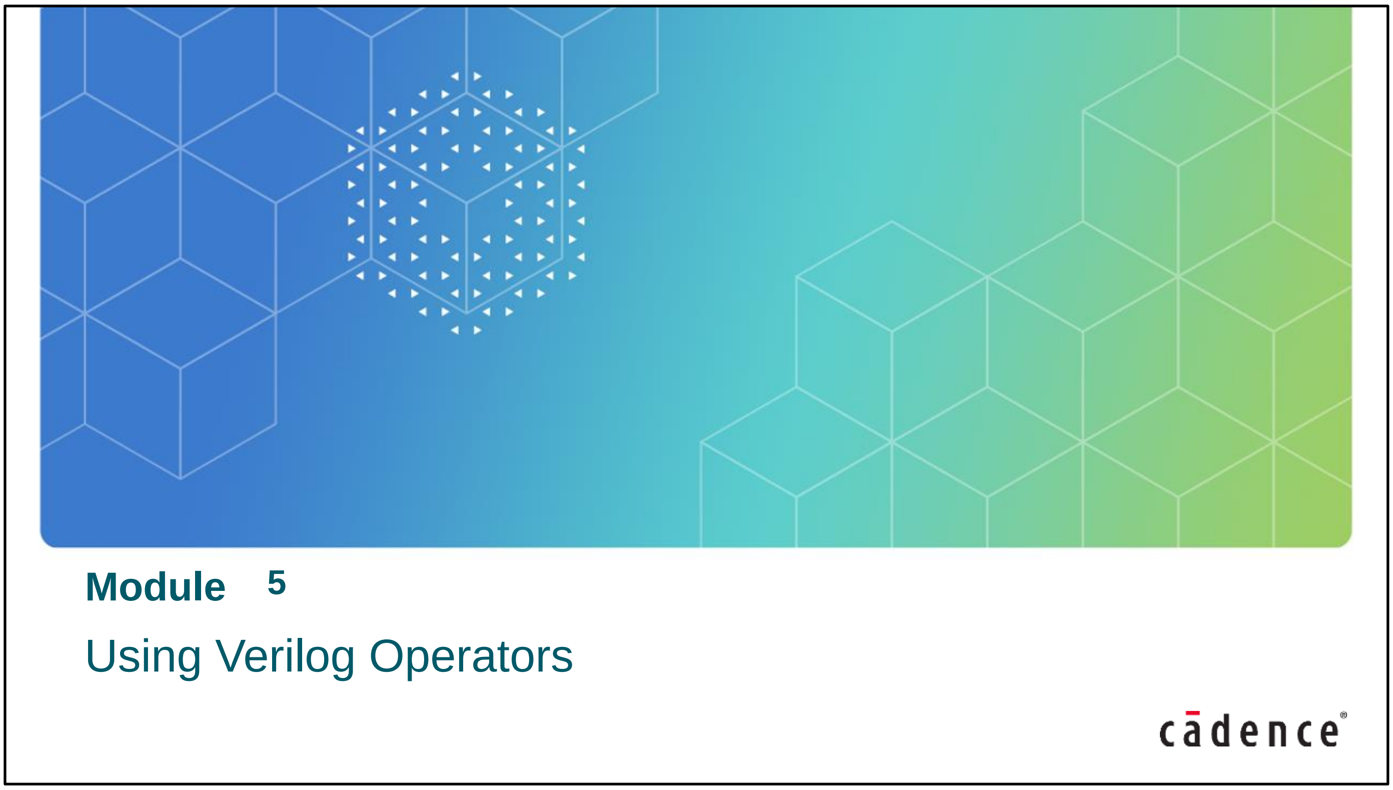
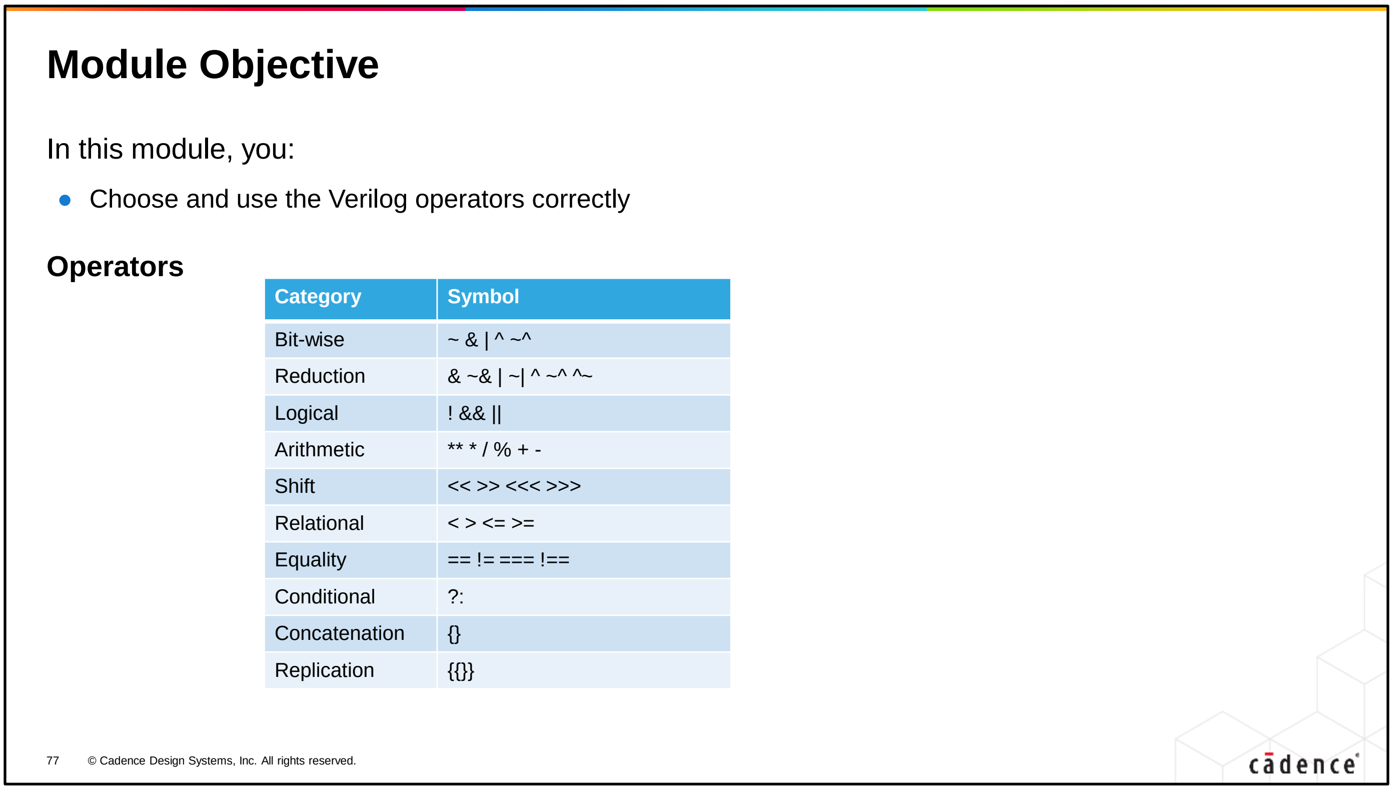


Module 5

Using Verilog Operators

cādence®

This module provides an explanation and example for each Verilog operator.



Module Objective

In this module, you:

- Choose and use the Verilog operators correctly

Operators

Category	Symbol
Bit-wise	~ & | ^ ~^
Reduction	& ~& | ~| ^ ~^ ^~
Logical	! && ||
Arithmetic	** * / % + -
Shift	<< >> <<< >>>
Relational	< > <= >=
Equality	== != === !==
Conditional	?:
Concatenation	{}
Replication	{{}}

This page does not contain notes.

Bit-Wise Operators

not	~
and	&
or	|
xor	^
xnor	~^
xnor	^~

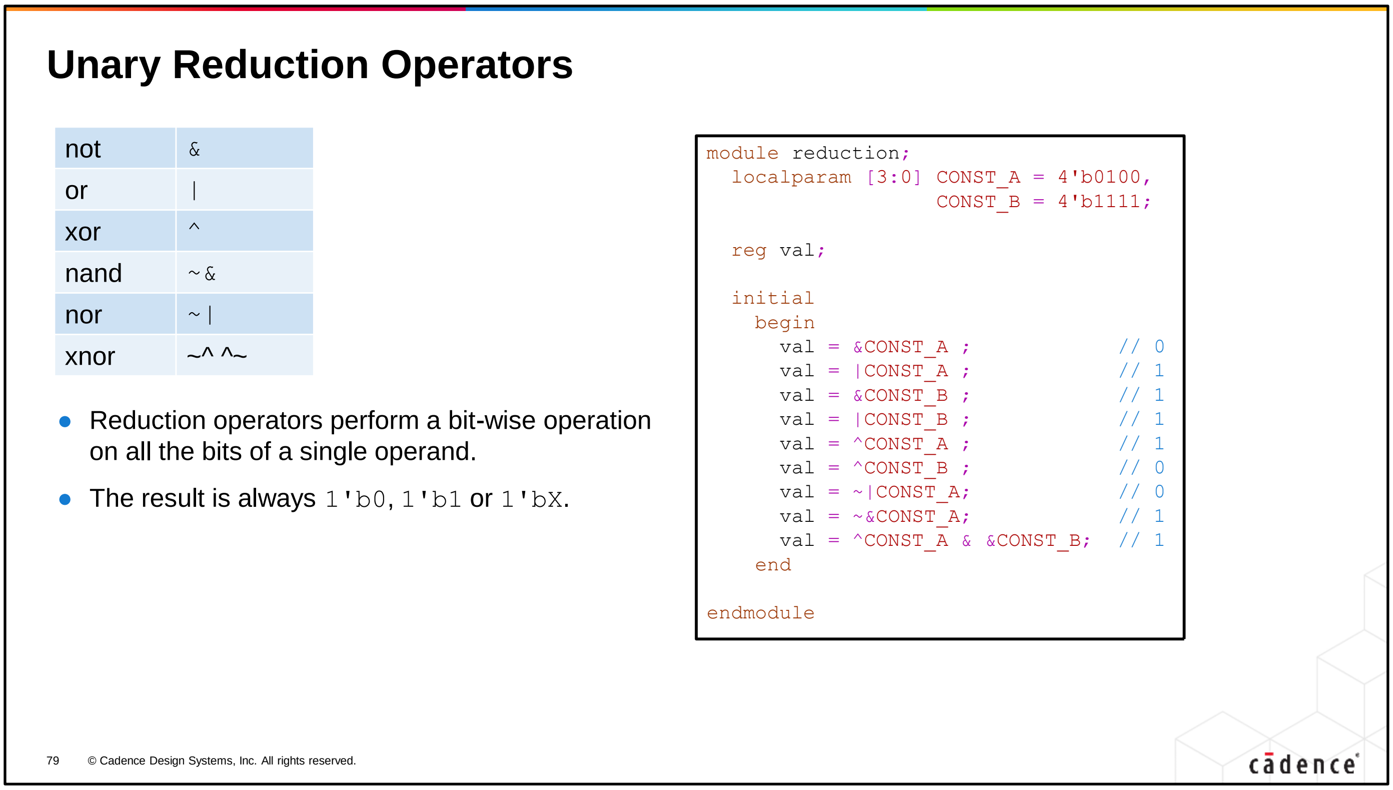
- Bit-wise operators operate on vectors.
- Operations are performed bit by bit on individual bits.
- Unknown bits in an operand do not necessarily lead to unknown bits in the result.

```
module bitwise;
  reg [3:0] rega, regb, regc;
  reg [3:0] num;

  initial
    begin
      rega = 4'b1001;
      regb = 4'b1010;
      regc = 4'b11x0;

      num = ~rega;        // num = 0110
      num = rega & 0;     // num = 0000
      num = rega & regb;  // num = 1000
      num = rega | regb;  // num = 1011
      num = regb & regc;  // num = 10x0
      num = regb | regc;  // num = 1110
    end

endmodule
```

The bitwise operators perform logical operations in a bitwise manner.

The bitwise unary negation operator inverts the logical sense of each bit of its operand. Each 0 becomes 1 and each 1 becomes 0, and each high-impedance bit becomes unknown.

The bitwise binary operators first zero-extend a smaller operand to match the width of a larger operand, and then perform logical operations on individual bit positions. Depending upon the operation, bits in one operand that are 0 or 1 can mask bits in the same position of the other operand that are high-impedance or unknown, so unknown bits in an operand do not necessarily produce unknown bits in the result.



Unary Reduction Operators

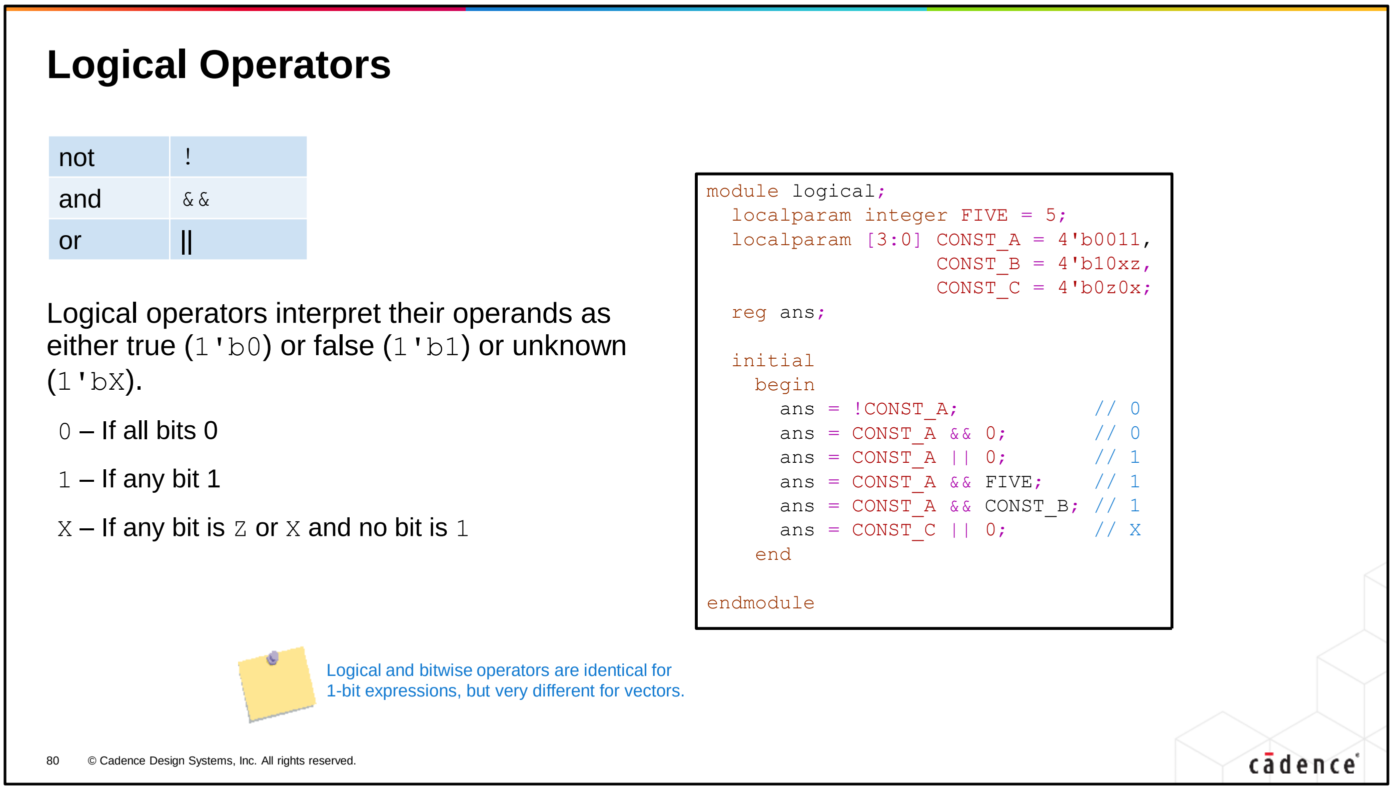
not	&
or	|
xor	^
nand	~&
nor	~|
xnor	~^ ^~

- Reduction operators perform a bit-wise operation on all the bits of a single operand.
- The result is always `1'b0`, `1'b1` or `1'bX`.

```
module reduction;
  localparam [3:0] CONST_A = 4'b0100,
                   CONST_B = 4'b1111;

  reg val;

  initial
    begin
      val = &CONST_A ;               // 0
      val = |CONST_A ;               // 1
      val = &CONST_B ;               // 1
      val = |CONST_B ;               // 1
      val = ^CONST_A ;               // 1
      val = ^CONST_B ;               // 0
      val = ~|CONST_A;               // 0
      val = ~&CONST_A;               // 1
      val = ^CONST_A & &CONST_B;     // 1
    end

endmodule
```

Unary reduction operators operate on all bits of a single operand to produce a single-bit result. The effect is as if they first applied the logical operation to the first two bits of the operand, then iteratively applied the logical operation to the current partial result and the next bit. The result of the operation is always a single bit that is 0, 1, or unknown (x).

You will see these same operators also used as bitwise binary operators. When used with a single operand, they are reduction operators.



Logical Operators

not	!
and	&&
or	||

Logical operators interpret their operands as either true (1'b0) or false (1'b1) or unknown (1'bX).

0 – If all bits 0

1 – If any bit 1

X – If any bit is Z or X and no bit is 1

```
module logical;
  localparam integer FIVE = 5;
  localparam [3:0] CONST_A = 4'b0011,
                   CONST_B = 4'b10xz,
                   CONST_C = 4'b0z0x;
  reg ans;

  initial
    begin
      ans = !CONST_A;           // 0
      ans = CONST_A && 0;       // 0
      ans = CONST_A || 0;       // 1
      ans = CONST_A && FIVE;    // 1
      ans = CONST_A && CONST_B; // 1
      ans = CONST_C || 0;       // X
    end

endmodule
```

Logical and bitwise operators are identical for 1-bit expressions, but very different for vectors.

80 © Cadence Design Systems, Inc. All rights reserved.

cādence

Logical operators reduce each operand to a single bit, and then perform a single bit operation.

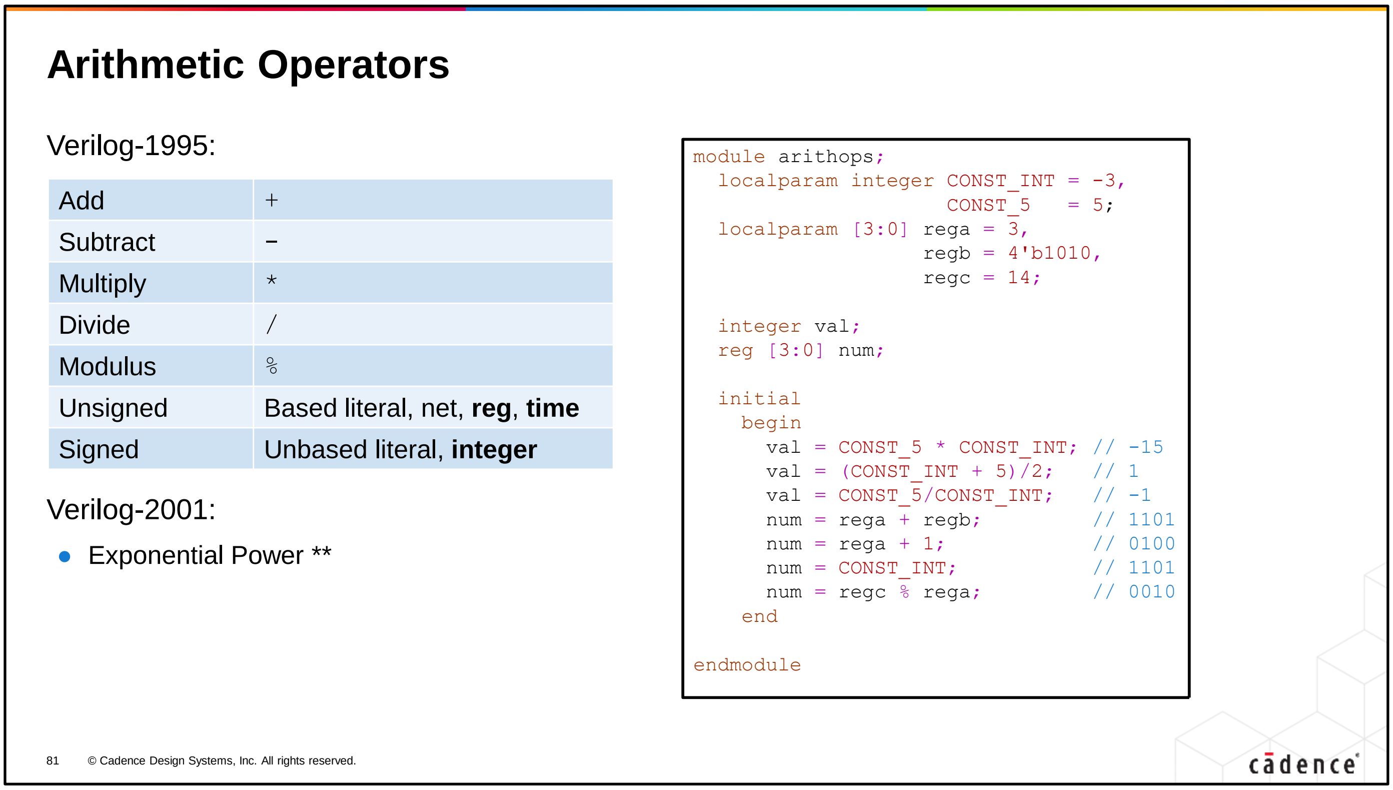
The rules for reduction of an operand are as follows:

- If the operand contains all zeroes, it reduces to logic 0.
- If the operand contains any ones, it reduces to logic 1.
- If the operand contains no ones, but does contain one or more high-impedance or unknown values, it reduces to unknown, because its logical value is unknown.

The unary logical negation operator then inverts the logical sense of its operand. 0 becomes 1 and 1 becomes 0.

The binary conjunction and disjunction operators produce the logical conjunction and disjunction, respectively, of their operands.

Design Tip: Generally logical operators should be avoided unless explicitly required. It is very easy to get into the habit of using logical operators everywhere. They are identical to bitwise operators for single bit expressions, but give very different results when using vectors.



Arithmetic Operators

Verilog-1995:

Add	+
Subtract	–
Multiply	*
Divide	/
Modulus	%
Unsigned	Based literal, net, **reg**, **time**
Signed	Unbased literal, **integer**

Verilog-2001:

- Exponential Power **

```
module arithops;
  localparam integer CONST_INT = -3,
                     CONST_5   = 5;
  localparam [3:0] rega = 3,
                   regb = 4'b1010,
                   regc = 14;

  integer val;
  reg [3:0] num;

  initial
    begin
      val = CONST_5 * CONST_INT; // -15
      val = (CONST_INT + 5)/2;   // 1
      val = CONST_5/CONST_INT;   // -1
      num = rega + regb;         // 1101
      num = rega + 1;            // 0100
      num = CONST_INT;           // 1101
      num = regc % rega;         // 0010
    end

endmodule
```

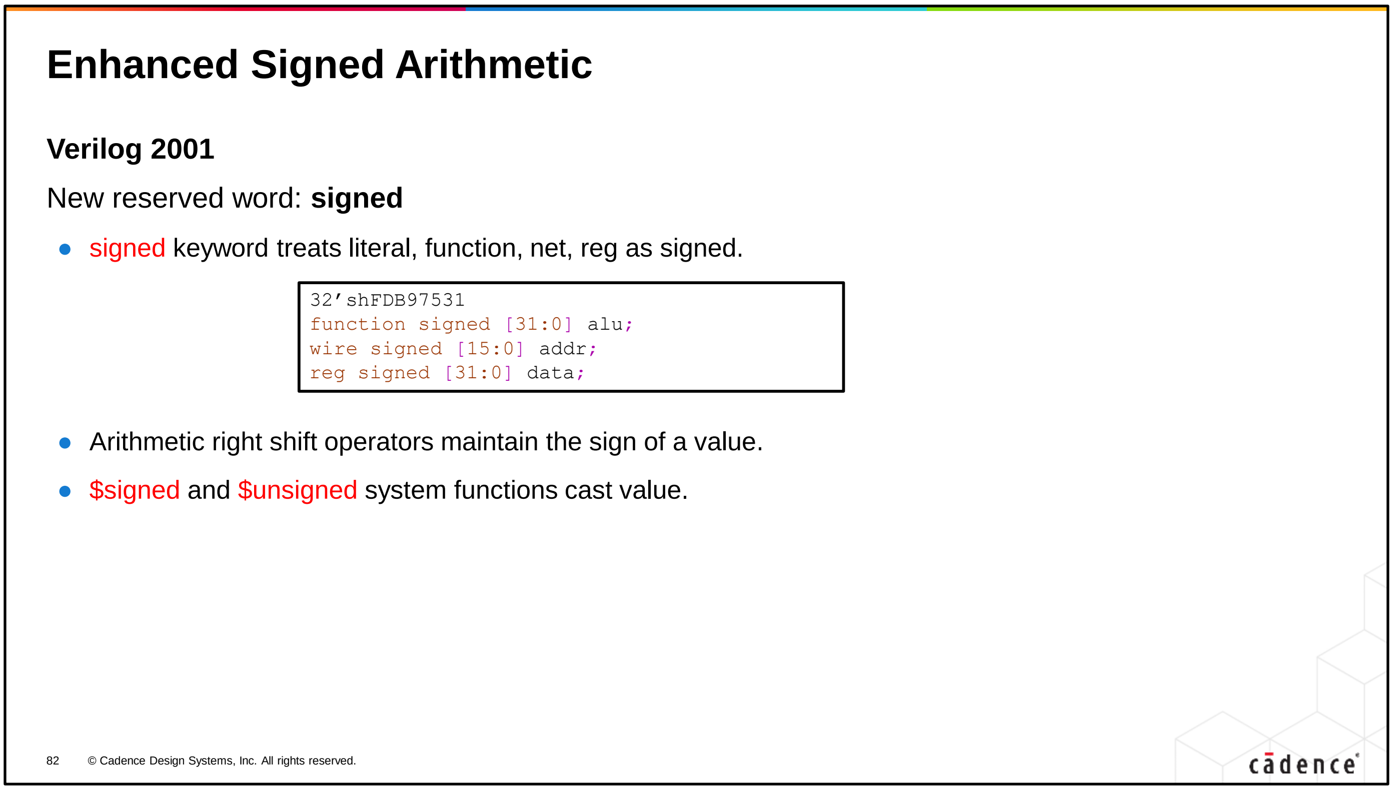
81 © Cadence Design Systems, Inc. All rights reserved.

The Verilog 1995 arithmetic operators are add, subtract, multiply, divide and modulus. Verilog 2001 added the power operator.

This example declares two *integer* parameters, three 4-bit vector *reg* parameters, an *integer* variable and a 4-bit vector *reg* variable, and performs some operations with them. See where integer division discards the fractional part. See where assigning 3 to an unsigned 4-bit vector *reg* keeps the same rightmost four bits but now interprets the value as +13.

Some additional information about arithmetic operators:

- Any z or x bit in either operand produces an unknown (x) result.
- Integer division discards any remaining fractional part.
- Division or modulo by 0 produces an unknown (x) result.
- Raising 0 to a negative power produces an unspecified result.
- Raising a negative value to a real power produces an unspecified result.



Enhanced Signed Arithmetic

Verilog 2001

New reserved word: **signed**

- signed keyword treats literal, function, net, reg as signed.

```
32'shFDB97531
function signed [31:0] alu;
wire signed [15:0] addr;
reg signed [31:0] data;
```

- Arithmetic right shift operators maintain the sign of a value.
- $signed and $unsigned system functions cast value.

The parser interprets an integer with no base specifier as a signed value in 2's complement form.

- `intA = -12 / 3 ;`

▪ The left operand is a 32-bit signed value 12, negated.

▪ The expression result is –4.

The parser interprets an integer with an unsigned base specifier as an unsigned value.

- `intA = -'d12 / 3 ;`

▪ The left operand is a 32-bit unsigned value 12, negated.

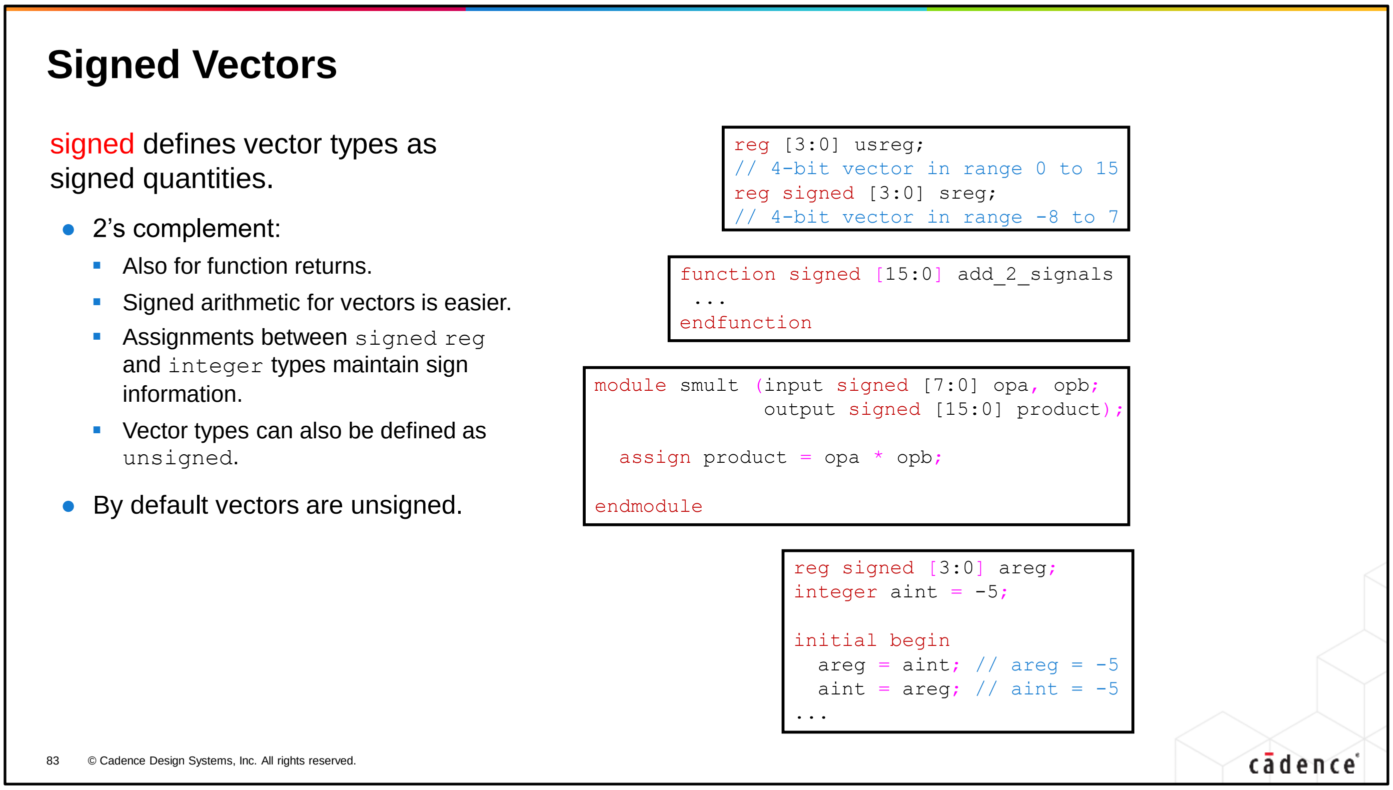
▪ The expression result is 1431655761.

The parser interprets an integer with an signed base specifier as an signed value.

- `intA = -4'sd12 / 3 ;`

▪ The left operand is a 4-bit pattern of 12 (1100) interpreted as a signed number (–4) and then negated (4).

▪ The expression result is 1.



Signed Vectors

signed defines vector types as signed quantities.

- 2’s complement:
  - Also for function returns.
  - Signed arithmetic for vectors is easier.
  - Assignments between `signed reg` and `integer` types maintain sign information.
  - Vector types can also be defined as `unsigned`.
- By default vectors are unsigned.

```
reg [3:0] usreg;
// 4-bit vector in range 0 to 15
reg signed [3:0] sreg;
// 4-bit vector in range -8 to 7
```

```
function signed [15:0] add_2_signals
 ...
endfunction
```

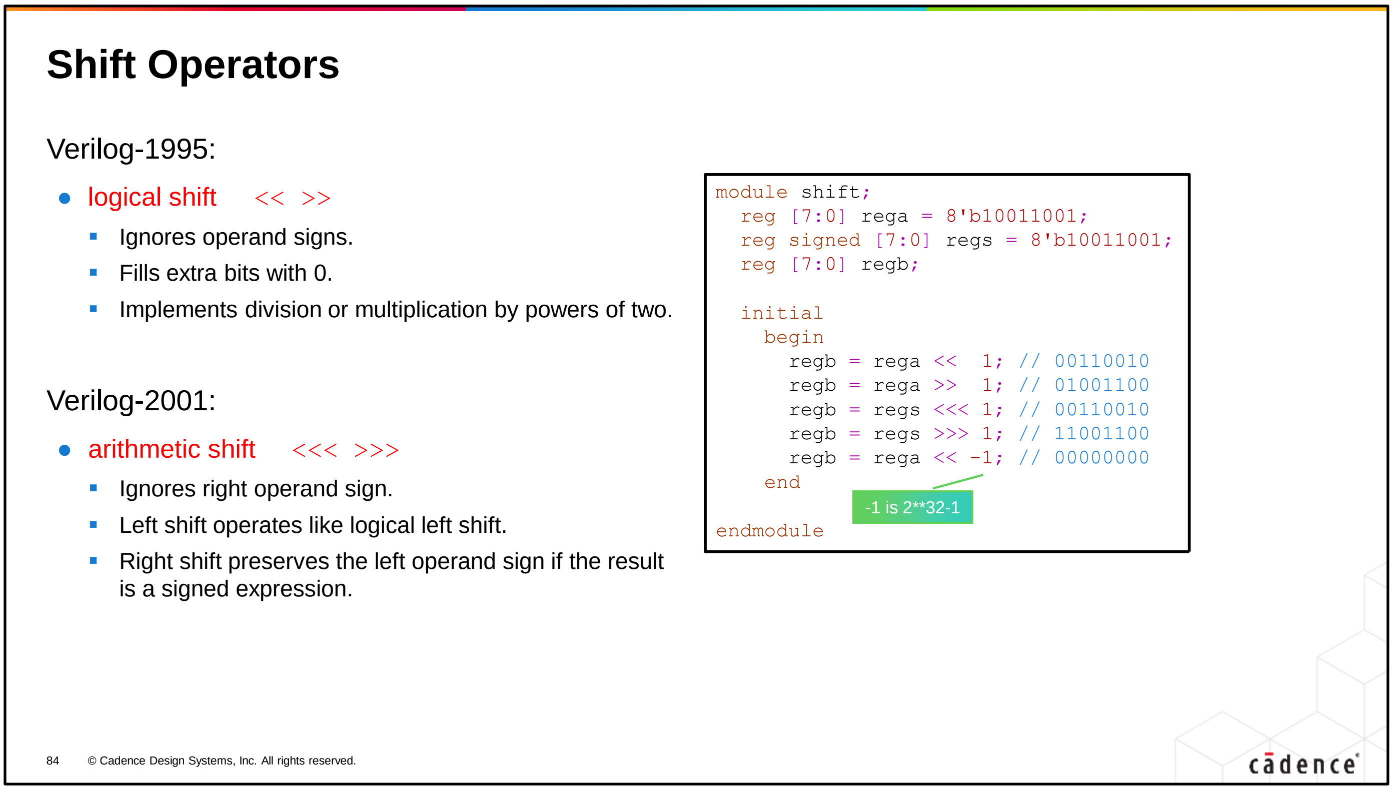
```
module smult (input signed [7:0] opa, opb;
              output signed [15:0] product);

  assign product = opa * opb;

endmodule
```

```
reg signed [3:0] areg;
integer aint = -5;

initial begin
  areg = aint; // areg = -5
  aint = areg; // aint = -5
...
```

83 © Cadence Design Systems, Inc. All rights reserved.

cadence

For Verilog-2001, you can declare signed vectors. Vectors are by default unsigned. An unsigned vector is treated as signed for accesses through a signed port.



Shift Operators

Verilog-1995:

- logical shift   << >>
  - Ignores operand signs.
  - Fills extra bits with 0.
  - Implements division or multiplication by powers of two.

Verilog-2001:

- arithmetic shift   <<< >>>
  - Ignores right operand sign.
  - Left shift operates like logical left shift.
  - Right shift preserves the left operand sign if the result is a signed expression.

```
module shift;
  reg [7:0] rega = 8'b10011001;
  reg signed [7:0] regs = 8'b10011001;
  reg [7:0] regb;

  initial
    begin
      regb = rega <<  1; // 00110010
      regb = rega >>  1; // 01001100
      regb = regs <<< 1; // 00110010
      regb = regs >>> 1; // 11001100
      regb = rega << -1; // 00000000
    end

endmodule
```

-1 is 2**32-1

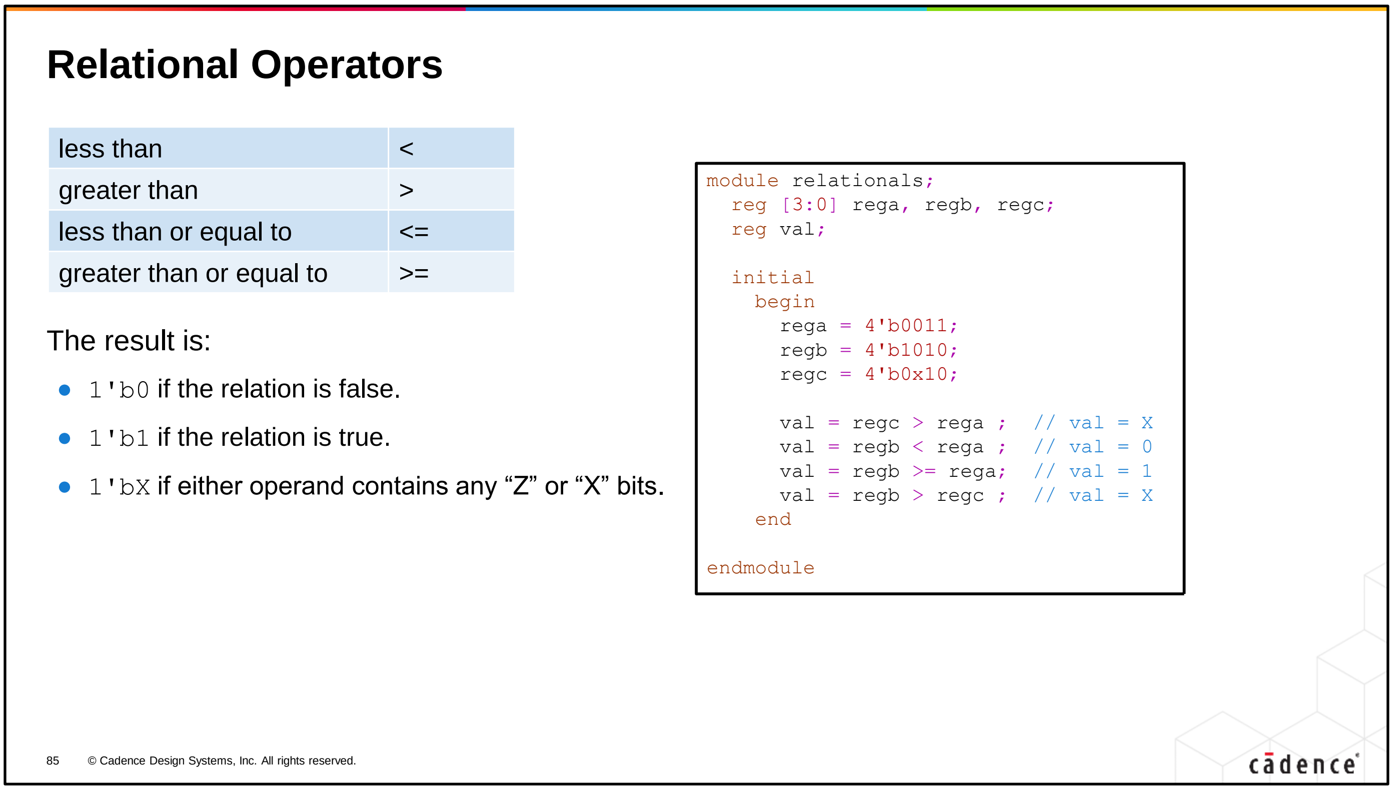
84 © Cadence Design Systems, Inc. All rights reserved.

cadence

The logical shift operators shift the left operand by the number of bit positions given by the right operand interpreted as a positive number, filling vacated bit positions with 0. You can use the logical shift operators to implement integer division or multiplication by powers of 2.

Verilog 2001 added the arithmetic shift operators. The arithmetic left shift operator operates exactly as does the logical left shift operator. The arithmetic right shift operator preserves the sign bit if the resulting expression is signed.

The example initializes an 8-bit vector rega to value 153 (8'h99). Left-shifting this value once is equivalent to doubling its value, and placing the result back into an 8-bit vector reg. The data truncates and the value gets back down to 50 (8'h32). Right-shifting rega value once is equivalent to halving its value and losing the fractional part, so produces 76 (8'h4c). Left-shifting by –1 is equivalent to left-shifting a very large number of times, so produces 0.



Relational Operators

less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=

The result is:

- `1'b0` if the relation is false.
- `1'b1` if the relation is true.
- `1'bX` if either operand contains any “Z” or “X” bits.

```
module relationals;
  reg [3:0] rega, regb, regc;
  reg val;

  initial
    begin
      rega = 4'b0011;
      regb = 4'b1010;
      regc = 4'b0x10;

      val = regc > rega ;  // val = X
      val = regb < rega ;  // val = 0
      val = regb >= rega;  // val = 1
      val = regb > regc ;  // val = X
    end

endmodule
```

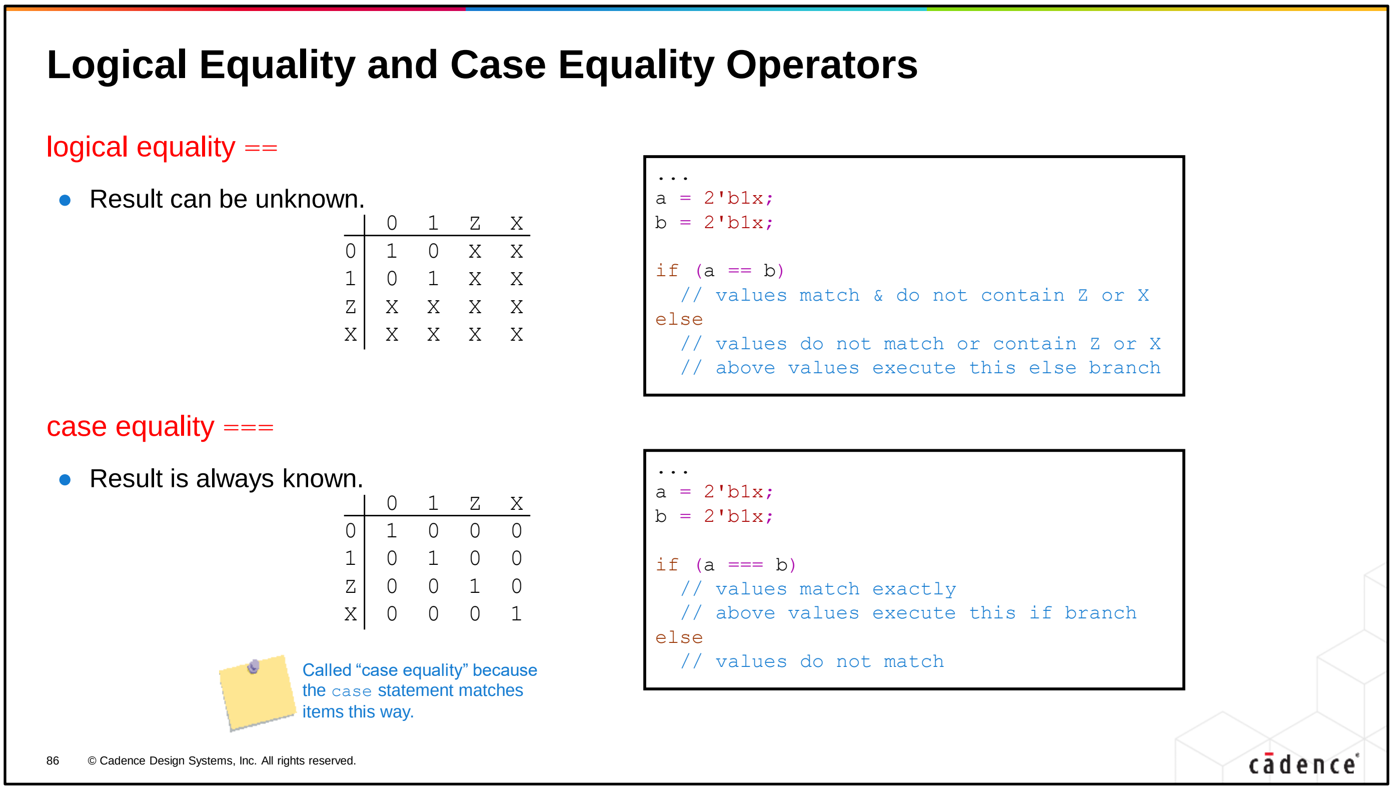
85 © Cadence Design Systems, Inc. All rights reserved.

If at least one operand is unsigned, the comparison treats both operands as unsigned and zero-extends a smaller operand to match the width of a wider operand.

If both operands are signed integer quantities, the operation sign-extends a smaller operand to match the width of a wider operand and treats the comparison as a signed integer comparison.

If at least one operand is real, the operation if necessary converts the other operand to real and treats the comparison as a real comparison.

The result is 0 if the relation is false and 1 if the relation is true. Any high-impedance (z) or unknown (x) bit in either operand produces an unknown result regardless of the truth of the relation.



Logical Equality and Case Equality Operators

logical equality ==

- Result can be unknown.

	0	1	Z	X
0	1	0	X	X
1	0	1	X	X
Z	X	X	X	X
X	X	X	X	X

```
...
a = 2'b1x;
b = 2'b1x;

if (a == b)
  // values match & do not contain Z or X
else
  // values do not match or contain Z or X
  // above values execute this else branch
```

case equality ===

- Result is always known.

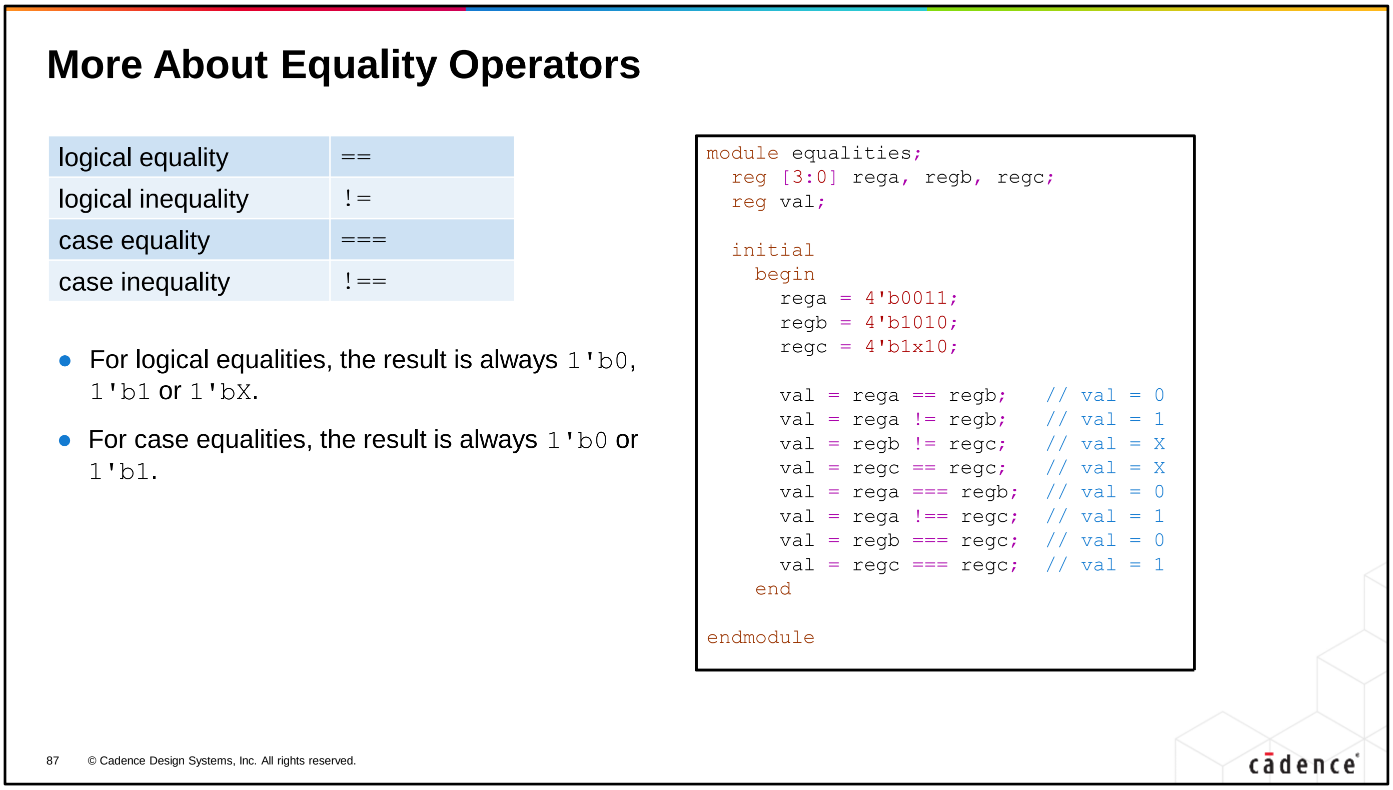
	0	1	Z	X
0	1	0	0	0
1	0	1	0	0
Z	0	0	1	0
X	0	0	0	1

```
...
a = 2'b1x;
b = 2'b1x;

if (a === b)
  // values match exactly
  // above values execute this if branch
else
  // values do not match
```

Called “case equality” because the `case` statement matches items this way.

The difference between logical equality and case equality is the handling of high-impedance and unknown values. The logical equality treats high-impedance and unknown bits as truly unknown bit values, while the case equality treats high-impedance and unknown bits as values to be matched. A case equality operation thus always produces a 0 or 1 result and never an unknown result.

The case equality operator gets its name from the fact that the case statement, which you will see later, matches items in the same manner. Some people informally call it the “identity” operator because it checks that the bits are identical instead of that their values are the same.



More About Equality Operators

logical equality	`==`
logical inequality	`!=`
case equality	`===`
case inequality	`!==`

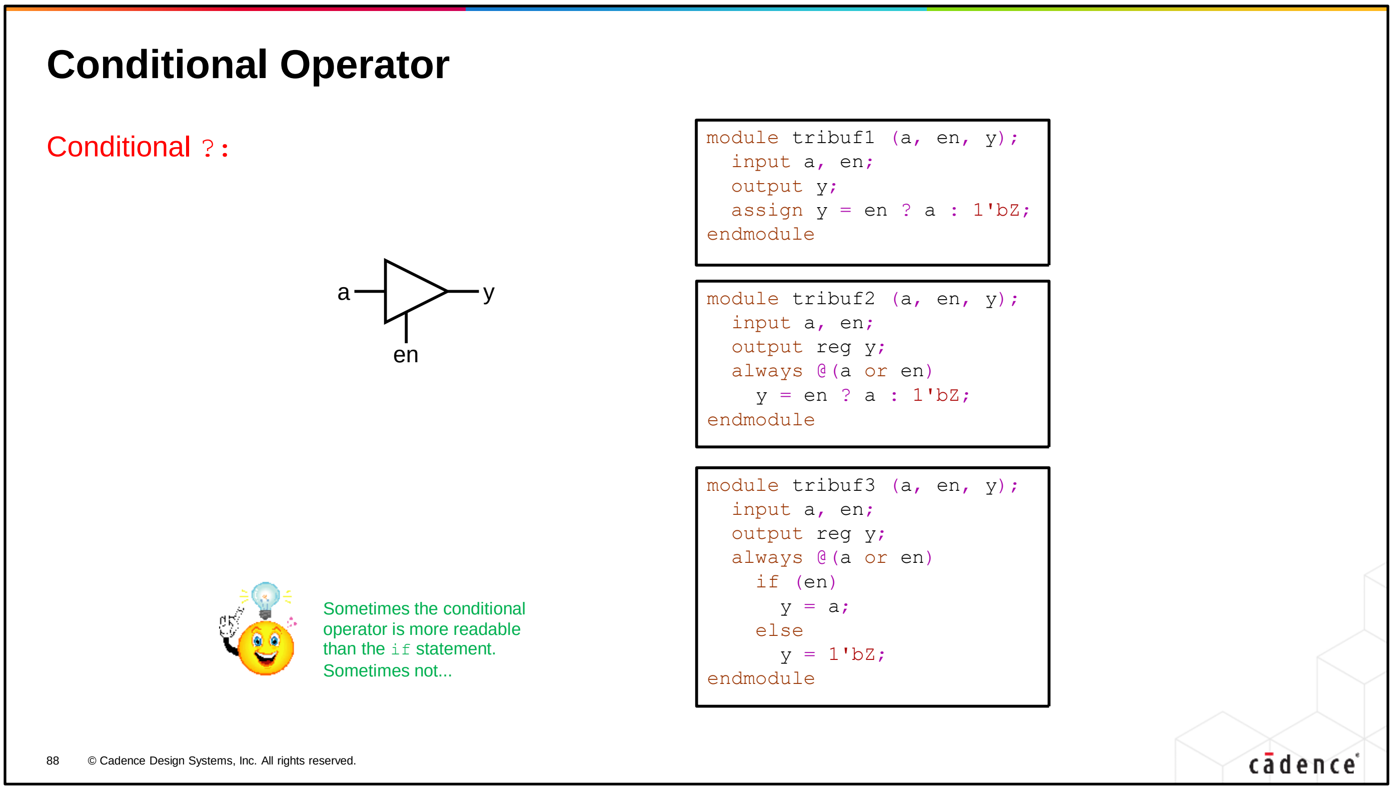
- For logical equalities, the result is always `1'b0`, `1'b1` or `1'bX`.
- For case equalities, the result is always `1'b0` or `1'b1`.

```
module equalities;
  reg [3:0] rega, regb, regc;
  reg val;

  initial
    begin
      rega = 4'b0011;
      regb = 4'b1010;
      regc = 4'b1x10;

      val = rega == regb;    // val = 0
      val = rega != regb;    // val = 1
      val = regb != regc;    // val = X
      val = regc == regc;    // val = X
      val = rega === regb;   // val = 0
      val = rega !== regc;   // val = 1
      val = regb === regc;   // val = 0
      val = regc === regc;   // val = 1
    end

endmodule
```

87 © Cadence Design Systems, Inc. All rights reserved.

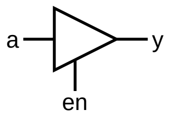
Here is another example illustrating the difference between the logical equality operators and the case equality operators.

The difference between logical equality and case equality is the handling of high-impedance (z) and unknown (x) values. The logical equality treats high-impedance and unknown bits as truly unknown bit values, while the case equality treats high-impedance and unknown bits as values to be matched. A case equality operation thus always produces a 0 or 1 result and never an unknown result.



Conditional Operator

Conditional `?:`



```
module tribuf1 (a, en, y);
  input a, en;
  output y;
  assign y = en ? a : 1'bZ;
endmodule
```

```
module tribuf2 (a, en, y);
  input a, en;
  output reg y;
  always @(a or en)
    y = en ? a : 1'bZ;
endmodule
```

```
module tribuf3 (a, en, y);
  input a, en;
  output reg y;
  always @(a or en)
    if (en)
      y = a;
    else
      y = 1'bZ;
endmodule
```

Sometimes the conditional operator is more readable than the `if` statement. Sometimes not...

88 © Cadence Design Systems, Inc. All rights reserved.

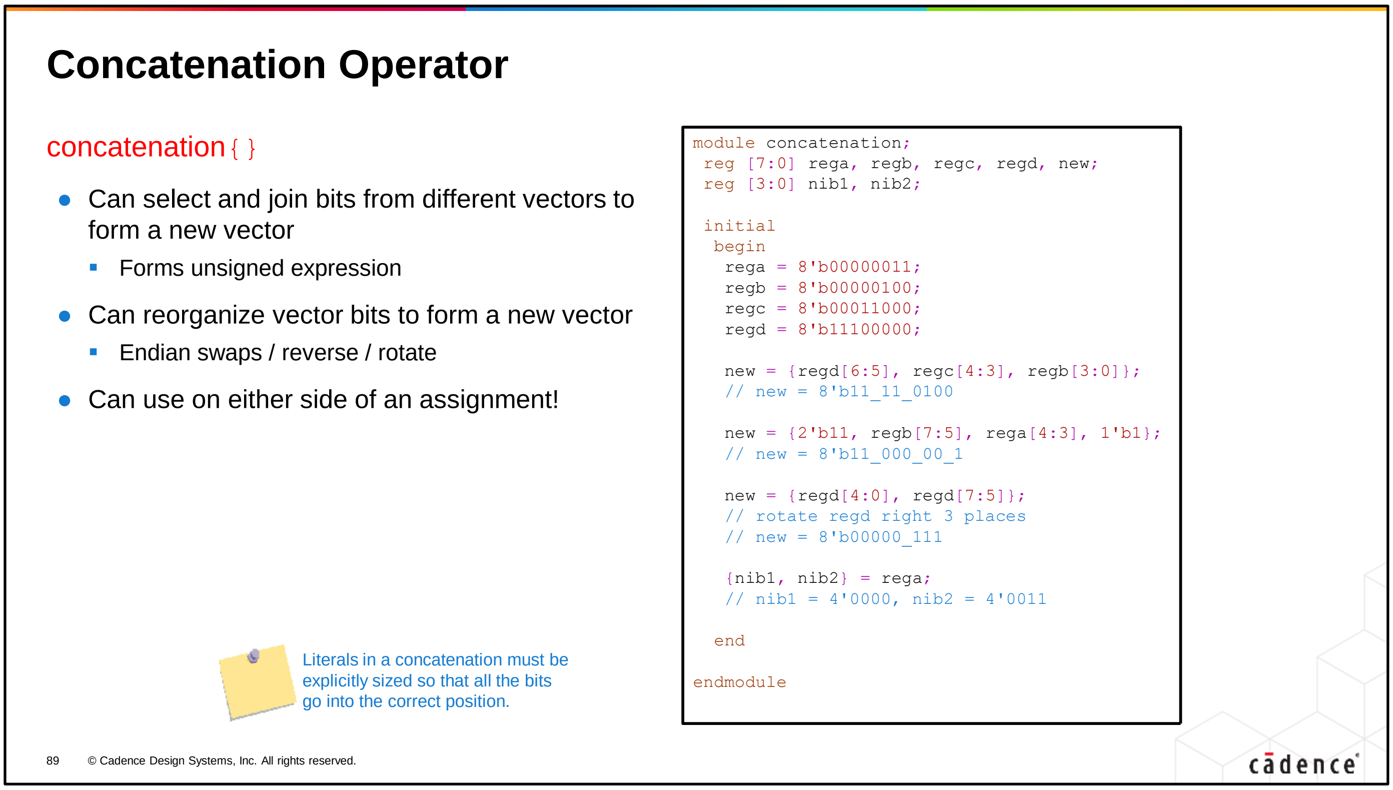
cādence

The conditional operator is a ternary operator. It has three operands.

- If the first operand, the condition expression, is 1, then the operation result is the value of the second operand.
- If the first operand is 0, then the operation result is the value of the third operand.
- If the first operand is unknown (x), then the operation result is the value of the second operand merged with the value of the third operand so that if any bit position has the same value in both operands, then that bit position of the result also has that value.

So the conditional operator selects between two operands like a 2-to-1 multiplexor.

The operands can be any expression. It is very common to nest conditional operations.



Concatenation Operator

concatenation { }

- Can select and join bits from different vectors to form a new vector
  - Forms unsigned expression
- Can reorganize vector bits to form a new vector
  - Endian swaps / reverse / rotate
- Can use on either side of an assignment!

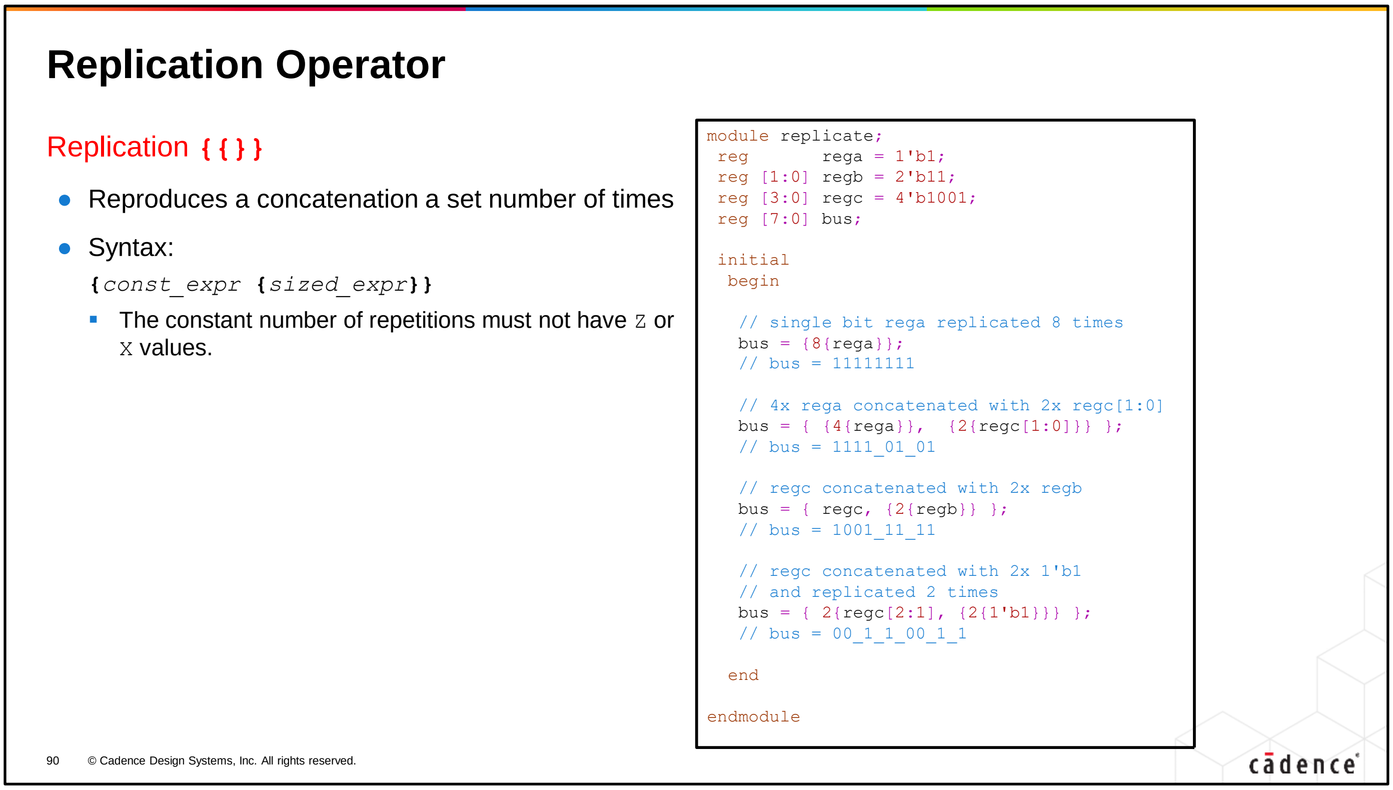
Literals in a concatenation must be explicitly sized so that all the bits go into the correct position.

```
module concatenation;
 reg [7:0] rega, regb, regc, regd, new;
 reg [3:0] nib1, nib2;

 initial
  begin
   rega = 8'b00000011;
   regb = 8'b00000100;
   regc = 8'b00011000;
   regd = 8'b11100000;

   new = {regd[6:5], regc[4:3], regb[3:0]};
   // new = 8'b11_11_0100

   new = {2'b11, regb[7:5], rega[4:3], 1'b1};
   // new = 8'b11_000_00_1

   new = {regd[4:0], regd[7:5]};
   // rotate regd right 3 places
   // new = 8'b00000_111

   {nib1, nib2} = rega;
   // nib1 = 4'0000, nib2 = 4'0011

  end

endmodule
```

A concatenation operation joins together the bits of one or more comma-separated operands. You must size literal constant operands so that the operation can determine exactly where to place each bit.

Here are some examples that fail to size their operands:

```
a[7:0] = {5'b01010, 2}; //decimal value 2 unsized
c[3:0] = {3'b011, 'b0}; //binary value 'b0 unsized
```



Replication Operator

Replication **{ { } }**

- Reproduces a concatenation a set number of times
- Syntax:

  `{const_expr {sized_expr}}`

  - The constant number of repetitions must not have `z` or `x` values.

```
module replicate;
 reg       rega = 1'b1;
 reg [1:0] regb = 2'b11;
 reg [3:0] regc = 4'b1001;
 reg [7:0] bus;

 initial
  begin

   // single bit rega replicated 8 times
   bus = {8{rega}};
   // bus = 11111111

   // 4x rega concatenated with 2x regc[1:0]
   bus = { {4{rega}},  {2{regc[1:0]}} };
   // bus = 1111_01_01

   // regc concatenated with 2x regb
   bus = { regc, {2{regb}} };
   // bus = 1001_11_11

   // regc concatenated with 2x 1'b1
   // and replicated 2 times
   bus = { 2{regc[2:1], {2{1'b1}}} };
   // bus = 00_1_1_00_1_1

  end

endmodule
```

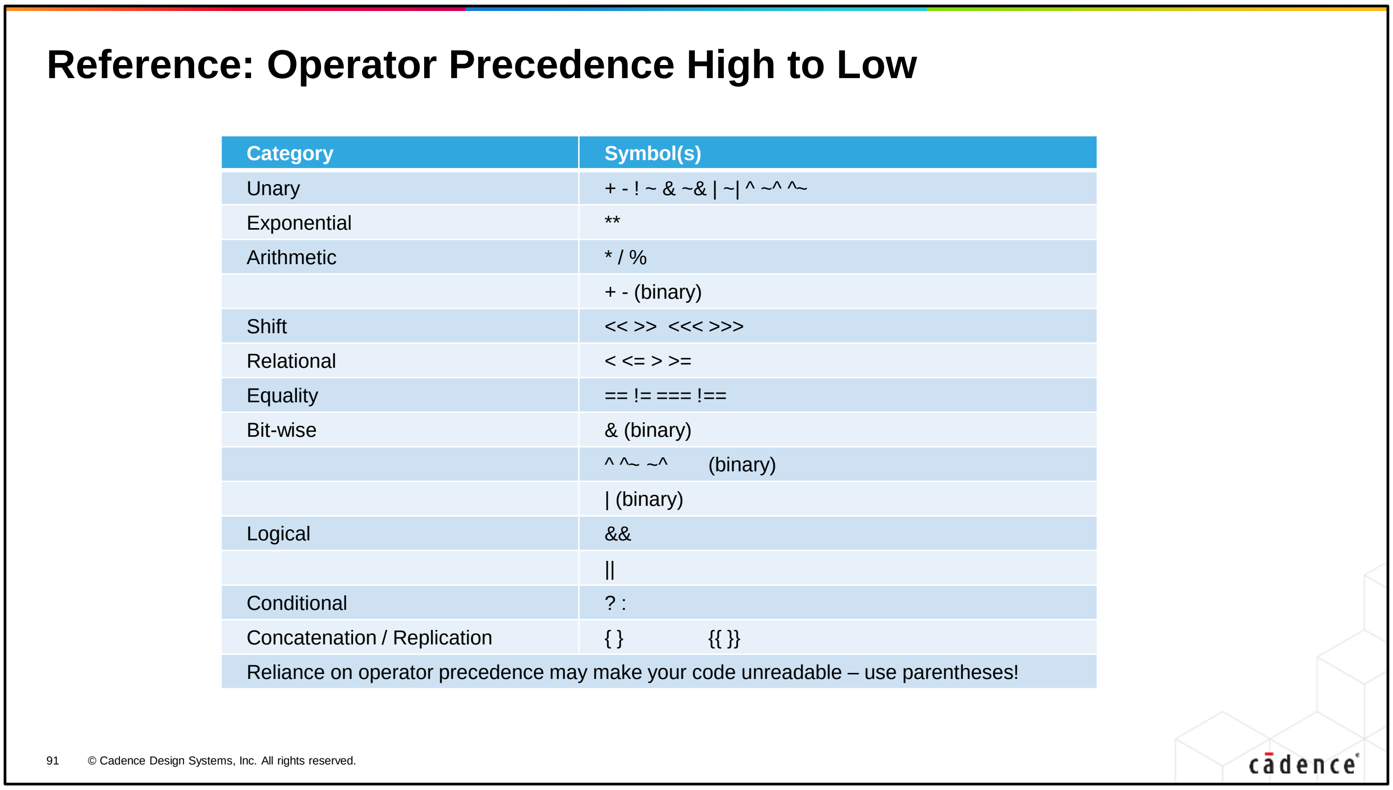
90 © Cadence Design Systems, Inc. All rights reserved.

cadence

A replication operation replicates a concatenation a nonnegative constant number of times. A replication operation applies only to a concatenation. You will not see it in any other context. The replication count cannot have any high-impedance (z) or unknown (x) values.

This example:

- 1st – Replicates the value of the single-bit rega variable value eight times to form an 8-bit value it assigns to the bus variable;
- 2nd – Concatenates four replications of the value of the single-bit rega variable with two replications of the value of the lowest two bits of the regc variable to form an 8-bit value it assigns to the bus variable;
- 3rd – Concatenates the value of the four-bit regc variable with two replications of the value of the two-bit regb variable to form an 8-bit value it assigns to the bus variable; and
- 4th – Concatenates the value of the middle two bits of the regc variable with two replications of the sized literal constant 1'b1, and replicates that 4-bit concatenation twice to form an 8-bit value it assigns to the bus variable.



Reference: Operator Precedence High to Low

Category	Symbol(s)
Unary	+ - ! ~ & ~& | ~| ^ ~^ ^~
Exponential	**
Arithmetic	* / %
	+ - (binary)
Shift	<< >> <<< >>>
Relational	< <= > >=
Equality	== != === !==
Bit-wise	& (binary)
	^ ^~ ~^ (binary)
	| (binary)
Logical	&&
	||
Conditional	? :
Concatenation / Replication	{ } {{ }}
Reliance on operator precedence may make your code unreadable – use parentheses!	

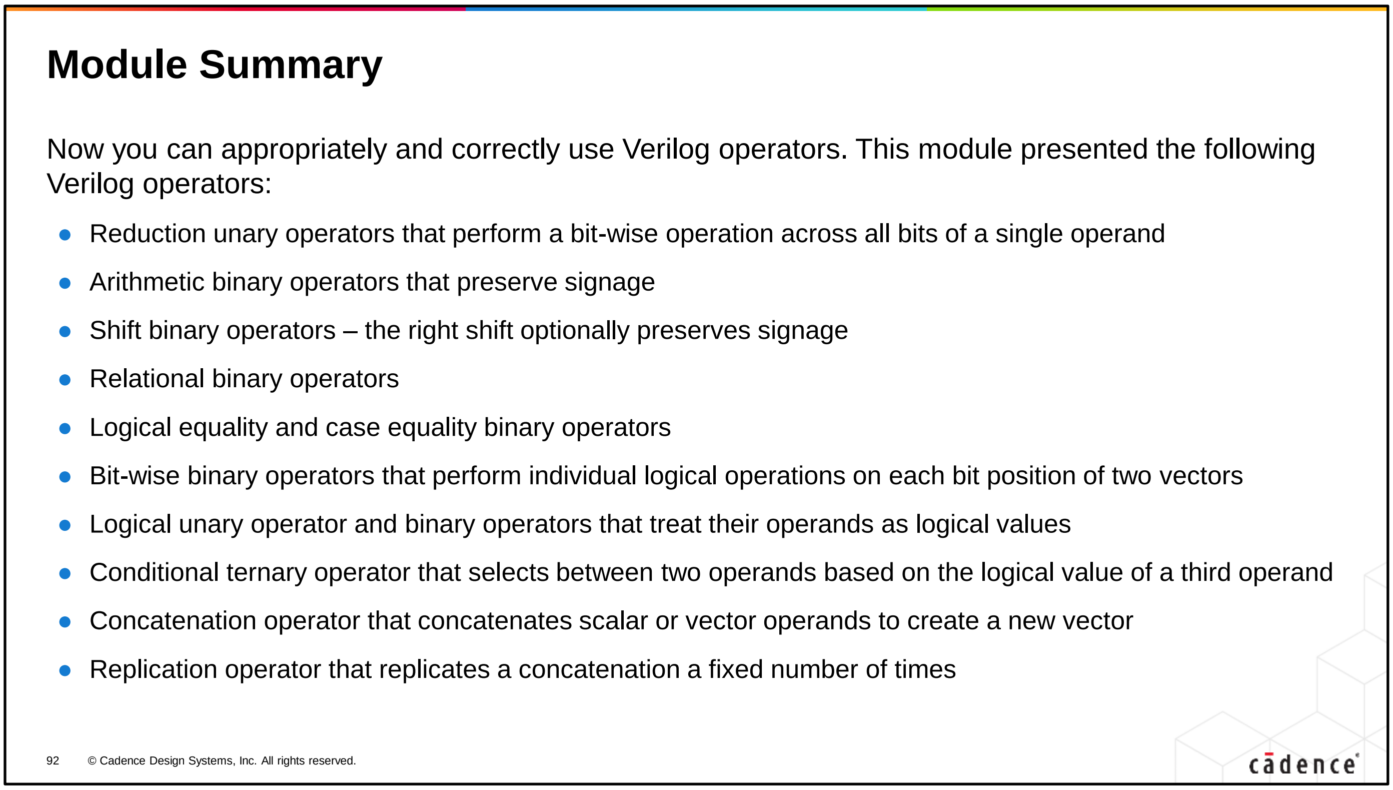
Your reliance on operator precedence can make your code unreadable to the great majority of people that do not memorize this table. Your co-workers will appreciate your appropriate use of parenthesized expressions.

Quickly now, without looking at this table, insert parentheses to indicate the grouping of these expressions…

```
a ? ~ b * c << d : e < f ^ g || h
```

Give up? Don’t you wish it was already parenthesized for you?

```
a ? ((~b * c) << d) : (((e < f) ^ g) || h)
```



Module Summary

Now you can appropriately and correctly use Verilog operators. This module presented the following Verilog operators:

- Reduction unary operators that perform a bit-wise operation across all bits of a single operand
- Arithmetic binary operators that preserve signage
- Shift binary operators – the right shift optionally preserves signage
- Relational binary operators
- Logical equality and case equality binary operators
- Bit-wise binary operators that perform individual logical operations on each bit position of two vectors
- Logical unary operator and binary operators that treat their operands as logical values
- Conditional ternary operator that selects between two operands based on the logical value of a third operand
- Concatenation operator that concatenates scalar or vector operands to create a new vector
- Replication operator that replicates a concatenation a fixed number of times

You can now appropriately and effectively use Verilog operators. To do that, you need to know what operators are available and what they do. This module provided an explanation and example for each Verilog operator.

Module Review

1. How wide is the result of a logical `&&` operation?
2. Explain the difference between the `&&` and `&` operators.
3. TRUE or FALSE: You must explicitly size literals in a concatenation.
4. Given the statement `regx = 4'b0101;` what is the value of:
   `bus = { 2 {regx[3:1], {3{1'b0,regx[0]}}} };`

cādence

This page does not contain notes.

Module Review Solutions

1. How wide is the result of a logical `&&` operation?
   - The logical && operator reduces each operand to a single bit (1'b0, 1'b1 or 1'bx), then ANDs these bits together to give a single bit result.
2. Explain the difference between the `&&` and `&` operators.
   - The logical && operator reduces each operand to a single bit, then ANDs these bits to give a single bit result.
     The bitwise & operator does a bit-by-bit AND of its operands, from LSB to MSB, producing a result which is the same width as the longest operand.
3. TRUE or FALSE: You must explicitly size literals in a concatenation.
   - TRUE. All operands in a concatenation must have a size. For example: {3{'b1}} is illegal because 'b1 is not sized, while {3{1'b1}} is legal.
4. Given the statement `regx = 4'b0101;` what is the value of:
   `bus = { 2 {regx[3:1], {3{1'b0,regx[0]}}} };`
   - Bus = 18'b010_01_01_01_010_01_01_01;

This page does not contain notes.

Module Exercise

Fix this code so that it preserves the sign of the value.

```
integer i;
always @(...)
  begin
    ...
    i = -4;   // -4
    ...
    i = i >> 1; // 2147483646
    ...
  end
```

This page does not contain notes.

Module Exercise Solution

Fix this code so that it preserves the sign of the value.

```
integer i;
always @(...)
  begin
    ...
    i = -4;   // -4
    ...
    i = i >> 1; // 2147483646
    ...
  end
```

Solution:

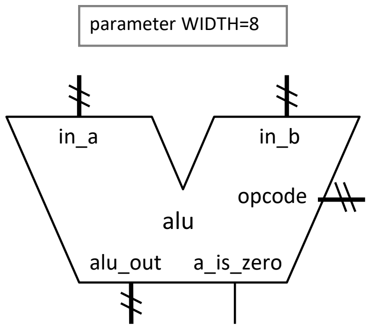
```
integer i;
always @(...)
  begin
    ...
    i = -4;   // -4
    ...
    i = i >>> 1; // -2
    ...
  end
```

This page does not contain notes.

Lab

Lab 5-1 Modeling the Arithmetic Logic Unit

- Use Verilog operators while describing a parameterized-width arithmetic logic unit (ALU).



Your objective is to appropriately choose and correctly use the Verilog operators.

For this lab, you use Verilog operators while describing a parameterized-width arithmetic/logic unit (ALU).