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 (CMDNG0) 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—l-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
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—PCEX 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—PCEX 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 pseudo-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 COMMAND 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
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

<table>
<thead>
<tr>
<th>Filename</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>B__filename</td>
<td>Binary (object) file</td>
</tr>
<tr>
<td>L__filename</td>
<td>Listing file</td>
</tr>
<tr>
<td>C__filename</td>
<td>Command file</td>
</tr>
<tr>
<td>filename</td>
<td>Source file</td>
</tr>
<tr>
<td>*filename</td>
<td>Saved (executable) file</td>
</tr>
<tr>
<td>M__filename</td>
<td>Map file</td>
</tr>
<tr>
<td>#filename</td>
<td>SEG runfile</td>
</tr>
</tbody>
</table>

**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.

1 March, 1979  2-1  FDR 3059
Instruction summary and description conventions

A  A Register (16-bits)
B  B Register (16 bits)
L  L Register [A\(\land\)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
\(\rightarrow\)  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

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

### 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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Definition</th>
<th>S</th>
<th>R</th>
<th>V</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN</td>
<td>Generic</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AP</td>
<td>Address Pointer</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>BRAN</td>
<td>Branch</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IBRN</td>
<td>Branch I-mode</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CHAR</td>
<td>Character</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECI</td>
<td>Generic Decimal</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIO</td>
<td>Programmed I/O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SHIFT</td>
<td>Shift</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MR</td>
<td>Memory Reference - Non I-mode</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>MRFR</td>
<td>Memory Reference - Floating Register</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MRNR</td>
<td>Memory Reference - Non Register</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGEN</td>
<td>Register Generic</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**GENERAL DATA STRUCTURES**

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

<table>
<thead>
<tr>
<th>Class</th>
<th>S</th>
<th>R</th>
<th>V</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integer (Unsigned)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>32-bit</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Integer (Signed)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>31-bit</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Floating Point</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>64-bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Decimal</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Character String</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Word</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>32-bit</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Halfword – 16 bit</strong></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>Byte</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Indirect Pointer (IP)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>32-bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>40-bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Address Pointer (AP)</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stacks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment Header</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame Header</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Argument Template</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Entry Control Block</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Queue Control Block</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PROCESSOR CHARACTERISTICS
Table 2-4. Processor Characteristics, lists the program visible portions of the hardware.

<table>
<thead>
<tr>
<th>Table 2-4. Processor Characteristics</th>
<th>S</th>
<th>R</th>
<th>V</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S, R mode</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V, I mode</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field Registers</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating Registers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Keys</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S, R mode</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V, I mode</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-Bit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>L-Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition Codes</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modals</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SAMPLE TERMINAL SESSION
***** FIRST, CREATE THE FILE *****
OK, ED
GO
INPUT
EDIT
TAB 10 15 25

INPUT
\SEG
\RLIT
STR\LDA\='456
\STA\BUFF1
\STA\BUFF2
\PRINT
***** 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: 02.00012S (0005) LDA  STCK
  000003: 04.000401L (0006) STA  BUFF2
  000004: 00.00611 (0007) PRN
  (0008) ***** DATA AREA *****
  000012 (0009) DYNM STCK(1)
       (0010) LINK
  000400> (0011) BUFF1 BSS 1
  000401> (0012) BUFF2 BSS 1
  000402> 00.00000 (0013) ECB  STR
       00014
       00011
       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

1 March, 1979
2-7
FDR 3059
TAB 10 15 25
L END
R \END\ECB1
N-1
    ECB STR
C / /ECB1/
ECB1 ECB STR
FILE

OK, PMA TTY
GO
0000 ERRORS (PMA-REV 15.0)
SLIST L_TTY
GO
SEG

(0001) SEG
(0002) RLIT
000000: 02.00005 (0003) STR
000001: 04.000400L (0004) STA BUFF1
000002: 02.000012S (0005) LDA STCK
000003: 04.000401L (0006) STA BUFF2
000004: 000611 (0007) PRTN
000008 ***** DATA AREA *****
000012 (0009) DYNM STCK(1)
0010 LINK
00400> (0011) BUFF1 BSS 1
00401> (0012) BUFF2 BSS 1
00402> 00000 (0013) ECB1 ECB STR
  00014
  00011
  00000
  177400
  014000
00422 (0014) END ECB1

00005: 00.000456A

TEXT SIZE: PROC 000006 LINK 000022 STACK 000013
BUFF1 00400L 0004 0011
BUFF2 00401L 0006 0012
ECB1 00402L 0013 0014
STCK 00012S 0005 0009
STR 000000 0003 0013

0000 ERRORS (PMA-REV 15.0)

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

<table>
<thead>
<tr>
<th>SEG. #</th>
<th>TYPE</th>
<th>LOW</th>
<th>HIGH</th>
<th>TOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>004001</td>
<td>PROC#</td>
<td>001000</td>
<td>001005</td>
<td>001005</td>
</tr>
<tr>
<td>004002</td>
<td>DATA</td>
<td>000002</td>
<td>000021</td>
<td>000021</td>
</tr>
</tbody>
</table>

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
3

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.

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

**FILE USAGE**

Three files may be involved during an assembly:

<table>
<thead>
<tr>
<th>File Type</th>
<th>PRIMOS File unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>1</td>
</tr>
<tr>
<td>Listing</td>
<td>2</td>
</tr>
<tr>
<td>Object</td>
<td>3</td>
</tr>
</tbody>
</table>
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 change 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:

0001 ERRORS (PMA-REV 15.2)

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
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.
LOADING R-MODE PROGRAMS

PB

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 PB 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 indirections 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/ {  FORCLOAD  } {  LIBRARY  } {  LOAD  } [pathname] [parameters]
Forceloads all modules in specified object file. See LOAD for parameters.

**HARDWARE definition**

Specifies expected level of instruction execution.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P450 and up</td>
<td>100</td>
</tr>
<tr>
<td>P350,P400</td>
<td>57</td>
</tr>
<tr>
<td>P300/FP</td>
<td>17 FP=Floating Point</td>
</tr>
<tr>
<td>P300</td>
<td>3</td>
</tr>
<tr>
<td>P200/HSA</td>
<td>1 HSA=High-speed arithmetic</td>
</tr>
<tr>
<td>P100/HSA</td>
<td>1</td>
</tr>
<tr>
<td>P200</td>
<td>0</td>
</tr>
<tr>
<td>P100</td>
<td>0</td>
</tr>
</tbody>
</table>

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 PBK 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 '777777', the loader requests more setbase values.

**MAP** [pathname] [option]

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

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

\begin{align*}
\text{P/} \& \text{FORCELLOAD} \quad \{ \text{pathname} \} \quad \{ \text{parameters} \}
\end{align*}

\begin{align*}
\text{LOAD} \quad \text{[pathname]} \quad \text{[parameters]}
\end{align*}

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

\text{PAUSE}

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

\begin{align*}
\text{PBRK} \quad \{ \text{symbol-name} \} \quad \{ \text{offset-1} \ldots \{ \text{offset-9} \} \}
\end{align*}

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.

\text{QUIT}

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

\text{SAVE \textit{pathname}}

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

\begin{align*}
\text{SETBASE} \quad \{ \text{base-start} \} \quad \{ \text{base-range} \}
\end{align*}

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

\text{SS symbol-name}

Save symbol. Exempts specified symbol from action of XPUNGE.

\begin{align*}
\text{SYMBOL} \quad \{ \text{symbol-name} \} \quad \{ \text{offset-1} \ldots \{ \text{offset-6} \} \}
\end{align*}

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.

\begin{align*}
\text{SZ} \quad \{ \text{YES} \} \quad \{ \text{NO} \}
\end{align*}

Permits/prohibits links in sector zero.

\text{VIRTUALBASE base-start to-sect or}

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

\text{XPUNGE dsymbols dbase}

Deletes COMMON symbols, other defined symbols, and base areas.
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 character. Loader error messages 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
RESUME

LOAD subcommands:

MODE option
Loads specified object file.

LOAD pathname
LIBRARY [filename]

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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Load Map Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>None, 0 or 4</td>
<td>Load state, base area, and symbol storage; symbols sorted by address</td>
</tr>
<tr>
<td>1</td>
<td>Load state only</td>
</tr>
<tr>
<td>2</td>
<td>Load state and base areas</td>
</tr>
<tr>
<td>3</td>
<td>Unsatisfied references only</td>
</tr>
<tr>
<td>6</td>
<td>Undefined symbols, sorted in alphabetical order</td>
</tr>
<tr>
<td>7</td>
<td>All symbols, sorted in alphabetic order</td>
</tr>
</tbody>
</table>

**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 VLI 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.

*CMHIGH*: The top of COMMON.

*SYM*: The number of symbols in the loader's symbol table. This is usually of no 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXXXX</td>
<td>Lowest location defined for this area</td>
</tr>
<tr>
<td>YYYYYY</td>
<td>Next available location if starting up from XXXXXX</td>
</tr>
<tr>
<td>ZZZZZZ</td>
<td>Next available location if starting down from WWWWWW</td>
</tr>
<tr>
<td>WWWWWW</td>
<td>Highest location defined for this area</td>
</tr>
</tbody>
</table>

**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 U11 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 accommodates 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:

<table>
<thead>
<tr>
<th>UII Value</th>
<th>CPU Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Prime 450 and up</td>
</tr>
<tr>
<td>57</td>
<td>Prime 350 or 400</td>
</tr>
<tr>
<td>17</td>
<td>Prime 300 with FP Hardware</td>
</tr>
<tr>
<td>3</td>
<td>Prime 300</td>
</tr>
<tr>
<td>1</td>
<td>Prime 100 with HSA or 200 with HSA</td>
</tr>
<tr>
<td>0</td>
<td>Prime 100 or 200</td>
</tr>
</tbody>
</table>

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.)

FDR 3059

4-6

1 March, 1979
LOADING R-MODE PROGRAMS

**START** 001000  *LOW* 000200  *HIGH* 007775  *PBKR* 106376
*CMLOW* 077777  *CMHIGH* 077777  *SYM* 000070  *UII* 000001

*BASE 000200  000225  000777  000777
*BASE 001534  001600  001605  001605
*BASE 002576  002660  002661  002661
*BASE 003624  003663  003665  003665
*BASE 004664  004706  004707  004707

**GHOST** 001025  FSWA  001031  FSWX  001037  FSIO  001113
FSA1  001606  FSA3  001606  FSA5  001613  FSA2  001621
FSA6  001627  FSA7  001645  FSCB  002326  FSIOBF  005405
WRASC  005587  IOCSS  005514  IOCSS1  005613  WATBL  005625
LUTBL  005644  PUTBL  005701  RSTBL  005736  OSADD07  005773
OSADD08  006136  OSAAR01  006200  PRSPES  006235  OERRTS  006427
ERRPES  007433  PRWFSS  007436  WTLHNS  007441  ERRSET  007444
FSER  007447  FSHT  007545  EXIT  007534  AC1  007537
AC2  007540  AC3  007541  AC4  007542  AC5  007543
TNOU  007544  TONL  007634  T1OU  007641  T1IB  007661
T1OB  007666  PSAT  007673  PSAT1  007675  GCHAR  007740
SCHAR  007755

COMMON BLOCKS

LIST  000001  007776  075400

A. Full Map [MAP]

**START** 001000  *LOW* 000200  *HIGH* 007775  *PBKR* 106376
*CMLOW* 077777  *CMHIGH* 077777  *SYM* 000070  *UII* 000001

*BASE 000200  000225  000777  000777
*BASE 001534  001500  001605  001605
*BASE 002576  002660  002661  002661

*BASE 003624  003663  003665  003665
*BASE 004664  004706  004707  004707

AC1  007537  AC2  007540  AC3  007541  AC4  007542
AC5  007543  ERRPES  007433  ERRSET  007444  EXIT  007534
FSA1  001605  FSA2  001621  FSA3  001606  FSA5  001613
FSA6  001627  FSA7  001645  FSCAT  007673  FSA1  007675
FSOB  002326  FSER  007447  FSHT  007545  FSIO  001113
FSIOBF  005405  FSWA  001831  FSIO  001837  GCHAR  007740
**GHOST** 001025  IOCSS  005514  IOCSS1  005613  WATBL  005625
OSADD08  006136  OSADD07  006200  OSADD08  006136  OERRTS  006427
PRSPES  006235  PRWFSS  007436  PUTBL  005701  RSTBL  005736
SCHAR  007755  T1IB  007661  T1OB  007666  T1OU  007641
TNOU  007544  TONL  007634  WATBL  005625  WRASC  005587
WTLHNS  007441

COMMON BLOCKS

007776  075400  LIST  000001

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 PFTNLIB and IFTNLB in the UFD LIB). Other libraries, such as VSRTLNB 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).

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:

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

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.
**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**

<table>
<thead>
<tr>
<th>*BASE</th>
<th>VVVVVV</th>
<th>WWWWW</th>
<th>XXXXX</th>
<th>YYYYY</th>
<th>ZZZZZZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>VVVVVV</td>
<td>Segment number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WWWWWW</td>
<td>Lowest location for base area</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XXXXXX</td>
<td>Next available location if starting up from lowest location</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YYYYYY</td>
<td>Next available location if starting down from highest location</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZZZZZZ</td>
<td>Highest location for base area</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 used 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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Word address within a segment.</td>
</tr>
<tr>
<td>segno</td>
<td>Segment number.</td>
</tr>
<tr>
<td>psegio</td>
<td>Procedure segment number.</td>
</tr>
<tr>
<td>lsegno</td>
<td>Linkage segment number.</td>
</tr>
<tr>
<td>[a][b][x]</td>
<td>Values for A, B, and X registers.</td>
</tr>
</tbody>
</table>

**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/ 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 \} [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 \} [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
\[
\begin{cases}
\text{segno addr} \\
\text{addr} \\
\text{addr segno addr lsegno}
\end{cases}
\]
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/
\[
\begin{cases}
\text{LIBRARY} \\
\text{FORGELoad} \\
\text{PL or IL} \\
\text{RL or LOAD}
\end{cases}
\] [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.
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
GO
THIS IS A TEST
OK,
```

User requests program
Execution begins
Output of program

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
.
.
.

FDR 3059

6-2

1 March, 1979
$LI
LOAD COMPLETE
$SA *FARLEY
$QU

OK, FUTIL
GO
>TO CMDNC0 ORDER
>COPY *FARLEY FARLEY
>QUIT

OK,

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

OK, FARLEY
GO
HGW FAR:

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 TRECOPY 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
$ .                          
  .                          
$ L1                         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 $$SEG 1/5707        
GO                           
0000 ERRORS (PMA-REV 16.2)  
OK, FILMEM                  
OK, LOAD                    
$$Z                        
$ER 2                       
$$M0 D64R                   
$$C0 173400                 
$$LO B $$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
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 CMDNC0 ORDER TO UFD=CMDNC0; 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.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Where Described</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM command</td>
<td>Described below</td>
</tr>
<tr>
<td>COMOUTPUT command</td>
<td>Described below</td>
</tr>
<tr>
<td>PMA Error Messages</td>
<td>Appendix C of this document</td>
</tr>
<tr>
<td>Load Maps</td>
<td>Sections 4 and 5 of this document</td>
</tr>
<tr>
<td>Debugging Utilities</td>
<td>Sections 18-21 of this document</td>
</tr>
</tbody>
</table>

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 EGB 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 occurred 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 SEGO**

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 *CMHIGH. 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>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTNLIB</td>
<td>FORTRAN Library</td>
<td>R</td>
</tr>
<tr>
<td>PFTNLB</td>
<td>FORTRAN Library pure procedures</td>
<td></td>
</tr>
<tr>
<td>IFTNLB</td>
<td>FORTRAN Library impure procedures</td>
<td></td>
</tr>
<tr>
<td>APPLIB</td>
<td>Application Library</td>
<td>R</td>
</tr>
<tr>
<td>VAPPLB</td>
<td>Application Library</td>
<td>V</td>
</tr>
<tr>
<td>SRTLIB</td>
<td>Sort Library-Files</td>
<td>R</td>
</tr>
<tr>
<td>VSRTL1</td>
<td>Sort Library-Files</td>
<td>V</td>
</tr>
<tr>
<td>MSORTS</td>
<td>Sort Library-Memory</td>
<td>R</td>
</tr>
<tr>
<td>MATHLIB</td>
<td>Matrix Routines</td>
<td>R</td>
</tr>
<tr>
<td>No Arguments</td>
<td>Main Program</td>
<td>One Argument</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>CALL SUBX</td>
<td>CALL SUBX</td>
<td>CALL SUBX</td>
</tr>
<tr>
<td>DAC A</td>
<td>DAC A</td>
<td>DAC A</td>
</tr>
<tr>
<td></td>
<td>DAC B</td>
<td>DAC B</td>
</tr>
<tr>
<td></td>
<td>DAC C</td>
<td>DAC C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Subroutine**

<table>
<thead>
<tr>
<th>No Arguments</th>
<th>Main Program</th>
<th>One Argument</th>
<th>Two or More Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENT SUBX</td>
<td>ENT SUBX</td>
<td>ENT SUBX</td>
<td></td>
</tr>
<tr>
<td>REL</td>
<td>REL</td>
<td>REL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBX DAC **</td>
<td>SUBX DAC **</td>
<td>SUBX DAC **</td>
<td></td>
</tr>
<tr>
<td>first instruction</td>
<td>CALL F$AT</td>
<td>CALL F$AT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DEC 1</td>
<td>DEC 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>APTR DAC **</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP SUBX,*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note**

CALL SUBX is equivalent to:

EXT SUBX
JST SUBX

Figure 8-1. SR Subroutine CALL Conventions
### Main Program

<table>
<thead>
<tr>
<th>No Arguments</th>
<th>One Argument</th>
<th>Two or More Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SUBX</td>
<td>CALL SUBX</td>
<td>CALL SUBX</td>
</tr>
<tr>
<td>AP A,SL</td>
<td>AP A,S</td>
<td>AP B,S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AP C,S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AP n,SL</td>
</tr>
</tbody>
</table>

### Subroutine

<table>
<thead>
<tr>
<th>ENT SUBX,SBX</th>
<th>ENT SUBX,SBX</th>
<th>ENT SUBX,SBX</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEG</td>
<td>SEG</td>
<td>SEG</td>
</tr>
<tr>
<td>SUBX first instruction</td>
<td>SUBX ARGT first instruction</td>
<td>SUBX ARGT first instruction</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>FRTN</td>
<td>FRTN</td>
<td>FRTN</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>LINK</td>
<td>DYNM APTR(3)</td>
<td>DYNM APTR(3)</td>
</tr>
<tr>
<td>SBX1 ECB</td>
<td>SBX1 ECB</td>
<td>SBX1 ECB</td>
</tr>
<tr>
<td>SUBX</td>
<td>,,APTR,1</td>
<td>,,APTR,n</td>
</tr>
<tr>
<td>DYNM BPTR(3)</td>
<td>DYNM CPTR(3)</td>
<td>DYNM DPTR(3)</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>DYNM nPTR(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Note

CALL SUBX is equivalent to:

- EXT SUBX
- PCL SUBX

---

**Figure 8-2. VI Subroutine CALL Conventions**
3

MACHINE FORMATS AND INSTRUCTIONS
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)

1 March, 1979
Floating Point - Double Precision 64 bits [RV1]

- 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:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14-16</td>
</tr>
</tbody>
</table>

- **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 \( F_x = F_1 - F_2 \), where \( F_x \) 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.
<table>
<thead>
<tr>
<th>BITNO</th>
<th>I</th>
<th>BR</th>
<th>WORDNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>5</td>
<td>6</td>
<td>7-8</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>14-bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>15-bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>16-bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

I  [Bit 1] Indirect bit  
X  [Bit 2] Index bit  

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

<table>
<thead>
<tr>
<th>F</th>
<th>RR</th>
<th>0</th>
<th>SEGNO</th>
<th>WORDNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2-3</td>
<td>4</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>F</th>
<th>RR</th>
<th>1</th>
<th>SEGNO</th>
<th>WORDNO</th>
<th>BITNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2-3</td>
<td>4</td>
<td>5</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

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.  

1 March, 1979  9-3  FDR 3059
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)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FREE POINTER</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>EXTENSION SEGMENT POINTER</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

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)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 - 0</td>
</tr>
<tr>
<td>1</td>
<td>STACK ROOT SEGMENT NUMBER</td>
</tr>
<tr>
<td>2</td>
<td>RETURN POINTER</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CALLER'S SAVED STACK BASE REGISTER</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CALLER'S SAVED LINK BASE REGISTER</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>CALLER'S SAVED KEYS</td>
</tr>
<tr>
<td>9</td>
<td>LOCATION FOLLOWING CALL</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Word</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag bits – set to one by CALF fault</td>
</tr>
<tr>
<td>1</td>
<td>Stack root segment number – for locating free pointer</td>
</tr>
<tr>
<td>2,3</td>
<td>Return pointer – segment number/word number of return location</td>
</tr>
<tr>
<td>4,5</td>
<td>Caller’s saved stack base register</td>
</tr>
<tr>
<td>6,7</td>
<td>Caller’s saved link base register</td>
</tr>
<tr>
<td>8</td>
<td>Caller’s saved keys</td>
</tr>
<tr>
<td>9</td>
<td>Word number of location following call - beginning of argument transfer templates, if any</td>
</tr>
<tr>
<td>10</td>
<td>Fault code</td>
</tr>
<tr>
<td>11,12</td>
<td>Fault address</td>
</tr>
<tr>
<td>13–15</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Entry control block (ECB) (VI)

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

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)

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

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)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>B (Bits 1-4)</td>
<td>Bit number</td>
</tr>
<tr>
<td>I (Bit 5)</td>
<td>Indirect bit</td>
</tr>
<tr>
<td>BR (Bits 7-8)</td>
<td>Base register</td>
</tr>
<tr>
<td></td>
<td>00 Procedure base (PB)</td>
</tr>
<tr>
<td></td>
<td>01 Stack base (SB)</td>
</tr>
<tr>
<td></td>
<td>10 Link base (LB)</td>
</tr>
<tr>
<td>L (Bit 9)</td>
<td>Last template for this call</td>
</tr>
<tr>
<td>S (Bit 10)</td>
<td>Store argument address. Last template for this argument.</td>
</tr>
<tr>
<td>WORDNO (Bits 17-32)</td>
<td>Word number offset from base register</td>
</tr>
</tbody>
</table>
**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 accessible.

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

**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 accessible.

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

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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSWPB</td>
<td>Diagnostic Status Word Procedure Base</td>
</tr>
<tr>
<td>RSAPTR</td>
<td>Register Save Pointer</td>
</tr>
<tr>
<td>GR</td>
<td>General Register</td>
</tr>
<tr>
<td>FAR0</td>
<td>Field Address Register 0</td>
</tr>
<tr>
<td>FLR0</td>
<td>Field Length Register 0</td>
</tr>
<tr>
<td>FAR1</td>
<td>Field Address Register 1</td>
</tr>
<tr>
<td>FLR1</td>
<td>Field Length Register 1</td>
</tr>
<tr>
<td>PB</td>
<td>Procedure Base</td>
</tr>
<tr>
<td>SB</td>
<td>Stack Base</td>
</tr>
<tr>
<td>LB</td>
<td>Link Base</td>
</tr>
<tr>
<td>XB</td>
<td>Temporary [auxiliary] base</td>
</tr>
<tr>
<td>DTAR</td>
<td>Descriptor Table address registers</td>
</tr>
<tr>
<td>KEYS</td>
<td>See below</td>
</tr>
<tr>
<td>MODALS</td>
<td>See below</td>
</tr>
<tr>
<td>OWNER</td>
<td>Pointer to PCB of process owning this register set</td>
</tr>
<tr>
<td>FCODE</td>
<td>Fault Code</td>
</tr>
<tr>
<td>FADDR</td>
<td>Fault Address</td>
</tr>
<tr>
<td>TIMER</td>
<td>1-millisecond process timer (used for time-slice)</td>
</tr>
</tbody>
</table>

#### MICROCODE SCRATCH

<table>
<thead>
<tr>
<th>ADR</th>
<th>HIGH</th>
<th>LOW</th>
<th>RS0</th>
<th>HIGH</th>
<th>LOW</th>
<th>RS1</th>
<th>HIGH</th>
<th>LOW</th>
<th>RS2</th>
<th>HIGH</th>
<th>LOW</th>
<th>RS3</th>
<th>HIGH</th>
<th>LOW</th>
<th>PRIME 300</th>
<th>PRIME 400</th>
<th>PRIME 500</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TR0</td>
<td>—</td>
<td>40</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>140</td>
<td>—</td>
<td>101</td>
<td>141</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>GR0</td>
<td>GR0</td>
<td>GR0</td>
</tr>
<tr>
<td>1</td>
<td>TR1</td>
<td>—</td>
<td>41</td>
<td>—</td>
<td>—</td>
<td>102</td>
<td>142</td>
<td>1(A)</td>
<td>—</td>
<td>143</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>GR1</td>
<td>GR1</td>
<td>GR1</td>
</tr>
<tr>
<td>2</td>
<td>TR2</td>
<td>—</td>
<td>42</td>
<td>—</td>
<td>—</td>
<td>103</td>
<td>144</td>
<td>—</td>
<td>—</td>
<td>2(B)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>GR2</td>
<td>GR2</td>
<td>GR2</td>
</tr>
<tr>
<td>3</td>
<td>TR3</td>
<td>—</td>
<td>43</td>
<td>—</td>
<td>—</td>
<td>104</td>
<td>144</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>GR3</td>
<td>GR3</td>
<td>GR3</td>
</tr>
<tr>
<td>4</td>
<td>TR4</td>
<td>—</td>
<td>44</td>
<td>—</td>
<td>—</td>
<td>105</td>
<td>145</td>
<td>3(S)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>GR4</td>
<td>GR4</td>
<td>GR4</td>
</tr>
<tr>
<td>5</td>
<td>TR5</td>
<td>—</td>
<td>45</td>
<td>—</td>
<td>—</td>
<td>106</td>
<td>146</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>GR5</td>
<td>GR5</td>
<td>GR5</td>
</tr>
<tr>
<td>6</td>
<td>TR6</td>
<td>—</td>
<td>46</td>
<td>—</td>
<td>—</td>
<td>107</td>
<td>147</td>
<td>0(X)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>GR6</td>
<td>GR6</td>
<td>GR6</td>
</tr>
<tr>
<td>7</td>
<td>TR7</td>
<td>—</td>
<td>47</td>
<td>—</td>
<td>—</td>
<td>110</td>
<td>150</td>
<td>13</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>FALR0</td>
<td>FALR0 (FAC0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>RDMX1</td>
<td>—</td>
<td>50</td>
<td>—</td>
<td>—</td>
<td>111</td>
<td>151</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>FALR0</td>
<td>FALR0 (FAC0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RDMX2</td>
<td>—</td>
<td>51</td>
<td>—</td>
<td>—</td>
<td>112</td>
<td>152</td>
<td>4(FAC)</td>
<td>5(FAC)</td>
<td>—</td>
<td>—</td>
<td>FALR1 (FAC)</td>
<td>FALR1 (FAC1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>RATMPL</td>
<td>—</td>
<td>52</td>
<td>—</td>
<td>—</td>
<td>113</td>
<td>153</td>
<td>6(FAC)</td>
<td>FAC</td>
<td>—</td>
<td>15</td>
<td>SB</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>RSGT1</td>
<td>—</td>
<td>53</td>
<td>—</td>
<td>—</td>
<td>114</td>
<td>154</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>PB</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>RSGT2</td>
<td>—</td>
<td>54</td>
<td>—</td>
<td>—</td>
<td>115</td>
<td>155</td>
<td>14</td>
<td>—</td>
<td>—</td>
<td>17</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>RECC1</td>
<td>—</td>
<td>55</td>
<td>—</td>
<td>—</td>
<td>116</td>
<td>156</td>
<td>16</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>RECC2</td>
<td>—</td>
<td>56</td>
<td>—</td>
<td>—</td>
<td>117</td>
<td>157</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>REOIV</td>
<td>—</td>
<td>57</td>
<td>—</td>
<td>—</td>
<td>118</td>
<td>158</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>ZERO</td>
<td>ONE</td>
<td>60</td>
<td>(20)</td>
<td>(21)</td>
<td>120</td>
<td>160</td>
<td>10</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>PBSAVE</td>
<td>—</td>
<td>61</td>
<td>—</td>
<td>—</td>
<td>121</td>
<td>161</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>RDMX3</td>
<td>—</td>
<td>62</td>
<td>(22)</td>
<td>(23)</td>
<td>122</td>
<td>162</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>RDMX4</td>
<td>—</td>
<td>63</td>
<td>—</td>
<td>—</td>
<td>123</td>
<td>163</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>C377</td>
<td>C377</td>
<td>64</td>
<td>(24)</td>
<td>(25)</td>
<td>124</td>
<td>164</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>—</td>
<td>—</td>
<td>65</td>
<td>—</td>
<td>—</td>
<td>125</td>
<td>165</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>owner</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>—</td>
<td>—</td>
<td>66</td>
<td>(26)</td>
<td>(27)</td>
<td>126</td>
<td>166</td>
<td>11(FCODE)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>—</td>
<td>—</td>
<td>67</td>
<td>—</td>
<td>—</td>
<td>127</td>
<td>167</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>PSWPB</td>
<td>—</td>
<td>70</td>
<td>(30)</td>
<td>(31)</td>
<td>130</td>
<td>170</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>PSWKEYS</td>
<td>—</td>
<td>71</td>
<td>—</td>
<td>—</td>
<td>131</td>
<td>171</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>FPA-FLA</td>
<td>—</td>
<td>72</td>
<td>(32)</td>
<td>(33)</td>
<td>132</td>
<td>172</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>PPB-PFB</td>
<td>—</td>
<td>73</td>
<td>—</td>
<td>—</td>
<td>133</td>
<td>173</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>DSRMMA</td>
<td>—</td>
<td>74</td>
<td>(34)</td>
<td>(35)</td>
<td>134</td>
<td>174</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>DSRSTAT</td>
<td>—</td>
<td>75</td>
<td>—</td>
<td>—</td>
<td>135</td>
<td>175</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>DSWPB</td>
<td>—</td>
<td>76</td>
<td>(36)</td>
<td>(37)</td>
<td>136</td>
<td>176</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>RSAPTR</td>
<td>—</td>
<td>77</td>
<td>—</td>
<td>—</td>
<td>137</td>
<td>177</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>dumps</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

*FDR 3059* 9-10 1 March 1979
**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)**

<table>
<thead>
<tr>
<th>Prime 300</th>
<th>Register</th>
<th>Prime 400</th>
<th>Prime 500</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>'04</td>
<td>12H</td>
<td>10H</td>
<td></td>
<td>S MANTISSA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 2 16</td>
</tr>
<tr>
<td>'05</td>
<td>12L</td>
<td>10L</td>
<td></td>
<td>MANTISSA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17 32</td>
</tr>
<tr>
<td>'06</td>
<td>13H</td>
<td>11H</td>
<td></td>
<td>EXONENT (EXCESS 128)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>33 48</td>
</tr>
</tbody>
</table>
Floating point register – double precision (RVI) (2 registers in I-mode)

<table>
<thead>
<tr>
<th>Register</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prime 300</td>
<td>Prime 400</td>
</tr>
<tr>
<td>'04</td>
<td>12H</td>
</tr>
<tr>
<td>'05</td>
<td>12L</td>
</tr>
<tr>
<td>'02</td>
<td>13L</td>
</tr>
<tr>
<td>'06</td>
<td>13H</td>
</tr>
</tbody>
</table>

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
- Stack Base Register
- Link Base Register
- Temporary Base Register

have the following format:

<table>
<thead>
<tr>
<th>0</th>
<th>RING</th>
<th>0</th>
<th>SEGNO</th>
<th>WORDNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

RING (Bits 2–3)  
SEGNO (Bits 5–16)  
WORDNO (Bits 17–32)

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.

<table>
<thead>
<tr>
<th>0</th>
<th>RING</th>
<th>0</th>
<th>SEGNO</th>
<th>WORDNO</th>
<th>LENGTH</th>
<th>BITNO</th>
<th>0</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2-3</td>
<td>4</td>
<td>5-16</td>
<td>17-32</td>
<td>33-48</td>
<td>49-52</td>
<td>53-59</td>
<td>60-64</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>C</th>
<th>DBL</th>
<th>—</th>
<th>Mode</th>
<th>0</th>
<th>Bits 9-16 of location 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4-6</td>
<td>7-8</td>
<td>9</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>C</th>
<th>0</th>
<th>L</th>
<th>MODE</th>
<th>F</th>
<th>X</th>
<th>LT</th>
<th>EQ</th>
<th>DEX</th>
<th>0</th>
<th>I</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4-6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

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) Condition code bits:
    LT negative
    EQ positive
EQ (Bit 10) 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”.

<table>
<thead>
<tr>
<th>E</th>
<th>V</th>
<th>0</th>
<th>CURREG</th>
<th>MIO</th>
<th>P</th>
<th>S</th>
<th>MCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3-8</td>
<td>9-11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15-16</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>OP</th>
<th>SHIFT-NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10 11 16</td>
</tr>
<tr>
<td>OP (Bits 1–10)</td>
<td>Opcode - Bits 3–6 are always zero</td>
</tr>
<tr>
<td>SHIFT-NO (Bits 11–16)</td>
<td>Two's complement of the number of places to be shifted. Zero means shift 63 places</td>
</tr>
</tbody>
</table>

I/O (SR)

<table>
<thead>
<tr>
<th>CLASS</th>
<th>1 1 0 0</th>
<th>FUNCTION</th>
<th>DEVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>6 7 10 11 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLASS (Bits 1–2)</td>
<td>Type of I/O instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>Sense</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bits 3–6</td>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUNCTION (Bits 7–10)</td>
<td>Subdivision of class. Device dependent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEVICE (Bits 11–16)</td>
<td>Device type</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DECIMAL (VI)

<table>
<thead>
<tr>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 16</td>
</tr>
<tr>
<td>OP (Bits 1–16)</td>
</tr>
</tbody>
</table>

CHARACTER (VI)

<table>
<thead>
<tr>
<th>OP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 16</td>
</tr>
<tr>
<td>OP (Bits 1–16)</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>OP</th>
<th>R</th>
<th>AD</th>
<th>S</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>7-9</td>
<td>10-11</td>
<td>12-14</td>
<td>15-16</td>
<td>17-32</td>
</tr>
</tbody>
</table>

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

MRGR-memory reference general register:

<table>
<thead>
<tr>
<th>OP</th>
<th>110</th>
<th>OP</th>
<th>AD</th>
<th>S</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>4-6</td>
<td>7-9</td>
<td>10-11</td>
<td>12-14</td>
<td>15-16</td>
<td>17-32</td>
</tr>
</tbody>
</table>

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

MRFR-memory reference floating register:

<table>
<thead>
<tr>
<th>OP</th>
<th>110</th>
<th>OP</th>
<th>FR</th>
<th>OP</th>
<th>AD</th>
<th>S</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>4-6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10-11</td>
<td>12-14</td>
<td>15-16</td>
<td>17-32</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>AD</th>
<th>S</th>
<th>B</th>
<th>Effective Address/Instruction Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&gt;0</td>
<td>—</td>
<td>(D+B)*+S (indirect, post-index)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>—</td>
<td>(D+B)* (indirect)</td>
</tr>
<tr>
<td>2</td>
<td>&gt;0</td>
<td>—</td>
<td>(D+B+S)* (pre-index, indirect)</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>—</td>
<td>(D+B)* (indirect)</td>
</tr>
<tr>
<td>1</td>
<td>&gt;0</td>
<td>—</td>
<td>D+B+S (indexed)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>—</td>
<td>D+B (direct)</td>
</tr>
<tr>
<td>0</td>
<td>≥0</td>
<td>0</td>
<td>REG-REG (S specifies source register)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Immediate Type 1</td>
</tr>
<tr>
<td>0</td>
<td>&gt;0</td>
<td>1</td>
<td>Immediate Type 2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Immediate Type 3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>Floating Reg Source [FRO]</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>Undefined (will not generate UII)</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>Floating Reg source [FR1]</td>
</tr>
<tr>
<td>0</td>
<td>4-7</td>
<td>2</td>
<td>Undefined (will not generated UII)</td>
</tr>
<tr>
<td>0</td>
<td>—</td>
<td>3</td>
<td>Undefined (will not generate UII)</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>D</th>
<th>—</th>
<th>8</th>
<th>9</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Require a 16-bit literal (no L suffix)

**Immediate type 2**

<table>
<thead>
<tr>
<th>—</th>
<th>D</th>
<th>8</th>
<th>9</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>—</th>
<th>D</th>
<th>8</th>
<th>9</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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≤240 (R-Mode), or D≤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.


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:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Addressing Bits</th>
<th>Size of Addressable Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>16S</td>
<td>14</td>
<td>16K</td>
</tr>
<tr>
<td>32S</td>
<td>15</td>
<td>32K</td>
</tr>
<tr>
<td>32R</td>
<td>15</td>
<td>32K</td>
</tr>
<tr>
<td>64R</td>
<td>16</td>
<td>64K</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Type</th>
<th>No. Words</th>
<th>(S^1)</th>
<th>(D^2)</th>
<th>(CB^3)</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>1</td>
<td>0</td>
<td>0 – 777</td>
<td>—</td>
<td>SR</td>
</tr>
<tr>
<td>Sector Relative</td>
<td>1</td>
<td>1</td>
<td>0 – 777</td>
<td>—</td>
<td>S</td>
</tr>
<tr>
<td>Procedure Relative</td>
<td>1</td>
<td>1</td>
<td>-241 to +255</td>
<td>—</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-224 to +255</td>
<td>—</td>
<td>V</td>
</tr>
<tr>
<td>Stack Postincrement/</td>
<td>1</td>
<td>1</td>
<td>-256 to -241</td>
<td>2,3</td>
<td>R</td>
</tr>
<tr>
<td>Predecrement Relative</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Register</td>
<td>1</td>
<td>0</td>
<td>0 – 777</td>
<td>—</td>
<td>V</td>
</tr>
<tr>
<td>Long Reach</td>
<td>2</td>
<td>1</td>
<td>-256 to -241</td>
<td>0.2</td>
<td>R</td>
</tr>
<tr>
<td>Stack Relative</td>
<td>2</td>
<td>1</td>
<td>-256 to -241</td>
<td>1.3</td>
<td>R</td>
</tr>
<tr>
<td>Base Registers</td>
<td>2</td>
<td>1</td>
<td>-256 to -224</td>
<td>—</td>
<td>V</td>
</tr>
</tbody>
</table>

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=0, the nine bit displacement field is a location in the current sector. When S=1, 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.

<table>
<thead>
<tr>
<th>Assembler Notation</th>
<th>Location of ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LDA ADDR</strong></td>
<td><strong>Same Sector</strong></td>
</tr>
<tr>
<td>S=0</td>
<td>S=1</td>
</tr>
<tr>
<td>I=0</td>
<td>I=0</td>
</tr>
<tr>
<td>D=location in sector 0</td>
<td>D=displacement in same sector.</td>
</tr>
</tbody>
</table>

**LDA ADDR,***

<table>
<thead>
<tr>
<th>S=0</th>
<th>S=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>I=1</td>
<td>I=1</td>
</tr>
</tbody>
</table>

D=Location in sector 0 which contains a pointer defined by the program.

D=Location in same sector. It must contain a pointer defined by the program.

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.

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, 32R) bit indirect pointer to ADDR. Not permitted in 64R.
MEMORY REFERENCE INSTRUCTION FORMATS

**BASIC (one word, S-bit=0)**

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>OP</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- **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.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>I</th>
<th>X</th>
<th>S</th>
<th>D</th>
<th>EA</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>16S</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 to '777</td>
<td>0</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 to '777</td>
<td>0</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 to '777</td>
<td>I[0</td>
<td>D]</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0 to '777</td>
<td>I[0</td>
<td>D</td>
</tr>
<tr>
<td><strong>32S, 32R, 64R</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 to '777</td>
<td>0</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 to '777</td>
<td>0</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 to '777</td>
<td>I[0</td>
<td>D]</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0 to '777</td>
<td>I[0</td>
<td>D</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>OP</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- **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 concatenating 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

<table>
<thead>
<tr>
<th>Mode</th>
<th>I</th>
<th>X</th>
<th>S</th>
<th>D</th>
<th>EA</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>16S</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 to '777</td>
<td>P</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 to '777</td>
<td>P</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0 to '777</td>
<td>I(P</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 to '777</td>
<td>I(P</td>
<td>Indirect, preindexed</td>
</tr>
<tr>
<td>32S</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 to '777</td>
<td>P</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 to '777</td>
<td>P</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0 to '777</td>
<td>I(P</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 to '777</td>
<td>I(P</td>
<td>Indirect, postindexed</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>OP</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Mode</th>
<th>I</th>
<th>X</th>
<th>S</th>
<th>D</th>
<th>EA</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>32R, 64R</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-240 to +255</td>
<td>P+D</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-240 to +255</td>
<td>P+D+X</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-240 to +255</td>
<td>I(P+D)</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-240 to +255</td>
<td>I(P+D)+X</td>
<td>Indirect, postindexed</td>
</tr>
<tr>
<td>64V</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-224 to +255</td>
<td>P+D</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-224 to +255</td>
<td>P+D+X</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-224 to +255</td>
<td>I(P+D)</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-224 to +255</td>
<td>I(P+D)+X</td>
<td>Indirect, postindexed</td>
</tr>
</tbody>
</table>
STACK PREDECREMENT, POSTINCREMENT (One word, S-bit=1) 32R, 64R

<table>
<thead>
<tr>
<th>1</th>
<th>X</th>
<th>OP</th>
<th>11000</th>
<th>XX</th>
<th>CB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6 7</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Addressing</th>
<th>Mode</th>
<th>I</th>
<th>X</th>
<th>S</th>
<th>CB</th>
<th>EA</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32R, 64R</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>SP</td>
<td>Postincrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>I(SP)+X</td>
<td>Postincrement, indirect, post-indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>I(SP)</td>
<td>Postincrement, indirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>SP-1</td>
<td>Predecrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>I(SP-1)+X</td>
<td>Predecrement, indirect, post-indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>I(SP-1)</td>
<td>Predecrement, indirect</td>
</tr>
</tbody>
</table>

Note

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

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

<table>
<thead>
<tr>
<th>1</th>
<th>X</th>
<th>OP</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6 7</td>
<td>8 16</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>S</th>
<th>D</th>
<th>Address</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>'0'-'7</td>
<td>register location</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'10-'377</td>
<td>SB+D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'400-'777</td>
<td>LB+D</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>'0-'377 if D+X&lt;10 then</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EA=register location else SB+D+X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'400-'777</td>
<td>LB+D+X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>'0-'7</td>
<td>I(REG)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'10-'777</td>
<td>I(PB(D)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>'0-'77</td>
<td>I[PBD]+X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'100-'777</td>
<td>I[PBD]+X</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>OP</th>
<th>11000</th>
<th>XX</th>
<th>CB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

A

<table>
<thead>
<tr>
<th></th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>I (Bit 1)</td>
<td>Indirect Bit</td>
</tr>
<tr>
<td>X (Bit 2)</td>
<td>Index Bit</td>
</tr>
<tr>
<td>OP (Bits 3-6)</td>
<td>Opcode</td>
</tr>
<tr>
<td>Bits 7-12</td>
<td>110000</td>
</tr>
<tr>
<td>XX (Bits 13-14)</td>
<td>Opcode extension</td>
</tr>
<tr>
<td>CB (Bits 15-16)</td>
<td>Class Bits</td>
</tr>
<tr>
<td>A (Bits 17-32)</td>
<td>Address word</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>I</th>
<th>X</th>
<th>S</th>
<th>CB</th>
<th>EA</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>32R, 64R</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A+X</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>I(A)</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>I(A)+X</td>
<td>Indirect, preindexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>I(A)+X</td>
<td>Indirect, postindexed</td>
</tr>
</tbody>
</table>
STACK RELATIVE (Two Word, S-bit = 1)

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>OP</th>
<th>11000</th>
<th>XX</th>
<th>CB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>12</td>
</tr>
</tbody>
</table>

\[ A \]

| 17 | 32 |

I (Bit 1)  
X (Bit 2)  
OP (Bits 3-6)  
Bits 7-12  
XX (Bits 13-14)  
CB (Bits 15-16)  
A (Bits 17-32)

Indirect Bit  
Index Bit  
Opcode  
110000  
Opcode extension  
Class Bits  
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.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>I</th>
<th>X</th>
<th>S</th>
<th>CB</th>
<th>EA</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>32R, 64R</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A+SP</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A+SP+X</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>I[A+SP]</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>I[A+SP+X]</td>
<td>Indirect, preindexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>I[A+SP]+X</td>
<td>Indirect, postindexed</td>
</tr>
</tbody>
</table>

TWO WORD MEMORY REFERENCE

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>OP</th>
<th>11000</th>
<th>Y</th>
<th>XX</th>
<th>BR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3 - 6</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

\[ A \]

I (Bit 1)  
X (Bit 2)  
OP (Bit 3-6)  
Bits 7-12  
Y (Bit 12)  
XX (Bits 13-14)  
BR (Bits 15-16)  
A (Bits 17-32)

Indirect bit  
X bit  
Opcode  
110000  
Y bit  
Opcode extension  
Base register: 00=PB, 01=SB, 10=LB, 11=XB  
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>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>Y</th>
<th>BR</th>
<th>Effective Address</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>PB D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>SB + D</td>
<td></td>
<td>Direct</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>LB + D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>XB + D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>PBi D + Y</td>
<td>Indexed by Y</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>SBi D + Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>LBi D + Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>XBi D + Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>PBi D + X</td>
<td>Indexed by X</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>SBi D + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>LBi D + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>XBi D + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>[PB D]</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>[SB D]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>[LB D]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>[XB D]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[PB D + Y]</td>
<td>Pre-indexed by Y</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>[SB D + Y]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>[LB D + Y]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>[XB D + Y]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>[PB D] + Y</td>
<td>Post-indexed by Y</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>[SB D] + Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>[LB D] + Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>[XB D] + Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>[PB D] + X</td>
<td>Pre-indexed by X</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>[SB D] + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>[LB D] + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>[XB D] + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>[PB D] + X</td>
<td>Post-indexed by X</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>[SB D] + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>[LB D] + X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>[XB D] + X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>opcode</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Instruction

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>14-bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Indirect address word

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

Indirection: Multiple levels.

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>S</th>
<th>D</th>
<th>EA</th>
<th>Assembler Notation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0D</td>
<td>LDA ADDR</td>
<td>Direct</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0D+X</td>
<td>LDA ADDR,1</td>
<td>Indexed</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0D</td>
<td>LDA ADDR,*</td>
<td>Indirect</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0D+X</td>
<td>LDA ADDR,1*</td>
<td>Indirect, preindexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>D</td>
<td>LDA ADDR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>D+X</td>
<td>LDA ADDR,1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>IP</td>
<td>D</td>
<td>LDA ADDR,*</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>IP</td>
<td>D+X</td>
<td>LDA ADDR,1*</td>
</tr>
</tbody>
</table>

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:

```
I | X | opcode | S | D
---|---|--------|---|---
 1 | 2 | 3      | 6 | 7 | 8-16
```

Instruction Word

```
I | 15-bit address
---|----------------
 1 | 2 | 16
```

Indirect address word

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.

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>S</th>
<th>D</th>
<th>EA</th>
<th>Assembler</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 to ’777</td>
<td>0D</td>
<td>LDA ADDR</td>
<td>Direct</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 to ’777</td>
<td>0D+X</td>
<td>LDA ADDR,1</td>
<td>Indexed</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 to ’777</td>
<td>I[0D]</td>
<td>LDA ADDR,*</td>
<td>Indirect</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0 to ’777</td>
<td>I[0D+X]</td>
<td>LDA ADDR,1*</td>
<td>Indirect, preindexed</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100 to ’777</td>
<td>I[0D]+X</td>
<td>LDA ADDR,*1</td>
<td>Indirect, postindexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 to ’777</td>
<td>P.D</td>
<td>LDA ADDR</td>
<td>Direct</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 to ’777</td>
<td>P.D+X</td>
<td>LDA ADDR,1</td>
<td>Indexed</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0 to ’777</td>
<td>I[P.D]</td>
<td>LDA ADDR,*</td>
<td>Indirect</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 to ’777</td>
<td>I[P.D]+X</td>
<td>LDA ADDR,1*</td>
<td>Indirect, postindexed</td>
</tr>
</tbody>
</table>

P: Contents of program counter prior to instruction fetch (pointing at instruction).

0D: 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:

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>opcode</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Instruction Word:
S=0 or S=1
D≥-240

Instruction Word:
S=1
D<-240

Address Word:
Long Reach and
Stack Relative

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>opcode</th>
<th>110000</th>
<th>XX</th>
<th>CB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>12</td>
</tr>
</tbody>
</table>

Indirect Address
Word

Indexing: One level.

Indirection: Multiple levels.

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>S</th>
<th>CB</th>
<th>D</th>
<th>EA</th>
<th>Assembler</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 to '777</td>
<td>LDA ADDR</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 to '777</td>
<td>LDA ADDR,1</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 to '777</td>
<td>LDA ADDR,*</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 to '777</td>
<td>LDA ADDR,1*</td>
<td>Indirect, preindexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100 to '777</td>
<td>I(0D)+X</td>
<td>LDA ADDR,*1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-240 to +255</td>
<td>P+D</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-240 to +255</td>
<td>P+D+X</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-240 to +255</td>
<td>I(P+D)</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-240 to +255</td>
<td>I(P+D)+X</td>
<td>Indirect, postindexed</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>SP</td>
<td>Postincrement</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>I(SP)+X</td>
<td>Postincrement, indirect, postindexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>I(SP)</td>
<td>Postincrement, indirect</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>SP-1</td>
<td>Predecrement</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>I(SP-1)+X</td>
<td>Predecrement, indirect, postindexed</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>I(SP-1)</td>
<td>Predecrement, indirect</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Assembler</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>ADDR</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>ADDR,1</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>ADDR,*</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>ADDR,1*</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>ADDR,*1</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>@+</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>@+,*1</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>@+,*</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>-@</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>-@,*1</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>-@,*</td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>A</td>
<td>LDA% ADDR</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>A+X</td>
<td>LDA% ADDR,X</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>I(A)</td>
<td>LDA% ADDR,*</td>
</tr>
<tr>
<td>1 1 1 2</td>
<td>I[A+X]</td>
<td>LDA% ADDR,X*</td>
</tr>
<tr>
<td>1 1 1 2</td>
<td>I(A)+X</td>
<td>LDA% ADDR,*X</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>A+SP</td>
<td>LDA @+ADDR</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>A+SP+X</td>
<td>LDA @+ADDR,X</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>I(A+SP)</td>
<td>LDA @+ADDR,*</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>I[A+SP+X]</td>
<td>LDA @+ADDR,X*</td>
</tr>
<tr>
<td>1 1 1 3</td>
<td>I(A+SP)+X</td>
<td>LDA @+ADDR,*X</td>
</tr>
</tbody>
</table>

**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

EA = (S)
S = (S) + 1

S = (S) - 1
EA = (S)

X = 0?

I = 1?

I = [EA] bit 1
EA = [EA] bits 2-16

X = 1?

EA = EA + (X)

DONE
64R summary

Address length: 16 bits; 64K word address space

Format:

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>opcode</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
</table>
| 1 | 2 | 3      | 6 | 7 | 8 — 16 | Instruction Word  
S=0 or S=1  
D ≤ -240 |

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>opcode</th>
<th>CB</th>
<th>XX</th>
</tr>
</thead>
</table>
| 1 | 2 | 3      | 6  | 7  | 12 | 13-14 | 15-16 | Instruction  
S=1  
D < -240 |

| A | 17 | 32 |

Address Word:  
Long Reach and  
Stack Relative  

| 16-bit address | 1 | 16 |

Indirect Address  
Word

Indexing: One level.

Indirection: One level.

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>S</th>
<th>CB</th>
<th>D</th>
<th>EA</th>
<th>Assembler Notation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>0 to '777</td>
<td>0D</td>
<td>LDA ADDR</td>
<td>Direct</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>—</td>
<td>0 to '777</td>
<td>0D+X</td>
<td>LDA ADDR,1</td>
<td>Indexed</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>0 to '777</td>
<td>I(0D)</td>
<td>LDA ADDR,*</td>
<td>Indirect</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>—</td>
<td>0 to '77</td>
<td>I(0D+X)</td>
<td>LDA ADDR,1*</td>
<td>Indirect, preindexed</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>—</td>
<td>'100 to '777</td>
<td>I(0D)+X</td>
<td>LDA ADDR*1</td>
<td>Indirect, postindexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>-240 to +255</td>
<td>P+D</td>
<td>LDA ADDR</td>
<td>Direct</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>-240 to +255</td>
<td>P+D+X</td>
<td>LDA ADDR,1</td>
<td>Indexed</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>—</td>
<td>-240 to +255</td>
<td>I(P+D)</td>
<td>LDA ADDR,*</td>
<td>Indirect</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>-240 to +255</td>
<td>I(P+D)+X</td>
<td>LDA ADDR,*1</td>
<td>Indirect, postindexed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>SP</td>
<td>LDA @+</td>
<td>Postincrement</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>I(SP)+X</td>
<td>LDA @+,*1</td>
<td>Postincrement, indirect, postindexed</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>I(SP)</td>
<td>LDA @+,*</td>
<td>Postincrement, indirect</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>SP-1</td>
<td>LDA @-</td>
<td>Predecrement</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>I(SP-1)+X</td>
<td>LDA @-,*1</td>
<td>Predecrement, indirect, postindexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Operation</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 3</td>
<td>I(SP-1)</td>
<td>LDA -@,*</td>
<td>Predecrement, indirect</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>A</td>
<td>LDA% ADDR</td>
<td>Direct, long reach</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>A+X</td>
<td>LDA% ADDR,X</td>
<td>Indexed, long reach</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>I(A)</td>
<td>LDA% ADDR,*</td>
<td>Indirect, long reach</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>I(A+X)</td>
<td>LDA% ADDR,X*</td>
<td>Indirect, preindexed long reach</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>I(A)+X</td>
<td>LDA% ADDR,*X</td>
<td>Indirect, postindexed long reach</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>A+SP</td>
<td>LDA @+ADDR</td>
<td>Direct, stack relative</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>A+SP+X</td>
<td>LDA @+ADDR,X</td>
<td>Indexed, stack relative</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>I(A+SP)</td>
<td>LDA @+ADDR,*</td>
<td>Indirect, stack relative</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>I(A+SP+X)</td>
<td>LDA @+ADDR,X*</td>
<td>Indirect, preindexed, stack relative</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 3</td>
<td>I(A+SP)+X</td>
<td>LDA @+ADDR,*X</td>
<td>Indirect, postindexed, stack relative</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**P**
Contents of program counter after instruction fetch (pointing at instruction plus 1).

**0D**
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:

```
<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>OP</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7-16</td>
</tr>
</tbody>
</table>
```

Instruction Word

Indirect address word

Indexing: One level

Indirection: One level

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>S</th>
<th>D</th>
<th>EA</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-224 to +255</td>
<td>P+D</td>
<td>Direct</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-224 to +255</td>
<td>P+D+X</td>
<td>Indexed</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-224 to +255</td>
<td>I(P+D)</td>
<td>Indirect</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-224 to +255</td>
<td>I(P+D)+X</td>
<td>Indirect, postindexed</td>
</tr>
</tbody>
</table>

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:

```
<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>OP</th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7-16</td>
</tr>
</tbody>
</table>
```

Instruction Word

Indirect address word

Indexing: One level

Indirection: One level
### 64V Two Word Memory Reference

**Address length:** 28 bits; 4096 64K segments

**Format:**

<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>S</th>
<th>D</th>
<th>EA</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0~7</td>
<td>register location</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'10~777</td>
<td>SB+D</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'400~777</td>
<td>LB+D</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0~377</td>
<td>if D+X&lt;10 then EA = register location * else SB+D+X</td>
<td>Indexed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'400~777</td>
<td>LB+D+X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0~7</td>
<td>I(REG)</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'10~777</td>
<td>I(PB/D)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0~77</td>
<td>I(PB/D+X)</td>
<td>Indirect, preindexed postindexed</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>'100~777</td>
<td>I(PB/D)+X</td>
<td></td>
</tr>
</tbody>
</table>

- **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.

**Indexing:** X and Y

**Indirection:** 48 bit word

---

1 March, 1979
<table>
<thead>
<tr>
<th>I</th>
<th>X</th>
<th>Y</th>
<th>BR</th>
<th>Effective Address</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>PB[D]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>SB[D]</td>
<td>Direct</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>LB[D]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>XB[D]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>PB[D]+Y</td>
<td>Indexed by Y</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>SB[D]+Y</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>LB[D]+Y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>XB[D]+Y</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>PB[D]+X</td>
<td>Indexed by X</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>SB[D]+X</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>LB[D]+X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>XB[D]+X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>I(PB[D])</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>I(SB[D])</td>
<td>Indirect</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>I(LB[D])</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>I(XB[D])</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>I(PB[D]+Y)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>I(SB[D]+Y)</td>
<td>Preindexed by Y</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>I(LB[D]+Y)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>I(XB[D]+Y)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>I(PB[D]+Y)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>I(SB[D]+Y)</td>
<td>Postindexed by Y</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>I(LB[D]+Y)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>I(XB[D]+Y)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>I(PB[D]+X)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>I(SB[D]+X)</td>
<td>Preindexed by X</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>I(LB[D]+X)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>I(XB[D]+X)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>I(PB[D]+X)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>I(SB[D]+X)</td>
<td>Postindexed by X</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>I(LB[D]+X)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>I(XB[D]+X)</td>
<td></td>
</tr>
</tbody>
</table>
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)
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=001013, 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:

Word 1 =

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Word 2 =

16-bit word address in current procedure segment

1 16

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Branch if LT bit set and EQ bit cleared</td>
</tr>
<tr>
<td>≤</td>
<td>Branch if LT bit set or EQ bit set</td>
</tr>
<tr>
<td>=</td>
<td>Branch if EQ bit set</td>
</tr>
<tr>
<td>≠</td>
<td>Branch if EQ bit cleared</td>
</tr>
<tr>
<td>≥</td>
<td>Branch if LT bit cleared or EQ bit set</td>
</tr>
<tr>
<td>&gt;</td>
<td>Branch if LT bit cleared and EQ bit cleared</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCLT addr</td>
<td>If CC&lt;, then addr→PC</td>
<td>141604</td>
</tr>
<tr>
<td>BCLE addr</td>
<td>If CC≤, then addr→PC</td>
<td>141600</td>
</tr>
<tr>
<td>BCEQ addr</td>
<td>If CC=, then addr→PC</td>
<td>141602</td>
</tr>
<tr>
<td>BCNE addr</td>
<td>If CC≠, then addr→PC</td>
<td>141603</td>
</tr>
<tr>
<td>BCGE addr</td>
<td>If CC≥, then addr→PC</td>
<td>141605</td>
</tr>
<tr>
<td>BCGT addr</td>
<td>If CC&gt;, then addr→PC</td>
<td>141601</td>
</tr>
</tbody>
</table>
Test magnitude condition and branch

These instructions have the following format:

\[
\begin{cases}
\text{LT} \\
\text{LE} \\
\text{EQ} \\
\text{GE} \\
\text{GT}
\end{cases}
\]

Branch to addr if \( \text{L}=1 \) and condition code

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).

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMLT addr</td>
<td>If ( \text{L}=1 ) and ( \text{CC}_&lt; ), then addr( \rightarrow )PC</td>
<td>141707</td>
</tr>
<tr>
<td>BMLE addr</td>
<td>If ( \text{L}=1 ) and ( \text{CC}_\leq ), then addr( \rightarrow )PC</td>
<td>141711</td>
</tr>
<tr>
<td>BMEQ addr</td>
<td>If ( \text{L}=1 ) and ( \text{CC}_= ), then addr( \rightarrow )PC</td>
<td>141602</td>
</tr>
<tr>
<td>BMNE addr</td>
<td>If ( \text{L} \neq 1 ) and ( \text{CC}_\neq ), then addr( \rightarrow )PC</td>
<td>141603</td>
</tr>
<tr>
<td>BMGE addr</td>
<td>If ( \text{L}=1 ) and ( \text{CC}_\geq ), then addr( \rightarrow )PC</td>
<td>141606</td>
</tr>
<tr>
<td>BMGT addr</td>
<td>If ( \text{L}=1 ) and ( \text{CC}_&gt; ), then addr( \rightarrow )PC</td>
<td>141710</td>
</tr>
</tbody>
</table>

Test C-bit and branch

Branch if C-bit

\[
\begin{cases}
0 \\
1
\end{cases}
\]

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

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

Test L-bit

Test L-bit

\[
\begin{cases}
0 \\
1
\end{cases}
\]

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

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

Branch on register

These instructions have the following format:

\[
\begin{cases}
\text{A-Register (blank)} \\
\text{L-Register (L)} \\
\text{Floating-Register (F)}
\end{cases}
\]

Branch if

\[
\begin{cases}
\text{LT} \\
\text{LE} \\
\text{EQ} \\
\text{NE} \\
\text{GE} \\
\text{GT}
\end{cases}
\]

\[
0
\]

1 March, 1979

11-3

FDR 3059
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, OPCODE = see chart below. C = unchanged, L = unchanged, CC = result.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLT addr</td>
<td>If A&lt;0, then addr→PC</td>
<td>140614</td>
</tr>
<tr>
<td>BLE addr</td>
<td>If A≤0, then addr→PC</td>
<td>140610</td>
</tr>
<tr>
<td>BEQ addr</td>
<td>If A=0, then addr→PC</td>
<td>140612</td>
</tr>
<tr>
<td>BNE addr</td>
<td>If A≠0, then addr→PC</td>
<td>140613</td>
</tr>
<tr>
<td>BGE addr</td>
<td>If A≥0, then addr→PC</td>
<td>140615</td>
</tr>
<tr>
<td>BGT addr</td>
<td>If A&gt;0, then addr→PC</td>
<td>140611</td>
</tr>
<tr>
<td>BLT addr</td>
<td>If L&lt;0, then addr→PC</td>
<td>140614</td>
</tr>
<tr>
<td>BLE addr</td>
<td>If L≤0, then addr→PC</td>
<td>140700</td>
</tr>
<tr>
<td>BEQ addr</td>
<td>If L=0, then addr→PC</td>
<td>140702</td>
</tr>
<tr>
<td>BNE addr</td>
<td>If L≠0, then addr→PC</td>
<td>140703</td>
</tr>
<tr>
<td>BGE addr</td>
<td>If L≥0, then addr→PC</td>
<td>140615</td>
</tr>
<tr>
<td>BGT addr</td>
<td>If L&gt;0, then addr→PC</td>
<td>140701</td>
</tr>
<tr>
<td>BFLT addr</td>
<td>If F&lt;0, then addr→PC</td>
<td>141614</td>
</tr>
<tr>
<td>BFLE addr</td>
<td>If F≤0, then addr→PC</td>
<td>141610</td>
</tr>
<tr>
<td>BFEQ addr</td>
<td>If F=0, then addr→PC</td>
<td>141612</td>
</tr>
<tr>
<td>BFNE addr</td>
<td>If F≠0, then addr→PC</td>
<td>141613</td>
</tr>
<tr>
<td>BFGE addr</td>
<td>If F≥0, then addr→PC</td>
<td>141615</td>
</tr>
<tr>
<td>BFGT addr</td>
<td>If F&gt;0, then addr→PC</td>
<td>141611</td>
</tr>
</tbody>
</table>

**Increment or decrement X or Y and branch**

\[
\begin{align*}
\text{Increment} & \quad \begin{cases}
X \\
Y
\end{cases} \\
\text{Decrement} & \quad \begin{cases}
X \\
Y
\end{cases}
\end{align*}
\]

by 1 then branch to addr if result ≠ 0

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIX addr</td>
<td>X+1→X; if X≠0 then addr→PC</td>
<td>141334</td>
</tr>
<tr>
<td>BIY addr</td>
<td>Y+1→Y; if Y≠0 then addr→PC</td>
<td>141324</td>
</tr>
<tr>
<td>BDX addr</td>
<td>X−1→X; if X≠0 then addr→PC</td>
<td>140734</td>
</tr>
<tr>
<td>BDY addr</td>
<td>Y−1→Y; if Y≠0 then addr→PC</td>
<td>140724</td>
</tr>
</tbody>
</table>

**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 ('20).

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. |

<table>
<thead>
<tr>
<th>Condition code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Field 0=field 1</td>
</tr>
<tr>
<td>LT</td>
<td>Field 0&lt;field 1</td>
</tr>
<tr>
<td>GT (LT and EQ)</td>
<td>Field 0&gt;field 1</td>
</tr>
</tbody>
</table>

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

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:

<table>
<thead>
<tr>
<th>L</th>
<th>O</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>7-8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcode (E)</th>
<th>Mnemonic</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPC</td>
<td>Copy M characters from source to destination.</td>
</tr>
<tr>
<td>1</td>
<td>INL</td>
<td>Insert literal character M.</td>
</tr>
<tr>
<td>2</td>
<td>SKC</td>
<td>Skip M characters.</td>
</tr>
<tr>
<td>3</td>
<td>BLK</td>
<td>Supply M blanks (ASCII ‘240).</td>
</tr>
</tbody>
</table>

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

\[0 \rightarrow A(1-8)\]

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

\[0 \rightarrow A(9-16)\]

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

\[0 \rightarrow A\]

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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Code</th>
<th>Size of Decimal Digit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leading Separate Sign</td>
<td>0</td>
<td>8</td>
<td>A plus sign (+) or a space represents a positive number. Operations generate +. A minus sign (-) represents negative number.</td>
</tr>
<tr>
<td>Trailing Separate Sign</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Packed Decimal</td>
<td>3</td>
<td>4</td>
<td>Use 4-bit nibble to represent each digit, followed by sign nibble. Requires odd number of digits and must start on byte boundary.</td>
</tr>
<tr>
<td>Leading Embedded Sign</td>
<td>4</td>
<td>8</td>
<td>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. Embedded sign characters are as follows:</td>
</tr>
<tr>
<td>Trailing Embedded Sign</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Digit</th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,+</td>
<td>-,</td>
</tr>
<tr>
<td>1</td>
<td>1 A</td>
<td>J</td>
</tr>
<tr>
<td>2</td>
<td>2 B</td>
<td>K</td>
</tr>
<tr>
<td>3</td>
<td>3 C</td>
<td>L</td>
</tr>
<tr>
<td>4</td>
<td>4 D</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>5 E</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>6 F</td>
<td>O</td>
</tr>
<tr>
<td>7</td>
<td>7 G</td>
<td>P</td>
</tr>
<tr>
<td>8</td>
<td>8 H</td>
<td>Q</td>
</tr>
<tr>
<td>9</td>
<td>9 I</td>
<td>R</td>
</tr>
</tbody>
</table>
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 | 1-6 | B | 7 | C | 8 | T | 9 | D | 10 | E | 11 | F | 12 | G | 13 | H | 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:

<table>
<thead>
<tr>
<th>Fault</th>
<th>DEX Type (High)</th>
<th>Sub Code (Low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Divide by zero</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Conversion</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

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

- XAD Decimal add
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.

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>+ Source + Destination</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>+ Source − Destination</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>− Source + Destination</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>− Source − Destination</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>A</th>
<th>E</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>14-16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>30-32</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14-16</td>
<td>17</td>
<td>22</td>
<td>23-29</td>
<td>30-32</td>
</tr>
</tbody>
</table>

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:

\[ \begin{align*}
\text{GT} &= \text{Field } 2 > \text{Field } 1 \\
\text{EQ} &= \text{Field } 2 = \text{Field } 1 \\
\text{LT} &= \text{Field } 2 < \text{Field } 1
\end{align*} \]

\text{MODES}=V, \text{FORMAT}=\text{DECI, OPCODE}=001102, \text{C}=\text{unchanged, L}=\text{unchanged, CC}=\text{result.}

\begin{itemize}
\item \textbf{XDTB} \hspace{1em} \text{Decimal to binary conversion}
\end{itemize}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
A & E & H \\
\hline
1-6 & 7 & 13 & 14-16 & 17 & 29 & 30-32 \\
\hline
\end{tabular}
\end{table}

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

\begin{itemize}
\item 0 \hspace{1em} 16 \text{ Bits, returned in A.}
\item 1 \hspace{1em} 32 \text{ Bits, returned in L.}
\item 2 \hspace{1em} 64 \text{ Bits, returned in L/E.}
\end{itemize}

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. \text{MODES}=V, \text{FORMAT}=\text{DECI, OPCODE}=001146, \text{C}=\text{unchanged, L}=\text{unchanged, CC}=\text{unspecified.}

\begin{itemize}
\item \textbf{XDV} \hspace{1em} \text{Decimal divide}
\end{itemize}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
A & B & C \\
\hline
1-6 & 7 & 8 & 9 & 10 & 11 & 32 \\
\hline
\end{tabular}
\end{table}

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 \leq source. \text{MODES}=V, \text{FORMAT}=\text{DECI, OPCODE}=001107, \text{C}=\text{unchanged, L}=\text{unchanged, CC}=\text{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:

```
<table>
<thead>
<tr>
<th>L</th>
<th>0</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2-4</td>
<td>5-8</td>
<td>9</td>
</tr>
</tbody>
</table>
```

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**

```
<table>
<thead>
<tr>
<th>A</th>
<th>—</th>
<th>B</th>
<th>C</th>
<th>—</th>
<th>T</th>
<th>—</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>7-8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14-16</td>
<td>17-22</td>
<td>23-29</td>
<td>30-32</td>
</tr>
</tbody>
</table>
```

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.**

1 March, 1979

11-13
### Table 11-2. Edit Sub-operations

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ZS</td>
<td>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.</td>
</tr>
<tr>
<td>01</td>
<td>IL</td>
<td>Insert literal M in destination field.</td>
</tr>
<tr>
<td>02</td>
<td>SS</td>
<td>Set zero suppress character to M.</td>
</tr>
<tr>
<td>03</td>
<td>ICS</td>
<td>Insert literal M if the significance flag set; otherwise insert zero suppress character.</td>
</tr>
<tr>
<td>04</td>
<td>ID</td>
<td>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.</td>
</tr>
<tr>
<td>05</td>
<td>ICM</td>
<td>Insert M if sign is minus; otherwise insert zero suppress character.</td>
</tr>
<tr>
<td>06</td>
<td>ICP</td>
<td>Insert M if sign is plus; otherwise insert zero suppress character.</td>
</tr>
<tr>
<td>07</td>
<td>SFC</td>
<td>Set floating character to M.</td>
</tr>
<tr>
<td>10</td>
<td>SFP</td>
<td>Set floating character to M if sign plus; otherwise set floating edit character to zero suppress character.</td>
</tr>
<tr>
<td>11</td>
<td>SFM</td>
<td>Set floating character to M if sign minus; otherwise set floating edit character to zero suppress character.</td>
</tr>
<tr>
<td>12</td>
<td>SFS</td>
<td>Set floating character to sign.</td>
</tr>
<tr>
<td>13</td>
<td>JZ</td>
<td>Jump M+1 locations ahead in edit sub program if source field equals zero.</td>
</tr>
<tr>
<td>14</td>
<td>FS</td>
<td>Fill next M characters with zero suppress character.</td>
</tr>
<tr>
<td>15</td>
<td>SF</td>
<td>Set significance flag.</td>
</tr>
<tr>
<td>16</td>
<td>IS</td>
<td>Insert sign.</td>
</tr>
</tbody>
</table>

**X MV** Decimal move

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14-16</td>
</tr>
</tbody>
</table>

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

\[ L + \text{FAR} \rightarrow \text{FAR} \]

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

\[ [\text{EA}]_{48} \rightarrow \text{FAR} \]

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

\[ \text{DATA} \rightarrow \text{FLR} \]

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

\[ \text{FAR} \rightarrow [\text{EA}]_{32} \text{ or } [\text{EA}]_{48} \]

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

\[ \text{FLR} \rightarrow \text{L} \]

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

\[ \text{L} \rightarrow \text{FLR} \]

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 –32639 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 \(\pm30\) bits or about \(\pm1\) 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**

<table>
<thead>
<tr>
<th>Register 11 (Precision)</th>
<th>Register 12</th>
<th>Type of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>Double</td>
<td></td>
</tr>
<tr>
<td>$100</td>
<td>$200</td>
<td>—</td>
</tr>
<tr>
<td>$101</td>
<td>$201</td>
<td>—</td>
</tr>
<tr>
<td>$102</td>
<td>—</td>
<td>(EA)</td>
</tr>
<tr>
<td>$103</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Overflow/Underflow (Exponent exceeds approx. 10 (\pm9800))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Division by zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attempt to store single precision exponent exceeding 8-bit memory format ( (&gt;127, &lt; -128))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attempt to form integer exceeding capacity. INT: A/B (30) bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTA: A (15) bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTL: L (31) bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note**

$ indicates hexadecimal codes
Table 11-4. Floating Point Mantissa and Exponent Ranges

<table>
<thead>
<tr>
<th>Field</th>
<th>Single Precision-Memory</th>
<th>Single Precision-Register</th>
<th>Double Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mantissa (two’s complement)</td>
<td>23 + Sign</td>
<td>31 + Sign</td>
<td>47 + Sign</td>
</tr>
<tr>
<td>Precision</td>
<td>±8,388.607</td>
<td>±2,147,483.647</td>
<td>±140,737,488,355,327</td>
</tr>
<tr>
<td>Exponent</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Range</td>
<td>-128 to +127</td>
<td>-32896 to +32639</td>
<td>-32896 to +32639</td>
</tr>
<tr>
<td></td>
<td>(10±38)</td>
<td>(10+9823,-9902)</td>
<td>(10+9823,-9902)</td>
</tr>
</tbody>
</table>

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

\[(EA)_{64}\to F\]

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

\[(EA)_{16^4}\to X\]

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

\[F\times(EA)_{64}\to F\]

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

\[F-(EA)_{64}\to F\]

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

\[F+(EA)_{32}\to F\]

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(AB)→F

Take the 31-bit integer in the combined AB 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 proceeding 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. 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)→AB
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
  \[
  \text{INT}(F)\rightarrow L
  \]

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. \(\text{MODES}=V, \text{FORMAT}=\text{GEN}, \text{OPCODE}=140533, C=\text{overflow}, L=\text{unspecified}, CC=\text{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
  \[
  A+1\rightarrow A
  \]

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. \(\text{MODES}=SRV, \text{FORMAT}=\text{GEN}, \text{OPCODE}=141206, C=\text{overflow}, L=\text{carry}, CC=\text{result}.

A2A  Add two to A
  \[
  A+2\rightarrow A
  \]

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. \(\text{MODES}=SRV, \text{FORMAT}=\text{GEN}, \text{OPCODE}=140304, C=\text{overflow}, L=\text{carry}, CC=\text{result}.

ACA  Add C-bit to A
  \[
  A+C\text{-bit}\rightarrow A
  \]

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. \(\text{MODES}=SRV, \text{FORMAT}=\text{GEN}, \text{OPCODE}=141216, C=\text{overflow}, L=\text{carry}, CC=\text{result}.

ADD addr  Add
  \[
  A+[EA]_{16}\rightarrow A
  \]

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, the 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 \(2^{*}15\). \(\text{MODES}=SRV, \text{FORMAT}=MR, \text{OPCODE}=06, C=\text{overflow}, L=\text{carry}, CC=\text{result}.

ADL addr  Add long
  \[
  L+[EA]_{32}\rightarrow L
  \]

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 \(2^{*}31\). \(\text{MODES}=V, \text{FORMAT}=MR, \text{OPCODE}=06\ 03, C=\text{overflow}, L=\text{carry}, CC=\text{result}.

FDR 3059
11–22
1 March, 1979
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


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

\[ A(1) \rightarrow C \text{-bit} 0 \rightarrow A(1) \]

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. \text{MODES} = \text{SRV}, \text{FORMAT} = \text{GEN}, \text{OPCODE} = 140320, C = \text{result}, L = \text{unspecified}, CC = \text{unchanged}.

DAD addr  Double add


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 \(\geq 2^{**30}\) or \(\leq -2^{**30}\), 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. \text{MODES} = \text{SR}, \text{FORMAT} = \text{MR}, \text{OPCODE} = 06, C = \text{overflow}, L = \text{carry}, CC = \text{result}.

DBL  Enter double precision mode

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

DIV addr  Divide

\[ A[B]/[EA]16 \rightarrow A:REM = B \]

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 \(A*\text{[addr]}+B\) 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. \text{MODES} = \text{SR}, \text{FORMAT} = \text{MR}, \text{OPCODE} = 17, C = \text{overflow}, L = \text{unspecified}, CC = \text{unspecified}.

DIV addr  Divide

\[ L/[EA]16 \rightarrow A:REM = B \]

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 \(A*\text{[addr]}+B\) 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. \text{MODES} = \text{V}, \text{FORMAT} = \text{MR}, \text{OPCODE} = 17, C = \text{overflow}, L = \text{unspecified}, CC = \text{unspecified}.

DSB addr  Double subtract


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 \(\geq 2^{**30}\) or \(\leq -2^{**30}\), 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**=un-
specified, **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
\[
A(2\text{-}16)\rightarrow B(2\text{-}16); 0\rightarrow B(1); A(1)\rightarrow A(2\text{-}16)
\]
Convert the 16-bit integer in register A to a 31-bit integer in A\(\overline{B}\) 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\(\overline{B}\) simplifies integer arithmetic. MODES=SR, FORMAT=GEN, OPCODE=000211, C=unchanged, L=unchanged, CC=unchanged.

PIDA | Position for integer divide
\[
A(1\text{-}16)\rightarrow L(17\text{-}32); A(1)\rightarrow A(2\text{-}16)
\]
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
\[
L\rightarrow E; L(1)\rightarrow L(2\text{-}32)
\]
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
\[
B(2\text{-}16)\rightarrow A(2\text{-}16)
\]
Convert the 31-bit integer in registers A\(\overline{B}\) 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
\[
L(17\text{-}31)\rightarrow A(1\text{-}16)
\]
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
\[
L(E)(33\text{-}64)\rightarrow L(1\text{-}32)
\]
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
\[
A-1\rightarrow A
\]
Subtract 1 from the 16-bit integer in register A and put the result into A. If the number decremented is \(2^{*}\times15\), set C and give a result of \(2^{*}\times15\)-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

\[-A\rightarrow A\]

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

\[-L\rightarrow L\]

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-diagnostics 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

\[ A \text{ AND } [EACH 16 \rightarrow A] \]

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.

<table>
<thead>
<tr>
<th>A BIT</th>
<th>Memory Bit</th>
<th>Resulting Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

MODES=SRV, FORMAT=MR, OPCODE=03. C=unchanged, L=unchanged, CC=unchanged.

► ANL addr AND long

\[ L \text{ AND } [EACH 32 \rightarrow L] \]

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

\[ \text{NOT. A} \rightarrow A \]

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

\[ A \text{ XOR } [EACH 16 \rightarrow A] \]

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.

<table>
<thead>
<tr>
<th>A BIT</th>
<th>Memory Bit</th>
<th>Resulting Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

MODES=SRV, FORMAT=MR, OPCODE=05. C=unchanged, L=unchanged, CC=unchanged.

► ERL addr Exclusive OR long

\[ L \text{ XOR } [EACH 32 \rightarrow L] \]

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.

\[
\begin{array}{c}
\text{A Register (Blank)} \\
\text{Condition Code (C)} \\
\text{L Register (L)} \\
\text{Floating Point (F)} \\
\text{Register}
\end{array} \quad \begin{array}{c}
\text{LT} \\
\text{LE} \\
\text{EQ} \\
\text{NE} \\
\text{GE} \\
\text{GT}
\end{array}
\] 0, then 1→A; else 0→A

A-Register test

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLT</td>
<td>If A &lt; 0, then 1→A; else 0→A</td>
<td>140410</td>
</tr>
<tr>
<td>LLE</td>
<td>If A \leq 0, then 1→A; else 0→A</td>
<td>140411</td>
</tr>
<tr>
<td>LEQ</td>
<td>If A = 0, then 1→A; else 0→A</td>
<td>140413</td>
</tr>
<tr>
<td>LNE</td>
<td>If A \neq 0, then 1→A; else 0→A</td>
<td>140412</td>
</tr>
<tr>
<td>LGE</td>
<td>If A \geq 0, then 1→A; else 0→A</td>
<td>140414</td>
</tr>
<tr>
<td>LGT</td>
<td>If A &gt; 0, then 1→A; else 0→A</td>
<td>140415</td>
</tr>
</tbody>
</table>

MODES=SRV, FORMAT=GEN, C=unchanged, L=unchanged, CC=result.

Condition code test

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGLT</td>
<td>If CC &lt; 0, then 1→A; else 0→A</td>
<td>141500</td>
</tr>
<tr>
<td>LCLE</td>
<td>If CC \leq 0, then 1→A; else 0→A</td>
<td>141501</td>
</tr>
<tr>
<td>LGEQ</td>
<td>If CC = 0, then 1→A; else 0→A</td>
<td>141503</td>
</tr>
<tr>
<td>LCNE</td>
<td>If CC \neq 0, then 1→A; else 0→A</td>
<td>141502</td>
</tr>
<tr>
<td>LGGE</td>
<td>If CC \geq 0, then 1→A; else 0→A</td>
<td>141504</td>
</tr>
<tr>
<td>LGGT</td>
<td>If CC &gt; 0, then 1→A; else 0→A</td>
<td>141505</td>
</tr>
</tbody>
</table>

MODES=V, FORMAT=GEN, C=unchanged, L=unchanged, CC=unchanged.

L register test

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLLL</td>
<td>If L &lt; 0, then 1→A; else 0→A</td>
<td>140410</td>
</tr>
<tr>
<td>LLLE</td>
<td>If L \leq 0, then 1→A; else 0→A</td>
<td>141511</td>
</tr>
<tr>
<td>LLEQ</td>
<td>If L = 0, then 1→A; else 0→A</td>
<td>141513</td>
</tr>
<tr>
<td>LLNE</td>
<td>If L \neq 0, then 1→A; else 0→A</td>
<td>141512</td>
</tr>
<tr>
<td>LLGE</td>
<td>If L \geq 0, then 1→A; else 0→A</td>
<td>140414</td>
</tr>
<tr>
<td>LLGT</td>
<td>If L &gt; 0, then 1→A; else 0→A</td>
<td>141515</td>
</tr>
</tbody>
</table>

MODES=V, FORMAT=GEN, C=unchanged, L=unchanged, CC=result.
Floating register test

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFLT</td>
<td>If F &lt; 0, then 1→A; else 0→A</td>
<td>141110</td>
</tr>
<tr>
<td>LFLE</td>
<td>If F ≤ 0, then 1→A; else 0→A</td>
<td>141111</td>
</tr>
<tr>
<td>LFEQ</td>
<td>If F = 0, then 1→A; else 0→A</td>
<td>141113</td>
</tr>
<tr>
<td>LFNE</td>
<td>If F ≠ 0, then 1→A; else 0→A</td>
<td>141112</td>
</tr>
<tr>
<td>LFGE</td>
<td>If F ≥ 0, then 1→A; else 0→A</td>
<td>141114</td>
</tr>
<tr>
<td>LFGT</td>
<td>If F &gt; 0, then 1→A; else 0→A</td>
<td>141115</td>
</tr>
</tbody>
</table>

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.

- **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:

<table>
<thead>
<tr>
<th>Information</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>AP</td>
</tr>
<tr>
<td>PA</td>
<td>Segment number page table</td>
</tr>
<tr>
<td>TVA</td>
<td>L</td>
</tr>
</tbody>
</table>

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 interrupt, 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. MODER=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. MODER=V, FORMAT=GEN, OPCODE=001710, C=unspecified, L=unspecified, CC=unspecified. Restricted instruction.

**MIA addr**  Microcode indirect A

Microcode entrance. MODER=V, FORMAT=MR, OPCODE=12 01, C=unchanged, L=unchanged, CC=unchanged.

**MIB addr**  Microcode indirect B

Microcode entrance. MODER=V, FORMAT=MR, OPCODE=13 01, C=unchanged, L=unchanged, CC=unchanged.

**NOP**  No operation

PC+1→PC

Do nothing, but go on to the next instruction. MODER=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 physical page number is provided right-justified in the L register. The high-order bit of L is set to indicate a complete purge. MODER=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:

<table>
<thead>
<tr>
<th>Word</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Save Mask</td>
</tr>
<tr>
<td>2-5</td>
<td>FALR1 (FAC)</td>
</tr>
<tr>
<td>6-9</td>
<td>FALR 0</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
</tr>
<tr>
<td>11-13</td>
<td>—</td>
</tr>
<tr>
<td>14</td>
<td>Y,S</td>
</tr>
<tr>
<td>15-17</td>
<td>—</td>
</tr>
<tr>
<td>18-19</td>
<td>E</td>
</tr>
<tr>
<td>20</td>
<td>A,LH</td>
</tr>
<tr>
<td>21</td>
<td>B,LL</td>
</tr>
<tr>
<td>22-25</td>
<td>—</td>
</tr>
<tr>
<td>26-27</td>
<td>XB</td>
</tr>
</tbody>
</table>

Save Mask:

<table>
<thead>
<tr>
<th>MUST BE ZERO</th>
<th>FALR 1, FAC</th>
<th>FALR 0</th>
<th>X</th>
<th>Y, S</th>
<th>E</th>
<th>L, B, A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 — 4</td>
<td>5 — 6</td>
<td>7 — 8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15-16</td>
</tr>
</tbody>
</table>

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

\[ A \leftrightarrow B \]

Move the contents of register A to register B and the contents of register B to register A. \( \text{MODES=SRV, FORMAT=GEN, OPCODE=000201, C=unchanged, L=unchanged, CC=unchanged.} \)

ICA  Interchange characters in A

\[ A(1-8) \leftrightarrow A(9-16) \]

Move the contents of register A bits 1–8 to bits 9–16 and the contents of bits 9–16 to bits 1–8. \( \text{MODES=SRV, FORMAT=GEN, OPCODE=141340, C=unchanged, L=unchanged, CC=unchanged.} \)

ICL  Interchange and clear left

\[ A(1-8) \rightarrow A(9-16); 0 \rightarrow A(1-8) \]

Move the contents of register A bits 1–8 to bits 9–16 and then clear the left byte (bits 1–8). \( \text{MODES=SRV, FORMAT=GEN, OPCODE=141140, C=unchanged, L=unchanged, CC=unchanged.} \)

ICR  Interchange and clear right

\[ A(9-16) \rightarrow A(1-8); 0 \rightarrow A(9-16) \]

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. \( \text{MODES=SRV, FORMAT=GEN, OPCODE=141240, C=unchanged, L=unchanged, CC=unchanged.} \)

ILE  Interchange L and E

\[ L \leftrightarrow E \]

Move the contents of register L to register E and the contents of register E to register L. \( \text{MODES=V, FORMAT=GEN, OPCODE=141414, C=unchanged, L=unchanged, CC=unchanged.} \)

IMA addr  Interchange memory and the A register

\[ [EA]16 \leftrightarrow A \]

Store the contents of the A register in location addr and load the original contents of location addr into the A register. \( \text{MODES=SRV, FORMAT=MR, OPCODE=13, C=unchanged, L=unchanged, CC=unchanged.} \)

LDA addr  Load the A register

\[ [EA]16 \rightarrow A \]

Load the contents of location addr into the A register. \( \text{MODES=SRV, FORMAT=MR, OPCODE=02, C=unchanged, L=unchanged, CC=unchanged.} \)

LDL addr  Load long

\[ [EA]32 \rightarrow L \]

Move the 32-bit quantity at location addr to register L. \( \text{MODES=V, FORMAT=MR, OPCODE=02 03, C=unchanged, L=unchanged, CC=unchanged.} \)
_para
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.

_para
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=S R V, FORMAT=MR, OPCODE=35, C=unchanged, L=unchanged, CC=unchanged.

Para
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.

Para
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=S R V, FORMAT=MR, OPCODE=04, C=unchanged, L=unchanged, CC=unchanged.

Para
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.

Para
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.

\textbf{MODES}=V, \textbf{FORMAT}=GEN, \textbf{OPCODE}=000605, \textbf{C}=\text{unspecified}, \textbf{L}=\text{unspecified}, \textbf{CC}=\text{unspecified.}

\begin{itemize}
  \item \textbf{CEA} Compute effective address
  \end{itemize}

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. \textbf{MODES}=SR, \textbf{FORMAT}=GEN, \textbf{OPCODE}=000111, \textbf{C}=\text{unchanged}, \textbf{L}=\text{unchanged, CC}=\text{unchanged.}

\begin{itemize}
  \item \textbf{CREP addr} Call recursive entry procedure
  \end{itemize}

\[
(P)+ \rightarrow \lfloor(S)+1 \rfloor \\
\text{EA} \rightarrow (P)
\]

Increment the program counter, \textit{P}, and load \textit{P}+1 into the location following the one specified by the current R-mode stack pointer. Load \textit{addr} into the program counter and continue execution from that location.

The \textit{CREP} instruction performs subroutine linkage for recursive or reentrant procedures. \textit{CREP} stores the return address in the second word of a stack frame created by the \textit{ENTR} instruction, rather than in the destination address as in a \textit{JST}.

\textbf{MODES}=R, \textbf{FORMAT}=MR, \textbf{OPCODE}=10 02, \textbf{C}=\text{unchanged, L}=\text{unchanged, CC}=\text{unchanged.}

\begin{itemize}
  \item \textbf{EAA addr} Effective address to A register
  \end{itemize}

\[
\text{EA} \rightarrow \text{A}
\]

Calculate the effective address and load it into register \textit{A}. The contents of \textit{addr} are unaffected and the original contents of register \textit{A} are overwritten and lost. \textbf{MODES}=R, \textbf{FORMAT}=MR, \textbf{OPCODE}=01 01, \textbf{C}=\text{unchanged, L}=\text{unchanged, CC}=\text{unchanged.}

\begin{itemize}
  \item \textbf{EAL addr} Effective address to L
  \end{itemize}

\[
\text{EA} \rightarrow \text{L}
\]

Calculate the effective address and put it into the \textit{L} register. \textbf{MODES}=V, \textbf{FORMAT}=MR, \textbf{OPCODE}=01 01, \textbf{C}=\text{unchanged, L}=\text{unchanged, CC}=\text{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

\[ X = X + 1; \text{if } X = 0, \text{ then } \text{EA} \rightarrow \text{PC}; \text{ else } \text{PC} = \text{PC} + 1 \]

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 \( A \leq 0 \), then \( \text{EA} \rightarrow \text{PC} \)

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 \( \text{EA} \rightarrow \text{PC} \)

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

\( \text{EA} \rightarrow \text{PC} \)

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 \( A \neq 0 \), then \( \text{EA} \rightarrow \text{PC} \)

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

\( \text{PC} \rightarrow [\text{EA}]_{16}; \text{EA} + 1 \rightarrow \text{PC} \)

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:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Preset Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>16S</td>
<td>I, X</td>
</tr>
<tr>
<td>32S, 32R</td>
<td>I</td>
</tr>
<tr>
<td>64, 64V</td>
<td>-</td>
</tr>
</tbody>
</table>
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**

Dealocates 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**

\[ [(S)+1] \rightarrow P[S] \rightarrow S \]

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:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>C</th>
<th>L</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>INBC</td>
<td>001217</td>
<td>unspecified</td>
<td>unspecified</td>
<td>unspecified</td>
</tr>
<tr>
<td>INBN</td>
<td>001215</td>
<td>unspecified</td>
<td>unspecified</td>
<td>unspecified</td>
</tr>
<tr>
<td>INEC</td>
<td>001216</td>
<td>unspecified</td>
<td>unspecified</td>
<td>unspecified</td>
</tr>
<tr>
<td>INEN</td>
<td>001214</td>
<td>unspecified</td>
<td>unspecified</td>
<td>unspecified</td>
</tr>
<tr>
<td>NFYB</td>
<td>001211</td>
<td>unspecified</td>
<td>unspecified</td>
<td>unspecified</td>
</tr>
<tr>
<td>NFEY</td>
<td>001210</td>
<td>unspecified</td>
<td>unspecified</td>
<td>unspecified</td>
</tr>
<tr>
<td>WAIT</td>
<td>000315</td>
<td>unchanged</td>
<td>unchanged</td>
<td>unchanged</td>
</tr>
</tbody>
</table>

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 two’s 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 ← A1—A16 ← 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
\[ \overline{A1—A16} \quad C \]
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
A1 ← A2 — A16 ← 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—A1—A16—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
\[ \overline{C} \quad \overline{A1—A16} \]
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
A1—A2—A16—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—A1—A16 ← B1—B16 ← 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

\[ A_1 \rightarrow A_{16} \leftarrow B_1 \rightarrow B_{16} \rightarrow C \]

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

\[ A_1 \leftarrow A_2 \rightarrow A_{16} \leftarrow B_1 \rightarrow B_{16} \rightarrow 0 \]

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

\[ C \leftarrow L_1 \rightarrow L_2 \rightarrow L_{32} \rightarrow 0 \]

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

\[ 0 \rightarrow A_1 \rightarrow A_{16} \rightarrow B_1 \rightarrow B_{16} \rightarrow C \]

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

\[ C \leftarrow A_1 \rightarrow A_{16} \rightarrow B_1 \rightarrow B_{16} \rightarrow C \]

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

\[ A_1 \rightarrow A_2 \rightarrow A_{16} \rightarrow B_1 \rightarrow B_{16} \rightarrow C \]

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

\[ L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow C \]

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=unchanged, CC=unchanged.

SKIP CONDITIONAL—SKIP

DRX  Decrement and replace X

\[ X-1 \rightarrow X; \text{if } X=0 \text{ then } PC+1 \rightarrow PC \]

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

\[ [\text{EA}]_{16}+1 \rightarrow [\text{EA}]_{16}; \text{if } [\text{EA}]_{16}=0 \text{ then } PC+1 \rightarrow PC \]

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

\[ X+1 \rightarrow X; \text{if } X=0 \text{ then } PC+1 \rightarrow PC \]

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Selector</th>
<th>Bit 7</th>
<th>Skip on Condition</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>1</td>
<td></td>
<td>None [no-op]</td>
<td>'101000</td>
</tr>
<tr>
<td>SKP</td>
<td>0</td>
<td></td>
<td>Skip unconditionally</td>
<td>'100000</td>
</tr>
<tr>
<td>SMI</td>
<td>8</td>
<td>1</td>
<td>A Minus (A(1) = 1)</td>
<td>'101400</td>
</tr>
<tr>
<td>SPL</td>
<td>8</td>
<td>0</td>
<td>A Plus (A(1) = 0)</td>
<td>'100400</td>
</tr>
<tr>
<td>SLN</td>
<td>10</td>
<td>1</td>
<td>LSB Nonzero (A(16) = 1)</td>
<td>'101100</td>
</tr>
<tr>
<td>SLZ</td>
<td>10</td>
<td>0</td>
<td>LSB Zero (A(16) = 0)</td>
<td>'100100</td>
</tr>
<tr>
<td>SNZ</td>
<td>11</td>
<td>1</td>
<td>A Nonzero</td>
<td>'101040</td>
</tr>
<tr>
<td>SZE</td>
<td>11</td>
<td>0</td>
<td>A Zero</td>
<td>'100040</td>
</tr>
<tr>
<td>R SS1</td>
<td>12</td>
<td>1</td>
<td>Sense Switch 1 Set</td>
<td>'101020</td>
</tr>
<tr>
<td>R SR1</td>
<td>12</td>
<td>0</td>
<td>Sense Switch 1 Reset</td>
<td>'100020</td>
</tr>
<tr>
<td>R SS2</td>
<td>13</td>
<td>1</td>
<td>Sense Switch 2 Set</td>
<td>'101010</td>
</tr>
<tr>
<td>R SR2</td>
<td>13</td>
<td>0</td>
<td>Sense Switch 2 Reset</td>
<td>'100010</td>
</tr>
<tr>
<td>R SS3</td>
<td>14</td>
<td>1</td>
<td>Sense Switch 3 Set</td>
<td>'101004</td>
</tr>
<tr>
<td>R SR3</td>
<td>14</td>
<td>0</td>
<td>Sense Switch 3 Reset</td>
<td>'100004</td>
</tr>
<tr>
<td>R SS4</td>
<td>15</td>
<td>1</td>
<td>Sense Switch 4 Set</td>
<td>'101002</td>
</tr>
<tr>
<td>R SR4</td>
<td>15</td>
<td>0</td>
<td>Sense Switch 4 Reset</td>
<td>'100002</td>
</tr>
<tr>
<td>R SSS</td>
<td>12–15</td>
<td>1</td>
<td>All Sense Switches 1–4 Set</td>
<td>'101036</td>
</tr>
<tr>
<td>R SSR</td>
<td>12–15</td>
<td>0</td>
<td>Any of Sense Switches 1–4 Reset</td>
<td>'100036</td>
</tr>
<tr>
<td>SSC</td>
<td>16</td>
<td>1</td>
<td>Set C</td>
<td>'101001</td>
</tr>
<tr>
<td>SRC</td>
<td>16</td>
<td>0</td>
<td>Clear C</td>
<td>'100001</td>
</tr>
</tbody>
</table>

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.
Instruction definitions - I
ADDRESSING MODE—ADMOD

Defined in Section 11.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E16S</td>
<td>Enter 16S Mode</td>
</tr>
<tr>
<td>E32R</td>
<td>Enter 32R Mode</td>
</tr>
<tr>
<td>E32S</td>
<td>Enter 32S Mode</td>
</tr>
<tr>
<td>E64R</td>
<td>Enter 64R Mode</td>
</tr>
<tr>
<td>E64V</td>
<td>Enter 64V Mode</td>
</tr>
<tr>
<td>E32I</td>
<td>Enter 32I Mode</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Word 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>REGISTER</td>
</tr>
<tr>
<td>0</td>
<td>OP-CODE</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Word 2 = 16-bit word address in current procedure segment

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Branch if LT bit set and EQ bit cleared</td>
</tr>
<tr>
<td>≤</td>
<td>Branch if LT bit set or EQ bit set</td>
</tr>
<tr>
<td>=</td>
<td>Branch if EQ bit set</td>
</tr>
<tr>
<td>≠</td>
<td>Branch if EQ bit cleared</td>
</tr>
<tr>
<td>&gt;</td>
<td>Branch if LT bit cleared or EQ bit set</td>
</tr>
<tr>
<td>&gt;</td>
<td>Branch if LT bit cleared and EQ bit cleared</td>
</tr>
</tbody>
</table>

Test Relation to 0 and branch if true
These instructions have the following format:

\[
\begin{align*}
\text{Branch if} & \quad \left\{ \begin{array}{l}
\text{Register [R]} \\
\text{Half-Register [H]} \\
\text{Floating-Register [F]} \\
\end{array} \right. \\
\text{then} & \quad \left\{ \begin{array}{l}
\text{LT} \\
\text{LE} \\
\text{EQ} \\
\text{NE} \\
\text{GE} \\
\text{GT} \\
\end{array} \right. \\
\text{result} & \quad 0
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRLT R.addr</td>
<td>If R&lt;0, then addr→PC</td>
<td>104</td>
</tr>
<tr>
<td>BRLE R.addr</td>
<td>If R≤0, then addr→PC</td>
<td>100</td>
</tr>
<tr>
<td>BREQ R.addr</td>
<td>If R=0, then addr→PC</td>
<td>102</td>
</tr>
<tr>
<td>BRNE R.addr</td>
<td>If R≠0, then addr→PC</td>
<td>103</td>
</tr>
<tr>
<td>BRGE R.addr</td>
<td>If R≥0, then addr→PC</td>
<td>105</td>
</tr>
<tr>
<td>BRGT R.addr</td>
<td>If R&gt;0, then addr→PC</td>
<td>101</td>
</tr>
<tr>
<td>BHLT RH.addr</td>
<td>If RH&lt;0, then addr→PC</td>
<td>104</td>
</tr>
<tr>
<td>BHLE RH.addr</td>
<td>If RH≤0, then addr→PC</td>
<td>110</td>
</tr>
<tr>
<td>BHEQ RH.addr</td>
<td>If RH=0, then addr→PC</td>
<td>112</td>
</tr>
<tr>
<td>BHNE RH.addr</td>
<td>If RH≠0, then addr→PC</td>
<td>113</td>
</tr>
<tr>
<td>BHGE RH.addr</td>
<td>If RH≥0, then addr→PC</td>
<td>105</td>
</tr>
<tr>
<td>BHGT RH.addr</td>
<td>If RH&gt;0, then addr→PC</td>
<td>111</td>
</tr>
<tr>
<td>BFLT F.addr</td>
<td>If F&lt;0, then addr→PC</td>
<td>124</td>
</tr>
<tr>
<td>BFLE F.addr</td>
<td>If F≤0, then addr→PC</td>
<td>120</td>
</tr>
<tr>
<td>BFSEQ F.addr</td>
<td>If F=0, then addr→PC</td>
<td>122</td>
</tr>
<tr>
<td>BFNE F.addr</td>
<td>If F≠0, then addr→PC</td>
<td>123</td>
</tr>
<tr>
<td>BFGE F.addr</td>
<td>If F≥0, then addr→PC</td>
<td>125</td>
</tr>
<tr>
<td>BFGT F.addr</td>
<td>If F&gt;0, then addr→PC</td>
<td>121</td>
</tr>
</tbody>
</table>

Test register bit and branch


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:

\[
\begin{align*}
\left\{ \begin{array}{l}
\text{Increment} \\
\text{Decrement} \\
\end{array} \right. & \quad \left\{ \begin{array}{l}
\text{Register} \\
\text{Half-Register} \\
\end{array} \right. \\
& \quad \text{by} \quad \left\{ \begin{array}{l}
1 \\
2 \\
4 \\
\end{array} \right. \\
& \quad \text{then branch if result≠0}
\end{align*}
\]

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.
INSTRUCTION DEFINITIONS—I

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR1 R,addr</td>
<td>R+1→R: if R≠0, then addr→PC</td>
<td>130</td>
</tr>
<tr>
<td>BR2 R,addr</td>
<td>R+2→R: if R≠0, then addr→PC</td>
<td>131</td>
</tr>
<tr>
<td>BR4 R,addr</td>
<td>R+4→R: if R≠0, then addr→PC</td>
<td>132</td>
</tr>
<tr>
<td>BHI1 RH,addr</td>
<td>RH+1→RH: if RH≠0, then addr→PC</td>
<td>140</td>
</tr>
<tr>
<td>BHI2 RH,addr</td>
<td>RH+2→RH: if RH≠0, then addr→PC</td>
<td>141</td>
</tr>
<tr>
<td>BHI4 RH,addr</td>
<td>RH+4→RH: if RH≠0, then addr→PC</td>
<td>142</td>
</tr>
<tr>
<td>BRD1 R,addr</td>
<td>R−1→R: if R≠0, then addr→PC</td>
<td>134</td>
</tr>
<tr>
<td>BRD2 R,addr</td>
<td>R−2→R: if R≠0, then addr→PC</td>
<td>135</td>
</tr>
<tr>
<td>BRD4 R,addr</td>
<td>R−4→R: if R≠0, then addr→PC</td>
<td>136</td>
</tr>
<tr>
<td>BHD1 RH,addr</td>
<td>RH−1→RH: if RH≠0, then addr→PC</td>
<td>144</td>
</tr>
<tr>
<td>BHD2 RH,addr</td>
<td>RH−2→RH: if RH≠0, then addr→PC</td>
<td>145</td>
</tr>
<tr>
<td>BHD4 RH,addr</td>
<td>RH−4→RH: if RH≠0, then addr→PC</td>
<td>146</td>
</tr>
</tbody>
</table>

**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
- **BGT**   Branch on condition code greater than
- **BLE**   Branch on condition code less than or equal
- **BCLT**  Branch on condition code less than
- **BNE**   Branch on condition code not equal
- **BHR**   Branch if C-Bit=0
- **BS**    Branch if C-Bit=1
- **BLS**   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
- **BMN**   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 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
- **ZFL** 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**

\[ 0 \rightarrow RH(1-8) \]

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**

\[ 0 \rightarrow RH(9-16) \]

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:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAD</td>
<td>Decimal Add</td>
</tr>
<tr>
<td>XBTBD</td>
<td>Binary to Decimal Conversion</td>
</tr>
<tr>
<td>XCM</td>
<td>Decimal Compare</td>
</tr>
<tr>
<td>XDTB</td>
<td>Decimal to Binary Conversion</td>
</tr>
<tr>
<td>XDV</td>
<td>Decimal Divide</td>
</tr>
<tr>
<td>XED</td>
<td>Numeric Edit</td>
</tr>
<tr>
<td>XMP</td>
<td>Decimal Multiply</td>
</tr>
<tr>
<td>XMV</td>
<td>Decimal Move</td>
</tr>
</tbody>
</table>

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. \text{FORMAT}=\text{RGEN}, \text{FLR0 OPCODE}=165, \text{FLR1 OPCODE}=175. \text{C}=\text{unchanged}, \text{L}=\text{unchanged}, \text{CC}=\text{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 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 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 32768 is an overflow: decreasing it below -32768 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

\text{FR1|EA|32→FR}

Add the floating point number at \text{addr} to the contents of the floating point number in floating point register \text{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. \text{FORMAT}=\text{MRFR}, \text{FR0 OPCODE}=14, \text{FR1 OPCODE}=16. \text{C}=\text{overflow}, \text{L}=\text{unspecified}, \text{CC}=\text{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. 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**

  \[ FR-[EA]_{32} \rightarrow FR \]

  Subtract the contents of addr from floating point register FR by aligning exponents, and proceeding 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**

  \[ FR \rightarrow [EA] _{32} \]

  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**

  \[ \text{Int}(FR) \rightarrow R \]

  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=unchanged.

- **INTH FR,R  Convert Floating Point to Halfword Integer**

  \[ \text{Int}(FR) \rightarrow RH \]

  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=unchanged.

### Double Precision—64 Bits

- **DBLE FR  Convert Single to Double**

  \[ FR \rightarrow FR \]

  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**

  \[ FR+[EA]_{64} \rightarrow FR \]

  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=RG, 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=MRF, 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=MRF, 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=MRF, 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=MRF, 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=MRF, 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. \text{FORMAT}=\text{MRGR}, \text{OPCODE}=02, C=\text{overflow}, L=\text{carry}, CC=\text{result}.

\begin{itemize}
  \item \textbf{ADLR R} Add Link to Register
    \begin{itemize}
      \item if keys \{L\}=1 then R+11=R
    \end{itemize}
  
  If the L bit is set in the keys then add 1 to the contents of register R. \text{FORMAT}=\text{RGEN}, \text{OPCODE}=014, C=\text{overflow}, L=\text{carry}, CC=\text{result}.
\end{itemize}

\begin{itemize}
  \item \textbf{AH R,addr} Add Halfword
    \begin{itemize}
      \item \text{RH}+[\text{EA}]16=\text{RH}
    \end{itemize}
    
    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. \text{FORMAT}=\text{MRGR}, \text{OPCODE}=12, C=\text{overflow}, L=\text{carry}, CC=\text{result}.
\end{itemize}

\begin{itemize}
  \item \textbf{C R,addr} Compare Fullword
    \begin{itemize}
      \item \text{R}:::[\text{EA}]32; set CC.
    \end{itemize}
    
    Arithmetically compare the 32-bit integer in R with the 32-bit integer at addr and set the condition codes to reflect the results. \text{FORMAT}=\text{MRGR}, \text{OPCODE}=61, C=\text{unchanged}, L=\text{carry}, CC=\text{result}.
\end{itemize}

\begin{itemize}
  \item \textbf{CH R,addr} Compare Halfword
    \begin{itemize}
      \item \text{RH}:::[\text{EA}]16; set CC.
    \end{itemize}
    
    Arithmetically compare bits 1–16 of register R with the 16-bit integer at addr and set the condition codes to reflect the results. \text{FORMAT}=\text{MRGR}, \text{OPCODE}=71, C=\text{unchanged}, L=\text{carry}, CC=\text{result}.
\end{itemize}

\begin{itemize}
  \item \textbf{CHS R} Change Sign
    \begin{itemize}
      \item \text{R[1]}=\text{R[1]}
    \end{itemize}
    
    Change bit 1 of register R to its opposite. \text{FORMAT}=\text{RGEN}, \text{OPCODE}=040, C=\text{unchanged}, L=\text{unchanged}, CC=\text{unchanged}.
\end{itemize}

\begin{itemize}
  \item \textbf{CSR R} Copy Sign
    \begin{itemize}
      \item \text{R[1]}=\text{C}; \text{0}=\text{R[1]}
    \end{itemize}
    
    Copy the sign bit of register R, \{bit 1\}, into C and zero R[1]. \text{FORMAT}=\text{RGEN}, \text{OPCODE}=041, C=R[1], L=\text{unchanged}, CC=\text{unchanged}.
\end{itemize}

\begin{itemize}
  \item \textbf{D R,addr} Divide Fullword
    \begin{itemize}
      \item \text{R}/[\text{EA}]32=\text{R}; \text{Remainder}=\text{R}+1
    \end{itemize}
    
    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. \text{FORMAT}=\text{MRGR}, \text{OPCODE}=62, C=\text{overflow/\text{div}} by 0, L=\text{unspecified}, CC=\text{unspeciﬁed}.
\end{itemize}

\begin{itemize}
  \item \textbf{DH R,addr} Divide Halfword
    \begin{itemize}
      \item \text{R}/[\text{EA}]16=\text{RH}; \text{Remainder}=\text{RH}
    \end{itemize}
\end{itemize}
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→RR+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

\[ R+1 \rightarrow R \]

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. \( \text{FORMAT=RGEN, OPCODE=50, C=overflow, L=unspecified, CC=unspecified} \).

PIMH R  Position Half Register After Multiply

\[ \text{RL} \rightarrow \text{RH} \]

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. \( \text{FORMAT=RGEN, OPCODE=51, C=overflow, L=unspecified, CC=unspecified} \).

S R,addr  Subtract Fullword

\[ R-|[EA]|32 \rightarrow R \]

Subtract the 32-bit integer at addr from 32-bit integer in register R, and put the result into R. \( \text{FORMAT=MRGR, OPCODE=22, C=overflow, L=carry, CC=result} \).

SH R,addr  Subtract Halfword

\[ \text{RH-|[EA]|16} \rightarrow \text{RH} \]

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. \( \text{FORMAT=MRGR, OPCODE=32, C=overflow, L=carry, CC=result} \).

SSM R  Set Sign Minus

\[ 1 \rightarrow R(1) \]

Set the sign bit of register R, (bit 1), equal to one. \( \text{FORMAT=RGEN, OPCODE=042, C=unchanged, L=unchanged, CC=unchanged} \).

SSP R  Set Sign Plus

\[ 0 \rightarrow R(1) \]

Set the sign bit of register R, (bit 1), equal to zero. \( \text{FORMAT=RGEN, OPCODE=043, C=unchanged, L=unchanged, CC=unchanged} \).

TC R  Two's Complement Register

\[ -R+1 \rightarrow R \]

Replace the contents of register R by its two's complement. \( \text{FORMAT=RGEN, OPCODE=046, C=overflow, L=carry, CC=result} \).

TCH R  Two's Complement Half Register

\[ -\text{RH}+1 \rightarrow \text{RH} \]

Replace the contents of RH by its two's complement. \( \text{FORMAT=RGEN, OPCODE=047, C=overflow, L=carry, CC=result} \).
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:

\[
\begin{align*}
\text{Condition Codes (C)} & \quad \text{LT} \\
\text{Register (blank)} & \quad \text{LE} \\
\text{If} & \quad \text{EQ} \\
\text{Half Register (H)} & \quad \text{0, then } 1\rightarrow R; \text{ else } 0\rightarrow R \\
\text{Floating-Point [F]} & \quad \text{NE} \\
\text{Register} & \quad \text{GE} \\
\text{GT} & \\
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCLT R</td>
<td>If CC&lt;0. then 1→R: else 0→R</td>
<td>150</td>
</tr>
<tr>
<td>LCLE R</td>
<td>If CC ≤ 0. then 1→R: else 0→R</td>
<td>151</td>
</tr>
<tr>
<td>LCEQ R</td>
<td>If CC=0. then 1→R: else 0→R</td>
<td>153</td>
</tr>
<tr>
<td>LCNE R</td>
<td>If CC≠0. then 1→R: else 0→R</td>
<td>152</td>
</tr>
<tr>
<td>LGGE R</td>
<td>If CC ≥ 0. then 1→R: else 0→R</td>
<td>154</td>
</tr>
<tr>
<td>LGGT R</td>
<td>If CC&gt;0. then 1→R: else 0→R</td>
<td>155</td>
</tr>
</tbody>
</table>

FORMAT=RGEN.  C=unchanged. L=unchanged. CC=unchanged.
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
  \[ R \leftarrow |EA|32 \]
  Swap the contents of R and addr. FORMAT=MRGR, OPCODE=41. C=unchanged. L=unchanged. CC=unchanged.

- **ICBL R** Interchange Bytes and Clear Left
  \[ RH[1-8] \leftarrow RH[9-16]; 0 \rightarrow RH[1-8] \]
  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
  \[ RH[9-16] \leftrightarrow RH[1-8]; 0 \rightarrow RH[9-16] \]
  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

\[ RH \leftrightarrow RL:0 \rightarrow RH \]

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

\[ RH \leftrightarrow RL:0 \rightarrow RL \]

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

\[ RH \leftrightarrow [EA]16 \]

Swap the contents of RH and addr. FORMAT=MRGR, OPCODE=51. C=unchanged. L=unchanged, CC=unchanged.

IRB R Interchange Register Bytes

\[ RH[1-8] \leftrightarrow RH[9-16] \]


IRH R Interchange Register Halves

\[ RH \leftrightarrow RL \]


L R,addr Load Fullword

\[ [EA]32 \rightarrow R \]

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

\[ [EA]16 \rightarrow RH \]

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

\[ [EA]16.LS.1 \rightarrow RH \]

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=result.

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 in the 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**

\[ EA \rightarrow XB \]

Store the effective address of addr in the temporary base register. **FORMAT=MRN.** **OPCODE=52.** **C=unchanged.** **L=unchanged.** **CC=unchanged.**

**JMP addr  Jump**

\[ EA \rightarrow PC \]

Jump to addr. **FORMAT=MRN.** **OPCODE=51.** **C=unchanged.** **L=unchanged.** **CC=unchanged.**

**JSR R,addr  Jump to Subroutine**

\[ PC(16-32) \rightarrow RH; EA \rightarrow PC \]

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**

\[ PC \rightarrow XB; EA \rightarrow PC \]

Jump to addr and save the full 32-bit return address in XB. **FORMAT=MRN.** **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


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.
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:

<table>
<thead>
<tr>
<th>R</th>
<th>RESTRICTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>regular instruction</td>
</tr>
<tr>
<td>R</td>
<td>instruction causes a restricted mode fault if executed in other than right 0</td>
</tr>
<tr>
<td>P</td>
<td>instruction may cause a fault depending on address</td>
</tr>
<tr>
<td>W</td>
<td>writable control store instruction, may be programmed in wcs to cause a fault</td>
</tr>
<tr>
<td>M</td>
<td>Machine specific—use only on specified CPU. Usually an instruction reserved for operating system, such as EPMJ.</td>
</tr>
</tbody>
</table>

MNEMONIC

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:

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN</td>
<td>Generic</td>
</tr>
<tr>
<td>AP</td>
<td>Address Pointer</td>
</tr>
<tr>
<td>BRAN</td>
<td>Branch</td>
</tr>
<tr>
<td>IBRN</td>
<td>I-mode Branch</td>
</tr>
<tr>
<td>CHAR</td>
<td>Character</td>
</tr>
<tr>
<td>DECI</td>
<td>Decimal</td>
</tr>
<tr>
<td>PIO</td>
<td>Programmed I/O</td>
</tr>
<tr>
<td>SHFT</td>
<td>Shift</td>
</tr>
<tr>
<td>MR</td>
<td>Memory Reference—non I-mode</td>
</tr>
<tr>
<td>MRFR</td>
<td>Memory Reference—Floating Register</td>
</tr>
<tr>
<td>MRNR</td>
<td>Memory Reference—Non Register</td>
</tr>
<tr>
<td>RGEN</td>
<td>Register Generic</td>
</tr>
</tbody>
</table>
### Function of instruction

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADMOD</td>
<td>Addressing Mode</td>
</tr>
<tr>
<td>BRAN</td>
<td>Branch</td>
</tr>
<tr>
<td>CHAR</td>
<td>Character</td>
</tr>
<tr>
<td>CLEAR</td>
<td>Clear field</td>
</tr>
<tr>
<td>DECI</td>
<td>Decimal Arithmetic</td>
</tr>
<tr>
<td>FIELD</td>
<td>Field Register</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Floating Point Arithmetic</td>
</tr>
<tr>
<td>INT</td>
<td>Integer</td>
</tr>
<tr>
<td>INTGY</td>
<td>Integrity</td>
</tr>
<tr>
<td>IO</td>
<td>Input/Output</td>
</tr>
<tr>
<td>KEYS</td>
<td>Keys</td>
</tr>
<tr>
<td>LOGIC</td>
<td>Logical Operations</td>
</tr>
<tr>
<td>LTSTS</td>
<td>Logical Test and Set</td>
</tr>
<tr>
<td>MCTL</td>
<td>Machine Control</td>
</tr>
<tr>
<td>MOVE</td>
<td>Move</td>
</tr>
<tr>
<td>PCTLJ</td>
<td>Program Control and Jump</td>
</tr>
<tr>
<td>PRCEX</td>
<td>Process Exchange</td>
</tr>
<tr>
<td>QUEUE</td>
<td>Queue Control</td>
</tr>
<tr>
<td>SHIFT</td>
<td>Register Shift</td>
</tr>
<tr>
<td>SKIP</td>
<td>Skip</td>
</tr>
</tbody>
</table>

### MODE

Addressing modes in which instruction functions as defined:

- **S**: Sectored
- **R**: Relative
- **V**: 64V (P400–P500)
- **I**: 32I (P500)

### Column

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td><strong>2</strong></td>
<td><strong>3</strong></td>
</tr>
<tr>
<td>Prime 100, 200, 300 series</td>
<td>Prime 400 series</td>
<td>Prime 500 series</td>
</tr>
</tbody>
</table>

**Codes are:**

- **Not implemented. Do not use this mnemonic on this CPU.**
- **Implemented by standard hardware.**
- **Implemented by hardware option or Ull library if option is not present.**

### 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
### INSTRUCTION SUMMARY CHART

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

<table>
<thead>
<tr>
<th>R</th>
<th>MNEM</th>
<th>OP CODE</th>
<th>RI</th>
<th>FORM</th>
<th>FUNC</th>
<th>MODE</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>C</th>
<th>CC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>02</td>
<td>YY</td>
<td>MRGR</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add Fullword</td>
<td></td>
</tr>
<tr>
<td>A1A</td>
<td>141206</td>
<td>GEN</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add One to A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2A</td>
<td>140304</td>
<td>GEN</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add Two to A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABQ</td>
<td>141716</td>
<td>AP</td>
<td>QUEUE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>Add to Bottom of Queue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARQ</td>
<td>134</td>
<td>AP</td>
<td>QUEUE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>7</td>
<td>Add to Bottom of Queue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACA</td>
<td>141216</td>
<td>GEN</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add C-Bit to A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>06</td>
<td>MR</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADL</td>
<td>06</td>
<td>MR</td>
<td>INT</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add Long</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADLL</td>
<td>141000</td>
<td>GEN</td>
<td>INT</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add Link Bit to L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADLR</td>
<td>014</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add Link to R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td>12</td>
<td>YY</td>
<td>MRGR</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Add Halfword</td>
<td></td>
</tr>
<tr>
<td>ALFA</td>
<td>001301</td>
<td>GEN</td>
<td>FIELD</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Add L to Field Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALFA</td>
<td>001311</td>
<td>GEN</td>
<td>FIELD</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Add L to Field Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALL</td>
<td>0414XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>A Left Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALR</td>
<td>0416XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>A Left Rotate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALS</td>
<td>0415XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>—</td>
<td>A Left Shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANA</td>
<td>03</td>
<td>MR</td>
<td>LOGIC</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANL</td>
<td>03</td>
<td>MR</td>
<td>FIELD</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>AND Long</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARFA</td>
<td>0161</td>
<td>RGEN</td>
<td>FIELD</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>Add R to Field Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARFA</td>
<td>171</td>
<td>RGEN</td>
<td>FIELD</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>Add R to Field Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARG</td>
<td>000605</td>
<td>GEN</td>
<td>PCTL</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Argument Transfer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARL</td>
<td>0404XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>A Right Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARR</td>
<td>0406XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>A Right Rotate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARS</td>
<td>0406XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>A Right Shift</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATQ</td>
<td>141717</td>
<td>AP</td>
<td>QUEUE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>7</td>
<td>Add to Top of Queue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATQ</td>
<td>135</td>
<td>AP</td>
<td>QUEUE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>7</td>
<td>Add to Top of Queue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCEQ</td>
<td>141602</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if CC = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCGE</td>
<td>141605</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if CC &gt; 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCGT</td>
<td>141601</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if CC &lt; 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCLE</td>
<td>141600</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if CC &lt; 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCLT</td>
<td>141604</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if CC &lt; 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCNE</td>
<td>141603</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if CC &lt; 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCR</td>
<td>141705</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if C-Bit = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCS</td>
<td>141704</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if C-Bit = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BDX</td>
<td>140734</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Decrement X and branch if X &lt;NE= 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BDY</td>
<td>140724</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Decrement Y and branch if Y =NE= 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td>140612</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>4</td>
<td>Branch if A = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td>140612</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>4</td>
<td>Branch if F = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFEQ</td>
<td>122</td>
<td>IBRN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>Branch if F = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFGE</td>
<td>141615</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Branch if F = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFGE</td>
<td>125</td>
<td>IBRN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>Branch if F = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFGT</td>
<td>141611</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>4</td>
<td>Branch if F = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFGT</td>
<td>122</td>
<td>IBRN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>Branch if F = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFLE</td>
<td>141610</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>4</td>
<td>Branch if F = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFLE</td>
<td>120</td>
<td>IBRN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>Branch if F = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 March, 1979  13-3  FDR 3059
<table>
<thead>
<tr>
<th>R</th>
<th>MNEM</th>
<th>CODE</th>
<th>OP</th>
<th>CODE</th>
<th>_OPCODE</th>
<th>RI</th>
<th>FORM</th>
<th>FUNC</th>
<th>MODE</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>C</th>
<th>CC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFLT</td>
<td>141614</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if F &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFLT</td>
<td>124</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if F &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFNE</td>
<td>141613</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if F ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFNE</td>
<td>123</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if F ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGE</td>
<td>140615</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if A ≥ 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BGT</td>
<td>140611</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if A &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHD1</td>
<td>144</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Decrement H by One; Branch if H ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHD2</td>
<td>145</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Decrement H by Two; Branch if H ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHD3</td>
<td>146</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Decrement H by Four; Branch if H ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHEQ</td>
<td>112</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if H = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHEG</td>
<td>105</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if H &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHT</td>
<td>111</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if H &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHI1</td>
<td>140</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment H by One; Branch if H ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHI2</td>
<td>141</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment H by Two; Branch if H ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHI4</td>
<td>142</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment H by One; Branch if H ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BJWT</td>
<td>110</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if H &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRJ</td>
<td>104</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if H &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>113</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if H is not equal to 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIX</td>
<td>141334</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment X and Branch if X ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLY</td>
<td>141324</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment Y and Branch if Y ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLE</td>
<td>140610</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if A &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLEG</td>
<td>140702</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if L = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLEG</td>
<td>140615</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if L &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLG</td>
<td>140701</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if L &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLEU</td>
<td>140614</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if L &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLN</td>
<td>140703</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if L ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLO</td>
<td>141707</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if L-Bit = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLS</td>
<td>141706</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if L-Bit = 1 (Set)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLT</td>
<td>140614</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if A &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMEQ</td>
<td>141602</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if Magnitude = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMGE</td>
<td>141706</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if Magnitude ≥ 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMGT</td>
<td>141710</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if Magnitude &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMLE</td>
<td>141711</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if Magnitude ≤ 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMLT</td>
<td>141707</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if Magnitude &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMNE</td>
<td>141603</td>
<td>BRAN</td>
<td>BRAN</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if Magnitude is not equal to 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRBR</td>
<td>140813</td>
<td>BRAN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if A ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRBS</td>
<td>000-037</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Branch if R bit n = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRD1</td>
<td>134</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Decrement R by One; Branch if R ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRD2</td>
<td>135</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Decrement R by Two; Branch if R ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRD4</td>
<td>136</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Decrement R by Four; Branch if R ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BREQ</td>
<td>102</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if R = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRGE</td>
<td>105</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if R &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRGT</td>
<td>101</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if R &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRI1</td>
<td>130</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment R by one and branch if ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRI2</td>
<td>131</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment R by two and branch if ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRI4</td>
<td>132</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment R by four and branch if ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRIE</td>
<td>100</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if R &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRET</td>
<td>104</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if R &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRNE</td>
<td>103</td>
<td>IBN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>Branch if R ≠E 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>61</td>
<td>YY</td>
<td>MRCR</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>1</td>
<td>1</td>
<td>Compare Fullword</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>CAY</td>
<td>000411</td>
<td>GEN</td>
<td>IO</td>
<td>SRVI</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Clear Active Interrupt</td>
<td></td>
</tr>
<tr>
<td>CAL</td>
<td>141050</td>
<td>GEN</td>
<td>CLEAR</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Clear A Left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALF</td>
<td>000705</td>
<td>GEN</td>
<td>PCTL</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Call Fault Handler</td>
<td></td>
</tr>
<tr>
<td>CAR</td>
<td>141044</td>
<td>GEN</td>
<td>CLEAR</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Clear A Right Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAS</td>
<td>11</td>
<td>MR</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Compare A and Skip</td>
<td></td>
</tr>
<tr>
<td>CAZ</td>
<td>140214</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Compare A with Zero</td>
<td></td>
</tr>
<tr>
<td>CAY</td>
<td>000111</td>
<td>GEN</td>
<td>PCTL</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Compute Effective Address</td>
<td></td>
</tr>
<tr>
<td>CGT</td>
<td>001314</td>
<td>GEN</td>
<td>BRAN</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>GCT</td>
<td>026</td>
<td>GEN</td>
<td>BRAN</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CH</td>
<td>71</td>
<td>YY</td>
<td>MRCR</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>1</td>
<td>1</td>
<td>Compare Halfword</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHS</td>
<td>140024</td>
<td>GEN</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Change Sign</td>
<td></td>
</tr>
<tr>
<td>R MNEM</td>
<td>OF CODE</td>
<td>Ri FORM</td>
<td>FUNC</td>
<td>MODE 1</td>
<td>2</td>
<td>3</td>
<td>C</td>
<td>CC</td>
<td>DESCRIPTION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>---------</td>
<td>------</td>
<td>--------</td>
<td>----</td>
<td>----</td>
<td>---</td>
<td>----</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHS</td>
<td>040</td>
<td>RGEN</td>
<td>INT</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>Change Sign</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLS</td>
<td>113</td>
<td>MR</td>
<td>LOGIC</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>1</td>
<td>-</td>
<td>Compare L and Skip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMA</td>
<td>140401</td>
<td>GEN</td>
<td>LOGIC</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Complement A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMH</td>
<td>045</td>
<td>RGEN</td>
<td>LOGIC</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>Complement H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMR</td>
<td>44</td>
<td>RGEN</td>
<td>LOGIC</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>Complement R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR</td>
<td>050</td>
<td>RGEN</td>
<td>CLEAR</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>Clear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRA</td>
<td>140040</td>
<td>GEN</td>
<td>CLEAR</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Clear A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRB</td>
<td>140015</td>
<td>GEN</td>
<td>CLEAR</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Clear B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRBL</td>
<td>062</td>
<td>RGEN</td>
<td>CLEAR</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>Clear High Byte 1 Left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRBR</td>
<td>061</td>
<td>RGEN</td>
<td>CLEAR</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>Clear High Byte 2 Right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRE</td>
<td>141404</td>
<td>GEN</td>
<td>CLEAR</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Clear E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREP</td>
<td>100</td>
<td>MR</td>
<td>PCTLJ</td>
<td>R</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Call Recursive Entry Procedure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRHL</td>
<td>054</td>
<td>RGEN</td>
<td>CLEAR</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>Clear Left Half Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRHR</td>
<td>055</td>
<td>RGEN</td>
<td>CLEAR</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>Clear Right Half Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRL</td>
<td>140010</td>
<td>GEN</td>
<td>CLEAR</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Clear L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRLE</td>
<td>141410</td>
<td>GEN</td>
<td>CLEAR</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Clear L and E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSA</td>
<td>1400320</td>
<td>GEN</td>
<td>MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>5</td>
<td>-</td>
<td>Copy Sign of A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSR</td>
<td>041</td>
<td>RGEN</td>
<td>MOVE</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>5</td>
<td>Copy Sign of R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R CXCS</td>
<td>000174</td>
<td>GEN</td>
<td>MCTL</td>
<td>VI</td>
<td>H</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Control Extended Control Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>62</td>
<td>YY</td>
<td>MRGR</td>
<td>INT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Divide Fullword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAD</td>
<td>06</td>
<td>MR</td>
<td>INT</td>
<td>SR</td>
<td>B</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Double Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBLE</td>
<td>000007</td>
<td>GEN</td>
<td>INT</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Enter Double Precision Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFC</td>
<td>140547</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>C</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Double Floating Complement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFCM</td>
<td>141444</td>
<td>GEN</td>
<td>FLPT</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Double Floating Complement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFCS</td>
<td>1102</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Double Floating Compare and Skip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFD</td>
<td>11330</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Double Floating Divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFVD</td>
<td>1702</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>D</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Double Floating Divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFL</td>
<td>0103</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Double Floating Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFLD</td>
<td>0202</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Double Floating Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFLX</td>
<td>1552</td>
<td>MR</td>
<td>FLPT</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Load Double Floating Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFM</td>
<td>2817</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Double Floating Multiply</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFMQ</td>
<td>1602</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>D</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Double Floating Multiply</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFS</td>
<td>2123</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Double Floating Subtract</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFSB</td>
<td>0702</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Double Floating Subtract</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFST</td>
<td>17133</td>
<td>NN</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Double Floating Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFST</td>
<td>0402</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Double Floating Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DH</td>
<td>72</td>
<td>YY</td>
<td>MRGR</td>
<td>INT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Divide Halfword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DH1</td>
<td>130</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Decrement H by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DH2</td>
<td>131</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Decrement H by 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>17</td>
<td>MR</td>
<td>INT</td>
<td>V</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>17</td>
<td>MR</td>
<td>INT</td>
<td>SR</td>
<td>B</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLD</td>
<td>02</td>
<td>MR</td>
<td>MOVE</td>
<td>SR</td>
<td>B</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Double Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DM</td>
<td>60</td>
<td>NN</td>
<td>MRNR</td>
<td>INT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>1</td>
<td>-</td>
<td>Decrement Fullword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMH</td>
<td>70</td>
<td>NN</td>
<td>MRNR</td>
<td>INT</td>
<td>I</td>
<td>-</td>
<td>H</td>
<td>1</td>
<td>-</td>
<td>Decrement Halfword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR1</td>
<td>124</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Decrement R by One</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR2</td>
<td>125</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Decrement R by Two</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRX</td>
<td>140210</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Decrement and Replace X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS8</td>
<td>07</td>
<td>MR</td>
<td>INT</td>
<td>SR</td>
<td>B</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Double Subtract</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DST</td>
<td>04</td>
<td>MR</td>
<td>MOVE</td>
<td>SR</td>
<td>B</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Double Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DVL</td>
<td>1703</td>
<td>MR</td>
<td>INT</td>
<td>V</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Divide Long</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E16S</td>
<td>000011</td>
<td>GEN</td>
<td>ADMOD</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Enter 16S Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2I</td>
<td>00010</td>
<td>GEN</td>
<td>ADMOD</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Enter 32I Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2R</td>
<td>00011</td>
<td>GEN</td>
<td>ADMOD</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Enter 32R Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2S</td>
<td>000013</td>
<td>GEN</td>
<td>ADMOD</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Enter 32S Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E4R</td>
<td>000011</td>
<td>GEN</td>
<td>ADMOD</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Enter 64R Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E64V</td>
<td>000010</td>
<td>GEN</td>
<td>ADMOD</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>Enter 64V Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAA</td>
<td>0101</td>
<td>MR</td>
<td>MOVE</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Effective Address to a Register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAFA</td>
<td>0003000</td>
<td>AP</td>
<td>FIELD</td>
<td>VI</td>
<td>H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Effective Address to Field Address Register 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>MNEMONIC</td>
<td>CODE</td>
<td>OP FIELD</td>
<td>FORM</td>
<td>FUNC</td>
<td>MODE</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>C</td>
<td>CC</td>
<td>DESCRIPTION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>----------</td>
<td>------</td>
<td>----------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAFA</td>
<td>001310</td>
<td>AP</td>
<td>FIELD</td>
<td>VI</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Effective Address to Field Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAL</td>
<td>01 01</td>
<td>MR</td>
<td>PCTL</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Effective Address to L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EALB</td>
<td>42</td>
<td>NN</td>
<td>MRNR</td>
<td>PCTL</td>
<td>I</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Effective Address to LB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAR</td>
<td>02</td>
<td>MR</td>
<td>PCTL</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Effective Address to R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAXB</td>
<td>52</td>
<td>NN</td>
<td>MRNR</td>
<td>PCTL</td>
<td>I</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Effective Address to XB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAXB</td>
<td>02</td>
<td>MR</td>
<td>PCTL</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Effective Address to XB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>EIO</td>
<td>34</td>
<td>NN</td>
<td>MRGR</td>
<td>IO</td>
<td>I</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Execute I/O</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>EIO</td>
<td>14</td>
<td>MR</td>
<td>IO</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Execute I/O</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>EIMC</td>
<td>000503</td>
<td>GEN</td>
<td>INTCY</td>
<td>SVRI</td>
<td>E</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Machine Check Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>ENBR</td>
<td>000401</td>
<td>GEN</td>
<td>IO</td>
<td>SVRI</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enable Interrupts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENTR</td>
<td>01 03</td>
<td>MR</td>
<td>PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Recursive Procedure Stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPMJ</td>
<td>000217</td>
<td>MR</td>
<td>MCTL</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Paging Mode and Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>EPMX</td>
<td>000237</td>
<td>MR</td>
<td>MCTL</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Paging Mode and Jump to XCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERA</td>
<td>05</td>
<td>MR</td>
<td>LOGIC</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Exclusive OR to A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERL</td>
<td>05 03</td>
<td>MR</td>
<td>LOGIC</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Exclusive OR to L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>ERMJ</td>
<td>000701</td>
<td>MR</td>
<td>MCTL</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Restricted Execution Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>ERMX</td>
<td>000721</td>
<td>MR</td>
<td>MCTL</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Restricted Execution Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>ESIM</td>
<td>000415</td>
<td>GEN</td>
<td>IO</td>
<td>SRVI</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Standard Interrupt Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>EVIM</td>
<td>000417</td>
<td>GEN</td>
<td>IO</td>
<td>SRVI</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Vectored Interrupt Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>EVMJ</td>
<td>000703</td>
<td>MR</td>
<td>MCTL</td>
<td>SR</td>
<td>F</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Vectorized Mode and Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>EVMX</td>
<td>000723</td>
<td>MR</td>
<td>MCTL</td>
<td>SR</td>
<td>F</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Enter Virtual Mode and Jump to WCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FA</td>
<td>14 16</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAD</td>
<td>06 01</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC</td>
<td>04 06</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>1</td>
<td>Floating Compare</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCM</td>
<td>140530</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>C</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Complement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCM</td>
<td>100</td>
<td>RGEN</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Complement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCS</td>
<td>11 01</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Compare and Skip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD</td>
<td>30 32</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDBL</td>
<td>140016</td>
<td>GEN</td>
<td>FLPT</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>Convert Single to Double Float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDV</td>
<td>17 01</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>D</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Divide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FL</td>
<td>00 02</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>Floating Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLD</td>
<td>02 01</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>Floating Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOT</td>
<td>140550</td>
<td>GEN</td>
<td>FLPT</td>
<td>R</td>
<td>C</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Convert 31-Bit Integer to Float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLT</td>
<td>105 115</td>
<td>RGEN</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Convert Integer to Floating</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLTH</td>
<td>140532</td>
<td>GEN</td>
<td>FLPT</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Convert Integer to Floating</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLTI</td>
<td>102 112</td>
<td>RGEN</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Convert Halfword to Floating</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLTJ</td>
<td>140535</td>
<td>GEN</td>
<td>FLPT</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Convert Long Integer to Floating</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLX</td>
<td>15 01</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>Load Double Word Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FM</td>
<td>24 26</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Multiply</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMP</td>
<td>16 01</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>D</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Multiply</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRN</td>
<td>140534</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>D</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Round</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRN</td>
<td>107</td>
<td>RGEN</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Round</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td>20 22</td>
<td>YY</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Subtract</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSB</td>
<td>07 01</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Floating Subtract</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSCT</td>
<td>140515</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>C</td>
<td>H</td>
<td>Floating Skip if &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSJE</td>
<td>140514</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>C</td>
<td>H</td>
<td>Floating Skip &lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSMJ</td>
<td>140512</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>C</td>
<td>H</td>
<td>Floating Skip if Minus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSNZ</td>
<td>140511</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>C</td>
<td>H</td>
<td>Floating Skip if Not Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSPL</td>
<td>140513</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>C</td>
<td>H</td>
<td>Floating Skip if Plus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FST</td>
<td>10 12</td>
<td>NN</td>
<td>MRFR</td>
<td>FLPT</td>
<td>I</td>
<td>H</td>
<td>3</td>
<td>Floating Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTE</td>
<td>04 01</td>
<td>MR</td>
<td>FLPT</td>
<td>RV</td>
<td>A</td>
<td>H</td>
<td>3</td>
<td>Floating Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSIZE</td>
<td>140510</td>
<td>GEN</td>
<td>FLPT</td>
<td>RV</td>
<td>C</td>
<td>H</td>
<td>Floating Skip if Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>HLT</td>
<td>000000</td>
<td>GEN</td>
<td>MCTL</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>Halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>47</td>
<td>YN</td>
<td>MRGR</td>
<td>MOV</td>
<td>I</td>
<td>H</td>
<td>Interchange Register and Memory-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fullword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IAB</td>
<td>000201</td>
<td>GEN</td>
<td>MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>Interchange A and B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICA</td>
<td>141340</td>
<td>GEN</td>
<td>MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>Interchange Characters in A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICRL</td>
<td>065</td>
<td>RGEN</td>
<td>MOVE</td>
<td>I</td>
<td>H</td>
<td>Interchange Bytes and Clear Left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICBR</td>
<td>066</td>
<td>RCEN</td>
<td>MOVE</td>
<td>I</td>
<td>H</td>
<td>Interchange Bytes and Clear Right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### INSTRUCTION SUMMARY CHART

<table>
<thead>
<tr>
<th>R</th>
<th>MNEM</th>
<th>OP CODE</th>
<th>FORM</th>
<th>FUNC</th>
<th>MODE</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>C</th>
<th>CC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICHL</td>
<td>060</td>
<td>RGEN</td>
<td>MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Interchange Halves and Clear Left</td>
<td></td>
</tr>
<tr>
<td>ICHR</td>
<td>051</td>
<td>RGEN</td>
<td>MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Interchange Halves and Clear Right</td>
<td></td>
</tr>
<tr>
<td>ICL</td>
<td>141740</td>
<td>GEN</td>
<td>MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Interchange and Clear Left</td>
<td></td>
</tr>
<tr>
<td>ICR</td>
<td>141240</td>
<td>GEN</td>
<td>MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Interchange and Clear Right</td>
<td></td>
</tr>
<tr>
<td>IH</td>
<td>51</td>
<td>YN MRGR</td>
<td>MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Interchange Register and Memory</td>
<td></td>
</tr>
<tr>
<td>IH1</td>
<td>126</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Increment by One</td>
<td></td>
</tr>
<tr>
<td>IH2</td>
<td>127</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Increment by Two</td>
<td></td>
</tr>
<tr>
<td>ILE</td>
<td>141414</td>
<td>GEN</td>
<td>MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Interchange L and E</td>
<td></td>
</tr>
<tr>
<td>IM</td>
<td>40</td>
<td>NN MRNR</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>1</td>
<td>Increment Fullword</td>
<td></td>
</tr>
<tr>
<td>IMA</td>
<td>13</td>
<td>MR MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Interchange Memory and A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMH</td>
<td>50</td>
<td>NN MRNR</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>1</td>
<td>Increment Halfword</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>IN</td>
<td>54</td>
<td>PIO</td>
<td>IO</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Input to A</td>
</tr>
<tr>
<td>R</td>
<td>INBC</td>
<td>001217</td>
<td>AP PRCEX</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>Interrupt Notify</td>
</tr>
<tr>
<td>R</td>
<td>INBN</td>
<td>001215</td>
<td>AP PRCEX</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>Interrupt Notify</td>
</tr>
<tr>
<td>R</td>
<td>INEC</td>
<td>001210</td>
<td>AP PRCEX</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>Interrupt Notify</td>
</tr>
<tr>
<td>R</td>
<td>INEN</td>
<td>001214</td>
<td>AP PRCEX</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>Interrupt Notify</td>
</tr>
<tr>
<td>R</td>
<td>INH</td>
<td>001001</td>
<td>GEN</td>
<td>IO</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Inhibit Interrupts</td>
</tr>
<tr>
<td>R</td>
<td>INK</td>
<td>000043</td>
<td>GEN KEYS</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Input Keys</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>INK</td>
<td>070</td>
<td>RGEN KEYS</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Save Keys</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>INT</td>
<td>140554</td>
<td>GEN FLPT</td>
<td>R</td>
<td>C</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>—</td>
<td>Convert Floating to Integer</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>INT</td>
<td>103,113</td>
<td>GEN FLPT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>—</td>
<td>Convert Floating to Integer</td>
</tr>
<tr>
<td>R</td>
<td>INTA</td>
<td>140531</td>
<td>GEN FLPT</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>—</td>
<td>Convert Floating to Integer</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>INTH</td>
<td>101,111</td>
<td>GEN FLPT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>—</td>
<td>Convert Floating to Halfword Integer</td>
</tr>
<tr>
<td>R</td>
<td>INTL</td>
<td>140533</td>
<td>GEN FLPT</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>—</td>
<td>Convert Floating to Integer Long</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>IR1</td>
<td>122</td>
<td>RGEN INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Increment R by One</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>IR2</td>
<td>123</td>
<td>RGEN INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Increment R by Two</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>IRB</td>
<td>064</td>
<td>RGEN MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Interchange Bytes</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>IRH</td>
<td>057</td>
<td>RGEN MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Interchange Halves</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>IRS</td>
<td>12</td>
<td>MR SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment Memory Replace and Skip</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>IRTC</td>
<td>000603</td>
<td>GEN IO</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>6</td>
<td>—</td>
<td>Interrupt Return</td>
</tr>
<tr>
<td>R</td>
<td>IRTN</td>
<td>000601</td>
<td>GEN IO</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>6</td>
<td>—</td>
<td>Interrupt Return</td>
</tr>
<tr>
<td>R</td>
<td>IRX</td>
<td>140114</td>
<td>GEN SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Increment and Replace X</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>ITLB</td>
<td>000615</td>
<td>GEN MCTL</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Invalidate STLX entry</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JDX</td>
<td>15</td>
<td>MR PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump and Decrement X</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JEQ</td>
<td>02</td>
<td>MR PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump if = 0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JGE</td>
<td>07</td>
<td>MR PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump if ≥ 0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JGT</td>
<td>05</td>
<td>MR PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump if &gt; 0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JIX</td>
<td>15</td>
<td>MR PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump and Increment X</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JLE</td>
<td>04</td>
<td>MR PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump if &lt; 0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JLT</td>
<td>06</td>
<td>MR PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump if &lt; 0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JMP</td>
<td>51</td>
<td>NN MRNR PCTL</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Jump</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JNE</td>
<td>03</td>
<td>MR PCTL</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JNE</td>
<td>03</td>
<td>MR PCTL</td>
<td>R</td>
<td>A</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump if ≠ 0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JSR</td>
<td>73</td>
<td>NN MRGR PCTL</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Jump to Subroutine</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JST</td>
<td>0</td>
<td>MR PCTL</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump and Store PC</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JSX</td>
<td>35</td>
<td>MR PCTL</td>
<td>RV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump and Store Return in X</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JSXB</td>
<td>61</td>
<td>NN MRNR PCTL</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Jump and Store Return in XB</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>JSY</td>
<td>14</td>
<td>MR PCTL</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Jump and Store Return in Y</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>L</td>
<td>01</td>
<td>MR PCTL</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Load</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LCEQ</td>
<td>141503</td>
<td>GEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test CC Equal to 0 and Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LCG</td>
<td>141504</td>
<td>GEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test CC Equal to 0 and Set R</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LGGE</td>
<td>141505</td>
<td>GEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test ≥ 0 and Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LGGT</td>
<td>141505</td>
<td>GEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test ≥ 0 and Set R</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LGT</td>
<td>155</td>
<td>GEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test CC &gt; 0 and Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LGLE</td>
<td>141501</td>
<td>GEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test CC &gt; 0 and Set R</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LCLT</td>
<td>141500</td>
<td>GEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test CC &lt; 0 and Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LCLE</td>
<td>141502</td>
<td>GEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test CC &lt; 0 and Set R</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LCNE</td>
<td>152</td>
<td>RGEN LTSTS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Test CC &lt; 0 and Set R</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>MNEMONIC</td>
<td>OP CODE</td>
<td>RI</td>
<td>FORM</td>
<td>FUNC</td>
<td>MODE</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>CC</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>---</td>
<td>----------</td>
<td>---------</td>
<td>----</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>-------------</td>
</tr>
<tr>
<td>L</td>
<td>LDA</td>
<td>02</td>
<td>MR</td>
<td>MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Load A</td>
</tr>
<tr>
<td>L</td>
<td>LDAR</td>
<td>44</td>
<td>NN</td>
<td>MRGR</td>
<td>MOVE</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Load From Addressed Register</td>
</tr>
<tr>
<td>L</td>
<td>LDC</td>
<td>165</td>
<td>RGEN</td>
<td>CHAR</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>7</td>
<td>Character</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LDC</td>
<td>172</td>
<td>RGEN</td>
<td>CHAR</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>7</td>
<td>Character</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LDC</td>
<td>003</td>
<td>CHAR</td>
<td>CHAR</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>Load Character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LDC</td>
<td>001312</td>
<td>CHAR</td>
<td>CHAR</td>
<td>V</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>Load Character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LDL</td>
<td>02</td>
<td>03</td>
<td>MR</td>
<td>MOVE</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Load Long</td>
</tr>
<tr>
<td>L</td>
<td>LDLR</td>
<td>05</td>
<td>01</td>
<td>MR</td>
<td>MOVE</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Load From Addressed Register</td>
</tr>
<tr>
<td>L</td>
<td>LDX</td>
<td>35</td>
<td>MR</td>
<td>MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Load X</td>
</tr>
<tr>
<td>L</td>
<td>LDY</td>
<td>35</td>
<td>01</td>
<td>MR</td>
<td>MOVE</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Load Y</td>
</tr>
<tr>
<td>L</td>
<td>LEQ</td>
<td>002</td>
<td>GEN</td>
<td>LTSTS</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>A = 0; Set A</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LEQ</td>
<td>003</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set R</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LF</td>
<td>016</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Logic Set A False</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LF</td>
<td>140113</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B = 0; Set A</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFEQ</td>
<td>023</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set R</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFGT</td>
<td>023034</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &gt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFLE</td>
<td>140114</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B = 0; Set R</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFLE</td>
<td>025</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFLE</td>
<td>021031</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set R</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFFI</td>
<td>001030</td>
<td>GEN</td>
<td>LTSTS</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Load Field Length Register 0</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFFI</td>
<td>001313</td>
<td>GEN</td>
<td>LTSTS</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Load Field Length Register 1</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFFT</td>
<td>001111</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFFT</td>
<td>020030</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set R</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFGT</td>
<td>141112</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B = 0; Set A</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LFGT</td>
<td>022032</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set R</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LGT</td>
<td>002</td>
<td>SLTS</td>
<td>SRV</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &gt; 0; Set A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LGT</td>
<td>004</td>
<td>SLTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &gt; 0; Set A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LGT</td>
<td>025</td>
<td>SLTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Load Halfword Left Shifted by 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LHLT</td>
<td>14</td>
<td>MRGR</td>
<td>MOVE</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Load Halfword Left Shifted by 2</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LHLT</td>
<td>001</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LHLT</td>
<td>000</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &lt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>LHLT</td>
<td>012</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test B &gt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LIOT</td>
<td>000044</td>
<td>GEN</td>
<td>LTSTS</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Load I/O TLB (Prime 700 only)</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLE</td>
<td>140411</td>
<td>GEN</td>
<td>LTSTS</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test A &lt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLE</td>
<td>001</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R &lt; 0; Set R</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLEQ</td>
<td>141513</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R &gt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLGE</td>
<td>140414</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R = 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLGT</td>
<td>141515</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R &gt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLM</td>
<td>0410XX</td>
<td>SLFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Long Left Logical Phase 1</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLM</td>
<td>141511</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R &gt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLM</td>
<td>140410</td>
<td>GEN</td>
<td>LTSTS</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R &lt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLM</td>
<td>141512</td>
<td>SLTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R &lt; 0; Set A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLT</td>
<td>0412XX</td>
<td>SLFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Long Left Shift 1</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLS</td>
<td>0411XX</td>
<td>SLFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>2</td>
<td>Long Left Shift 2</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LLT</td>
<td>140410</td>
<td>GEN</td>
<td>LTSTS</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test A &lt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LTT</td>
<td>000</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R &lt; 0; Set R</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LMCM</td>
<td>000501</td>
<td>GEN</td>
<td>INTGY</td>
<td>SRV 1</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>Leave Machine Check Mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LNE</td>
<td>140412</td>
<td>GEN</td>
<td>LTSTS</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test A &lt; 0; Set A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LNE</td>
<td>002</td>
<td>GEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Test R &gt; 0; Set R</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LPID</td>
<td>006S17</td>
<td>GEN</td>
<td>MCTL</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>Load Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LPMJ</td>
<td>000215</td>
<td>MR</td>
<td>MCTL</td>
<td>SR</td>
<td>F</td>
<td>--</td>
<td>--</td>
<td>Leave Paging Mode and JUMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LPMX</td>
<td>000235</td>
<td>MR</td>
<td>MCTL</td>
<td>SR</td>
<td>F</td>
<td>--</td>
<td>--</td>
<td>Leave Paging Mode and JUMP to XCX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LPSW</td>
<td>000711</td>
<td>AP</td>
<td>MCTL</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>7</td>
<td>Load Program Status Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>MNEM</td>
<td>OP CODE</td>
<td>RI</td>
<td>FORM</td>
<td>FUNC</td>
<td>MODE 1</td>
<td>2</td>
<td>3</td>
<td>C</td>
<td>CC</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>---</td>
<td>-------</td>
<td>--------</td>
<td>----</td>
<td>------</td>
<td>------</td>
<td>--------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>-------------</td>
</tr>
<tr>
<td>LRL</td>
<td>0400XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>4</td>
<td>--</td>
<td>Long Right Logical</td>
<td></td>
</tr>
<tr>
<td>LRR</td>
<td>0402XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>4</td>
<td>--</td>
<td>Long Right Rotate</td>
<td></td>
</tr>
<tr>
<td>LRS</td>
<td>0401XX</td>
<td>SHIFT</td>
<td>SHIFT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>4</td>
<td>--</td>
<td>Long Right Shift</td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td>140417</td>
<td>GEN</td>
<td>LTSTS</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Set A = 1</td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td>017</td>
<td>RGEN</td>
<td>LTSTS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>4</td>
<td>Set R = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>LWCS</td>
<td>001770</td>
<td>GEN</td>
<td>MCTL</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Load Writable Control Store</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>42</td>
<td>YY</td>
<td>MRGR</td>
<td>INT</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>3</td>
<td>--</td>
<td>Multiply Fullword</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>MDER</td>
<td>001304</td>
<td>GEN</td>
<td>INTGY</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Memory Diagnostic Enable Interleave</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>MDII</td>
<td>001305</td>
<td>GEN</td>
<td>INTGY</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Inhibit Interleaved</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>TDVR</td>
<td>001324</td>
<td>GEN</td>
<td>INTGY</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Write Interleaved</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>MDIS</td>
<td>001306</td>
<td>GEN</td>
<td>INTGY</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Read Syndrome Bits</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>MDRC</td>
<td>001307</td>
<td>GEN</td>
<td>INTGY</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Load Write Control Register</td>
<td></td>
</tr>
<tr>
<td>MH</td>
<td>52</td>
<td>YY</td>
<td>MRGR</td>
<td>INT</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Multiply Halfword</td>
<td></td>
</tr>
<tr>
<td>MIA</td>
<td>64</td>
<td>NN</td>
<td>MRGR</td>
<td>MCTL</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Microcode Entrance</td>
<td></td>
</tr>
<tr>
<td>MIA</td>
<td>12</td>
<td>01</td>
<td>MR</td>
<td>MCTL</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>Microcode Entrance</td>
<td></td>
</tr>
<tr>
<td>MIB</td>
<td>74</td>
<td>NN</td>
<td>MRGR</td>
<td>MCTL</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Microcode Entrance</td>
<td></td>
</tr>
<tr>
<td>MIB</td>
<td>13</td>
<td>01</td>
<td>MR</td>
<td>MCTL</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>Microcode Entrance</td>
<td></td>
</tr>
<tr>
<td>MPL</td>
<td>16</td>
<td>03</td>
<td>MR</td>
<td>INT</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>3</td>
<td>Multiply Long</td>
<td></td>
</tr>
<tr>
<td>MPY</td>
<td>16</td>
<td>MR</td>
<td>INT</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>3</td>
<td>Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>03</td>
<td>YY</td>
<td>MRGR</td>
<td>LOGIC</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>AND Fullword</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>NFRB</td>
<td>001211</td>
<td>AP</td>
<td>PRCX</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Notify</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>NYB</td>
<td>001210</td>
<td>AP</td>
<td>PRCX</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Notify</td>
<td></td>
</tr>
<tr>
<td>NH</td>
<td>13</td>
<td>YY</td>
<td>MRGR</td>
<td>LOGIC</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>AND Halfword</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>000001</td>
<td>GEN</td>
<td>MCTL</td>
<td>SRVI</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>No Operation</td>
<td></td>
</tr>
<tr>
<td>NRM</td>
<td>000101</td>
<td>GEN</td>
<td>INT</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Normalize</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>23</td>
<td>YY</td>
<td>MRGR</td>
<td>LOGIC</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>OR Fullword</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>OCP</td>
<td>14</td>
<td>PIO</td>
<td>IO</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Output Control Pulse</td>
</tr>
<tr>
<td>OIA</td>
<td>33</td>
<td>YY</td>
<td>MRGR</td>
<td>LOGIC</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>OR Halfword</td>
<td></td>
</tr>
<tr>
<td>ORA</td>
<td>02</td>
<td>02</td>
<td>MR</td>
<td>LOGIC</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>Inclusive OR</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>OTA</td>
<td>74</td>
<td>PIO</td>
<td>IO</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Output from A</td>
</tr>
<tr>
<td>OKT</td>
<td>000405</td>
<td>GEN</td>
<td>KEYS</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>6</td>
<td>Restore Keys</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTK</td>
<td>071</td>
<td>RGEN</td>
<td>KEYS</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>7</td>
<td>6</td>
<td>Restore Keys</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCL</td>
<td>41</td>
<td>NN</td>
<td>MRGR</td>
<td>PCTL</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Procedure Call</td>
<td></td>
</tr>
<tr>
<td>PCL</td>
<td>10</td>
<td>02</td>
<td>MR</td>
<td>PCTL</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Procedure Call</td>
<td></td>
</tr>
<tr>
<td>PID</td>
<td>000211</td>
<td>GEN</td>
<td>INT</td>
<td>SR</td>
<td>B</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Position for Integer Divide</td>
<td></td>
</tr>
<tr>
<td>PID</td>
<td>052</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Position for Integer Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIDA</td>
<td>000115</td>
<td>GEN</td>
<td>INT</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Position for Integer Divide</td>
<td></td>
</tr>
<tr>
<td>PIDH</td>
<td>053</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Position for Integer Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIDL</td>
<td>000305</td>
<td>GEN</td>
<td>INT</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Position Long for Integer Divide</td>
<td></td>
</tr>
<tr>
<td>PIM</td>
<td>000205</td>
<td>GEN</td>
<td>INT</td>
<td>SR</td>
<td>B</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Position After Multiply</td>
<td></td>
</tr>
<tr>
<td>PIM</td>
<td>50</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Position After Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIMA</td>
<td>000015</td>
<td>GEN</td>
<td>INT</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Position After Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIMH</td>
<td>51</td>
<td>RGEN</td>
<td>INT</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Position After Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIML</td>
<td>000301</td>
<td>GEN</td>
<td>INT</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>3</td>
<td>5</td>
<td>Position After Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>PTBL</td>
<td>000664</td>
<td>GEN</td>
<td>MCTL</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Purge TLB [Prime 750 only]</td>
<td></td>
</tr>
<tr>
<td>RBQ</td>
<td>141715</td>
<td>AP</td>
<td>QUEUE</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>7</td>
<td>Remove From Bottom of Queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBQ</td>
<td>133</td>
<td>AP</td>
<td>RGEN</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>7</td>
<td>Remove From Bottom of Queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>CBB</td>
<td>140200</td>
<td>GEN</td>
<td>KEYS</td>
<td>SRVI</td>
<td>H</td>
<td>H</td>
<td>5</td>
<td>--</td>
<td>Clear C-Bit (Reset)</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>RMC</td>
<td>000021</td>
<td>GEN</td>
<td>INTGY</td>
<td>SRVI</td>
<td>E</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Clear Machine Check</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>ROT</td>
<td>24</td>
<td>NN</td>
<td>MRGR</td>
<td>SHIFT</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>4</td>
<td>--</td>
<td>Rotate</td>
</tr>
<tr>
<td>RRST</td>
<td>000717</td>
<td>AP</td>
<td>MCTL</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Register Restore</td>
<td></td>
</tr>
<tr>
<td>RSAV</td>
<td>000715</td>
<td>AP</td>
<td>MCTL</td>
<td>VI</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Register Save</td>
<td></td>
</tr>
<tr>
<td>RII</td>
<td>000105</td>
<td>GEN</td>
<td>PCTL</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTQ</td>
<td>141714</td>
<td>AP</td>
<td>QUEUE</td>
<td>V</td>
<td>--</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>Remove From Top of Queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTQ</td>
<td>132</td>
<td>RGEN</td>
<td>QUEUE</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>--</td>
<td>7</td>
<td>Remove From Top of Queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>22</td>
<td>YY</td>
<td>MRGR</td>
<td>INT</td>
<td>I</td>
<td>--</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Subtract Fullword</td>
<td></td>
</tr>
<tr>
<td>SIA</td>
<td>140110</td>
<td>GEN</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Subtract One from A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SZA</td>
<td>140310</td>
<td>GEN</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Subtract Two from A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR</td>
<td>10026X</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Skip on A Bit Clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAS</td>
<td>10126X</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>--</td>
<td>--</td>
<td>Skip on A Bit Set</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 March, 1979
<table>
<thead>
<tr>
<th>R</th>
<th>MNEM</th>
<th>OP CODE</th>
<th>RI</th>
<th>FORM</th>
<th>FUNC</th>
<th>MODE</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>C</th>
<th>CC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL</td>
<td>07</td>
<td>03</td>
<td>MR</td>
<td>INT</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Subtract Long</td>
<td></td>
</tr>
<tr>
<td>SCA</td>
<td>000041</td>
<td>GEN</td>
<td>INT</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Load Shift Count into A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCB</td>
<td>140000</td>
<td>GEN</td>
<td>KEYS</td>
<td>SRV1</td>
<td>H</td>
<td>H</td>
<td>5</td>
<td>—</td>
<td>—</td>
<td>Set C-Bit in Keys</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGL</td>
<td>000005</td>
<td>GEN</td>
<td>INT</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Enter Single Precision Mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGT</td>
<td>100220</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if A Greater Than Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SH</td>
<td>32</td>
<td>YY</td>
<td>MRGR</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>Subtract Halfword</td>
<td></td>
</tr>
<tr>
<td>SHA</td>
<td>15</td>
<td>NN</td>
<td>MRGR</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift Arithmetic</td>
<td></td>
</tr>
<tr>
<td>SHL</td>
<td>05</td>
<td>NN</td>
<td>MRGR</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift Logical</td>
<td></td>
</tr>
<tr>
<td>SHL1</td>
<td>076</td>
<td>RGEN</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift H Left One</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL2</td>
<td>077</td>
<td>RGEN</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift H Left Two</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR1</td>
<td>120</td>
<td>RGEN</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift H Right One</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR2</td>
<td>121</td>
<td>RGEN</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift H Right Two</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SKP</td>
<td>100000</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SKS</td>
<td>34</td>
<td>PIO</td>
<td>IO</td>
<td>SR</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Satisfied</td>
<td></td>
</tr>
<tr>
<td>SL1</td>
<td>072</td>
<td>RGEN</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift R Left One</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SL2</td>
<td>073</td>
<td>RGEN</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift R Left Two</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLE</td>
<td>101220</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if A Less Than or Equal to Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLN</td>
<td>101100</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if LSB Nonzero (A[16]=1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMCR</td>
<td>100200</td>
<td>GEN</td>
<td>INTGY</td>
<td>SRV</td>
<td>E</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip on Machine Check Reset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMCS</td>
<td>101200</td>
<td>GEN</td>
<td>INTGY</td>
<td>SRV</td>
<td>E</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip on Machine Check Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMI</td>
<td>101400</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if A Minus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SNR</td>
<td>10024X</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip on Sense Switch Clear</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SNS</td>
<td>10124X</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip on Sense Switch Set</td>
<td></td>
</tr>
<tr>
<td>SNZ</td>
<td>101040</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if A Non-Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SPL</td>
<td>100400</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if A Plus</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SR1</td>
<td>100020</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Sense Switch 1 Cleared</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SR2</td>
<td>100010</td>
<td>RGEN</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift R Right One</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SR3</td>
<td>100004</td>
<td>RGEN</td>
<td>SHFT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>4</td>
<td>—</td>
<td>Shift R Right Two</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SR4</td>
<td>100002</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Sense Switch 3 Cleared</td>
<td></td>
</tr>
<tr>
<td>SRC</td>
<td>100001</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Sense Switch 4 Cleared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SS1</td>
<td>101020</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Sense Switch 1 Cleared</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SS2</td>
<td>101010</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Sense Switch 2 Cleared</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SS3</td>
<td>101004</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Sense Switch 3 Cleared</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SS4</td>
<td>101002</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Sense Switch 4 Cleared</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SSC</td>
<td>101001</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if C-Bit is Set</td>
<td></td>
</tr>
<tr>
<td>SSM</td>
<td>140500</td>
<td>GEN</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Set Sign Minus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSM</td>
<td>140100</td>
<td>GEN</td>
<td>INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Set Sign Minus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSM</td>
<td>140100</td>
<td>GEN</td>
<td>INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Set Sign Plus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SSP</td>
<td>100036</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if Any Sense Switch is Clear</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>SSS</td>
<td>101036</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if All Sense Switches are Set</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>ST</td>
<td>21</td>
<td>NN</td>
<td>MRGR</td>
<td>MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>Store Fullword</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td>04</td>
<td>001200</td>
<td>MR</td>
<td>MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Store A</td>
<td></td>
</tr>
<tr>
<td>STAC</td>
<td>04</td>
<td>001200</td>
<td>AP</td>
<td>MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>—</td>
<td>Store A Conditionally</td>
<td></td>
</tr>
<tr>
<td>STAR</td>
<td>54</td>
<td>001200</td>
<td>NN</td>
<td>MRGR</td>
<td>MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>Store into Addressed Register</td>
<td></td>
</tr>
<tr>
<td>STC 0</td>
<td>166</td>
<td>176</td>
<td>RGEN</td>
<td>CHAR</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>7</td>
<td>Store Character</td>
<td></td>
</tr>
<tr>
<td>STC 0</td>
<td>001322</td>
<td>CHAR</td>
<td>CHAR</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>—</td>
<td>Store Character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STC 0</td>
<td>001322</td>
<td>CHAR</td>
<td>CHAR</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>—</td>
<td>Store Character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STC 0</td>
<td>157</td>
<td>AP</td>
<td>MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>7</td>
<td>Store Conditional Fullword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STCH</td>
<td>136</td>
<td>AP</td>
<td>MOVE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Stack Extend</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STCH</td>
<td>00135</td>
<td>AP</td>
<td>MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Stack Extend</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STCH</td>
<td>00135</td>
<td>GEN</td>
<td>PCTLJ</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Store Field Address Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STCH</td>
<td>00135</td>
<td>AP</td>
<td>MOVE</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Store Field Address Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STCH</td>
<td>00135</td>
<td>AP</td>
<td>FIELD</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Store Field Address Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STCH</td>
<td>00135</td>
<td>AP</td>
<td>MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>—</td>
<td>Store L Conditionally</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>STLR</td>
<td>03</td>
<td>01</td>
<td>MR</td>
<td>MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>Store into Addressed Register</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>MNEM</td>
<td>OP CODE</td>
<td>RI</td>
<td>FORM</td>
<td>FUNC</td>
<td>MODE 1</td>
<td>2</td>
<td>3</td>
<td>C</td>
<td>CC</td>
<td>DESCRIPTION</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>STPM</td>
<td>000024</td>
<td>GEN</td>
<td>MCTL</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Store Processor Model Number</td>
<td></td>
</tr>
<tr>
<td>STX</td>
<td>15</td>
<td>MR MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Store X</td>
<td></td>
</tr>
<tr>
<td>STY</td>
<td>35 02</td>
<td>MR MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Store Y</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>07</td>
<td>MR INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Subtract</td>
<td></td>
</tr>
<tr>
<td>SVC</td>
<td>000505</td>
<td>GEN</td>
<td>PCTLJ</td>
<td>SRVI</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Supervisor Call</td>
<td></td>
</tr>
<tr>
<td>SZE</td>
<td>100040</td>
<td>GEN</td>
<td>SKIP</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Skip if A Zero</td>
<td></td>
</tr>
<tr>
<td>TAB</td>
<td>140314</td>
<td>GEN MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer A to B</td>
<td></td>
</tr>
<tr>
<td>TAK</td>
<td>001015</td>
<td>GEN KEYS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>7</td>
<td>6</td>
<td>—</td>
<td>—</td>
<td>Move A to Keys</td>
<td></td>
</tr>
<tr>
<td>TAX</td>
<td>140504</td>
<td>GEN MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer A to X</td>
<td></td>
</tr>
<tr>
<td>TAY</td>
<td>140505</td>
<td>GEN MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer A to Y</td>
<td></td>
</tr>
<tr>
<td>TBA</td>
<td>140604</td>
<td>GEN MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer B to A</td>
<td></td>
</tr>
<tr>
<td>TC</td>
<td>046</td>
<td>RGEN INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Two's Complement R</td>
<td></td>
</tr>
<tr>
<td>TCA</td>
<td>140407</td>
<td>GEN INT</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Two's Complement A</td>
<td></td>
</tr>
<tr>
<td>TCH</td>
<td>047</td>
<td>RGEN INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Two's Complement H</td>
<td></td>
</tr>
<tr>
<td>TCL</td>
<td>141210</td>
<td>GEN INT</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>2</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Two's Complement Long</td>
<td></td>
</tr>
<tr>
<td>TFL0</td>
<td>001323</td>
<td>GEN FIELD</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer Field Length to L</td>
<td></td>
</tr>
<tr>
<td>TFLR</td>
<td>001333</td>
<td>GEN FIELD</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer Field Length to L</td>
<td></td>
</tr>
<tr>
<td>TFLR</td>
<td>163</td>
<td>RGEN FIELD</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Move Field Length to R</td>
<td></td>
</tr>
<tr>
<td>TQL</td>
<td>173</td>
<td>RGEN FIELD</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Move Field Length to R</td>
<td></td>
</tr>
<tr>
<td>TKA</td>
<td>001005</td>
<td>GEN KEYS</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Move Keys to A</td>
<td></td>
</tr>
<tr>
<td>TLFL</td>
<td>001321</td>
<td>GEN FIELD</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer L to Field Length Register</td>
<td></td>
</tr>
<tr>
<td>TLFL</td>
<td>001331</td>
<td>GEN FIELD</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer L to Field Length Register</td>
<td></td>
</tr>
<tr>
<td>TM</td>
<td>44</td>
<td>NN MRNR MCTL</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Test Memory Fullword</td>
<td></td>
</tr>
<tr>
<td>TMH</td>
<td>54</td>
<td>NN MRNR INT</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Test Memory Halfword</td>
<td></td>
</tr>
<tr>
<td>TRFL</td>
<td>165</td>
<td>RGEN FIELD</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer R to Field Length Register</td>
<td></td>
</tr>
<tr>
<td>TRFL</td>
<td>175</td>
<td>RGEN FIELD</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer R to Field Length Register</td>
<td></td>
</tr>
<tr>
<td>TSTQ</td>
<td>141757</td>
<td>AP QUEUE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>7</td>
<td>—</td>
<td>—</td>
<td>Test Queue</td>
<td></td>
</tr>
<tr>
<td>TSTQ</td>
<td>104</td>
<td>RGEN QUEUE</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>7</td>
<td>—</td>
<td>—</td>
<td>Test Queue</td>
<td></td>
</tr>
<tr>
<td>TXA</td>
<td>141034</td>
<td>GEN MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer X to A</td>
<td></td>
</tr>
<tr>
<td>TYA</td>
<td>141124</td>
<td>GEN MOVE</td>
<td>V</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Transfer Y to A</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>VIRY</td>
<td>000311</td>
<td>GEN</td>
<td>INTGY</td>
<td>SVRI</td>
<td>G</td>
<td>H</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>Verify</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>WAIR</td>
<td>000315</td>
<td>AP</td>
<td>PRCX</td>
<td>VI</td>
<td>—</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>43</td>
<td>YY MRGR LOGIC</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Writeable Control Store</td>
<td></td>
</tr>
<tr>
<td>WCS</td>
<td>0018XX</td>
<td>GEN</td>
<td>MCTL</td>
<td>RVI</td>
<td>O</td>
<td>O</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>XAD</td>
<td>001100</td>
<td>DECI DECI</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>3</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Decimal Add</td>
<td></td>
</tr>
<tr>
<td>XBT</td>
<td>001145</td>
<td>DECI DECI</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Binary to Decimal Conversion</td>
<td></td>
</tr>
<tr>
<td>XCA</td>
<td>140104</td>
<td>GEN MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Exchange and Clear A</td>
<td></td>
</tr>
<tr>
<td>XCB</td>
<td>140204</td>
<td>GEN MOVE</td>
<td>SRV</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Exchange and Clear B</td>
<td></td>
</tr>
<tr>
<td>XCM</td>
<td>001102</td>
<td>DECI DECI</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Decimal Compare</td>
<td></td>
</tr>
<tr>
<td>XDTB</td>
<td>001146</td>
<td>DECI DECI</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>5</td>
<td>—</td>
<td>—</td>
<td>Decimal to Binary Conversion</td>
<td></td>
</tr>
<tr>
<td>XDV</td>
<td>001107</td>
<td>DECI DECI</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Decimal Divide</td>
<td></td>
</tr>
<tr>
<td>XEC</td>
<td>01 02</td>
<td>MR PCTLJ</td>
<td>RV</td>
<td>F</td>
<td>H</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td>XED</td>
<td>001112</td>
<td>DECI DECI</td>
<td>VI</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Numeric Edit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XH</td>
<td>53</td>
<td>YY MRGR LOGIC</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Exclusive OR Fullword</td>
<td></td>
</tr>
<tr>
<td>XMP</td>
<td>001104</td>
<td>DECI DECI</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>3</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Decimal Multiply</td>
<td></td>
</tr>
<tr>
<td>XMV</td>
<td>001101</td>
<td>DECI DECI</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Decimal Move</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>XVRY</td>
<td>001113</td>
<td>MCTL GEN</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>Verify XIS</td>
<td></td>
</tr>
<tr>
<td>ZCM</td>
<td>001117</td>
<td>CHAR CHAR</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Compare Character Field</td>
<td></td>
</tr>
<tr>
<td>ZED</td>
<td>001111</td>
<td>CHAR CHAR</td>
<td>VI</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Character Edit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZFIL</td>
<td>001116</td>
<td>CHAR CHAR</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Fill Character Field</td>
<td></td>
</tr>
<tr>
<td>ZM</td>
<td>43</td>
<td>NN MRNR CLEAR</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Clear Fullword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMH</td>
<td>53</td>
<td>NN MRNR CLEAR</td>
<td>I</td>
<td>—</td>
<td>—</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Clear Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMV</td>
<td>001114</td>
<td>CHAR CHAR</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Move Character Field</td>
<td></td>
</tr>
<tr>
<td>ZMVD</td>
<td>001115</td>
<td>CHAR CHAR</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Move Equal Length Fields</td>
<td></td>
</tr>
<tr>
<td>ZTRN</td>
<td>001110</td>
<td>CHAR CHAR</td>
<td>VI</td>
<td>—</td>
<td>U</td>
<td>H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Translate Character Fields</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>REL</th>
<th>PSEUDO-OPERATION - USE RELOCATABLE ADDRESSING</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA = '123</td>
<td>MACHINE INSTRUCTION - LITERAL</td>
</tr>
<tr>
<td>CALL EXIT</td>
<td>PSEUDO-OPERATION - SUBROUTINE CALL</td>
</tr>
<tr>
<td>END</td>
<td>PSEUDO-OPERATION - END OF SOURCE CODE</td>
</tr>
</tbody>
</table>

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**
  - Column 1 contains an apostrophe (’). The rest of the line is used as a page title for subsequent pages.

- **Heading Lines**
  - 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

\[ A + 3 \]

Comment

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 \(\text{ALPA/5}\).

BETA.GE.A+$FF Logic expression consisting of a variable \(\text{BETA}\) 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 assembler 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)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRGR</td>
<td>op r.addr</td>
</tr>
<tr>
<td>MRFR</td>
<td>op f.addr</td>
</tr>
<tr>
<td>MRNR</td>
<td>op addr</td>
</tr>
<tr>
<td>IBRN</td>
<td>( { r } \text{.word} )</td>
</tr>
<tr>
<td>* IBRN bit test</td>
<td>op r.bitno. word</td>
</tr>
<tr>
<td>RGEN</td>
<td>op r</td>
</tr>
<tr>
<td>GEN</td>
<td>op</td>
</tr>
<tr>
<td>AP</td>
<td>op ap</td>
</tr>
<tr>
<td>** RGEN field register</td>
<td>op falr.r</td>
</tr>
<tr>
<td>** BRAN field register</td>
<td>op falr.data</td>
</tr>
<tr>
<td>** AP field register</td>
<td>op falr.ap</td>
</tr>
</tbody>
</table>

* 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)
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
RLIT
ENT

Sets up segmented address space
Puts literals in the procedure area
Entry point declarations

Code

Executable code

Data Area

DYNM
LINK
ECB

Stack variable declarations
defines linkage area containing static variables
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-

Calling sequence (Fortran):

INTEGER ARY(10), NZERO, ADDARY, RESULT
... RESULT = ADDARY (ARY, NZERO)

Calling sequence (PMA):

CALL ADDARY
AP ARY,S
AP NZERO,SL
... RESULT RETURNED IN (A) (GR2H)

ENT ADDARY,ADDECB ENTRY DECLARATION
SEGR 321-MODE SEGMENTED ADDRESSING
PLACE LITERALS IN PROCEDURE FRAME

ADDARY EQ *
ECB CAUSES CONTROL TO BE PASSED HERE ON CALL
TRANSFER POINTERS TO ARGUMENTS

LR 1,-10  # ELEMENTS TO ADD
ZMH NZERO,* INITIALIZE # ELEMENTS WHICH = 0
EH 3,=0 INITIALIZE ACCUMULATOR

ADDLP EQ *

LA 2,ARY,* PICK UP NEXT ARRAY ELEMENT
BRNE 2,LP18 TEST ZERO
IMH NZERO,* VALUE IS ZERO, BUMP COUNTER
AR 3,2 COMPUTE NEW SUM (RESULT => GR3H)
IMH ARY+1 UPDATE WORD# OF POINTER TO NEXT ARRAY ELEMENT
BH1 1,ADDLP BRANCH IF NOT DONE, PROCESS NEXT ELEMENT

TH 3,SUM SO WE CAN PRINT SUM
CALL TNOA
AP =C'RESULT IS ','S
AP =10,SL
CALL TODEC PRINT DECIMAL RESULT
AP SUM,SL
CALL TONL PRINT NEW-LINE

---THRU HERE WHEN DONE - RETURN WITH SUM IN THE GR2H (A-REGISTER).

---DATA DEFINITION:

DYNM ARY(3),NZERO(3),SUM
(0064) *
(0065) *
(0066) LINK
(0067) *

000000> 000000 (0068) ADDECB ECB ADDARY,,ARY,2
000022
000012
000002
177400
010000

(0069) *
(0070) *
(0071) *

000420 (0072) EWD

00043> 000000.000000E
00044> 000000.000000E
00045> 000000.000000E
00046> 000000.000000E
00047> 000000.000000E
00045> 000000.000000E

TEXT SIZE: PROC 000051 LINK 000026 STACK 000021
ADDARY 000000 0031 0068
ADDECB 000000 0068
ADDLP 000007 0039 0045
ARY 000012S 0040 0044 0063 0068
LP1O 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) *
(0010) *
(0011) *
(0012) *
(0013) *
(0014) *
(0015) *
(0016) *
(0017) *
(0018) *
(0019) *
(0020) *
(0021) *
(0022) *
(0023) *
(0024) *
(0025) *
(0026) *
(0027) *
(0028) *
(0029) *
(0030) *
(0031) *
(0032) *
(0033) *
(0034) *
(0035) *
(0036) *
(0037) *
(0038) *
(0039) *
(0040) *
(0041) *
(0042) *
(0043) *
(0044) *
(0045) *
(0046) *
(0047) *
(0048) *
(0049) *
(0050) *
(0051) *
(0052) *
(0053) *
(0054) *
(0055) *
(0056) *
(0057) *
(0058) *
(0059) *
(0060) *
(0061) *
(0062) *
(0063) *
(0064) *
(0065) *
(0066) *
(0067) *
(0068) *
(0069) *
(0070) *
(0071) *
(0072) *
(0073) *
(0074) *

Calling sequence (Fortran):

INTEGER ARY(10), NZERO, ADDARY, RESULT
RESULT = ADDARY (ARY, NZERO)

Calling sequence (PMA):

CALL ADDARY
DAC ARY ARRAY WHOSE ELEMENTS ARE TO BE SUMMED
DAC NZERO RETURNED # ELEMENTS = 0
DEC 0 (TO TERMINATE ARG LIST)
STA RESULT RESULT RETURNED IN (A)

---ENTRY DECLARATION---
ENT ADDARY C64R CHECK 64R MODE ADDRESSING VIOLATIONS
REL ADDARY RELATIVE MODE ASSEMBLY

0000: 00.000000A ADDARY DAC ** RETURN ADDRESS SAVED HERE BY 'JST' INSTRUCTION
00001: 10.000000E CALL PSAT TRANSFER ARGUMENTS' ADDRESSES
00002: 00.000000E DATA 2 (2 ARGUMENTS)
00003: 00.000000A ARY DAC ** PTR TO ARRAY
00004: 00.000000A NZERO DAC ** PTR TO COUNTER FOR ELEMENTS W/ VALUE = 0
00005: 35.0000003 LDX -10 # ELEMENTS TO ADD
00006: 14.0000000 CRA INITIALIZE COUNTERS
00007: 04.0000032 STA SUM # ELEMENTS WHICH = 0
00010: 44.0000000 STA NZERO,* NOT DONE, PROCESS NEXT ELEMENT

---LOOP TO PERFORM ACTUAL ADDITION---

00011: 42.0000003 LDA ARY,* PICK UP NEXT ARRAY ELEMENT
00012: 10148100 SNZ TEST ZERO
00013: 52.000004 IRS NZERO,* VALUE IS ZERO, BUMP COUNTER
00014: 06.0000032 ADD SUM COMPUTE NEW SUM
00015: 04.0000032 STA SUM BUMP POINTER TO NEXT ARRAY ELEMENT
00016: 12.0000003 IRS ARY TEST DONE
00017: 140114 IRX ADDLP EQU * NOT DONE, PROCESS NEXT ELEMENT
00018: 01.0000011 JMP ADDLP NOT DONE, PROCESS NEXT ELEMENT

---PRINT RESULT ON USER TERMINAL---

00021: 10.000000E CALL TNOUA TEXT TO BE PRINTED
00022: 00.000034 DAC =C'RESULT IS ' CHARACTERS
00023: 00.000041 DAC =10 # CHARACTERS
00024: 000000 DEC 0 PRINT DECIMAL RESULT
00025: 10.000000E CALL TODSC PRINT DECIMAL RESULT
00026: 10.000000E CALL TONL PRINT NEW-LINE

---THRU HERE WHEN DONE - RETURN WITH SUM IN THE A-REGISTER.

00030: 02.000032 LDA SUM TEMPORARY SUM
00031: 41.0000000 JMP ADDARY,* BACK TO CALLER

---DATA DECLARATION---

00032: SUM BSS 1 TEMPORARY SUM
00033: 00.177766A
00034: 00.151305A
00035: 00.151725A

FDR 3059 14-12 1 March. 1979
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
PSAT 00000E 0033
NZERO 000004 0036 0041 0047
SUM 000032 0040 0049 0049 0060 0064 0070
TNOUA 00000E 0055
TODEC 00000E 0059
TONL 00000E 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) *
(0010) * Calling sequence (Fortran):
(0011) *
(0012) * INTEGER ARY(10), NZERO, ADDARY, RESULT
(0013) *
(0014) * RESULT = ADDARY (ARY, NZERO)
(0015) *
(0016) * Calling sequence (PMA):
(0017) *
(0018) * CALL ADDARY
(0019) * AP ARY,S
(0020) * AP NZERO,SL
(0021) * ...
(0022) * RESULT RETURNED IN (A) (GR2H)
(0023) *
000400 000424 *
000426 000440 *
ENT ADDARY,ADDECB ENTRY DECLARATION
000442 000456 *
SEG 64V-MODE SEGMENTED ADDRESSING
000458 000472 *
RLIT PLACE LITERALS IN PROCEDURE FRAME
000474 000488 *
000490 0004a0 *
0004a2 0004b6 *
0004b8 0004d2 *
ADDARY EQU * EBC CAUSES CONTROL TO BE PASSED HERE ON CALL
0004d4 0004e8 *
ARGT TRANSFER POINTERS TO ARGUMENTS
0004ea 000502 *
000504 000518 *
LDX -=10 # ELEMENTS TO ADD
000520 000534 *
CRA INITIALIZE COUNTERS
000536 000550 *
STA SUM # ELEMENTS WHICH = 0
000552 000566 *
000568 000578 *
000580 000594 *
ADDLP EQUS* PICK UP NEXT ARRAY ELEMENT
000596 0005a0 *
LDA ARY,* TEST ZERO
0005a2 0005b6 *
BNE LP10 VALUE IS ZERO, BUMP COUNTER
0005b8 0005cc *
LP10 ADD SUM COMPUTE NEW SUM
0005cd 0005e6 *
STA SUM
0005e8 000602 *
IRS NZERO,* PRINT DECIMAL RESULT
000604 000618 *
IRS ARY+1 UPDATE WORD4 OF POINTER TO NEXT ARRAY ELEMENT
00061a 000634 *
BIX ADDLP BRANCH IF NOT DONE, PROCESS NEXT ELEMENT
000636 00064a *
00064c 000660 *
000662 00067c *
CALL TNOUA PRINT RESULT ON USER TERMINAL...
00067e 000698 *
CALL TODEC PRINT NEW-LINE
0006a0 0006b4 *
CALL TOWL PRINT NEW-LINE
0006b6 0006c0 *
(0057) ---THRU HERE WHEN DONE - RETURN WITH SUM IN THE A-REGISTER.
0006c2 0006d6 *
LDA SUM BACK TO CALLER
0006d8 0006e2 *
0006e4 0006f8 *
(0065) * DYNM ARY(3),NZERO(3),SUM
000700 000714 *
000716 00072a *
00072c 000740 *
000742 000756 *
000758 00076c *
000770 000784 *
000786 0007a0 *
0007a2 0007b6 *
0007b8 0007c2 *
0007c4 0007d8 *
0007da 0007e4 *
0007e6 0007f0 *
0007f2 0007f6 *
ADDARY ECB ADDARY,,ARY,2
```

(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  

00420: 000000.000000E  
00422: 000000.000000E  
00424: 000000.000000E  

TEXT SIZE: PROC 000046 LINK 000026 STACK 000021

ADDARY 000000 0031 0069  
ADDCBC 000000L 0069  
ADDP 000000 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  
T_NOU 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.

<table>
<thead>
<tr>
<th>Precision</th>
<th>Address Mode</th>
<th>Range</th>
</tr>
</thead>
</table>
| Single    | SRVI         | From: -32.768 \[2^{15}\]  
             |              | To: +32.767 \[2^{16}-1\] |
| Double    | SR           | From: -1.073.741.824 \[2^{30}\]  
             |              | To: +1.073.741.823 \[2^{31}\]  |
| Double    | VI           | From: -2.147.483.648 \[2^{31}\]  
             |              | To: +2.147.483.647 \[2^{32}\]  |

**Decimal:** Whole number, base 10 quantities

Data Type Symbol: none

<table>
<thead>
<tr>
<th>Precision</th>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>DATA 123</td>
<td>000173</td>
</tr>
<tr>
<td></td>
<td>DEC 123</td>
<td>000173</td>
</tr>
<tr>
<td></td>
<td>123</td>
<td>000173</td>
</tr>
<tr>
<td>Double Precision</td>
<td>DATA 123L</td>
<td>000000 000173</td>
</tr>
<tr>
<td></td>
<td>DEC 123L</td>
<td>000000 000173</td>
</tr>
<tr>
<td></td>
<td>123L</td>
<td>000000 000173</td>
</tr>
</tbody>
</table>

1 March, 1979  
15-1  
FDR 3059
### Table 15-1. Numeric Constants

<table>
<thead>
<tr>
<th>Class</th>
<th>Source</th>
<th>Symbol</th>
<th>Pseudo-Op</th>
<th>Binary Scale</th>
<th>Notes</th>
<th>Exponent</th>
<th>Notes</th>
<th>Precision</th>
<th>Symbol</th>
<th>Expression</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>D</td>
<td>DEC</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>Single</td>
<td>—</td>
<td>YES</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Double</td>
<td>L²</td>
<td>NO</td>
<td>1234L</td>
</tr>
<tr>
<td>O</td>
<td></td>
<td>OCT</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Single</td>
<td>—</td>
<td>YES</td>
<td>’123</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OCT</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Double</td>
<td>L³</td>
<td>NO</td>
<td>’1234L</td>
</tr>
<tr>
<td>H</td>
<td>S</td>
<td>HEX</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Single</td>
<td>—</td>
<td>YES</td>
<td>$A8</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>HEX</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Double</td>
<td>L</td>
<td>NO</td>
<td>$A8L</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>%</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Single</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B⁺</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Double</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BB</td>
<td>Req. E</td>
<td>—</td>
<td>—</td>
<td>Opt.</td>
<td>Double</td>
<td>—</td>
<td>NO</td>
<td>125BB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BBB</td>
<td>Req. E</td>
<td>—</td>
<td>—</td>
<td>Opt.</td>
<td>Triple</td>
<td>—</td>
<td>NO</td>
<td>125BBB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BBBB</td>
<td>Req. E</td>
<td>—</td>
<td>—</td>
<td>Opt.</td>
<td>Quadruple</td>
<td>—</td>
<td>NO</td>
<td>125BBBB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>4</td>
<td>D</td>
<td>Req.</td>
<td>Double</td>
<td>—</td>
<td>—</td>
<td>NO</td>
<td>1.23D-6</td>
</tr>
</tbody>
</table>

- **Binary**
- **Decimal**
- **Floating Point**
- **Fixed Point**
- **Octal**: Whole number, base 8 quantities
  - Data Type Symbol: Apostrophe ('), letter O plus single quotes (O’

#### Precision

<table>
<thead>
<tr>
<th>Precision</th>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA <code>123</code></td>
<td>000123</td>
<td></td>
</tr>
<tr>
<td>DATA <code>O123</code></td>
<td>000123</td>
<td></td>
</tr>
<tr>
<td>OCT 123</td>
<td>000123</td>
<td></td>
</tr>
<tr>
<td>'123'</td>
<td>000123</td>
<td></td>
</tr>
<tr>
<td>O'123'</td>
<td>000123</td>
<td></td>
</tr>
<tr>
<td>Double Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA <code>123L</code></td>
<td>0000000 000123</td>
<td></td>
</tr>
<tr>
<td>OCT 123L</td>
<td>0000000 000123</td>
<td></td>
</tr>
<tr>
<td>'123L'</td>
<td>0000000 000123</td>
<td></td>
</tr>
</tbody>
</table>

#### Hexadecimal

- Whole number, base 16 quantities. The hexadecimal digit values are:

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 9</td>
<td>0 - 9</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

- Data Type Symbol: Dollar Sign [$], letter X plus single quotes (‘’)
<table>
<thead>
<tr>
<th>Precision</th>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Precision</strong></td>
<td>DATA $30BF'</td>
<td>030277</td>
</tr>
<tr>
<td></td>
<td>DATA 'X30BF</td>
<td>030277</td>
</tr>
<tr>
<td></td>
<td>HEX 30BF</td>
<td>030277</td>
</tr>
<tr>
<td></td>
<td>'X30BF'</td>
<td>030277</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>DATA $30BFL</td>
<td>000000 030277</td>
</tr>
<tr>
<td></td>
<td>HEX 30BFL</td>
<td>000000 030277</td>
</tr>
<tr>
<td></td>
<td>$30BFL</td>
<td>000000 030277</td>
</tr>
</tbody>
</table>

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’ ).

<table>
<thead>
<tr>
<th>Precision</th>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Precision</strong></td>
<td>DATA %11100101</td>
<td>000345</td>
</tr>
<tr>
<td></td>
<td>DATA '11100101'</td>
<td>000345</td>
</tr>
<tr>
<td></td>
<td>% '11100101'</td>
<td>000345</td>
</tr>
<tr>
<td></td>
<td>'B11100101'</td>
<td>000345</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>DATA %11100101L</td>
<td>000000 000345</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>B4</td>
<td>B5</td>
<td>B6</td>
<td>B7</td>
<td>B8</td>
<td>B9</td>
<td>B10</td>
<td>B11</td>
<td>B12</td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>123B15</td>
<td>000173</td>
</tr>
<tr>
<td>123B7</td>
<td>075400</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 1 1 1 0 1 1</td>
</tr>
<tr>
<td>0 1 1 1 1 0 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

1 March, 1979
15-3
FDR 3059
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.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Listing Representation</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>123B18</td>
<td>000017</td>
<td>![Bit Pattern Image]</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Listing Representation</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.5B7</td>
<td>075600</td>
<td>![Bit Pattern Image]</td>
</tr>
</tbody>
</table>

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, and 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 \( K_{10} \) \( (2^{*-n}) \), where \( K_{10} \) is the decimal constant, \( K_{10} \) 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:**

<table>
<thead>
<tr>
<th>Precision</th>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Precision</td>
<td>123E1BB17</td>
<td>000463 100000</td>
</tr>
<tr>
<td>Single Precision</td>
<td>123E0B15</td>
<td>000173</td>
</tr>
<tr>
<td>Single Precision</td>
<td>123E1B15</td>
<td>002316</td>
</tr>
</tbody>
</table>

**Non-SEG examples:**

<table>
<thead>
<tr>
<th>Precision</th>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Precision</td>
<td>123E1BB17</td>
<td>000463 040000</td>
</tr>
<tr>
<td>Single Precision</td>
<td>123E0B15</td>
<td>000173</td>
</tr>
<tr>
<td>Single Precision</td>
<td>123E1B15</td>
<td>002316</td>
</tr>
</tbody>
</table>
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).

<table>
<thead>
<tr>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.28E2</td>
<td>040000 000210</td>
</tr>
<tr>
<td>1.28</td>
<td>050753 102601</td>
</tr>
<tr>
<td>-11.28</td>
<td>122702 107604</td>
</tr>
<tr>
<td>1.28E-14</td>
<td>071512 145122</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C’AB’</td>
<td>140702</td>
</tr>
<tr>
<td>C’A’</td>
<td>140640</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Listing Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C’A’</td>
<td>140640</td>
</tr>
<tr>
<td>R’A’</td>
<td>000301</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>S</th>
<th>MANTISSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MANTISSA</th>
<th>EXPONENT (EXCESS 128)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
</tbody>
</table>

### B. Double Precision Floating Point

<table>
<thead>
<tr>
<th>S</th>
<th>MANTISSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MANTISSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MANTISSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
</tr>
<tr>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EXPONENT (EXCESS 128)</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

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.
- `'A` ASCII constant.
- `ALPHA` Symbol.
- `1.23E2` Invalid because it is a floating point number; it does not have a single precision integer value.
- `'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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABSYM</td>
<td>LABSYM LDA LOC</td>
<td>LABSYM is a label symbol whose value is a location (program counter value) of the instruction LDA LOC.</td>
</tr>
<tr>
<td>DATSYM</td>
<td>DATSYM DATA <code>10</code></td>
<td>DATSYM is a label symbol whose value is the location (program counter value) of the constant <code>10</code>.</td>
</tr>
<tr>
<td>ADSYM</td>
<td>ADSYM DAC LOC</td>
<td>ADSYM is a label symbol whose value is the location (program counter value) of the address constant LOC.</td>
</tr>
<tr>
<td>ABSSYM</td>
<td>ABSSYM EQU <code>10</code></td>
<td>ABSSYM is a symbol whose value is <code>10</code>.</td>
</tr>
<tr>
<td>CHRSYM</td>
<td>CHRSYM EQU <code>'A</code></td>
<td>CHRSYM is a symbol whose octal value is 140640.</td>
</tr>
</tbody>
</table>

**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>
<thead>
<tr>
<th>Table 15-2. Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mode</strong></td>
</tr>
<tr>
<td>Absolute</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Relative (non-SEG) Procedure (SEG)</td>
</tr>
<tr>
<td>Relative Common</td>
</tr>
<tr>
<td>Stack Relative</td>
</tr>
<tr>
<td>Stack Relative</td>
</tr>
<tr>
<td>Stack Relative</td>
</tr>
<tr>
<td>Stack Relative</td>
</tr>
<tr>
<td>External</td>
</tr>
<tr>
<td>Procedure Absolute</td>
</tr>
<tr>
<td>Linkage Base Relative</td>
</tr>
<tr>
<td>Linkage Base Relative</td>
</tr>
<tr>
<td>Temporary Base Relative</td>
</tr>
</tbody>
</table>

Notes:
1. If PB/4 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result (Octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>'3+4</td>
<td>0000017</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>'10-3</td>
<td>0000005</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>'20*10</td>
<td>0000200</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>'20/10</td>
<td>0000002</td>
</tr>
</tbody>
</table>

Logical operators: Perform a logical operation on two 16-bit operands:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result (Octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.OR.</td>
<td>Logical OR</td>
<td>'123.OR.'456</td>
<td>0000577</td>
</tr>
<tr>
<td>.XOR.</td>
<td>Logical XOR</td>
<td>'123.XOR.'456</td>
<td>0000575</td>
</tr>
<tr>
<td>.AND.</td>
<td>Logical AND</td>
<td>'123.AND.'456</td>
<td>0000002</td>
</tr>
</tbody>
</table>
Relational operators: Perform a comparison of two 16-bit operands with a result of 0 if false and 1 if true.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relation</th>
<th>Example</th>
<th>Result (Octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.EQ.</td>
<td>Equal</td>
<td>'123.EQ.'123</td>
<td>000001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'123.EQ.'456</td>
<td>000000</td>
</tr>
<tr>
<td>.NE.</td>
<td>Not equal</td>
<td>'123.NE.'123</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'123.NE.'456</td>
<td>000001</td>
</tr>
<tr>
<td>.GT.</td>
<td>Greater than</td>
<td>'123.GT.'123</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'456.GT.'123</td>
<td>000001</td>
</tr>
<tr>
<td>.GE.</td>
<td>Greater than</td>
<td>'123.GE.'123</td>
<td>000001</td>
</tr>
<tr>
<td></td>
<td>or equal</td>
<td>'123.GE.'456</td>
<td>000000</td>
</tr>
<tr>
<td>.LE.</td>
<td>Less than</td>
<td>'123.LE.'123</td>
<td>000001</td>
</tr>
<tr>
<td></td>
<td>or equal</td>
<td>'123.LE.'456</td>
<td>000001</td>
</tr>
<tr>
<td>.LT.</td>
<td>Less than</td>
<td>'123.LT.'456</td>
<td>000001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'456.LT.'123</td>
<td>000000</td>
</tr>
</tbody>
</table>

Shift operators: Perform logical right or left shift of an expression, using the syntax:

\[
\text{argument-expression} \{ .LS. \text{shift-count-expression} \} \{ .RS. \}
\]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Result (Octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LS.</td>
<td>Left shift</td>
<td>'123.LS.'3</td>
<td>001230</td>
</tr>
<tr>
<td>.RS.</td>
<td>Right shift</td>
<td>'123.RS.'3</td>
<td>000012</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>* /</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
</tr>
<tr>
<td></td>
<td>.RS. .LS.</td>
</tr>
<tr>
<td></td>
<td>.GT. .GE. .EQ. .NE. .LE. .LT.</td>
</tr>
<tr>
<td></td>
<td>.AND.</td>
</tr>
<tr>
<td></td>
<td>.OR.</td>
</tr>
<tr>
<td>Lowest</td>
<td>.XOR.</td>
</tr>
</tbody>
</table>

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

000005: 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 procedure origin, the FIN will have the effect of:

```
HERE   EQU   *
LINK
FIN
ORG   HERE
```

**ASSEMBLER ATTRIBUTES**

Assembler attributes 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.
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>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Class</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Set mode to absolute</td>
<td>AC</td>
<td>Not in SEG/SEGR MODE</td>
</tr>
<tr>
<td>AP</td>
<td>Argument pointer</td>
<td>AD</td>
<td>SEG/SEGR mode</td>
</tr>
<tr>
<td>BACK</td>
<td>Loop back</td>
<td>CA</td>
<td>Macros only</td>
</tr>
<tr>
<td>BCI</td>
<td>Define ASCII string</td>
<td>DD</td>
<td></td>
</tr>
<tr>
<td>BES</td>
<td>Allocate block ending with symbol</td>
<td>SA</td>
<td></td>
</tr>
<tr>
<td>BSS</td>
<td>Allocate block starting with symbol</td>
<td>SA</td>
<td></td>
</tr>
<tr>
<td>BSZ</td>
<td>Allocate block set to zeros</td>
<td>SA</td>
<td></td>
</tr>
<tr>
<td>C64R</td>
<td>Check 64R</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>External subroutine reference</td>
<td>PL</td>
<td></td>
</tr>
<tr>
<td>CENT</td>
<td>Conditional entry</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>COMM</td>
<td>FORTRAN compatible COMMON</td>
<td>SA</td>
<td></td>
</tr>
<tr>
<td>D16S</td>
<td>Use 16S address mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>D32I</td>
<td>Use 32I address mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>D32R</td>
<td>Use 32R address mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>D32S</td>
<td>Use 32S address mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>D64R</td>
<td>Use 64R address mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>D64V</td>
<td>Use 64V address mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>DAC</td>
<td>Address definition (Prime 100-300)</td>
<td>AD</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>Define data constant</td>
<td>DD</td>
<td></td>
</tr>
<tr>
<td>DDM</td>
<td>Use default mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>Define decimal integer constant</td>
<td>DD</td>
<td></td>
</tr>
<tr>
<td>DFTB</td>
<td>Define table block</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>DFVT</td>
<td>Define value table</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>DUII</td>
<td>Define UII</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>DYMN</td>
<td>Define stack relative symbol</td>
<td>SD</td>
<td></td>
</tr>
<tr>
<td>DYNT</td>
<td>Direct entry definition</td>
<td>PL</td>
<td></td>
</tr>
<tr>
<td>ECB</td>
<td>Entry control block</td>
<td>PL</td>
<td></td>
</tr>
<tr>
<td>EJCT</td>
<td>Eject page</td>
<td>LC</td>
<td></td>
</tr>
<tr>
<td>ELM</td>
<td>Enter loader mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>Reverse conditional assembly</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>End of source statements</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>ENDC</td>
<td>End conditional assembly area</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>ENDM</td>
<td>End macro definition</td>
<td>MD</td>
<td></td>
</tr>
<tr>
<td>ENT</td>
<td>Define entry point</td>
<td>PL</td>
<td></td>
</tr>
<tr>
<td>EVEN</td>
<td>Outputs NOP if current address not even</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>EQU</td>
<td>Fixed symbol definition</td>
<td>SD</td>
<td></td>
</tr>
<tr>
<td>EXT</td>
<td>External reference</td>
<td>PL</td>
<td></td>
</tr>
<tr>
<td>FAIL</td>
<td>Force error message</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>FIN</td>
<td>Insert literals</td>
<td>LT</td>
<td></td>
</tr>
<tr>
<td>GO</td>
<td>Forward reference</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>HEX</td>
<td>Define hexadecimal integer constant</td>
<td>DD</td>
<td></td>
</tr>
<tr>
<td>IFTT</td>
<td>If table true</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>IFTF</td>
<td>If table false</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>IFVT</td>
<td>If value true</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>IFVF</td>
<td>If value false</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>If true</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>IFx</td>
<td>Arithmetic conditional if</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>Indirect pointer</td>
<td>AD</td>
<td></td>
</tr>
<tr>
<td>LINK</td>
<td>Put code in linkage segment</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>LIR</td>
<td>Load if required</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>LIST</td>
<td>Enable listing</td>
<td>LC</td>
<td></td>
</tr>
<tr>
<td>LSMD</td>
<td>List macro expansions data only</td>
<td>LC</td>
<td></td>
</tr>
<tr>
<td>LST</td>
<td>List macro expansions</td>
<td>LC</td>
<td></td>
</tr>
<tr>
<td>MAC</td>
<td>Begin macro definition</td>
<td>MD</td>
<td></td>
</tr>
<tr>
<td>N64R</td>
<td>Not 64R</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>NLSM</td>
<td>Don’t list macro expansions</td>
<td>LC</td>
<td></td>
</tr>
<tr>
<td>NLST</td>
<td>Inhibit listing</td>
<td>LC</td>
<td></td>
</tr>
<tr>
<td>OCT</td>
<td>Define octal integer constant</td>
<td>DD</td>
<td></td>
</tr>
<tr>
<td>ORG</td>
<td>Define origin location</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>PCVH</td>
<td>Print cross reference values in HEX</td>
<td>LC</td>
<td></td>
</tr>
<tr>
<td>PROC</td>
<td>Put code in procedure segment</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>REL</td>
<td>Set mode to relocatable</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>RLIT</td>
<td>Optimize literals</td>
<td>LT</td>
<td></td>
</tr>
<tr>
<td>SAY</td>
<td>Print message</td>
<td>MD</td>
<td></td>
</tr>
<tr>
<td>SCT</td>
<td>Select code within macro</td>
<td>MD</td>
<td></td>
</tr>
<tr>
<td>SCTL</td>
<td>Select code from macro list</td>
<td>MD</td>
<td></td>
</tr>
<tr>
<td>SDM</td>
<td>Set default mode</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>SEG</td>
<td>Segmentation assembly - 64V</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>SEGR</td>
<td>Segmentation assembly - 32I</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>Changeable symbol definition</td>
<td>SD</td>
<td></td>
</tr>
<tr>
<td>SETB</td>
<td>Set base sector</td>
<td>LO</td>
<td></td>
</tr>
<tr>
<td>SUBR</td>
<td>Define entry point</td>
<td>PL</td>
<td></td>
</tr>
<tr>
<td>SYML</td>
<td>Allows long external names</td>
<td>PL</td>
<td></td>
</tr>
<tr>
<td>VFD</td>
<td>Define variable fields</td>
<td>DD</td>
<td></td>
</tr>
<tr>
<td>XAC</td>
<td>External address definition</td>
<td>AD</td>
<td></td>
</tr>
<tr>
<td>XSET</td>
<td>Changeable symbol definition</td>
<td>SD</td>
<td></td>
</tr>
</tbody>
</table>

*SEGR mode*

- Must be first statement in listing
- Must be first statement in listing
Figure 16-1. Pseudo-operations.
ASSEMBLY CONTROL PSEUDO-OPERATIONS (AC)

Affect the actions of PMA during the assembly process.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Set mode to absolute</td>
<td>Not in SEG/SEGR mode</td>
</tr>
<tr>
<td>C64R</td>
<td>Check 64R</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>End of source statements</td>
<td></td>
</tr>
<tr>
<td>EVEN</td>
<td>Outputs NOP if current address not even.</td>
<td></td>
</tr>
<tr>
<td>LINK</td>
<td>Put code in linkage segment</td>
<td>SEG/SEGR mode</td>
</tr>
<tr>
<td>ORG</td>
<td>Define origin location</td>
<td></td>
</tr>
<tr>
<td>PROC</td>
<td>Put code in procedure segment</td>
<td>SEG/SEGR mode</td>
</tr>
<tr>
<td>REL</td>
<td>Set mode to relocatable</td>
<td>Not in SEG/SEGR mode</td>
</tr>
<tr>
<td>SEG</td>
<td>Segmentation assembly - 64V</td>
<td>Must appear before executable code</td>
</tr>
<tr>
<td>SEGR</td>
<td>Segmentation assembly -32I</td>
<td>Must appear before executable code</td>
</tr>
</tbody>
</table>

► 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 SEGR — 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>Argument pointer</td>
<td>SEG/SEGR mode</td>
</tr>
<tr>
<td>DAC</td>
<td>Local address definition</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>Indirect pointer</td>
<td>SEG/SEGR mode</td>
</tr>
<tr>
<td>XAC</td>
<td>External address definition</td>
<td></td>
</tr>
</tbody>
</table>

**[label] AP address-expression [,.modifier]**

Generates an argument pointer in the form used by the 64 V/321 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:

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Address Length</th>
<th>Index</th>
<th>Indirection</th>
</tr>
</thead>
<tbody>
<tr>
<td>16S</td>
<td>14</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>32S</td>
<td>15</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>32R</td>
<td>15</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>64R</td>
<td>16</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>64V</td>
<td>16</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

*[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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACK,</td>
<td>Loop back</td>
<td>Macros only</td>
</tr>
<tr>
<td>BACK TO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFTB</td>
<td>Define table block</td>
<td></td>
</tr>
<tr>
<td>DFVT</td>
<td>Define value table</td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>Reverse conditional assembly</td>
<td></td>
</tr>
<tr>
<td>ENDC</td>
<td>End conditional assembly area</td>
<td></td>
</tr>
<tr>
<td>FAIL</td>
<td>Force error message</td>
<td></td>
</tr>
<tr>
<td>GO</td>
<td>Forward reference</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>Conditional</td>
<td></td>
</tr>
<tr>
<td>IFx</td>
<td>Conditional</td>
<td></td>
</tr>
<tr>
<td>IFTF</td>
<td>If table false</td>
<td></td>
</tr>
<tr>
<td>IFTT</td>
<td>If table true</td>
<td></td>
</tr>
<tr>
<td>IFVF</td>
<td>If value false</td>
<td></td>
</tr>
<tr>
<td>IFVT</td>
<td>If value true</td>
<td></td>
</tr>
</tbody>
</table>

➤ 

\[
\text{LABEL-1} \left\{ \text{BACK, BACK TO} \right\} \text{LABEL-2};
\]

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:

\[
\begin{align*}
A & \quad \text{EQU} \quad 5 \\
X & \quad \text{DFTB} \quad (A,1),(B,2) \\
X & \quad \text{EQU} \quad 6
\end{align*}
\]

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] 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.
\[\text{GO} \quad \{\text{GO TO} \}\]

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCI</td>
<td>Define ASCII string</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>Define data constant</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>Define decimal integer constant</td>
<td></td>
</tr>
<tr>
<td>HEX</td>
<td>Define hexadecimal integer constant</td>
<td></td>
</tr>
<tr>
<td>OCT</td>
<td>Define octal integer constant</td>
<td></td>
</tr>
<tr>
<td>VFD</td>
<td>Define variable fields</td>
<td></td>
</tr>
</tbody>
</table>

\[\text{[label]} \text{BCI} \left\{ \text{string} \right\}
\quad \{n, \text{string} \}\]

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.

\[\text{[label]} \text{DATA} \left[\text{[absolute-expression-1]}\right] \text{absolute-expression-2}, \ldots\]

Defines \(\text{absolute-expression-2} \ \text{absolute-expression-1}\) 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.

\[\text{[label]} \text{DEC} \text{decimal-integer-constant}, \ldots\]

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJCT</td>
<td>Eject page</td>
<td></td>
</tr>
<tr>
<td>LIST</td>
<td>Enable listing</td>
<td></td>
</tr>
<tr>
<td>LSMD</td>
<td>List macro expansions data only</td>
<td></td>
</tr>
<tr>
<td>LSTM</td>
<td>List macro expansions</td>
<td></td>
</tr>
<tr>
<td>NLSM</td>
<td>Don’t list macro expansions</td>
<td></td>
</tr>
<tr>
<td>NLST</td>
<td>Inhibit listing</td>
<td></td>
</tr>
<tr>
<td>PCVH</td>
<td>Print cross reference values in hexadecimal</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIN</td>
<td>Insert literals</td>
<td></td>
</tr>
<tr>
<td>RLIT</td>
<td>Optimize literals</td>
<td></td>
</tr>
</tbody>
</table>

[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:

<table>
<thead>
<tr>
<th>REL</th>
<th>REL</th>
<th>REL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0001)</td>
<td>RLIT</td>
<td>'123 IS AN OCTAL LITERAL</td>
</tr>
<tr>
<td>(0002)</td>
<td>LDA = '123</td>
<td>'123 IS AN OCTAL LITERAL</td>
</tr>
<tr>
<td>000008:</td>
<td>02.00001</td>
<td>000003:</td>
</tr>
<tr>
<td>000001:</td>
<td>00.000123A</td>
<td>LDA = '123</td>
</tr>
<tr>
<td>*</td>
<td>02.00001</td>
<td>000005:</td>
</tr>
<tr>
<td>000003:</td>
<td>04.000005</td>
<td>LDA = '456</td>
</tr>
<tr>
<td>*</td>
<td>02.000006</td>
<td>000006:</td>
</tr>
<tr>
<td>000004:</td>
<td>STA BUFF</td>
<td>END '10</td>
</tr>
<tr>
<td>000005:</td>
<td>00.00010A</td>
<td>END END</td>
</tr>
<tr>
<td>000006:</td>
<td>BUFF</td>
<td></td>
</tr>
<tr>
<td>000009:</td>
<td>00.000456A</td>
<td></td>
</tr>
</tbody>
</table>

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 DUUI, 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.

1 March, 1979

16-13

FDR 3059
<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENT</td>
<td>Conditional entry</td>
<td></td>
</tr>
<tr>
<td>D16S</td>
<td>Use 16S addressing mode</td>
<td></td>
</tr>
<tr>
<td>D32S</td>
<td>Use 32S addressing mode</td>
<td></td>
</tr>
<tr>
<td>D32R</td>
<td>Use 32R addressing mode</td>
<td></td>
</tr>
<tr>
<td>D64R</td>
<td>Use 64R addressing mode</td>
<td></td>
</tr>
<tr>
<td>D64V</td>
<td>Use 64V addressing mode</td>
<td></td>
</tr>
<tr>
<td>D32I</td>
<td>Use 32I addressing mode</td>
<td></td>
</tr>
<tr>
<td>DDM</td>
<td>Use default mode</td>
<td></td>
</tr>
<tr>
<td>DUII</td>
<td>Define UII</td>
<td></td>
</tr>
<tr>
<td>ELM</td>
<td>Enter loader mode</td>
<td></td>
</tr>
<tr>
<td>LIR</td>
<td>Load if required</td>
<td></td>
</tr>
<tr>
<td>N64R</td>
<td>Not 64R</td>
<td></td>
</tr>
<tr>
<td>SDM</td>
<td>Set default mode</td>
<td>Not 64V or 32I</td>
</tr>
<tr>
<td>SETB</td>
<td>Set base sector</td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–9</td>
<td>Must be 0</td>
</tr>
<tr>
<td>10</td>
<td>Prime 450 and up</td>
</tr>
<tr>
<td>11</td>
<td>Prime 300, 400</td>
</tr>
<tr>
<td>12</td>
<td>Undefined</td>
</tr>
<tr>
<td>13</td>
<td>Double Precision Floating Point</td>
</tr>
<tr>
<td>14</td>
<td>Single Precision Floating Point</td>
</tr>
<tr>
<td>15</td>
<td>Prime 300 Only</td>
</tr>
<tr>
<td>16</td>
<td>High Speed Arithmetic</td>
</tr>
</tbody>
</table>

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 ‘Né’ error.

SDM absolute-expression

Directs the loader to set its default addressing mode to absolute-expression. Legal values of the expression are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16S Mode</td>
</tr>
<tr>
<td>1</td>
<td>32S Mode</td>
</tr>
<tr>
<td>2</td>
<td>64R Mode</td>
</tr>
<tr>
<td>3</td>
<td>32R Mode</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENDM</td>
<td>End macro definition</td>
</tr>
<tr>
<td>MAC</td>
<td>Begin macro definition</td>
</tr>
<tr>
<td>SAY</td>
<td>Print message</td>
</tr>
<tr>
<td>SCT</td>
<td>Select code within macro</td>
</tr>
<tr>
<td>SCTL</td>
<td>Select code from list within macro</td>
</tr>
</tbody>
</table>

**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**

{\( \text{dummy-words, \ldots} \)
{\( \text{argument-values, \ldots} \)}
{\( \text{argument-identifiers, \ldots} \)}

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 column 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:

<table>
<thead>
<tr>
<th>Marker</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Ordinary marker line.</td>
</tr>
<tr>
<td>%1</td>
<td>If the preceding section of code was assembled, continue assembly from this marker to the next marker.</td>
</tr>
<tr>
<td>%2</td>
<td>If no other sections of code have been assembled, assemble from this line to the next marker.</td>
</tr>
<tr>
<td>%/</td>
<td>End of control range.</td>
</tr>
</tbody>
</table>

The %2 marker is useful to identify a section of code that is to be assembled if the argument value of the SCT statement is out of range.

<table>
<thead>
<tr>
<th>Argument Value</th>
<th>Assembly Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Assemble from the SCT statement to the first % marker; then skip to the %/ line.</td>
</tr>
<tr>
<td>1</td>
<td>Skip to the first % marker; assemble from there to the second % marker; then skip to the %/ marker.</td>
</tr>
<tr>
<td>n</td>
<td>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.</td>
</tr>
<tr>
<td>-n</td>
<td>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.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Assembly Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>-n</td>
</tr>
</tbody>
</table>

**Expression comparison**: The position on the argument list of the expression which equals absolute-expression-1 determines the result:

<table>
<thead>
<tr>
<th>Expression Position</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute-expression-1=absolute-expression-2</td>
<td>SCT 0</td>
</tr>
<tr>
<td>absolute-expression-1=absolute-expression-n</td>
<td>SCT n</td>
</tr>
<tr>
<td>no match</td>
<td>SCT -n</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>External subroutine reference</td>
<td></td>
</tr>
<tr>
<td>DYNMT</td>
<td>Direct entry call</td>
<td></td>
</tr>
<tr>
<td>ECB</td>
<td>Define entry control block</td>
<td>SEG/SEGR</td>
</tr>
<tr>
<td>EXT</td>
<td>Flag external reference</td>
<td></td>
</tr>
<tr>
<td>SUBR,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENT</td>
<td>Define entry point</td>
<td></td>
</tr>
</tbody>
</table>

**[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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry-point</td>
<td>Procedure relative value; entry point for subroutine.</td>
</tr>
<tr>
<td>link-base</td>
<td>Link base register value.</td>
</tr>
<tr>
<td>displacement</td>
<td>Displacement in stack frame for argument list. May be stack relative or absolute expression.</td>
</tr>
<tr>
<td>n-arguments</td>
<td>Number of arguments expected; default is zero.</td>
</tr>
<tr>
<td>stack-size</td>
<td>Initial stack frame size. Default is maximum area specified in DYNM statements.</td>
</tr>
<tr>
<td>keys</td>
<td>CPU keys for procedure. Default is 64V addressing mode, all other keys zero.</td>
</tr>
</tbody>
</table>

For example:

```
ENT ECBNAM
LAB1 LDA ='123
LINK
ECBNAM EOB 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSS</td>
<td>Allocate block starting with symbol</td>
<td></td>
</tr>
<tr>
<td>BES</td>
<td>Allocate block ending with symbol</td>
<td></td>
</tr>
<tr>
<td>BSZ</td>
<td>Allocate block set to zeroes</td>
<td></td>
</tr>
<tr>
<td>COMM</td>
<td>FORTRAN compatible COMMON</td>
<td></td>
</tr>
</tbody>
</table>

[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 COMMON 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNM</td>
<td>Declare stack relative</td>
<td></td>
</tr>
<tr>
<td>EQU</td>
<td>Symbol definition</td>
<td>R and V only</td>
</tr>
<tr>
<td>SET</td>
<td>Symbol definition</td>
<td></td>
</tr>
<tr>
<td>XSET</td>
<td>Symbol definition</td>
<td></td>
</tr>
</tbody>
</table>
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.
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:

\[
\begin{align*}
&\text{REL} \\
&\text{LSTM} \\
&\text{TRANSFER MAC} \\
&\text{LDA } <J> \\
&\text{STA } <K> \\
&\text{ENDM} \\
&J \text{ EQU 1} \\
&K \text{ EQU 2} \\
&\text{ENDM}
\end{align*}
\]

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:

<table>
<thead>
<tr>
<th>Assigned Local Label</th>
<th>Evaluation As</th>
<th>In Macro Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>002ABC</td>
<td>0002</td>
</tr>
<tr>
<td>X3A</td>
<td>1739X3A</td>
<td>1739</td>
</tr>
</tbody>
</table>

MACRO CALLS

A macro call is a special type of statement that uses the name of a defined macro in the operation field:

\[
\text{[label] macro-name \{ arguments, \ldots \} \{ dummy words, \ldots \} \{ argument identifiers, \ldots \}}
\]
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.

<table>
<thead>
<tr>
<th>Variable Field</th>
<th>Argument &lt;1&gt;</th>
<th>Argument &lt;2&gt;</th>
<th>Argument &lt;3&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A+3</td>
<td>A+3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X,Y-1,Z+A-1</td>
<td>X</td>
<td>Y-1</td>
<td>Z+A-1</td>
</tr>
<tr>
<td>X,B,C (Z3X2)</td>
<td>X</td>
<td>B-C</td>
<td>Z3X2</td>
</tr>
<tr>
<td>(A, B-1), C</td>
<td>A, B-1</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>(X, Y, [Z1+Z2],3)</td>
<td>X,Y,[Z1+Z2],3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In the following call to the TRANSFER macro.

```
TRANSFER ARG1 '1770
```

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 '1770
```

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:

```plaintext
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:

```plaintext
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:
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 BUFFER. 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 WAIT 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 100,(INPUT,SAVE)
INPUT BSS 2
SAVE  BSS 2
END

is assembled as:

WAIT 100,(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

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>L</th>
<th>ADR</th>
<th>F</th>
<th>X</th>
<th>N</th>
<th>Z</th>
<th>0</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Bit</td>
<td>Name</td>
<td>Meaning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>Carry Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>Precision: 0=Single; 1=Double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>L</td>
<td>Carry out of most significant bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4–6</td>
<td>ADR</td>
<td>Addressing Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00=16S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01=32S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011=32R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010=64R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110=64V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>0=Floating point exception faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>1=Integer exception faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>N</td>
<td>Negative result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Z</td>
<td>Zero result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11–16</td>
<td></td>
<td>Must be zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

► **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
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>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>TAP</th>
<th>PSD</th>
<th>VPSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>Access and print or alter contents of memory word</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>BREAKPOINT</td>
<td>Insert up to 10 breakpoints in program (TAP permits only 1 breakpoint)</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>BR</td>
<td>Display contents of base registers</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>COPY</td>
<td>Copy block to block</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>DEFINE</td>
<td>Define local symbols</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>DUMP</td>
<td>Print contents of block (or, in PSD and VPSD, write contents to optional file)</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>EFFECTIVE</td>
<td>Search for effective address under mask</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>Execute a subroutine</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>Execute segmented program</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>FILL</td>
<td>Fill block with constant</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>GO</td>
<td>Proceed from breakpoint</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>JUMPRACE</td>
<td>Trace JMP, JST, HLT instructions</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>KEYS</td>
<td>Update CPU status</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>LB</td>
<td>Load link base register</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>LIST</td>
<td>Print contents of address</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>LS</td>
<td>Load external symbols from map file and enter symbolic address mode</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>MAP</td>
<td>Type out all symbols with their values</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>MODE</td>
<td>Address mode selection</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>MONITOR</td>
<td>Execute program, reporting any reference to the specified effective address</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>NOT-EQUAL</td>
<td>Not-equal search for constant under mask</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>OPEN</td>
<td>Open file for memory dump or symbols</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>PATCH</td>
<td>Patch object program</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>PRINT</td>
<td>Print parameters</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>PROCEED</td>
<td>Remove breakpoint and proceed</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>QUIT</td>
<td>Return to PRIMOS (or SEG if SEG's VPSD)</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>RELOCATE</td>
<td>Alter relocation constant</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>RUN</td>
<td>Run object program</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>SB</td>
<td>Load stack base register</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>SEARCH</td>
<td>Search memory block for constant under mask</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>SN</td>
<td>Set segment number</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>SYMBOL</td>
<td>Enable/disable use of symbols in address typeout</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>TRACE</td>
<td>Trace object program</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Update memory word</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Compare contents of one block of memory with another</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>VERSION</td>
<td>Print PSD version and restart location</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>WHERE</td>
<td>List location and remaining repeat counts for all breakpoints</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>XB</td>
<td>Load temporary base register</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>XR</td>
<td>Load X register</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>YR</td>
<td>Load Y register</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>ZERO</td>
<td>Zero breakpoint location</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSD20</td>
<td>'20000</td>
<td>Use when your system is small. i.e., 16k</td>
</tr>
<tr>
<td>HPSP</td>
<td>'150000</td>
<td>Use when your program is so large that it overlays PSD</td>
</tr>
<tr>
<td>PSD</td>
<td>'60000</td>
<td>Normal use</td>
</tr>
<tr>
<td>VPSD</td>
<td>'60000 of segment '4000</td>
<td>Normal use</td>
</tr>
<tr>
<td>VPSD16</td>
<td>'160000</td>
<td>Use when your program is so large that it overlays VPSD</td>
</tr>
</tbody>
</table>

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**

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.

**Usage/Consequence**

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.

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.

Used when runfile exists. When EXECUTE is given, the registers are as SEG initialized them. 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 accessor 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 defined 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)

<table>
<thead>
<tr>
<th>Format</th>
<th>Code</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>:A</td>
<td>Two characters accepted first may not be: &gt;= @ % , .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]</td>
<td>Two characters are printed. An @ is substituted for a nonprinting character</td>
</tr>
<tr>
<td>Binary</td>
<td>:B</td>
<td>Takes a sequence of 16 1's and 0's</td>
<td>Prints a sequence of sixteen 1's and 0's</td>
</tr>
<tr>
<td>Decimal</td>
<td>:D</td>
<td>Accepts up to five decimal (0-9) digits</td>
<td>Prints decimal digits</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>:H</td>
<td>Accepts up to four hexadecimal (0-9, A, B, D, C, E, F) digits</td>
<td>Prints hexadecimal digits</td>
</tr>
<tr>
<td>Octal</td>
<td>:O</td>
<td>Accepts up to six octal (0-7) digits</td>
<td>Prints octal digits</td>
</tr>
<tr>
<td>Symbolic</td>
<td>:S</td>
<td>Symbolic instructions [See below]</td>
<td>Symbolic instructions</td>
</tr>
<tr>
<td>AP *</td>
<td>:P</td>
<td>Symbolic instructions</td>
<td>Prints address pointers</td>
</tr>
<tr>
<td>Long *</td>
<td>:L</td>
<td>Accepts 32 bit octal integers</td>
<td>Prints 32 bit octal integers</td>
</tr>
</tbody>
</table>

*-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:

\[ \text{constant} [\pm \text{constant}] \]

**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.
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>
<thead>
<tr>
<th>Terminator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>Alters contents of current location (if a value is given), moves to current location +1, and prints its contents.</td>
</tr>
<tr>
<td>* or †</td>
<td>Alters contents of current location (if a value is given), moves to current location –1, and prints its contents.</td>
</tr>
<tr>
<td>/ or ?</td>
<td>Exits from access mode. Does not close current location.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>Carry Bit</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>Precision: 0=Single; 1=Double</td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>Not used</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>Not used</td>
</tr>
<tr>
<td>5–6</td>
<td>ADR</td>
<td>Addressing Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00=16S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01=32S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11=32R</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10=64R</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>Not used</td>
</tr>
<tr>
<td>8</td>
<td>–</td>
<td>Not used</td>
</tr>
<tr>
<td>9–16</td>
<td>SHIFT COUNT</td>
<td>Shift count – low order 8 bits of the floating point accumulator exponent register</td>
</tr>
</tbody>
</table>

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 subprocessor 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.
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>
<thead>
<tr>
<th>Terminator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>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.</td>
</tr>
<tr>
<td>/ or ?</td>
<td>Exits from access mode. Does not close current location.</td>
</tr>
<tr>
<td>.n(CR)</td>
<td>Moves to current location +n and prints its contents (n is octal).</td>
</tr>
<tr>
<td>.-n(CR)</td>
<td>Moves to current location -n and prints its contents (n is octal).</td>
</tr>
<tr>
<td>@</td>
<td>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.</td>
</tr>
<tr>
<td>(</td>
<td>Goes to effective address without indirection, but saves current location as return address.</td>
</tr>
<tr>
<td>\</td>
<td>Returns to the return address saved by the last @.</td>
</tr>
<tr>
<td>)</td>
<td>Returns to the return address saved by the last (.</td>
</tr>
<tr>
<td>=</td>
<td>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.</td>
</tr>
<tr>
<td>!</td>
<td>Close a location, setting it to a new value if one was supplied, and return to command mode.</td>
</tr>
</tbody>
</table>
**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 :FFFF
F 1011 1020 :AAAA
```

Example:

```
F 1000 1010 :FFFF
F 1011 1020 :AAAA
D 1000 1020
C 1010 1016 1012
D 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 1000
RE 1000
DE FOO 1000
DE FOO >3
DE FOO :AX
DE FOO :HF
DE FOO :A X
DE FOO :SLT
```

```
DE FOO = OCTAL 1000
SD RELOCATION COUNTER
DE FOO = OCTAL 1003
DE FOO = 'XX'
DE FOO = 5
```

```
DE FOO = OCTAL 1012
```

```
DE BAR FOO
DE FOO (1+3)
DE FOO :HF
DE FOO :AX
DE FOO :A X
DE FOO :SLT
```

**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, optionally, 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 1000 2000
    0 0 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 by pre-filling it with a visual pattern.

Example:

    F 1000 1007 :HFFFF
    D 1000 1007
    1000 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 overridden 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

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th></th>
<th>ADR</th>
<th></th>
<th>SHIFT COUNT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

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
GO
$ LOAD B PROG
LOAD COMPLETE
$ SAVE $PROG
$ MAP LSYM 10
$ QUIT
OK, RESTORE $PROG
OK, PSD
GO
$ 0 LSYM 1 1
$ LS
$ 0 014
```

\[ \text{MA} \]

Types the symbols and their definitions.

\[ \text{MO} \quad \{ \begin{align*}
\text{D16S} \\
\text{D32S} \\
\text{D32R} \\
\text{D64R}
\end{align*} \]

\( \text{D16S} \) means use 16S address mode; \( \text{D32S} \), use 32S address mode; \( \text{D32R} \), use 32R address mode; and \( \text{D64R} \), 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.

\[ \text{MONITOR [start-add] [a-reg] [b-reg] address} \]

Traces the object program from \( \text{start-add} \) (the default is the current location) looking for \( \text{address} \). 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.

\[ \text{NOT-EQUAL block-start block-end n-match [mask]} \]

Searches memory between \( \text{block-start} \) and \( \text{block-end} \) for words not equal to \( \text{n-match} \) under an optional \( \text{16-bit mask} \).

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 -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.)
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>
<thead>
<tr>
<th>Terminator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>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.</td>
</tr>
<tr>
<td>/ or ?</td>
<td>Exits from access mode. Does not close current location.</td>
</tr>
<tr>
<td>.n(CR)</td>
<td>Moves to current location +n and prints its contents (n is octal).</td>
</tr>
<tr>
<td>.−n(CR)</td>
<td>Moves to current location −n and prints its contents (n is octal).</td>
</tr>
<tr>
<td>@</td>
<td>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.</td>
</tr>
<tr>
<td>(</td>
<td>Goes to effective address without indirection, but saves current location as return address.</td>
</tr>
<tr>
<td>\</td>
<td>Returns to the return address saved by the last @.</td>
</tr>
<tr>
<td>)</td>
<td>Returns to the return address saved by the last .</td>
</tr>
<tr>
<td>=</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 procedure 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 :HAAAAA | 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:

DMPFIL 1 2
D 1000 2000
0 0 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 "i" will switch to access mode and display the location referenced by the field address register in ASCII. A "j" 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 "i"

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>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>ADR</th>
<th>SHIFT COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4 5 6 7 8 9</td>
</tr>
<tr>
<td>Bit</td>
<td>Name</td>
<td>Meaning</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>Carry Bit</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>Precision; 0=Single; 1=Double</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>–</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>5–6</td>
<td>ADR</td>
<td>Addressing Mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00=16S</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01=32S</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11=32R</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10=64R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>–</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>9–16</td>
<td>SHIFT COUNT</td>
<td>Shift count—low order 8 bits of the floating point accumulator exponent register.</td>
<td></td>
</tr>
</tbody>
</table>

Table 21-2. Key Values: R and S Modes
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.

<table>
<thead>
<tr>
<th>Label</th>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Number of arguments in current macro call</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Current macro call number</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>A-register</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>B-register</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>X-register</td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td>103</td>
<td>Current character pointer</td>
</tr>
<tr>
<td>CCM</td>
<td>104</td>
<td>Character count max of source line</td>
</tr>
<tr>
<td>105</td>
<td>Used by dynm (must precede cdyn)</td>
<td></td>
</tr>
<tr>
<td>CDYN</td>
<td>106</td>
<td>Current dynamic storage pointer</td>
</tr>
<tr>
<td>MDYN</td>
<td>107</td>
<td>Maximum dynamic stack space used</td>
</tr>
<tr>
<td>108</td>
<td>(spare)</td>
<td></td>
</tr>
<tr>
<td>MCLS</td>
<td>109</td>
<td>Macro list control</td>
</tr>
<tr>
<td>MCRC</td>
<td>110</td>
<td>Current extent of macro call number</td>
</tr>
<tr>
<td>111</td>
<td>(spare)</td>
<td></td>
</tr>
<tr>
<td>MCRN</td>
<td>112</td>
<td>Current macro nest number</td>
</tr>
<tr>
<td>MODE</td>
<td>113</td>
<td>Current mode of assembler</td>
</tr>
<tr>
<td>NCRD</td>
<td>114</td>
<td>Current record number (card number)</td>
</tr>
<tr>
<td>NERR</td>
<td>115</td>
<td>Number of lines in program with errors</td>
</tr>
<tr>
<td>NMFL</td>
<td>116</td>
<td>No-macro-search flag (0=search)</td>
</tr>
<tr>
<td>PASS</td>
<td>117</td>
<td>Pass 1=0, pass 2=1</td>
</tr>
<tr>
<td>RPL</td>
<td>118</td>
<td>Current program counter value</td>
</tr>
<tr>
<td>STAK</td>
<td>119</td>
<td>Current temporary store stack limit</td>
</tr>
<tr>
<td>TC</td>
<td>120</td>
<td>Last character fetched</td>
</tr>
<tr>
<td>TCHB</td>
<td>121</td>
<td>TC held back flag</td>
</tr>
<tr>
<td>TCNT</td>
<td>122</td>
<td>TC repeat count</td>
</tr>
<tr>
<td>IFLG</td>
<td>123</td>
<td>Indirect operator flag (0=indirect)</td>
</tr>
<tr>
<td>DFVL</td>
<td>124</td>
<td>Table search value</td>
</tr>
<tr>
<td>SEG</td>
<td>125</td>
<td>Seg mode flag (0, 1, -1)</td>
</tr>
<tr>
<td>ABM</td>
<td>126</td>
<td>Current abstract machine</td>
</tr>
<tr>
<td>0=S,R and 1=V,l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMB</td>
<td>127</td>
<td>Procedure size max</td>
</tr>
<tr>
<td>LBM</td>
<td>128</td>
<td>Link size max</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Character</th>
<th>Octal</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL-P</td>
<td>'220</td>
<td>is interpreted as a .BREAK.</td>
</tr>
<tr>
<td>.CR.</td>
<td>'215</td>
<td>is interpreted as a newline (.NL.)</td>
</tr>
<tr>
<td>'&quot;'</td>
<td>'242</td>
<td>is interpreted as a character erase</td>
</tr>
<tr>
<td>'?'</td>
<td>'277</td>
<td>is interpreted as line kill</td>
</tr>
<tr>
<td>'/'</td>
<td>'334</td>
<td>is interpreted as a logical tab (Editor)</td>
</tr>
</tbody>
</table>
### Table B-1. ASCII Character Set (Non-Printing)

<table>
<thead>
<tr>
<th>Octal Value</th>
<th>ASCII Char</th>
<th>Comments/Prime Usage</th>
<th>Control Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>NULL</td>
<td>Null character – filler</td>
<td>@</td>
</tr>
<tr>
<td>201</td>
<td>SOH</td>
<td>Start of header (communications)</td>
<td>A</td>
</tr>
<tr>
<td>202</td>
<td>STX</td>
<td>Start of text (communications)</td>
<td>B</td>
</tr>
<tr>
<td>203</td>
<td>ETX</td>
<td>End of text (communications)</td>
<td>C</td>
</tr>
<tr>
<td>204</td>
<td>EOT</td>
<td>End of transmission (communications)</td>
<td>D</td>
</tr>
<tr>
<td>205</td>
<td>ENQ</td>
<td>End of I.D. (communications)</td>
<td>E</td>
</tr>
<tr>
<td>206</td>
<td>ACK</td>
<td>Acknowledge affirmative (communications)</td>
<td>F</td>
</tr>
<tr>
<td>207</td>
<td>BEL</td>
<td>Audible alarm (bell)</td>
<td>G</td>
</tr>
<tr>
<td>210</td>
<td>BS</td>
<td>Back space on position (carriage control)</td>
<td>H</td>
</tr>
<tr>
<td>211</td>
<td>HT</td>
<td>Physical horizontal tab</td>
<td>I</td>
</tr>
<tr>
<td>212</td>
<td>LF</td>
<td>Line feed; ignored as terminal input</td>
<td>J</td>
</tr>
<tr>
<td>213</td>
<td>VT</td>
<td>Physical vertical tab (carriage control)</td>
<td>K</td>
</tr>
<tr>
<td>214</td>
<td>FF</td>
<td>Form feed (carriage control)</td>
<td>L</td>
</tr>
<tr>
<td>215</td>
<td>CR</td>
<td>Carriage return (carriage control)</td>
<td>M</td>
</tr>
<tr>
<td>216</td>
<td>SO</td>
<td>RRS-red ribbon shift</td>
<td>N</td>
</tr>
<tr>
<td>217</td>
<td>SI</td>
<td>BRS-black ribbon shift</td>
<td>O</td>
</tr>
<tr>
<td>220</td>
<td>DLE</td>
<td>RCP-relative copy (2)</td>
<td>P</td>
</tr>
<tr>
<td>221</td>
<td>DC1</td>
<td>RHT-relative horizontal tab (3)</td>
<td>Q</td>
</tr>
<tr>
<td>222</td>
<td>DC2</td>
<td>HLF-half line feed forward (carriage control)</td>
<td>R</td>
</tr>
<tr>
<td>223</td>
<td>DC3</td>
<td>RVT-relative vertical tab (4)</td>
<td>S</td>
</tr>
<tr>
<td>224</td>
<td>DC4</td>
<td>HLR-half line feed reverse (carriage control)</td>
<td>T</td>
</tr>
<tr>
<td>225</td>
<td>NAK</td>
<td>Negative acknowledgement (communications)</td>
<td>U</td>
</tr>
<tr>
<td>226</td>
<td>SYN</td>
<td>Synchronocity (communications)</td>
<td>V</td>
</tr>
<tr>
<td>227</td>
<td>ETB</td>
<td>End of transmission block (communications)</td>
<td>W</td>
</tr>
<tr>
<td>230</td>
<td>CAN</td>
<td>Cancel</td>
<td>X</td>
</tr>
<tr>
<td>231</td>
<td>EM</td>
<td>End of Medium</td>
<td>Y</td>
</tr>
<tr>
<td>232</td>
<td>SUB</td>
<td>Substitute</td>
<td>Z</td>
</tr>
<tr>
<td>233</td>
<td>ESC</td>
<td>Escape</td>
<td>]</td>
</tr>
<tr>
<td>234</td>
<td>FS</td>
<td>File separator</td>
<td>\</td>
</tr>
<tr>
<td>235</td>
<td>GS</td>
<td>Group separator</td>
<td>:</td>
</tr>
<tr>
<td>236</td>
<td>RS</td>
<td>Record separator</td>
<td>.</td>
</tr>
<tr>
<td>237</td>
<td>US</td>
<td>Unit separator</td>
<td>_</td>
</tr>
</tbody>
</table>

**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>
<thead>
<tr>
<th>Octal Value</th>
<th>ASCII Character</th>
<th>Octal Value</th>
<th>ASCII Character</th>
<th>Octal Value</th>
<th>ASCII Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>.SP. (1)</td>
<td>300</td>
<td>@</td>
<td>340</td>
<td>'9'</td>
</tr>
<tr>
<td>241</td>
<td>!</td>
<td>301</td>
<td>A</td>
<td>341</td>
<td>a</td>
</tr>
<tr>
<td>242</td>
<td>&quot; (2)</td>
<td>302</td>
<td>B</td>
<td>342</td>
<td>b</td>
</tr>
<tr>
<td>243</td>
<td># (3)</td>
<td>303</td>
<td>C</td>
<td>343</td>
<td>c</td>
</tr>
<tr>
<td>244</td>
<td>$</td>
<td>304</td>
<td>D</td>
<td>344</td>
<td>d</td>
</tr>
<tr>
<td>245</td>
<td>%</td>
<td>305</td>
<td>E</td>
<td>345</td>
<td>e</td>
</tr>
<tr>
<td>246</td>
<td>&amp;</td>
<td>306</td>
<td>F</td>
<td>346</td>
<td>f</td>
</tr>
<tr>
<td>247</td>
<td>' (4)</td>
<td>307</td>
<td>G</td>
<td>347</td>
<td>g</td>
</tr>
<tr>
<td>250</td>
<td>(</td>
<td>310</td>
<td>H</td>
<td>350</td>
<td>h</td>
</tr>
<tr>
<td>251</td>
<td>)</td>
<td>311</td>
<td>I</td>
<td>351</td>
<td>i</td>
</tr>
<tr>
<td>252</td>
<td>*</td>
<td>312</td>
<td>J</td>
<td>352</td>
<td>j</td>
</tr>
<tr>
<td>253</td>
<td>+</td>
<td>312</td>
<td>K</td>
<td>353</td>
<td>k</td>
</tr>
<tr>
<td>254</td>
<td>, (5)</td>
<td>314</td>
<td>L</td>
<td>354</td>
<td>l</td>
</tr>
<tr>
<td>255</td>
<td>-</td>
<td>315</td>
<td>M</td>
<td>355</td>
<td>m</td>
</tr>
<tr>
<td>256</td>
<td>.</td>
<td>316</td>
<td>N</td>
<td>356</td>
<td>n</td>
</tr>
<tr>
<td>257</td>
<td>/</td>
<td>317</td>
<td>O</td>
<td>357</td>
<td>o</td>
</tr>
<tr>
<td>260</td>
<td>0</td>
<td>320</td>
<td>P</td>
<td>360</td>
<td>p</td>
</tr>
<tr>
<td>261</td>
<td>1</td>
<td>321</td>
<td>Q</td>
<td>361</td>
<td>q</td>
</tr>
<tr>
<td>262</td>
<td>2</td>
<td>322</td>
<td>R</td>
<td>362</td>
<td>r</td>
</tr>
<tr>
<td>263</td>
<td>3</td>
<td>323</td>
<td>S</td>
<td>363</td>
<td>s</td>
</tr>
<tr>
<td>264</td>
<td>4</td>
<td>324</td>
<td>T</td>
<td>364</td>
<td>t</td>
</tr>
<tr>
<td>265</td>
<td>5</td>
<td>325</td>
<td>U</td>
<td>365</td>
<td>u</td>
</tr>
<tr>
<td>266</td>
<td>6</td>
<td>326</td>
<td>V</td>
<td>366</td>
<td>v</td>
</tr>
<tr>
<td>267</td>
<td>7</td>
<td>327</td>
<td>W</td>
<td>367</td>
<td>w</td>
</tr>
<tr>
<td>270</td>
<td>8</td>
<td>330</td>
<td>X</td>
<td>370</td>
<td>x</td>
</tr>
<tr>
<td>271</td>
<td>9</td>
<td>331</td>
<td>Y</td>
<td>371</td>
<td>y</td>
</tr>
<tr>
<td>272</td>
<td>:</td>
<td>332</td>
<td>Z</td>
<td>372</td>
<td>z</td>
</tr>
<tr>
<td>273</td>
<td>;</td>
<td>333</td>
<td></td>
<td></td>
<td>373</td>
</tr>
<tr>
<td>274</td>
<td>&lt;</td>
<td>334</td>
<td>\</td>
<td>374</td>
<td>\</td>
</tr>
<tr>
<td>275</td>
<td>=</td>
<td>335</td>
<td></td>
<td></td>
<td>375</td>
</tr>
<tr>
<td>276</td>
<td>&gt;</td>
<td>336</td>
<td>~ (7)</td>
<td>376</td>
<td>~ (10)</td>
</tr>
<tr>
<td>277</td>
<td>? (6)</td>
<td>337</td>
<td>-- (8)</td>
<td>377</td>
<td>DEL (11)</td>
</tr>
</tbody>
</table>

**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 hit ('200) has been added for Prime usage.
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
ERROR MESSAGES

L00: IMPROPER LABEL (CONSTANT OR TERMINATOR IN LABEL FIELD)
L01: EXTERNAL VARIABLE DISCLAIMED 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 DISCLAIMED 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 DESCENSORIZATION ('LOAD' MODE)
S01: INDIRECT_DAC DISCLAIMED IN C64R MODE
S02: 64V INSTRUCTION DISCLAIMER 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
V00: 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/DFTV ARGUMENT
V17: CLOSE PARENTHESIS MISSING ON DFTB/DFTV 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
ERROR MESSAGES

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: 321 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. SAvе the runfile and give the EXecute command.

CMS
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.

xxxxxxx 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 CCommon 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.
C  ERROR MESSAGES

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

Attempt to perform operations in segments to which user has no right.

****AD

Overflow or underflow in double-precision addition/subtraction [A$66, SS$66].

ALL REMOTE UNITS IN USE

Attempt made to assign a remote unit when none are available. (Network error) [E$FUIU]

**** ALOG/ALOGIN 10 - ARGUMENT <= 0

Argument not greater than zero used in logarithm [ALOGIN, ALOG 10] function.

filename ALREADY EXISTS

Attempt to create a file or UFD with the name of one already existing. [CZ]

ALREADY EXISTS

Attempt made to create, in the UFD, a sub-UFD with the same name as one already existing. [CREASS] [E$EXST]

****AT

Both arguments are zero in the ATAN2 function.

**** ATAN2 - BOTH ARGUMENTS = 0

Both arguments are zero in the ATAN2 function.

**** ATTDEV - BAD UNIT

Incorrect logical device unit number in the ATTDEV subroutine call.

BAD CALL TO SEARCH

Error in calling the SEARCH subroutine, e.g., incorrect parameter. [SA]

BAD DAM FILE

The DAM file specified has been corrupted - either by the programmer or by a system problem. [SS]

BAD DAM FILE

The DAM file specified has been corrupted - either by the programmer or by a system problem. [PRWFS$, SRCHS$]. [ESBDAM]
BAD FAM SVC
System problem; will not be seen by applications programmer. [ESBFSV]

BAD KEY
Incorrect key value specified in subroutine argument. [ATCH$$, RDEN$$, SATR$$, SRCH$$, SGDR$$] [ESBKEY]

BAD PARAMETER
Incorrect parameter value in subroutine call. [SA]

BAD PASSWORD
Incorrect password specified in ATTACH subroutine. Returns to PRIMOS level attached to no UFD. [AN]

BAD PASSWORD
Incorrect password specified in ATCH$$ subroutine. Returns to PRIMOS level attached to no UFD. [ATCH$$] [ESBPAS]

Note
To protect UFD privacy the system does not allow the user to trap BAD PASSWORD errors.

BAD RTNREC
System error.

BAD SEGDIR UNIT
Error generated in accessing segment directory. i.e., PRIMOS file unit specified is not a segment directory. [SRCH$$] [ESBSUN]

BAD SEGMENT NUMBER
Attempt made to access segment number outside valid range. [ESBSGN]

BAD SVC
Bad supervisor call. In FORTRAN usually caused by program writing over itself.

BAD TRUNCATE OF SEGDIR
Error encountered in truncating segment directory. [SGDR$$] [ESBTRN]

BAD UFD
UFD has become corrupted. [ATCH$$, CREA$$, GPASS$$, RDEN$$, SATR$$, SRCH$$] [ESBUFD]. Calls to RDEN$$ return this as a trappable error; other commands return to the PRIMOS command level.

BAD UNIT NUMBER
PRIMOS file unit number specified is invalid - outside legal range. [PRWF$$, RDEN$$, SRCH$$, SGDR$$]. [ESBUNT]

BEGINNING OF FILE
Attempt was made to access locations before the beginning of the file. [PRWF$$, RDEN$$, SGDR$$] [ESBOF]
**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$$] [ESBFTS]

**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. [ESDVJU]

**DEVICE NOT ASSIGNED**

File System

Attempt was made to perform I/O operations on a device before assigning that device. [ESNASS]

**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. [ESDNS]

**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$$] [ESDNTE]

**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$$, SGDR$$]. [ESDKFL]

**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$$, SGDR$$. [ESDISK]
**DISK WRITE-PROTECTED**  
File System  
An attempt has been made to write to a disk which is WRITE-protected. [ESWTPE]

**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 \( \leq 0 \)  
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. [PRWFSS, RDENS$, SGDRSS] [E$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. [ESFABT]
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$$] [ESFDEL]

FILE OPEN ON DELETE                   File System
    Attempt made to delete a file which is open. [SRCH$$] [ESFDEL]

FILE TOO BIG                          File System
    Attempt made to increase size of segment directory beyond size limit. (SGDRSS)
    [ESFITB]

****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

   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. [CREASS, SRCHSS] [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$$, GPASS$$, SATR$$, SRCH$$, SGDR$$) [ESNRIT]

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. [ESROOM]

NO TIME  
File System  
Clock not started. System error. [ESNTIM]

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$$, CREA$$, GPASS$$, SATR$$, SRCH$$). [ESNATT] 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 UUI, 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$$] [ESNTSD]

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$$, GPASS$$, SRCH$$). [ESNTUD]

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$$, GPASS$$, SATR$$, SRCH$$) [ESFNTF]
C  ERROR MESSAGES

filename NOT FOUND IN SEGDIR File System
Filenanme 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$$, CREA$$, GPASS$$, PRWF$$, 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
  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$$] [ES$SUNO]

SEM OVERFLOW
  File System
  System error [ES$EMO]

**** 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. [ESTMUL]

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. [CREAS$, SRCH$$] [ES$DFDL]
C  ERROR MESSAGES

UF D OVR FL OW
   No more room in UF D.

UNIT IN USE
   Attempt to open file on PRIMOS file unit already in use. [SI].

UNIT IN USE
   Attempt to open file on PRIMOS file unit already in use. [SRCH$$.] [ESUIUS]

UNIT NOT OPEN
   Attempt to perform operations with a file unit number on which no file has been
   opened. [PD, SD]

UNIT NOT OPEN
   Attempt to perform operations with a file unit number on which no file has been
   opened. [PRWF$$, RDEN$$, SRCH$$, SGDR$$]. [ESUNOP]

UNIT OPEN ON DELETE
   Attempt to delete file without having first closed it. [SD]

****WN n
   Device error or end-of-file in WRITE statement on FORTRAN logical unit n.

****XX
   Integer argument >32767.
Index

x

Usage in PMA 14-5

C-bit to A, ACA, (SRV) 11-22

APPLIB, system library 8-1

fullword, A, (I) 12-9

Application library 8-1

halfword, AH, (I) 12-10

ARFA (I) 12-3

L bit to L, ADLL, (V) 11-23

ARGT (I) 12-20

L to field address, ALFA, (V) 11-15

ARGT (V) 11-43

link to register, ADLL, (I) 12-10

Argument

long, ADL, (V) 11-22

identifiers, macro 17-2, 17-3, 17-4

one to A, ALA (SRV) 11-22

pointer pseudo-operation, AP 16-7

register to field address

references, macro 17-2, 17-4

register, ARFA, (I) 12-5

substitution, macro 17-3

to bottom of queue, ABQ, (I) 12-21

transfer template, AP 9-7

to bottom of queue, ABQ, (V) 11-49

transfer, ARGT, (I) 12-20

to top of queue, ATQ, (I) 12-21

transfer, ARGT, (V) 11-43

to top of queue, ATQ, (V) 11-49

value expressions, macro 17-3

two to A, A2A (SRV) 11-22

values in parentheses,

ADD (SRV) 11-22

macro 17-3

Adding module to a SEG runfile 5-7

values, macro 17-1, 17-3, 17-4

Address

Address

definition pseudo-operations,

Argus, macro 17-2

AD 18-1

Arithmetic instruction register

formation 18-4

usage (I-mode only) 11-10

special case selection 9-18

Arithmetic operators 15-8

template 4-4, 10-3

character set B-I

space, virtual 4-5

character string 15-5

truncation (SR) 10-2

constants 18-3

Assembler

Assembly control pseudo-opera-

tions, AC 16-1

attribute references 17-2

Asterisk (current location), PMA 14-6

attributes 15-11, A-1

Asterisk, double, PMA 14-6

top and bottom messages A-1

ATQ (I) 12-21

error messages C-1

ATQ (V) 11-49

formats (I) 14-7

Attributes, assembler 15-11, 17-2, A-1

messages 3-2

Assembly control pseudo-

operations, AC 16-1

Assembly, pseudo-operation 16-8

Assembly, register relative, re-

terence instruction formats 10-6

Base area problems, how to re-

direct 4-5, 4-6, 5-8

Base area, R-mode load map
description 4-4

Base areas, R-mode load map

description 4-4

Base areas, SEG load map
template 5-6

Base register relative, memory refer-

cence instruction formats 10-6

Base registers 9-12

Base registers, PMA formats 14-6

BCEQ (I) 12-3

Base registers, PMA formats

BCEQ (V) 11-2

14-6

BCEQ (V) 11-2

BCGE (I) 12-3

BCGE (V) 11-2

BCGT (I) 12-3

BCGT (V) 11-2

BCGT (V) 11-2

BCI pseudo operation 16-10

BCLR (I) 12-3

BCLR (V) 11-2

BCLT (I) 12-3

BCLT (V) 11-2

BCNE (V) 11-2

1 March 1979 X-1

FDR 3059
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) 12-8
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
BRDT; (I) 12-3
BRD2; (I) 12-3
BRD4; (I) 12-3
BREQ; (I) 12-2
BRGE; (I) 12-2
BRGT; (I) 12-2
BR1; (I) 12-3
BR2; (I) 12-3
BR4; (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 (SRV) 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
CIT (I) 12-3
CIT (V) 11-4
CH (I) 12-10
Change signs, CHS, (I) 12-10
Change sign, CHS, (SRV) 11-23
CHAR (I) 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
C, CRE, (V) 11-8
high byte 1 left, CRBI, (I) 12-4
high byte 2 right, CRBR, (I) 12-4
L and E, CRL, (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
CMOA (SRV) 11-32
CMDCN60, 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, CMDCN60 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
COMPUTPUT 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
Composed COTO, CFT, (I) 12-3
Composed COTO, CFT, (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 16-5, 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, XBDTD, [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.
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 (CX) 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) 11-12
Decimal to binary conversion, XCTB, (V) 11-12
Declare stack relative pseudo-operation, DYNM 16-20
Decrement and replace X, DRX,
(SRV) 11-93
half register by 1, DHl,
(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
DF (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
dFSF (RV) 11-18
DFSTB pseudo-operation 16-8
DFSTT 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 5-8
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
DML (I) 11-11
dMH (I) 12-11
Double add, DAD, (SR) 11-24
Double asterisk (initial zero)
PM (I) 14-6
Double floating
add, DFM, (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, CFS, (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, CFA, (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, DFS, (RV) 11-17
floating point register 9-12
floating store, DFB, (RV) 11-18
floating subtract, DFBF,
(RV) 11-18
Double subtract, DS, (RV) 11-24
DR1 (I) 12-11
complement, FCM, [I] 12-7
divide, FD, [I] 12-7
dividide, FDM, (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, FPNT (I) 12-6
t 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, (RV) 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, FSEZ, (RV) 11-21
store, FST, [I] 12-8
store, FST, (RV) 11-21
subtract, PS, (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
FLTHT (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 (1-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
FSD (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
FTPUMB library use with SEG 5-3
FTPUMB, 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, (RV) 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, (RV) 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
BCE 12-3
BCCT 12-3
BCL 12-3
BCLE 12-3
BCN 12-3
BCR 12-3
BCS 12-3
BFEQ 12-2
BGE 12-2
BFGT 12-2
BFLE 12-2
BFT 12-2
BFNE 12-2
BD1 12-3
BD2 12-3
BD4 12-3
BGE 12-2
BGT 12-2
BHL 12-3
BH2 12-3
BHI 12-3
BH1 12-3
BHLE 12-2
BHLT 12-2
BFNEB 12-2
BL 12-3
BLS 12-3
BMGE 12-3
BMGT 12-3
BMLE 12-3
BMLT 12-3
BMNE 12-3
BMNEQ 12-3
BRBR 12-2
BRBS 12-2
BRDI 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
BNE 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
DF 12-8
DFC 12-8
DFCM 12-9
DFD 12-9
DFF 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
DR 12-11
DR1 12-11
D16S 12-1
E321 12-1
E32R 12-1
E3S 12-1
E64R 12-1
E4V 12-1
E4A 12-6
EALB 12-19
EAR 12-19
EAXB 12-20
EIO 12-14
EMCM 12-14
ENB 12-14
ESBM 12-14
EVIM 12-14
FA 12-6
FC 12-7
FCD 12-7
FD 12-7
FL 12-7
FLT 12-7
FMT 12-7
FM 12-7
FRN 12-7
FS 12-8
FST 12-8
<table>
<thead>
<tr>
<th>Index</th>
<th>1 March 1979</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLT 12-17</td>
<td>MDWC 12-14</td>
</tr>
<tr>
<td>I 12-17</td>
<td>MH 12-12</td>
</tr>
<tr>
<td>ICBL 12-17</td>
<td>N 12-15</td>
</tr>
<tr>
<td>ICBL 12-18</td>
<td>NFYB 12-20</td>
</tr>
<tr>
<td>IBCR 12-17</td>
<td>NFYE 12-20</td>
</tr>
<tr>
<td>ICHR 12-18</td>
<td>NH 12-15</td>
</tr>
<tr>
<td>IH 12-18</td>
<td>NOP 12-17</td>
</tr>
<tr>
<td>IH2 12-11</td>
<td>O 12-15</td>
</tr>
<tr>
<td>IH1 12-11</td>
<td>OH 12-15</td>
</tr>
<tr>
<td>IM 12-12</td>
<td>OTK 12-14</td>
</tr>
<tr>
<td>IMH 12-12</td>
<td>PCL 12-20</td>
</tr>
<tr>
<td>INBN 12-20</td>
<td>PID 12-12</td>
</tr>
<tr>
<td>INEC 12-20</td>
<td>PIDH 12-12</td>
</tr>
<tr>
<td>INEN 12-20</td>
<td>PIM 12-13</td>
</tr>
<tr>
<td>INH 12-14</td>
<td>PIMH 12-13</td>
</tr>
<tr>
<td>INK 12-14</td>
<td>PRTN 12-20</td>
</tr>
<tr>
<td>INT 12-8</td>
<td>PTLB 12-17</td>
</tr>
<tr>
<td>INTH 12-8</td>
<td>RBQ 12-21</td>
</tr>
<tr>
<td>IR1 12-12</td>
<td>RGB 12-15</td>
</tr>
<tr>
<td>IR2 12-12</td>
<td>RMCE 12-14</td>
</tr>
<tr>
<td>IRB 12-18</td>
<td>ROT 12-21</td>
</tr>
<tr>
<td>IRH 12-18</td>
<td>RRST 12-17</td>
</tr>
<tr>
<td>RTC 12-14</td>
<td>RSAR 12-17</td>
</tr>
<tr>
<td>RTX 12-14</td>
<td>RTQ 12-21</td>
</tr>
<tr>
<td>TLBL 12-17</td>
<td>S 12-13</td>
</tr>
<tr>
<td>JMF 12-20</td>
<td>SCB 12-15</td>
</tr>
<tr>
<td>JSR 12-20</td>
<td>SH 12-13</td>
</tr>
<tr>
<td>JSXB 12-20</td>
<td>SHA 12-22</td>
</tr>
<tr>
<td>L 12-18</td>
<td>SHL 12-23</td>
</tr>
<tr>
<td>LCEQ 12-16</td>
<td>SHL1 12-22</td>
</tr>
<tr>
<td>LCGE 12-16</td>
<td>SHR1 12-23</td>
</tr>
<tr>
<td>LGT 12-16</td>
<td>SL1 12-22</td>
</tr>
<tr>
<td>LGT 12-16</td>
<td>SL1 12-22</td>
</tr>
<tr>
<td>LDAR 12-18</td>
<td>SQR 12-22</td>
</tr>
<tr>
<td>LDC 12-4</td>
<td>SRL 12-22</td>
</tr>
<tr>
<td>LEG 12-16</td>
<td>SSM 12-13</td>
</tr>
<tr>
<td>LF 12-17</td>
<td>ST 12-19</td>
</tr>
<tr>
<td>LFQ 12-16</td>
<td>SSP 12-13</td>
</tr>
<tr>
<td>LFGE 12-16</td>
<td>STAR 12-19</td>
</tr>
<tr>
<td>LGT 12-16</td>
<td>STC 12-4</td>
</tr>
<tr>
<td>LFOE 12-16</td>
<td>STCD 12-19</td>
</tr>
<tr>
<td>LFGE 12-16</td>
<td>STCH 12-19</td>
</tr>
<tr>
<td>LFQ 12-16</td>
<td>STECH 12-20</td>
</tr>
<tr>
<td>LGT 12-16</td>
<td>STFA 12-6</td>
</tr>
<tr>
<td>LGT 12-16</td>
<td>STH 12-19</td>
</tr>
<tr>
<td>LGT 12-16</td>
<td>STPM 12-17</td>
</tr>
<tr>
<td>LH 12-18</td>
<td>SVC 12-20</td>
</tr>
<tr>
<td>LHEQ 12-16</td>
<td>TC 12-13</td>
</tr>
<tr>
<td>LGHE 12-16</td>
<td>TCH 12-13</td>
</tr>
<tr>
<td>LHGT 12-16</td>
<td>TFLR 12-5</td>
</tr>
<tr>
<td>LHL1 12-18</td>
<td>TM 12-14</td>
</tr>
<tr>
<td>LHL2 12-19</td>
<td>TMH 12-14</td>
</tr>
<tr>
<td>LHL1 12-16</td>
<td>TSTQ 12-21</td>
</tr>
<tr>
<td>LHL2 12-16</td>
<td>VIRY 12-14</td>
</tr>
<tr>
<td>LHL1 12-16</td>
<td>WAIT 12-20</td>
</tr>
<tr>
<td>LHL2 12-16</td>
<td>WCS 12-17</td>
</tr>
<tr>
<td>LHLT 12-16</td>
<td>X 12-15</td>
</tr>
<tr>
<td>LHNE 12-18</td>
<td>XAD 12-5</td>
</tr>
<tr>
<td>LHNE 12-18</td>
<td>XBTB 12-5</td>
</tr>
<tr>
<td>LHT 12-17</td>
<td>XCM 12-5</td>
</tr>
<tr>
<td>LIOT 12-17</td>
<td>XDBT 12-5</td>
</tr>
<tr>
<td>LLE 12-16</td>
<td>XED 12-5</td>
</tr>
<tr>
<td>LLI 12-16</td>
<td>XH 12-16</td>
</tr>
<tr>
<td>LLI 12-16</td>
<td>XMV 12-5</td>
</tr>
<tr>
<td>LMT 12-16</td>
<td>XVRY 12-14</td>
</tr>
<tr>
<td>LMCM 12-14</td>
<td>XVRY 12-17</td>
</tr>
<tr>
<td>MT 12-14</td>
<td>ZCM 12-4</td>
</tr>
<tr>
<td>M2 12-13</td>
<td>ZDV 12-5</td>
</tr>
<tr>
<td>MDEI 12-14</td>
<td>ZED 12-4</td>
</tr>
<tr>
<td>MDII 12-14</td>
<td>ZEH 12-4</td>
</tr>
<tr>
<td>MDII 12-14</td>
<td>ZM 12-5</td>
</tr>
<tr>
<td>MDIW 12-14</td>
<td>ZMH 12-5</td>
</tr>
<tr>
<td>MDJW 12-14</td>
<td>ZMV 12-4</td>
</tr>
<tr>
<td>MDRS 12-14</td>
<td>ZMVD 12-4</td>
</tr>
<tr>
<td>MDRS 12-14</td>
<td>ZTRN 12-4</td>
</tr>
</tbody>
</table>

**l-mode**

- Instruction formats 9-16
- Purpose 9-16

**l/O** 9-14

- input/output (I) 12-14
- input/output (V) 11-29

**lAB** (SRV) 11-40

**lCA** (SRV) 11-40

**lCB** (I) 12-17

**lCB** (I) 12-18

**lCHR** (I) 12-17

**lCHR** (I) 12-18

**lGL** (SRV) 11-40

**lICR** (SRV) 11-40

IF pseudo-operation 16-9

- If (SRV) 11-34

**lFTNL** library, use with SEG 5-2

**lFTNL** system library 8-1

**lFTP** pseudo-operation 16-9

**lFV** pseudo-operation 16-9

**lFVT** pseudo-operation 16-9

**lFX** pseudo-operation 16-9

**lIH** halfword 12-18

IH (I) 12-11

**lIH2** (I) 12-11

**lILE** (V) 11-40

**lIM** (I) 12-12

**lIMA** (SRV) 11-40

Images saved by load 6-2

**lIMH** (I) 12-12

**Immediate**

- type 1, l-mode instruction format 9-18
- type 2, l-mode instruction format 9-18
- type 3, l-mode instruction format 9-18

**lINA** (SR) 11-30

**lINB** (I) 12-20

**lINB** (V) 11-49

**lINB** (V) 11-49

**lINB** (I) 12-20

**lInclusive OR, or, (V)** 11-33

Increment and replace X, IRX.

(RSRV) 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,

BHIL (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, IRX,

(SRV) 11-53

register by 1 and branch, BRIL,

(I) 12-3

register by 1, IR1, (I) 12-12

register by 2 and branch, BRIL2,

(I) 12-3

1 March 1979  X-7  FDR 3059
register by 2, IR2, (I) 12-12
register by 4 and branch, BR4, 12-3
X and branch if not zero, BX, 11-4
Y and branch if not zero, BY, 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,
L, (I) 12-17
register and memory, halfword,
IH, (I) 12-18
register bytes, IRB, (I) 12-18
register halfwords and clear
eight 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
INBN, (I) 12-20
INBN, (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 STL8 entry, ITLB,
(I) 12-17
Invalidate STL8 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
IRB (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
JSX (I) 12-20
JSX (V) 11-47
JSY (V) 11-47
Jump and
decrement X, JX, (R) 11-45
increment X, JX, (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 13-8
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
LCEQ (V) 11-33
LCEG (I) 12-16
LGST (I) 12-16
LCLE (V) 11-33
LCL (I) 12-16
LCL (V) 11-33
LCNE (V) 11-33
LGN (I) 12-16
LDA (SRV) 11-40
LDAR (I) 12-18
Index X

LDC (l) 12-4
LDC (V) 11-5
LDO (V) 11-40
LDMR (V) 11-41
LDX (SRV) 11-41
LDY (V) 11-41
Leave machine check mode, LMCI (I) 12-14
Leave machine check mode, LMCM (I) 11-28
Leave paging mode and jump (Prime 300), LPMI (I) 11-36
LEQ (V) 11-33
LEQ (I) 12-16
LF (I) 12-17
LFQ (I) 12-16
LFEQ (I) 12-16
LFGT (I) 12-16
LFLG (I) 12-16
LFLF (I) 12-6
LFLI (V) 11-15
LFLT (I) 12-16
LFNE (I) 12-16
LGE (V) 11-33
LGE (I) 12-16
LGTE (I) 12-16
LGTV (I) 11-33
LGT (I) 12-16
LHI (I) 12-18
LHEQ (I) 12-16
LHE (I) 12-16
LHEQ (I) 12-16
LHLE (I) 12-19
LHLS (I) 12-18
LHPE (I) 12-16
LHPL (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
ULI 4-4
Line format, PMA 14-3
Line terminator 18-6
Lines, PMA 14-1
Link frame description in load map 3-6
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, LCI 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 I 14-9
LLE (V) 11-33
LLE (I) 12-16
LLEQ (V) 11-33
LLGE (V) 11-33
LLGT (V) 11-33
LLL (SRV) 11-31
LLLE (V) 11-33
LLLT (V) 11-33
LLNLN (V) 11-33
LLR (SRV) 11-52
LLS (SRV) 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, Rmode 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, LHLI (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) 11-40
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, LPFW, (I) 12-17
program status word, LPFW, (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 pseudooperation, DAC 16-7
Local labels within macros 17-2
Location pointer 16-6
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, LTSTTS, (I) 12-16
test and set, LTSTTS, (V) 11-33
Long
left logical, LLL, (SRV) 11-31
left rotate, LLR, (SRV) 11-52
left shift, LS, (SR) 11-52
left shift, LS, (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
LPM (I) 11-36
LPMX (SR) 11-36
LPFW (I) 12-17
LPFW (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
LTSTTS (I) 12-16
LTSTTS (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

1 March 1979

X-9

FDR 3059
X  Index

MATHLIB, 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)  12-14
  read syndrome bits, MDRS, (V)  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
  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
Nesting calls  17-5
Nesting macros  17-5
NFE (I)  12-20
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  4-5
  Object file (SEQ)
    reorganization  5-8
  Object files  3-1
  Object files in SEG  5-8
  OCP (SR)  11-30
  OCT pseudo-operation  16-11
  Octal constants  15-2
  OH (I)  12-15
  One word memory reference, indirect word  9-3
  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 (V)  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
  PCTL (I)  12-19
  PCTL (V)  11-43
  PCVH pseudo-operation  16-12
  Percent sign, PMA  14-5
  PFTNLB library, use with
Index X

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
point sign 14-5
stack 14-6
stack 14-8
statements 14-1, 14-2
symbolic addresses 14-5
symbols 14-4
types 14-4
PMA V or I mode code, how to write 14-8
Position
after multiply, PIM, [I] 12-13
following multiply integer, 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
for integer divide, PID, [V] 11-26
for integer divide, PID, [V] 11-26
for integer divide-long, PIDL, [V] 11-26
half register after multiply, PIMH, [I] 12-13
half register for integer divide, PIMA, [I] 12-12
Pound sign (#) assembler
notation 14-5, 15-11
Powers of 10 [E] 15-4
PRCSEX, 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 2-2
PROC pseudo-operation 16-6
Procedure
call, PCL, [I] 12-20
call, PCL, [V] 11-47
frame description in load
maps 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
maps 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
DFB 16-8
DFVT 16-8
DUII 16-15
DYNM 16-21
DYNF 16-19
DYNF 16-19
EJECT 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
IP 16-7
IPFT 16-9
IPVT 16-9
IPVT 16-9
IPVT 16-9
IPVT 16-9
IPVT 16-9
IPVT 16-9
LINK 16-5
LIR 16-15
LIST 16-11
LSMD 16-11
LSTM 16-11
MAC 16-16
N64R 16-15
NLSM 18-12
NLSI 16-12
OCF 16-11
ORG 16-6
PCVH 16-12
PROC 16-6
REL 16-6
RLT 16-12
SAY 16-10
SCT 16-17
SCT 16-17
SCT 16-17
SDM 16-15
SEG 64V 16-6
SEGR, 321 16-6
SET 16-21
SETB 16-15
SUBR 16-20
VFD 16-11
XAC 16-7
XSET 16-21
PTLB [I] 12-17

1 March 1979

FDR 3059
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, (SRV) 11-38
Resultant mode 15-9
Return from R-mode recursive procedure, RTN, (SRV) 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 (SRV) 11-38
RSAV (I) 12-17
RSAV (SRV) 11-38
RTN (SRV) 11-48
RTQ (I) 12-21
RTQ (SRV) 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
SGTL 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
loads 5-7
VPSD 18-3
SEG level commands 5-9
SEG pseudo-operation 16-6
Segment directory subfields 3-7
Segment usage by SEG 5-7
Segmentation 10-1
Segmented programs, loading 5-1
Segmented runfiles 5-1, 5-2, 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, SHL2, (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 13-8
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, SRR, (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 15-3
Single precision integer value 15-5
Skip conditional, SKIP (V) 11-53
Skip group, Skip, (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] (I) 11-55
SML, (SRV) 11-55
condition A nonzero, SNZ, (SRV) 11-55
condition A plus [a] (I) 11-55
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 lab nonzero, [a] (I) 11-55
SLN, (SRV) 11-55
condition lab zero, [a] (I) 11-55
SLZ, (SRV) 11-55
condition sense switch 1 reset, SRL, (SRV) 11-55
condition sense switch 1 set, SSI, (SRV) 11-55
condition sense switch 2 reset, SRZ, (SRV) 11-55
condition sense switch 2 set, SSZ, (SRV) 11-55
condition sense switch 3 reset, SRR, (SRV) 11-55
condition sense switch 3 set, SSZ, (SRV) 11-55
condition sense switch 4 reset, SR4, (SRV) 11-55
condition sense switch 4 set, SS4, (SRV) 11-55
condition set C, SCC, (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
SoR 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
SR3 (SRV) 11-55
SR4 (SRV) 11-55
SRC 11-55
SRL (SRV) 11-55
SRTLIB, system library 8-1
SST1 (SRV) 11-55
SST2 (SRV) 11-55
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, LFLE, (V) 11-34
floating register less than zero and set R, LFLE, (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, LH, (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, LLLT, (V) 11-33
L register less than or equal to zero and set A, LLEQ, (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 not equal to zero and set R, LLT, (I) 12-16
register less than zero and set R, LLLT, (I) 12-16
Text conventions 2-1
TFFL (V) 11-15
TFLR (I) 12-5
Three word memory reference, indirect pointer 9-3
TWA (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) 11-45
field length register to L, TFFL, (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, ZTR, (I) 12-4
Triple asterisk (dummy instruction), PMA 14-5
TSTQ (I) 12-21
TSTQ (V) 11-50
Two word memory reference, V-mode 10-6
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
U11 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
AZA 11-22
ABQ 11-49
ACA 11-22
ADD 11-22
ADL 11-22
ADL 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

BGGE 11-2
BGT 11-2
BGLE 11-2
BGNE 11-2
BCR 11-3
BGS 11-3
BDX 11-3
BDB 11-4
BEQ 11-4
BFEQ 11-4
BFGE 11-4
BFGT 11-4
BFL 11-4
BFLE 11-4
BFPL 11-4
BFNE 11-4
BG 11-4
BGT 11-4
BXX 11-4
BIY 11-4
BLEQ 11-4
BLGE 11-4
BLGT 11-4
BLL 11-4
BLN 11-4
BLE 11-4
BLR 11-3
BLS 11-3
BLT 11-4
BMEQ 11-3
BN 11-3
BM 11-3
BMLT 11-3
BMN 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
CRLE 11-8
CSA 11-24
CXCS 11-34
DFAD 11-17
DFCM 11-17
DFGS 11-17
DFDV 11-17
DFLD 11-18
DFLX 11-18
DFMP 11-18
DFSB 11-18
DFST 11-18
DIV 11-24
DRLX 11-53
DVL 11-25
E18S 11-1
E32I 11-1
E32R 11-1
E32S 11-1
E64R 11-1
E64V 11-1
E6AF 11-15

<table>
<thead>
<tr>
<th>X</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAL 11-44</td>
<td>LFNE 11-34</td>
</tr>
<tr>
<td>EALB 11-45</td>
<td>LGE 11-33</td>
</tr>
<tr>
<td>EA XB 11-45</td>
<td>LGT 11-33</td>
</tr>
<tr>
<td>EJ O 11-29</td>
<td>LIOT 11-35</td>
</tr>
<tr>
<td>E MCM 11-28</td>
<td>LLE 11-33</td>
</tr>
<tr>
<td>ENB 11-39</td>
<td>LLEQ 11-33</td>
</tr>
<tr>
<td>ERA 11-32</td>
<td>LLCE 11-33</td>
</tr>
<tr>
<td>ERL 11-32</td>
<td>LLGT 11-33</td>
</tr>
<tr>
<td>ESIM 11-30</td>
<td>LLL 11-51</td>
</tr>
<tr>
<td>E VIM 11-30</td>
<td>LLLE 11-33</td>
</tr>
<tr>
<td>EXB 11-43</td>
<td>LLLT 11-33</td>
</tr>
<tr>
<td>FAD 11-18</td>
<td>LLNE 11-33</td>
</tr>
<tr>
<td>FCM 11-18</td>
<td>LLR 11-52</td>
</tr>
<tr>
<td>FCS 11-19</td>
<td>LLS 11-52</td>
</tr>
<tr>
<td>FCST 11-20</td>
<td>LTT 11-33</td>
</tr>
<tr>
<td>FD BL 11-19</td>
<td>LMCM 11-28</td>
</tr>
<tr>
<td>FDV 11-19</td>
<td>LNE 11-33</td>
</tr>
<tr>
<td>FL D 11-19</td>
<td>LPID 11-36</td>
</tr>
<tr>
<td>FLTA 11-20</td>
<td>LPSW 11-36</td>
</tr>
<tr>
<td>FLT L 11-20</td>
<td>LRL 11-52</td>
</tr>
<tr>
<td>FMP 11-20</td>
<td>LRR 11-52</td>
</tr>
<tr>
<td>FRN 11-20</td>
<td>LRS 11-53</td>
</tr>
<tr>
<td>FSB 11-20</td>
<td>LT 11-34</td>
</tr>
<tr>
<td>FS LE 11-20</td>
<td>LWGS 11-37</td>
</tr>
<tr>
<td>FSM I 11-21</td>
<td>MDEI 11-28</td>
</tr>
<tr>
<td>FS NZ 11-21</td>
<td>MDII 11-28</td>
</tr>
<tr>
<td>FP SL 11-21</td>
<td>MDIW 11-28</td>
</tr>
<tr>
<td>FST 11-21</td>
<td>MDRS 11-28</td>
</tr>
<tr>
<td>FSZE 11-21</td>
<td>MDWC 11-28</td>
</tr>
<tr>
<td>HLT 11-35</td>
<td>MIA 11-37</td>
</tr>
<tr>
<td>IAB 11-40</td>
<td>MIB 11-37</td>
</tr>
<tr>
<td>I CL 11-40</td>
<td>MPL 11-25</td>
</tr>
<tr>
<td>IRC 11-40</td>
<td>MPY 11-25</td>
</tr>
<tr>
<td>I LE 11-40</td>
<td>NF YB 11-49</td>
</tr>
<tr>
<td>IMA 11-40</td>
<td>NF YE 11-49</td>
</tr>
<tr>
<td>INB C 11-49</td>
<td>NOP 11-37</td>
</tr>
<tr>
<td>INB C 11-49</td>
<td>NOP 11-53</td>
</tr>
<tr>
<td>INEC 11-49</td>
<td>PCL 11-47</td>
</tr>
<tr>
<td>INEN 11-49</td>
<td>PIDA 11-28</td>
</tr>
<tr>
<td>INH 11-30</td>
<td>PIDL 11-26</td>
</tr>
<tr>
<td>INT 11-21</td>
<td>PIMA 11-26</td>
</tr>
<tr>
<td>INTA 11-21</td>
<td>PIML 11-26</td>
</tr>
<tr>
<td>IN TL 11-22</td>
<td>PRTN 11-48</td>
</tr>
<tr>
<td>IRS 11-53</td>
<td>PT LB 11-37</td>
</tr>
<tr>
<td>IRX 11-53</td>
<td>RBQ 11-50</td>
</tr>
<tr>
<td>ITBL 11-35</td>
<td>RCB 11-31</td>
</tr>
<tr>
<td>J MP 11-46</td>
<td>RMC 11-28</td>
</tr>
<tr>
<td>JST 11-46</td>
<td>RRST 11-38</td>
</tr>
<tr>
<td>JSX 11-47</td>
<td>RSAV 11-38</td>
</tr>
<tr>
<td>JS XB 11-47</td>
<td>RTQ 11-50</td>
</tr>
<tr>
<td>JSY 11-47</td>
<td>SIA 11-26</td>
</tr>
<tr>
<td>LCEQ 11-33</td>
<td>S2A 11-27</td>
</tr>
<tr>
<td>LCGE 11-33</td>
<td>SAR 11-53</td>
</tr>
<tr>
<td>LG T 11-33</td>
<td>SAS 11-33</td>
</tr>
<tr>
<td>LCLE 11-33</td>
<td>SBL 11-27</td>
</tr>
<tr>
<td>LCLT 11-33</td>
<td>SCB 11-31</td>
</tr>
<tr>
<td>LCNE 11-33</td>
<td>SGT 11-54</td>
</tr>
<tr>
<td>LDA 11-40</td>
<td>SKP 11-54</td>
</tr>
<tr>
<td>LCD 11-5</td>
<td>SLE 11-54</td>
</tr>
<tr>
<td>LDL 11-40</td>
<td>SLN 11-55</td>
</tr>
<tr>
<td>LDLR 11-41</td>
<td>SLZ 11-55</td>
</tr>
<tr>
<td>LDX 11-41</td>
<td>SMCR 11-29</td>
</tr>
<tr>
<td>LDY 11-41</td>
<td>SMCS 11-29</td>
</tr>
<tr>
<td>LEQ 11-33</td>
<td>SMI 11-55</td>
</tr>
<tr>
<td>LF 11-34</td>
<td>RMQ 11-54</td>
</tr>
<tr>
<td>LF EQ 11-34</td>
<td>SNS 11-54</td>
</tr>
<tr>
<td>LFGE 11-34</td>
<td>SNZ 11-55</td>
</tr>
<tr>
<td>LF GT 11-34</td>
<td>SPL 11-55</td>
</tr>
<tr>
<td>LFLE 11-34</td>
<td>SR 11-55</td>
</tr>
<tr>
<td>LFL I 11-15</td>
<td>SR 11-55</td>
</tr>
<tr>
<td>LF L T 11-34</td>
<td>SR 11-55</td>
</tr>
</tbody>
</table>

```
| 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 | STL C 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 |
| TAX 11-31 | TAX 11-42 |
| TAY 11-42 | TBA 11-43 |
| TCA 11-28 | TCL 11-28 |
| TFLL 11-15 | TKA 11-31 |
| TLF 11-15 | TLF 11-15 |
| TSTQ 11-50 | TXA 11-43 |
| TAY 11-43 | TAY 11-43 |
| VIRY 11-29 | WAIT 11-49 |
| WCS 11-39 | XAD 11-10 |
| XAD 11-10 | XBDT 11-11 |
| XGA 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 | ZE L 11-6 |
| ZFL 11-6 | ZMV 11-6 |
| ZMVD 11-7 | ZTRN 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

FDR 3059
X-16
1 March 1979
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
  XBTĐ [I] 12-5
  XBTĐ [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
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:  

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).

**Loading library subroutines**

Standard FORTRAN mathematical and input/output functions are implemented by subroutines in the library file FTNLIB in the LIB UF D. 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 MATHLIB, 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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Load Map Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>None, 0 or 4</td>
<td>Load state, base area, and symbol storage; symbols sorted by address</td>
</tr>
<tr>
<td>1</td>
<td>Load state only</td>
</tr>
<tr>
<td>2</td>
<td>Load state and base areas</td>
</tr>
<tr>
<td>3</td>
<td>Unsatisfied references only</td>
</tr>
<tr>
<td>6</td>
<td>Undefined symbols, sorted in alphabetical order</td>
</tr>
<tr>
<td>7</td>
<td>All symbols, sorted in alphabetical order</td>
</tr>
</tbody>
</table>

**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 U11 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.

*CMLLOW*: The low end of COMMON.

*CMLHIGH*: 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    XXXXX    YYYYY    ZZZZZZ    WWWWWW

 XXXXX    Lowest location defined for this area
 YYYYY    Next available location if starting up from XXXXX
 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 runfile.

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
FORCLOAD
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.
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 [OFF] [ON]
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/ [(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 [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 [NO]**
Controls use of sector zero base area in procedure segments.

**S/LIBRARY FORCELOAD [pathname] [addr psegno lsegno]**
\[
\begin{align*}
\text{S/} & \quad \text{LIBRARY} \\
& \quad \text{FORCELOAD} \\
& \quad \text{[pathname]} \quad \text{[addr psegno lsegno]}
\end{align*}
\]
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.

\[
\begin{align*}
\text{SK} & \left\{ \begin{array}{l}
\text{ssize} \\
\text{segno addr} \\
\text{ssize 0 esegno} \\
\text{segno addr esegno}
\end{array} \right. \\
\end{align*}
\]
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.
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>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTNLIB</td>
<td>FORTAN Library</td>
<td>R</td>
</tr>
<tr>
<td>PFTNLB</td>
<td>FORTAN Library pure procedures</td>
<td>V</td>
</tr>
<tr>
<td>IFTNLB</td>
<td>FORTAN Library impure procedures</td>
<td>V</td>
</tr>
<tr>
<td>PLIGLB</td>
<td>PL/1 V and I Mode Library</td>
<td>V</td>
</tr>
<tr>
<td>NPFTNLB</td>
<td>FORTAN Library V-Mode, Pure, non-shared</td>
<td>V</td>
</tr>
<tr>
<td>APPLIB</td>
<td>Application Library</td>
<td>R</td>
</tr>
<tr>
<td>VAPPLIB</td>
<td>Application Library</td>
<td>R</td>
</tr>
<tr>
<td>SRTLIB</td>
<td>Sort Library-Files</td>
<td>V</td>
</tr>
<tr>
<td>VSRTLI</td>
<td>Sort Library-Files</td>
<td>R</td>
</tr>
<tr>
<td>MSORTS</td>
<td>Sort Library-Memory</td>
<td>R</td>
</tr>
<tr>
<td>MATHLIB</td>
<td>Matrix Routines</td>
<td>R</td>
</tr>
<tr>
<td>No Arguments</td>
<td>Main Program</td>
<td>Two or More Arguments</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>CALL SUBX</td>
<td>CALL SUBX</td>
<td>CALL SUBX</td>
</tr>
<tr>
<td>DAC A</td>
<td>DAC A</td>
<td>DAC A</td>
</tr>
<tr>
<td></td>
<td>DAC B</td>
<td>DAC B</td>
</tr>
<tr>
<td></td>
<td>DAC C</td>
<td>DAC C</td>
</tr>
<tr>
<td></td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td></td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td></td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td></td>
<td>DAC Ø</td>
<td>DAC Ø</td>
</tr>
</tbody>
</table>

**Subroutine**

<table>
<thead>
<tr>
<th>ENT SUBX</th>
<th>ENT SUBX</th>
<th>ENT SUBX</th>
</tr>
</thead>
<tbody>
<tr>
<td>REL</td>
<td>REL</td>
<td>REL</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>SUBX DAC **</td>
<td>SUBX DAC **</td>
<td>SUBX DAC **</td>
</tr>
<tr>
<td>first instruction</td>
<td>CALL F$AT</td>
<td>CALL F$AT</td>
</tr>
<tr>
<td>·</td>
<td>DEC 1</td>
<td>DEC 3</td>
</tr>
<tr>
<td>:</td>
<td>APTR DAC **</td>
<td>APTR DAC **</td>
</tr>
<tr>
<td>:</td>
<td>BPTR DAC **</td>
<td>BPTR DAC **</td>
</tr>
<tr>
<td>:</td>
<td>CPTR DAC **</td>
<td>CPTR DAC **</td>
</tr>
<tr>
<td>JMP SUBX,*</td>
<td>first instruction</td>
<td></td>
</tr>
<tr>
<td>·</td>
<td>·</td>
<td></td>
</tr>
<tr>
<td>·</td>
<td>·</td>
<td></td>
</tr>
<tr>
<td>JMP SUBX,*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry-point</td>
<td>Procedure relative value: entry point for subroutine.</td>
</tr>
<tr>
<td>link-base</td>
<td>Link base register value.</td>
</tr>
<tr>
<td>displacement</td>
<td>Displacement in stack frame for argument list. May be stack relative or absolute expression.</td>
</tr>
<tr>
<td>n-arguments</td>
<td>Number of arguments expected; default is zero.</td>
</tr>
<tr>
<td>stack-size</td>
<td>Initial stack frame size. Default is maximum area specified in DYNM statements.</td>
</tr>
<tr>
<td>keys</td>
<td>CPU keys for procedure. Default is 64V addressing mode, all other keys zero.</td>
</tr>
</tbody>
</table>

For example:

```
    ENT   ECB Nam
    Lab1  LDA  =123
          LINK
    ECB Nam  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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSS</td>
<td>Allocate block starting with symbol</td>
<td></td>
</tr>
<tr>
<td>BES</td>
<td>Allocate block ending with symbol</td>
<td></td>
</tr>
<tr>
<td>BSZ</td>
<td>Allocate block set to zeroes</td>
<td></td>
</tr>
<tr>
<td>COMM</td>
<td>FORTRAN compatible COMMON</td>
<td></td>
</tr>
</tbody>
</table>

[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 COMMON 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNM</td>
<td>Declare stack relative</td>
<td>R and V only</td>
</tr>
<tr>
<td>EQU</td>
<td>Symbol definition</td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>Symbol definition</td>
<td></td>
</tr>
<tr>
<td>XSET</td>
<td>Symbol definition</td>
<td></td>
</tr>
</tbody>
</table>
USER SURVEY
Tell us how we’re doing, and we’ll send you a free Programmer’s Companion.

<table>
<thead>
<tr>
<th>Your name</th>
<th>Company or School</th>
<th>Address</th>
<th>City, State, Zip</th>
</tr>
</thead>
</table>

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?
   a. If YES, which model?
   b. Is it networked with other Prime computers?
   c. Is it networked with any of these?
   d. Which of these software packages do you use?
   e. Have you read any other Prime documents?
   f. If YES, which ones?
4. Are you presently evaluating Prime?
   Is the documentation playing a part?
5. What book are you reviewing?
6. My initial reaction to this book was:
7. *After reading it my reaction was:*
   If BETTER or WORSE why?
8. How often have you used this book?
9. Did the book have the content you expected?
   If NO, why?
10. Did you find the organization useful?
    If NO, why?

- YES  NO
- 450  550  650  750  OTHER
- YES  NO
- IBM  CDC  UNIVAC  HONEYWELL
- FORTRAN  FORTRAN 77  COBOL
- MIDAS  PL/I-G  DBMS  POWER
- RPGII  FORMS  PASCAL  SPSS
- RJE  DPTX  PRIMENET
- YES  NO
- YES  NO
- YES  NO
- EXCELLENT  GOOD  FAIR
- VERY GOOD  FAIR
- BETTER  THE SAME  WORSE
- EVERY DAY  FAIRLY OFTEN
- VERY OFTEN  JUST GOT IT
- YES  NO
- YES  NO
11. How did you find the examples?
12. How did you find the illustrations?
13. Could you locate the information you needed?
14. Was the index adequate?
15. Please evaluate our writing style and edit quality:
   a. Clarity
   b. Tone
   c. Technical level
   d. General writing quality
   e. Editorial quality
16. Which of these manufacturers documentation have you used the most?

   How is Prime compared to theirs?

   Any comment on your rating?
17. Please evaluate the graphic quality:
   a. Rate the general presentation

   b. Do you like the paper color?
   c. Did you find any quality defects?
   d. Do you like the way we've used color for abbreviations and user input?
   e. Did the shading over the tables and charts make them EASIER or HARDER to read?
   f. Do you like the Programmer's Companion concept?
   g. Which form of bindery do you find most useful?
18. Do you know about the AIDUS program?
19. Any other comments:

<table>
<thead>
<tr>
<th>Question</th>
<th>Response Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>11. How did you find the examples?</td>
<td>☐ Too many ☐ About right ☐ Too few</td>
</tr>
<tr>
<td>12. How did you find the illustrations?</td>
<td>☐ Too many ☐ About right ☐ Too few</td>
</tr>
<tr>
<td>13. Could you locate the information you needed?</td>
<td>☐ Yes ☐ No</td>
</tr>
<tr>
<td>14. Was the index adequate?</td>
<td>☐ Yes ☐ No</td>
</tr>
<tr>
<td>15. Please evaluate our writing style and edit quality:</td>
<td>☐ Very clear ☐ Average ☐ Unclear</td>
</tr>
<tr>
<td>a. Clarity</td>
<td>☐ Correct ☐ Neutral ☐ Stilted</td>
</tr>
<tr>
<td>b. Tone</td>
<td>☐ Too high ☐ About right ☐ Too low</td>
</tr>
<tr>
<td>c. Technical level</td>
<td>☐ Excellent ☐ Average ☐ Fair ☐ Poor</td>
</tr>
<tr>
<td>d. General writing quality</td>
<td>☐ Excellent ☐ Average ☐ Fair ☐ Poor</td>
</tr>
<tr>
<td>e. Editorial quality</td>
<td>☐ IBM ☐ Digital ☐ DG</td>
</tr>
<tr>
<td>f. HONEYWELL</td>
<td>☐ HP ☐ CDC ☐ UNIVAC</td>
</tr>
<tr>
<td>g. GE</td>
<td>☐ BURROUGHS ☐ WANG</td>
</tr>
<tr>
<td>h. XEROX</td>
<td>☐ MUCH BETTER ☐ A LITTLE WORSE</td>
</tr>
<tr>
<td>i. SAME</td>
<td>☐ A LITTLE BETTER ☐ MUCH WORSE</td>
</tr>
<tr>
<td>17. Please evaluate the graphic quality:</td>
<td>☐ Excellent ☐ Good ☐ Poor</td>
</tr>
<tr>
<td>a. Rate the general presentation</td>
<td>☐ Very good ☐ Fair</td>
</tr>
<tr>
<td>b. Do you like the paper color?</td>
<td>☐ Yes ☐ No</td>
</tr>
<tr>
<td>c. Did you find any quality defects?</td>
<td>☐ Yes ☐ No</td>
</tr>
<tr>
<td>d. Do you like the way we've used color for abbreviations and user input?</td>
<td>☐ Yes ☐ No</td>
</tr>
<tr>
<td>e. Did the shading over the tables and charts make them EASIER or HARDER to read?</td>
<td>☐ Easier ☐ Harder</td>
</tr>
<tr>
<td>f. Do you like the Programmer's Companion concept?</td>
<td>☐ Yes ☐ No ☐ Haven't seen one</td>
</tr>
<tr>
<td>g. Which form of bindery do you find most useful?</td>
<td>☐ Bound ☐ Loose-leaf</td>
</tr>
<tr>
<td>19. Any other comments:</td>
<td>☐ Yes ☐ No</td>
</tr>
</tbody>
</table>
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
☐ FORTRAN 77  ☐ BASIC/VM
☐ ASSEMBLY  ☐ POWER
☐ ADMINISTRATOR  ☐ WORD PROCESSING