CRAY RESEARCH, INC.

CRAY-1® AND CRAY X-MP COMPUTER SYSTEMS

CAL ASSEMBLER VERSION 1 REFERENCE MANUAL

SR-0000

RECORD OF REVISION CRAY RESEARCH, INC. PUBLICATION NUMBER SR-0000

Each time this manual is revised and reprinted, all changes issued against the previous version in the form of change packets are incorporated into the new version and the new version is assigned an alphabetic level. Between reprints, changes may be issued against the current version in the form of change packets. Each change packet is assigned a numeric designator, starting with 01 for the first change packet of each revision level.

Every page changed by a reprint or by a change packet has the revision level and change packet number in the lower righthand corner. Changes to part of a page are noted by a change bar along the margin of the page. A change bar in the margin opposite the page number indicates that the entire page is new; a dot in the same place indicates that information has been moved from one page to another, but has not otherwise changed.

Requests for copies of Cray Research, Inc. publications and comments about these publications should be directed to:

CRAY RESEARCH, INC.,
1440 Northland Drive,
Mendota Heights, Minnesota 55120

Revision	Description
	April 1975 - Original printing
	June 1975 - This revision corrects various typographical and technical errors.
B	December 1975 - This revision corrects various typographical errors. Changes have been made to the block transfer instructions. The title of this manual has been changed to the Preliminary CAL Assembler Reference Manual.
C	July 1976 - This manual describes CAL Version i, which executes on the CRAY-1. The manual replaces all previous versions of this publication. Since the manual has been completely rewritten, change bars are not used to indicate changes to the manual.
D	December 1976 - This revision provides much of the information relating to conditional pseudo instructions and macro/opdef operations that was missing from revision C.
D-01	June 1977 - Changes in this revision include the addition of the 042i00 instruction, START pseudo instruction, changes to the exchange package format, and additional minor technical changes.
E	August 1977 - This printing is a reprint of revision D with the D-01 packet incorporated. There are no additional changes.
F	April 1978 - Changes in this revision include the addition of DFI and EFI instructions, the MICSIZE pseudo instruction, changes to the mode (M) register, the addition of DEBUG to the CAL control statement, provision for up to 10 system texts, and other minor technical changes.
F-01	July 1978 - This revision includes minor technical changes to bring documentation in line with released version 1.02 of the CAL assembler.

Revision	Description
F-02	October 1978 - This revision brings documentation in line with released version 1.03 of the CAL assembler.
G	January 1979 - This revision is the same as Revision F with change packets F-01 and F-02 incorporated.
G-01	April 1979 - This change packet adds LIST options XNS, NXNX, WEM, and NWEM; describes the Vector Population Instructions Option and the Monitor Mode Interrupt Option; describes the instructions associated with these options; and includes other technical changes that bring the document into agreement with version 1.06 of the CAL assembler.
G-02	July 1979 - This revision includes minor technical changes to bring the document into agreement with version 1.06 of the CAL assembler.
G-03	December 1979 - This revision includes minor technical changes to bring the document into agreement with version 1.07 of the CAL assembler.
G-04	April 1980 - This revision supports all models of the CRAY-1, including the CRAY-1A, CRAY-1B, and CRAY-1 S Series Computer Systems. The publication number has been changed from 2240000 to SR-0000.
H	April 1980 - This reprint is the same as Revision G with change packets G-01, G-02, G-03, and G-04 incorporated.
H-01	October 1980 - This change packet brings the manual into agreement with the released version 1.09. Major changes include binary system text generation with the T parameter in the CAL control statement and new CAL messages.
I	February 1981 - This reprint is the same as Revision H with change packet H-01 incorporated.
I-01	June 1981 - This change packet brings the manual into agreement with the released version 1.10. Errata corrections are also included. Major changes include the addition of TEXT, ENDTEXT, and MODULE pseudo instructions.

I-02 April 1982 - This change packet brings the manual into agreement with version 1.11 of the assembler. Major changes include the following additions: the ALIGN pseudo instruction, the WRP, NWRP, WMR, and NWMR options to the CAL control statement and the LIST pseudo instruction, the predefined micro $QUAL, two warning errors: Y1 - EXTERNAL DECLARATION ERROR and Y2 - MACRO REDEFINED, and a logfile message. Miscellaneous technical and editorial changes are also included.

J February 1983 - This rewrite obsoletes all previous versions of the manual. The manual is reorganized; hardware information has been deleted and can be found in the appropriate Cray mainframe reference manual. Changes include adding CRAY X-MP symbolic machine instructions; the CPU=*type* CAL control statement parameter; and the $CPU predefined micro. This rewrite brings the manual into agreement with the released version 1.12 of the CAL Assembler.

J-01 July 1983 - This change packet brings the manual into agreement with version 1.12 of the assembler. Major changes were the correction of errata.

PREFACE

The CAL Assembler Version 1 allows the user to express symbolically all hardware functions of a mainframe for a Cray Research, Inc., CRAY-1 or CRAY X-MP computer. This detailed and precise level of programming is of special aid in tailoring programs to the architecture of a Cray mainframe and writing programs requiring code that is optimized to the hardware.

Augmenting the instruction repertoire of CAL is a versatile set of pseudo instructions that provides the user with a variety of options for generating macro instructions, controlling list output, organizing programs, etc.

Except where indicated the content of this manual applies to all series of Cray Research, Inc., computers. Detailed information concerning a specific Cray mainframe is given in one of the following Cray mainframe reference manuals:

HR-0029	CRAY-1 S Series Mainframe Reference Manual
HR-0032	CRAY X-MP Series Mainframe Reference Manual
HR-0064	CRAY-1 M Series Mainframe Reference Manual

Detailed information about the Cray Operating System (COS) is presented in separate Cray Research, Inc., publications.

The system macro instructions available with CAL are described in the Macros and Opdefs Reference Manual, CRI publication SR-0012.

CONTENTS

3. SYMBOLIC MACHINE INSTRUCTIONS (continued)

4. PSEUDO INSTRUCTIONS (continued)

4. PSEUDO INSTRUCTIONS (continued)

APPENDIX SECTION

FIGURES

APPENDIX SECTION

FIGURES

TABLES

INDEX

INTRODUCTION 1

The Cray Research, Inc., Cray Assembly Language (CAL) provides the user with a powerful symbolic language for generation of object code to be loaded and executed on the mainframe of a CRAY-1 or CRAY X-MP Computer System.

CAL source statements consist of symbolic machine instructions and pseudo instructions. The symbolic machine instructions provide a means of expressing symbolically all functions of a Cray mainframe. Pseudo instructions allow programmer control of the assembly process.

Features inherent to CAL include:

- Free-field source statement format. Size and location of source statement fields are largely controlled by the user.
- Control of local and common blocks. The programmer can assign code or data segments to specific areas.
- Preloaded data. Data areas can be defined during assembly and loaded with the program.
- Data notation. Data can be designated in integer, floating-point, and character code notation.
- Word and parcel address arithmetic. Addresses can be specified as either word or parcel addresses.
- Binary control. The programmer can specify object code as either absolute or relocatable.
- Listing control. The programmer can control the content of the assembler listing.
- Micro coding. A character string can be defined in a program and substituted for each occurrence of its micro name in the program.
- Macro coding. Sequences of code are defined in a program or on a library, are substituted for each occurrence of the macro name in the program, and use parameters supplied with the macro call.

EXECUTION OF THE CAL ASSEMBLER

The CAL assembler executes under control of the Cray Operating System (COS). It has no hardware requirements beyond those required for the minimum system configuration.

The assembler is loaded and begins executing as a result of the CAL control statement called from a user job deck. Control statement parameters specify characteristics of an assembler run such as the dataset containing source statements and list output. See section 5 of this publication for a description of the CAL control statement.

The source statements can include more than one CAL program module. The assembler assembles each program module as it is encountered on the source dataset. Two passes are made by the assembler for each program module to be assembled. During the first pass, the assembler reads each source language statement instruction, expands sequences such as macro instructions, generates the machine function codes, and assigns memory. During the second pass, the assembler assigns block origins, substitutes values for symbolic operands and addresses, and generates the object code and an associated listing.

The loader is called to load the program module and begin its execution through a control statement in the user's job deck. If the program is relocatable, the loader performs any loading and linking of program modules required to complete the program in memory. These program modules are linked through references to external symbols.

CRAY ASSEMBLY LANGUAGE 2

This section presents the general rules and statement syntax for coding a Cray Assembly Language (CAL) program.

SOURCE LINE FORMAT

A CAL source statement consists of one to eight source lines. A source line is a maximum of 90 characters. The entire line is recorded in the list output dataset generated during a CAL assembly. The assembler interprets only the first 72 columns of a line. Remaining character positions may be ignored.

CONTINUATION LINE

A comma in column 1 indicates a continuation line. Columns 2 through 72 are then a continuation of the previous line. Up to seven continuation lines are allowed. Additional lines beyond seven are treated as comments.

COMMENT STATEMENT

An asterisk in column 1 indicates a comment statement. The assembler lists comment statements, but they have no effect on the program.

STATEMENT FORMAT

With the exception of the comment statement, each statement consists of a location field, a result field, an operand field, and a comment field. Fields are described in the following paragraphs and are separated by one or more blank characters. Statement format is essentially free field.

LOCATION FIELD

The location field begins in column 1 or 2 of a line and is terminated by a blank. The location field has no entry if columns 1 and 2 are blank. The content of the location field consists of a name, symbol, or error code and depends upon the requirements of the result field.

RESULT FIELD

The result field begins with the first nonblank character following the location field. It cannot begin before column 3 or after column 34. A blank terminates the result field. The result field has no entry if only blank characters occur between the location field and column 35. A blank result field following a nonblank location field produces an informative error.

OPERAND FIELD

The operand field begins with the first nonblank character following a nonempty result field and is terminated by one or more blanks. If the result field terminates before column 33, the operand field must begin before column 35; otherwise, the field is considered empty. However, if the result field extends beyond column 32, the operand field must follow one blank separator and can begin after column 35.

COMMENT FIELD

The comment field is optional and begins with the first nonblank character following the operand field or if the operand field is empty, does not begin before column 35. If the result field extends beyond column 32 and no operand entry is provided, two or more blanks must precede the comment field. The comment field can be the only field supplied in a statement.

CODING CONVENTIONS

Although CAL statements are essentially free field, adoption of a convention such as is suggested here provides more uniform and more readable listings.

Beginning Column	Field
1	Blank, asterisk, or comma
1-8	Location field entry, left-justified
9	Blank
10-18	Result field entry, left-justified
19	Blank
20-33	Operand field entry, left-justified
34	Blank
35	Beginning of comment field

LINE EDITING

CAL processes source statements sequentially from the source dataset. A macro or opdef definition is not immediately interpreted but is saved and interpreted each time it is called. Before interpreting a statement, CAL performs two operations referred to as editing: concatenation and micro substitution.

CONCATENATION

CAL examines each line for the underscore (concatenation) character and deletes it so that the two adjoining columns are linked before the statement is interpreted.

MICRO SUBSTITUTION

The CAL assembler searches for quotation marks (") which serve to delimit micro names. The first quotation mark indicates the beginning of a micro name; the second quotation mark identifies the end of a micro name. Before a statement is interpreted, CAL replaces the micro name by the character string comprising the micro.

NAMES

A name is one to eight characters. The first character of a name must be alphabetic (A through Z), a dollar sign ($), a percent sign (%), or an at sign (@). Characters other than the first can also be decimal digits (0 through 9).

Names are used to identify the following types of information:

- Program modules
- Blocks
- Macro instructions
- Micro character strings
- Conditional sequences
- Duplicated sequences

Unlike symbols, a name does not have a value or an attribute associated with it and cannot be used in expressions.

Different types of names do not conflict with each other or with symbols. For example, a micro can have the same name as a macro and a program module can have the same name as a block.

REGISTER DESIGNATORS

A Cray computer system supports the following groups of operating registers:

- 8 address registers represented by A*n* or A.*x*
- 64 intermediate address registers represented by B*n* or B.*x*
- 8 shared intermediate address registers represented by SB*n* or SB.*x*†
- 8 scalar registers represented by S*n* or S.*x*
- 64 intermediate scalar registers represented by T*n* or T.*x*
- 8 shared intermediate scaler registers represented by ST*n* or ST.*x*†
- 8 vector registers represented by V*n* or V.*x*
- 32 semaphore registers represented by SM*n* or SM.*x*†

For the A, SB, S, ST, and V registers, *n* is a single digit in the range 0 through 7 and *x* is a symbol or a numeric constant. The value is truncated and an error is generated if *x* does not have a value in the range 0 through 7.

† Supported on CRAY X-MP Computer Systems only

For the B and T registers, *n* is one or two octal digits in the range 0 through 77 (octal) and *x* is a symbol or a numeric constant. The value is truncated and an error is generated if *x* does not have a value in the range 0 through 77 (octal).

For the SM registers, *n* is one or two octal digits in the range 0 through 37 (octal) and *x* is a symbol or a numeric constant. The value is truncated and an error is generated if *x* does not have a value in the range 0 through 37 (octal).

If *x* is a symbol, it can be used before it is defined but must be defined before program end. The symbol is evaluated during pass 2.

For additional information on registers, see the appropriate Cray mainframe reference manual.

SYMBOLS

A symbol is one to eight characters that identifies a value and its associated attributes (see following description of symbol attributes). The first character of a symbol must be alphabetic (A through Z), a dollar sign ($), a percent sign (%), or an at sign (@). Characters other than the first can also be decimal digits (0 through 9).

A warning error is issued if a symbol is defined as one of the following register designators. CRAY X-MP specific registers get a warning error only when CAL is generating code for a CRAY X-MP mainframe (see CPU=*type* option on the CAL control statement).

A*n*, FA*n*
B*n*, B*nn*
SB*n*†
S*n*, FS*n*, PS*n*, ZS*n*, QS*n*
T*n*, T*nn*
ST*n*†
V*n*, FV*n*, PV*n*, ZV*n*, QV*n*
SM*n*,† SM*nn*†
RT, VM, CA, CL, CE, XA, VL, CI, SB, SM,† MC†

In the above, *n* is a single octal digit.

† Supported on CRAY X-MP Computer Systems only

SYMBOL DEFINITION

The process of associating a symbol with a value and attributes is known as symbol definition. This association can occur in the following ways.

- A symbol used in the location field of a symbolic machine instruction or certain pseudo instructions is defined as an address having the current value of the location counter and having parcel-address or word-address attributes and relocatable or absolute attributes.

- A symbol used in the location field of a symbol-defining pseudo instruction is defined as having the value and attributes derived from an expression in the operand field of instruction. The type of symbol-defining pseudo instruction can cause the symbol to have a redefinable attribute. When a symbol is redefinable, a second attempt to define it must be through use of a redefinable pseudo instruction causing the symbol to be assigned a new value and attributes.

- A symbol defined in a program module other than the module being currently assembled can be defined as having the attribute of external in the current program module. The true value of an external symbol is not known within the current program module.

SYMBOL ATTRIBUTES

Two or more attributes are assigned to a symbol when it is defined. These attributes are described in the following paragraphs.

Word address, parcel address, or value

Each symbol is assigned an attribute of word address, parcel address, or value. A symbol is assigned a word-address attribute if it appears in the location field of a pseudo instruction such as a BSS or BSSZ which defines words or if it is equated to an expression having a word-address attribute. A 22-bit value is associated with a word-address symbol.

A symbol is assigned a parcel-address attribute if it appears in the location field of a symbolic machine instruction or certain pseudo instructions. A 24-bit value is associated with a parcel-address symbol.

A symbol has a value attribute if it does not have a word-address or parcel-address attribute. A 64-bit value is associated with a value symbol.

Relocatable, external, or absolute

Each symbol is assigned the attribute of relocatable, external, or absolute.

A symbol is assigned an attribute of relocatable if it appears in a relocatable assembly in the location field of a machine instruction, BSS pseudo instruction, or data generation pseudo instruction such as BSSZ, CON, etc. A symbol is also relocatable if it is equated to an expression that is relocatable.

A symbol is assigned the attribute of external if it is defined by an EXT pseudo instruction. An external symbol defined in this manner has a value attribute and a value of 0. A symbol is also assigned the attribute of external if it is equated to an expression that is external. Such a symbol assumes the value of the expression and can have an attribute of parcel address, word address, or value.

If a symbol is neither relocatable nor external, it is assigned the attribute of absolute in a relocatable assembly. In an absolute assembly, symbols that would be relocatable in a relocatable assembly are assigned the attribute of absolute. An exception occurs when the absolute program module is divided into local blocks through use of BLOCK pseudo instructions. In this case, symbols defined in local blocks other than the initial (nominal) block are assigned an attribute of relocatable during pass 1 and absolute during pass 2. The use of blocks is described further under Block Control in section 4.

Common

A relocatable symbol is assigned an additional attribute of common if it is defined in a common block rather than a local block. Common blocks are allowed only in relocatable assemblies. The use of common blocks is described under Block Control in section 4.

Redefinable

In addition to its other attributes, a symbol is assigned the attribute of redefinable if it is defined by certain pseudo instructions such as SET. A redefinable symbol can be defined more than once in a program module and can have different values and attributes at different times during an assembly. When such a symbol is referenced, its most recent definition is used by the assembler.

SYMBOL REFERENCE

The occurrence of a symbol in a field other than the location field constitutes a reference to the symbol and causes the value and attributes of the symbol to be used in place of the symbol.

A symbol reference can contain a prefix which causes the usual value and attributes associated with the symbol to be altered according to the prefix. The prefix affects only the specific reference with which it occurs. For details, refer to Prefixed Symbols or Constants later in this section.

QUALIFIED SYMBOLS

A symbol other than a global symbol can be rendered unique to a code sequence by specifying a symbol qualifier to be appended to all symbols defined within the sequence. The option to qualify symbols is initiated by one QUAL pseudo instruction and terminated by the next. If a symbol defined in the code sequence is referred to from within the sequence, it can be referred to without qualification. If, however, the symbol is referred to from outside of the code sequence in which it was defined, it must be referred to in the form */qualifier/symbol*, where *qualifier* is a 1-character to 8-character name and is defined through the use of a QUAL pseudo instruction.

GLOBAL DEFINITIONS

Before the first IDENT pseudo instruction and between program modules (that is, between the END pseudo that terminates one program module and the IDENT that begins the next program module), CAL recognizes sequences of instructions that do not generate code but define symbols, macro and opdef instructions, and micros.

Definitions occurring before an IDENT pseudo instruction are considered global and can be referred to without redefinition from within any of the program modules that occur subsequent to the definition. Micros, redefinable symbols, and symbols of the form %%*xxxxxx*, where *x* is any nonblank character, represent an exception. While they can occur in such sequences, they are local to the program module that follows and are not known to the assembler after the next END pseudo instruction is encountered. Global symbols cannot be qualified.

SPECIAL ELEMENTS

The following designators can occur as elements of expressions and have special meaning to the assembler.

- `*` Denotes a value equal to the current location counter with parcel-address attribute and absolute or relocatable attribute, depending on type of assembly

- `*O` Denotes a value equal to the current value of the origin counter with parcel-address attribute and absolute or relocatable attribute

- `*W` Denotes a value equal to the current value of the word-bit-position counter with absolute and value attributes

- `*P` Denotes a value equal to the current value of the parcel-bit-position counter with absolute and value attributes

Expression elements are described later in this section. Counters are described under Block Control in section 4.

DATA NOTATION

In this publication, italicized lowercase letters, numbers, or symbols indicate variable information. Use of underlining in presenting parameter options indicates default options. Use of parenthesis ( ) indicates optional information; use of brackets [ ] indicates required information.

Data can be in the form of numeric or character constants, data items, or literals. These forms are described and illustrated in the following paragraphs.

NUMERIC CONSTANTS

A numeric constant can be expressed in integer or floating-point notation. An integer constant has the following format:

(*prefix*) [*integer*] (*binary scale*)

A floating-point constant has the following format:

```
        [integer.         ]
(prefix)[integer.fraction](decimal exponent)(binary scale)
        [        .fraction]
or
(prefix)[integer][decimal exponent](binary scale)
```

prefix Numeric base used for the *integer*, *fraction*, *decimal exponent*, and *binary scale*. If no prefix is used, base is determined by the default mode of the assembler or by the BASE pseudo instruction. *prefix* can be one of the following:

O'	Octal (integer only)
D'	Decimal (default mode)
X'	Hexadecimal (integer only)

integer and/or *fraction*
A non-empty string of digits as required by *prefix*

decimal exponent
Power of 10 by which the *integer* and/or *fraction* is to be multiplied; indicates whether the constant is to be single precision (one 64-bit word) or double precision (two 64-bit words). *n* is an integer in the base specified by *prefix*. If no *decimal exponent* is provided, the constant occupies one word.

E*n* or E+*n*	Positive decimal exponent, single precision
E-*n*	Negative decimal exponent, single precision
D*n* or D+*n*	Positive decimal exponent, double precision
D-*n*	Negative decimal exponent, double precision

binary scale
The *integer* and/or *fraction* is to be multiplied by a power of 2. *n* is an integer in the base specified by *prefix*.

S*n* or S+*n*	Positive binary exponent
S-*n*	Negative binary exponent

An integer constant is evaluated as a 64-bit twos-complement integer. Refer to figure 3-6 in section 3 for the twos-complement integer formats. A floating-point constant is evaluated as a 1-word or 2-word quantity, depending on the precision specified. See figure 3-7 in section 3 for the floating-point data formats.

Example:

```
Location  Result  Operand          Comment
1         10      20               35

          CON     D'1.5
          A4      O'50
          CON     D'1.0E-6
          VWD     40/0,D'24/ADDR
SYM       =       O'1777752
          CON     1S63             sign bit
```

CHARACTER CONSTANTS

Character constants are expressed using the following format:

(*prefix*) ['*character string*'] (*suffix*)

prefix Character set used for stored constant:

- A ASCII character set (default)
- C Control Data Display Code
- E EBCDIC character set

character string

Appendix F lists the character set.

A string of zero or more characters from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate a single apostrophe.

suffix Justification and fill of character string:

- H Left-justified, blank fill (default)
- L Left-justified, zero fill
- R Right-justified, zero fill
- Z Left-justified, zero fill, at least one trailing binary zero character guaranteed

Example:

Location	Result	Operand	Comment
1	10	20	35
	S3	'*'R	
	CON	A'ABC'L	
	VWD	24/'OUT'	

DATA ITEMS

A character or data item can be used in the operand field of the DATA, CON, and VWD pseudo instruction and in literals. The length of the data field occupied by a data item is determined by its type and size.

An integer data item has the following format:

(*sign*) (*prefix*) [*integer*] (*binary scale*)

A floating-point data item has the following format:

(*sign*) (*prefix*) [*integer*.] (*decimal exponent*) (*binary scale*)
[*integer*.*fraction*]
[.*fraction*]

or

(*sign*) (*prefix*) [*integer*] [*decimal exponent*] (*binary scale*)

An integer data item occupies one 64-bit word. A floating-point data item occupies one word if single precision and two words if double precision.

A character string data item has the following format:

(*prefix*) ['*character string*'] (*count*) (*suffix*)

In the above notation, descriptions given for numeric and character constants apply. The two added options, *sign* for numeric data items and *count* for character string data items, have the following significance:

sign Data item is to be stored ones or twos complemented or uncomplemented

+ or omitted	Uncomplemented
-	Negated (twos complemented)
#	Ones complemented

count Length of the field in number of characters into which the data item is to be placed. *count* can only be used with a DATA pseudo instruction. If *count* is not supplied, the length is the number of words needed to hold the character string. If a count field is present, the length is the character count times the character width, so length is not necessarily an integral number of words. The character width is 8 bits for ASCII or EBCDIC, 6 bits for Control Data Display Code.

If an asterisk is in the count field, then the actual number of characters in the string is used as the count. The case where two apostrophes are used to represent a single apostrophe is counted as a single character.

If the base is M (mixed), CAL assumes that count is decimal. Refer to section 4 for a description of mixed base.

Example:

Location	Result	Operand	Comment
1	10	20	35
	DATA	'ERROR IN DSN'	
	DATA	-D'1.5E2	
	DATA	+O'20	
	VWD	40/0,24/O'200	

LITERALS

A literal is a read-only constant and has the following format when used as an element of an expression. *data item* represents any of the formats for data items previously described.

= [*data item*]

The first use of a literal value in an expression causes the assembler to store the data item in one or more words in a special, local block known as the literals block. The value used in the expression in place of the literal data item is the address at which the literal is stored. A subsequent reference to the literal value in an expression does not cause another store into the literals block; the address of the previously stored value is again used. This process avoids duplication of read-only data. A reference to a literal does not cause generation of new entries if the bit pattern of words previously stored in the literals block matches the bit pattern of the new data.

Because the address of literal, rather than its value, is used in evaluating expression elements, a literal has an attribute of relocatable in a relocatable assembly and during pass 1 of an absolute assembly. However, a literal has an attribute of absolute on pass 2 of an absolute assembly.

Examples of literals:

```
Location  Result  Operand      Comment
1         10      20           35

          A2      =O'101       Load address of word
                               containing 101
          S3      ='A'
          S4      =-2.1E2,0    Load -2.1E2 to S4
```

PREFIXED SYMBOLS, CONSTANTS, OR SPECIAL ELEMENTS

A symbol, constant, or special element can be prefixed by a P. or a W. causing the value to assume an attribute of parcel address or word address, respectively, in the expression in which the reference appears.

A prefix does not permanently alter the attribute of a symbol; the effect of a prefix is for the current reference only.

P. - PARCEL-ADDRESS PREFIX

A symbol, special element, or constant can be prefixed by P. to specify the attribute of parcel address. If a symbol, *sym*, has the attribute of word address, the value of P.*sym* is the value of *sym* multiplied by four. A P. prefix to a symbol with value attribute or to a constant does not cause the value to be multiplied by four but it can be used to assign the parcel-address attribute.

Example:

```
Location  Result  Operand      Comment
1         10      20           35

ADDR      CON     P.ADDR
          JAZ     *+P.10
```

W. - WORD-ADDRESS PREFIX

A symbol, special element, or constant can be prefixed by W. to specify the attribute of word address. If a symbol, *sym*, has the attribute of parcel address, the value of W.*sym* is the value of *sym* divided by four. A W. prefix to a symbol with value-address attribute or to a constant does not cause the value to be divided by four but it can be used to assign the word-address attribute to the symbol or constant.

Examples:

Location	Result	Operand	Comment
1	10	20	35
	A0	W.ADDR	
	A4	W.BUFF+O'100	

EXPRESSIONS

The result and operand fields for many source statements consist of entries known to CAL as expressions. An expression consists of one or more terms joined by special characters referred to as adding operators. A blank or a comma terminates an expression. A term consists of one or more elements joined by special characters referred to as multiplying operators. Thus, an expression can be diagrammed as follows:

Add op_1 (optional)	$Term_1$	Add op_2	$Term_2$ . . .	Add op_n	$Term_n$	comma or blank

Any term in an expression can be diagrammed as follows:

$Element_1$	Mult op_1	$Element_2$ . . .	Mult op_m	$Element_m$

The multiplying operators complete all multiplication and division before the adding operators complete addition or subtraction.

ADDING OPERATORS

An adding operator joins two terms or precedes the first term of an expression. The two adding operators are:

+	Addition
-	Subtraction

MULTIPLYING OPERATORS

A multiplying operator joins two elements. Multiplying operators are:

*	Multiplication
/	Division

ELEMENTS

An element is a symbol, constant, literal, or special element. It can also be one of these preceded by a complement (#) operator. However, an element preceded by # must be absolute. Examples of elements follow.

SIGMA	Symbol	O'77S3	Numeric constant
*	Special element	A'ABC'R	Character constant
*W	Special element	=A'ABC'	Literal

Attributes of elements are assigned by using the SET or = pseudo instructions to define the attributes or by implication when the element is used.

TERMS

A term is an element or two or more elements joined by multiplying operators. Only one relocatable or external element can occur in a term. The following rules apply for terms.

- Two consecutive elements are illegal.
- The element to the right of / must be an absolute element; that is, it must be a constant or an absolute symbol or, in an absolute assembly, a literal or a special element as well as a constant or an absolute symbol.
- An external symbol, if present, must be the only element of the term and if preceded by an adding operator, that operator must be +.

- An element cannot be null; that is, two consecutive multiplying operators or a multiplying operator not followed by an element is illegal.

- A term containing / must have an attribute of absolute up to the point at which the / is encountered (see the description of term attributes).

- Division by 0 produces an error.

TERM ATTRIBUTES

Attributes assigned to a term depend on the elements and operators comprising the term.

Every term is assigned an attribute of either external, absolute, or relocatable. A term assumes the attribute of external if it consists of a single external symbol. A term assumes the attribute of absolute if it contains only absolute elements. A term assumes an attribute of relocatable if it contains one relocatable element and no external symbols.

Every term assumes an attribute of parcel address, word address, or value. The term attribute can vary as each element in the term is evaluated. The term's final attribute is the attribute in effect when the final (rightmost) element of the term is evaluated. As CAL encounters each element in the left-to-right scan of a term, it assigns an attribute to the term based on the operator, if any, preceding the element, the attribute of any previous partial term, and the attribute of the element currently being evaluated.

In the following rules, consider that P, W and V denote an element being incorporated into the term and having an attribute of parcel address, word address, or value, respectively. Consider, also, that *pterm*, *wterm*, and *vterm* denote the attribute of the partial term resulting from all elements evaluated before the current element. The following rules can then be stated.

- Following evaluation of the element, a new partial term is assigned a parcel-address attribute if the partial term, operator, and new element are one of the following combinations:

  P
  *pterm**V
  pterm/V
  *vterm**P

- Following evaluation of the element, a new partial term is assigned a word-address attribute if the partial term, operator, and new element are one of the following combinations:

  W
  *wterm**V
  wterm/V
  *vterm**W

- Following evaluation of the element, a new partial term is assigned a value attribute if the partial term, operator, and new element are one of the following combinations:

  V
  *vterm**V
  pterm/P
  wterm/W
  vterm/V

- In addition, any of the following combinations results in an attribute of value being assigned but is accompanied by a warning error.

  *pterm**W
  *wterm**P
  pterm/W
  wterm/P
  vterm/P
  vterm/W
  *pterm**P
  *wterm**W

EXPRESSION EVALUATION

Expressions are evaluated from left to right. Each term is evaluated from left to right, with CAL performing 64-bit integer multiplication or division as each multiplying operator is encountered. When a complete term has been evaluated, it is added or subtracted from the sum of the previous terms.

The assembler treats each element as 64-bit twos-complement integer. Sign extension is performed for elements with 22-bit (word-address) or 24-bit (parcel address) values. Character constants are left-justified or right-justified within a field width equal to the destination field. Complemented elements are complemented in the rightmost bits in a field width equal to the destination field.

A relocatable term has a 64-bit integer coefficient associated with it, equal to the value of the term obtained when a one is substituted for the relocatable element. The value of a relocatable term is the value of the relocatable element multiplied by the coefficient.

The coefficient of each relocatable term is added to the coefficient or subtracted from the coefficient maintained for the corresponding relocatable block represented in the expression.

EXPRESSION ATTRIBUTES

Expressions can be assigned the following attributes by the assembler.

- Relocatable, external, or absolute
- Parcel address, word address, or value

RELOCATABLE, EXTERNAL, or ABSOLUTE

An expression is relocatable if the coefficient for every block represented in the expression is 0, except for one block which must have a coefficient of +1 (positive relocation). An expression error occurs if a coefficient does not equal 0 or +1, or if more than one coefficient is nonzero.

An expression is external if the expression contains one external term and if the coefficients of all relocatable blocks are 0. An expression error occurs if more than one external term is present.

An expression is absolute if no external terms are present and the coefficients of all relocatable blocks are 0.

PARCEL ADDRESS, WORD ADDRESS, OR VALUE

An expression has parcel-address attribute if at least one term has parcel-address attribute and all other terms have value or parcel-address attributes.

An expression has word-address attribute if at least one term has word-address attribute and all other terms have value or word-address attributes.

All other expressions have value attributes. A warning error occurs if an expression has terms with both word-address attributes and parcel-address attributes.

An expression value is truncated to the field size of the expression destination. A warning error occurs if the leftmost bits lost in truncation are not all zeros or all ones with the leftmost remaining bit also 1 (that is, a negative quantity).

A null (empty) expression is treated as an absolute value of 0.

If an error other than a warning error occurs in evaluating an expression, the expression is treated as an absolute value of 0.

Examples of expressions:

ALPHA	An expression consisting of a single term
*W+BETA	Two terms; *W and BETA.
GAMMA/4+DELTA*5	Two terms, each consisting of two elements
MU-NU*2+*	Three terms; the first consisting only of MU, the second consisting of NU*2, and the third consisting only of the special element *.
O'100+=O'100	Two terms; a constant and the address of a literal.

In the following examples, R and S are relocatable symbols in the same block, COM is relocatable in a common block, X and Y are external, and A and B are absolute. The location counter is currently in the block containing R and S.

The following expressions are relocatable:

*	
W.*+B	
R+2	
COM+R-S	R and S cancel
3**-R-S	3** cancels -R and -S
=A'LITERAL'	Relocatable except in an absolute assembly, pass 2
X+R	Error; external and relocatable.
R+S	Error; relocation coefficient of 2.
R/16*16	Error; division of relocatable element is illegal.

The following expressions are external:

X+2	
Y-100	
X+R-*	R, -* cancel relocation
X+2**-R-S	Relocatable terms 2**, -R, -S cancel each other
-X+2	Error; external cannot be negated.
X+Y	Error; more than one external.
X/Z	Error; division of an external element is illegal.

The following expressions are absolute:

```
A+B
'A'R-1
2*R-S-*          Relocation of terms all cancel
1/2*R            Equivalent to 0*R
A*(R-S)          Error; parentheses not allowed
```

CHART METHOD OF EXPRESSION ATTRIBUTE EVALUATION

As shown in the following charts, if a symbol, literal, special element, or constant has the attribute of the left column (1st Term) and is added, subtracted, multiplied, or divided by a symbol, literal, special element, or constant with the attribute of the top horizontal row (2nd Term), then the resulting attribute is determined at the intersection of the column and row by the arithmetic operator position (upper left corner of table).

+ | - / * | /	V	P	W
V	V | V / V | V	P | P / P | V³	W | W / W | V³
P	P | P / P | P	P | P / V³ | V	V³ | V³ / V³ | V³
W	W | W / W | W	V³ | V³ / V³ | V³	W | W / V³ | V

2nd Term

1st Term

V - Value
P - Parcel
W - Word
3 - Warning Message

+ | - / * | /	V	P	W
V	v | v / v | v	p | x / x^3 | x^3	w | x / x | x^3
P	p | p / x | x	x | p^* / x^3 | x	x^3 | x^3 / x^3 | x^3
W	w | w / x | x	x^3 | v^3 / x^3 | x^3	x | w^* / x^3 | x

2nd Term

1st Term

V - Value
P - Parcel
W - Word
X - Error Message
3 - Warning Message
* - Absolute

SYMBOLIC MACHINE INSTRUCTIONS 3

Each CRAY-1 or CRAY X-MP mainframe machine instruction can be represented symbolically in Cray Assembly Language (CAL). The assembler identifies a symbolic instruction according to its syntax and generates a binary machine instruction in the object code. An instruction is generated in the block in use when the instruction is interpreted.

INSTRUCTION FORMAT

Each instruction is either a 1-parcel (16-bit) instruction or a 2-parcel (32-bit) instruction. Instructions are packed four parcels per word. Parcels are numbered 0 through 3 from left to right and any parcel position can be addressed in branch instructions. A 2-parcel instruction begins in any parcel of a word and can span a word boundary. For example, a 2-parcel instruction beginning in the fourth parcel of a word ends in the first parcel of the next word. No padding to word boundaries is required. Figure 3-1 illustrates the general form of instructions.

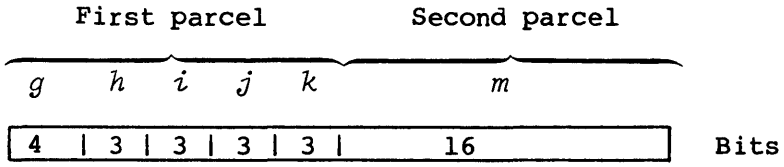


Figure 3-1. General form for instructions

Four variations of this general format use the fields differently; two forms are 1-parcel formats and two are 2-parcel formats. The formats of these four variations are described below.

1-PARCEL INSTRUCTION FORMAT WITH DISCRETE *j* AND *k* FIELDS

The most common of the 1-parcel instruction formats uses the *i*, *j*, and *k* fields as individual designators for operand and result registers (see figure 3-2). The *g* and *h* fields define the operation code. The *i* field designates a result register and the *j* and *k* fields designate

operand registers. Some instructions ignore one or more of the *i*, *j*, and *k* fields. The following types of instructions use this format.

- Arithmetic
- Logical
- Double shift
- Floating-point constant

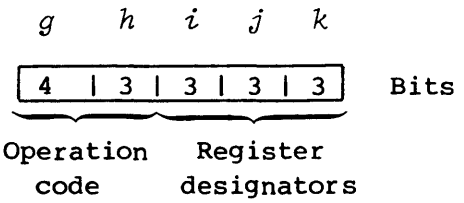


Figure 3-2. 1-parcel instruction format with discrete *j* and *k* fields

1-PARCEL INSTRUCTION FORMAT WITH COMBINED *j* AND *k* FIELDS

Some 1-parcel instructions use the *j* and *k* fields as a combined 6-bit field (see figure 3-3). The *g* and *h* fields contain the operation code, and the *i* field is generally a destination register identifier. The combined *j* and *k* fields generally contain a constant or a B or T register designator. The branch instruction 005 and the following types of instructions use the 1-parcel instruction format with combined *j* and *k* fields.

- Constant
- B and T register block memory transfer
- B and T register data transfer
- Single shift
- Mask

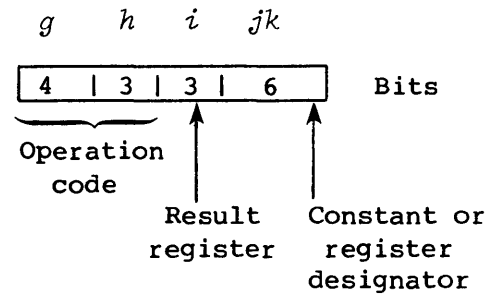


Figure 3-3. 1-parcel instruction format with combined *j* and *k* fields

2-PARCEL INSTRUCTION FORMAT WITH COMBINED *j*, *k*, AND *m* FIELDS

The instruction type for a 22-bit immediate constant uses the combined *j*, *k*, and *m* fields to hold the constant. The 7-bit *gh* field contains an operation code, and the 3-bit *i* field designates a result register. The instruction type using this format transfers the 22-bit *jkm* constant to an A or S register.

The instruction type used for scalar memory transfers also requires a 22-bit *jkm* field for an address displacement. This instruction type uses the 4-bit *g* field for an operation code, the 3-bit *h* field to designate an address index register, and the 3-bit *i* field to designate a source or result register. (See special register values.)

Figure 3-4 shows the two general applications for the 2-parcel instruction format with combined *j*, *k*, and *m* fields.

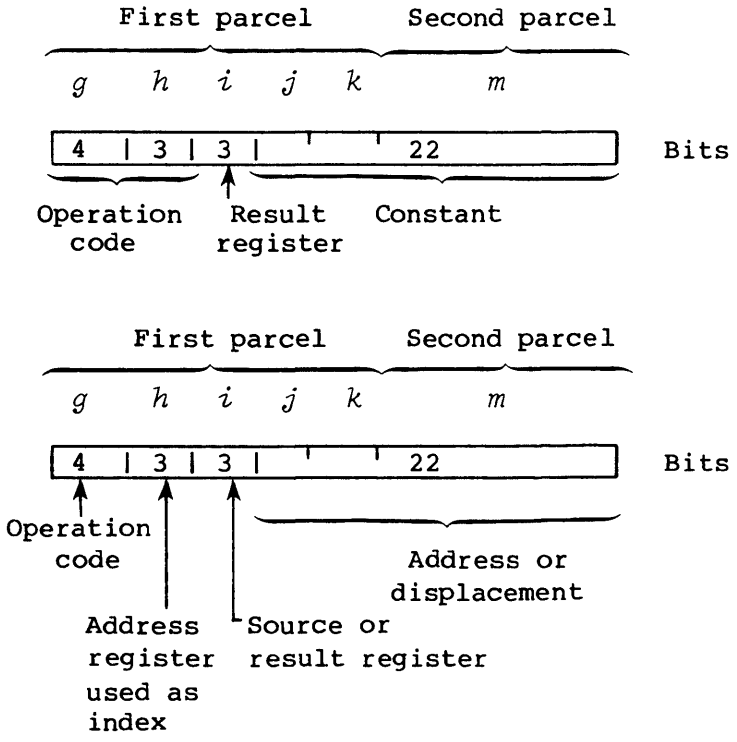


Figure 3-4. 2-parcel instruction format with combined *j*, *k*, and *m* fields

NOTE

When using an immediate constant having a parcel value, and that is relocatable, the result of the relocation will be incorrect if the loader-determined actual address within the user's field length is greater than 1,048,575 because the resulting relocated value will have more than 22 significant bits.

2-PARCEL INSTRUCTION FORMAT WITH COMBINED *i*, *j*, *k*, AND *m* FIELDS

The 2-parcel branch instruction type uses the combined *i*, *j*, *k*, and *m* fields to contain a 24-bit address that allows branching to an instruction parcel (see figure 3-5). A 7-bit operation code (*gh*) is followed by an *ijkm* field. The high-order bit of the *i* field is unused.

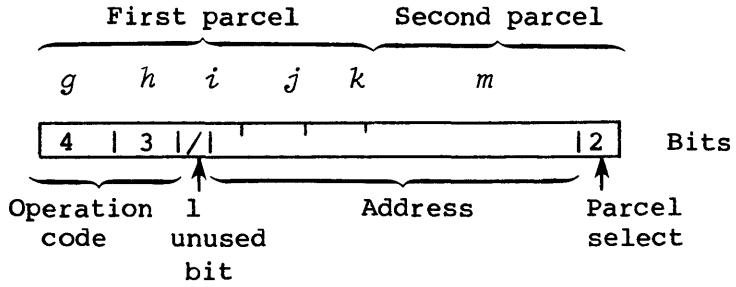


Figure 3-5. 2-parcel instruction format with combined *i*, *j*, *k*, and *m* fields

SPECIAL REGISTER VALUES

If the S0 and A0 registers are referenced in the *j* or *k* fields of certain instructions, the contents of the respective register are not used; instead, a special operand is generated. The special value is available regardless of existing A0 or S0 reservations (and in this case are not checked). This use does not alter the actual value of the S0 or A0 register. If S0 or A0 is used in the *i* field as the operand, the actual value of the register is provided. The table below shows the special register values.

Field	Operand value
A*h*, *h*=0	0
A*i*, *i*=0	(A0)
A*j*, *j*=0	0
A*k*, *k*=0	1
S*i*, *i*=0	(S0)
S*j*, *j*=0	0
S*k*, *k*=0	2**63

SYMBOLIC NOTATION

This section describes the notation used for coding symbolic machine instructions. Instructions are described in the following functional categories:

- Register entries
- Inter-register transfers
- Memory transfers
- Integer arithmetic operations
- Floating-point arithmetic operations
- Logical operations
- Bit counts
- Shift operations
- Program branches and exits
- Monitor operations

Within each functional category, each instruction is presented, explained, and followed by an example. Instructions are summarized by functional category below. Appendix A of this publication contains a cross reference to this section with the octal machine code as the primary index. For descriptions of functional units, see the appropriate Cray mainframe reference manual.

GENERAL REQUIREMENTS

Register designators and the location, result, and operand fields have the following general requirements.

SYMBOLIC MACHINE INSTRUCTIONS

LOGICAL OPERATIONS

Si Sj&Sk	Vi Sj&Vk	Vi Vj&Vk
Si Sj&SB		
Si SB&Sj		
Si #Sk&Sj		
Si #SB&Sj		
Si Sj!Sk	Vi Sj!Vk	Vi Vj!Vk
Si Sj!SB		
Si SB!Sj		
Si Sj\Sk	Vi Sj\Vk	Vi Vj\Vk
Si Sj\SB		
Si SB\Sj		
Si #Sj\Sk		
Si #Sj\SB		
Si #SB\Sj		
	VM Vj,Z	
	VM Vj,N	
	VM Vj,P	
	VM Vj,M	
Si Sj!Si&Sk		
Si Sj!Si&SB	Vi Sj!Vk&VM	Vi Vj!Vk&VM
	Vi #VM&Vk	

INTEGER ARITHMETIC OPERATIONS

Ai Aj+Ak		
Ai Aj+1		
Ai Aj-Ak		
Ai Aj-1		
Ai Aj*Ak		
Si Sj+Sk	Vi Sj+Vk	Vi Vj+Vk
Si Sj-Sk	Vi Sj-Vk	Vi Vj-Vk

SHIFT INSTRUCTIONS

S0 Si<exp	S0 Si>exp
Si Si<exp	Si Si>exp
Si Si,Sj<Ak	Si Sj,Si>Ak
Si Si,Sj<1	Si Sj,Si>1
Si Si<Ak	Si Si>Ak
Vi Vj<Ak	Vi Vj>Ak
Vi Vj<1	Vi Vj>1
Vi Vj,Vj<Ak	Vi Vj,Vj>Ak
Vi Vj,Vj<1	Vi Vj,Vj>1

REGISTER ENTRY INSTRUCTIONS

Ai exp	Si <exp
Ai -1	Si #>exp
	Si >exp
Si exp	Si #<exp
Si 0	Si SB
Si 1	Si #SB
Si -1	
Si 1.	Vi,Ak 0
Si 2.	Vi 0
Si 4.	
Si 0.4	SMjk 1 †, TS†
Si 0.6	SMjk 0†
	SMjk 1†

FLOATING-POINT OPERATIONS

EFI		
DFI		
Si Sj+FSk	Vi Sj+FVk	Vi Vj+FVk
Si +FSk	Vi +FVk	
Si Sj-FSk	Vi Sj-FVk	Vi Vj-FVk
Si -FSk	Vi -FVk	
Si Sj*FSk	Vi Sj*FVk	Vi Vj*FVk
Si Sj*HSk	Vi Sj*HVk	Vi Vj*HVk
Si Sj*RSk	Vi Sj*RVk	Vi Vj*RVk
Si Sj*ISk	Vi Sj*IVk	Vi Vj*IVk
Si /HSj		Vi /HVj

PROGRAM BRANCHES AND EXITS

J exp	
J Bjk	
JAZ exp	JSZ exp
JAN exp	JSN exp
JAP exp	JSP exp
JAM exp	JSM exp
R exp	
EX	ERR
EX exp††	ERR exp††

BIT COUNT INSTRUCTIONS

Ai PSj	Vi PVj
Ai QSj	Vi QVj
Ai ZSj	

MONITOR OPERATIONS

CA,Aj Ak	CCI
CL,Aj Ak	ECI
CI,Aj	DCI
MC,Aj†	ERI†
XA Aj	DRI†
RT Sj	CLN 0†
PCI Sj	CLN 1†
IP 1†	CLN 2†
IP 0†	CLN 3†

INTER-REGISTER TRANSFERS

Ai Ak	Si Sk
Ai -Ak	Si -Sk
	Si #Sk
Ai Sj	Si Ak
	Si +Ak
Ai VL†	Si +FAk
Ai Bjk	Si Tjk
Ai SBj†	Si STj†
Ai CI	Si Vj,Ak
Ai CA,Aj	Si VM
Ai CE,Aj	Si RT
	Si SM†
	Si SRj†
Bjk Ai	Tjk Si
SBj Ai†	STj Si†
	Vi Vk
	Vi -Vk
	Vi,Ak Sj
VL Ak	VM Sj
VL 1	VM 0
	SM Si†

MEMORY TRANSFERS

DBM†		
EBM†		
CMR†		
	(store)	(load)
	,A0 Bjk,Ai	Bjk,Ai ,A0
	0,A0 Bjk,Ai	Bjk,Ai 0,A0
	,A0 Tjk,Ai	Tjk,Ai ,A0
	0,A0 Tjk,Ai	Tjk,Ai 0,A0
	exp,Ah Ai	Ai exp,Ah
	exp,0 Ai	Ai exp,0
	exp, Ai	Ai exp,
	,Ah Ai	Ai ,Ah
	exp,Ah Si	Si exp,Ah
	exp,0 Si	Si exp,0
	exp, Si	Si exp,
	,Ah Si	Si ,Ah
	,A0,Ak Vj	Vi ,A0,Ak
	,A0,1 Vj	Vi ,A0,1

REGISTER	VALUE
Ah, h=0	0
Ai, i=0	(A0)
Aj, j=0	0
Ak, k=0	1
Si, i=0	(S0)
Sj, j=0	0
Sk, k=0	2^{63}

LOGICAL OPERATORS

&	0101
	1100
	0100
!	0101
	1100
	1101
\	0101
	1100
	1001

† CRAY X-MP Computer Systems only
†† CRAY-1 Computer Systems only

Register designators

A, B, SB†, S, T, ST†, SM†, and V registers can be referenced with numeric or symbolic designators as described in section 2.

In the symbolic notation, the *h*, *i*, *j*, and *k* designators indicate the field of the machine instruction into which the register designator constant or symbol value is placed. An expression (*exp*) occupies the *jk*, *ijk*, *jkm*, or *ijkm* fields depending on the operation code and magnitude of the expression value.

Supporting registers have the following designators:

CA	Current Address
CL	Channel Limit
CI	Channel Interrupt flag
CE	Channel Error flag
RT	Real-time Clock
SB	Sign bit (S*k*, with *k*=0)
VL	Vector Length
VM†	Vector Mask
XA	Exchange Address
MC†	Master Clear
SM†	Semaphore

Location field

The location field of a symbolic instruction optionally contains a location symbol. When a symbol is present, it is assigned a parcel address as indicated by the value of the location counter after the force to parcel boundary occurs.

Result field

The result field of a symbolic machine instruction can consist of one, two, or three subfields separated by commas. A subfield can be null or can contain a register designator or an expression specifying a memory address which indicates the register or memory location to receive the results of the operation. The result field, in some cases, contains a mnemonic indicating the function being performed (for example, J for jump or EX for exit).

† CRAY X-MP Computer Systems only

Operand field

The operand field of a symbolic machine instruction consists of no subfield or one or more subfields separated by commas. A subfield can be null, can contain an expression (with no register designators), or can consist of register designators and operators.

The following special characters can appear in the operand field of symbolic machine instructions and are used by the assembler in determining the operation to be performed.

+	Arithmetic sum of adjoining registers
-	Arithmetic difference of adjoining registers
*	Arithmetic product of adjoining registers
/	Reciprocal of approximation
#	Use ones complement
>	Shift value or form mask from left to right
<	Shift value or form mask from right to left
&	Logical product of adjoining registers
!	Logical sum of adjoining registers
\	Logical difference of adjoining registers

In some instructions, register designators are prefixed by the following letters which have special meaning to the assembler:

F	Floating-point operation
H	Half-precision operation
R	Rounded operation
I	Reciprocal iteration
P	Population count
Q	Parity count
Z	Leading-zero count

SPECIAL SYNTAX FORMS

The CAL instruction repertoire has been expanded for the convenience of programmers to allow for special forms of symbolic instructions. Because of this expansion, certain Cray machine instructions can be generated from two or more different CAL instructions. For example, both of the following instructions

```
VL  Ak   (where k=0)
VL  1
```

generate an instruction 00200, which causes a 1 to be entered into the VL register. The first instruction is the basic form of the Enter VL instruction and takes advantage of the special case where (A*k*)=1 if *k*=0; the second instruction is a special syntax form providing the programmer with a more convenient notation for the special case.

Any of the operations performed by special instructions can be performed using instructions in the basic set. Instructions having a special syntax form are identified as such in the instruction description described later in this section.

In several cases, a single syntax form of an instruction can result in any of several different machine instructions being generated. In these cases, which provide for entering the value of an expression into an A register or into an S register or for shifting S register contents, the assembler determines which instruction to generate from characteristics of the expression. For details, refer to the Entries into A registers and Entries into S registers instructions and to the shift instruction.

REGISTER ENTRY INSTRUCTIONS

Instructions in this category provide for entering values such as constants, expression values, or masks directly into registers.

Entries into A registers

The following syntax and its special form enter a quantity into A*i*. This syntax differs from most CAL symbolic instructions in that the assembler generates any of four Cray machine instructions depending on the form, value, and attributes of the expression.

	Result	Operand	Description	Machine instruction
	A*i*	*exp*	Enter *exp* into A*i*	020*ijkm* or 021*ijkm* or 022*ijk*
†	A*i*	-1	Enter -1 into A*i*	031*i*00

If the form of the expression is explicitly -1, the assembler generates an instruction 031*i*00 to efficiently enter the value -1. This instruction executes in the Address Add functional unit.

† Special syntax format

The assembler generates an instruction 022*ijk* where the *jk* fields contain the 6-bit value of the expression if the following conditions are true:

- The value of the expression is positive and less than 64.
- All symbols are previously defined and neither relocatable nor external.

If either of the conditions is not true, the assembler generates either the 2-parcel instruction 020*ijkm* or 021*ijkm*. If the expression has a positive value or is external, instruction 020*ijkm* is generated with the value entered in the 22-bit *jkm* field. If the expression value is negative, instruction 021*ijkm* is generated with the ones complement of the expression value entered into the 22-bit *jkm* field.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
022310		A3	0'10	
0212 00000010		A2	#0'10	
	AREG	=	2	
0212 00000007		A.AREG	-0'10	
0202 00000130		A2	0'130	
0203 00000021		A3	VAL+1	VAL=20 (octal)
0204 01777777		A4	0'1777777	
0205 00051531		A5	A'SY'R	
0206 00000000		A6	#MINUS1	MINUS1=-1
031300		A3	-1	
		EXT	X	
0204 17777777		A4	X-1	020*ijkm* used if expression is external

Entries into S registers

Several instructions can be used to enter a quantity into S registers. The following syntax enters a quantity into S*i*. Either the 2-parcel 040*ijkm* instruction or the 2-parcel 041*ijkm* instruction is generated depending on the value of the expression.

Result	Operand	Description	Machine instruction
S*i*	*exp*	Enter *exp* into S*i*	040*ijkm* or 041*ijkm*

If the expression has a positive value or is external, instruction 040*ijkm* is generated with the 22-bit *jkm* field containing the expression value. If the expression has a negative value, instruction 041*ijkm* is generated with the 22-bit *jkm* field containing the ones complement of the expression value.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
0402 00000130			S2	0'130
	SREG	=	3	
0403 00000021		S.SREG	VAL+1	VAL=20 (octal)
0404 01777777		S4	0'1777777	
0405 00051531		S5	A'SY'R	
0406 00000000		S6	#MINUS1	MINUS1=-1
0412 00000000		S2	-1	
0413 00000002		S3	#2	
0414 01777776		S4	-0'1777777	
0404 00000003		S4	#VAL2	VAL2=3
		EXT	X	
0401 17777777		S1	X-1	040*ijkm* used if expression is external

The following syntax forms are initially recognized by the assembler as the symbolic instruction S*i* *exp*. The assembler then checks the expression to see if it has any of the following forms. If it finds one of the forms in the exact syntax shown, it generates the corresponding Cray machine instruction. If none of these forms is found, instruction 040*ijkm* or 041*ijkm* is generated as previously described. These special forms allow more efficient instructions for entering often used values into S1.

	Result	Operand	Description	Machine instruction
†	S*i*	0	Clear S*i*	043*i*00
†	S*i*	1	Enter 1 into S*i*	042*i*77
†	S*i*	-1	Enter -1 into Si	042*i*00
	S*i*	1.	Enter 1 into S*i* as normalized floating-point constant	071*i*50
	S*i*	2.	Enter 2 into S*i* as normalized floating-point constant	071*i*60
	S*i*	4.	Enter 4 into S*i* as normalized floating-point constant	071*i*70
	S*i*	0.4	Enter 0.5 into S*i* as normalized floating-point constant	071*i*40
	S*i*	0.6	Enter 0.75*(2**48) into S*i* as normalized floating-point constant	071*i*30

The syntax form S*i* 0.6 is useful for extracting the integer part of a floating-point quantity (that is, fix) as illustrated in the examples.

Instructions 043*i*00, 042*i*77, and 042*i*00 execute in the Scalar Logical functional unit.

† Special syntax form

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
	FIX	=	6	
071630		S.FIX	0.6	
071240		S2	0.4	
071350		S3	1.	
071460		S4	2.	
071570		S5	4.	
043600		S6	0	Clear S6
042677		S6	1	Set S6 to 1
	*	Fix a floating-point number in S1		
	*	Separate integer and fractional parts		
071230		S2	0.6	
062312		S3	S1+FS2	
023130		A1	S3	Integer part
063332		S3	S3-FS2	Floating-point integer part
063113		S1	S1-FS3	Fractional part

The following syntax and its special form generate a mask of ones from the right. The assembler evaluates the expression to determine the mask length. All symbols in the expression must be previously defined.

	Result	Operand	Description	Machine instruction
	S*i*	<*exp*	Form ones mask in S*i* from right	042*ijk*
†	S*i*	#><*exp*		

In the first form, the mask length is the value of the expression. In the second form, the mask length is 64 minus the expression value. The mask length must be a positive integer not exceeding 64; 64 minus the mask length is inserted into the *jk* fields of the instruction. If the value of the expression is 0 for the first form or 64 for the second form, the assembler generates instruction 043*i*00.

Instruction 042*ijk* executes in the Scalar Logical functional unit.

† Special syntax form

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
042273		S2	<5	
042273		S2	#>0'73	
042366		S3	<D'10	
042400		S4	<0'100	
043500		S5	<0	

The following syntax and its special form generate a mask of ones from the left. The assembler evaluates the expression to determine the mask length. All symbols in the expression must be previously defined.

	Result	Operand	Description	Machine instruction
	S*i*	>*exp*	Form ones mask in S*i* from left	043*ijk*
†	S*i*	#<*exp*		

In the first form, the mask length is the value of the expression. In the second form, the mask length is 64 minus the expression value. The mask length must be a positive integer not exceeding 64 and is inserted into the *jk* fields of the instruction. If the expression value is 64 for the first form or 0 for the second form, the assembler generates instruction 042*i*00.

Instruction 043*ijk* executes in the Scalar Logical functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
043205		S2	>5	
043205		S2	#<0'73	
043312		S3	<D'10	
042400		S4	<0'100	
043500		S5	<0	

† Special syntax form

The following syntax can be used to set or clear the sign bit of S*i*. The first syntax sets the sign bit of S*i* and zeros all other bits.

Result	Operand	Description	Machine instruction
† S*i*	SB	Enter sign bit into S*i*	051*i*00
† S*i*	#SB	Enter ones complement of sign bit in S*i*	047*i*00

The second syntax clears the sign bit and sets all other bits.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
	I	=	1	
051100		S.I	SB	
047200		S2	#SB	

Entries into V registers

Two instructions can be used to enter a quantity into V registers.

The following syntax zeros element (A*k*) of register V*i*. The low-order 6 bits of A*k* determine which element is zeroed. The second element of register V*i* is zeroed (that is, element 1) if the *k* designator is 0.

Result	Operand	Description	Machine instruction
† V*i*,A*k*	0	Clear element (A*k*) of register V*i*	077*i*0*k*

† Special syntax form

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
077602		V6,A2	0	

The following syntax zeros elements of V*i*. The number of elements zeroed is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
† V*i*	0	Clear V*i*	145*iii*

The 145*iii* instruction executes in the Vector Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
140500		V5	0	

Entries into Semaphore register††

Three instructions can be used to test and set, clear, or set a semaphore.

The following syntax tests and sets the semaphore designated by *jk*. If the semaphore is set, issue is held until another CPU clears that semaphore. If the semaphore is clear, instruction 0034*jk* issues and sets the semaphore.

Result	Operand	Description	Machine instruction
SM*jk*	1,TS	Test and set semaphore *jk*, $0 \leq jk \leq 31_{10}$	0034*jk*

† Special syntax form
†† CRAY X-MP Computer Systems only

If all CPUs in a cluster are holding issue on a test and set, the DL flag is set in the Exchange Package (if not in monitor mode) and an exchange occurs. If an interrupt occurs while a test and set instruction is holding in the CIP register, the WS flag in the Exchange Package sets, CIP and NIP registers clear, and an exchange occurs with the P register pointing to the test and set instruction.

The SM register is 32 bits with SM0 being the most significant bit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
003407		SM7	1,TS	

The following syntax clears the semaphore designated by *jk*.

Result	Operand	Description	Machine instruction
SM*jk*	0	Clear semaphore *jk*, $0 \leq jk \leq 31_{10}$	0036*jk*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
003607		SM7	0	

The following syntax sets the semaphore designated by *jk*.

Result	Operand	Description	Machine instruction
SM*jk*	1	Set semaphore *jk*, $0 \leq jk \leq 31_{10}$	0037*jk*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
003707		SM7	1	

INTER-REGISTER TRANSFER INSTRUCTIONS

Instructions in this group provide for transferring the contents of one register to another register. In some cases, the register contents can be complemented, converted to floating-point format, or sign extended as a function of the transfer.

Transfers to A registers

The machine instructions and related CAL syntax for transferring the contents from one register to A registers are described below.

The following syntax enters the contents of register A*k* into register A*i*. The value 1 is entered if the *k* designator is 0.

Result	Operand	Description	Machine instruction
† A*i*	A*k*	Transmit (A*k*) to A*i*	030*i*0*k*

Instruction 030*i*0*k* executes in the Address Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
030602		A6	A2	

† Special syntax form

Result	Operand	Description	Machine instruction
† A*i*	-A*k*	Transmit negative of (A*k*) to A*i*	031*i*0*k*

The following syntax enters the negative (twos complement) of the contents of register A*k* into register A*i*. The value -1 is entered into A*i* if the *k* designator is 0.

Instruction 031*i*0*k* executes in the Address Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
031703		A7	-A3	

The following syntax transmits the low-order 24 bits of the contents of register S*j* to register A*i*. A*i* is zeroed if the *j* designator is 0.

Result	Operand	Description	Machine instruction
A*i*	S*j*	Transmit (S*j*) to A*i*	023*ij*0

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
023420		A4	S2	

† Special syntax form

The following syntax enters the contents of the VL register into A*i*.

Result	Operand	Description	Machine instruction
† A*i*	VL	Transmit (VL) to A*i*	023*i*01

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
023201		A2	VL	

The following syntax transfers the contents of register B*jk* to A*i*.

Result	Operand	Description	Machine instruction
A*i*	B*jk*	Transmit (B*jk*) to A*i*	024*ijk*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
024517		A5	B17	
	SVNTN	=	0'17	
024517		A5	B.SVNTN	

† CRAY X-MP Computer Systems only

The following syntax transfers the contents of the SBj register shared between the CPUs to Ai.

Result	Operand	Description	Machine instruction
† Ai	SBj	Transfer (SBj) to Ai	026ij7

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
026007		A0	SB0	
026017		A0	SB1	

The following syntax enters the channel number of the highest priority interrupt request into Ai.

Result	Operand	Description	Machine instruction
Ai	CI	Channel number to Ai	033i00

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
033100		A1	CI	

† CRAY X-MP Computer Systems only

The following syntax enters the contents of the Current Address (CA) register for the channel specified by the contents of A*j* into register A*i*.

Result	Operand	Description	Machine instruction
A*i*	CA,A*j*	Address of channel (A*j*) to A*i*	033*ij*0

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
033230		A2	CA,A3	

The following syntax enters the error flag for the channel specified by the contents of A*j* into the low-order 7 bits of A*i*. The high-order bits of A*i* are cleared. The error flag can be cleared only in monitor mode using the CI,A*j* instruction.

Result	Operand	Description	Machine instruction
A*i*	CE,A*j*	Error flag of channel (A*j*) to A*i*	033*ij*1

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
033341		A3	CE,A4	

Transfers to S registers

The machine instructions and related CAL syntax for transferring the contents from one register to S registers are described below.

The following syntax enters the contents of register S*k* into register S*i*. The sign bit is entered into S*i* if the *k* designator is 0.

Result	Operand	Description	Machine instruction
† S*i*	S*k*	Transmit (S*k*) to S*i*	051*i*0*k*

Instruction 051*i*0*k* executes in the Scalar Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
051701		S7	S1	

The following syntax enters the negative (twos complement) of the contents of S*k* into S*i*. The sign bit is entered if the *k* designator is 0.

Result	Operand	Description	Machine instruction
† S*i*	-S*k*	Transmit negative of (S*k*) to S*i*	061*i*0*k*

Instruction 061*i*0*k* uses the Scalar Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
061506		S5	-S6	

† Special syntax form

The following syntax forms the ones complement of the contents of register S*k* and enters the value into S*i*. The complement of the sign bit is entered into S*i* if the *k* designator is 0.

Result	Operand	Description	Machine instruction
† S*i*	#S*k*	Transmit ones complement of (S*k*) to S*i*	047*i*0*k*

Instruction 047*i*0*k* uses the Scalar Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
047203		S2	#S3	

The following syntax transfers the 24-bit value in register A*k* into the low-order 24 bits of register S*i*. The value is treated as an unsigned integer. The high-order bits of S*i* are zeroed. A value of 1 is entered into S*i* when the *k* designator is 0.

Result	Operand	Description	Machine instruction
S*i*	A*k*	Transmit (A*k*) to S*i* without sign extension	071*i*0*k*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
071707		S7	A7	

† Special syntax form

The following syntax transfers the 24-bit value in register A*k* into the low-order 24 bits of register S*i*. The value is treated as a signed integer and the sign bit of the contents of register A*k* is extended to the high-order bits of S*i*. A value of 1 is entered into S*i* when the *k* designator is 0.

Result	Operand	Description	Machine instruction
S*i*	+A*k*	Transmit (A*k*) to S*i* with sign extension	071*i*1*k*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
071717		S7	+A7	

The following syntax transmits the contents of register A*k* to S*i* as an unnormalized floating-point value. The result can then be added to 0 to normalize. When the *k* designator is 0, an unnormalized floating-point 1 is entered into S*i*.

Result	Operand	Description	Machine instruction
S*i*	+FA*k*	Transmit (A*k*) to S*i* as an unnormalized floating-point value	071*i*2*k*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
071324		S3	+FA4	

The following syntax enters the contents of register T*jk* into register S*i*.

Result	Operand	Description	Machine instruction
S*i*	T*jk*	Transmit (T*jk*) to S*i*	074*ijk*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
074306		S3	T6	
074566		S5	T66	
074541		S.ARG	T.TEMP	ARG=5, TEMP=41 (octal)

The following syntax enters the contents of register ST*j* into register S*i*.

Result	Operand	Description	Machine instruction
† S*i*	ST*j*	Read (ST*j*) register to S*i*	072*ij*3

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
072003		S0	ST0	
072013		S0	ST1	

† CRAY X-MP Computer Systems only

The following syntax enters the contents of the element of V*j* indicated by the contents of the low-order 6 bits of A*k* into S*i*. The second element, that is, element 1, is selected if the *k* designator is 0.

Result	Operand	Description	Machine instruction
S*i*	V*j*,A*k*	Transmit (V*j*, element (A*k*)) to S*i*	076*ijk*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
076456		S4	V5,A6	
	I	=	4	
	J	=	5	
	K	=	6	
076456		S.I	V.J,A.K	

The following syntax enters the 64-bit contents of the VM register into register S*i*. The VM register is normally read after having been set by instruction 1750*jk*.

Result	Operand	Description	Machine instruction
S*i*	VM	Transmit (VM) to S*i*	073*i*00

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
073200		S2	VM	

The following syntax enters the 64-bit contents of the real-time clock into register S*i*. The clock is incremented by one each clock period. The real-time clock can be reset only when in monitor mode using instruction 072*i*00.

Result	Operand	Description	Machine instruction
S*i*	RT	Transmit (RTC) to S*i*	072*i*00

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
072700		S7	RT	

The following syntax enters the values of all of the semaphores into S*i*. The 32-bit SM register is left justified in S*i* with SM00 occupying the sign bit.

Result	Operand	Description	Machine instruction
† S*i*	SM	Read semaphore to S*i*	072*i*02

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
072002		S0	SM	
072602		S6	SM	

† CRAY X-MP Computer Systems only

The following syntax enters the contents of the Status register into S*i*.

Result	Operand	Description	Machine instruction
† S*i*	SR*j*	Transmit (SR*j*) to S*i*; *j*=0	073*ij*1

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
073001		S0	SR0	
073301		S3	SR0	

Transfers to intermediate registers

The machine instructions and related CAL syntax for transferring the contents from one register to intermediate registers are described below.

The following syntax enters the contents of register A*i* into register B*jk*.

Result	Operand	Description	Machine instruction
B*jk*	A*i*	Transmit (A*i*) to B*jk*	025*ijk*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
025634		B34	A6	
025634		B.THRTY4	A6	THRTY4=34 (octal)

† CRAY X-MP Computer Systems only

The following syntax transfers the contents of register A*i* into register SB*j*, which is shared between the CPUs in the same cluster.

Result	Operand	Description	Machine instruction
† SB*j*	A*i*	Transfer (A*i*) to SB*j*	027*ij*7

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
027007		SB0	A0	
027107		SB0	A1	

The following syntax enters the contents of register S*i* into register T*jk*.

Result	Operand	Description	Machine instruction
T*jk*	S*i*	Transmit (S*i*) to T*jk*	075*ijk*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
075306		T6	S3	
075566		T66	S5	
075541		T.TEMP	S5	TEMP=41 (octal)

The following syntax transfers the contents of register S*i* into register ST*j*, which is shared between the CPUs in the same cluster.

† CRAY X-MP Computer Systems only

Result	Operand	Description	Machine instruction
† ST*j*	S*i*	Transfer (S*i*) to ST*j*	073*ij*3

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
073003		ST0	S0	
073103		ST0	S1	

Transfers to V registers

The machine instructions and related CAL syntax for transferring the contents from one register to V registers are described below.

The following syntax transmits the contents of the elements of register V*k* to the elements of register V*i*. The number of elements involved is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
†† V*i*	V*k*	Transmit (V*k*) to V*i*	142*i*0*k*

Instruction 142*i*0*k* executes in the Vector Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
142102		V1	V2	

† CRAY X-MP Computer Systems only
†† Special syntax form

The following syntax transmits the twos complement of the contents of elements of register V*k* to the elements of register V*i*. The number of elements involved is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
† V*i*	-V*k*	Transmit twos complement of (V*k*) to V*i*	156*i*0*k*

Instruction 156*i*0*k* executes in the Vector Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
156102		V1	-V2	

The following syntax transmits the contents of register S*j* to an element of V*i* as determined by the low-order 6 bits of the contents of A*k*. Element 1, the second element of V*i*, is selected if the *k* designator is 0.

Result	Operand	Description	Machine instruction
V*i*,A*k*	S*j*	Transmit (S*j*) to V*i* element (A*k*)	077*ijk*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
077167		V1,A7	S6	

† Special syntax form

Transfer to Vector Mask register

The following syntax and its special form transmit the contents of register S*j* to the VM register. The VM register is zeroed if the *j* designator is 0; the special form accommodates this case.

	Result	Operand	Description	Machine instruction
	VM	S*j*	Transmit (S*j*) to VM	0030*j*0
†	VM	0	Clear VM	003000

This instruction may be used in conjunction with the vector merge instructions where an operation is performed depending on the contents of the VM register.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
003040		VM	S4	
003000		VM	0	Clear VM

Transfer to Vector Length register

The following syntax and its special form enters the low-order 7 bits of the contents of register A*k* into the VL register.

	Result	Operand	Description	Machine instruction
	VL	A*k*	Transmit (A*k*) to VL	00200*k*
†	VL	1	Enter 1 into VL	002000

The contents of the VL register determines the number of operations performed by a vector instruction. Since a vector register has 64 elements, from 1 to 64 operations can be performed. The number of operations is (VL) modulo 64. A special case exists such that when (VL) modulo 64 is 0, then the number of operations performed is 64.

† Special syntax form

In this publication, a reference to register V*i* implies operations involving the first *n* elements where *n* is the vector length unless a single element is explicitly noted as in the instructions S*i* V*j*,A*k* and V*i*,A*k* S*j*.

Vector operations controlled by the contents of VL begin with element 0 of the vector registers.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
002003		VL	A3	

If (A3)=6, then (VL)=6 following instruction execution and subsequent vector instructions operate on elements 0 through 5 of vector registers.

Code generated	Location	Result	Operand	Comment
	1	10	20	35
002000		VL	1	

Since the *k* designator is 0, (VL)=1 and vector instructions operate on only one element, element 0.

Code generated	Location	Result	Operand	Comment
	1	10	20	35
002005		VL	A5	

If (A5)=0, then (VL)=64 and vector instructions operate on all 64 elements of the vectors.

Transfers to Semaphore register

The following syntax sets the semaphores from 32 high-order bits of S*i*. SM00 receives the sign bit of S*i*.

Result	Operand	Description	Machine instruction
† SM	S*i*	Load semaphores from S*i*	073*i*02

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
073002		SM	S0	
073102		SM	S1	
073502		SM	S5	

MEMORY TRANSFERS

This category includes instructions that transfer data between registers and memory, enable and disable concurrent block memory transfers, and assure completion of memory references.

Bidirectional memory transfers†

The following syntax forms disable and enable the bidirectional memory mode. Block reads and writes can operate concurrently in bidirectional memory mode. If the bidirectional memory mode is disabled, only block reads can operate concurrently.

† CRAY X-MP Computer Systems only

Result	Operand	Description	Machine instruction
DBM		Disable bidirectional memory transfers	002500
EBM		Enable bidirectional memory transfers	002600

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
002500		DBM		
002600		EBM		

Memory references†

The following syntax assures completion of all memory references within a particular CPU issuing the instruction. This instruction does not issue until all memory references before this instruction are at the stage of execution where completion occurs in a fixed amount of time. For example, a load of any data that has been stored by the CPU issuing instruction CMR is assured of receiving the updated data if the load is issued after the CMR instruction. Synchronization of memory references between processors can be done by this instruction in conjunction with semaphore instructions.

Result	Operand	Description	Machine instruction
CMR		Complete memory references	002700

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
002700		CMR		

† CRAY X-MP Computer Systems only

Stores

Several instructions store data from registers into memory.

Either of the following syntax forms can be used to store words from B registers directly into memory. A0 contains the address of the first word of memory to receive data. The *jk* designator specifies the first B register to be used in the transfer. Subsequent B register contents are stored in consecutive words of memory.

Result	Operand	Description	Machine instruction
,A0	B*jk*,A*i*	Store (A*i*) words starting at	035*ijk*
† 0,A0	B*jk*,A*i*	B*jk* to memory starting at (A0)	

Processing of B registers is circular. B00 is processed after B77 if the count specified in A*i* is not exhausted after B77 is processed. The low-order 7 bits of the contents of A*i* specify the number of words transmitted. If 128>(A*i*)>64, wraparound occurs.

If (A*i*)=0, no words are transferred. Note also that if *i*=0, (A0) is used for the block length as well as the starting memory address. The CAL assembler issues a warning message in this case.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
035522		,A0	B22,A5	
	BB	=	0'22	
	FWAR	=	5	
035522		0,A0	B.BB,A.FWAR	

Either of the following syntax forms can be used to store words from T registers directly into memory. A0 contains the address of the first word of memory to receive data. The *jk* designator specifies the first T register to be used in the transfer. Subsequent T register contents are stored in consecutive words of memory. Processing of T registers is circular. T00 is processed after T77 if the count specified in A*i* is not exhausted after T77 is processed. The low-order 7 bits of the contents of register A*i* specify the number of words transmitted. If 128>(A*i*)>64, wraparound occurs.

† Special syntax form

Result	Operand	Description	Machine instruction
,A0	T*jk*,A*i*	Store (A*i*) words starting at	037*ijk*
† 0,A0	T*jk*,A*i*	T*jk* to memory starting at (A0)	

If (A*i*)=0, no words are transferred. Note also that if *i*=0, (A0) is used for the block length as well as the starting memory address. CAL issues a warning message in this case.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
37522		,A0	T22,A5	
	TT	=	O'22	
	FWAR	=	5	
037522		0,A0	T.TT,A.FWAR	

The following syntax forms store 24 bits from register A*i* directly into memory. The high-order bits of the memory word are zeroed. The memory address is determined by adding the address in register A*h* to the expression value.

Result	Operand	Description	Machine instruction
exp,A*h*	A*i*	Store (A*i*) to (A*h*) + *exp*	11*hijkm*
† *exp*,0	A*i*	Store (A*i*) to *exp*	110*ijkm*
† *exp*,	A*i*	Store (A*i*) to *exp*	110*ijkm*
† ,A*h*	A*i*	Store (A*i*) to (A*h*)	11*hi*000

Only the value of the expression is used if the *h* designator is 0 or a zero or blank field is used in place of A*h*. Only the contents of A*h* is used if the expression is omitted. An expression, if present, must not have a parcel-address attribute or an assembly error occurs.

† Special syntax form

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
1101 00004520		CON1,A0	A1	
1102 00004520		CON1,0	A2	
1113 00004521		CON1+1,A1	A3	
1124 17777777		-1,A2	A4	
1105 00003000		ADDR,	A5	
1146 00004647		CON,A4	A6	
1146 00000000		,A4	A6	
1161 00000001		1,A6	A1	
1172 00000177		O'177,A7	A2	

The following syntax forms store the contents of register S*i* directly into memory. The memory address is determined by adding the address in register A*h* to the expression value.

	Result	Operand	Description	Machine instruction
	exp,A*h*	S*i*	Store (S*i*) to (A*h*) + *exp*	13*hijkm*
†	*exp*,0	S*i*	Store (S*i*) to *exp*	130*ijkm*
†	*exp*,	S*i*	Store (S*i*) to *exp*	130*ijkm*
†	,A*h*	S*i*	Store (S*i*) to (A*h*)	13*hi*000

Only the value of the expression is used if the *h* designator is 0 or if a zero or blank field is used in place of A*h*. Only the contents of A*h* is used if the expression is omitted. An expression, if present, must not have a parcel-address attribute or an assembly error occurs.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
1301 00004520		CON1,A0	S1	
1302 00004520		CON1,0	S2	
1346 00000000		,A4	S6	
1324 17777777		-1,A2	S4	
1305 00003000		ADDR,	S5	

† Special syntax form

The following syntax and its special form store words from elements of register V*j* directly into memory. A0 contains the starting memory address. This address is incremented by the contents of register A*k* for each word transmitted. The contents of A*k* can be positive or negative allowing both forward and backward streams of references. If the *k* designator is 0 or if 1 replaces A*k* in the result field of the instruction, the address is incremented by 1.

Result	Operand	Description	Machine instruction
,A0,A*k*	V*j*	Store (V*j*) to memory starting at (A0) incremented by (A*k*)	1770*jk*
† ,A0,1	V*j*	Store (V*j*) to a memory in consecutive addresses starting with (A0)	1770*j*0

The number of elements transferred is determined by the contents of the VL register.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
177032		,A0,A2	V3	
177030		,A0,1	V3	

Loads

Several instructions can be used to load data from memory into registers.

Either of the following syntax forms can be used to transfer words from memory directly into B registers. A0 contains the address of the first word of memory to be transferred. The *jk* designator specifies the first B register to be used in the transfer. The low-order 24 bits of consecutive words of memory are loaded into consecutive B registers.

† Special syntax form

Result	Operand	Description	Machine instruction
B*jk*,A*i*	,A0	Read (A*i*) words starting at B*jk* from memory starting at (A0)	034*ijk*
† B*jk*,A*i*	0,A0		

Processing of B registers is circular. B00 is loaded after B77 if the count specified in A*i* is not exhausted after B77 is loaded. The low-order 7 bits of the contents of A*i* specify the number of words transmitted. If 128>(A*i*)>64, wraparound occurs.

If (A*i*)=0, no words are transferred. Note also that if *i*=0, (A0) is used for the block length as well as the starting memory address. The CAL assembler issues a warning message in this case.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
034407		B7,A4	,A0	
	BB	=	O'22	
	FWAR	=	5	
034522		B.BB,A.FWAR	0,A0	

Either of the following syntax forms can be used to transfer words from memory directly into T registers. A0 contains the address of the first word of memory to be transferred. The *jk* designator specifies the first T register to be used in the transfer. The loading of T registers is circular. T00 is loaded after T77 if the count specified in A*i* is not exhausted after T77 is loaded. The low-order 7 bits of the contents of A*i* specify the number of words transmitted. If 128>(A*i*)>64, wraparound occurs.

† Special syntax form

	Result	Operand	Description	Machine instruction
	T*jk*,A*i*	,A0	Read (A*i*) words starting at	036*ijk*
†	T*jk*,A*i*	0,A0	T*jk* from memory starting at (A0)	

If (A*i*)=0, no words are transferred. If *i*=0, (A0) is used for the block length and for the starting memory address. The CAL assembler issues a warning message in this case.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
036407		T7,A4	,A0	
	TT	=	O'22	
	FWAR	=	5	
036522		T.TT,A.FWAR	O,A0	

The following syntax forms load the low-order 24 bits of a memory word directly into an A register. The memory address is determined by adding the address in the register A*h* to the expression value. Only the value of the expression is used if the *h* designator is 0 or a zero or blank field is used in place of A*h*. Only the contents of A*h* is used if the expression is omitted. An expression, if present, must not have a parcel-address attribute or an assembly error occurs.

	Result	Operand	Description	Machine instruction
	A*i*	*exp*,A*h*	Read from ((Ah) + *exp*) to A*i*	10*hijkm*
†	A*i*	*exp*,0	Read from (*exp*) to A*i*	100*ijkm*
†	A*i*	*exp*,	Read from (*exp*) to A*i*	100*ijkm*
†	A*i*	,A*h*	Read from (A*h*) to A*i*	10*hi*000

† Special syntax form

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
1001 00004520		A1	CON1,A0	
1002 00004520		A2	CON1,0	
1013 00004521		A3	CON1+1,A1	
1024 17777777		A4	-1,A2	
1005 00003000		A5	ADDR,	
1046 00004647		A6	CON,A4	
1046 00000000		A6	,A4	
1061 00000001		A1	1,A6	
1072 00000177		A2	O'177,A7	

The following syntax forms load the contents of a memory word directly into an S register. The memory address is determined by adding the address in register A*h* to the expression value. Only the value of the expression is used if the *h* designator is 0 or a zero or blank field is used in place of A*h*. Only the contents of A*h* is used if the expression is omitted. An expression, if present, must not have a parcel-address attribute or an assembly error occurs.

Result	Operand	Description	Machine instruction
S*i*	*exp*,A*h*	Read from ((A*i*) + *exp*) to S*i*	12*hijkm*
† S*i*	*exp*,0	Read from (*exp*) to S*i*	120*ijkm*
† S*i*	*exp*,	Read from (*exp*) to S*i*	120*ijkm*
† S*i*	,A*h*	Read from (A*h*) to S*i*	12*hi*000

† Special syntax form

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
1201 00004520		S1	CON1,A0	
1202 00004520		S2	CON1,0	
1213 00004521		S3	CON1+1,A1	
1224 17777777		S4	-1,A2	
1205 00003000		S5	ADDR,	
1246 00004647		S6	CON,A4	
1246 00000000		S6	,A4	
1261 00000001		S1	1,A6	
1272 00000177		S2	O'177,A7	

The following syntax and its special form load words into elements of register V*i* directly from memory. A0 contains the starting memory address. This address is incremented by the contents of register A*k* for each word transmitted. The contents of A*k* can be positive or negative allowing both forward and backward streams of references. If the *k* designator is 0 or if 1 replaces A*k* in the operand field of the instruction, the address is incremented by 1.

Result	Operand	Description	Machine instruction
V*i*	,A0,A*k*	Read from memory starting at (A0) incremented by (A*k*) and load into V*i*	176*i*0*k*
† V*i*	,A0,1	Read from consecutive memory addresses starting with (A0) and load into V*i*	176*i*00

The number of elements transferred is determined by the contents of the VL register.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
176201		V2	,A0,A1	
176500		V5	,A0,1	

† Special syntax form

INTEGER ARITHMETIC OPERATIONS

Integer arithmetic operations obtain operands from registers and return results to registers. No direct memory references are allowed.

The assembler recognizes several special syntax forms for incrementing or decrementing register contents, such as the operands A*i*+1 and A*i*-1; however, these references actually result in register references such that the 1 becomes a reference to A*k* with *k*=0.

All integer arithmetic, whether 24-bit or 64-bit, is twos complement and is so represented in the registers as illustrated in figure 3-6 (the zero bit is the sign). The Address Add functional unit and Address Multiply functional unit perform 24-bit arithmetic. The Scalar Add functional unit and the Vector Add functional unit perform 64-bit arithmetic.

No overflow is detected by Integer Functional units.

Multiplication of two fractional operands can be accomplished using the floating-point multiply instruction. The Floating-point Multiply functional unit recognizes the conditions where both operands have zero exponents as a special case and returns the high-order 48 bits of the result as an unnormalized fraction. Division of integers would require that they first be converted to floating-point format and then divided using the floating-point units.

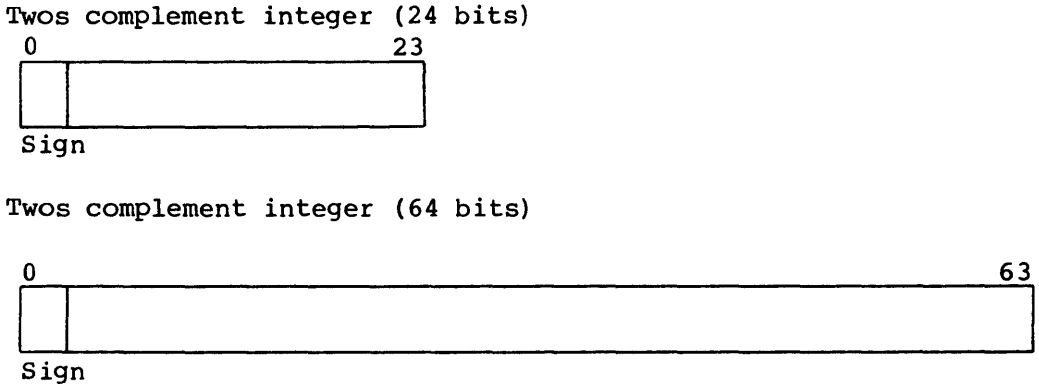


Figure 3-6. Integer data formats

24-bit integer arithmetic

The machine instructions and related CAL syntax for performing 24-bit integer arithmetic operations are described in the following paragraphs.

The following syntax and its special form add the contents of register A*j* to the contents of register A*k* and enter the result into register A*i*. A*k* is transmitted to A*i* when the *j* designator is 0 and the

k designator is nonzero. One is transmitted to A*i* when the *j* and *k* designators are both 0. (A*j*)+1 is transmitted to A*i* when the *j* designator is nonzero and the *k* designator is 0. The assembler allows an alternate form of the instruction when the *k* designator is 0.

	Result	Operand	Description	Machine instruction
	A*i*	A*j*+A*k*	Integer sum of (A*j*) and (A*k*) to A*i*	030*ijk*
†	A*i*	A*j*+1	Integer sum of (A*j*) and 1 to A*i*	030*ij*0

Instruction 030*ijk* executes in the Address Integer Add functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
030123		A1	A2+A3	
030102		A1	A2	
030230		A2	A3+1	

The following syntax and its special form subtract the contents of register A*k* from the contents of register A*j* and enter the result into register A*i*. The negative of A*k* is transmitted to A*i* when the *j* designator is 0 and the *k* designator is nonzero. A -1 is transmitted to A*i* when the *j* and *k* designators are both 0. (A*j*)-1 is transmitted to A*i* when the *j* designator is nonzero and the *k* designator is 0.

	Result	Operand	Description	Machine instruction
	A*i*	A*j*-A*k*	Integer difference of (A*j*) less (A*k*) to A*i*	031*ijk*
†	A*i*	A*j*-1	Integer difference of (A*j*) less 1 to A*i*	031*ij*0

† Special syntax form

The special form represents the case where (A*k*)=1 if *k*=0.

Instruction 031*ijk* executes in the Address Integer Add functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
031456		A4	A5-A6	
031102		A1	-A2	
031450		A4	A5-A1	

The following syntax forms the integer product of the contents of register A*j* and register A*k* and enters the low-order 24 bits of the result into A*i*. A*i* is cleared when the *j* designator is 0. A*j* is transmitted to A*i* when the *k* designator is 0 and the *j* designator is nonzero.

Result	Operand	Description	Machine instruction
A*i*	A*j**A*k*	Integer product of (A*j*) and (A*k*) to A*i*	032*ijk*

Instruction 032*ijk* executes in the Address Integer Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
032712		A7	A1*A2	

64-bit integer arithmetic

The machine instructions and related CAL syntax for performing 64-bit integer arithmetic operations are described in the following paragraphs.

The following syntax adds the contents of register S*k* to the contents of register S*j* and enters the result into S*i*. S*k* is transmitted to S*i* if the *j* designator is 0 and the *k* designator is nonzero. The high-order bit of S*i* is set and all other bits of S*i* are cleared if the *j* and *k* designators are both 0.

Result	Operand	Description	Machine instruction
S*i*	S*j*+S*k*	Integer sum of (S*j*) and (S*k*) to S*i*	060*ijk*

Instruction 060*ijk* executes in the Scalar Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
060237		S2	S3+S7	
060405		S4	S0+S5	

The following syntax adds the contents of S*j* to each element of V*k* and enters the results into elements of V*i*. Elements of V*k* are transmitted to V*i* if the *j* designator is 0.

Result	Operand	Description	Machine instruction
V*i*	S*j*+V*k*	Integer sums of (S*j*) and (V*k*) to V*i*	154*ijk*

The number of operations performed is determined by the contents of the VL register.

Instruction 154*ijk* executes in the Vector Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
154213		V2	S1+V3	

The following syntax adds the contents of elements of register V*j* to the contents of corresponding elements of register V*k* and enters the results into elements of register V*i*.

Result	Operand	Description	Machine instruction
V*i*	V*j*+V*k*	Integer sums of (V*j*) and (V*k*) to V*i*	155*ijk*

The number of operations performed is determined by the contents of the VL register.

Instruction 155*ijk* executes in the Vector Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
155456		V4	V5+V6	

The following syntax subtracts the contents of register S*k* from the contents of register S*j* and enters the result into S*i*. The high-order bit of S*i* is set and all other bits of S*i* are cleared when the *j* and *k* designators are both 0. The negative (twos complement) of S*k* is transmitted to S*i* if the *j* designator is 0 and the *k* designator is nonzero.

Result	Operand	Description	Machine instruction
S*i*	S*j*-S*k*	Integer difference of (S*j*) less (S*k*) to S*i*	061*ijk*

Instruction 061ijk executes in the Scalar Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
061123		S1	S2-S3	

The following syntax subtracts the contents of each element of Vk from the contents of register Sj and enters the results into elements of register Vi. The negative (twos complement) of each element of Vk is transmitted to Vi if the j designator is 0.

Result	Operand	Description	Machine instruction
Vi	Sj-Vk	Integer differences of (Sj) and (Vk) to Vi	156ijk

The number of operations performed is determined by the contents of the VL register.

Instruction 156ijk executes in the Vector Integer Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
156712		V7	S1-V2	

The following syntax subtracts the contents of elements of register Vk from the contents of corresponding elements of register Vj and enters the results into elements of register Vi.

Result	Operand	Description	Machine instruction
Vi	Vj-Vk	Integer differences of (Vj) less (Vk) to Vi	157ijk

The number of operations performed is determined by the contents of the VL register.

Instruction 157*ijk* executes in the Vector Integer Add functional unit.

Example:

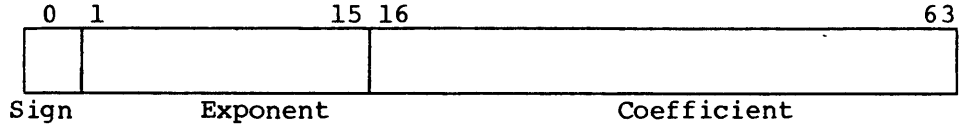
Code generated	Location	Result	Operand	Comment
	1	10	20	35
157345		V3	V4-V5	

FLOATING-POINT ARITHMETIC

All floating-point arithmetic operations use registers as the source of operands and return results to registers.

Floating-point numbers are represented in a standard format throughout the CPU. This format is a packed representation of a binary coefficient and an exponent or power of two. The coefficient is a 48-bit signed fraction. The sign of the coefficient is separated from the rest of the coefficient as shown in figure 3-7. Since the coefficient is signed magnitude, it is not complemented for negative values.

Single-precision, floating-point number



Double-precision, floating-point number

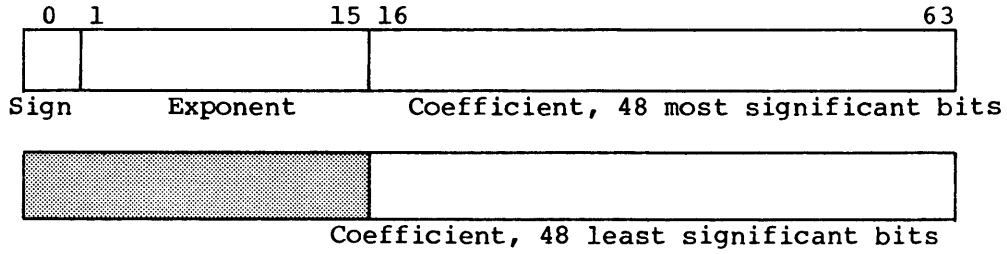


Figure 3-7. Floating-point data formats

The exponent of the floating-point format is represented as a biased integer in bits 1 through 15. The bias that is added to the exponents is 40000 (octal). The positive range of exponents is 40000 (octal) through 57777 (octal). The negative range of exponents is 37777 (octal) through 20000 (octal). Thus, the unbiased range of exponents is the following:

2^{-20000_8} through 2^{+17777_8}

In terms of decimal values, the floating-point format of the Cray computer allows the expression of numbers accurate to about 15 decimal digits in the approximate decimal range of 10^{-2466} through 10^{+2466}.

Double-precision floating-point numbers are represented in two 64-bit words. The format is a software convention, as there is no hardware for double-precision floating-point arithmetic.

The format of the first word as defined and used in the Cray Operating System (COS) is the same as a single-precision floating-point number, with the low-order 48 bits of the second word providing the least significant bits of a 96-bit coefficient. The high-order 16 bits of the second word are normally 0 and are not used.

Normalized floating-point number

A nonzero floating-point number in packed format is normalized if the most significant bit of the coefficient is nonzero. This condition implies that the coefficient has been shifted to the left as far as possible and therefore, the floating-point number has no leading zeros in the coefficient.

When a floating-point number has been created by inserting an exponent of 40060 (octal) into a word containing a 48-bit integer, the result should be normalized before being used in a floating-point operation. Normalization is accomplished by adding the unnormalized floating-point operand to 0. Since S0 provides a 64-bit zero value when used in the S*j* field of an instruction, a normalize of an operand in S*k* can be performed using the S*i* +FS*k* instruction and a normalize of operands in V*k* can be performed using the V*i* +FV*k* instruction.

Floating-point range errors

Overflow of the floating-point range is indicated by an exponent value of 60000 (octal) or greater in packed format. Underflow is indicated by an exponent value of 17777 (octal) or less in packed format. Detection of the overflow condition initiates an interrupt if the Floating-point Mode flag is set in the Mode register and monitor mode is not in effect. Detection of these conditions by the floating-point units is described in detail in the appropiate Cray mainframe reference manual.

Result	Operand	Description	Machine instruction
EFI		Enable floating-point interrupt	0021*xx*
DFI		Disable floating-point interrupt	0022*xx*

The EFI and DFI instructions provide for setting and clearing the interrupt flag in the Mode register. These instructions do not check the previous state of the flag; there is no testing of the flag.

Floating-point addition and subtraction

The machine instructions and related CAL syntax for performing floating-point addition and subtraction are described in the following paragraphs.

The following syntax and its special form produce the floating-point sum of the contents of S*j* and the contents of S*k* registers and enter the result into S*i*. The result is normalized even if the operands are unnormalized. The *k* designator is not normally 0. In the special form, the *j* designator is assumed to be 0 so that the normalized contents of S*k* are entered into S*i*.

	Result	Operand	Description	Machine instruction
	S*i*	S*j*+FS*k*	Floating-point sum of (S*j*) and (S*k*) to S*i*	062*ijk*
†	S*i*	+FS*k*	Normalize (S*k*) to S*i*	062*i*0*k*

Instruction 062*ijk* executes in the Floating-point Add functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
062345		S3	S4+FS5	
062404		S4	+FS4	

† Special syntax form

The following syntax forms the floating-point sums of the contents of S*j* and elements of register V*k* to elements of register V*i*. The results are normalized even if the operands are unnormalized. The number of operations performed depends on the contents of the VL register. The special form of the instruction normalizes the contents of the elements of V*k* and enters the results into elements of V*i*.

	Result	Operand	Description	Machine instruction
	V*i*	S*j*+FV*k*	Floating-point sums of (S*j*) and (V*k*) to V*i*	170*ijk*
†	V*i*	+FV*k*	Normalize (V*k*) to V*i*	170*i*0*k*

Instruction 170*ijk* executes in the Floating-point Add functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
170712		V7	S1+FV2	
170501		V5	+FV1	Normalize (V1) to V5

The following syntax forms the floating-point sums of the contents of elements of V*j* and elements of V*k* and enters the results into the elements of register V*i*. The results are normalized even if the operands are unnormalized. The number of operations performed is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	V*j*+FV*k*	Floating-point sums of (V*j*) and (V*k*) to V*i*	171*ijk*

Instruction 171*ijk* executes in the Floating-point Add functional unit.

† Special syntax form

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
171234		V2	V3+FV4	

The following syntax forms the floating-point difference of the contents of register Sj less the contents of register Sk and enters the normalized result into Si. The result is normalized even if the operands are unnormalized.

	Result	Operand	Description	Machine instruction
	Si	Sj-FSk	Floating-point difference of (Sj) less (Sk) to Si	063ijk
†	Si	-FSk	Transmit the negative of (Sk) as a normalized floating-point value	063i0k

The negative (twos complement) of the floating-point quantity in Sk is transmitted to Si as a normalized floating-point number if the j designator is 0 and the k designator is nonzero. The special form accommodates this special case. The k designator is normally nonzero.

Instructions 063ijk and 063i0k execute in the Floating-point Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
063761		S7	S6-FS1	

The following syntax forms the floating-point differences of the contents of Sj and elements of register Vk and enters the results into register Vi. The results are normalized even if the operands are unnormalized. The negatives (twos complements) of floating-point quantities in elements of Vk are transmitted to Vi if the j designator is 0. The special form accommodates this special case. The number of operations performed is determined by the contents of the VL register.

† Special syntax form

	Result	Operand	Description	Machine instruction
	V*i*	S*j*-FV*k*	Floating-point differences of (S*j*) less (V*k*) to V*i*	172*ijk*
†	V*i*	-FV*k*	Transmit normalized negative of (V*k*) to V*i*	172*i*0*k*

Instruction 172*ijk* and 172*i*0*k* execute in the Floating-point Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
172516		V5	S1-FV6	

The following syntax forms the floating-point differences of the contents of elements of register V*j* less the contents of elements of registers V*k* and enters the results into elements of register V*i*. The results are normalized even if the operands are unnormalized. The number of operations performed is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	V*j*-FV*k*	Floating-point differences of (V*j*) less (V*k*) to V*i*	173*ijk*

Instruction 173*ijk* executes in the Floating-point Add functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
173712		V7	V1-FV2	

† Special syntax form

Floating-point multiplication

The machine instructions and related CAL syntax for performing floating-point multipication are described in the following paragraphs.

The following syntax forms the floating-point product of the contents of S*j* and S*k* and enters the result into S*i*. The result is not normalized if either operand is unnormalized.

Result	Operand	Description	Machine instruction
S*i*	S*j**FS*k*	Floating-point product of (S*j*) and (S*k*) to S*i*	064*ijk*

Instruction 064*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
064234		S2	S3*FS4	

The following syntax forms the floating-point products of the contents of S*j* and elements of V*k* and enters the results into elements of V*i*. The results are not normalized if the operands are unnormalized. The number of operations performed is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	S*j**FV*k*	Floating-point products of (S*j*) and (V*k*) to V*i*	160*ijk*

Instruction 160*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
160727		V6	S2*FV7	

The following syntax forms the floating-point products of the contents of elements of V*j* and elements of V*k* and enters the results into elements of V*i*. The results are not normalized if the operands are unnormalized. The number of operations performed is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	V*j**FV*k*	Floating-point products of (V*j*) and (V*k*) to V*i*	161*ijk*

Instruction 161*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
161123		V1	V2*FV3	

The following syntax forms the half-precision rounded floating-point product of the contents of the S*j* and S*k* registers and enters the result into S*i*. The result is not normalized if the operands are unnormalized. The low-order 18 bits of the result are zeroed. This instruction can be used in a divide algorithm when only 30 bits of accuracy are required.

Result	Operand	Description	Machine instruction
S*i*	S*j**HS*k*	Half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*	065*ijk*

Instruction 065*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
065167		S1	S6*HS7	

The following syntax forms the half-precision rounded floating-point products of the contents of the S*j* register and the contents of elements of the V*k* register and enters the results into elements of V*i*. The results are not normalized if the operands are unnormalized. The low-order 18 bits of the results are zeroed.

Result	Operand	Description	Machine instruction
V*i*	S*j**HV*k*	Half-precision rounded floating-point products of (S*j*) and (V*k*) to V*i*	162*ijk*

The number of operations performed by this instruction is determined by the contents of the VL register. This instruction can be used in a divide algorithm when only 30 bits of accuracy are required.

Instruction 162*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
162456		V4	S5*HV6	

The following syntax forms the half-precision rounded floating-point products of the contents of elements of the V*j* register and elements of the V*k* register and enters the results into elements of V*i*. The results are not normalized if the operands are unnormalized. The low-order 18 bits of the results are zeroed.

Result	Operand	Description	Machine instruction
V*i*	V*j**HV*k*	Half-precision rounded floating-point products of (V*j*) and (V*k*) to V*i*	163*ijk*

The number of operations performed by this instruction is determined by the contents of the VL register. This instruction can be used in a divide algorithm when only 30 bits of accuracy are required.

Instruction 163*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
163712		V7	V1*HV2	

The following syntax forms the rounded floating-point product of the contents of the S*j* register and the contents of the S*k* register and enters the result into S*i*. The result is not normalized if the operands are unnormalized.

Result	Operand	Description	Machine instruction
S*i*	S*j**RS*k*	Rounded floating-point product of (S*j*) and (S*k*) to S*i*	066*ijk*

Instruction 066*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
066147		S1	S4*RS7	

The following syntax forms the rounded floating-point products of the contents of the S*j* register and the contents of elements of V*k* and enters the results into elements of V*i*. The results will not be normalized if the operands are unnormalized. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	S*j**RV*k*	Rounded floating-point products of (S*j*) and (V*k*) to V*i*	164*ijk*

Instruction 164*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
164314		V3	S1*RV4	

The following syntax forms the rounded floating-point products of the contents of elements of V*j* and elements of V*k* and enters the results into elements of V*i*. The results will not be normalized if the operands are unnormalized. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	V*j**RV*k*	Rounded floating-point products of (V*j*) and (V*k*) to V*i*	165*ijk*

Instruction 165*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
165567		V5	V6*RV7	

Reciprocal iteration

The machine instructions and related CAL syntax for performing floating-point reciprocal iteration are described in the following paragraphs.

The following syntax forms 2 minus the floating-point product of the contents of S*j* and S*k* and enters the result into S*i*. The result is not normalized if the operands are unnormalized. The instruction is used in the divide sequence illustrated in the example for the reciprocal approximation instruction S*i* /HS*j*.

Result	Operand	Description	Machine instruction
S*i*	S*j**IS*k*	2-floating-point product of (S*j*) and (S*k*) to S*i*	067*ijk*

Instruction 067*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
67323		S3	S2*IS3	

The following syntax forms 2 minus the floating-point products of the contents of S*j* and the contents of elements of V*k* and enters the results into elements of V*i*. The results are not normalized if the operands are unnormalized. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	S*j**IV*k*	2-floating-point products of (S*j*) and (V*k*) to V*i*	166*ijk*

Instruction 166*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
166123		V1	S2*IV3	

The following syntax forms 2 minus the floating-point products of contents of elements of V*j* and elements of V*k* and enters the results into elements of V*i*. The results are not normalized if the operands are unnormalized. This instruction is used in the divide sequence. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	V*j**IV*k*	2-floating-point products of (V*j*) and (V*k*) to V*i*	167*ijk*

Instruction 167*ijk* executes in the Floating-point Multiply functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
167456		V4	V5*IV6	

Reciprocal approximation

The machine instructions and related CAL syntax for forming an approximation to the reciprocal of a floating-point value are described in the following paragraphs.

The following syntax forms an approximation to the reciprocal of the floating-point value in S*j* and enters the result into S*i*. The result is meaningless if the contents of S*j* is unnormalized or 0. This instruction is used in the divide sequence as illustrated in the following example.

Result	Operand	Description	Machine instruction
S*i*	/HS*j*	Floating-point reciprocal approximation of (S*j*) to S*i*	070*ij*0

Instruction 070*ij*0 executes in the Floating-point Reciprocal functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
	*	Divide S1 by S2; result to S1		
070320		S3	/HS2	Approximate reciprocal
064113		S1	S1*FS3	Approximate result
067223		S2	S2*IS3	Correction factor
064112		S1	S1*FS2	
	*	Divide S1 by S2 with result accurate to 30 bits		
070320		S3	/HS2	
065313		S3	S1*HS3	
	*	Integer divide A1 by A2;		
	*	Result to A3		
071222		S2	+FA2	Denominator
071121		S1	+FA1	Numerator
062202		S2	S0+FS2	Normalize
062101		S1	S0+FS1	
070220		S2	/HS2	Reciprocal approximation to 1/D
065110		S1	S1*HS2	Rounded half-precision multiply
071230		S2	0.6	
062112		S1	S1+FS2	Fix quotient
023310		A3	S1	24-bit signed result to A3

The following syntax forms the approximations to the reciprocals of the floating-point values in elements of V*j* and enters the results into elements of V*i*. The results are meaningless if the contents of elements are unnormalized or 0. This instruction is used in the divide sequence. The number of operations performed by the instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	/HV*j*	Floating-point reciprocal approximation of (V*j*) to V*i*	174*ij*0

Instruction 174*ij*0 executes in the Floating-point Reciprocal functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
	*	Divide elements of V1 by elements of V2;		
	*	Result to V6		
174320		V3	/HV2	
161413		V4	V1*FV3	
167532		V5	V3*IV2	
161645		V6	V4*FV5	
		Divide elements of V1 by elements of V2;		
	*	Results accurate to 30 bits, results to V6		
174320	*	V3	/HV2	
165613		V6	V1*HV3	
	*	Divide S1 by elements of V2;		
	*	Result to V6		
174320		V3	/HV2	
160413		V4	S1*FV3	
167532		V5	V3*IV2	
161645		V6	V4*FV5	

LOGICAL OPERATIONS

The Scalar and Vector Logical functional units perform bit-by-bit manipulation of 64-bit quantities. Operations provide for logical products, logical differences, logical sums, logical equivalence, and merges.

A logical product (& operator) is the AND function:

```
Operand 1  1010
Operand 2  1100
Result     1000
```

A logical difference (\ operator) is the exclusive OR function:

```
Operand 1  1010
Operand 2  1100
Result     0110
```

A logical sum (! operator) is the inclusive OR function:

```
Operand 1  1010
Operand 2  1100
Result     1110
```

A logical equivalence function:

```
Operand 1  1010
Operand 2  1100
Result     1001
```

A logical merge combines two operands depending on a ones mask in a third operand. The result is defined by (operand 2 & mask)!(operand 1 & #mask) as in the following example:

```
Mask       11110000
Operand 1  11001100
Operand 2  10101010
Result     10101100
```

Logical products

The machine instructions and related CAL syntax for forming logical products are described in the following paragraphs.

The following syntax forms the logical product of the contents of S*j* and S*k* and enters the result into S*i*. If the *j* and *k* designators have the same nonzero value, the contents of S*j* is transmitted to S*i*. If the *j* designator is 0, register S*i* is zeroed. If the *j* designator is nonzero and the *k* designator is 0, the sign bit of the contents of S*j* is extracted.

	Result	Operand	Description	Machine instruction
	Si	Sj&Sk	Logical product of (Sj) and (Sk) to Si	044ijk
†	Si	Sj&SB	Sign bit of (Sj) to Si	044ij0
†	Si	SB&Sj	Sign bit of (Sj) to Si; $j \neq 0$	044ij0

The two special forms of the instruction accommodate this case. The two forms perform identical functions; however, j must not be equal to 0 in the second form; if j is equal to 0, an error results.

Instruction 044ijk executes in the Scalar Logical functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
044235		S2	S3&S5	
044655		S6	S5&S5	S5 to S6
044307		S3	S0&S7	Clear S3
044160		S1	S6&SB	Get sign of S6
044160		S1	SB&S6	Get sign of S6

The following syntax forms the logical products of the contents of Sj and the contents of elements of Vk and enters the results into elements of Vi. If the j designator is 0, elements of register Vi are zeroed. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
Vi	Sj&Vk	Logical products of (Sj) and (Vk) to Vi	140ijk

Instruction 140ijk executes in the Vector Logical functional unit.

† Special syntax form

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
140123		V1	S2&V3	

The following syntax forms the logical products of the contents of elements of register V*j* and elements of register V*k* and enters the results into elements of V*i*. If the *j* designator is the same as the *k* designator, the contents of V*j* elements is transmitted to V*i* elements.

The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	V*j*&V*k*	Logical products of (V*j*) and (V*k*) to V*i*	141*ijk*

Instruction 141*ijk* executes in the Vector Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
141257		V2	V5&V7	
141033		V0	V3&V3	

The following syntax forms the logical product of the contents of S*j* and the ones complement of the contents of S*k* and enters the result into S*i*. If the *j* and *k* designators have the same value or if the *j* designator is 0, register S*i* is zeroed. If the *j* designator is nonzero and the *k* designator is 0, the contents of S*j* with the sign bit cleared is transmitted to S*i*.

The special syntax form accommodates this case.

	Result	Operand	Description	Machine instruction
	S*i*	#S*k*&S*j*	Logical product of (S*j*) and #(S*k*) to S*i*	045*ijk*
†	S*i*	#SB&S*j*	(S*j*) with sign bit cleared to S*i*	045*ij*0

Instruction 045*ijk* executes in the Scalar Logical functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
045271		S2	#S1&S7	
045430		S4	#SB&S3	Clear S4
045506		S5	#S6&S0	Clear S5
045670		S6	#SB&S7	Clear sign bit

Logical sums

The machine instructions and related CAL syntax for forming logical sums are described in the following paragraphs.

The following syntax forms the logical sum of the contents of S*j* and the contents of S*k* and enters the result into S*i*. If the *j* and *k* designators have the same nonzero value, the contents of S*j* is transmitted to S*i*. If the *j* designator is 0 and the *k* designator is nonzero, the contents of S*k* is transmitted to S*i*.

	Result	Operand	Description	Machine instruction
	S*i*	S*j*!S*k*	Logical sum of (S*j*) and (S*k*) to S*i*	051*ijk*
†	S*i*	S*j*!SB	Logical sum of (S*j*) and sign bit to S*i*	051*ij*0
†	S*i*	SB!S*j*	Logical sum of sign bit and (S*j*) to S*i*; *j*≠0	051*ij*0

† Special syntax form

If the j designator is nonzero and the k designator is 0, the contents of Sj with the sign bit set to 1 are transmitted to Si. The two special syntax forms provide for this case. If the j and k designators are both 0, a ones mask consisting of only the sign bit is entered into Si.

The two special forms perform an identical function but in the second form, j must not equal 0; if j equals 0, an error results.

Instruction 051ijk executes in the Scalar Logical functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
051472		S4	S7!S2	
051366		S3	S6!S6	
051710		S7	SB!S1	

The following syntax forms the logical sums of the contents of Sj and the contents of elements of Vk and enters the results into elements of Vi. The contents of Vj elements are transmitted to Vi elements if the j designator is 0. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
Vi	Sj!Vk	Logical sums of (Sj) and (Vk) to Vi	142ijk

Instruction 142ijk executes in the Vector Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
142615		V6	S1!V5	

The following syntax forms the logical sums of the contents of elements of V*j* and elements of V*k* and enters the results into elements of V*i*.

Result	Operand	Description	Machine instruction
V*i*	V*j*!V*k*	Logical sums of (V*j*) and (V*k*) to V*i*	143*ijk*

If the *j* and *k* designators are equal, the contents of V*j* elements are transmitted to V*i*. The number of operations performed by this instruction is determined by the contents of the VL register.

Instruction 143*ijk* executes in the Vector Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
143714		V7	V1!V4	

Logical differences

The machine instructions and related CAL syntax for forming logical differences are described in the following paragraphs.

The following syntax forms the logical difference of the contents of S*j* and the contents of S*k* and enters the result into S*i*. If the *j* and *k* designators are the same nonzero value, S*i* is zeroed. If the *j* designator is 0 and the *k* designator is nonzero, the contents of S*k* is transmitted to S*i*. If the *j* designator is nonzero and the *k* designator is 0, the sign bit of the contents of S*j* is complemented and the result is transmitted to S*i*.

The two special syntax forms provide for this case. The two forms perform identical functions; however, in the second form, *j* must not equal 0; if *j* equals 0, an error results.

	Result	Operand	Description	Machine instruction
	S*i*	S*j*\S*k*	Logical difference of (S*j*) and (S*k*) to S*i*	046*ijk*
†	S*i*	S*j*\SB	Enter (S*j*) into S*i* with sign bit toggled	046*ij*0
†	S*i*	SB\S*j*	Enter (S*j*) into S*i* with sign bit toggled; *j*≠0	046*ij*0

Instruction 046*ijk* executes in the Scalar Logical functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
046123		S1	S2\S3	
046455		S4	S5\S5	Clear S4
046506		S5	S0\S6	S6 to S5
046770		S7	S7\SB	Toggle sign bit

The following syntax forms the logical differences of the contents of S*j* and the contents of elements of V*k* and enters the results into elements of V*i*. If the *j* designator is 0, the contents of V*k* elements are entered into V*i* elements. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	S*j*\V*k*	Logical differences of (S*j*) and (V*k*) to V*i*	144*ijk*

Instruction 144*ijk* executes in the Vector Logical functional unit.

† Special syntax form

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
144267		V2	S6\V7	

The following syntax forms the logical differences of the contents of elements of V*j* and elements of V*k* and enters the results into elements of V*i*. If the *j* and *k* designators are equal, the V*i* elements are zeroed. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	V*j*\V*k*	Logical differences of (V*j*) and (V*k*) to V*i*	145*ijk*

Instruction 145*ijk* executes in the Vector Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
145513		V5	V1\V3	

Logical equivalence

The following syntax forms the logical equivalence of the contents of S*j* and the contents of S*k* and enters the result into S*i*. Bits of S*i* are set to 1 when the corresponding bits of the contents of S*j* and the contents of S*k* are both 1 or both 0.

If the *j* and *k* designators have the same nonzero value, the contents of S*i* is set to all ones. If the *j* designator is 0 and the *k* designator is nonzero, the ones complement of the contents of S*k* is transmitted to S*i*. If the *j* designator is nonzero and the *k* designator is 0, all bits other than the sign bit of the contents of S*j* are complemented and the result transmitted to S*i*.

The two special forms of the instruction accommodate this case. The two forms perform identical functions; however, in the second form, *j* must not equal 0; if *j* equals 0, an error results.

	Result	Operand	Description	Machine instruction
	S*i*	#S*j*\S*k*	Logical equivalence of (S*j*) and (S*k*) to S*i*	047*ijk*
†	S*i*	#S*j*\SB	Logical equivalence of (S*j*) and sign bit to S*i*	047*ij*0
†	S*i*	#SB\S*j*	Logical equivalence of sign bit and (S*j*) to S*i*; *j*≠0	047*ij*0

Instruction 047*ijk* executes in the Scalar Logical functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
047345		S3	#S4\S5	
047260		S2	#S6\SB	
047260		S2	#SB\S6	

Vector mask

The following syntax forms create a mask in the VM register. The 64 bits of the VM register correspond to the 64 elements of V*j*. Elements of V*j* are tested for the specified condition and if the condition is true for an element, the corresponding bit is set to 1 in the VM register. If the condition is not true, the bit is zeroed.

Result	Operand	Description	Machine instruction
VM	V*j*,Z	Set VM bits for zero elements of V*j*	1750*j*0
VM	V*j*,N	Set VM bits for nonzero elements of V*j*	1750*j*1
VM	V*j*,P	Set VM bits for positive elements of V*j*	1750*j*2
VM	V*j*,M	Set VM bits for negative elements of V*j*	1750*j*3

† Special syntax form

The number of elements tested is determined by the contents of the VL register; however, the entire VM register is zeroed before elements of V*j* are tested. If the contents of an element is 0, it is considered positive. Element 0 corresponds to bit 0, element 1 to bit 1, etc. from left to right in the register.

These instructions execute in the Vector Logical functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
175050		VM	V5,Z	
175061		VM	V6,N	
175072		VM	V7,P	
175013		VM	V1,M	

Merge

The machine instructions and related CAL syntax for performing merge operations are described in the following paragraphs.

The following syntax merges the contents of S*j* with the contents of S*i* depending on the ones mask in S*k*.

	Result	Operand	Description	Machine instruction
	S*i*	S*j*!S*i*&S*k*	Scalar merge of (S*i*) and (S*j*) to S*i*	050*ijk*
†	S*i*	S*j*!S*i*&SB	Scalar merge of (S*i*) and sign bit of (S*j*) to S*i*	050*ij*0

The result is defined by (S*j*&S*k*)!(S*i*&#S*k*) as in the following example:

(S*k*) = 11110000
(S*i*) = 11001100
(S*j*) = 10101010
(S*i*) = 10101100

† Special syntax form

This instruction is intended for merging portions of 64-bit words into a composite word. Bits of S*i* are cleared when the corresponding bits of S*k* are 1 if the *j* designator is 0 and the *k* designator is nonzero. The sign bit of S*j* replaces the sign bit of S*i* if the *j* designator is nonzero and the *k* designator is 0 as provided for by the special syntax form of the instruction. The sign bit of S*i* is cleared if the *j* and *k* designators are both 0.

Instruction 050*ijk* executes in the Scalar Logical functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
050123		S1	S2!S1&S3	
050760		S7	S6!S7&S0	

The following syntax transmits the contents of S*j* or the contents of element *n* of V*k* to element *n* of V*i* depending on the ones mask in the VM register. The contents of S*j* is transmitted if bit *n* of VM is 1; the contents of element *n* of V*k* is transmitted if bit *n* of VM is 0.

Element *n* of V*i* is zeroed if the *j* designator is 0 and bit *n* of VM is 1. The number of merge operations performed is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	S*j*!V*k*&VM	Vector merge of (S*j*) and (V*k*) to V*i*	146*ijk*

Instruction 146*ijk* executes in the Vector Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
146726		V7	S2!V6&VM	

For the above instruction, assume the following initial register conditions exist:

```
(VL) = 4
(VM) = 0 60000 0000 0000 0000 0000
(S2) = -1
Element 0 of V6 = 1
Element 1 of V6 = 2
Element 2 of V6 = 3
Element 3 of V6 = 4
```

After instruction execution, the first four elements of V7 are modified as follows:

```
Element 0 of V7 = 1
Element 1 of V7 = -1
Element 2 of V7 = -1
Element 3 of V7 = 4
```

The remaining elements of V7 are unaltered.

The following syntax transmits the contents of element *n* of V*j* or element *n* of V*k* to element *n* of V*i* depending on the ones mask in the VM register. The contents of the V*j* element is transmitted if bit *n* of VM is 1; the contents of the V*k* element is transmitted if bit *n* of VM is 0. The number of merge operations performed is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
V*i*	V*j*!V*k*&VM	Vector merge of (V*j*) and (V*k*) to V*i*	147*ijk*

Instruction 147*ijk* executes in the Vector Logical functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
147123		V1	V2!V3&VM	

Assume the following initial register conditions exist for the above instruction:

```
            (VL) = 4
            (VM) = 0 60000 0000 0000 0000 0000
Element 0 of V2 = 1
Element 1 of V2 = 2
Element 2 of V2 = 3
Element 3 of V2 = 4
Element 0 of V3 = -1
Element 1 of V3 = -2
Element 2 of V3 = -3
Element 3 of V3 = -4
```

After instruction execution, the first four elements of V*i* have been modified as follows:

```
Element 0 of V1 = -1
Element 1 of V1 = 2
Element 2 of V1 = 3
Element 3 of V1 = -4
```

The remaining elements of V1 are unaltered.

The following syntax zeros element *n* of register V*i* or transmits the contents of element *n* of V*k* to element *n* of V*i* depending on the ones mask in the VM register. If bit *n* of the VM is 1, element *n* of V*i* is zeroed; if bit *n* is 0, element *n* of V*k* is transmitted. The number of operations performed by this instruction is determined by the contents of the VL register.

Result	Operand	Description	Machine instruction
† V*i*	#VM&V*k*	Vector merge of (V*k*) and zero to V*i*	146*i*0*k*

Instruction 146*i*0*k* executes in the Vector Logical functional unit.

† Special syntax form

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
146607		V6	#VM&V7	

Assume the following initial register conditions for the above instruction:

```
          (VL) = 4
          (VM) = 0 50000 0000 0000 0000 0000
Element 0 of V7 = 1
Element 1 of V7 = 2
Element 2 of V7 = 3
Element 3 of V7 = 4
```

After instruction execution, the first four elements of V6 have been modified as follows:

```
Element 0 of V6 = 1
Element 1 of V6 = 0
Element 2 of V6 = 3
Element 3 of V6 = 0
```

SHIFT INSTRUCTIONS

The Scalar Shift functional unit and Vector Shift functional unit shift 64-bit quantities or 128-bit quantities. A 128-bit quantity is formed by concatenating two 64-bit quantities. The number of bits a value is shifted left or right is determined by the value of an expression for some instructions and by the contents of an A register for other instructions. If the count is specified by an expression, the value of the expression must not exceed 64.

The following syntax shifts the contents of S*i* left by the amount specified by the expression and enters the result into S0. The shift count must be a positive integer value not exceeding 64. If the shift count is 64, an instruction 053*ijk* is generated. The shift is end off with zero fill. The contents of S*i* is not altered.

Result	Operand	Description	Machine instruction
S0	S*i*<*exp*	Shift (S*i*) left *exp* places to S0	052*ijk*

Instruction 052*ijk* executes in the Scalar Shift functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
052305		S0	S3<5	
052724		S0	S7<VAL+4	
053200		S0	S2<D'64	

The following syntax shifts the contents of S*i* right by the amount specified by the expression and enters the result into S0. The shift count must be a positive integer value not exceeding 64. The assembler stores 64 minus the shift count in the *jk* field of the instruction. If the shift count is 0, instruction 052*ijk* is generated. The shift is end off with zero fill. The contents of S*i* is not altered.

Result	Operand	Description	Machine instruction
S0	S*i*>*exp*	Shift (S*i*) right *exp* places to S0	053*ijk*

Instruction 053*ijk* executes in the Scalar Shift functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
053373		S0	S3>5	
053066		S0	S0>D'10	
053754		S0	S7>VAL+4	
052100		S0	S1>0	

The following syntax shifts the contents of S*i* left by the amount specified by the expression and enters the result into S*i*. The shift count must be a positive integer value not exceeding 64. If the shift count is 64, instruction 055*ijk* is generated. The shift is end off with zero fill.

Result	Operand	Description	Machine instruction
S*i*	S*i*<*exp*	Shift (S*i*) left *exp* places to S*i*	054*ijk*

Instruction 054*ijk* executes in the Scalar Shift functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
054703		S7	S7<3	
054622		S6	S6<VAL+2	
055300		S3	S3<D'64	

The following syntax shifts the contents of S*i* right by the amount specified by the expression and enters the result into S*i*. The shift count must be a positive integer value not exceeding 64. The assembler stores 64 minus the shift count in the *jk* field of the instruction. If the shift count is 0, instruction 054*ijk* is generated. The shift is end off with zero fill.

Result	Operand	Description	Machine instruction
S*i*	S*i*>*exp*	Shift (S*i*) right *exp* places to S*i*	055*ijk*

Instruction 055*ijk* executes in the Scalar Shift functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
055775		S7	S7>3	
055656		S6	S6>VAL+2	
054300		S3	S3>0	

The following syntax and its special forms form a 128-bit quantity by concatenating the contents of S*i* and the contents of S*j*, shift the quantity left by an amount specified by the low-order bits of A*k* and enter the high-order bits of the result into S*i*. The shift is end off with zero fill.

Replacing the A*k* reference with 1 is the same as setting the *k* designator to 0; a reference to A0 provides a shift count of 1. Omitting the S*j* reference is the same as setting the *j* designator to 0; the contents of S*i* are concatenated with a word of zeros.

	Result	Operand	Description	Machine instruction
	S*i*	S*i*,S*j*<A*k*	Left shift by (A*k*) of (S*i*) and (S*j*) to S*i*	056*ijk*
†	S*i*	S*i*,S*j*<1	Left shift by 1 of (S*i*) and (S*j*) to S*i*	056*ij*0
†	S*i*	S*i*<A*k*	Left shift by (A*k*) of (S*i*) to S*i*	056*i*0*k*

S*i* is cleared if the shift count exceeds 127. The shift is a left circular shift of the contents of S*i* if the shift count does not exceed 64 and the *i* and *j* designators are equal and nonzero. The instruction produces the same result as the S*i* S*i*<*exp* instruction if the shift count does not exceed 63 and the designator is 0.

Instruction 056*ijk* executes in the Scalar Shift functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
056235		S2	S2,S3<A5	
056340		S3	S3,S4<1	Left 1 place
056604		S6	S6<A4	

† Special syntax form

The following syntax and its special forms produce a 128-bit quantity by concatenating the contents of Sj and the contents of Si, shift the quantity right by an amount specified by the low-order 7 bits of the contents of Ak and enter the low-order bits of the result into Si. The shift is end off with zero fill.

	Result	Operand	Description	Machine instruction
	Si	Sj,Si>Ak	Right shift by (Ak) of (Sj) and (Si) to Si	057ijk
†	Si	Sj,Si>1	Right shift by 1 of (Sj) and (Si) to Si	057ij0
†	Si	Si>Ak	Right shift by (Ak) of (Si) to Si	057i0k

Replacing the Ak reference with 1 is the same as setting the k designator to 0; a reference to A0 provides a shift count of 1. Omitting the Sj reference is the same as setting the j designator to 0; the contents of Si is concatenated with a word of zeros.

Si is cleared if the shift count exceeds 127. The shift is a right circular shift of the contents of Si if the shift count does not exceed 64 and the i and j designators are equal and nonzero. The instruction produces the same result as the Si Si>*exp* instruction if the shift count does not exceed 63 and the j designator is 0.

Instruction 057ijk executes in the Scalar Shift functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
057235		S2	S3,S2>A5	
057604		S6	S6>A4	
057340		S3	S4,S3>1	Right 1 place

The following syntax and its special form shift the contents of the elements of register Vj to the left by the amount specified by the contents of Ak and enter the results into the elements of Vi. The number of elements involved is determined by the contents of the VL register. For each element, the shift is end off with zero fill. Elements of Vi are zeroed if the shift count exceeds 63. Element contents are shifted left 1 place if the k designator is 0; this can be specified through the special form of the instruction.

† Special syntax form

	Result	Operand	Description	Machine instruction
	V*i*	V*j*<A*k*	Shift (V*j*) left (A*k*) places to V*i*	150*ijk*
†	V*i*	V*j*<1	Shift (V*j*) left 1 place to V*i*	150*ij*0

Instruction 150*ijk* executes in the Vector Shift functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
150123		V1	V2<A3	
150450		V4	V5<1	Left 1 place

The following syntax and its special form shift the contents of the elements of register V*j* to the right by the amount specified by the contents of A*k* and enter the results into the elements of V*i*. The number of elements involved is determined by the contents of the VL register. For each element, the shift is end off with zero fill. Elements of V*i* are zeroed if the shift count exceeds 63. Element contents are shifted right one place if the *k* designator is 0; a special form of the instruction accommodates this feature.

	Result	Operand	Description	Machine instruction
	V*i*	V*j*>A*k*	Shift (V*j*) right (A*k*) places to V*i*	151*ijk*
†	V*i*	V*j*>1	Shift (V*j*) right 1 place to V*i*	151*ij*0

Instruction 151*ijk* executes in the Vector Shift functional unit.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
151341		V3	V4>A1	
151450		V4	V5>1	Right 1 place

† Special syntax form

The following syntax and its special form shift 128-bit quantities from elements of V*j* by the amount specified in A*k* and enter the result into elements of V*i*. Element *n* of V*j* is concatenated with element *n*+1 and the 128-bit quantity is shifted left by the amount specified in A*k*. The shift is end off with zero fill. The high-order 64 bits of the results are transmitted to element *n* of V*i*.

Result	Operand	Description	Machine instruction
V*i*	V*j*,V*j*<A*k*	Double shift (V*j*) left (A*k*) places to V*i*	152*ijk*
† V*i*	V*j*,V*j*<1	Double shift (V*j*) left one place to V*i*	152*ij*0

The number of elements involved is determined by the contents of the VL register. The last element of V*j*, as determined by VL, is concatenated with 64 bits of zeros. The 128-bit quantities are shifted left 1 place if the *k* designator is 0; the special form of the instruction accommodates this feature.

Instruction 152*ijk* executes in the Vector Shift functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
152541		V5	V4,V4<A1	

Assume the following initial register conditions for the above instruction:

```
          (VL) = 4
          (A1) = 3
Element 0 of V4 = 0 00000 0000 0000 0000 0007
Element 1 of V4 = 0 60000 0000 0000 0000 0005
Element 2 of V4 = 1 00000 0000 0000 0000 0006
Element 3 of V4 = 1 60000 0000 0000 0000 0007
```

† Special syntax form

After instruction execution, the first four elements of V5 have been modified as follows:

```
Element 0 of V5 = 0 00000 0000 0000 0000 0073
Element 1 of V5 = 0 00000 0000 0000 0000 0054
Element 2 of V5 = 0 00000 0000 0000 0000 0067
Element 3 of V5 = 0 00000 0000 0000 0000 0070
```

The remaining elements of V5 are unaltered.

The following syntax and its special form shift 128-bit quantities from elements of V*j* by the amount specified in A*k* and enter the result into elements of V*i*. Element *n*-1 of V*j* is concatenated with element *n* and the 128-bit quantity is shifted right by the amount specified in A*k*. The shift is end off with zero fill. The low-order 64 bits are transmitted to element *n* of V*i*.

Result	Operand	Description	Machine instruction
V*i*	V*j*,V*j*>A*k*	Double shift (V*j*) right (A*k*) places to V*i*	153*ijk*
† V*i*	V*j*,V*j*>1	Double shift (V*j*) right one place to V*i*	153*ij*0

The number of elements involved is determined by the contents of the VL register. The first element of V*j* is concatenated with 64 bits of zeros. The 128-bit quantities are shifted right one place if the *k* designator is 0; the special form of the instruction accommodates this feature.

Instruction 153*ijk* executes in the Vector Shift functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
153026		V0	V2,V2>A6	

† Special syntax form

Assume the following initial register conditions for the above instruction.

```
          (VL) = 4
          (A6) = 3
Element 0 of V2 = 0 00000 0000 0000 0000 0017
Element 1 of V2 = 0 60000 0000 0000 0000 0005
Element 2 of V2 = 1 00000 0000 0000 0000 0006
Element 3 of V2 = 1 60000 0000 0000 0000 0007
```

After instruction execution, the first four elements of V0 have been modified as follows:

```
Element 0 of V0 = 0 00000 0000 0000 0000 0001
Element 1 of V0 = 1 66000 0000 0000 0000 0000
Element 2 of V0 = 1 30000 0000 0000 0000 0000
Element 3 of V0 = 1 56000 0000 0000 0000 0000
```

The remaining elements of V0 are unaltered.

BIT COUNT INSTRUCTIONS

The instructions described in this category provide for counting the number of bits in an S or V register or counting the number of leading 0 bits in an S or V register.

Scalar population count

The following syntax counts the number of 1 bits in the contents of S*j* and enters the result into A*i*. A*i* is zeroed if the *j* designator is 0.

Result	Operand	Description	Machine instruction
A*i*	PS*j*	Population count of (S*j*) to A*i*	026*ij*0

Instruction 026*ij*0 executes in the Scalar Leading Zero/Population Count functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
026720		A7	PS2	

Vector population count

The following syntax counts the number of 1 bits in the elements of register V*j* and enters the result into the elements of register V*i*. The number of elements involved is determined by the VL register.

Result	Operand	Description	Machine instruction
† V*i*	PV*j*	Population count of (V*j*) to (V*i*)	174*ij*1

Instruction 174*ij*1 executes in the Reciprocal Approximation functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
174311		V3	PV1	Pop count of V1 to V3

Scalar population count parity

The following syntax enters a 0 in A*i* if S*j* has an even number of 1 bits in S*j* and enters a 1 in S*j* if it has an odd number of 1 bits.

Result	Operand	Description	Machine instruction
† A*i*	QS*j*	Population count parity of (S*j*) to A*i*	026*ij*1

† The instruction is optional on the CRAY-1 models A and B.

Instruction 026ij1 executes in the Scalar Leading Zero/Population Count functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
026271		A2	QS7	Pop count of S7 to A2

Vector population count parity

The following syntax enters a 0 or 1 into the elements of Vi depending on whether the elements of Vj have an even or odd number of 1 bits. A 0 is entered into element n of Vi if there is an even number of 1 bits in element n of Vj; a 1 is entered into element n of Vi if there is an odd number of 1 bits in element n of Vj. The number of elements involved is determined by the VL register.

Result	Operand	Description	Machine instruction
† Vi	QVj	Population count parity of (Vj) to (Vi)	174ij2

Instruction 174ij2 executes in the Reciprocal Approximation functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
174502		V5	QV2	Pop count parity of V2 to V5

† The instruction is optional on the CRAY-1 models A and B.

Scalar leading zero count

The following syntax counts the number of leading zeros in the contents of S*j* and enters the result into A*i*. A*i* is set to 64 if the *j* designator is 0.

Result	Operand	Description	Machine instruction
A*i*	ZS*j*	Leading zero count of (S*j*) to A*i*	027*ij*0

Instruction 027*ij*0 executes in the Scalar Leading Zero/Population Count functional unit.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
027130		A1	ZS3	

BRANCH INSTRUCTIONS

Instructions in this category include conditional and unconditional branch instructions. The branch address is specified by an expression for some instructions and by the contents of a B register for other instructions. An address is always taken to be a parcel address when the instruction is executed. If an expression has a word-address attribute, the assembler issues an error message.

Unconditional branch instructions

There are two unconditional branch instructions. The following syntax sets the P register to the parcel address specified by the low-order 24 bits of the expression. Execution continues at that address.

Result	Operand	Description	Machine instruction
J	*exp*	Jump to *exp*	006*ijkm*

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
006 00002124b		J	TAG1	
006 00001753a		J	LDY3+1	
006 00004533c		J	*+3	

The following syntax sets the P register to the parcel address specified by the contents of register B*jk*. Execution continues at that address.

Result	Operand	Description	Machine instruction
J	B*jk*	Jump to (B*jk*)	0050*jk*

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
005017		J	B17	
005017		J	B.RTNADDR	RTNADDR=17 (octal)

Conditional branch instructions

There are several conditional branch instructions. The following syntax forms test the contents of A0 for the specified condition. If the condition is satisfied, the P register is set to the parcel address specified by the low-order 24 bits of the expression. Execution continues at that address. If the condition is not satisfied, execution continues with the instruction following the branch instruction. For the JAP and JAM instructions, a zero value in A0 is considered positive.

Result	Operand	Description	Machine instruction
JAZ	*exp*	Branch to *exp* if (A0)=0	010*ijkm*
JAN	*exp*	Branch to *exp* if (A0)≠0	011*ijkm*
JAP	*exp*	Branch to *exp* if (A0) positive	012*ijkm*
JAM	*exp*	Branch to *exp* if (A0) negative	013*ijkm*

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
010 00002243d		JAZ	TAG3+2	
011 00004520a		JAN	P.CON1	
012 0000222lc		JAP	TAG2	
013 00002124b		JAM	TAG1	

The following syntax forms test the contents of S0 for the specified condition. If the condition is satisfied, the P register is set to the parcel address specified by the low-order 24 bits of the expression. Execution continues at that address.

If the condition is not satisfied, execution continues with the instruction following the branch instruction. For the JSP and JSM instructions, a zero value in S0 is considered positive.

Result	Operand	Description	Machine instruction
JSZ	*exp*	Branch to *exp* if (S0)=0	014*ijkm*
JSN	*exp*	Branch to *exp* if (S0)≠0	015*ijkm*
JSP	*exp*	Branch to *exp* if (S0) positive	016*ijkm*
JSM	*exp*	Branch to *exp* if (S0) negative	017*ijkm*

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
014 0000222lc		JSZ	TAG2	
015 00002124d		JSN	TAG1+2	
016 00004533c		JSP	*+3	
017 00002367c		JSM	TAG4	

Return jump

The following syntax sets register B00 to the address of the parcel following the instruction. The P register is then set to the parcel address specified by the low-order 24 bits of the expression. Execution continues at that address.

Result	Operand	Description	Machine instruction
R	*exp*	Return jump to *exp*; set B00 to (P)+2	007*ijkm*

The purpose of the instruction is to provide a return linkage for subroutine calls. The subroutine is entered via a return jump. The subroutine returns to the caller at the instruction following the call by executing a branch to the contents of the B register containing the saved address.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
007 00001142d		R	HELP	

Normal exit

The following syntax and its special form cause an exchange sequence. The contents of the instruction buffers are voided by the exchange sequence. If monitor mode is not in effect, the Normal Exit flag in the F register is set. All instructions issued before this instruction are run to completion.

Result	Operand	Description	Machine instruction
EX		Normal exit	004000
† †† EX	*exp*	Normal exit	004*ijk*

When the results of previously issued instructions have arrived at the operating registers, an exchange occurs to the Exchange Package designated by the contents of the Exchange Address (XA) register. The program address stored in the Exchange Package is advanced one parcel from the address of the normal exit instruction. This instruction is used to issue a monitor request from a user program.

The expression in the operand field is optional and has no effect on instruction execution; the low-order 9 bits of the expression value are placed in the *ijk* fields of the instruction.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
004000		EX		
004027		EX	27	

Error exit

The following syntax and its special form are treated as an error condition and an exchange sequence occurs. The contents of the instruction buffers are voided by the exchange sequence. If monitor mode is not in effect, the Error Exit flag in the F register is set. All instructions issued before this instruction are run to completion.

Result	Operand	Description	Machine instruction
ERR		Error exit	000000
† †† ERR	*exp*	Error exit	000*ijk*

† Special syntax form
†† CRAY-1 Computer Systems only

When the results of previously issued instructions have arrived at the operating registers, an exchange occurs to the Exchange Package designated by the contents of the Exchange Address (XA) register. The program address stored in the Exchange Package on the terminating exchange sequence is advanced by one parcel from the address of the error exit instruction.

The error exit instruction is not generally used in program code. This instruction is used to halt execution of an incorrectly coded program that branches to an unused area of memory or into a data area.

The expression in the operand field is optional and has no effect on instruction execution; the low-order 9 bits of the expression value are placed in the *ijk* fields of the instruction.

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
000000		ERR		
000017		ERR	D'15	

MONITOR INSTRUCTIONS

Instructions described in this category are executed only when the CPU is in monitor mode. An attempt to execute one of these instructions when not in monitor mode is treated as a no-op.

The instructions perform specialized functions useful to the operating system.

Channel control

The machine instructions and related CAL syntax for channel control are described in the following paragraphs.

The following syntax sets the Current Address (CA) register for the channel indicated by the contents of A*j* to the value specified in A*k*. It then activates the channel.

Result	Operand	Description	Machine instruction
CA,A*j*	A*k*	Set the channel (A*j*) current address to (A*k*) and begin the I/O sequence	0010*jk*

Before this instruction is issued, the Channel Limit (CL) register should be initialized. As the transfer progresses, the address in CA is incremented. When the contents of CA equals the contents of CL, the transfer is complete for the words at the initial address in CA through one less than the address in CL.

When the *j* designator is 0 or when the contents of A*j* is less than 2 or greater than 25, the instruction executes as a pass instruction. When the *k* designator is 0, CA is set to 1.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001035		CA,A3	A5	

The following syntax sets the Channel Limit (CL) register for the channel indicated by the contents of A*j* to the address specified in A*k*.

Result	Operand	Description	Machine instruction
CL,A*j*	A*k*	Set the channel (A*j*) limit address to (A*k*)	0011*jk*

The instruction is usually issued before issuing the CA,A*j* A*k* instruction.

When the *j* designator is 0 or when the contents of A*j* is less than 2 or greater than 25, the instruction is executed as a pass instruction. When the *k* designator is 0, CL is set to 1.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001134		CL,A3	A4	

The following syntax clears the interrupt flag and error flag for the channel indicated by the contents of A*j*.

Result	Operand	Description	Machine instruction
CI,A*j*		Clear channel (A*j*) interrupt flag	0012*j*0

When the *i* designator is 0 or when the contents of A*j* is less than 2 or greater than 25, the instruction is executed as a pass instruction.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001210		CI,A1		

The following syntax clears the device Master Clear. If (A*j*) represents an output channel, the master clear is set; if (A*j*) represents an input channel, the ready flag is cleared.

Result	Operand	Description	Machine instruction
† MC,A*j*		Clear channel (A*j*) interrupt flag and error flag; set device master-clear (output channel); clear device ready-held (input channel)	0012*j*1

† CRAY X-MP Computer Systems only

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001241		MC,A4		
001201		MC,A0		

Set exchange address

The following syntax transmits bits 12 through 19 of register A*j* to the Exchange Address (XA) register.

Result	Operand	Description	Machine instruction
XA	A*j*	Enter XA register with (A*j*)	0013*j*0

If the *j* designator is 0, the XA register is cleared.

A monitor program activates a user job by initializing the XA register with the address of the user job's Exchange Package and then executing a normal exit (EX).

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001350		XA	A5	

Set real-time clock

The following syntax transmits the contents of register S*j* to the Real-Time clock register. When the *j* designator is 0, the Real-Time Clock register is cleared.

Result	Operand	Description	Machine instruction
RT	S*j*	Enter RTC with (S*j*)	0014*j*0

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001420		RT	S2	
001400		RT	S0	

Programmable clock interrupt instructions

These instructions are supported on Cray computers only if the Programmable Clock Interrupt Option is included.

The following syntax loads the low-order 32 bits from the S*j* register into the Interrupt Interval register (II) and the Interrupt Countdown counter (ICD). The interrupt countdown counter is a 32-bit counter that is decremented by one each clock period until the contents of the counter is equal to 0. At this time, it sets the real-time clock (RTC) interrupt request. The counter is then set to the interval value held in the Interrupt Interval register and repeats the countdown to zero cycle. When an RTC interrupt request is set, it remains set until a clear clock interrupt (CCI) instruction is executed.

Result	Operand	Description	Machine instruction
PCI	S*j*	Set program interrupt interval	0014*j*4

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001434		PCI	S3	Load the low-order 32 bits from (S3) to (II)

The following syntax clears a real-time clock (RTC) interrupt.

Result	Operand	Description	Machine instruction
CCI		Clear clock interrupt	001405

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001405		CCI		Clear clock interrupt

The following syntax enables real-time clock (RTC) interrupts at a rate determined by the value in the Interrupt Interval (II) register.

Result	Operand	Description	Machine instruction
ECI		Enable clock interrupts	001406

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001406		ECI		Enable clock interrupt

The following syntax disables real-time clock (RTC) interrupts until an enable clock interrupt (ECI) instruction is executed.

Result	Operand	Description	Machine instruction
DCI		Disable clock interrupts	001407

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001407		DCI		Disable clock interrupt

Interprocessor interrupt instructions†

The following syntax handles interprocessor interrupt requests. When the k designator is 1, the instruction sets the internal CPU interrupt request in the other CPU. If the other CPU is not in monitor mode, the ICP (Interrupt from Internal CPU) flag sets in the F register causing an interrupt. The request remains until cleared by the receiving CPU.

Result	Operand	Description	Machine instruction
IP	1	Set interprocessor interrupt request of other processor	001401
IP	0	Clear interprocessor interrupt request from other processor	001402

When the k designator is 2, the instruction clears the internal CPU interrupt request set by another CPU.

† CRAY X-MP Computer Systems only

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001401		IP	1	Set interprocessor interrupt request
001402		IP	0	Clear interprocessor interrupt request

Cluster number instructions†

The following syntax sets the cluster number to j to make the following cluster selections:

CLN = 0 No cluster; all shared register and semaphore operations are no-ops, (except SB, ST, or SM register reads, which return a 0 value to Ai or Si).

CLN = 1 Cluster 1

CLN = 2 Cluster 2

CLN = 3 Cluster 3

Each of clusters 1, 2, and 3 has a separate set of SM, SB, and ST registers.

Result	Operand	Description	Machine instruction
CLN	0	Cluster number = 0	001403
CLN	1	Cluster number = 1	001413
CLN	2	Cluster number = 2	001423
CLN	3	Cluster number = 3	001433

† CRAY X-MP Computer Systems only

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
001403		CLN	0	
001413		CLN	1	
001423		CLN	2	
001433		CLN	3	

Operand range error interrupt instructions†

The following syntax forms set and clear the Operand Range Mode flag in the M register. The two instructions do not check the previous state of the flag. When set, the Operand Range Mode flag enables interrupts on operand (address) range errors.

Result	Operand	Description	Machine instruction
ERI		Enable interrupt on (address) range error	002300
DRI		Disable interrupt on (address) range error	002400

Examples:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
002300		ERI		
002400		DRI		

† CRAY X-MP Computer Systems only

PSEUDO INSTRUCTIONS 4

INTRODUCTION

The Cray Assembly Language (CAL) includes a set of instructions known as pseudo instructions that direct the assembler in its task of interpreting the source statements and generating an object program.

Some pseudo instructions are required by the assembler; others are optional. If certain optional instructions are not used, the assembler uses a default setting.

RULES FOR PSEUDO INSTRUCTIONS

Each program module begins with an IDENT instruction and ends with an END instruction. Symbol, micro, macro, and opdef definitions occurring within the program module are cleared before assembling the next program module.

A symbol, macro, or opdef can be defined before the first IDENT pseudo instruction or between an END and a subsequent IDENT pseudo instruction. Such a definition is considered global and can be referenced in any subsequent program module. (Refer to Global Definitions, section 2.)

Micros and redefinable symbols can be defined locally only. Micros and redefinable symbols appearing before the first IDENT or between an END and subsequent IDENT are cleared after assembling the next program module.

Symbolic machine instructions and the pseudo instructions listed below must appear within a program module. They are allowed outside of an IDENT to END sequence only within opdef or macro definitions.

ABS	BSS	DATA	LOC	QUAL	VWD
ALIGN	BSSZ	ELSE	LOCAL	REP	
BITP	COMMENT	ENDTEXT	MICSIZE	SKIP	
BITW	COMMON	ENTRY	MODULE	START	
BLOCK	CON	EXT	ORG	TEXT	

In an absolute program module, the ABS pseudo instruction must appear before any symbolic machine instruction or any of the other above pseudo instructions. The LOCAL pseudo instruction must occur after a macro or opdef prototype statement or DUP or ECHO pseudo instructions, except for intervening comment statements. All other pseudo instructions, macro definitions, and opdef definitions can appear anywhere.

INSTRUCTION DESCRIPTIONS

Pseudo instructions are classified according to their applications, as follows:

Class	Pseudo instructions
Program control	IDENT, END, ABS, COMMENT
Loader linkage	ENTRY, EXT, MODULE, START
Mode control	BASE, QUAL
Block control	BLOCK, COMMON, ORG, LOC, BITW, BITP, BSS, ALIGN
Error control	ERROR, ERRIF
Listing control	LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, ENDTEXT
Symbol definition	=, SET, MICSIZE
Data definition	CON, BSSZ, DATA, VWD, REP
Conditional assembly	IFA, IFC, IFE, ENDIF, SKIP, ELSE
Instruction definition	MACRO, LOCAL, OPDEF, OPSYN, ENDM
Code duplication	DUP, ENDDUP, STOPDUP, ECHO
Micro definition	MICRO, OCTMIC, DECMIC

PROGRAM CONTROL

The pseudo instructions described in this subsection define the limits of a program module and define the type of assembly to be performed.

IDENT - Identify program module

The IDENT pseudo instruction identifies a program module and marks its beginning. The name of the module appears in the heading of the listing produced by CAL (if the title pseudo has not been used) and in the Program Descriptor Table (PDT) of the binary load module.

Format:

Location	Result	Operand
ignored	IDENT	*name*

name Name of the program module; a name must meet the requirements for names given in section 2.

Example:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	WOQXF	

END - End program module

The END pseudo instruction is the final statement of a program module. It causes the assembler to take the following actions:

- Reset the numeric base for assembly to decimal.
- Clear the base, list, qualification, and block stacks.
- Terminate any skipping, macro or opdef definitions, or repeated code.
- Reset the list control options to those determined by the CAL control statement.

Format:

Location	Result	Operand
ignored	END	

ABS - Assemble absolute binary

The ABS pseudo instruction designates that a program module is to be assembled as an absolute rather than a relocatable load module.

Format:

Location	Result	Operand
ignored	ABS	*ignored*

COMMENT - Define Program Descriptor Table comment

The COMMENT pseudo instruction defines a character string to be entered as an informational comment in the Program Descriptor Table (PDT) of the binary load data. The character string is entered as 0 to 10 words of left-justified, blank-filled ASCII data.

If a subprogram contains more than one COMMENT pseudo, the character string from the last COMMENT pseudo is inserted in the PDT.

Format:

Location	Result	Operand
ignored	COMMENT	'*character string*'

'*character string*'
An ASCII character string of 0 to 80 characters

Example:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	CAL	
	COMMENT	'COPYRIGHT CRAY RESEARCH, INC. 1976'	

LOADER LINKAGE

The pseudo instructions described in this subsection provide for loading multiple object program modules and linking them into a single executable program (ENTRY and EXT); defining the contents of the PDT module type field (MODULE); and specifying the main program entry (START).

ENTRY - Specify entry symbols

The ENTRY pseudo instruction specifies symbolic addresses or values which can be referred to by other program modules linked by the loader. Each entry symbol must be a relocatable or absolute symbol defined within the program module.

Format:

Location	Result	Operand
ignored	ENTRY	$symbol_1$,$symbol_2$,...,$symbol_n$

$symbol_i$ An unqualified symbol

Example:

Location	Result	Operand	Comment
1	10	20	35
	ENTRY	EPTNME,TREG	
	.		
	.		
	.		
EPTNME	=	*	
TREG	=	O'17	

EXT - Specify external symbols

The EXT pseudo instruction specifies linkage to symbols that are defined as entry symbols in other program modules. They can be referred to from within the program module but must not be defined within the program module. Symbols specified on the EXT instruction are assembled as having absolute and value attributes with a value of 0. An EXT pseudo instruction is flagged with a warning error and treated as a do-nothing instruction in an absolute assembly.

Format:

Location	Result	Operand
ignored	EXT	$sym_1, sym_2, \ldots, sym_n$

sym_i An unqualified symbol

Examples:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	A	
	.		
	.		
	.		
	ENTRY	VALUE	
VALUE	=	-2.0	
	.		
	.		
	.		
	IDENT	B	
	EXT	VALUE	
	CON	VALUE	The 64-bit external value -2.0 will be stored here by the loader

MODULE - Define program module type for loader

The MODULE pseudo instruction defines the contents of the Program Descriptor Table (PDT) module-type field.

Format:

Location	Result	Operand
ignored	MODULE	*modtype*

modtype Value of the PDT type field:

modtype	Significance
RELOCOVL	Module is a relocatable overlay.

START - Specify program entry

The START pseudo instruction specifies the main program entry. A relocatable program uses the START psuedo as the symbolic address where execution begins following the loading of the program. The named symbol can optionally be an entry symbol specified in an ENTRY pseudo instruction.

If the loader encounters more than one main entry in the program modules being loaded, then execution of the program begins at the first encountered main program entry.

Only one main program entry can be named in a program module.

Format:

Location	Result	Operand
ignored	START	*symbol*

symbol An unqualified symbol

Mode control pseudo instructions define the characteristics of an assembly. The BASE pseudo determines whether notation for numeric data is assumed to be octal or decimal. The QUAL pseudo instruction permits symbols to be defined as qualified or unqualified.

BASE - Declare base for numeric data

The BASE pseudo instruction allows specification of the base of numeric data as being octal, decimal, or mixed, when the base is not explicitly specified by an O' or D' prefix. The default is decimal.

Format:

Location	Result	Operand
ignored	BASE	*base*

base Required single character as follows:

O Octal; all numeric data is assumed to be octal.
D Decimal; all numeric data is assumed to be decimal.
M Mixed; numeric data is assumed to be octal, except for numeric data used for the following, which is assumed to be decimal:

- Statement counts in DUP and conditional statements
- Line count in SPACE
- Bit position or count in BITW, BITP, or VWD
- Character counts as in MICRO, OCTMIC, DECMIC, and data items

* Reverts to use of the previous base in the stack. Each occurrence of a BASE pseudo instruction other than BASE * causes an entry in the stack. Each BASE * removes an entry from the stack and causes the base in use before the current base to be resumed. If the stack is empty when BASE * is encountered, the CAL default mode (decimal) is used.

Example:

```
Location  Result  Operand  Comment
1         10      20       35

          BASE    O        Change base from default to octal
          VWD     50/12    Field size and constant
          .       .        value both octal
          .       .
          .       .
          BASE    D        Change base from octal to decimal
          VWD     40/10    Field size and constant
          .       .        value both decimal
          .       .
          .       .
          BASE    M        Change from decimal to mixed base
          VWD     40/12    Field size decimal; constant
          .       .        value octal.
          .       .
          .       .
          BASE    *        Resume decimal base
          BASE    *        Resume octal base
          BASE    *        Stack empty; resume decimal base
```

QUAL - Qualify symbols

A QUAL pseudo instruction begins or ends a code sequence in which all symbols defined are either qualified by a qualifier specified by the QUAL or are unqualified. Until the first use of a QUAL pseudo instruction, symbols are defined as unqualified. Global symbols cannot be qualified. Thus, QUAL pseudo instructions must not occur before IDENT.

A qualifier applies to symbols only. Names used for blocks, conditional sequences, duplicated sequences, macros, micros, externals, and formal parameters are not affected.

Format:

```
Location  Result  Operand
ignored   QUAL    qualification
```

qualification
Symbol qualifier. Indicates whether symbols are to be qualified or unqualified and, if qualified, indicates the qualifier to be used. The field can contain a *qualifier*, * , or no entry.

qualifier
A 1-character to 8-character name; causes all symbols defined until the next QUAL pseudo instruction to be qualified. Being qualified means that such a symbol can be referenced with or without the qualifier within any sequence in which the qualifier is in effect; however, if the symbol is referenced while some other qualifier is in effect, the reference must be in the form:

/qualifier/symbol

When a symbol is referenced without a *qualifier*, CAL first attempts to find it qualified by the *qualifier* in effect. If the qualified symbol is not defined, CAL attempts to find it in the list of unqualified symbols. The symbol is undefined if both of these searches fail.

* An * resumes use of the qualifier in effect previous to the current qualification. Each occurrence of a QUAL other than a QUAL * causes an entry in a qualification stack. Each QUAL * removes an entry from the stack and causes the qualification in effect to be resumed. If the stack is empty when QUAL * is encountered, symbols are defined unqualified.

No entry
If the operand field of the QUAL is empty, symbols are defined as unqualified until the next occurrence of a QUAL pseudo instruction. An unqualified symbol can be referenced without qualification from any place in the program module, or in the case of global symbols, from any program module assembled after the symbol definition.

Example:

Location	Result	Operand	Comment
1	10	20	35
	.		System default is unqualified
	.		
	.		
ABC	=	1	ABC is defined unqualified
	QUAL	JVR	Symbols will be qualified by JVR
ABC	=	2	
	J	XYZ	
XYZ	S1	+FA2	
	.		
	.		
	.		
	QUAL	DCK	Symbols will be qualified by DCK
ABC	=	3	
	J	/JVR/XYZ	
	.		
	.		
	.		
	QUAL	*	Resume use of JVR
	.		
	.		
	.		
	QUAL		Symbols will be unqualified
A	IFA	DEF,ABC	Test for ABC being defined
B	IFA	DEF,/JVR/ABC	Test for /JVR/ABC being defined
C	IFA	DEF,/DCK/ABC	Test for /DCK/ABC being defined

BLOCK CONTROL

A program, whether assembled into absolute binary or relocatable binary, can be divided into sections called blocks. As assembly of a program proceeds, the user explicitly or implicitly assigns code to specific blocks or reserves areas of a block. The assembler assigns locations in a block consecutively as it encounters instructions or data destined for the block.

By dividing a program into blocks, a programmer can conveniently separate executable sequences of code from nonexecutable data. When no BLOCK or COMMON pseudo instructions are used, all assignment of code is implicitly designated. Two blocks are used, the nominal block and the literals block. In this case, the nominal block is used for all code other than that generated by the occurrence of a literal reference as described in section 2 of this manual. The first occurrence of a reference to a specific literal causes an entry for that literal to be made in the

literals block. At program end, these two blocks are concatenated to form a single program block that is identified to the loader by the name of the program as given on the IDENT pseudo instruction.

When a BLOCK pseudo instruction is used, all code generated or memory reserved (other than literals) from the occurrence of one BLOCK instruction up to the occurrence of the next BLOCK or COMMON instruction is assigned to the designated block. Until the first BLOCK or COMMON instruction, the nominal block is used. Blocks defined by BLOCK instructions are referred to as local blocks because at program end, all of the blocks are concatenated with the nominal block and literals block to form the program block. That is, blocks exist local to the assembly and are invisible to the relocatable loader.

The nominal block is always the first block in the program block. All other local blocks including the literals block are appended in the order that the blocks are first referenced in a BLOCK instruction. The location of the literals block is determined by the first occurrence of a literal. Data is generated in the literals block implicitly by the occurrence of a literal. Explicit data generation or memory reservation is not allowed in the literals block.

For a relocatable assembly, COMMON pseudo instructions are allowed. When a COMMON instruction is used, all code (other than literals) generated or memory reserved from the occurrence of one COMMON instruction up to the occurrence of the next COMMON or BLOCK instruction is assigned to the designated common block. At program end, each common block is identified to the loader by its COMMON name and is available for reference by another program. A common block that is named (labeled) can contain data; a common block that is unnamed (blank) cannot contain data; only memory reservation instructions can be used with this block.

CAL maintains a pushdown stack of block names. It makes an entry in the stack each time a BLOCK or COMMON pseudo instruction names a block to be used and deletes an entry from the stack each time a BLOCK or COMMON pseudo contains * to indicate resumption of the block previously in use. The block in use is always the top entry in the stack. The size of the stack is an assembler option. If the size is exceeded, entries are deleted from the bottom to make room for new entries and an error flag is issued. If the program contains more BLOCK * or COMMON * instructions than there are entries in the stack, the assembler uses the nominal block.

For each block used in a program, CAL maintains an origin counter, a location counter, and a bit position counter. When a block is first established or its use is resumed, CAL uses the counters for that block. During pass 1 of the assembler, the origin and location counters for a block are initially 0. During pass 2, as the assembler constructs the program, it assigns an initial value to each local block origin counter and location counter. Thus, expressions containing relocatable symbols are evaluated differently in pass 2 than in pass 1.

Origin counter

The origin counter controls the relative location of the next word to be assembled or reserved in the block. It is possible to reserve blank memory areas simply by using either the ORG or BSS pseudo instructions to advance the origin counter. When the special element *O is used in an expression, the assembler replaces it with the current parcel-address value of the origin counter for the block in use. W.*O can be used to obtain the word-address value of the origin counter.

Location counter

The location counter is normally the same value as the origin counter and is used by the assembler for defining symbolic addresses within a block. The counter is incremented whenever the origin counter is incremented. It is possible through use of the LOC pseudo instruction to adjust the location counter so that it differs in value from the origin counter or so that it refers to the address relative to a block other than the one currently in use. When the special element * is used in an expression, the assembler replaces it by the current parcel-address value of the location counter for the block in use. W.* can be used to obtain the word-address value of the location counter.

Word-bit-position counter

As instructions and data are assembled and placed into a word, CAL maintains a pointer indicating the next available bit within the word currently being assembled. This pointer is known as the word-bit-position counter. It is 0 when a new word is begun and is incremented by 1 for each completed bit in the word. Its maximum value is 63 for the rightmost bit in the word. When a word is completed, the origin and location counters are incremented by 1 and the word-bit-position counter is reset to 0 for the next word.

When the special element *W is used in an expression, the assembler replaces it with the current value of the word-bit-position counter. The normal advancement of the word-bit-position counter is in increments of 16, 32, and 64 as 1-parcel and 2-parcel instructions or words are generated. This normal advancement can be altered, however, through use of the BITW, BITP, and VWD pseudo instructions.

Force word boundary

The assembler completes a partial word and sets the word-bit-position and parcel-bit-position counters to 0 if either of the following conditions is true:

- The current instruction is an ORG, LOC, BSS, BSSZ, CON, or ALIGN pseudo instruction.

- The current instruction is a DATA or VWD pseudo instruction and the instruction has an entry in the location field.

If an ALIGN pseudo instruction is used, unused parcels are zero filled; otherwise, unused parcels are filled with pass instructions if the last code generating instruction in the current block is not a DATA or VWD instruction. If the last code generating instruction in the current block is DATA or VWD, the unused parcels are zero filled. The S1 S1&S1 instruction is used as the pass instruction.

Parcel-bit-position counter

In addition to the word-bit-position counter, CAL also maintains a counter that points to the next bit to be assembled in the current parcel. This pointer is known as the parcel-bit-position counter. It is 0 when a new parcel is begun and advances by 1 for each completed bit in the parcel. Its maximum value is 15 for the rightmost bit in a parcel. When a parcel is completed, the parcel-bit-position counter is reset to 0.

When the special element *P is used in an expression, CAL replaces it with the current value of the parcel-bit-position counter.

The parcel-bit-position counter will be set to 0 following assembly of most instructions. The pseudo instructions BITW, BITP, DATA, and VWD can cause the counter to be nonzero.

Force parcel boundary

The assembler completes a partially filled parcel and sets the parcel-bit-position counter to 0 if the current instruction is a symbolic machine instruction.

BLOCK - Local block assignment

A BLOCK pseudo instruction establishes or resumes use of a block of code (a local block) within a program module. Each block has its own location, origin, and bit position counters.

Format:

Location	Result	Operand
ignored	BLOCK	*name*

name The content of this field indicates which block will be used for assembling code until the occurrence of the next BLOCK or COMMON pseudo instruction.

name Name of local block

* Resume use of block in use prior to current block

blank Resume use of nominal block

Example:

Location	Result	Operand	Comment
1	10	20	35
	.		Nominal block in use
	.		
	.		
	BLOCK	A	Use block A
	.		
	.		
	.		
	BLOCK		Use nominal block
	.		
	.		
	.		
	BLOCK	*	Return to use of block A

COMMON - Common block assignment

A COMMON pseudo instruction establishes or resumes use of a common block for a relocatable assembly. COMMON is illegal in an absolute assembly.

Data cannot be defined in the blank common block; only storage reservation can be defined.

Format:

Location	Result	Operand
ignored	COMMON	*name*

name Common block to be defined

name Name of a labeled common block

* Resumes use of block in use before current block (cannot be a common block)

blank Blank common block

Example:

Location	Result	Operand	Comment
1	10	20	35
	.		Nominal block
	.		
	.		
	COMMON	FIRST	Labeled common block FIRST
	.		
	.		
	.		
	COMMON		Blank common
	.		
	.		
	.		
	COMMON	*	Return to FIRST

ORG - Set *O counter

The ORG pseudo instruction resets the location and origin counters to the value specified. The expression must have a value or word-address attribute. If the expression has a value attribute, it is assumed to be a word address.

The first occurrence of the ORG instruction in an absolute assembly indicates the address at which binary output begins, and subsequent ORG instructions cannot specify a value lower than the first ORG value. If ORG is omitted, an origin of 0 is assumed.

Format:

Location	Result	Operand
ignored	ORG	*exp*

exp Relocatable expression with positive relocation within block currently in use. In an absolute assembly, *exp* must be absolute if in the nominal block. If the expression is blank, the word address of the next available word in the block is used.

All symbols used in the expression must be previously defined. A force to word boundary occurs before the expression is evaluated.

Example:

Location	Result	Operand	Comment
1	10	20	35
	ORG	O'200	Absolute assembly
	ORG	W.*+O'200	

BSS - Block save

The BSS pseudo instruction reserves a block of memory in a program or a common block. A force to word boundary occurs and then the number of words specified by the operand field expression is reserved. Data is not generated by this pseudo instruction. The block of memory is reserved by increasing the location and origin counters.

Format:

Location	Result	Operand
symbol	BSS	*exp*

symbol Optional symbol. Assigned the word address of the location counter after the force to word boundary occurs.

exp An absolute expression with word-address or value attribute and with all symbols previously defined. The expression value must be positive. A force to word boundary occurs before the expression is evaluated.

The left margin of the listing shows the octal word count.

Example:

```
Location  Result  Operand      Comment
1         10      20           35

          BSS     4
A         CON     'NAME'
          CON     1
          CON     2
          BSS     A+16-W.*     reserve 13 more words
```

LOC - Set * counter

The LOC pseudo instruction resets the location counter to the first parcel of the word address specified. The location counter is used for assigning address values to location field symbols. Changing the location counter allows code to be assembled and loaded at one location, controlled by the origin counter, then moved and executed at another address, controlled by the location counter.

Format:

```
Location  Result  Operand
ignored   LOC     exp
```

exp Relocatable expression with positive relocation, not necessarily within the block currently in use. The expression can also be absolute. All symbols used in the expression must be previously defined. A force word boundary occurs before the expression is evaluated.

Example:

```
Location  Result    Operand     Comment
1         10        20          35
*         In this example, the code is generated and loaded at
*         location 10000 and must be moved by the user to 200
*         before execution
          ABS
          ORG       10000
          LOC       200
A         A1        0
          .
          .
          .
          J         A
```

BITW - Set *W counter

The BITW pseudo instruction sets the current bit position relative to the current word to the value specified. A value of 64 indicates the following instruction is to be assembled at the beginning of the next word (force word boundary). If the counter is set lower than its current value, any code previously generated in the overlapping portion of the word is ORed with any new code.

Format:

```
Location  Result    Operand
ignored   BITW      exp
```

exp An expression with absolute value attribute with positive value less than or equal to 64. When the base is M (mixed), CAL assumes that *exp* is decimal.

Example:

```
Location  Result    Operand     Comment
1         10        20          35
          BITW      D'39
```

BITP - Set *P counter

The BITP pseudo instruction sets the bit position relative to the current parcel to the value specified. A value of 16 forces a parcel boundary. If the current position is in the middle of a parcel with a value of 16, the bit position is set to the beginning of the next parcel; otherwise, the bit position is not changed. If the counter is set lower than its current value, any code previously generated in the overlapping portion of the word is ORed with any new code.

Format:

Location	Result	Operand
ignored	BITP	*exp*

exp An expression with absolute value attribute with positive value less than or equal to 16. When the base is M (mixed), CAL assumes that *exp* is decimal.

Example:

Location	Result	Operand	Comment
1	10	20	35
	BITP	D'14	

ALIGN - Align on an instruction buffer boundary

The ALIGN pseudo instruction ensures that the code following the instruction is aligned on an instruction buffer boundary. An offset is calculated to determine the next 20 (octal) or 40 (octal) word boundary from the current location counter, depending on the machine for which CAL is targeting code (see CPU=*type* option on the CAL control statement). The offset is added to the location and origin counter for the current block. Code is not generated within this offset.

The offset is calculated relative to the beginning of a block. Each local block encountering an ALIGN pseudo by means of the location counter is aligned. The loader is notified to align the program block on an instruction buffer boundary. For each common block encountering an ALIGN pseudo, information is sent to the loader to align that specific block on an instruction buffer boundary.

Format:

Location	Result	Operand
symbol	ALIGN	*ignored*

symbol An optional symbol; it is assigned the parcel address of the location counter after alignment.

The octal value in the output listing immediately to the left of the location field indicates the number of full parcels skipped.

Example using a CRAY-1 Computer:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
4d+	L	=	*	
d 006 00000020a+		J	A	
20a 53	A	ALIGN		

ERROR CONTROL

Two pseudo instructions, ERROR and ERRIF, allow the programmer to generate an assembly error condition.

ERROR - Unconditional error generation

The ERROR pseudo instruction unconditionally sets an assembly error flag.

Format:

Location	Result	Operand
error	ERROR	*ignored*

error A valid error flag character as defined in Appendix C. P is used if this field is null.

Example:

Location	Result	Operand	Comment
1	10	20	35
	IFE ERROR . . .	ABC,LT,DEF,1	

ERRIF - Conditional error generation

The ERRIF pseudo instruction conditionally sets an assembly error flag.

Format:

Location	Result	Operand
error	ERRIF	exp_1,*op*,exp_2

error A valid error flag character(s) as defined in Appendix C. P is used if this field is null.

exp_1, exp_2 Expressions to be compared. Any symbols must have been defined previously.

Expressions are evaluated in pass 2, whereas expressions in other conditional pseudo instructions are evaluated in pass 1. In pass 2, address expressions in local blocks have been relocated relative to the beginning of the program block rather than relative to the local block.

op Operator. Specifies a relation to be satisfied by exp_1 and exp_2 that causes generation of an error. For LT, LE, GT, and GE, only the values of the expressions are examined. The word-address, parcel-address or value attributes and the relocatable, external, or absolute attributes are not compared.

op	Significance
LT	Less than; the value of exp_1 must be less than the value of exp_2.
LE	Less than or equal to; the value of exp_1 must be less than or equal to the value of exp_2.

op	Significance
GT	Greater than; the value of exp_1 must be greater than the value of exp_2.
GE	Greater than or equal to; the value of exp_1 must be greater than or equal to the value of exp_2.
EQ	Equal; the value of exp_1 must be equal to the value of exp_2. The expressions must both be absolute, or both be external relative to the same external symbol, or both be relocatable in the program block or the same common block. The word-address, parcel-address or value attributes must be the same.
NE	Not equal. The two expressions, exp_1 and exp_2, do not satisfy the conditions required for EQ described above.

Example:

Location	Result	Operand	Comment
1	10	20	35
P	ERRIF	ABC,LT,DEF	

LISTING CONTROL

Listing control pseudo instructions allow the programmer to control the content and format of the listing produced by the assembler. The LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, and ENDTEXT listing control pseudo instructions are described in the following paragraphs. These pseudo instructions are not ordinarily listed.

LIST - List control

The LIST pseudo instruction controls the listing. An END pseudo instruction causes options to be reset to the default values.

Format:

Location	Result	Operand
name	LIST	$option_1,option_2,\ldots,option_n$

name Optional list name. If a name is present, the instruction is ignored unless a matching name is specified on a LIST parameter on the CAL control statement. For example, if LIST=*name* appears on the CAL control statement, LIST pseudos with a matching name are not ignored. LIST pseudos with a blank location field are always processed. All of the option names given below can be specified as CAL control statement parameters. The selection of an option on the CAL control statement overrides the enabling or disabling of the corresponding feature by a LIST pseudo.

If L=0 is specified on the CAL control statement, listing output is not generated. In this case, LIST pseudos and list options specified on the CAL control statement have no effect.

option$_i$ Listing option. Specifies that a particular listing feature be enabled or disabled. There can be zero, one, or more options specified or an *. The options allowed are listed below. Defaults are underlined. If no options are specified, OFF is assumed.

* Return to the preceding LIST pseudo.

[ON / OFF]

ON Enable source statement listing. Source statements and code generated are listed.

OFF or blank operand field
Disable source statement listing. Only statements with errors are listed while this option is selected. Listing control pseudo instructions are also listed if LIS option is enabled.

[XRF / NXRF]

XRF Enable cross reference. Symbol references are accumulated and a cross reference listing is produced.

NXRF Disable cross reference. Symbol references are not accumulated. If this option is selected when the END pseudo is encountered, no cross reference is produced. This does not affect the $XRF written by CAL.

[XNS / NXNS]

XNS Include nonreferenced local symbols in the reference. Local symbols that were not referenced in the listing output are included in the cross reference listing.

NXNS Exclude nonreferenced local symbols in the cross reference. If this option is selected when the END pseudo is encountered, local symbols that were not referenced in the listing output are not included in the cross reference.

[DUP / NDUP]

DUP Enable listing of duplicated statements. Statements generated by DUP and ECHO expansions are listed. Conditional statements and skipped statements generated by DUP and ECHO are not listed unless the macro conditional list feature is enabled (MIF).

NDUP Disable listing of duplicated statements. Statements generated by DUP and ECHO are not listed.

[MAC / NMAC]

MAC Enable listing of macro expansions. Statements generated by macro and opdef calls are listed. Conditional statements and skipped statements generated by macro and opdef calls are not listed unless the macro conditional list feature is enabled (MIF).

NMAC Disable listing of macro expansions. Statements generated by macro calls are not listed.

[MIF / NMIF]

Conditional statements and skipped statements in source code are listed regardless of whether this option is enabled or disabled.

MIF Enable macro conditional listing. Conditional statements and skipped statements generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, are listed. The listing of macro expansions or the listing of duplicated statements must also be enabled.

NMIF Disable macro conditional listing. Conditional statements and skipped statements are not listed.

[MIC / NMIC]

Source statements containing a micro reference or a concatenation character are listed before editing regardless of whether this option is enabled or disabled.

MIC Enable listing of generated statements before editing. Statements which are generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, and which contain a micro reference or concatenation character are listed before and after editing. The listing of macro expansions or the listing of duplicated statements must also be enabled.

NMIC Disable listing of generated statements before editing. Statements generated by a macro or opdef call, or by a DUP or ECHO pseudo instruction, are not listed before editing.

[LIS / NLIS]

LIS Enable listing of the pseudo instructions LIST, SPACE, EJECT, TITLE, and SUBTITLE. These statements are listed regardless of whether the source statement listing is enabled.

NLIS Disable listing of these pseudo instructions.

[WEM / NWEM]

WEM Enable warning errors. Each statement containing a warning error is written to the source listing and the error listing. A logfile message is issued giving the number of warning errors.

NWEM Disable warning errors. Warning errors are ignored.

[TXT / NTXT]

TXT Enable global text source listing. Each statement following a TEXT pseudo instruction is listed through the ENDTEXT instruction, if the listing is otherwise enabled.

NTXT Disable global text source listing. Statements following a TEXT pseudo instruction through the following ENDTEXT instruction are not listed.

[WRP / NWRP]

WRP Enable warning error message for a relocatable parcel address within a 22-bit expression of an instruction 020, 021, 040, or 041.

NWRP Disable warning error message for a relocatable parcel address within a 22-bit expression of an instruction 020, 021, 040, or 041.

[WMR / NWMR]

WMR Enable warning error message for macro and opdef redefinition. If the name of a macro is the same as a currently defined pseudo instruction or macro, a warning message is issued. If an opdef syntax is being redefined, a warning message is also issued.

NWMR Disable warning error message for macro and opdef redefinition.

SPACE - List blank lines

The SPACE pseudo instruction specifies the number of blank lines to be inserted in the listing.

Format:

Location	Result	Operand
ignored	SPACE	*count*

count An absolute expression specifying the number of blank lines to insert in the listing. When the base is M (mixed), CAL assumes that *count* is decimal.

EJECT - Begin new page

The EJECT pseudo instruction causes a page eject on the output listing.

Format:

Location	Result	Operand
ignored	EJECT	*ignored*

TITLE - Specify listing title

The TITLE pseudo instruction specifies the main title to be printed on the listing.

Format:

Location	Result	Operand
ignored	TITLE	'*character string*'

'*character string*'
A character string to be printed as the main title on subsequent pages of the listing. A maximum of 64 characters is allowed.

SUBTITLE - Specify listing subtitle

The SUBTITLE pseudo instruction specifies the subtitle to be printed on the listing. The instruction also causes a page eject.

Format:

Location	Result	Operand
ignored	SUBTITLE	'*character string*'

'*character string*'
A character string to be printed as the subtitle on subsequent pages of the listing. A maximum of 64 characters is allowed.

TEXT - Declare beginning of global text source

Source lines following the TEXT pseudo instruction through the next ENDTEXT pseudo instruction are treated as *text* source statements. These statements are listed only when the TXT listing option is enabled. A symbol defined in *text* source is treated as a system text symbol for cross reference purposes. That is, such a symbol is not listed in the cross reference unless there is a reference to the symbol from a listed statement. The */block/* or system text *name* column of the cross reference listing contains the text name, unless the symbol is a COMMON block symbol. In this case the COMMON block name appears in this column.

Symbols defined in *text* source are global if the text appears before an IDENT pseudo instruction. Symbols in *text* source are local to a program module if the text appears between IDENT and END pseudo instructions.

The TEXT pseudo instruction is listed if the listing is on or if the LIS listing option is enabled regardless of other listing options.

The TEXT and ENDTEXT pseudo instructions have no effect within system text.

Format:

Location	Result	Operand
name	TEXT	'*character string*'

name Optional name of text. *name* is used as the name of the text source following until the next ENDTEXT pseudo instruction. It is associated with any symbols defined in the text, and is listed in the *name* column of the cross reference listing.

'*character string*'
An optional character string to be printed as the subtitle on subsequent pages of the listing. This operand and the TXT option cause a page eject. A maximum of 64 characters is allowed. If the operand field is blank then the subtitle is not affected and no page eject occurs. If the operand field is nonblank then the preceding subtitle is lost and replaced by the character string in the operand field.

ENDTEXT - Terminate global text source

The ENDTEXT pseudo instruction terminates *text* source initiated by a TEXT instruction. An IDENT or END pseudo instruction also terminates *text* source. The ENDTEXT instruction is not listed unless the TXT option is enabled with the exception that if the LIS option is enabled, the ENDTEXT instruction is listed regardless of other listing options.

Format:

Location	Result	Operand
ignored	ENDTEXT	

Example (with TXT option off):

Source listing:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	TEXT	
A	=	2	
TXTNAME	TEXT	'AN example.'	
B	=	3	
C	=	4	
	ENDTEXT		
	A1	A	
	A2	B	
	END		

Output listing:

```
Code generated          Location  Result    Operand     Comment
                        1         10        20          35

                                  IDENT     TEXT
                        2         A         =           2
                                  TXTNAME   TEXT        'An example.'
0a  022102                                  A1          A
 b  022203                                  A2          B
                                                        END
```

Cross reference:

Value	Symbol	/block/ or Name	Symbol references
2	A		1: 2 D 1: 4
3	B	TXTNAME	1: 5

SYMBOL DEFINITION

The pseudo instructions =, SET, and MICSIZE define symbols used in the program.

Requirements for symbols are given in section 2 of this publication.

= - Equate symbol

The = pseudo instruction defines a symbol with the value and attributes determined by the expression. The symbol is not redefinable.

Format:

Location	Result	Operand
symbol	=	*exp,attribute*

symbol An unqualified symbol. The symbol is implicitly qualified by the current qualifier. The symbol must not be defined already. The location field can be blank.

exp Any expression

attribute P, W, or V indicating parcel, word, or value attribute (optional). Attribute, if present, is used instead of the expression's attribute. An expression with word-address attribute is multiplied by four if a parcel-address attribute is specified; an expression with parcel-address attribute is divided by four if word-address attribute is specified. A relocatable expression cannot be specified as having value attribute.

Example:

Location	Result	Operand	Comment
1	10	20	35
SYMB	=	A*B+100/4	

SET - Set symbol

The SET pseudo instruction resembles the = pseudo instruction. However, a symbol defined by SET is redefinable.

Format:

Location	Result	Operand
symbol	SET	*exp,attribute*

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
100	SIZE	=	O'100	
22	PARAM	SET	D'18	
10	WORD	SET	*W	
10	PARCEL	SET	*P	
	SIZE	=	SIZE+1	(Illegal)
24	PARAM	SET	PARAM+2	(Legal)

MICSIZE - Set redefinable symbol to micro size

The MICSIZE pseudo instruction defines the symbol in the location field as an absolute symbol with a value equal to the number of characters in the micro string whose name is in the operand field. Another SET or MICSIZE instruction with the same symbol redefines the symbol to the new value.

Format:

Location	Result	Operand
symbol	MICSIZE	*name*

symbol An unqualified symbol. The symbol is implicitly qualified by the current qualifier. The location field can be blank.

name The name of a micro string previously defined.

DATA DEFINITION

Data definition instructions are the only pseudo instructions that generate object binary. The only other instructions that are translated into object binary are the symbolic machine instructions. An instruction that generates binary cannot be used in a blank common block.

- CON Places an expression value into one or more words
- BSSZ Generates words of zero
- DATA Generates one or more words of numeric or character data
- VWD Generates a variable-width field of word-oriented data
- REP Generates loader duplication table entries

CON - Generate constant

The CON pseudo instruction generates one or more full words of binary data. This pseudo always forces to a word boundary.

Format:

Location	Result	Operand
symbol	CON	$exp_1, exp_2, \ldots, exp_n$

symbol Optional symbol assigned the word-address value of the location counter after the force to word boundary occurs.

exp$_i$ An expression whose value is to be inserted into a single 64-bit word. If an expression is null, a single zero word is generated. A force word boundary occurs before any operand field expressions are evaluated. A double-precision, floating-point constant is not allowed.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
0000000000000007777017				
	A	CON	O'7777017	
		CON	A	

BSSZ - Generate zeroed block

The BSSZ pseudo instruction causes a block of words containing zeros to be generated. A force to word boundary occurs, and then the number of zero words specified by the operand field expression is generated.

Format:

Location	Result	Operand
symbol	BSSZ	*exp*

symbol Optional symbol. Assigned the word-address value of the location counter after the force to word boundary occurs.

exp An absolute expression with word-address or value attribute and with all symbols previously defined. The expression value must be positive and specifies the number of 64-bit words containing zeros to be generated. A blank operand field results in no data generation.

The left margin of the listing shows the octal word count.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
144		BSSZ	D'100	

DATA - Generate data words

The DATA pseudo instruction generates data from the items listed. The length of the field generated for each data item depends on the type of constant involved. A word boundary is not forced between data items.

Format:

Location	Result	Operand
symbol	DATA	$data_1, data_2, \ldots, data_n$

symbol Optional symbol assigned the address value of the location counter after a force to word boundary. If no symbol is present, a force to word boundary does not occur.

$data_i$ A numeric or character data item

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
00000000000000005252		DATA	O'5252,A'ABC'R	
00000000000020241103				
0405022064204010020040		DATA	'ABCD'	
0425062164404010020040		DATA	'EFGH'	
040502206420		DATA	'ABCD'*	
10521443510		DATA	'EFGH'*	
00000000000000000000		DATA	'ABCD'12R	
040502206420				
10521443510		DATA	'EFGHIJ'*	
044512				
0405022064204010020040	LL2	DATA	'ABCD'	
00000000000000000144		DATA	100	
037743527461670430 2142		DATA	1.25E-9	
0521102225144022251440		DATA	'THIS IS A MESSAGE'*L	
0404402324252324640507				
0424				
000		VWD	8/0	

VWD - Variable word definition

The VWD pseudo instruction allows data to be generated in fields from 0 to 64 bits wide. Fields can cross word boundaries. Data begins at the current bit position unless a symbol is used. If a symbol is used, a force word boundary occurs and the data begins at the new current bit position.

Format:

Location	Result	Operand
symbol	VWD	$n_1/exp_1,n_2/exp_2,\ldots,n_m/exp_m$

symbol Optional symbol. If present, a force word boundary occurs.

n_i Field width, specifying the number of bits in the field. A numeric constant or symbol, with absolute and value attributes. The value of n_i must be positive and less than or equal to 64. When the base is M (mixed), CAL assumes that n_i is decimal.

exp_i An expression whose value is to be inserted in the field.

Example:

In the following example, the value of SIGN is 1, the value of FC is 0, the value of ADD is 653 (octal), and the value of DSN is $IN in ASCII code.

```
Code generated          Location  Result  Operand                 Comment
                        1         10      20                      35

                                  BASE    M
                        PDT       BSS     0
1000000000000023440515            VWD     1/SIGN,3/0,60/A'"NAM"'R
10000000653                       VWD     1/1,6/FC,24/ADD
                  41    REMDR     =       64-*W
          00011044516             VWD     REMDR/DSN
```

REP - Loader replication directive

The REP pseudo instruction generates loader duplication table entries which direct the loader to load one or more copies of a block of binary data words. The loader generates the copies immediately upon encountering the table. The data words to be copied must have been generated by previous assembler statements and cannot contain any external or relocatable fields.

The REP pseudo instruction is allowed only in relocatable assemblies and cannot be used in a blank common block.

Storage for all copies of the source data word must be reserved by the user with BSS, ORG, or other pseudo instructions. The source data words and all copies must be within the current program block or common block.

A force word boundary occurs before evaluating the operand field expressions.

Format:

Location	Result	Operand
ignored	REP	*ct,swa,inc,bsz*

ct Duplication count. An absolute expression with word address or value attribute, value greater than 0. The duplication count specifies the number of times the data words are to be copied.

swa Source word address. A relocatable expression with word-address attribute, specifying the first word address of the data words to be copied.

inc Increment. An absolute expression with word-address or value attribute with positive value. The loader stores copies of the source data words at *swa+inc*, *swa*+2**inc*,... until the duplication count is exhausted.

Default value is 1 if the third subfield is null or missing. The loader does no duplication if the increment has zero value. The increment must have a value less than 256.

bsz Block size. An absolute expression with word address or value attribute and with a positive value not greater than the value of *inc*. The block size specifies the number of words in the block at *swa* which are to be copied. The default value for *bsz* is 1 if the fourth subfield is null or missing.

Example:

Location	Result	Operand	Comment
1	10	20	35
N	=	D'64	
A	DATA	''	Word or space characters
	REP	N-1,A	Fill the array with spaces
	BSS	N-1	Reserve storage for the array
B	CON	1.0	
	CON	2.0	
	BSS	18	
	ORG	B	Complex constant (1.0,2.0)
	REP	9,B,2,2	Fill array B with complex constant

CONDITIONAL ASSEMBLY

The instructions described in this section permit optional assembly or skipping of source code. The conditional pseudo instructions IFA, IFC, or IFE determine whether a sequence of instructions following the test is to be skipped or assembled. The end of the conditional sequence is determined by a count of instructions provided on the test instruction or by an ENDIF pseudo instruction with a matching location field name.

The ELSE pseudo instruction provides a means of reversing the effect of a previous IFA, IFE, IFC, SKIP, or ELSE instruction. The SKIP pseudo instruction unconditionally skips following statements.

When skipping under control of a statement count, comment statements (asterisk in column 1) and continuation lines (comma in column 1) are not included in the statement count.

IFA - Test expression attribute for assembly condition

The IFA pseudo instruction tests an attribute of an expression. If the expression has the specified attribute, assembly continues with the next statement. If the attribute test is failed, then subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

If an assembly error is detected, assembly continues with the next instruction.

Formats:

Location	Result	Operand
ifname	IFA	*attribute,exp*
	IFA	*attribute,exp,count*

ifname Optional name of conditional sequence of code

count Statement count. An absolute expression with positive value. When the base is M (mixed), CAL assumes that count is decimal. A count parameter is required if *ifname* is missing, otherwise it is ignored. A missing or null count subfield gives a zero count if *ifname* is not present.

attribute A mnemonic signifying an attribute of *exp*. An expression has only one of the attributes PA, WA, or VAL and has only one of the attributes EXT, REL, or ABS.

An attribute can also be any of the following letters preceded by a complement sign (#) indicating that the second subfield does not satisfy the corresponding condition.

Mnemonic	Attribute
PA	Parcel address
WA	Word address
VAL	Value
EXT	External
REL	Relocatable
ABS	Absolute
COM	Relocatable; relative to a common block.
DEF	All symbols in the expression *exp* have been previously defined.
SET	The symbol in the second subfield is a redefinable symbol.
REG	The second subfield contains a valid A, B, S, T, or V register designator.
MIC	The name in the second subfield is a micro name.

exp Expression. The second subfield must either be a valid expression, symbol, name, or character string depending on the attribute mnemonic.

For PA, WA, VAL, EXT, REL, ABS, and COM, the second subfield must be a valid expression with all symbols previously defined.

For DEF, the second subfield must be a valid expression.

For SET, the second subfield must be a valid defined symbol.

For REG, the second subfield can be a string of any characters except blank or comma.

For MIC, the second subfield must be a valid name.

Expressions are evaluated in pass 1. Expressions that are relocatable addresses in local blocks have values relative to the beginning of the local block rather than the program block. Address expressions in a local block other than the nominal block on an absolute assembly are considered relocatable in pass 1.

IFE - Test expressions for assembly condition

The IFE pseudo instruction tests a pair of expressions for a condition under which code is to be assembled if the relation specified by the operation (*op*) is satisfied. That is, if the relationship is true, then assembly resumes with the next statement. If the relationship is not satisfied (is false), then subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

If an assembly error is detected, assembly continues with the next statement.

Formats:

Location	Result	Operand
ifname	IFE	exp_1, op, exp_2
	IFE	$exp_1, op, exp_2, count$

ifname Optional name of a conditional sequence of code

exp_1, *exp_2* Expressions to be compared. All symbols in the expression must be previously defined.

Expressions are evaluated in pass 1. Expressions that are relocatable addresses in local blocks have values relative to the beginning of the local block rather than the program block. Address expressions in a local block other than the nominal block in an absolute assembly are considered relocatable in pass 1.

count Statement count; must be an absolute expression with positive value. When the base is M, mixed, CAL assumes that count is decimal. A count parameter is required if *ifname* is missing, otherwise it is ignored. A missing or null count subfield gives a zero count.

op Specifies relation to be satisfied by *exp_1* and *exp_2*. It must be one of the following:

op	Significance
LT	Less than; the value of *exp_1* must be less than the value of *exp_2*.
LE	Less than or equal to; the value of *exp_1* must be less than or equal to *exp_2*.
GT	Greater than; the value of *exp_1* must be greater than the value of *exp_2*.
GE	Greater than or equal to; the value of *exp_1* must be greater than or equal to *exp_2*.
EQ	Equal; the value of *exp_1* must be equal to the value of *exp_2*. The expressions must both be absolute, or both be external relative to the same external symbol, or both be relocatable in the same block. The word-address, parcel-address or value attributes must be the same.
NE	Not equal; the expressions *exp_1* and *exp_2* do not satisfy the conditions required for EQ described above.

IFC - Test character strings for assembly condition

The IFC pseudo instruction tests a pair of character strings for a condition under which code is to be assembled if the relation specified by the operation (*op*) is satisfied. That is, if the relationship is not satisfied (is false), then subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

If an assembly error is detected, assembly continues with the next statement.

Formats:

Location	Result	Operand
ifname	IFC	'$char_1$',*op*,'$char_2$'
	IFC	'$char_1$',*op*,'$char_2$',*count*

ifname Optional name of a conditional sequence of code

op Operator. Specifies relation to be satisfied by $char_1$ and $char_2$. *op* must be one of the following:

op	Significance
LT	Less than
LE	Less than or equal to
GT	Greater than
GE	Greater than or equal to
EQ	Equal to
NE	Not equal to

'$char_1$', '$char_2$' Character strings to be compared. The first and third subfields can be null (empty) indicating a null character string.

The ASCII character code value of each character in $char_1$ is compared with the value of each character in $char_2$, beginning at the left and continuing until an inequality is found or until the longer string is exhausted. A zero value is substituted for missing characters in the shorter string.

Refer to Appendix F for the ASCII character code values.

Micros and formal parameters can be contained in the character strings.

A character string can be delimited by a character other than an apostrophe. Any ASCII character other than a comma or space can be used. Two consecutive occurrences of the delimiting character indicate a single such character. For example,

```
AIF IFC =O'100=,EQ,*ABCD***
```

compares the character strings O'100 and ABCD*.

count Statement count; must be an absolute expression with positive value. A missing or null count subfield gives a zero count. If the base is M (mixed), CAL assumes that count is decimal. A count parameter is required if *ifname* is missing; otherwise, it is ignored.

SKIP - Unconditionally skip statements

The SKIP pseudo instruction unconditionally skips subsequent statements. If a location field name is present, skipping stops when an ENDIF or ELSE with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

Format:

Location	Result	Operand
ifname	SKIP	*count*

ifname Optional name of conditional sequence of code

count Statement count; must be an absolute expression with positive value. If the base is M (mixed), CAL assumes that count is decimal. A count parameter is required if *ifname* is missing; otherwise, it is ignored. A missing or null count subfield gives a zero count.

ENDIF - End conditional code sequence

The ENDIF pseudo instruction terminates skipping initiated by an IFA, IFE, IFC, ELSE, or SKIP pseudo instruction with the same location field name. Otherwise, ENDIF acts as a do-nothing pseudo instruction. ENDIF has no effect on skipping, which is controlled by a statement count.

Format:

Location	Result	Operand
ifname	ENDIF	

ifname Required name of conditional code sequence

NOTE

While skipping, an END statement is recognized and terminates skipping.

ELSE - Toggle assembly condition

The ELSE pseudo instruction terminates skipping initiated by an IFA, IFC, IFE, ELSE, or SKIP pseudo instruction with the same location field name. If statements are currently being skipped under control of a statement count, ELSE has no effect.

If the assembler is not currently skipping statements, ELSE initiates skipping. Skipping is terminated by an ENDIF or ELSE pseudo instruction with a matching location field name.

Format:

Location	Result	Operand
ifname	ELSE	

ifname Required name of conditional sequence of code

Examples of conditional assembly

```
Location  Result  Operand              Comment
1         10      20                   35

          IFA     #DEF,A,1
A         =       10                   Define A if not already defined

BTEST     IFA     ABS,SYM
X         ERR                          Generate X error if SYM is
                                       absolute
BTEST     ELSE
          A1      SYM                  Assemble if SYM not absolute
BTEST     ENDIF

*         Assemble BSSZ instruction if W.* is less than BUF,
*         otherwise assemble ORG
          IFE     W.*,LT,BUF,2
          BSSZ    BUF-W.*              Generate words of zero to
*                                      address BUF
          SKIP    1                    Skip next statement
          ORG     BUF

          IFC     '"L"',EQ,,1
          ERR                          Error if micro string
*                                      defined by L is empty

X         IFC     'ABCD',GT,'ABC'      ABCD is greater than ABC

Y         IFC     ' ',GT,              Single space is greater
*                                      than null string
Z         IFC     '''',EQ,*'*          Single apostrophe equals
*                                      single apostrophe
```

INSTRUCTION DEFINITION

The CAL assembler allows a programmer to identify a sequence of instructions to be saved for assembly at a later point in the source program. When the sequence is defined, CAL stores it in a list of definitions but does not assemble the sequence. Each time the defined sequence is referenced, it is placed in the source program and is assembled. Four types of defined sequences are recognized: macro, opdef, dup, and echo.

A macro or opdef definition identifies a sequence of instructions that is referenced at a later point in the source program by a single instruction, the macro, or opdef call. Each time the macro or opdef call occurs, the definition sequence is placed in the source program. For a macro call, the name in the result field matches the name associated with

the macro. For an opdef call, the syntax, or form, of the instruction matches the syntax associated with the opdef definition. Thus, a macro call resembles a pseudo instruction and an opdef call resembles a symbolic machine instruction.

A dup or echo definition identifies a sequence of instructions that is assembled repeatedly, immediately following the definition. The number of times the sequence is assembled depends on the parameters on the DUP or ECHO pseudo.

A macro or opdef is defined as global if it occurs before the IDENT that begins the program module. Macro and opdef definitions are local if they occur within an IDENT, END sequence. Every local definition is removed from the assembler tables at the end of a program module. A global definition can be referenced in any program module following the definition.

Dup and echo definitions are removed from the assembler tables as soon as the definition sequence has been assembled the proper number of times.

A definition has three parts: a header, a body, and an end.

Definition header

The header consists of a MACRO, OPDEF, DUP, or ECHO instruction, a prototype statement for a macro or opdef definition, and, optionally, LOCAL pseudo instructions. For a macro, the prototype statement provides a name and a list of formal parameters and default arguments. For an opdef, the prototype statement supplies the syntax and the formal parameters. LOCAL pseudo instructions identify symbols that CAL must render unique to the assembly each time the definition sequence is placed in the source program.

Definition body

The body of the definition begins with the first instruction following the header. The body consists of a series of CAL instructions other than END and can include other definitions and calls. However, a definition used within another definition is not recognized until the definition in which it is contained is called. Therefore, an inner definition cannot be called before the outer definition is called for the first time.

A comment statement identified by an asterisk in column 1 is ignored in the definition header or definition body. Such comments are not saved as a part of the definition sequence. Comment fields on other statements in the body of a definition are saved.

The body of the definition is saved before editing for micros, concatenation marks, and lowercase comments. Editing occurs when the definition is assembled each time it is called. An inner nested definition is not edited until it is called. ENDDUP, ENDM, END, and LOCAL pseudo instructions and prototype statements cannot contain any micros or concatenation characters. These statements are not edited when they occur in a definition.

Definition end

The end of a macro or opdef definition is signaled by an ENDM pseudo instruction with the proper name in the location field. The end of a dup or echo definition is signaled by a statement count or by an ENDDUP with the proper name in the location field.

Assembly source stack

Each time a definition sequence of code is referenced (called), an entry is made in a pushdown stack, called the assembly source stack. The most recent entry indicates the current source of statements to be assembled. When a definition is called within a definition sequence being assembled, another entry is made in the stack, and assembly continues with the new definition sequence belonging to the inner, or nested, call. When the end of a definition sequence is reached, the most recent stack entry is removed and assembly continues with the previous stack entry. When the stack becomes empty, assembly continues with statements from the source file.

An inner nested call can be recursive, that is, it can reference the same definition that is referenced by an outer call.

The depth of nested calls permitted by CAL is limited only by the amount of memory available.

An inner definition must be entirely contained within the next outer definition.

Skipping of statements due to conditional assembly must not extend beyond the end of a definition sequence being assembled. An error is generated and skipping is terminated if this condition occurs.

The sequence field in the right margin of the listing shows the definition name and nesting depth for definition sequences being assembled.

Formal parameters

Formal parameters are defined in the definition header. Three types of formal parameters are recognized: positional, keyword, and local. Formal parameters are recognized in the definition body whenever they are delimited by a space, comma, beginning or end of a statement, or any of the following characters:

! " # & ' ( ) * + - . / < = > __(underline)

There can be from 0 to 511 formal parameters. Positional, keyword, and local parameters must all have unique names within a given definition.

Formal parameter names should not be END, ENDM, ENDDUP, or LOCAL. When the definition is referenced, substitution of actual arguments occur in any pseudo instruction with these names contained in any inner definition.

MACRO - Macro definition

The MACRO pseudo instruction is the first statement of a macro definition. The macro header consists of the MACRO pseudo instruction, a prototype statement, and optional LOCAL pseudo instructions.

Format:

Location	Result	Operand	
ignored	MACRO		HEADER:
lfp	*name*	$p_1,p_2,\ldots,p_n,e_1=d_1,e_2=d_2,\ldots,e_m=d_m$	Prototype statement
	LOCAL	$sym_1,\ldots,sym_r$	Optional LOCAL pseudo instructions
	.		DEFINITION BODY
	.		
	.		
name	ENDM		DEFINITION END

Prototype statement parameters:

lfp Optional location field parameter. It must be a valid name. If present, it is a positional parameter.

name Name of the macro, must be valid name. If the name is the same as a currently defined pseudo instruction or macro, then this definition redefines the operation associated with the name, and a warning message is issued to the logfile (see Appendixes C and D).†

p_i Positional parameter, must be a valid name. There can be none, one, or more positional parameters.

e_i Keyword parameter, must be a valid name. There can be none, one, or more keyword parameters.

d_i Default argument for keyword parameter e_i. An argument string can consist of any string of ASCII characters except comma or blank.

If the first character of the default argument d_i is a left parenthesis, then the string must be terminated by a matching right parenthesis. Such an argument is called an embedded argument, and consists of all characters between the enclosing parentheses. An embedded string can contain commas and blanks, and can also contain pairs of matching left and right parentheses.

A space or comma following the equal sign specifies a null (empty) character string as the default argument.

The default argument for a positional parameter is an empty string.

An inner macro definition must be entirely contained within the outer definition.

Macro calls

A macro definition can be called by an instruction of the following format:

Location	Result	Operand
loc	*name*	$a_1,a_2,\ldots,a_j,f_1=b_1,f_2=b_2,\ldots,f_k=b_k$

loc Location field argument; must be a valid name. If a location field parameter is specified on the macro definition, this symbol is optional. It is substituted wherever the location field parameter occurs in the definition.

If no location field parameter is specified in the definition, this field must be empty.

name Macro name; must match the name specified in the macro definition.

† Warning error depends on the WMR and NWMR features of the CAL control statement or the LIST pseudo instruction.

a_i Actual argument string corresponding to positional parameters in the definition prototype statement.

The first argument, a_1, is substituted for the first positional parameter, p_1, in the prototype operand field, the second argument, a_2, is substituted for the second positional parameter, p_2, etc. If the number of operand subfields is less than the number of positional parameters, null argument strings are used for the missing arguments.

Two consecutive commas indicate a null (empty) argument string.

f_i A keyword parameter. Each keyword parameter f_i must match a keyword parameter in the macro definition. The keyword parameters can be listed in any order; they do not need to match the order given in the macro definition. The default arguments specified in the macro definition are used as the actual argument for missing keyword parameters.

Keyword parameters are not recognized until after n subfields (n commas), where n is the number of positional parameters in the operand field of the macro definition.

b_i Actual argument string for keyword parameter f_i. A space or comma following the equal sign indicates a null (empty) argument string.

An actual argument string can consist of any ASCII characters except comma or blank. A comma separates subfields and a blank terminates the operand field.

If the first character of the actual argument is a left parenthesis, then the string must be terminated by a matching right parenthesis. Such an argument is called an embedded argument and consists of all characters between the enclosing parentheses. An embedded string can contain commas and blanks, and can also contain pairs of matching left and right parentheses.

The actual argument string for each positional and keyword parameter is substituted in the definition sequence wherever the formal parameter occurs. Embedded argument strings are substituted without the enclosing parentheses.

OPDEF - Operation definition

The OPDEF pseudo instruction is the first statement of an opdef definition.

Format:

Location	Result	Operand	
			HEADER:
name	OPDEF		Macro pseudo
lfp	*synres*	*synop*	Prototype statement
	LOCAL	$sym_1,\ldots,sym_r$	Optional LOCAL pseudo
	.		instructions
	.		
	.		DEFINITION BODY
name	ENDM		DEFINITION END

name Name of the OPDEF definition. This name is used as an identification of the definition and has no association with names appearing in the result field of instructions. The name must match the name in the location field of the ENDM pseudo instruction which ends the definition. This name is also listed in the sequence of lines generated by an OPDEF call.

lfp Optional location field parameter; must be a valid symbol. If present, the location field parameter is a positional parameter.

synres Result field syntax; must be one, two, or three subfields specifying a valid result field syntax. Positional parameters are indicated by register symbols and expression designators.

synop Operand field syntax; must be one, two, or three subfields specifying a valid operand field syntax. Positional parameters are indicated by register symbols and expression designators.

Symbolic instruction syntax

The symbolic instruction syntax forms allowed by CAL can be described in terms of italic letters defining possible character strings representing registers, expressions, mnemonics, operators, and combinations of these strings.

If the syntax is the same as a currently defined opdef, the definition redefines the operation associated with the syntax and a warning message is issued (see Appendixes C and D).†

An OPDEF must match the syntax formed by combining one of the patterns from the result field syntax with any of the patterns in the operand field syntax as shown below.

Result field syntax	Operand field syntax
r	
c	*f*
c,w	*f,g*
c,w,w	*f,m*
w,w	*f,g,w*
w,w,w	
m	

Expressions

The letter *x* represents an expression. It is signified in the prototype statement by a name whose first character is *@*. The name is taken as a formal parameter.

Registers

The letters *a*, *b*, *sb*,†† *s*, *t*, *st*,†† *sm*,†† and *v* represent the A, B, SB,†† S, T, ST,†† SM,†† and V register designators, which must be signified in the opdef prototype statement by one of the following character strings.

a :	A.*sym*	where *sym* is a valid symbol and is used as a positional parameter
b :	B.*sym*	
sb:	SB.*sym*††	
s :	S.*sym*	
t :	T.*sym*	
st:	ST.*sym*††	
sm:	SM.*sym*††	
v :	V.*sym*	

† Warning error depends on the WMR and NWMR features of the CAL control statement or the LIST pseudo instruction

†† For CRAY X-MP Computer Systems only

The letters *r*, *d*, and *e* represent sets of registers and are defined as follows:

r:	*d*:	*e*:
a	*a*	*s*
b	*s*	*v*
s	SB	SB
SB	*v*	
t		
v		
sb		
st		
sm		

The letter *c* represents the special register designators.

c:
CA
CE
CI
CL
MC
RT
SM
VL
VM
XA

The letter *m* represents a mnemonic, such as JAZ, which is signified by one, two, or three alphabetic characters, A-Z, except for those character strings represented by *c* and SB.

Combinations

w:	*y*:	*z*:
null	*r*	*r*
r	*x*	*x*
x		VM

i:	*j*:	*k*:	*l*:
r	< *y*	&*z*	< *y*
c	> *y*	*z*	> *y*
< *y*	&*z*	!*z*	
> *y*	*z*		
#< *y*	!*z*		
#> *y*	+*y*		
#*r*	-*y*		
#VM	**r*		
+*r*	/*r*		
-*r*	+F*e*		
+F*d*	-F*e*		
-F*d*	**Fe*		
/H*e*	**He*		
P*e*	**Re*		
Z*s*	**Ie*		
ZSB	/F*e*		
	/R*e*		

f:	*g*:
null	null
x	*y*
i	*yl*
ij	
ijk	

Exceptions

1. An expression *x*, occurring in a string *f* or *g*, must terminate the string.

2. The form *f,m* in the operand field is allowed only if the result field is VM.

Examples of prototype statements (examples of the form *w,w f,g*):

Location	Result	Operand	Comment
1	10	20	35
	@1,A.AREG1	,A.AREG2	Formal parameters @1,AREG1 and AREG2
	,A.REG	@X,S.SREG	Formal parameters REG, @X, SREG

LOCAL - Specify local symbols

The LOCAL pseudo instruction specifies symbols that are defined only within the macro or opdef definition. The LOCAL pseudo instruction also defines any of the named symbols used within an inner definition or call that are not defined as local to that inner usage.

On each macro/opdef call and each repetition of a dup or echo definition sequence, the assembler creates a unique symbol for each local parameter and substitutes the created symbol for the local parameter on each occurrence within the definition. The symbol created for local parameters has the form *%%nnnnnn*, where *n* is an octal digit.

A symbol not defined as local in a definition can be referenced outside an assembly of the definition sequence.

One or more LOCAL pseudo instructions can appear in a macro, opdef, dup, or echo definition. The LOCAL pseudo instructions must immediately follow the macro or opdef prototype statement or DUP or ECHO pseudo instructions, except for intervening comment statements.

Format:

Location	Result	Operand
ignored	LOCAL	$sym_1, sym_2, \ldots, sym_n$

sym_i Symbols that are to be rendered local to the definition

ENDM - End macro or opdef definition

The body of a macro or opdef definition is terminated by an ENDM pseudo instruction.

Format:

Location	Result	Operand
name	ENDM	

name Name of a macro or opdef definition sequence. The name must match the name appearing in the result field of the macro prototype or the location field name in an OPDEF instruction.

Opdef calls

An opdef definition can be called by an instruction with the same result and operand field syntax as specified in the opdef prototype statement.

Format:

Location	Result	Operand
sym	*synr*	*syno*

sym Location field argument, must be a valid symbol. If a location field parameter is specified in the opdef definition, this symbol is optional. *sym* is substituted wherever the location field parameter occurs in the definition.

If no location field parameter is specified in the opdef definition, this field must be empty.

synr Result field, consisting of one, two, or three subfields with the same syntax as specified in the opdef prototype statement

syno Operand field, consisting of none, one, two, or three subfields with the same syntax as specified in the opdef prototype statement

The character strings *synr* and *syno* must be exactly as specified in the opdef definition, except where an expression *x* or a register *r* is indicated.

An expression must appear wherever an expression *x* is indicated in the prototype statement. The actual argument string consists of the characters in the expression up to the terminating space or comma. The actual argument string is substituted in the definition sequence wherever the corresponding formal parameter *x* occurs.

An A, B, SB,† S, T, ST,† SM,† or V register designator must appear wherever the character string A.*sym*, B.*sym*, SB.*sym*,† S.*sym*, T.*sym*, ST.*sym*,† SM.*sym*†, or V.*sym*, respectively, appeared in the prototype statement.

If the register designator is of the form A*n*, B*n*, SB*n*,† S*n*, T*n*, ST*n*,† SM*n*,† or V*n*, then the actual argument string is *n*, where *n* is a single octal digit or two octal digits (B, T, and SM† only). If the register designator is of the form A.*x*, B.*x*, SB.*x*,† S.*x*, T.*x*, ST.*x*,† SM.*x*,† or V.*x*, where *x* is a symbol or numeric constant, then the actual argument string is *x*.

Examples of macro and opdef definitions and calls

Example 1. Macro with positional parameters.

Macro definition:

Location	Result	Operand	Comment
1	10	20	35
	MACRO		
LBL	READ	DN,UDA,CT	
LBL	A1	DN+1	
	A2	UDA	
	A3	CT	
	R	$RWDR	
	EXT	$RWDR	
READ	ENDM		

Macro call and expansion. The second argument in the call is an embedded argument. The expansion starts on line two.

Location	Result	Operand	Comment
1	10	20	35
	READ	FILE,(BUF,0),3	
A1		FILE+1	
	A2	BUF,0	
	A3	3	
	R	$RWDR	
	EXT	$RWDR	

† For CRAY X-MP Computer Systems only.

Note that the location field parameter was omitted on the macro call. The result and operand fields of the first line of the expansion were shifted left three character positions because a null argument was substituted for the 3-character parameter, LBL.

If only one space appeared between the location field parameter and result field in the macro definition and if a null argument were substituted for the location parameter, the result field would be shifted into the location field in column 2. Therefore, at least two spaces should always appear between a parameter in the location field and the first character in the result field in a definition.

Example 2. Macro with positional and keyword parameters.

Macro definition:

```
Location  Result  Operand                    Comment
1         10      20                         35

          MACRO
          TABLE   TABN,VAL1=#0,VAL2=,VAL3=0
          BLOCK   TABLES
TABN      CON     'TABN'L
          CON     VAL1
          CON     VAL2
          CON     VAL3
          BLOCK   *                          resume use of previous block
TABLE     ENDM
```

Macro call and expansion. The expansion starts on line 2.

```
Location  Result  Operand                    Comment
1         10      20                         35

          TABLE   TABA,VAL3=4,VAL2=A
          BLOCK   TABLES
TABA      CON     'TABA'L
          CON      #0
          CON      A
          CON      4
          BLOCK   *                          resume use of previous block
```

Example 3. Opdef illustrating a scalar floating-point divide sequence.

Opdef definition:

```
Location  Result    Operand          Comment
1         10        20               35

FDV       OPDEF                      Scalar floating-point
L         S.R1      S.R2/FS.R3       divide prototype statement
          ERRIF     R1,EQ,R2
          ERRIF     R1,EQ,R3
L         S.R1      /HS.R3
          S.R2      S.R2*FS.R1
          S.R3      S.R3*IS.R1
          S.R1      S.R2*FS.R3
FDV       ENDM
```

Opdef call and expansion. The expansion starts on line 2.

```
Location  Result    Operand          Comment
1         10        20               35

A         S4        S3/FS2           Divide S3 by S2, result to S4
          ERRIF     4,EQ,3
          ERRIF     4,EQ,2
A         S.4       /HS.2
          S.3       S.3*FS.4
          S.2       S.2*IS.4
          S.4       S.3*FS.2
```

Example 4. Opdef defining a conditional jump where a jump occurs if the A register values are equal.

Opdef definition:

```
Location  Result    Operand          Comment
1         10        20               35

JEQ       OPDEF
L         JEQ       A.A1,A.A2,@TAG
          AO        A_A1-A_A2
          JAZ       @TAG
JEQ       ENDM
```

Opdef call and expansion. The expansion starts on line 2.

Location	Result	Operand	Comment
1	10	20	35
	JEQ	A3,A6,GO	
	AO	A3-A5	
	JAZ	GO	

OPSYN - Synonymous operation

The OPSYN pseudo instruction defines or redefines a name in the location field as being the same as the named operation in the operand field. A previous definition with a name matching the location field name is no longer available. Any pseudo instruction or macro can be redefined in this manner.

An operation defined by OPSYN is global if the OPSYN pseudo occurs before the IDENT pseudo that begins a program module, and it is local if the OPSYN pseudo appears with an IDENT, END sequence. Global operations can be referenced in any program module following the definition. Every local operation is removed at the end of a program module, making any previous global definition with the same name available again.

Format:

Location	Result	Operand
name$_1$	OPSYN	*name*$_2$

name$_1$ A valid name or the name of a defined operation such as a pseudo instruction or macro. *name*$_1$ must not be blank.

name$_2$ The name of a defined operation. If *name*$_2$ is blank, then *name*$_1$ becomes a do-nothing pseudo instruction.

Example:

OPSYN with a macro definition to redefine the pseudo instruction IDENT.

OPSYN definition:

```
Location  Result   Operand      Comment
1         10       20           35

IDENTT    OPSYN    IDENT
          MACRO
          IDENT    NAME
          LIST     OFF,NXRF
NAME      LIST     ON,XRF       Processed if LIST=NAME on CAL
                                statement
          IDENTT   NAME
IDENT     ENDM
```

OPSYN call and expansion. The expansion starts on line 2.

```
Location  Result   Operand      Comment
1         10       20           35

          IDENT    A
          LIST     OFF,NXRF
A         LIST     ON,XRF
          IDENTT   A
```

CODE DUPLICATION

CAL provides a set of four instructions (DUP, ECHO, ENDDUP, and STOPDUP) that allow multiple assemblies of sequences of source statements.

DUP - Duplicate code

The DUP pseudo instruction introduces the definition of a sequence of code that is assembled repetitively immediately following the definition. The dup sequence is assembled the number of times specified on the DUP pseudo instruction. The dup sequence to be repeated consists of statements following the DUP pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The dup sequence ends when the statement count is exhausted or when ENDDUP with a matching location field name is encountered.

A nested inner dup definition must be entirely contained in the outer definition.

STOPDUP can be used to override the repetition count.

Formats:

Location	Result	Operand
dupname	DUP	*times*
	DUP	*times,count*

dupname Name of the dup sequence; required if the count field is null or missing. *dupname* is used to match an ENDDUP name if no count field is present. *dupname* is also used in the sequence field of the listing for the dup expansion.

times An absolute expression with positive value specifying number of times to repeat the code sequence. If the value is 0, the code is skipped.

count Optional absolute expression with positive value specifying the number of statements to be duplicated. LOCAL pseudo instructions and comment statements (* in column 1) are ignored for the purpose of this count. Statements are counted before expansion of nested macro or opdef calls or dup or echo sequences.

ECHO - Duplicate code with varying arguments

The ECHO pseudo instruction introduces the definition of a sequence of code that is assembled repetitively immediately following the definition. On each repetition, the actual arguments are substituted for the formal parameters until the longest argument list is exhausted. Null characters are substituted for the formal parameters once shorter argument lists are exhausted. The echo sequence to be repeated consists of statements following the ECHO pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The echo sequence ends with an ENDDUP that has a matching location field name.

A nested inner echo definition must be entirely contained in the outer definition.

STOPDUP can be used to override the repetition count determined by the number of arguments in the longest argument list.

Format:

Location	Result	Operand
dupname	ECHO	e_1=$list_1$,e_2=$list_2$,...,e_n=$list_n$

dupname Name of the echo sequence, must not be empty. *dupname* must match the location field name in the ENDDUP instruction that terminates the echo sequence.

e_i Formal parameter name. There can be none, one, or more formal parameters, e_i.

$list_i$ List of actual arguments. The list can be a single argument a_{i1} or a parenthesized list of arguments $a_{i1},a_{i2},\ldots,a_{im}$ where each a_{ij} is an actual argument to be substituted for e_i in the echo sequence. Each actual argument a_{ij} can be an ASCII character string not containing blanks or commas or can itself be an embedded argument containing a list of arguments a_{ij} enclosed in matching parentheses. An embedded argument can contain blanks or commas and matched pairs of parentheses.

The argument a_{i1} is substituted for e_i in the echo sequence on the first repetition, a_{i2} is substituted for e_i on the second repetition, etc.

A comma immediately followed by another comma or closing right parenthesis specifies a null (empty) character string as the argument.

ENDDUP - End duplicated code

The ENDDUP pseudo instruction ends the definition of the code sequence to be repeated. An ENDDUP pseudo instruction terminates a dup or echo sequence with the same name. ENDDUP has no effect on dup or echo sequences terminated by a statement count.

Format:

Location	Result	Operand
dupname	ENDDUP	

dupname Name of a dup sequence

STOPDUP - Stop duplication

The STOPDUP pseudo instruction stops duplication of a code sequence indicated by a DUP or ECHO pseudo instruction. It overrides the repetition count. Assembly of the current repetition of the dup sequence is terminated immediately. STOPDUP terminates the innermost dup or echo sequence with the same name. STOPDUP does not affect the definition of the code sequence to be duplicated.

Format:

Location	Result	Operand
dupname	STOPDUP	

dupname Name of a dup sequence

Examples of duplicated sequences

Example 1. Use DUP pseudo instruction to define an array with values 0,1,2,3.

DUP definition:

Location	Result	Operand	Comment
1	10	20	35
S	=	W.*	
	DUP	3,1	
	CON	W.*-S	

DUP expansion:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
				(writer comment; not generated by CAL)
		CON	W.*-S	(W.*-S=0)
00000000000000000000		CON	W.*-S	(W.*-S=1)
00000000000000000001		CON	W.*-S	(W.*-S=2)
00000000000000000002		CON	W.*-S	(W.*-S=3)

Example 2. Illustration of nested duplication.

ECHO definition:

```
Location  Result   Operand              Comment
1         10       20                   35

ECHO      ECHO     RI=(A,S),RJK=(B,T)
I         SET      0
DUPI      DUP      8
JK        SET      0
DUPJK     DUP      64
          RI.I     RJK.JK
JK        SET      JK+1
DUPJK     ENDDUP
I         SET      I+1
DUPI      ENDDUP
ECHO      ENDDUP
```

ECHO and DUP expansion:

```
Location  Result   Operand              Comment
1         10       20                   35

                                        (writer comment; not generated
                                        by CAL)
          A.I      B.JK                 I=0,JK=0
          A.I      B.JK                 I=0,JK=1
          .        .                    .
          .        .                    .
          .        .                    .
          A.I      B.JK                 I=8,JK=64
          S.I      T.JK                 I=0,JK=0
          S.I      T.JK                 I=0,JK=1
          .        .                    .
          .        .                    .
          .        .                    .
          S.I      T.JK                 I=8,JK=64
```

Example 3. Use STOPDUP to terminate duplication.

STOPDUP definition:

```
Location  Result    Operand     Comment
1         10        20          35

T         SET       0
A         DUP       1000
T         SET       T+1
          IFE       T,EQ,3,1    Terminate duplication when T=3
A         STOPDUP
          CON       T
A         ENDDUP
```

STOPDUP expansion:

```
Location  Result    Operand     Comment
1         10        20          35

T         SET       T+1
          CON       T
T         SET       T+1
          CON       T
T         SET       T+1
A         STOPDUP
```

MICRO DEFINITION

Through the use of micros, a programmer is able to assign a name to a character string and subsequently refer to the character string through use of its name. A reference to a micro results in the character string being substituted for the name before assembly of the source statement containing the reference. Boldface indicates a CAL response.

Micro references

A programmer refers to a micro by using the micro name enclosed by quotation marks anywhere in a source statement other than a comment line. If column 72 of a line is exceeded as a result of a micro substitution, the assembler creates additional continuation lines. No replacement takes place if the micro name is unknown or if one of the micro marks has been omitted.

Example:

A micro named PFX is defined as ID. A reference to PFX is in the location field of a line:

Location	Result	Operand	Comment
1	10	20	35
"PFX"TAG	S0	S1	

However, before the line is interpreted, CAL substitutes the definition for PFX producing the following line:

Location	Result	Operand	Comment
1	10	20	35
IDTAG	S0	S1	

MICRO - Micro definition

The MICRO pseudo instruction assigns a name to a character string.

Formats:

Location	Result	Operand
name	MICRO	'*character string*',exp_1,exp_2
name	MICRO	'*character string*',exp_1
name	MICRO	'*character string*'

name Micro name. If name is previously defined, the previous micro definition is lost.

'*character string*'
A character string, which can include previously defined micros

To specify a single apostrophe in a character string, use two adjacent apostrophes. These are counted as a single character in the string.

A character string can be delimited by a character other than an apostrophe. Any ASCII character other than a comma or space can be used. Two consecutive occurrences of the delimiting character indicate a single such character. For example, a micro consisting of the single character * could be specified as '*' or ****.

exp_1 Absolute expression indicating number of characters in the micro character string.

The micro character string is terminated either by the character count or the final apostrophe of the character string, whichever occurs first. The string is considered empty if exp_1 has a 0 or negative value. exp_1 is considered very large if it is null. In this case the string is terminated by the final apostrophe.

exp_2 Absolute expression indicating starting character. The micro character string is considered to begin with the first character of the character string if exp_2 is null or has the value of 0 or 1 or is negative.

Example:

Location	Result	Operand	Comment
1	10	20	35
MIC	MICRO	'THIS IS A MICRO STRING'	
MIC1	MICRO	****	Micro string is 1 asterisk
MIC2	MICRO	'"MIC"',1	Micro consisting of 1st character of the micro string represented by MIC
MIC2	**MICRO**	**'THIS IS A MICRO STRING',1**	
MIC4	MICRO	'"MIC"',2,2	Micro consisting of 2nd and 3rd characters of micro string represented by MIC
MIC4	**MICRO**	**'THIS IS A MICRO STRING',2,2**	
MIC5	MICRO		Blank operand field defines an empty string

OCTMIC and DECMIC - Octal and decimal micros

These pseudo instructions convert the value of an expression into a character string that is assigned a micro name.

Format:

Location	Result	Operand
name	OCTMIC	*exp,count*
name	DECMIC	*exp,count*

name Micro name

exp An absolute expression to be converted to up to 8 characters representing the octal (or decimal) value

count An expression providing an optional character count less than or equal to 8. If this parameter is present, leading zeros are supplied to provide the requested number of characters.

Example of MICSIZE and DECMIC:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
26	V	MICSIZE	MIC	The value of V is the number of characters in the micro string represented by MIC
2	VOCT	DECMIC	V,2	VOCT is a micro name There are "VOCT" characters in MIC **There are 26 characters in MIC**

Example of OCTMIC:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IP	=	O'20
		VAL	OCTMIC	IP
		MSG	DATA	'THE VALUE OF IP IS "VAL"'
			DATA	**'THE VALUE OF IP IS 20'**

Predefined micros

In addition to the above micros, the CAL assembler provides the following predefined micros.

$DATE Current date '*mm/dd/yy*'

$JDATE Julian date '*yy/ddd*'

$TIME Time of day '*hh:mm:ss*'

$MIC Micro character (ASCII 042)

$CNC Concatenation character (ASCII 137)

$QUAL Name of qualifier in effect; if none, then null string.

$CPU Target machine: 'CRAY 1' or 'CRAY XMP'

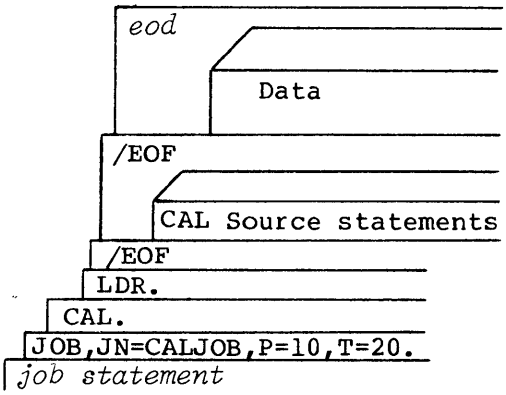
Example: Use of predefined micro $DATE

Location	Result	Operand	Comment
1	10	20	35
		DATA	'THE DATE IS "$DATE"'
		DATA	**'THE DATE IS 06/23/82'**

CAL EXECUTION 5

The CAL assembler is loaded and executed on the Cray mainframe through placement of a CAL control statement in the job control statement file of a Cray Operating System (COS) job deck. Loading and executing of the assembled program is initiated by a call to the loader through an LDR control statement as described in the CRAY-OS Version 1 Reference Manual, publication SR-0011. ASSIGN control statements for datasets used by the assembled program must be placed in the job deck before the LDR control statement.

Sample job deck order:



Default datasets are $IN for source statements, $OUT for list output, and $BLD for binary load data.

The CAL control statement, system text, and binary system text are described in this section.

CAL CONTROL STATEMENT

A CAL control statement has the following format:

CAL,CPU=*type*,I=*idn*,L=*ldn*,B=*bdn*,E=*edn*,ABORT,DEBUG,*options*,LIST=*name*,

S=*sdn*,SYM=*sym*,T=*bst*,X=*xdn*.

Parameters are order independent and are optional. Parameters are processed in the order they appear. In the case of duplication or contradictory specification of parameters, the latter specification is used.

Parameters:

CPU=*type* Cray computer to execute CAL source code. The default is the machine on which CAL is executing. *type* can be CRAY-1 or CRAY-XMP.

I=*idn* Name of dataset containing source statement input. The default is $IN. CAL reads source statements from dataset *idn* until an end-of-file is encountered.

L=*ldn* Name of dataset into which list output is written. The default is $OUT. CAL writes one file of output. If L=0, no listing is written.

B=*bdn* Name of dataset to receive binary load data. The default is $BLD. CAL writes binary load data to this dataset, one record per program module. An end-of-file is not written. If B=0, no binary load data is written.

E=*edn* Name of dataset on which error listing is written. The default is no error listing if the list output is on $OUT, otherwise, the default is $OUT. CAL writes source statements containing errors to this dataset as one file. Specifying E causes an error listing to be generated on a dataset named $OUT. If the error dataset name, *edn*, is the same as the listing dataset name, then list output is written.

ABORT Abort mode. If this parameter is present, CAL aborts the job after assembling all program modules if any fatal errors were encountered during assembly. If this parameter is omitted or if fatal errors were not encountered, CAL exits normally and job processing continues with the next control statement in the job deck.

DEBUG Debug mode. If this parameter is omitted and fatal errors occur in a program, CAL writes a binary record containing only a Program Description Table (PDT) with the Fatal Error flag set. The loader ignores a program module that has this flag set.

When the DEBUG parameter is present, CAL writes a full binary record with the fatal error flag clear, whether or not fatal errors are encountered. The loader will attempt to load and execute the module.

options Listing control options. Any of the following listing control options can be specified to enable or disable a listing feature. The selection of an option on the CAL control statement overrides the enabling or disabling of the corresponding feature on a LIST pseudo instruction. Refer to the description of the LIST pseudo in section 4 for more details about these options.

Defaults are underlined.

ON	Enables source statement listing
OFF	Disables source statement listing
XRF	Enables cross reference
NXRF	Disables cross reference
XNS	Includes non-referenced symbols in cross reference
NXNS	Does not include non-referenced symbols in cross reference
DUP	Enables listing of duplicated statements
NDUP	Disables listing of duplicated statements
LIS	Enables listing of listing control pseudo instructions
NLIS	Disables listing of listing control pseudo instructions
MAC	Enables listing of macro expansions
NMAC	Disables listing of macro expansions
MIC	Enables listing of generated statements before editing
NMIC	Disables listing of generated statements before editing
MIF	Enables macro conditional listing
NMIF	Disables macro conditional listing
WEM	Enables warning errors
NWEM	Disables warning errors
TXT	Enables global text source listing
NTXT	Disables global text source listing
WRP	Enables relocatable parcel-address warning error
NWRP	Disables relocatable parcel-address warning error
WMR	Enables warning error message for macro and opdef redefinitions
NWMR	Disables warning error message for macro and opdef redefinitions

LIST=*name* Name of LIST pseudo instructions to be processed. A LIST pseudo instruction with a matching location field name is not ignored. A LIST pseudo with a nonblank location field name that does not match a name specified on the CAL control statement is ignored. A *name* can be a single name or can be a list of names separated by colons, (for example, LIST=TASK1:TASK2:TASK7). If just LIST is specified, all LIST pseudo instructions are processed, regardless of the location field name.

S=*sdn* Name of dataset containing system text file. The default is $SYSTXT. If S=0 is specified, no system text is used. *sdn* can be a single dataset name or can be a list of up to ten dataset names separated by colons, (for example, S=$SYSTXT:OURTXT:MYTXT). The system texts are processed in the order of appearance. (See description of system text later in this section.)

SYM=*sym* Name of dataset where the optional symbol table is to be written. The default is no symbol table dataset generated by CAL. If SYM is specified without a value, the symbol text is written to the same dataset as the binary load data.

T=*bst* Binary system text. Specifies dataset name to which all global macros, opdefs, symbols, and OPSYN assignments are written. The default, equivalent to specifying T=0, is no binary system text written. If T is specified without a value, the binary dataset is written to $BST.

X=*xdn* Binary symbol table for the global cross reference generator, SYSREF. Each record contains cross reference information for the global symbols in one particular program unit. The default, equivalent to specifying X=0, is to write no global cross reference records. If X is specified without a value, the information is written to $XRF. (See description of binary system text later in this section.)

Example of CAL statement:

```
CAL,I=$IN,E,ABORT.
```

This CAL statement specifies that source statements are on $IN, errors are written on $OUT, list output is suppressed, binary load data is written on $BLD, the system text is on $SYSTXT, and no binary system text is written. The job aborts if fatal errors are encountered.

NOTE

Input datasets and system text datasets (such as $SYSTEXT) that are permanent datasets having the same names as the local datasets need not be accessed (via the ACCESS control statement) before calling CAL. Note also that the IDs for these datasets must be null.

SYSTEM TEXT

System text allows for definition of global macros, opdefs, micros, and symbols that are commonly used. These macros, opdefs, micros, and symbols are defined in a system text that is separate from the user's source statement input and that is assembled before the user's source. All global definitions contained in the system text are preserved for reference in the user's programs.

System text symbols referenced by the user are identified in the cross reference listing by the system text dataset name.

System text can contain any CAL statements that are allowed in normal source input. Typically however, a system text consists of macro, opdef, micro, and symbol definitions followed by an IDENT and END pseudo. While assembling system text, CAL suppresses writing binary load data and list output, except for statements that contain errors.

An IDENT and an END pseudo are not required at the end of a system text, but, if present, facilitate assembling the system text separately as a program module for the purpose of obtaining a listing.

BINARY SYSTEM TEXT

A binary system text is a preassembled version of a source system text. A binary system text is generated as a result of the presence of the T option on the CAL control statement. When T is specified, all global macros, opdefs, symbols, and OPSYN assignments are written to the specified dataset in an internal CAL format.

The specified dataset can thereafter be used with the S option, as if using the source system text. CAL determines whether a system text is in source or in binary format. When multiple system texts are used, binary and source versions can be mixed. The effect is as if all of the source versions were present.

NOTE

Use of binary system text generally reduces assembly time.

Examples:

1. CAL,I=SOURCE1,S=0,T=BINARY1.

2. CAL,I=SOURCE3,S=0,T=BINARY3.

3. CAL,I=MYPROG,S=BINARY1:SOURCE2:BINARY3.

In examples 1 and 2, binary versions of source system texts SOURCE1 and SOURCE3 are created. If S=0 had not been specified, CAL would have assembled $SYSTXT by default. The global macros, opdefs, and symbols in $SYSTXT would have been copied into the binary system texts being generated.

In example 3, the binary texts generated by examples 1 and 2 are used. The effect is as if the following statement had been written instead of example 3:

CAL,I=MYPROG,S=SOURCE1:SOURCE2:SOURCE3.

APPENDIX SECTION

INSTRUCTION SUMMARIES A

This appendix includes an instruction summary for CRAY-1 mainframes (Models A and B, CRAY-1 S Series, and CRAY-1 M Series) and an instruction summary for CRAY X-MP mainframes.

INSTRUCTION SUMMARY FOR CRAY-1 COMPUTERS

CRAY-1	CAL	Page	Unit	Description
000*xxx*	ERR	3-95	-	Error exit
†000*ijk*	ERR *exp*	3-95	-	Error exit
††0010*jk*	CA,A*j* A*k*	3-97	-	Set the channel (A*j*) current address to (A*k*) and begin the I/O sequence
††0011*jk*	CL,A*j* A*k*	3-97	-	Set the channel (A*j*) limit address to (A*k*)
††0012*jx*	CI,A*j*	3-98	-	Clear channel (A*j*) interrupt flag
††0013*jx*	XA A*j*	3-99	-	Enter XA register with (A*j*)
††0014*j*0	RT S*j*	3-100	-	Enter RTC register with (S*j*)
††§0014*j*4	PCI S*j*	3-100	-	Enter interval register with (S*j*)
††§0014*x*5	CCI	3-101	-	Clear PCI request
††§0014*x*6	ECI	3-101	-	Enable PCI request
††§0014*x*7	DCI	3-102	-	Disable PCI request

† Special syntax form
†† Privileged to monitor mode
§ Programmable clock (optional on CRAY-1 Models A and B)

CRAY-1	CAL	Page	Unit	Description
0020*xk*	VL A*k*	3-33	-	Transmit (A*k*) to VL register
†0020*x*0	VL 1	3-33	-	Transmit 1 to VL register
0021*xx*	EFI	3-53	-	Enable interrupt on floating-point error
0022*xx*	DFI	3-53	-	Disable interrupt on floating-point error
003*xjx*	VM S*j*	3-33	-	Transmit (S*j*) to VM register
†003*x*0*x*	VM 0	3-33	-	Clear VM register
004*xxx*	EX	3-95	-	Normal exit
†004*ijk*	EX *exp*	3-95	-	Normal exit
005*xjk*	J B*jk*	3-92	-	Jump to (B*jk*)
006*ijkm*	J *exp*	3-92	-	Jump to *exp*
007*ijkm*	R *exp*	3-94	-	Return jump to *exp*; set B00 to P.
010*ijkm*	JAZ *exp*	3-93	-	Branch to *exp* if (A0)=0
011*ijkm*	JAN *exp*	3-93	-	Branch to *exp* if (A0)≠0
012*ijkm*	JAP *exp*	3-93	-	Branch to *exp* if (A0) ≥0
013*ijkm*	JAM *exp*	3-93	-	Branch to *exp* if (A0) <0
014*ijkm*	JSZ *exp*	3-93	-	Branch to *exp* if (S0)=0
015*ijkm*	JSN *exp*	3-93	-	Branch to *exp* if (S0)≠0
016*ijkm*	JSP *exp*	3-93	-	Branch to *exp* if (S0) ≥0
017*ijkm*	JSM *exp*	3-93	-	Branch to *exp* if (S0) <0
†††020*ijkm*	A*i* *exp*	3-9	-	Transmit *exp*=*jkm* to A*i*

† Special syntax form

††† Instruction is generated depending on the value of the expression as described in section 3.

CRAY-1	CAL	Page	Unit	Description
†††021*ijkm*	A*i* *exp*	3-9	-	Transmit *exp*=ones complement of *jkm* to A*i*
†††022*ijk*	A*i* *exp*	3-9	-	Transmit *exp*=*jk* to A*i*
023*ijx*	A*i* S*j*	3-19	-	Transmit (S*j*) to A*i*
024*ijk*	A*i* B*jk*	3-20	-	Transmit (B*jk*) to A*i*
025*ijk*	B*jk* A*i*	3-29	-	Transmit (A*i*) to B*jk*
026*ij*0	A*i* PS*j*	3-88	Pop/LZ	Population count of (S*j*) to A*i*
§§026*ij*1	A*i* QS*j*	3-89	Pop/LZ	Population count parity of (S*j*) to A*i*
027*ijx*	A*i* ZS*j*	3-91	Pop/LZ	Leading zero count of (S*j*) to A*i*
030*ijk*	A*i* A*j*+A*k*	3-46	A Int Add	Integer sum of (A*j*) and (A*k*) to A*i*
†030*i*0*k*	A*i* A*k*	3-18	A Int Add	Transmit (A*k*) to A*i*
†030*ij*0	A*i* A*j*+1	3-46	A Int Add	Integer sum of (A*j*) and 1 to A*i*
031*ijk*	A*i* A*j*-A*k*	3-46	A Int Add	Integer difference of (A*j*) less (A*k*) to A*i*
† †††031*i*00	A*i* -1	3-9	A Int Add	Transmit -1 to A*i*
†031*i*0*k*	A*i* -A*k*	3-19	A Int Add	Transmit the negative of (A*k*) to A*i*
†031*ij*0	A*i* A*j*-1	3-46	A Int Add	Integer difference of (A*j*) less 1 to A*i*
032*ijk*	A*i* A*j**A*k*	3-47	A Int Mult	Integer product of (A*j*) and (A*k*) to A*i*
033*i*0*x*	A*i* CI	3-21	-	Channel number to A*i* (*j*=0)

† Special syntax form

††† Instruction is generated depending on the value of the expression as described in section 3.

§§ Vector Population Count (optional on CRAY-1 Models A and B)

CRAY-1	CAL	Page	Unit	Description
033*ij*0	A*i* CA,A*j*	3-22	-	Address of channel (A*j*) to A*i* (*j*≠0; *k*=0)
033*ij*1	A*i* CE,A*j*	3-22	-	Error flag of channel (A*j*) to A*i* (*j*≠0; *k*=1)
034*ijk*	B*jk*,A*i* ,A0	3-41	Memory	Read (A*i*) words to B register *jk* from (A0)
†034*ijk*	B*jk*,A*i* 0,A0	3-41	Memory	Read (A*i*) words to B register *jk* from (A0)
035*ijk*	,A0 B*jk*,A*i*	3-37	Memory	Store (A*i*) words at B register *jk* to (A0)
†035*ijk*	0,A0 B*jk*,A*i*	3-37	Memory	Store (A*i*) words at B register *jk* to (A0)
036*ijk*	T*jk*,A*i* ,A0	3-41	Memory	Read (A*i*) words to T register *jk* from (A0)
†036*ijk*	T*jk*,A*i* 0,A0	3-41	Memory	Read (A*i*) words to T register *jk* from (A0)
037*ijk*	,A0 T*jk*,A*i*	3-38	Memory	Store (A*i*) words at T register *jk* to (A0)
†037*ijk*	0,A0 T*jk*,A*i*	3-38	Memory	Store (A*i*) words at T register *jk* to (A0)
040*ijkm*	S*i* *exp*	3-11	-	Transmit *jkm* to S*i*
041*ijkm*	S*i* *exp*	3-11	-	Transmit *exp*=ones complement of *jkm* to S*i*
042*ijk*	S*i* <*exp*	3-13	S Logical	Form ones mask *exp*=64-*jk* bits in S*i* from the right
†042*ijk*	S*i* #>*exp*	3-13	S Logical	Form ones mask *exp*=64-*jk* bits in S*i* from the right
†042*i*77	S*i* 1	3-12	S Logical	Enter 1 into S*i*
†042*i*00	S*i* -1	3-12	S Logical	Enter -1 into S*i*

† Special syntax form

CRAY-1	CAL	Page	Unit	Description
043*ijk*	S*i* >*exp*	3-14	S Logical	Form ones mask *exp*=*jk* bits in S*i* from the left
†043*ijk*	S*i* #<*exp*	3-14	S Logical	Form ones mask *exp*=*jk* bits in S*i* from the left
†043*i*00	S*i* 0	3-12	S Logical	Clear S*i*
044*ijk*	S*i* S*j*&S*k*	3-68	S Logical	Logical product of (S*j*) and (S*k*) to S*i*
†044*ij*0	S*i* S*j*&SB	3-68	S Logical	Sign bit of (S*j*) to S*i*
†044*ij*0	S*i* SB&S*j*	3-68	S Logical	Sign bit of (S*j*) to S*i* (*j*≠0)
045*ijk*	S*i* #S*k*&S*j*	3-70	S Logical	Logical product of (S*j*) and ones complement of (S*k*) to S*i*
†045*ij*0	S*i* #SB&S*j*	3-70	S Logical	(S*j*) with sign bit cleared to S*i*
046*ijk*	S*i* S*j*\S*k*	3-73	S Logical	Logical difference of (S*j*) and (S*k*) to S*i*
†046*ij*0	S*i* S*j*\SB	3-73	S Logical	Toggle sign bit of S*j*, then enter into S*i*
†046*ij*0	S*i* SB\S*j*	3-73	S Logical	Toggle sign bit of S*j*, then enter into S*i* (*j*≠0)
047*ijk*	S*i* #S*j*\S*k*	3-75	S Logical	Logical equivalence of (S*k*) and (S*j*) to S*i*
†047*i*0*k*	S*i* #S*k*	3-24	S Logical	Transmit ones complement of (S*k*) to S*i*
†047*ij*0	S*i* #S*j*\SB	3-75	S Logical	Logical equivalence of (S*j*) and sign bit to S*i*
†047*ij*0	S*i* #SB\S*j*	3-75	S Logical	Logical equivalence of (S*j*) and sign bit to S*i* (*j*≠0)
†047*i*00	S*i* #SB	3-15	S Logical	Enter ones complement of sign bit into S*i*

† Special syntax form

CRAY-1	CAL	Page	Unit	Description
050*ijk*	Si Sj!Si&Sk	3-76	S Logical	Logical product of (S*i*) and (S*k*) complement ORed with logical product of (S*j*) and (S*k*) to S*i*
†050*ij*0	Si Sj!Si&SB	3-76	S Logical	Scalar merge of (S*i*) and sign bit of (S*j*) to S*i*
051*ijk*	Si Sj!Sk	3-70	S Logical	Logical sum of (S*j*) and (S*k*) to S*i*
†051*i*0*k*	Si Sk	3-23	S Logical	Transmit (S*k*) to S*i*
†051*ij*0	Si Sj!SB	3-70	S Logical	Logical sum of (S*j*) and sign bit to S*i*
†051*ij*0	Si SB!Sj	3-70	S Logical	Logical sum of (S*j*) and sign bit to S*i* (*j*≠0)
†051*i*00	Si SB	3-15	S Logical	Enter sign bit into S*i*
052*ijk*	S0 Si<*exp*	3-80	S Shift	Shift (S*i*) left *exp*=*jk* places to S0
053*ijk*	S0 Si>*exp*	3-81	S Shift	Shift (S*i*) right *exp*=64-*jk* places to S0
054*ijk*	Si Si<*exp*	3-82	S Shift	Shift (S*i*) left *exp*=*jk* places
055*ijk*	Si Si>*exp*	3-82	S Shift	Shift (S*i*) right *exp*=64-*jk* places
056*ijk*	Si Si,Sj<Ak	3-83	S Shift	Shift (S*i* and S*j*) left (A*k*) places to S*i*
†056*ij*0	Si Si,Sj<1	3-83	S Shift	Shift (S*i* and S*j*) left one place to S*i*
†056*i*0*k*	Si Si<Ak	3-83	S Shift	Shift (S*i*) left (A*k*) places to S*i*
057*ijk*	Si Sj,Si>Ak	3-84	S Shift	Shift (S*j* and S*i*) right (A*k*) places to S*i*
†057*ij*0	Si Sj,Si>1	3-84	S Shift	Shift (S*j* and S*i*) right one place to S*i*

† Special syntax form

CRAY-1	CAL	Page	Unit	Description
†057*i*0*k*	S*i* S*i*>A*k*	3-84	S Shift	Shift (S*i*) right (A*k*) places to S*i*
060*ijk*	S*i* S*j*+S*k*	3-48	S Int Add	Integer sum of (S*j*) and (S*k*) to S*i*
061*ijk*	S*i* S*j*-S*k*	3-49	S Int Add	Integer difference of (S*j*) and (S*k*) to S*i*
†061*i*0*k*	S*i* -S*k*	3-23	S Int Add	Transmit negative of (S*k*) to S*i*
062*ijk*	S*i* S*j*+FS*k*	3-53	Fp Add	Floating-point sum of (S*j*) and (S*k*) to S*i*
†062*i*0*k*	S*i* +FS*k*	3-53	Fp Add	Normalize (S*k*) to S*i*
063*ijk*	S*i* S*j*-FS*k*	3-55	Fp Add	Floating-point difference of (S*j*) and (S*k*) to S*i*
†063*i*0*k*	S*i* -FS*k*	3-55	Fp Add	Transmit normalized negative of (S*k*) to S*i*
064*ijk*	S*i* S*j**FS*k*	3-57	Fp Mult	Floating-point product of (S*j*) and (S*k*) to S*i*
065*ijk*	S*i* S*j**HS*k*	3-59	Fp Mult	Half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*
066*ijk*	S*i* S*j**RS*k*	3-61	Fp Mult	Full-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*
067*ijk*	S*i* S*j**IS*k*	3-62	Fp Mult	2-floating-point product of (S*j*) and (S*k*) to S*i*
070*ijx*	S*i* /HS*j*	3-64	Fp Rcpl	Floating-point reciprocal approximation of (S*j*) to S*i*
071*i*0*k*	S*i* A*k*	3-24	-	Transmit (A*k*) to S*i* with no sign extension
071*i*1*k*	S*i* +A*k*	3-25	-	Transmit (A*k*) to S*i* with sign extension

† Special syntax form

CRAY-1	CAL	Page	Unit	Description
071*i*2*k*	S*i* +FA*k*	3-25	-	Transmit (A*k*) to S*i* as unnormalized floating-poin number
071*i*3*x*	S*i* 0.6	3-12	-	Transmit constant 0.75*2**48 to S*i*
071*i*4*x*	S*i* 0.4	3-12	-	Transmit constant 0.5 to S*i*
071*i*5*x*	S*i* 1.	3-12	-	Transmit constant 1.0 to S*i*
071*i*6*x*	S*i* 2.	3-12	-	Transmit constant 2.0 to S*i*
071*i*7*x*	S*i* 4.	3-12	-	Transmit constant 4.0 to S*i*
072*ixx*	S*i* RT	3-28	-	Transmit (RTC) to S*i*
073*ixx*	S*i* VM	3-27	-	Transmit (VM) to S*i*
074*ijk*	S*i* T*jk*	3-26	-	Transmit (T*jk*) to S*i*
075*ijk*	T*jk* S*i*	3-30	-	Transmit (S*i*) to T*jk*
076*ijk*	S*i* V*j*,A*k*	3-27	-	Transmit (V*j*, element (A*k*)) to S*i*
077*ijk*	V*i*,A*k* S*j*	3-32	-	Transmit (S*j*) to V*i* element (A*k*)
†077*i*0*k*	V*i*,A*k* 0	3-15	-	Clear V*i* element (A*k*)
10*hijkm*	A*i* *exp*,A*h*	3-42	Memory	Read from ((A*h*)+*exp*) to A*i* (A0=0)
†100*ijkm*	A*i* *exp*,0	3-42	Memory	Read from (*exp*) to A*i*
†100*ijkm*	A*i* *exp*,	3-42	Memory	Read from (*exp*) to A*i*
†10*hi*000	A*i* ,A*h*	3-42	Memory	Read from (A*h*) to A*i*
11*hijkm*	*exp*,A*h* A*i*	3-38	Memory	Store (A*i*) to (A*h*)+*exp* (A0=0)

† Special syntax form

CRAY-1	CAL	Page	Unit	Description
†110*ijkm*	*exp*,0 A*i*	3-38	Memory	Store (A*i*) to *exp*
†110*ijkm*	*exp*, A*i*	3-38	Memory	Store (A*i*) to *exp*
†11*hi*000	,A*h* A*i*	3-38	Memory	Store (A*i*) to (A*h*)
12*hijkm*	S*i* *exp*,A*h*	3-43	Memory	Read from ((A*h*)+*exp*) to S*i* (A0=0)
†120*ijkm*	S*i* *exp*,0	3-43	Memory	Read from (*exp*) to S*i*
†120*ijkm*	S*i* *exp*,	3-43	Memory	Read from (*exp*) to S*i*
†12*hi*000	S*i* ,A*h*	3-43	Memory	Read from (A*h*) to S*i*
13*hijkm*	*exp*,A*h* S*i*	3-39	Memory	Store (S*i*) to (A*h*)+*exp* (A0=0)
†130*ijkm*	*exp*,0 S*i*	3-39	Memory	Store (S*i*) to *exp*
†130*ijkm*	*exp*, S*i*	3-39	Memory	Store (S*i*) to *exp*
†13*hi*000	,A*h* S*i*	3-39	Memory	Store (S*i*) to (A*h*)
140*ijk*	V*i* S*j*&V*k*	3-68	V Logical	Logical products of (S*j*) and (V*k*) to V*i*
141*ijk*	V*i* V*j*&V*k*	3-69	V Logical	Logical products of (V*j*) and (V*k*) to V*i*
142*ijk*	V*i* S*j*!V*k*	3-71	V Logical	Logical sums of (S*j*) and (V*k*) to V*i*
†142*i*0*k*	V*i* V*k*	3-31	V Logical	Transmit (V*k*) to V*i*
143*ijk*	V*i* V*j*!V*k*	3-72	V Logical	Logical sums of (V*j*) and (V*k*) to V*i*
144*ijk*	V*i* S*j*\V*k*	3-73	V Logical	Logical differences of (S*j*) and (V*k*) to V*i*
145*ijk*	V*i* V*j*\V*k*	3-74	V Logical	Logical differences of (V*j*) and (V*k*) to V*i*
†145*iii*	V*i* 0	3-16	V Logical	Clear V*i*

† Special syntax form

CRAY-1	CAL	Page	Unit	Description
146*ijk*	V*i* S*j*!V*k*&VM	3-77	V Logical	Transmit (S*j*) if VM bit=1; (V*k*) if VM bit=0 to V*i*.
†146*i*0*k*	V*i* #VM&V*k*	3-79	V Logical	Vector merge of (V*k*) and 0 to V*i*
147*ijk*	V*i* V*j*!V*k*&VM	3-78	V Logical	Transmit (V*j*) if VM bit=1; (V*k*) if VM bit=0 to V*i*.
150*ijk*	V*i* V*j*<A*k*	3-85	V Shift	Shift (V*j*) left (A*k*) places to V*i*
†150*ij*0	V*i* V*j*<1	3-85	V Shift	Shift (V*j*) left one place to V*i*
151*ijk*	V*i* V*j*>A*k*	3-85	V Shift	Shift (V*j*) right (A*k*) places to V*i*
†151*ij*0	V*i* V*j*>1	3-85	V Shift	Shift (V*j*) right one place to V*i*
152*ijk*	V*i* V*j*,V*j*<A*k*	3-86	V Shift	Double shift (V*j*) left (A*k*) places to V*i*
†152*ij*0	V*i* V*j*,V*j*<1	3-86	V Shift	Double shift (V*j*) left one place to V*i*
153*ijk*	V*i* V*j*,V*j*>A*k*	3-87	V Shift	Double shift (V*j*) right (A*k*) places to V*i*
†153*ij*0	V*i* V*j*,V*j*>1	3-87	V Shift	Double shift (V*j*) right one place to V*i*
154*ijk*	V*i* S*j*+V*k*	3-48	V Int Add	Integer sums of (S*j*) and (V*k*) to V*i*
155*ijk*	V*i* V*j*+V*k*	3-49	V Int Add	Integer sums of (V*j*) and (V*k*) to V*i*
156*ijk*	V*i* S*j*-V*k*	3-50	V Int Add	Integer differences of (S*j*) and (V*k*) to V*i*
†156*i*0*k*	V*i* -V*k*	3-32	V Int Add	Transmit negative of (V*k*) to V*i*
157*ijk*	V*i* V*j*-V*k*	3-50	V Int Add	Integer differences of (V*j*) and (V*k*) to V*i*

† Special syntax form

CRAY-1	CAL	Page	Unit	Description
160*ijk*	V*i* S*j**FV*k*	3-57	Fp Mult	Floating-point products of (S*j*) and (V*k*) to V*i*
161*ijk*	V*i* V*j**FV*k*	3-58	Fp Mult	Floating-point products of (V*j*) and (V*k*) to V*i*
162*ijk*	V*i* S*j**HV*k*	3-59	Fp Mult	Half-precision rounded floating-point products of (S*j*) and (V*k*) to V*i*
163*ijk*	V*i* V*j**HV*k*	3-60	Fp Mult	Half-precision rounded floating-point products of (V*j*) and (V*k*) to V*i*
164*ijk*	V*i* S*j**RV*k*	3-61	Fp Mult	Rounded floating-point products of (S*j*) and (V*k*) to V*i*
165*ijk*	V*i* V*j**RV*k*	3-62	Fp Mult	Rounded floating-point products of (V*j*) and (V*k*) to V*i*
166*ijk*	V*i* S*j**IV*k*	3-63	Fp Mult	2-floating-point products of (S*j*) and (V*k*) to V*i*
167*ijk*	V*i* V*j**IV*k*	3-63	Fp Mult	2-floating-point products of (V*j*) and (V*k*) to V*i*
170*ijk*	V*i* S*j*+FV*k*	3-54	Fp Add	Floating-point sums of (S*j*) and (V*k*) to V*i*
†170*i*0*k*	V*i* +FV*k*	3-54	Fp Add	Normalize (V*k*) to V*i*
171*ijk*	V*i* V*j*+FV*k*	3-54	Fp Add	Floating-point sums of (V*j*) and (V*k*) to V*i*
172*ijk*	V*i* S*j*-FV*k*	3-56	Fp Add	Floating-point differences of (S*j*) and (V*k*) to V*i*
†172*i*0*k*	V*i* -FV*k*	3-56	Fp Add	Transmit normalized negatives of (V*k*) to V*i*
173*ijk*	V*i* V*j*-FV*k*	3-56	Fp Add	Floating-point differences of (V*j*) and (V*k*) to V*i*

† Special syntax form

CRAY-1	CAL	Page	Unit	Description
174*ij*0	V*i* /HV*j*	3-66	Fp Rcpl	Floating-point reciprocal approximations of (V*j*) to V*i*
§§174*ij*1	V*i* PV*j*	3-89	V Pop	Population counts of (V*j*) to V*i*
§§174*ij*2	V*i* QV*j*	3-90	V Pop	Population count parities of (V*j*) to V*i*
175*xj*0	VM V*j*,Z	3-75	V Logical	VM=1 where (V*j*)=0
175*xj*1	VM V*j*,N	3-75	V Logical	VM=1 where (V*j*)≠0
175*xj*2	VM V*j*,P	3-75	V Logical	VM=1 where (V*j*) positive
175*xj*3	VM V*j*,M	3-75	V Logical	VM=1 where (V*j*) negative
176*ixk*	V*i* ,A0,A*k*	3-44	Memory	Read (VL) words to V*i* from (A0) incremented by (A*k*)
†176*ix*0	V*i* ,A0,1	3-44	Memory	Read (VL) words to V*i* from (A0) incremented by 1
177*xjk*	,A0,A*k* V*j*	3-40	Memory	Store (VL) words from V*j* to (A0) incremented by (A*k*)
†177*xj*0	,A0,1 V*j*	3-40	Memory	Store (VL) words from V*j* to (A0) incremented by 1

Legend:

A	Address
Fp	Floating-point
Int	Integer
Pop	Population/Parity
Pop/LZ	Population/Leading Zero
Rcpl	Reciprocal Approximation
S	Scalar
V	Vector

† Special syntax form

§§ Vector Population Count (optional on CRAY-1 Models A and B)

INSTRUCTION SUMMARY FOR CRAY X-MP COMPUTERS

CRAY X-MP	CAL	Page	Unit	Description
000000	ERR	3-95	-	Error exit
††0010*jk*	CA,A*j* A*k*	3-97	-	Set the channel (A*j*) current address to (A*k*) and begin the I/O sequence
††0011*jk*	CL,A*j* A*k*	3-97	-	Set the channel (A*j*) limit address to (A*k*)
††0012*j*0	CI,A*j*	3-98	-	Clear channel (A*j*) interrupt flag; clear device master-clear (output channel)
††0012*j*1	MC,A*j*	3-98	-	Clear channel (A*j*) interrupt flag; set device master-clear (output channel); clear device ready-held (input channel).
††0013*j*0	XA A*j*	3-99	-	Enter XA register with (A*j*)
††0014*j*0	RT S*j*	3-100	-	Enter RTC register with (S*j*)
††001401	IP 1	3-102	-	Set interprocessor interrupt
††001402	IP 0	3-102	-	Clear interprocessor interru
††001403	CLN 0	3-103	-	Enter CLN register with 0
††001413	CLN 1	3-103	-	Enter CLN register with 1
††001423	CLN 2	3-103	-	Enter CLN register with 2
††001433	CLN 3	3-103	-	Enter CLN register with 3
††0014*j*4	PCI S*j*	3-100	-	Enter II register with (S*j*)
††001405	CCI	3-101	-	Clear PCI request
††001406	ECI	3-101	-	Enable PCI request
††001407	DCI	3-102	-	Disable PCI request
00200*k*	VL A*k*	3-33	-	Transmit (A*k*) to VL register

†† Privileged to monitor mode

CRAY X-MP	CAL	Page	Unit	Description
†002000	VL 1	3-33	-	Transmit 1 to VL register
002100	EFI	3-53	-	Enable interrupt on floating-point error
002200	DFI	3-53	-	Disable interrupt on floating-point error
002300	ERI	3-104	-	Enable operand range interrupts
002400	DRI	3-104	-	Disable operand range interrupts
002500	DBM	3-36	-	Disable bidirectional memory transfers
002600	EBM	3-36	-	Enable bidirectional memory transfers
002700	CMR	3-36	-	Complete memory references
0030*j*0	VM S*j*	3-33	-	Transmit (S*j*) to VM register
†003000	VM 0	3-33	-	Clear VM register
0034*jk*	SM*jk* 1,TS	3-16	-	Test & set semaphore *jk* in SM
0036*jk*	SM*jk* 0	3-17	-	Clear semaphore *jk* in SM
0037*jk*	SM*jk* 1	3-17	-	Set semaphore *jk* in SM
004000	EX	3-95	-	Normal exit
0050*jk*	J B*jk*	3-92	-	Jump to (B*jk*)
006*ijkm*	J *exp*	3-92	-	Jump to *exp*
007*ijkm*	R *exp*	3-94	-	Return jump to *exp*; set B00 to P.
010*ijkm*	JAZ *exp*	3-93	-	Branch to *exp* if (A0)=0

† Special syntax form

CRAY X-MP	CAL	Page	Unit	Description
011*ijkm*	JAN *exp*	3-93	-	Branch to *exp* if (A0)≠0
012*ijkm*	JAP *exp*	3-93	-	Branch to *exp* if (A0) positive; 0 is positive.
013*ijkm*	JAM *exp*	3-93	-	Branch to *exp* if (A0) negative
014*ijkm*	JSZ *exp*	3-93	-	Branch to *exp* if (S0)=0
015*ijkm*	JSN *exp*	3-93	-	Branch to *exp* if (S0)≠0
016*ijkm*	JSP *exp*	3-93	-	Branch to *exp* if (S0) positive; 0 is positive.
017*ijkm*	JSM *exp*	3-93	-	Branch to *exp* if (S0) negative
†††020*ijkm*	A*i* *exp*	3-9	-	Transmit *exp*=*jkm* to A*i*
†††021*ijkm*	A*i* *exp*	3-9	-	Transmit *exp*=ones complement of *jkm* to A*i*
†††022*ijk*	A*i* *exp*	3-9	-	Transmit *exp*=*jk* to A*i*
023*ij*0	A*i* S*j*	3-19	-	Transmit (S*j*) to A*i*
023*i*01	A*i* VL	3-20	-	Transmit (VL) to A*i*
024*ijk*	A*i* B*jk*	3-20	-	Transmit (B*jk*) to A*i*
025*ijk*	B*jk* A*i*	3-29	-	Transmit (A*i*) to B*jk*
026*ij*0	A*i* PS*j*	3-88	Pop/LZ	Population count of (S*j*) to A*i*
026*ij*1	A*i* QS*j*	3-89	Pop/LZ	Population count parity of (S*j*) to A*i*
026*ij*7	A*i* SB*j*	3-21	-	Transmit (SB*j*) to A*i*
027*ij*0	A*i* ZS*j*	3-91	Pop/LZ	Leading zero count of (S*j*) to A*i*
027*ij*7	SB*j* A*i*	3-30	-	Transmit (A*i*) to SB*j*

††† Instruction is generated depending on the value of the expression as described in section 3.

CRAY X-MP	CAL	Page	Unit	Description
030*ijk*	A*i* A*j*+A*k*	3-46	A Int Add	Integer sum of (A*j*) and (A*k*)
†030*i*0*k*	A*i* A*k*	3-18	A Int Add	Transmit (A*k*) to A*i*
†030*ij*0	A*i* A*j*+1	3-46	A Int Add	Integer sum of (A*j*) and 1 to
031*ijk*	A*i* A*j*-A*k*	3-46	A Int Add	Integer difference of (A*j*) le (A*k*) to A*i*
† †††031*i*00	A*i* -1	3-9	A Int Add	Transmit -1 toA*i*
†031*i*0*k*	A*i* -A*k*	3-19	A Int Add	Transmit the negative of (A*k*) to A*i*
†031*ij*0	A*i* A*j*-1	3-46	A Int Add	Integer difference of (A*j*) less 1 to A*i*
032*ijk*	A*i* A*j**A*k*	3-47	A Int Mult	Integer product of (A*j*) and (A*k*) to A*i*
033*i*00	A*i* CI	3-21	-	Channel number to A*i* (*j*=0)
033*ij*0	A*i* CA,A*j*	3-22	-	Address of channel (A*j*) to A*i* (*j*≠0; *k*=0)
033*ij*1	A*i* CE,A*j*	3-22	-	Error flag of channel (A*j*) to A*i* (*j*≠0; *k*=1)
034*ijk*	B*jk*,A*i* ,A0	3-41	Memory	Read (A*i*) words to B register *jk* from (A0)
†034*ijk*	B*jk*,A*i* 0,A0	3-41	Memory	Read (A*i*) words to B register *jk* from (A0)
035*ijk*	,A0 B*jk*,A*i*	3-37	Memory	Store (A*i*) words at B register *jk* to (A0)
†035*ijk*	0,A0 B*jk*,A*i*	3-37	Memory	Store (A*i*) words at B register *jk* to (A0)
036*ijk*	T*jk*,A*i* ,A0	3-41	Memory	Read (A*i*) words to T register *jk* from (A0)

† Special syntax form

††† Instruction is generated depending on the value of the expression as described in section 3.

CRAY X-MP	CAL	Page	Unit	Description
†036*ijk*	T*jk*,A*i* 0,A0	3-41	Memory	Read (A*i*) words to T register *jk* from (A0)
037*ijk*	,A0 T*jk*,A*i*	3-38	Memory	Store (A*i*) words at T register *jk* to (A0)
†037*ijk*	0,A0 T*jk*,A*i*	3-38	Memory	Store (A*i*) words at T register *jk* to (A0)
040*ijkm*	S*i* *exp*	3-11	-	Transmit *jkm* to S*i*
041*ijkm*	S*i* *exp*	3-11	-	Transmit *exp*=ones complement of *jkm* to S*i*
042*ijk*	S*i* <*exp*	3-13	S Logical	Form ones mask *exp* bits in S*i* from the right; *jk* field gets 64-*exp*.
†042*ijk*	S*i* #>*exp*	3-13	S Logical	Form zeros mask *exp* bits in S from the left; *jk* field gets 64-*exp*.
†042*i*77	S*i* 1	3-12	S Logical	Enter 1 into S*i*
†042*i*00	S*i* -1	3-12	S Logical	Enter -1 into S*i*
043*ijk*	S*i* >*exp*	3-14	S Logical	Form ones mask *exp* bits in S*i* from the left; *jk* field gets *exp*.
†043*ijk*	S*i* #<*exp*	3-14	S Logical	Form zeros mask *exp* bits in S*i* from the right; *jk* field gets 64-*exp*.
†043*i*00	S*i* 0	3-12	S Logical	Clear S*i*
044*ijk*	S*i* S*j*&S*k*	3-68	S Logical	Logical product of (S*j*) and (S*k*) to S*i*
†044*ij*0	S*i* S*j*&SB	3-68	S Logical	Sign bit of (S*j*) to S*i*
†044*ij*0	S*i* SB&S*j*	3-68	S Logical	Sign bit of (S*j*) to S*i* (*j*≠0)

† Special syntax form

CRAY X-MP	CAL	Page	Unit	Description
045*ijk*	S*i* #S*k*&S*j*	3-70	S Logical	Logical product of (S*j*) and ones complement of (S*k*) to S*i*
†045*ij*0	S*i* #SB&S*j*	3-70	S Logical	(S*j*) with sign bit cleared to S*i*
046*ijk*	S*i* S*j*\S*k*	3-73	S Logical	Logical difference of (S*j*) and (S*k*) to S*i*
†046*ij*0	S*i* S*j*\SB	3-73	S Logical	Toggle sign bit of S*j*, then enter into S*i*
†046*ij*0	S*i* SB\S*j*	3-73	S Logical	Toggle sign bit of S*j*, then enter into S*i* (*j*≠0)
047*ijk*	S*i* #S*j*\S*k*	3-75	S Logical	Logical equivalence of (S*k*) and (S*j*) to S*i*
†047*i*0*k*	S*i* #S*k*	3-24	S Logical	Transmit ones complement of (S*k*) to S*i*
†047*ij*0	S*i* #S*j*\SB	3-75	S Logical	Logical equivalence of (S*j*) and sign bit to S*i*
†047*ij*0	S*i* #SB\S*j*	3-75	S Logical	Logical equivalence of (S*j*) and sign bit to S*i* (*j*≠0)
†047*i*00	S*i* #SB	3-15	S Logical	Enter ones complement of sign bit into S*i*
050*ijk*	S*i* S*j*!S*i*&S*k*	3-76	S Logical	Logical product of (S*i*) and (S*k*) complement ORed with logical product of (S*j*) and (S*k*) to S*i*
†050*ij*0	S*i* S*j*!S*i*&SB	3-76	S Logical	Scalar merge of (S*i*) and sign bit of (S*j*) to S*i*
051*ijk*	S*i* S*j*!S*k*	3-70	S Logical	Logical sum of (S*j*) and (S*k*) to S*i*
†051*i*0*k*	S*i* S*k*	3-23	S Logical	Transmit (S*k*) to S*i*

† Special syntax form

CRAY X-MP	CAL	Page	Unit	Description
†051*ij*0	S*i* S*j*!SB	3-70	S Logical	Logical sum of (S*j*) and sign bit to S*i*
†051*ij*0	S*i* SB!S*j*	3-70	S Logical	Logical sum of (S*j*) and sign bit to S*i* (*j*≠0)
†051*i*00	S*i* SB	3-15	S Logical	Enter sign bit into S*i*
052*ijk*	S0 S*i*<*exp*	3-80	S Shift	Shift (S*i*) left *exp*=*jk* places to S0
053*ijk*	S0 S*i*>*exp*	3-81	S Shift	Shift (S*i*) right *exp*=64-*jk* places to S0
054*ijk*	S*i* S*i*<*exp*	3-82	S Shift	Shift (S*i*) left *exp*=*jk* places
055*ijk*	S*i* S*i*>*exp*	3-82	S Shift	Shift (S*i*) right *exp*=64-*jk* places
056*ijk*	S*i* S*i*,S*j*<A*k*	3-83	S Shift	Shift (S*i* and S*j*) left (A*k*) places to S*i*
†056*ij*0	S*i* S*i*,S*j*<1	3-83	S Shift	Shift (S*i* and S*j*) left one place to S*i*
†056*i*0*k*	S*i* S*i*<A*k*	3-83	S Shift	Shift (S*i*) left (A*k*) places to S*i*
057*ijk*	S*i* S*j*,S*i*>A*k*	3-84	S Shift	Shift (S*j* and S*i*) right (A*k*) places to S*i*
†057*ij*0	S*i* S*j*,S*i*>1	3-84	S Shift	Shift (S*j* and S*i*) right one place to S*i*
†057*i*0*k*	S*i* S*i*>A*k*	3-84	S Shift	Shift (S*i*) right (A*k*) places to S*i*
060*ijk*	S*i* S*j*+S*k*	3-48	S Int Add	Integer sum of (S*j*) and (S*k*) to S*i*
061*ijk*	S*i* S*j*-S*k*	3-49	S Int Add	Integer difference of (S*j*) and (S*k*) to S*i*
†061*i*0*k*	S*i* -S*k*	3-23	S Int Add	Transmit negative of (S*k*) to S*i*

† Special syntax form

CRAY X-MP	CAL		Page	Unit	Description
062*ijk*	S*i*	S*j*+FS*k*	3-53	Fp Add	Floating-point sum of (S*j*) and (S*k*) to S*i*
†062*i*0*k*	S*i*	+FS*k*	3-53	Fp Add	Normalize (S*k*) to S*i*
063*ijk*	S*i*	S*j*-FS*k*	3-55	Fp Add	Floating-point difference of (S*j*) and (S*k*) to S*i*
†063*i*0*k*	S*i*	-FS*k*	3-55	Fp Add	Transmit normalized negative of (S*k*) to S*i*
064*ijk*	S*i*	S*j**FS*k*	3-57	Fp Mult	Floating-point product of (S*j*) and (S*k*) to S*i*
065*ijk*	S*i*	S*j**HS*k*	3-59	Fp Mult	Half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*
066*ijk*	S*i*	S*j**RS*k*	3-61	Fp Mult	Full-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*
067*ijk*	S*i*	S*j**IS*k*	3-62	Fp Mult	2-floating-point product of (S*j*) and (S*k*) to S*i*
070*ij*0	S*i*	/HS*j*	3-64	Fp Rcpl	Floating-point reciprocal approximation of (S*j*) to S*i*
071*i*0*k*	S*i*	A*k*	3-24	-	Transmit (A*k*) to S*i* with no sign extension
071*i*1*k*	S*i*	+A*k*	3-25	-	Transmit (A*k*) to S*i* with sign extension
071*i*2*k*	S*i*	+FA*k*	3-25	-	Transmit (A*k*) to S*i* as unnormalized floating-point number
071*i*30	S*i*	0.6	3-12	-	Transmit constant 0.75*2**48 to S*i*
071*i*40	S*i*	0.4	3-12	-	Transmit constant 0.5 to S*i*
071*i*50	S*i*	1.	3-12	-	Transmit constant 1.0 to S*i*

† Special syntax form

CRAY X-MP	CAL	Page	Unit	Description
071*i*60	S*i* 2.	3-12	-	Transmit constant 2.0 to S*i*
071*i*70	S*i* 4.	3-12	-	Transmit constant 4.0 to S*i*
072*i*00	S*i* RT	3-28	-	Transmit (RTC) to S*i*
072*i*02	S*i* SM	3-28	-	Transmit (SM) to S*i*
072*ij*3	S*i* ST*j*	3-26	-	Transmit (ST*j*) to S*i*
073*i*00	S*i* VM	3-27	-	Transmit (VM) to S*i*
073*ij*1	S*i* SR*j*	3-29	-	Transmit (SR*j*) to S*i* (*j*=0)
073*i*02	SM S*i*	3-35	-	Transmit (S*i*) to SM
073*ij*3	ST*j* S*i*	3-31	-	Transmit (S*i*) to ST*j*
074*ijk*	S*i* T*jk*	3-26	-	Transmit (T*jk*) to S*i*
075*ijk*	T*jk* S*i*	3-30	-	Transmit (S*i*) to T*jk*
076*ijk*	S*i* V*j*,A*k*	3-27	-	Transmit (V*j*, element (A*k*)) to S*i*
077*ijk*	V*i*,A*k* S*j*	3-32	-	Transmit (S*j*) to V*i* element (A*k*)
†077*i*0*k*	V*i*,A*k* 0	3-15	-	Clear V*i* element (A*k*)
10h*ijkm*	A*i* *exp*,A*h*	3-42	Memory	Read from ((A*h*)+*exp*) to A*i* (A0=0)
†100*ijkm*	A*i* *exp*,0	3-42	Memory	Read from (*exp*) to A*i*
†100*ijkm*	A*i* *exp*,	3-42	Memory	Read from (*exp*) to A*i*
†10h*i*00 0	A*i* ,A*h*	3-42	Memory	Read from (A*h*) to A*i*
11h*ijkm*	*exp*,A*h* A*i*	3-38	Memory	Store (A*i*) to (A*h*)+*exp* (A0=0)
†110*ijkm*	*exp*,0 A*i*	3-38	Memory	Store (A*i*) to *exp*
†110*ijkm*	*exp*, A*i*	3-38	Memory	Store (A*i*) to *exp*

† Special syntax form

CRAY X-MP	CAL	Page	Unit	Description
†11*hi*00 0	,A*h* A*i*	3-38	Memory	Store (A*i*) to (A*h*)
12*hijkm*	S*i* *exp*,A*h*	3-43	Memory	Read from ((A*h*)+*exp*) to S*i* (A0=0)
†120*ijkm*	S*i* *exp*,0	3-43	Memory	Read from *exp* to S*i*
†120*ijkm*	S*i* *exp*,	3-43	Memory	Read from *exp* to S*i*
†12*hi*00 0	S*i* ,A*h*	3-43	Memory	Read from (A*h*) to S*i*
13*hijkm*	*exp*,A*h* S*i*	3-39	Memory	Store (S*i*) to (A*h*)+*exp* (A0=0)
†130*ijkm*	*exp*,0 S*i*	3-39	Memory	Store (S*i*) to *exp*
†130*ijkm*	*exp*, S*i*	3-39	Memory	Store (S*i*) to *exp*
†13*hi*00 0	,A*h* S*i*	3-39	Memory	Store (S*i*) to (A*h*)
140*ijk*	V*i* S*j*&V*k*	3-68	V Logical	Logical products of (S*j*) and (V*k*) to V*i*
141*ijk*	V*i* V*j*&V*k*	3-69	V Logical	Logical products of (V*j*) and (V*k*) to V*i*
142*ijk*	V*i* S*j*!V*k*	3-71	V Logical	Logical sums of (S*j*) and (V*k*) to V*i*
†142*i*0*k*	V*i* V*k*	3-31	V Logical	Transmit (V*k*) to V*i*
143*ijk*	V*i* V*j*!V*k*	3-72	V Logical	Logical sums of (V*j*) and (V*k*) to V*i*
144*ijk*	V*i* S*j* V*k*	3-73	V Logical	Logical differences of (S*j*) and (V*k*) to V*i*
145*ijk*	V*i* V*j* V*k*	3-74	V Logical	Logical differences of (V*j*) and (V*k*) to V*i*
†145*iii*	V*i* 0	3-16	V Logical	Clear V*i*
146*ijk*	V*i* S*j*!V*k*&VM	3-77	V Logical	Transmit (S*j*) if VM bit=1; (V*k*) if VM bit=0 to V*i*.
†146*i*0*k*	V*i* #VM&V*k*	3-79	V Logical	Vector merge of (V*k*) and 0 to V*i*

† Special syntax form

CRAY X-MP	CAL	Page	Unit	Description
147*ijk*	V*i* V*j*!V*k*&VM	3-78	V Logical	Transmit (V*j*) if VM bit=1; (V*k*) if VM bit=0 to V*i*.
150*ijk*	V*i* V*j*<A*k*	3-85	V Shift	Shift (V*j*) left (A*k*) places to V*i*
†150*ij*0	V*i* V*j*<1	3-85	V Shift	Shift (V*j*) left one place to V*i*
151*ijk*	V*i* V*j*>A*k*	3-85	V Shift	Shift (V*j*) right (A*k*) places to V*i*
†151*ij*0	V*i* V*j*>1	3-85	V Shift	Shift (V*j*) right one place to V*i*
152*ijk*	V*i* V*j*,V*j*<A*k*	3-86	V Shift	Double shift (V*j*) left (A*k*) places to V*i*
†152*ij*0	V*i* V*j*,V*j*<1	3-86	V Shift	Double shift (V*j*) left one place to V*i*
153*ijk*	V*i* V*j*,V*j*>A*k*	3-87	V Shift	Double shift (V*j*) right (A*k*) places to V*i*
†153*ij*0	V*i* V*j*,V*j*>1	3-87	V Shift	Double Shift (V*j*) right one place to V*i*
154*ijk*	V*i* S*j*+V*k*	3-48	V Int Add	Integer sums of (S*j*) and (V*k*) to V*i*
155*ijk*	V*i* V*j*+V*k*	3-49	V Int Add	Integer sums of (V*j*) and (V*k*) to V*i*
156*ijk*	V*i* S*j*-V*k*	3-50	V Int Add	Integer differences of (S*j*) and (V*k*) to V*i*
†156*i*0*k*	V*i* -V*k*	3-32	V Int Add	Transmit negative of (V*k*) to V*i*
157*ijk*	V*i* V*j*-V*k*	3-50	V Int Add	Integer differences of (V*j*) and (V*k*) to V*i*
160*ijk*	V*i* S*j**FV*k*	3-59	Fp Mult	Floating-point products of (S*j*) and (V*k*) to V*i*

† Special syntax form

CRAY X-MP	CAL	Page	Unit	Description
161*ijk*	V*i* V*j**FV*k*	3-58	Fp Mult	Floating-point products of (V*j*) and (V*k*) to V*i*
162*ijk*	V*i* S*j**HV*k*	3-59	Fp Mult	Half-precision rounded floating-point products of (S*j*) and (V*k*) to V*i*
163*ijk*	V*i* V*j**HV*k*	3-60	Fp Mult	Half-precision rounded floating-point products of (V*j*) and (V*k*) to V*i*
164*ijk*	V*i* S*j**RV*k*	3-61	Fp Mult	Rounded floating-point products of (S*j*) and (V*k*) to V*i*
165*ijk*	V*i* V*j**RV*k*	3-62	Fp Mult	Rounded floating-point products of (V*j*) and (V*k*) to V*i*
166*ijk*	V*i* S*j**IV*k*	3-63	Fp Mult	2-floating-point products of (S*j*) and (V*k*) to V*i*
167*ijk*	V*i* V*j**IV*k*	3-63	Fp Mult	2-floating-point products of (V*j*) and (V*k*) to V*i*
170*ijk*	V*i* S*j*+FV*k*	3-54	Fp Add	Floating-point sums of (S*j*) and (V*k*) to V*i*
†170*i*0*k*	V*i* +FV*k*	3-54	Fp Add	Normalize (V*k*) to V*i*
171*ijk*	V*i* V*j*+FV*k*	3-54	Fp Add	Floating-point sums of (V*j*) and (V*k*) to V*i*
172*ijk*	V*i* S*j*-FV*k*	3-56	Fp Add	Floating-point differences of (S*j*) and (V*k*) to V*i*
†172*i*0*k*	V*i* -FV*k*	3-56	Fp Add	Transmit normalized negatives of (V*k*) to V*i*
173*ijk*	V*i* V*j*-FV*k*	3-56	Fp Add	Floating-point differences of (V*j*) and (V*k*) to V*i*
174*ij*0	V*i* /HV*j*	3-66	Fp Rcpl	Floating-point reciprocal approximations of (V*j*) to V*i*

† Special syntax form

CRAY X-MP	CAL	Page	Unit	Description
174*ij*1	V*i* PV*j*	3-89	V Pop	Population counts of (V*j*) to V*i*
174*ij*2	V*i* QV*j*	3-90	V Pop	Population count parities of (V*j*) to V*i*
1750*j*0	VM V*j*,Z	3-75	V Logical	VM=1 where (V*j*)=0
1750*j*1	VM V*j*,N	3-75	V Logical	VM=1 where (V*j*)≠0
1750*j*2	VM V*j*,P	3-75	V Logical	VM=1 if (V*j*) positive; 0 is positive.
1750*j*3	VM V*j*,M	3-75	V Logical	VM=1 if (V*j*) negative
176*i*0*k*	V*i* ,A0,A*k*	3-44	Memory	Read (VL) words to V*i* from (A0) incremented by (A*k*)
†176*i*00	V*i* ,A0,1	3-44	Memory	Read (VL) words to V*i* from (A0) incremented by 1
1770*jk*	,A0,A*k* V*j*	3-40	Memory	Store (VL) words from V*j* to (A0) incremented by (A*k*)
†1770*j*0	,A0,1 V*j*	3-40	Memory	Store (VL) words from V*j* to (A0) incremented by 1

Legend:

A	Address
Fp	Floating-point
Int	Integer
Pop	Population/Parity
Pop/LZ	Population/Leading Zero
Rcpl	Reciprocal Approximation
S	Scalar
V	Vector

† Special syntax form

PSEUDO INSTRUCTION INDEX B

Name	Definition	Page
ORG	Set *O counter	4-15
QUAL	Qualify symbols	4-8
REP	Loader replication directive	4-35
SET	Set symbol	4-30
SKIP	Unconditionally skip statements	4-41
SPACE	List blank lines	4-26
START	Specify program entry	4-6
STOPDUP	Stop duplication	4-61
SUBTITLE	Specify listing subtitle	4-27
TEXT	Declare beginning of global text source	4-27
TITLE	Specify listing title	4-26
VWD	Variable word definition	4-34

ASSEMBLY ERRORS C

Two types of errors, fatal errors and warning errors, can occur during assembly. Table C-1 lists fatal errors and table C-2 lists warning errors. Fatal errors cause CAL to abort the job unless a DEBUG parameter is present on the CAL control statement. Warning errors have no effect on the assembly process. The error code consists of a single alpha character or an alpha character and a digit.

Table C-1. Fatal assembly errors

Error type	Definition
C	NAME, SYMBOL, CONSTANT OR DATA ITEM ERROR Indicates any of a number of possible errors Examples: • Illegal character, too many characters, or illegal separator in a name, symbol, constant, or data item • Floating-point exponent underflow or overflow • Double-precision floating-point in an expression • Count field in character constant exceeds 800 • Missing right apostrophe in a character string • Parentheses in an embedded parameter not matched properly • Embedded argument not followed by blank or comma
D	DOUBLE DEFINED SYMBOL OR DUPLICATE PARAMETER NAME Examples: • Symbol previously defined; the first definition holds. No error is given if the second definition results in the same value and attributes. • A formal parameter in a definition has the same name as a previously defined parameter. The parameter is ignored.
E	DEFINITION OR CONDITIONAL SEQUENCE ILLEGALLY NESTED

Table C-1. Fatal errors (continued)

Error type	Definition
F	TOO MANY ENTRIES Examples: • Number of local and common blocks exceeds 1024 • Number of common blocks exceeds 125 • Number of external names exceeds 4095 • Number of entry names exceeds 5461 • Location or origin counter word address exceeds 4,194,303
I	INSTRUCTION PLACEMENT ERROR Treats the instruction as a null (blank) pseudo instruction Examples: • COMMON not allowed in an absolute assembly • ABS not allowed after a symbolic machine instruction or restricted pseudo instruction • IDENT not allowed after IDENT without an intervening END • Symbolic machine instruction, restricted pseudo instruction, or literal appears outside an IDENT, END sequence • Illegal instruction in blank common • END pseudo instruction within a macro expansion
L	LOCATION FIELD ERROR Indicates an invalid name in the location field of a pseudo instruction or macro or opdef call or prototype statement
N	RELOCATABLE FIELD ERROR Indicates an error in a relocatable field. More than one main program entry is named in a program module.
O	OPERAND FIELD ERROR Indicates any of a number of possible errors in the operand field

Table C-1. Fatal errors (continued)

Error type	Definition
	Examples: • Error in operand field of a pseudo instruction • Syntax error in operand field of a symbolic machine instruction
P	PROGRAMMER ERROR Indicates error generated by ERROR or ERRIF pseudo instruction
R	RESULT FIELD ERROR Indicates a syntax error in result field of a symbolic machine instruction
S	SYNTAX ERROR Indicates a syntax error in an instruction Examples: • Undefined pseudo instruction or symbolic machine instruction • A or S register not S0 when required • Registers not the same when required
T	TYPE ERROR Word address, parcel address, or value type not as required for an expression or constant
U	UNDEFINED SYMBOL OR OPERATION Examples: • Reference to a symbol that is not defined • Undefined operation in operand field of OPSYN pseudo instruction
V	REGISTER EXPRESSION OR FIELD WIDTH ERROR Indicates an inconsistency between an expression attribute and field width defined

Table C-1. Fatal errors (continued)

Error type	Definition
	Examples: • Relocatable attribute not allowed for field width or register expression • External attribute not allowed for field width or A, S, or V register expression • Word-address or parcel-address attribute not allowed for field width or register expression • Field width symbol or constant (in VWD) not terminated by a slash (/).
X	EXPRESSION ERROR Expression contains illegal attribute, separator value, etc., for application. Examples: • Expression element not terminated by space, comma, or expression operator • Complement (#) of external or relocatable element not allowed • Negative expression value in BSS, BSSZ, ORG, or LOC pseudo instruction • Expression in ORG not relative to current block • Shift or mask count exceeds 64 or is negative in symbolic machine instruction • Expression relocatable or external when relocatable or external attribute is not allowed • More than one element in a term is external or relocatable, or external element is not the only element in a term. • More than one external element in an expression, or minus sign precedes an external element. • Expression relocatable relative to more than one block after cancellation of relocatable terms with opposite signs • Expression is both external and relocatable.

Table C-2. Warning assembly errors

Error type	Definition
W	PROGRAMMER WARNING ERROR Error can be generated by the ERROR or ERRIF pseudo instruction.
W1	LOCATION FIELD SYMBOL IGNORED Location symbol not used in a pseudo instruction and is ignored
W2	BAD LOCATION SYMBOL Examples: • Illegal character or too many characters • Symbol is the same as a register designator with or without a special prefix (that is, FV*n* or ZS*n*) or the same as a sign bit designator (SB). Refer to SYMBOLS, section 2 for details.
W3	EXPRESSION ELEMENT TYPE ERROR Value, parcel-address, or word-address attribute not allowed for an element in an expression
W4	POSSIBLE SYMBOLIC MACHINE INSTRUCTION ERROR Register usage is unconventional. Examples: • V*i*=V*j* or V*i*=V*k* constituting a recursive vector operation† • A0 used for length and address register in instructions 034 through 037

† Assembling for a CRAY-1 Computer System only

Table C-2. Warning errors (continued)

Error type	Definition
W5	TRUNCATION ERROR Examples: • Expression value exceeds field size, result truncated • Relocatable expression with parcel-address attribute appears in the 22-bit field of instruction 020, 021, 040, or 041.† • Division by zero (zero result) • External expression in zero-width field • Value of register symbol or constant exceeds 7 for A, S, or V, or 77 for B or T • Double-precision floating-point constant in an expression. The result is truncated to one 64-bit word.
W6	LOCATION FIELD SYMBOL NOT DEFINED Examples: • Illegal character or too many characters • Expression defining the symbol contains an undefined symbol • Micro name on a MICSIZE instruction is not previously defined
W7	MICRO SUBSTITUTION ERROR A quotation mark encountered in CAL source was not followed by a previously defined micro name or was not terminated by a second quotation mark.
W8	ADDRESS COUNTER BOUNDARY ERROR Examples: • * (or *0) used in an expression when the location (or origin) counter is not a parcel boundary • W.* (or W. *0) used in an expression when the location (or origin) counter is not a word boundary

† Warning error depends on the WRP and NWRP features of the CAL control statement or the LIST pseudo instruction

Table C-2. Warning errors (continued)

Error type	Definition
Y1	EXTERNAL DECLARATION ERROR EXT not allowed in an absolute assembly; statement ignored.
Y2	MACRO OR OPDEF REDEFINED† A macro name has been previously defined, or syntax has been previously defined for this opdef.

† Warning error depends on the WMR and NWMR features of the CAL control statement or the LIST pseudo instruction

LOGFILE MESSAGES D

CAL supports four classes of logfile messages: abort, fatal, informative, and warning. Each class is defined below.

Abort: CAL aborts

Fatal:

ABORT option	DEBUG option	Result
off	off	PDT fatal error flag set
off	on	PDT fatal error flag clear
on	off	CAL aborts when assembly of all modules in the current input is complete
on	on	CAL aborts when assembly of all modules in the current input is complete

Warning: Possible error detected, no action taken.

Informative: Informative message

The following messages to the job and system logfiles are issued by CAL.

CA000 - [CAL] INTERNAL 'CAL' ERROR DETECTED AT P = *paddress*

CLASS: Abort (immediate)

CAUSE: CAL has detected an internal error at parcel address *paddress* and is unable to proceed.

ACTION: Refer the problem to a Cray Research analyst.

CA001 - [CAL] CAL VERSION *x.xx* (*mm/dd/yy*) - *cpu*

CLASS: Informative

CAUSE: A message issued at the beginning of each assembly indicating the version number *x.xx*, the date *mm/dd/yy* CAL was assembled, and the type of Cray computer, *cpu*, for which CAL is targeting the generated code.

ACTION: Not applicable

CA002 - [CAL] ASSEMBLY TIME: *nnnnn.nnnn* CPU SECONDS

CLASS: Informative

CAUSE: All programs in the current file of the source dataset have been assembled. *nnnnn.nnnn* is the assembly time in floating-point CPU seconds.

ACTION: Not applicable

CA003 - [CAL] MEMORY WORDS: *mwords* + I/O BUFFERS: *iobuffers*

CLASS: Informative

CAUSE: All programs in the current file of the source dataset have been assembled. *mwords* is the decimal number of memory words required in the user portion of the job field. *iobuffers* is the decimal number of words needed for the I/O table and buffer area of this job field.

ACTION: Not applicable

CA004 - [CAL] ASSEMBLY ERRORS

CLASS: Abort

CAUSE: User set the ABORT flag on the CAL control statement and CAL encountered fatal errors during assembly.

ACTION: Either remove the ABORT flag from the CAL control statement or correct all fatal errors found by CAL.

CA010 - [CAL] 1 WARNING ERROR, PROGRAM MODULE *pname*
or
CA010 - [CAL] *n* WARNING ERRORS, PROGRAM MODULE *pname*

CLASS: Warning

CAUSE: CAL issues this message for all source lines from the previous program module (if any) through program module *pname*, in which warning errors have been detected. *pname* is equivalent to the name used on a particular IDENT pseudo statement.

ACTION: Correct all warning errors. See Appendix C for a list of warning errors.

CA011 - [CAL] 1 FATAL ERROR, PROGRAM MODULE *pname*
or
CA011 - [CAL] *n* FATAL ERRORS, PROGRAM MODULE *pname*

CLASS: Fatal

CAUSE: CAL issues this message for all source lines from the previous program module (if any) through program module *pname*, in which fatal errors have been detected. *pname* is equivalent to the name used on a particular IDENT pseudo statement.

ACTION: Correct all fatal errors. See Appendix C for a list of fatal errors.

CA012 = [CAL] MISSING IDENT STATEMENT

CLASS: Warning

CAUSE: An END pseudo on the source dataset occurred before an IDENT pseudo instruction.

ACTION: Check the source dataset for matching IDENT and END pseudo instructions.

CA013 - [CAL] MISSING END STATEMENT, PROGRAM MODULE *pname*

CLASS: Warning

CAUSE: On the source dataset, an end-of-file occurred before an END pseudo instruction corresponding to the IDENT pseudo in program module *pname*. *pname* is equivalent to the name used on that IDENT pseudo statement.

ACTION: Check the source dataset for matching IDENT and END pseudo instructions.

CA014 - [CAL] EMPTY SOURCE FILE, DN = *dname*

CLASS: Warning

CAUSE: An end-of-file or end-of-data was encountered on the source dataset before any source statements.

ACTION: Check the job control statements and the source dataset for a problem that causes a null file.

CA015 - [CAL] 1 LINE EXCEEDS 90 CHARACTERS, DN = *dname*
or
CA015 - [CAL] *n* LINES EXCEED 90 CHARACTERS, DN = *dname*

CLASS: Warning

CAUSE: The given number of records in the named dataset contain more than 90 characters. The most typical cause is UPDATE sequence numbers that extend past column 90. (CAL truncates the long records to 90 characters). This message may also be issued when a binary dataset is erroneously read.

ACTION: If the records exceed 90 characters, break up the long records with continuation lines.

CA016 - [CAL] OPEN ERROR, DN = *dname*

CLASS: Abort

CAUSE: The dataset *dname* was not found in the user's local environment or in the system directory.

ACTION: Access or create the dataset *dname*.

CA017 - [CAL] INVALID CPU TYPE SPECIFIED: *cpu*

CLASS: Warning

CAUSE: An invalid CPU parameter *cpu* was passed on the CAL control statement. CAL defaults to the mainframe on which it is currently executing.

ACTION: Correct the CPU type on the CAL job control statement.

CA030 - [CAL] BAD BINARY TEXT, DN = *dname*, (ERROR CODE = *cc*)

CLASS: Fatal

CAUSE: An error (see below) was discovered in the binary system text *dname*.

Error code	Meaning
P1	Prologue field BSTTT≠1
P2	Prologue field BSTWC less than LE@BSTPR
P3	EOR encountered while prologue was being read
P4	EOF, EOD, or null record encountered while prologue was being read
H1	EOF, EOD, or null record encountered while subtable header was being read
H2	Header field BSTTT≠1
H3	Header field BSTWC less than 1
H4	Header field BSTID not recognized
M1	EOR encountered while TMDF was being read
M2	EOF, EOD, or null record encountered while TMDF was being read
M3	Length of TMDF entry less than 0
M4	Length of TMDF entry=0
M5	Global word count exceeded during TMDF processing
S1	EOR encountered while TSYM entry was being read
S2	EOR, EOD, or null record encountered while TSYM entry was being read
S3	Global word count exceeded during TSYM processing
E1	Epilogue field BSTWC≠1
E2	Global word count not equal to sum of subtable word counts

ACTION: Generate a new binary system text from the original source system text and rerun the job with the new binary system text; or rerun the job with the source system text in place of the binary system text; or show listing and DSDUMP output of offending binary system text to a Cray Research analyst.

CA031 - [CAL] *symbol* DOUBLY-DEFINED IN BINARY TEXT *dname*

CLASS: Fatal

CAUSE: The named *symbol* is defined in the named binary system text but was defined differently in a previous system text.

ACTION: Remove one of the offending definitions from the source system texts, generate a new binary system text, and resubmit job.

CA032 - [CAL] MACRO *opsyn* NOT FOUND, BINARY TEXT *dname*

CLASS: Fatal

CAUSE: The named binary system text contains an OPSYN pseudo instruction of the form name OPSYN *opsyn*, but no macro or pseudo-op with the name *opsyn* is known to the assembler.

ACTION: Correct the spelling of opsyn; or remove the OPSYN pseudo instruction from the named system text; or define the offending macro in a previous system text or before the OPSYN directive in the named system text.

CA033 - [CAL] MACRO *mname* REDEFINED IN BINARY TEXT *dname*

CLASS: Warning

CAUSE: A definition for the named macro appears in the named dataset, but the macro has been previously defined.

ACTION: The new definition will be used; if not intentional, remove the unwanted macro definition.

CA0034 - [CAL] OPDEF *oname* REDEFINED IN BINARY TEXT *dname*

CLASS: Warning

CAUSE: A definition for the named opdef appears in the named dataset, but the opdef's syntax has been previously defined.

ACTION: The new definition will be used; if not intentional, remove the unwanted opdef definition.

FORMAT OF ASSEMBLER LISTINGS E

The CAL assembler generates a source statement listing and a cross reference listing as determined by list pseudo instructions and by options on the CAL control statement as discussed in section 4 and section 5.

Every page of list output produced by the CAL assembler contains two 132-character header lines. The first line contains the title, type of Cray mainframe, version of CAL, date and time of assembly, and a global page number over all programs assembled by the current CAL assembly.

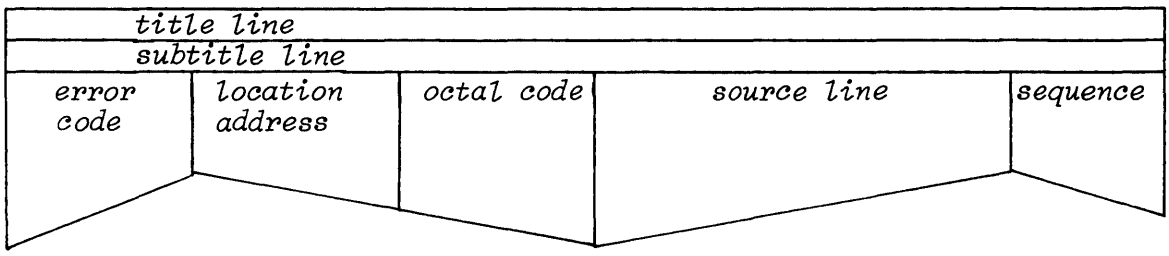
The title is taken from a TITLE pseudo instruction if present or from the operand field of the IDENT pseudo instruction. The second line contains the subtitle specified by a SUBTITLE pseudo if present, a local block name if other than the nominal block, a symbol qualifier if in effect, and a local page number which is reset for each new program until the local page number is used in the cross-reference listings generated by CAL and SYSREF.

Example of page header:

1	66	76	96	105	115
|*title*	|*cpu type*	|CAL *version*	|*date*	|*time*	|Page *n* |
|*subtitle*	|unused	|Block:*bname*	|Quaulifier:*qualname*		| (*n*) |

SOURCE STATEMENT LISTING

The listing for the source statements of a CAL program is organized into five columns of information.



Error codes

The first column contains up to seven characters indicating errors that have been detected for the current statement. If all the errors do not fit in seven columns, the seventh character is a +, indicating that all errors are not shown. Error codes are described in Appendix C.

Location addresses

The second column gives the parcel or word address at which the current statement is assembled. If the statement is a machine instruction, the address is listed as a parcel address with the parcel identifier a, b, c, or d appended to the word address. Parcels are lettered from left to right in the word.

Octal code The third column of information contains the octal equivalent of the instruction or value.

If the instruction or value represents an address, the octal code has a suffix as follows:

+	Positive relocation in program block
-	Negative relocation in program block
C	Common block
X	External symbol
None	Absolute address

For a symbol defined through the SET, MICSIZE, or = pseudo instruction, the column contains the octal value of the symbol.

For a BSS or BSSZ instruction, the column contains the octal value of the number of words reserved.

For an ALIGN instruction, the column contains the octal value of the number of full parcels skipped.

For a MICRO, OCTMIC, or DECMIC instruction, the column contains the octal value of the number of characters in the micro string.

Source line

The fourth column presents columns 1 through 72 of each source line.

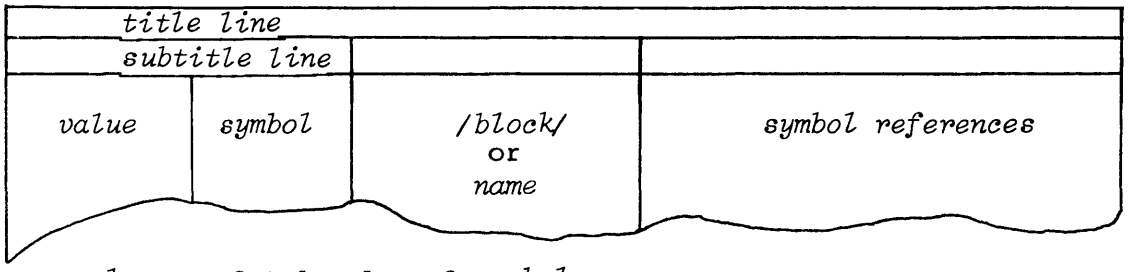
Sequence field

The rightmost columns either contain the information taken from columns 73 through 90 of the source line image or contain an identifier if the line is an expansion of a macro or opdef.

CROSS REFERENCE LISTING

The assembler generates a cross reference table with the following format. Symbols are listed alphabetically and grouped by qualifier. Each qualified group of symbols is headed by the message SYMBOL QUALIFIER IS *qualname*.

Global symbols which are not referenced are not listed in the cross reference. Symbols of the form %%*xxxxxx*, where *x* is any ASCII character, are not listed in the cross reference.



value Octal value of *symbol*

symbol A symbol with parcel-address attribute has a, b, c, or d appended to indicate the parcel in the word. A relocatable symbol has a + suffix if it has positive relocation relative to the program block, a - suffix if negative relocation relative to the program block, and a C suffix if it is relocated relative to a common block. An external symbol has an X suffix. An undefined symbol has a U suffix.

/block/ A relocatable symbol relocated relative to a common block has the common block name enclosed in slant bars. Blank common is indicated by //.

name A global symbol defined by the user is indicated by *GLOBAL*. A global symbol defined in a system text is indicated by the system text dataset name. A symbol defined in text between TEXT and ENDTEXT pseudo instructions is indicated by the associated TEXT name.

symbol references
One or more references to the symbol in the following format:

page : *line x*

page Local decimal number, of page containing reference. The local page number appears in parentheses at the right end of the second title line, which is also called the subtitle line.

line Decimal number of line containing reference

x Type of reference, as follows:

blank Symbol value is used at this point.

D Symbol defined at this reference; that is, it appears in the location field of an instruction or is defined by a SET, =, or EXT pseudo instruction.

E Declares the symbol as an entry name.

F Symbol used in an expression on an IFE, IFA, or ERRIF conditional pseudo instruction.

R Symbol used in an address expression in a memory read instruction or as a B or T register symbol in an instruction which reads the B or T register.

S Symbol used in an address expression in a memory store instruction or as a B or T register symbol in an instruction which stores a new value in the B or T register.

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	EBCDIC	CDC DISPLAY CODE
NUL	000	12-0-9-8-1	00	None
SOH	001	12-9-1	01	None
STX	002	12-9-2	02	None
ETX	003	12-9-3	03	None
EOT	004	9-7	37	None
ENQ	005	0-9-8-5	2D	None
ACK	006	0-9-8-6	2E	None
BEL	007	0-9-8-7	2F	None
BS	010	11-9-6	16	None
HT	011	12-9-5	05	None
LF	012	0-9-5	25	None
VT	013	12-9-8-3	0B	None
FF	014	12-9-8-4	0C	None
CR	015	12-9-8-5	0D	None
S0	016	12-9-8-6	0E	None
SI	017	12-9-8-7	0F	None
DLE	020	12-11-9-8-1	10	None
DC1	021	11-9-1	11	None
DC2	022	11-9-2	12	None
DC3	023	11-9-3	13	None
DC4	024	9-8-4	3C	None
NAK	025	9-8-5	3D	None
SYN	026	9-2	32	None
ETB	027	0-9-6	26	None

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	EBCDIC	CDC DISPLAY CODE
CAN	030	11-9-8	18	None
EM	031	11-9-8-1	19	None
SUB	032	9-8-7	3F	None
ESC	033	0-9-7	27	None
FS	034	11-9-8-4	1C	None
GS	035	11-9-8-5	1D	None
RS	036	11-9-8-6	1E	None
US	037	11-9-8-7	1F	None
Space	040	None	40	55
!	041	12-8-7	5A	66
"	042	8-7	7F	64
#	043	8-3	7B	60
$	044	11-8-3	5B	53
%	045	0-8-4	6C	63
&	046	12	50	67
'	047	8-5	7D	70
(	050	12-8-5	4D	51
)	051	11-8-5	5D	52
*	052	11-8-4	5C	47
+	053	12-8-6	4E	45
,	054	0-8-3	6B	56
-	055	11	60	46
.	056	12-8-3	4B	57
/	057	0-1	61	50
0	060	0	F0	33
1	061	1	F1	34
2	062	2	F2	35
3	063	3	F3	36
4	064	4	F4	37

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	EBCDIC	CDC DISPLAY CODE
5	065	5	F5	40
6	066	6	F6	41
7	067	7	F7	42
8	070	8	F8	43
9	071	9	F9	44
:	072	8-2	7A	00
;	073	11-8-6	5E	77
<	074	12-8-4	4C	72
=	075	8-6	7E	54
>	076	0-8-66E	6E	73
?	077	0-8-7	6F	71
@	100	8-4	7C	74
A	101	12-1	C1	01
B	102	12-2	C2	02
C	103	12-3	C3	03
D	104	12-4	C4	04
E	105	12-5	C5	05
F	106	12-6	C6	06
G	107	12-7	C7	07
H	110	12-8	C8	10
I	111	12-9	C9	11
J	112	11-1	D1	12
K	113	11-2	D2	13
L	114	11-3	D3	14
M	115	11-4	D4	15
N	116	11-5	D5	16
O	117	11-6	D6	17
P	120	11-7	D7	20
Q	121	11-8	D8	21

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	EBCDIC	CDC DISPLAY CODE
R	122	11-9	D9	22
S	123	0-2	E2	23
T	124	0-3	E3	24
U	125	0-4	E4	25
V	126	0-5	E5	26
W	127	0-6	E6	27
X	130	0-7	E7	30
Y	131	0-8	E8	31
Z	132	0-9	E9	32
[	133	12-8-2	AD	61
\	134	0-8-2	E0	75
]	135	11-8-2	BD	62
^	136	11-8-7	5F	76
_	137	0-8-5	6D	65
`	140	8-1	79	None
a	141	12-0-1	81	None
b	142	12-0-2	82	None
c	143	12-0-3	83	None
d	144	12-0-4	84	None
e	145	12-0-5	85	None
f	146	12-0-6	86	None
g	147	12-0-7	87	None
h	150	12-0-8	88	None
i	151	12-0-9	89	None
j	152	12-11-1	91	None
k	153	12-11-2	92	None
l	154	12-11-3	93	None
m	155	12-11-4	94	None
n	156	12-11-5	95	None

CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	EBCDIC	CDC DISPLAY CODE
o	157	12-11-6	96	None
p	160	12-11-7	97	None
q	161	12-11-8	98	None
r	162	12-11-9	99	None
s	163	11-0-2	A2	None
t	164	11-0-3	A3	None
u	165	11-0-4	A4	None
v	166	11-0-5	A5	None
w	167	11-0-6	A6	None
x	170	11-0-7	A7	None
y	171	11-0-8	A8	None
z	172	11-0-9	A9	None
{	173	12-0	C0	None
|	174	12-11	6A	None
}	175	11-0	D0	None
~	176	11-0-1	A1	None
DEL	177	12-9-7	07	None

CODING EXAMPLES G

This appendix gives examples of efficient coding methods for long vectors, a loop counter, alternate tests on the contents of S registers, and circular shifts.

LONG VECTORS

When vectors have more than 64 elements, the vector should be segmented into groups of 64 elements and a residue before processing. The following example shows an efficient way to do this.

Location	Result	Operand	Comment
1	10	20	35
	A1	FWA	Vector first word address
	A2	LWA+1	Vector last word address+1
	A0	A1-A2	-vector length
	A3	A1-A2	
	S2	<6	
	S1	A3	
	JAP	ERROR	Error if vector length<0
	S1	#S1&S2	
	A3	S1	(A3)=63 if vector length is multiple of 64
	A3	A3+1	First segment length
LOOP	VL	A3	Set vector length, read vector segment, and perform vector computations
	...		Store result
	A1	A1+A3	Increment current position
	A0	A1-A2	
	A3	D'64	
	JAN	LOOP	Loop for all segments

LOOP COUNTER

The following example shows an efficient way to count the number of passes through loops when the number of passes does not exceed 64.

```
Location  Result    Operand     Comment
1         10        20          35

          S0        >COUNT      (mask with length=loop count)
LOOP      S0        S0<1        Shift mask
                                Perform computations
                    ...
          JSM       LOOP        Loop required number of times
```

ALTERNATE TESTS ON THE CONTENTS OF S REGISTERS

Usually S0 is used to test the contents of S registers to determine if the contents are positive, negative, zero, or nonzero. The population count and leading zero count instructions can be used to test the contents of S registers for these conditions in A0. This is useful when the contents of S0 cannot be destroyed or when one S register test needs to be made right after another.

```
Location  Result    Operand     Comment
1         10        20          35

          A0        PS3
          JAZ       SZR         If S3=0

          A0        PS3
          JAN       SNZ         If S3≠0

          A0        ZS3
          JAN       SPL         If S3≥0

          A0        ZS3
          JAZ       SMI         If S3<0
```

CIRCULAR SHIFTS

The double shift instructions (056 and 057) can be used to shift an S register circularly.

Location	Result	Operand	Comment
1	10	20	35
	S7	S7,S7<A2	
or:			
	S7	S7,S7>A2	

STRUCTURED PROGRAMMING MACROS H

The following structured programming macros which are contained in $SYSTXT and are available for use in programs written in CAL are described in the Macros and Opdefs Reference Manual, CRI publication SR-0012.

- $GOSUB
- $GOTO
- $IF, $ELSEIF, $ELSE, and $ENDIF
- $JUMP
- $LOOP, $EXITLP, and $ENDLOOP
- $RETURN
- $SUBR

CONDITIONS AND SPECIAL MACROS H

CONDITIONS

Conditions such as those used by the $IF macros can be very complex. Several classes of conditions are shown in the following paragraphs.

CONDITIONS ON A0 AND S0

Condition	Meaning
AZ...	(A0) equal to 0
AN...†	(A0) not equal to 0
AP...	(A0) greater than or equal to 0
AM...	(A0) less than 0
SZ...	(S0) equal to 0
SN...†	(S0) not equal to 0
SP...	(S0) greater than or equal to 0
SM...	(S0) less than 0

In the above, "..." means any number (including zero) of letters. For example, AZ means the same as AZero.

CONDITIONS ON A AND S REGISTERS

Condition	Meaning
A_n[=operand],Z...	(A_n) equal to 0
A_n[=operand],N...†	(A_n) not equal to 0
A_n[=operand],P...	(A_n) greater than or equal to 0
A_n[=operand],M...	(A_n) less than 0
S_n[=operand],Z...	(S_n) equal to 0
S_n[=operand],N...†	(S_n) not equal to 0
S_n[=operand],P...	(S_n) greater than or equal to 0
S_n[=operand],M...	(S_n) less than 0

In the above, n is any integer between 0 and 7 inclusive, "..." is any number of letters, and the portion in brackets is optional. If an

† Condition means not equal to 0; Ne, NE, Ng, or NG is not an acceptable condition.

operand is specified, an instruction is generated to set the indicated register to the operand's value. If an operand contains embedded commas or blanks, then the entire assignment must be enclosed in parentheses. For example, A3,Minus is true if (A3)is less than 0; A0=PS2,Zero is true if the population count of (S2) is 0; (S0=JOE,0),Plus is true if the content of memory word JOE is positive.

RELATIONAL CONDITIONS

Relational conditions are of the following forms:

A*m*[=operand],relation,A*n*[=operand] and
S*m*[=operand],relation,S*n*[=operand]

The *m* and *n* are integers between 1 and 7 inclusive, operands are as described in conditions on A and S registers, and relation is one of EQ, NE, LT, LE, GT, or GE and has the same meaning as in FORTRAN.

For example, A2,EQ,A3 is true if (A2) equals (A3); (S1=JOE,0),LT,S2=4 is true if the content of memory word JOE is less than 4.

BIT SET CONDITIONS

Condition	Meaning
constant,IN,S*m*[=operand]	True if bit number *constant* is set in register "S*m*". Bits are numbered sequentially, with the sign bit being 0.
S*n*[=operand],ALLIN,S*o*[=operand]	True if every bit set in register "S*n*" is also set in register "S*o*".
S*n*[=operand],ONEIN,S*o*[=operand]	True if at least one bit set in "S*n*" is also set in "S*o*".

In the above, *constant* is any CAL expression yielding an integer constant; *m* is any integer between 0 and 7 inclusive; *n* and *o* are any integers between 1 and 7 inclusive; and operands are as described in Conditions on A and S registers. For example, D'63,IN,S0=T.JOE is true if the content of T.JOE is odd; S2<3,ALLIN,S3 is true if the last 3 bits of S3 are set; S1,ONEIN,(S2=ERROR,0) is true if any bit set in S1 is also set in memory word ERROR.

COMPOUND CONDITIONS

Conditions on A0 and S0, conditions on A and S registers, relational conditions, and bit set conditions are called simple conditions. Both simple and compound conditions can be combined in various ways to form new compound conditions:

Condition	Meaning
NOT,(*cond*)	*cond* is not true
(*cond*$_1$),AND,(*cond*$_2$)	*cond*$_1$ and *cond*$_2$ are both true
(*cond*$_1$),OR,(*cond*$_2$)	*cond*$_1$ is true, *cond*$_2$ is true, or both are true.

In the above, *cond*, *cond*$_1$, and *cond*$_2$ are any conditions, simple or compound. The parentheses are required. For example, NOT,(S1,EQ,S2) is true if (S1) is not equal to S2; (AMinus),OR,(SMinus) is true if (A0) is less than 0 or (S0)is less than 0 or both; and ((A1,GE,A2='A'R),AND,(A1,LE,A2='Z'R)), OR ((A1,GE,A2='a'R),AND,(A1,LE,A2='z'R)) is true if A1 contains an uppercase or lowercase letter.

SPECIAL MACROS

The following macros are contained in $SYSTXT and are available for use in programs written in CAL. Unlike the majority of macros in $SYSTXT, these are independent of the operating system.

$IF MACRO

The $IF macro operates in the same manner as the similar structure in FORTRAN when used with the attendant $ELSEIF, $ELSE, and $ENDIF macros. The $ELSEIF and $ELSE macros are optional. If both are included, an $ELSEIF macro cannot follow a $ELSE macro.

The conditions that can be used with $IF or $ELSEIF are described under conditions of this appendix.

$IF groups can be nested within other $IF groups up to a level of 10 deep.

The value of an IF or ELSEIF condition is treated as either true or false. If true, the block that follows is executed; if false, it is skipped. The ELSE statement, if present, must follow any ELSEIF statements that belong to the same IF group. Within each IF group, no

more than one block is executed (once a block is executed, the remaining blocks in the same IF group are skipped). If none of the blocks in a group have been executed when an ELSE statement is encountered, then the ELSE block is executed if present. A block can be null (that is, it can contain no statements to be executed).

Example:

Location	Result	Operand	Comment
1	10	20	35
	$IF	condition	
	assembly code . . .		This code is executed if the $IF condition is true.
	$ELSEIF	condition	
	assembly code . . .		If the $IF condition is false and the $ELSEIF condition is true, this code is executed.
	$ELSE		
	assembly code . . .		If both of the above conditions are false, this code is executed.
	$ENDIF		

Examples of conditions used with $IF, $ELSEIF, and $ELSE are shown below.

Example 1:

```
$IF       AZ
  S1        A3
  S2        A4
$ELSEIF   SZ
  A1        S2
  A2        S3
  $IF       AP
    A1        A2
    A3        S4
  $ENDIF
$ELSE
  S1        S2*FS3
$ENDIF
```

Example 2:

```
$IF       S2,LT,S4
  A1        2
$ELSEIF   A5,GE,A1
  A1        5
$ELSEIF   S1,EQ,S7=123
  A2        6
$ELSE
  $IF       A2,NE,A5=ABC
    A3        4
  $ELSEIF   S5,GT,S7=LABEL
    A1        5
  $ENDIF
$ENDIF
```

$GOTO MACRO

The $GOTO macro offers CAL users a computed GO TO statement.

NOTE

Unlike the 1-based FORTRAN computed GO TO, this GO TO statement is 0-based.

Example:

Location	Result	Operand	Comment
1	10	20	35
	$GOTO	Ai, (label$_0$, label$_1$...label$_n$), Aj	

Register Ai is a scratch register, and register Aj holds a value that determines to which label the jump takes place. For instance, if Aj=1 the jump is to label$_1$. If Aj is greater than n, no jump takes place, and control falls through to the next instruction.

DATA GENERAL CAL I

Data General CAL is a development tool used by Cray Research, Inc. and is not a CRI supported product. It may be used on the MCU for the CRAY-1 Model A and B systems and for Models S/250, S/500 and S/1000 of the S Series CRAY-1 Computer Systems.

SUMMARY OF DIFFERENCES BETWEEN CPU CAL AND DATA GENERAL CAL

- Expression evaluation

  Data General CAL evaluates expressions from left to right without regard for term operators. CPU CAL forms elements into a term and incorporates the term into the sum of previously evaluated terms.

- Continuation lines

  Data General CAL does not allow continuation lines.

- Operand field

  CPU CAL handles the case where a result field extends beyond column 34 and an operand field begins after column 35. Data General CAL does not handle this case.

- Line editing

  Data General CAL does not support concatenation and micros.

- Qualified symbols

  Data General CAL does not support qualified symbols.

- Special elements

  The only special element supported by Data General CAL is *.

- Data notation

  Data elements in Data General CAL can be octal integers (0 prefix), decimal integers (D prefix), or a character string (A prefix) that can fit into 64 or fewer bits. The only suffix supported is for character justification and fill (H, L, or R).

- Numeric base

  For Data General CAL, if the 0 or D prefix is omitted from a numeric element, it is assumed to be octal. For CPU CAL, the default can be set by a BASE pseudo but is decimal if no BASE pseudo is supplied.

- Register designators

  The designators for A, S, and V registers for Data General CAL must be numeric, not symbolic. Designators for B and T registers may be symbolic but must be defined before their use in an instruction.

- Pseudo instructions

  Data General CAL supports the following subset of pseudo instructions.

  ABS Optional in Data General CAL which assembles only absolute code

  BSS Unused parcels are padded with pass instructions (Sl Sl&Sl) not with zeros as in CPU CAL

  BSSZ No differences

  CON The operand field can contain only one entry in Data General CAL

  EJECT No differences

  END No differences

  ENTRY In Data General CAL, the operand field can contain only one entry

  IDENT In Data General CAL, statements preceding IDENT and between END and IDENT are taken as comments

  LIST A non-empty operand field enables the listing for Data General CAL

  ORG No differences for absolute assembly

- Symbolic machine instructions

  Data General CAL symbol instructions are a subset of CPU CAL symbolic machine instructions except for the following which are recognized by Data General CAL but not by CPU CAL.

Location	Result	Operand	Comment
1	10	20	35
	,	B*jk*,A*i*	
	,	T*jk*,A*i*	
	B*jk*,A*i*	,	
	T*jk*,A*i*	,	
	,,1	V*i*	
	,,A0	V*i*	
	V*i*	,,A0	
	V*i*	,,1	

Special syntax forms for Data General CAL are a subset of CPU CAL symbolic machine instructions. The following symbolic machine instructions are recognized by CPU CAL but not by Data General CAL.

Location	Result	Operand	Comment
1	10	20	35
	ERR	*exp*	
	VL	1	
	VM	0	
	EX	*exp*	
	A*i*	-1	
	B*jk*,A*i*	0,A0	
	0,A0	B*jk*,A*i*	
	T*jk*,A*i*	0,A0	
	0,A0	T*jk*,A*i*	
	S*i*	S*j*&SB	
	S*i*	#SB&S*j*	
	S*i*	S*j*\SB	
	S*i*	SB\S*j*	
	S*i*	#S*j*\SB	
	S*i*	#SB\S*j*	
	S*i*	#SB	
	S*i*	S*j*!S*i*&SB	
	S*i*	S*j*!SB	
	S*i*	SB!S*j*	
	S*i*	SB	
	S*i*	S*i*,S*j*<1	
	S*i*	S*j*,S*i*>1	
	S*i*	+FS*k*	
	S*i*	-FS*k*	

Location	Result	Operand	Comment
1	10	20	35
	Vi,Ak	0	
	Ai	exp,	
	Ai	,Ah	
	exp,	Ai	
	,Ah	Ai	
	Si	exp,	
	Si	,Ah	
	exp,	Si	
	,Ah	Si	
	Vi	0	
	Vi	Vk	
	Vi	#VM&Vk	
	Vi	Vj<1	
	Vi	Vj>1	
	Vi	Vj,Vj<1	
	Vi	Vj,Vj>1	
	Vi	Vj-Vk	
	Vi	+FVk	
	Vi	-FVk	
	VM	Vj,Z	
	Vi	,A0,1	
	,A0,1	Vj	

- Execution of Data General CAL assembler

  Name: CAL

  Format: CAL *filename*

  Purpose: To assemble a CAL assembly language source file. Output can be an absolute binary file, a listing file, or both.

  Switches:

  Global: By default, output of an assembly is an absolute binary file (no listing file). Switches other than those specified are ignored.

  /E - List only lines with errors on listing file; no effect if L or P switches not selected.

  /L - Listing file is produced on *filename*.LS.

  /N - No absolute binary file is produced.

/O - Override effect of LIST pseudo-instructions; no effect if L or P switches not selected.

/P - Listing on printer; overridden by L switch.

/X - Produce cross referencing of symbol table; no effect if L or P switches not selected.

Local: None

Extensions: On input, search for *filename*.

On output, produce *filename*.SV for absolute binary and *filename*.LS for listing (global L switch selected).

The source file name specified on the call cannot have an extension and is limited to ten characters.

Examples: In these examples, each statement must be terminated with a carriage return.

```
CAL Z
```

This example causes assembly of CAL source file Z, producing an absolute binary file called Z.SV.

```
CAL/N/L A
```

This example causes assembly of file A, producing as output a listing file A.LS. No binary file is produced.

```
CAL/P/X EXAMP
```

This example causes assembly of file EXAMP, producing an assembly listing with cross-referenced symbol table, output to the line printer, and an absolute binary file EXAMP.SV.

- Execution of generated binary under COS

A binary generated by Data General CAL can execute on the CRAY-1 under COS if the following steps are taken.

1. Block the binary as a separate dataset using the B option. (See BLOCK utility in Data General Station (DGS) Operator's Guide, CRI publication SG-0006.)

2. Stage the dataset to the CRAY-1.

3. Access the dataset from a job.

4. Execute the dataset by specifying the dataset name as the verb of a control statement. (Note that the LDR utility is not able to load the dataset.)

INDEX

INDEX

READERS COMMENT FORM

CAL Assembler Version 1 Reference Manual SR-0000 J

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention:
PUBLICATIONS

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.

STAPLE

READERS COMMENT FORM

CAL Assembler Version 1 Reference Manual SR-0000 J

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention:
PUBLICATIONS

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.

STAPLE