

# 4

*Figure 4-0*  
*Example 4-0*  
*Syntax 4-0*  
*Table 4-0*

## Expressions

This chapter describes the operators and operands available in the Verilog HDL, and how to use them to form expressions.

An expression is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Alternatively, an expression is any legal operand—for example, a net bit-select. Wherever a value is needed in a Verilog HDL statement, an expression can be given. However, several statement constructs limit an expression to a constant expression. A constant expression consists of constant numbers and predefined parameter names only, but can use any of the operators defined in Table 4-1.

For their use in expressions, integer and time data types share the same traits as the data type reg. Descriptions pertaining to register usage apply to integers and times as well.

An operand can be one of the following:

- number (including real)
- net
- register, integer, time
- net bit-select
- register bit-select
- net part-select
- register part-select
- memory element
- a call to a user-defined function or system defined function that returns any of the above

## 4.1 Operators

The symbols for the Verilog HDL operators are similar to those in the C language. Table 4-1 lists these operators.

| Verilog Language Operators |                       |
|----------------------------|-----------------------|
| { }                        | concatenation         |
| + - * /                    | arithmetic            |
| %                          | modulus               |
| > >= < <=                  | relational            |
| !                          | logical negation      |
| &&                         | logical and           |
|                            | logical or            |
| ==                         | logical equality      |
| !=                         | logical inequality    |
| ===                        | case equality         |
| !==                        | case inequality       |
| ~                          | bit-wise negation     |
| &                          | bit-wise and          |
|                            | bit-wise inclusive or |
| ^                          | bit-wise exclusive or |
| ^~ or ~^                   | bit-wise equivalence  |
| &                          | reduction and         |
| ~&                         | reduction nand        |
|                            | reduction or          |
| ~                          | reduction nor         |
| ^                          | reduction xor         |
| ~^ or ^~                   | reduction xnor        |
| <<                         | left shift            |
| >>                         | right shift           |
| ? :                        | conditional           |

Table 4-1: Operators for Verilog language

Not all of the operators listed above are valid with real expressions. Table 4-2 is a list of the operators that are legal when applied to real numbers.

| Operators for Real Expressions |                  |
|--------------------------------|------------------|
| unary +    unary -             | unary operators  |
| +   -   *   /                  | arithmetic       |
| >   >=   <   <=                | relational       |
| !   &&                         | logical          |
| ==   !=                        | logical equality |
| ?:                             | conditional      |
| or                             | logical          |

Table 4-2: Legal operators for use in real expressions

The result of using logical or relational operators on real numbers is a single-bit scalar value.

Table 4-3 lists operators that are *not allowed* to operate on real numbers.

| Disallowed Operators for Real Expressions |               |
|---|---------------|
| {}  | concatenate   |
| %   | modulus       |
| ===    !==                                | case equality |
| ~   &                                     | bit-wise      |
| ^   ^~   ~^                               | reduction     |
| &   ~&       ~                            |               |
| <<   >>                                   | shift         |

Table 4-3: Operators not allowed for real expressions

See Section 3.10.3 for more information on use of real numbers.

### 4.1.1 Binary Operator Precedence

The precedence order of binary operators (and the ternary operator `?:`) is the same as the precedence order for the matching operators in the C language. Verilog has two equality operators not present in C; they are discussed in Section 4.1.6. Table 4-4 summarizes the precedence rules for Verilog's binary and ternary operators.


| Operator Precedence Rules |  |
|---------------------------|--|
| ! ~                       | highest precedence<br><br><br><br>lowest precedence |
| * / %                     |  |
| + -                       |  |
| << >>                     |  |
| < <= > >=                 |  |
| == != === !==             |  |
| &                         |  |
| ^ ^~                      |  |
|                           |  |
| &&                        |  |
|                           |  |
| ?: (ternary operator)     |  |

Table 4-4: Precedence rules for operators

Operators on the same line in Table 4-4 have the same precedence. Rows are in order of decreasing precedence, so, for example, `*`, `/`, and `%` all have the same precedence, which is higher than that of the binary `+` and `-` operators.

All operators associate left to right. Associativity refers to the order in which a language evaluates operators having the same precedence. Thus, in the following example `B`, is added to `A` and then `C` is subtracted from the result of `A+B`.

$$A + B - C$$

When operators differ in precedence, the operators with higher precedence apply first. In the following example, B is divided by C (division has higher precedence than addition) and then the result is added to A.

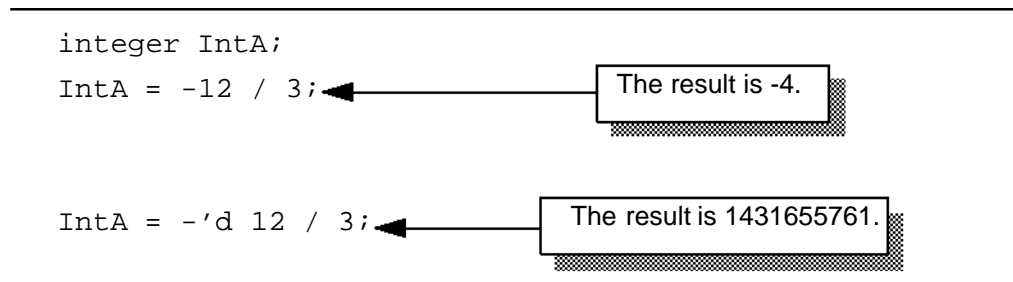
A + B / C

Parentheses can be used to change the operator precedence.

(A + B) / C // not the same as A + B / C

### 4.1.2 Numeric Conventions in Expressions

Operands can be expressed as based and sized numbers—with the following restriction: The Verilog language interprets a number of the form `sss 'f nnn`, when used directly in an expression, as the unsigned number represented by the two's complement of `nnn`. Example 4-1 shows two ways to write the expression “minus 12 divided by 3.” Note that `-12` and `-d12` both evaluate to the same bit pattern, but in an expression `-d12` loses its identity as a signed, negative number.



Example 4-1: Number format in expressions

### 4.1.3 Arithmetic Operators

The binary arithmetic operators are the following:

+   -   \*   /   % (the modulus operator)

Integer division truncates any fractional part. The modulus operator—for example,  $y \% z$ , gives the remainder when the first operand is divided by the second, and thus is zero when  $z$  divides  $y$  exactly. The result of a modulus operation takes the sign of the first operand. Table 4-5 gives examples of modulus operations.

| Modulus Expression | Result | Comments  |
|--------------------|--------|---|
| $10 \% 3$          | 1      | $10/3$ yields a remainder of 1  |
| $11 \% 3$          | 2      | $11/3$ yields a remainder of 2  |
| $12 \% 3$          | 0      | $12/3$ yields no remainder  |
| $-10 \% 3$         | -1     | the result takes the sign of the first operand  |
| $11 \% -3$         | 2      | the result takes the sign of the first operand  |
| $-4'd12 \% 3$      | 1      | $-4'd12$ is seen as a large, positive number that leaves a remainder of 1 when divided by 3 |

Table 4-5: Examples of modulus operations

The unary arithmetic operators take precedence over the binary operators. The unary operators are the following:

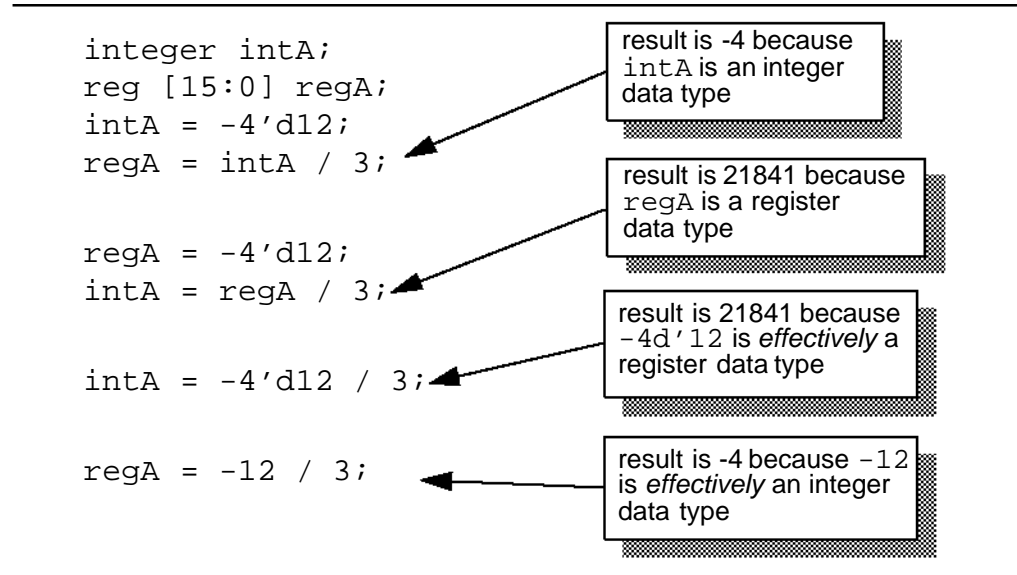
+    -

For the arithmetic operators, if any operand bit value is the unknown value  $x$ , then the entire result value is  $x$ .

#### 4.1.4 Arithmetic Expressions with Registers and Integers

An arithmetic operation on a register data type behaves differently than an arithmetic operation on an integer data type. The Verilog language sees a register data type as an unsigned value and an integer type as a signed value. As a result, when you assign a value of the form  $\text{<size><base\_format><number>}$  to a *register* and then use that register as an expression operand, you are actually using a positive number that is the two's complement of  $nnn$ . In contrast, when you assign a value of

the form `-<size><base_format><number>` to an *integer* and then use that integer as an expression operand, the expression evaluates using signed arithmetic. Example 4-2 shows various ways to divide minus twelve by three using integer and register data types in expressions.



Example 4-2: Modulus operation with registers and integers

### 4.1.5 Relational Operators

Table 4-2 lists and defines the relational operators.

| Relational Operators |                              |
|----------------------|------------------------------|
| a < b                | a less than b                |
| a > b                | a greater than b             |
| a <= b               | a less than or equal to b    |
| a >= b               | a greater than or equal to b |

Table 4-6: The relational operators defined

The relational operators in Table 4-2 all yield the scalar value 0 if the specified relation is false, or the value 1 if the specified relation is true. If, due to unknown bits in the operands, the relation is ambiguous, then the result is the unknown value (x).

**Please note:** If Verilog-XL tests a value that is x or z, then the result of that test is *False*.

All the relational operators have the same precedence. Relational operators have lower precedence than arithmetic operators. The following examples illustrate the implications of this precedence rule:

```
a < size - 1      // this construct is the same as
a < (size - 1)   // this construct, but . . .
size - (1 < a)   // this construct is not the same
size - 1 < a     // as this construct
```

Note that when `size - (1 < a)` evaluates, the relational expression evaluates first and then either zero or one is subtracted from `size`. When `size - 1 < a` evaluates, the `size` operand is reduced by one and then compared with `a`.

### 4.1.6 Equality Operators

The equality operators rank just lower in precedence than the relational operators. Table 4-2 lists and defines the equality operators.

| Equality Operators   |   |
|----------------------|---|
| <code>a == b</code>  | a equal to b, including x and z         |
| <code>a != b</code>  | a not equal to b, including x and z     |
| <code>a === b</code> | a equal to b, result may be unknown     |
| <code>a !== b</code> | a not equal to b, result may be unknown |

Table 4-7: The equality operators defined

All four equality operators have the same precedence. These four operators compare operands bit for bit, with zero filling if the two operands are of unequal bit-length. As with the relational operators, the result is 0 if false, 1 if true.

For the `==` and `!=` operators, if either operand contains an `x` or a `z`, then the result is the unknown value (`x`).

For the `===` and `!==` operators, the comparison is done just as it is in the procedural case statement. Bits that are `x` or `z` are included in the comparison and must match for the result to be true. The result of these operators is always a known value, either 1 or 0.

### 4.1.7 Logical Operators

The operators logical AND (`&&`) and logical OR (`||`) are logical connectives. Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. The result of the evaluation of a logical comparison is one (defined as *true*), zero (defined as *false*), or, if the result is ambiguous, then the result is the unknown value (`x`). For example, if register `alpha` holds the integer value 237 and `beta` holds the value zero, then the following examples perform as described:

```
regA = alpha && beta; // regA is set to 0
regB = alpha || beta; // regB is set to 1
```

The precedence of `&&` is greater than that of `||`, and both are lower than relational and equality operators. The following expression ANDs three sub-expressions without needing any parentheses:

```
a < size-1 && b != c && index != lastone
```

However, it is recommended for readability purposes that parentheses be used to show very clearly the precedence intended, as in the following rewrite of the above example:

```
(a < size-1) && (b != c) && (index != lastone)
```

A third logical operator is the unary logical negation operator `!`. The negation operator converts a non-zero or true operand into 0 and a zero or false operand into 1. An ambiguous truth value remains as `x`. A common use of `!` is in constructions like the following:

```
if (!inword)
```

## Expressions

### Operators

In some cases, the preceding construct makes more sense to someone reading the code than the equivalent construct shown below:

```
if (inword == 0)
```

Constructions like `if (!inword)` read quite nicely (“if not inword”), but more complicated ones can be hard to understand. The first form is slightly more efficient in simulation speed than the second.

### 4.1.8 Bit-Wise Operators

The bit operators perform bit-wise manipulations on the operands—that is, the operator compares a bit in one operand to its equivalent bit in the other operand to calculate one bit for the result. The logic tables in Table 4-8 show the results for each possible calculation.

bit-wise unary negation

|   |   |
|---|---|
| ~ |   |
| 0 | 1 |
| 1 | 0 |
| x | x |

bit-wise binary AND operator

|   |   |   |   |
|---|---|---|---|
| & | 0 | 1 | x |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x |
| x | 0 | x | x |

bit-wise binary inclusive  
OR operator

|   |   |   |   |
|---|---|---|---|
|   | 0 | 1 | x |
| 0 | 0 | 1 | x |
| 1 | 1 | 1 | 1 |
| x | x | 1 | x |

bit-wise binary exclusive  
OR operator

|   |   |   |   |
|---|---|---|---|
| ^ | 0 | 1 | x |
| 0 | 0 | 1 | x |
| 1 | 1 | 0 | x |
| x | x | x | x |

bit-wise binary exclusive  
NOR operator

|    |   |   |   |
|----|---|---|---|
| ^~ | 0 | 1 | x |
| 0  | 1 | 0 | x |
| 1  | 0 | 1 | x |
| x  | x | x | x |

Table 4-8: Bit-wise operators logic tables

Care should be taken to distinguish the bit-wise operators `&` and `|` from the logical operators `&&` and `||`. For example, if `x` is 1 and `y` is 2, then `x & y` is 0, while `x && y` is 1. When the operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

### 4.1.9 Reduction Operators

The unary reduction operators perform a bit-wise operation on a single operand to produce a single bit result. The first step of the operation applies the operator between the first bit of the operand and the second—using the logic tables in Table 4-9. The second and subsequent steps apply the operator between the one-bit result of the prior step and the next bit of the operand—still using the same logic table.

reduction unary  
AND operator

| <code>&amp;</code> | 0 | 1 | x |
|--------------------|---|---|---|
| 0                  | 0 | 0 | 0 |
| 1                  | 0 | 1 | x |
| x                  | 0 | x | x |

reduction unary inclusive  
OR operator

| <code> </code> | 0 | 1 | x |
|----------------|---|---|---|
| 0              | 0 | 1 | x |
| 1              | 1 | 1 | 1 |
| x              | x | 1 | x |

reduction unary exclusive  
OR operator

| <code>^</code> | 0 | 1 | x |
|----------------|---|---|---|
| 0              | 0 | 1 | x |
| 1              | 1 | 0 | x |
| x              | x | x | x |

Table 4-9: Reduction operators logic tables

Note that the reduction unary NAND and reduction unary NOR operators operate the same as the reduction unary AND and OR operators, respectively, but with their outputs negated. The effective results produced by the unary reduction operators are listed in Table 4-10 and Table 4-11.

| Results of Unary &,  , ~&, and ~ <br>Reduction Operations |   |   |    |   |
|---|---|---|----|---|
| Operand   | & |   | ~& | ~ |
| no bits set   | 0 | 0 | 1  | 1 |
| all bits set  | 1 | 1 | 0  | 0 |
| some bits set,<br>but not all                             | 0 | 1 | 1  | 0 |

Table 4-10: AND, OR, NAND, and NOR unary reduction operations

| Results of Unary ^ and ~^<br>Reduction Operators |   |    |
|--|---|----|
| Operand  | ^ | ~^ |
| odd number of bits set                           | 1 | 0  |
| even number of bits set<br>(or none)             | 0 | 1  |

Table 4-11: Exclusive OR and exclusive NOR unary reduction operations

### 4.1.10 Syntax Restrictions

The Verilog language imposes two syntax restrictions intended to protect description files from a typographical error that is particularly hard to find. The error consists of transposing a space and a symbol. Note that the constructs on line 1 below do *not* represent the same syntax as the similar constructs on line 2.

- |           |        |
|-----------|--------|
| 1. a & &b | a    b |
| 2. a && b | a    b |

In order to protect users from this type of error, Verilog requires the use of parentheses to separate a reduction or or and operator from a bit-wise or or and operator. Table 4-12 shows the syntax that requires parentheses:

| Invalid Syntax | Equivalent Syntax |
|----------------|-------------------|
| a & &b         | a & (&b)          |
| a    b         | a   ( b)          |

Table 4-12: Syntax equivalents for syntax restriction

### 4.1.11 Shift Operators

The shift operators, << and >>, perform left and right shifts of their left operand by the number of bit positions given by the right operand. Both shift operators fill the vacated bit positions with zeroes. Example 4-3 illustrates this concept.

---

```

module shift;
  reg [3:0] start, result;
  initial
  begin
    start = 1;
    result = (start << 2);
  end
endmodule

```

1.) Start is set to 0001.  
2.) Result is set to 0100.

---

*Example 4-3: Use of shift operator*

In this example, the register `result` is assigned the binary value 0100, which is 0001 shifted to the left two positions and zero filled.

### 4.1.12 Conditional Operator

The conditional operator has three operands separated by two operators in the following format:

```
cond_expr ? true_expr : false_expr
```

If `cond_expr` evaluates to false, then `false_expr` is evaluated and used as the result. If the conditional expression is true, then `true_expr` is evaluated and used as the result. If `cond_expr` is ambiguous, then both `true_expr` and `false_expr` are evaluated and their results are compared, bit by bit, using Table 4-13 to calculate the final result. If the lengths of the operands are different, the shorter operand is lengthened to match the longer and zero filled from the left (the high-order end).

| ?: | 0 | 1 | x | z |
|----|---|---|---|---|
| 0  | 0 | x | x | x |
| 1  | x | 1 | x | x |
| x  | x | x | x | x |
| z  | x | x | x | x |

Table 4-13: Conditional operator ambiguous condition results

The following example of a tri-state output bus illustrates a common use of the conditional operator.

```
wire [15:0] busa = drive_busa ? data : 16'bz;
```

The bus called `data` is driven onto `busa` when `drive_busa` is 1. If `drive_busa` is unknown, then an unknown value is driven onto `busa`. Otherwise, `busa` is not driven.

### 4.1.13 Concatenations

A concatenation is the joining together of bits resulting from two or more expressions. The concatenation is expressed using the brace characters { and }, with commas separating the expressions within. The next example concatenates four expressions:

```
{a, b[3:0], w, 3'b101}
```

The previous example is equivalent to the following example:

```
{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}
```

Unsize constant numbers are not allowed in concatenations. This is because the size of each operand in the concatenation is needed to calculate the complete size of the concatenation.

Concatenations can be expressed using a repetition multiplier as shown in the next example.

```
{4{w}} // This is equivalent to {w, w, w, w}
```

The next example illustrates nested concatenations.

```
{b, {3{a, b}}} // This is equivalent to  
              // {b, a, b, a, b, a, b}
```

The repetition multiplier must be a constant expression.

## 4.2 Operands

As stated before, there are several types of operands that can be specified in expressions. The simplest type is a reference to a net or register in its complete form—that is, just the name of the net or register is given. In this case, all of the bits making up the net or register value are used as the operand.

If just a single bit of a vector net or register is required, then a bit-select operand is used. A part-select operand is used to reference a group of adjacent bits in a vector net or register.

A memory element can be referenced as an operand.

A concatenation of other operands (including nested concatenations) can be specified as an operand.

A function call is an operand.

### 4.2.1 Net and Register Bit Addressing

Bit-selects extract a particular bit from a vector net or register. The bit can be addressed using an expression. The next example specifies the single bit of `acc` that is addressed by the operand `index`.

```
acc[index]
```

The actual bit that is accessed by an address is, in part, determined by the declaration of `acc`. For instance, each of the declarations of `acc` shown in the next example causes a particular value of `index` to access a *different* bit:

```
reg [15:0] acc;  
reg [1:16] acc;
```

If the bit select is out of the address bounds or is `x`, then the value returned by the reference is `x`.

Several contiguous bits in a vector register or net can be addressed, and are known as **part-selects**. A part-select of a vector register or net is given with the following syntax:

```
vect[ms_expr:ls_expr]
```

Both expressions must be constant expressions. The first expression must address a more significant bit than the second expression. Compiler errors result if either of these rules is broken.

The next example and the bullet items that follow it illustrate the principles of bit addressing. The code declares an 8-bit register called `vect` and initializes it to a value of 4. The bullet items describe how the separate bits of that vector can be addressed.

```
reg [7:0] vect;  
vect = 4;
```

- if the value of `addr` is 2, then `vect[addr]` returns 1
- if the value of `addr` is out of bounds, then `vect[addr]` returns `x`
- if `addr` is 0, 1, or 3 through 7, `vect[addr]` returns 0
- `vect[3:0]` returns the bits 0100
- `vect[5:1]` returns the bits 00010
- `vect[<expression that returns x>]` returns `x`
- `vect[<expression that returns z>]` returns `x`
- if any bit of `addr` is `x/z`, then the value of `addr` is `x`

## 4.2.2 Memory Addressing

Section 3.8 discussed the declaration of memories. This section discusses memory addressing. The next example declares a memory of 1024 8-bit words:

```
reg [7:0] mem_name[0:1023];
```

The syntax for a memory address consists of the name of the memory and an expression for the address—specified with the following format:

```
mem_name[addr_expr]
```

The `addr_expr` can be any expression; therefore, memory indirections can be specified in a single expression. The next example illustrates memory indirection:

```
mem_name[mem_name[3]]
```

In the above example, `mem_name[3]` addresses word three of the memory called `mem_name`. The value at word three is the index into `mem_name` that is used by the memory address `mem_name[mem_name[3]]`. As with bit-selects, the address bounds given in the declaration of the memory determine the effect of the address expression. If the index is out of the address bounds or is `x`, then the value of the reference is `x`.

There is no mechanism to express bit-selects or part-selects of memory elements directly. If this is required, then the memory element has to be first transferred to an appropriately sized temporary register.

### 4.2.3 Strings

String operands are treated as constant numbers consisting of a sequence of 8-bit ASCII codes, one per character, with no special termination character.

Any Verilog HDL operator can manipulate string operands. The operator behaves as though the entire string were a single numeric value.

Example 4-4 declares a string variable large enough to hold 14 characters and assigns a value to it. The example then manipulates the string using the concatenation operator.

Note that when a variable is larger than required to hold the value being assigned, the contents after the assignment are padded on the left with zeros. This is consistent with the padding that occurs during assignment of non-string values.

---

```
module string_test;
    reg [8*14:1] stringvar;
    initial
        begin
            stringvar = "Hello world";
            $display("%s is stored as %h",
                stringvar,stringvar);
            stringvar = {stringvar,"!!!"};
            $display("%s is stored as %h",
                stringvar,stringvar);
        end
    endmodule
```

---

*Example 4-4: Concatenation of strings*

The result of running Verilog on the above description is:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

#### 4.2.4 String Operations

The common string operations *copy*, *concatenate*, and *compare* are supported by Verilog operators. Copy is provided by simple assignment. Concatenation is provided by the concatenation operator. Comparison is provided by the equality operators. Example 4-4 and Example 4-5 illustrate assignment, concatenation, and comparison of strings.

When manipulating string values in vector variables, at least  $8*n$  bits are required in the vector, where  $n$  is the number of characters in the string.

#### 4.2.5 String Value Padding and Potential Problems

When strings are assigned to variables, the values stored are padded on the left with zeros. Padding can affect the results of comparison and concatenation operations. The comparison and concatenation operators do not distinguish between zeros resulting from padding and the original string characters.

Example 4-5 illustrates the potential problem.

---

```

reg [8*10:1] s1, s2;
initial
begin
    s1 = "Hello";
    s2 = " world!";
    if ({s1,s2} == "Hello world!")
        $display("strings are equal");
end

```

---

*Example 4-5: Comparing string variables*

The comparison in the example above fails because during the assignment the string variables get padded as illustrated in the next example:

```

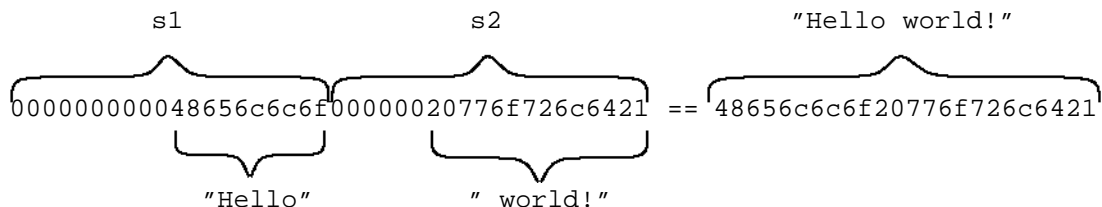
s1 = 000000000048656c6c6f
s2 = 00000020776f726c6421

```

The concatenation of s1 and s2 includes the zero padding, resulting in the following value:

```
000000000048656c6c6f00000020776f726c6421
```

Since the string "Hello world" contains no zero padding, the comparison fails, as shown below:



The above comparison yields a result of zero, which is equivalent to false.

## 4.2.6 Null String Handling

The null string (" ") is equivalent to the value zero (0).

## 4.3 Minimum, Typical, Maximum Delay Expressions

Verilog HDL delay expressions can be specified as three expressions separated by colons. This triple is intended to represent minimum, typical, and maximum values—in that order. The appropriate expression is selected by the compiler when Verilog-XL is run. The user supplies a command-line option to select which of the three expressions will be used on a global basis. In the absence of a command-line option, Verilog-XL selects the second expression (the “typical” delay value). The syntax is as follows:

```
<mintypmax_expression>  
 ::= <expression>  
 || = <expression1> : <expression2> : <expression3>
```

*Syntax 4-1: Syntax for <mintypmax\_expression>*

The three expressions follow these conventions:

- expression1 is less than or equal to expression2
- expression2 is less than or equal to expression3

Verilog models typically specify three values for delay expressions. The three values allow a design to be tested with minimum, typical, or maximum delay values. In the following example, one of the three specified delays will be executed before the simulation executes the assignment; if the user does not select one, the simulator will take the default.

```
always @A  
X = #(3:4:5) A;
```

The command-line option `+mindelays` selects the minimum expression in all expressions where `min:typ:max` values have been specified. Likewise, `+typdelays` selects all the typical expressions and `+maxdelays` selects all the maximum expressions. Verilog-XL defaults to the second value when a two or three-part delay expression is specified.

Values expressed in `min:typ:max` format can be used in expressions. The next example shows an expression that defines a single triplet of delay values. The minimum value is the sum of `a+d`; the typical value is `b+e`; the maximum value is `c+f`, as follows:

```
a:b:c + d:e:f
```

The next example shows some typical expressions that are used to specify `min:typ:max` format values:

```
val - 32'd 50: 32'd 75: 32'd 100
```

The `min:typ:max` format can be used wherever expressions can appear, both in source text files and in interactive commands.

## 4.4 Expression Bit Lengths

Controlling the number of bits that are used in expression evaluations is important if consistent results are to be achieved. Some situations have a simple solution, for example, if a bit-wise AND operation is specified on two 16-bit registers, then the result is a 16-bit value. However, in some situations it is not obvious how many bits are used to evaluate an expression, what size the result should be, or whether signed or unsigned arithmetic should be used.

For example, when is it necessary to perform the addition of two 16-bit registers in 17 bits to handle a possible carry overflow? The answer depends on the context in which the addition takes place. If the 16-bit addition is modeling a real 16-bit adder that loses or does not care about the carry overflow, then the model must perform the addition in 16 bits. If the addition of two 16-bit unsigned numbers can result in a significant 17th bit, then assign the answer to a 17-bit register.

### 4.4.1

## An Example of an Expression Bit Length Problem

During the evaluation of an expression, interim results take the size of the largest operand (in the case of an assignment, this also includes the left-hand side). You must therefore take care to prevent loss of a significant bit during expression evaluation. This section describes an example of the problems that can occur.

The expression  $(a + b \gg 1)$  yields a 16-bit result, but cannot be assigned to a 16-bit register without the potential loss of the high-order bit. If  $a$  and  $b$  are 16-bit registers, then the result of  $(a+b)$  is 16 bits wide—unless the result is assigned to a register wider than 16 bits. If answer is a 17-bit register, then  $(\text{answer} = a + b)$  yields a full 17-bit result. But in the expression  $(a + b \gg 1)$ , the sum of  $(a + b)$  produces an interim result that is only 16 bits wide. Therefore, the assignment of  $(a + b \gg 1)$  to a 16-bit register loses the carry bit *before* the evaluation performs the one-bit right shift.

There are two solutions to a problem of this type. One is to assign the sum of  $(a+b)$  to a 17-bit register before performing the shift and then shift the 17-bit answer into the 16-bits that your model requires. An easier solution is to use the following trick.

### The problem:

Evaluate the expression  $(a+b)\gg 1$ , assigning the result to a 16-bit register without losing the carry bit. Variables  $a$  and  $b$  are both 16-bit registers.

### The solution:

Add the integer zero to the expression. The expression evaluates as follows:

1.  $0 + (a+b)$  evaluates—the result is as wide as the widest term, which is the 32-bit zero
2. the 32-bit sum of  $0 + (a+b)$  is shifted right one bit

This trick preserves the carry bit until the shift operation can move it back down into 16 bits.

### 4.4.2

## Verilog Rules for Expression Bit Lengths

In the Verilog language, the rules governing the expression bit lengths have been formulated so that most practical situations have a natural solution.

The number of bits of an expression (known as the size of the expression) is determined by the operands involved in the expression and the context in which the expression is given.

A self-determined expression is one where the bit length of the expression is solely determined by the expression itself—for example, an expression representing a delay value.

A context-determined expression is one where the bit length of the expression is determined by the bit length of the expression *and* by the fact that it is part of another expression. For example, the bit size of the right-hand side expression of an assignment depends on itself and the size of the left-hand side.

## Expressions

### Expression Bit Lengths

Table 4-14 shows how the form of an expression determines the bit lengths of the results of the expression. In Table 4-14,  $i$ ,  $j$ , and  $k$  represent expressions of an operand, and  $L(i)$  represents the bit length of the operand represented by  $i$ .

| Expression   | Bit length                   | Comments                         |
|--|------------------------------|----------------------------------|
| unsized constant number  | same as integer (usually 32) |                                  |
| sized constant number  | as given                     |                                  |
| $i \text{ op } j$<br>where op is:<br>+ - * / %<br>&   ^ ~            | $\max(L(i), L(j))$           |                                  |
| +i and -i  | $L(i)$                       |                                  |
| ~i   | $L(i)$                       |                                  |
| $i \text{ op } j$<br>where op is<br>=== !== == != &&   <br>> >= < <= | 1 bit                        | all operands are self-determined |
| op i<br>where op is<br>& ~&   ~  ^ ~^                                | 1 bit                        | all operands are self-determined |
| $i \gg j$<br>$i \ll j$   | $L(i)$                       | $j$ is self-determined           |
| $i ? j : k$  | $\max(L(j), L(k))$           | $i$ is self-determined           |
| {i,..j}  | $L(i)+..+L(j)$               | all operands are self-determined |
| { i { j, .. , k } }  | $i*(L(j)+..+L(k))$           | all operands are self-determined |

Table 4-14: Bit lengths resulting from expressions