Chapter 3
EXPRESSIONS

3.1 INTRODUCTION

An expression represents a value which is used as an operand in an Assembler instruction or directive. An expression is a combination of symbols, constants, operators, and parentheses. Expressions may contain user-defined labels and their associated integer or floating point values, and/or any combination of integers, floating point numbers, or ASCII literal strings. In general, white space (a blank or tab) is not allowed between the terms and operators of an Assembler expression. Expressions otherwise follow the conventional rules of algebra and boolean arithmetic.

3.2 ABSOLUTE AND RELATIVE EXPRESSIONS

An expression may be either **relative** or **absolute**. An absolute expression is one which consists only of absolute terms, or is the result of two relative terms with opposing signs. A relative expression consists of a relative term by itself or only in combination with absolute terms.

When the Assembler is operating in relative mode all address expressions must adhere to the above definitions for absolute or relative expressions. This is because only these types of expressions will retain a meaningful value after program relocation. For example, when relative terms are paired with opposing signs, the result is the difference between the two relative terms, which is an absolute value. However, if two positive relative terms are added together the result is unpredictable based on the computed values of the terms at relocation time.

3.3 EXPRESSION MEMORY SPACE ATTRIBUTE

A symbol is associated with either an integer or a floating point value which is used in place of the symbol during the expression evaluation. Each symbol also carries a memory space attribute of either **X**, **Y**, **L**, **P**rogram, **E**MI, or **N**one. Constants and floating point expressions always have a memory space attribute of **N**one. The result of an expression will always have a memory space attribute associated with it. The unary logical negate operator, relational operators, and some functions return values that have a memory space attribute of **N**. The result of an expression that has only one operand (and possibly the unary negate or unary minus operator) always has the memory attribute of that operand. Ex-

pressions that involve two or more operands and operators other than those mentioned above derive the memory space attribute of the result by examining the operands on the left and right side of an operator as shown in the following chart:

		Left Operand Memory Space Attribute				
		X	**Y**	**L**	**P**	**EN**
Right Operand Memory Space Attribute	**X**	X	*	X	*	*X
	Y	*	Y	Y	*	*Y
	L	X	Y	L	*	*L
	P	*	*	*	P	*P
	E	*	*	*	*	EE
	N	X	Y	L	P	EN

* = Represents an illegal operation that will result in an error.

Notice that **L** memory space is regarded as a union of both **X** and **Y** space. In expressions that have one element that has a memory space attribute of **L** and another element with a memory space attribute of either **X** or **Y**, the result will have the more restrictive memory space attribute (**X** or **Y**).

The memory space attribute is regarded by the Assembler as a type, in the same sense that high level languages use type for variables. Symbols that are assigned memory space attributes of **X**, **Y**, **L**, **P**, or **E** are assumed to be addresses and therefore can only have values between zero and the maximum address of the target processor. Only symbols that have a memory space attribute of **N** can have values greater than the maximum address of the target machine.

Memory space attributes become important when an expression is used as an address. Errors will occur when the memory space attribute of the expression result does not match the explicit or implicit memory space specified in the source code. Memory spaces are explicit when the address has any of the following forms:

X:<address expression>
Y:<address expression>
L:<address expression>
P:<address expression>
E:<address expression>

The memory space is implicitly **P** when an address is used as the operand of a **DO**, branch, or jump-type instruction.

Expressions used for immediate addressing can have any memory space attribute.

3.4 INTERNAL EXPRESSION REPRESENTATION

Expression value representation internal to the Assembler is dependent on the word size of the target processor. The Assembler supports a word and a double word integer format internally. The actual storage size of an expression value is dependent upon the magnitude of the result, but the Assembler is capable of representing signed integers up to 64 bits in length. These longer integer representations are useful when performing data initialization in **L** memory space.

Internal floating point representation is almost entirely dependent upon the host environment, but in general floating point values are stored in double precision format. This means that there are ordinarily 64 bits of storage allotted for a floating point number by the Assembler, with 11 bits of exponent, 53 bits of mantissa, and an implied binary point.

3.5 CONSTANTS

Constants represent quantities of data that do not vary in value during the execution of a program.

3.5.1 Numeric Constants

Numeric constants can be in one of three bases:

Binary Binary constants consist of a percent sign (**%**) followed by a string of binary digits (0,1).

Example: %11010

Hexadecimal Hexadecimal constants consist of a dollar sign (**$**) followed by a string of hexadecimal digits (0-9, A-F, a-f).

Example: $12FF, $12ff

Decimal Decimal constants can be either floating point or integer. Integer decimal constants consist of a string of decimal (0-9) digits optionally preceded by a grave accent (`). Floating point constants are indicated either by a preceding, following, or included decimal point or by the presence of an upper or lower case ‘E’ followed by the exponent.

Example:

12345 (integer)
6E10 (floating point)

.6 (floating point)
2.7e2 (floating point)

A constant may be written without a leading radix indicator if the input radix is changed using the **RADIX** directive. For example, a hexadecimal constant may be written without the leading dollar sign (**$**) if the input radix is set to16 (assuming an initial radix of 10). The default radix is10. See Chapter 6 on the **RADIX** directive for more information.

3.5.2 String Constants

String constants that are used in expressions are converted to a concatenated sequence of ASCII bytes (right aligned), as shown below. Strings used in expressions are limited to the long word size of the target processor; subsequent characters in the string are ignored. Null strings (strings that have no characters) have a value of 0.

String constants greater than the maximum number of characters can be used in expressions, but the Assembler will truncate the value and will use only those characters that will fit in a DSP long word. In this case, a warning will be printed. This restriction also applies to string constants using the string concatenation operator. Handling of string constants by the **DC** and **DCB** directives is an exception to this rule; see Chapter 6 for a description.

Examples:

'ABCD'	($41424344)
"'79'	($00273739)
'A'	($00000041)
''	($00000000) - null string
'abcdef'	($61626364)
'abc'++'de'	($61626364)

3.6 OPERATORS

Some of the Assembler operators can be used with both floating point and integer values. If one of the operands of the operator has a floating point value and the other has an integer value, the integer will be converted to a floating point value before the operator is applied and the result will be floating point. If both operands of the operator are integers, the result will be an integer value. Similarly, if both the operands are floating point, the result will be a floating point value.

3.6.1 Unary operators

plus	(**+**)	
minus	(**-**)	
one’s complement	(**~**)	- Integer only
logical negate	(**!**)	

The unary plus operator returns the value of its operand.

The unary minus operator returns the negative of its operand.

The one's complement operator returns the one's complement of its operand. It cannot be used with a floating point operand.

The unary logical negation operator returns an integer 1 (memory space attribute **N**one) if the value of its operand is 0 and will return a 0 otherwise. For example, if the symbol BUF had a value of 0, then **!**BUF would have a value of 1. If BUF had a value of 1000, **!**BUF would have a value of 0.

3.6.2 Arithmetic operators

addition	(**+**)
subtraction	(**-**)
multiplication	(*****)
division	(***/***)
mod	(**%**)

The addition operator yields the sum of its operands.

The subtraction operator yields the difference of its operands.

The multiplication operator yields the product of its operands.

The divide operator yields the quotient of the division of the first operand by the second. For integer operands the divide operation will produce a truncated integer result.

The mod operator applied to integers will yield the remainder from the division of the first operand by the second. If the mod operator is used with floating point operands, the mod operator will apply the following rules:

```
Y % Z = Y     if Z = 0
      = X     if Z <> 0
```

where X has the same sign as Y, is less than Z, and satisfies the relationship:

$$Y = i * Z + X$$

where i is an integer.

3.6.3 Shift operators

shift left	(**<<**)	- Integer only
shift right	(**>>**)	- Integer only

The shift left operator causes the left operand to be shifted to the left (and zero-filled) by the number of bits specified by the right operand.

The shift right operator causes the left operand to be shifted to the right by the number of bits specified by the right operand. The sign bit will be extended.

Shift operators cannot be applied to floating point operands.

3.6.4 Relational operators

less than	(**<**)
less than or equal	(**<=**)
greater than	(**>**)
greater than or equal	(**>=**)
equal	(**==**)
not equal	(**!=**)

Relational operators all work the same way. If the indicated condition is true, the result of the expression is an integer 1. If it is false, the result of the expression is an integer 0. In either case, the memory space attribute of the result is **N**one.

For example, if D has a value of 3 and E has a value of 5, then the result of the expression D**<**E is 1, and the result of the expression D**>**E is 0. Each operand of the conditional operators can be either floating point or integer. Test for equality involving floating point values should be used with caution, since rounding error could cause unexpected results. Relational operators are primarily intended for use with the conditional assembly **IF** directive, but can be used in any expression.

3.6.5 Bitwise operators

AND	(**&**)	- Integer only
OR	(**|**)	- Integer only
exclusive OR	(**^**)	- Integer only

The bitwise AND operator yields the bitwise AND function of its operands.

The bitwise OR operator yields the bitwise OR function of its operands.

The bitwise exclusive OR operator yields the bitwise exclusive OR function of its operands.

Bitwise operators cannot be applied to floating point operands.

3.6.6 Logical operators

Logical AND	(**&&**)
Logical OR	(**||**)

The logical AND operator returns an integer 1 if both of its operands are nonzero; otherwise, it returns an integer 0.

The logical OR operator returns an integer 1 if either of its operands is nonzero; otherwise it returns an integer 0.

The types of the operands may be either integer or floating point; the memory space attribute of the result is **N**one. Logical operators are primarily intended for use with the conditional assembly **IF** directive, but can be used in any expression.

3.7 OPERATOR PRECEDENCE

Expressions are evaluated with the following operator precedence:

1. parenthetical expression (innermost first)
2. unary plus, unary minus, one's complement, logical negation
3. multiplication, division, mod
4. addition, subtraction
5. shift
6. relational operators: less, less or equal, greater, greater or equal
7. relational operators: equal, not equal
8. bitwise AND, OR, EOR
9. logical AND, OR

Operators of the same precedence are evaluated left to right. Valid operands include numeric constants, literal ASCII strings, and symbols. The one's complement, shift, and bitwise operators cannot be applied to floating point operands. That is, if the evaluation of an expression (after operator precedence has been applied) results in a floating point number on either side of any of these operators, an error will be generated.

3.8 FUNCTIONS

The Assembler has several built-in functions to support data conversion, string comparison, and transcendental math computations. Functions may be used as terms in any arbitrary expression. Functions may have zero or more arguments, but must always be followed by open and closed parentheses. Function arguments which are expressions must be absolute expressions except where noted. Arguments containing external references are not allowed. There must be no intervening spaces between the function name

and the opening parenthesis, and there must be no spaces between comma-separated arguments.

Assembler functions can be grouped into five types:

1. Mathematical functions
2. Conversion functions
3. String functions
4. Macro functions
5. Assembler mode functions

3.8.1 Mathematical Functions

The mathematical functions comprise transcendental, random value, and min/max functions, among others:

ABS - Absolute value
ACS - Arc cosine
ASN - Arc sine
AT2 - Arc tangent
ATN - Arc tangent
CEL - Ceiling function
COH - Hyperbolic cosine
COS - Cosine
FLR - Floor function
L10 - Log base 10
LOG - Natural logarithm
MAX - Maximum value
MIN - Minimum value
POW - Raise to a power
RND - Random value
SGN - Return sign
SIN - Sine
SNH - Hyperbolic sine
SQT - Square root
TAN - Tangent
TNH - Hyperbolic tangent
XPN - Exponential function

3.8.2 Conversion Functions

The conversion functions provide conversion between integer, floating point, and fixed point fractional values:

CVF	- Convert integer to floating point
CVI	- Convert floating point to integer
CVS	- Convert memory space
FLD	- Shift and mask operation
FRC	- Convert floating point to fractional
LFR	- Convert floating point to long fractional
LNG	- Concatenate to double word
LUN	- Convert long fractional to floating point
RVB	- Reverse bits in field
UNF	- Convert fractional to floating point

3.8.3 String Functions

String functions compare strings, return the length of a string, and return the position of a substring within a string:

LEN	- Length of string
POS	- Position of substring in string
SCP	- Compare strings

3.8.4 Macro Functions

Macro functions return information about macros:

ARG	- Macro argument function
CNT	- Macro argument count
MAC	- Macro definition function
MXP	- Macro expansion function

3.8.5 Assembler Mode Functions

Miscellaneous functions having to do with Assembler operation:

CCC	- Cumulative cycle count
CHK	- Current instruction/data checksum
CTR	- Location counter type
DEF	- Symbol definition function
EXP	- Expression check
INT	- Integer check
LCV	- Location counter value
LST	- **LIST** directive flag value
MSP	- Memory space
REL	- Relative mode function

Individual descriptions of each of the Assembler functions follow. They include usage guidelines, functional descriptions, and examples.

@**ABS(**<expression>**)**

Returns the absolute value of <expression> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
MOVE    #@ABS(VAL),D4.S          ; load absolute value
```

@**ACS(**<expression>**)**

Returns the arc cosine of <expression> as a floating point value in the range zero to pi. The result of <expression> must be between -1 and 1. The memory space attribute of the result will be **N**one.

Example:

```
ACOS    =    @ACS(-1.0)          ; ACOS  =  3.141593
```

@**ARG(**<symbol> | <expression>**)**

Returns integer 1 if the macro argument represented by <symbol> or <expression> is present, 0 otherwise. If the argument is a symbol it must be single-quoted and refer to a dummy argument name. If the argument is an expression it refers to the ordinal position of the argument in the macro dummy argument list. A warning will be issued if this function is used when no macro expansion is active. The memory space attribute of the result will be **N**one.

Example:

```
IF      @ARG(TWIDDLE)            ; twiddle factor provided?
```

@**ASN(**<expression>**)**

Returns the arc sine of <expression> as a floating point value in the range -pi/2 to pi/2. The result of <expression> must be between -1 and 1. The memory space attribute of the result will be **N**one.

Example:

```
ARCSINE   SET     @ASN(-1.0)          ; ARCSINE  =  -1.570796
```

@**AT2(**<expr1,expr2>**)**

Returns the arc tangent of <expr1>/<expr2> as a floating point value in the range -pi to pi. Expr1 and expr2 must be separated by a comma. The memory space attribute of the result will be **N**one.

Example:

```
ATAN      EQU     @AT2(-1.0,1.0)      ; ATAN  =  -0.7853982
```

@**ATN(**<expression>**)**

Returns the arc tangent of <expression> as a floating point value in the range -pi/2 to pi/2. The memory space attribute of the result will be **N**one.

Example:

```
MOVE      #@ATN(1.0),D0.S             ; load arc tangent
```

@CCC()

Returns the cumulative cycle count as an integer. Useful in conjunction with the **CC**, **NOCC**, and **CONTCC** Assembler options (see the **OPT** directive). The memory space attribute of the result will be **N**one.

Example:

```
IF        @CCC() > 200                ; cycle count > 200?
```

@**CEL(**<expression>**)**

Returns a floating point value which represents the smallest integer greater than or equal to <expression>. The memory space attribute of the result will be **N**one.

Example:

```
CEIL      SET     @CEL(-1.05)         ; CEIL  =  -1.0
```

@CHK()

Returns the current instruction/data checksum value as an integer. Useful in conjunction with the **CK**, **NOCK**, and **CONTCK** Assembler options (see the **OPT** directive). Note that assignment of the checksum value with directives other than **SET** could cause phasing errors due to different generated instruction values between passes. The memory space attribute of the result will be **N**one.

Example:

```
CHKSUM  SET     @CHK()          ; reserve checksum value
```

@CNT()

Returns the count of the current macro expansion arguments as an integer. A warning will be issued if this function is used when no macro expansion is active. The memory space attribute of the result will be **N**one.

Example:

```
ARGCNT  SET     @CNT()          ; squirrel away arg count
```

@COH(<expression>**)**

Returns the hyperbolic cosine of <expression> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
HYCOS   EQU     @COH(VAL)       ; compute hyperbolic cosine
```

@COS(<expression>**)**

Returns the cosine of <expression> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
        DC      -@COS(@CVF(COUNT)*FREQ)  ; compute cosine value
```

@CTR({**L** | **R**}**)**

If **L** is specified as the argument, returns the counter number of the load location counter. If **R** is specified, returns the counter number of the runtime location counter. The counter number is returned as an integer value with memory space of **N**one.

Example:

```
CNUM    =       @CTR(R)         ; runtime counter number
```

@CVF(<expression>**)**

Converts the result of <expression> to a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
FLOAT   SET   @CVF(5)          ; FLOAT  =  5.0
```

@CVI(<expression>**)**

Converts the result of <expression> to an integer value. This function should be used with caution since the conversions can be inexact (e.g., floating point values are truncated). The memory space attribute of the result will be **N**one.

Example:

```
INT     SET   @CVI(-1.05)      ; INT  =  -1
```

@CVS({**X** | **Y** | **L** | **P** | **E** | **N**},<expression>**)**

Converts the memory space attribute of <expression> to that specified by the first argument; returns <expression>. See section 3.3 for more information on memory space attributes. The <expression> may be relative or absolute.

Example:

```
LOADDR  EQU   @CVS(X,TARGET)   ; set LOADDR to X:TARGET
```

@DEF(<symbol>**)**

Returns an integer 1 (memory space attribute **N**) if <symbol> has been defined, 0 otherwise. <symbol> may be any label not associated with a **MACRO** or **SECTION** directive. If <symbol> is quoted it is looked up as a **DEFINE** symbol; if it is not quoted it is looked up as an ordinary label.

Example:

```
        IF      @DEF(ANGLE)     ; assemble if ANGLE defined
```

@EXP(<expression>**)**

Returns an integer 1 (memory space attribute **N**) if the evaluation of <expression> would not result in errors. Returns 0 if the evaluation of <expression> would cause an error. No error will be output by the Assembler if <expression> contains an error. No test is made by the Assembler for warnings. The <expression> may be relative or absolute.

Example:

```
        IF      !@EXP(@FRC(VAL))        ; skip on error
```

@FLD(<base>,<value>,<width>[,<start>]**)**

Shift and mask <value> into <base> for <width> bits beginning at bit <start>. If <start> is omitted, zero (least significant bit) is assumed. All arguments must be positive integers and none may be greater than the target word size. Returns the shifted and masked value with a memory space attribute of **N**one.

Example:

```
SWITCH  EQU  @FLD(TOG,1,1,7)      ; turn eighth bit on
```

@FLR(<expression>**)**

Returns a floating point value which represents the largest integer less than or equal to <expression>. The memory space attribute of the result will be **N**one.

Example:

```
FLOOR   SET  @FLR(2.5)            ; FLOOR = 2.0
```

@FRC(<expression>**)**

For binary fractional DSPs (DSP56000) this functions performs scaling and convergent rounding to obtain the fractional representation of the floating point <expression> as an integer. For floating point DSPs (DSP96000) this function simply returns the binary representation of <expression> as an integer. The memory space attribute of the result will be **N**one.

Example:

```
FRAC    EQU  @FRC(FLT)+1          ; compute saturation
```

@INT(<expression>**)**

Returns an integer 1 (memory space attribute **N**) if <expression> has an integer result, 0 otherwise. The <expression> may be relative or absolute.

Example:

```
IF      @INT(TERM)                ; insure integer value
```

@L10(<expression>**)**

Returns the base 10 logarithm of <expression> as a floating point value. <expression> must be greater than zero. The memory space attribute of the result will be **N**one.

Example:

```
LOG     EQU  @L10(100.0)          ; LOG = 2
```

@LCV({**L** | **R**}[,{**L** | **H** | <expression>}]**)**

If **L** is specified as the first argument, returns the memory space attribute and value of the load location counter. If **R** is specified, returns the memory space attribute and value of the runtime location counter. The optional second argument indicates the **L**ow, **H**igh, or numbered counter and must be separated from the first argument by a comma. If no second argument is present the default counter (counter 0) is assumed.

The **@LCV** function will not work correctly if used to specify the runtime counter value of a relocatable overlay. This is because the resulting value is an overlay expression, and overlay expressions may not be used to set the runtime counter for a subsequent overlay. See the **ORG** directive (Chapter 6) for more information.

Also, **@LCV**(**L**,...) will not work inside a relocatable overlay. In order to obtain the load counter value for an overlay block, origin to the load space and counter immediately before the overlay and use **@LCV(L)** to get the beginning load counter value for the overlay.

Example:

```
ADDR    =    @LCV(R)          ; save runtime address
```

@LEN(<string>**)**

Returns the length of <string> as an integer. The memory space attribute of the result will be **N**one.

Example:

```
SLEN    SET  @LEN('string')   ; SLEN = 6
```

@LFR(<expression>**)**

For binary fractional DSPs (DSP56000) this functions performs scaling and convergent rounding to obtain the fractional representation of the floating point <expression> as a long integer. For floating point DSPs (DSP96000) this function simply returns the binary representation of <expression> as a long integer. The memory space attribute of the result will be **N**one.

Example:

```
LFRAC   EQU  @LFR(LFLT)       ; store binary form
```

@**LNG(**<expr1>,<expr2>**)**

Concatenates the single word <expr1> and <expr2> into a double word value such that <expr1> is the high word and <expr2> is the low word. The memory space attribute of the result will be **N**one.

Example:

```
LWORD   DC      @LNG(HI,LO)             ; build long word
```

@**LOG(**<expression>**)**

Returns the natural logarithm of <expression> as a floating point value. <expression> must be greater than zero. The memory space attribute of the result will be **N**one.

Example:

```
LOG     EQU     @LOG(100.0)             ; LOG  =  4.605170
```

@**LST()**

Returns the value of the **LIST** directive flag as an integer, with a memory space attribute of **N**one. Whenever a **LIST** directive is encountered in the Assembler source, the flag is incremented; when a **NOLIST** directive is encountered, the flag is decremented.

Example:

```
DUP     @CVI(@ABS(@LST()))              ; list unconditionally
```

@**LUN(**<expression>**)**

Converts the double-word <expression> to a floating point value. For fractional DSPs (DSP56000) <expression> should represent a binary fraction. For floating point DSPs (DSP96000) <expression> should represent a binary floating point number. The memory space attribute of the result will be **N**one.

Example:

```
DBLFRC  EQU     @LUN($3FE0000000000000)     ;DBLFRC  =  0.5
```

@**MAC(**<symbol>**)**

Returns an integer 1 (memory space attribute **N**) if <symbol> has been defined as a macro name, 0 otherwise.

Example:

```
IF      @MAC(DOMUL)             ; expand macro
```

@MAX(<expr1>[,...,<exprN>]**)**

Returns the greatest of <expr1>,...,<exprN> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
MAX     DC      @MAX(1.0,5.5,-3.25)     ; MAX = 5.5
```

@MIN(<expr1>[,...,<exprN>]**)**

Returns the least of <expr1>,...,<exprN> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
MIN     DC      @MIN(1.0,5.5,-3.25)     ; MIN = -3.25
```

@MSP(<expression>**)**

Returns the memory space attribute of <expression> as an integer value:

None = 0
X space = 1
Y space = 2
L space = 3
P space = 4
E space = 5

The <expression> may be relative or absolute.

Example:

```
MEM     SET     @MSP(ORIGIN)            ; save memory space
```

@MXP()

Returns an integer 1 (memory space attribute **N**) if the Assembler is expanding a macro, 0 otherwise.

Example:

```
IF      @MXP()                          ; macro expansion active?
```

@POS(<str1>,<str2>[,<start>]**)**

Returns the position of string <str2> in <str1> as an integer, starting at position <start>. If <start> is not given the search begins at the beginning of <str1>. If the

<start> argument is specified it must be a positive integer and cannot exceed the length of the source string. The memory space attribute of the result will be **N**one.

Example:

```
ID      EQU   @POS('DSP96000','96')      ; ID  =  3
```

@POW(<expr1>,<expr2>**)**

Returns <expr1> raised to the power <expr2> as a floating point value. <expr1> and <expr2> must be separated by a comma. The memory space attribute of the result will be **N**one.

Example:

```
BUF     EQU   @CVI(@POW(2.0,3.0))      ; BUF  =  8
```

@REL()

Returns an integer 1 (memory space attribute **N**) if the Assembler is operating in relative mode, 0 otherwise.

Example:

```
        IF    @REL()                   ; in relative mode?
```

@RND()

Returns a random value in the range 0.0 to 1.0. The memory space attribute of the result will be **N**one.

Example:

```
SEED    DC    @RND()                   ; save initial seed value
```

@RVB(<expr1>[,<expr2>]**)**

Reverse the bits in <expr1> delimited by the number of bits in <expr2>. If <expr2> is omitted the field is bounded by the target word size. Both expressions must be single word integer values.

Example:

```
REV     EQU   @RVB(VAL)                ; reverse all bits in value
```

@SCP(<str1>,<str2>**)**

Returns an integer 1 (memory space attribute **N**) if the two strings compare, 0 otherwise. The two strings must be separated by a comma.

Example:

```
IF      @SCP(STR,'MAIN')        ; does STR equal MAIN?
```

@SGN(<expression>**)**

Returns the sign of <expression> as an integer: -1 if the argument is negative, 0 if zero, 1 if positive. The memory space attribute of the result will be **N**one. The <expression> may be relative or absolute.

Example:

```
IF      @SGN(INPUT)             ; is sign positive?
```

@SIN(<expression>**)**

Returns the sine of <expression> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
DC      @SIN(@CVF(COUNT)*FREQ)  ; compute sine value
```

@SNH(<expression>**)**

Returns the hyperbolic sine of <expression> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
HSINE   EQU   @SNH(VAL)         ; hyperbolic sine
```

@SQT(<expression>**)**

Returns the square root of <expression> as a floating point value. <expression> must be positive. The memory space attribute of the result will be **N**one.

Example:

```
SQRT    EQU   @SQT(3.5)         ; SQRT  =  1.870829
```

@**TAN(**<expression>**)**

Returns the tangent of <expression> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
MOVE    #@TAN(1.0),D1.S          ; load tangent
```

@**TNH(**<expression>**)**

Returns the hyperbolic tangent of <expression> as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
HTAN    =       @TNH(VAL)        ; hyperbolic tangent
```

@**UNF(**<expression>**)**

Converts <expression> to a floating point value. For fractional DSPs (DSP56000) <expression> should represent a binary fraction. For floating point DSPs (DSP96000) <expression> should represent a binary floating point number. The memory space attribute of the result will be **N**one.

Example:

```
FRC     EQU     @UNF($400000)    ; FRC  =  0.5
```

@**XPN(**<expression>**)**

Returns the exponential function (base e raised to the power of <expression>) as a floating point value. The memory space attribute of the result will be **N**one.

Example:

```
EXP     EQU     @XPN(1.0)        ; EXP  =  2.718282
```