Cray Assembly Language [4]

This section presents the general rules and statement syntax for Cray Assembly Language (CAL). This section describes the following instruction syntax:

- Register designators
- Names
- Symbols
- Data
- Special elements
- Element prefixes for symbols, constants, or special elements
- Expressions
- Expression evaluation
- Expression attributes

Register designators
4.1

Register designators are used in symbolic machine instructions and opdefs to specify the register to be used for an operation. CAL accepts register mnemonics specified in uppercase, lowercase, or mixed case. Each Cray PVP system supports all or a subset of simple and complex registers.

Complex registers are members of a set of registers that are identical in function and architecture. The set of registers is identified by a letter. The specific register within the set is specified by an octal number up to 4 octal digits in length or a constant. For example, you specify register `S1` from the set of S registers.

The following example illustrates complex register designation:

```
      A1     SyM          ; CAL permits mixed case in any combination
                          ; with the following restriction: matching
                          ; names must be entered in the same manner.
 REG  =      3
      A.REG  A1           ; Register A3 gets the contents of A1
      S1     s2           ; Register S1 gets the contents of S2.
```

A *simple* register has a predefined function that cannot be redefined. These registers are identified by register names that are comprised of only letters.

The following example illustrates simple register designation:

```
      S1     RT           ; Register S1 gets the contents of the RT
                          ; register
```

Table 3 lists register designations for CRAY Y-MP, CRAY C90, CRAY J90, and CRAY T90 series systems. By convention, B and T registers are written using two octal digits.

> **Note:** A, B, and SB registers are expanded to 64 bits (32 bits in C90 mode) on CRAY T90 systems. Several new instructions use the 64-bit A registers for logical and shift operations. See appendix E, page 369, for more information on the CRAY T90 instruction set.

Table 3. Register designations

Register type	CRAY Y-MP Number Mnemonic Size (in bits)	CRAY C90 Number Mnemonic Size (in bits)	CRAY J90 Number Mnemonic Size (in bits)	CRAY T90 Number Mnemonic Size (in bits)
Data registers (A registers)	8 A0 – A7 32	8 A0 – A7 32	8 A0 – A7 32	8 A0 – A7 64
Data registers (B registers)	64 B00 – B77 32	64 B00 – B77 32	64 B00 – B77 32	64 B00 – B77 64

Table 3. Register designations
(continued)

Register type	CRAY Y-MP Number Mnemonic Size (in bits)	CRAY C90 Number Mnemonic Size (in bits)	CRAY J90 Number Mnemonic Size (in bits)	CRAY T90 Number Mnemonic Size (in bits)
Scalar registers (S registers)	8 `S0 – S7` 64	8 `S0 – S7` 64	8 `S0 – S7` 64	8 `S0 – S7` 64
Transfer registers (T registers)	64 `T00 – T77` 64	64 `T00 – T77` 64	64 `T00 – T77` 64	64 `T00 – T77` 64
Vector registers (V registers)	8 `V0 – V7` 64 (64 elements)	8 `V0 – V31` 64 (128 elements)	8 `V0 – V31` 64 (64 elements)	8 `V0 – V63` 64 (128 elements)
Shared address registers (SB registers)	8/cluster `SB0 – SB7` 32	8/cluster `SB0 – SB7` 32	8/cluster `SB – SB7` 32	16/cluster `SB0 – SB7` 64
Semaphore register (SM register)	32/cluster `SM0 – SM37` 1	32/cluster `SM0 – SM37` 1	32/cluster `SM0 – SM37` 1	64/cluster `SM0 – SM77` 1
Status registers (SR registers)	1 `SR` 32	8 `SR0 – SR7` 64	1 `SR` 32	8 `SR0 – SR7` 64
Shared scalar registers (ST registers)	8 `ST0 – ST7` 64	8 `ST0 – ST7` 64	8 `ST0 – ST7` 64	16 `ST0 – ST15` 64
Channel address register (CA register)	1/channel `CA` 32	1/channel `CA` 32	1/channel `CA` 32	1/channel `CA` 64
Channel error register (CE register)	1/channel `CE` 32	1/channel `CE` 32	1/channel `CE` 32	1/channel `CE` 64

Table 3. Register designations
(continued)

Register type	CRAY Y-MP Number Mnemonic Size (in bits)	CRAY C90 Number Mnemonic Size (in bits)	CRAY J90 Number Mnemonic Size (in bits)	CRAY T90 Number Mnemonic Size (in bits)
Channel interrupt register (CI register)	1/channel `CI` 32	1 `CI` 6	1/channel `CI` 1	1/channel `CI` 32
Channel limit register (CL register)	1/channel `CL` 32	1/channel `CL` 32	1/channel `CL` 1	1/channel `CL` 32
Master clear register (MC register)	1/channel `MC` 1	1/channel `MC` 1	1/channel `MC` 1	1/channel `MC` 1
Real-time clock register (RT register)	1 `RT` 64	1 `RT` 64	1 `RT` 64	1 `RT` 64
Vector length register (VL register)	1 `VL` 7	1 `VL` 8	1 `VL` 7	1 `VL` 8
Vector mask registers (VM registers)	1 `VM` 64	2 `VM0`, `VM1` 32	1 `VM` 64	2 `VM0`, `VM1` 64
Exchange address register (XA register)	1 `XA` 8	1 `XA` 8	1 `XA` 10	1 `XA` 16
Program address register (P register)	1 `P` 24	1 `P` 32	1 `P` 24	1 `PA` 32

Vector length register (VL) 4.1.1

On CRAY C90 systems, vector registers have been increased to 128 elements and the vector mask has been increased to 128 bits. On CRAY Y-MP systems and CRAY J90 systems, the `VL` register has 7 bits. On CRAY C90 systems and CRAY T90 systems the `VL` register has 8 bits. To facilitate portable CAL code, the following symbols are available (UNICOS 7.0 and later) on all Cray PVP systems:

- `MAX$VL` (maximum vector length)
- `L2$MAX$VL` ($\log_2$ of `MAX$VL` for shift and mask operations)
- `A$MAX$VL` (a micro that yields a valid A register operand equal to `MAX$VL`)

These symbols can be used in lieu of conditional code. For example:

```
C90IFC /"$CPU"/,EQ,/CRAY C90/
      S1  <7
C90ELSE
      S1  <6
C90ENDIF
      S0  S1&S2           ; Vector residual
      S2  S1&S2
      A2  S2
      $IF S0,Zero         ; If no residual
C90IFC /"$CPU"/,EQ,/CRAY C90/
      A2  D'128           ; First VL = 128
C90ELSE
      A2  D'64            ; First VL = 64
C90ENDIF
      $ENDIF
```

can be replaced with:

```
S1    <L2$MAX$VL
S0    S1&S2    ; Vector residual
S2    S1&S2
A2    S2
$IF   S0,Zero  ; If no residual
      A2  "A$MAX$VL"      ; First VL
$ENDIF
```

Vector mask register (VM)
4.1.2

The vector mask register on CRAY Y-MP systems and CRAY J90 systems is a 64-bit register, on CRAY C90 systems there are two 32-bit vector mask registers, and on CRAY T90 systems there are two 64-bit vector mask registers. The vector mask (VM) register is accessed through the following instructions:

```
Si   VM0  ; Get first half of VM
Si   VM1  ; Get second half of VM
VM0  Si   ; Set first half of VM
VM1  Si   ; Set second half of VM
```

If the routine does not manipulate the vector mask, no portable CAL changes are necessary. The changes necessary for a portable CAL module depend on the types of manipulations required of the vector mask. A routine that only logically combines conditions together (through AND and OR operations) can create opdefs to simulate a two-word VM register on all Cray PVP systems. For example:

```
C12XY  IFC   /"$CPU"/,NE,/CRAY C90/
       VM0    OPDEF
       S.REG  VM0
      *
      S.REG  VM
*
     VM0      ENDM
     VM0      OPDEF
     VM0      S.REG
*
     VM       S.REG
*
     VM0      ENDM
     VM1      OPDEF
     S.REG              VM1
*
     VM1      ENDM
     VM1      OPDEF
     VM1      S.REG
*    VM1      ENDM
C12XY  ENDIF
```

Using the above opdefs, a system that has a 64-bit `VM` register can be made to look as though it has a 128-bit `VM` register. A portable code sequence such as the following could be written:

```
S1    VM0     ; First half of VM
S2    VM1     ; Possible second half of VM
VM    V7,Z    ; Test second condition
S3    VM0     ; First half of VM
S4    VM1     ; Possible second half of VM
S1    S1!S3   ; Combine conditions
S2    S2!S4
VM0   S1      ; Set VM
VM1   S2
```

This code can then assemble on any Cray PVP system. The instructions that actually reference the second half of the `VM` register do not generate code except on CRAY C90 systems. The only extra code for other systems is the `S2 S2!S4` instruction. The `S`*i* `VM1` opdef could be modified to set `S`*i* to zero if more sophisticated vector mask operations were to be performed.

Names
4.2

Names do not have an associated value or attribute and cannot be used in expressions. Names that are 1 to 8 characters in length are used to identify the following types of information:

- Macro instructions
- Micro character strings
- Conditional sequences
- Duplicated sequences

The first character must be one of the following:

- Alphabetic character (A through Z or a through z)
- Dollar sign ($)
- Percent sign (%)
- At sign (@)

Characters 2 through 8 can also be decimal digits (0 through 9).

Names that are 1 to 255 characters in length can be used to identify the following types of information:

- Program modules
- Sections

The first character must be one of the valid name characters or the underscore (_) character. Characters 2 through 255 can also be decimal digits (0 through 9).

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 section.

Examples of valid and not valid names:

Valid	Comment
`count`	Lowercase is permitted
`@ADD`	`@` legal beginning character
`_SUBTRACT`	_ beginning character and 9 characters are legal
`ABCDE465`	Combinations of letters and digits are legal if the first character is legal

Not valid	Comment
`9knt`	Begins with a numeric character
`JOHNJONES`	Contains more than 8 characters
`Y+Z3`	Contains an illegal character
`+YZ3`	Begins with `+`

Note: UNICOS supports the Source Code Control System (SCCS) and UNICOS source manager (USM). If you plan to use SCCS or USM to store your CAL program, avoid using the 3-character string `%`*U*`%`; where *U* is any uppercase letter. SCCS and USM replace these strings throughout your source program with other text. Because this type of string is allowed within identifiers and long-identifiers, avoid using it in names, long names, and symbols.

The underscore character (_) also is used as the concatenation character by CAL (see subsection 3.3, page 41). Usually the assembler edits this character out of a source line. To insert this character into a long name, either disable editing or use the predefined concatenation micro (`$CNC`). To disable editing, use either the invocation statement or the `EDIT` pseudo instruction.

Symbols
4.3

A *symbol* is an identifier that can be from 1 to 255 characters long and has an associated value and attributes. You can use symbols in expressions and in the following ways:

- In the location field of a source statement to define the symbol for use in the program assign it a value and certain characteristics called attributes.
- In the operand or result field of a source statement to reference the symbol.
- In loader linkage

A symbol can be local or global depending on where the symbol is defined; that is, a symbol used within a single program module is local and a symbol used by a number of program segments is global (see subsection 3.1.2, page 35). A symbol also can be unique to a code sequence (see subsection 4.3.1.2, page 71).

CAL generates symbols of the following form (where *n* is a decimal digit):

`%%`*nnnnnn*

Symbols that begin with the character sequence `%%` are discarded at the end of a program segment regardless of whether they are redefinable or defined in the global definitions part, and regardless of whether they are user-defined or generated by CAL.

For more detailed information about symbols generated by CAL, see the description of the `LOCAL` pseudo instruction in subsection 6.11, page 184.

If a symbol is properly identified and defined as one of the registers reserved by CAL, a warning message is issued.

Symbols can be used if they are:

- Specified as unqualified or qualified.
- Defined or associated with a value and attributes.
- Assigned address, relative, and redefinable attributes.
- Referenced by using the value rather than the symbol itself.

Symbol qualification 4.3.1

Symbols defined within a program module (between IDENT and END pseudo instructions) can be unqualified or qualified. They are unqualified unless preceded by the QUAL pseudo instruction (see the QUAL pseudo instruction in appendix A, page 189, for more information).

Unqualified symbol 4.3.1.1

The following statements describe ways in which unqualified symbols can be referenced:

- Unqualified symbols defined in an unqualified code sequence can be referenced without qualification from within that sequence.
- If the symbol has not been redefined within the current qualifier, unqualified symbols can be referenced without qualification from within the current qualifier.
- Unqualified symbols can be referenced from within the current qualifier by using the form //*symbol*.

Unqualified symbols are defined as follows:

```
symbol   =    n    ; symbol is equal to n
```

The following example illustrates unqualified symbol definition:

```
       EDIT   OFF
       IDENT  TEST
SYM_1  =      *          ; SYM_1 has a value equal to the location
                         ; counter.
       A1     SYM_1      ; Register A1 gets SYM_1's value.
SYM_2  SET    2          ; SYM_2 is redefinable
SYM_3  =      3          ; SYM_3 is not redefinable.
```

Qualified symbols 4.3.1.2

You can make a symbol that is not a global symbol unique to a code sequence by specifying a symbol qualifier that will be appended to all symbols defined within the sequence. The `QUAL` pseudo instruction qualifies symbols (see the `QUAL` pseudo instruction in appendix A, page 189).

Qualified symbols must be defined with respect to the following rules:

- A qualified symbol cannot be defined with a label that is reserved for registers.
- Symbols can be qualified only in a program module.

Qualified symbols can be referenced as follows:

- If a qualified symbol defined in a code sequence is referenced from within that sequence, it can be referenced without qualification.
- If a qualified symbol is referenced outside of the code sequence in which it was defined, it must be referenced in the form /*qualifier*/*symbol*. The *qualifier* variable is a 1- to 8-character identifier defined by the `QUAL` pseudo instruction and the *symbol* variable is a 1- to 255-character identifier.

Qualified symbols are defined as follows:

qualified_symbol = /[*identifier*]/*symbol*

The following example illustrates the use of qualified symbols:

```
      IDENT    TEST
SYM1  =        1                 ; Assignment
      QUAL     NAME1             ; Declare qualifier name
SYM1  =        2                 ; Qualified symbol SYM1
      S1       SYM1              ; Register S1 gets 2 (qualified SYM1)
      S1       //SYM1            ; Register S1 gets 1 (unqualified SYM1)
      S1       /NAME1/SYM1       ; Register S1 gets 2 (qualified SYM1)
      QUAL     *                 ; Pop the top of the qualifier stack
      S1       SYM1              ; Register S1 gets 1
      S1       //SYM1            ; Register S1 gets 1
      S1       /NAME1/SYM1       ; Register S1 gets 2
      END
```

Symbol definition
4.3.2

A symbol is defined by assigning it a value and attributes. The value and attributes of a symbol depend on how the program uses the symbol. The assignment can occur in the following three ways:

- When a symbol is used in the location field of a symbolic machine instruction or certain pseudo instructions, it is defined as follows:
  - It has the address of the current value of the location counter (for a description of counters, see subsection 3.6.3, page 56).
  - It has parcel-address or word-address attributes.
  - It is absolute, immobile, or relocatable.
  - It is not redefinable.
- 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 the instruction. Some symbol-defining pseudo instructions cause the symbol to have a redefinable attribute. When a symbol is redefinable, a redefinable pseudo instruction must be used to define the symbol the second time. Redefinition of the symbol causes it to be assigned a new value and attributes.
- A symbol can be defined as external to the current program module. A symbol is external if it is defined in a program module other than the module currently being assembled. The true value of an external symbol is not known within the current program module.

The following are examples of a symbol:

```
START  =    *             ; The symbol START has the current value of
                          ; the location counter and cannot be
                          ; redefined.
PARAM  SET  D'18          ; The symbol PARAM is equal to the decimal
                          ; value 18 and can be redefined.
       EXT  SECOND        ; Identifies SECOND as an external symbol.
```

Symbol attributes 4.3.3

When a symbol is defined, it assumes two or more attributes. These attributes are in three categories as follows:

- Address
- Relative
- Redefinable

Every symbol is assigned one attribute from each of the first two categories. Whether a symbol is assigned the redefinable attribute depends on how the symbol is used. Each symbol has a value of up to 64 bits associated with it.

Address attributes 4.3.3.1

Each symbol is assigned one of the following address attributes:

- Word address

  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`) that defines words, if it is equated to an expression having a word-address attribute, or if word is explicitly stated in the operand field of an `EXT` pseudo instruction.

- Parcel address

  A symbol is assigned a parcel-address attribute if it appears in the location field of a symbolic machine instruction or certain pseudo instructions, if it is equated to an expression having a parcel-address attribute, or if parcel is explicitly stated in the operand field of an `EXT` pseudo instruction.

- Value

  A symbol has a value attribute if it does not have word-address or parcel-address attributes, or if value is explicitly stated in the operand field of an `EXT` pseudo instruction. All globally defined symbols have an address attribute of value.

- Absolute

  A symbol is assigned the relative attribute of absolute when the current location counter is absolute and it appears in the location field of a machine instruction, `BSS` pseudo instruction, or data generation pseudo instruction such as `BSSZ` or `CON` or if it is equated to an expression that is absolute. All globally defined symbols have a relative attribute of absolute. The symbol is known only at assembly time.

Relative attributes 4.3.3.2

Each symbol is assigned one of the following relative attributes:

- Immobile

  A symbol is assigned the relative attribute of immobile when the current location counter is immobile and it appears in the location field of a machine instruction, `BSS` pseudo instruction, or data generation pseudo instruction such as `BSSZ` or `CON` or if it is equated to an expression that is immobile. The symbol is known only at assembly time.

- Relocatable

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

- External

  A symbol is assigned the relative attribute of external when it is defined by an `EXT` pseudo instruction. An external symbol defined in this manner is entered in the symbol table with a value of `0`. The address attribute of an external symbol is specified as value (`V`), parcel (`P`), or word (`W`); the default is value.

  A symbol is also assigned the relative 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.

  Note: The assignment of an unknown variable with a register at assembly time is made by using a symbol with a relative attribute of external.

In the following example, register `s1` is loaded with variable `ext1` at assembly time:

```
       ext    test1         ; Variable extl is defined as an external
                            ; variable
       s1     ext1          ; ext1 transmits value to register s1
       end
       ident  test2
       entry  ext1
ext1   =      3             ; When the two modules are linked, register
                            ; S1 gets 3.
       end
```

Redefinable attributes
4.3.3.3

In addition to its other attributes, a symbol is assigned the attribute of *redefinable* if it is defined by the `SET` or `MICSIZE` pseudo instructions. A redefinable symbol can be defined more than once in a program segment and can have different values and attributes at various times during an assembly. When such a symbol is referenced, its most recent definition is used by the assembler. All redefinable symbols are discarded at the end of a program segment without regard to whether they were defined in the global definitions.

The following example illustrates the redefinable attribute:

```
       IDENT  TEST
SYM1   =      1               ; Not redefinable
SYM2   SET    2               ; Redefinable
SYM1   SET    2               ; Error: SYM1 previously defined as 1
SYM2   SET    3               ; Redefinable
       END
```

Symbol reference
4.3.4

When a symbol is in a field other than the location field, the symbol is being referenced. Reference to a symbol within an expression causes the value and attributes of the symbol to be used in place of the symbol. Symbols can be found in the operand fields of pseudo instructions.

A symbol reference within an expression can contain a prefix that causes the value and attributes associated with the symbol to be altered. The prefix affects only the specific reference in which it occurs. For details, see subsection 4.6, page 90.

The following example illustrates a symbol reference:

```
        S1     SYM1+1       ; Register S1 gets the value of SYM1+1.
                            ; SYM1+1 is an example of a symbol in an
                            ; operand field used as an expression.
        IFA    DEF,SYM1     ; Symbols can also be used outside of an
                            ; expression.  In this instance, SYM1 is
                            ; not used within an expression; it is a
                            ; symbol.
```

Data
4.4

Some instructions manipulate data. CAL instructions use data of the following types:

- Constants
- Data items
- Literals

The subsections that follow describe these types of data.

Constants
4.4.1

Constants can be defined as floating, integer, or character.

Floating constant
4.4.1.1

A *floating constant* is evaluated as a one- or two-word quantity, depending on the precision specified. (See the floating-point data format figures in the appropriate symbolic machine instruction manual.)

The floating constant is defined as follows:

[*decimal-prefix*] *floating-decimal* [*binary-scale decimal-integer*]

In the preceding definition, variables are defined as follows:

- *decimal-prefix*

  This variable specifies the numeric base for the *floating-decimal* and/or the *decimal-integer* variables. `D'` or `d'` specifies a *decimal-prefix* and is the only prefix available for a floating constant.

- *floating-decimal*

  The *floating-decimal* variable can include the *decimal-integer*, *decimal-fraction* and/or *decimal-exponent* variables. A *decimal-integer* is a nonempty string of decimal digits. A *decimal-integer* or a *decimal-fraction* is a nonempty string of decimal digits representing a whole number, a mixed number, or a fraction.

  A *floating-decimal* can be defined as follows:

  – A *decimal-integer* followed by a *decimal-fraction* with an optional *decimal-exponent* and *decimal-integer*. For example:

    n.*n* or *n*.*n*E*n* or *n*.*n*E+*n* or *n*.*n*D*n* or *n*.*n*D+*n*

  – A *decimal-integer* followed by a period (.) with a *decimal-exponent* and *decimal-integer*. For example:

    n. or *n*.E*n* or *n*.E+*n* or *n*.*n*D*n* or *n*.*n*D+*n*

  – A *decimal-integer* followed by a *decimal-exponent* and *decimal-integer*. For example:

    *n*E*n* or *n*E+*n* or *n*D*n* or *n*D+*n*

  – A *decimal-fraction* followed by an optional *decimal-exponent* and *decimal-integer*. For example:

    .*n* or .*n*E*n* or .*n*E+*n* or .*n*D*n* or .*n*D+*n*

- *decimal-exponent*

  The power of 10 by which the integer and/or *fraction* will be multiplied; indicates whether the constant will be single precision (`E` or `e`; one 64-bit word) or double precision (`D` or `d`; 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. *decimal-exponents* are defined as follows:

  – `E`*n* (Positive decimal exponent, single precision)
  – `E+`*n* (Positive decimal exponent, single precision)
  – `E–`*n* (Negative decimal exponent, single precision)
  – `D`*n* (Positive decimal exponent, double precision)
  – `D+`*n* (Positive decimal exponent, double precision)
  – `D–`*n* (Negative decimal exponent, double precision)

- *binary-scale decimal-integer*

  The *integer* and/or *fraction* will be multiplied by a power of 2. Binary scale is specified with `S` or `s` and an optional add-operator (+ or –). *n* is an integer in the base specified by the *decimal-prefix*. For example:

`S`*n* or `S+`*n*	Positive binary exponent
`s`*n* or `s+`*n*	Positive binary exponent
`S–`*n* or `s–`*n*	Negative binary exponent

  Note: Double-precision floating-point numbers are truncated to single-precision floating-point numbers if pseudo instructions, which can reserve only one memory word (such as the `CON` pseudo instruction) are used.

The following examples illustrate floating constants:

```
        CON    D'1.5          ; Mixed decimal of the form n.n.
        CON    4.5E+10        ; Single-precision floating constant of
                              ; the form n.nE+n.
        CON    4.D+15         ; Double-precision floating constant of
                              ; the form n.D+n.
        CON    D'1.0E-6       ; Negative floating constant of the form
                              ; n.nE-n.
        CON    1000e2         ; Single precision floating constant of
                              ; the form nD+n.
SYM     =      1777752d+10    ; Double-precision floating constant of
                              ; the form nD+n.
```

Integer constant
4.4.1.2

An *integer constant* is evaluated as a 64-bit twos complement integer. The integer constant is defined as follows:

base-integer [*binary-scale base-integer*]
octal-prefix octal-integer [*binary-scale octal-integer*]
decimal-prefix decimal-integer [*binary-scale decimal-integer*]
hex-prefix hex-integer [*binary-scale hex-integer*]

In the preceding definition, variables are defined as follows:

- *base-integer*

  A string of decimal digits (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`) of any length

- *binary-scale*

  The integer and/or fraction that will be multiplied by a power of 2. *binary-scale* is specified with `S` or `s` and an optional add-operator (+ or −). *n* is an integer in the base specified by the *decimal-prefix*. For example:

  `S`*n* or `S+`*n* (positive binary exponent)

  `s`*n* or `s+`*n* (positive binary exponent)

  `s−`*n* or `S−`*n* (negative binary exponent)

- *base-integer*, *octal-prefix*, *decimal-prefix*, or *hex-prefix*

  Numeric base used for the integer. If no prefix is used, base-integer is determined by the default mode of the assembler or by the `BASE` pseudo instruction. A prefix can be one of the following:

`D'` or `d'`	Decimal (default mode)
`O'` or `o'`	Octal
`X'` or `x'`	Hexadecimal

- *octal-integer*

  A string of octal integers (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`) of any length

- *decimal-integer*

  A string of decimal integers (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`) of any length

- *hex-integer*

  A string of hexadecimal integers (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A` or `a`, `B` or `b`, `C` or `c`, `D` or `d`, `E` or `e`, `F` or `f`) of any length

The following examples illustrate integer constants:

```
        S1     O'1234567     ; Octal-prefix followed by octal-integer.
        A4     D'50          ; Integer-constant of the form
                             ; decimal-prefix followed by
                             ; decimal-integer.
 SYM    =      x'ffffffa     ; Integer-constant of the form hex-prefix
                             ; followed by hex-integer.
```

Character constants
4.4.1.3

The character constant is defined as follows:

[*character-prefix*] *character-string* [*character-suffix*]

In the preceding definition, variables are defined as follows:

- *character-prefix*

  The character set used for the stored constant:

`A` or `a`	ASCII character set (default)
`C` or `c`	Control Data display code
`E` or `e`	EBCDIC character set

- *character-string*

  The default is a string of zero or more characters (enclosed in apostrophes) from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate one apostrophe.

- *character-suffix*

  The justification and fill of a character string:

`H` or `h`	Left-justified, blank-filled (default)
`L` or `l`	Left-justified, zero-filled
`R` or `r`	Right-justified, zero-filled
`Z` or `z`	Left-justified, zero-filled, at least one trailing binary zero character guaranteed

The following examples illustrate character constants:

```
        S3   `*'R        ; ASCII character set (default) right
                         ; justified, zero filled.
        CON  A'ABC'L     ; ASCII character set left justified, zero
                         ; filled.
        S1   E'XYZ'H     ; EBCDIC character set left justified, blank
                         ; filled.
        CON  C'OUT       ; CDC character set left justified, blank
                         ; filled (default).
        VWD  32/'EFG'    ; ASCII character set left justified, blank
                         ; filled within a 32-bit field (all default).
```

Data items 4.4.2

A *character* or *data* item can be used in the operand field of the DATA pseudo instruction and in literals. The length of the data field occupied by a data item is determined by its type and size. Data items can be floating, integer, or character. The subsections that follow describe these types of data items.

Floating data item 4.4.2.1

Single-precision floating data items occupy one word and double-precision floating data items occupy two words. A floating data item is defined as follows:

```
[ sign ] floating-constant
```

In the preceding definition, the *sign* variable is defined as follows:

- *sign*

  The sign variable determines how the floating data item will be stored. The sign variable can be specified as follows:

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

Note: Although syntactically correct, # is not permitted; a semantic error is generated with floating data.

- *floating-constant*

  The syntax for a floating data item is the same as the syntax for floating constants. Floating constants are described in subsection 4.4.1.1, page 76.

The following example illustrates floating constants for data items:

```
        DATA   D'1345.567      ; Decimal floating data item of the form
                               ; n.n.
        DATA   1345.E+1        ; Decimal floating data item of the form
                               ; n.E+n.
        DATA   4.5E+1O         ; Single-precision floating constant of
                               ; the form n.nE+n.
        DATA   4.D+15          ; Double-precision floating constant of
                               ; the form n.D+n.
        DATA   D'1.0E-6        ; Negative floating constant of the form
                               ;  n.nE-n.
        DATA   1000e2          ; Single-precision floating constant of
                               ; the form nen.
        DATA   1.5S2           ; Floating binary scale data item of the
                               ; form n.nSn.
```

Integer data item
4.4.2.2

An integer data item occupies one 64-bit word and is defined as follows:

[*sign*] *integer-constant*

In the preceding definition the *sign* variable defines the form of a data item to be stored. The *sign* variable can be replaced in the integer data item definition with any of the following:

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

The syntax for *integer-constant* is described in subsection 4.4.1.2, page 79.

The following example illustrates integer constants for data:

```
         DATA   +o'20          ; Octal integer
         VWD    40/0,24/O'200
```

Character data item
4.4.2.3

The character data item is as follows:

[*character-prefix*] *character-string* [*character-count*] [*character suffix*]

In the preceding definition, variables are defined as follows:

- *character-prefix*

  This variable specifies the character set used for the stored constant. It is specified as follows:

`A` or `a`	ASCII character set (default)
`C` or `c`	Control Data display code
`E` or `e`	EBCDIC character set

- *character-string*

  The default is a string of zero or more characters (enclosed in apostrophes) from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate one apostrophe.

- *character-count*

  The length of the field, in number of characters, into which the data item will be placed. 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; therefore, length is not necessarily an integral number of words. The character width is 8 bits for ASCII or EBCDIC, and 6 bits for control data display code.

If an asterisk is in the count field, the actual number of characters in the string is used as the count. Two apostrophes that are used to represent one apostrophe are counted as one character.

If the base is mixed, CAL assumes that the count is decimal. See appendix A, page, 189 for information on the `base` pseudo instructions.

- *character-suffix*

  This variable specifies justification and fill of the character string as follows:

`H` or `h`	Left-justified, blank-filled (default)
`L` or `l`	Left-justified, zero-filled
`R` or `r`	Right-justified, zero-filled
`Z` or `z`	Left-justified, zero-filled, at least one trailing zero character guaranteed

The following example illustrates character data items:

```
DATA   A'ERROR IN DSN'   ; ASCII character set left justified and
                         ; blank fill by default; two words
DATA   E'error in dsn'R  ; EBCDIC character set right justified,
                         ; zero filled; stored in two words.
DATA   'Error'           ; Default ASCII character set left
                         ; justified and blank filled by default
                         ; stored in one word.
```

Literals
4.4.3

Literals are read-only data items whose storage is controlled by CAL. Specifying a literal lets you implicitly insert a constant value into memory. The actual storage of the literal value is the responsibility of the assembler. Literals can be used only in expressions because the address of a literal, rather than its value, is used.

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 of memory known as the *literals section*. Subsequent references to a literal value do not produce multiple copies of the same literal.

Because literals can map into the same location in the literals section, CAL checks for the presence of matching literals before new entries are added. This check is made bit by bit. If the current string is identical to any string currently stored in the literals section, CAL maps that string to the location of the matching string. If the current string is not identical to any of the strings currently stored, the current string is considered to be unique, and is assigned a location in the literals section.

The following special syntaxes are in effect for literals:

- Literals always have the following attributes:
  - Relocatable (relative) to a constant section
  - Word (address)
- Literals cannot be specified as character strings of zero bits. The actual constant within a literal must have a bit length greater than 0. In actual use, you must specify at least one 6-bit character for the CDC character set or one 8-bit character for the ASCII (default) and EDCDIC character sets.
- By default, literals always fall on full-word boundaries. Trailing blanks are added to fill the word to the next word boundary.

When used as an element of an expression, a literal is defined as follows:

=data-item

A data item for literals is the same as data items for constants. Data items for constants are described in subsection 4.4.2, page 82.

Single-precision literals are stored in one 64-bit word (default). Double-precision literals are stored in two 64-bit words. The following example shows how literals can be specified with single or double precision:

```
        CON    =1.5         ; Single-precision literal
        CON    =1.sD1       ; Double-precision literal
```

Figure 6 illustrates how the ASCII character `a` is stored by either of the following instructions (^ represents a blank character):

```
CON   =‘a’H

CON   =‘a’
```

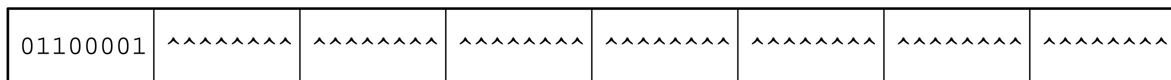


Figure 6. ASCII character with left-justification and blank-fill

Figure 7 illustrates how the ASCII character `a` is stored by any of the following instructions (^ represents a blank character):

```
CON   =‘a’L

CON   ‘a’R

CON   –‘a’S
```

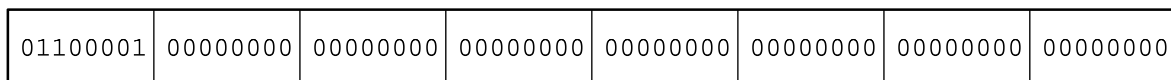


Figure 7. ASCII character with left-justification and zero-fill

Figure 8 illustrates how the ASCII character `a` is stored by the following instruction (^ represents a blank character):

```
CON   =‘a’R
```

This example illustrates how the ASCII character `a` is stored when `=‘a’R` is specified.

00000000	00000000	00000000	00000000	00000000	00000000	00000000	01100001

Figure 8. ASCII character with right-justification and zero-fill

Figure 9 illustrates how the ASCII character `a` is stored by the following instruction (^ represents a blank character):

```
CON   =‘a’*R
```

01100001	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 9. ASCII character with right-justification in 8 bits

The three character sets available to CAL are declared as follows:

```
CON   =‘A’       ; 8-bit ASCII character.

CON   =A‘A’      ; 8-bit ASCII character.

CON   =C‘A’      ; 6-bit CDC character.

CON   =E‘A’      ; 8-bit EBCDIC character.
```

The following example illustrates the use of the `H`, `L`, `R`, or `Z` options when specifying literals:

```
CON   =‘AB’3        ; Left-justified with one blank-padded on the
                    ; right (default).
CON   =‘AB’3H       ; Left-justified with one blank-padded on the
                    ; right (default).
CON   =‘AB’6R       ; Right-justified, filled with four leading
                    ; zeros.
CON   =‘AB’6Z       ; Left-justified, padded with four trailing
                    ; zeros
```

Special elements
4.5

Special elements are used to obtain the current value of the location counter, the origin counter, the word pointer, and the parcel pointer. Special elements can occur as elements of expressions. For a description of expression elements, see subsection 4.7, page 94. The origin, location, word-bit-position, and parcel-bit-position counters are described in section 3, page 33.

Elements that have special meanings to the assembler are described as follows:

- `*`. Location counter.

  The asterisk (`*`) denotes a value equal to the current value of the location counter with parcel-address attribute and absolute, immobile, or relocatable attributes. The location counter is absolute if the `LOC` pseudo instruction modified it by using an expression that has a relative attribute of absolute. The location counter is immobile if it is relative to either a `STACK` or `TASKCOM` section. The location counter is relocatable in all other cases.

- `*A` or `*a`. Absolute location counter.

  The `*A` or `*a` denotes a value equal to the current value of the location counter with parcel-address and absolute attributes.

- `*B` or `*b`. Absolute origin counter.

  The `*B` or `*b` denotes a value equal to the current value of the origin counter relative to the beginning of the section with parcel-address and absolute attributes.

- `*O` or `*o`. Origin counter.

  The `*O` or `*o` denotes a value equal to the current value of the origin counter relative to the beginning of the current section. The origin counter has an address attribute of parcel. If the current section is a section with a type of `STACK` or `TASKCOM`, it has an immobile attribute. In all other cases, it has a relative attribute of relocatable.

- `*W` or `*w`. Word pointer.

  The `*W` or `*w` denotes a value equal to the current value of the word-bit position counter with absolute and value attributes. `*W` is relative to the word and the word-bit-position counter is almost always equal to 0, 16, 32, or 48. CAL issues a warning message when the word-bit-position counter has a value other than 0 (not pointing at a word boundary) and is used in an expression.

- `*P` or `*p`. Parcel pointer.

  The `*P` or `*p` denotes a value equal to the current value of the parcel-bit-position counter with absolute and value attributes. The range of possible values for `*P` is 0 through 15. CAL issues a warning message when the parcel-bit-position counter has a value other than 0 (not pointing at a parcel boundary) and is used in an expression. The following statement defines where you are within a parcel, and it is almost always 0:

  ```
  SYM1 = *P
  ```

Element prefixes

4.6

A symbol, constant, or special element can be prefixed by an element prefix (`P.` or `p.` for parcel or `W.` or `w.` for word) causing the value to assume parcel-address or word-address attributes, respectively, in the expression in which the reference appears.

A prefix does not permanently alter the attribute of a symbol. A prefix only effects the current reference.

Parcel-address prefix
4.6.1

A symbol, special element, or constant can be prefixed by `P.` or `p.` to specify the attribute of parcel address. If a symbol (*sym*) has the attribute of word address, the value of `P.`*sym* or `p.`*sym* is the value of *sym* multiplied by 4. Each Cray word is divided into 4 parcels that are designated as `a`, `b`, `c`, and `d`. Each parcel has a 2-bit value associated with it; 00_2 for `a`, 01_2 for `b`, 10_2 for `c`, and 11_2 for `d`. To find the exact parcel being addressed, multiply the word address by 4. For example, the following word-address attributes are translated into parcel-address attributes:

Word	Equation	Value	Parcel representation
2	2x4	O'10	2a
4	4x4	O'20	4a
0	0x4	O'0	0a

A `P.` or `p.` specified for an element with value-address attribute does not cause the value to be multiplied by 4; however, the `P.` or `p.` prefix can be used to assign the parcel-address attribute to the element.

A `P.` or `p.` specified for an element with parcel-address attribute does not alter its characteristics.

Figure 10, page 92, shows the octal numbering of parcels `a`, `b`, `c`, and `d` in a 6-word block.

	Parcel a	Parcel b	Parcel c	Parcel d
Word 0	0	1	2	3
Word 1	4	5	6	7
Word 2	10	11	12	13
Word 3	14	15	16	17
Word 4	20	21	22	23
Word 5	24	25	26	27

Figure 10. Word/parcel conversion for 6 words

The following example illustrates the use of the parcel-address prefix:

```
SYM1    =     *            ; SYM1 is equal to the location counter with
                           ; parcel and relocatable attributes.
        S1    SYM1         ; Register S1 gets the relocatable parcel
                           ; address of SYM1.
        S1    P.SYM1       ; The same value that was generated by the
                           ; last statement is produced.
```

Word-address prefix
4.6.2

A symbol, special element, or constant can be prefixed by `W.` or `w.` to specify the attribute of word address. If a symbol (*sym*) has the attribute of parcel address, the value of `W.`*sym* or `w.`*sym* is the value of *sym* divided by 4. When converting from parcel-address attribute to a word-address attribute, divide the parcel address by 4. When the conversion is completed, the result is always understood to be pointing at parcel `a`.

If the parcel address is not pointing at a word boundary, CAL issues a warning message and truncates the division to a word boundary. For example, the following parcel address attributes are converted into word-address attributes:

Parcel representation	Value	Equation	Word	Truncation warning
`0c`	2	2/4	0	Yes
`3a`	14	14/4	3	No
`5c`	26	26/4	5	Yes
`0a`	0	0/4	0	No
`6a`	30	30/4	0	No

A `W.` or `w.` prefix specified for an element with a value-address attribute does not cause the value to be divided by 4. However, the `W.` or `w.` prefix can be used to assign the word-address attribute to the element.

A `W.` or `w.` prefix specified for an element with a word-address attribute does not alter its characteristics.

The following example illustrates the use of a word-address prefix:

```
SYM2    =     W.*            ; Word and relocatable attributes.
        A0    W.ADDR
        A4    W.BUFF+0'100
```

Expressions
4.7

The result and operand fields for many source statements contain expressions. An *expression* consists of one or more terms joined by special characters referred to as adding operators (*add-operator*). A *term* consists of one or more special elements, constants, symbols, or literals (*prefixed-element*) joined by multiplying operators (*multiply-operator*). Figure 11 diagrams an expression and Figure 12 diagrams a term.

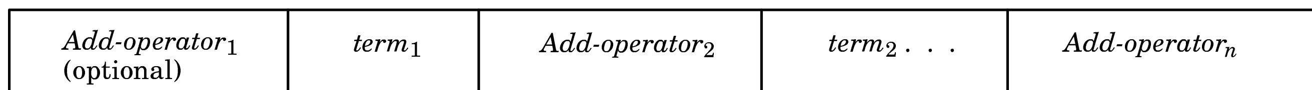


Figure 11. Diagram of an expression

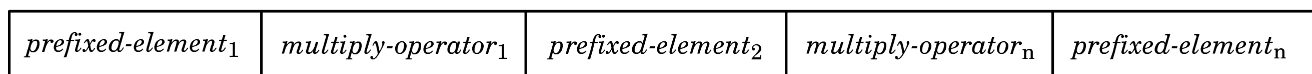


Figure 12. Diagram of a term

An expression is defined as follows:

embedded-argument or [*add-operator*] *term* { add-operator *term* }

The variables listed in the previous definition are defined in the subsections that follow.

Add-operator
4.7.1

An *add-operator* joins two terms in an expression or precedes the first term of an expression. *Add-operators* include the plus sign (+) and the minus sign (–) and perform addition and subtraction.

Terms
4.7.2

A term consists of one or more *prefixed-elements* joined by special characters referred to as *multiply-operators*. The multiply-operators complete all multiplication (*) and division (/) before the add-operators complete addition or subtraction.

A *term* is defined as follows:

prefixed-element { *multiply-operator prefixed-element* }

The following general rules apply:

- Only one *prefixed-element* within a term can have a relative attribute of immobile or relocatable. All other *prefixed-elements* in that term must have relative attributes of absolute.
- A *prefixed-element* with a relative attribute of external must be the only *prefixed-element* of the term. If preceded by an *add-operator*, that operator must be a `+`.
- The *prefixed-element* to the right of a slash `(/)` must have a relative attribute of absolute.
- A term that contains a slash `(/)` must have an attribute of absolute up to the point at which the `/` is encountered (see subsection 4.7.2.3, page 97).
- Division by 0 produces an error.

The following example illustrates the use of terms:

```
SYM     =     *            ; Relocatable and parcel attributes.
        S1    SYM1         ; One term within an expression.
        S2    SYM*1+1      ; Two terms within an expression.
        S3    1*2*3/4      ; Every prefixed-element preceding a / must
                           ; have the attribute of absolute and the
                           ; prefixed-element following the / must have
                           ;  an attribute of absolute.
```

The following are examples of terms:

Term	Description
`SIGMA*5`	Two elements, `SIGMA` and `5`, are joined by a multiplying operator.
`DELTA`	A single-element term.

4.7.2.1 *Prefixed-elements*

A *prefixed-element* is defined as follows:

```
[ # ] [ element-prefix ] element
```

The variables in the previous definition are defined as follows:

- *complement character* (`#`)

  If an element is prefixed with the *complement character* (`#`), the element itself must have a relative attribute of absolute.

- *element-prefix*

  If an element is prefixed with an *element-prefix*, the attribute of the element is as follows:

`P. or p.`	Parcel-address attribute
`W. or w.`	Word-address attributes

  For more information about *element-prefixes*, see subsection 4.6, page 90.

- *element*

  An element can be a special element, constant, symbol, or literal. Elements can be optionally preceded by a complement character (`#`) or an *element-prefix* (`P.` or `W.`). For more information about *element*, see subsection 4.5, page 89.

The following are examples of elements:

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

Multiply-operator 4.7.2.2

A *multiply-operator* joins two *prefixed-elements*. *Multiply-operators* are the asterisk (*) which specifies multiplication and the slash (/) which specifies division.

Term attributes 4.7.2.3

Each *prefixed-element* in a term has a relative and an address attribute associated with it. CAL assigns relative and address attributes to the entire term by evaluating each *prefixed-element* in the term.

The relative and address attributes for a term vary as CAL evaluates each *prefixed-element* in the term. The final attribute of the term is the attribute in effect when the final (rightmost) element of the term is evaluated. As CAL encounters each *prefixed-element* in the left-to-right scan of a term, it assigns an attribute to the term based on the *multiply-operator* (if any) preceding the *prefixed-element*, the attribute of any previous partial term, and the attribute of the *prefixed-element* currently being evaluated.

Relative attributes (the *prefixed-elements* and *multiply-operators* that compose a term) determine the relative attributes of the term.

CAL assigns every term a relative attribute determined by the following rules:

- A term assumes the attributes of *absolute* if every *prefixed-element* is absolute. For example:

  ```
  2*4/3*4
  ```

  In the above example, absolute (`2`) `*` absolute (`4`) is evaluated as absolute. Absolute (`2*4`) `/` absolute (`3`) is evaluated as absolute. Absolute (`2*4/3`) `*` absolute (`4`) is evaluated as absolute.

- A term assumes an attribute of *immobile* if it contains one *prefixed-element* with immobile attributes, zero or more *prefixed-elements* with absolute attributes, and no *prefixed-elements* with relocatable or external attributes. Thus, an immobile term can contain one immobile *prefixed-element* with the remaining *prefixed-elements* being absolute. For example:

  ```
  STKSYM*3
  ```

  In the above example, immobile (`STKSYM`) `*` absolute (`3`) is evaluated as immobile.

- A term assumes an attribute of *relocatable* if it contains one *prefixed-element* with relocatable attributes, zero or more *prefixed-elements* with absolute attributes, and no *prefixed-elements* with immobile or external attributes. Thus, a relocatable term can contain one relocatable *prefixed-element* with the remaining *prefixed-elements* being absolute.

  ```
  2*SYM1*2
  ```

  In the above example, absolute (`2`) `*` relocatable (`SYM1`) is evaluated as relocatable. Relocatable (`2*SYM1`) `*` absolute (`2`) is evaluated as relocatable.

- A term assumes the attribute of *external* if it consists of one *prefixed-element* and the *prefixed-element* is external. For example:

  ```
  EXT1
  ```

  In the above example, one external (`EXT1`) element is evaluated as external.

  ```
  EXT2*SYM1
  ```

  In the above example, external (`EXT2`) `*` relocatable (`SYM1`) produces an error.

  In the following example, Absolute (`4`) `*` relocatable (`SYM1`) is evaluated as relocatable; relocatable (`4*SYM1`) `/ 4` produces an error:

  ```
  4*SYM1/4
  ```

  All *prefixed-elements* to the left of the `/` must have a relative attribute of absolute. See general rules for terms in subsection 4.7.2, page 95.

CAL assigns one of the following address attributes to every term:

- Parcel-address
- Word-address
- Value

Figure 13, page 100, indicates how address attributes are assigned to terms and partial terms. *Vterm*, *Pterm*, and *Wterm* denote the attribute of the partial term resulting from all elements evaluated before the current element. In Figure 13, `P`, `W`, and `V` denote an element being incorporated into the term and having an attribute of *parcel-address*, *word-address*, or *value*, respectively.

If a partial term has the address attribute of the left column and is multiplied or divided by a *prefixed-element* with the address attribute of the top horizontal row, the resulting attribute is determined at the intersection of the column and row by the arithmetic operator position in the upper-left corner of the table.

The results for multiplication and division are given in the top (`*`) and bottom (`/`) halves of each box on the chart, respectively. For example, if partial term *Vterm* is multiplied by a *prefixed-element* with an address attribute of word, the address attribute for the new partial term is word.

A 2-digit value following an address attribute indicates that although a result is specified, a warning message is issued that corresponds to the 2-digit superscript. For example, if the partial term *interm* is divided by a *prefixed-element* with an address attribute of parcel, the result is value and message 84 is issued as follows:

```
Partial term with value address is divided by parcel element
```

See appendix B, page 281, for the text associated with messages.

* /	V	P	W
Vterm	V / V	P / V84	W / V86
Pterm	P / P	P80 / V	V82 / V87
Wterm	W / W	V81 / V85	V83 / V

Rows: Partial term; Columns: 2nd term

V – Value
P – Parcel
W – Word
nn – Warning message number

Figure 13. Address attribute assignment chart

Expression evaluation
4.8

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 multiply-operator is encountered. Expressions are defined as follows:

embedded-argument |[*add-operator*] *term* { *add-operator* *term* }

Note: The *embedded-argument* is intended for use with macros and opdefs and should not be included in expressions. Although the *embedded-argument* is syntactically correct, the CAL expression evaluator cannot evaluate expressions that contain *embedded-arguments*. See the following examples.

```
sym1   =    1          ; Valid expression
sym2   =    (1)        ; Syntactically correct, but CAL issues
                       ; error message.
```

An *embedded-argument* can be any *argument-character* that is enclosed in parentheses. For example:

```
        MACRO
        FRED    p1,p2
ABC     =       p1
        S1      ABC*p2
FRED    ENDM
        FRED    (1+2),3     ; (1+2) is the embedded argument
```

When a complete term is evaluated, it is added or subtracted from the sum of the previous terms. CAL does not check for overflow and underflow.

The assembler treats each element as a 64-bit twos complement integer. Character constants are left- or right-justified within a field width equal to the destination field. If the field width is shorter than the length of the character constant, a warning message is issued. Elements are complemented in the rightmost bits of a field width equal to the destination field.

Note: CAL processes *floating-constants* as expected when they are specified as one uncomplemented *prefixed-element* within an expression. If *floating-constants* are used in any other way, an appropriate warning message is issued and integer arithmetic is used to evaluate the expression. CAL processes the *floating-constants* within the expressions of the following examples as expected:

```
A       CON     1.0
B       CON     -1.0
C       CON     4.5
D       CON     .3
E       CON     -.75
```

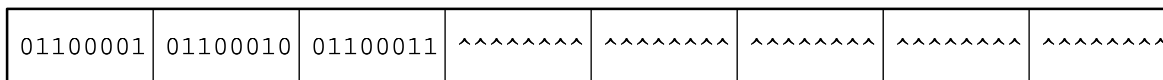
CAL issues an appropriate warning message and evaluates the *floating-constants* within the expressions of the following examples by using integer arithmetic:

```
G       CON     1.0+2.0
H       CON     -1*3.4
I       CON     -#1.0
```

This example demonstrates how the result of a VWD with a 9-bit destination field is stored; ^ represents a blank space.

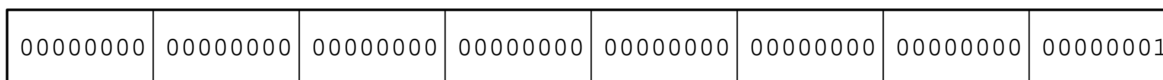
```
        VWD     D’9/’abc’+1   ; The terms of the expression ‘abc’ and 1
```

Figure 14 and Figure 15 contain the binary representations of the ASCII character strings "abc" and 1, respectively.



01100001	01100010	01100011	^^^^^^^^	^^^^^^^^	^^^^^^^^	^^^^^^^^	^^^^^^^^

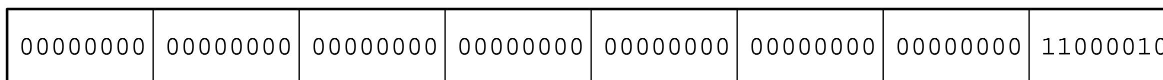
Figure 14. 64-bit binary representation of ASCII abc, left-justified



00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000001

Figure 15. 64-bit binary representation of 1

Because the character constant is left-justified by default within a field width equal to the 9 bits specified in the example, the 64-bit representation of "abc" is actually as in Figure 16.



00000000	00000000	00000000	00000000	00000000	00000000	00000000	11000010

Figure 16. Binary representation of ASCII abc, right-justified in 9 bits

CAL adds the value `1` (Figure 15, page 102) to the value shown in Figure 16, page 102 (`011000010`), and stores it in the destination field (see Figure 17). CAL issues a warning message stating that the character string `"abc"` has been truncated. The destination field contains a value of 303 (`011000011`).

011000011

Figure 17. Result of `VWD` with 9-bit destination field

The following example demonstrates that elements are complemented in the rightmost bits of a field width equal to the destination field:

```
        VWD    D'4/#1+1   ; The terms of the expression are the
                          ; complement of 1 and the value 1.  The
                          ; destination field is 4-bits wide.
```

Figure 18 and Figure 19 contain the complement of `1` and the binary representation of the value 1 (0001), respectively.

11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111110

Figure 18. 64-bit binary representation of the complement of 1

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000001

Figure 19. 64-bit binary representation of 1

Figure 20 shows that the actual value of the complement of 1 is stored in the rightmost bits of a word in memory.

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00001110

Figure 20. Binary representation of the complement of 1 stored in the rightmost bits of a 4-bit field

The binary value 1110 (Figure 20) is stored in the destination field, and CAL adds the value 1 to the destination field; the result (1111) is shown as the rightmost 4 bits as in Figure 20 and is stored as shown in Figure 21.

1111

Figure 21. Result of VWD with 4-bit destination field

Evaluating immobile and relocatable terms with coefficients 4.8.1

An immobile term has one immobile *prefixed-element*, no relocatable or external *prefixed-elements*, and zero or more absolute *prefixed-elements*. A relocatable term has one relocatable *prefixed-element*, no immobile or external *prefixed-elements*, and zero or more absolute *prefixed-elements*.

An immobile term has an associated 64-bit integer coefficient equal to the value of the term obtained when a 1 is substituted for the immobile element. The value of an immobile term is the value of the immobile element multiplied by the coefficient.

A relocatable term has an associated 64-bit integer coefficient equal to the value of the term obtained when a 1 is substituted for the relocatable element. The value of a relocatable term is the value of the relocatable element multiplied by the coefficient.

Each section has two relative section coefficients, one represents an immobile relative attribute and one represents a relocatable relative attribute. These relative section coefficients are initialized to 0 before the evaluation of each expression. As each term is evaluated within an expression, the coefficient of the term is either added to or subtracted from the corresponding

coefficient of the corresponding section depending on the sign immediately preceding the term. When each term within an expression has been evaluated, the expression is assigned a relative attribute as follows:

- Absolute; if the expression contains no external terms and all of the coefficients for all of the sections are 0.
- Immobile; if the expression contains no external terms and all of the coefficients for all of the sections are 0, except for one immobile coefficient that must have a value of 1. The expression is immobile relative to the section with the coefficient of 1.
- Relocatable; if the expression contains no external terms and all of the coefficients for all of the sections are 0 except for one relocatable coefficient that must have a value of 1. The expression is relocatable relative to the section with the coefficient of 1.
- External; if the expression contains one external term and all of the coefficients for all of the sections are 0.
- Not valid; all other cases.

For example, if `SYMBOL` is assumed to be relocatable, `SYMBOL*2+1-SYMBOL` is considered a valid expression when evaluated by CAL. Because `SYMBOL` is relocatable, substituting 1 for `SYMBOL` generates three terms (`1*2`, `+1`, and `-1`). The first term (`1*2`) includes the relocatable term `SYMBOL`. A value of 2 is stored with the coefficient maintained by CAL for the relocatable section to which `SYMBOL` is relative. The second term (`+1`) is absolute and does not affect the evaluation of the relocatable coefficient. The third term (`-1`) includes the relocatable term `SYMBOL`. A 1 is subtracted from the coefficient maintained by CAL for the relocatable section `SYMBOL`.

When the entire term is evaluated, the coefficient associated with the relocatable term `SYMBOL` equals 1. Because all relocatable terms within the expression are relative to one section and the final coefficient of the section is 1, the expression is relocatable relative to that section.

Every relocatable symbol is relative to a section. All sections contain an initial coefficient of 0 before expression evaluation. The operator immediately preceding a relocatable term is the operator associated with that term. For example, the coefficient for `SYMBOL` is maintained as `-1`. When the sign of a coefficient is

not indicated, it is assumed to be positive. The coefficient for `SYMBOL*1` is maintained as `+1*1`. If `1a` (`100`) is substituted for `SYMBOL` in the following expression; the binary that will be evaluated is `100*010+001-100`:

```
SYMBOL*2+1-SYMBOL
```

CAL evaluates the string from left to right. The following partial results are obtained:

```
100*010=1000
1000+0001=1001
1001-0100=0101=1b
```

The final result (`1b`) is the result that you would expect to be generated. The following example demonstrates the correct and incorrect use of a relocatable term:

```
       IDENT
SYMBOL =     *                 ; SYMBOL is given a value equal to the
                               ; current location counter.
       S1    SYMBOL*2+1-SYMBOL ; When evaluated, this expression
                               ; produces a value equal to the current
                               ; location counter plus 1.  The value is
                               ; relocatable.
       S1    SYMBOL*2+1        ; When evaluated, this expression
                               ; produces a value equal to twice the
                               ; current location counter plus 1.  The
                               ; value is not relocatable.  CAL
                               ; produces an error message.
       END
```

In the preceding example, the term `SYMBOL*2+1` is not relocatable because the results generated depend on the location of the module by the loader. If the loader puts the module at 400, `SYMBOL*2+1=801`. If the loader puts the module at 200, `SYMBOL*2+1=401`. If a term is evaluated and found to be not relocatable, CAL issues an error-level diagnostic message.

The following example illustrates the use of relocatable terms:

```
        IDENT       TEST
SNAME1  SECTION
SYMBOL1 BSS         4
SYMBOL2 =           w.*
        BSS         5
SNAME2  SECTION
SYMBOL3 BSS         3
        SECTION     *
SYMBOL4 =           3*SYMBOL2+SYMBOL3-1SYMBOL2-2*SYMBOL1
        END
```

In the previous example, the expression `3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1` contains five terms, four of which are relocatable; it is evaluated as follows:

Term	Value of coefficient	Attribute
`3*SYMBOL2`	`3*1`	Relocatable (relative to `SNAME1`)
`+SYMBOL3`	`+1`	Relocatable (relative to `SNAME2`)
`-1`		`Absolute`
`-SYMBOL2`	`-1`	Relocatable (relative to `SNAME1`)
`2*SYMBOL1`	`-2*1`	Relocatable (relative to `SNAME1`)

The coefficients for the `SNAME1` and `SNAME2` sections were initialized to 0 before the expression was evaluated. The main section has a coefficient of 0. When the coefficients for the relocatable terms relative to `SNAME1` are evaluated, the result is `0` (`+3-1-2`). When the coefficients for the relocatable terms for `SNAME2` are evaluated, the result (`+1`) is 1.

`SYMBOL4` obtains a relative attribute of relocatable because one section in the expression has a coefficient of 1 (`SNAME2`) and all other sections (`SNAME1`) maintained for the expression have coefficients of 0. The final expression is relocatable relative to `SNAME2`, because `SNAME2` is the section with the coefficient of 1.

The address attribute of the expression is evaluated, as follows:

Term	Partial term	Attribute
3*SYMBOL2	Value*word	Word (see Figure 13, page 100)
+SYMBOL3	Word	Word (see Figure 13, page 100)
-1	Value	Value (see Figure 13, page 100)
-SYMBOL2	Word	Word (see Figure 13, page 100)
2*SYMBOL1	Value*word	Word (see Figure 13, page 100)

The address attribute for the entire expression is word. For a description of the manner in which *parcel-address*, *word-address*, and value attributes are assigned to entire expressions, see subsection 4.9, page 110.

The value of the expression 3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1 = O'7. It is calculated as follows:

Term	Result	Description
3*SYMBOL2	3*4=0'14	SYMBOL2 begins with word 4 in section SNAME1; 4 is substituted for SYMBOL2.
SYMBOL3	0	SYMBOL3 begins with word 0 in section SNAME2; 0 is substituted for SYMBOL3.
-1	-1	Term 3 is absolute; no substitution.
-SYMBOL2	-4	SYMBOL2 begins with word 4 in section SNAME1; 4 is substituted for SYMBOL2.
2*SYMBOL1	-2*0=0	SYMBOL1 begins with word 0 in section SNAME1; 0 is substituted for SYMBOL1.

When the values for the terms (O'14+0-1-4-0) are substituted for the (3*SYMBOL2+SYMBOL3-1-SYMBOL2-2*SYMBOL1) expression, the result is 7.

The following example illustrates the use of immobile terms:

```
        ident          test
taskc   section        taskcom
tcsym   bss            4
        section        *
symbol  =              taskc+tcsym-taskc
```

In the preceding example, the `taskc+tcsym−taskc` expression contains three terms, two that are relocatable and one that is immobile. The expression is evaluated as follows:

Term	Value of coefficient	Attribute
`taskc`	`+1`	Relocatable (relative to `taskc`)
`+tcsym`	`+1`	Immobile (relative to `taskc`)
`−taskc`	`−1`	Relocatable (relative to `taskc`)

The relative section coefficients for relocatable `taskc` and immobile `tcsym` were initialized to 0 before the expression was evaluated. When the coefficients for the relocatable terms relative to `taskc` are evaluated, the result (`+1−1=0`) is 0. When the coefficient for the immobile term (`tcsym`) is evaluated, the result (`+1`) is 1. Because the term with the relative attribute of immobile has the coefficient of 1, the entire expression is assigned a relative attribute of immobile.

The address attribute of the expression is evaluated as follows:

Term	Partial term	Attribute
`* taskc`	Word	word (see Figure 13, page 100)
`+tcsym`	Word	word (see Figure 13, page 100)
`−taskc`	Word	word (see Figure 13, page 100)

The address attribute for the entire expression is word. For a description of the manner in which *parcel-address*, *word-address*, and *value* attributes are assigned to entire expressions, see subsection 4.9, page 110.

The value of the expression `taskc+tcsym–taskc` is calculated as follows:

Term	Result	Description
`taskc`	`0`	`taskc` is assigned a value of 0 relative to the task common section `taskc`; 0 is substituted for `taskc`.
`+tcsym`	`0`	`tcsym` begins with word 0 in `taskcom` section `taskc`; `0` is substituted for `tcsym`.
`–taskc`	`0`	`taskc` is assigned a value of 0 relative to the task common section `taskc`; 0 is substituted for `taskc`.

When the values for the terms (`0+0–0`) are substituted for the expression (`taskc+tcsym–taskc`), the result is 0.

Expression attributes
4.9

To determine the expression attributes for a full expression, evaluate the terms within an expression. The assembler can assign the following attributes to an expression:

- Relative

  Relative attributes are classified as follows:
  - Absolute
  - Immobile
  - Relocatable
  - External

- Address

  Address attributes are classified as follows:
  - Parcel-address
  - Word-address
  - Value

Relative attributes 4.9.1

Every expression assumes one of the relative attributes as follows:

- An expression is *absolute* if no external terms are present and the coefficients of all other sections are 0.
- An expression is *immobile* if the coefficient is 0 for each section within the current module represented in the expression. The exception is when one section has a coefficient of +1 (positive relocation) and is immobile with respect to that expression.
- An expression is *relocatable* if the coefficient for every section within the current module represented in the expression is 0. The exception is when one section has a coefficient of +1 (positive relocation) and is relocatably associated with that expression. 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 it contains one external term and if the coefficients of all sections are 0. An expression error occurs if more than one external term is present. All external terms defined with the EXT pseudo instruction have a value of 0 associated with them.

The following are examples of relative attributes (see section 3, page 33, for a description of sections):

```
       IDENT  TEST
       EXT    EXT1
SNAME1 SECTION
SYM1   BSS    4
SYM2   =      W.*
       BSS    5
       SYM4   EXT1+SYM1       ; Illegal external term and relocatable
                              ; terms with coefficients of 1 in the
                              ; same expression.
       SYM5   EXT1+SYM1-SYM2  ; Legal; SYM1 (+1) and SYM2 (-1) cancel
                              ; each other and produce a coefficient
                              ; of 0 for the expression.  The value of
                              ; the expression EXT1+SYM1-SYM2 is 4
                              ; (0+0-4).
       END
```

Address attributes
4.9.2

Each expression assumes an address attribute as follows:

- An expression has a *parcel-address* attribute if at least one term has a *parcel-address* attribute and all other terms have *value* or *parcel-address* attributes.
- An expression has a *word-address* attribute if at least one term has a *word-address* attribute and all other terms have *value* or *word-address* attributes.
- All other expressions have *value* attributes. A warning message is issued if an expression has terms with both *parcel-address* and *word-address* attributes.

Truncating expression values
4.9.3

An expression value is truncated to the field size of the expression destination.

The following example illustrates expression value truncation:

```
SYM1   BSS    4
SYM2   =      -1         ; 64 bits
       VWD    5/-1       ; 5 bits
       VWD    3/5        ; 3-bit destination field, value of 5
       VWD    2/5        ; 2-bit destination field, value of 5,
                         ; truncation message issued.
       VWD    3/exp      ; 3-bit destination field, the range of
                         ; values is as follows: -4 ≤ exp ≥ 7.
```

A warning message is issued if the leftmost bits lost in truncation are not all 0's or all 1's with the leftmost remaining bit also 1 (that is, a negative quantity).

In the preceding example, truncation occurs in statement `VWD 5/-1` (see Figure 22), but an error message is not generated because the part that was truncated included all 1's and the leftmost bit of the 5-bit field is also a 1.

11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
Truncated							Result

Figure 22. 64-bit binary representation of –1

11111

Figure 23. Truncated value of –1 stored in a 5-bit field

Truncation occurs in statement `VWD 3/5`. An error message is not generated, because the truncated part was all 0's. The result is truncated and stored as shown in Figure 24.

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000101
Truncated							

Figure 24. 64-bit binary representation of 5

101

Figure 25. Truncated value of 5 stored in a 3-bit field

Truncation occurs in statement `VWD 2/5`. CAL generates a warning message, because a combination of 1's and 0's is truncated. The result is truncated and stored as shown in Figure 26.

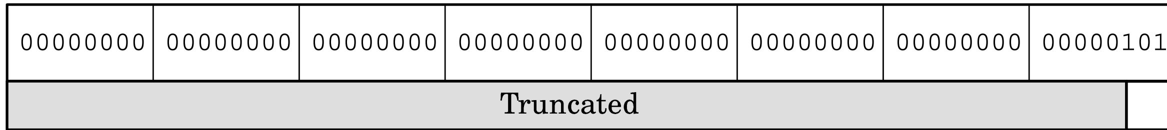


Figure 26. 64-bit binary representation of 5

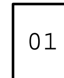


Figure 27. Truncated value of 5 stored in a 2-bit field

If the values generated by the statement in `VWD 3/`*exp* are in the range from –4 through 7, a warning message is not generated.

If a message of error-level is issued for an expression it causes the expression to have a relative attribute of absolute, an address attribute of value, and a value of 0.

The following are examples of expressions:

Expression	Description
`ALPHA`	An expression consisting of one 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, `P` and `Q` are immobile symbols in the same section, `R` and `S` are relocatable symbols in the same section, `COM` is relocatable in a common section, `X` and `Y` are external, and `A` and `B` are absolute. The location counter is currently in the section containing `R` and `S`.

The following expressions are absolute:

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

The following expressions are immobile:

`P+B`	
`Q+3`	
`COM+P-Q`	`P` and `Q` cancel.
`X+P`	Error; external and immobile.
`R+P`	Error; relocatable and immobile.
`P+Q`	Error; immobile coefficient of 2.
`Q/16*16`	Error; division of immobile element is illegal.

The following expressions are relocatable:

`*`	
`w.*+B`	
`R+2`	
`COM+R-S`	`R` and `S` cancel.
`3**-R-S`	`3**` cancels `-R` and `-S`.
`X+R`	Error; external and relocatable.
`R+S`	Error; relocation coefficient of 2.
`Q+S`	Error; immobile and relocatable.
`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.