black ribbon shift	^O
220	DLE	RCP-relative copy *(2)*	^P
221	DC1	RHT-relative horizontal tab *(3)*	^Q
222	DC2	HLF-half line feed forward (carriage control)	^R
223	DC3	RVT-relative vertical tab *(4)*	^S
224	DC4	HLR-half line feed reverse (carriage control)	^T
225	NAK	Negative acknowledgement (communications)	^U
226	SYN	Synchronocity (communications)	^V
227	ETB	End of transmission block (communications)	^W
230	CAN	Cancel	^X
231	EM	End of Medium	^Y
232	SUB	Substitute	^Z
233	ESC	Escape	^[
234	FS	File separator	^\
235	GS	Group separator	^]
236	RS	Record separator	^^
237	US	Unit separator	^_

Notes

1. Interpreted as .NL. at the terminal.
2. .BREAK. at terminal. Relative copy in file; next byte specifies number of bytes to copy from corresponding position of preceding line.
3. Next byte specifies number of spaces to insert.
4. Next byte specifies number of lines to insert.

Conforms to ANSI X3.4–1968

The parity bit ('200) has been added for Prime-usage. Non-printing characters (^c) can be entered at most terminals by typing the (control) key and the character key simultaneously.

Table B-2. ASCII Character Set (Printing)

Octal Value	ASCII Character	Octal Value	ASCII Character	Octal Value	ASCII Character
240	.SP. *(1)*	300	@	340	`*(9)*
241	!	301	A	341	a
242	"*(2)*	302	B	342	b
243	# *(3)*	303	C	343	c
244	$	304	D	344	d
245	%	305	E	345	e
246	&	306	F	346	f
247	' *(4)*	307	G	347	g
250	(	310	H	350	h
251	)	311	I	351	i
252	*	312	J	352	j
253	+	312	K	353	k
254	, *(5)*	314	L	354	l
255	-	315	M	355	m
256	.	316	N	356	n
257	/	317	O	357	o
260	0	320	P	360	p
261	1	321	Q	361	q
262	2	322	R	362	r
263	3	323	S	363	s
264	4	324	T	364	t
265	5	325	U	365	u
266	6	326	V	366	v
267	7	327	W	367	w
270	8	330	X	370	x
271	9	331	Y	371	y
272	:	332	Z	372	z
273	;	333	[	373	{
274	<	334	\	374	¦
275	=	335	]	375	}
276	>	336	ˆ *(7)*	376	~ *(10)*
277	? *(6)*	337	_*(8)*	377	DEL *(11)*

Notes

1. Space forward one position
2. Terminal usage - erase previous character
3. £ in British use
4. Apostrophe/single quote
5. Comma
6. Terminal usage - kill line
7. 1963 standard ↑; terminal use - logical escape
8. 1963 standard ←
9. Grave
10. 1963 standard ESC
11. Rubout - ignored

Conforms to ANSI X3.4-1968
1963 variances are noted

The parity bit ('200) has been added for Prime usage.

C
Error messages

INTRODUCTION

Error messages are given in the following order:

1. PMA Error Messages
2. Loader Error Messages
3. SEG Loader Error Messages
4. Run-time Error Messages

In each group errors are listed alphabetically.

Run-time error messages beginning with a filename, device name, UFDname, etc., are alphabetized according to the first word which is constant. The user should have no trouble in determining this word (the second word in the message). Leading asterisks, etc., are ignored in alphabetizing. All run-time errors have been grouped together to facilitate lookup by the user.

PMA ERROR MESSAGES

C00:	INSTRUCTION IMPROPERLY TERMINATED
F00:	ILLEGAL TERMINATOR ON ARGUMENT # EXPRESSION
F01:	UNRECOGNIZED OPERATOR IN EXPRESSION
F02:	FAIL PSEUDO-OP ENCOUNTERED
F03:	OPERAND FIELD EMPTY; OPERAND REQUIRED
G00:	GO-TO OR BACK-TO USED OUTSIDE OF MACRO OR ARGUMENT IS NOT SYMBOL
G01:	END/ENDM PSEUDO-OP IS WITHIN GO-TO OR BACK-TO SKIP AREA
I00:	TAG MODIFIER ILLEGAL ON GENERIC, I/O, OR SHIFT INSTRUCTION
I01:	TAG MODIFIED NOT PERMITTED ON 32I MODE FIELD INSTRUCTION
I03:	CAN'T MAKE THIS INSTRUCTION SHORT (#)
I04:	ILLEGAL TAG MODIFIED FIELD ON 64V MODE LDX CLASS INSTRUCTION
I05:	TAG MODIFIED NOT PERMITTED ON 64V MODE BRANCH INSTRUCTION
I06:	ILLEGAL INDIRECT OR INDEX SPECIFICATION WITH COMMON/ EXTERNAL SYMBOL
I07:	INDEX SPECIFIED INVALID WITH AP/IP PSEUDO-OP
I08:	TAG MODIFIED FIELD NOT PERMITTED ON 32I MODE BRANCH INSTRUCTION

L00: IMPROPER LABEL (CONSTANT OR TERMINATOR IN LABEL FIELD)
L01: EXTERNALL VARIABLE DISALLOWED IN LITERAL
L02: ILLEGAL ARGUMENT IN EQU, SET, OR XSET
M00: SYMBOL MULTIPLY DEFINED
N00: 'END' STATEMENT ENCOUNTERED WITHIN MACRO OR IF
O00: UNRECOGNIZED OPCODE OR 32I-ONLY OPCODE IN NON-32I MODE
O01: THIS MEMORY REFERENCE INSTRUCTION ONLY PERMITTED IN 64V MODE
O02: THIS MEMORY REFERENCE INSTRUCTION ONLY PERMITTED IN S/R MODE
P00: MISMATCHED PARENTHESIS
Q00: AP ONLY PERMITTED IN 64V/32I MODE
Q01: IP ONLY PERMITTED IN 64V/32I MODE
Q02: ENDM PSEUDO-OP DISALLOWED OUTSIDE OF MACRO DEFINITION
R00: ARITHMETIC STACK OVERFLOW: REDUCE THE COMPLEXITY OF THE EXPRESSION AND TRY AGAIN
R01: MULTIPLY DEFINED MACRO OR MACRO NAME FIELD EMPTY
S00: INSTRUCTION REQUIRES DESCECTORIZATION ('LOAD' MODE)
S01: INDIRECT DAC DISALLOWED IN C64R MODE
S02: 64V INSTRUCTION DISALLOWED IN C64R MODE
T00: SYNTAX ERROR IN 32I MODE TAG MODIFIED FIELD
U00: UNDEFINED SYMBOL IN ADDRESS FIELD OR EXPRESSION
U01: UNDEFINED SYMBOL IN 6ORG' OR 6SETB'
V01: CONTENTS OF BIT FIELD OUT OF RANGE
V02: UNRECOGNIZED OPERATOR IN EXPRESSION
V03: FUNCTION CODE OR DEVICE ADDRESS OUT OF RANGE IN I/O INSTRUCTION
V04: SHIFT COUNT OUT OF RANGE IN SHIFT INSTRUCTION
V05: NO COMMA FOLLOWS FAR SPECIFICATION IN FIELD ADDRESS INSTRUCTION
V06: NO COMMA FOLLOWS REGISTER # IN 32I MODE REGISTER GENERIC
V07: NO COMMA FOLLOWS REGISTER # IN 32I MODE FLOATING PT REGISTER GENERIC
V08: NO COMMA FOLLOWS REGISTER # IN 32I MODE BIT TEST INSTRUCTION
V09: NO COMMA FOLLOW BIT # IN 32I MODE BIT TEST INSTRUCTION
V10: BAD DELIMITER IN 32I MODE GENERAL REGISTER MEMORY REFERENCE INSTRUCTION
V11: BAD DELIMITER IN 32I MODE SHIFT INSTRUCTION
V12: BAD SHIFT COUNT IN 32I MODE SHIFT INSTRUCTION
V13: ILLEGAL TAG MODIFIED FIELD FOR 32I MODE SHIFT INSTRUCTION
V14: BAD DELIMITER FOLLOWS REGISTER # IN 32I MODE PIO INSTRUCTION
V15: LABEL REQUIRED ON DFTB/DFTV PSEUDO-OP
V16: OPEN PARENTHESIS MISSING ON DFTB/DFVT ARGUMENT
V17: CLOSE PARENTHESIS MISSING ON DFTB/DFVT ARGUMENT
V18: LABEL REQUIRED ON IFTF, IFTT, IFVT, IFVF PSEUDO-OP
V19: SYMBOL NOT FOUND IN IFTF, IFTT, IFVT, IFVF PSEUDO-OP
V20: ABS/REL PSUEDO-OP ILLEGAL IN SEG/SEGR MODE

V21:	SEG/SEGR PSEUDO-OP SPECIFIED AFTER CODE HAS BEEN GENERATED
V22:	PROC/LINK SPECIFICATION ONLY ALLOWED IN SEG/SEGR MODE
V23:	FIELD OUT OF RANGE IN DDM PSEUDO-OP
V24:	ILLEGAL ARGUMENT FOLLOWS 'EXT' PSEUDO-OP
V25:	'END' PSEUDO-OP ENCOUNTERED WITHIN MACRO
V26:	SYNTAX ERROR IN DYMN PSEUDO-OP ARGUMENT(S)
V27:	ILLEGAL ARGUMENT FOLLOWS SUBR/ENT PSEUDO-OP
V28:	16 BITS NOT DEFINED BY VFD PSEUDO-OP (UNDEFINED BITS SET TO 0)
V29:	OPERAND MISSING OR UNRECOGNIZED OPERATOR IN EXPRESSION
V30:	UNTERMINATED CHARACTER STRING
V31:	VALUE OVERFLOW IN FLOATING POINT NORMALIZE
V32:	VALUE OVERFLOW IN FLOATING POINT (RE-)NORMALIZE
V33:	SIGNIFICANCE LOST IN SCALED BINARY DATUM
V34:	FLOATING POINT VALUE OUT OF RANGE
V35:	'BCI' PSEUDO-OP REPEAT COUNT ERROR
V36:	ILLEGAL SYMBOL TYPE IN 'BCI' REPEAT COUNT SPECIFICATION
V37:	'CALL' PSEUDO-OP FOLLOWED BY CONSTANT OR TERMINATOR
V38:	BAD ADDRESS FIELD FOLLOWING 'COMN' PSEUDO-OP
V39:	ILLEGAL REPEAT COUNT IN DATA DEFINITION PSEUDO-OP
V40:	ILLEGAL ARGUMENT FOLLOW DEC/OCT PSEUDO-OP
V41:	RLIT SPECIFIED AFTER CODE HAS BEEN GENERATED
V42:	WCS ENTRANCE OUT OF RANGE—MUST BE 0–63
V43:	SYML NOT PERMITTED AFTER CODE HAS BEEN GENERATED
V44:	SYML ONLY PERMITTED IN SEG/SEGR MODE
X00:	32I MODE REGISTER SPECIFICATION ERROR
Y00:	PHASE ERROR—THE VALUE OF THE SYMBOL DEFINED ABOVE DIFFERS BETWEEN PASS 1 AND PASS 2
Z00:	ILLEGAL ABSOLUTE REFERENCE IN SEG/SEGR MODE
Z01:	ABSOLUTE REFERENCE OUTSIDE OF 0–7 DISALLOWED IN SEG
Z02:	ABSOLUTE REFERENCE IN AP/IP DISALLOWED
Z03:	ONLY 1 EXTERNAL NAME IS ALLOWED WITHIN AN EXPRESSION
Z04:	THE MODE ASSOCIATED WITH THE RESULT OF THE EXPRESSION IS ILLEGAL WITH SPECIFIED INSTRUCTION
Z05:	THE RESULTANT MODE OF THIS EXPRESSION IS ILLEGAL WHEN USED WITH THE SPECIFIED OPCODE OR PSEUDO-OP
Z06:	MORE THAN 1 OPERAND IS NON ABS/REL OR THE RIGHT-HAND OPERAND IS NON ABS/REL
Z07:	AN EXTERNAL NAME IS NOT PERMITTED
Z08:	NON-16-BIT INTEGER IS ILLEGAL IN AN EXPRESSION

LOADER ERROR MESSAGES

ALREADY EXISTS!

An attempt is being made to define a new symbol; however, the symbol name is already a defined symbol in the symbol table.

BAD OBJECT FILE

The object text is not recognizable. This usually occurs when an attempt is made to load source code or when the object text was compiled or assembled for segmented loading.

BASE SECTOR 0 FULL

All locations in the sector zero base area have been used. Use the AU command to generate base areas at regular intervals, or use the SETB or LOAD commands to specifically place base areas.

CAN'T DEFER COMMON, OLD OBJECT TEXT

The Defer Common command has been given and a module created with a pre-Rev. 14 compiler or assembler has been encountered. It is not possible to defer Common in this case. The module must be recreated with a Rev. 15 or later compiler or assembler.

CAN'T - PLEASE SAVE

The EXecute command has been given for a run file which has required virtual loading. SAve the runfile and give the EXecute command.

CM$

Command line error. Unrecognized command given. Not fatal.

COMMON OUT OF REACH

COMMON above '100000 is out of reach of the current load mode (16S, 32S or 32R). Use the MOde command to set the load mode to 64R.

COMMON TOO LARGE

Definition of this COMMON block causes COMMON to wrap around through zero. Moving the top of COMMON - with the COMMON command - may help.

sname ILLEGAL COMMON REDEFINITION

An attempt is being made to redefine COMMON block **sname** to a longer length. The user's program should be examined for consistent COMMON definitions. At the very least the longest definition for a COMMON block should be first.

xxxxxx MULTIPLE INDIRECT

A module loading in 64R mode requires a second level of indirection at location **xxxxxx**. This message usually results when an attempt is made to load code compiled or assembled for 32R mode in 64R mode. It can also happen if code has accidentally been loaded into base areas as the result of a bad load command sequence.

sname xxxxxx NEED SECTOR ZERO LINK

At location **xxxxxx** a link is required for desectoring the instruction. No base areas are within reach except sector zero. The last referenced symbol was **sname**. This message is only generated when the SZ command has been given. Sname may be the

name of a COMMON block, the name of the routine to which the link should be made, or the name of the module being loaded.

xxxxxx NO POST BASE AREA, OLD OBJECT TEXT

A post base area has been specified for module which was created with a pre-Rev.14 compiler or assembler. No base area is created. Recreate the object text with a Rev. 15 or later compiler or assembler. This is not a fatal error.

PROGRAM-COMMON OVERLAP

The module being loaded is attempting to load code into an area reserved for COMMON. Use the loader's COmmon command to move COMMON up higher.

PROGRAM TOO LARGE

The program has loaded into the last location in memory and has wrapped around to load in Location 0. The program size must be decreased. Alternatively, compile in 64V mode and use SEG.

REFERENCE TO UNDEFINED COMMON

An attempt is being made to link to a COMMON name which has not been defined. This usually happens to users creating their own translators.

SECTORED LOAD MODE INVALID

A module compiled or assembled to load in R mode has been loaded in S mode. Use the MOde command to reset the load mode. It might be a good idea to be sure that all modules are correctly written, since the default load mode is 32R.

SYMBOL NOT FOUND

An attempt is being made to equate two symbols with the SYmbol command and the old symbol does not exist.

SYMBOL TABLE FULL

The symbol table has expanded down to location '4000. The last buffer cannot be assigned to the symbol table. Rebuild LOAD to load in higher memory locations, or reduce the number of symbols in the load.

SYMBOL UNDEFINED

An attempt is being made to equate two symbols; however, the old symbol is an undefined symbol in the symbol table.

64R LOAD MODE INVALID

A module compiled or assembled to run in only 32K of memory is being loaded in 64R mode. Recompile or reassemble or change the load mode with the loader's MOde command.

SEG LOADER ERROR MESSAGES

BAD OBJECT FILE

User is attempting to load file which has faulty code. The file may not be an object file or it may be incorrectly compiled. Fatal error, the load must be aborted.

CAN'T LOAD IN SECTORED MODE

The Loader is attempting to load code in sectored mode which has not been compiled in sectored mode. This could arise if trying to load a module compiled or assembled in 16S or 32S mode. It is unlikely that the average applications programmer will encounter this. Fatal error, abort load.

CAN'T LOAD IN 64V OR 64R MODE

The Loader is attempting to load code in 64V mode which is not compiled in that mode. This would arise if:

1. A program was compiled in a mode other than 64V.
2. A PMA module is written in code other than 64V and its mode is not specified.

In case 1, the user should recompile the program.

In case 2, which the average applications programmer is unlikely to encounter, the PMA module must be modified. Fatal error, abort load.

COMMAND ERROR

An unrecognized command was entered or the filenames/parameters following the command are incorrect. Usually not fatal.

EXTERNAL MEMORY REFERENCE TO ILLEGAL SEGMENT

An attempt was made to load a 64R mode program, causing a reference to an illegal segment number. Recompile in 64V mode. Fatal error, abort load.

ILLEGAL SPLIT ADDRESS

Incorrect use of the Loader's SPLIT command. Segments may be split at '4000 boundaries only (i.e., '4000, '10000, '14000, etc.). Not fatal; resplit segment.

MEMORY REFERENCE TO COMMON IN ILLEGAL SEGMENT

An attempt was made to load a 64R mode program wherein COMMON would be allocated to an illegal segment number. Recompile in 64V mode. Fatal error, abort load.

NO FREE SEGMENTS TO ASSIGN

All SEG's segments have been allocated; no more are available at present. Use SYMBOL command to eliminate COMMON from assigned segments, thus reducing the number of assigned segments required. (User may need larger version of SEG and PRIMOS). Fatal error, abort load.

NO ROOM IN SYMBOL TABLE

Unlikely to occur; no user solution. A new issue of SEG with a bigger symbol table is required. Check with analyst. As a temporary measure, user may try to reduce number of symbols used in program. Fatal error, abort load.

REFERENCE TO UNDEFINED SEGMENT

Almost always caused by improper use of the SYMBOL command to allocate initialized COMMON. Initialized COMMON cannot be located with the SYMBOL command; use R/SYMBOL or A/SYMBOL instead.

SECTOR ZERO BASE AREA FULL

Extremely unlikely to occur. Not correctable at applications level. Check with analyst. Fatal error, abort load.

SEGMENT WRAP AROUND TO ZERO

An attempt has been made to load a 64R mode program. The program has exceeded 64K and is trying to be loaded over code previously loaded. Recompile in 64V mode. Fatal error, abort load.

RUN-TIME ERROR MESSAGES

ACCESS VIOLATION — **64V mode**

Attempt to perform operations in segments to which user has no right.

****AD — **R-mode function**

Overflow or underflow in double-precision addition/subtraction (A$66,S$66).

ALL REMOTE UNITS IN USE — **File System**

Attempt made to assign a remote unit when none are available. (Network error) [E$FUIU]

****** ALOG/ALOG 10 - ARGUMENT <=0** — **V-mode function**

Argument not greater than zero used in logarithm (ALOG, ALOG 10) function.

filename ALREADY EXISTS — **Old file call**

Attempt to create a file or UFD with the name of one already existing. [CZ]

ALREADY EXISTS — **File System**

Attempt made to create, in the UFD, a sub-UFD with the same name as one already existing. (CREA$$) [E$EXST]

****AT — **R-mode function**

Both arguments are zero in the ATAN2 function.

****** ATAN2 - BOTH ARGUMENTS = 0** — **V-mode function**

Both arguments are zero in the ATAN2 function.

****** ATTDEV - BAD UNIT** — **V-mode call**

Incorrect logical device unit number in the ATTDEV subroutine call.

BAD CALL TO SEARCH — **Old file call**

Error in calling the SEARCH subroutine, e.g., incorrect parameter. [SA]

BAD DAM FILE — **Old file call**

The DAM file specified has been corrupted - either by the programmer or by a system problem. [SS]

BAD DAM FILE — **File System**

The DAM file specified has been corrupted - either by the programmer or by a system problem. (PRWF$$, SRCH$$). [E$BDAM]

BAD FAM SVC **File System**

System problem; will not be seen by applications programmer. [E$BFSV]

BAD KEY **File System**

Incorrect key value specified in subroutine argument. (ATCH$$, RDEN$$, SATR$$, SRCH$$, SGDR$$) [E$BKEY]

BAD PARAMETER **Old file call**

Incorrect parameter value in subroutine call. [SA]

BAD PASSWORD **Old file call**

Incorrect password specified in ATTACH subroutine. Returns to PRIMOS level attached to no UFD. [AN]

BAD PASSWORD **File System**

Incorrect password specified in ATCH$$ subroutine. Returns to PRIMOS level attached to no UFD. [ATCH$$] [E$BPAS]

Note

To protect UFD privacy the system does not allow the user to trap BAD PASSWORD errors.

BAD RTNREC **PRIMOS**

System error.

BAD SEGDIR UNIT **File System**

Error generated in accessing segment directory, i.e., PRIMOS file unit specified is not a segment directory. (SRCH$$) [E$BSUN]

BAD SEGMENT NUMBER **File System**

Attempt made to access segment number outside valid range. [E$BSGN]

BAD SVC **PRIMOS**

Bad supervisor call. In FORTRAN usually caused by program writing over itself.

BAD TRUNCATE OF SEGDIR **File System**

Error encountered in truncating segment directory. (SGDR$$) [E$BTRN]

BAD UFD **File System**

UFD has become corrupted. (ATCH$$, CREA$$, GPAS$$, RDEN$$, SATR$$, SRCH$$) [E$BUFD]. Calls to RDEN$$ return this as a trappable error; other commands return to the PRIMOS command level.

BAD UNIT NUMBER **File System**

PRIMOS file unit number specified is invalid - outside legal range. (PRWF$$, RDEN$$, SRCH$$, SGDR$$). [E$BUNT]

BEGINNING OF FILE **File System**

Attempt was made to access locations before the beginning of the file. (PRWF$$, RDEN$$, SGDR$$) [E$BOF]

******BN n** — **R-mode function**

Device error in REWIND command on FORTRAN logical unit n.

BUFFER TOO SMALL — **File System**

Buffer as defined is not large enough to accomodate entry to be read into it. (RDEN$$) [E$BFTS]

****** DATAN - BAD ARGUMENT** — **V-mode function**

The second argument in the DATAN2 function is zero.

******DE** — **R-mode function**

The exponent of a double-precision number has overflowed.

DEVICE IN USE — **File System**

Attempt was made to ASSIGN a device currently assigned to another user. [E$DVIU]

DEVICE NOT ASSIGNED — **File System**

Attempt was made to perform I/O operations on a device before assigning that device. [E$NASS]

DEVICE NOT STARTED — **File System**

Attempt was made to access a disk not physically or logically connected to the system. If disk must be accessed, systems manager must start it up. [E$DNS]

****** DEXP - ARGUMENT TOO LARGE** — **V-mode function**

The argument of the DEXP function is too large; i.e., it will give a result outside the legal range.

****** DEXP - OVERFLOW/UNDERFLOW** — **V-mode function**

An overflow or underflow condition occurred in calculating the DEXP function.

DIRECTORY NOT EMPTY — **File System**

Attempt was made to delete a non-empty directory. (SRCH$$) [E$DNTE]

DISK FULL — **Old file call**

No more room for creating/extending any type of file on disk. [DJ]

DISK FULL — **File System**

No more room for creating/extending any type of file on disk. (CREA$$, PRWF$$, SRCH$$, SGDR$$). [E$DKFL]

Note

Space may be made available. Use the internal PRIMOS commands ATTACH, LISTF, and DELETE to remove files which are no longer needed.

DISK I/O ERROR — **File System**

A read/write error was encountered in accessing disk. Returns immediately to PRIMOS level. Not correctable by applications programmer. (ATTCH$$, CREA$$, GPAS$$, PRWF$$, RDEN$$, SATR$$, SRCH$$, SGDR$$). [E$DISK]

DISK WRITE-PROTECTED — **File System**

An attempt has been made to write to a disk which is WRITE-protected. [E$WTPR]

DK ERROR — **Old file call**

A read/write error was encountered in accessing disk. [WB]

******DL** — **R-mode function**

Argument was not greater than zero in DLOG or DLOG2 function.

****** DLOG/DLOG2 - ARGUMENT< = 0** — **V-mode function**

Argument not greater than zero was used in DLOG or DLOG2 function.

******DN n** — **R-mode function**

Device error (end of file) on FORTRAN logical unit n.

****** DSIN/DCOS - ARGUMENT RANGE ERROR** — **V-mode function**

Argument outside legal range for DSIN or DCOS function.

****** DSQRT - ARGUMENT <0** — **V-mode function**

Negative argument in DSQRT function.

****** DT** — **R-mode function**

Second argument is zero in DATAN2 function. (D$22)

DUPLICATE NAME — **Old file call**

Attempt to create/rename a file with the name of an existing file. [CZ]

******DZ** — **R-mode function**

Attempt to divide by zero (double-precision).

END OF FILE — **File System**

Attempt to access location after the end of the file. (PRWF$$, RDEN$$, SGDR$$) [E$EOF]

******EQ** — **R-mode function**

Exponent overflow. (A$81)

******EX** — **R-mode function**

Exponent function value too large in EXP or DEXP function.

****** EXP - ARGUMENT TOO LARGE** — **V-mode function**

The argument of the EXP function is too large, i.e., it will give a result outside the legal range.

****** EXP - OVERFLOW** — **V-mode function**

Overflow occurred in calculating the EXP function.

FAM ABORT — **File System**

System error. [E$FABT]

FAM - BAD STARTUP **File System**

System error. [E$FBST]

FAM OP NOT COMPLETE **File System**

Network error. [E$FONC]

******FE** **R-mode function**

Error in FORMAT statement. FORMAT statements are not completely checked at compile time. (F$IO)

FILE IN USE **File System**

Attempt made to open a file already opened or to close/delete a file opened by another user, etc. (SRCH$$) [E$FDEL]

FILE OPEN ON DELETE **File System**

Attempt made to delete a file which is open. (SRCH$$) [E$FDEL]

FILE TOO BIG **File System**

Attempt made to increase size of segment directory beyond size limit. (SGDR$$) [E$FITB]

******FN n** **R-mode function**

Device error in BACKSPACE command on FORTRAN logical unit n.

****** F$BN - BAD LOGICAL UNIT** **V-mode function**

FORTRAN logical unit number out of range.

****** F$FLEX - DOUBLE-PRECISION DIVIDE BY ZERO** **64V mode**

Attempt has been made to divide by zero.

****** F$FLEX - DOUBLE-PRECISION EXPONENT OVERFLOW** **64V mode**

Exponent of a double-precision number has exceeded maximum.

****** F$FLEX - REAL => INTEGER CONVERSION ERROR** **64V mode**

Magnitude of real number too great for integer conversion.

****** F$FLEX - SINGLE-PRECISION DIVIDE BY ZERO** **64V mode**

Attempt has been made to divide by zero.

****** F$FLEX - SINGLE-PRECISION EXPONENT OVERFLOW** **64V mode**

Exponent of a single-precision number has exceeded maximum.

****** F$IO - FORMAT ERROR** **V-mode function**

Incorrect FORMAT statement. FORMAT statements are not completely checked at compile time.

****** F$IO - FORMAT/DATA MISMATCH** **V-mode function**

Input data does not correspond to FORMAT statement.

****** F$IO - NULL READ UNIT** **V-mode function**

FORTRAN logical unit for READ statement not configured properly.

******II** **R-mode function**

Exponentiation exceeds integer size. (E$11)

ILLEGAL INSTRUCTION AT octal-location **R mode and 64V mode**

An instruction at octal-location cannot be identified by the computer.

ILLEGAL NAME **File System**

Illegal name specified for a file or UFD. (CREA$$, SRCH$$) [E$BNAM]

ILL REMOTE REF **File System**

Attempt to perform network operations by user not on network. [E$IREM]

ILLEGAL SEGNO **64V mode**

Program references a non-existent segment or a segment number greater than those available to the user.

ILLEGAL TREENAME **File System**

The string specified for a treename is syntactically incorrect. [E$ITRE]

******IM** **R-mode function**

Overflow or underflow occurred during a multiply. (M$11, E$11)

filename IN USE **Old file call**

Attempt made to open a file already opened, or to close/delete a file opened by another user, etc. [SI]

INVALID FAM FUNCTION CODE **File System**

System error. [E$FIFC]

****** I**I - ARGUMENT ERROR** **V-mode function**

Exponentiation exceeds integer size.

******LG** **R-mode function**

Argument not greater than zero in ALOG or ALOG10 function.

MAX REMOTE USERS EXCEEDED **File System**

No more users may access the network. [E$TMRU]

NAME TOO LONG **File System**

Length of name in argument list exceeds 32 characters. [E$NMLG]

NO AVAILABLE SEGMENTS **64V mode**

Additional segment(s) required - none available. User should log out to release assigned segments and try again later.

NO PHANTOMS AVAILABLE **File System**

An attempt has been made to spawn a phantom. All configured phantoms are already in use. [E$NPHA]

NO RIGHT **File System**

User does not have access right to file, or does not have write access in UFD when attempting to create a sub-UFD. (CREA$$, GPAS$$, SATR$$, SRCH$$, SGDR$$) [E$NRIT]

NO ROOM **File System**

An attempt has been made to add to a table of assignable devices with a DISKS or ASSIGN AMLC command and the table is already filled. [E$ROOM]

NO TIME **File System**

Clock not started. System error. [E$NTIM]

NO UFD ATTACHED **Old file call**

User not attached to a UFD [AL, SL]. Usually occurs after attempt to attach with a bad password.

NO UFD ATTACHED **File System**

User not attached to a UFD. (ATCH$$, CREA$$, GPAS$$, SATR$$, SRCH$$). [E$NATT] Usually occurs after attempt to attach with a bad password.

NO VECTOR **R and 64V mode**

User error in program has caused PRIMOS to attempt to access an unloaded element.

1. A UII, PSU, or FLEX to location 0
2. Trap to location 0
3. SVC switch on, SVC trap and location '65 is 0.

NOT A SEGDIR **File System**

Attempt to perform segment director operations on a file which is not a segment directory. (SRCH$$) [E$NTSD]

NOT A UFD **Old file call**

Attempt to perform UFD operations on a file which is not a UFD. [AR]

NOT A UFD **File System**

Attempt to perform UFD operations on a file which is not a UFD. (ATCH$$, GPAS$$, SRCH$$). [E$NTUD]

device-name NOT ASSIGNED **PRIMOS**

User program has attempted to access an I/O device which has not been assigned to the user by a PRIMOS command.

filename NOT FOUND **Old file call**

File specified in subroutine call not found. [AH, SH]

filename NOT FOUND **File System**

File specified in subroutine call not found. (ATCH$$, GPAS$$, SATR$$, SRCH$$) [E$FNTF]

filename NOT FOUND IN SEGDIR — **File System**

Filename specified in subroutine call not found in specified segment directory. (SRCH$$, SGDR$$) [E$FNTS]

NULL READ UNIT — **PRIMOS**

Program has attempted to read with a bad unit number. This may be caused by a program overwriting itself (array out of bounds).

OLD PARTITION — **File System**

Attempt to perform, in an old file partition, an operation possible only in a new file partition; e.g., date/time information access. (SATR$$) [E$OLDP]

******PA n** — **R-mode function**

PAUSE statement n (octal) encountered during program execution

****** PAUSE n** — **V-mode function**

PAUSE statement n (octal) encountered during program execution.

POINTER FAULT — **64V mode**

Reference has been made to an argument or instruction not in memory. The two usual causes of this are an incomplete load (unsatisfied references), or incomplete argument list in a subroutine or function call.

POINTER MISMATCH — **PRIMOS**

Internal file pointers have become corrupted. No user remedial action possible. System Administrator must correct. [PC, DC, AC]

PROGRAM HALT AT octal-location — **R mode and 64V mode**

Program control has been lost. The program has probably written over itself or the load was incomplete (R-mode).

PRWFIL BOF — **Old file call**

Attempt by PRWFIL subroutine to access location before beginning of file. [PG]

PRWFIL EOF — **Old file call**

Attempt by PRWFIL subroutine to access location after end of file. [PE]

PRWFIL POINTER MISMATCH — **Old file call**

The internal file pointers in the PRWFIL subroutine have become corrupted.

PRWFIL UNIT NOT OPEN — **Old file call**

The PRWFIL subroutine is attempting to perform operations using a PRIMOS file unit number on which no file is open.

PTR MISMATCH — **File System**

Internal file pointers have become corrupted. No user remedial action possible. (ATCH$$, CREA$$, GPAS$$, PRWF$$, RDEN$$, SATR$$, SRCH$$, SGDR$$). [E$PTRM]. Consult system manager.

REMOTE LINE DOWN — **File System**

Remote call-in access to computer not enabled. [E$RLDN]

******RI** — **R-mode function**

Argument is too large for real-to-integer conversion. (C$12)

******RN n** — **R-mode function**

Device error or end-of-file in READ statement on FORTRAN logical unit n.

******SE** — **R-mode function**

Single precision exponent overflow.

SEG-DIR ER — **Old file call**

Error encountered in segment directory operation. [SQ]

SEGDIR UNIT NOT OPEN — **File System**

Attempt has been made to reference a segment directory which is not open. (SRCH$$) [E$SUNO]

SEM OVERFLOW — **File System**

System error [E$SEMO]

****** SIN/COS - ARGUMENT TOO LARGE** — **V-mode function**

Argument too large for SIN or COS function.

******SQ** — **R-mode function**

Negative argument in SQRT or DSQRT function.

****** SQRT - ARGUMENT<0** — **V-mode function**

Negative argument in SQRT function.

******ST n** — **R-mode function**

STOP statement n (octal) encountered during program execution.

****** STOP n** — **V-mode function**

STOP statement n (octal) encountered during program execution.

******SZ** — **R-mode function**

Attempt to divide by zero (single-precision).

TOO MANY UFD LEVELS — **File System**

Attempt to create more than 72 levels of sub-UFDs. This error occurs only on old file partitions; new file partitions have no limit on UFD levels. [E$TMUL]

UFD FULL — **Old file call**

No more room in UFD. [SK]

UFD FULL — **File System**

UFD has no room for more files and/or sub-UFD's. Occurs only in old file partitions. (CREA$$, SRCH$$) [E$FDFL]

UFD OVERFLOW **Old file call**

No more room in UFD.

UNIT IN USE **Old file call**

Attempt to open file on PRIMOS file unit already in use. [SI].

UNIT IN USE **File System**

Attempt to open file on PRIMOS file unit already in use. (SRCH$$). [E$UIUS]

UNIT NOT OPEN **Old file call**

Attempt to perform operations with a file unit number on which no file has been opened. [PD, SD]

UNIT NOT OPEN **File System**

Attempt to perform operations with a file unit number on which no file has been opened. (PRWF$$, RDEN$$, SRCH$$, SGDR$$). [E$UNOP]

UNIT OPEN ON DELETE **Old file call**

Attempt to delete file without having first closed it. [SD]

****WN n **R-mode function**

Device error or end-of-file in WRITE statement on FORTRAN logical unit n.

****XX **R-mode function**

Integer argument >32767.

#, usage in PMA 14-5
%, usage in PMA 14-5
&, ampersand character in macros 17-2
***, dummy instruction 14-5
**, initial zero 14-6
*, current location 14-6
*CMHIGH, R-mode load map entry 4-3
*CMLOW, R-mode load map entry 4-3
*HIGH, R-mode load map entry 4-3
*HIGH, SEG load map entry 5-4
*LOW, R-mode load map entry 4-3
*LOW, SEG load map entry 5-4
*PBRK, R-mode load map entry 4-3
*STACK, SEG load map entry 5-4
*START, R-mode load map entry 4-3
*START, SEG load map entry 5-4
*SYM, R-mode load map entry 4-3
*SYM, SEG load map entry 5-4
*UII, R-mode load map entry 4-4
16S address calculation flowcharts 10-11
16S summary 10-10
32R address calculation flowcharts 10-16
32R summary 10-13
32S (includes 32R when S=0) summary 10-12
32S address calculation 10-13
64R address calculation flowcharts 10-23
64R summary 10-20
64V address calculation flowcharts 10-31
64V base register relative 10-28
64V procedure relative 10-28
64V two word memory reference 10-29
:, assembler notation 15-10
=, PMA literals 14-6
@ (non-SEG mode) assembler notation 15-8

A

A left logical, ALL, (SRV) 11-51
A left rotate, ALR, (SRV) 11-51
A left shift, ALS, (SRV) 11-51
A register details 3-3
A right logical, ARL, (SRV) 11-51
A right rotate, ARR, (SRV) 11-51
A right shift, ARS, (SRV) 11-51
A (I) 12-9
A2A (SRV) 11-22
ABQ (I) 12-21
ABQ (V) 11-49
ABS 16-5
Absolute integers 17-2
AC, assembly control pseudo-operations 16-1
ACA (SRV) 11-22
Access mode 18-6
AD, address definition pseudo-operations 16-1
Add
  C-bit to A, ACA, (SRV) 11-22
  fullword, A, (I) 12-9
  halfword, AH, (I) 12-10
  L bit to L, ADLL, (V) 11-23
  L to field address, ALFA, (V) 11-15
  link to register, ADLR, (I) 12-10
  long, ADL (V) 11-22
  one to A, ALA (SRV) 11-22
  register to field address register, ARFA, (I) 12-5
  to bottom of queue, ABQ, (I) 12-21
  to bottom of queue, ABQ, (V) 11-49
  to top of queue, ATQ, (I) 12-21
  to top of queue, ATQ, (V) 11-49
  two to A, A2A (SRV) 11-22
ADD (SRV) 11-22
Adding modules to a SEG runfile 5-7
Addition operator 15-9
Address
  definition pseudo-operations, AD 16-1
  formation 18-4
  special case selection 9-18
  mode, selecting the 4-5
  pointer (AP) 9-2
  resolution, role of assembler 10-3
  resolution, role of loader 4-4, 10-3
  space, virtual 4-5
  truncation (SR) 10-2
Addresses
  relative 18-4
  symbolic, PMA 14-5
Addressing mode, ADMOD, (I) 12-1
Addressing mode, ADMOD, (V) 11-1
ADL (V) 11-22
ADLL (V) 11-23
ADLR (I) 12-10
ADMOD (I) 12-1
ADMOD (V) 11-1
Advanced debugging techniques 7-2
Advanced features in loader, use of 4-4
Advanced SEG features 5-7
AH (I) 12-10
ALA (SRV) 11-22
ALFA (V) 11-15
ALL (SRV) 11-51
ALR (SRV) 11-51
ALS (SRV) 11-51
Ampersand character in macros (&) 17-2
ANA (SRV) 11-32
AND fullword, N, (I) 12-15
AND halfword, NH, (I) 12-15
AND long, ANL, (V) 11-32
AND to A, ANA, (SRV) 11-32
ANL (V) 11-32
AP, argument transfer template 9-7
AP, pseudo-operation 16-7
APPLIB, system library 8-1
Application library 8-1
ARFA (I) 12-5
ARGT (I) 12-20
ARGT (V) 11-43
Argument
  identifiers, macro 17-2, 17-3, 17-4
  pointer pseudo-operation, AP 16-7
  references, macro 17-2, 17-4
  substitution, macro 17-3
  transfer template, AP 9-7
  transfer, ARGT, (I) 12-20
  transfer, ARGT, (V) 11-43
  value expressions, macro 17-3
  values in parentheses, macro 17-3
  values, macro 17-1, 17-3, 17-4
Arguments, macro 17-2
Arithmetic instruction register usage (I-mode only) 11-10
Arithmetic operators 15-8
ARL (SRV) 11-51
ARR (SRV) 11-51
ARS (SRV) 11-51
ASCII 18-5
  character set B-1
  character strings 15-5
  constants 18-5
Assembler
  attribute references 17-2
  attributes 15-11, A-1
  error messages C-1
  formats (I) 14-7
  messages 3-2
Assembly control pseudo-operations, AC 16-1
Asterisk (current location), PMA 14-6
Asterisk, double, PMA 14-6
ATQ (I) 12-21
ATQ (V) 11-49
Attributes, assembler 15-11, 17-2, A-1

B

BACK pseudo-operation 16-8
Base area problems, how to resolve 4-5, 4-6, 5-8
Base areas, R-mode load map description 4-4
Base areas, SEG load map entry 5-6
Base register relative, memory refer-reference instruction formats 10-6
Base registers 9-12
Base registers, PMA formats 14-6
BCEQ (I) 12-3
BCEQ (V) 11-2
BCGE (I) 12-3
BCGE (V) 11-2
BCGT (I) 12-3
BCGT (V) 11-2
BCI pseudo operation 16-10
BCLE (I) 12-3
BCLE (V) 11-2
BCLT (I) 12-3
BCLT (V) 11-2
BCNE (V) 11-2

BCNE (I) 12-3
BCR (I) 12-3
BCR (V) 11-3
BCS (I) 12-3
BCS (V) 11-3
BDI1 (I) 12-3
BDX (V) 11-4
BDY (V) 11-4
Begin macro definition pseudo-operation, MAC 16-16
BEQ (V) 11-4
BES pseudo-operation 16-20
BFEQ (I) 12-2
BFEQ (V) 11-4
BFGE (I) 12-2
BFGE (V) 11-4
BFGT (V) 11-4
BFGT (I) 12-2
BFLE (V) 11-4
BFLE (I) 12-2
BFLT (I) 12-2
BFLT (V) 11-4
BFNE(I) 12-2
BFNE (V) 11-4
BGE (V) 11-4
BGT (V) 11-4
BHD1 (I) 12-3
BHD2 (I) 12-3
BHD4 (I) 12-3
BHGE (I) 12-2
BHGT (I) 12-2
BHI2 (I) 12-3
BHI4 (I) 12-3
BHLE (I) 12-2
BHLT (I) 12-2
BHNE (I) 12-2
Binary 18-5
  constants 15-3
  exponent 15-5
  fraction 15-5
  point 15-3, 15-4
  scaling 15-3, 15-5
  to decimal conversion, XBTD, (I) 12-5
BIX, increment X and branch if not zero, (V) 11-4
BIY, increment Y and branch if not zero, (V) 11-4
BLE, branch if A register less than or equal to zero, (V) 11-4
BLEQ, branch if L register equal to zero, (V) 11-4
BLGE, branch if L register greater than or equal to zero - (V) 11-4
BLGT, branch if L register greater than zero, (V) 11-4
BLLE, branch if L register less than or equal to zero, (V) 11-4
BLLT, branch if L register less than zero, (V) 11-4
BLNE, branch if L register not equal to zero, (V) 11-4
Block allocation 16-20
BLR, branch if L-bit reset, (I) 12-3
BLR, branch if L-bit reset, (V) 11-3
BLS, branch if L-bit set, (I) 12-3
BLS, branch if L-bit set, (V) 11-3
BLT, branch if A register less than zero, (V) 11-4
BMEQ, branch if magnitude is equal to zero, (I) 12-3
BMEQ, branch if magnitude is equal to zero, (V) 11-3
BMGE, branch if magnitude is greater than or equal to zero, (I) 12-3
BMGE, branch if magnitude is greater than or equal to zero, (V) 11-3
BMGT, branch if magnitude is greater than zero, (V) 11-3
BMGT, branch if magnitude is greater than zero, (I) 12-3
BMLE, branch if magnitude is less than or equal to zero, (I) 12-3
BMLE, branch if magnitude is less than or equal to zero, (V) 11-3
BMLT, branch if magnitude is less than zero, (I) 12-3
BMLT, branch if magnitude is less than zero, (V) 11-3
BMNE, branch if magnitude is not equal to zero, (I) 12-3
BMNE, branch if magnitude is not equal to zero, (V) 11-3
BNE, branch if A register not equal to zero, (V) 11-4
BRAN, branch (I) 12-1
BRAN, branch (V) 11-2
Branch instruction format 9-16
Branch if
  A register equal to zero, BEQ, (V) 11-4
  A register greater than or equal to zero, BGE, (V) 11-4
  A register greater than zero, BGT, (V) 11-4
  A register less than or equal to zero, BLE, (V) 11-4
  A register less than zero, BLT, (V) 11-4
  A register not equal to zero, BNE, (V) 11-4
  C-bit reset (equals zero), BCR, (V) 11-3
  C-bit reset, BCR, (I) 12-3
  C-bit set, BCS, (I) 12-3
  C-bit set, BCS, (V) 11-3
  condition code equal, BCEQ, (V) 11-2
  condition code equal, BCEQ, (I) 12-3
  condition code greater than or equal BCGE, (V) 11-2
  condition code greater than or equal, BCGE, (I) 12-3
  condition code greater than, BCGT, (V) 11-2
  condition code greater than, BCGT, (I) 12-3
  condition code less than or equal, BCLE, (V) 11-2
  condition code less than or equal, BCLE, (I) 12-3
  condition code less than BCLT, (V) 11-2
  condition code less than, BCLT, (I) 12-3
  condition code not equal, BCNE, (V) 11-2
  condition code not equal, BCNE, (I) 12-3
  floating register equal to zero, BFEQ, (I) 12-2
  floating register equal to zero, BFEQ, (V) 11-4
  floating register greater than or equal to zero, BFGE, (I) 12-2
  floating register greater than or equal to zero, BFGE, (V) 11-4
  floating register greater than zero, BFGT, (I) 12-2
  floating register greater than zero, BFGT, (V) 11-4
  floating register less than or equal to zero, BFLE, (I) 12-2
  floating register less than or equal to zero, BFLE, (V) 11-4
  floating register less than zero, BFLT, (I) 12-2
  floating register less than zero, BFLT, (V) 11-4
  floating register not equal to zero, BFNE, (I) 12-2
  floating register not equal to zero, BFNE, (V) 11-4
  half register greater than or equal to zero, BHGE, (I) 12-2
  half register greater than zero, BHGT, (I) 12-2
  half register less than or equal to zero, BHLE, (I) 12-2
  half register less than zero, BHLT, (I) 12-2
  half register not equal to zero, BHNE, (I) 12-2
  L register equal to zero, BLEQ, (V) 11-4
  L register greater than or equal to zero, BLGE, (V) 11-4
  L register greater than zero, BLGT, (V) 11-4
  L register less than or equal to zero, BLLE, (V) 11-4
  L register less than zero, BLLT, (V) 11-4
  L register not equal to zero, BLNE, (V) 11-4
  L-bit reset, BLR, (V) 11-3
  L-bit reset, BLR, (I) 12-3
  L-bit set, BLS, (V) 11-3
  L-bit set, BLS, (I) 12-3
  magnitude is equal to zero, BMEQ, (I) 12-3
  magnitude is equal to zero, BMEQ, (V) 11-3
  magnitude is greater than or equal to zero, BMGE, (I) 12-3
  magnitude is greater than or equal to zero, BMGE, (V) 11-3
  magnitude is greater than zero, BMGT, (I) 12-3

magnitude is greater than zero, BMGT, (V) 11-3
magnitude is less than or equal to zero, BMLE, (I) 12-3
magnitude is less than or equal to zero, BMLE, (V) 11-3
magnitude is less than zero, BMLT, (I) 12-3
magnitude is less than zero, BMLT, (V) 11-3
magnitude is not equal to zero, BMNE, (I) 12-3
magnitude is not equal to zero, BMNE, (V) 11-3
register bit reset, BRBR, (I) 12-2
register bit set, BRBS, (I) 12-2
register equals zero, BREQ, (I) 12-2
register greater than or equal to zero, BRGE, (I) 12-2
register greater than zero, BRGT, (I) 12-2
register less than zero, BRLT, (I) 12-2
register less than or equal to zero, BRLE, (I) 12-2
register not equal to zero, BRNE, (I) 12-2
Branch, BRAN (I) 12-1
Branch, BRAN, (V) 11-2
BRBR (I) 12-2
BRBS (I) 12-2
BRD1 (I) 12-3
BRD2 (I) 12-3
BRD4 (I) 12-3
BREQ (I) 12-2
BRGE (I) 12-2
BRGT (I) 12-2
BRI1 (I) 12-3
BRI2 (I) 12-3
BRI4 (I) 12-3
Bringing a program into memory 6-1
BRLE (I) 12-2
BRLT (I) 12-2
BRNE (I) 12-2
BSS pseudo-operation 16-20
BSZ pseudo-operation 16-20

C

C, compare fullword, (I) 12-10
C-bit 9-14
C64R, check 64R pseudo-operation 16-5
CA, conditional assembly pseudo-operations 16-1
CAI (I) 12-14
CAI (SRV) 11-29
CAL (SRV) 11-7
CALF stack frame header 9-5
CALF (I) 12-20
Call conventions (SR) 8-2
Call conventions (VI) 8-3
Call fault handler, CALF, (I) 12-20
CALL macro 17-1, 17-2
Call recursive entry procedure, CREP, (R) 11-44
CALL pseudo-operation 16-18
CAR (SRV) 11-7
CAS (SRV) 11-23
CAZ (SRV) 11-23
CEA (SRV) 11-44
CENT pseudo-operation 16-14
CGT (I) 12-3
CGT (V) 11-4
CH (I) 12-10
Change signs, CHS, (I) 12-10
Change sign, CHS, (SRV) 11-23
CHAR (V) 11-5
CHAR (I) 12-3
Character 9-14
operations, CHAR (V) 11-5
(ASCII) constants 15-5
edit, ZED, (I) 12-4
operations, CHAR (I) 12-3
CHS (I) 12-10
CHS (SRV) 11-23
Class bits in R-mode long reach instruction format 10-7
Class bits in R-mode stack instruction format 10-6, 10-7
Clear
A left byte, CAL, (SRV) 11-7
A right byte, CAR, (SRV) 11-7
A register, CRA (SRV) 11-7
active interrupt, CAI, (I) 12-14
active interrupt, CAI, (SRV) 11-29
B register, CRB (SRV) 11-8
E, CRE, (V) 11-8
high byte 1 left, CRBI, (I) 12-4
high byte 2 right, CRBR, (I) 12-4
L and E, CRLE, (V) 11-8
left halfword, CRHL, (I) 12-5
long, CRL, (SRV) 11-8
machine check, RMC, (I) 12-14
machine check, RMC, (SRV) 11-28
register and memory, CLEAR (I) 12-4
register, CLEAR (V) 11-7
register, CR, (I) 12-4
right halfword, CRHR, (I) 12-5
Clear (V) 11-7
Clear (I) 12-4
Clearing memory with FILMEM 4-1
CLS (V) 11-23
CMA (SRV) 11-32
CMDNC0, command UFD 6-2
CMH (I) 12-15
CMR (I) 12-15
Colon, assembler notation 15-10
COMM pseudo-operation 15-8
COMM, FORTRAN compatible COMMON pseudo-operation 16-20
Command files, loader subcommands in 4-1
Command line
format 18-4
operands 18-5
options, PMA 3-1
Command summary, SEG 5-8
Command UFD, CMDNC0 6-2
Command UFD, installation in the 6-2
Commands, SEG level 5-9
Comments, PMA 14-4
COMMON
block descriptions in R-mode load maps 4-4
blocks 5-6
blocks, SEG load map entry 5-6
data area 15-8
locating with SEG 5-8
COMOUTPUT files 7-2
Compare
A with skip, CAZ, (SRV) 11-23
A with zero, CAZ, (SRV) 11-23
character field, ZCM, (I) 12-4
character field, ZCM, (V) 11-5
CLS, (V) 11-23
fullword, C, (I) 12-10
halfword, CH, (I) 12-10
Complement
A, CMA, (SRV) 11-32
Complement half register, CMH, (I) 12-15
Complement register, CMR, (I) 12-15
Complement, FCM, (RV) 11-19
Compute effective address, CEA, (SRV) 11-44
Computed GOTO, CGT, (I) 12-3
Computed GOTO, CGT, (V) 11-4
Concordance (cross reference) 3-1, 3-5
Condition code bits 9-14
Condition code test 11-33
Conditional assembly 17-6
Conditional assembly pseudo-operations, CA 16-1
Constant expressions 18-5, 18-6
Constants 15-1, 15-8, 18-5
integer 15-1
numeric 15-1, 15-2
PMA 14-4
Control extended control store, CXCS, (I) 12-17
Control extended control store, CXCS, (V) 11-34
Control word format 11-10
Conventions
filename 2-1
instruction summary and description 2-2
Prime 2-1
sign 15-9
space 15-9
text 2-1
Convert
31-bit integer to float, FLOT, (R) 11-19
binary to decimal, XBTD, (V) 11-11
float to integer, INT, (V) 11-21
float to integer, INTA, (V) 11-21
float to long integer, INTL, (V) 11-22
floating point to halfword integer, INTH, (I) 12-8
floating point to integer, INT, (I) 12-8
halfword integer to floating point, FLTH, (I) 12-7
integer to float, FLTA,

(V) 11-20
integer to floating point, FLT, (I) 12-7
long integer to float, FLTL, (V) 11-20
single to double float, FDBL, (V) 11-19
single to double, DBLE, (I) 12-8
Copy sign of A, CSA, (SRV) 11-24
Copy sign, CSR, (I) 12-10
CR (I) 12-4
CRA (SRV) 11-7
CRB (SRV) 11-8
CRBI (I) 12-4
CRBR (I) 12-4
CRE (V) 11-8
Creating a system command 6-2
CREP (R) 11-44
CRHL (I) 12-5
CRHR (I) 12-5
CRL (SRV) 11-8
CRLE (V) 11-8
Cross-reference listing (concordance) 3-5
CSA (SRV) 11-24
CSR (I) 12-10
CXCS (I) 12-17
CXCS (V) 11-34

D

D (I) 12-10
D16S pseudo-operation 16-14
D32I pseudo-operation 16-14
D32R pseudo-operation 16-14
D32S pseudo-operation 16-14
D64R, R-mode load command 4-10
D64R pseudo-operation 16-14
D64V pseudo-operation 16-14
DAC pseudo-operation 16-7
DAD (SRV) 11-24
Data
constants 15-1
definition pseudo-operations, DD 16-1
structures, mode usage 2-4
structures, useful in debugging 7-1
types, decimal 11-8
DATA pseudo-operation 16-10
DBL (SRV) 11-24
DBLE (I) 12-8
DD, data definition pseudo-operations 16-1
DDM pseudo-operation 16-14
Debugging 7-1
access violation message 7-1
advanced techniques 7-2
floating exception (FLEX) 7-1
illegal Segno message 7-1
interactive programs 18-1
MO-memory overflow 7-1
no vector message 7-1
pointer fault message 7-1
PRIMOS severe errors 7-2
severe PRIMOS errors 7-1
stack overflow 7-1
using COMOUTPUT files 7-1
using the PM command 7-1
utilities 7-1
DEC pseudo-operation 16-10
DECI 11-8, 12-5
Decimal 9-2, 9-14, 18-5
add, XAD, (I) 12-5
add, XAD, (V) 11-10
arithmetic, DEC 11-8
arithmetic, DECI (I) 12-5
compare, XCM, (I) 12-5
compare, XCM, (V) 11-11
constants 15-1
constants, fixed point 15-3
constants, floating point 15-5
control word format (VI) 9-2
data types 11-8
divide, XDV, (I) 12-5
divide, XDV, (V) 11-12
exception (CEX) 11-10
exponent 15-5
fraction 15-5
integer 15-5
move, XMV, (I) 12-5
move, XMV, (V) 11-14
multiply, XMP, (I) 12-5
multiply, XMP, (V) 11-13
Decimal to binary conversion, XCTB, (I) 12-5
Decimal to binary conversion, XCTB, (V) 11-12
Declare stack relative pseudo-operation, DYNM 16-21
Decrement
and replace X, DRX, (SRV) 11-53
half register by 1, DH1, (I) 12-11
half register by 2 and branch, BHD2, (I) 12-3
half register by 2, DH2, (I) 12-11
half register by 4 and branch, BHD4, (I) 12-3
memory fullword, DM, (I) 12-11
memory halfword, DMH, (I) 12-11
register by 1 and branch, BRD1, (I) 12-3
register by 1, DR1, (I) 12-11
register by 2 and branch, BRD2, (I) 12-3
register by 2, DR2, (I) 12-11
register by 4 and branch, BRD4, (I) 12-3
X and branch if not zero, BDX, (V) 11-4
Y and branch if not zero, BDY, (V) 11-4
Definition, macro 17-1, 17-2, 17-4
Deleting SEG runfiles 5-7
DEX, decimal exception 11-10
DFA (I) 12-8
DFAD (RV) 11-17
DFC (I) 12-8
DFCM (I) 12-9
DFCM (RV) 11-17
DFCS (RV) 11-17
DFD (I) 12-9
DFDV (RV) 11-17
DFL (I) 12-9
DFLD (RV) 11-18
DFLX (V) 11-18
DFM (I) 12-9
DFMP (RV) 11-18
DFS (I) 12-9
DFSB (RV) 11-18
DFST (I) 12-9
DFST (RV) 11-18
DFTB pseudo-operation 16-8
DFVT pseudo-operation 16-8
DH (I) 12-10
DH1 (I) 12-11
DH2 (I) 12-11
Direct entry calls 5-6
Direct entry links, SEG load map entry 5-6
Displacement field meaning
base register relative 10-6
basic format 10-4
procedure relative format 10-5
sector relative format 10-4
two word memory reference (V-mode) 10-8
DIV (SR) 11-24
DIV (V) 11-24
Divide
fullword, D, (I) 12-10
halfword, DH, (I) 12-10
long, DVL, (V) 11-25
Divide, DIV, (SR) 11-24
Divide, DIV, (V) 11-24
DLD (SR) 11-39
DM (I) 12-11
DMH (I) 12-11
Double add, DAD, (SR) 11-24
Double asterisk (initial zero) PMA 14-6
Double floating
add, DFA, (I) 12-8
compare, DFC, (I) 12-8
complement, DFCM, (I) 12-9
divide, DFD, (I) 12-9
load, DFL, (I) 12-9
multiply, DFM, (I) 12-9
store, CFST, (I) 12-9
subtract, DFS, (I) 12-9
Double load, DLD, (SR) 11-39
Double precision 15-1
64 bits, floating point 9-2
floating add, CFAD, (RV) 11-17
floating complement, DFCM, (RV) 11-17
floating divide, DFDV, (RV) 11-17
floating load index, DFLX, (V) 11-18
floating load, DFLD, (RV) 11-18
floating multiply, DFMP, (RV) 11-18
floating point 15-5
floating point compare and skip, DFCS, (RV) 11-17
floating point register 9-12
floating store, DFSB, (RV) 11-18
floating subtract, DFSB, (RV) 11-18
Double subtract, DSB, (SRV) 11-24
DR1 (I) 12-11

DR2 (I) 12-11
DRX (SRV) 11-53
DSB (SR) 11-24
DUII pseudo-operation 16-15
Dummy instruction, triple asterisk, PMA 14-5
Dummy words 17-1, 17-2, 17-4
DVL (V) 11-25
DYNM pseudo-operation 15-8, 16-21
DYNT pseudo-operation 16-19

E

E16S (I) 12-1
E16S (SRV) 11-1
E32I (I) 12-1
E32I (SRV) 11-1
E32R (I) 12-1
E32R (SRV) 11-1
E32S (I) 12-1
E32S (SRV) 11-1
E64R (I) 12-1
E64R (SRV) 11-1
E64V (I) 12-1
E64V (SRV) 11-1
EAA (R) 11-44
EAFA (I) 12-6
EAFA (V) 11-15
EAL (V) 11-44
EALB (V) 11-45
EALB (I) 12-19
EAR (I) 12-19
EAXB (I) 12-20
EAXB (V) 11-45
ECB
- description in load map 5-6
- entry control block 9-6
- pseudo-operation 16-19

Edit
- character field, ZED, (V) 11-5
- program word 11-6
- sub-operations 11-14

Effective address formation 10-1
Effective address formation (PSD and VPSD only) 18-4
Effective address to
- A register, EAA, (R) 11-44
- field address register, EAFA, (V) 11-15
- L register, EAL, (V) 11-44
- link base, EALB, (I) 12-19
- link base, EALB, (V) 11-45
- register, EAR, (I) 12-19
- temporary base, EAXB, (I) 12-20
- XB, EAXB, (V) 11-45

EIO (I) 12-14
EIO (V) 11-29
EJCT pseudo-operation 16-11
Elements of PMA 14-4
ELM pseudo-operation 16-15
ELSE pseudo-operation 16-8
EMCM (I) 12-14
EMCM (SRV) 11-28
Enable interrupts, ENB, (SRV) 11-30
Enable interrupts, ENB, (I) 12-14
ENB (SRV) 11-30
ENB (I) 12-14
END pseudo-operation 16-5
ENDC pseudo-operation 16-9
ENDM pseudo-operation 16-16, 17-2
ENT pseudo-operation 16-20
Enter
- 16S mode, E16S, (SRV) 11-1
- 16S mode, E16S, (I) 12-1
- 32I mode, E32I, (I) 12-1
- 32I mode, E32I, (SRV) 11-1
- 32R mode, E32R, (I) 12-1
- 32R mode, E32R, (SRV) 11-1
- 32S mode, E32S, (I) 12-1
- 32S mode, E32S, (SRV) 11-1
- 64R mode, E64R, (I) 12-1
- 64R mode, E64R, (SRV) 11-1
- 64V mode, E64V, (I) 12-1
- 64V mode, E64V, (SRV) 11-1
- double precision mode-DBL, (SR) 11-24
- loader mode pseudo-operation, ELM 16-15
- machine check mode, EMCM, (I) 12-14
- machine check mode, EMCM, (SRV) 11-28
- paging mode and jump (Prime 300), EPMJ, (R) 11-34
- paging mode and jump to XCS (Prime 300), EPMX, (SR) 11-34
- R-mode recursive procedure stack, ENTR, (R) 11-45
- restricted execution mode and jump to XCS (Prime 300), ERMX, (SR) 11-35
- single precision mode, SGL, (SR) 11-27
- standard interrupt mode, ESIM, (SRV) 11-30
- standard interrupt mode, ESIM, (I) 12-14
- vector interrupt mode, EVIM, (I) 12-14
- vectored interrupt mode, EVIM, (SRV) 11-30
- virtual mode and jump (Prime 300), EVMJ, (SR) 11-35
- virtual mode and jump (Prime 300), EVMX, (SR) 11-35

ENTR (R) 11-45
Entry control block, ECB 9-6
EPMJ (SR) 11-34
EPMX (SR) 11-34
EQU pseudo-operation 16-21
Equals sign (literals) 14-6
ERA (SRV) 11-32
ERL (V) 11-32
ERMX (SR) 11-35
Error messages
- PMA 7-1
- run-time 6-5

Errors (system), SEG's action 5-1
Escape character assembler notation 15-10
ESIM (I) 12-14
ESIM (SRV) 11-30
- pseudo-operation 16-5

EVIM (I) 12-14
EVIM (SRV) 11-30
EVMJ (SR) 11-35
EVMX (SR) 11-35
Examples, PMA 14-9
Exception codes, floating point 11-6
Exceptions, floating point 11-16, 12-6
Excess-128 notation 15-5
Exchange and clear the A register, XCA, (SRV) 11-43
Exchange and clear the B register, XCB, (SRV) 11-43
Exclamation mark, assembler notation 15-10
Exclusive OR
- fullword, X, (I) 12-15
- halfword, XH, (I) 12-16
- long, ERL, (V) 11-32
- to A, ERA, (SRV) 11-32

Execute I/O, EIO, (I) 12-14
Execute I/O, EIO, (V) 11-29
Execute, XEC, (RV) 11-49
Executing PMA programs 6-1
Execution of segmented runfiles 6-2
Execution of unsegmented runfiles 6-1
Exponent
- binary 15-5
- decimal 15-5

Expressions 15-1, 15-5, 15-8, 15-10, 18-5
- constant 18-5
- PMA 14-4
- EXT pseudo-operation 15-8, 16-19

F

FA (I) 12-6
FAD (R) 11-18
FAIL pseudo-operation 16-9
Fault pointers 5-6
FC (I) 12-7
FCM (RV) 11-19
FCM (I) 12-7
FCS (RV) 11-19
FD (I) 12-7
FDBL (V) 11-19
FDV (RV) 11-19
Field address and length registers 9-12
Field operations, FIELD, (I) 12-5
Field operations, FIELD, (V) 11-15
FIELD (I) 12-5
FIELD (V) 11-15
File types, PMA 3-1
Filename conventions 2-1
Fill character field, ZFH, (I) 12-4
Fill field, ZFIL, (V) 11-6
FILMEM, PRIMOS clear memory command 4-1
FIN pseudo-operation 16-12
FIN, use of 15-10
Fixed point decimal constants 15-3, 15-4
FL (I) 12-7
FLD (RV) 11-19
Floating
- add, FA, (I) 12-6
- add, FAD, (RV) 11-18
- compare and skip, FCS, (RV) 11-19
- compare, FC, (I) 12-7

complement, FCM, (I) 12-7
divide, FD, (I) 12-7
divide, FDV, (RV) 11-19
load index, FLX, (RV) 11-20
load, FL, (I) 12-7
load, FLD, (RV) 11-19
multiply, FM, (I) 12-7
multiply, FMP, (RV) 11-20
point arithmetic, FLPT (I) 12-6
point arithmetic, FLPT (V) 11-16
point decimal constants 15-5
point exception codes 11-16
point exceptions 11-16, 12-6
point mantissa and exponent ranges 11-17
point register, double precision 9-12
point register, single precision, (RVI) 9-10
point registers, 9-12
point, double precision 15-5
point, double precision 64 bits 9-2
point, single precision 15-5
round, FRN, (I) 12-7
skip if greater than zero, FSCT, (RV) 11-20
skip if less than or equal to zero, FSLE, (RV) 11-20
skip if minus, FSMI, (RV) 11-21
skip if not zero-FSNZ, (RV) 11-21
skip if plus, FSPL, (RV) 11-21
skip if zero, FSZE, (RV) 11-21
store, FST, (I) 12-8
store, FST, (RV) 11-21
subtract, FS, (I) 12-8
subtract, FSB, (RV) 11-20

FLOT (R) 11-19
FLPL (RV) 11-21
FLPT, floating point arithmetic 11-16, 12-6
FLT (I) 12-7
FLTA (V) 11-20
FLTH (I) 12-7
FLTL (V) 11-20
FLX (RV) 11-20
FM (I) 12-7
FMP (RV) 11-20
Force loading 4-6
Format specifier 18-4
Formats
assembler (I-mode), PMA 14-7
assembler, PMA 14-7
instruction 9-14
instruction, PMA 14-6
FORTRAN libraries 8-1
Fraction, binary 15-5
FRN (I) 12-7
FRN (RV) 11-20
FS (I) 12-8
FSB (RV) 11-20
FSCT (RV) 11-20
FSLE (RV) 11-20
FSMI (RV) 11-21
FSNZ (RV) 11-21
FST (I) 12-8
FST (RV) 11-21
FSZE (RV) 11-21
FTNLIB library, use with SEG 5-3
FTNLIB, system library 8-1

G

General data structures, mode usage 2-4
General registers, 32 bits 9-10
Generic
AP instruction formats 9-16
instruction formats 9-14
non-register instruction formats 9-17
register instruction formats 9-17
GO pseudo-operation 16-9

H

Halt, HLT, (I) 12-17
Halt, HLT, (SRV) 11-35
Header, stack segment 9-4
HEX pseudo-operation 16-11
Hexadecimal 18-5
Hexadecimal constants 15-2
HLT, halt, (I) 12-17
HLT, halt, (SRV) 11-35
HPSD 18-3

I

I-mode instructions
A 12-9
ABQ 12-21
ADLR 12-10
AH 12-10
ARFA 12-5
ARGT 12-20
ATQ 12-21
BCEQ 12-3
BCGE 12-3
BCGT 12-3
BCLE 12-3
BCLT 12-3
BCNE 12-3
BCR 12-3
BCS 12-3
BFEQ 12-2
BFGE 12-2
BFGT 12-2
BFLE 12-2
BFLT 12-2
BFNE 12-2
BHD1 12-3
BHD2 12-3
BHD4 12-3
BHGE 12-2
BHGT 12-2
BHI1 12-3
BHI2 12-3
BHI4 12-3
BHLE 12-2
BHLT 12-2
BHNE 12-2
BLR 12-3
BLS 12-3
BMGE 12-3
BMGT 12-3
BMLE 12-3
BMLT 12-3
BMNE 12-3
BMEQ 12-3
BRBR 12-2
BRBS 12-2
BRD1 12-3
BRD2 12-3
BRD4 12-3
BREQ 12-2
BRGE 12-2
BRGT 12-2
BRI1 12-3
BRI2 12-3
BRI4 12-3
BRLE 12-2
BRLT 12-2
BRNE 12-2
C 12-10
CAI 12-14
CALF 12-20
CGT 12-3
CH 12-10
CHS 12-10
CMH 12-15
CMR 12-15
CR 12-4
CRBI 12-4
CRBR 12-4
CRHL 12-5
CRHR 12-5
CSR 12-10
CXCS 12-17
D 12-10
DBLE 12-8
DFA 12-8
DFC 12-8
DFCM 12-9
DFD 12-9
DFL 12-9
DFM 12-9
DFS 12-9
DFST 12-9
DH 12-10
DH1 12-11
DH2 12-11
DM 12-11
DMH 12-11
DR2 12-11
DR1 12-11
E16S 12-1
E32I 12-1
E32R 12-1
E32S 12-1
E64R 12-1
E64V 12-1
EAFA 12-6
EALB 12-19
EAR 12-19
EAXB 12-20
EIO 12-14
EMCM 12-14
ENB 12-14
ESIM 12-14
EVIM 12-14
FA 12-6
FC 12-7
FCM 12-7
FD 12-7
FL 12-7
FLT 12-7
FLTH 12-7
FM
FRN 12-7
FS 12 8
FST 12-8

HLT 12-17
I 12-17
ICBL 12-17
ICBL 12-18
IBCR 12-17
ICHR 12-18
IH 12-18
IH2 12-11
IH1 12-11
IM 12-12
IMH 12-12
INBN 12-20
INEC 12-20
INEN 12-20
INH 12-14
INK 12-14
INT 12-8
INTH 12-8
IR1 12-12
IR2 12-12
IRB 12-18
IRH 12-18
IRTC 12-14
IRTN 12-14
ITLB 12-17
JMP 12-20
JSR 12-20
JSXB 12-20
L 12-18
LCEQ 12-16
LCGE 12-16
LCGT 12-16
LCLE 12-16
LCLT 12-16
LCNE 12-16
LDAR 12-18
LDC 12-4
LEG 12-16
LF 12-17
LFEQ 12-16
LFGE 12-16
LFGT 12-16
LFLE 12-16
LFLI 12-6
LFLT 12-16
LFNE 12-16
LGE 12-16
LGT 12-16
LH 12-18
LHEQ 12-16
LHGE 12-16
LHGT 12-16
LHL1 12-18
LHL2 12-19
LHLE 12-16
LHLT 12-16
LHNE 12-16
LIOT 12-17
LLE 12-16
LLT 12-16
LMCM 12-14
LNE 12-16
LPID 12-17
LPSW 12-17
LT 12-17
LWCS 12-17
M 12-12
MDEI 12-14
MDII 12-14
MDIW 12-14
MDRS 12-14
MDWC 12-14
MH 12-12
N 12-15
NFYB 12-20
NFYE 12-20
NH 12-15
NOP 12-17
O 12-15
OH 12-15
OTK 12-14
PCL 12-20
PID 12-12
PIDH 12-12
PIM 12-13
PIMH 12-13
PRTN 12-20
PTLB 12-17
RBQ 12-21
RCB 12-15
RMC 12-14
ROT 12-21
RRST 12-17
RSAV 12-17
RTQ 12-21
S 12-13
SCB 12-15
SH 12-13
SHA 12-22
SHL 12-23
SHL 12-22
SHL1 12-22
SHR1 12-23
SHR2 12-23
SL1 12-22
SL1 12-22
SR2 12-22
SRL 12-22
SSM 12-13
SSP 12-13
ST 12-19
STAR 12-19
STC 12-4
STCD 12-19
STCH 12-19
STEX 12-20
STFA 12-6
STH 12-19
STPM 12-17
SVC 12-20
TC 12-13
TCH 12-13
TFLR 12-5
TM 12-14
TMH 12-14
TSTQ 12-21
VIRY 12-14
WAIT 12-20
WCS 12-17
X 12-15
XAD 12-5
XBTD 12-5
XCM 12-5
XDTB 12-5
XED 12-5
XH 12-16
XMV 12-5
XVRY 12-14
XVRY 12-17
ZCM 12-4
ZDV 12-5
ZED 12-4
ZFH 12-4
ZM 12-5
ZMH 12-5
ZMV 12-4
ZMVD 12-4
ZTRN 12-4
I-mode
  instruction formats 9-16
  PMA 14-6
  purpose 9-16
I/O 9-14
  input/output (I) 12-14
  input/output (V) 11-29
IAB (SRV) 11-40
ICA (SRV) 11-40
ICBL (I) 12-17
ICBL (I) 12-18
ICBR (I) 12-17
ICHR (I) 12-18
ICL (SRV) 11-40
ICR (SRV) 11-40
IF pseudo-operation 16-9
If (SRV) 11-34
IFTNLB library, use with SEG 5-2
IFTNLB, system library 8-1
IFTT pseudo-operation 16-9
IFVF pseudo-operation 16-9
IFVT pseudo-operation 16-9
IFX pseudo-operation 16-9
IH halfword, (I) 12-18
IH1 (I) 12-11
IH2 (I) 12-11
ILE (V) 11-40
IM (I) 12-12
IMA (SRV) 11-40
Images saved by load 6-2
IMH (I) 12-12
Immediate
  type 1, I-mode instruction format 9-18
  type 2, I-mode instruction format 9-18
  type 3, I-mode instruction format 9-18
INA (SR) 11-30
INBC (I) 12-20
INBC (V) 11-49
INBC (V) 11-49
INBN (I) 12-20
Inclusive OR, ora, (V) 11-33
Increment
  and replace X, IRX, (SRV) 11-53
  half register by 1 and branch, BHIL, (I) 12-3
  half register by 1, IH, (I) 12-11
  half register by 2 and branch, BHI2, (I) 12-3
  half register by 2, IH1, (I) 12-11
  memory fullword, IM, (I) 12-12
  memory halfword, IMH, (I) 12-12
  memory replace and skip, IRS, (SRV) 11-53
  register by 1 and branch, BRIL, (I) 12-3
  register by 1, IR1, (I) 12-12
  register by 2 and branch, BRI2, (I) 12-3

register by 2, IR2, (I) 12–12
register by 4 and branch, BRI4, (I) 12–3
X and branch if not zero, BIX, (V) 11–4
Y and branch if not zero, BIY, (V) 11–4
Index registers 9–17, 18–4
Indexing 10–2
Indexing, PMA 14–5
Indirect
links, 32R vs. 64R 4–5
pointer pseudo-operation, IP 16–7
pointer, three word memory reference 9–3
pointer, two word memory reference 9–3
word, one word memory reference 9–3
Indirection 10–2
Indirection, PMA 14–5
INEC (I) 12–20
INEC (V) 11–49
INEN (I) 12–20
INEN (V) 11–49
INH (I) 12–14
INH (SRV) 11–30
Inhibit interrupts, INH, (SRV) 11–30
Inhibit interrupts, INH, (I) 12–14
INK (I) 12–14
INK (SRV) 11–31
Input keys, INK, (I) 12–14
Input keys, INK, (SR) 11–31
Input parameters 18–4
Input to A, INA, (SR) 11–30
Input/output formats (PSD and VPSD only) 18–4
Input/output, I/O (I) 12–14
Input/output, I/O (V) 11–29
Installation in the command UFD 6–2
Instruction
description conventions 2–1
format (SRV), memory reference, PMA 14–5
format, memory reference 10–2
format, mnemonic definitions 2–3
formats 9–14
formats, I-mode 9–16
formats, PMA 14–6
function group definitions 2–2
range, relative 10–3
range, sectored 10–3
summary and description conventions 2–2
summary chart 13–1
INT (V) 11–21
INT (I) 12–8
INT (I) 12–9
INT (V) 11–22
INTA (V) 11–21
Integer arithmetic, INT (I) 12–9
Integer arithmetic, INT (V) 11–22
Integer constants 15–1
Integer value single precision 15–5
Integers, absolute 17–2
Integers, decimal 15–5
Integrity check for hardware, INTGY (I) 12–14
Integrity check for hardware, INTGY (V) 11–28
Interactive debugging programs 18–1
Interchange
and clear left, ICL, (SRV) 11–40
and clear right, ICR, (SRV) 11–40
A and B registers, IAB, (SRV) 11–40
bytes and clear left, ICBL, (I) 12–17
bytes and clear right, ICBR, (I) 12–17
halfwords and clear left, ICBL, (I) 12–18
L and E, ILE, (V) 11–40
memory and the A register, IMA, (SRV) 11–40
register and memory, fullword, I, (I) 12–17
register and memory, halfword, IH, (I) 12–18
register bytes, IRB, (I) 12–18
register halfwords and clear right, ICHR, (I) 12–18
register halves, IRH, (I) 12–18
Interfacing with the system libraries 8–1
Interlude program in CMDNC0 6–5
Interlude programs 6–2
Interrupt notify
INBC, (I) 12–20
INBC, (V) 11–49
INBC, (V) 11–49
INBN, (I) 12–20
INEC, (I) 12–20
INEC, (V) 11–49
INEN, (I) 12–20
INEN, (V) 11–49
Interrupt return
IRTC, (I) 12–14
IRTN, (I) 12–14
INTGY (I) 12–14
INTGY (V) 11–28
INTH(I) 12–8
INTL (V) 11–22
Invalidate STLB entry, ITLB, (I) 12–17
Invalidate STLB entry, ITLB, (V) 11–35
IP pseudo-operation 16–7
IPVT pseudo-operation 16–16
IR1 (I) 12–12
IR2 (I) 12–12
IRB (I) 12–18
IRH (I) 12–18
IRS (SRV) 11–53
IRTC (I) 12–14
IRTN (I) 12–14
IRX (SRV) 11–53
ITLB (I) 12–17
ITLB (V) 11–35

J

JDX (R) 11–45
JEQ (R) 11–45
JGE (R) 11–45
JGT (R) 11–45
JIX (R) 11–46
JLE (R) 11–46
JLT (R) 11–46
JMP (I) 12–20
JMP (SRV) 11–46
JNE (R) 11–46
JSR (I) 12–20
JST (SRV) 11–46
JSX (RV) 11–47
JSXB (I) 12–20
JSXB (V) 11–47
JSY (V) 11–47
Jump and
decrement X, JCX, (R) 11–45
increment X, JIX, (R) 11–46
set XB, JSXB, (I) 12–20
set XB, JSXB, (V) 11–47
set Y, JSY, (V) 11–47
store return in X, JSX, (RV) 11–47
store, JST, (SRV) 11–46
Jump if
equal to zero, JEQ, (R) 11–45
greater than or equal to zero, JGE, (R) 11–45
greater than zero, JGT, (R) 11–45
less than or equal to zero, JLE, (R) 11–46
less than zero, JLT, (R) 11–46
not equal to zero, JNE, (R) 11–46
Jump to subroutine, JSR, (I) 12–20
Jump, JMP, (I) 12–20
Jump, JMP, (SRV) 11–46

K

Key manipulation, KEYS (I) 12–14
Key manipulation, KEYS (V) 11–31
Keys (SR) 9–12
Keys (VI) 9–12
KEYS, key manipulation (I) 12–14
KEYS, key manipulation (V) 11–31

L

L, load fullword, (I) 12–18
L-bit 9–14
Label, PMA 14–4
Labels 15–8
Language structure, PMA 14–1
LB% assembler notation 15–8
LC, listing control pseudo-operations 16–1
LCEG (V) 11–33
LCEQ (I) 12–16
LCGE (V) 11–33
LCGE (I) 12–16
LCGT (I) 12–16
LCLE (V) 11–33
LCLE (I) 12–16
LCLT (V) 11–33
LCLT (I) 12–16
LCNE (V) 11–33
LCNE (I) 12–16
LDA (SRV) 11–40
LDAR (I) 12–18

LDC (I) 12-4
LDC (V) 11-5
LDL (V) 11-40
LDLR (V) 11-41
LDX (SRV) 11-41
LDY (V) 11-41
Leave machine check mode, LMCM, (I) 12-14
Leave machine check mode, LMCM, (SRV) 11-28
Leave paging mode and jump (Prime 300), LPMJ, (SR) 11-36
LEQ (V) 11-33
LEQ (I) 12-16
LF (I) 12-17
LFEQ (I) 12-16
LFGE (I) 12-16
LFGT (I) 12-16
LFLE (I) 12-16
LFLI (I) 12-6
LFLI (V) 11-15
LFLT (I) 12-16
LFNE (I) 12-16
LGE (V) 11-33
LGE (I) 12-16
LGT (V) 11-33
LGT (I) 12-16
LH (I) 12-18
LHEQ (I) 12-16
LHGE (I) 12-16
LHGT (I) 12-16
LHL1 (I) 12-18
LHL2 (I) 12-19
LHLE (I) 12-16
LHLT (I) 12-16
LHNE (I) 12-16
LIB UFD library, use with SEG 5-3
Library
  subroutines, loading 4-3, 5-3
  files sort 8-1
  memory sort 8-1
  UII 4-4
Line format, PMA 14-3
Line terminator 18-6
Lines, PMA 14-1
Link frame description in load map 5-6
LINK pseudo-operation 16-5
Linkage area, PMA 14-9
LIOT (V) 11-35
LIOT (I) 12-17
LIR pseudo-operation 16-15
List pseudo-operation 16-11
Listing control pseudo-operations, LC 16-1
Listing
  files 3-1
  format 3-2
  page headers 3-1
  user generated messages in 3-1
Literal pool 15-10
Literal pseudo-operations LT 16-1
Literals 15-1, 15-10, 18-5
Literals, PMA 14-9
LLE (V) 11-33
LLE (I) 12-16
LLEQ (V) 11-33
LLGE (V) 11-33
LLGT (V) 11-33
LLL (SRV) 11-51
LLLE (V) 11-33
LLLT (V) 11-33
LLNE (V) 11-33
LLR (SRV) 11-52
LLS (SR) 11-52
LLS (V) 11-52
LLT (V) 11-33
LLT (I) 12-16
LMCM (I) 12-14
LMCM (SRV) 11-28
LNE (V) 11-33
LNE (I) 12-16
LO loader control pseudo-operations 16-1
LOAD, R-mode linking loader 4-1
LOAD
  addressed register, LDAR, (I) 12-18
  character, LDC, (I) 12-4
  character, LDC, (V) 11-5
  command summary, R-mode 4-8
  field address register EAFA, (I) 12-6
  field length register immediate, LFLI, (I) 12-6
  field length register immediate, LFLI, (V) 11-15
  fullword, L, (I) 12-18
  halfword left shifted by 1, LHL1, (I) 12-18
  halfword shifted by 2, LHL2, (I) 12-19
  halfword, LH, (I) 12-18
  I/O TLB, LIOT, (V) 11-35
  L from addressed register, LDRL, (V) 11-41
  long, LDL, (V) 11-40
  map entry (R-mode), *CMLOW 4-3
  map entry (R-mode), *HIGH 4-3
  map entry (R-mode), *LOW 4-3
  map entry (R-mode), *PBRK 4-3
  map entry (R-mode), *START 4-3
  maps 4-3
  maps in SEG 5-3
  procedures, basic 4-2
  process ID, LPID, (I) 12-17
  process ID, LPID, (V) 11-36
  program status word, LPSW, (I) 12-17
  program status word, LPSW, (V) 11-36
  shift count into A, SCA, (SR) 11-27
  state 4-3
  subprocessor commands, SEG 5-10
  the A register, LDA, (SRV)
  error messages 4-1
  functions 4-4
Loading
  library subroutines 4-3, 5-3
  on page boundaries 4-6
  order of (R-mode) 4-2
  order of (SEG) 5-3
  R-mode programs 4-1
  SEG mode programs 5-2
Local address definition pseudo-operation, DAC 16-7
Local labels within macros 17-2
Location pointer 18-6
Logic
  set A false, LF, (SRV) 11-34
  set A true, LT, (SRV) 11-34
  set false, LF, (I) 12-17
  set true, LT, (I) 12-17
Logical
  operations (I) 12-15
  operators 15-8
  test and set, LTSTS, (I) 12-16
  test and set, LTSTS, (V) 11-33
Long
  left logical, LLL, (SRV) 11-51
  left rotate, LLR, (SRV) 11-52
  left shift, LLS, (SR) 11-52
  left shift, LLS, (V) 11-52
  reach memory reference instruction formats 10-6
  right logical, LRL, (SRV) 11-52
  right rotate, LRR, (SRV) 11-52
  right shift, LRS, (SR) 11-52
  right shift, LRS, (V) 11-53
LPID (I) 12-17
LPID (V) 11-36
LPMJ (SR) 11-36
LPMX (SR) 11-36
LPSW (I) 12-17
LPSW (V) 11-36
LRL (SRV) 11-52
LRR (SRV) 11-52
LRS (SR) 11-52
LRS (V) 11-53
LSMD pseudo-operation 16-11
LSTM 17-6
LSTM pseudo-operation 16-11
LSTMD 17-6
LT, literal pseudo-operations 16-1
LT, (SRV) 11-34
LT (I) 12-17
LTSTS (I) 12-16
LTSTS (V) 11-33
LWCS (I) 12-17
LWCS (V) 11-37

M

M (I) 12-12
MAC pseudo-operation 16-16, 17-2
Machine control, MCTL, (I) 12-17
Machine control, MCTL, (V) 11-34
Macro
  call number 17-2
  calls 15-11, 17-1, 17-2
  definition pseudo-operations, MD 16-1
  definitions 15-11, 17-1, 17-2, 17-4, 17-5, 17-6
  facility 17-1
  listing 17-6
  nesting 17-5
  self documentation of 17-3
Map files, using 5-3

MATHLB, system library 8-1
Matrix routines 8-1
MCTL (I) 12-17
MCTL (V) 11-34
MD, macro definition pseudo-operations 16-1
MDEI (V) 11-28
MDEI (I) 12-14
MDII (V) 11-28
MDII (I)12-14
MDIW (I) 12-14
MDIW (V) 11-28
MDRS (I) 12-14
MDRS (V) 11-28
MDWC (I) 12-14
MDWC (V) 11-28
Memory addressing
  address truncation 10-1
  indexing 10-1
  indirection 10-1
  parameters 10-1
Memory diagnostic
  enable interleave, MDEI, (V) 11-28
  enable interleave, MDEI, (I) 12-14
  inhibit interleave, MDII, (V) 11-28
  inhibit interleave, MDII, (I) (I) 12-14
  read syndrome bits, MDRS, (I) 12-14
  read syndrome bits, MDRS, (V) 11-28
  write control register, MDWC, (V) 11-28
  write interleaved, MDIW, (I) 12-14
  write interleaved, MDIW, (V) 11-28
Memory organization 10-1
Memory overflow errors (MO) 7-3
Memory reference 9-17
  concepts (SRV) 10-1
  floating register, MRFR 9-17
  general register, MRGR 9-17
  instruction formats 9-16, 10-1, 10-2
  instruction formats (SRV), PMA 14-5
  instruction formats, base register relative 10-1
  instruction formats, base registers 10-1
  instruction formats, basic 10-1, 10-4
  instruction formats, long reach 10-1
  instruction formats, procedure relative 10-1
  instruction formats, sector relative 10-1
  instruction formats, stack postincrement 10-1
  instruction formats, stack predecrement 10-1
  instruction formats, stack relative 10-1
  non register, MRNR 9 17
  two word 10-8
Messages, assembler 3-2
MH (I) 12-12
MIA (V) 11-37
MIB (V) 11-37
Microcode indirect A, MIA, (V) 11-37
Microcode indirect A, MIB, (V) 11-37
MO, memory overflow errors 7-3
Modals 9-14
Mode 18-4
Mode usage, general data structures 2-4
Mode, access 18-6
Mode, resultant 15-9
Modify subprocessor commands 5-13
Move
  character field, ZMV, (I) 12-4
  character field, ZMV, (V) 11-6
  data, MOVE, (I) 12-17
  data, MOVE, (V) 11-39
  equal length fields, ZMVD, (I) 12-4
  equal length fields, ZMVD, (V) 11-7
Move (I) 12-17
Move (V) 11-39
MPL (V) 11-25
MPY (V) 11-25
MRFR, memory reference floating register 9-17
MRGR, memory reference general register 9-17
MRNR, memory reference non register 9-17
MSORTS, system library 8-1
Multiply fullword, M, (I) 12-12
Multiply halfword, MH, (I) 12-12
Multiply long, MPL, (V) 11-25
Multiply, MPY, (V) 11-25

N

N (I) 12-15
N64R pseudo-operation 16-15
Negative numbers 15-5
Negative scaling 15-4
Nested calls 17-5
Nesting macros 17-5
NFE (I) 12-20
NFYB (I) 12-20
NFYB (V) 11-49
NFYE (V) 11-49
NH (I) 12-15
NLSM pseudo-operation 16-12, 17-6
NLST pseudo-operation 16-12
No operation, NOP, (I) 12-17
No operation, NOP, (SRV) 11-37
Non-register generic instruction formats 9-17
NOP (I) 12-17
NOP (SRV) 11-37, 11-55
Normalization 11-16, 12-6
Normalize, NRM, (SR) 11-25
Normalized fraction 15-5
Not 64R pseudo-operation, N64R 16-15
Notify,
  NFYB, (I) 12-20
  NFYB, (V) 11-49
  NFYE, (I) 12-20
  NFYE, (V) 11-49
NRM (SR) 11-25
Numeric
  constants 15-1, 15-2
  edit, XEC, (I) 12-5
  edit, XED, (V) 11-13

O

O (I) 12-15
Object code, loader use of 4-5
Object file (SEG) reorganization 5-8
Object files 3-1
Object files in SEG 5-8
OCP (SR) 11-30
OCT pseudo-operation 16-11
Octal 18-5
Octal constants 15-2
OH (I) 12-15
One word memory reference, indirect word 9-3
Operand field, PMA 14-5
Operand, PMA 14-4
Operation field, PMA 14-5
Operation, PMA 14-4
Operations 15-8
Operator
  addition 15-9
  priority 15-9
  subtraction 15-9
Operators 15-8, 15-9
  arithmetic 15-8
  logical 15-8
  relational 15-9
  shift 15-8, 15-9
Optimizing program performance, SEG 5-7
Optimizing program size, SEG 5-7
Options, PMA command line 3-1
OR fullword, O, (I) 12-15
OR halfword, OH, (I) 12-15
ORA (V) 11-33
ORG pseudo-operation 16-6
Organization of memory 10-1
OTA (SR) 11-30
OTK (I) 12-14
OTK (SR) 11-31
Out from A, OTA, (SR) 11-30
Output control pulse, OCP, (SR) 11-30
Output keys, OTK, (I) 12-14
Output keys, OTK, (SR) 11-31
Output values 18-4

P

Page headers listing 3-1
Parameter passing (SR) 8-1
Parameter passing (VI) 8-1
Pathname, use of in SEG 5-1, 5-8
Pathnames, use of 4-5
PB% assembler notation 15-8
PCL stack frame header 9-4
PCL (I) 12-20
PCL (V) 11-47
PCTLJ (I) 12-19
PCTLJ (V) 11-43
PCVH pseudo-operation 16-12
Percent sign, PMA 14-5
PFTNLB library, use with

SEG 5-2
PFTNLB, system library 8-1
PID (I) 12-12
PID (SR) 11-26
PIDA (V) 11-26
PIDH (I) 12-12
PIDL (V) 11-26
PIM (I) 12-13
PIM (SR) 11-26
PIMA (V) 11-26
PIMH (I) 12-13
PIML (V) 11-26
PL, program linking pseudo-operations 16-1
PM 18-3
PM command 7-1
PMA concepts and facilities
asterisk (current location) 14-6
asterisk, double (initial zero) 14-6
asterisk, triple (dummy instruction) 14-5
base registers 14-6
code 14-9
command line options 3-1
comments 14-4
constants 14-4
elements 14-4
error messages 7-1, C-1
examples 14-9
expressions 14-4
file types 3-1
formats 14-7
I-mode 14-6
indexing 14-5
indirection 14-5
instruction formats 14-6
label 14-4
language structure 14-1
line format 14-3
lines 14-1
linkage area 14-9
literals 14-9
memory reference instruction format (SRV) 14-5
operand 14-4
operand field 14-5
operation field 14-5
percent sign 14-5
pound sign (#) 14-5
stack 14-6
stack 14-8
statements 14-1, 14-2
symbolic addresses 14-5
symbols 14-4
syntax 14-4
types 14-1
PMA V or I mode code, how to write 14-8
Position
after multiply, PIM, (I) 12-13
following integer multiply, PIM, (SR) 11-26
following integer multiply, PIMA, (V) 11-26
following integer multiply-long, PIML, (V) 11-26
for integer divide, PID, (i) 12-12 (I) 12-12
for integer divide, PID, (SR) 11-26
for integer divide, PIDA, (V) 11-26
for integer divide-long, PIDL, (V) 11-26
half register after multiply, PIMH, (I) 12-13
half register for integer divide, PIDH, (I) 12-12
Pound sign(#) assembler notation 14-5, 15-11
Powers of 10 (E) 15-4
PRCEX, process exchange (I) 12-20
PRCEX, process exchange (V) 11-49 (restricted) 11-49
Precision 15-4
double 15-1
single 15-1
Prime conventions 2-1
PRIMOS severe errors, debugging 7-2
PROC pseudo-operation 16-6
Procedure
call, PCL, (I) 12-20
call, PCL, (V) 11-47
frame description in load map 5-6
relative memory reference instruction formats 10-4
return, PRTN, (I) 12-20
return, PRTN, (V) 11-48
structure load map description 5-6
Procedures with no names, identification in load maps 5-6
Process exchange, PRCEX, (V) 11-49
Process exchange, PRCEX, (I) 12-20
Processor characteristics 9-8
Program control and jump, PCTLJ, (I) 12-19
Program control and jump, PCTLJ, (V) 11-43
Program linking pseudo-operations, PL 16-1
Program memory images saved by load 6-2
PRTN, procedure return, (I) 12-20
PRTN, procedure return, (V) 11-48
PSD (Prime symbolic debugger) 18-1, 18-3,18-4,20-1
command summary 20-1
terminators 20-1
using 18-3
PSD and VPSD input/output formats 18-4
PSD, VPSD terminators 21-1
PSD30 18-3
Pseudo-operation summary 16-2
Pseudo-operations
ABS 16-5
AP 16-7
BACK 16-8
BCI 16-10
BES 16-20
BSS 16-20
BSZ 16-20
C64R 16-5
CALL 16-18
CENT 16-14
COMM 16-20
D16S 16-14
D32I 16-14
D32R 16-14
D32S 16-14
D64R 16-14
D64V 16-14
DAC 16-7
DATA 16-10
DDM 16-14
DEC 16-10
DFTB 16-8
DFVT 16-8
DUII 16-15
DYNM 16-21
DYNT 16-19
ECB 16-19
EJCT 16-11
ELM 16-15
ELSE 16-8
END 16-5
ENDC 16-9
ENDM 16-16
ENT 16-20
EQU 16-21
EVEN 16-5
EXT 16-19
FAIL 16-9
FIN 16-12
GO 16-9
HEX 16-11
IF 16-9
IFTF 16-9
IFTT 16-9
IFVF 16-9
IFVT 16-9
IFX 16-9
IP 16-7
IPVT 16-16
LINK 16-5
LIR 16-15
LIST 16-11
LSMD 16-11
LSTM 16-11
MAC 16-16
N64R 16-15
NLSM 16-12
NLST 16-12
OCT 16-11
ORG 16-6
PCVH 16-12
PROC 16-6
REL 16-6
RLIT 16-12
SAY 16-16
SCT 16-17
SCTL 16-17
SDM 16-15
SEG, 64V 16-6
SEGR, 32I 16-6
SET 16-21
SETB 16-15
SUBR 16-20
VFD 16-11
XAC 16-7
XSET 16-21
PTLB (I) 12-17

PTLB (V) 11-37
Purge TLB, PTLB, (I) 12-17
Purge TLB, PTLB, (V) 11-37
Purpose of I-mode 9-16

Q

Question mark (?) 18-4
Queue control block (VI) 9-7
Queue management, QUEUE (I) 12-20
Queue management, QUEUE (V) 11-49
QUEUE, queue management (I) 12-20
QUEUE, queue management (V) 11-49

R

R registers 9-8
R-mode instructions
- A2A 11-22
- ACA 11-22
- ADD 11-22
- ALA 11-22
- ALL 11-51
- ALR 11-51
- ALS 11-51
- ANA 11-32
- ARL 11-51
- ARR 11-51
- ARS 11-51
- CAJ 11-29
- CAL 11-7
- CAR 11-7
- CAS 11-23
- CAZ 11-23
- CEA 11-44
- CHS 11-23
- CMA 11-32
- CRA 11-7
- CRB 11-8
- CREP 11-44
- CRL 11-8
- CSA 11-24
- DAD 11-24
- DBL 11-24
- DFAD 11-17
- DFCM 11-17
- DFCS 11-17
- DFDV 11-17
- DFLD 11-18
- DFMP 11-18
- DFSB 11-18
- DFST 11-18
- DIV 11-24
- DLD 11-39
- DRX 11-53
- DSB 11-24
- E16S 11-1
- E32I 11-1
- E32R 11-1
- E32S 11-1
- E64R 11-1
- E64V 11-1
- EAA 11-44
- EMCM 111-28
- ENB 11-39
- EBTR 11-45
- EPMJ 11-34
- ERA 11-32
- ERMX 11-35
- ESIM 11-30
- EVIM 11-30
- EVMJ 11-35
- EVMX 11-35
- EXB 11-43
- FAD 11-18
- FCM 11-19
- FCS 11-19
- FCST 11-20
- FDV 11-19
- FLD 11-19
- FLOT 11-19
- FMP 11-20
- FRN 11-20
- FSB 11-20
- FSLE 11-20
- FSMI 11-21
- FSNZ 11-21
- FSPL 11-21
- FST 11-21
- FSZE 11-21
- HLT 11-35
- IAB 11-40
- ICL 11-40
- ICR 11-40
- IMA 11-40
- INA 11-30
- INH 11-30
- INK 11-31
- IRS 11-53
- IRX 11-53
- JDX 11-45
- JED 11-45
- JGE 11-45
- JGT 11-45
- JIX 11-46
- JLE 11-46
- JLT 11-46
- JMP 11-46
- JNE 11-46
- JST 11-46
- JSX 11-47
- LDA 11-40
- LDX 11-41
- LF 11-34
- LLL 11-51
- LLR 11-52
- LLS 11-52
- LMCM 11-28
- LPMI 11-36
- LPMX 11-36
- LRL 11-52
- LRR 11-52
- LRS 11-52
- LT 11-34
- NOP 11-37
- NOP 11-55
- NRM 11-25
- OCP 11-30
- OTA 11-30
- OTK 11-31
- PID 11-26
- PIM 11-26
- RCB 11-31
- RMC 11-28
- SLA 11-26
- S2A 11-27
- SAR 11-53
- SAS 11-53
- SCA 11-27
- SCB 11-31
- SGL 11-27
- SGT 11-54
- SKP 11-54, 11-55
- SKS 11-31
- SLE 11-54
- SLN 11-55
- SLZ 11-55
- SMCR 11-29
- SMCS 11-29
- SMI 11-55
- SNR 11-54
- SNS 11-54
- SNZ 11-55
- SPL 11-55
- SR2 11-55
- SR3 11-55
- SR4 11-55
- SRC 11-55
- SRL 11-55
- SS1 11-55
- SS2 11-55
- SS3 11-55
- SS4 11-55
- SSC 11-55
- SSM 11-27
- SSP 11-27
- SSR 11-55
- SSS 11-55
- STA 11-41
- STR 11-48
- STX 11-42
- SUB 11-27
- SVC 11-39
- SZE 11-55
- TCA 11-28
- VIRY 11-29
- WCS 11-39
- XCA 11-43
- XEC 11-49

R-mode programs, Loading 4-1
RBQ (V) 11-50
RCB (I) 12-15
RCB (SRV) 11-31
Recursive entry, R-mode 11-44
Register description VI-mode 9-9
Register generic 9-17
Register, index 18-4
Register save, RSAV, (I) 12-17
Register to register instruction formats 9-17
Registers (R) 9-8
Registers (S) 9-8
Registers (VI) 9-9
Registers, base, PMA 14-6
REL pseudo-operation 16-6
Relational operators 15-9
Relative addresses 18-4
Relative base 10-6
Relative reach 10-1
Relocatable mode 18-4
Relocation constant 18-4
Remove
- from bottom of queue, RBC, (I) 12-21
- from bottom of queue, RBC, (V) 11-50
- from top of queue, RTQ, (I) 12-21
- from top of queue, RTQ, (V) 11-50

Replacing modules in a SEG

runfile 5-7
Reset C-bit (clear), RCB, (I) 12-15
Reset C-bit, RCB, (SRV) 11-31
Restore registers, RRST, (I) 12-17
Restore registers, RRST, (V) 11-38
Resultant mode 15-9
Return from R-mode recursive procedure, RTN, (SR) 11-48
RLIT pseudo-operation 15-10, 16-12
RMC (I) 12-14
RMC (SRV) 11-28
ROT (I) 12-21
Rotate, ROT, (I) 12-21
Round up, FRN, (RV) 11-20
RRST (I) 12-17
RRST (V) 11-38
RSAV (I) 12-17
RSAV (V) 11-38
RTN (SR) 11-48
RTQ (I) 12-21
RTQ (V) 11-50
Run-time error messages 6-5
Runfiles 4-5
saved by SEG's loader 6-3
segmented 5-7
segmented, advantages of 5-1
RVEC parameters 7-1
RVI, single precision, floating point register 9-10

S

S registers 9-8
S (I) 12-13
S2A (SRV) 11-27
SA, storage allocation pseudo-operations 16-1
Sample terminal session 2-5
SAR (SRV) 11-53
SAS (SRV) 11-53
Save registers, RSAV, (V) 11-38
SAY pseudo operation, usage 3-1, 16-16
SB% (seg modes) assembler notation 15-8
SBL (V) 11-27
SCA (SR) 11-27
Scale differential 9-2
Scaling, binary 15-5
Scaling, negative 15-4
SCB (I) 12-15
SCB (SRV) 11-31
SCT pseudo-operation 16-17
SCTL pseudo-operation 16-17
SD, symbol definition pseudo-operations 16-1
SDM pseudo-operation 16-15
Sector relative memory reference instruction formats 10-4
Sectors 10-1
SEG 18-3
filename 1/1 18-3
load map components 5-4
loader 5-7
VPSD 18-3
SEG level commands 5-9
SEG pseudo-operation 16-6
Segment directory subfiles 5-7
Segment usage by SEG 5-7
Segmentation 10-1
Segmented programs, loading 5-1
Segmented runfiles 5-1, 5-7, 6-2, 6-3
SEGR pseudo-operation 16-6
Selecting the address mode 4-5
Self documentation of macros 17-3
Semicolon, assembler notation 15-10
Set C-bit, SCB, (I) 12-15
Set C-bit, SCB, (SRV) 11-31
Set sign minus, SSM, (I) 12-13
Set sign minus, SSM, (SRV) 11-27
Set sign plus, SSP, (I) 12-13
Set sign plus, SSP, (SRV) 11-27
SGL (SR) 11-27
SGT (SRV) 11-54
SH (I) 12-13
SHA (I) 12-22
Shift 9-14
arithmetic, SHA, (I) 12-22
data 12-21
group 11-50
half register left 1, SHL1, (I) 12-22
half register left 2, SHL, (I) 12-23
half register right 1, SHR1, (I) 12-23
half register right 2, SHR2, (I) 12-23
logical, SHL, (I) 12-22
operators 15-8
operators 15-9
register left 1, SL1, (I) 12-22
register left 2, SL2, (I) 12-22
register right 1, SRL, (I) 12-22
register right 2, SR2, (I) 12-22
SHL (I) 12-23
SHL 12-22
SHL1 (I) 12-22
SHR1 (I) 12-23
SHR2 (I) 12-23
Sign bit 15-4
Sign conventions 15-9
Single precision 15-1
Single precision floating point 15-5
Single precision integer value 15-5
Skip conditional, SKIP (V) 11-53
Skip group, SKP, (SRV) 11-54
Skip if
A greater than zero, SGT, (SRV) 11-54
less than or equal to zero, SLE, (SRV) 11-54
condition A minus (a(1)=1), SMI, (SRV) 11-55
condition A nonzero, SNZ, (SRV) 11-55
condition A plus (a(1)=0), SPL, (SRV) 11-55
condition A zero, SZE, (SRV) 11-55
condition all sense switches 1-4 set, SSS, (SRV) 11-55
condition any of sense switches 1-4 reset, SSR, (SRV) 11-55
condition clear C, SRC, (SRV) 11-55
condition lsb nonzero (a(16)=1), SLN, (SRV) 11-55
condition lsb zero (a(16)=0), SLZ, (SRV) 11-55
condition sense switch 1 reset, SRL, (SRV) 11-55
condition sense switch 1 set, SS1, (SRV) 11-55
condition sense switch 2 reset, SR2, (SRV) 11-55
condition sense switch 2 set, SS2, (SRV) 11-55
condition sense switch 3 reset, SR3, (SRV) 11-55
condition sense switch 3 set, SS3, (SRV) 11-55
condition sense switch 4 reset, SR4, (SRV) 11-55
condition sense switch 4 set, SS4, (SRV) 11-55
condition set C, SSC, (SRV) 11-55
satisfied, SKS, (SR) 11-31
A bit reset, SAR, (SRV) 11-53
A bit set, SAS, (SRV) 11-53
machine check reset, SMCR, (SRV) 11-29
machine check set, SMCS, (SRV) 11-29
sense switch reset, SNR, (SRV) 11-54
sense switch set, SNS, (SRV) 11-54
Skip unconditionally, SKP, (SRV) 11-55
SKIP, skip conditional (V) 11-53
SKP (SRV) 11-54
SKS (SR) 11-31
SL1 (I) 12-22
SL2 (I) 12-22
SLA (SRV) 11-26
SLE (SRV) 11-54
SLN (SRV) 11-55
SLZ (SRV) 11-55
SMCR (SRV) 11-29
SMCS (SRV) 11-29
SMI (SRV) 11-55
SNR (SRV) 11-54
SNS (SRV) 11-54
SNZ (SRV) 11-55
Sort library 8-1
Source files 3-1
Space conventions 15-9
Special case selection address formation 9-18
SPL (SRV) 11-55
SR address truncation 10-2
SR keys 9-12
SR subroutine call conventions 8-2
SR1 (I) 12-22
SR2 (I) 12-22
SR2 (SRV) 11-55
SR3 (SRV) 11-55
SR4 (SRV) 11-55
SRC 11-55
SRL (SRV) 11-55
SRTLIB, system library 8-1
SS1 (SRV) 11-55
SS2 (SRV) 11-55

SS3 (SRV) 11-55
SS4 (SRV) 11-55
SSC (SRV) 11-55
SSM (I) 12-13
SSM (SRV) 11-27
SSP (I) 12-13
SSP (SRV) 11-27
SSR (SRV) 11-55
SSS (SRV) 11-55
ST (I) 12-19
ST.SIZE, SEG load map entry 5-6
STA (SRV) 11-41
STAC (V) 11-41
Stack
  assignment (SEG) 5-8
  extend, STEX, (I) 12-10
  extend, STEX, (V) 11-48
  frame description in load map 5-6
  frame header CALF 9-5
  frame header PCL 9-4
  overflow, SEG load map information 5-3
  postincrement address formation 10-6
  predecrement address formation 10-6
  relative (R-mode) memory reference instruction formats 10-8
  segment header 9-4
Stack, PMA 14-6, 14-8
Stack, SEG assignment of 5-8
STAR (I) 12-19
START 18-3
Starting a loaded program 6-1
Statements, PMA 14-1, 14-2
STC (V) 11-5
STC (I) 12-4
STCD (I) 12-19
STCH (I) 12-19
STEX (I) 12-20
STEX (V) 11-48
STFA (I) 12-6
STFA (V) 11-15
STH (I) 12-19
STL (V) 11-41
STLC (V) 11-42
STLR (V) 11-42
Storage allocation pseudo-operations, SA 16-1
Store
  A conditionally, STAC, (V) 11-41
  addressed register, STAR, (I) 12-19
  character, STC, (I) 12-4
  character, STC, (V) 11-5
  conditional fullword, STCC, (I) 12-19
  conditional fullword, STCH, (I) 12-19
  field address register, STFA, (I) 12-6
  field address register, STFA, (V) 11-15
  fullword, ST, (I) 12-19
  halfword, STH, (I) 12-19
  L conditionally, STLC, (V) 11-42
  L into addressed register, STLR, (V) 11-42
  long, STL, (V) 11-41
  process model number, STPM, (I) 12-17
  processor model number, STPM, (V) 11-38
  the A register, STA, (SRV) 11-41
  X register, STX, (SRV) 11-42
  Y, STY, (V) 11-42
STPM (I) 12-17
STPM (V) 11-38
STX (SRV) 11-42
STY (V) 11-42
SUB (SRV) 11-27
SUBR pseudo-operation 16-20
Subroutine call conventions, (SR) 8-2
Subroutine call conventions, (VI) 8-3
Subroutine calling 8-1
Subroutines, loading library 4-3, 5-3
Subtract
  fullword, S, (I) 12-13
  halfword, SH, (I) 12-13
  long, SBL, (V) 11-27
  one from A, SLA, (SRV) 11-26
  two from A, S2A, (SRV) 11-27
Subtract, SUB, (SRV) 11-27
Subtraction operator 15-9
Supervisor call, SVC, (I) 12-20
Supervisor call, SVC, (SRV) 11-39
SVC (I) 12-20
SVC (SRV) 11-39
Symbol definition pseudo-operations, SD 16-1
Symbol definitions in load maps 4-4
Symbolic addresses, PMA 14-5
Symbolic instruction format 18-5
Symbols, PMA 14-4
Syntax, PMA 14-4
System error messages 4-1, C
System errors, SEG's action 5-1
System libraries 8-1
System programming features 4-6
SZE (SRV) 11-55

T

TAB (V) 11-42
TAK (V) 11-31
TAP (Trace And Patch) 18-1
  command summary 19-1
  terminators 19-1
  using 18-1
TAX (V) 11-42
TAY (V) 11-42
TBA (V) 11-43
TC (I) 12-13
TCA (SRV) 11-28
TCH (I) 12-13
TCL (V) 11-28
Terminating long operations 18-1, 18-3
Terminators 18-4
Terminators, PSD 20-1
Terminators, TAP 19-1
Terminators, VPSD 21-1
Terms 15-5
Test
  A register equal to zero and set A, LEG, (V) 11-33
  A register greater than or equal to zero and set A, LGE, (V) 11-33
  A register greater than zero and set A, LGE, (V) 11-33
  A register greater than zero and set R, LGT, (I) 12-16
  A register less than or equal to zero and set A, LLE, (V) 11-33
  A register less than zero and set A, LLT, (V) 11-33
  A register not equal to zero and set A, LNE, (V) 11-33
  condition code equal to zero and set A, LCEQ, (V) 11-33
  condition code equal to zero and set R, LCEQ, (I) 12-16
  condition code greater than or equal to zero and set A, LCEG, (V) 11-33
  condition code greater than or equal to zero and set R, LCGE (I) 12-16
  condition code greater than zero and set A, LCGT, (V) 11-33
  condition code greater than zero and set R, LCGT, (I) 12-16
  condition code less than or equal to zero and set A, LCLE, (V) 11-33
  condition code less than or equal to zero and set R, LCLE, (I) 12-16
  condition code less than zero and set A, LCLT, (V) 11-33
  condition code less than zero and set R, LCLT, (I) 12-16
  condition code not equal to zero and set A, LCNE, (V) 11-33
  condition code not equal to zero and set R, LCNE, (I) 12-16
  floating register equal to zero and set A, LFEQ, (V) 11-34
  floating register equal to zero and set R, LFEQ, (I) 12-16
  floating register greater than or equal to zero and set A, LFGE, (V) 11-34
  floating register greater than or equal to zero and set R, LFGE, (I) 12-16
  floating register greater than zero and set A, LFGT, (V) 11-34
  floating register greater than zero and set R, LFGT, (I) 12-16
  floating register less than or

equal to zero and set A, LFLE, (V) 11-34
floating register less than or equal to zero and set R, LFLE, (I) 12-16
floating register less than zero and set A, LFLT, (V) 11-34
floating register less than zero and set R, LFLT, (I) 12-16
floating register not equal to zero and set A, LFNE, (V) 11-34
floating register not equal to zero and set R, LFNE, (I) 12-16
half register equal to zero and set R, LHEQ, (I) 12-16
half register greater than or equal to zero and set R, LHGE, (I) 12-16
half register greater than zero and set R, LHGT, (I) 12-16
half register less than or equal to zero and set R, LHLE, (I) 12-16
half register not equal to zero and set R, LHNE, (I) 12-16
L register equal to zero and set A, LLEQ, (V) 11-33
L register greater than or equal to zero and set A, LLGE, (V) 11-33
L register greater than zero and set A, LLGT, (V) 11-33
L register less than or equal to zero and set A, LLLE, (V) 11-33
L register less than zero and set A, LLLT, (V) 11-33
L register not equal to zero and set A, LLNE, (V) 11-33
L-bit 11-3
memory fullword, TM, (I) 12-14
memory halfword, TMH, (I) 12-14
queue, TSTQ, (I) 12-21
queue, TSTQ, (V) 11-50
register equal to zero and set R, LEQ, (I) 12-16
register greater than or equal to zero and set R, LGE, (I) 12-16
register less than or equal to zero and set R, LLE, (I) 12-16
register less than zero and set R, LLT, (I) 12-16
register not equal to zero and set R, LNE, (I) 12-16
Text conventions 2-1
TFLL (V) 11-15
TFLR (I) 12-5
Three word memory reference, indirect pointer 9-3
TKA (V) 11-31
TLFL (V) 11-15
TM (I) 12-14
TMH (I) 12-14
TOP, SEG load map entry 5-6
Transfer
A or B, TAB, (V) 11-42
A to keys, TAK, (V) 11-31
A to X, TAX, (V) 11-42
A to Y, TAY, (V) 11-42
B to A, TBA, (V) 11-43
field length register to L, TFLL, (V) 11-15
field length to register, TFLR, (I) 12-5
keys to A, TKA, (V) 11-31
X to A, TXA, (V) 11-43
Y to A, TYA, (V) 11-43
Translate character field, ZTRN, (V) 11-7
Translate character fields, ZTRR, (I) 12-4
Triple asterisk (dummy instruction), PMA 14-5
TSTQ (I) 12-21
TSTQ (V) 11-50
Two word memory reference (V-mode) 10-8
Two word memory reference, indirect pointer 9-3
Two's complement
A, TCA, (SRV) 11-28
half register, TCH, (I) 12-13
long, TCL, (V) 11-28
register, TC, (I) 12-13
TXA (V) 11-43
TYA (V) 11-43
Types, PMA 14-1

U

UII
handling 4-6
library 4-4
library, loader use of 4-4
Unsatisfied references 5-7
Unsegmented runfiles, execution of 6-1
Using the assembler, terminal session example 2-5
Using the R-mode loader under PRIMOS 4-1
Using VPSD 18-3

V

V-mode instructions
A2A 11-22
ABQ 11-49
ACA 11-22
ADD 11-22
ADL 11-22
ADLL 11-23
ALA 11-22
ALFA 11-15
ALL 11-51
ALR 11-51
ALS 11-51
ANA 11-32
ANL 11-32
ARGT 11-43
ARL 11-51
ARR 11-51
ARS 11-51
ATQ 11-49
BCEQ 11-2
BCGE 11-2
BCGT 11-2
BCLE 11-2
BCLT 11-2
BCNE 11-2
BCR 11-3
BCS 11-3
BDX 11-4
BDY 11-4
BEQ 11-4
BFEQ 11-4
BFGE 11-4
BFGT 11-4
BFLE 11-4
BFLT 11-4
BFNE 11-4
BGE 11-4
BGT 11-4
BIX 11-4
BIY 11-4
BLEQ 11-4
BLGE 11-4
BLGT 11-4
BLLE 11-4
BLLT 11-4
BLNE 11-4
BLE 11-4
BLR 11-3
BLS 11-3
BLT 11-4
BMEQ 11-3
BMGE 11-3
BMGT 11-3
BMLE 11-3
BMLT 11-3
BMNE 11-3
BNE 11-4
CAI 11-29
CAL 11-7
CAR 11-7
CAS 11-23
CAZ 11-23
CGT 11-4
CLS 11-23
CMA 11-32
CRA 11-7
CRB 11-8
CRE 11-8
CRL 11-8
CRLE 11-8
CSA 11-24
CXCS 11-34
DFAD 11-17
DFCM 11-17
DFCS 11-17
DFDV 11-17
DFLD 11-18
DFLX 11-18
DFMP 11-18
DFSB 11-18
DFST 11-18
DIV 11-24
DRX 11-53
DVL 11-25
E16S 11-1
E32I 11-1
E32R 11-1
E32S 11-1
E64R 11-1
E64V 11-1
EAFA 11-15

EAL 11-44
EALB 11-45
EAXB 11-45
EIO 11-29
EMCM 11-28
ENB 11-39
ERA 11-32
ERL 11-32
ESIM 11-30
EVIM 11-30
EXB 11-43
FAD 11-18
FCM 11-19
FCS 11-19
FCST 11-20
FDBL 11-19
FDV 11-19
FLD 11-19
FLTA 11-20
FLTL 11-20
FMP 11-20
FRN 11-20
FSB 11-20
FSLE 11-20
FSMI 11-21
FSNZ 11-21
FSPL 11-21
FST 11-21
FSZE 11-21
HLT 11-35
IAB 11-40
ICL 11-40
ICR 11-40
ILE 11-40
IMA 11-40
INBC 11-49
INBC 11-49
INEC 11-49
INEN 11-49
INH 11-30
INT 11-21
INTA 11-21
INTL 11-22
IRS 11-53
IRX 11-53
ITBL 11-35
JMP 11-46
JST 11-46
JSX 11-47
JSXB 11-47
JSY 11-47
LCEQ 11-33
LCGE 11-33
LCGT 11-33
LCLE 11-33
LCLT 11-33
LCNE 11-33
LDA 11-40
LDC 11-5
LDL 11-40
LDLR 11-41
LDX 11-41
LDY 11-41
LEQ 11-33
LF 11-34
LFEQ 11-34
LFGE 11-34
LFGT 11-34
LFLE 11-34
LFLI 11-15
LFLT 11-34
LFNE 11-34
LGE 11-33
LGT 11-33
LIOT 11-35
LLE 11-33
LLEQ 11-33
LLGE 11-33
LLGT 11-33
LLL 11-51
LLLE 11-33
LLLT 11-33
LLNE 11-33
LLR 11-52
LLS 11-52
LLT 11-33
LMCM 11-28
LNE 11-33
LPID 11-36
LPSW 11-36
LRL 11-52
LRR 11-52
LRS 11-53
LT 11-34
LWCS 11-37
MDEI 11-28
MDII 11-28
MDIW 11-28
MDRS 11-28
MDWC 11-28
MIA 11-37
MIB 11-37
MPL 11-25
MPY 11-25
NFYB 11-49
NFYE 11-49
NOP 11-37
NOP 11-55
PCL 11-47
PIDA 11-26
PIDL 11-26
PIMA 11-26
PIML 11-26
PRTN 11-48
PTLB 11-37
RBQ 11-50
RCB 11-31
RMC 11-28
RRST 11-38
RSAV 11-38
RTQ 11-50
S1A 11-26
S2A 11-27
SAR 11-53
SAS 11-53
SBL 11-27
SCB 11-31
SGT 11-54
SKP 11-54
SLE 11-54
SLN 11-55
SLZ 11-55
SMCR 11-29
SMCS 11-29
SMI 11-55
RMQ 11-54
SNS 11-54
SNZ 11-55
SPL 11-55
SR2 11-55
SR3 11-55
SR4 11-55
SRC 11-55
SRL 11-55
SS1 11-55
SS2 11-55
SS3 11-55
SS4 11-55
SSC 11-55
SSM 11-27
SSP 11-27
SSR 11-55
SSS 11-55
STA 11-41
STAC 11-41
STC 11-5
STEX 11-48
STFA 11-15
STL 11-41
STLC 11-42
STLR 11-42
STPM 11-38
STX 11-42
STY 11-42
SUB 11-27
SVC 11-39
SZE 11-55
TAB 11-42
TAK 11-31
TAX 11-42
TAY 11-42
TBA 11-43
TCA 11-28
TCL 11-28
TFLL 11-15
TKA 11-31
TLFL 11-15
TSTQ 11-50
TXA 11-43
TYA 11-43
VIRY 11-29
WAIT 11-49
WCS 11-39
XAD 11-10
XBTD 11-11
XCA 11-43
XCM 11-11
XDTB 11-12
XDV 11-12
XEC 11-49
XED 11-13
XMP 11-13
XMV 11-14
XVRY 11-29
ZCM 11-5
ZED 11-5
ZFIL 11-6
ZMV 11-6
ZMVD 11-7
ZTRN 11-7
Values, output 18-4
VAPPLB library, use with
SEG 5-2, 5-3
VAPPLB, system library 8-1
Variables, macro 17-2
Verify the XIS board (Prime 500), XVRY, (V) 11-29
Verify XIS, XVRY, (I) 12-14
Verify XIS, XVRY, (I) 12-17
Verify, VIRY, (I) 12-14
Verify, VIRY, (SRV) 11-29
VFD pseudo-operation 16-11

VI
decimal control word format 9-2
keys 9-12
queue control block 9-7
registers 9-9
subroutine call conventions 8-3
Virtual address space 4-5
Virtual loading 4-5
VIRY (I) 12-14
VIRY (SRV) 11-29
VPSD (virtual symbolic debugger) 18-1, 18-3, 18-4
command summary 21-1
terminators 21-1
using 18-3
VPSD16 18-3
VSRTLB library, use with SEG 5-2, 5-3
VSRTLB, system library 8-1

W

WAIT (I) 12-20
WAIT (V) 11-49
WCS (I) 12-17
WCS (RV) 11-39
Writable control store, WCS, (I) 12-17
Writable control store, WCS, (RV) 11-39
Writing and debugging a program, example 2-5
Writing V or I mode code in PMA 14-8

X

X (I) 12-15
XAC pseudo-operation 16-7
XAD (I) 12-5
XAD (V) 11-10
XB% assembler notation 15-8
XBTD (I) 12-5
XBTD (V) 11-11
XCA (SRV) 11-43
XCB (SRV) 11-43
XCM (I) 12-5
XCM (V) 11-11
XDTB (I) 12-5
XDTB (V) 11-12
XDV (I) 12-5
XDV (V) 11-12
XEC (RV) 11-49
XED (I) 12-5
XED (V) 11-13
XH (I) 12-16
XMP (I) 12-5
XMP (V) 11-13
XMV (I) 12-5
XMV (V) 11-14
XSET pseudo-operation 16-21
XVRY (V) 11-29
XVRY (I) 12-14, 12-17

Z

ZCM (I) 12-4
ZCM (V) 11-5
ZED (I) 12-4
ZED (V) 11-5
Zero memory fullword, ZM, (I) 12-5
Zero memory halfword, ZMH, (I) 12-5
Zero, VPSD 21-6
ZFH, fill character field, (I) 12-4
ZFIL, fill field, (V) 11-6
ZM, zero memory fullword, (I) 12-5
ZMH, zero memory halfword, (I) 12-5
ZMV, move character field, (I) 12-4
ZMV, move character field, (V) 11-6
ZMVD, move equal length fields, (I) 12-4
ZMVD, move equal length fields, (V) 11-7
ZTRN, translate character field, (V) 11-7
ZTRN, translate character fields, (I) 12-4

Ø1 JAN 8Ø
REV. 17.

AIDUS

Change sheet package

This is your AIDUS Change Sheet package for FDR3059, **The Assembly Language Programmer's Guide.** It contains replacement pages to update your book to Master Disk Revision 17.

Two types of changes are indicated on these change pages. Changes that are specific to Revision 17 are indicated by the following symbol: <17>. The bar extending up and down from the symbol, points out the overall area where Revision 17 changes were made. Other changes, (errors fixed, information missing at Revision 16 or earlier, or editorial changes) are shown by a simple bar in the inner margin. All pages with changes of either type are now dated 1 january 1980 in the folio line.

Change Sheet Package Number: COR3059-001
Date: January, 1980
Revision Number: 17
Number of pages enclosed: 14 **Pages with changes:** 8
List of pages enclosed (pages with changes are underlined): 4-3, 4-4, 5-9, 5-10, 5-11, 5-12, 5-13, 6-0, 6-1, 6-2, 8-1, 8-2, 16-19, 16-20

Copyright© 1980 by Prime Computer, Incorporated
Published by Prime Computer, Incorporated
Technical Publications Department
500 Old Connecticut Path
Framingham, MA 01701

The information contained on these change pages is subject to change without notice and should not be construed as a commitment by Prime Computer, Incorporated. Prime Computer, Incorporated assumes no responsibility for any errors that may appear in this package.

PR1ME and PRIMOS are registered trademarks of Prime Computer.

PRIMENET and THE PROGRAMMER'S COMPANION are trademarks of Prime Computer, Inc.

Printing date: January, 1980

All correspondence on suggested changes to this document should be addressed to:

Rosemary Shields
Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

3. Other Prime libraries (LI filename).
4. Standard FORTRAN library (LI).

Loading library subroutines

Standard FORTRAN mathematical and input/output functions are implemented by subroutines in the library file FTNLIB in the LIB UFD. The appropriate subroutines from the file are loaded by the LIBRARY command given without a filename argument. If subroutines from other libraries are used, such as MATHLB, SRTLIB, or APPLIB, additional LIBRARY commands are required which include the desired library as an argument.

LOAD MAPS

During loading the loader collects information about the results of the load process, which can be printed at the terminal (or written to a file) by the MAP command:

MAP [pathname] [option]

The information in the map can be consulted to diagnose problems in loading, or to optimize placement of modules, linkage areas and COMMON in complex loads.

Load information is printed in four sections, as shown in Figure 4-1. The amount of information printed is controlled by MAP option codes such as:

Option	Load Map Information
None, 0 or 4	Load state, base area, and symbol storage; symbols sorted by address
1	Load state only
2	Load state and base areas
3	Unsatisfied references only
6	Undefined symbols, sorted in alphabetical order
7	All symbols, sorted in alphabetic order

Load state

The load state area shows where the program has been loaded, the start-of-execution location, the area occupied by COMMON, the size of the symbol table, and the UII status. All locations are octal numbers.

***START:** The location at which execution of the loaded program will begin. The default is '1000.

***LOW:** The *lowest* memory image location occupied by the program. Executable code normally starts at '1000, but sector 0 address links (if any) begin at '200.

***HIGH:** The *highest* memory image location occupied by the program (excluding any area reserved for initialized COMMON).

***PBRK:** "Program Break": The next available location for loading. It normally is the location following the last loaded module, but can be moved by PBRK or the LOAD family of commands.

***CMLOW:** The low end of COMMON.

***CMHGH:** The top of COMMON.

***SYM:** The number of symbols in the loader's symbol table. This is usually of not concern unless the symbol space crowds out the last remaining runfile buffer area. (There is room for about 4000 symbols before this is a risk.)

***UII:** A code representing the hardware required to execute the instructions in loaded modules. Codes and other information are described later in this section.

Base areas

The base area map includes the lowest, highest and next available locations for all defined base areas. Each line contains four addresses as follows:

***BASE XXXXXX YYYYYY ZZZZZZ WWWWWW**

XXXXXX	Lowest location defined for this area
YYYYYY	Next available location if starting up from XXXXXX
ZZZZZZ	Next available location if starting down from WWWWWW
WWWWWW	Highest location defined for this area

Symbol storage

The symbol storage listing consists of every defined label or external reference name printed four per line in the following format:

namexx NNNNNN

or

****namexx NNNNNN**

NNNNNN is a six-digit octal address. The ** flag means the reference is unsatisfied (i.e., has not been loaded).

Symbols are listed by ascending address (default) or in alphabetical order (MA 6 or MA 7). The list may be restricted to unsatisfied references only (MA 3 or MA 6).

COMMON blocks

The low end and size of each COMMON area are listed, along with the name (if any). Every map includes a reference to the special COMMON block LIST, defined as starting at location 1.

LOADER CONCEPTS

When standard loading goes well, the user can ignore most of the loader's advanced features. However, situations can arise where some detailed knowledge of the loader's tasks, can optimize size or performance of a runfile, or even make a critical load possible. From that viewpoint, the main tasks of the loader are:

- Convert block-format object code into a run-time version of the program (executable machine instructions, binary data and data blocks).
- Resolve address linkages (translate symbolic names of variables, subroutine entry points, data items etc. into appropriate binary address values).
- Perform address resolution (discussed later).
- Detect and flag errors such as unresolved external references, memory overflow, etc.
- Build (and, on request, print) a load map. The map may also be written to a file.
- Reserve COMMON areas as specified by object modules.
- Keep track of runfile's hardware execution requirements and make user aware of need to load subroutines from UII library.

SEG-LEVEL COMMANDS

Commands at SEG level are entered in response to the "#" prompt.

DELETE [pathname]

Deletes a saved SEG runfiles.

HELP

Prints abbreviated list of SEG commands at terminal.

[V]LOAD[*] [pathname]

Defines runfile name and invokes virtual loader for creation of new runfile (if name did not exist) or appending to existing runfile (if name exists). If **pathname** is omitted, SEG requests one.

MAP pathname-1 [pathname-2] [map-option]

Prints a loadmap of runfile (**pathname-1** or current loadfile (*)) at terminal or optional file (**pathname-2**).

Option	Load Map information
0	Full map (default)
1	Extent map only
2	Extent map and base areas
3	Undefined symbols only
4	Full map (identical to 0)
5	System programmer's map
6	Undefined symbols, alphabetical order
7	Full map, sorted alphabetically
10	Symbols by ascending address
11	Symbols alphabetically

MODIFY [filename]

Invokes MODIFY subprocessor to create a new runfile or modify an existing runfile.

PARAMS [filename]

Displays the parameters of a SEG runfile.

PSD

Invokes VPSD debugging utility.

QUIT

Returns to PRIMOS command level and closes all open files.

RESTORE [pathname]

Restores a SEG runfile to memory for examination with VPSD.

RESUME [pathname]

Restores runfile and begins execution.

SAVE [pathname]

Synonym for **MODIFY**.

SHARE [pathname]

Converts portions of SEG runfile corresponding to segments below '4001 into R-mode-like runfiles.

SINGLE [pathname] segno

Creates an R-mode-like runfile for any segment.

TIME [pathname]

Prints time and date of last runfile modification.

VERSION

Displays SEG version number.

VLOAD

See **LOAD**.

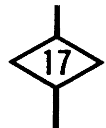
LOAD SUBPROCESSOR COMMANDS

ATTACH [ufd-name] [password] [ldisk] [key]

Attaches to directory.

AUTOMATIC base-area-size

Automatically places base areas between procedures.



A/SYMBOL symbolname [segtype] segno size

Defines a symbol in memory and reserves space for it using absolute segment numbers.

COMMON { **[ABS]** / **REL** } **segno**

Relocates COMMON using absolute or relative segment numbers.

D/ { **IL** / **LOAD** / **LIBRARY** / **FORCELOAD** / **PL** or **RL** }

Continues a load using parameters of previous load command.

Note

D/ and F/ may be combined, as in D/F/LI.

EXECUTE [a] [b] [x]

Performs SAVE and executes program.

F/ { IL | LOAD | LIBRARY | FORCELOAD | PL | RL } [pathname][addr psegno lsegno]

Forceloads all routines in object file.

IL [addr psegno lsegno]

Loads impure FORTRAN library IFTNLB

INITIALIZE [pathname]

Initializes and restarts load subprocessor.

LIBRARY [filename] [addr psegno lsegno]

Loads library file (PFTNLB and IFTNLB if no filename specified).

LOAD [pathname] [addr psegno lsegno]

Loads object file.

MAP [pathname] option

Generates load map (see SEG-level MAP command).

MIXUP { [ON] | OFF }

Mixes procedure and data in segments and permits loading of linkage and common areas in procedure segments. *Not* reset by INITIALIZE.

MV [start-symbol move-block desegno]

Moves portion of loaded file (for libraries). If options are omitted, information is requested.

OPERATOR option

Enables or removes system privileges 0=enable, 1=remove. *Caution: this command is intended only for knowledgeable creators of specialized software.*

PL [addr psegno [segno]

Loads pure FORTRAN library, PFTNLB.

P/ { IL | LOAD | LIBRARY | FORCELOAD | PL | RL } [pathname] option [psegno] [lsegno]

Loads on a page boundary. The **options** are: PR=procedure only, DA=link frames only, none =both procedure and link frames.

QUIT

Performs SAVE and returns to PRIMOS command level.

RETURN

Performs SAVE and returns to SEG command level.

RL pathname [addr psegno lsegno]

Replaces a binary module in an established runfile.

R/SYMBOL symbol-name [segtype] segno size

Defines a symbol in memory and reserves space for it using relative segment assignment. (Default=data segment).

SAVE [a] [b] [x]

Saves the results of a load on disk.

SETBASE segno length

Creates base area for desectorization.

SPLIT [**segno addr** | **addr** | **addr segno addr lsegno**]

Splits segment into data and procedure portions. Formats 2 and 3 allow R mode execution if all loaded information is in segment 4000.

SS symbol-name

Saves symbol; prevents XPUNGE from deleting symbol-name.

STACK size

Sets minimum stack size.

SYMBOL new-symbol-name { **old-symbol-name** | **segno addr** | ***** } [**octal-number**]

Defines a symbol at a location and/or assigns a value of an already defined symbol or a constant.

SZ psegno { **YES** | **[NO]** }

Controls use of sector zero base area in procedure segments.

S/ { **LIBRARY** | **FORCELOAD** | **PL or IL** | **RL or LOAD** } **[pathname] [addr psegno lsegno]**

Loads an object file in specified absolute segments.

XP dsymbol dbase

Expunges symbol from symbol table and deletes base information.

symbol	Action
0	Delete all defined symbols—including COMMON area.
1	Delete only entry points, leaving COMMON areas.

dbase	Action
0	Retain all base information.
1	Retain only sector zero information.
2	Delete all base information.

MODIFY SUBPROCESSOR COMMANDS

NEW **pathname**

Writes a new copy of SEG runfile to disk.

PATCH **segno baddr taddr**

Adds a patch (loaded between **baddr** and **taddr**) to an existing runfile and saves it on disk.

RETURN

Returns to SEG command level.

SK { **ssize** | **segno addr** | **ssize 0 esegno** | **segno addr esegno** }

Specifies stack size (**ssize**) and location. **esegno** specifies an extension stack segment.

START **segno addr**

Changes program execution starting address.

WRITE

Writes all segments above '4000 of current runfile to disk.

6

Executing

This section treats the following topics:

- Execution of program memory images saved by the linking loader.
- Execution of segmented runfiles saved by SEG's loader.
- Installation of programs in the command UFD (CMDNC0).
- Use of run time.

EXECUTION OF UNSEGMENTED RUNFILES

Use the PRIMOS RESUME command to execute an unsegmented runfile:

RESUME pathname

where **pathname** is an R-mode runfile in the current UFD.

Programs which are resident in the user's memory may be executed by a START command:

START

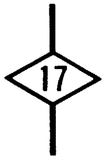
RESUME

RESUME brings the memory-image program **pathname** from the disk into the user's memory, loads the initial register settings, and begins execution of the program. Its format is:

RESUME pathname

Example:

```
OK, R *TEST             User requests program
GO                      Execution begins
THIS IS A TEST          Output of program

OK,                     PRIMOS requests next command
```



Note

As of Rev. 17 PRIMOS no longer prints GO in response to a command.

RESUME should *not* be used for segmented (64V or 32I mode) programs; use the SEG command (discussed later) instead.

START

Once a program is resident in memory (e.g., by a previous RESUME command) you can use START to initialize the registers and begin execution. Its format is:

START [start-address]

Upon completion of the program, control returns to PRIMOS command level.

EXECUTION OF SEGMENTED RUNFILES

Use the SEG command to begin execution of a segmented program; e.g. **SEG pathname** where **pathname** is a SEG runfile. SEG loads the runfile into segmented memory and starts execution. SEG should be used for runfiles created by SEG's Loader; it should not be used for program memory images created by the LOAD utility.

Example:

```
OK, SEG #TEST           user requests program
GO                      execution begins
THIS IS A TEST          output of program

OK,                     PRIMOS requests next command
```

Upon completion of program execution, control returns to the PRIMOS command level.

You may restart a SEG runfile by the command: S 1000, provided both the SEG runfile and the copy of SEG used to invoke it are in memory.

INSTALLATION IN THE COMMAND UFD (CMDNC0)

Run-time programs in the command UFD (CMDNC0) can be invoked by keying in the program name alone. This feature of PRIMOS is useful if a number of users invoke this program. Only one copy of the program need reside on the disk in UFD CMDNC0.

Even more space is saved during execution by multiple users if the program uses shared code (64V and 32I mode only).

Program memory images saved by LOAD

Installation in the command UFD is extremely simple, providing you have access to the password. The runtime version of the program is copied into UFD CMDNC0 using PRIMOS' FUTIL file handling utility.

Example: Assume you have written a utility program called FARLEY. This utility acts as a "tickler" for dates. Using FARLEY, each user builds a file with important dates. The FARLEY utility program, upon request, prints out upcoming events or occasions of interest to the user.

Note

This utility does not necessarily actually exist; it is used as a plausible example.

First, assemble the program.

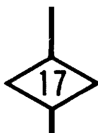
```
OK, PMA FARLEY -64R      Assemble in 64R mode
GO
0000 ERRORS (PMA-REV 16.2)ASSEMBLER MESSAGE
OK, LOAD                 Invoke the Loader
GO
$LO B_FARLEY             Load the object file; the default
                           name is used
$                        Load other required modules
  .
  .
  .
```

Most of the commonly used subroutines — I/O, math functions and EXIT, are either embedded in the operating system or are in one of the FORTRAN libraries. LOAD and SEG automatically load the appropriate library when you type the command LI during a loading sequence. Other libraries, such as APPLIB and MATHLIB require the specification of their name following LI — e.g. LI APPLIB causes the application library to be searched for unresolved references.

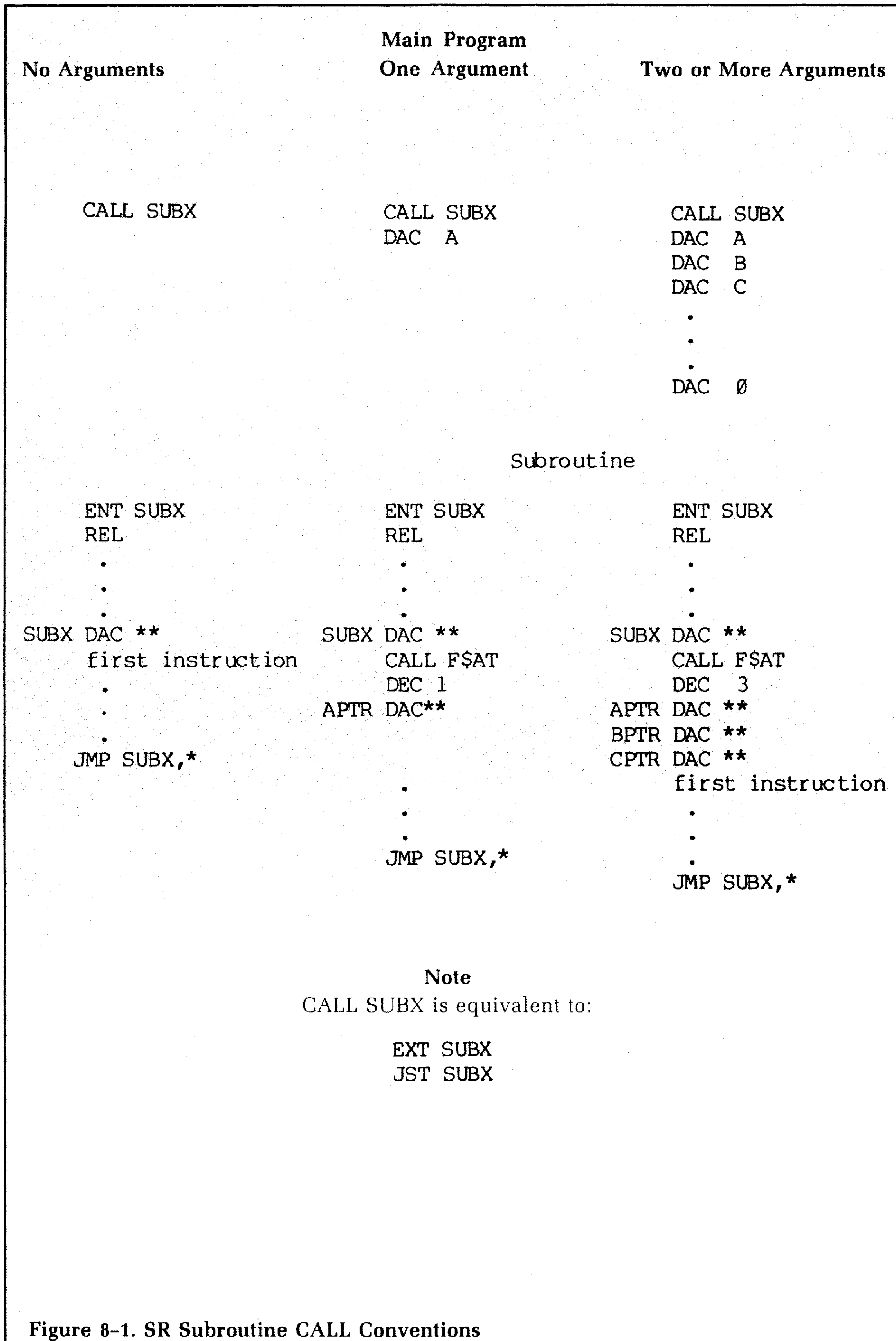
Table 8-1 lists the commonly available system libraries. See the Reference Guide, PRIMOS Subroutines for complete descriptions of the system subroutines.

All routines, regardless of mode, should use the CALL pseudo-operation to call subroutines. S and R-mode arguments use DAC pointers; V, and I-mode arguments use AP pointers (see Section 16 for the DAC and AP pseudo-operation formats). Figure 8-1 illustrates the SR calling sequences and associated subroutine code; Figure 8-2 illustrates VI calling sequences and associated subroutine code.

Table 8-1. System Libraries



Name	Description	Mode
FTNLIB	FORTRAN Library	**R**
PFTNLB	FORTRAN Library pure procedures	**V**
IFTNLB	FORTRAN Library impure procedures	**V**
PLIGLB	PL/I V and I Mode Library	
NPFTNLB	FORTRAN Library V-Mode, Pure, non-shared	
APPLIB	Application Library	**R**
VAPPLB	Application Library	**V**
SRTLIB	Sort Library-Files	**R**
VSRTLI	Sort Library-Files	**V**
MSORTS	Sort Library-Memory	**R**
MATHLB	Matrix Routines	**R**



Main Program

No Arguments	One Argument	Two or More Arguments

```
CALL SUBX           CALL SUBX            CALL SUBX
                    DAC  A               DAC  A
                                         DAC  B
                                         DAC  C
                                         .
                                         .
                                         .
                                         DAC  Ø
```

Subroutine

```
     ENT SUBX            ENT SUBX             ENT SUBX
     REL                 REL                  REL
     .                   .                    .
     .                   .                    .
     .                   .                    .
SUBX DAC **         SUBX DAC **          SUBX DAC **
     first instruction   CALL F$AT            CALL F$AT
     .                   DEC 1                DEC  3
     .              APTR DAC**           APTR DAC **
     .                                   BPTR DAC **
     JMP SUBX,*                          CPTR DAC **
                         .                    first instruction
                         .                    .
                         .                    .
                         JMP SUBX,*           .
                                              JMP SUBX,*
```

Note

CALL SUBX is equivalent to:

```
EXT SUBX
JST SUBX
```

Figure 8-1. SR Subroutine CALL Conventions

external. Unlike EXT, there is no conflict between a local variable and a CALL operand with the same symbol.

The operand must contain a single symbol (not an expression) of up to 6 characters, of which the first must be alphabetic. A ,1 for indexing and * for indirect addressing is optional.

In 64V and 32I modes, CALL generates a PCL instruction to an external symbol.

▶ DYNT address-expression

Defines a direct entry point into the operating system. System libraries only.

▶ [label] ECB entry-point, [link base], displacement, n-arguments [, stack-size] [, keys]

Generates an entry control block to define a procedure entry. It must go in the linkage frame with the subroutine entry point pointing to the ECB.

Parameter	Meaning
entry-point	Procedure relative value; entry point for subroutine.
link-base	Link base register value.
displacement	Displacement in stack frame for argument list. May be stack relative or absolute expression.
n-arguments	Number of arguments expected; default is zero.
stack-size	Initial stack frame size. Default is maximum area specified in DYNM statements.
keys	CPU keys for procedure. Default is 64V addressing mode, all other keys zero.

For example:

```
        ENT   ECBNAM
LAB1    LDA   ='123
        LINK
ECBNAM  ECB   LAB1
        END
```

If the default value for a parameter is desired, the parameter may be omitted, leaving only its delimiting comma. Any string of trailing commas may be omitted.

Note

The main program — that which you call PRIMOS level using SEG#Name — is a subroutine to SEG and must, therefore, have an ECB and the ECB name on the END statement. It need not have an ENT because SEG will give a dummy entry point name to a routine called at this level.

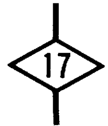
▶ [label] EXT symbol

Identifies external variables. The names appearing in the operand of this statement are flagged as external references. Whenever other statements in the main program reference one of these names, a special block of object text is generated that notifies the loader to fill in the address properly. The assembler fills the address fields with zeroes.

Names defined by the EXT pseudo-operations are unique only in the first 6 characters (loader restriction) and should not appear in a label field internal to the program.

▶ **[label] SUBR symbol-1** *or* **ENT symbol-1 [, symbol-2]**

Link subroutine entry points to external names used in CALL, XAC or EXT statements in calling programs. SUBR and ENT are identical in effect. **symbol-1** is the external name used by calling program, whereas, **symbol-2** is the entry name used in subroutine, if different from symbol-1. All SUBR and ENT pseudo-operations must appear before any generated code.



▶ **SYML** *SEG/SEGR*

Allows long external names up to eight characters to be generated by the assembler. Must follow SEG or SEGR but precede any generated code.

STORAGE ALLOCATION PSEUDO-OPERATIONS (SA)

Control the allocation of storage within the program address space.

Name	Function	Restriction
BSS	Allocate block starting with symbol	
BES	Allocate block ending with symbol	
BSZ	Allocate block set to zeroes	
COMM	FORTRAN compatible COMMON	

▶ [label] { **BSS** / **BES** / **BSZ** } **absolute-expression**

Allocates a block of words of the size specified in the **absolute-expression** starting at the current location count. If there is a label, it is assigned to the first word of the block (BSS and BSZ) or to the last word of the block (BES). For BSZ, all words within the block are set to zeroes.

▶ **[label] COMM symbol [(absolute-expression)]**

Defines FORTRAN-compatible named COMMON areas. These areas are allocated by the loader. The **label** assigns a name to the block as a whole, and the operand field specifies named variables or arrays within the block. Additional COMM statements with the same block name are treated as continuations. **symbol** alone reserves a single location: the optional **absolute-expression** reserves locations equal to its value. In SEG mode, the loader sets up a 32-bit indirect pointer in the linkage segment which points to the common area.

SYMBOL DEFINING PSEUDO-OPERATIONS (SD)

Variables used as address symbols are usually defined when they appear in the label field of an instruction or pseudo-operation statement. Symbols so defined are given the relocation mode and value of the program counter at that location. The EQU, SET and XSET statements make it possible to equate symbols to any numerical value, including ones that lie outside the range of addresses in a program.

Name	Function	Restrictions
DYNM	Declare stack relative	*R and V only*
EQU	Symbol definition	
SET	Symbol definition	
XSET	Symbol definition	

USER SURVEY

Tell us how we're doing, and we'll send you a free Programmer's Companion.

Your name ______________________

Company or School ______________________

Address ______________________

City, State, Zip ______________________

---

1. What is your job title or function? ______________________

2. What specific task describes what you do? ______________________

3. Does your company or school own a Prime computer? ☐ YES ☐ NO

a. If YES, which model? ☐ 450 ☐ 550 ☐ 650 ☐ 750 ☐ OTHER

b. Is it networked with other Prime computers? ☐ YES ☐ NO

c. Is it networked with any of these? ☐ IBM ☐ CDC ☐ UNIVAC ☐ HONEYWELL

d. Which of these software packages do you use?

☐ FORTRAN ☐ COBOL ☐ BASIC/VM
☐ FORTRAN 77 ☐ PL/I-G ☐ POWER
☐ MIDAS ☐ DBMS ☐ SPSS
☐ RPGII ☐ FORMS ☐ PRIMENET
☐ RJE ☐ PASCAL ☐ OAS
☐ DBG ☐ DPTX

e. Have you read any other Prime documents? ☐ YES ☐ NO

f. If YES, which ones? ______________________

4. Are you presently evaluating Prime? ☐ YES ☐ NO
Is the documentation playing a part? ☐ YES ☐ NO

5. What book are you reviewing? ______________________

6. My initial reaction to this book was: ☐ EXCELLENT ☐ GOOD ☐ FAIR ☐ VERY GOOD ☐ FAIR

7. After reading it my reaction was: ☐ BETTER ☐ THE SAME WORSE
If BETTER or WORSE why? ______________________

8. How often have you used this book? ☐ EVERY DAY ☐ FAIRLY OFTEN ☐ VERY OFTEN ☐ JUST GOT IT

9. Did the book have the content you expected? ☐ YES ☐ NO
If NO, why? ______________________

10. Did you find the organization useful? ☐ YES ☐ NO
If NO, why? ______________________

11. How did you find the examples? ☐ TOO MANY ☐ ABOUT RIGHT ☐ TOO FEW

12. How did you find the illustrations? ☐ TOO MANY ☐ ABOUT RIGHT ☐ TOO FEW

13. Could you locate the information you needed? ☐ YES ☐ NO

14. Was the index adequate? ☐ YES ☐ NO

15. Please evaluate our writing style and edit quality?

    a. Clarity ☐ VERY CLEAR ☐ AVERAGE ☐ UNCLEAR
    b. Tone ☐ CORRECT ☐ NEUTRAL ☐ STILTED
    c. Technical level ☐ TOO HIGH ☐ ABOUT RIGHT ☐ TOO LOW
    d. General writing quality ☐ EXCELLENT ☐ AVERAGE ☐ FAIR ☐ POOR
    e. Editorial quality ☐ EXCELLENT ☐ AVERAGE ☐ FAIR ☐ POOR

16. Which of these manufacturers documentation have you used the most?

    ☐ IBM ☐ DIGITAL ☐ DG
    ☐ HP ☐ CDC ☐ UNIVAC
    ☐ HONEYWELL ☐ BURROUGHS ☐ WANG
    ☐ GE ☐ XEROX ☐ ____________

    How is Prime compared to theirs?

    ☐ MUCH BETTER ☐ A LITTLE WORSE
    ☐ A LITTLE BETTER ☐ MUCH WORSE
    ☐ SAME

    Any comment on your rating? ____________________

17. Please evaluate the graphic quality:

    a. Rate the general presentation ☐ EXCELLENT ☐ GOOD ☐ POOR ☐ VERY GOOD ☐ FAIR
    b. Do you like the paper color? ☐ YES ☐ NO
    c. Did you find any quality defects? ☐ YES ☐ NO
    d. Do you like the way we've used color for abbreviations and user input? ☐ YES ☐ NO
    e. Did the shading over the tables and charts make them EASIER or HARDER to read? ☐ EASIER ☐ HARDER
    f. Do you like the Programmer's Companion concept? ☐ YES ☐ NO ☐ HAVEN'T SEEN ONE
    g. Which form of bindery do you find most useful? ☐ BOUND ☐ LOOSE-LEAF

18. Do you know about the AIDUS program? ☐ YES ☐ NO

19. Any other comments:

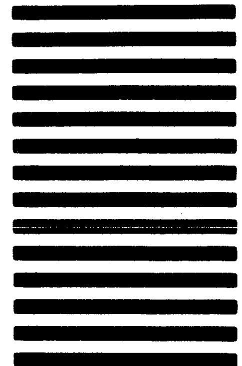
____
____
____
____
____
____
____
____
____
____
____

20. What book don't we offer that you'd like to see? ____

Thank you for filling out the survey. Check off which Programmer's Companion you would like to receive.

- ☐ PRIMOS
- ☐ FORTRAN 77
- ☐ ASSEMBLY
- ☐ ADMINISTRATOR
- ☐ FORTRAN
- ☐ BASIC/VM
- ☐ POWER
- ☐ WORD PROCESSING

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PR1ME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, MA 01760