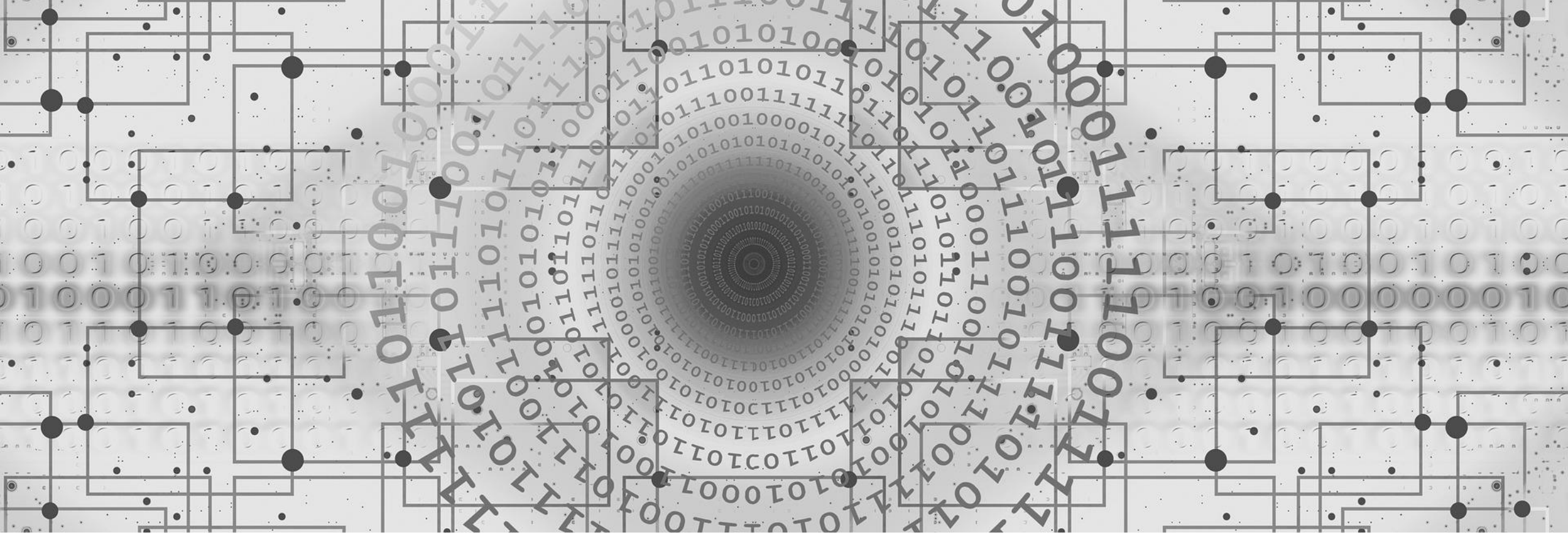


SystemVerilog Operators and Number System

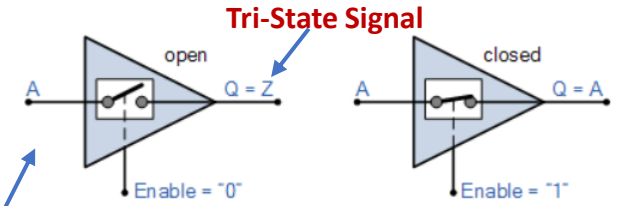
ECE-111 Advanced Digital Design Project

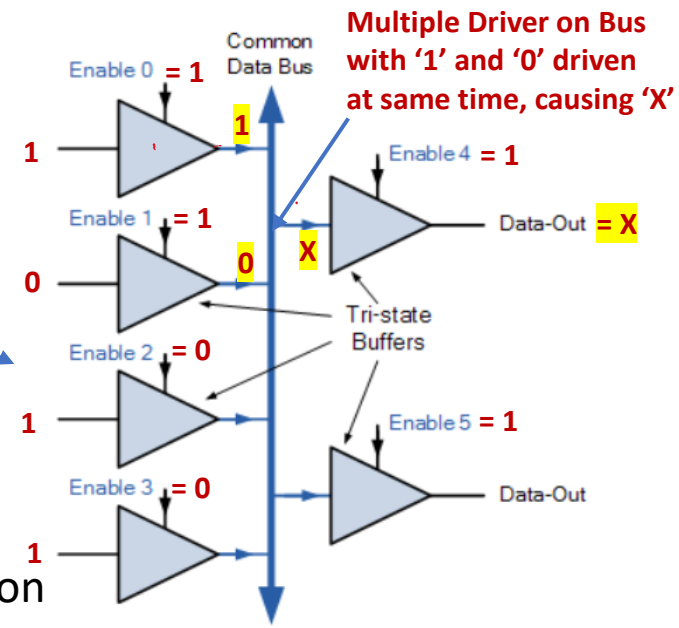
UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

SystemVerilog Data Value Set

- For hardware modeling, SystemVerilog uses a four-value set to represent actual value that can occur in silicon
- These 4-Value Sets are described in the table below :

Value	Abstract State
0	▪ Represents an abstract digital low state
1	▪ Represents an abstract digital high state
Z	▪ Represents abstract digital high-impedance state (tri-stated signal) ▪ In multi-driver circuit, a value of 0 or 1 will override a Z
X	▪ Represents unknown value or indicates uncertainity (**Simulation Only**) ▪ Indicates a wire or a register is uninitialized, or it indicates unintentionally same wire is driven simultaneously to different logic 0 or 1 by different drivers (also known as multi-driver circuit)





Note :

- Values of **'0', '1',** and **'Z'** are an abstraction of values that can exist in actual silicon
- The value of **'X'** is not an actual silicon value. It is only has meaning in Simulation.
- Simulators use **'X'** value to indicate uncertainty in how actual hardware would behave under specific conditions if there are registers/wires not initialized or there exists unintentional multiple driver on a wire or a register

SystemVerilog vs Verilog Data Types

Type	Mode	State	Size	Sign	SV/Verilog	Representation
reg	integer	4-state	user-defined	unsigned	Verilog	Equivalent **to var logic**
logic	integer	4-state	user-defined	unsigned	SystemVerilog	Infers a **var logic** except for input/inout ports wire logic is inferred
shortint	integer	2-state	16-bit	signed	SystemVerilog	Equivalent to **var bit**[15:0]
int	integer	2-state	32-bit	signed	SystemVerilog	Equivalent to **var bit**[31:0]. Synthesis compilers treats as 4-state **integer** type
longint	integer	2-state	64-bit	signed	SystemVerilog	Equivalent to **var logic**[63:0]
byte	integer	2-state	8-bit	signed	SystemVerilog	Equivalent to **var logic**[7:0]
bit	integer	2-state	user-defined	unsigned	SystemVerilog	default 1-bit size
integer	integer	4-state	32-bit	signed	Verilog	Equivalent to **var logic**[31:0]
real	floatingpoint	2-state	-	-	Verilog	Cannot be synthesized
shortreal	integer	2-state	-	-	SystemVerilog	Cannot be synthesized
realtime	floatingpoint	2-state	-	-	Verilog	Cannot be synthesized
time	Integer	4-state	64-bit	unsigned	Verilog	Cannot be synthesized

SystemVerilog Operators

[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

How to represent numbers in SystemVerilog

N'Bxx

```
8'b0000_0001
```

- **(N) Number of bits**
  - Expresses how many bits will be used to store the value
- **(B) Base**
  - Can be b (binary), h (hexadecimal), d (decimal), o (octal)
- **(xx) Number**
  - The value expressed in base, apart from numbers it can also have X and Z as values.
  - Underscore _ can be used to improve readability

x and z are not case-sensitive.

SystemVerilog Number System

Verilog	Stored Number	Verilog	Stored Number
4’b1001	1001	4’d5	0101
8’b1001	0000 1001	12’hFA3	1111 1001 0011
8’b0000_1001	0000 1001	8’o12	00 001 010
8’bxX0X1zZ1	XX0X 1ZZ1	4’h7	0111
‘b01	0000 .. 0001	12’h0	0000 0000 0000

Arithmetic Operators

Binary Operators	Description	Expressions
+ , -	Add , Subtract	c = a + b ; d = a – b;
* , /	Multiply , Divide	c = a * b; c = a/b;
%	Modulus	c = a % b;
**	Power operator	c = a ** b;

If a = 0 and b < 0 then for *c = a**b*, value of c is ***x***.

Unary Operator	Description	Expressions
+	Unary (sign) plus	c = +a; d = b/+a;
-	Unary (sign) minus	c = -a; d = b * -a;

Unary signed operators are used to assign a positive or negative sign to the operand.
By default, an operand is assumed to have '+' sign.

Arithmetic Operators : Binary Operator Examples

// Initial Values:

a = 4'b001;
b = 4'b0100;
d = 6; e = 4; f = 2;

// Assuming data type: logic [3:0] x, y, z, u, v ;

- assign x = a + b; // Adds *a* and *b* , evaluates to 4'b0101
- assign y = b – 3; // Subtracts 3 from *b*, evaluates to 4'b001
- assign z = d/e; // Divides *d* by *e*, evaluates to 4'b001, fractional part is truncated
- assign v = f**f; // Calculates *f* to the power of *f*, *v* evaluates to 4'b0100
- assign u = e**f; /* Calculates *e* to the power of *f*, *u* evaluates to 4'b0000. Since *u* is 4 bits wide, 4 least significant bits of the result of *e**f (i.e. 16)* get assigned to u. */

NOTE: If any operand has a value of x, then the result of any of these expressions is x, as if an operand's value is not fully known, the result cannot be known either.

Arithmetic Operators : Unary Operator Examples

- Unary arithmetic operators are used to assign a sign to an operand.
- For example,

```
a = 4; b = 2;
x = -a;          // x will hold value of -4
y = -a + -b;     // y will hold value of -6
```

- Negative numbers are represented in 2's complement
- For example,

```
logic [3:0]   nibble, out0;
logic [7:0]   out1;
initial begin
    nibble =  -1;          // i.e. 4'b1111
end
assign  out0  =  -nibble;  // i.e. 4'b0001
assign out1  =   nibble;   //i.e. 8'b0000_1111  GOTCHA!
```

- A sized negative number is not sign extended when assigned to a variable, unless signed variables are used

Arithmetic Operators

Modulus operator yields the remainder from division of two numbers
It works like the modulus operator in C
Modulus is synthesible

```
3  %  2;  //evaluates to  1
16 %  4;  //evaluates to  0
-7 %  2;  //evaluates to  -1, takes sign of first operand
 7 % -2;  //evaluates to  1, takes sign of first operand
```

Arithmetic Operators : Signed Arithmetic

- Use negative numbers only as type integer or real
- Avoid the use of <#bits>'<base><number> in expressions. These are converted to unsigned 2's complement numbers, which leads to unexpected results in simulation and synthesis.
- For example,

```
byte in;                  // signed 8-bit variables
int   out1, out2;         // signed 32-bit variables
initial begin
   in = -5;
   out1 = in + 1;          // OK!: -5 + 1 = -4 (literal 1 is signed)
   out2 = in + 1'b1;       // GOTCHA! : -5 + 1'b1 = 252 (literal 1'b1 is unsigned)
end
```

- An expression that has ANY unsigned operands will always result in an unsigned value

Arithmetic Operators

- The logic gate realization depends on several variables
  - coding style
  - synthesis tool used
  - synthesis constraints (more later on this)
- So, when we say "+", is it a...
  - ripple-carry adder
  - look-ahead-carry adder (how many bits of lookahead to be used?)
  - carry-save adder

When writing RTL code, keep in mind what will eventually be needed
Continually thinking about structure, timing, size, power

Logical Operators

Logical Operator	Description	Expressions
&&	Logical AND operation	c = a && b;
||	Logical OR operation	c = a || b;
!	Logical NOT – Unary	c = !b; d = !(a && b);

- Logical operators evaluate variables or expressions to a **1 bit** result:
  - 0, if the relation is false
  - 1, if the relation is true
  - x, if the *comparison is ambiguous*
- Logical operators consider their operands to be Boolean values
- Operands not equal to zero are equivalent to 1 or True. Operands with value zero are false or 0
- An ambiguous value may evaluate to unknown (1'bx)

  If a = 4'bxx00, then !a is x

  If a = 4'b1x00, then !a is 0 ← ***GOTCHA!***

  If at least a single bit's value is 1, then irrespective of other bit values, a is anything but 0. Hence, logical negation of a non-zero is ZERO.
- Expressions with &&, || are evaluated from left to right

Logical Operators : Examples

- **Logical operation on variables** : if a = 3 and b = 0, then,
  - (a && b) // evaluates to 0, since b is zero
  - (b || a) // evaluates to 1, since a is a non-zero value
  - !a // evaluates to 0, since NOT of a non-zero value is zero
  - !b // evaluates to 1, since NOT of a zero is one

- **Logical operation on unknown values** : if a = 2'b0x and b = 2'b10, then,
  - (a && b) // evaluates to x

- **Logical operation on expressions** : if a = 2 and b = 3, then,
  - (a == 2) && (b == 3) // evaluates to 1, since both the comparison expressions are true
  - (a == 2) && !b // evaluates to 0, since !b is 0 or false

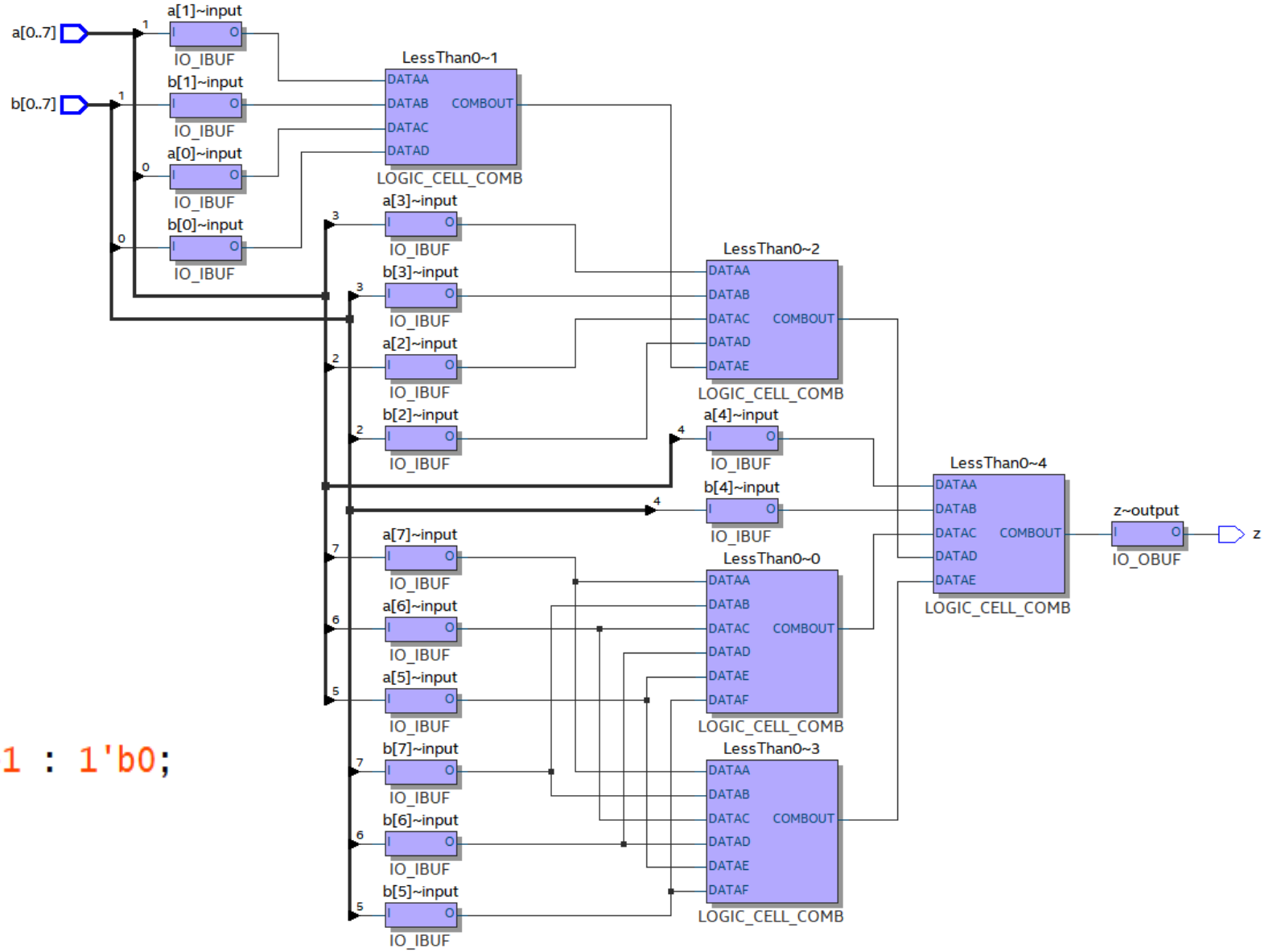
Relational Operators

Relational Operator	Description	Expressions
>	Greater than	c = a > b;
<	Less than	c = a < b;
>=	Greater than or equal to	c = a>= b;
<=	Less than or equal to	c = a<=b;

- Relational or comparison are binary operators that:
  - return a logical 1 if expression is true
  - return a logical 0 if expression is false
  - x, if *any operand has an unknown (x) bit value*
- For example, if a = 4, b = 3, x = 4'b1010, y = 4'b1101, z = 4'b1xxx, then:
  - (a <= b) // evaluates to logical 0, since the expression is false
  - (a > b) // evaluates to logical 1, since the expression holds true
  - y >=x // evaluates to logical 1
  - y < z // evaluates to ***x***

Relational Operators

These operators are expensive and slow at gate-level!
Let's look at an example.




```
module less_than (
   input [7:0] a,b,
   output      z
   );
   assign z = (a < b) ? 1'b1 : 1'b0;

endmodule: less_than
```

Bitwise Operators

Relational Operator	Description	Expressions
&	Bitwise AND	c = a & 3;
|	Bitwise OR	c = a | b;
~	Bitwise NOT	c = ~ a;
^	Bitwise XOR	c = a ^ b;
&~	Bitwise NAND	c = a &~ 5;
|~	Bitwise NOR	c = 2 |~ 4;
^~	Bitwise XNOR	c = a ^~ b;

- Bitwise operators operate bit by bit
- Results in x, 1 or 0 bit values
- Mismatched length operands are zero extended
- x and z are treated the same

Bitwise Operators : Examples

- Examples:

  if a = 3'b101, b= 3'b011, c = 3'b1x0, then:

  - y = a & b; // 3'b001
  - y = a | c ; // 3'b1x1, since 0 | x = x
  - y = a & c; // 3'b100, since 0 & x = 0
  - y = a ^ c; // 3'b0x1, since 0 ^ x = x; also, 1 ^ x = x
  - y = ~b; // 3'b100
  - y = ~c; // 3'b0x1, since ~x = x (unknown/ambiguous)

- Difference between **logical** AND, OR, NOT and **bitwise** AND, OR NOT :

  if a = 3'b100, b= 3'b010, c = 3'b1x0

  - a & b // evaluates to 0
  - a && b // evaluates to 1
  - ~a // evaluates to 3'b011 or 3
  - !a // evaluates to 0
  - ~c // evaluates to 3'b0x1
  - !c // evaluates to 0

WHY?

a and b are both non-zero values. If we do a logical AND of to non-zero (TRUE) values, then result is 1(True).

Reduction Operators

Relational Operator	Description	Expressions
&	AND	c = &a;
|	OR	c = |b;
^	XOR	c = ^b;
&~	NAND	c = &~ a;
|~	NOR	c = |~ b;
^~	XNOR	c = ^~ b;

- Reduction operators are unary operators, in the sense that they work on 1 vector operand
- Performs bit-wise operation on all bits of the operand
- Works from right to left, bit by bit
- Returns a 1-bit result

- For example, if x = 4'b1010

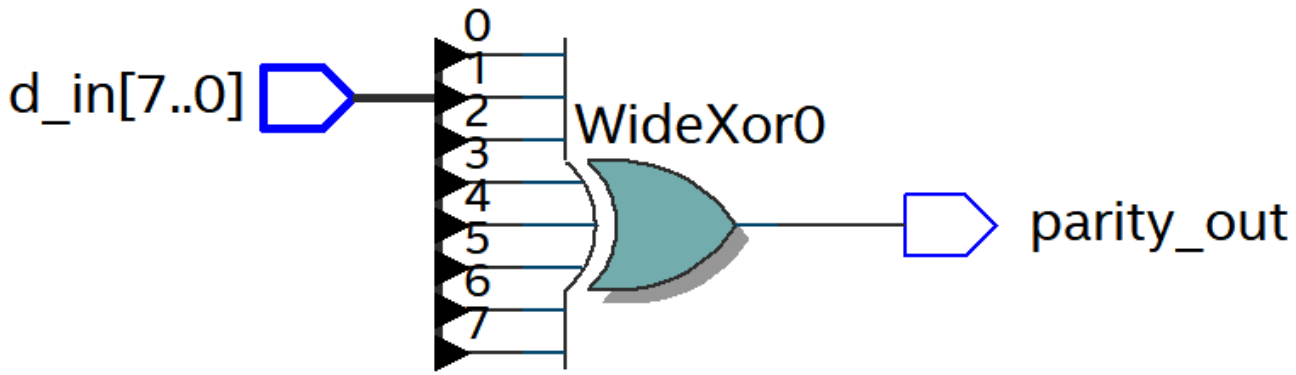
```
&x     // equivalent to 1 & 0 & 1 & 0; results in 1'b0
|x     // equivalent to 1 | 0 | 1 | 0; results in 1'b1
^x     // equivalent to  1 ^ 0 ^ 1 ^ 0; results in 1'b0
```

Reduction Operators

A good example of usage of *XOR reduction operator* is parity generation.

```
module parity8 (
    input [7:0] d_in,
    output parity_out
);

    assign parity_out = ^d_in;
endmodule: parity8
```



Equality Operators

Equality Operator	Description	Expressions
==	Logical equality, result may be unknown	(a == b) ; c = (a==b)
!=	Logical inequality, result may be unknown	(a != b) ; c = (a != b)
===	Case equality, including x and z	a === b
!==	Case inequality, including x and z	a ==! b;

- Equality operators return a logical 1 if expression is true or 0 if expression is false
- Operands are compared bit by bit
- If operands are of unequal length, zero filling is done.
- For logical equality/inequality operators, if any operand has an x or z bit, result is x (unknown)
- Case equality/inequality operators match x and z bits of the operands as well.
- For example, a = 4'b1x01, b = 4'b1x01, m = 4'b1010, n = 4'b1101, o = 4'b1xxx, then:
  - (a == b) // evaluates to **x**, since both operands have x
  - (a === b) // evaluates to logical **1**, since x bits match as well
  - n != m // evaluates to logical 1, since n and m aren't equal
  - n !== o // evaluates to logical 1
- **NOTE**: === and !== synthesize to == and != in design

Shift Operators

Relational Operator	Description	Expressions
>>	Logical shift right	c = a >> 3;
<<	Logical shift left	c = b << 3;
>>>	Arithmetic shift right	c = a >>> 1;
<<<	Arithmetic shift left	c = a <<< 1;

- Shifts the vector to a side by the given number of positions
- Logical shift operators fill the vacant bit positions with zeros
- Arithmetic shift left operator fills the vacant bit positions that pop up on the left with zeros
- Arithmetic shift right operator fills the vacant bit positions on the right with 0, if result variable is unsigned. If result variable is signed, then vacant bits are filled with the MSBit (i.e. the sign bit)

Shift Operators : Examples

- **Logical shift operators**: if a = 4'b1100, b=4'b01xx, then,
  - c = a << 2; // c = 4'b0000
  - c = a >> 3; // c = 4'b0001
  - c = b >> 1; // c = 4'b001x

- **Arithmetic shift operators**:

```
logic signed [3:0]  x, y, z;
logic [3:0]         u;
initial begin
   x = 4'b1010;
end
y = (x >>> 2);      // y will have value 4'b1110
u = (x >>> 2);      // u will have value 4'b0010   ← GOTCHA!
z = (z <<< 2);      // z will have value 4'b1000
```

Concatenation and Replication Operators

Concatenation Operator {,}

- Provides a way to append busses or wires to make busses
- The operands must be sized
- Expressed as operands in braces separated by commas

```
//let a = 1'b1, b = 2'b00, c = 2'b10, d = 3'b110
y = {b, c} // y is then 4'b0010
y = {a, b, c, d, 3'b001} // y is then 11'b10010110001
y = {a, b[0], c[1]} // y is then 3'b101
```

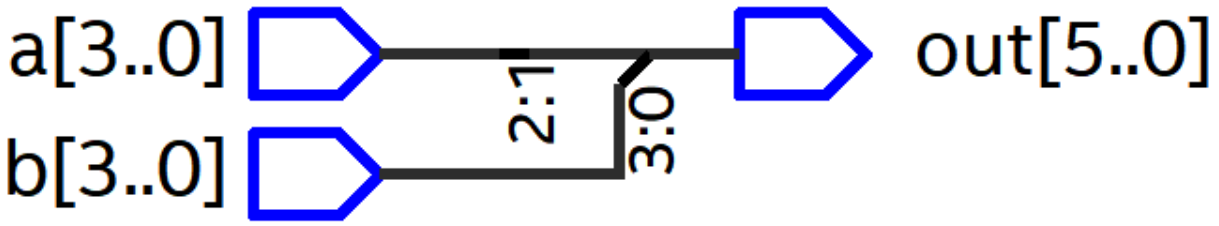
Replication Operator { { } }

- Repetitive concatenation of the same number
- Operands are number of repetitions, and the bus or wire

```
//let a = 1'b1, b = 2'b00, c = 2'b10, d = 3'b110
y = { 4{a} }          // y = 4'b1111
y = { 4{a}, 2{b} }    // y = 8'b11110000
y = { 4{a}, 2{b}, c } // y = 8'b1111000010
```

Concatenation Operator : Example

```
module concat (
    input [3:0] a,
    input [3:0] b,
    output [5:0] out
);
    assign out = {a[2:1], b};
endmodule : concat
```



Conditional Operator

- Widely used operator in RTL modeling. Also known as **Ternary** operator !!
- Similar to **if-else** statement
- Conditional operator often behaves like a hardware multiplexer
- Can be used in continuous **assign** statement and also within **always** procedural blocks

conditional operator	Syntax	Example Usage
? :	conditional expression **?** true expression **:** false expression	assign out= p **?** a **:** b

If **"p"** is true then assign value of **"a"** to **"out"** otherwise assign value of **"b"** to **"out"**

- Conditional expression listed before **"?"** is evaluated first as true or false
  - If evaluation result is true, then **true expression** is evaluated
  - If evaluation result is false, then **false expression** is evaluated
  - If evaluation result is unknown "x", then conditional operator performs bit by bit comparison of the two possible return values
    - If corresponding bits are both 0, a 0 is returned for that bit position
    - If corresponding bits are both 1, a 1 is returned for that bit position
    - If corresponding bits differ or if either has "x" or "z" value, an "x" is return for that bit position

Conditional Operator

- **Example :**

```
logic sel, mode;
logic [3:0] a, b, mux_out;
assign mux_out = (sel & mode) ? a : b;
```

Scenario	Value of "sel"	Value of "mode"	Value of "a"	Value of "b"	Result of conditional expr (sel & mode)	Final value assigned to "mux_out"
1	1'b1	1'b1	4'b0101	4'b1110	True (1)	4'b0101
2	1'b0	1'b1	4'b0101	4'b1110	False (0)	4'b1110
3	1'b1	1'bx	4'b0101	4'b1110	Unknown (x)	4'bx1xx
4	1'b1	1'bx	4'b011x	4'b0z10	Unknown (x)	4'b0x1x

Note : bitwise and ("**&**") will return value "x" if one of the operand is "x" and another operand is "1"

Precedence of Operators in SystemVerilog

Highest Precedence

() [] :: .
+ - ! ~ & ~& | ~| ^ ~^ ^~ ++ -- unary
**
* / %
+ - (binary)
<< >> <<< >>>
< <= > >= inside dist
== != === !== =?= !?=
& (binary)
^ ~^ ^~ (binary)
| (binary)
&&
||
? : (conditional operator)
>
= += -= *= /= %= &= ^= |= <<= >>= <<<= >>>= := :/ <=
{} {{}}

Lowest Precedence