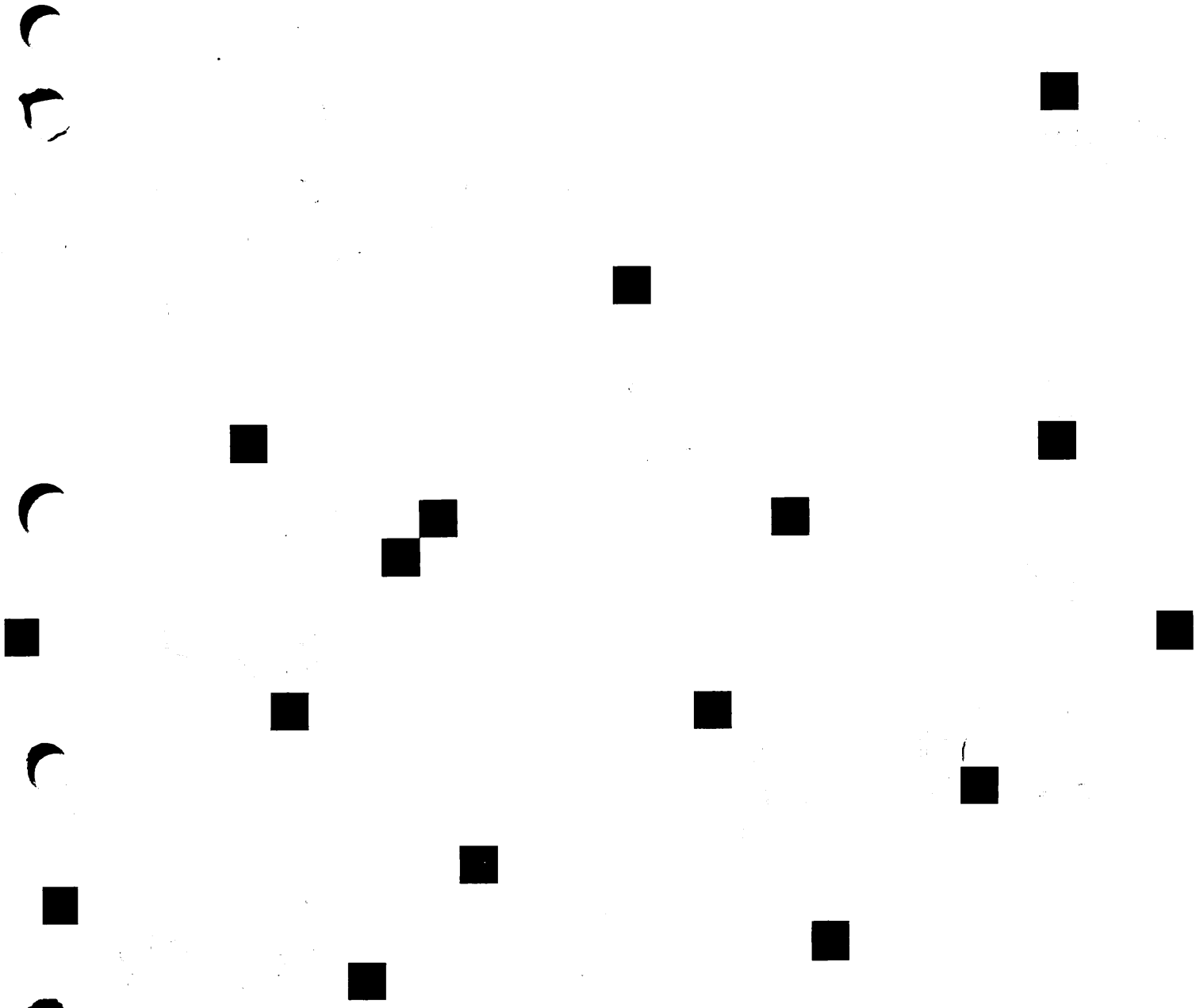


Prime™

Assembly Language Programmer's Guide

Revision 21.0

DOC3059-2LA

Assembly Language Programmer's Guide

Second Edition

by

Len Bruns

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 21.0 (Rev. 21.0).

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

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

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

PRINTING HISTORY

First Edition	(FDR3059-101)	March 1979 for Release 16.3
Update 1	(COR3059-001)	January 1980 for Release 17.2
Update 2	(PTU2600-104)	June 1983 for Release 19.2
Second Edition	(DOC3059-2LA)	July 1987 for Release 21.0

CREDITS

Editorial:	Thelma Henner
Project Support:	Margaret Taft
Illustration:	Mingling Chang
Document Preparation:	Celeste Henry
Production:	Judy Gordon

HOW TO ORDER TECHNICAL DOCUMENTS

To order copies of documents, or to obtain a catalog and price list:

United States Customers	International
Call Prime Telemarketing, toll free, at 1-800-343-2533, Monday through Friday, 8:30 a.m. to 5:00 p.m. (EST).	Contact your local Prime subsidiary or distributor.

CUSTOMER SUPPORT

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (within Massachusetts)	1-800-541-8888 (within Alaska)
1-800-343-2320 (within other states)	1-800-651-1313 (within Hawaii)

For other locations, contact your Prime representative.

SURVEYS AND CORRESPONDENCE

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Contents

About This Book

The Assembly Language Programmer's Guide, Second Edition, documents the use of the Prime Macro Assembler (PMA) as implemented at PRIMOS Revision 21.0. It replaces the first edition of the same guide and its various updates. It is a completely rewritten guide, whose salient features are

- Reorganization of the text into a sequence of chapters that more closely parallels an actual assembly. Introductory chapters give an overview of the assembler and describe its invocation and command line options. The remaining chapters discuss coding the program, defining and calling macros and subroutines, linking the program, executing, and debugging.

- Removal of most material that is duplicated in other volumes. Where required for an understanding of the subject under discussion, this material has been replaced by references to the appropriate manuals. This guide is, therefore, a more compact but no longer self-contained reference text; it is intended to be used in conjunction with other manuals. A list of reference documents appears later in this preface.

- Elimination of discussions of S mode and R mode. Use of these older addressing modes is declining, and the user is urged to do all new programming in the current V, I, and IX addressing modes. For those users who need to maintain existing S-mode and R-mode programs, the relevant information in the first edition of this guide is still valid. The System Architecture Reference Guide and the Instruction Sets Guide also continue to present information on S mode and R mode.

Summary of Chapters and Appendices

This book contains the following chapters and appendices:

Chapter 1 is an overview of the current implementation of the assembler; Chapter 2 describes its method of invocation and the various command line options available.

Chapter 3 is a detailed description of the language elements: statements and their components; terms and expressions; and the functions of statement components and fields.

Chapters 4 through 7 discuss the coding of the four major categories of pseudo-operations and their functions and requirements.

Chapters 8 through 10 discuss, respectively, the instruction sets for V mode, I mode, and IX mode.

Chapter 11 describes the coding and calling of macro routines and the logic capabilities of macro processing, while Chapter 12 is devoted to several methods of calling local and external subroutines.

Chapter 13 discusses simple linking of assembled programs using the SEG and BIND linkers, with references to other documents for more complex linking tasks.

Chapter 14 briefly describes the invocation of linked programs, and goes on to discuss V-mode and I-mode debugging in some detail.

Three appendices provide reference material in the form of a list of assembler error messages (Appendix A); a summary of the V, I, and IX mode instruction sets (Appendix B); and a description of the Prime Extended Character Set (Prime ECS) and its use (Appendix C).

Reference Documents

The following guides are frequently referred to in the text of this book.

System Architecture Reference Guide, DOC9473-2LA

Instruction Sets Guide, DOC9474-2LA

SEG and LOAD Reference Guide, DOC3524-192 and update, UPD3524-21A, for Rev. 19.4

Programmer's Guide to BIND and EPFs, DOC8691-1LA

Advanced Programmer's Guide, Vol. I: BIND and EPFs, DOC10055-1LA

Advanced Programmer's Guide, Vol. II: File System, DOC10056-2LA

Advanced Programmer's Guide, Vol. III: Command Environment, DOC10057-1LA

Advanced Programmer's Guide, Vol. 0: Introduction and Error Codes, DOC10066-2LA

Subroutines Reference Guide, Vol. I: Language Interface, DOC10080-2LA

Subroutines Reference Guide, Vol. II: File System, DOC10081-1LA and update, UPD10081-11A

Subroutines Reference Guide, Vol. III: Operating System, DOC10082-1LA and update, UPD10082-11A

Subroutines Reference Guide, Vol. IV: User Libraries, DOC10083-1LA and update, UPD10083-11A

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, and in examples throughout this document. Examples illustrate the uses of these commands and statements in typical applications.

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase.	SLIST
	In PMA statements, uppercase elements are entered exactly as shown, and only in uppercase.	DATA value
lowercase	In command formats and PMA statements, words coded in lowercase indicate variables for which you must substitute a value.	LOGIN user-id DATA value
Abbreviations in format statements	If an uppercase word in a command format has an abbreviation, either the abbreviation is underscored or the name and abbreviation are placed within braces.	LOGOUT { SET_QUOTA { SQ
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or more of these items.	LD [-BRIEF [-SIZE
Braces { }	Braces enclose a list of items. Choose one and only one of these items.	CLOSE { filename { ALL
Vertical bars within brackets [*A|B|C|]	Vertical bars within brackets offer a choice among two or more items. Choose either none or one of these items; do not choose more than one. Asterisked value is the default.	-L [*YES|NO|TTY]

Convention	Explanation	Example
Ellipsis ...	An ellipsis indicates that the preceding item may be entered more than once in a format.	DATA value, ...
Parentheses ( )	In command or statement formats, you must enter parentheses exactly as shown.	SPOOL FILE(1 2 3) XFER MAC (FROM)=1
Hyphen -	Wherever a hyphen appears as the first character of an option, it is a required part of that option.	SPOOL -LIST
Underscore in examples	In examples, user input is underscored but system prompts and output are not.	OK, RESUME MY_PROG This is the output of MY_PROG.CPL OK,
Apostrophe '	An apostrophe preceding a number indicates that the number is in octal.	'200
Angle brackets in examples < >	In examples, the name of a key enclosed within angle brackets indicates that you press that key.	OK, ED <RETURN>

1
Introduction

The Prime Macro Assembler (PMA) at PRIMOS Revision 21.0 incorporates several enhancements over previous versions. Some of these are related to the operation of the assembler itself, while others are concerned with new PRIMOS and hardware facilities. The following items describe the enhancements.

- The assembler is capable of assembling much larger object programs than in the past, owing to a new method of storing the symbol table. Symbol table size has until now been limited by the bounds of a single segment that also contains the assembler's executable code. The new storage mechanism uses a segment apart from the execution segment; moreover, it is capable of adding more symbol storage segments when the need arises.

- Two assembler attributes that were previously designated as spare are no longer so designated. Assembly language programmers who used these attributes for their own purposes can no longer do so. Attribute #108 is designated as reserved, while attribute #111 contains a parity value established by whether the length of the character string defined by the most recent DATA, BCI, or BCZ pseudo-operation was even (parity is 0) or odd (parity is 1). Assembler attributes are listed in Chapter 11.

- In addition to formerly-supported S, R, V, and I address modes, IX mode is supported as of Revision 21.0. IX mode operates on 50 Series Models 2550™, 9650™, 9750™, 9950™, 9955™, and 9955II™. It includes a small set of additional instructions that enable operations involving C language pointers and characters. These instructions are described in Chapter 10.

- The assembler supports an I mode addressing enhancement that permits a general-register-relative format in memory reference instructions and adds two indirect pointer related instructions, AIP and LIP. GRR format (described in Chapter 9) improves the performance of programs that must address large arrays that could potentially cross segment boundaries. This format operates only on the 50 Series models listed for IX mode in the preceding paragraph.

- The assembler supports the Prime extended character set (Prime ECS). The extended character set and its implications for assembly-language programs are described in Appendix C.

- A new subroutine calling mechanism, shortcall, is described in Chapter 12. Shortcall provides a much faster transfer of control to and from a PMA-written subroutine than the traditional PCL/PRTN mechanism. Shortcall, from a high-level language caller's point of view, is currently implemented only in Fortran 77. I-mode PMA shortcalled routines can take advantage of GRR, register-to-register, and immediate addressing formats to further enhance overall program efficiency.

- The assembler produces binary files that are compatible with the BIND linker, introduced at Revision 19.4, to create runfiles in executable program (EPF) format. A description of a simple BIND session appears in Chapter 13.

2
Using PMA

The Prime Macro Assembler (PMA) is a three-pass assembler. The first pass creates a symbol table containing internal symbols and their segment-relative displacements, and identifies external references. The second pass uses the symbol table to resolve references to the internal symbols, generates object code blocks for input to the linker and, optionally, creates a listing. The third pass 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]
```

pathname specifies the pathname of your source PMA program. The standard pathname conventions apply. File naming conventions are described following the option descriptions.

PMA supports the following options. Default options are indicated by asterisks (*).

```
    -BINARY, -B [* YES|NO|pathname]              -RESET
    -ERRLIST                                     -ROUND
  * -EXPLIST                                   * -XREFL
    -INPUT [pathname], -I [pathname]             -XREFS
    -LISTING, -L [* YES|NO|TTY|SPOOL|pathname]
```

Brief descriptions of these options are given below.

-BINARY [* YES|NO|pathname]

Specifies binary (object) file.

[YES] gives source-program.BIN if source program name has .PMA suffix, otherwise gives B_source-program. Binary file is in the home directory.
[NO] gives no binary file.
[pathname] allows complete specification of binary file.

-ERRLIST

Generates errors-only listing (overrides pseudo-operation NLIST).

* -EXPLIST

Generates full assembly listing (overrides pseudo-operation NLIST).

-INPUT [pathname]

Specifies source program.

[pathname] is the name of source program. (Do not use if name immediately follows the PMA command). Standard pathname conventions apply.

-LISTING [* YES|NO|TTY|SPOOL|pathname]

Specifies listing file.

[YES] gives source-program.LIST if source program has .PMA suffix, otherwise gives L_source-program. Listing file is in the home directory.
[NO] gives no listing file.
[TTY] displays assembly listing at the terminal.
[SPOOL] puts listing file into line printer spool queue.
[pathname] allows complete specification of listing file.

-RESET

Resets A, B, and X Register settings.

-ROUND

Rounds rather than truncates conversion of real numbers to decimal.

* -XREFL

Generates complete cross reference listing.

-XREFS

Omits from cross reference list symbols that are defined but not used.

FILE NAMING CONVENTIONS

For consistency with Prime's other language processors, the pathname of the source file should be suffixed with a language name code. For the assembler, the code is .PMA. The form of the source filename affects the form of the default names of the binary and listing files: if the source filename suffix is not .PMA, the default binary and listing filenames are prefixed by B_ and L_, respectively; if the source filename suffix is .PMA, the default binary and listing filenames are suffixed by .BIN and .LIST, respectively. The .PMA form is recommended, both for consistency and for ease of use in subsequent operations such as linking and invoking the resulting programs.

The defaults for both binary and listing filenames can be overridden by specifying different pathnames as arguments to the -BINARY and -LISTING options, respectively. The -LISTING option also accepts TTY as an argument to cause the assembly listing to appear at your terminal.

FILE USAGE

Three files may be involved during an assembly:

File Type	PRIMOS File unit
Source	1
Listing	2
Binary	3

PMA automatically opens files for listing and binary output. They are closed at the termination of each assembler run.

The PRIMOS commands LISTING and BINARY permit you to concatenate two or more listing files, and two or more binary files, respectively. These commands, when used before a series of PMA invocations, open file units 2 (for the listing file) and 3 (for the binary file). The assembler uses them to write the files, and leaves them open when it returns control to PRIMOS. Each subsequent invocation of the assembler appends its listing and binary outputs to those already written. The files can be closed by use of the PRIMOS CLOSE command. Refer to the PRIMOS Commands Reference Guide for descriptions of these commands.

ASSEMBLER MESSAGES

After the assembler processes a program's END statement, it prints a message, terminates assembly, and returns control to PRIMOS command level. The message contains a decimal error count, the assembler version number, and a copyright statement:

<nn> ERRORS [PMA <version> Copyright (c) Prime Computer, Inc. <year>]

LISTING FORMAT

Figure 2-1 shows a section of a typical assembly listing and illustrates the main features.

When the assembly listing file is printed using the SPOOL command with no options, each page begins with a header and contains a page number. (Some Spooler options disable headers and pagination; refer to the description of the SPOOL command in the PRIMOS Commands Reference Guide.) The first statement in a program is used as the initial page header. If column 1 of any source statement contains an apostrophe ('), columns 3 through 80 of that statement become the header for all pages that follow, until a new header is specified.

At the end of the assembly listing appears a cross reference table containing each symbol's name (in alphabetical order), the symbol's address value with a code indicating the segment in which it resides, and a list of all line numbers defining or referring to the symbol. The 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 -XREFS option suppresses symbols which have been defined but not used.

```
     SAMPLE ASSEMBLER LISTING
                          (0001) '     SAMPLE ASSEMBLER LISTING
         SEG
                          (0001)            SEG
000000:      02.000015   (0002) START      LDA   M
000001:      16.000016   (0003)            MPY   X
000002: 015414.000017    (0004)            ADL   B
000004:        000015    (0005)            PIMA
000005:      04.000400L  (0006)            STA   OUT
000006: 061432.000422L   (0007)            CALL  TODEC
000010: 001300.000400L   (0008)            AP    OUT,SL
000012: 061432.000424L   (0009)            CALL  TONL
000014:        000611    (0010)            PRTN
000015:        000003    (0011) M          DATA  3
000016:        000005    (0012) X          DATA  5
000017:        000000    (0013) B          DATA  10L
000020:        000012
                          (0014)            LINK
000400>        000000    (0015) OUT        DEC   0
000401>        000000    (0016) ECB$       ECB   START
               000012
               000011
               000000
               177400
               014000

           000421         (0017)            END   ECB$


000421>      00.000000A
000422> 000000.000000E
000424> 000000.000000E

TEXT SIZE:   PROC 000021   LINK 000026   STACK 000012

B          000017  0004  0013
ECB$       000401L 0016  0017
M          000015  0002  0011
OUT        000400L 0006  0008  0015
START      000000  0002  0016
TODEC      000000E 0007
TONL       000000E 0009
X          000016  0003  0012


0000 ERRORS [PMA Rev. 21.0 Copyright (c) Prime Computer, Inc. 1986]
```

Sample Assembler Listing
Figure 2-1

ASSEMBLY LISTING SYMBOLOGY

The first two columns of an assembly listing show the octal representations of the addresses the assembler assigns to each area allocated for storage of machine instruction or data strings, and the generated strings themselves, if any. Appended to each entry in these columns is a code (a blank is considered a code). These codes are described in the following two sections.

Assignment Column Codes

The codes in the first assembly listing column indicate the segment to which the assembler assigns the accompanying generated string. The meanings of the codes are:

:	String is assigned to the procedure segment
>	String is assigned to the linkage segment

The assembler can also allocate space in the stack segment (by a DYNM pseudo-operation); these assignments have no entry in the first column, but have their stack-relative addresses listed in the second column with a blank assignment code.

Instruction and Data Column Codes

The second assembly listing column contains the assembler-generated instruction or data string. Its entries vary in length.

For data strings an entry contains six octal digits and represents one halfword (16 bits) of storage. For pseudo-operations such as DYNM, EQU, ORG, and END, it can also represent a 16-bit quantity that can be interpreted as either a numeric constant or an address, depending on how the assembler uses it. The entry is terminated by a blank.

For generic instructions (those that do not reference memory) an entry contains six digits and represents a 16-bit octal operation code. For memory reference instructions an entry consists of two parts separated by a period. The first part represents the operation code and contains two digits (for short-form instructions) or six digits (for long-form instructions). The second part represents a referenced memory address and is followed by a code indicating the storage class of the addressed item.

The codes associated with the instruction and data column have the following meanings:

blank	Addressed item is relative to the current module
A	Addressed item is an absolute number
E	Addressed item is external to the program
L	Addressed item is relative to the linkage base (LB)
P	Addressed item is relative to the procedure base (PB)
R	Addressed item is relative to a general register (I mode only)
S	Addressed item is relative to the stack base (SB)
X	Addressed item is relative to the auxiliary base (XB)

Other Listing Information

The remainder of the assembly listing consists of source program line images and, shown in parentheses, the assembler-assigned line number of each statement. These line numbers are for the benefit of the cross reference listing, described in the next section.

CROSS REFERENCE LISTING SYMBOLOGY

The cross reference listing shows, for each symbol defined or referenced in a program, the symbol name, its storage address and class, and one or more numbers indicating the statement line number in which it is defined or referenced.

The storage class codes that can appear in a cross reference listing are the same as those that can appear in the instruction and data column of the assembly listing; their meanings are also the same.

3
Language Structure

This chapter describes the structure and function of Prime Macro Assembler language statements and the elements with which they are constructed.

The PMA language structure is both flexible and simple. For example, here is a program which includes three pseudo-operations, a machine instruction and a literal.

```
SEG         Pseudo-operation -- assemble in V-mode
LDA ='301   Machine instruction with literal operand
CALL T10B   Pseudo-operation -- subroutine call (generates machine
            instruction).
END         Pseudo-operation -- defines end of source code.
```

LINES

Input to the assembler consists of statement, comment, and header lines. The basic unit of information is the line, which consists of fields separated by spaces. Line syntax is described later in this chapter.

All lines except comment and header lines must be entered in uppercase characters. Comment lines, header lines, and the comment fields of statement lines can be in uppercase or lowercase.

There are three basic line formats:

Comment Line	Column 1 contains an asterisk (*). The entire line is treated as a comment.
Header Line	Column 1 contains an apostrophe ('). The rest of the line is used as a page title for subsequent pages.
Statement	Statements are described in the following sections.

STATEMENTS

Every statement causes the assembler either to generate machine code (instructions or data) or to take some assembler or linker related action.

Statement Types

There are four kinds of statements:

Machine instructions
: Generate the instructions and data the program is to execute and use. Machine instructions are fully described in Chapters 8, 9, and 10, and in the Instruction Sets Guide.

Pseudo-operations
: Direct the assembler to perform some function during an assembly. With few exceptions, they do not generate machine instructions; they do, however, frequently generate data. Pseudo-operations are described in Chapters 4 through 7.

Macro definitions
: Delimit blocks of code or data (or both) that can be called as if they were instructions. This group also contains some pseudo-operations that provide a logic capability within a macro definition block. See Chapter 7.

Macro calls
: Invoke code previously defined in macro definitions. These are described in Chapter 11.

Statement Syntax

Statements can have up to four fields, delimited by spaces:

```
[label] operation [operand]...  [comment]
```

label — 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 are from 1 to 32 characters in length. The first character is a letter (A through Z), and the remaining characters can be letters, numerals (0 through 9), the dollar sign ($), or underscore (_).

operation — The operation field is the only field required in all types of instructions. It contains the mnemonic code for a machine instruction or a pseudo-operation. It is separated from the label field, if any, by one or more spaces.

operand — The number of operands and their meanings are operation-specific. Some statements do not require an operand, while others require one or more. The first operand is separated from the operation code by one or more spaces; multiple operands are separated by commas, and there must be no intervening spaces unless an operand is a literal that contains spaces. Literal operands and operands defining character constants can contain any character in the Prime ECS character set (see Appendix C).

comments — All text following either column 72 or two spaces after the last operand (10 spaces or following a colon in macro calls) is treated as a comment. Comments can contain any character in the Prime ECS character set.

Continuation Lines

Any statement can be interrupted by a semicolon (;) and continued on the next line. Any text following the semicolon is treated as a comment. Processing of the statement continues with the first nonspace character in the following line. Semicolons appearing within comments are not interpreted as continuation indicators. A semicolon that appears as a character in a literal must be preceded by the assembler's escape character, the exclamation point (!).

STATEMENT ELEMENTS

Statement elements -- labels, operation codes, and operands -- are composed of constants and symbols. These are made up of the subset of the printing ASCII characters defined for labels, above. The entire Prime extended character set (Prime-ECS), printing and nonprinting, can be used in comments, macro instruction operands, literals, and constants. Refer to Appendix C for a discussion of Prime-ECS.

Constants

Constants are explicit data values. They are most often used in operands of data-defining pseudo-operations and in literal operands. They can be used to represent bit configurations, absolute addresses, program-relative addresses (displacements), and data. A constant may be any of the following data types:

Decimal
Binary
Hexadecimal
Octal
Character
Address

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 through Z), and the remaining characters may be letters, numerals (0 through 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.

TERMS AND EXPRESSIONS

An operand can be constructed of one or more elements called terms, combined into an expression by use of one or more operators. These are described in this section.

Terms

A term is the smallest element that represents a distinct value. It represents a single precision signed integer and may be a constant or a symbol.

Every term, whether used alone or in an expression, has both a value and a mode. These attributes either are defined by the assembler and related to the procedure, stack, link, or common location counter, or they are 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 expression in their operand fields; labels, at the time of their definition, take the mode and value of the current location counter. Refer to the description of the ORG statement in Chapter 4 for a discussion of how the mode of the location counter is set). Some examples of terms are:

'123	Octal constant
C'A'	Character (string) constant
BETA	Symbol
1.23E2	Invalid because it is a floating point number; it does not have a single precision integer value
C'ABC'	Invalid because the value is too large for 16 bits

Value of a Term: The value of a term is its single precision numeric equivalent. It can represent either an address relative to some base in the program or an absolute number. Some examples of symbolic term definitions, usages, and values are shown below.

Symbol	Usage	Explanation
LABSYM	LABSYM LDA LOC	LABSYM is a label symbol whose value is the address (program counter value) of the instruction LDA LOC.
DATSYM	DATSYM DATA '10	DATSYM is a label symbol whose value is the address (program counter value) of the constant '10.
ADSYM	ADSYM DAC LOC	ADSYM is a label symbol whose value is the address (program counter value) of the address constant LOC.
ABSSYM	ABSSYM EQU '10	ABSSYM is a symbol whose value is '10.
CHRSYM	CHRSYM EQU C'A'	CHRSYM is a symbol whose value is A<space> ('140640).

Mode of a Term: The mode of a term defines whether the value associated with a symbol is absolute or relative to some base. A term can have one of the following modes:

Absolute	The value of the symbol is independent of its position relative to any base. Symbols equated to absolute terms or to the results of expressions involving only absolute terms have a mode of absolute.
Procedure Relative	The symbol is defined relative to the start of the module; it is identified by an asterisk if the current location counter's mode is procedure relative, or by reference to another procedure relative label.
Common Relative	The symbol is defined relative to a data area defined by a COMM pseudo-operation. This data area can be shared by several independently assembled or compiled routines.
External	The symbol is defined in a separately assembled module and is identified in the current module by an EXT pseudo-operation.
Stack Base Relative	The symbol is defined relative to the start of the current program's stack area. Variables defined by the DYNM pseudo-operation or by SB% + value have a mode of stack relative.
Procedure Absolute	The symbol is defined relative to the start of the procedure segment and is identified by PB% + value.
Linkage Base Relative	The symbol is defined relative to the start of the program's linkage area and is identified by LB% + value, or * if the current location counter's mode is linkage relative.
Auxiliary Base Relative	The symbol is defined relative to the contents of the auxiliary base register and is identified by XB% + value.

The mode of a term is represented internally in the assembler by a number from 0 through 7. The modes and their numeric equivalents are:

```
0    Absolute
1    Procedure relative
2    Common
3    External
4    Stack base relative
5    Procedure absolute
6    Linkage base relative
7    Auxiliary base relative
```

A term's mode number can be represented in an assembly statement by prefixing the symbol with a left bracket ([):

```
[ABC
```

It is thus possible to determine, for example, whether the mode of the term ABC is stack base relative or linkage base relative by a sequence such as

```
MODE    SET  [ABC
        IF   MODE .EQ. 4   statement_1
        IF   MODE .EQ. 6   statement_2
```

The assembler generates statement_1 if ABC is a stack based label, or statement_2 if it is a linkage based label.

Refer to the discussion of conditional assembly in Chapter 4 for a description of the use of IF and other conditional statements.

Expressions

Expressions contain one or more terms (constants or symbols) joined by operators. Expressions may contain arithmetic, logical, relational and shift operators.

Arithmetic Operators: Perform addition, subtraction, multiplication, and division operations:

Operator	Meaning	Example	Result (Octal)
+	Addition	'3+'4	000007
-	Subtraction	'10-'3	000005
*	Multiplication	'20*'10	000200
/	Division	'23/'10	000002

Division retains only the integer part of the quotient.

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

Operator	Meaning	Example	Result (Octal)
.OR.	Logical OR	'123.OR.'456	000577
.XOR.	Logical Exclusive OR	'123.XOR.'456	000575
.AND.	Logical AND	'123.AND.'456	000002

Relational Operators: Perform a comparison of two 16-bit operands with a result of 0 if false and 1 if true.

Operator	Relation	Example	Result (Octal)
.EQ.	equal	'123.EQ.'123	000001
		'123.EQ.'456	000000
.NE.	not equal	'123.NE.'123	000000
		'123.NE.'456	000001
.GT.	greater than	'123.GT.'123	000000
		'456.GT.'123	000001
.GE.	greater than or equal	'123.GE.'123	000001
		'123.GE.'456	000000
.LE.	less than or equal	'123.LE.'123	000001
		'123.LE.'456	000001
.LT.	less than	'123.LT.'456	000001
		'456.LT.'123	000000

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

```
argument-expression   .LS.   shift-count-expression
                      .RS.
```

Operator	Meaning	Example	Result (Octal)
.LS.	Left Shift	'123.LS.'3	001230
.RS.	Right Shift	'123.RS.'3	000012

Expression Conventions: The following conventions apply to the construction of expressions.

Spaces — Operators can be preceded and followed by a single space (more than one space causes the assembler to treat the rest of the line as a comment).

Signs — 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. Parentheses can be used to alter the order of evaluation.

Priority	Operator
Highest	* /
	+ -
	.RS. .LS.
	.GT. .GE. .EQ. .NE. .LE. .LT.
	.AND.
	.OR.
Lowest	.XOR.

Resultant Mode: For all operations except addition and subtraction, both operands' modes must be absolute; the resultant mode is absolute.

When an addition operator is used, no more than one of the terms can be relative. If one term is relative, the resultant mode is the mode of the relative term; if all terms are absolute, the resultant mode is absolute.

When a subtraction operator is used, one or both terms can be relative. If both are relative, they must be relative to the same base; the resultant mode is absolute. When one term is relative, it must be the first term; the resultant mode is the mode of the relative term.

For multiplication and division, both terms must be absolute; the resultant mode is absolute. For division, if the resultant value is not an integer, the fractional part is discarded and the value is the integer part.

FUNCTIONS OF STATEMENT FIELDS

This section describes the functions of the four fields of all assembly language statements.

Label Field

The label field equates a symbolic name appearing within it to a numeric value. The value can represent either the address of a program element (instruction or data) or a numeric constant. In either case, the value is a 16-bit integer quantity, and can range from 0 through 65535 decimal (0 through 177777 octal), inclusive.

A symbolic name is defined when it appears in a statement's label field. It is referenced when it appears in an operand field. This means that when a symbolic name, or label, appears in the label field, the assembler assigns a value to the name. The value assigned depends on the type of statement. For an instruction statement, the value is the location of the instruction relative to the start of the program. For a psuedo-operation, the value depends on the function of the pseudo-operation (See Chapters 4 through 7 for descriptions of these functions.)

When a label appears in the operand field of a statement, the assembler retrieves the assigned value and substitutes it for the symbolic name. For all machine instructions and most pseudo-operations, the relative placement in a program of statements that define labels and those that reference them is immaterial. A few pseudo-operations, however, require that labels used in their operand fields be defined before they are referenced. These requirements will be indicated where appropriate in the descriptions of pseudo-operations later in this guide.

Operation Field

The operation field of a statement contains the mnemonic operation code of an instruction or a pseudo-operation. An instruction mnemonic causes the generation of a machine instruction; a pseudo-operation mnemonic causes the assembler to take some action which may or may not result in the generation of machine code. An ORG pseudo-operation, for example, causes the assembler to reset its current location pointer to the value specified in its operand field, and assigns that value to the ORG's label if there is one; an EQU pseudo-operation simply assigns a numeric value to a label. A BSS pseudo-operation allocates a specified number of memory locations but does not fill them with anything; a DATA or DEC pseudo-operation allocates memory locations and stores specified values in them.

In an instruction statement, a % appended to the mnemonic forces the assembler to generate this instruction in long (32-bit) form, even though it would normally be generated in short form. The % notation is valid only in V mode. (Short and long form instructions are described in Chapters 8 and 9.)

A # appended to the mnemonic forces the assembler to generate this instruction in short (16-bit) form, even though it would normally be generated in long form (this can be done only in certain cases; see Chapters 8 and 9 for more information). The # notation is valid only in V mode.

Operand Field

The operand field, for those statements that require one, contains the representation of the program element to be acted upon. For a machine instruction it is normally an address expression, and may include indirection, indexing, and base register references. It can also, in certain cases, contain a numeric constant. Refer to TYPES OF ADDRESSING in Chapter 8 (V mode) or Chapter 9 (I mode) of this guide, and to Chapter 3 of the System Architecture Reference Guide. For pseudo-operations, the operand field performs a wide variety of functions, from defining the values of constants to controlling the actions of the linker after assembly is completed. Chapters 4 through 7 detail the actions and operand requirements of pseudo-operations.

Asterisk in the Operand Field: An asterisk in an operand field has two functions. One is to represent an address relative to the current value of the assembler location counter. When used by itself, its value is equal to the displacement, from the beginning of the procedure or linkage segment, of the statement the assembler is currently processing. It is frequently used along with a numeric increment (*+nn) or decrement (*-nn) to represent a displacement of nn halfwords relative to the current location.

The asterisk's other function is to indicate indirection, with or without indexing; in these cases it always appears following an address, possibly along with an index designator. It is separated from the address by a comma. Indirect addressing is described in Chapters 8 (V mode) and 9 (I mode).

Equal sign in the Operand Field: An equal sign defines a literal value that is to be used by an instruction. A literal is represented as a constant preceded by an equal sign. The following examples show the representations of various kinds of literal operands with a load instruction. In each case, what is loaded is the binary equivalent of the literal operand. If the binary form does not exactly fill the indicated number of bits, it is right-justified with leading zero bits for numeric literals, and left-justified with trailing space characters for character literals. Floating point literals always take exactly the indicated number of bits.

```
LDA  =123          decimal literal (16 bits)
LDL  =123L         long decimal literal (32 bits)
LDA  ='123         octal literal (16 bits)
LDA  =%110010      binary literal (16 bits)
LDA  =$2FF         hexadecimal literal (16 bits)
LDA  =C'AB'        character literal (16 bits)
LDL  =C'ABC'       character literal (32 bits), space-filled
LDL  =C'ABCD'      character literal (32 bits)
LDL  =Z'ABC'       character literal (32 bits), zero-filled
LDA  =EXPR         literal whose value is expression; see
                   the following text
FLD  =12.3E4       single-precision floating point (32 bits)
DFLD =12.3D4       double-precision floating point (64 bits)
QFLD =12.3Q4       quad-precision floating point (128 bits)
```

When a literal's value is defined by an expression, any symbols in that expression must be defined by EQU, SET, or XSET statements (described in Chapter 4). The expression's mode must be absolute and its value must be one that can be expressed as a 16-bit integer. (Refer to the discussion of terms and expressions, earlier in this chapter.) If SET or XSET is used to define a symbol, the symbol's value is that computed in the most recent SET or XSET defining that symbol.

Note

From the assembler's perspective, the size or type of a data literal does not have to match what is expected by an instruction using the literal. For example, a statement such as LDA =C'ABCD' (a 32-bit literal) is not flagged as an error, although only the first 16 bits of the literal are loaded into the A register (a 16-bit register).

In V mode, the assembler treats the numeric value of a literal as if it were a label assigned to a constant containing the literal's value. It reserves storage for the constant and stores the constant in that location. The assembler generates the constant's storage address as the operand of the instruction.

In I mode, a number of instructions permit a form of addressing known as immediate. (Refer to Immediate Addressing, in Chapter 9.) If an instruction allows immediate addressing and the literal value is expressible in 16 bits, the literal is stored in the second halfword of the instruction itself; otherwise it is treated and stored as in V mode.

Refer to the descriptions of the RLIT and FIN pseudo-operations in Chapter 5 for information on how and where literals are stored.

Comment Field

The comment field provides space for program documentation. It is generally used to describe the mechanics of a procedure. Unless otherwise noted, any text which begins two or more spaces after the last operand is treated by the assembler as a comment field. In a macro call, a comment field must either begin 10 or more spaces after the last operand, or be preceded by a colon (:).

PSEUDO-OPERATIONS

Pseudo-operation statements provide directions to the assembler or to the linker. Unlike machine instructions, they direct the actions of the assembler itself, rather than the actions of the assembled program. Some pseudo-operations generate machine code, but most do not. Those that do, define and allocate storage for data that the program is to use, in the form of data constants, address constants, or reserved areas for data storage or buffers.

Pseudo-operation functions described in this guide are of several classes. A list of these classes, and the chapters in which their detailed descriptions appear, follows.

AC	Assembly control (Chapter 4)
AD	Address definition (Chapter 5)
CA	Conditional assembly (Chapter 4)
DD	Data definition (Chapter 5)
LC	Listing control (Chapter 4)
LT	Literal control (Chapter 5)
LO	Loader control (Chapter 6)
MD	Macro definition (Chapter 7)
PL	Program linking (Chapter 6)
SA	Storage allocation (Chapter 5)
SD	Symbol definition (Chapter 4)

Table 3-1 contains an alphabetical listing of all the pseudo-operations, their functional class and their restrictions, if any.

All pseudo-operations have an operation field; most also have an operand field, separated from the operation field by spaces. Labels are usually optional, but some pseudo-operations either require a label to be present, or prohibit it.

The operation field contains the mnemonic name that identifies the pseudo-operation.

The operand field, for those pseudo-operations that require one, can contain one or more terms separated by single spaces or commas. Terms can be constants, symbols, or expressions as defined earlier in this chapter. In certain operations, such as BCI, terms can also consist of ASCII character strings.

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 addressing and indexing symbols. These are interpreted according to the addressing mode in effect when they are encountered.

Table 3-1
Pseudo-Operation Summary

Name	Function	Class	Comment
AP	Argument pointer	AD	
BACK	Loop back	CA	Macro definition only
BCI	Define ASCII string (blank fill)	DD	
BCZ	Define ASCII string (zero fill)	DD	
BES	Allocate block ending with symbol	SA	
BSS	Allocate block starting with symbol	SA	
BSZ	Allocate block set to zeros	SA	
CALL	External subroutine reference	PL	
CENT	Conditional entry	LO	
COMM	FORTRAN compatible common	SA	
D32I	Use 32I address mode	LO	
D64V	Use 64V address mode	LO	
DAC	Define 16-bit address constant	DD	
DATA	Define data constant	DD	
DEC	Define decimal integer constant	DD	
DFTB	Define table block	CA	
DFVT	Define value table	CA	
DUII	Define UII	LO	
DYMN	Define stack-relative symbol	SD	
DYNT	Direct entry definition	PL	
ECB	Entry control block	PL	
EJCT	Eject page	LC	
ELM	Enter loader mode	LO	
ELSE	Reverse conditional assembly	CA	
END	End of source statements	AC	
ENDC	End conditional assembly area	CA	
ENDM	End macro definition	MP	Macro definition only
ENT	Define entry point	PL	
EQU	Fixed symbol definition	SD	
EXT	External reference	PL	
FAIL	Force error message	CA	
FIN	Insert literals	LT	
GO	Forward reference	CA	
HEX	Define hexadecimal integer constant	DD	
IFTT	If table true	CA	
IFTF	If table false	CA	
IFVT	If value true	CA	
IF	If true	CA	
IFx	Arithmetic conditional if	CA	
IP	Indirect pointer	AD	
LINK	Put code in linkage segment	AC	
LIR	Load if required	LO	
LIST	Enable listing	LC	
LSMD	List macro expansions data only	LC	
LSTM	List macro expansions	LC	

Table 3-1 (continued)
Pseudo-Operation Summary

Name	Function	Class	Comment
MAC	Begin macro definition	MP	Macro definition only
NLSM	Don't List macro expansions	LC	
NLST	Inhibit listing	LC	
OCT	Define octal integer constant	DD	
ORG	Define origin location	AC	
PCVH	Print cross reference values in HEX	LC	
PROC	Put code in procedure segment	AC	
RLIT	Optimize literals	LT	
SAY	Print message	MP	
SCT	Select code within macro	MP	
SCTL	Select code from macro list	MP	
SEG	Segmentation assembly -- V-mode	AC	Must be first statement in source program
SEGR	Segmentation assembly -- I-mode	AC	Must be first statement in source program
SET	Changeable symbol definition	SD	
SETB	Set base sector	LO	
SUBR	Define entry point	PL	
SYML	Allow eight-character symbols	PL	
VFD	Define variable fields	DD	
XAC	External address definition	AD	
XSET	Changeable symbol defintion	SD	

MACHINE INSTRUCTIONS

Machine instruction statements generate the instructions that the assembled program is to execute. Machine instructions described in this guide are divided into several groups:

- Generic
- Branch and jump
- Memory reference
- Decimal
- Floating point
- Character
- Process control
- Restricted

All machine instructions are described in Chapters 8, 9, and 10 (for V, I, and IX modes respectively). A summary chart of all instructions for these modes is given in Appendix B.

RECOMMENDED PROGRAM STRUCTURE

PMA makes using the segmented architecture easy. The programmer can write straightforward code, such as LDA ADDR; the assembler, depending on the definition of ADDR, may generate a short or long instruction and may reference the stack area, the linkage area, the procedure area, or a temporary area. This is possible because symbols, during assembly, carry a great deal of state information with them.

The structure of a V-mode or I-mode program should reflect the system architecture design for the separation of code and data. The recommended structure is:

Prologue

SEG/SEGR	Indicates segmented addressing in V/I mode
RLIT	Puts literals in the procedure area
COMM	Declares FORTRAN-compatible COMMON areas
ENT	Declares entry point(s) to this program

Procedure/Stack Area

Executable code and dynamic storage

Data Area

LINK	Defines linkage area containing static variables
ECB	Declares entry control block for this program

End

END	Terminates assembly

All of the above declarations are pseudo-operations. Descriptions of these and other pseudo-operations appear in Chapters 4 through 7 of this guide.

4
Code Generation Pseudo-Operations

This chapter describes a group of pseudo-operations that control such things as the placement of generated code within a program (AC), equating symbols to absolute numeric values (SD), and conditional assembly of blocks of statements (CA). Assembly listing control pseudo-operations (LC) are also included in this chapter.

ASSEMBLY CONTROL PSEUDO-OPERATIONS (AC)

Assembly control pseudo-operations affect the placement of generated code and the addressing mode in which it is generated. The statements in this group are listed below.

Name	Function	Restrictions
END	End of source statements.	
LINK	Put code in linkage segment.	
ORG	Define origin location.	
PROC	Put code in procedure segment.	
SEG	Segmented assembly (V mode).	Must appear before any generated code.
SEGR	Segmented assembly (I mode).	Must appear before any generated code.

▶ [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. Refer to FIN and RLIT pseudo-operations in Chapter 5 for specific information on how these statements can affect literal placement.

The label field is permitted, but it serves no useful purpose and is usually omitted. While address-expression is indicated as optional, it is required for a main program, that is, the module that contains the entry point of the executable program called from the command processor. The address expression is the label specified in the ECB statement that defines the entry control block for this program. The typical sequence is:

```
BEGIN     first executable instruction
  .         .         .
  .         .         .

          LINK

          static data declarations
  .         .         .
  .         .         .

ECB_LOC   ECB     BEGIN
          END     ECB_LOC
```

End statements for modules called as subroutines do not need an operand; these modules' entry points are declared in any of several ways, as described in Chapter 12, USING SUBROUTINES.

▶ LINK

Places subsequent code in the linkage segment. The assembler's location counter mode is set to linkage relative and its value is set to one more than the highest value previously used in the linkage area. It starts at linkage-relative address '400. Linkage-relative mode is terminated by a PROC or COMM pseudo-operation.

The LINK pseudo-operation requires neither a label field nor an operand field.

▶ [label] ORG address-expression

Sets the assembler location counter equal to the mode and value of address-expression. Symbolic terms in the expression must have been previously defined. An expression containing an asterisk sets the mode and value to the current mode and value of the current location counter. The value may be modified by any terms that have absolute values, such as constants or symbols equated to constants.

The mode of the address expression may be absolute, procedure relative, linkage relative, or common, depending on the mode that was in effect when any symbolic term in address-expression was defined. The value of the location counter is set to the value of the address expression. If the mode of the address expression is absolute, then the mode of the location counter remains unchanged. In all other cases, whether relative, linkage, or common, both the mode and the value of the location counter are set to that of the address expression.

If a label appears in the label field, both the value and mode of the address expression are assigned to that label.

Be careful, when using an ORG statement in the procedure segment, that the preceding executable code is terminated by some kind of control transfer instruction such as a branch or jump; otherwise it is possible for execution to fall into an area containing data or uninitialized memory. If this happens, run-time errors such as ILLEGAL INSTRUCTION or ACCESS VIOLATION are likely to result.

▶ PROC

Places subsequent code in the procedure segment. The assembler's location counter mode is set to procedure relative and its value is set to one more than the highest value previously used in the procedure segment. It begins at procedure-relative address 0 (zero). Procedure-relative mode is terminated by a LINK or COMM pseudo-operation.

The assembler's location counter mode is procedure-relative by default at the beginning of an assembly.

The PROC pseudo-operation requires neither a label field nor an operand field.

▶ SEG [PURE / IMPURE]

Directs the assembler to create a segmented V-mode program. SEG must appear before any instruction, pseudo-operation, or macro call which generates instructions or data.

PURE and IMPURE are used only in a program that is to be linked by the BIND linker; they are ignored for programs linked by the SEG loader. (See Chapter 13, Linking and Loading, for more information on SEG and BIND.) PURE is the default, and need not be used if no locations within the procedure segment are modified by the program. IMPURE is required if the program modifies locations within the procedure segment. BIND in this case creates an impure (nonshared) procedure segment; that is, when both pure and impure procedures are linked in a single BIND run, separate shared and nonshared segments result.

SEG has the following effects:

- Sets the assembler into a three-pass mode, so that it can optimize stack and link frame references.
- Sets the instruction and address resolution mode to 64V.
- Initializes the assembler location counter to procedure relative zero.

▶ SEGR [PURE / IMPURE]

Directs the assembler to create a segmented I-mode program. SEGR must appear before any instruction, pseudo-operation, or macro call which generates instructions or data.

See the description of the SEG statement above for a discussion of the PURE and IMPURE operands.

SEGR has the following effects:

- Sets the assembler into a three-pass mode so that it can optimize stack and link frame references.
- Sets the instruction and address resolution mode to 32I.
- Initializes the assembler location counter to procedure relative zero.

CONDITIONAL ASSEMBLY PSEUDO-OPERATIONS (CA)

Conditional assembly pseudo-operations permit the selective inclusion or omission of one or more program statements, depending on a true/false condition test using internal assembler variables.

The statements in this group are listed below.

Name	Function	Restrictions
BACK	Loop back.	Macro definition only
DFTB	Define a symbol table	
DFVT	Define a value table	
ELSE	Start FALSE conditional assembly block	
ENDC	End conditional assembly block	
FAIL	Force error message	
GO	Skip forward	
IF	Start TRUE conditional assembly block	
IFx	Start TRUE conditional assembly block	

▶ [label-1] BACK [TO] label-2

Directs the assembler to repeat source statements that have already been assembled, beginning with the statement specified by label-2. label-1 and TO are optional (TO is used only to improve readability).

The operand label-2 must have been previously defined. It must appear in the label field of a SET statement whose operand is an asterisk (*). The correct sequence of statements is shown on the next page.

BACK statements are permitted only within a macro definition. They are normally used after one of the forms of IF pseudo-operations described below. Both BACK and label-2 must lie within the same MAC-ENDM range. (See Chapter 7 for descriptions of MAC and ENDM pseudo-operations.)

The relationship between SET and BACK statements is shown in the following structure.

```
MACNAME    MAC
  .         .
  .         .

label      SET    *
  .         .     .
  .         .     .
     code to be repeated
  .         .     .
  .         .     .

            IFx    condition
            BACK   label       go back if condition true
            ELSE/ENDC          required with IFx
  .         .     .
  .         .     .
            ENDM
```

▶ label DFTB (symbol-1, expression-1)[...]

Creates a table of symbols and their associated values. The symbol table is used in conjunction with the IFTF and IFTT statement (described later in this section). label is the table name, and symbol-1 is the name of a symbol whose value is an absolute number defined by expression-1. The symbols defined in a DFTB statement can be the same as symbols defined by instruction or other pseudo-operation statements; for example:

```
A       EQU     5
X       DFTB    (A,1),(B,5),(C,3)
X       LDA     A
```

There is no conflict between the As and Xs defined within the DFTB and those defined outside it.

If a DFTB statement has the same label as a previous DFTB statement, the contents of the second DFTB are appended to that of the first. If the same symbol occurs in both, the value associated with the second occurrence replaces that of the first.

▶ label DFVT (expression-1, expression-2)[, ...]

Creates a table of locator values and substitution values. The value table is used in conjunction with the IFVF and IFVT statement

(described later in this section). label is the table name; expression-1 is a locator value and expression-2 is a value to be substituted.

▶ ELSE

Causes the inclusion of statements following the ELSE if the result of a previous IFx statement (any IF with a qualifier appended, such as IFP or IFVT) is false. Statements are included until the matching ENDC statement is reached. A matching ENDC is one that is at the same nesting level as its corresponding IFx statement. ELSE statements that lie within IFx-ENDC pairs nested within the current level 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 assembly listing. You can use FAIL statements to indicate a failure in the logic controlling a block of conditionally assembled code. Embed a FAIL statement between IFx and ELSE statements or between ELSE and ENDC statements; the FAIL message is displayed if the wrong block of code is assembled, indicating a failure in the logic of the IFx condition during assembly.

▶ GO [TO] label

Causes the assembler to skip all subsequent statements until a statement having the specified label is found. Assembly continues at the labeled statement.

The GO statement is typically used in conjunction with a simple or structured IF statement (described below). The GO statement's operand must point forward to the destination label. The destination label must be within the same MAC-ENDM range as the GO statement. An error condition exists if the assembler reaches an END, MAC, or ENDM statement before finding the specified label.

▶ [label] IF logical-expression, statement

This form of IF is known as a simple IF. It conditionally assembles statement based on the result of a test. The label is optional.

The operand consists of a logical expression followed by a statement. If the expression is true, 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. Here is an example of a simple IF statement:

```
VALUE   SET   0
        IF    VALUE.EQ.0, DATA  C'SUCCESS'
VALUE   SET   1
        IF    VALUE.EQ.0, DATA  C'FAILURE'
```

The data constant SUCCESS is generated for the first IF, since VALUE has been set to zero by the SET statement, and the IF tests true. FAILURE is not generated, since VALUE has been set to 1; the second IF therefore tests false and skips the DATA statement.

The six possible logic conditions are:

expression_1.EQ.expression_2	expression_1 equal to expression_2
expression_1.GE.expression_2	expression_1 greater than or equal to expression_2
expression_1.GT.expression_2	expression_1 greater than expression_2
expression_1.LE.expression_2	expression_1 less than or equal to expression_2
expression_1.LT.expression_2	expression_1 less than expression_2
expression_1.NE.expression_2	expression_1 not equal to expression_2

The periods before and after the condition codes are part of the codes, and must be entered as shown. Single spaces can be used between expression_1, the condition code, and expression_2 to improve readability.

▶ [label] IFx logical-expression

This form of IF is known as a structured IF. It requires the structure shown on the following page. The label is optional. The variant x can have one of the values: M, N, P, Z, TF, TT, VF, or VT; the meanings of these values are given after the structure description.

```
[label]   IFx  logical-expression
   .      .        .
   .      .        .

   assemble this code if condition is true
   .      .        .
   .      .        .

          [ELSE
   .      .        .
   .      .        .

   assemble this code if condition is false
   .      .        .
   .      .        .]

          ENDC
   .      .        .
   .      .        .
   continue with non-conditional assembly
   .      .        .
```

The ELSE part of the structure can be omitted if, for the false condition, there is no conditional code to assemble. Assembly then continues with the nonconditional code following the ENDC statement.

For every IFx statement there must be a matching ENDC statement. IFx-ENDC pairs can be nested within each other. The nesting depth count is checked even in sections of code that are being skipped because of a nested IFx-ENDC block.

There are four expression magnitude variants to the IFx statement:

Variant	Meaning
IFM	expression value is minus (< 0)
IFN	expression value is not zero
IFP	expression value is plus (> 0)
IFZ	expression value is zero

Four additional structured IF statements take actions based on the presence or absence of symbols and values in tables defined by DFTB and DFVT statements.

The general form is:

```
label    IFx   symbol      (for IFTF or IFTT)

label    IFx   value       (for IFVF or IFVT)
```

IFTF Search for symbol in the table defined by a DFTB statement whose name is label. (The labels on the DFTB and the IFTF statements must match.) If the symbol is not found, assemble the code up to the matching ELSE or ENDC. If the symbol is found, put its value in assembler attribute #124 and assemble the code following the matching ELSE or ENDC.

IFTT Search for symbol in the table defined by a DFTB statement whose name is label. (The labels on the DFTB and the IFTT statements must match.) If the symbol is found, put its value in assembler attribute #124 and assemble the code up to the matching ELSE or ENDC. If the symbol is not found, assemble the code following the matching ELSE or ENDC.

IFVF Search for a locator value matching value in the table defined by a DFVT statement whose name is label. (The labels on the DFVT and the IFVF statements must match.) If the value is not found, assemble the code up to the matching ELSE or ENDC. If the value is found, put its substitution value in assembler attribute #124 and assemble the code following the matching ELSE or ENDC.

IFVT Search for a locator value matching value in the table defined by a DFVT statement whose name is label. (The labels on the DFVT and the IFVF statements must match.) If the value is found, put its substitution value in assembler attribute #124 and assemble the code up to the matching ELSE or ENDC. If the value is not found, assemble the code following the matching ELSE or ENDC.

In all of the above cases, the assembler can retrieve the value placed in attribute #124 in a (logically, not necessarily physically) later instruction or pseudo-operation by coding #124 as its operand. Thus, the sequence

```
SYMTAB    DFTB    (X,2)
VALTAB    DFVT    (1,5)
   .        .       .
   .        .       .

SYMTAB    IFTT    X
VAL_1     SET     #124
          ENDC
VALTAB    IFVT    1
VAL_2     SET     #124
          ENDC
   .        .       .
   .        .       .
```

sets VAL_1 to 2 as a result of looking up the symbol X in table SYMTAB, and sets VAL_2 to 5 as a result of locating the value 1 in VALTAB and using its substitution value 5.

SYMBOL DEFINING PSEUDO-OPERATIONS (SD)

Labels used to represent addresses are usually defined when they appear in the label field of an instruction or pseudo-operation statement. Symbols so defined are given the procedure-relative, link-relative, or common-relative mode and the value of the location counter at that location. The EQU, SET and XSET statements make it possible to equate symbols to any numeric value, including values that lie outside the range of addresses in a program.

The following symbol-defining pseudo-operations are described in this section.

Name	Function
DYNM	Declare stack-relative data or address constant
EQU	Symbol definition
SET	Symbol definition
XSET	Symbol definition

▶ DYNM

The DYNM pseudo-operation is used to define both the label and the length of a data or address constant; the constants are allocated in a stack frame. Refer to Chapter 8 of the System Architecture Reference Guide for a description of stacks and procedure calls.

One use of DYNM is to provide communication between a calling program and a called program so that arguments can be passed from one to the other. DYNM is used in the procedure segment of the called program to provide space for pointers created by the calling program's CALL pseudo-operation. These pointers accommodate the addresses of the calling program's arguments. The called program then references the arguments indirectly through the pointers, as shown in the example on the following page.

```
*
*     CALLED PROGRAM THAT ADDS TWO NUMBERS
*     AND RETURNS THE SUM
*
            SEG
            SUBR    ADD,ECBADD      IDENTIFIES ROUTINE FOR LINKER
ADD         ARGT                    REQUIRED FOR ARGUMENT PASSING
            LDA     QS,*            LOAD FIRST NUMBER
            ADD     RS,*            ADD SECOND NUMBER
            STA     SUM,*           STORE SUM
            PRTN                    RETURN TO CALLER
            DYNM    QS(3)           POINTER TO FIRST NUMBER
            DYNM    RS(3)           POINTER TO SECOND NUMBER
            DYNM    SUM(3)          POINTER TO SUM
            LINK
ECBADD      ECB     ADD,,QS,3
            END
```

The calling program contains a call sequence such as:

```
            CALL    ADD
            AP      NUM_1,S         FIRST NUMBER
            AP      NUM_2,S         SECOND NUMBER
            AP      TOTAL,SL        SUM
```

in which NUM_1 corresponds positionally to QS in the called program ADD, NUM_2 corresponds to RS, and TOTAL to SUM. The argument names, as can be seen in this example, do not need to be the same; the order of the APs and their corresponding DYNMs, however, must agree, and they must both be contiguous.

The CALL statement initiates the creation of the pointers in the stack frame of the called routine and the ARGT instruction in the called routine completes the process. The called program's ECB operands include the label of the first argument pointer (QS) and the number of arguments expected (3). Refer to the description of the ECB pseudo-operation in Chapter 6 for a fuller discussion of ECBs.

DYNM also identifies and allocates any other temporary space needed for data used wholly within the called program, and only for the duration of the program's execution. This data is dynamic in that PRIMOS allocates space for it when the program is called, and deallocates it when the program returns to its caller.

Each DYNM statement must define the size of the item being allocated. This is the function of the number in parentheses following the data name. Argument pointers are always defined as three halfwords long; other items can be of any length suitable to their purposes. If a length is not specified, it defaults to one halfword (16 bits).

▶ EQU, SET, and XSET

These pseudo-operations equate a label with an absolute number. There are two permissible formats:

Format 1:

$$\text{symbol} \left\{ \begin{matrix} \text{EQU} \\ \text{SET} \\ \text{XSET} \end{matrix} \right\} \text{absolute-expression [,symbol = absolute-expression] ...}$$

Format 2:

$$\left\{ \begin{matrix} \text{EQU} \\ \text{SET} \\ \text{XSET} \end{matrix} \right\} \text{symbol = absolute-expression, ...}$$

In format 1, the symbol in the label field is equated to the absolute expression, which may be any expression that is valid in the current addressing mode. Any symbols used in the expression must already be defined. The label field is required.

Format 1 can include one or more symbol = value expressions, enabling several symbols to be defined by one EQU, SET, or XSET statement.

In format 2, equality expressions in the operand field assign numeric values to symbols. This format allows one or more equality expressions, separated by commas.

EQU, SET, and XSET all perform the same functions; however, a symbol defined by EQU cannot be redefined, while a symbol once defined by SET or XSET can be redefined by subsequent SET or XSET statements without causing an error message.

EQU statements are normally used outside of conditional assembly and macro definition blocks; SET and XSET are useful within these blocks when a label needs to be equated to different values under different conditions. SET, for example, can be used to increment or decrement a counter used by an IF/BACK sequence controlling repeated generation of a block of statements in a macro definition, as shown on the following page.

```
MACNAME   MAC
  .         .
  .         .

COUNT     SET    5              INITIALIZE COUNT
REPT      SET    *
  .         .     .
  .         .     .
    code to be repeated
  .         .     .
  .         .     .

          SET   COUNT=COUNT-1   DECREMENT COUNT
          IFN   COUNT           REPEAT IF COUNT
          BACK  REPT              NOT ZERO
          ENDC
 .         .     .
 .         .     .
          ENDM
```

The only difference between SET and XSET is that symbols defined by XSET do not appear in the assembler's cross reference listing.

LISTING CONTROL PSEUDO-OPERATIONS (LC)

The listing control pseudo-operations are used to control the contents of the assembler listing. They include the statements in the following list.

Name	Function
EJCT	Eject page
LIST	Enable listing
LSMD	List macro expansions
LSTM	List macro expansions with logic control lines
NLSM	Don't list macro expansions
NLST	Inhibit listing
PCVH	Print cross reference values in hexadecimal

▶ EJCT

Causes the listing device to eject the page (execute a form feed), print the current page header 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. Also, if the listing is printed via the SPOOL command, the SPOOL options used may affect the printing of the page header and page number. Refer to the description of the SPOOL command in the PRIMOS Commands Reference Guide for details.

▶ 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 an NLST statement has previously inhibited listing.

▶ LSMD

Lists macro call statements plus macro-generated instructions and data.

▶ LSTM

Lists macro call statements plus all macro-generated lines, including logic control lines such as such as SET, GO, IF, and BACK.

▶ NLSM

Inhibits listing of statements generated by macro expansion. Only the macro call is listed. NLSM is overridden if the -EXPLIST command line option is specified when the assembler is invoked (see Chapter 2). NLSM also suppresses the listing of local variables (those preceded by &) in the assembler's cross reference listing.

▶ NLST

Inhibits listing of all subsequent statements until a LIST statement is encountered. NLST is overridden if the -EXPLIST command line option is specified when invoking the assembler (see Chapter 2).

LIST and NLST may be used together in source text to select sections to be listed. The LSTM, LSMD, and NLSM statements provide control of listing for macro definitions.

▶ PCVH

Prints symbol values in the cross reference in hexadecimal instead of octal.

5
Constant Definition Pseudo-Operations

This chapter describes a group of pseudo-operations that create static data and data areas of various kinds that the machine instructions will use for computation and data storage. This group includes four classes of pseudo-operations that perform address definition (AD), data definition (DD), literal control (LT), and storage allocation (SA) functions.

Two kinds of static data are discussed in this chapter: address constants and data constants. Address constants are used primarily for indirect addressing. (See TYPES OF ADDRESSING in Chapters 8 and 9 of this guide, and the System Architecture Reference Guide for fuller discussions of addressing.) Data constants are numeric or alphanumeric entities that the program uses for computation and display purposes. These are described in detail later in this chapter.

A third kind of data, the literal constant, is not generated by a pseudo-operation, but by being used in the operand field of a machine instruction. Refer to Operand Field in Chapter 3 for a description of literal constant formats.

There are two literal-related pseudo-operations, whose function is to determine where in the assembled program the constants will be stored; that is, whether they will be stored in the procedure segment or in the linkage segment. These are described later in this chapter.

One other group of pseudo-operations reserves a specified number of locations for such things as input and output buffers and temporary storage areas. These do not load anything into the areas; they simply reserve space and assign the label, if any, to the space.

ADDRESS DEFINITION PSEUDO-OPERATIONS (AD)

This group of pseudo-operations creates address constants that statements can use for indirect addressing, subroutine argument passing, and temporary storage of addresses.

Name	Function
AP	Argument pointer template
DAC	Local address definition
IP	Indirect pointer
XAC	External address definition

▶ [label] AP address-expression [,modifier]

The AP pseudo-operation generates a template in the form used by the Procedure Call instruction (PCL) to create indirect pointers for subroutine argument passing via the subroutine's stack frame. address-expression is an argument label, written in any V-mode or I-mode memory reference format except indexed.

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.
```

Indirect argument pointers and argument templates are described in detail in Chapter 8 of the System Architecture Reference Guide.

▶ [label] DAC address-expression

The DAC pseudo-operation generates a 16-bit address constant, containing the address represented by address-expression. An instruction can use the address constant as an indirect pointer to a location within the same segment as the instruction. When used for indirection, the DAC operand must be direct and the referencing instruction must be in short (16-bit) form; that is, its operation code must include a terminating # sign. Its operand can be indirect or indirect indexed.

```
          LDA#    ADCON,*     or
          LDA#    ADCON,*X
           .        .
           .        .

ADCON     DAC     TABLE
TABLE     DEC     0
          DEC     1
           .        .
           .        .
```

The address expression (TABLE in the above example) is the label of a location within the segment. When a DAC defines an indirect pointer, its argument must be direct, since V mode and I mode allow only one level of indirection, and that level is used in the referencing instruction. Refer to the discussion of indirect addressing in Chapter 8 of this guide.

If a DAC simply defines a storage location for an address, that address can be direct or indirect in both V mode and I mode. In V mode only, the address can also be indirect post-indexed by X.

The DAC statement can also provide storage space in a subroutine called by a JST instruction (in V mode only). In this usage it allocates storage for the address to which control is to return after the subroutine completes execution. (Refer to the description of the jump-and-store instructions in Chapter 8.) It must appear immediately before the first executable instruction of the subroutine, in the following form:

```
label DAC **
```

In this case, label is required, and is the label used in the operand field of the JST instruction. This subroutine calling technique is valid only in nonshared segments; that is, the program must be linked by the SEG loader, or, if it is linked by BIND, the procedure segment must be designated as impure. (See the description of the SEG or SEGR pseudo-operation in Chapter 4.)

▶ [label] IP address-expression

The IP pseudo-operation generates a 32-bit V-mode or I-mode indirect pointer containing the address represented by address-expression. It is functionally the same as the DAC in indirect addressing, but it can refer to locations outside the referencing segment. address-expression can be any of the following: procedure relative, linkage relative, common, or external. This means that address-expression can contain a label defined when the assembler's location counter mode is procedure-relative, linkage-relative, or common-relative, or when the

label is defined in an EXT pseudo-operation. The linker supplies the value of the pointer.

IPs, unlike DACs, should not appear in the procedure segment, but in the linkage segment (that is, following a LINK pseudo-operation). Otherwise, an error will result during linking of the assembled program.

▶ [label] XAC symbol

The XAC pseudo-operation generates a 16-bit pointer containing the address of symbol, which is defined in an external module. The symbol name may be the same as a local symbol without conflict. XAC is like DAC except that it references external symbols. The address of the external symbol is supplied by the linker.

Because the pointer is only 16 bits long, the module containing the definition of the external symbol must be linked into the same segment as the referencing module.

DATA DEFINITION PSEUDO-OPERATIONS (DD)

This group of pseudo-operations allocates space for and initializes data constants to known starting values. Data can appear in any program segment. However, it is important to note that no data that the program will modify (by storing into it or performing arithmetic, logical, or character operations on it) should ever appear in a procedure segment, which in V mode and I mode is considered to be a pure segment. No assembler or linker error messages are generated if this rule is violated, but a run-time error (access violation) will occur if the program is linked with BIND and invoked by the RESUME command.

For coding convenience, the assembler accepts a variety of data declaration formats. Simple coding conventions allow the programmer to use decimal, octal, hexadecimal, and binary integers, decimal floating point, and character constants. The assembler interprets the notation and generates one or more data elements in the proper internal binary format.

The following pseudo-operations define data constants.

Name	Function
BCI	Define character constant (space fill)
BCZ	Define character constant (null fill)
DATA	Define numeric or character constant
DEC	Define numeric constant
HEX	Define hexadecimal integer constant
OCT	Define octal integer constant
VFD	Define variable fields

```
[label] BCI 'string'
[label] BCI n,string
[label] BCZ 'string'
```

Loads character strings by packing the specified characters two per halfword, starting with the leftmost 8 bits. Assembled halfwords are loaded starting at the current location, and label is assigned the current location's value.

If the n, is omitted, the length of the string within the delimiter characters determines the number of halfwords to allocate. In a BCI of this form, the string must begin and end with a pair of nonnumeric characters which do not appear in the string itself. A string can be up to 72 characters long.

The single quote is the typical delimiter, but if it appears in the string, then some other character such as a slash (/) or double quote (") can replace it:

```
CONS1   BCI  'PRIME COMPUTER'
CONS2   BCZ  /YOU'RE OUT!/
```

The BCI packs 14 characters into 7 halfwords whose contents are PR, IM, E<space>, CO, MP, UT, and ER. The BCZ packs 11 characters into 6 halfwords, the last of which is padded on the right with a null.

Both BCI and BCZ pad on the right when the string contains an odd number of characters. BCI pads with a space character ('240), while BCZ pads with a null character ('000).

When the n, is included (valid only for BCI), it defines the number of halfwords to allocate. The constant can consist of any number of characters up to twice the value of n. If the constant is fewer than 2 * n characters long, the assembler pads the n unused halfwords on the right with the appropriate number of spaces:

```
CONS3   BCI  3,HI
CONS4   BCI  20,
```

The first BCI loads HI into the first of three halfwords, padding the second and third with spaces. The second BCI loads 20 halfwords with spaces; this technique is useful for reserving a buffer initialized to spaces. Note that when n, is present, no delimiters are used around the string.

▶ [label] DATA constant(s)

The DATA pseudo-operation can be used in a variety of ways to define almost any form of constant or group of constants. Data items are stored starting at the current location, and label is assigned the value of the current location.

Character Constants: The DATA statement can define one or more character constants in one of the following forms:

```
[label] DATA j'string'
[label] DATA n(j'string')
```

where j specifies the justification within the allocated halfwords if the length of string is odd; j is either C or R, for left or right justification, respectively. If C is used, string can be up to 32 characters long; if its length is odd, it is padded on the right with a space. If R is used, string can be only one character; the leftmost byte is null ('000).

n specifies the number of occurrences of the constant to generate. If n is used, the string specification must be enclosed in parentheses.

The only valid delimiter in a string-defining DATA statement is the single quote. Therefore, a DATA statement cannot define a string containing a single quote; use the BCI statement as shown previously.

In both of the above forms, label is assigned the memory location of the first halfword allocated for the string.

A group of character constants (not all the same) can be generated by a DATA statement of the following form:

```
[label] DATA constant-1,constant-2,...
```

where constant-n can assume either of the forms shown in the DATA statement descriptions previously shown. The following is a valid DATA statement:

```
CONS5 DATA C'HI',R'X',3(C'HELLO')
```

The generated constants consist of one occurrence of HI, one occurrence of <null>X, and three occurrences of HELLO<space>.

Integer Constants: The DATA statement can generate decimal, octal, hexadecimal, and binary integer constants in either 16-bit or 32-bit form. The form of the statement is:

```
                          number
[label]   DATA    [k]  +number [L]
                       -number
```

k defines the base of the number to its right: if it is omitted, the base is decimal; a single quote (') indicates octal; a dollar sign ($) indicates hexadecimal; and a percent sign (%) indicates binary. In all cases, the digits following k must be valid for the base denoted by k. The value of number can be in the range -2**15 to +2**15 - 1, inclusive; that is, it must be a number that can be represented by 15 bits plus a sign bit.

When L follows the number, it indicates that a 32-bit (double precision) constant is to be generated. This enables values of number in the range -2**31 to +2**31 - 1, inclusive -- numbers that can be represented by 31 bits plus a sign bit. label is assigned the address of the leftmost halfword.

Whether the L is used or not, no error message is generated if the limiting values are exceeded; the assembler discards the high-order bits.

The characters O, X, and B can be used in place of the symbols described above to designate the base of a constant as octal, hexadecimal, and binary, respectively. When these designations are used, the value specifications must be enclosed in single quotes (see the examples below).

Integer constants are always right-justified in their storage locations. If there are fewer than 16 significant bits in a constant declared with the L option, they are stored in the rightmost halfword, and the leftmost halfword is zero-filled. The partial assembly listing shown below illustrates how integer constants can be generated.

```
000435>       000024   (0012) C08        DATA  20          decimal
000436>       000000   (0013)            DATA  20L
000437>       000024
000440>       177760   (0014)            DATA  O'-20'      octal
000441>       000000   (0015)            DATA  '20L
000442>       000020
000443>       000040   (0016)            DATA  X'20'       hex
000444>       177777   (0017)            DATA  $-20L
000445>       177740
000446>       000064   (0018)            DATA  B'110100'   binary
000447>       177777   (0019)            DATA  %-110100L
000450>       177714
```

Fixed Point Constants and Scaling: Noninteger decimal constants can be specified by using a binary scaling technique. Binary scaling indicates the location of an implied binary point after a specified bit in the binary representation of the decimal constant. It is used in a DATA statement of the following form:

```
[label]   DATA   nBm
```

where n is a decimal number that may include a decimal point, B represents one to four occurrences of the letter B, and m is the scale factor.

The following examples show some data declarations and the resulting octal and binary representations.

Constant	Octal Representation	Bit Pattern
123B15	000173	0000000001111011.
123.5BB15	000173 100000	0000000001111011.1000000000000000
123B7	075400	01111011.00000000
123.5B7	075600	01111011.10000000
0.5B0	040000	0.100000000000000

In these examples, the B indicates that scaling is in effect, and the number following it (the scale factor) specifies the bit position of the implied binary point. The first bit of the binary string is the sign bit.

Single, double, triple, and quadruple precision can be specified by using 1, 2, 3, or 4 Bs, respectively, to generate 16, 32, 48, and 64 bit binary equivalents of the decimal numbers.

The assembler generates an error message if the number of bits to the left of the binary point is not sufficient to contain the integer part of the constant. A scaling factor greater than the number of available bits (for example, B18) results in right-end truncation of the generated constant (123B18 results in 000017 octal) with no assembler error indication. The notation BB18 would supply sufficient bits for the truncated portion to be carried over into the second 16 bits.

Floating Point Constants: The DATA statement is used to generate single (32-bit), double (64-bit), and quad (128-bit) precision floating point constants. The three forms are shown below.

```
[label]   DATA   number[E[-]exp]      single precision
[label]   DATA   numberD[-]exp       double precision
[label]   DATA   numberQ[-]exp       quad precision
```

For single precision floating point numbers, either a decimal point or the E notation must be present. They can also be used together. If the E notation is absent, number must include a decimal point. Whether to use a decimal point when the E notation is present depends on how the number and its exponent are represented. Thus, the single precision floating point integer 123 can be represented as 123., .123E3, 1.23E2, 12.3E1, 123E0, 1230.E-1, and so on. Double and quad precision numbers must always use the D and Q notations, respectively.

For a description of how floating point numbers are represented in the floating point registers and in memory, refer to Chapter 6 of the System Architecture Reference Guide.

▶ [label] DEC numeric-constant[,...]

The DEC pseudo-operation defines numeric constants. It operates in precisely the same way as the DATA statement with numeric operands, and all but one of the numeric formats accepted by the DATA statement can be used with DEC. The exception is the multiple-occurrence form of operand, n(number).

▶ [label] HEX hexadecimal-constant[L][,...]

The HEX pseudo-operation defines hexadecimal integers by converting the hexadecimal representation in the operand to 16-bit integer values. The effect is the same as using a DATA pseudo-operation with a $number operand.

In the absence of the L qualifier, the assembler generates a 16-bit constant whose value can range between -2**15 and +2**15 - 1, inclusive. This corresponds to hexadecimal values between -8000 and +7FFF. If the L qualifier is present, a 32-bit constant is generated. Its value can be between -2**31 and +2**31 - 1 (hexadecimal -80000000 and +7FFFFFFF), inclusive. In either case, no error message is generated if these values are exceeded; the assembler discards the high-order bits.

▶ [label] OCT octal-constant[L][,...]

The OCT pseudo-operations defines octal integers by converting the octal representation in the operand to 16-bit integer values. The effect is the same as using a DATA pseudo-operation with a 'number operand.

In the absence of the L qualifier, the assembler generates a 16-bit constant whose value can range between -2**15 and +2**15 - 1, inclusive. This corresponds to octal values between -100000 and +77777. If the L qualifier is present, a 32-bit constant is generated. Its value can be between -2**31 and +2**31 - 1 (octal -20000000000 and +17777777777), inclusive. In either case, no error message is generated if these values are exceeded; the assembler discards the high-order bits.

▶ [label] VFD size-1,value-1[,size-2,value-2]...

The VFD pseudo-operation permits a 16-bit halfword to be formed with subfields of varying length. In the operand pairs, size-n gives the subfield size in bits, and value-n gives the value. size is expressed as a decimal integer. value can be specified as decimal, octal, hexadecimal, binary, or character, specified as for a DATA statement. Thus, the following is a valid VFD statement:

```
[label]   VFD   3,6,8,R'A',5,'12
```

The first size/value pair represents the most signficant (leftmost) subfield; subsequent size/value pairs load less significant subfields of the 16-bit halfword. For any subfield, if the binary equivalent of its value will not fit into its specified subfield size, the leftmost overflow bits are dropped. No error message is generated. If the entire halfword is not specified, the least significant bits are zero-filled.

An error message results if the sum of the subfield sizes exceeds 16 bits.

LITERAL CONTROL PSEUDO-OPERATIONS (LT)

This group of pseudo-operations governs the placement of literals in the assembled program. See also the description of the END pseudo-operation, described under Assembly Control Pseudo-Operations in Chapter 4.

Name	Function
FIN	Insert literals
RLIT	Optimize literals

▶ [label] FIN

The FIN pseudo-operation 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. label takes the address and mode of this location. Processing of subsequent statements begins at the location following the last literal in the current pool.

By using FIN, you can distribute literals (especially those defined in short form instructions) throughout the procedure segment of a program to keep them within range of their defining instructions, thus reducing the number of out-of-range indirect address pointers that the loader must create to access them. (The direct addressing range of short form instructions is from -224 to +255 locations relative to the instruction's location. The linker generates indirect pointers for addresses outside this range.)

Each time a FIN is encountered, the assembler closes one literal pool and opens a new one. This means that two identically-valued literals defined before a FIN statement are allocated at the same memory address, while two such literals, one defined before and the other after a FIN statement, are allocated different memory addresses.

Be careful, when using the FIN statement, that you end the preceding executable code with a control transfer instruction such as a jump or branch; otherwise the program will attempt to interpret the data in the literal pool as instructions, and will very likely produce a run-time error condition.

▶ RLIT

The RLIT pseudo-operation, if used, must appear immediately after the initial SEG or SEGR statement in a program. In the absence of an RLIT statement, all literals are placed in the linkage segment.

The presence of an RLIT statement causes all literals to be placed in the procedure segment. The FIN statement controls their location within the procedure segment as described in the previous section.

RLIT and FIN statements interact in the following way.

If RLIT is specified and a FIN occurs while in linkage-relative mode, the FIN will act as if the following sequence had been coded (only the FIN is actually generated):

```
PROC        switch to procedure-relative mode
FIN         put literals in the procedure segment
LINK        restore the linkage-relative mode
```

Correspondingly, if RLIT is not specified and a FIN occurs while in procedure-relative mode, the FIN will act as if the following sequence had been coded (only the FIN is actually generated):

```
LINK        switch to linkage-relative mode
FIN         put literals in the linkage segment
PROC        restore the procedure-relative mode
```

STORAGE ALLOCATION PSEUDO-OPERATIONS (SA)

This group of pseudo-operations allocates storage without (except in one case) assigning initial values to the storage locations. They can be thought of as simply reserving a certain amount of space for future data storage. They are typically used to reserve buffer space for input and output operations.

Your program should never rely on the initial contents of uninitialized memory. Any data using uninitialized storage should be put there by the program itself, either by reading into it from an external medium or by executing instructions which explicitly store into it.

The following pseudo-operations constitute the storage allocation statements.

Name	Function
BSS	Allocate block starting with symbol
BES	Allocate block ending with symbol
BSZ	Allocate block and set to zeroes
COMM	FORTRAN (FTN and F77) compatible COMMON

```
          BSS
▶ [label] BES   absolute-expression
          BSZ
```

Each of these pseudo-operations allocates a block of halfwords of the size specified by absolute-expression, starting at the current location count. If there is a label, BSS and BSZ assign it to the first halfword of the block; BES assigns it to the location following the last halfword of the block. BSZ, in addition to allocating space for the block, initializes the block to all zero bits.

Since storage allocated by these statements is almost invariably meant to be written into (that is, modified by the program), the statements should never be coded in a pure procedure segment. They should appear either in a linkage segment or in a procedure segment that is designated as impure (see the description of the SEG or SEGR pseudo-operation in Chapter 4).

▶ [label] COMM symbol [(absolute-expression)]

Defines FORTRAN-compatible named COMMON areas. These areas are allocated by the linker. label assigns a name to the block as a whole, while symbol specifies named variables or arrays within the block. Additional COMM statements with the same block name are treated as continuations of the block. symbol alone reserves a single location; the optional (absolute-expression) reserves a number of locations equal to its value. The loader creates in the linkage segment a 32-bit indirect pointer which points to the common area.

The COMM statement must appear before any statement that generates code, either in the procedure segment or in the linkage segment. It should immediately follow the SEG or SEGR statement, or the RLIT statement, if there is one.

6
Loading and Linking Pseudo-Operations

The pseudo-operations described in this chapter control the actions of the linker during program linking and provide the mechanism by which separately assembled or compiled programs can be identified and called from the current program.

LOADER CONTROL PSEUDO-OPERATIONS (LO)

The loader control pseudo-operations provide control information for the SEG loader or BIND linker. Addressing mode control pseudo-operations (D64V, D32I) control the assembler memory reference instruction processing as well as loader address resolution mode. Mode commands entered during loading set only the loader's current mode, and are overridden by mode control pseudo-operations in the program.

Incompatible instructions such as a 64V-mode instruction in 32I mode are flagged by the assembler. The addressing mode of the program is determined by the SEG or SEGR pseudo-operation, described in Chapter 4.

The DUII, LIR and CENT statements simplify the preparation of library packages that automatically load instruction simulation modules appropriate to the machine in which the code is to be executed.

The following pseudo-operations provide the loader control functions.

Name	Function
CENT	Conditional entry
D64V	Use 64V addressing mode
D32I	Use 32I addressing mode
DUII	Define UII
ELM	Enter loader mode
LIR	Load if required

▶ 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) directs it to load the module. This is true even if the module would have been loaded by virtue of a match between symbol and an unresolved external reference.

Typically, a module containing a CENT statement will be part of a library.

▶ D64V

The D64V pseudo-operation directs the assembler and the linker to use 64V address resolution for the following instructions, even though a SEGR pseudo-operation appears in the program. V-mode assembly continues until a D32I statement is encountered, or to the end of the program.

It is important to emphasize that this pseudo-operation affects only the assembler and the linker; a corresponding machine instruction, E64V, must accompany it to cause the execution mode to switch from I mode to V mode. D64V/E64V sequence is commonly coded in the form

```
D64V:E64V
```

▶ D32I

The D32I pseudo-operation directs the assembler and the loader to use 32I address resolution, even though a SEG pseudo-operation appears in the program. I-mode assembly continues until a D64V statement is encountered, or to the end of the program.

It is important to emphasize that this pseudo-operation affects only the assembler and the linker; a corresponding machine instruction, E32I, must accompany it to cause the execution mode to switch from V mode to I mode. D32I/E32I sequence is commonly coded in the form

```
D32I:E32I
```

▶ DUII absolute-expression-1, absolute-expression-2

The DUII pseudo-operation directs the loading of an unimplemented-instruction (UII) emulation package. absolute expression-1 is a bit mask defining instruction sets that the UII package emulates, and absolute-expression-2 is a bit mask defining hardware instruction sets that must be present to execute the UII package.

Bit number	Meaning
1-9	Must be 0
10	Prime 500
11	Prime 400
12	Undefined
13	Double Precision Floating Point
14	Single Precision Floating Point
15	Prime 300 Only
16	High Speed Arithmetic

▶ ELM

The ELM pseudo-operation directs the loader to generate an enter addressing mode instruction in the current loader addressing mode at the current counter.

▶ LIR absolute-expression

The LIR pseudo-operation 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 cause loading. Bit assignments are the same as for the DUII statement.

PROGRAM LINKING PSEUDO-OPERATIONS (PL)

This group of pseudo-operations governs the interaction between the assembler and the loader in resolving address references between main programs and external subroutines.

Name	Function
CALL	External subroutine call
DYNT	Direct entrance call
ECB	Define entry control block
EXT	Flag external reference
SUBR, ENT	Define entry point
SYML	Allow long (8-character) external names

▶ [label] CALL symbol

The CALL pseudo-operation generates a PCL instruction that transfers control to a location in an external program or subroutine. It combines the functions of the PCL instruction, an EXT pseudo-operation, and an IP pseudo-operation in that it identifies the name given in symbol as an external label which the linker will use to resolve the indirect pointer. Thus, CALL eliminates the need for an explicit EXT statement to identify an external symbol, and for an explicit IP statement to provide an indirect pointer, as would be required if the PCL instruction itself were coded. Figure 12-1 in Chapter 12 shows the differences in calling subroutines by the CALL method and the PCL method.

▶ DYNT entry-point-name

The DYNT pseudo-operation identifies a direct entrance point into a subroutine library. For entry points in Prime-supplied system libraries, this statement is unnecessary; it is used only to define entry points in user-created libraries. Its effect is to store the

entry point name in the program, where it will be resolved into the address of the entry point by the dynamic linking mechanism. Volume I of the Advanced Programmer's Guide describes DYNTs and the dynamic linking mechanism in detail.

▶ [label] ECB entry-point, [link-base], [displacement],
[n-arguments], [stack-size], [keys]

The ECB pseudo-operation generates an entry control block by which calling and called programs can communicate with each other. It must appear in the link frame. label is the name of the ECB, by which the program containing the ECB is called by a CALL statement in another program. If this program is to be called from other than command level, it must also have a SUBR or ENT statement (described later in this section) whose operand is the ECB label.

The operand functions are described below.

Operand	Function
entry-point	Entry point in the procedure segment of the program. It is the label of the first executable instruction in the program.
link-base	This operand is not currently used. The comma following it, however, must be included if more operands follow. The link base defaults to '-400 ('177400).
displacement	Used only if there are arguments to be passed to the this program when it is called. It is the label of an argument pointer defined in this program in the operand field of a DYNM pseudo-operation (described in Chapter 4). If only one argument is to be passed, it is the label of that argument's pointer. If more than one argument is to be passed, it is the label of the first of these arguments' pointers. All of the argument pointers must appear as consecutive arguments in one or more consecutive DYNM statements. (Other DYNM statements can be used to define dynamic storage that is not related to argument passing; these DYNMs must not intervene between those used to pass arguments.)
n-arguments	Number of arguments expected; this number must agree with the number of DYNM statements that define the argument pointers. The default is zero.

stack-size	Initial stack frame size. The default is sum of the sizes specified in all DYNM statements, plus 10 (decimal) words for stack frame overhead.
keys	CPU keys for procedure. The default is 64V addressing mode ('14000) if the SEG pseudo-operation was used, or 32I addressing mode ('10000) if the SEGR pseudo-operation was used. Keys are discussed in detail in Chapter 5 of the System Architecture Reference Guide.

If the default value for any operand is desired, the operand can be omitted, leaving only its trailing comma. Any string of trailing commas can be omitted.

Note

Any program that is invoked from command level (that is, by the SEG or RESUME command) is entered from SEG or RESUME via a PCL instruction; the invoked program must therefore contain a labeled ECB and must have the ECB label as the operand of its END statement. It need not have an ENT or SUBR statement because SEG and RESUME provide a dummy entry point name to a program called in this way.

▶ [label] EXT symbol

Identifies variables defined in external programs. The name appearing in the operand of this statement is flagged as an external reference. Whenever other statements in the main program reference one of these names, a special block of object text is generated that notifies the linker to supply the appropriate address. The assembler fills the address fields with zeros.

An EXT statement is required if calls are made to an external program through a PCL instruction in this program; it is not needed if a call is made through a CALL pseudo-operation, since CALL implicitly performs the processing of an EXT statement in addition to generating the PCL instruction.

Names defined by symbol must be unique in the first 6 characters (8 characters if a SYML pseudo-operation appears in the program) and should not appear as a label within the program.

```
[label]   SUBR   symbol-1[, symbol-2]
[label]   ENT    symbol-1[, symbol-2]
```

The SUBR or ENT pseudo-operation matches an entry point in a called program to the label appearing in the operand field of a CALL, XAC or EXT statement in a calling program. SUBR and ENT are identical in effect.

symbol-1 and symbol-2 supply the names of the entry-point and the ECB of the called program. The details of how this is done and when and how to use the optional symbol-2 are discussed in Chapter 12.

SYML

The SYML pseudo-operation allows the declaration of external names up to eight characters long. In the absence of this statement, external names are limited to six characters (this is a linker restriction).

This statement, if used, must follow SEG or SEGR and precede any generated code.

7
Macro Definition Pseudo-Operations

This chapter describes a group of pseudo-operations used in coding macro definition blocks. A macro definition block consists of a group of statements -- instructions and pseudo-operations -- that can be called repeatedly from anywhere outside the block; they are useful for saving coding time and effort when the same sequence of statements must be used more than once in a program.

The following pseudo-operations are described in this chapter:

Name	Description
ENDM	End a macro definition
MAC	Begin a macro definition
SAY	Print a message
SCT	Select code within a macro
SCTL	Select code from comparison list

See also the descriptions of the conditional assembly pseudo-operations given in Chapter 4. Additional information on the definition and use of macros is given in Chapter 11.

MACRO DEFINITION BLOCK

A macro definition block contains, in addition to ordinary statements, pseudo-operations that are unique to macro definitions; they define the beginning and end of the block, and provide a degree of logic in the inclusion or rejection of subgroups of statements within the definition block. These pseudo-operations are described in this chapter; some other conditional assembly statements, which can be used within or outside of macro definitions, are described in Chapter 4.

A macro definition always has a name; the macro is called by coding that name in the operation field of an assembler statement. A statement having a macro name in its operation field is known as a macro call. If the macro expects arguments, argument values are coded in the operand field of the call. These values are substituted for argument references (strings of the form <number>) wherever they appear in the macro definition block. For example, if it were frequently necessary in a program to transfer one halfword of data from one memory location to another, the following macro definition could be used.

```
TRANSFER   MAC
           LDA <1>
           STA <2>
           ENDM
```

Then, from anywhere else in the program (including from within another macro definition block), the macro can be called by a statement such as

```
           TRANSFER  LOC_1,LOC_2
```

where LOC_1 and LOC_2 are labels on the source and destination data items within the program. The integers enclosed in angle brackets are the argument references. The numbers correspond to the positions of the arguments in the macro call's operand field. During assembly they are replaced by the argument values specified in the call. Thus, in the example, <1> is replaced by LOC_1 and <2> is replaced by LOC_2.

The code for the call and the generated statements would appear in the following form on the assembly listing (if LSTM is in effect -- see Chapter 4).

```
           TRANSFER  LOC_1,LOC_2
           LDA       LOC_1
           STA       LOC_2
```

Optional dummy words and argument identifiers can be used to improve readability and increase flexibility of argument positioning. These are described in Chapter 11.

A macro definition block must appear before any call to that macro. Macro definition blocks can contain calls to other macros, provided the called macro's definition block appears before the call to it. A macro definition cannot, however, contain another macro definition; that is, a MAC pseudo-operation cannot appear between another MAC and an ENDM statement.

MACRO DEFINITION PSEUDO-OPERATIONS (MD)

▶ ENDM

The ENDM pseudo-operation terminates a macro definition. ENDM must be the last statement in a macro definition.

▶ label MAC [dummy-word,...] [argument-identifier,...]

The MAC pseudo-operation begins the definition of the macro. Its name is given in the label field. The name is formed in the same way as the label on an instruction or pseudo-operation. Following the MAC statement are statements that make up the macro definition. The definition ends with an ENDM statement.

▶ [label] SAY ASCII-expression

The SAY pseudo-operation 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. An example of how a SAY message appears in an assembly listing is shown in Figure 7-1. Any argument references appearing within the message are replaced, before the message is generated, by corresponding values given in the macro call.

If a listing device is assigned, SAY statements generate output regardless of the status of the listing options. The PMA invocation command assigns the listing device and some listing options, as described in Chapter 2. Macro listing options are defined by listing control pseudo-operations within a program; these are described in Chapter 4.

▶ [label] SCT absolute-expression

The SCT pseudo-operation assembles selected code groups based on the value of absolute-expression. The expression must be a constant or an expression that can be evaluated as a single-precision number. It can also be an argument reference (<n>). The argument value may be

positive or negative, with a range between -4000 and +4000. This value determines which code groups are asembled.

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 range.

Code Groups: Code under the control of an SCT statement must be in groups delimited by one of four types of marker lines. Marker lines have a percent symbol (%) in column 1, either by itself or followed by a second character. Marker line functions are described below.

Marker	Function
%	Code group delimiter line. Increments code group count. If the count matches the value of the SCT argument, assemble from this marker to the next % marker or to the %/ marker, whichever occurs first.
%1	If any statements in the code group containing this marker were assembled, continue assembly from this marker to the next marker of any kind; then skip to the %/ marker if not already there. %1 markers increment the code group count.
%2	If no statements between the SCT and this marker have been assembled, assemble from this marker to the next marker of any kind; then skip to the %/ marker if not already there. %2 markers increment the code group count.
%/	End of control range for the current SCT.

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. When used in this way, it should be used only as the last code group in an SCT range.

Function of the Expression Value: The value of the absolute expression is essentially a counter pointing to a particular code group within the range of the SCT statement. In the following example there are five code groups, each consisting of one instruction. The first code group is considered code group zero, and begins with the SCT statement; it is not preceded by a % marker.

```
LOAD  MAC
      SCT   <1>
      LDA   LOC_0           code group 0
%
      LDA   LOC_1           code group 1
%
      LDA   LOC_2           code group 2
%
      LDA   LOC_3           code group 3
%
      LDA   LOC_4           code group 4
%2
      SAY   ARGUMENT ERROR IN CALL TO 'LOAD' MACRO
%/
      ENDM
```

A call to the LOAD macro with a numeric value between 0 and 4 as its first argument substitutes that value for the <1> in the SCT statement, and causes the generation of the corresponding code group. An argument value of 5 or greater causes the error message to be displayed. Figure 7-1 shows an assembly using an SCT statement.

A general description of assembler action for various argument values is given below.

Argument Value	Assembler Action
0	Assemble from the SCT statement to the first % marker; then skip to the %/ line.
1	Skip to the first % marker; assemble from there to the second % marker; then skip to the %/ marker.
n	Skip to the nth % marker, if any. Assemble from there to marker n+1; then skip to the %/ marker. If there is no nth % marker, proceed as for -n.
-n	Skip to a %2 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.

▶ [label] SCTL absolute-expression, argument-1, [argument-2,...]

The SCTL pseudo-operation assembles selected code groups. The result of a comparison between absolute-expression and the items in the argument list controls the selection of the code group. absolute-expression and each item in the argument list must be an expression that can be evaluated as a single-precision number. Any of them can be (or contain) an argument reference (<n>).

The argument value may be positive or negative, with a range between -32768 and +32767. This value determines which code group is assembled. Code groups are defined as for the SCT statement, described above.

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.

Expression Comparison: The ordinal position in the argument list of the argument that equals absolute-expression determines which code group is selected.

Expression Comparison	Selection
absolute-expression = argument-1	code group 0
absolute-expression = argument-2	code group 1
absolute-expression = argument-n	code group n-1
no match	same as SCT -n

The SCTL statement functions like the SCT statement, but uses argument values rather than code group numbers to select code groups:

```
LOAD  MAC
      SCTL  <1>, 15, 27, -3, 250, -99
      LDA   LOC_0
%
      LDA   LOC_1
%
      LDA   LOC_2
%
      LDA   LOC_3
%
      LDA   LOC_4
%2
      SAY   ARGUMENT ERROR IN CALL TO 'LOAD' MACRO
%/
      ENDM
```

The macro expects calls whose argument values match those in the argument list of the SCTL statement in order to produce useful code. If a call argument value does not match one of those in the list, the error message is generated and no code is produced.

Assume that this macro is called with the statements

```
LOAD   27
LOAD   -99
LOAD   17
```

For each LOAD call, the number 27, -99, or 17 is substituted for the <1> in the SCTL statement. In the first LOAD call, the equality of the argument value 27 with 27 in the SCTL argument list (argument 2) results in the selection of code group 1; in the second call, equality with argument 5 causes code group 4 to be selected. The third case produces the error message because no match is found between 17 and any of the items in the argument list.

Figure 7-2 shows an assembly listing using an SCTL statement. Note that in the assemblies in both Figures 7-1 and 7-2, the same code groups are selected; the selection method in each, however, is different.

```
    SEG
                        (0001)        SEG
                        (0002) LOAD   MAC
                        (0003)        SCT    <1>
                        (0004)        LDA    LOC_0
                        (0005) %
                        (0006)        LDA    LOC_1
                        (0007) %
                        (0008)        LDA    LOC_2
                        (0009) %
                        (0010)        LDA    LOC_3
                        (0011) %
                        (0012)        LDA    LOC_4
                        (0013) %2
                        (0014)        SAY    ARGUMENT ERROR IN MACRO CALL
                        (0015) %/
                        (0016)        ENDM
                        (0017) *
                        (0018) *
              000000    (0019) ST      EQU    *
                        (0020)         LOAD   1
000000:     02.000400L  (ML01)        LDA    LOC_1
                        (0021)         LOAD   4
000001:     02.000401L  (ML01)        LDA    LOC_4
                        (0022)         LOAD   5
ARGUMENT ERROR IN MACRO CALL
000002:       000611    (0023)         PRTN
                        (0024)         LINK
000400>       000001    (0025) LOC_1   DEC    1
000401>       000004    (0026) LOC_4   DEC    4
000402>       000000    (0027) ECB$    ECB    ST
              000012
              000011
              000000
              177400
              014000

          000422        (0028)         END    ECB$
```

Example of SCT Pseudo-Operation
Figure 7-1

```
    SEG
                        (0001)        SEG
                        (0002) LOAD   MAC
                        (0003)        SCTL  <1>, 15, 27, -3, 250, -99
                        (0004)        LDA   LOC_0
                        (0005) %
                        (0006)        LDA   LOC_1
                        (0007) %
                        (0008)        LDA   LOC_2
                        (0009) %
                        (0010)        LDA   LOC_3
                        (0011) %
                        (0012)        LDA   LOC_4
                        (0013) %2
                        (0014)        SAY   ARGUMENT ERROR IN MACRO CALL
                        (0015) %/
                        (0016)        ENDM
                        (0017) *
                        (0018) *
               000000   (0019) ST      EQU   *
                        (0020)         LOAD  27
000000:      02.000400L (ML01)         LDA   LOC_1
                        (0021)         LOAD  -99
000001:      02.000401L (ML01)         LDA   LOC_4
                        (0022)         LOAD  17
ARGUMENT ERROR IN MACRO CALL
000002:         000611  (0023)         PRTN
                        (0024)         LINK
000400>         000033  (0025) LOC_1   DEC   27
000401>         177635  (0026) LOC_4   DEC   -99
000402>         000000  (0027) ECB$    ECB   ST
                000012
                000011
                000000
                177400
                014000

           000422       (0028)         END   ECB$
```

Example of SCTL Pseudo-Operation
Figure 7-2

8
Machine Instructions -- V Mode

This chapter describes the set of machine instructions that is available to you when your program is executing in 64V addressing mode (commonly referred to simply as V mode). It also describes the various types of addressing usable in V mode, as well as the registers available to V-mode programs.

TYPES OF ADDRESSING

V-mode programs use both short and long form instructions. Short form (16-bit) instructions can address the first 256 halfwords of both the stack and link segments, as well as 224 halfwords before and 255 halfwords after the current location in the procedure segment. Direct long form (32-bit) instructions can address all locations in any 128-Kbyte segment pointed to by one of the four base registers; indirect long form (32-bit) instructions can address all locations in up to 4096 128-Kbyte segments.

V-mode addresses at execution time (known as effective addresses) are virtual addresses consisting of a segment number and an offset. The offset identifies a particular halfword relative to the beginning of the segment. Segment numbers and offsets are commonly represented (in listings such as memory maps produced by linkers) as two octal numbers separated by a slash:

```
4001/1025
```

The offset of the first halfword in any segment is 0; therefore the address shown above represents the (octal) 1026th halfword in segment (octal) 4001. Segmented addressing is fully described in the System Architecture Reference Guide.

Segment numbers (at least in their resolved form) do not appear in an assembly listing; generally only the offsets appear, with a code indicating the type of segment in which they will eventually reside. It is the linking operation performed by SEG or BIND that (among other things) establishes segment numbers based on whether the data or instruction in question is defined in a procedure, stack, or linkage segment of the program. For information on program linking and how segments are allocated, refer to the Seg and Load Reference Guide and the Programmer's Guide to BIND and EPFs.

The assembler keeps track of segment information so that, when you use a label as an operand in an instruction, you do not need to concern yourself with specifying the segment in which that label was defined.

The processor hardware uses various components of an instruction to decide which of four addressing forms to use in the instruction's execution:

- Direct
- Indexed
- Indirect
- Indirect indexed

These addressing forms are summarized below. Detailed information on addressing can be found in Chapter 3 of the System Architecture Reference Guide.

Direct Addresses

A direct address is indicated by an operand of the form

```
LABEL1 [ +|- LABEL2 ...  +|- LABELn ]
```

The operand is an expression consisting of the simple symbolic name of an element defined somewhere within the referencing segment, or a combination of such names which, when combined arithmetically, yield a single numeric displacement relative to the beginning, and within the limits, of the segment. (Expressions and their evaluations are described under TERMS AND EXPRESSIONS, in Chapter 3.) In certain cases, you can also use absolute numbers or other literal values; the assembler resolves them to displacements. The hardware, at execution time, adds the displacement to the contents of the appropriate base register (procedure, link, stack, or auxiliary) to form the effective address.

Indexed Address

An indexed address is indicated by an operand of the form

```
LABEL,X
```

Indexing provides a convenient way to address successive elements in a table or to count iterations in a loop.

Here, LABEL is normally a single symbolic name whose displacement value is determined as in direct addressing. The processor then adds this displacement to the current contents of an index register (in this case the X register), whose value must have been previously set by the program. V-mode instructions can also use a second index register, the Y register. (V-mode register usage is described later in this chapter.) The result of the addition is the effective address.

Indirect Address

An indirect address is indicated by an operand of the form

```
LABEL,*
```

Indirect addresses enable a program to make a memory reference to a location through an indirect pointer. A primary use of indirect addressing is to refer to locations outside of the segment making the reference. Prime processors pass arguments to subroutines by address rather than by value. Therefore, subroutines (which frequently reside outside their callers' segments and cannot access or be accessed by their callers using direct addressing) must use indirect pointers to communicate with their callers.

In an indirect address, LABEL is a single symbolic name whose displacement value is determined as in direct addressing. However, rather than using the resulting address directly as the target of the operation, the processor interprets it as the address of a pointer which, in turn, contains the address of the target. Only one level of indirection is allowed in V-mode programs; that is, the contents of the pointer cannot itself be an indirect address.

Since an indirect pointer is a 32-bit address, it can contain segment as well as displacement information, and can thus address across segment boundaries, which addresses that contain only displacement information cannot do. Indirect pointers can also be 48 bit entities, and are used when it is necessary to address specific bits within the target.

Indirect operands normally cause the assembler to generate 32-bit (long form) instructions, which expect 32-bit pointers, declared by using the IP pseudo-operation. For some instructions, if the target is within the same segment as the instruction (and can therefore be reached with a short form instruction), you can generate a 16-bit indirect pointer

and force the generation of a matching short form instruction. You do this by declaring the pointer using a DAC pseudo-operation and appending a "#" to the operation code of the instruction:

Long form indirection			Short form indirection		
	JMP	LABEL1,*		JMP#	LABEL2,*
	.	.		.	.
	.	.		.	.
	.	.		.	.
LABEL1	IP	TARG_1	LABEL2	DAC	TARG_2

The object represented by TARG_1 can be in any segment (including the local segment); the object represented by TARG_2 must be within the local segment. Furthermore, to be accessible to the short-form JMP# instruction, LABEL2 must be within -224 and +255 halfwords of the JMP# instruction.

The instruction summaries later in this chapter indicate which instructions have short forms.

Indirect Indexed Address

Indirect indexed addresses take one of four forms, depending on whether you use the X register or the Y register, and whether indexing is performed before or after indirection:

LABEL,X*	pre-indexed by X
LABEL,*X	post-indexed by X
LABEL,Y*	pre-indexed by Y
LABEL,*Y	post-indexed by Y

In pre-indexing, the processor adds the contents of the index register to the value of LABEL, and uses the result as an indirect address to form the effective address. In post-indexing, the processor resolves the indirect address of LABEL, then adds the index register contents to it to form the effective address.

Each form of indirect indexed addressing has its usefulness in particular situations. Pre-indexing is useful in stepping through a table of addresses, as in the first of the following examples. Post-indexing is used primarily in stepping through a table of data, as in the second example.

Example 1 (pre-indexed indirect):

```
            SEG
START       LDX   =-8
LOOP        LDA   IPTAB+8,X*    pre-indexed through IP table to data
            STA   NUMBER
            STX   SAVEX
            CALL  TODEC
            AP    NUMBER,SL
            CALL  TONL
            LDX   SAVEX
            BIX   ADD2
ADD2        BIX   LOOP
            PRTN
*
            LINK
IPTAB       IP    TEST1
            IP    TEST2
            IP    TEST3
            IP    TEST4
NUMBER      BSS   1
SAVEX       BSS   1
TEST3       DATA  11
TEST1       DATA  22
TEST2       DATA  33
TEST4       DATA  44
ECB$        ECB   START
            END   ECB$
```

Example 2 (post-indexed indirect):

```
            SEG
            RLIT
START       LDX    =5
LOOP        LDA    TABLE,*X    post-indexed to table of data
            STA    NUMBER
            STX    SAVEX
            CALL   TODEC
            AP     NUMBER,SL
            CALL   TONL
            LDX    SAVEX
            BDX    LOOP
            PRTN
*
            LINK
TABLE       IP     TAB-1
NUMBER      BSS    1
SAVEX       BSS    1
TAB         DATA   1, 3, 5, 7, 9
ECB$        ECB    START
            END    ECB$
```

Both of these programs print out a list of data items, one per line. The first accesses the items through a table of indirect pointers (IPs) which in turn contain the addresses of the data items; the second accesses the items in the data table itself. In both cases the tables consist of equal-length entries: in the first they are 32-bit pointers, and in the second they are 16-bit constants. This accounts for the difference in the number of increments (BIX) or decrements (BDX) to the X register in each case (two in the first example and one in the second).

The CALL statements generate procedure call (PCL) instructions, which may destroy the contents of the X register (among others) if argument passing is required (as it is for the calls to the TODEC subroutine). The register must therefore be saved (STX) before the calls and restored (LDX) after the calls.

REGISTER USAGE

The following registers are available to V-mode programs; their sizes are given in bits.

Register	Size	Function
A	16	Accumulator (high half of L register)
B	16	Extension to A Register (Low half of L register)
L	32	Concatenated A and B registers; operations involving L overlay contents of A and B, and vice versa
E	32	Extension to L register; used in double precision multiply and divide operations
X	16	Index register
Y	16	Index register
FAR0	32	Field address register 0
FLR0	32	Field length register 0
FAR1	32	Field address register 1
FLR1	32	Field Length register 1
FAC	64	Floating point accumulator (single precision); concatenation of FAR1 and FLR1
DAC	64	Floating point accumulator (double precision); concatenation of FAR1 and FLR1
QAC	128	Floating point accumulator (quad precision); concatenation of FAR0, FLR0, FAR1 and FLR1
PB	32	Procedure base register
SB	32	Stack base register
LB	32	Link base register
XB	32	Auxiliary base register

The registers listed above are available to all V-mode programs, executing in both ring 0 and ring 3. These, as well as others available only to ring 0 programs, are described in the discussion of user register files in Chapter 9 of the System Architecture Reference Guide. If your program uses FAC, pay particular attention to the discussion of overlap between floating point and field registers in that chapter.

Saving and Restoring Registers

If your program uses PCL instructions (generated by CALL statements), bear in mind that PCL may destroy the contents of the X, Y, and XB registers. The code that is executed as a result of the CALL may use other registers (such as A, B, E, FAC, L, and Y). It is therefore prudent to provide for saving the contents of any registers whose contents are established before the CALL and are required after the call. You can do this easily by use of the register save (RSAV) and register restore (RRST) instructions; refer to the System Architecture Reference Guide for more information on saving and restoring registers.

Register Usage Between V Mode and I Mode

You should be aware that when your program switches from V mode to I mode (as it might when calling a subroutine), there is a correspondence between registers in the user register set in the two modes. For example, the V-mode L register corresponds to I-mode general register 2 (GR2); the V-mode A and B registers correspond to the high and low halves, respectively, of I-mode register GR2. Therefore, if your V-mode program calls an I-mode subroutine and expects a result to be available in the A register upon return from the subroutine, the subroutine should load the return value in the high half of GR2 before returning to its caller.

A complete list of user registers and their intermode correspondences is given in the discussion of user register files in Chapter 9 of the System Architecture Reference Guide.

THE V-MODE INSTRUCTION SET

This section briefly describes the instructions that you can use in V-mode programs. Complete descriptions of all V-mode instructions can be found, listed alphabetically, in Chapter 2 of the Instructions Sets Guide.

V-mode instructions can be arranged in the following groups:

- Generic (accumulator, shift, and skip)
- Branch and jump
- Memory reference
- Decimal
- Floating point
- Character
- Process-related operations
- Restricted operations

Generic Instructions

The generic instruction group comprises three subgroups: accumulator generic, shift generic, and skip generic. All are short form (16-bit) instructions.

Accumulator Generic Instructions: This subgroup contains chiefly instructions that manipulate the contents of the accumulator in various ways and under various conditions. The accumulator is (rather loosely) equivalent to the A register, and the terms are often used interchangeably. To be strictly accurate, however, the A register should be thought of as the high (leftmost) half of the L register; any operation on the A register therefore affects the L register, and vice versa. Operations on the B register, which is the low (rightmost) half of the L register, also affect the L register, and vice versa. The E register, referred to in several of the instructions described below, is a 32-bit extension of the L register, and is used in double precision (long) integer multiply and divide operations. The Keys register contains information related to the current addressing mode, the results of arithmetic and compare operations, overflow conditions, and other indicators.

You will note, in the summaries below, that there are a number of instructions that operate on the A register, but have no equivalents that operate on the B register. You cannot, for example, complement

the B register directly. If you need to perform such an operation, use a sequence such as the following:

```
IAB         Interchange A and B
CMA         Complement A
IAB         Interchange A and B
```

The following functional groups summarize the accumulator generic instructions. Instructions that perform more than one function (for example transfer and clear) may appear in more than one group.

ACCUMULATOR INCREMENT/DECREMENT OPERATIONS

```
A1A         Add 1 to A
A2A         Add 2 to A
S1A         Subtract 1 from A
S2A         Subtract 2 from A
```

ACCUMULATOR CLEAR OPERATIONS

```
CRA         Clear A to zero
CRB         Clear B to zero
CRE         Clear E to zero
CRL         Clear L to zero
CRLE        Clear L and E to zero
XCA         Exchange A and B;  clear A to zero
XCB         Exchange A and B;  clear B to zero
```

ACCUMULATOR TRANSFER OPERATIONS

```
IAB         Interchange A and B
ILE         Interchange L and E
TAB         Transfer A to B
TAK         Transfer A to Keys register
TAX         Transfer A to X
TAY         Transfer A to Y
TBA         Transfer B to A
TKA         Transfer Keys register to A
TXA         Transfer X to A
TYA         Transfer Y to A
XCA         Exchange A and B;  clear A to zero
XCB         Exchange A and B;  clear B to zero
```

ACCUMULATOR BYTE OPERATIONS

CAL	Clear left byte of A to zero
CAR	Clear right byte of A to zero
ICA	Interchange bytes of A
ICL	Interchange bytes of A; clear left byte to zero
ICR	Interchange bytes of A; clear right byte to zero

ACCUMULATOR COMPLEMENT OPERATIONS

CMA	One's-complement A
TCA	Two's-complement A
TCL	Two's-complement L

ACCUMULATOR SIGN OPERATIONS

CHS	Change sign (complement bit 1) of A
CSA	Copy sign (bit 1) of A to C bit of Keys register; clear bit 1 of A to zero
SSM	Set sign (bit 1) of A to 1
SSP	Set sign (bit 1) of A to 0

OPERATIONS WITH KEYS REGISTER

ACA	Add C bit of Keys register to bit 16 of A
ADLL	Add L bit of Keys register to bit 32 of L
CSA	Copy sign (bit 1) of A to C bit of Keys register; clear bit 1 of A to zero
RCB	Reset C bit of Keys register to 0
SCB	Set C bit of Keys register to 1
TAK	Transfer A to Keys register
TKA	Transfer Keys register to A

LOGIC OPERATIONS

LF	Set A to "false" (0)
LT	Set A to "true" (1)

CONDITION CODE TEST AND SET

LCEQ	Set A to 1 if condition code EQ; else set A to 0
LCGE	Set A to 1 if condition code GE; else set A to 0
LCGT	Set A to 1 if condition code GT; else set A to 0
LCLE	Set A to 1 if condition code LE; else set A to 0
LCLT	Set A to 1 if condition code LT; else set A to 0
LCNE	Set A to 1 if condition code NE; else set A to 0

A REGISTER VALUE TEST AND SET

LEQ	Set A to 1 if A = 0; else set A to 0
LGE	Set A to 1 if A >= 0; else set A to 0
LGT	Set A to 1 if A > 0; else set A to 0
LLE	Set A to 1 if A <= 0; else set A to 0
LLT	Set A to 1 if A < 0; else set A to 0
LNE	Set A to 1 if A <> 0; else set A to 0

L REGISTER VALUE TEST AND SET

LLEQ	Set A to 1 if L = 0; else set A to 0
LLGE	Set A to 1 if L >= 0; else set A to 0
LLGT	Set A to 1 if L > 0; else set A to 0
LLLE	Set A to 1 if L <= 0; else set A to 0
LLLT	Set A to 1 if L < 0; else set A to 0
LLNE	Set A to 1 if L <> 0; else set A to 0

FLOATING ACCUMULATOR VALUE TEST AND SET

LFEQ	Set A to 1 if FAC = 0; else set A to 0
LFGE	Set A to 1 if FAC >= 0; else set A to 0
LFGT	Set A to 1 if FAC > 0; else set A to 0
LFLE	Set A to 1 if FAC <= 0; else set A to 0
LFLT	Set A to 1 if FAC < 0; else set A to 0
LFNE	Set A to 1 if FAC <> 0; else set A to 0

Shift Generic Instructions: This subgroup contains instructions that shift the contents of the A or L register a specified number of bits leftward or rightward. There are three types of shifts: logical, rotate, and arithmetic.

Logical shifts move the register contents a specified number of bits in a specified direction, storing each bit shifted out in the C bit and LINK (bits 1 and 3) of the Keys register. Since only one bit is available for storage, only the last bit shifted is available for use in subsequent operations; all intervening bits are lost. Leftward shifts are zero-filled on the right; rightward shifts are zero-filled on the left.

Rotate shifts move the register contents a specified number of bits in a specified direction, storing each bit shifted out in the C bit and LINK (bits 1 and 3) of the Keys register, and also copying the bit to the opposite end of the register. Thus, in a leftward shift, each bit shifted out is reproduced at the right end of the register; in a rightward shift, each bit shifted out is reproduced at the left end of the register. Only the last bit shifted is stored in the Keys register.

Arithmetic shifts move the register contents a specified number of bits in a specified direction.

For a leftward arithmetic shift, bits shifted out are lost, and bits on the right end are zero-filled. The C bit of the Keys register is initially set to zero. If the shift causes a sign change (bit 1 of the register changes from 0 to 1 or from 1 to 0), the C bit is set to 1.

For a rightward arithmetic shift, each bit shifted out is stored in the C bit and LINK (bits 1 and 3) of the Keys register. The sign bit is propagated to the right. That is, if the original value of bit 1 is zero, the left-end bits are zero-filled; if the original value of bit 1 is one, the left-end bits are one-filled.

The following functional groups summarize the shift generic instructions.

LOGICAL SHIFT OPERATIONS

ALL n	Logical shift A left n bits
ARL n	Logical shift A right n bits
LLL n	Logical shift L left n bits
LRL n	Logical shift L right n bits

ROTATE SHIFT OPERATIONS

ALR n	Rotate shift A left n bits
ARR n	Rotate shift A right n bits
LLR n	Rotate shift L left n bits
LRR n	Rotate shift L right n bits

ARITHMETIC SHIFT OPERATIONS

ALS n	Arithmetic shift A left n bits
ARS n	Arithmetic shift A right n bits
LLS n	Arithmetic shift L left n bits
LRS n	Arithmetic shift L right n bits

Skip Generic Instructions: Skip instructions (and branch and jump instructions, described later in this chapter) alter the normal sequential flow of control in a program. Skip instructions test some condition and skip if the tested condition is true.

Most skip instructions test for a true or false condition and skip zero or one halfword, but some can skip zero, one, or two halfwords, based on the result of a test for a greater-than, equal, or less-than condition. The latter are typically followed by two short-form jump (JMP#) instructions and a third instruction of any type. The two jump instructions transfer control to code that handles two of the tested conditions; the third instruction handles the remaining condition. (Refer to the description of the JMP instruction, below, for valid destinations of short form jumps.) The following example illustrates the technique.

```
CAZ              Compare A register to zero
JMP#   addr_1      Process A > 0 condition
JMP#   addr_2      Process A = 0 condition
<inst>             Process A < 0 condition
  .      .
  .      .
```

It is most important to remember, when using skip instructions, that the instructions skip halfwords (16 bits), not instructions. Therefore only short-form instructions (or instructions that can be forced to short form, such as the JMP# instructions illustrated above) should follow any of the skip generic instructions.

The following functional groups summarize the skip generic instructions.

TEST ACCUMULATOR AND SKIP 1 HALFWORD

SAR n	Skip if bit n of A = 0; n is a number from 1 through 16
SAS n	Skip if bit n of A = 1; n is a number from 1 through 16
SGT	Skip if A > 0
SKP	Generic skip (see Instruction Sets Guide)
SLE	Skip if A <= 0
SLN	Skip if bit 16 of A = 1
SLZ	Skip if bit 16 of A = 0
SMI	Skip if A < 0 (bit 1 = 1)
SNZ	Skip if A <> 0
SPL	Skip if A >= 0 (bit 1 = 0)
SZE	Skip if A = 0

COMPARE ACCUMULATOR AND SKIP 0, 1, OR 2 HALFWORDS

CAZ	Compare A to zero and skip: 0 halfwords if A > 0 1 halfword if A = 0 2 halfwords if A < 0

INCREMENT/DECREMENT X REGISTER AND SKIP 1 HALFWORD

IRX	Add 1 to X; skip if X = 0
DRX	Subtract 1 from X; skip if X = 0

TEST C BIT OF KEYS REGISTER AND SKIP 1 HALFWORD

SRC	Skip if C bit = 0
SSC	Skip if C bit = 1

Branch Instructions

Branch instructions, like the skip instructions just described, alter the normal sequential flow of control in a program. Branch instructions can test the following conditions and transfer control to a specifed address if the tested condition is true:

- Contents of the accumulator (A register) with respect to zero
- Contents of the L register with respect to zero
- Contents of the floating accumulator (FAC) with respect to zero
- State of the condition codes (CC)
- State of the C bit of the Keys register
- State of the L bit of the Keys register and condition codes (magnitude)
- Contents of the X or Y register with respect to zero after incrementing or decrementing

Branch instructions can accept only direct addresses. Therefore, they can transfer control only to locations within the same segment as themselves. Jump instructions, described later in this chapter, can transfer control to other segments through indexing or indirection (as well as to their own segments through direct addressing).

The following functional groups summarize the branch instructions. All are long form instructions.

BRANCH ON ACCUMULATOR WITH RESPECT TO ZERO

BEQ addr	Branch to addr if A = 0
BGE addr	Branch to addr if A >= 0
BGT addr	Branch to addr if A > 0
BLE addr	Branch to addr if A <= 0
BLT addr	Branch to addr if A < 0
BNE addr	Branch to addr if A <> 0

BRANCH ON L REGISTER WITH RESPECT TO ZERO

BLEQ addr	Branch to addr if L = 0
BLGE addr	Branch to addr if L >= 0
BLGT addr	Branch to addr if L > 0
BLLE addr	Branch to addr if L <= 0
BLLT addr	Branch to addr if L < 0
BLNE addr	Branch to addr if L <> 0

BRANCH ON FLOATING ACCUMULATOR WITH RESPECT TO ZERO

```
BFEQ addr    Branch to addr if FAC = 0
BFGE addr    Branch to addr if FAC >= 0
BFGT addr    Branch to addr if FAC > 0
BFLE addr    Branch to addr if FAC <= 0
BFLT addr    Branch to addr if FAC < 0
BFNE addr    Branch to addr if FAC <> 0
```

BRANCH ON CONDITION CODE IN KEYS REGISTER

```
BCEQ addr    Branch to addr if EQ bit = 1
BCGE addr    Branch to addr if LT bit = 0 or EQ bit = 1
BCGT addr    Branch to addr if LT bit = 0 and EQ bit = 0
BCLE addr    Branch to addr if LT bit = 1 or EQ bit = 1
BCLT addr    Branch to addr if LT bit = 1 and EQ bit = 0
BCNE addr    Branch to addr if EQ bit = 0
```

BRANCH ON MAGNITUDE CONDITION

```
BMEQ addr    Branch to addr if EQ bit = 1 (same as BCEQ)
BMGE addr    Branch to addr if L bit = 1 (same as BLS)
BMGT addr    Branch to addr if L bit = 1 and EQ bit = 0
BMLE addr    Branch to addr if L bit = 0 and EQ bit = 1
BMLT addr    Branch to addr if L bit = 0 (same as BLR)
BMNE addr    Branch to addr if EQ bit = 0 (same as BCNE)
```

BRANCH ON STATE OF C/L BIT OF KEYS REGISTER

```
BCR addr     Branch to addr if C bit = 0
BCS addr     Branch to addr if C bit = 1
BLR addr     Branch to addr if L bit = 0
BLS addr     Branch to addr if L bit = 1
```

BRANCH ON X/Y REGISTER AFTER INCREMENT/DECREMENT

```
BDX addr     Decrement X by 1;  branch to addr if not zero
BDY addr     Decrement Y by 1;  branch to addr if not zero
BIX addr     Increment X by 1;  branch to addr if not zero
BIY addr     Increment Y by 1;  branch to addr if not zero
```

Computed Go To Instruction

The computed go to (CGT) instruction is a multi-directional form of branch instruction, capable of transferring control to any of several destination addresses in the same segment as the CGT instruction itself, depending on a preset value in the A register. It is functionally identical to a FORTRAN computed GO TO statement. A typical sequence is:

```
LDA   destination_number
CGT
DATA  number_of_destinations
DAC   destination_1
DAC   destination_2
 .       .       .
 .       .       .
DAC   destination_n
instruction
```

Let n be the number of addresses at which processing can continue. In this sequence, the LDA instruction (or any other instruction or series of instructions that establish a value in the A register) loads the A register with a destination_number between 1 and n, inclusive, depending on which of the destinations control is to be transferred to. The DATA statement defines the number of valid destinations, plus 1 to account for invalid A register settings. Thus, if there are 4 valid addresses at which processing can continue, the value of the number_of_destinations operand is 5. If the value in the A register is, say, 2, then the CGT instruction transfers control to destination_2. An invalid value in the A register (a value less than 1 or greater than number_of_destinations - 1) transfers control to instruction to perform error processing.

The destination addresses must be in the local segment, and must be specified as short (16-bit) addresses; hence the use of a DAC pseudo-operation. Refer to Chapter 5 for a description of the DAC pseudo-operation.

Instruction can be any statement that generates a machine instruction. For example, it could be a CALL statement to a subroutine that prints an error message and then exits the program.

Jump Instructions

V-mode programs can use any of five jump instructions to alter the normal sequence of control. They are all unconditional transfers; they are independent of any previous test conditions. Four of them are jump-and-store instructions, normally used to transfer control to subroutines from which control is expected to return to the code following the jump instruction. (Refer to Chapter 12 for a discussion

of subroutine calling.) The remaining one is a one-way jump to code from which no return is expected.

One-way Jump Instruction: A one-way jump instruction is used whenever a control transfer without a subsequent return is required. One of its typical uses is to return from a subroutine to which control has been transferred by one of the four jump-and-store instructions.

The assembler generates a jump instruction in either short or long form, depending on its operand. The short form results for any of the following cases:

```
JMP   LABEL      (to LABEL, in the current segment)
JMP   *+nn       (nn halfwords after current program counter)
JMP   *-nn       (nn halfwords before current program counter)
JMP   LABEL,X    (to LABEL, in the current segment, indexed by X)
JMP   PB%,X      (to address in PB register, indexed by X)
```

JMP with any other form of operand (that is, indirect, indirect indexed, indexed by Y, or any form of XB-relative or SB-relative) generates the long form.

With some operands, jump instructions that are normally generated in long form can be forced to short form by using the JMP# form of the mnemonic operation code. These forced short-form jumps, together with those itemized above, are the ones that should be used after a three-way skip instruction such as CAZ or CAS. Short-form jumps can be forced with the following operands (LABEL must be within -224 and +255 halfwords of the JMP# instruction).

```
JMP#  LABEL,*    (LABEL, in the current segment, indirect)
JMP#  LABEL,*X   (LABEL, indirect, post-indexed by X)
JMP#  LABEL,X*   (LABEL, indirect, pre-indexed by X)
JMP#  PB%,*      (procedure base, indirect)
JMP#  PB%,X      (procedure base, indexed by X)
JMP#  PB%,*X     (procedure base, post-indexed by X)
JMP#  PB%,X*     (procedure base, pre-indexed by X)
```

Jump-and-Store Instructions: Each jump-and-store instruction stores a return address in a specific place (see the descriptions below); the code to which control is transferred must therefore know which form of jump was used to transfer to it so that it can make the appropriate return. The four jump-and-store instructions and their storage and return mechanisms are summarized below. (Further details are given in the Instruction Sets Guide.)

JST and JSY are generated as either short or long form, depending on their operands. They can, for some operands, be forced to short form by appending "#" to the operation code (see the description of the JMP instruction, above).

JST[#] addr — Store the address following the JST in addr. Jump to addr + 1. Return by JMP addr,*.

JST can jump only to a location in the local segment. Since it modifies addr, a location within the segment, it cannot be used in a shared (pure code) segment.

JSX addr — Store the address following the JSX in the X register. Jump to addr. Return by JMP PB%,X.

JSX can jump only to a location in the local segment. It can be used in shared segments, since it does not modify a location in the segment. addr cannot be indexed.

JSXB addr — Store the address following the JSXB in the auxiliary base (XB) register. Jump to addr. Return by JMP XB%.

JSXB can jump to nonlocal segments as well as to the local segment. It can be used in shared segments, since it does not modify a location in the segment.

JSY[#] addr — Store the address following the JSY in the Y register. Jump to addr. Return by JMP PB%,Y.

JSY can jump only to a location in the local segment. It can be used in shared segments, since it does not modify a location in the segment.

Memory Reference Instructions

This section describes a group of instructions that can refer to or modify the contents of memory locations; that is, they can read from and write to memory.

Because the memory reference instruction group comprises a large number of instructions, their descriptions are divided into several subgroups:

- Memory/register transfer operations
- Memory/register logic operations
- Memory test and skip operations
- Integer operations
- Decimal operations
- Floating point operations
- Character and field operations

All memory reference instructions described in this section are by default long (32-bit) instructions. Some, however, can be forced to short (16-bit) form by using the "#" designator described in the previous section for JMP instructions. These are indicated in the instruction lists in this section by "[#]" following the mnemonic operation code.

Memory/Register Transfer Operations: The instructions in this group transfer the contents of a memory location to a register, or transfer the contents of a register to a memory location. Also included here are instructions that calculate an effective address and load it into a register.

MEMORY/REGISTER TRANSFER OPERATIONS

EAL addr	Load effective address into L register
EALB addr	Load effective address into link base register
EAXB addr	Load effective address into auxiliary base register
IMA[#] addr	Interchange memory and A register
LDA[#] addr	Load 16 bits from memory into A register
LDL addr	Load 32 bits from memory into L register
LDLR addr	Load from addressed register into L register (see Instruction Sets Guide)
LDX[#] addr	Load 16 bits from memory into index register X; addr cannot be indexed
LDY addr	Load 16 bits from memory into index register Y; addr cannot be indexed
RRST addr	Restore user registers (see Instruction Sets Guide)
RSAV addr	Save user registers (see Instruction Sets Guide)
STA[#] addr	Store 16 bits from A register into memory
STAC addr	If contents of B register equals contents of addr, store 16 bits of A register into addr; (addr is generated as a 32-bit address pointer)
STL addr	Store 32 bits from L register into memory
STLC addr	If contents of E register equals contents of addr, store 32 bits of L register into addr; (addr is generated as a 32-bit address pointer)
STLR addr	Store L into addressed register (see Instruction Sets Guide)
STX[#] addr	Store index register X into memory; addr cannot be indexed
STY addr	Store index register Y into memory; addr cannot be indexed

Memory/Register Logic Operations: The instructions in this group perform logic operations on a register based on the contents of a memory address.

MEMORY/REGISTER LOGIC OPERATIONS

ANA[#] addr	Logical AND memory to A register
ANL addr	Logical AND memory to L register
ERA[#] addr	Exclusive OR memory to A register
ERL addr	Exclusive OR memory to L register
ORA addr	Inclusive OR memory to A register

Memory Test and Skip Operations: The instructions in the following group test a memory location in various ways, and skip or do not skip based on the result.

MEMORY TEST AND SKIP OPERATIONS

CAS[#] addr	Compare A register to memory and skip: 0 halfwords if A > memory 1 halfword if A = memory 2 halfwords if A < memory
CLS addr	Compare L register to memory and skip: 0 halfwords if L > memory 1 halfword if L = memory 2 halfwords if L < memory
IRS[#] addr	Increment memory and skip if 0

Integer Operations: The instructions in this group perform binary integer arithmetic operations involving a register and a memory location. (Decimal and floating point operations are described later in this chapter.)

In all cases, one of the operands of the integer operation must have been stored in the appropriate register before the operation is performed. Use the positioning instructions listed below to properly position operands in the L and E registers before DIV and DVL operations, and after MPY and MPL operations. See the Instruction Sets Guide for details on how positioning is done.

Unless otherwise noted, the result of the operation is stored in the register, not in the memory location. A store instruction (appropriate to the register type) must follow the operation if the result is to be used later and if the same register (or part of it) is used in intervening operations. For example, an integer operation leaving its result in the A register, followed by one leaving its result in the L register, destroys the result in the A register, since A is the upper half of L. An STA instruction should appear between the two integer operations.

In some cases, consecutive, or chained, integer operations involving the same register or parts of a register are valid, provided the the memory operands are of the correct length. For example, an algebraic expression such as

```
Y = M * X + B
```

can be coded as the following instruction sequence:

```
        LDA   M
        MPY   X
        ADL   B
        PIMA
        STA   Y
         .    .
         .    .

M       DATA  3
X       DATA  2
B       DATA  5L
Y       BSS   1
```

The DATA statements declare M and X as 16-bit quantities for the MPY operation, and B as a 32-bit quantity for the ADL operation. The result of the sequence (Y) is in the L register. The PIMA instruction properly positions the result in the A register for further 16-bit operations; omit the PIMA if subsequent operations involve 32-bit quantities.

Exception conditions can occur for integer operations if the results are outside the range of the result location. In an operation such as the example above, if the result in the L register is greater than +32767 or less than -32768, the PIMA instruction would cause an overflow condition to be set. The Instruction Sets Guide describes, for each instruction, the result limits and the disposition of exception conditions for that instruction.

POSITIONING OPERATIONS

PIDA	Position before integer divide
PIDL	Position before long integer divide
PIMA	Position after integer multiply
PIML	Position after long integer multiply

INTEGER ARITHMETIC OPERATIONS

ADD[#] addr	Add 16-bit memory to A register
ADL addr	Add 32-bit memory to L register
DIV[#] addr	Divide 32-bit A/B register by 16-bit memory; 16-bit quotient in A register, 16-bit remainder in B register
DVL addr	Divide 64-bit L/E register by 32-bit memory; 32-bit quotient in L register, 32-bit remainder in E register
MPL addr	Multiply L register by 32-bit memory; high half of result in L register, low half of result in E register
MPY[#] addr	Multiply A register by 16-bit memory; high half of result in A register, low half of result in B register
SBL addr	Subtract 32-bit memory from L register
SUB[#] addr	Subtract 16-bit memory from A register

Decimal Operations: The instructions in this group perform decimal arithmetic operations involving two memory locations. Since the amount of information required to perform a decimal operation cannot be contained in a single instruction such as XAD (decimal add), information about the operands' addresses and characteristics (length, data type, scale differential, etc.) must be stored elsewhere. The setup operations are described below.

Before performing any decimal operation, your program must store the following information in the indicated registers:

Operand address 1	Field address register 0 (FAR0)
Operand address 2	Field address register 1 (FAR1)
Control word	L register

For the decimal edit (XED) instruction only, a fourth setup operation is necessary: the address of the beginning of the edit control subprogram must be loaded into the auxiliary base (XB) register. The EAXB instruction is used for this purpose (see Memory/Register Transfer Operations, earlier in this chapter).

The field address registers are loaded by using EAFA instructions (see Character and Field Operations, later in this chapter, and the EAFA instruction description in the Instruction Sets Guide).

The control word and the operand addresses can be defined and loaded as shown below for a decimal add:

```
            EAFA  0,DATA_0
            EAFA  1,DATA_1
            LDL   CTL_WD     (or  LDL   ='02004212001L)
            XAD
             .      .
             .      .
CTL_WD      DATA  '02004212001L
DATA_0      DATA  2030
DATA_1      DATA  59846
```

Each decimal instruction description given in the Instruction Sets Guide defines the control word fields required by that instruction. You must determine the corresponding bit patterns in each case and transform the required bits into their octal equivalent (you may also use a decimal or hexadecimal equivalent, or the bit string itself). Chapter 6 of the System Architecture Reference Guide describes the control word and its fields in detail.

Whether you declare the control word as a separate data item or use a literal value, be sure that you define it as a long (32-bit) quantity by appending L to either declaration. Otherwise only (the low-order) 16 bits will be allocated; the LDL will nonetheless load 32 bits, giving unpredictable results for the decimal operation.

The example above uses simple unsigned decimal declarations as operands. Various other declarations can be used, depending on whether your program uses packed decimal, leading or trailing sign, or separate or embedded sign representations. These representations are described in detail under DECIMAL DATA in Chapter 6 of the System Architecture Reference Guide.

Unless otherwise noted, the result of a decimal operation is stored in the field represented by the address in FAR1 (field 2).

Arithmetic exception conditions can occur for decimal operations if the results are outside the range of the result location. The Instruction Sets Guide describes, for each instruction, the disposition of exception conditions for that instruction.

Some decimal operations use several of the general registers during their execution. It is the program's responsibility to save and restore these registers when necessary.

The decimal operations are summarized below. All are short form instructions.

DECIMAL ARITHMETIC OPERATIONS

XAD	Decimal add or subtract, depending on control word
XCM	Decimal compare
XDV	Decimal divide field 2 by field 1; quotient and remainder in field 2 (see Instruction Sets Guide)
XMP	Decimal multiply (see Instruction Sets Guide for setup and result placement in field 2)
XMV	Decimal move

DECIMAL CONVERSION AND EDITING OPERATIONS

XBTD	Convert binary to decimal (see Instruction Sets Guide for location of binary number); result in field 1, does not alter field 2 or FAR1
XDTB	Convert decimal to binary; source in field 1, does not alter field 2 or FAR1 (see Instruction Sets Guide for location of binary number)
XED	Edit under control of a subprogram (see Instruction Sets Guide for setup and control program information)

Floating Point Operations: The instructions in this group perform operations on single-precision, double-precision, or quad-precision floating point numbers. The four arithmetic operations can be performed, as can a variety of load, store, test, skip, and other such operations. Instructions that branch as a result of tests on the floating accumulator are summarized under Branch Instructions, earlier in this chapter.

Most operations involve the use of a group of user registers known collectively as floating accumulators (FACs). The accumulators occupy the same physical locations as the field address and field length registers FAR0, FLR0, FAR1, and FLR1. Chapter 6 of the System Architecture Reference Guide gives details on the structure of the accumulators; accumulator and memory storage capacities for floating point numbers; and normalization, rounding, and overflow conditions. See also Chapter 9 of the same volume for some cautions on floating point and field register overlap.

The following subgroups summarize the floating point operations. Unless otherwise stated, instruction mnemonics that begin with F operate on single-precision numbers; those beginning with D operate on double-precision numbers; those beginning with Q operate on quad-precision numbers. FAC, DAC, and QAC, refer to the floating accumulator for single, double, and quad precision, respectively.

FLOATING ACCUMULATOR OPERATIONS

DFCM	Two's-complement DAC mantissa
DFCS addr	Compare DAC with memory and skip: 0 halfwords if DAC > memory 1 halfword if DAC = memory 2 halfwords if DAC < memory
DFLD addr	Load DAC from memory (does not normalize before loading)
DFST addr	Store DAC into memory (does not normalize before storing)
FCM	Two's-complement FAC mantissa
FCS addr	Compare FAC with memory and skip: 0 halfwords if FAC > memory 1 halfword if FAC = memory 2 halfwords if FAC < memory
FLD addr	Load FAC from memory
FSGT	Skip 1 halfword if FAC > 0; used for all precisions
FSLE	Skip 1 halfword if FAC <= 0; used for all precisions
FSMI	Skip 1 halfword if FAC < 0; used for all precisions
FSNZ	Skip 1 halfword if FAC <> 0; used for all precisions
FSPL	Skip 1 halfword if FAC > 0; used for all precisions
FST addr	Store FAC into memory (does not normalize before storing)
FSZE	Skip 1 halfword if FAC = 0; used for all precisions
QFCM	Two's-complement QAC
QFCS addr	Compare QAC with memory and skip: 0 halfwords if QAC > memory 1 halfword if QAC = memory 2 halfwords if QAC < memory
QFLD addr	Load QAC from memory
QFST addr	Store QAC into memory (does not normalize before storing)

FLOATING POINT CONVERSION OPERATIONS

FCDQ	Convert double to quad
FDBL	Convert single to double
FLTA	Convert A register integer to floating point and store in FAC
FLTL	Convert L register integer to floating point and store in FAC
INTA	Convert DAC to integer and store in A register (ignores fractional part)
INTL	Convert DAC to integer and store in L register (ignores fractional part)
QINQ	Convert QAC to integer and store in QAC (see Instruction Sets Guide)
QIQR	Convert and round QAC to integer and store in QAC (see Instruction Sets Guide)

FLOATING POINT ARITHMETIC OPERATIONS

DFAD addr	Add memory to DAC
DFDV addr	Divide DAC by memory
DFMP addr	Multiply DAC by memory
DFSB addr	Subtract memory from DAC
FAD addr	Add memory to FAC
FDV addr	Divide FAC by memory
FMP addr	Multiply FAC by memory
FSB addr	Subtract memory from FAC
QFAD addr	Add memory to QAC
QFDV addr	Divide QAC by memory
QFMP addr	Multiply QAC by memory
QFSB addr	Subtract memory from QAC

FLOATING POINT ROUNDING OPERATIONS

DRN	Round quad to double; store in bits 1 through 64 of QAC (see Instruction Sets Guide for rounding rules)
DRNM	Round quad to double towards minus infinity; store in bits 1 through 64 of QAC (see Instruction Sets Guide for rounding rules)
DRNP	Round quad to double towards plus infinity; store in bits 1 through 64 of QAC (see Instruction Sets Guide for rounding rules)
DRNZ	Round quad to double towards zero; store in bits 1 through 64 of QAC (see Instruction Sets Guide for rounding rules)
FRN	Round double to single; store in bits 1 through 48 of DAC (see Instruction Sets Guide for rounding rules)
FRNM	Round double to single towards minus infinity; store in bits 1 through 48 of DAC (see Instruction Sets Guide for rounding rules)
FRNP	Round double to single towards plus infinity; store in bits 1 through 48 of DAC (see Instruction Sets Guide for rounding rules)
FRNZ	Round double to single towards zero; store in bits 1 through 48 of DAC (see Instruction Sets Guide for rounding rules)

FLOATING POINT LOAD INDEX OPERATIONS

DFLX addr	Load X register with 4 times 16-bit contents of memory; addr cannot be indexed
FLX addr	Load X register with 2 times 16-bit contents of memory; addr cannot be indexed
QFLX addr	Load X register with 8 times 16-bit contents of memory; addr cannot be indexed

Character and Field Operations: V-mode programs can operate on characters and character strings (fields) with the aid of the field address registers FAR0 and FAR1 and the field length registers FLR0 and FLR1. These registers operate in pairs, FAR0/FLR0 and FAR1/FLR1. (These are the same registers used in the decimal operations described earlier in this chapter.) The discussion below summarizes the requirements common to many of the character and field operations; detailed information can be found in Chapter 6 of the System Architecture Reference Guide and in the Instruction Sets Guide. In this description, the terms character and byte are equivalent, and represent either the high-order or low-order eight bits of a halfword.

In operations in which only one pair of FAR/FLR registers is involved (such as LDC and STC), either pair (0 or 1) can be specified. For operations involving both pairs (such as ZED or ZMV), FAR0/FLR0 represents the source string and FAR1/FLR1 represents the destination string.

Before and during a character or field operation the FAR contains the address of the next character to be operated on, while the FLR contains the number of characters yet to be processed. Both registers must be preloaded with the appropriate address and length information before the operation can begin. Use the EAFA instruction to load the beginning address of the field into FAR, and the LFLI instruction to load the field length (in bytes) into FLR. (See the instruction summaries below, and the detailed descriptions in the Instruction Sets Guide.)

The initializing instructions are coded as shown below. far or flr represents the field register pair number (0 or 1).

```
EAFA far,addr        initialize for left byte at addr, OR
EAFA far,addr+8B     initialize for right byte at addr

LFLI flr,number      initialize field length in bytes
```

After execution of the EAFA instruction, the FAR contains a segment/offset address and the FLR contains the character offset (see the next paragraph) within the halfword at the segment/offset location.

After execution of the LFLI instruction, bits 44 through 64 of the FLR contain the length of the field (the number of characters) to be operated on. In addition to the field length information, the FLR contains, in bits 1 to 4, the character offset which, during the character or field operation, is updated to point to the next character to be processed. The bit field can thus be considered a four-bit extension of the segment/offset address in FAR. The value in the bit field at any given time is either '0000'b or '1000'b (0 or 8 decimal), representing the left or right byte, respectively, at the segment/offset location. Each time the bit field is updated to 0, the offset part of the address is incremented by 1.

For the edit and translate (ZED and ZTRN) instructions, an additional setup operation is necessary: the address of the beginning of the edit control subprogram (for ZED) or the beginning of the translation table (for ZTRN) must be loaded into the auxiliary base (XB) register. The EAXB instruction is used for this purpose (See Memory/Register Transfer Operations, earlier in this chapter.)

Character and field operations are of two types; those that process one character per invocation of an instruction (LDC and STC), and those that operate on an entire field with one invocation of an instruction (such as ZED, ZFIL, ZMV). LDC and STC can be used to process consecutive characters one at a time, and are often coded in a loop. The loop is typically terminated by a branch on condition code equal (BCEQ); this is possible because the LDC or STC instruction decrements the length field in FLR for each character processed, and sets the condition codes equal when the count reaches zero. The field operations other than LDC and STC do not need programmed loops to operate on fields; their looping is internal to the instruction, and the condition codes at their termination are usually indeterminate.

Character and field operations use several of the general registers during their execution. It is the program's responsibility to save and restore these registers when necessary.

The following groups summarize the instructions used in character and field operations. The designations far and flr represent the number (0 or 1) of the field register pair.

FIELD REGISTER OPERATIONS

ALFA far	Add contents of L register to FARfar
EAFA far,addr	Load memory address into FARfar
LFLI flr,n	Load n into FLRflr
STFA far,addr	Store contents of FARfar into memory (stores 32 or 48 bits -- see Instruction Sets Guide)
TFLL flr	Transfer contents of FLRflr to L register
TLFL flr	Transfer contents of L register to FLRflr; maximum allowable number is 2**20 (the number of bits in a 64K segment)

CHARACTER AND FIELD OPERATIONS

LDC flr	If FLRflr is nonzero, load character pointed to by FARflr into bits 9 through 16 of A register; else set condition codes equal
STC flr	If FLRflr is nonzero, store bits 9 through 16 of A register in location pointed to by FARflr; else set condition codes equal
ZCM	Compare fields at FAR0 (F1) and FAR1 (F2); if F1 > F2, set condition codes GT if F1 = F2, set condition codes EQ if F1 < F2, set condition codes LT
ZED	Edit character field (see Instruction Sets Guide for edit control information)
ZFIL	Fill field starting at FAR1 with the character in bits 9 to 16 of L register; FLR1 specifies length of string to fill
ZMV	Move field starting at FAR0 to field starting at FAR1; FLRs define lengths of fields (see Instruction Sets Guide for treatment of unequal length fields)
ZMVD	Move field starting at FAR0 to equal length field starting at FAR1; FLR1 defines length of fields
ZTRN	Translate field starting at FAR0 and store in field starting at FAR1; FLR1 defines length of fields, XB contains beginning address of translation table (see Instruction Sets Guide for the translation algorithm)

Process-Related Operations

The instructions in this group are concerned with various aspects of the control of a process and its related procedures. Chapters 8 and 9 of the System Architecture Reference Guide discusses process exchange and procedure calls in detail. Appendix C of the same volume describes process exchange on the Prime 850.

Only summary lists of these instructions are presented in this chapter; the Instruction Sets Guide goes into further detail on each one.

ADDRESS MODE CHANGE OPERATIONS

E16S	Enter 16S address mode
E32I	Enter 32I address mode
E32R	Enter 32R address mode
E32S	Enter 32S address mode
E64R	Enter 64R address mode
E64V	Enter 64V address mode

INTER-PROCECURE TRANSFER OPERATIONS

ARGT	Argument transfer
CALF addr	Call fault handler whose ECB is at addr
PCL addr	Call procedure whose ECB is at addr
PRTN	Procedure return
STEX	Stack extend

QUEUE MANAGEMENT OPERATIONS

ABQ addr	Add entry in A register to bottom of queue pointed to by addr
ATQ addr	Add entry in A register to top of queue pointed to by addr
RBQ addr	Remove from bottom of queue pointed to by addr and store in A register
RTQ addr	Remove from top of queue pointed to by addr and store in A register
TSTQ addr	Set A register to number of items in queue pointed to by addr

HARDWARE-RELATED OPERATIONS

SMCR	Skip 1 halfword if machine check flag reset (0)
SMCS	Skip 1 halfword if machine check flag set (1)
SSSN	Store system serial number in memory block specified by XB register

MISCELLANEOUS OPERATIONS

HLT	If not in ring 0, simulate a processor halt and display a message
NOP	No operation; proceed to next instruction
STTM	Store process timer in memory specified by XB register
SVC	Supervisor call
XEC addr	Execute a single instruction at addr

Restricted Instructions

The instructions in this group deal mainly with the manipulation of system data structures that are essential to PRIMOS operation, and are therefore protected against access by the casual user. They can be executed by users who have access to ring 0. Refer to Chapter 5 of the System Architecture Reference Guide for further information on these instructions. Chapter 10 of the same volume contains information on the role of interrupts process exchange.

Only summary lists of these instructions are presented in this chapter; the Instruction Sets Guide goes into further detail on each one.

INTERRUPT HANDLING OPERATIONS

ENB	Enable interrupts
ENBL	Enable interrupts (local)
ENBM	Enable interrupts (mutual)
ENBP	Enable interrupts (process)
INBC addr	Interrupt notify beginning; clear active interrupt
INBN addr	Interrupt notify beginning
INEC addr	Interrupt notify end; clear active interrupt
INEN addr	Interrupt notify end
INH	Inhibit interrupts
INHL	Inhibit interrupts (local)
INHM	Inhibit interrupts (mutual)
INHP	Inhibit interrupts (process)
IRTC	Interrupt return; clear active interrupt
IRTN	Interrupt return

INPUT/OUTPUT OPERATION

EIO addr	Execute I/O

ADDRESS TRANSLATION OPERATIONS

ITLB	Invalidate STLB entry
LIOT addr	Load IOTLB
PTLB	Purge TLB

PROCESS EXCHANGE OPERATIONS

LPID	Load Process ID register from bits 1 through 10 of A register
LPSW addr	Load program status word from memory

SEMAPHORE OPERATIONS

NFYB addr	Notify semaphore at addr; use LIFO queuing
NFYE addr	Notify semaphore at addr; use FIFO queuing
WAIT addr	Wait on semaphore at addr

MISCELLANEOUS OPERATIONS

RMC	Reset machine check flag to 0
RTS	Reset time slice
STPM	Store processor model number and microcode revision number in memory block specified by XB register

9
Machine Instructions -- I Mode

This chapter describes the set of machine instructions that is available to you when your program is executing in 32I addressing mode (commonly referred to simply as I mode). It also describes the various types of addressing usable in I mode, as well as the registers available to I mode programs.

TYPES OF ADDRESSING

I-mode programs use both short and long form instructions. All instructions that refer to memory locations, either for data manipulation or as destinations in control transfers, are long form (32-bit) instructions. Those that manipulate registers and perform operations unrelated to memory are mostly short form (16-bit).

Direct long form instructions can address all locations in any 128-Kbyte segment pointed to by one of the four base registers; indirect long form instructions can address all locations in up to 4096 128-Kbyte segments.

I mode also supports general register relative, register to register, and immediate addressing for many instructions. These addressing forms are described under Addressing Through Registers, in this section. The instructions that allow these addressing forms are so identified in the instruction summaries later in this chapter.

I-mode memory addresses at execution time (known as effective addresses) are virtual addresses consisting of a segment number and an

offset. The offset identifies a particular word relative to the beginning of the segment. Segment numbers and offsets are commonly represented (in listings such as memory maps produced by linkers) as two octal numbers separated by a slash:

```
4001/1025
```

The offset of the first word in any segment is 0; therefore the address shown above represents the (octal) 1026th word in segment (octal) 4001. Segmented addressing is fully described in the System Architecture Reference Guide.

Segment numbers (at least in their resolved form) do not appear in an assembly listing; generally only the offsets appear, with a code indicating the type of segment in which they will eventually reside. It is the linking operation performed by SEG or BIND that (among other things) establishes segment numbers based on whether the data or instruction in question is defined in a procedure, stack, or linkage segment of the program. For information on program linking and how segments are allocated, refer to the SEG and LOAD Reference Guide and the Programmer's Guide to BIND and EPFs.

The assembler keeps track of segment information so that, when you use a label as an operand in an instruction, you do not need to concern yourself with specifying the segment in which that label was defined.

The processor hardware uses various components of an instruction to decide which of six addressing forms to use in the instruction's execution:

- Direct
- Indexed
- Indirect
- Indirect indexed
- General register relative
- Register to register
- Immediate

These addressing forms are summarized below. Detailed information on addressing can be found in Chapter 3 of the System Architecture Reference Guide.

Direct Address

A direct address is indicated by an operand of the form

```
LABEL1 [ +|- LABEL2 ...  +|- LABELn ]
```

The operand is the simple symbolic name of an element defined somewhere within the referencing segment, or a combination of such names which, when combined arithmetically, yield a single numeric displacement within the limits of the segment. In certain cases, you can also use absolute numbers or other literal values; the assembler resolves them to displacements. The processor, at execution time, adds the displacement to the contents of the appropriate base register (procedure, link, or stack) to form the effective address.

Indexed Address

Indexing provides a convenient way to address successive elements in a table or to count iterations in a loop.

An indexed address is indicated by an operand of the form

```
LABEL,gr
```

where gr is the designation of one of the general registers 1 through 7. I-mode instructions can use any of the general registers 1 through 7 as an index register. (I-mode register usage is described later in this chapter.)

Here, LABEL is normally a single symbolic name whose displacement value is determined as in direct addressing. The processor then adds this displacement to the current contents of a general register, whose value must have been previously set by the program. The result of the addition is the effective address.

Indirect Address

An indirect address is indicated by an operand of the form

```
LABEL,*
```

Indirect addresses enable a program to make a memory reference to a location through an indirect pointer. A primary use of indirect addressing is to refer to locations outside of the segment making the reference. Prime processors pass arguments to subroutines by address rather than by value. Therefore, subroutines (which frequently reside outside their callers' segments and cannot access or be accessed by their callers using direct addressing) must use indirect pointers to communicate with their callers.

In an indirect address, LABEL is a single symbolic name whose displacement value is determined as in direct addressing. However, rather than using the resulting address directly as the target of the operation, the processor interprets it as the address of a pointer which, in turn, contains the address of the target. Only one level of indirection is allowed in I-mode programs; that is, the contents of the pointer cannot itself be an indirect address.

Since an indirect pointer is a 32-bit address, it can contain segment as well as displacement information, and can thus address across segment boundaries, which addresses that contain only displacement information cannot do. Indirect pointers can also be 48 bit entities, and are used when it is necessary to address specific bits within the target.

Indirect Indexed Address

Indirect indexed addresses take one of two forms, depending on whether indexing is performed before or after indirection:

LABEL,gr*	pre-indexed by general register gr
LABEL,*gr	post-indexed by general register gr

In pre-indexing, the processor adds the contents of the register to the value of LABEL, and uses the result as an indirect address to form the effective address. In post-indexing, the processor resolves the indirect address of LABEL, then adds the register contents to it to form the effective address.

Each form of indirect indexed addressing has its usefulness in particular situations. Pre-indexing is useful in stepping through a table of addresses, as in the first of the following examples. Post-indexing is used primarily in stepping through a table of data, as in the second example.

Example 1 (pre-indexed indirect):

```
          SEGR
START     LH    7,=-8
LOOP      LH    2,TABLE+8,7*    pre-indexed through IP table to data
          STH   2,NUMBER
          STH   7,SAVE7
          CALL  TODEC
          AP    NUMBER,SL
          CALL  TONL
          LH    7,SAVE7
          BHI2  7,LOOP
          PRTN
*
          LINK
TABLE     IP    TEST1
          IP    TEST2
          IP    TEST3
          IP    TEST4
NUMBER    BSS   1
SAVE7     BSS   1
TEST3     DATA  11
TEST1     DATA  22
TEST2     DATA  33
TEST4     DATA  44
ECB$      ECB   START
          END   ECB$
```

Example 2 (post-indexed indirect):

```
          SEGR
          RLIT
START     LH     7,=5
LOOP      LH     2,TABLE,*7    post-indexed to table of data
          STH    2,NUMBER
          STH    7,SAVE7
          CALL   TODEC
          AP     NUMBER,SL
          CALL   TONL
          LH     7,SAVE7
          BHD1   7,LOOP
          PRTN
*
          LINK
TABLE     IP     TAB-1
NUMBER    BSS    1
SAVE7     BSS    1
TAB       DATA   1, 3, 5, 7, 9
ECB$      ECB    START
          END    ECB$
```

Both of these programs print out a list of data items, one per line; the first addresses the items through a table of indirect pointers (IPs) which in turn contain the addresses of the data items, while the second accesses the items in the data table itself. In both cases the tables consist of equal-length entries: in the first they are 32-bit pointers, and in the second they are 16-bit constants. This accounts for the difference in the incrementing or decrementing technique used in the two examples. The first uses an increment of 2 (BHI2) to step through the 32-bit pointers; the second used a decrement of 1 (BHD1) to step through the 16-bit data items.

Note that the CALL statements generate procedure call (PCL) instructions, which destroy the contents of register 7, among others, if argument passing is required (as it is for the calls to the TODEC subroutine). The register must therefore be saved before the calls and restored after the calls. (Refer to the description of the PCL instruction in the Instruction Sets Guide.)

Addressing Through Registers

Many I-mode memory reference instructions have, in addition to a memory reference format, two forms of addressing in which a general register replaces a memory address. These are known as general register relative and register to register addressing.

General Register Relative Addressing: This addressing format is useful in the manipulation of data within a structured record or data storage area, in which a given field is always at the same position relative, say, to the beginning of the record or to a primary key field. General register relative addressing involves two registers. The first is used in the normal way for source, destination, or result storage, while the second is used in much the same way as a base register and contains, rather than a data value, a segment/offset address that points to a memory location. The instruction does its work at this location or some location relative to it.

An instruction in general register relative format is always 32 bits long. Bits 1 through 16 contain, in addition to the operation code, two general register designator fields. One specifies the number of the source, destination, or result register; the other the number of the register containing the address pointer. Bits 17 through 32 contain a positive or negative augment to be applied to the address pointer. Some examples are:

```
L   1,R0%     Load GR1 with the data at the address contained in
                GR0
A   1,R0%+2   Add to the contents of GR1 the data at 2 locations
                after the address contained in GR0
ST  1,R0%-4   Store the contents of GR1 at 4 locations before the
                address in GR0
```

Bits 17 through 32 of each instruction contain the augment 0, +2 or -4. (Negative augments are in two's-complement form.) If no augment is specified, as in the first instruction, bits 17 through 32 contain 0. GR0 must be preloaded with the address relative to which these actions are performed. The sequence shown above can be coded in the following way:

```
            SEGR
      .      .
      .      .

            EAR     0,STOR      preload address of STOR into R0
            L       1,R0%       load contents of STOR into GR1
            A       1,R0%+2     add contents of STOR+2 to GR1
            ST      1,R0%-4     store GR1 into STOR-4
      .      .
      .      .

            LINK
            BSS     4           location of STOR-4
STOR        DEC     10L         location of STOR
            DEC     5L          location of STOR+2
```

In the instruction summary lists later in this chapter, instructions that can operate in the general register relative format are indicated by a G following the instruction format.

Register to Register Addressing: This form of addressing manipulates one register with respect to another, as in a transfer of the contents of one register to another. Some examples are:

```
L   1,2     Load contents of GR2 into GR1
ST  0,7     Store contents of GR0 into GR7
A   7,6     Add contents of GR6 to contents of GR7
```

In this form, the instruction is always 16 bits long; the instruction contains, in addition to the operation code, fields designating the source and destination register numbers. The second register contains the value that the displacement address would normally point to.

In the instruction summary lists later in this chapter, instructions that can operate in the register to register format are indicated by an R following the instruction format.

Immediate Addressing

Some instructions that allow one or both of the register formats also allow an immediate format. In this form of addressing, instructions are always 32 bits long. Bits 17 through 32 contain, instead of a displacement, a numeric operand value coded as a literal:

```
A   2,=10      or
A   2,=10L
```

Literals are coded as described under Operand Field, in Chapter 3. The above example adds the numeric value 10 to the contents of general register 2. The literal can be coded as either short (16 bits, without the L) or long (32 bits, with the L). The way it is coded determines what part of the register is affected. The short literal generates what is known as a type 1 immediate instruction, which affects the high half of the register; the long literal generates a type 2 immediate instruction, which affects the entire register. Single and double precision floating point literals can also be coded as operands of single and double precision floating point instructions; these generate type 3 immediate instructions.

Refer to the 32I Mode Summary table in Appendix B of the Instruction Sets Guide for the bit structure that determines the type of an immediate instruction.

In the type 1 immediate format, whether the low half of the register is affected is determined by the operation: a type 1 load instruction loads into the high half of the register and zero-fills the low half. A type 1 add or subtract instruction affects the high half and leaves the low half unchanged.

Regardless of whether the literal is 16 or 32 bits long, if the numeric value of the constant will fit into 16 bits, it is placed in bits 17 through 32 of the instruction; otherwise the instruction in effect becomes a memory reference format instruction whose displacement points to a literal storage location in memory. Thus, any integer constant whose value is between -2**15 and +2**15 - 1, or any string constant of one or two characters, can be represented in an immediate format instruction.

In the instruction summary lists later in this chapter, instructions that can operate in the immediate format are indicated by an I following the instruction format.

REGISTER USAGE

The registers listed on the following page are available to I-mode programs; their sizes are given in bits.

Register	Size	Function
0-7	32	General Registers 0 through 7
FAR0	32	Field address register 0
FAR1	32	Field address register 1
FLR0	32	Field length register 0
FLR1	32	Field Length register 1
FAC0	64	Floating point accumulator 0 (single precision); concatenation of FAR0 and FLR0
FAC1	64	Floating point accumulator 1 (single precision); concatenation of FAR1 and FLR1
DAC0	64	Floating point accumulator 0 (double precision); concatenation of FAR0 and FLR0
DAC1	64	Floating point accumulator 1 (double precision); concatenation of FAR1 and FLR1
QAC	128	Floating point accumulator (quad precision); concatenation of FAR0, FLR0, FAR1 and FLR1
PB	32	Procedure base register
SB	32	Stack base register
LB	32	Link base register
XB	32	Auxiliary base register

The registers listed above are available to all I-mode programs, executing in both ring 0 and ring 3. These, as well as others available only to ring 0 programs, are described in the discussion of user register files in Chapter 9 of the System Architecture Reference Guide. If your program uses a FAC, pay particular attention to the discussion of overlap between floating point and field registers in that chapter.

Saving and Restoring Registers

If your program uses PCL instructions (generated by CALL statements), bear in mind that PCL destroys the contents of the GR3, GR5, GR7, FAC1, and XB registers. The code that is executed as a result of the CALL may use other registers (such as GR0, GR2, and FAC0). It is therefore prudent to provide for saving the contents of any registers whose contents are established before the CALL and are required after the call. This can be done easily by use of the register save (RSAV) and register restore (RRST) instructions; refer to the System Architecture Reference Guide for more information on saving and restoring registers.

The same precautions apply to most of the decimal operations and character and field operations. Each of these instruction descriptions in the Instruction Sets Guide indicates which registers, if any, the instruction uses.

Register Usage Between I and V Modes

You should be aware that when your program switches from I mode to V mode (as it might when calling a subroutine), there is a correspondence between registers in the user register set in the two modes. For example, the V-mode L register corresponds to I-mode general register 2 (GR2); the V-mode A and B registers correspond to the high and low halves, respectively, of I-mode register GR2. Therefore, if your I-mode program calls a V-mode subroutine and expects the result to occupy all 32 bits of general register 2 upon return from the subroutine, the subroutine should load the return value in the L register before returning to its caller.

A complete list of user registers and their intermode correspondences is given in the discussion of user register files in Chapter 9 of the System Architecture Reference Guide.

THE I-MODE INSTRUCTION SET

This section briefly describes the instructions that can be used in I-mode programs. Complete descriptions of all I-mode instructions can be found, listed alphabetically, in Chapter 3 of the Instructions Sets Guide.

In the summary instruction lists in the rest of this chapter, instructions that operate on the high half of GRr will affect bits 1 through 16 of the 32-bit register and may affect bits 17 through 32; those that operate on the low half will affect bits 17 through 32 and may affect bits 1 through 16. Instructions that are designated as operating simply on GRr affect the entire 32 bits.

I-mode instructions can be arranged in the following groups:

- Generic (register and shift)
- Branch and jump
- Memory reference
- Decimal
- Floating point
- Character
- Process-related operations
- Restricted operations

Generic Instructions

The generic instruction group comprises two subgroups: register generic and shift generic.

Register Generic Instructions: This subgroup contains chiefly instructions that manipulate the contents of the eight general registers GR0 through GR7 in various ways and under various conditions.

REGISTER INCREMENT/DECREMENT OPERATIONS

DH1	r	Decrement high half of GRr by 1
DH2	r	Decrement high half of GRr by 2
DR1	r	Decrement GRr by 1
DR2	r	Decrement GRr by 2
IH1	r	Increment high half of GRr by 1
IH2	r	Increment high half of GRr by 2
IR1	r	Increment GRr by 1
IR2	r	Increment GRr by 2

REGISTER CLEAR OPERATIONS

CR	r	Clear GRr to zero
CRHL	r	Clear high half of GRr to zero
CRHR	r	Clear low half of GRr to zero
ICHL	r	Interchange halfwords of GRr and clear high half
ICHR	r	Interchange halfwords of GRr and clear low half

REGISTER TRANSFER OPERATIONS

IRH	r	Interchange halfwords of GRr
INK	r	Transfer Keys register to GRr
OTK	r	Transfer GRr to Keys register
ICHL	r	Interchange halfwords of GRr and clear high half
ICHR	r	Interchange halfwords of GRr and clear low half

REGISTER BYTE OPERATIONS

CRBL	r	Clear bits 1 through 8 of GRr to zero
CRBR	r	Clear bits 9 through 16 of GRr to zero
ICBL	r	Interchange bits 1 through 8 and 9 through 16 of GRr; clear bits 1 through 8 to zero
ICBR	r	Interchange bits 1 through 8 and 9 through 16 of GRr; clear bits 9 through 16 to zero
IRB	r	Interchange bits 1 through 8 and 9 through 16 of GRr

REGISTER COMPLEMENT OPERATIONS

CMH	r	One's-complement high half of GRr
CMR	r	One's-complement GRr
TC	r	Two's-complement GRr
TCH	r	Two's-complement high half of GRr

ACCUMULATOR SIGN OPERATIONS

CHS	r	Change sign (complement bit 1) of GRr
CSR	r	Copy sign (bit 1) of GRr to C bit of Keys register; clear bit 1 of GRr to zero
SSM	r	Set sign (bit 1) of GRr to 1
SSP	r	Set sign (bit 1) of GRr to 0

OPERATIONS WITH KEYS REGISTER

ADLR	r	Add L bit of Keys register to bit 32 of GRr
RCB		Reset C bit of Keys register to 0
SCB		Set C bit of Keys register to 1
INK	r	Transfer Keys register to high half ofGRr
OTK	r	Transfer high half of GRr to Keys register

LOGIC OPERATIONS

LF	r	Set high half of GRr to false (bit 16 = 0)
LT	r	Set high half of GRr to true (bit 16 = 1)

CONDITION CODE (CC) TEST AND SET REGISTER

LCEQ	r	Set high half of GRr to 1 if CC is EQ; else set high half of GRr to 0
LCGE	r	Set high half of GRr to 1 if CC is GE; else set high half of GRr to 0
LCGT	r	Set high half of GRr to 1 if CC is GT; else set high half of GRr to 0
LCLE	r	Set high half of GRr to 1 if CC is LE; else set high half of GRr to 0
LCLT	r	Set high half of GRr to 1 if CC is LT; else set high half of GRr to 0
LCNE	r	Set high half of GRr to 1 if CC is NE; else set high half of GRr to 0

REGISTER VALUE TEST AND SET

LEQ r	Set bit 16 of GRr to 1 if GRr = 0; else set to 0
LGE r	Set bit 16 of GRr to 1 if GRr >= 0; else set to 0
LGT r	Set bit 16 of GRr to 1 if GRr > 0; else set to 0
LHEQ r	Set bit 16 of GRr to 1 if high half of GRr = 0; else set to 0
LHGE r	Set bit 16 of GRr to 1 if high half of GRr >= 0; else set to 0
LHGT r	Set bit 16 of GRr to 1 if high half of GRr > 0; else set to 0
LHLE r	Set bit 16 of GRr to 1 if high half of GRr <= 0; else set to 0
LHLT r	Set bit 16 of GRr to 1 if high half of GRr < 0; else set to 0
LHNE r	Set bit 16 of GRr to 1 if high half of GRr <> 0; else set to 0
LLE r	Set bit 16 of GRr to 1 if GRr <= 0; else set to 0
LLT r	Set bit 16 of GRr to 1 if GRr < 0; else set to 0
LNE r	Set bit 16 of GRr to 1 if GRr <> 0; else set to 0

FLOATING ACCUMULATOR VALUE TEST AND SET

LFEQ f,r	Set bit 16 of GRr to 1 if FACf = 0; else set to 0
LFGE f,r	Set bit 16 of GRr to 1 if FACf >= 0; else set to 0
LFGT f,r	Set bit 16 of GRr to 1 if FACf > 0; else set to 0
LFLE f,r	Set bit 16 of GRr to 1 if FACf <= 0; else set to 0
LFLT f,r	Set bit 16 of GRr to 1 if FACf < 0; else set to 0
LFNE f,r	Set bit 16 of GRr to 1 if FACf <> 0; else set to 0

Shift Generic Instructions: This subgroup contains instructions that shift the contents of a register a specified number of bits leftward or rightward. Either the high half of the register or the whole register can be shifted. There are three types of shifts: logical, rotate, and arithmetic.

Logical shifts move the register contents a specified number of bits in a specified direction, storing each bit shifted out in the C bit of the Keys register. Since only one bit is available for storage, only the last bit shifted is available for use in subsequent operations; all intervening bits are lost. Leftward shifts are zero-filled on the right; rightward shifts are zero-filled on the left.

Eight logical shift instructions contain in their operation codes directions for shifting a register or half register one or two bits leftward or rightward. A ninth instruction requires an operand to specify shifts of more than two bits. Refer to the description of the SHL instruction in the Instruction Sets Guide for the format of the operand.

Rotate shifts move the register contents a specified number of bits in a specified direction, storing each bit shifted out in the C bit of the

Keys register, and also copying the bit to the opposite end of the register. Thus, in a leftward shift, each bit shifted out is reproduced at the right end of the register; in a rightward shift, each bit shifted out is reproduced at the left end of the register. The last bit shifted is stored in the C bit of the Keys register.

There is only one rotate shift instruction. It requires an operand specifying the register size, shift direction, and number of bits to shift. Refer to the description of the ROT instruction in the Instruction Sets Guide for the format of the operand.

Arithmetic shifts move the register contents a specified number of bits in a specified direction.

For a leftward arithmetic shift, the C bit of the Keys register is initially set to zero. If a sign change occurs (bit 1 of the register changes from 0 to 1 or from 1 to 0) as a result of the shift, the C bit is set to 1. Bits shifted out are lost, and bits on the right end are zero-filled.

For a rightward arithmetic shift, each bit shifted out is stored in the C bit of the Keys register. The sign bit is propagated to the right. That is, if the original value of bit 1 is zero, the left-end bits are zero-filled; if the original value of bit 1 is one, the left-end bits are one-filled.

There is only one arithmetic shift instruction. It requires an operand specifying the register size, shift direction, and number of bits to shift. Refer to the description of the SHA instruction in the Instruction Sets Guide for the format of the operand.

The following groups summarize the shift generic instructions.

LOGICAL SHIFT OPERATIONS

SHL	r,addr	Logical shift high half or all of GRr as specified by contents of addr
SHL1	r	Logical shift high half of GRr left 1 bit
SHL2	r	Logical shift high half of GRr left 2 bits
SHR1	r	Logical shift high half of GRr right 1 bit
SHR2	r	Logical shift high half of GRr right 2 bits
SL1	r	Logical shift GRr left 1 bit
SL2	r	Logical shift GRr left 2 bits
SR1	r	Logical shift GRr right 1 bit
SR2	r	Logical shift GRr right 2 bits

ROTATE SHIFT OPERATIONS

ROT	r,addr	Rotate shift high half or all of GRr as specified by contents of addr

ARITHMETIC SHIFT OPERATIONS

SHA	r,addr	Arithmetic shift high half or all of GRr as specified by contents of addr

Branch Instructions

Branch instructions alter the normal sequential flow of control in a program. Branch instructions can test the following conditions and transfer control to a specifed address if the tested condition is true:

- Contents of the high half or all of a register with respect to zero
- Contents of a floating accumulator (FAC0 or FAC1) with respect to zero
- State of the condition codes (CC) or register bits
- State of the C bit of the Keys register
- State of the L bit of the Keys register and condition codes (magnitude)

Branch instructions can accept only direct addresses. Therefore, they can transfer control only to locations within the same segment as themselves. Jump instructions, described later in this chapter, can transfer control to other segments through indexing or indirection (as well as to their own segments through direct addressing).

The following functional groups summarize the branch instructions. All are long form instructions.

BRANCH ON REGISTER WITH RESPECT TO ZERO

```
BHEQ r,addr   Branch to addr if high half of GRr = 0
BHGE r,addr   Branch to addr if high half of GRr >= 0
BHGT r,addr   Branch to addr if high half of GRr > 0
BHLE r,addr   Branch to addr if high half of GRr <= 0
BHLT r,addr   Branch to addr if high half of GRr < 0
BHNE r,addr   Branch to addr if high half of GRr <> 0
BREQ r,addr   Branch to addr if GRr = 0
BRGE r,addr   Branch to addr if GRr >= 0
BRGT r,addr   Branch to addr if GRr > 0
BRLE r,addr   Branch to addr if GRr <= 0
BRLT r,addr   Branch to addr if GRr < 0
BRNE r,addr   Branch to addr if GRr <> 0
```

BRANCH ON FLOATING ACCUMULATOR WITH RESPECT TO ZERO

```
BFEQ f,addr  Branch to addr if FACf = 0
BFGE f,addr  Branch to addr if FACf >= 0
BFGT f,addr  Branch to addr if FACf > 0
BFLE f,addr  Branch to addr if FACf <= 0
BFLT f,addr  Branch to addr if FACf < 0
BFNE f,addr  Branch to addr if FACf <> 0
```

BRANCH ON CONDITION CODE OR REGISTER BIT

```
BCEQ addr       Branch to addr if EQ bit = 1
BCGE addr       Branch to addr if LT bit = 0 or EQ bit = 1
BCGT addr       Branch to addr if LT bit = 0 and EQ bit = 0
BCLE addr       Branch to addr if LT bit = 1 or EQ bit = 1
BCLT addr       Branch to addr if LT bit = 1 and EQ bit = 0
BCNE addr       Branch to addr if EQ bit = 0
BRBR r,b,addr   Branch to addr if bit b of GRr = 0;  b is  a  number
                  from 1 through 32
BRBS r,b,addr   Branch to addr if bit b of GRr = 1;  b is  a  number
                  from 1 through 32
```

BRANCH ON MAGNITUDE CONDITION

```
BMEQ addr    Branch to addr if EQ bit = 1 (same as BCEQ)
BMGE addr    Branch to addr if L bit = 1 (same as BLS)
BMGT addr    Branch to addr if L bit = 1 and EQ bit = 0
BMLE addr    Branch to addr if L bit = 0 and EQ bit = 1
BMLT addr    Branch to addr if L bit = 0 (same as BLR)
BMNE addr    Branch to addr if EQ bit = 0 (same as BCNE)
```

BRANCH ON STATE OF C/L BIT OF KEYS REGISTER

```
BCR addr     Branch to addr if C bit = 0
BCS addr     Branch to addr if C bit = 1
BLR addr     Branch to addr if L bit = 0
BLS addr     Branch to addr if L bit = 1
```

BRANCH ON REGISTER VALUE AFTER INCREMENT/DECREMENT

```
BHD1 r,addr  Decrement high half of GRr by 1; branch to addr if
              not zero
BHD2 r,addr  Decrement high half of GRr by 2; branch to addr if
              not zero
BHD4 r,addr  Decrement high half of GRr by 4; branch to addr if
              not zero
BHI1 r,addr  Increment high half of GRr by 1; branch to addr if
              not zero
BHI2 r,addr  Increment high half of GRr by 2; branch to addr if
              not zero
BHI4 r,addr  Increment high half of GRr by 4; branch to addr if
              not zero
BRD1 r,addr  Decrement GRr by 1;  branch to addr if not zero
BRD2 r,addr  Decrement GRr by 2;  branch to addr if not zero
BRD4 r,addr  Decrement GRr by 4;  branch to addr if not zero
BRI1 r,addr  Increment GRr by 1;  branch to addr if not zero
BRI2 r,addr  Increment GRr by 2;  branch to addr if not zero
BRI4 r,addr  Increment GRr by 4;  branch to addr if not zero
```

Computed Go To Instruction

The computed go to (CGT) instruction is a multi-directional form of branch instruction, capable of transferring control to any of several destination addresses in the same segment as the CGT instruction itself, depending on a preset value in a general register. It is functionally identical to a FORTRAN computed GO TO statement. A typical sequence is:

```
LH    destination_number
CGT r
DATA  number_of_destinations
DAC   destination_1
DAC   destination_2
 .       .       .
 .       .       .
DAC   destination_n
instruction
```

Let n be the number of addresses at which processing can continue. In this sequence, the LH instruction (or any other instruction or series of instructions that establish a value in the the high half of GRr) loads the register with a value of destination_number between 1 and n, inclusive, depending on which of the destinations control is to be transferred to. The DATA statement defines the number of valid destinations, plus 1 to account for an invalid register setting. Thus, if there are 4 valid addresses at which processing can continue, the value of the number_of_destinations operand is 5. If the value in GRr is, say, 2, then the CGT instruction transfers control to destination_2. An invalid value in GRr (a value less than 1 or greater than number_of_destinations - 1) transfers control to instruction to perform error processing.

The destination addresses must be in the local segment, and must be specified as 16-bit addresses; hence the use of a DAC pseudo-operation. Refer to Chapter 5 for a description of the DAC pseudo-operation.

Instruction can be any statement that generates a machine instruction. For example, it could be a CALL statement to a subroutine that prints an error message and then exits the program.

Jump Instructions

I-mode programs can use any of three jump instructions to alter the normal sequence of control. They are all unconditional transfers; they are independent of any previous test conditions. Two of them are jump-and-store instructions, normally used to transfer control to subroutines from which control is expected to return to the code following the jump instruction. The remaining one is a one-way jump to code from which no return is expected.

One-way Jump Instruction: A one-way jump instruction is used whenever a control transfer without a subsequent return is required. One of its typical uses is to return from a subroutine to which control has been transferred by one of the two jump-and-store instructions. Any of the following operand forms is permissible:

```
JMP addr          direct, to local segment only
JMP addr,r        indexed by r, to local segment only
JMP addr,*        indirect through IP, to any segment
JMP addr,*r       indirect through IP post-indexed by r,
                    to any segment
JMP addr,r*       indirect through IP pre-indexed by r,
                    to any segment
JMP Rr%           through general register r, to any segment
JMP XB%           through auxiliary base register,
                    to any segment
```

Jump-and-Store Instructions: A jump-and-store instruction stores a return address in a specific place (see the descriptions below); the code to which control is transferred must therefore know which form of jump was used to transfer to it so that it can make the appropriate return.

The two jump-and-store instructions and their storage and return mechanisms are summarized below. (Further details are given in the Instruction Sets Guide.)

JSR r,addr — Store the address following the JSR in the high half of GRr. Jump to addr. Return by JMP PB%,r

JSR can jump only to a location in the local segment. Be careful that, in the code being jumped to, the register used to save the return address is not used for other purposes; do not, for example use GR3, GR5, or GR7 if the target code contains CALL statements that pass arguments.

JSXB addr — Store the address following the JSXB in the auxiliary base (XB) register. Jump to addr. Return by JMP XB%.

JSXB can jump to nonlocal segments as well as to the local segment. The caution given above regarding register integrity applies also to the XB register when the target code contains CALL statements that pass arguments.

You can save the XB register into an unused general register at the beginning of the target routine if the target routine requires the XB register, and restore it just before the return. The following code will do this for you:

```
TARGET  LDAR   1,'17    store XB into GR1
          .       .
          .       .

        code requiring use of XB register
          .       .
          .       .

        STAR   1,'17    restore XB from GR1
        JMP    XB%      return to caller
```

The XB register is register '17 in the user register set. Refer to the System Architecture Reference Guide for information on addressing registers.

Memory Reference Instructions

This section describes a group of instructions that can refer to or modify the contents of memory locations; that is, they can read from and write to memory.

Because the memory reference instruction group comprises a large number of instructions, their descriptions are divided into several subgroups:

- Memory/register transfer operations
- Memory/register logic operations
- Memory test
- Integer operations
- Decimal operations
- Floating point operations
- Character and field operations

As stated earlier in this chapter, many of the memory reference instructions can be generated in register and immediate formats, in addition to their memory reference format. The instructions for which one, two, or all three of these formats are valid are indicated in the summaries below by an R for register to register format, a G for general register relative format, or an I for immediate format.

Memory/Register Transfer Operations: The instructions in this group transfer the contents of a memory location to a register, or transfer the contents of a register to a memory location. Also included here are instructions that calculate an effective address and load it into a register.

MEMORY/REGISTER TRANSFER OPERATIONS

EALB	addr	G	Load effective address into link base register
EAR	r,addr	G	Load effective address into GRr
EAXB	addr	G	Load effective address into auxiliary base register
I	r,addr	RG	Interchange GRr and 32-bit memory
IH	r,addr	RG	Interchange high half of GRr and 16-bit memory
L	r,addr	RGI	Load GRr from 32-bit memory
LDAR	r,addr		Load from addressed register into GRr (see Instruction Sets Guide)
LH	r,addr	RGI	Load high half of GRr from 16-bit memory

MEMORY/REGISTER TRANSFER OPERATIONS (continued)

LHL1 r,addr	RG	Shift contents of addr left 1 bit; then load into high half of GRr
LHL2 r,addr	RG	Shift contents of addr left 2 bits; then load into high half of GRr
LHL3 r,addr	RG	Shift contents of addr left 3 bits; then load into high half of GRr
RRST addr		Restore user registers (see Instruction Sets Guide)
RSAV addr		Save user registers (see Instruction Sets Guide)
ST r,addr	G	Store GRr into 32-bit memory at addr
STAR r,addr		Store GRr into addressed register (see Instruction Sets Guide)
STCD r,addr		If contents of GRr+1 equals 32-bit contents of addr, store GRr into addr; (addr is generated as a 32-bit address pointer)
STCH r,addr		If contents of low half of GRr equals 16-bit contents of addr, store high half of GRr into addr; (addr is generated as a 32-bit address pointer)
STH r,addr	G	Store high half of GRr into 16-bit memory at addr

Memory/Register Logic Operations: The instructions in this group perform logic operations on a register based on the contents of a memory address.

MEMORY/REGISTER LOGIC OPERATIONS

N r,addr	RGI	Logical AND 32-bit memory at addr to GRr
NH r,addr	RGI	Logical AND 16-bit memory at addr to high half of GRr
O r,addr	RGI	Inclusive OR 32-bit memory at addr to GRr
OH r,addr	RGI	Inclusive OR 16-bit memory at addr to high half of GRr
X r,addr	RGI	Exclusive OR 32-bit memory at addr to GRr
XH r,addr	RGI	Exclusive OR 16-bit memory at addr to high half of GRr

Memory Test Operations: The instructions in the following group test a memory location in various ways and set the condition codes in the keys register based on the result.

MEMORY TEST OPERATIONS

C	r,addr	RGI	Compare GRr to 32-bit memory at addr; if: GRr > memory, set CC to GT, LINK bit to 1 GRr = memory, set CC to EQ, LINK bit to 1 GRr < memory, set CC to LT, LINK bit to 0
CH	r,addr	RGI	Compare high half of GRr to 16-bit memory at addr; if: high half of GRr > memory, set CC to GT, LINK bit to 1 high half of GRr = memory, set CC to EQ, LINK bit to 1 high half of GRr < memory, set CC to LT, LINK bit to 0
TM	addr	G	Test 32-bit memory at addr; if: memory > 0, set CC to GT memory = 0, set CC to EQ memory < 0, set CC to LT
TMH	addr	G	Test 16-bit memory at addr; if: memory > 0, set CC to GT memory = 0, set CC to EQ memory < 0, set CC to LT

Integer Operations: The instructions in this group perform binary integer arithmetic operations involving a register and a memory location. Several instructions that increment or decrement memory without involving a register are also included. (Decimal and floating point arithmetic operations are described later in this chapter.)

In all cases in which a register is involved, one of the operands of the integer operation must have been stored in the appropriate register before the operation is performed. Use the positioning instructions listed below to properly position operands in the registers before D and DH operations, and after M and MH operations. See the Instruction Sets Guide for details on how positioning is done.

Except for the memory increment and decrement instructions, the result of the operation is stored in the register, not in the memory location. A store instruction must follow the operation if the result is to be used later and if the same register (or part of it) is used in intervening operations. For example, an integer operation leaving its result in the high half of a general register, followed by one leaving its result in the (entire) same register, destroys the result in the high half of the register. An STH instruction should appear between the two integer operations.

In some cases, consecutive, or chained, integer operations involving the same register or parts of a register are valid, provided the the memory operands are of the correct length. For example, an algebraic expression such as

```
Y = M * X + B
```

can be coded as the following instruction sequence:

```
        LDH   1,M
        MH    1,X
        A     1,B
        PIMH  1
        STH   1,Y
         .     .
         .     .

M       DATA  3
X       DATA  2
B       DATA  5L
Y       BSS   1
```

The DATA statements declare M and X as 16-bit quantities for the MH operation, and B as a 32-bit quantity for the A operation. The result of the sequence (Y) after the A operation occupies all of GR1. The PIMH instruction positions the result in the high half of GR1 for further operations involving 16-bit integers. If subsequent operations involve 32-bit integers the PIMH is omitted.

Exception conditions can occur for some integer operations if the results are outside the range of the result location. In an operation such as the example above, if the result in GR1 is greater than +32767 or less than -32768, the PIMH instruction causes an overflow condition to be set. The Instruction Sets Guide describes, for each instruction, the result limits and the disposition of exception conditions for that instruction.

POSITIONING OPERATIONS

PID	r	Position dividend in GRr and GRr+1 before 32-bit integer divide; r must be an even number
PIDH	r	Position dividend in GRr before 16-bit integer divide
PIM	r	Position product in GRr+1 into GRr after 32-bit integer multiply; r must be an even number
PIMH	r	Position product in GRr into high half of GRr after 16-bit integer multiply

INTEGER ARITHMETIC OPERATIONS

A	r,addr	RGI	Add 32-bit memory at addr to GRr
AH	r,addr	RGI	Add 16-bit memory at addr to high half of GRr
D	r,addr	RGI	Divide 64-bit GRr|GRr+1 by 32-bit memory at addr; 32-bit quotient in GRr, 32-bit remainder in GRr+1; r must be an even number
DH	r,addr	RGI	Divide 32-bit GRr by 16-bit memory at addr; 16-bit quotient in high half of GRr, 16-bit remainder in low half of GRr
DM	addr	G	Subtract 1 from 32-bit memory at addr
DMH	addr	G	Subtract 1 from 16-bit memory at addr
IM	addr	G	Add 1 to 32-bit memory at addr
IMH	addr	G	Add 1 to 16-bit memory at addr
M	r,addr	RGI	Multiply 32-bit GRr by 32-bit memory at addr; high half of result in GRr, low half of result in GRr+1; r must be an even number
MH	r,addr	RGI	Multiply high half of GRr by 16-bit memory at addr; high half of result in high half of GRr, low half of result in low half of GRr
S	r,addr	RGI	Subtract 32-bit memory at addr from GRr
SH	r,addr	RGI	Subtract 16-bit memory at addr from high half of GRr
ZM	addr	G	Clear 32-bit memory at addr to zero
ZMH	addr	G	Clear 16-bit memory at addr to zero

Decimal Operations: The instructions in this group perform decimal arithmetic operations involving two memory locations. Since the amount of information required to perform a decimal operation cannot be contained in a single instruction such as XAD (decimal add), information about the operands' addresses and characteristics (length, data type, scale differential, and the like) must be stored elsewhere. The setup operations are described below.

Before performing any decimal operation, your program must store the following information in the indicated registers:

Operand address 1	Field address register 0 (FAR0)
Operand address 2	Field address register 1 (FAR1)
Control word	General register 2 (GR2)

For the decimal edit (XED) instruction only, a fourth setup operation is necessary: the address of the beginning of the edit control subprogram must be loaded into the auxiliary base (XB) register. The EAXB instruction is used for this purpose (see Memory/Register Transfer Operations, earlier in this chapter).

The field address registers are loaded by using EAFA instructions (see Character and Field Operations, later in this chapter, and the EAFA instruction description in the Instruction Sets Guide).

The control word and the operand addresses can be defined and loaded as shown below for a decimal add:

```
            EAFA  0,DATA_0
            EAFA  1,DATA_1
            L     2,CTL_WD   (or L   2,='02004212001L)
            XAD
             .      .
             .      .
CTL_WD      DATA  '02004212001L
DATA_0      DATA  2030
DATA_1      DATA  59846
```

Each decimal instruction description given in the Instruction Sets Guide defines the control word fields required by that instruction. You must determine the corresponding bit patterns in each case and transform the required bits into their octal equivalent (you may also use a decimal or hexadecimal equivalent, or the bit string itself). Chapter 6 of the System Architecture Reference Guide describes the control word and its fields in detail.

Whether you declare the control word as a separate data item or as literal, be sure that you define it as a long (32-bit) quantity by appending L to either declaration. Otherwise only (the low-order) 16 bits will be allocated; the L instruction will nonetheless load 32 bits, giving unpredictable results for the decimal operation.

The example above uses simple unsigned decimal declarations as operands. The control word can specify other declarations, depending on whether your program uses packed decimal, leading or trailing sign, or separate or embedded sign representations of its data. These representations are described in detail under DECIMAL DATA in Chapter 6 of the System Architecture Reference Guide.

Unless otherwise noted, the result of a decimal operation is stored in the field represented by the address in FAR1 (field 2).

Arithmetic exception conditions can occur for decimal operations if the results are outside the range of the result location. The Instruction Sets Guide describes, for each instruction, the disposition of exception conditions for that instruction.

Some decimal operations use several of the general registers during their execution. It is the program's responsibility to save and restore these registers when necessary.

The decimal operations are summarized below. All are short (16-bit) instructions.

DECIMAL ARITHMETIC OPERATIONS

XAD	Decimal add or subtract, depending on control word
XCM	Decimal compare
XDV	Decimal divide field 2 by field 1; quotient and remainder in field 2 (see Instruction Sets Guide)
XMP	Decimal multiply (see Instruction Sets Guide for setup and result placement in field 2)
XMV	Decimal move

DECIMAL CONVERSION AND EDITING OPERATIONS

XBTD	Convert binary to decimal (see Instruction Sets Guide for location of binary number); result in field 1, does not alter field 2 or FAR1
XDTB	Convert decimal to binary; source in field 1, does not alter field 2 or FAR1 (see Instruction Sets Guide for location of binary number)
XED	Edit under control of a subprogram (see Instruction Sets Guide for setup and control program information)

Floating Point Operations: The instructions in this group perform operations on single-precision, double-precision, or quad-precision floating point numbers. The four arithmetic operations can be performed, as can a variety of load, store, test, and other such operations. Instructions that branch as a result of tests on the floating accumulator are summarized under Branch Instructions, earlier in this chapter.

Most operations involve the use of a group of user registers known collectively as floating accumulators (FACs). The accumulators occupy the same physical locations as the field address and field length registers FAR0, FLR0, FAR1, and FLR1. The System Architecture Reference Guide gives details on the structure of the accumulators; accumulator and memory storage capacities for floating point numbers; and normalization, rounding, and overflow conditions. The same volume lists some cautions on floating point and field register overlap.

The subgroups in this section summarize the floating point operations. Unless otherwise stated, instruction mnemonics that begin with F operate on single-precision numbers; those beginning with D operate on double-precision numbers; those beginning with Q operate on quad-precision numbers. FAC, DAC, and QAC, refer to the floating accumulators for single, double, and quad precision, respectively.

For those instructions that allow the register-to-register (R) format, addr can be a floating accumulator number, either 0 or 1, whichever is not designated by f. Using a register number in place of a memory address indicates that the operation is performed between the two single or double floating point accumulators. This technique is not applicable to quad precision operations, since there is only one quad floating accumulator.

FLOATING ACCUMULATOR OPERATIONS

DFC	f,addr	RGI	Compare DACf with 64-bit memory at addr; if: DACf > memory, set CC to GT DACf = memory, set CC to EQ DACf < memory, set CC to LT
DFCM	f		Two's-complement mantissa of DACf
DFL	f,addr	RGI	Load DACf from 64-bit memory at addr (does not normalize before loading)
DFST	f,addr	G	Store DACf into 64-bit memory at addr (does not normalize before storing)
FC	f,addr	RGI	Compare FACf with 32-bit memory at addr; if: FACf > memory, set CC to GT FACf = memory, set CC to EQ FACf < memory, set CC to LT
FCM	f		Two's-complement mantissa of FACf
FL	f,addr	RGI	Load FACf from 32-bit memory at addr (does not normalize before loading)
FST	f,addr	G	Store FAC into memory (does not normalize before storing)
QFC	addr	GI	Compare QAC with 128-bit memory at addr; if: QAC > memory, set CC to GT QAC = memory, set CC to EQ QAC < memory, set CC to LT
QFCM			Two's-complement mantissa of QAC
QFLD	addr	GI	Load QAC from 128-bit memory at addr (does not normalize before loading)
QFST	addr	G	Store QAC into 128-bit memory at addr (does not normalize before storing)

FLOATING POINT CONVERSION OPERATIONS

DBLE	f	Convert single precision FACf to double precision; store in DACf
FCDQ		Convert double to quad precision by clearing FAC0 to zero
FLT	f,r	Convert 32-bit integer in GRr to floating point; store in DACf
FLTH	f,r	Convert 16-bit integer in high half of GRr to floating point; store in FACf
INT	f,r	Convert double-precision DACf to 32-bit integer; store in GRr
INTH	f,r	Convert double-precision DACf to 16-bit integer; store in high half of GRr
QINQ		Convert QAC to integer and store in QAC (see Instruction Sets Guide)
QIQR		Convert and round QAC to integer and store in QAC (see Instruction Sets Guide)

FLOATING POINT ARITHMETIC OPERATIONS

DFA	f,addr	RGI	Add 64-bit memory at addr to DACf
DFD	f,addr	RGI	Divide DACf by 64-bit memory at addr
DFM	f,addr	RGI	Multiply DACf by 64-bit memory at addr
DFS	f,addr	RGI	Subtract 64-bit memory at addr from DACf
FA	f,addr	RGI	Add 32-bit memory at addr to FACf
FD	f,addr	RGI	Divide FACf by 32-bit memory at addr
FM	f,addr	RGI	Multiply FACf by 32-bit memory at addr
FS	f,addr	RGI	Subtract 32-bit memory at addr from FACf
QFAD	addr	GI	Add 128-bit memory at addr to QAC
QFDV	addr	GI	Divide QAC by 128-bit memory at addr
QFMP	addr	GI	Multiply QAC by 128-bit memory at addr
QFSB	addr	GI	Subtract 128-bit memory at addr from QAC

FLOATING POINT ROUNDING OPERATIONS

DRN	Round quad to double; store in bits 1 through 64 of QAC (see Instruction Sets Guide for rounding rules)
DRNM	Round quad to double towards minus infinity; store in bits 1 through 64 of QAC (see Instruction Sets Guide for rounding rules)
DRNP	Round quad to double towards plus infinity; store in bits 1 through 64 of QAC (see Instruction Sets Guide for rounding rules)
DRNZ	Round quad to double towards zero; store in bits 1 through 64 of QAC (see Instruction Sets Guide for rounding rules)
FRN f	Round double to single; store in bits 1 through 48 of DACf (see Instruction Sets Guide for rounding rules)
FRNM f	Round double to single towards minus infinity; store in bits 1 through 48 of DACf (see Instruction Sets Guide for rounding rules)
FRNP f	Round double to single towards plus infinity; store in bits 1 through 48 of DACf (see Instruction Sets Guide for rounding rules)
FRNZ f	Round double to single towards zero; store in bits 1 through 48 of DACf (see Instruction Sets Guide for rounding rules)

Character and Field Operations: I-mode programs can operate on characters and character strings (fields) with the aid of the field address registers FAR0 and FAR1 and the field length registers FLR0 and FLR1. These are the same registers used in the decimal operations described earlier in this chapter. Field registers operate in pairs, FAR0/FLR0 and FAR1/FLR1.

The discussion below summarizes the character and field operations; detailed information can be found in the System Architecture Reference Guide and in the Instruction Sets Guide. In this description, the terms character and byte are equivalent, and represent either the high-order or low-order eight bits of a halfword.

In operations in which only one pair of FAR/FLR registers is involved (such as LDC and STC), either pair (0 or 1) can be specified. For operations involving both pairs (such as ZED or ZMV) FAR0/FLR0 always represents the source string and FAR1/FLR1 always represents the destination string.

Before and during a character or field operation the FAR contains the address of the next character to be operated on, while the FLR contains the number of characters yet to be processed. Both registers must be preloaded with the appropriate address and length information before the operation can begin. Use the EAFA instruction to load the beginning address of the field into FAR, and the LFLI instruction to load the field length (in bytes) into FLR. (See the instruction summaries below, and the detailed descriptions in the Instruction Sets Guide.)

After execution of the EAFA instruction, the FAR contains a segment/offset address and the FLR contains the character offset (see the next paragraph) within the halfword at the segment/offset location.

After execution of the LFLI instruction, bits 44 through 64 of the FLR contain the length of the field (the number of characters) to be operated on. In addition to the field length information, the FLR contains a bit field (in bits 1 through 4) which, during the character or field operation, is updated to point to the next character to be processed. The bit field can thus be considered a four-bit extension of the segment/offset address in FAR. The value in the bit field at any given time is either '0000'b or '1000'b (0 or 8 decimal), representing the left or right byte, respectively, at the segment/offset location. Each time the bit field is updated to 0, the offset part of the address is incremented by 1.

The initializing instructions are coded as shown below. far or flr represents the field register pair number (0 or 1).

```
EAFA far,addr       initialize for left byte at addr    OR
EAFA far,addr+8B    initialize for right byte at addr

LFLI flr,number     initialize field length in bytes
```

For the edit and translate (ZED and ZTRN) instructions, an additional setup operation is necessary: the address of the beginning of the edit control subprogram (for ZED) or the beginning of the translation table (for ZTRN) must be loaded into the auxiliary base (XB) register. The EAXB instruction is used for this purpose (see Memory/Register Transfer Operations, earlier in this chapter).

Character and field operations are of two types; those that process one character per invocation of an instruction (LDC and STC), and those that operate on an entire field with one invocation of an instruction (such as ZED, ZFIL, ZMV). LDC and STC can be coded in a loop to process consecutive characters one at a time. The loop is typically

repeated by a branch on condition code not equal (BCNE); this is possible because the LDC or STC instruction decrements the length field in FLR for each character processed, and sets the condition codes to equal when the count reaches zero.

The following example illustrates a possible method of transferring an eight-character string, one character at a time, from one memory field to another, using both sets of field registers and general register 3.

```
        EAFA   0,FROM
        LFLI   0,8
        EAFA   1,TO
        LFLI   1,8
LOOP    LDC    0,3
        STC    1,3
        BCNE   LOOP
         .       .
         .       .

        LINK
FROM    DATA   'ABCDEFGH'
TO      BSS    4
         .       .
```

The BCNE instruction tests only the most recent setting of the condition codes. In the example above, this is the condition set by the STC instruction; the conditions set by the LDC instruction are lost.

The field operations other than LDC and STC do not need programmed loops to operate on fields; their looping is internal to the instruction, and the condition codes at their termination are usually indeterminate.

Character and field operations use several of the general registers during their execution. It is the program's responsibility to save and restore these registers when necessary.

The following groups summarize the instructions used in character and field operations. The designations far and flr represent the number (0 or 1) of the field register pair.

FIELD REGISTER OPERATIONS

ARFA far,r	Add contents of GRr to FARfar
EAFA far,addr	Load memory address into FARfar
LFLI flr,n	Load n into FLRflr
STFA far,addr	Store contents of FARfar into memory (stores 32 or 48 bits -- see Instruction Sets Guide)
TFLR flr,r	Transfer contents of FLRflr to GRr
TRFL flr,r	Transfer contents of GRr to FLRflr; maximum allowable number is 2**20 (the number of bits in a 64K segment)

CHARACTER AND FIELD OPERATIONS

LDC flr,r	If FLRflr is non-zero, load character pointed to by FARflr into bits 9 through 16 of the high half of GRr; else set condition codes equal
STC flr,r	If FLRflr is non-zero, store bits 9 through 16 of the high half of GRr in location pointed to by FARflr; else set condition codes equal
ZCM	Compare fields at FAR0 (F1) and FAR1 (F2) if F1 > F2, set condition codes GT if F1 = F2, set condition codes EQ if F1 < F2, set condition codes LT
ZED	Edit character field (see Instruction Sets Guide for edit control information)
ZFIL	Fill field starting at FAR1 with the character in bits 9 through 16 of GR2; FLR1 specifies length of string to fill
ZMV	Move field starting at FAR0 to field starting at FAR1; FLRs define lengths of fields (see Instruction Sets Guide for treatment of unequal length fields)
ZMVD	Move field starting at FAR0 to equal length field starting at FAR1; FLR1 defines length of fields
ZTRN	Translate field starting at FAR0 and store in field starting at FAR1; FLR1 defines length of fields, XB contains beginning address of translation table (see Instruction Sets Guide for the translation algorithm)

Process-Related Operations

The instructions in this group are concerned with various aspects of the control of a process and its related procedures. Chapters 8, 9, and 10 of the System Architecture Reference Guide discussion processes and procedures in detail.

Only summary lists of these instructions are presented in this chapter; the Instruction Sets Guide goes into further detail on each one.

ADDRESS MODE CHANGE OPERATIONS

E16S	Enter 16S address mode
E32I	Enter 32I address mode
E32R	Enter 32R address mode
E32S	Enter 32S address mode
E64R	Enter 64R address mode
E64V	Enter 64V address mode

INTER-PROCECURE TRANSFER OPERATIONS

ARGT	Argument transfer
CALF addr	Call fault handler whose ECB is at addr
PCL addr	Call procedure whose ECB is at addr
PRTN	Procedure return
STEX	Stack extend

QUEUE MANAGEMENT OPERATIONS

ABQ r,addr	Add entry in high half of GRr to bottom of queue pointed to by addr
ATQ r,addr	Add entry in high half of GRr to top of queue pointed to by addr
RBQ r,addr	Remove from bottom of queue pointed to by addr and store in high half of GRr
RTQ r,addr	Remove from top of queue pointed to by addr and store in high half of GRr
TSTQ r,addr	Set high half of GRr to number of items in queue pointed to by addr

MISCELLANEOUS OPERATIONS

HLT	If not in ring 0, simulate a processor halt and display a message
NOP	No operation; proceed to next instruction
SSSN	Store system serial number in memory block specified by XB register
STTM	Store process timer in memory specified by XB register
SVC	Supervisor call

Restricted Instructions

The instructions in this group deal mainly with the manipulation of system data structures that are essential to PRIMOS operation, and are therefore protected against access by the casual user. They can be executed by users who have access to ring 0. Refer to Chapter 5 of the System Architecture Reference Guide for further information on these instructions.

Only summary lists of these instructions are presented in this chapter; the Instruction Sets Guide goes into further detail on each one.

INTERRUPT HANDLING OPERATIONS

ENB	Enable interrupts
ENBL	Enable interrupts (local)
ENBM	Enable interrupts (mutual)
ENBP	Enable interrupts (process)
INBC addr	Interrupt notify beginning; clear active interrupt
INBN addr	Interrupt notify beginning
INEC addr	Interrupt notify end; clear active interrupt
INEN addr	Interrupt notify end
INH	Inhibit interrupts
INHL	Inhibit interrupts (local)
INHM	Inhibit interrupts (mutual)
INHP	Inhibit interrupts (process)
IRTC	Interrupt return; clear active interrupt
IRTN	Interrupt return

INPUT/OUTPUT OPERATION

EIO addr	Execute I/O

ADDRESS TRANSLATION OPERATIONS

ITLB	Invalidate STLB entry
LIOT addr	Load IOTLB
PTLB	Purge TLB

PROCESS EXCHANGE OPERATIONS

LPID	Load Process ID register from bits 1 through 10 of GR2
LPSW addr	Load program status word from memory

SEMAPHORE OPERATIONS

NFYB addr	Notify semaphore at addr; use LIFO queuing
NFYE addr	Notify semaphore at addr; use FIFO queing
WAIT addr	Wait on semaphore at addr

MISCELLANEOUS OPERATIONS

RMC	Reset machine check flag to 0
RTS	Reset time slice
STPM	Store processor model number and microcode revision number in memory block specified by XB register

10
Machine Instructions -- IX Mode

32 IX mode (commonly referred to simply as IX mode) comprises a small set of instructions, in addition to those described for I mode in Chapter 9, that are executable on the 2550, 9650, 9750, 9950, 9955, and 9955II processors. These instructions permit the initialization and modification of indirect pointers and provide an interface between C language programs and assembly language routines that have to manipulate C language pointers.

INDIRECT POINTER-RELATED INSTRUCTIONS

Two IX-mode instructions, LIP (Load Indirect Pointer) and AIP (Add Indirect Pointer), enable a program to load the contents of an indirect pointer into a general register and to add to or subtract from the value of the pointer stored in the register. Memory locations relative to the indirect pointer can then be addressed by instructions that allow the general register relative format, described in Chapter 9.

For example, assume that a series of identically-structured data areas are to be processed. Each structure contains 40 16-bit halfwords, and in each structure, the contents of the fourth and tenth halfwords are to be added together and stored in the 28th halfword. This sequence could be coded as follows:

```
          LIP    1,STRUC_AD     load GR1 for 1st structure
LOOP      LH     2,R1%+3        load 4th halfword into GR2
          AH     2,R1%+9        add 10th halfword to GR2
```

```
            STH     2,R1%+27        store sum into 28th halfword
            AIP     1,FORTY         increment for next structure
            C       1,STRC_END      test for loop end
            BCNE    LOOP            repeat loop
             .          .
             .          .

            LINK
STRUC_AD    IP      STRUC_1         address of 1st structure
FORTY       DATA    40L             size of each structure
STRUC_1     BSZ     400             ten 40-character structures
STRC_END    IP      *               address of end of structure
             .          .
             .          .
```

The data area STRUC_1 is assumed to have been previously loaded with data from some source such as an external file containing the 40-~~character~~ half-word structures. The test for the end of the loop compares the current contents of GR1 with the contents of the IP following the structure (STRC_END).

The LIP instruction loads the 32-bit contents of the indirect pointer STRUC_AD in general register 1. The pointer consists of both a segment number and an offset. All processing within the structure is done through GR1 by instructions in the general register relative format. The number by which the register is incremented by the AIP instruction must be a long (32-bit) integer; the increment is applied to the offset portion of the address in the register. The indirect pointer itself is not modified.

The formats of the two indirect pointer-related instructions are shown below. The G indicates that the general register relative form of the instruction is permitted. Refer to the Instruction Sets Guide for details of the operation of these instructions.

INDIRECT POINTER OPERATIONS

AIP r,addr	G	Add 32-bit addr to contents of GRr
LIP r,addr		Load 32-bit indirect pointer at addr into GRr

C LANGUAGE-RELATED INSTRUCTIONS

Seven IX-mode instructions enable a program to load and store characters and manipulate pointers on behalf of a C language program.

A C language pointer differs from an ordinary indirect pointer in that it contains a bit that determines whether a character is being loaded from or stored into the left or right byte of a halfword. Refer to the discussion of the C language pointer in Chapter 3 of the System Architecture Reference Guide for more information on this pointer

format.

The general method for handling the C language pointer is to first load the pointer into a general register using the LIP instruction described in the previous section. (Do not use GR0 for this purpose; the assembler will display an error message.) Then use the LCC and SCC instructions to load and store characters, and the ICP and DCP instructions to increment and decrement from one character to the next. The sample program below uses these instructions to load A into GR4 and B into GR6.

```
            SEGR
ST          NOP
            LIP     7,CHAR_AD
            LCC     4,R7%
            ICP     7
            LCC     6,R7%
            PRTN
            LINK
CHAR_AD     IP      CHARS
CHARS       DATA    C'ABCD'
ECB$        ECB     ST
            END     ECB$
```

Additional instructions enable addition to or subtraction from the pointer (ACP), comparing the pointer value to some other value (CCP), and testing for a null pointer (TCNP). All of these instructions are described in detail in the Instruction Sets Guide. The instructions are summarized below. An R indicates that the instruction can be coded in register to register format; a G indicates general register relative format; an I indicates immediate format. All are described in Chapter 9.

C LANGUAGE-RELATED OPERATIONS

ACP d,s	RI	Add the two's-complement contents of GRs to the C pointer in GRd; result in GRd
CCP d,s	R	Compare the C pointer in GRs to the C pointer in GRd; if: GRd > GRs, set CC to GT GRd = GRs, set CC to EQ GRd < GRs, set CC to LT
DCP r		Decrement C pointer in GRr by one byte
ICP r		Increment C pointer in GRr by one byte
LCC r,addr	G	Load character at addr into bits 9 through 16 of GRr
SCC r,addr	G	Store character from bits 9 through 16 of GRr into addr
TCNP addr	R	Test C pointer at addr for null (bits 4 through 32 equal to zero); sets CC to EQ if null, else set CC to NE

11
Macro Facility

The macro facility enables you to define frequently used sequences of instructions, data, and pseudo-operations, and to invoke these sequences where required in an assembly language program. These sequences are known as macros; they are defined by macro definitions, and the statements used to invoke them are macro calls. Use of macros relieves the programmer from repetitious coding of the same instruction sequences.

For example, the macro call

```
    TRANSFER DATA TO SAVE
```

can be made to generate the instruction sequence

```
    LDA     DATA
    STA     SAVE
```

The name of the macro in this example is TRANSFER; a statement with a macro name in the operation field is a call to the macro having that name.

Once a macro function has been defined, you can call it any number of times within a program. You can supply argument values, such as DATA and SAVE in the above example, with each call. You can also add dummy words such as TO or FROM to increase readability. Dummy words are identified during macro definition and are not treated as arguments in a macro call.

The example below illustrates the TRANSFER macro definition, a call to it, and the code generated by the call. The discussion that follows the example describes each element.

```
            SEG
TRANSFER    MAC     TO              Begin definition; TO is a dummy word
<0>         SET     *               Provide for label on generated code
            LDA     <1>             Instruction using first argument
            STA     <2>             Instruction using second argument
            ENDM                    End definition
LABEL_1     TRANSFER  ADD_1 TO ADD_2            Call to TRANSFER
            LDA     ADD_1           Generated statement w/ 1st argument
            STA     ADD_2           Generated statement w/ 2nd argument
             .        .
             .        .
```

MACRO DEFINITION

A macro must be defined before it is called. It is good practice to define all macros before any of the main body of the program is coded, although this is not a requirement of the assembler.

Each macro definition begins with a MAC pseudo-operation. The MAC statement must have a label (TRANSFER in the example). This label is the macro name by which it is called. It can also have optional dummy words (TO) and argument identifiers in the operand field. (Dummy words and argument identifiers are described later in this chapter.) Statements that make up the macro definition follow, terminated by an ENDM pseudo-operation.

Argument References

Argument references are expressions enclosed within angle brackets (< >). Any part of a statement within a macro definition may contain an argument reference. The expression may contain symbols, integers, or both, provided any symbols can be evaluated as single-precision (16-bit) integers, and that the entire expression is reducible to a single-precision integer value. The example on the next page shows the use of symbols and constants in argument references.

```
            SEG
TRANSNO     MAC
            LDA     <J>
            STA     <K+2-J-1>
            LDX     <0>
<0>         SET     *
            DATA    C'XX'
            ENDM
J           EQU     1
K           EQU     2
MACNO       TRANSNO   AA  BB
            LDA     AA
            STA     BB
            LDX     MACNO
            PRTN
            LINK
AA          BSS     1
BB          BSS     2
            END
```

The expressions in the angle brackets are reducible to integer values because the symbols J and K are equated to integer values by the EQU statements.

A zero within angle brackets (or an expression that reduces to a value of zero) is replaced by whatever, if anything, is in the label field of the macro call statement. In the example above, the label field of the call to TRANSNO contains the label MACNO. This label replaces the <0> in the label field of the SET statement. (The SET statement does not appear in the generated code unless an LSTM pseudo-operation occurs somewhere before the call.) The MACNO label also replaces the <0> in the operand field in the LDX statement, and is resolved to its address value in the generated LDX instruction.

Use the <0> form of argument reference as a label field only in a SET or XSET statement. If it is used for an instruction or data element and the macro is called more than once, the assembler generates an error message (SYMBOL MULTIPLY DEFINED). The label that replaces the <0> in the SET or XSET statement is assigned the value and mode of the assembler's current location counter.

Assembler Attribute References

The assembler maintains a set of attributes that contain information vital to the progress of the assembly. Associated with each attribute is an attribute number and an attribute value. Most attributes are for the assembler's internal use, and hold such things as the current character pointer, statement line number, current program counter, and macro processing counters and flags.

Attributes are referenced by an attribute number preceded by the number sign character (#). The attribute number can be a symbol or an expression within parentheses, as long as any symbols have been previously defined as 16-bit integers and the expression represents a value of 0, 1, or 100 through 128 (decimal). The attribute value can be any 16-bit quantity, and can represent either a character or a numeric element, depending on how the assembler uses it.

Attribute references can be used both in macro definitions and in macro calls. An example of the use of attributes is the operation of the IFx (structured IF) pseudo-operation, described in Chapter 4.

A list of the assembler attributes, their numbers, and their meanings is given at the end of this chapter.

Local Labels Within Macros

Local labels, which do not conflict with labels outside of the macro, can be assigned and referenced within a macro definition by prefixing the label with the ampersand character (&). When the macro is called, the ampersand is replaced by a four-digit macro call number, thereby assuring the label's uniqueness regardless of the number of times the same macro is called. Each macro call within an assembly increments the macro call number by 1; the current call number is stored in assembler attribute #001.

The numeric prefix does not appear in the listing, but the label may appear more than once, depending on the number of times the macro is called.

Examples:

Assigned Local Label	Evaluated As	In Macro Call No.
&ABC	0002ABC	0002
&X3A	1739X3A	1739

MACRO CALLS

A macro call is a statement that uses as its operation code the name of a predefined macro:

```
                          argument, ...
[label]  macro-name       dummy-word, ...
                          argument-identifier, ...
```

For each macro call, the assembler generates the code contained in the macro definition, starting at the current location. This is known as

expanding the macro call. Argument references in the macro definition are replaced by argument values from the macro call's operand field.

Several calls to the same macro may not, for each call, generate the same code. The macro definition may contain blocks of code that are conditionally assembled (see the discussions of conditional assembly in Chapter 4 and the SCT and SCTL pseudo-operations in Chapter 7). Whether or not these blocks are generated can depend on, for example, the value of a call argument or the contents of an assembler attribute.

Argument Values

The operand 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 operand 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. Parentheses can be used in forming sublists of arguments for macro calls nested within another macro definition. See NESTING MACROS, later in this chapter.

Argument Substitution

During expansion of a macro call, the assembler substitutes the argument values in the macro call operand 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 argument expression in the macro call replaces argument reference 1, the second, reference 2, and so on. Some examples are shown below.

Argument Field	Argument <1>	Argument <2>	Argument <3>
A	A	0	0
A+3	A+3	0	0
X,Y-1,Z*A-1	X	Y-1	Z*A-1
X,B-C (Z3X2)	X	B-C	Z3X2
(A, B-1), C	A, B-1	C	0
(X, Y, (Z1+Z2),3)	X,Y,(Z1+Z2),3	0	0

In a call to the TRANSFER macro such as

```
TRANSFER  ARG1, ARG2+3
```

the variable ARG1 is argument 1 and the expression ARG2+3 is argument 2. Thus, the TRANSFER macro call shown would be assembled as:

```
LDA  ARG1
STA  ARG2+3
```

Argument references in a macro definition that do not have corresponding argument values in a macro call are set to zero by the assembler.

Using Macro Calls as Documentation

An ordinary macro call such as

```
TRANSFER  ARG1, ARG2
```

although functionally complete, does not provide, from a documentary point of view, a complete description of its operation. It is not possible, by looking at the call, to tell in which direction the transfer occurs. By using additional words in the operand field of a macro call, the programmer can show more precisely the nature of the function. Macro calls can be made self-describing by a combination of meaningful argument symbols such as SOURCE, TARGET, and MESSAGE; 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 serve to link arguments with argument references in some sequence other than the default positional order.

Dummy Words: Dummy words applicable to a given macro are defined in the operand field of the MAC statement that starts the macro definition. For example:

```
TRANSFER   MAC  FROM   TO
             .
             .
             .
             .
           ENDM
```

In the above example, FROM and TO are dummy words. In any subsequent call to this macro, the assembler ignores the words FROM and TO; all other expressions in the operand 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. For example, when the TRANSFER macro is called by

```
TRANSFER FROM ALPHA TO BETA
```

the assembler ignores the FROM and TO, and assembles the macro as if the call statement were

```
TRANSFER ALPHA, BETA
```

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 can be used. If the first character of a dummy word string is a left parenthesis, all characters, including spaces and commas, up to the matching right parenthesis are considered part of the dummy word. The parentheses are not considered part of the string.

Argument Identifiers: While the self-describing effect of dummy words improves the description of a macro call, 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 an argument number with a dummy word, regardless of the order in which arguments occur in the call. 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 operand field of a MAC statement that starts a macro definition. To define an argument identifier, set a dummy word, in parentheses, equal to the desired argument number:

```
TRANSFER    MAC   (FROM)=1,   (TO)=2
            .
            .
            ENDM
```

When a call to the macro uses an argument identifier in its operand field, the first nondummy expression immediately following the identifier is taken as the value of the argument:

```
TRANSFER FROM ALPHA TO BETA
TRANSFER TO BETA FROM ALPHA
```

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 can be intermixed in the same call. Ordinary dummy words are ignored, as described previously.

Arguments that are not associated with identifier words receive values in the usual positional order. For example, the macro defined by:

```
MASK    MAC (BY)=2,  (IN)=3, STORE, AND
        LDA   <1>
        ANA   ='<2>
        STA   <3>
        ENDM
```

can be called by

```
        MASK   INPUT BY 17 AND STORE IN BUFF1.
```

Using the identifier words BY and IN, argument 2 is given a value 17 and argument 3 is given the value BUFF1. The only remaining element in the call is INPUT, so it is assigned to the first argument reference that is not associated with an identifier, which is argument reference <1>.

NESTING MACROS

Macro definitions may contain calls within them, as in the following example:

The WAIT1 macro, which calls another macro, TRANSFER, is defined by:

```
WAIT1   MAC
        LDX      =<1>
        BDX      *-1
        TRANSFER <2>
        ENDM
```

WAIT1 is called by

```
WAIT1   100, (ARG1, ARG2)
```

It is assembled as:

```
LDX     =100
BDX     *-1
LDA     ARG1
STA     ARG2
```

The WAIT1 macro interprets the list in parentheses as a single argument (argument 2). During the expansion of WAIT1, the two elements replace WAIT1's argument reference <2>, but without the parentheses, thus passing two arguments to the call to TRANSFER.

Macro definitions, while they can contain calls to other macros, cannot contain other macro definitions. Further, a called macro must be defined before the macro that calls it.

CONDITIONAL ASSEMBLY

There are a number of pseudo-operations which allow the programmer to conditionally include or omit parts of a macro during expansion. Conditional assembly pseudo-operations are discussed in Chapter 4. See also the discussion of the SCT and SCTL statements in Chapter 7.

MACRO LISTING

Three pseudo-operations provide different levels of listing detail for macro calls and the resulting generated statements.

LSTM Lists macro statements and all lines generated by expansion of the macro: instructions, data, and macro related pseudo-operations.

LSMD Lists macro call statements and lines that generate instructions, data, and non-macro pseudo-operations; this is the default.

NLSM Lists only the macro call statement; no generated code or pseudo-operations are listed.

Each of these pseudo-operations remains in effect until a new macro listing control pseudo-operation is specified.

ASSEMBLER ATTRIBUTE LIST

The following assembler attributes are valid as of PRIMOS Revision 21.0:

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

12
Using Subroutines

A subroutine is a block of executable code to which a program can transfer control to perform some function, and from which control is returned to the program when the function is completed. Transferring control to the subroutine is known as calling the subroutine, and when the subroutine completes its function, it returns to the program that called it, usually to the instruction following the call. The calling and called routines are often referred to as simply the caller and the callee.

Subroutines and subroutine calls can be implemented in several ways in assembly language programs. A subroutine can be local to the caller; that is, it is contained in the same assembly module as its caller. Local subroutine code resides between the same SEG or SEGR and END statements as the code that calls it, and is assembled each time the calling code is assembled. Nonlocal, or external, subroutines, on the other hand, reside in separately assembled modules and need not be assembled each time their callers are assembled.

LOCAL SUBROUTINES

There are several ways of calling and returning from a local subroutine. The principal difference in call and return methods lies in the location in which the return address is stored by the caller at the time of the call, and the way in which the callee uses that location at the time of the return.

Local Calls in V Mode

Calls to local subroutines in V Mode are made by one of the four jump instructions that, at the time of the jump, store the address of the instruction following the jump at some location known to the subroutine. These are the JST, JSX, JSY, and JSXB instructions (see Chapter 8). Each is illustrated in the following sections by examples that call a subroutine, SQUARE, that calculates the squares of two numbers and returns the results in the A register. The numbers to be squared are passed to the subroutine also in the A register.

JST Call: The JST (Jump and Store) instruction stores the address of the next instruction in a 16-bit halfword at the target address, and then transfers control to the location following that halfword.

```
          LDA   =25
          JST   SQUARE
          STA   X25
          LDA   =13
          JST   SQUARE
          STA   X13
SQUARE    DAC   **       return address stored here
          STA   TEMP     control transferred to here
          MPY   TEMP
          PIMA
          JMP#  SQUARE,* return indirectly via address in SQUARE;
          LINK
X25       BSS   1
X13       BSS   1
TEMP      BSS   1
```

The JST call is perhaps the most straightforward of all of the calling mechanisms, but it is unsuited for use in pure procedure segments, because it alters a location (the DAC at SQUARE) in the procedure segment. If this method is used, the procedure must be declared impure by using the IMPURE operand of the SEG pseudo-operation (see Chapter 4).

JSX Call: The JSX (Jump and Store in X Register) instruction stores the address of the instruction following the jump in the X register, and then transfers control to the target address.

```
        LDA   =25
        JSX   SQUARE
        STA   X25
        LDA   =13
        JSX   SQUARE
        STA   X13
SQUARE  STA   TEMP    control transferred to here
        MPY   TEMP
        PIMA
        JMP   PB%,X   return via PB register indexed by X
        LINK
X25     BSS   1
X13     BSS   1
TEMP    BSS   1
```

The JSX call can be used in pure procedure segments because the X register is independent of any segment; nothing in the procedure segment is modified by storing a return address in this register.

If the X register is used in a subroutine's processing, the subroutine should begin by saving the register contents with an STX instruction, and end by restoring it with an LDX instruction just before the return JMP. Alternatively, a JSY or JSXB call can be used, as described below.

JSY Call: The JSY (Jump and Store in Y Register) instruction stores the address of the instruction following the jump in the Y register, and then transfers control to the target address.

```
        LDA   =25
        JSY   SQUARE
        STA   X25
        LDA   =13
        JSY   SQUARE
        STA   X13
SQUARE  STA   TEMP    control transferred to here
        MPY   TEMP
        PIMA
        JMP   PB%,Y   return via PB register indexed by Y
        LINK
X25     BSS   1
X13     BSS   1
TEMP    BSS   1
```

The JSY call can be used in pure procedure segments because the Y register is independent of any segment; nothing in the procedure segment is modified by storing a return address in this register.

If the Y register is used in a subroutine's processing, the subroutine should begin by saving the register contents with an STY instruction,

and end by restoring it with an LDY instruction just before the return JMP. Alternatively, a JSXB call can be used, as described below.

JSXB Call: The JSXB (Jump and Store in Auxiliary Base Register) instruction stores the address of the instruction following the jump in the XB register, and then transfers control to the target address.

```
          LDA   =25
          JSXB  SQUARE
          STA   X25
          LDA   =13
          JSXB  SQUARE
          STA   X13
SQUARE    STA   TEMP     control transferred to here
          MPY   TEMP
          PIMA
          JMP   XB%      return via XB register
          LINK
X25       BSS   1
X13       BSS   1
TEMP      BSS   1
```

The JSXB call can be used in pure procedure segments because the XB register is independent of any segment; nothing in the procedure segment is modified by storing a return address in this register.

If the XB register is used in a subroutine's processing (for example, by a ZED or ZTRN instruction, or a call to a nonlocal subroutine), the subroutine must save the XB register contents before using XB, and restore it before the return JMP. Saving the XB register, in turn, uses the L register, so arguments passed in the A, B, or L register must also be saved. A possible sequence for register preservation is:

```
STL   TEMP_1   save argument if passed in A, B, or L register
LDLR  '17      save XB in TEMP_2
STL   TEMP_2     via L register
LDL   TEMP_1   restore argument to L register
 .      .
 .      .      A, B, L, and XB can now be used
 .      .        to process the argument

STL   TEMP_1   save subroutine result if returned in A, B, or L
 .      .
 .      .      additional processing that may involve A, B, or L,
 .      .        such as a call to a nonlocal procedure

LDL   TEMP_2   restore XB from TEMP_2
STLR  '17        via L register
LDL   TEMP_1   restore result in L
JMP   XB%      return
```

Note

The assembler displays an error message on the LDLR and STLR instructions. However, the correct code is generated, and the subroutine executes correctly.

An alternate method of return is to use an indirect jump via TEMP_2:

```
JMP    TEMP_2,*
```

This removes the need for the LDL and STLR instructions used to restore XB from TEMP_2.

The temporary storage areas TEMP_1 and TEMP_2 must both be two halfwords long.

Local Calls in I Mode

Calls to local subroutines in I Mode are made by one of the two jump instructions that, at the time of the jump, store the address of the instruction following the jump at some location known to the subroutine. These are the JSR and JSXB instructions (see Chapter 9). Each is illustrated in the following sections by examples that call the SQUARE subroutine to calculate the squares of two numbers and return the results in a general register. The numbers to be squared are passed to the subroutine in the same general register.

JSR Call: The JSR (Jump and Store in General Register) instruction stores the address of the next instruction in a general register, and then transfers control to the target address.

```
          LH    2,=25
          JSR   1,SQUARE   store return in GR1
          STH   2,X25
          LH    2,=13
          JSR   1,SQUARE
          STH   2,X13
SQUARE    STH   2,TEMP     control transferred to here
          MH    2,TEMP
          PIMH  2
          JMP   PB%,1      return via PB indexed by GR1
          LINK
X25       BSS   1
X13       BSS   1
TEMP      BSS   1
```

The JSR call can be used in pure procedure segments because the general registers are independent of any segment; nothing in the procedure segment is modified by storing a return address in a general register.

Note that the general register used for the return cannot be GR0, since only GR1 through GR7 can be used for indexing. GR0 can, however, be used for general-purpose temporary storage within the subroutine. Also, be aware of the registers used in the subroutine, and avoid, if possible, using one of these for the return; if this is unavoidable, save and restore the return register using STH and LH instructions.

JSXB Call: The JSXB (Jump and Store in Auxiliary Base Register) instruction stores the address of the instruction following the jump in the XB register, and then transfers control to the target address.

```
          LH    2,=25
          JSXB  SQUARE
          STH   2,X25
          LH    2,=13
          JSXB  SQUARE
          STH   2,X13
SQUARE    STH   2,TEMP     control transferred to here
          MH    2,TEMP
          PIMH  2
          JMP   XB%        return via XB register
          LINK
X25       BSS   1
X13       BSS   1
TEMP      BSS   1
```

The JSXB call can be used in pure procedure segments because the XB register is independent of any segment; nothing in the procedure segment is modified by storing a return address in this register.

If the XB register is used in a subroutine's processing (for example, by a ZED or ZTRN instruction or a CALL pseudo-operation), the subroutine should begin by saving the XB register contents in an unused register r with an LDAR r,'17 instruction, and end by restoring it with an STAR r,'17 instruction just before the return JMP. Alternatively, a JSR call can be used, as described previously.

EXTERNAL SUBROUTINES

Prime supplies a large number of standard subroutines for use by programs written in any of its programming languages. These subroutines are grouped into libraries, described in the Programmer's Guide to BIND and EPFs, and in Volume I of the Subroutines Reference Guide.

The standard subroutines are written in assembly language or any of several high-level languages. They are assembled or compiled independently of any program that calls them, and are therefore, from the point of view of the calling program, nonlocal, or external subroutines.

External subroutines can also be user-written, assembled, and collected into user libraries. Writing and building subroutine libraries is discussed in detail in Volume I of the Advanced Programmer's Guide.

The assembler, when assembling a calling program, cannot resolve references to entry points and other locations within an external procedure. It is the function of Prime's linkers, BIND and SEG, and the dynamic linking mechanism to do this. The calling program uses the EXT pseudo-operation to identify external locations to be resolved after assembly is completed.

The linker locates the external modules by means of a set of search rules and entrypoint lists. Prime provides a set of search rules that include all of the standard libraries. These rules can, however, be modified to suit the user's particular needs; a search rule can, for example, be deleted if the associated library is never used, or one can be added to accommodate a library of user-written subroutines. A complete description of the use of search rules appears in Volume II of the Advanced Programmer's Guide.

External Calls

The standard method of calling external subroutines is through the CALL pseudo-operation, which performs several implicit functions. The calling mechanism is the same for both V mode and I mode. The CALL statement:

- Generates an implied EXT statement to identify its target as an entrypoint to a procedure outside the calling program
- Generates an explicit PCL (procedure call) instruction to effect the transfer of control
- Provides an indirect pointer (IP) in which to place the address of the external entrypoint at execution time

The procedure call mechanism is described in detail in Chapter 8 of the System Architecture Reference Guide; indirect pointers are filled in during either linking or execution, as described in Volume I of the Advanced Programmer's Guide.

Entrypoints to Called Routines

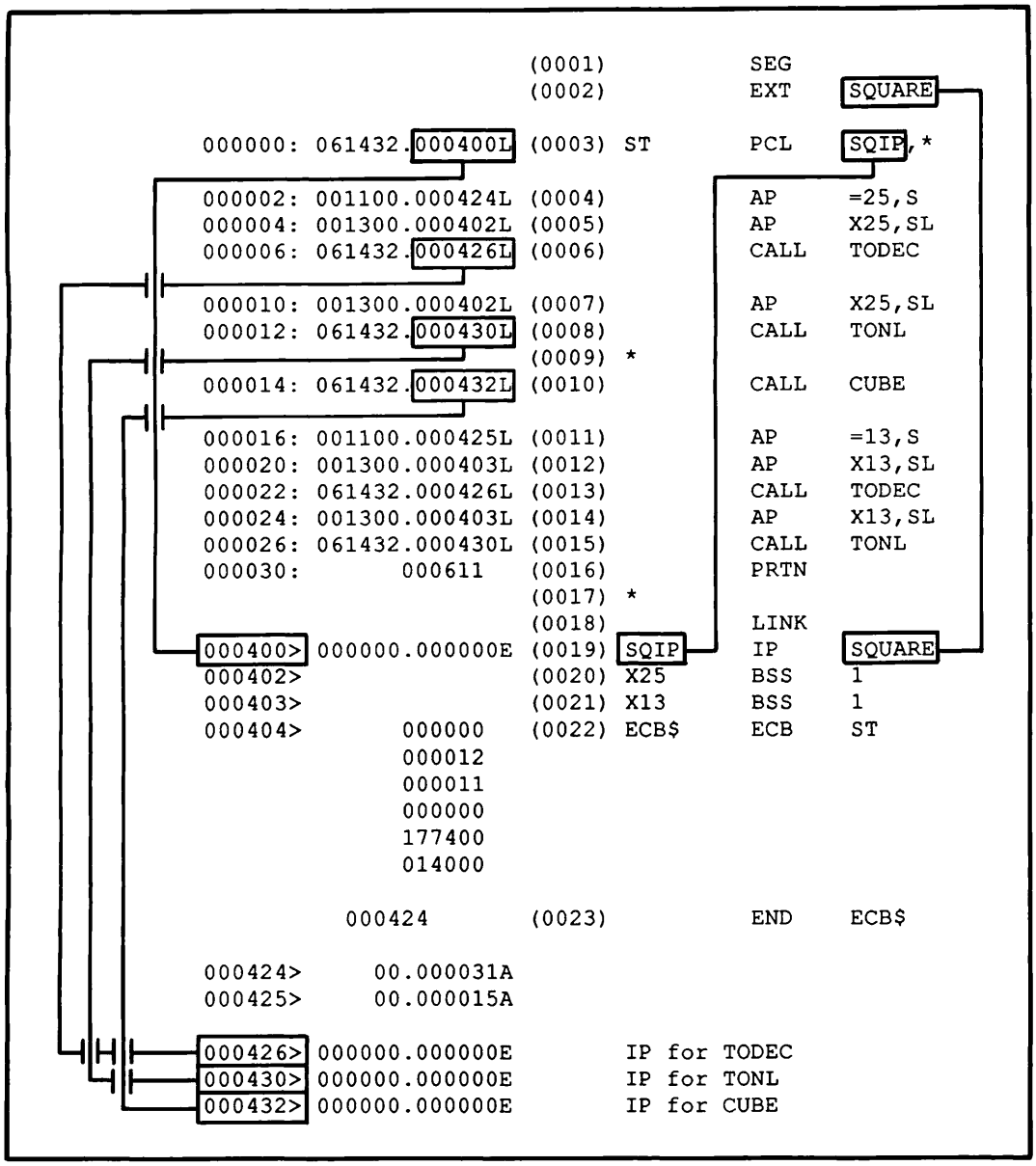
The primary requirement that must be met in order that one program may transfer control to (or call) another is that the caller must know, at execution time, the address of the first executable instruction in the called routine. In local subroutines, this information can be provided by the assembler, since all locations are known at assembly time. In nonlocal cases, the callees' addresses are not known at the time the caller is assembled; they must be provided by other means: by way of either external symbol and entrypoint declarations for resolution by the linker, or dynamic entrypoints for resolution by the dynamic linking mechanism. Only the former are described here; the latter are discussed in Volume I of the Advanced Programmer's Guide.

External Symbols in a Calling Program: An external symbol, that is, one that exists in a called program, is identified to a calling program by an EXT pseudo-operation (see Chapter 6). This symbol is the same as that used as the operand of the CALL pseudo-operation that calls the external program.

If, instead of a CALL statement, a PCL instruction is used to call an external program, both the EXT and the IP statements must be explicitly coded. The IP operand must be the same as the EXT operand; the PCL operand must reference the label of the IP statement indirectly. Since these functions are implicitly performed by the CALL statement, the CALL is the simpler and less error-prone of the two methods. Figure 12-1 shows an assembled program that calls two user-written subroutines SQUARE and CUBE, and two Prime-supplied subroutines TODEC and TONL. The SQUARE subroutine is called by the PCL method; the rest use the CALL statement.

To summarize, the calling program must do three things that the CALL pseudo-operation does implicitly. It must

- Provide an explicit EXT statement identifying the subroutine's entrypoint as a location external to the calling program, as shown in line 2 of Figure 12-1.
- Provide an explicit labeled indirect pointer (IP) whose operand matches that of the EXT statement. This is the statement whose label is SQIP (line 19). Note that it is coded in the Link segment.
- Code the operand of the PCL instruction (line 3) as an indirect reference to the label of the indirect pointer at line 19.



Assembly Language CALL vs. PCL Mechanism
Figure 12-1

All of these details are handled automatically when you use a CALL pseudo-operation. The assembler holds the EXTs internally as indicators that it must generate IPs for the calls; the EXTs are not listed.

The IPs for the CALL statements are generated as shown at the bottom of the listing. Note that they, too, are in the Link segment, as indicated by the > sign appended to their storage addresses.

Entrypoint Declaration in a Called Program: The resolution of a caller's external symbols is completed from the callee's side by a combination of the ENT or SUBR pseudo-operation and the callee's ECB. The key function is to provide a correlation between the entrypoint name and the ECB, which contains information necessary to the callee's execution.

The ENT is used in this discussion; the SUBR statement could be used with identical results. In the ENT pseudo-operation

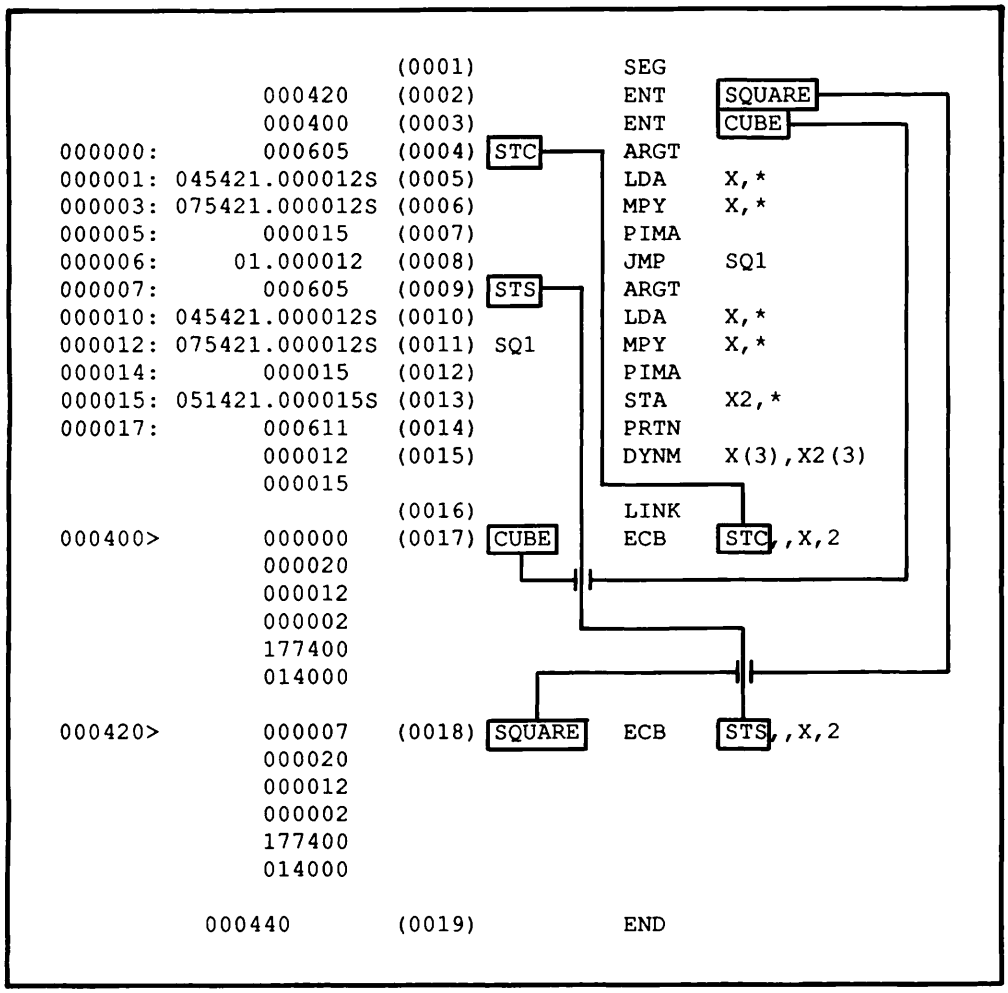
```
[label] ENT symbol-1[,symbol-2]
```

symbol-1 is the external name used by the calling program; it is this symbol that enables the linker to resolve the external symbol of the same name in the calling program. In addition to matching that symbol, it provides the entrypoint-to-ECB correlation in one of three ways:

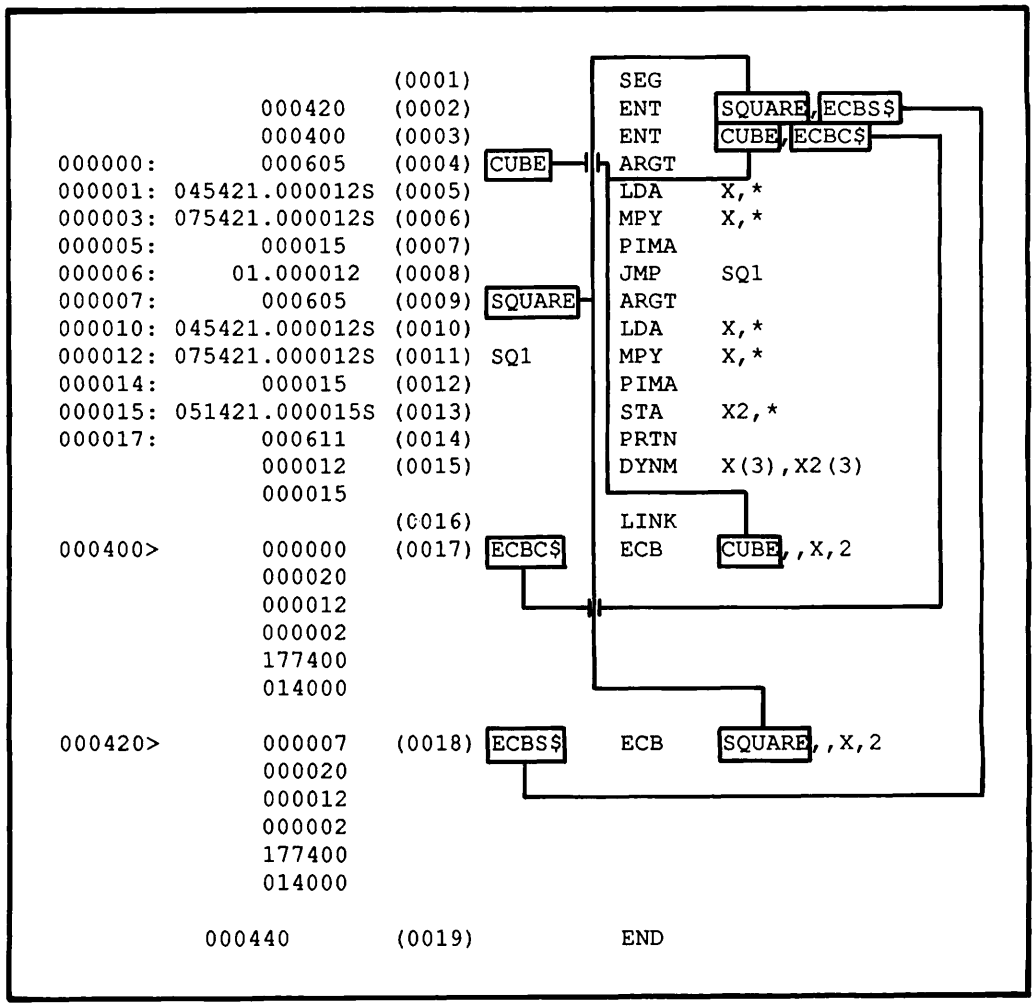
- It can match the name given in the label field of the called program's ECB. In this case, symbol-2 is not used. The ECB's starting address operand contains the label of the first executable instruction. Figure 12-2 shows this configuration.

- It can match the name given in the label field of the called program's first executable instruction. In this case, symbol-2 is required, and matches the name given in the label field of the called program's ECB. The ECB's starting address operand must also contain the name given in symbol-1. This configuration is shown in Figure 12-3.

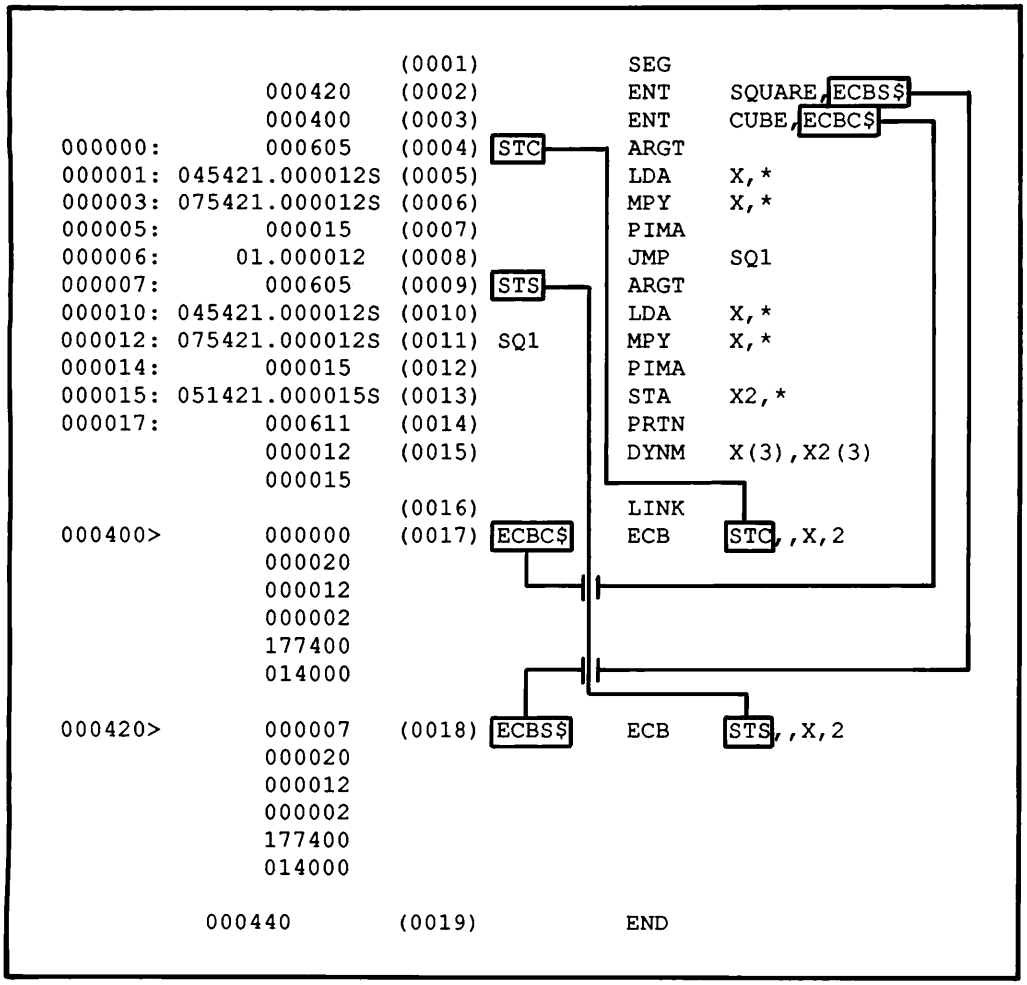
- It can match neither of the above; it simply identifies symbol-1 as an entrypoint. In this case, symbol-2 is required, and matches the name given in the label field of the called program's ECB. The ECB's starting address operand contains the label of the first executable instruction. Figure 12-4 illustrates this option.



Entrypoint Declaration -- Option 1
Figure 12-2



Entrypoint Declaration -- Option 2
Figure 12-3



Entrypoint Declaration -- Option 3
Figure 12-4

The relationship among the ENT operands, the first executable instruction label, and the ECB is summarized below.

	ENT	First Inst Label	ECB	
ENT	SYM_1	START	SYM_1 ECB	START ...
ENT	SYM_1,SYM_2	SYM_1	SYM_2 ECB	SYM_1 ...
ENT	SYM_1,SYM_2	START	SYM_2 ECB	START ...

Note that in Figures 12-2, 12-3, and 12-4, each example shows a subroutine with two entry points, SQUARE and CUBE. These entrypoints match the external labels declared or implied in the calling program whose code is shown in Figure 12-1.

Argument Passing in External Calls

In the examples referred to in the previous section, each entrypoint has a matching ECB, which supplies, in addition to the starting location, the location of the pointer to the first passed argument and the number of arguments. Refer to the descriptions of the ECB statement in Chapter 6, and the DYNM statement in Chapter 4 for additional information.

Argument passing is initiated by the PCL or CALL statement in the calling program; it is completed by the ARGT instruction in the called subroutine. The ARGT must be the first executable instruction in any subroutine that expects arguments. It must not be used in a subroutine that expects no arguments; that is, a subroutine that is called without any APs following it, such as the TONL call in Figure 12-1.

Returning from an External Call

Part of the function of the call and return mechanism is to allocate temporary storage in a stack area for use by the called routine, and to deallocate this storage at the subroutine's completion. The stack area contains the argument pointers, and can also accommodate any other temporary data that may be needed for the callee's execution.

Any subroutine that is called by a PCL or CALL statement with the expectation of returning to its caller must return by executing a PRTN instruction. The PRTN instruction, in addition to returning to the caller, deallocates the callee's stack space and returns it to the system for use by other users.

THE SHORTCALL MECHANISM

As stated previously, the PCL/PRTN mechanism performs, at the call, a great many functions related to argument passing, allocating stack space, and switching between V mode and I mode if necessary. At the return, some of these functions are performed in reverse. All of these operations occur without any intervention on the programmer's part, but while convenient from a programming point of view, they take an appreciable amount of time. A small subroutine of, say, 10 to 15 lines could conceivably take less time to execute than it would take to transfer control to it and and back, and if this subroutine is called very often in the course of a calling program's execution, the overhead could become prohibitive.

This overhead can be significantly reduced by using an alternative call and return mechanism known as the shortcall interface, or simply shortcall. Shortcall requires that the called routine be coded in assembly language; at Revision 21.0, the language of the calling program can be only FORTRAN 77 (F77).

The shortcall interface is implemented differently for V mode and I mode. The FORTRAN side of the interface is described in the FORTRAN F77 Reference Guide; the assembly language side is described in the following sections.

General Considerations

The assembly programmer, in order to properly handle subroutine arguments in the called procedure, needs to be aware of the differences between the standard call mechanism and the shortcall mechanism, and needs to know about the alignment of certain types of data in the calling program. A compilation listing of the calling program, in expanded listing format, should always be available for determining argument alignment and the sequence in which arguments are passed.

CALL/Shortcall Differences: Following is a list of the main differences between the PCL/CALL and shortcall mechanisms:

- V-mode calling programs execute a shortcall by means of a JSXB instruction; I-mode programs use a JMP instruction.
- Shortcall does not switch execution modes. This means that a V-mode calling program can call only V-mode subroutines, and an I-mode calling program can call only I-mode subroutines.
- Shortcall does not use stack space in the called procedure for argument passing. Arguments are passed differently in V mode and I mode; the mechanisms are described in detail later in this chapter.

- A shortcalled procedure is not required to have an ECB, since shortcall does not need information relating to execution mode, number of arguments, and argument pointer location that is normally contained in a called procedure's ECB.

- The calling program is responsible for providing the necessary space in its own stack frame for the number of arguments each shortcalled procedure expects. In FORTRAN 77, this is done through the SHORTCALL statement.

Argument Alignment: Certain datatypes can be either aligned (allocated on halfword boundaries) or unaligned (allocated on other than halfword boundaries) in the calling program. The shortcalled assembly language procedure needs to be aware of the alignment of an argument in order to retrieve it correctly.

Normally, only the following datatypes could be potentially unaligned:

- F77 CHARACTER*n

- F77 LOGICAL*1

These datatypes are always passed as unaligned, regardless of whether the actual arguments at each instance of a call are aligned or not. All other datatypes are always aligned in the calling program and passed as aligned data.

The setup instructions preparatory to a shortcall operation can be determined by examining an expanded listing of the calling program.

Argument Passing in V mode

Arguments are normally passed by reference, that is, pointers to the arguments are passed, rather than the arguments themselves. There are four distinct cases of argument passing to shortcalled procedures in V mode. The first case is that in which no arguments are being passed. No argument processing is required in the called procedure. The remaining three cases have arguments passed as follows:

One argument, aligned: A pointer to the argument is passed in the L register. The called procedure code required to access the argument is:

```
DYNM  TEMP(2)   storage for address of argument
STL   TEMP      store argument address
inst  TEMP,*    any operation on argument
```

The inst statement references the argument indirectly through TEMP.

One argument, unaligned: Special care must be taken in passing an unaligned object, since an unaligned object requires a 48-bit pointer. In this case, the calling procedure generates code to address the argument in FAR0, then loads the first two halfwords of this address into the L register. The extension bit (bit 4) is set to 1 in this address. It is up to the called procedure to test for this bit, and to retrieve the proper address from FAR0 if it is set. The following code can be used to do this:

```
DYNM  TEMP(3)
STL   TEMP      store L register
SAS   4         skip the JMP if bit 4 is 1
JMP#  *+3       jump if bit 4 is 0
STFA  0,TEMP    store FAR0
inst  TEMP,*    any operation on argument
```

More than one argument: A list of pointers to the arguments is constructed in memory by the caller. The space set aside for each pointer in the list is always three halfwords, although for aligned arguments only the first two halfwords are necessary. The address of the first list element is passed in the L register.

The arguments are most easily accessed as displacements relative to the XB register. However, since the XB register is used to store the caller's return address, the contents of this register must be saved before using it to access arguments, and restored just before the return. The L register, used to pass the argument list pointer to the callee, is used in the process of saving the return address, so it, too, must be saved temporarily. The following code sequence performs the required saves and sets up the XB register for argument access:

```
DYNM   TEMP_1(2),TEMP_2(2)
STL    TEMP_1              save argument list pointer
EAL    XB%                 save
STL    TEMP_2                return address
EAXB   TEMP_1,*            load saved pointer into XB
inst   XB%+0,*             operation on first argument
inst   XB%+3,*             operation on second argument
inst   XB%+6,*             operation on third argument
EAXB   TEMP_2,*            restore return address to XB
JMP    XB%                 return via XB
```

Note that the arguments are addressed in increments of three halfwords (XB%+0, XB%+3, ...); the increment corresponds to the 48-bit length of each pointer.

The V-mode argument passing mechanism may be affected if the shortcalled procedure is a function that returns a value. See the section Non-Register-Sized Return Value later in this chapter.

Register Saving and Restoring in V mode: Prime's usual convention for procedure calling via PCL is for the caller to save any registers whose contents are to be preserved over an external call. For efficiency's sake, however, this is not the best practice for shortcalled procedures.

For V-mode shortcalled procedures, the assembly programmer can assume that the L, X and Y registers are safe to use without having to save or restore them. No other registers are saved at the call site by the caller; therefore, the called program must be wary of modifying registers other than those just listed without saving and restoring them. Unnecessary saves can be eliminated if, upon examination of the caller's code, it is found that modification of unsaved registers does no damage.

Shortcall in I Mode

The implementation of the I-mode shortcall interface is designed to take advantage of the I mode general register based architecture. There are therefore differences between the shortcall interfaces in V mode and I mode. (For purposes of this discussion, I mode and IX mode are considered to be the same architecture.)

Call and Return Mechanism: The first difference between shortcalling in V mode and I mode is that of the mechanism used to transfer control to the callee and back. The I-mode return address is passed in R0, not the XB register. The code sequence used by the caller to invoke an I-mode shortcalled procedure, and the code sequence needed to return from it are:

Caller		Callee	
EAR	0,*+4	ST	0,TEMP
JMP	<callee>	JMP	TEMP,*

The EAR/JMP code sequence is faster than a JSXB. Also, saving and restoring R0 if it is needed for computation is more efficient than saving and restoring the XB register. If R0 is not needed by the callee for computation, it need not be saved, and the callee can return via a JMP R0%.

Argument Passing: Arguments are normally passed by reference (similar to V mode). However, two and three halfword pointers (pointers to

aligned and unaligned arguments, respectively) are treated separately in I mode.

Aligned Arguments: Aligned arguments are passed via the general registers R1 through R4. This means that a maximum of four pointers to aligned arguments can be passed via registers to a shortcalled procedure. For example, if pointers to three 32-bit integers A, B, and C, are passed in general registers 1, 2, and 3, respectively, the following instructions could operate on them:

```
L     6,R1%      load A into GR6
M     6,R2%      multiply A by B
PIM   6          position after multiply
A     6,R3%      add C to A*B
```

If more than 4 aligned arguments are passed, R1 through R3 are used for the first three as shown above, and R4 contains a pointer to a memory-resident list (in the calling program) of two-word pointers that point to the 4th through last arguments. Thus, to continue the previous example, if a fourth and fifth argument D and E were to be passed, they would be referenced as follows, after the code shown above:

```
EAXB  R4%        store pointer in XB
S     6,XB%,*    subtract D from A*B+C
D     6,XB%+2,*  divide A*B+C-D by E
```

Unaligned Arguments: Unaligned arguments are passed by reference via the FARs: FAR0 and FAR1, in sequence. The FARs are stored by STFA instructions into 48-bit indirect pointers in the callee's address space, and the arguments are addressed indirectly through these pointers:

```
DYNM  PTR_1(3),PTR_2(3)
STFA  0,PTR_1
STFA  1,PTR_2
inst  PTR_1,*    address first argument
inst  PTR_2,*    address second argument
```

If more than two unaligned arguments are passed, a pointer to the first is passed via FAR0, and FAR1 contains a pointer to a memory-resident list (in the calling program) of 48-bit pointers to the remaining unaligned arguments. The second through last arguments are addressed indirectly through the XB register:

```
DYNM  PTR_1(3),PTR_2(3)
STFA  0,PTR_1
STFA  1,PTR_2
EAXB  PTR_2,*     load pointer into XB
inst  PTR_1,*     address first argument
inst  XB%,*       address second argument
inst  XB%+3,*     address third argument
inst  XB%+6,*     address fourth argument
```

Mixed Arguments: Argument order is not strictly adhered to if a mixture of aligned and unaligned arguments is passed to an I-mode shortcalled procedure: aligned arguments are always passed via the general registers R1 through R4, unaligned arguments are always passed via the FARs. Within the aligned and unaligned classes of arguments, however, argument order is maintained.

Register Saving and Restoring: The callee is generally free to use any registers used to pass arguments without restoring them before returning. If the callee needs registers other than these, however, it is the callee's responsibility to save and restore them. Here, too, as in V mode, unnecessary saves can be eliminated if, upon examination of the caller's code, it is found that modification of unsaved registers does no harm.

Shortcalled Functions in V mode and I mode

The primary difference between subroutines and functions is that a function returns a value while a subroutine does not. Because Prime supports multi-language programming and procedure calling (PCL) between the V-mode and I-mode instruction sets, certain mechanisms by which functions return their values have been defined and must be adhered to. All types of functions, whether compiled or written in assembly language, called by PCL or shortcalled, must return their values according to these mechanisms; the programmer of a shortcalled function therefore needs to be aware of how these mechanisms work.

There are two classes of function return values: one class is those values that fit exactly into registers; the other is those that do not. Different mechanisms are used to return each of these classes of values. General guidelines are given here to help the assembly programmer in determining how to return function values. If unsure of how the caller expects the function to return its value, the programmer can compile the calling program, requesting an expanded listing, and can then examine the generated code.

Register-Sized Return Values: The types of values that are returned in registers from called functions include integers, floating point values, and other datatypes whose bit lengths match that of a register. Note carefully that what is passed to a function is a set of one or

more argument pointers, while what is passed from the function is a return value. The data types returned and the registers in which they are expected by a FORTRAN caller are shown below.

Data Types Returned	Expected Register (I mode in parentheses)
integer*2; logical*2; logical*1 (register bit 8)	A (GR2H)
integer*4; logical*4;	L (GR2)
real*4	FAC (FAC1)
real*8	DFAC (DFAC1)
real*16	QFAC

It is up to the assembly programmer to ensure that the shortcalled function return value is loaded into the appropriate type of register before returning to the caller.

Non-Register-Sized Return Values: If the return value does not fit exactly into a register, a pointer to a compiler-created space in the caller's stack is passed to the shortcalled function. This space is where the result value of the function must be put by the callee. The pointer is passed as the first argument in the V-mode function's argument list, and pointed to, as usual, by the L register. If other arguments are normally passed to the shortcalled function, these arguments are all moved up one position in the argument list, so that function argument 1 takes the space normally taken by subroutine argument 2, function argument 2 that of subroutine argument 3, and so on.

In I mode, the function return value pointer is passed as the first aligned argument, in R1. R2 through R4 will then contain the function's usual aligned arguments in the manner described above, except that they are, as in V mode, moved up one position in the argument list. Since R4 remains the last register that can be used to pass aligned arguments, the maximum number of arguments that can be passed to a shortcalled function of this type before the caller has to resort to a memory-resident argument list is reduced to 3.

13
Program Linking

When the assembler has completed its processing of one or more assembly language source programs, the object files it has created must be linked to create a file that can be loaded into memory and executed. The Prime linkers, SEG and BIND, combine object files whose pathnames are specified in linker subcommands into executable entities known as runfiles. Runfiles created by SEG are invoked by issuing the SEG command, specifying the pathname at which it was saved when linking was completed; those created by BIND are invoked by the RESUME command, also specifying the pathname at which it was filed. Unless you specify otherwise, both SEG and BIND obtain the assembled object files from, and store their respective runfiles in, the home directory to which you are attached when you invoke the linker.

This chapter briefly describes the characteristics of the two linkers and shows some examples of their use in doing simple linking tasks. For information on more complex applications, refer to the SEG and LOAD Reference Guide and the Programmer's Guide to BIND and EPFs.

Both SEG and BIND accept V-mode and I-mode object files as input to the linking process (I mode, for the purposes of this chapter, should be understood to also include IX mode). SEG can also link programs written in the older R mode, but these programs must be converted to V mode or I mode before BIND can link them. There is a discussion of conversion requirements and procedures in the Programmer's Guide to BIND and EPFs.

DIFFERENCES BETWEEN SEG AND BIND

One of the major differences between the two linkers is that SEG, in addition to performing the linking function for external references, common areas, and the like, also creates an image of (loads) the linked program into memory and then writes an image of memory to the runfile. The result is that, for default links, all executable programs are tied to the same memory segment (segment 4001). Therefore, if two programs were linked with SEG using the default segment assignment, it is not possible to interrupt the execution of one program, invoke the other, and then return to the first, since the second program will have overlaid part or all of the first.

BIND, on the other hand, performs the linking functions, but does not tie the linked code to any specific memory segment. It creates imaginary segments whose run-time segment numbers are determined by PRIMOS. BIND leaves the loading part of the linking and loading task for PRIMOS to perform when the program is invoked. PRIMOS loads the invoked program into any unoccupied memory segment it can find. This enables you to interrupt a program executing in one segment and load a second program (into a different segment, since PRIMOS considers the first program's segment still to be occupied) and execute it. You can then return to the first program, since it has not been overlaid by the second.

A second major difference between SEG and BIND is that BIND, by default, creates sharable procedures by separating executable code and modifiable data into separate segments. SEG can create sharable procedures, but it is a complex activity that includes specification of certain segment numbers, resharing of the PRIMOS operating system, and agreements between the programmer and the System Administrator. The procedure is described in detail in the SEG and LOAD Reference Guide.

There are many other differences between the two linkers, most of which make BIND the linker of choice over SEG. These differences are summarized in Chapter 1 of the Programmer's Guide to BIND and EPFs and described in more detail in Volume I of the Advanced Programmer's Guide.

USING THE SEG LINKER

The SEG linker is invoked by the PRIMOS SEG command, specifying the -LOAD option:

```
SEG -LOAD
```

Linking with SEG is an interactive procedure in which, for each subcommand, SEG issues a dollar sign ($) prompt. Figure 13-1 illustrates the procedure. In it, the program illustrated in Chapter 12 (named CALLSQ) is linked with its called subroutine module

(containing the entrypoints SQUARE and CUBE) to create the executable program CALLSQ.SEG. The resulting link map is shown in Figure 13-2.

USING THE BIND LINKER

The BIND linker is invoked by the PRIMOS command BIND:

```
BIND [options]
```

Linking with BIND can be interactive, working much like SEG, or you can include a series of options on the command line. In the latter case, the options specify the functions that would have been performed in response to subcommands in the interactive mode. When in the interactive mode, BIND issues a colon (:) as a prompt for each subcommand. Both methods are shown in Figure 13-3, using the same program and subroutine as in the SEG example above. The resulting link map is shown in Figure 13-4.

```
OK, SEG -LOAD
[SEG Rev. 21.0 Copyright (c) 1986, Prime Computer, Inc.]
$ LO CALLSQ    (Link calling program CALLSQ.BIN)
$ LO SQCU1     (Link subroutine module SQCU1.BIN)
$ LI           (Link system library routines TODEC and TONL)
LOAD COMPLETE
$ SAVE         (File CALLSQ.SEG)
$ MAP CALLSQ.SMAP   (Create map file -- optional)
$ Q            (Exit)
OK,
```

Sample SEG Terminal Session
Figure 13-1

```
*START  4002  000004  *STACK  4001  001206  *SYM    000016

SEG. #   TYPE        LOW      HIGH      TOP
 4001    PROC##    001000    001222    001204
 4002    DATA      000000    000115    000115


ROUTINE       ECB          PROCEDURE      ST. SIZE  LINK FR.
 ####      4002  000004     4001  001000    000012  000034  4002  177400
 CUBE      4002  000034     4001  001032    000020  000040  4002  177434
 SQUARE    4002  000054     4001  001041    000020  000040  4002  177434
 TODEC     4002  000074     4001  001052    000020  000022  4002  177474

DIRECT ENTRY LINKS
 T1OB      4001  001176     TONL      4001  001202

COMMON BLOCKS

OTHER SYMBOLS
```

Sample SEG Link Map
Figure 13-2

```
Using BIND interactively:


    OK, BIND
    [BIND Rev. 21.0 Copyright (c) 1986, Prime Computer, Inc.]
    : LO CALLSQ    (Link calling program CALLSQ.BIN)
    : LO SQCU1     (Link subroutine module SQCU1.BIN)
    : LI           (Link system library routines TODEC and TONL)
    BIND COMPLETE
    : MAP CALLSQ.MAP     (Create map file -- optional)
    : FILE         (File CALLSQ.RUN and Exit)
    OK,



Using BIND with command line options:


    OK, BIND CALLSQ -LO CALLSQ SQCU1 -LI -MAP CALLSQ.MAP
    BIND COMPLETE
    OK,
```

Sample BIND Terminal Session
Figure 13-3

```
Map of CALLSQ

  START ECB: -0002/000004

  Segment Type      Low    High   Top
   -0002  DATA    000000 000115 000116
   +0000  PROC    001000 001204 001206


PROCEDURES:
  Name       ECB address    Initial PB% Stack size  Link size Initial LB%
            -0002/000004  +0000/001000   000012     000034  -0002/177400
  CUBE      -0002/000034  +0000/001032   000020     000040  -0002/177434
  SQUARE    -0002/000054  +0000/001041   000020     000040  -0002/177434
  TODEC     -0002/000074  +0000/001052   000020     000022  -0002/177474

DYNAMIC LINKS:
  T1OB      +0000/001176
  TONL      +0000/001202

COMMON AREAS:

OTHER SYMBOLS:

UNDEFINED SYMBOLS:
```

Sample BIND Link Map
Figure 13-4

14
Program Execution and Debugging

PROGRAM EXECUTION

After your program has been assembled and linked, you can load it for execution in any of several ways.

- From command level; use the SEG command for programs linked with the SEG linker, or the RESUME command for programs linked with the BIND linker. Refer to the PRIMOS Commands Reference Guide for descriptions of these commands. (If your linked program has been placed in the command directory CMDNC0, you can also invoke it as an external command by simply supplying its name as if it were a PRIMOS command.)

- From another program, using any of several calling sequences; this procedure is described in detail in Volume III of the Advanced Programmer's Guide. Additional information can be found in the Programmer's Guide to BIND and EPFs.

- From command level in conjunction with one of the assembly language debugging utilities VPSD or IPSD; this method is described in this chapter.

PROGRAM DEBUGGING

There are two interactive debugging utilities that you can use for troubleshooting your programs. One of them, VPSD, is used for V mode programs; the other, IPSD, is an extension of VPSD that handles I mode (including IX mode) as well as V mode programs. IPSD differs from VPSD in its method of invocation and in some of its subcommands and displays. It is otherwise identical to VPSD.

The following sections describe the invocation methods for VPSD, the VPSD subcommands, and the IPSD extensions.

Using VPSD

VPSD is a symbolic routine that can handle the PRIME-400 and 50 Series segmented addressing modes (except I mode), and all of the PRIME-300 addressing modes.

There are two versions of VPSD: stand-alone VPSD and SEG's VPSD. Both reside in segment '4000. There are three ways to enter VPSD, each of which has slightly different consequences for debugging:

Action	Usage/Consequence
• Load the object file using SEG's loader. Then return to # level with the RE command and issue the SEG command PSD. Obtain the starting address of SEG's VPSD with the VERSION command. Memory may now be examined and breakpoints set. Type EX to start the program. If it crashes, issue the PRIMOS level PM command to obtain the data at crash time. Then issue the PRIMOS command START using SEG's VPSD starting address.	Used when no runfile exists. When EXECUTE is given, the registers are as SEG initialized them. Preserves the entire program contents exactly as it was at the time of the crash, except for the program counter whose value you obtain via the PM command.
• Load the runfile and enter VPSD via the SEG filename 1/1 command.	Used when runfile exists. When EXECUTE is given, the registers are as SEG initialized them.
• Load and execute the runfile using SEG. When the program crashes, use the PRIMOS command VPSD to call the stand-alone version of VPSD.	Use only if SEG's VPSD has been destroyed. The registers are not preserved.

VPSD Subcommand Line Format

Each VPSD subcommand 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. Subcommands may be terminated by a carriage return or a semicolon.

The ACCESS subcommand 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 subcommand 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 subcommand string and return to subcommand level.

If more than five octal digits are entered, only the last 16 bits are used.

Effective Address Formation: VPSD processes input and output in all addressing modes except I mode. The mode is set by the MODE subcommand.

When an index register is needed, the current value of the X or Y register is used.

When VPSD prints an address, it applies the same address formation process as the hardware, using the current values of the registers. For relative addresses, the location set by the last use of the ACCESS subcommand is used as the value of the P register.

Relocation Constant: VPSD can process addresses in a relocatable mode 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 '1000, both A >50 and A 1050 would address location '1050.

The relocation constant is set by the RELOCATE subcommand. Setting the relocation constant to 0 disables this mode.

For all output, any address larger than the relocation address is displayed as >n, where n is the address minus the relocation address.

Input/Output Formats: The default subcommand line scan is octal, but VPSD can accept input parameters and print output values in several other 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 affect only the current input line, but affect all output lines until you type a new format specifier. Table 14-1 describes the format specifiers. The following example illustrates their effects.

Table 14-1
Input/Output Formats

Format	Code	Input	Output
ASCII	:A	Two characters accepted first may not be: > = @ %, .NL./? + -: *( ) or blank Second is required and may not be: /, ?, <CR> Note: to input ASCII characters in any format use 'cc (single quote followed by two characters)	Two characters are printed. An @ is substituted for a non-printing character
Binary	:B	Takes a sequence of up to 16 1's and 0's	Prints a sequence of sixteen 1's and 0's
Decimal	:D	Accepts up to five decimal (0-9) digits	Prints decimal digits
Hex	:H	Accepts up to four hexadecimal (0-9, A, B, C, D, E, F) digits	Prints hexadecimal digits
Octal	:O	Accepts up to six octal (0-7) digits	Prints octal digits
Symbolic	:S	Symbolic instructions (See note)	Symbolic instructions
AP	:P	Symbolic instructions	Prints address pointers
Long	:L	Accepts up to 11 octal integers	Prints 32-bit octal integers

Note: Constants entered in :S mode are octal.

Fill and Dump Example:

F 100 200 :HAFAF	Fills octal locations 100 to 200 with hexadecimal digits AFAF
D 120 130	The display on the terminal will be in hexadecimal.

Symbolic Instruction Format: enables you to display standard PMA symbolic instruction format for output and to enter symbolic input in ACCESS subcommands. The only restrictions are:

- Expressions -- only + and - operations
- No literals
- Input is valid only in ACCESS mode; e.g., S 100 200 :SA1A is not valid
- The suffixes +1C and +nB may be used to indicate character and bit offsets

Constants entered in :S mode are octal

Subcommand Operands: Subcommand operands may be constants, constant expressions, or symbols. The format of a constant is:

```
[: format] [>]    + digits          [: format]
                  ASCII-constant
```

where:

format = format specifier (see Table 14-1)

> = relocatable mode

ASCII-constant = two-letter constant in the format described in Table 14-1

digit = decimal, octal, binary or hexadecimal, depending on which format is in control

The format of a constant expression is:

```
constant [+ constant]
```

Current Location Pointer: In ACCESS mode, a current location pointer is maintained, starting with the value of the start-address operand of the ACCESS cmmand. The location pointer determines the next location to be accessed.

During each access operation, VPSD replaces the value in the addressed 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.

VPSD Subcommands

This section gives a brief description of each of the VPSD subcommands. When entering subcommands, use only the underlined letter or letters in the subcommand.

▶ ACCESS address

Accesses a word in memory. VPSD displays address and its contents and then waits for keyboard input in the following form:

```
[:format-symbol] [value] [:new-format-symbol] terminator
```

:format-symbol is one of the optional input/output format symbols (see Table 14-1). 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 addressed location. The format is the current input/output format.

:new-format-symbol is one of the optional input/output format symbols (see Table 14-1). 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 14-2.

Long instructions are entered and printed in the same way as for the assembler; for example, LDA% 2000.

▶ BREAKPOINT location

Sets a breakpoint at the specified location. If the program is later executed and control reaches the breakpoint location, VPSD displays CPU status and awaits further subcommands. Up to ten breakpoints may be inserted.

Table 14-2
VPSD Terminators

Terminator	Function
CR	Alters contents of current location (if a value is given), moves to current location +1 and prints its contents.
^	Alters contents of current location (if a value is given), moves to current location -1 and prints its contents.
/ or?	Exits from access mode. Does not close current location.
.n(CR)	Moves to current location +n and prints its contents (n is octal).
.-n(CR)	Moves to current location -n and prints its contents (n is octal).
@	For memory reference instructions of the form "INST* location" only. Saves a return address (current location +1), moves to the effective address location, and prints its contents. Subsequent accesses (terminated by CR, comma, ,. or . -) are relative to the effective address. A \ returns to the return address.
(	Goes to effective address without indirection, but saves current location as return address.
\	Returns to the return address saved by the last @.
)	Returns to the return address saved by the last (.
=	For memory reference instructions only; calculates and prints the effective address and its contents. No change is made to the current location or its contents. If the instruction references a register, the contents of the register are printed.

▶ BREGISTER

Displays 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 target 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 area.

Example:

```
F 1000  1010  :HFFFF    Fill locations 1000-1010 with hex FFFF
F 1011  1020  :HAAAA    Fill locations 1011-1020 with hex AAAA
D 1000  1020            Display locations 1000-1020
 ---------
C 1010  1016  1012      Propagate alternate words of FFFF and AAAA
D 1000  1020            Display locations 1000-1020
```

▶ DUMP block-start block-end [words-per-line]

Displays 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 is the number of words to be printed per line.

You must open a file before dumping to it (see the OPEN subcommand). 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 subcommand with no parameters prior to issuing the DUMP subcommand.

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 words-per-line is specified. If you request dump output in symbolic (:S) format, specify a words-per-line value of 3 or less for a more readable display.

Example:

```
$ O DMPFIL 1 2          Open dump file
$ D 1000 2000           Dump locations '1000 through '2000
$ O 0 1 4               Close dump file
```

▶ 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 valid formats.

EFFECTIVE is useful in finding locations where a particular address 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 subcommand and shown in the keys by the PRINT subcommand.

▶ 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  1
   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 ^ goes back to the previous one. A (will switch to access mode and display the location referenced by the field address register in ASCII. A) will return to FA mode.

▶ FL regno

Accesses field length register regno. New values may be entered to replace old ones. A carriage return advances to the next register and a ^ goes back to the previous one.

▶ KEYS value

Sets the contents of the keys register to the specified octal value. See Chapter 5 of the System Architecture Guide for the format of the Keys register.

▶ LB seg-no word-no

Loads the link base register with 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.

▶ MODE 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.

MODE controls the way effective addresses are interpreted by setting the address mode bits of the Keys register. Other Keys register bits are unaffected. MODE gives you a way of setting just the address mode.

D64V prints the segment and word number for all addresses (initial segment number is '4000) and interprets instructions as the Prime 400 and 50 Series 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 as a DUMP output file. key can be: 1 (open for reading), 2 (open for writing), 3 (open for reading and writing) or 4 (close).

These key values are the same as for the PRIMOS OPEN command.

▶ PRINT

Prints CPU/VPSD parameters in octal as follows:

prgctr: breakpoint a-reg b-reg x-reg keys relcon y-reg

prgctr the program counter at the time of the breakpoint

relcon the 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 issues a RUN subcommand to the current program counter address. Registers and keys are loaded with specified values, if any.

▶ QUIT

Returns to the PRIMOS operating system. In SEG's VPSD, QUIT 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 beginning at start-add. Prior to program entry, a-reg, b-reg, x-reg, and keys are optionally loaded with specified values. Control does not return to VPSD unless a breakpoint is encountered.

Use the SN subcommand before issuing R to specify the segment in which to run; start-add is the 16-bit offset within that segment.

▶ SB seg-no word-no

Loads the stack base register with segment number seg-no and 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.

If a mask is not specified, the entire word is tested. When a match is found, the address and its contents are displayed, and the search continues until location block-end has been tested.

▶ SN seg-no

Use seg-no as the segment number for all subcommands where only a halfword offset 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 displays the address and contents of each location which does not match the corresponding word in the 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 crashes after a RUN subcommand, you can issue a PR subcommand, 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 segment number seg-no and word number word-no.

▶ XREGISTER value

Loads the X register with value -- for example, before executing a RUN subcommand 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.)

Using IPSD

The following sections describe IPSD, its features that are not found in VPSD, and how to use it. Also found here is information on the use and purpose of IPSD0 and IPSD16. These two utilities provide the same level of support for segment 4000 programs and data as is currently found in VPSD and VPSD16.

IPSD is an extension of VPSD that supports the debugging of I mode programs. IPSD can be used on both SEG runfiles EPF runfiles. It supports the V mode quad instructions and all of I mode. (This includes GRR, 32IX, and the I mode instructions added for C support.) IPSD also contains some additional features that improve upon VPSD's user interface. Since the IPSD user interface is nearly identical to that of VPSD, only the differences between VPSD and IPSD are described.

When IPSD is invoked, an outer shell interprets the command line, maps in the user program, and then invokes the actual IPSD debugger. The shell passes command lines to the user program and can provide for the execution of the user program as a command function if so desired.

Invoking IPSD

IPSD is invoked like any other command. Its usage is as follows:

```
IPSD [pathname] [program_command_line] [-FCN]
```

Where:

pathname is the pathname of an EPF or segdir program to be mapped in and executed under IPSD. The .RUN and .SEG suffixes are optional.

program_command_line is the command line expected by the program. (See the sections below on passing user program command lines.)

The -FCN option instructs IPSD to execute the user program as an EPF command function. (See the section below on execution as a command function.)

Invoking IPSD With No Arguments: If IPSD is invoked with no arguments, then the debugger is merely started up and no user program is mapped in. The initial register settings are as follows:

```
PB = 4000/0
LB = current LB
SB = current SB
XB = 7777/0
```

Passing Commandlines to an EPF User Program: IPSD will pass a command line to the user program if requested by the user. The user can request this by specifying the command line as the program_command_line argument to the IPSD command. For example,

```
OK, ipsd my_prog.run foo bar -option a -option b
```

In this case, the program my_prog will be invoked with the command line foo bar -option a -option b.

IPSD assumes that an EPF requiring a command line receives this command line via the standard EPF method, i.e. as the first argument to the main entrypoint of the EPF program. If IPSD is invoked upon an EPF user program that expects at least 1 argument, then IPSD always expects the user to provide a command line for the user program. If the user does not supply a command line, IPSD will ask for one. For example, if my_prog is an EPF that expects 1 or more arguments to its main entrypoint, then this is what IPSD would do:

```
OK, ipsd my_prog.run

[IPSD Rev. 21.0.0 Copyright (c) Prime Computer, Inc. 1986]

$ex
Your EPF program accepts arguments, one probably being its
command line argument.  Enter your program's command line,
if any.
--->x y z       (enter my_prog's command line)
```

In this example, the character string x y z would be passed as the first argument to my_prog. If my_prog does not care about the command line, enter a carriage return at the prompt.

Another possibility is that an EPF user program obtains its command line by calling RDTK$$. If IPSD is invoked on such a program with a user program command line, then IPSD will request reentry of the command line, since it must be provided interactively. For example, if my_prog is an EPF whose main entrypoint does not expect any arguments, this is what IPSD would do:

```
OK, ipsd my_prog.run x y z

[IPSD Rev. 21.0.0 Copyright (c) Prime Computer, Inc. 1986]

$ex
If your program requires a command line, then it obtains it
by using RDTK$$.  In this case, it must be entered
interactively.  Enter your program's command line, if any.
--->x y z
```

In this example, the Primos routine COMANL is called in order to get the character string x y z into the command line buffer accessed by RDTK$$. That is why the command line must be entered interactively.

Passing Commandlines to a Seg Runfiles: IPSD has no way of knowing whether or not a seg runfile requires a command line. For this reason, if the user program is a seg runfile, IPSD will always ask the user for a command line whether or not one is actually required. Here is an example of executing a seg runfile under IPSD:

```
OK, ipsd my_prog.seg

[IPSD Rev. 21.0.0 Copyright (c) Prime Computer, Inc. 1986]

$ex
Enter your program's command line, if any.
--->x y z
```

If my_prog, indeed, required a command line, then the character string x y z would be made available to it via the RDTK$$ interface. If my_prog did not require a command line, enter a carriage return.

If you mistakenly try to supply a seg runfile with a command line via the IPSD command, then IPSD will do this:

```
OK, ipsd my_prog.seg x y z

[IPSD Rev. 21.0.0 Copyright (c) Prime Computer, Inc. 1986]

$ex
If your program requires a command line, then it obtains it
by using RDTK$$.  In this case, it must be entered
interactively.  Enter your program's command line, if any.
--->x y z
```

As stated earlier, a command line to a seg runfile must be supplied interactively. That is why IPSD asks the user to reenter the command line.

Executing an EPF Program as a Command Function: IPSD can be instructed to invoke an EPF program as a command function. The specification of the -FCN command line option to IPSD causes this to occur.

When -FCN is specified, the COM_PROC_FLAGS argument to the main entrypoint of the EPF has the .COMMAND_FUNCTION_CALL flag set to 1.

Note

A seg runfile cannot be called as a command function.

Features Supported by IPSD But Not VPSD

The items described in the following sections are IPSD extensions over those described previously for VPSD.

Instructions and Instruction Sets: IPSD supports the V mode quad precision floating point instructions and all of the I-mode instruction set. This includes GRR, 32IX, and the instructions added for C language support.

▶ The MOde Subcommand:

IPSD supports the subcommand MO D32I to set the current address mode to I mode. If IPSD is invoked without specifying a user program, then the address mode defaults to I mode. Otherwise, the address mode is that of the user program's main program block.

I-Mode Breakpoints:

IPSD supports two types of breakpoints, V-mode and I-mode. The nature of the breakpoints is determined by the address mode in effect when the EX (execute), PR (proceed), or R (run) subcommand is given.

Note

An I-mode breakpoint occupies two words, while a V-mode breakpoint occupies only one word. For this reason, you must exercise care in the placement of breakpoints in I-mode programs. You must avoid placing an I-mode breakpoint such that control could be transferred from any other part of the program to the second word of the breakpointed instruction.

Preservation of Breakpoints After PRoceed Subcommand:

As in VPSD, IPSD removes the current breakpoint before allowing execution to continue. However, IPSD removes that breakpoint only temporarily. It is automatically re-inserted when the next breakpoint is encountered.

The P (Print) Subcommand:

When in I mode, the keys and the general and floating point registers are displayed in response to the P (print) command. For the general registers, the two halves are shown as halfword octal numbers, followed by the fullword octal contents of the register (leading zeros are suppressed). Two registers are displayed per line, thus:

```
GR0=      0      0           0  GR1= 177777 177772 37777777772
```

The floating point registers are each shown as four halfword octal numbers followed by the decimal value of the register contents. One register is displayed per line:

```
FR0= 104121 165605  17270    202  -0.37400000000000E 0001
```

When in V mode, the registers are displayed as in VPSD, except that the floating point register is also shown using the format described in the preceding paragraph.

The FR, GR, and HR Subcommands:

These subcommands permit the display and setting of the floating point, general, and half registers. The following sample dialogues illustrate

their use. Underlined entries show what you type to specify the register to display and to set register contents; / terminates each subcommand, returning you to the VPSD prompt.

```
$ FR 1<CR>
FR1=      0      0      0      0   0.00000000000000  12.5<CR>
FR0=      0      0      0      0   0.00000000000000  -3.74<CR>
FR1=  62000      0      0    204   0.12500000000000E 0002 <CR>
FR0= 104121 165605  17270    202  -0.37400000000000E 0001 /
$

$ GR 1<CR>
R1   0  177772<CR>
R2   0  12<CR>
R3   0  ^
R2   12  ^
R1   177772  /
$

$ GR 5<CR>
R5   123456  <CR>
R6   0  123456<CR>
R7   0  ^
R6   123456  /
$

$ HR 1<CR>
H1   0  177777<CR>
H2   0  ^
H1   177777  /
$

$ P<CR>
K= 10000
GR0=      0      0           0  GR1= 177777 177772 37777777772
GR2=      0     12          12  GR3=      0      0           0
GR4=      0      0           0  GR5=      0 123456      123456
GR6=      0 123456      123456  GR7=      0      0           0
FR0= 104121 165605  17270    202  -0.37400000000000E 0001
FR1=  62000      0      0    204   0.12500000000000E 0002
```

▶ Recognition of Erase and Kill Characters:

IPSD recognizes the Erase and Kill characters, as long as they occur before a line terminator such as <CR>, ?, /, @, =, !, (,), \, or ^. Note that the question-mark cannot be used as a line terminator when it has been defined to be the line kill character.

▶ Use With CPL or COMINPUT Files:

IPSD can be invoked from within a CPL program or a COMINPUT file.

▶ Default Arguments for D (Dump):

The D (dump) subcommand in IPSD has default arguments. If the third argument (number of words per line) is omitted, the dump will show three locations on one line when in symbolic mode, and eight locations on a line otherwise. If the second argument (dump-to address) is also omitted or is zero, '100 locations are dumped. If the first argument (dump-from address) is also omitted or is zero, dumping starts at the location last accessed. To start dumping at location '0000000, use a first argument of zero and a non-zero second argument.

▶ Immediate Operands for I-mode Memory Reference Instructions:

In denoting immediate operands in I-mode, it is necessary to use the left bracket instead of an equal sign. For example,

```
L 2,[123L
```

will be interpreted as L 2,=123L.

Immediate operands of MRGR-type instructions must be in octal. Immediate operands of MRFR-type instructions must be in decimal. The latter are converted to sixteen-bit values in which the right byte represents the characteristic. Thus, only seven significant bits of the mantissa are kept, with the remaining bits truncated. For example:

```
FL 1,[2.015625
```

will be interpreted as FL 1,=40202F, although the correct octal encoding of 2.015625 would have a mantissa of '402 and a characteristic of '202.

Restrictions

If IPSD is to be run on a seg runfile, then that runfile cannot use segments 4000 or 4037. Segment 4000 is reserved for IPSD itself. Segment 4037 is reserved for IPSD's stack extension.

IPSD0 and IPSD16

IPSD0 and IPSD16 are two utilities that offer some assistance in diagnosing problems in programs that need to reside in or use data in segment 4000. These utilities are not useful for controlling the execution of a program, but they can aid in the debugging task by enabling you to inspect data in segment 4000 without overwriting it.

IPSD0 is loaded below 4000/160000 and, therefore, is used to inspect data above 4000/160000.

IPSD16 is loaded above 4000/160000 and, therefore, is used to inspect data below 4000/160000.

Usage:

```
IPSD0

IPSD16
```

Their subcommand interfaces are the same as IPSD's. Upon invocation, IPSD0 and IPSD16 merely start up and enter an interactive mode. The initial register settings are as follows:

```
PB = 4000/0
LB = current LB
SB = current SB
XB = undefined
```

Appendices

A
Assembler Error Messages

Table A-1 on the following pages lists the messages that the assembler can display in response to syntax and other error conditions encountered during an assembly.

Table A-1
Assembler 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 FIELD NOT PERMITTED ON 64V MODE BRANCH INSTRUCTION

I06: ILLEGAL INDIRECT OR INDEX SPECIFICATION WITH COMMON/EXTERNAL SYMBOL

I07: INDEX SPECIFIED INVALID WITH AP/IP PSEUDO-OP

I08: TAG MODIFIED FIELD NOT PERMITTED ON 32I MODE BRANCH INSTRUCTION

L00: IMPROPER LABEL (CONSTANT OR TERMINATOR IN LABEL FIELD)

L01: EXTERNAL VARIABLE DISALLOWED IN LITERAL

L02: ILLEGAL ARGUMENT IN EQU, SET, OR XSET

M00: SYMBOL MULTIPLY DEFINED

N00: 'END' STATEMENT ENCOUNTERED WITHIN MACRO OR IF

Table A-1 (continued)
Assembler Error Messages

O00: UNRECOGNIZED OPCODE OR 32I-ONLY OPCODE IN NON-32I MODE

O01: THIS MEMORY REFERENCE INSTRUCTION ONLY PERMITTED IN 64V MODE

O02: THIS MEMORY REFERENCE INSTRUCTION ONLY PERMITTED IN S/R MODE

P00: MISMATCHED PARENTHESIS

Q00: AP ONLY PERMITTED IN 64V/32I MODE

Q01: IP ONLY PERMITTED IN 64V/32I MODE

Q02: ENDM PSEUDO-OP DISALLOWED OUTSIDE OF MACRO DEFINITION

R00: ARITHMETIC STACK OVERFLOW: REDUCE THE COMPLEXITY OF THE EXPRESSION AND TRY AGAIN

R01: MULTIPLY DEFINED MACRO OR MACRO NAME FIELD EMPTY

S00: INSTRUCTION REQUIRES DESECTORIZATION ('LOAD' MODE)

S01: INDIRECT DAC DISALLOWED IN C64R MODE

S02: 64V INSTRUCTION DISALLOWED IN C64R MODE

T00: SYNTAX ERROR IN 32I MODE TAG MODIFIED FIELD

U00: UNDEFINED SYMBOL IN ADDRESS FIELD OR EXPRESSION

U01: UNDEFINED SYMBOL IN 'ORG' OR 'SETB'

V01: CONTENTS OF BIT FIELD OUT OF RANGE

V02: UNRECOGNIZED OPERATOR IN EXPRESSION

V03: FUNCTION CODE OR DEVICE ADDRESS OUT OF RANGE IN I/O INSTRUCTION

V04: SHIFT COUNT OUT OF RANGE IN SHIFT INSTRUCTION

V05: NO COMMA FOLLOWS FAR SPECIFICATION IN FIELD ADDRESS INSTRUCTION

Table A-1 (continued)
Assembler Error Messages

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 FOLLOWS 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/DFVT PSEUDO-OP

V16: OPEN PARENTHESIS MISSING ON DFTB/DFVT ARGUMENT

V17: CLOSE PARENTHESIS MISSING ON DFTB/DFVT ARGUMENT

V18: LABEL REQUIRED ON IFTF, IFTT, IFVT, IFVF PSEUDO-OP

V19: SYMBOL NOT FOUND IN IFTF, IFTT, IFVT, IFVF PSEUDO-OP

V20: ABS/REL PSEUDO-OP ILLEGAL IN SEG/SEGR MODE

V21: SEG/SEGR PSEUDO-OP SPECIFIED AFTER CODE HAS BEEN GENERATED

V22: PROC/LINK SPECIFICATION ONLY ALLOWED IN SEG/SEGR MODE

V23: FIELD OUT OF RANGE IN DDM PSEUDO-OP

V24: ILLEGAL ARGUMENT FOLLOWS 'EXT' PSEUDO-OP

V25: 'END' PSEUDO-OP ENCOUNTERED WITHIN MACRO

Table A-1 (continued)
Assembler Error Messages

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 FOLLOWS 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

V45: IMPROPER ARGUMENT TO SEG OR SEGR PSEUDO-OP

V46: INVALID ESCAPE CODE IN CHARACTER STRING

V47: CHARACTER STRING TOO LONG

V48: WARNING: NUMERIC CONSTANT CHARACTER NOT AFFECTED BY ASCII-8 SWITCH

Table A-1 (continued)
Assembler Error Messages

X00: 32I MODE REGISTER SPECIFICATION ERROR

Y00: PHASE ERROR - THE VALUE OF THE SYMBOL DEFINED ABOVE DIFFERS BETWEEN PASS 1 AND PASS 2

Z00: ILLEGAL ABSOLUTE REFERENCE IN SEG/SEGR MODE

Z01: ABSOLUTE REFERENCE OUTSIDE OF 0-7 DISALLOWED IN SEG/SEGR MODE

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 THE 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

B
Instruction Summary Chart

This appendix consists of a summary chart of the V mode and I mode instruction sets. Each instruction's mnemonic operation code is followed by its octal code, format, functional group, addressing mode, CBIT, LINK, and condition code information, and a phrase describing its function.

The columns in each chart are as follows:

R — Restrictions:

Blank	Instruction can be executed in any ring.
R	Instruction causes a restricted mode fault if executed in other than Ring 0.
P	Instruction may cause a fault depending on the effective address value.

Mnem — A mnemonic name recognized by the assembler.

Opcode — Octal operation code portion of the instruction.

RI — Register (R) and Immediate (I) forms, if available.

Form Format of instruction:

Mnemonic	Definition
AP	Address Pointer
BRAN	Branch
CHAR	Character
DECI	Decimal
GEN	Generic
GR	General Register -- non Memory Reference
IBRN	I Mode Branch
MR	Memory Reference -- Non I Mode
MRFR	Memory Reference -- Floating Register
MRGR	Memory Reference -- General Register
MRNR	Memory Reference -- Non Register
PIO	Programmed I/O
RGEN	Register Generic
SHFT	Shift

Func Function of instruction:

Mnemonic	Definition
ADMOD	Addressing Mode
BRAN	Branch
CHAR	Character
CLEAR	Clear Field
CPTR	C Language Pointer
DECI	Decimal Arithmetic
FIELD	Field Register
FLPT	Floating Point Arithmetic
GRR	General Register Relative
INT	Integer
INTGY	Integrity
IO	Input/Output
KEYS	Keys
LOGIC	Logical Operations
LTSTS	Logical Test and Set
MCTL	Machine Control
MOVE	Move
PCTLJ	Program Control and Jump
PRCEX	Process Exchange
QUEUE	Queue Control
SHIFT	Register Shift
SKIP	Skip

Mode Addressing modes of instructions:

Mode	Name
V	64V
I	32I
VI	64V or 32I
IX	32IX

CL How instruction affects the CBIT and LINK.

Code	Definition
-	CBIT and LINK are unchanged
1	CBIT = unchanged; LINK = carry
2	CBIT = overflow status; LINK = carry
3	CBIT = overflow status; LINK = indeterminate
4	CBIT and LINK = shift extension
5	CBIT = result; LINK = indeterminate
6	CBIT and LINK are indeterminate
7	CBIT and LINK are loaded by the instruction
8	CBIT = result; LINK = unchanged
9	CBIT = unchanged; LINK = indeterminate
*	CBIT and LINK values vary among processors; see individual instruction description

CC How instruction affects the condition codes.

Code	Definition
-	Condition codes are unchanged.
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 indeterminate.
6	Condition codes are loaded by instruction.
7	Condition codes show special results for this instruction.

Description A brief description of the instruction. For I-mode register operations, an R designates a full (32-bit) register; an r designates a half (16-bit) register.

Table B-1
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	A1A	141206		GEN	INT	V	2	1	Add One to A
	A2A	140304		GEN	INT	V	2	1	Add Two to A
	A	02	RI	MRGR	INT	I	2	1	Add Fullword
	ABQ	141716		AP	QUEUE	V	-	7	Add Entry to Bottom of Queue
	ABQ	134		AP	QUEUE	I	-	7	Add Entry to Bottom of Queue
	ACA	141216		GEN	INT	V	2	1	Add CBIT to A
	ACP	55	RI	GR	CPTR	IX	-	-	Add C Pointer
	ADD	06		MR	INT	V	2	1	Add
	ADL	06 03		MR	INT	V	2	1	Add Long
	ADLL	141000		GEN	INT	V	2	1	Add LINK to L
	ADLR	014		RGEN	INT	I	2	1	Add LINK to R
	AH	12	RI	MRGR	INT	I	2	1	Add Halfword
	AIP	75		MRGR	GRR	IX	2	1	Add Indirect Pointer
	ALFA 0	001301		GEN	FIELD	V	6	-	Add L to FAR 0
	ALFA 1	001311		GEN	FIELD	V	6	-	Add L to FAR 1
	ALL	0414XX		SHFT	SHIFT	V	4	-	A Left Logical
	ALR	0416XX		SHFT	SHIFT	V	4	-	A Left Rotate
	ALS	0415XX		SHFT	SHIFT	V	3	-	A Arithmetic Left Shift
	ANA	03		MR	LOGIC	V	-	-	AND to A
	ANL	03 03		MR	LOGIC	V	-	-	AND to A Long
	ARFA 0	161		RGEN	FIELD	I	-	-	Add R to FAR 0
	ARFA 1	171		RGEN	FIELD	I	-	-	Add R to FAR 1
	ARGT	000605		GEN	PCTLJ	VI	6	5	Argument Transfer
	ARL	0404XX		SHFT	SHIFT	V	4	-	A Right Logical
	ARR	0406XX		SHFT	SHIFT	V	4	-	A Right Rotate
	ARS	0405XX		SHFT	SHIFT	V	4	-	A Arithmetic Right Shift
	ATQ	141717		AP	QUEUE	V	-	7	Add Entry to Top of Queue
	ATQ	135		AP	QUEUE	I	-	7	Add Entry to Top of Queue
	BCEQ	141602		BRAN	BRAN	VI	-	-	Branch on Condition Code EQ
	BCGE	141605		BRAN	BRAN	VI	-	-	Branch on Condition Code GE
	BCGT	141601		BRAN	BRAN	VI	-	-	Branch on Condition Code GT
	BCLE	141600		BRAN	BRAN	VI	-	-	Branch on Condition Code LE
	BCLT	141604		BRAN	BRAN	VI	-	-	Branch on Condition Code LT
	BCNE	141603		BRAN	BRAN	VI	-	-	Branch on Condition Code NE
	BCR	141705		BRAN	BRAN	VI	-	-	Branch on CBIT Reset to 0

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	BCS	141704		BRAN	BRAN	VI	-	-	Branch on CBIT Set to 1
	BDX	140734		BRAN	BRAN	V	-	-	Branch on Decremented X
	BDY	140724		BRAN	BRAN	V	-	-	Branch on Decremented Y
	BEQ	140612		BRAN	BRAN	V	-	4	Branch on A Equal to 0
	BFEQ	141612		BRAN	BRAN	V	-	4	Branch on F Equal to 0
	BFEQ	122		IBRN	BRAN	I	-	4	Branch on F Equal to 0
	BFGE	141615		BRAN	BRAN	V	-	4	Branch on F Greater Than or Equal to 0
	BFGE	125		IBRN	BRAN	I	-	4	Branch on F Greater Than or Equal to 0
	BFGT	141611		BRAN	BRAN	V	-	4	Branch on F Greater Than 0
	BFGT	121		IBRN	BRAN	I	-	4	Branch on F Greater Than 0
	BFLE	141610		BRAN	BRAN	V	-	4	Branch on F Less Than or Equal to 0
	BFLE	120		IBRN	BRAN	I	-	4	Branch on F Less Than or Equal to 0
	BFLT	141614		BRAN	BRAN	V	-	4	Branch on F Less Than 0
	BFLT	124		IBRN	BRAN	I	-	4	Branch on F Less Than 0
	BFNE	141613		BRAN	BRAN	V	-	4	Branch on F Not Equal to 0
	BFNE	123		IBRN	BRAN	I	-	4	Branch on F Not Equal to 0
	BGE	140615		BRAN	BRAN	V	-	4	Branch on A Greater Than or Equal to 0
	BGT	140611		BRAN	BRAN	V	-	4	Branch on A Greater Than 0
	BHD1	144		IBRN	BRAN	I	-	-	Branch on r Decremented by 1
	BHD2	145		IBRN	BRAN	I	-	-	Branch on r Decremented by 2
	BHD4	146		IBRA	BRAN	I	-	-	Branch on r Decremented by 4
	BHEQ	112		IBRN	BRAN	I	-	4	Branch on r Equal to 0
	BHGE	115		IBRN	BRAN	I	-	4	Branch on r Greater Than or Equal to 0
	BHGT	111		IBRN	BRAN	I	-	4	Branch on r Greater Than 0
	BHI1	140		IBRN	BRAN	I	-	-	Branch on r Incremented by 1
	BHI2	141		IBRN	BRAN	I	-	-	Branch on r Incremented by 2

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	BHI4	142		IBRN	BRAN	I	-	-	Branch on r Incremented by 4
	BHLE	110		IBRN	BRAN	I	-	4	Branch on r Less Than or Equal to 0
	BHLT	114		IBRN	BRAN	I	-	4	Branch on r Less Than 0
	BHNE	113		IBRN	BRAN	I	-	4	Branch on r Not Equal to 0
	BIX	141334		BRAN	BRAN	V	-	-	Branch on Incremented X
	BIY	141324		BRAN	BRAN	V	-	-	Branch on Incremented Y
	BLE	140610		BRAN	BRAN	V	-	4	Branch on A Less Than or Equal to 0
	BLEQ	140702		BRAN	BRAN	V	-	4	Branch on L Equal to 0
	BLGE	140615		BRAN	BRAN	V	-	4	Branch on L Greater Than or Equal to 0
	BLGT	140701		BRAN	BRAN	V	-	4	Branch on L Greater Than 0
	BLLE	140700		BRAN	BRAN	V	-	4	Branch on L Less Than or Equal to 0
	BLLT	140614		BRAN	BRAN	V	-	4	Branch on L Less Than 0
	BLNE	140703		BRAN	BRAN	V	-	4	Branch on L Not Equal to 0
	BLR	141707		BRAN	BRAN	VI	-	-	Branch on LINK Reset to 0
	BLS	141706		BRAN	BRAN	VI	-	-	Branch on LINK Set to 1
	BLT	140614		BRAN	BRAN	V	-	4	Branch on A Less Than 0
	BMEQ	141602		BRAN	BRAN	VI	-	-	Branch on Magnitude Condition EQ
	BMGE	141706		BRAN	BRAN	VI	-	-	Branch on Magnitude Condition GE
	BMGT	141710		BRAN	BRAN	VI	-	-	Branch on Magnitude Condition GT
	BMLE	141711		BRAN	BRAN	VI	-	-	Branch on Magnitude Condition LE
	BMLT	141707		BRAN	BRAN	VI	-	-	Branch on Magnitude Condition LT
	BMNE	141603		BRAN	BRAN	VI	-	-	Branch on Magnitude Condition NE
	BNE	140613		BRAN	BRAN	V	-	4	Branch on A Not Equal to 0
	BRBR	040-07		IBRN	BRAN	I	-	-	Branch on Register Bit Reset to 0
	BRBS	000-03		IBRN	BRAN	I	-	-	Branch on Register Bit Set to 1
	BRD1	134		IBRN	BRAN	I	-	-	Branch on R Decremented by 1

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	BRD2	135		IBRN	BRAN	I	-	-	Branch on R Decremented by 1
	BRD4	136		IBRN	BRAN	I	-	-	Branch on R Decremented by 4
	BREQ	102		IBRN	BRAN	I	-	4	Branch on R Equal to 0
	BRGE	105		IBRN	BRAN	I	-	4	Branch on R Greater Than or Equal to 0
	BRGT	101		IBRN	BRAN	I	-	4	Branch on R Greater Than 0
	BRI1	130		IBRN	BRAN	I	-	-	Branch on R Incremented by 1
	BRI2	131		IBRN	BRAN	I	-	-	Branch on R Incremented by 2
	BRI4	132		IBRN	BRAN	I	-	-	Branch on R Incremented by 4
	BRLE	100		IBRN	BRAN	I	-	4	Branch on R Less Than or Equal to 0
	BRLT	104		IBRN	BRAN	I	-	4	Branch on R Less Than 0
	BRNE	103		IBRN	BRAN	I	-	4	Branch on R Not Equal to 0
	C	61	RI	MRGR	INT	I	1	1	Compare Fullword
	CAL	141050		GEN	CLEAR	V	-	-	Clear A Left Byte
	CALF	000705		AP	PCTLJ	VI	6	5	Call Fault Handler
	CAR	141044		GEN	CLEAR	V	-	-	Clear A Right Byte
	CAS	11		MR	SKIP	V	1	1	Compare A and Skip
	CAZ	140214		GEN	SKIP	V	1	1	Compare A with 0
	CCP	45	R	GR	CPTR	IX	-	1	Compare C Pointer
	CGT	001314		GEN	BRAN	V	6	5	Computed GOTO
	CGT	026		RGEN	BRAN	I	6	5	Computed GOTO
	CH	71	RI	MRGR	INT	I	1	1	Compare Halfword
	CHS	140024		GEN	INT	V	-	-	Change Sign
	CHS	040		RGEN	INT	I	-	-	Change Sign
	CLS	11 03		MR	LOGIC	V	1	1	Compare L and Skip
	CMA	140401		GEN	LOGIC	V	-	-	Complement A
	CMH	045		RGEN	LOGIC	I	-	-	Complement r
	CMR	44		RGEN	LOGIC	I	-	-	Complement R
	CR	056		RGEN	CLEAR	I	-	-	Clear R to 0
	CRA	140040		GEN	CLEAR	V	-	-	Clear A to 0
	CRB	140015		GEN	CLEAR	V	-	-	Clear B to 0
	CRBL	062		RGEN	CLEAR	I	-	-	Clear R High Byte 1 (Bits 1-8)
	CRBR	063		RGEN	CLEAR	I	-	-	Clear R High Byte 2 (Bits 9-16)
	CRE	141404		GEN	CLEAR	V	-	-	Clear E to 0
	CRHL	054		RGEN	CLEAR	I	-	-	Clear R Left Halfword

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	CRHR	055		RGEN	CLEAR	I	-	-	Clear R Right Halfword
	CRL	140010		GEN	CLEAR	V	-	-	Clear L to 0
	CRLE	141410		GEN	CLEAR	V	-	-	Clear L and E to 0
	CSA	140320		GEN	MOVE	V	5	-	Copy Sign of A
	CSR	041		RGEN	MOVE	I	5	-	Copy Sign of R
	D	62	RI	MRGR	INT	I	3	5	Divide Fullword
	DBLE	106		RGEN	FLPT	I	-	-	Convert Single to Double Precision Floating
	DCP	160		RGEN	CPTR	IX	-	-	Decrement C Pointer
	DFA	15,17	RI	MRFR	FLPT	I	3	5	Double Precision Floating Add
	DFAD	06 02		MR	FLPT	V	3	5	Double Precision Floating Add
	DFC	05,07	RI	MRFR	FLPT	I	-	1	Double Precision Floating Compare
	DFCM	140574		GEN	FLPT	V	3	5	Double Precision Floating Complement
	DFCM	144		RGEN	FLPT	I	3	5	Double Precision Floating Complement
	DFCS	11 02		MR	FLPT	V	6	5	Double Precision Floating Compare and Skip
	DFD	31,33	RI	MRFR	FLPT	I	3	5	Double Precision Floating Divide
	DFDV	17 02		MR	FLPT	V	3	5	Double Precision Floating Divide
	DFL	01,03	RI	MRFR	FLPT	I	-	-	Double Precision Floating Load
	DFLD	02 02		MR	FLPT	V	-	-	Double Precision Floating Load
	DFLX	15 02		MR	FLPT	V	-	-	Double Precision Floating Load Index
	DFM	25,27	RI	MRFR	FLPT	I	3	5	Double Precision Floating Multiply
	DFMP	16 02		MR	FLPT	V	3	5	Double Precision Floating Multiply
	DFS	21,23	RI	MRFR	FLPT	I	3	5	Double Precision Floating Subtract
	DFSB	07 02		MR	FLPT	V	3	5	Double Precision Floating Subtract
	DFST	04 02		MR	FLPT	V	-	-	Double Precision Floating Store
	DFST	11,13		MRFR	FLPT	I	-	-	Double Precision Floating Store
	DH	72	RI	MRGR	INT	I	3	5	Divide Halfword

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	DH1	130		RGEN	INT	I	2	1	Decrement r by 1
	DH2	131		RGEN	INT	I	2	1	Decrement r by 2
	DIV	17		MR	INT	V	3	5	Divide
	DM	60		MRNR	INT	I	-	1	Decrement Memory Fullword
	DMH	70		MRNR	INT	I	-	1	Decrement Memory Halfword
	DR1	124		RGEN	INT	I	2	1	Decrement R by 1
	DR2	125		RGEN	INT	I	2	1	Decrement R by 2
	DRN	040300		GEN	FLPT	VI	3	5	Double Round From Quad
	DRNM	140571		GEN	FLPT	VI	8	5	Double Round From Quad Towards Negative Infinity
	DRNP	040301		GEN	FLPT	VI	3	5	Double Round From Quad Towards Positive Infinity
	DRNZ	040302		GEN	FLPT	VI	3	5	Double Round From Quad Towards Zero
	DRX	140210		GEN	SKIP	V	-	-	Decrement and Replace X
	DVL	17 03		MR	INT	V	3	5	Divide Long
	E16S	000011		GEN	ADMOD	VI	-	-	Enter 16S Mode
	E32I	001010		GEN	ADMOD	VI	-	-	Enter 32I Mode
	E32R	001013		GEN	ADMOD	VI	-	-	Enter 32R Mode
	E32S	000013		GEN	ADMOD	VI	-	-	Enter 32S Mode
	E64R	001011		GEN	ADMOD	VI	-	-	Enter 64R Mode
	E64V	000010		GEN	ADMOD	VI	-	-	Enter 64V Mode
	EAFA 0	001300		AP	FIELD	VI	-	-	Effective Address to FAR 0
	EAFA 1	001310		AP	FIELD	VI	-	-	Effective Address to FAR 1
	EAL	01 01		MR	PCTLJ	V	-	-	Effective Address to L
	EALB	13 02		MR	PCTLJ	V	-	-	Effective Address to LB
	EALB	42		MRNR	PCTLJ	I	-	-	Effective Address to LB
	EAR	63		MRGR	PCTLJ	I	-	-	Effective Address to R
	EAXB	12 02		MR	PCTLJ	V	-	-	Effective Address to XB
	EAXB	52		MRNR	PCTLJ	I	-	-	Effective Address to XB
R	EIO	14 01		MR	IO	V	-	7	Execute I/O
R	EIO	34		MRGR	IO	I	-	7	Execute I/O
R	ENB	000401		GEN	IO	VI	-	-	Enable Interrupts
R	ENBL	000401		GEN	IO	VI	-	-	Enable Interrupts (Local)
R	ENBM	000400		GEN	IO	VI	-	-	Enable Interrupts (Mutual)
R	ENBP	000402		GEN	IO	VI	-	-	Enable Interrupts (Process)

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	ERA	05		MR	LOGIC	V	-	-	Exclusive OR to A
	ERL	05 03		MR	LOGIC	V	-	-	Exclusive OR to L
	FA	014,16	RI	MRFR	FLPT	I	3	5	Floating Add
	FAD	06 01		MR	FLPT	V	3	5	Floating Add
	FC	04,06	RI	MRFR	FLPT	I	-	1	Floating Compare
	FCDQ	140571		GEN	FLPT	VI	-	-	Floating Convert Double to Quad
	FCM	140530		GEN	FLPT	V	3	5	Floating Complement
	FCM	100		RGEN	FLPT	I	3	5	Floating Complement
	FCS	11 01		MR	FLPT	V	6	5	Floating Compare and Skip
	FD	30,32	RI	MRFR	FLPT	I	3	5	Floating Divide
	FDBL	140016		GEN	FLPT	V	-	-	Floating Convert Single to Double
	FDV	17 01		MR	FLPT	V	3	5	Floating Divide
	FL	00,02	RI	MRFR	FLPT	I	-	-	Floating Load
	FLD	02 01		MR	FLPT	V	-	-	Floating Load
	FLT	105,11		RGEN	FLPT	I	6	5	Convert Integer to Floating Point
	FLTA	140532		GEN	FLPT	V	6	5	Convert Integer to Floating Point
	FLTH	102,11		RGEN	FLPT	I	6	5	Convert Halfword Integer to Floating Point
	FLTL	140535		GEN	FLPT	V	6	5	Convert Long Integer to Floating Point
	FLX	15 01		MR	FLPT	V	-	-	Floating Load Index
	FM	24,26	RI	MRFR	FLPT	I	3	5	Floating Multiply
	FMP	16 01		MR	FLPT	V	3	5	Floating Multiply
	FRN	140534		GEN	FLPT	V	3	5	Floating Round
	FRN	107		RGEN	FLPT	I	3	5	Floating Round
	FRNM	040320		GEN	FLPT	V	3	5	Floating Round Towards Negative Infinity
	FRNM	146		RGEN	FLPT	I	3	5	Floating Round Towards Negative Infinity
	FRNP	040303		GEN	FLPT	V	3	5	Floating Round Towards Positive Infinity
	FRNP	145		RGEN	FLPT	I	3	5	Floating Round Towards Positive Infinity
	FRNZ	040321		GEN	FLPT	V	3	5	Floating Round Towards Zero
	FRNZ	147		RGEN	FLPT	I	3	5	Floating Round Towards Zero
	FS	20,22	RI	MRFR	FLPT	I	3	5	Floating Subtract
	FSB	07 01		MR	FLPT	V	3	5	Floating Subtract

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	FSGT	140515		GEN	FLPT	V	-	5	Floating Skip If Greater Than 0
	FSLE	140514		GEN	FLPT	V	-	5	Floating Skip If Less Than or Equal to 0
	FSMI	140512		GEN	FLPT	V	-	5	Floating Skip If Minus
	FSNZ	140511		GEN	FLPT	V	-	5	Floating Skip If Not Equal to 0
	FSPL	140513		GEN	FLPT	V	-	5	Floating Skip If Plus
	FST	04 01		MR	FLPT	V	3	5	Floating Store
	FST	10,12		MRFR	FLPT	I	3	5	Floating Store
	FSZE	140510		GEN	FLPT	V	-	5	Floating Skip If Equal to 0
R	HLT	000000		GEN	MCTL	VI	-	-	Halt
	I	41	R	MRGR	MOVE	I	-	-	Interchange R and Memory Fullword
	IAB	000201		GEN	MOVE	V	-	-	Interchange A and B
	ICA	141340		GEN	MOVE	V	-	-	Interchange Bytes of A
	ICBL	065		RGEN	MOVE	I	-	-	Interchange Bytes and Clear Left
	ICBR	066		RGEN	MOVE	I	-	-	Interchange Bytes and Clear Right
	ICHL	060		RGEN	MOVE	I	-	-	Interchange Halfwords and Clear Left
	ICHR	061		RGEN	MOVE	I	-	-	Interchange Halfwords and Clear Right
	ICL	141140		GEN	MOVE	V	-	-	Interchange Bytes and Clear Left
	ICP	167		RGEN	CPTR	IX	-	-	Increment C Pointer
	ICR	141240		GEN	MOVE	V	-	-	Interchange Bytes and Clear Right
	IH	51	R	MRGR	MOVE	I	-	-	Interchange r and and Memory Halfword
	IH1	126		RGEN	INT	I	2	1	Increment r by 1
	IH2	127		RGEN	INT	I	2	1	Increment r by 2
	ILE	141414		GEN	MOVE	V	-	-	Interchange L and E
	IM	40		MRNR	INT	I	-	1	Increment Memory Fullword
	IMA	13		MR	MOVE	V	-	-	Interchange Memory and A
	IMH	50		MRNR	INT	I	-	1	Increment Memory Halfword
R	INBC	001217		AP	PRCEX	VI	6	5	Interrupt Notify Beginning, Clear Active Interrupt
R	INBN	001215		AP	PRCEX	VI	6	5	Interrupt Notify Beginning

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
R	INEC	001216		AP	PRCEX	VI	6	5	Interrupt Notify End, Clear Active Interrupt
R	INEN	001214		AP	PRCEX	VI	6	5	Interrupt Notify End
R	INH	001001		GEN	IO	VI	-	-	Inhibit Interrupts
R	INHL	001001		GEN	IO	VI	-	-	Inhibit Interrupts (Local)
R	INHM	001000		GEN	IO	VI	-	-	Inhibit Interrupts (Mutual)
R	INHP	001002		GEN	IO	VI	-	-	Inhibit Interrupts (Process)
	INK	070		RGEN	KEYS	I	-	-	Input Keys
	INT	103,11		RGEN	FLPT	I	3	5	Convert Floating Point to Integer
	INTA	140531		GEN	FLPT	V	3	5	Convert Floating Point to Integer
	INTH	101,11		RGEN	FLPT	I	3	5	Convert Floating Point to Halfword Integer
	INTL	140533		GEN	FLPT	V	3	5	Convert Floating Point to Integer Long
	IR1	122		RGEN	INT	I	2	1	Increment R by 1
	IR2	123		RGEN	INT	I	2	1	Increment R by 2
	IRB	064		RGEN	MOVE	I	-	-	Interchange r Bytes
	IRH	057		RGEN	MOVE	I	-	-	Interchange R Halves
	IRS	12		MR	SKIP	V	-	-	Increment and Replace Memory
R	IRTC	000603		GEN	IO	VI	7	6	Interrupt Return, Clear Active Interrupt
R	IRTN	000601		GEN	IO	VI	7	6	Interrupt Return
	IRX	140114		GEN	SKIP	V	-	-	Increment and Replace X
R	ITLB	000615		GEN	MCTL	VI	6	5	Invalidate STLB Entry
	JMP	01		MR	PCTLJ	V	-	-	Jump
	JMP	51		MRNR	PCTLJ	I	-	-	Jump
	JSR	73		MRGR	PCTLJ	I	-	-	Jump to Subroutine
	JST	10		MR	PCTLJ	V	-	-	Jump and Store
	JSX	35 03		MR	PCTLJ	V	-	-	Jump and Save in X
	JSXB	14 02		MR	PCTLJ	V	-	-	Jump and Save in XB
	JSXB	61		MRNR	PCTLJ	I	-	-	Jump and Save in XB
	JSY	14		MR	PCTLJ	V	-	-	Jump and Save in Y
	L	01	RI	MRGR	MOVE	I	-	-	Load
	LCC	45		MRGR	CPTR	IX	-	7	Load C Character
	LCEQ	141503		GEN	LTSTS	V	-	-	Load A on Condition Code EQ
	LCEQ	153		RGEN	LTSTS	I	-	-	Load r on Condition Code EQ

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	LCGE	141504		GEN	LTSTS	V	-	-	Load A on Condition Code GE
	LCGE	154		RGEN	LTSTS	I	-	-	Load r on Condition Code GE
	LCGT	141505		GEN	LTSTS	V	-	-	Load A on Condition Code GT
	LCGT	155		RGEN	LTSTS	I	-	-	Load r on Condition Code GT
	LCLE	141501		GEN	LTSTS	V	-	-	Load A on Condition Code LE
	LCLE	151		RGEN	LTSTS	I	-	-	Load r on Condition Code LE
	LCLT	141500		GEN	LTSTS	V	-	-	Load A on Condition Code LT
	LCLT	150		RGEN	LTSTS	I	-	-	Load r on Condition Code LT
	LCNE	141502		GEN	LTSTS	V	-	-	Load A on Condition Code NE
	LCNE	152		RGEN	LTSTS	I	-	-	Load r on Condition Code NE
	LDA	02		MR	MOVE	V	-	-	Load A
P	LDAR	44		MRGR	MOVE	I	-	5	Load from Addressed Register
	LDC 0	001302		CHAR	CHAR	V	-	7	Load Character
	LDC 0	162		RGEN	CHAR	I	-	7	Load Character
	LDC 1	001312		CHAR	CHAR	V	-	7	Load Character
	LDC 1	172		RGEN	CHAR	I	-	7	Load Character
	LDL	02 03		MR	MOVE	V	-	-	Load Long
P	LDLR	05 01		MR	MOVE	V	-	5	Load from Addressed Register
	LDX	35 00		MR	MOVE	V	-	-	Load X
	LDY	35 01		MR	MOVE	V	-	-	Load Y
	LEQ	140413		GEN	LTSTS	V	-	4	Load A on A Equal to 0
	LEQ	003		RGEN	LTSTS	I	-	4	Load r on R Equal to 0
	LF	140416		GEN	LTSTS	V	-	5	Load False
	LF	016		RGEN	LTSTS	I	-	5	Load False
	LFEQ	141113		GEN	LTSTS	V	-	4	Load A on F Equal to 0
	LFEQ	023,03		RGEN	LTSTS	I	-	4	Load r on F Equal to 0
	LFGE	141114		GEN	LTSTS	V	-	4	Load A on F Greater Than or Equal to 0
	LFGE	024,03		RGEN	LTSTS	I	-	4	Load r on F Greater Than or Equal to 0
	LFGT	141115		GEN	LTSTS	V	-	4	Load A on F Greater Than 0

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	LFGT	025,03		RGEN	LTSTS	I	-	4	Load r on F Greater Than 0
	LFLE	141111		GEN	LTSTS	V	-	4	Load A on F Less Than or Equal to 0
	LFLE	021,03		RGEN	LTSTS	I	-	4	Load r on F Less Than or Equal to 0
	LFLI 0	001303		BRAN	FIELD	VI	-	-	Load FLR 0 Immediate
	LFLI 1	001313		BRAN	FIELD	VI	-	-	Load FLR 1 Immediate
	LFLT	141110		GEN	LTSTS	V	-	4	Load A on F Less Than 0
	LFLT	020,03		RGEN	LTSTS	I	-	4	Load r on F Less Than 0
	LFNE	141112		GEN	LTSTS	V	-	4	Load A on F Not Equal or Equal to 0
	LFLI 0	001303		BRAN	FIELD	VI	-	-	Load FLR 0 Immediate
	LFLI 1	001313		BRAN	FIELD	VI	-	-	Load FLR 1 Immediate
	LFLT	141110		GEN	LTSTS	V	-	4	Load A on F Less Than 0
	LFLT	020,03		RGEN	LTSTS	I	-	4	Load r on F Less Than 0
	LFNE	141112		GEN	LTSTS	V	-	4	Load A on F Not Equal or Equal to 0
	LFNE	022,03		RGEN	LTSTS	I	-	4	Load r on F Not Equal to 0
	LGE	140414		GEN	LTSTS	V	-	4	Load A on A Greater Than or Equal to 0
	LGE	004		RGEN	LTSTS	I	-	4	Load r on R Greater Than or Equal to 0
	LGT	140415		GEN	LTSTS	V	-	4	Load A on A Greater Than 0
	LGT	005		RGEN	LTSTS	I	-	4	Load r on R Greater Than 0
	LH	11	RI	MRGR	MOVE	I	-	-	Load Halfword
	LHEQ	013		RGEN	LTSTS	I	-	4	Load r on r Equal to 0
	LHGE	004		RGEN	LTSTS	I	-	4	Load r on r Greater Than or Equal to 0
	LHGT	015		RGEN	LTSTS	I	-	4	Load r on r Greater Than 0
	LHL1	04	R	MRGR	MOVE	I	-	-	Load Halfword Shifted Left by 1
	LHL2	14	R	MRGR	MOVE	I	-	-	Load Halfword Shifted Left by 2
	LHL3	35	R	MRGR	MOVE	I	-	-	Load Halfword Shifted Left by 3
	LHLE	011		RGEN	LTSTS	I	-	4	Load r on r Less Than or Equal to 0
	LHLT	000		RGEN	LTSTS	I	-	4	Load r on r Less Than 0
	LHNE	012		RGEN	LTSTS	I	-	4	Load r on r Not Equal to 0

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
R	LIOT	000044		AP	MCTL	VI	6	5	Load IOTLB
	LIP	65		MRGR	GRR	IX	-	-	Load Indirect Pointer
	LLE	140411		GEN	LTSTS	V	-	4	Load A on A Less Than or Equal to 0
	LLE	001		RGEN	LTSTS	I	-	4	Load r on R Less Than or Equal to 0
	LLEQ	141513		GEN	LTSTS	V	-	4	Load L on A Equal to 0
	LLGE	140414		GEN	LTSTS	V	-	4	Load L on A Greater Than or Equal to 0
	LLGT	141515		GEN	LTSTS	V	-	4	Load L on A Greater Than 0
	LLL	0410XX		SHFT	SHIFT	V	4	-	Long Left Logical
	LLLE	141511		GEN	LTSTS	V	-	4	Load L on A Less Than or Equal to 0
	LLLT	140410		GEN	LTSTS	V	-	4	Load L on A Less Than 0
	LLNE	141512		GEN	LTSTS	V	-	4	Load L on A Not Equal to 0
	LLR	0412XX		SHFT	SHIFT	V	4	-	Long Left Rotate
	LLS	0411XX		SHFT	SHIFT	V	3	5	Long Left Shift
	LLT	140410		GEN	LTSTS	V	-	4	Load A on A Less Than 0
	LLT	000		RGEN	LTSTS	I	-	4	Load r on R Less Than 0
	LNE	140412		GEN	LTSTS	V	-	4	Load A on A Not Equal to 0
	LNE	002		RGEN	LTSTS	I	-	4	Load r on R Not Equal to 0
R	LPID	000617		GEN	MCTL	VI	-	-	Load Process ID
R	LPSW	000711		AP	MCTL	VI	7	6	Load Process Status Word
	LRL	0400XX		SHFT	SHIFT	V	4	-	Long Right Logical
	LRR	0402XX		SHFT	SHIFT	V	4	-	Long Right Rotate
	LRS	0401XX		SHFT	SHIFT	V	4	-	Long Right Shift
	LT	140417		GEN	LTSTS	V	-	5	Load True
	LT	017		RGEN	LTSTS	I	-	5	Load True
	M	42	RI	MRGR	INT	I	*	-	Multiply Fullword
	MH	52	RI	MRGR	INT	I	3	5	Multiply Halfword
	MPL	16 03		MR	INT	V	*	-	Multiply Long
	MPY	16		MR	INT	V	3	-	Multiply
	N	03	RI	MRGR	LOGIC	I	-	-	AND Fullword
R	NFYB	001211		AP	PRCEX	VI	6	5	Notify
R	NFYE	001210		AP	PRCEX	VI	6	5	Notify
	NH	13	RI	MRGR	LOGIC	I	-	-	AND Halfword
	NOP	000001		GEN	MCTL	VI	-	-	No Operation
	O	23	RI	MRGR	LOGIC	I	-	-	OR Fullword
	OH	33	RI	MRGR	LOGIC	I	-	-	OR Halfword
	ORA	03 02		MR	LOGIC	V	-	-	Inclusive OR
	OTK	071		RGEN	KEYS	I	7	6	Output Keys

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	PCL	10 02		MR	PCTLJ	V	6	5	Procedure Call
	PCL	41		MRNR	PCTLJ	I	6	5	Procedure Call
	PID	052		RGEN	INT	I	-	-	Position for Integer Divide
	PIDA	000115		GEN	INT	V	-	-	Position for Integer Divide
	PIDH	053		RGEN	INT	I	-	-	Position r for Integer Divide
	PIDL	000305		GEN	INT	V	-	-	Position for Integer Divide Long
	PIM	050		RGEN	INT	I	3	5	Position after Multiply
	PIMA	000015		GEN	INT	V	3	5	Position after Multiply
	PIMH	051		RGEN	INT	I	3	5	Position r after Multiply
	PIML	000301		GEN	INT	V	3	5	Position after Multiply Long
	PRTN	000611		GEN	PCTLJ	VI	7	6	Procedure Return
R	PTLB	000064		GEN	MCTL	VI	6	5	Purge TLB
	QFAD	5 2 2		MR	FLPT	V	3	5	Quad Precision Floating Add
	QFAD	36		MRFR	FLPT	I	3	5	Quad Precision Floating Add
	QFC	47	RI	MRFR	FLPT	I	-	7	Quad Precision Floating Compare
	QFCM	140570		GEN	FLPT	V	3	5	Quad Precision Floating Complement
	QFCM	140570		GEN	FLPT	I	3	5	Quad Precision Floating Complement
	QFCS	5 2 6		MR	FLPT	V	6	5	Quad Precision Floating Compare and Skip
	QFDV	5 2 5		MR	FLPT	V	3	5	Quad Precision Floating Divide
	QFDV	46		MRFR	FLPT	I	3	5	Quad Precision Floating Divide
	QFLD	5 2 0		MR	FLPT	V	-	-	Quad Precision Floating Load
	QFLD	34		MRFR	FLPT	I	-	-	Quad Precision Floating Load
	QFLX	6 7		MR	FLPT	V	-	-	Quad Precision Floating Load Index
	QFMP	5 2 4		MR	FLPT	V	3	5	Quad Precision Floating Multiply
	QFMP	45		MRFR	FLPT	I	3	5	Quad Precision Floating Multiply

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	QFSB	5 2 3		MR	FLPT	V	3	5	Quad Precision Floating Subtract
	QFSB	37		MRFR	FLPT	I	3	5	Quad Precision Floating Subtract
	QFST	5 2 1		MR	FLPT	V	-	-	Quad Precision Floating Store
	QFST	35		MRFR	FLPT	I	-	-	Quad Precision Floating Store
	QINQ	140572		GEN	FLPT	VI	3	5	Quad to Integer, in Quad Convert
	QIQR	140573		GEN	FLPT	VI	3	5	Quad to Integer, in Quad Convert Rounded
	RBQ	141715		AP	QUEUE	V	-	7	Remove Entry from Bottom of Queue
	RBQ	133		AP	QUEUE	I	-	7	Remove Entry from Bottom of Queue
	RCB	140200		GEN	KEYS	VI	8	-	Reset CBIT to 0
R	RMC	000021		GEN	INTGY	VI	-	-	Reset Machine Check Flag to 0
	ROT	24		MRGR	SHIFT	I	4	-	Rotate
	RRST	000717		AP	MCTL	VI	-	-	Restore Registers
	RSAV	000715		AP	MCTL	VI	-	-	Save Registers
	RTQ	141714		AP	QUEUE	V	-	7	Remove Entry from Top of Queue
	RTQ	132		RGEN	QUEUE	I	-	7	Remove Entry from Top of Queue
R	RTS	000511		GEN	MCTL	VI	-	-	Reset Time Slice
	S	22	RI	MRGR	INT	I	2	1	Subtract Fullword
	S1A	140110		GEN	INT	V	2	1	Subtract 1 from A
	S2A	140310		GEN	INT	V	2	1	Subtract 2 from A
	SAR	10026X		GEN	SKIP	V	-	-	Skip on A Register Bit Reset to 0
	SAS	10126X		GEN	SKIP	V	-	-	Skip on A Register Bit Set to 1
	SBL	07 03		MR	INT	V	2	1	Subtract Long
	SCB	140600		GEN	KEYS	VI	5	-	Set CBIT to 1
	SCC	55		MRGR	CPTR	IX	-	-	Store C Character
	SGT	100220		GEN	SKIP	V	-	-	Skip on A Greater Than 0
	SH	32	RI	MRGR	INT	I	2	1	Subtract Halfword
	SHA	15		MRGR	SHIFT	I	4	-	Shift Arithmetic
	SHL	05		MRGR	SHIFT	I	4	-	Shift Logical
	SHL1	076		RGEN	SHIFT	I	4	-	Shift R Left 1
	SHL2	077		RGEN	SHIFT	I	4	-	Shift R Left 2

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	SHR1	120		RGEN	SHIFT	I	4	-	Shift R Right 1
	SHR2	121		RGEN	SHIFT	I	4	-	Shift R Right 2
	SKP	100000		GEN	SKIP	V	-	-	Skip
	SL1	072		RGEN	SHIFT	I	4	-	Shift R Left 1
	SL2	073		RGEN	SHIFT	I	4	-	Shift R Left 2
	SLE	101220		GEN	SKIP	V	-	-	Skip on A Less Than or Equal to 0
	SLN	101100		GEN	SKIP	V	-	-	Skip on LSB of A Nonzero
	SLZ	100100		GEN	SKIP	V	-	-	Skip on LSB of A Zero
	SMCR	100200		GEN	INTGY	V	-	-	Skip on Machine Check Reset to 0
	SMCS	101200		GEN	INTGY	V	-	-	Skip on Machine Check Set to 1
	SMI	101400		GEN	SKIP	V	-	-	Skip on A Minus
	SNZ	101040		GEN	SKIP	V	-	-	Skip on A Nonzero
	SPL	100400		GEN	SKIP	V	-	-	Skip on A Plus
	SR1	074		RGEN	SHIFT	I	4	-	Shift R Right 1
	SR2	075		RGEN	SHIFT	I	4	-	Shift R Right 2
	SRC	100001		GEN	SKIP	V	-	-	Skip on CBIT Reset to 0
	SSC	101001		GEN	SKIP	V	-	-	Skip on CBIT Set to 1
	SSM	140500		GEN	INT	V	-	-	Set Sign of A Minus
	SSM	042		RGEN	INT	I	-	-	Set Sign Minus
	SSP	140100		GEN	INT	V	-	-	Set Sign of A Plus
	SSP	043		RGEN	INT	I	-	-	Set Sign Plus
	SSSN	040310		GEN	MCTL	VI	6	5	Store System Serial Number
	ST	21		MRGR	MOVE	I	-	-	Store Fullword
	STA	04		MR	MOVE	V	-	-	Store A into Memory
	STAC	001200		AP	MOVE	V	-	7	Store A Conditionally
P	STAR	54		MRGR	MOVE	I	-	5	Store into Addressed Register
	STC 0	001322		CHAR	CHAR	V	-	7	Store Character
	STC 0	166		RGEN	CHAR	I	-	7	Store Character
	STC 1	001332		CHAR	CHAR	V	-	7	Store Character
	STC 1	176		RGEN	CHAR	I	-	7	Store Character
	STCD	137		AP	MOVE	I	-	7	Store Conditional Fullword
	STCH	136		AP	MOVE	I	-	7	Store Conditional Halfword
	STEX	001315		GEN	PCTLJ	V	6	5	Stack Extend
	STEX	027		RGEN	PCTLJ	I	6	5	Stack Extend
	STFA 0	001320		AP	FIELD	VI	-	-	Store FAR 0
	STFA 1	001330		AP	FIELD	VI	-	-	Store FAR 1
	STH	31		MRGR	MOVE	I	-	-	Store Halfword

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	STL	04 03		MR	MOVE	V	-	-	Store Long
	STLC	001204		AP	MOVE	V	-	7	Store L Conditionally
P	STLR	03 01		MR	MOVE	V	-	5	Store L into Addressed Register
R	STPM	000024		GEN	MCTL	VI	-	-	Store Processor Model Number
	STTM	000510		GEN	MCTL	VI	6	5	Store Process Timer
	STX	15		MR	MOVE	V	-	-	Store X
	STY	35 02		MR	MOVE	V	-	-	Store Y
	SUB	07		MR	INT	V	2	1	Subtract
	SVC	000505		GEN	PCTLJ	VI	-	-	Supervisor Call
	SZE	100040		GEN	SKIP	V	-	-	Skip on A Zero
	TAB	140314		GEN	MOVE	V	-	-	Transfer A to B
	TAK	001015		GEN	KEYS	V	7	6	Transfer A to Keys
	TAX	140504		GEN	MOVE	V	-	-	Transfer A to X
	TAY	140505		GEN	MOVE	V	-	-	Transfer A to Y
	TBA	140604		GEN	MOVE	V	-	-	Transfer B to A
	TC	046		RGEN	INT	I	3	1	Two's Complement R
	TCA	140407		GEN	INT	V	2	1	Two's Complement A
	TCH	047		RGEN	INT	I	3	1	Two's Complement r
	TCL	141210		GEN	INT	V	2	1	Two's Complement Long
	TCNP	76	R	MRNR	CPTR	IX	-	1	Test C Null Pointer
	TFLL 0	001323		GEN	FIELD	V	-	-	Transfer FLR 0 to L
	TFLL 1	001333		GEN	FIELD	V	-	-	Transfer FLR 1 to L
	TFLR 0	163		RGEN	FIELD	I	-	-	Transfer FLR 0 to R
	TFLR 1	173		RGEN	FIELD	I	-	-	Transfer FLR 1 to R
	TKA	001005		GEN	KEYS	V	-	-	Transfer Keys to A
	TLFL 0	001321		GEN	FIELD	V	-	-	Transfer L to FLR 0
	TLFL 1	001331		GEN	FIELD	V	-	-	Transfer L to FLR 1
	TM	44		MRNR	MCTL	I	-	1	Test Memory Fullword
	TMH	54		MRNR	INT	I	-	1	Test Memory Halfword
	TRFL 0	165		RGEN	FIELD	I	-	-	Transfer R to FLR 0
	TRFL 1	175		RGEN	FIELD	I	-	-	Transfer R to FLR 1
	TSTQ	141757		AP	QUEUE	V	-	7	Test Queue
	TSTQ	104		RGEN	QUEUE	I	-	7	Test Queue
	TXA	141034		GEN	MOVE	V	-	-	Transfer X to A
	TYA	141124		GEN	MOVE	V	-	-	Transfer Y to A
R	WAIT	000315		AP	PRCEX	I	-	-	Wait
	X	43	RI	MRGR	LOGIC	I	-	-	Exclusive OR Fullword
	XAD	001100		DECI	DECI	VI	3	1	Decimal Add
	XBTD	001145		DECI	DECI	VI	3	5	Binary to Decimal Conversion
	XCM	001102		DECI	DECI	VI	-	1	Decimal Compare
	XDTB	001146		DECI	DECI	VI	3	5	Decimal to Binary Conversion

Table B-1 (Continued)
Instruction Summary

R	Mnem	Opcode	RI	Form	Func	Mode	CL	CC	Description
	XDV	001107		DECI	DECI	VI	3	5	Decimal Divide
	XEC	01 02		MR	PCTLJ	V	-	-	Execute
	XED	001112		DECI	DECI	VI	-	-	Numeric Edit
	XH	53	RI	MRGR	LOGIC	I	-	-	Exclusive OR Halfword
	XMP	001104		DECI	DECI	VI	3	1	Decimal Multiply
	XMV	001101		DECI	DECI	VI	3	1	Decimal Move
	ZCM	001117		CHAR	CHAR	VI	6	7	Compare Character Field
	ZED	001111		CHAR	CHAR	VI	-	-	Character Field Edit
	ZFIL	001116		CHAR	CHAR	VI	6	5	Fill Field With Character
	ZM	43		MRNR	CLEAR	I	-	-	Clear Fullword
	ZMH	53		MRNR	CLEAR	I	-	-	Clear Halfword
	ZMV	001114		CHAR	CHAR	VI	6	5	Move Character Field
	ZMVD	001115		CHAR	CHAR	VI	6	5	Move Characters Between Equal Length Strings
	ZTRN	001110		CHAR	CHAR	VI	-	-	Character String Translate

C
Prime Extended Character Set

As of Rev. 21.0, Prime has expanded its character set. The basic character set remains the same as it was before Rev. 21.0: it is the ANSI ASCII 7-bit set (called ASCII-7), with the 8th bit turned on. However, the 8th bit is now significant; when it is turned off, it signifies a different character. Thus, the size of the character set has doubled, from 128 to 256 characters. This expanded character set is called the Prime Extended Character Set (Prime ECS).

The pre-Rev. 21.0 character set is a proper subset of Prime ECS. These characters have not changed. Software written before Rev. 21.0 will continue to run exactly as it did before. Software written at Rev. 21.0 that does not use the new characters needs no special coding to use the old ones.

Prime ECS support is automatic at Rev. 21.0. You may begin to use characters that have the 8th bit turned off. However, the extra characters are not available on most printers and terminals. Check with your System Administrator to find out whether you can take advantage of the new characters in Prime ECS.

Table C-1 shows the Prime Extended Character Set. The pre-Rev. 21.0 character set consists of the characters with decimal values 128 through 255 (octal values 200 through 377). The characters added at Rev. 21.0 all have decimal values less than 128 (octal values less than 200).

SPECIFYING PRIME ECS CHARACTERS

Direct Entry

On terminals that support Prime ECS, you can enter the printing characters directly; the characters appear on the screen as you type them. For information on how to do this, see the appropriate manual for your terminal.

A terminal supports Prime ECS if

- It uses ASCII-8 as its internal character set, and
- The TTY8 protocol is configured on your asynchronous line.

If you do not know whether your terminal supports Prime ECS, ask your System Administrator.

On terminals that do not support Prime ECS, you can enter any of the ASCII-7 printing characters (characters with a decimal value of 160 or higher) directly by just typing them.

Octal Notation

If you use the Editor (ED), you can enter any Prime ECS character on any terminal by typing

```
^octal-value
```

where octal-value is the three-digit octal number given in Table C-1. You must type all three digits, including leading zeroes.

Before you use this method to enter any of the ECS characters that have decimal values between 32 and 127, first specify the following ED command:

```
MODE CKPAR
```

This command permits ED to print as ^nnn any characters that have a first bit of 0.

Character String Notation

The way in which you specify Prime ECS characters in character strings in programs depends on the character that you wish to specify. You can

specify Prime ECS characters on any terminal by using one of the notations shown below. However, the characters themselves can only appear on a terminal that supports Prime ECS. Terminals that do not support Prime ECS will not display the characters correctly.

The following rules describe how to specify Prime ECS characters in character strings.

1. You can specify printing characters in character strings by enclosing them in single quotation marks ('). For example:

   ```
   'Quoted string'
   ```

   You can enter the characters using either direct entry or octal notation as described at the beginning of this section.

2. You can specify any character in Prime ECS that has a mnemonic as follows:

   ```
   \(mnemonic)
   ```

   where mnemonic is the Prime mnemonic shown for that character in Table C-1. The parentheses are essential. You can specify the mnemonic with either uppercase or lowercase characters. Some characters have more than one mnemonic; you may use any one of these. In the table, the alternatives are separated by a slash character (/). For example:

   ```
   'A string'\(FF)'with a form feed in it'
   ```

   The compiler interprets the above example as a single character string.

3. You can specify certain frequently used non-printing characters as

   ```
   \abbreviation
   ```

   where abbreviation is one of the following:

Abbreviation	Meaning
B	Backspace
E	Escape
F	Form feed
L	Line feed
N	New line
R	Carriage return
T	Horizontal tab
V	Vertical tab

For example:

```
'A string'\F'with a form feed in it'
```

4. You can specify control characters as

```
\^character
```

where ^character is listed under "Graphic" in Table C-1. For example:

```
'A string'\^L'with a form feed in it'
```

You may use the commercial at sign (@) in place of the caret (^).

5. You can specify control characters as

```
\Onnn          (octal notation)
\Hnn           (hexadecimal notation)
\nnn           (decimal notation)
```

where Onnn is the letter O followed by the three octal digits shown in Table C-1 for the character. Hnn and nnn are the hexadecimal and decimal equivalents, respectively. For example:

```
'A string'\O214'with a form feed in it'
```

A character specified with a backslash (that is, with notation 2, 3, 4, or 5)

- Must appear outside quotation marks
- Specifies a character string of length 1
- Can be specified by itself, or combined with one or more additional backslash-notation characters, or with one or more quoted character strings

Spaces between the Prime ECS character specification and the character string are not significant, but there must be no spaces within the character specification itself.

The following subroutine call example writes a string specified by Prime ECS syntax. It contains two occurrences of format 1 and one each of formats 2, 3, 4, and 5. \(CR) and \^M specify carriage returns; \N and \0212 specify newlines.

```
CALL    TNOUA
AP      =C'HELLO'\(CR)\N'THERE'\^M\0212,S
AP      =14,SL
```

This subroutine call produces the following output:

```
HELLO
THERE
```

SPECIAL MEANINGS OF PRIME ECS CHARACTERS

PRIMOS, or an applications program running on PRIMOS, may interpret some Prime ECS characters in a special way. For example, PRIMOS interprets ^P as a process interrupt. ED, the Editor, interprets the backslash (\) as a logical tab. If you wish to make use of the Prime ECS backslash character in a file you are editing with ED, you must define another character as your logical tab.

For a detailed description of how PRIMOS interprets the following Prime ECS characters, see the discussion in the Prime User's Guide of special terminal keys and special characters ^ \ " ? ^P ^S ^Q _ and ;.

ASSEMBLY PROGRAMMING CONSIDERATIONS

Remember that symbols can contain only uppercase letters, numbers, and the dollar sign and underscore characters ($ and _). These characters form a subset of the ASCII-7 character set.

Character strings, however, can contain any character in Prime ECS. Such strings can be declared as constant or literal strings or as arguments in subroutine calls that print or display character strings.

You can use notations 2, 3, 4, and 5, described above, alone or in combination with any quoted string in your program. You cannot, however, use these notations in symbols when writing your program, nor can your program's input files contain any of these notations. If your terminal does not support Prime ECS, you can enter as terminal input only those characters with decimal numbers greater than 127 (octal numbers greater than 177).

PRIME EXTENDED CHARACTER SET TABLE

Table C-1 contains all of the Prime ECS characters, arranged in ascending order. This order provides both the collating sequence and the way that comparisons are done for character strings. For each character, the table includes the graphic, the mnemonic, the description, and the binary, decimal, hexadecimal, and octal values. A blank entry indicates that the particular item does not apply to this character. The graphics for control characters are specified as ^character; for example, ^P represents the character produced when you type P while holding the control key down.

Characters with decimal values from 000 to 031 and from 128 to 159 are control characters.

Characters with decimal values from 032 to 127 and from 160 to 255 are graphic characters.

Table C-1
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	RES1	Reserved for future standardization	0000 0000	000	00	000
	RES2	Reserved for future standardization	0000 0001	001	01	001
	RES3	Reserved for future standardization	0000 0010	002	02	002
	RES4	Reserved for future standardization	0000 0011	003	03	003
	IND	Index	0000 0100	004	04	004
	NEL	Next line	0000 0101	005	05	005
	SSA	Start of selected area	0000 0110	006	06	006
	ESA	End of selected area	0000 0111	007	07	007
	HTS	Horizontal tabulation set	0000 1000	008	08	010
	HTJ	Horizontal tab with justify	0000 1001	009	09	011
	VTS	Vertical tabulation set	0000 1010	010	0A	012
	PLD	Partial line down	0000 1011	011	0B	013
	PLU	Partial line up	0000 1100	012	0C	014
	RI	Reverse index	0000 1101	013	0D	015
	SS2	Single shift 2	0000 1110	014	0E	016
	SS3	Single shift 3	0000 1111	015	0F	017
	DCS	Device control string	0001 0000	016	10	020
	PU1	Private use 1	0001 0001	017	11	021
	PU2	Private use 2	0001 0010	018	12	022
	STS	Set transmission state	0001 0011	019	13	023
	CCH	Cancel character	0001 0100	020	14	024
	MW	Message waiting	0001 0101	021	15	025
	SPA	Start of protected area	0001 0110	022	16	026
	EPA	End of protected area	0001 0111	023	17	027
	RES5	Reserved for future standardization	0001 1000	024	18	030
	RES6	Reserved for future standardization	0001 1001	025	19	031
	RES7	Reserved for future standardization	0001 1010	026	1A	032
	CSI	Control sequence introducer	0001 1011	027	1B	033
	ST	String terminator	0001 1100	028	1C	034
	OSC	Operating system command	0001 1101	029	1D	035
	PM	Privacy message	0001 1110	030	1E	036

Table C-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	APC	Application program command	0001 1111	031	1F	037
	NBSP	No-break space	0010 0000	032	20	040
¡	INVE	Inverted exclamation mark	0010 0001	033	21	041
¢	CENT	Cent sign	0010 0010	034	22	042
£	PND	Pound sign	0010 0011	035	23	043
¤	CURR	Currency sign	0010 0100	036	24	044
¥	YEN	Yen sign	0010 0101	037	25	045
¦	BBAR	Broken bar	0010 0110	038	26	046
§	SECT	Section sign	0010 0111	039	27	047
¨	DIA	Diaeresis, umlaut	0010 1000	040	28	050
©	COPY	Copyright sign	0010 1001	041	29	051
ª	FOI	Feminine ordinal indicator	0010 1010	042	2A	052
«	LAQM	Left angle quotation mark	0010 1011	043	2B	053
¬	NOT	Not sign	0010 1100	044	2C	054
	SHY	Soft hyphen	0010 1101	045	2D	055
®	TM	Registered trademark sign	0010 1110	046	2E	056
¯	MACN	Macron	0010 1111	047	2F	057
°	DEGR	Degree sign	0011 0000	048	30	060
±	PLMI	Plus/minus sign	0011 0001	049	31	061
²	SPS2	Superscript two	0011 0010	050	32	062
³	SPS3	Superscript three	0011 0011	051	33	063
´	AAC	Acute accent	0011 0100	052	34	064
µ	LCMU	Lowercase Greek letter µ, micro sign	0011 0101	053	35	065
¶	PARA	Paragraph sign, Pilgrow sign	0011 0110	054	36	066
·	MIDD	Middle dot	0011 0111	055	37	067
¸	CED	Cedilla	0011 1000	056	38	070
¹	SPS1	Superscript one	0011 1001	057	39	071
º	MOI	Masculine ordinal indicator	0011 1010	058	3A	072
»	RAQM	Right angle quotation mark	0011 1011	059	3B	073
¼	FR14	Common fraction one-quarter	0011 1100	060	3C	074

Table C-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
1/2	FR12	Common fraction one-half	0011 1101	061	3D	075
3/4	FR34	Common fraction three-quarters	0011 1110	062	3E	076
¿	INVQ	Inverted question mark	0011 1111	063	3F	077
À	UCAG	Uppercase A with grave accent	0100 0000	064	40	100
Á	UCAA	Uppercase A with acute accent	0100 0001	065	41	101
Â	UCAC	Uppercase A with circumflex	0100 0010	066	42	102
Ã	UCAT	Uppercase A with tilde	0100 0011	067	43	103
Ä	UCAD	Uppercase A with diaeresis	0100 0100	068	44	104
Å	UCAR	Uppercase A with ring above	0100 0101	069	45	105
Æ	UCAE	Uppercase diphthong Æ	0100 0110	070	46	106
Ç	UCCC	Uppercase C with cedilla	0100 0111	071	47	107
È	UCEG	Uppercase E with grave accent	0100 1000	072	48	110
É	UCEA	Uppercase E with acute accent	0100 1001	073	49	111
Ê	UCEC	Uppercase E with circumflex	0100 1010	074	4A	112
Ë	UCED	Uppercase E with diaeresis	0100 1011	075	4B	113
Ì	UCIG	Uppercase I with grave accent	0100 1100	076	4C	114
Í	UCIA	Uppercase I with acute accent	0100 1101	077	4D	115
Î	UCIC	Uppercase I with circumflex	0100 1110	078	4E	116
Ï	UCID	Uppercase I with diaeresis	0100 1111	079	4F	117
Ð	UETH	Uppercase Icelandic letter Eth	0101 0000	080	50	120
Ñ	UCNT	Uppercase N with tilde	0101 0001	081	51	121
Ò	UCOG	Uppercase O with grave accent	0101 0010	082	52	122
Ó	UCOA	Uppercase O with acute accent	0101 0011	083	53	123

Table C-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
Ô	UCOC	Uppercase O with circumflex	0101 0100	084	54	124
Õ	UCOT	Uppercase O with tilde	0101 0101	085	55	125
Ö	UCOD	Uppercase O with diaeresis	0101 0110	086	56	126
×	MULT	Multiplication sign used in mathematics	0101 0111	087	57	127
Ø	UCOO	Uppercase O with oblique line	0101 1000	088	58	130
Ù	UCUG	Uppercase U with grave accent	0101 1001	089	59	131
Ú	UCUA	Uppercase U with acute accent	0101 1010	090	5A	132
Û	UCUC	Uppercase U with circumflex	0101 1011	091	5B	133
Ü	UCUD	Uppercase U with diaeresis	0101 1100	092	5C	134
Ý	UCYA	Uppercase Y with acute accent	0101 1101	093	5D	135
Þ	UTHN	Uppercase Icelandic letter Thorn	0101 1110	094	5E	136
ß	LGSS	Lowercase German letter double s	0101 1111	095	5F	137
à	LCAG	Lowercase a with grave accent	0110 0000	096	60	140
á	LCAA	Lowercase a with acute accent	0110 0001	097	61	141
â	LCAC	Lowercase a with circumflex	0110 0010	098	62	142
ã	LCAT	Lowercase a with tilde	0110 0011	099	63	143
ä	LCAD	Lowercase a with diaeresis	0110 0100	100	64	144
å	LCAR	Lowercase a with ring above	0110 0101	101	65	145
æ	LCAE	Lowercase diphthong ae	0110 0110	102	66	146
ç	LCCC	Lowercase c with cedilla	0110 0111	103	67	147
è	LCEG	Lowercase e with grave accent	0110 1000	104	68	150
é	LCEA	Lowercase e with acute accent	0110 1001	105	69	151
ê	LCEC	Lowercase e with circumflex	0110 1010	106	6A	152

Table C-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
ë	LCED	Lowercase e with diaeresis	0110 1011	107	6B	153
ì	LCIG	Lowercase i with grave accent	0110 1100	108	6C	154
í	LCIA	Lowercase i with acute accent	0110 1101	109	6D	155
î	LCIC	Lowercase i with circumflex	0110 1110	110	6E	156
ï	LCID	Lowercase i with diaeresis	0110 1111	111	6F	157
ð	LETH	Lowercase Icelandic letter Eth	0111 0000	112	70	160
ñ	LCNT	Lowercase n with tilde	0111 0001	113	71	161
ò	LCOG	Lowercase o with grave accent	0111 0010	114	72	162
ó	LCOA	Lowercase o with acute accent	0111 0011	115	73	163
ô	LCOC	Lowercase o with circumflex	0111 0100	116	74	164
õ	LCOT	Lowercase o with tilde	0111 0101	117	75	165
ö	LCOD	Lowercase o with diaeresis	0111 0110	118	76	166
÷	DIV	Division sign used in mathematics	0111 0111	119	77	167
ø	LCOO	Lowercase o with oblique line	0111 1000	120	78	170
ù	LCUG	Lowercase u with grave accent	0111 1001	121	79	171
ú	LCUA	Lowercase u with acute accent	0111 1010	122	7A	172
û	LCUC	Lowercase u with circumflex	0111 1011	123	7B	173
ü	LCUD	Lowercase u with diaeresis	0111 1100	124	7C	174
ý	LCYA	Lowercase y with acute accent	0111 1101	125	7D	175
þ	LTHN	Lowercase Icelandic letter Thorn	0111 1110	126	7E	176
ÿ	LCYD	Lowercase y with diaeresis	0111 1111	127	7F	177

Table C-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	NUL	Null	1000 0000	128	80	200
^A	SOH/TC1	Start of heading	1000 0001	129	81	201
^B	STX/TC2	Start of text	1000 0010	130	82	202
^C	ETX/TC3	End of text	1000 0011	131	83	203
^D	EOT/TC4	End of transmission	1000 0100	132	84	204
^E	ENQ/TC5	Enquiry	1000 0101	133	85	205
^F	ACK/TC6	Acknowledge	1000 0110	134	86	206
^G	BEL	Bell	1000 0111	135	87	207
^H	BS/FE0	Backspace	1000 1000	136	88	210
^I	HT/FE1	Horizontal tab	1000 1001	137	89	211
^J	LF/NL/FE2	Line feed	1000 1010	138	8A	212
^K	VT/FE3	Vertical tab	1000 1011	139	8B	213
^L	FF/FE4	Form feed	1000 1100	140	8C	214
^M	CR/FE5	Carriage return	1000 1101	141	8D	215
^N	SO/LS1	Shift out	1000 1110	142	8E	216
^O	SI/LS0	Shift in	1000 1111	143	8F	217
^P	DLE/TC7	Data link escape	1001 0000	144	90	220
^Q	DC1/XON	Device control 1	1001 0001	145	91	221
^R	DC2	Device control 2	1001 0010	146	92	222
^S	DC3/XOFF	Device control 3	1001 0011	147	93	223
^T	DC4	Device control 4	1001 0100	148	94	224
^U	NAK/TC8	Negative acknowledge	1001 0101	149	95	225
^V	SYN/TC9	Synchronous idle	1001 0110	150	96	226
^W	ETB/TC10	End of transmission block	1001 0111	151	97	227
^X	CAN	Cancel	1001 1000	152	98	230
^Y	EM	End of medium	1001 1001	153	99	231
^Z	SUB	Substitute	1001 1010	154	9A	232
^[	ESC	Escape	1001 1011	155	9B	233
^\	FS/IS4	File separator	1001 1100	156	9C	234
^]	GS/IS3	Group separator	1001 1101	157	9D	235
^^	RS/IS2	Record separator	1001 1110	158	9E	236
^_	US/IS1	Unit separator	1001 1111	159	9F	237
	SP	Space	1010 0000	160	A0	240
!		Exclamation mark	1010 0001	161	A1	241
''		Quotation mark	1010 0010	162	A2	242
#	NUMB	Number sign	1010 0011	163	A3	243
$	DOLR	Dollar sign	1010 0100	164	A4	244
%		Percent sign	1010 0101	165	A5	245
&		Ampersand	1010 0110	166	A6	246

Table C-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
'		Apostrophe	1010 0111	167	A7	247
(		Left parenthesis	1010 1000	168	A8	250
)		Right parenthesis	1010 1001	169	A9	251
*		Asterisk	1010 1010	170	AA	252
+		Plus sign	1010 1011	171	AB	253
,		Comma	1010 1100	172	AC	254
–		Minus sign	1010 1101	173	AD	255
.		Period	1010 1110	174	AE	256
/		Slash	1010 1111	175	AF	257
0		Zero	1011 0000	176	B0	260
1		One	1011 0001	177	B1	261
2		Two	1011 0010	178	B2	262
3		Three	1011 0011	179	B3	263
4		Four	1011 0100	180	B4	264
5		Five	1011 0101	181	B5	265
6		Six	1011 0110	182	B6	266
7		Seven	1011 0111	183	B7	267
8		Eight	1011 1000	184	B8	270
9		Nine	1011 1001	185	B9	271
:		Colon	1011 1010	186	BA	272
;		Semicolon	1011 1011	187	BB	273
<		Less than sign	1011 1100	188	BC	274
=		Equal sign	1011 1101	189	BD	275
>		Greater than sign	1011 1110	190	BE	276
?		Question mark	1011 1111	191	BF	277
@	AT	Commercial at sign	1100 0000	192	C0	300
A		Uppercase A	1100 0001	193	C1	301
B		Uppercase B	1100 0010	194	C2	302
C		Uppercase C	1100 0011	195	C3	303
D		Uppercase D	1100 0100	196	C4	304
E		Uppercase E	1100 0101	197	C5	305
F		Uppercase F	1100 0110	198	C6	306
G		Uppercase G	1100 0111	199	C7	307
H		Uppercase H	1100 1000	200	C8	310
I		Uppercase I	1100 1001	201	C9	311
J		Uppercase J	1100 1010	202	CA	312
K		Uppercase K	1100 1011	203	CB	313
L		Uppercase L	1100 1100	204	CC	314
M		Uppercase M	1100 1101	205	CD	315
N		Uppercase N	1100 1110	206	CE	316

Table C-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
O		Uppercase O	1100 1111	207	CF	317
P		Uppercase P	1101 0000	208	D0	320
Q		Uppercase Q	1101 0001	209	D1	321
R		Uppercase R	1101 0010	210	D2	322
S		Uppercase S	1101 0011	211	D3	323
T		Uppercase T	1101 0100	212	D4	324
U		Uppercase U	1101 0101	213	D5	325
V		Uppercase V	1101 0110	214	D6	326
W		Uppercase W	1101 0111	215	D7	327
X		Uppercase X	1101 1000	216	D8	330
Y		Uppercase Y	1101 1001	217	D9	331
Z		Uppercase Z	1101 1010	218	DA	332
[	LBKT	Left bracket	1101 1011	219	DB	333
\	REVS	Reverse slash, backslash	1101 1100	220	DC	334
]	RBKT	Right bracket	1101 1101	221	DD	335
^	CFLX	Circumflex	1101 1110	222	DE	336
_		Underline, underscore	1101 1111	223	DF	337
`	GRAV	Left single quote, grave accent	1110 0000	224	E0	340
a		Lowercase a	1110 0001	225	E1	341
b		Lowercase b	1110 0010	226	E2	342
c		Lowercase c	1110 0011	227	E3	343
d		Lowercase d	1110 0100	228	E4	344
e		Lowercase e	1110 0101	229	E5	345
f		Lowercase f	1110 0110	230	E6	346
g		Lowercase g	1110 0111	231	E7	347
h		Lowercase h	1110 1000	232	E8	350
i		Lowercase i	1110 1001	233	E9	351
j		Lowercase j	1110 1010	234	EA	352
k		Lowercase k	1110 1011	235	EB	353
l		Lowercase l	1110 1100	236	EC	354
m		Lowercase m	1110 1101	237	ED	355
n		Lowercase n	1110 1110	238	EE	356
o		Lowercase o	1110 1111	239	EF	357
p		Lowercase p	1111 0000	240	F0	360
q		Lowercase q	1111 0001	241	F1	361
r		Lowercase r	1111 0010	242	F2	362
s		Lowercase s	1111 0011	243	F3	363
t		Lowercase t	1111 0100	244	F4	364

Table C-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
u		Lowercase u	1111 0101	245	F5	365
v		Lowercase v	1111 0110	246	F6	366
w		Lowercase w	1111 0111	247	F7	367
x		Lowercase x	1111 1000	248	F8	370
y		Lowercase y	1111 1001	249	F9	371
z		Lowercase z	1111 1010	250	FA	372
{	LBCE	Left brace	1111 1011	251	FB	373
|	VERT	Vertical line	1111 1100	252	FC	374
}	RBCE	Right brace	1111 1101	253	FD	375
~	TIL	Tilde	1111 1110	254	FE	376
	DEL	Delete	1111 1111	255	FF	377

Indexes

Index

E

F

For your convenience in locating subjects in the three volumes listed below, a composite index is presented on the following pages. Each entry includes one or more two-letter codes indicating the volume or volumes in which the subject is discussed. Each code is followed by one or more chapter-page references. Thus, the entry

BDY (V), AL: 8-15; IS: 2-16

indicates that information on the V-mode BDY instruction can be found on page 8-15 of the Assembly Language Programmer's Guide and on page 2-16 of the Instruction Sets Guide.

Key to Master Index:

Abbreviation	Document Title	Document Number
AL	Assembly Language Programmer's Guide	DOC3059-2LA
IS	Instruction Sets Guide	DOC9474-2LA
SA	System Architecture Reference Guide	DOC9473-2LA

Composite Index

C

N

SURVEY

READER RESPONSE FORM

DOC3059-2LA Assembly Language Programmer's Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   _____
   _____
   _____
   _____
   _____
   _____

3. What faults or errors in this manual gave you problems?

   _____
   _____
   _____
   _____
   _____
   _____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   _____
   _____

Name:_____ Position:_____

Company:_____

Address:_____

_____

_____ Postal Code:_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760