4

Expressions

In Verilog, expressions consist of a single operand or multiple operands separated by operators. Use expressions where a value is required in Verilog.

This chapter explains how to build and use expressions, using

- Constant-Valued Expressions
- Operators
- Operands
- Expression Bit-Widths

Constant-Valued Expressions

A constant-valued expression is an expression whose operands are either constants or parameters. HDL Compiler determines the value of these expressions.

In Example 4-1, `size-1` is a constant-valued expression. The expression `(op == ADD)? a + b : a – b` is not a constant-valued expression, because the value depends on the variable `op`. If the value of `op` is `1`, `b` is added to `a`; otherwise, `b` is subtracted from `a`.

Example 4-1 Valid Expressions

```
// all expressions are constant-valued,
// except in the assign statement.
module add_or_subtract( a, b, op, s );
// performs  s = a+b  if op is ADD
// performs  s = a-b  if op is not ADD
     parameter size=8;
     parameter ADD=1’b1;

     input  op;
     input  [size-1:0] a, b;
     output [size-1:0] s;
     assign s = (op == ADD) ? a+b : a-b;//not a constant-
                                         //valued expression
endmodule
```

The operators and operands in an expression influence the way a design is synthesized. HDL Compiler evaluates constant-valued expressions and does not synthesize circuitry to compute their value. If an expression contains constants, they are propagated to reduce the amount of circuitry required. HDL Compiler does synthesize circuitry for an expression that contains variables, however.

Operators

Operators identify the operation to be performed on their operands to produce a new value. Most operators are either unary operators, which apply to only one operand, or binary operators, which apply to two operands. Two exceptions are conditional operators, which take three operands, and concatenation operators, which take any number of operands.

HDL Compiler supports the types of operations listed in Table 4-1, which also lists the Verilog language operators HDL Compiler supports. A description of the operators and their order of precedence appears in the sections that follow the table.

Table 4-1 Verilog Operators Supported by HDL Compiler

Operator type	**Operator**	**Description**
Arithmetic operators	+ – * /	Arithmetic
	%	Modules
Relational operators	> >= < <=	Relational
Equality operators	==	Logical equality
	!=	Logical inequality
Logical operators	!	Logical NOT
	&&	Logical AND
	||	Logical OR
Bitwise operators	~	Bitwise NOT
	&	Bitwise AND
	|	Bitwise OR

Table 4-1 Verilog Operators Supported by HDL Compiler (continued)

Operator type	Operator	Description
	^	Bitwise XOR
	^~ ~^	Bitwise XNOR
Reduction operators	&	Reduction AND
	|	Reduction OR
	~&	Reduction NAND
	~|	Reduction NOR
	^	Reduction XOR
	~^ ^~	Reduction XNOR
Shift operators	<<	Shift left
	>>	Shift right
Conditional operator	? :	Conditions
Concatenation operator	{ }	Concatenation

In the following descriptions, the terms *variable* and *variable operand* refer to operands or expressions that are not constant-valued expressions. This group includes wires and registers, bit-selects and part-selects of wires and registers, function calls, and expressions that contain any of these elements.

Arithmetic Operators

Arithmetic operators perform simple arithmetic on operands. The Verilog arithmetic operators are

- Addition (+)

- Subtraction (–)
- Multiplication (*)
- Division (/)
- Modules (%)

You can use the +, –, and * operators with any operand form (constants or variables). The + and – operators can be used as either unary or binary operators. HDL Compiler requires that the / and % operators have constant-valued operands.

Example 4-2 shows three forms of the addition operator. The circuitry built for each addition operation is different, because of the different operand types. The first addition requires no logic, the second synthesizes an incrementer, and the third synthesizes an adder.

Example 4-2 Addition Operator

```
parameter size=8;
wire [3:0] a,b,c,d,e;

assign c = size + 2; //constant + constant
assign d = a + 1;    //variable + constant
assign e = a + b;    //variable + variable
```

Relational Operators

Relational operators compare two quantities and yield a `0` or `1` value. A `true` comparison evaluates to `1`; a `false` comparison evaluates to `0`. All comparisons assume unsigned quantities. The circuitry synthesized for relational operators is a bitwise comparator whose size is based on the sizes of the two operands.

The Verilog relational operators are

- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)

Example 4-3 shows the use of a relational operator.

Example 4-3 Relational Operator

```
function [7:0] max( a, b  );
input  [7:0] a,b;
    if ( a >= b )  max = a;
    else           max = b;
endfunction
```

Equality Operators

Equality operators generate a `0` if the expressions being compared are not equal and a `1` if the expressions are equal. Equality and inequality comparisons are performed by bit.

The Verilog equality operators are

- Equality (==)
- Inequality (`!=`)

Example 4-4 shows the equality operator testing for a JMP instruction. The output signal jump is set to 1 if the two high-order bits of instruction are equal to the value of parameter JMP; otherwise, jump is set to 0.

Example 4-4 Equality Operator

```
module is_jump_instruction (instruction, jump);
    parameter JMP = 2’h3;

    input  [7:0] instruction;
    output jump;
    assign jump = (instruction[7:6] == JMP);

endmodule
```

Handling Comparisons to X or Z

HDL Compiler always ignores comparisons to an X or a Z. If your code contains a comparison to an X or a Z, a warning message displays, indicating that the comparison is always evaluated to false, which might cause simulation to disagree with synthesis.

Example 4-5 shows code from a file called test2.v. HDL Compiler always assigns the variable B to the value 1, because the comparison to X is ignored.

Example 4-5 Comparison to X Ignored

```
always begin
    if (A == 1’bx)   //this is line 10
        B = 0;
    else
        B = 1;
end
```

When HDL Compiler reads this code, it generates the following warning message:

```
Warning: Comparisons to a "don’t care" are treated as always
being false in routine test2 line 10 in file ’test2.v’. This
may cause simulation to disagree with synthesis. (HDL-170)
```

For an alternative method of handling comparisons to `X` or `Z`, use the `translate_off` and `translate_on` directives to comment out the condition and its first branch (the `true` clause) so that only the `else` branch goes through synthesis.

Logical Operators

Logical operators generate a `1` or a `0`, according to whether an expression evaluates to `true` (`1`) or `false` (`0`). The Verilog logical operators are

- Logical `NOT` (`!`)
- Logical `AND` (&&)
- Logical `OR` (`||`)

The logical `NOT` operator produces a value of `1` if its operand is zero and a value of `0` if its operand is nonzero. The logical `AND` operator produces a value of `1` if both operands are nonzero. The logical `OR` operator produces a value of `1` if either operand is nonzero.

Example 4-6 shows some logical operators.

Example 4-6 Logical Operators

```
module is_valid_sub_inst(inst,mode,valid,unimp);

    parameterIMMEDIATE=2’b00, DIRECT=2’b01;
    parameterSUBA_imm=8’h80, SUBA_dir=8’h90,
             SUBB_imm=8’hc0, SUBB_dir=8’hd0;
    input [7:0] inst;
    input [1:0] mode;
    output valid, unimp;

    assign valid = (((mode == IMMEDIATE) && (
                  (inst == SUBA_imm) ||
                  (inst == SUBB_imm))) ||
                  ((mode == DIRECT) && (
                      (inst == SUBA_dir) ||
                      (inst == SUBB_dir))));

    assign unimp = !valid;
endmodule
```

Bitwise Operators

Bitwise operators act on the operand bit by bit. The Verilog bitwise operators are

- Unary negation (~)
- Binary `AND` (&)
- Binary `OR` (|)
- Binary `XOR` (^)
- Binary `XNOR` (`^~ or ~^`)

Example 4-7 shows some bitwise operators.

Example 4-7 Bitwise Operators

```
module full_adder( a, b, cin, s, cout );
     input  a, b, cin;
     output s, cout;

     assign s    = a ^ b ^ cin;
     assign cout = (a&b) | (cin & (a|b));
endmodule
```

Reduction Operators

Reduction operators take one operand and return a single bit. For example, the reduction `AND` operator takes the `AND` value of all the bits of the operand and returns a 1-bit result. The Verilog reduction operators are

- Reduction `AND` (&)
- Reduction `OR` (|)
- Reduction `NAND` (~&)
- Reduction `NOR` (~|)
- Reduction `XOR` (^)
- Reduction `XNOR` (^~ or ~^)

Example 4-8 shows the use of some reduction operators.

Example 4-8 Reduction Operators

```
module check_input ( in, parity, all_ones );
    input  [7:0] in;
    output parity, all_ones;

    assign parity   = ^ in;
    assign all_ones = & in;
endmodule
```

Shift Operators

A shift operator takes two operands and shifts the value of the first operand right or left by the number of bits given by the second operand.

The Verilog shift operators are

- Shift left (<<)
- Shift right (>>)

After the shift, vacated bits fill with zeros. Shifting by a constant results in minor circuitry modification (because only rewiring is required). Shifting by a variable causes a general shifter to be synthesized. Example 4-9 shows use of a shift-right operator to perform division by 4.

Example 4-9 Shift Operator

```
module divide_by_4( dividend, quotient );
    input  [7:0] dividend;
    output [7:0] quotient;

    assign quotient = dividend >> 2; //shift right 2 bits
endmodule
```

Conditional Operator

The conditional operator (`? :`) evaluates an expression and returns a value that is based on the truth of the expression.

Example 4-10 shows how to use the conditional operator. If the expression (op == ADD) evaluates to true, the value a + b is assigned to result; otherwise, the value a – b is assigned to result.

Example 4-10   Conditional Operator

```
module add_or_subtract( a, b, op, result );

    parameter ADD=1’b0;
    input  [7:0] a, b;
    input  op;
    output [7:0] result;

    assign result = (op == ADD) ? a+b : a-b;
endmodule
```

You can nest conditional operators to produce an `if...else if` construct. Example 4-11 shows the conditional operators used to evaluate the value of op successively and perform the correct operation.

Example 4-11 Nested Conditional Operator

```
module arithmetic( a, b, op, result );

    parameterADD=3’h0,SUB=3’h1,AND=3’h2,
             OR=3’h3, XOR=3’h4;

    input  [7:0] a,b;
    input  [2:0] op;
    output [7:0] result;

    assign result = ((op == ADD) ? a+b : (
                     (op == SUB) ? a-b : (
                     (op == AND) ? a&b : (
                     (op ==  OR) ? a|b : (
                     (op == XOR) ? a^b : (a))))));
endmodule
```

Concatenation Operators

Concatenation combines one or more expressions to form a larger vector. In the Verilog language, you indicate concatenation by listing all expressions to be concatenated, separated by commas, in curly braces ({}). Any expression, except an unsized constant, is allowed in a concatenation. For example, the concatenation `{1’b1,1’b0,1’b0}` yields the value `3’b100`.

You can also use a constant-valued repetition multiplier to repeat the concatenation of an expression. The concatenation `{1’b1,1’b0,1’b0}` can also be written as `{1’b1,{2{1’b0}}}` to yield `3’b100`. The expression `{2{`*expr*`}}` within the concatenation repeats `expr` two times.

Example 4-12 shows a concatenation that forms the value of a condition-code register.

Example 4-12 Concatenation Operator

```
output [7:0] ccr;
wire half_carry, interrupt, negative, zero, overflow, carry;
...
assign ccr = { 2’b00, half_carry, interrupt,
               negative, zero, overflow, carry };
```

Example 4-13 shows an equivalent description for the concatenation.

Example 4-13 Concatenation Equivalent

```
output [7:0] ccr;
...
assign ccr[7] = 1’b0;
assign ccr[6] = 1’b0;
assign ccr[5] = half_carry;
assign ccr[4] = interrupt;
assign ccr[3] = negative;
assign ccr[2] = zero;
assign ccr[1] = overflow;
assign ccr[0] = carry;
```

Operator Precedence

Table 4-2 lists the precedence of all operators, from highest to lowest. All operators at the same level in the table are evaluated from left to right, except the conditional operator (?:), which is evaluated from right to left.

Table 4-2 Operator Precedence

Operator	**Description**
[ ]	Bit-select or part-select
( )	Parentheses
! ~	Logical and bitwise negation
& | ~& ~| ^ ~^ ^~	Reduction operators
+ –	Unary arithmetic
{ }	Concatenation
* / %	Arithmetic
+ -	Arithmetic
<< >>	Shift
> >= < <=	Relational
== !=	Logical equality and inequality
&	Bitwise AND
^ ^~ ~^	Bitwise XOR and XNOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Conditional

Operands

You can use the following kinds of operands in an expression:

- Numbers
- Wires and registers
  - Bit-selects
  - Part-selects
- Function calls

The following sections explain each of these operands.

Numbers

A number is either a constant value or a value specified as a parameter. The expression `size-1` in Example 4-1 on page 4-2 illustrates how you can use both a parameter and a constant in an expression.

You can define constants as sized or unsized, in binary, octal, decimal, or hexadecimal bases. The default size of an unsized constant is 32 bits. See “Numbers” on page B-14 for a discussion of the number format.

Wires and Registers

Variables that represent wires as well as registers are allowed in an expression. If the variable is a multiple-bit vector and you use only the name of the variable, the entire vector is used in the expression.

Bit-selects and part-selects allow you to select single or multiple bits, respectively, from a vector. These are described in the next two sections.

Wires are described in “Module Statements and Constructs” on page 3-6, and registers are described in “Function Declarations” on page 5-3.

In the Verilog fragment shown in Example 4-14, a, b, and c are 8-bit vectors of wires. Because only the variable names appear in the expression, the entire vector of each `wire` is used in evaluation of the expression.

Example 4-14 Wire Operands

```
wire [7:0] a,b,c;
assign c = a & b;
```

Bit-Selects

A bit-select is the selection of a single bit from a `wire`, `register`, or `parameter` vector. The value of the expression in brackets (`[]`) selects the bit you want from the vector. The selected bit must be within the declared range of the vector. Example 4-15 shows a simple example of a bit-select with an expression.

Example 4-15 Bit-Select Operands

```
wire [7:0] a,b,c;
assign c[0] = a[0] & b[0];
```

Part-Selects

A part-select is the selection of a group of bits from a `wire`, `register`, or `parameter` vector. The part-select expression must be constant-valued in the Verilog language, unlike the bit-select operator. If a variable is declared with ascending or descending indexes, the part-select (when applied to that variable) must be in the same order.

You can also write the expression in Example 4-14 on page 4-17 with part-select operands, as shown in Example 4-16.

Example 4-16 Part-Select Operands

```
assign c[7:0] = a[7:0] & b[7:0]
```

Function Calls

Verilog allows you to call one function from inside an expression and use the return value from the called `function` as an operand. Functions in Verilog return a value consisting of 1 or more bits. The syntax of a function call is the function name followed by a comma-separated list of function inputs enclosed in parentheses. Example 4-17 uses the function call `legal` in an expression.

Example 4-17 Function Call Used as an Operand

```
assign error = ! legal(in1, in2);
```

Functions are described in “Function Declarations” on page 5-3.

Concatenation of Operands

Concatenation is the process of combining several single- or multiple-bit operands into one large bit vector. The use of the concatenation operator, a pair of braces (`{}`), is described in “Concatenation Operators” on page 4-13.

Example 4-18 shows two 4-bit vectors (nibble1 and nibble2) that are joined to form an 8-bit vector that is assigned to an 8-bit wire vector (byte).

Example 4-18 Concatenation of Operands

```
wire [7:0] byte;
wire [3:0] nibble1, nibble2;
assign byte = {nibble1,nibble2};
```

Expression Bit-Widths

The bit-width of an expression depends on the widths of the operands and the types of operators in the expression.

Table 4-3 shows the bit-width for each operand and operator. In the table, i, j, and k are expressions; L (i) is the bit-width of expression i.

To preserve significant bits within an expression, Verilog fills in zeros for smaller-width operands. The rules for this zero extension depend on the operand type. These rules appear in Table 4-3.

Verilog classifies expressions (and operands) as either self-determined or context-determined. A self-determined expression is one in which the width of the operands is determined solely by the expression itself. These operand widths are never extended.

Table 4-3 Expression Bit-Widths

Expression	Bit length	Comments
unsized constant	32 bits	Self-determined
sized constant	as specified	Self-determined
i + j	max(L(i),L(j))	Context-determined

Table 4-3 Expression Bit-Widths (continued)

Expression	**Bit length**	**Comments**
i – j	max(L(i),L(j))	Context-determined
i * j	max(L(i),L(j))	Context-determined
i / j	max(L(i),L(j))	Context-determined
i % j	max(L(i),L(j))	Context-determined
i & j	max(L(i),L(j))	Context-determined
i | j	max(L(i),L(j))	Context-determined
i ^ j	max(L(i),L(j))	Context-determined
i ^~ j	max(L(i),L(j))	Context-determined
~i	L(i)	Context-determined
i == j	1 bit	Self-determined
i !== j	1 bit	Self-determined
i && j	1 bit	Self-determined
i || j	1 bit	Self-determined
i > j	1 bit	Self-determined
i >= j	1 bit	Self-determined
i < j	1 bit	Self-determined
i <= j	1 bit	Self-determined
&i	1 bit	Self-determined
|i	1 bit	Self-determined
^i	1 bit	Self-determined
~&i	1 bit	Self-determined

Table 4-3 Expression Bit-Widths (continued)

Expression	Bit length	Comments
~|i	1 bit	Self-determined
~^i	1 bit	Self-determined
i >> j	L(i)	j is self-determined
{i{j}}	i*L(j)	j is self-determined
i << j	L(i)	j is self-determined
{i,...,j}	L(i)+...+L(j)	Self-determined
{i {j,...,k}}	i*(L(j)+...+L(k))	Self-determined
i ? j : k	Max(L(j),L(k))	i is self-determined

Example 4-19 shows a self-determined expression that is a concatenation of variables with known widths.

Example 4-19 Self-Determined Expression

```
output [7:0] result;
wire   [3:0] temp;

assign temp = 4’b1111;
assign result = {temp,temp};
```

The concatenation has two operands. Each operand has a width of 4 bits and a value of `4’b1111`. The resulting width of the concatenation is 8 bits, which is the sum of the width of the operands. The value of the concatenation is `8’b11111111`.

A context-determined expression is one in which the width of the expression depends on all the operand widths in the expression. For example, Verilog defines the resulting width of an addition as the greater of the widths of its two operands. The addition of two 8-bit quantities produces an 8-bit value; however, if the result of the addition is assigned to a 9-bit quantity, the addition produces a 9-bit result. Because the addition operands are context-determined, they are zero-extended to the width of the largest quantity in the entire expression.

Example 4-20 shows some context-determined expressions.

Example 4-20 Context-Determined Expressions

```
if ( ((1’b1 << 15) >> 15) == 1’b0 )
    //This expression is ALWAYS true.

if ( (((1’b1 << 15) >> 15) | 20’b0) == 1’b0 )
    //This expression is NEVER true.
```

The expression `((1’b1 << 15) >> 15)` produces a 1-bit `0` value (`1’b0`). The `1` is shifted off the left end of the vector, producing a value of `0`. The right shift has no additional effect. For a shift operator, the first operand (`1’b1`) is context-dependent; the second operand (`15`) is self-determined.

The expression `(((1’b1 << 15) >> 15) | 20’b0)` produces a 20-bit `1` value (`20’b1`). `20’b1` has a `1` in the least significant bit position and `0`s in the other 19 bit positions. Because the largest operand in the expression has a width of 20, the first operand of the shift is zero-extended to a 20-bit value. The left shift of 15 does not drop the `1` value off the left end; the right shift brings the `1` value back to the right end, resulting in a 20-bit `1` value (`20’b1`).