

Operators

Operator types

- The operators can be :

- arithmetic
- logical
- relational
- equality
- bit wise
- reduction
- shift
- concatenation
- replication
- conditional

Arithmetic operators

Expressions constitute operators and operands.

operation	symbol	operand
Multiply	*	binary
Divide	/	binary
Add	+	binary
Subtract	-	binary
Modulus	%	binary

Arithmetic operator examples

Eg: $a * b$ // multiply a and b

a / b // divide a by b

$a + b$ // add a and b

$a - b$ // subtract b from a

$a \% b$ // modulus of a by b

$a = 3'b011$ $b = 3'b010$ $d = 4$ $e = 3$

$c = a * b$ // $c = 3'b110$

$c = a / b$ // $c = 1$

$c = a + b$ // $c = 3'b101$

$c = a - b$ // $c = 3'b001$

$c = d / e$ // $c = 1$

Cont'd..

`13 % 4 //` evaluates to 1.

`-9 % 2 //` evaluates to -1, takes sign of the first
//operand

- In arithmetic operations, if any operand bit has a value x , then the result of the entire expression is x .
- The size of the result is determined by the size of the largest operand.

Logical operators

Logical operator evaluates always to a **one bit value** either true(1) or false (0) or x (unambiguous) . If any operand bit is either x or z it is equivalent to x

operation	symbol	operand
logical and	&&	binary
logical or		binary
logical not	!	binary

Logical operator examples

`a1 = 1'b0; // 0 is false;`

`a2 = 1'b1; // 1 is true`

`a1 && a2` is 0 (false)

`a1 || a2` is 1 (true)

`!a2` is 0 (false)

- For vector operands, a non-zero vector is treated as logical 1.

Cont'd..

Example:

a=10 b=00

a && b // evaluates to 0 (1 && 0)

a=2'b1x b=2'b11

a || b // is unknown, evaluates to x.

Relational operators

- Relational operations return logical 0 or 1. If there is any x or z bit in operand then it will return x.

Operation	Symbol	Operand
greater	>	Binary
less than	<	Binary
Greater than or equal to	>=	Binary
Less than or equal to	<=	Binary

Relational operator examples

$a = 5 \quad b = 6 \quad c = 2 \quad !x$

$a > b$ // evaluates to 0

$a \leq b$ // evaluates to 1

$b \geq c$ // evaluates to x

Equality operators

Equality operators are the following

Operation	Symbol	Operand
logical equality	==	binary
logical inequality	!=	binary
case equality	===	binary
case inequality	!==	binary

Equality operators

- Equality operator can return 1 or 0.
- Logical equality operator (`==` `!=`) will return `x` if any of the operand bit has `x`.
- **Case equality** operator compares both operand bit by bit including `x` and `z` bit. If it matches then returns 1 or else it returns 0. It doesn't return `x`.

Cont'd..

```
a=3; b=5; c=3'b100; d=3'b101; e=4'b1xxx;
```

```
f=4'b1xxx;
```

```
g=3'b1xxz
```

```
a != b // evaluates to 1.
```

```
e === f // evaluates to 1.
```

```
f === g // evaluates to 0. d == e // evaluates to x
```

Bitwise operators

Bitwise operations are performed on each bit of the operand

Operation	Symbol	Operand
Bitwise and	&	Binary
Bitwise or		Binary
Bitwise negation	~	Unary
Bitwise xor	^	Binary
Bitwise xnor	~^ or ^~	Binary

Bitwise operators Example

```
module fulladd_1(sum,carry,a,b,c);
```

```
input a,b,c;
```

```
output sum,carry;
```

```
wire sum,carry;
```

```
assign sum = (a^b)^c;
```

```
assign carry = (a&b) | (b&c) | (c&a);
```

```
endmodule
```

Bitwise operator examples

```
a = 3'b111;  b = 3'b101;  d = 3'b1x1;
```

```
c = ~a;      // c = 3'b000
```

```
c = a & b;    // c = 3'b101
```

```
c = a | b;    // c = 3'b111
```

```
c = a ^ b;    // c = 3'b010
```

```
c = a | d;    // c = 3'b1x1
```


Reduction operators

Reduction operators are unary operators

Operation	Symbol	Operand
reduction and	&	unary
reduction nand	~&	unary
reduction or		unary
reduction nor	~	unary
reduction xor	^	unary
reduction xnor	~^ or ~^	unary

Reduction operator examples

$x = 4'b01100$

$c = \&x \quad // \quad c = 0 \& 1 \& 1 \& 0 \& 0 \quad c = 0$

$c = |x \quad // \quad c = 0|1|1|0|0 \quad c = 1$

$c = ^x \quad // \quad c = 0^1^1^0^0 \quad c = 0$

Shift operators

Shift operator can be shift left or shift right

Operation	Symbol	Operand
shift right	>>	unary
shift left	<<	unary

Example:

`a = 4'b1011;`

`y = a >> 2; // y = 4'b0010, 0's filled in MSB`

`y = a << 2; // y = 4'b1100, 0's filled in LSB`

Concatenation operators

- Concatenation operator is used to append multiple operands.
- The operand must be **sized**.

```
a=3'b101; b=3'b111;
```

```
y = {a,b};           // y = 6'b101111
```

```
y = {a,b,3'b010};    // y =101111010
```

Replication operators

Replication operator is used to concatenate same number.

$a = 3'b101$ $b = 2'b10$

$y = \{2\{a\}\};$ // result of y is $6'b101101$

$y = \{2\{a\}, 2\{b\}\};$ // result of y is $10'b1011011010$

$y = \{2\{a\}, 2'b10\};$ // result of y is $6'b10110110$

Conditional operators

Conditional operator ? :

format:

`conditional_expr ? true_expr : false_expr;`

eg:

assign `out = control ? I1 : I2;`

control	out
1	I1
0	I2

Conditional Operator examples

```
module mux_con(out,s0,s1,i);
```

```
input s0,s1;
```

```
input [3:0]i;
```

```
output out;
```

```
wire out;
```


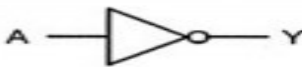






```
assign out = s1 ? ( s0 ? i[3]:i[2]) : (s0 ? i[1]:i[0]) ;
```

```
endmodule
```

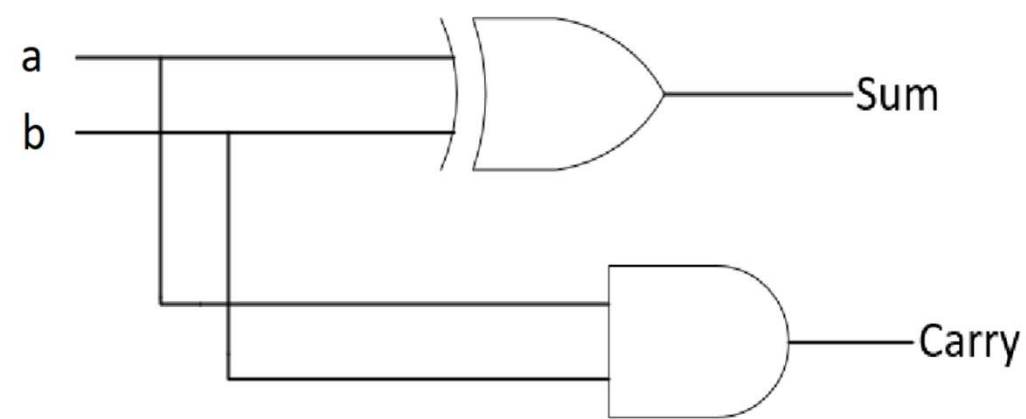
Operator precedence

+ - ~ !	unary
* / %	arithmetic
+ -	binary
<< >>	shift
< <= > >=	relational
& ~&	reduction and , nand
^ ~^	reduction exor, exnor
~	reduction or , nor
&&	logical and
	logical or
? :	conditional

Logic Gates

Logic function	Logic symbol	Truth table	Boolean expression															
Buffer		<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	Y	0	0	1	1	$Y = A$									
A	Y																	
0	0																	
1	1																	
Inverter (NOT gate)		<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0	$Y = \overline{A}$									
A	Y																	
0	1																	
1	0																	
2-input AND gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	$Y = A \cdot B$
A	B	Y																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
2-input NAND gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0	$Y = \overline{A \cdot B}$
A	B	Y																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
2-input OR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1	$Y = A + B$
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
2-input NOR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0	$Y = \overline{A + B}$
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
2-input EX-OR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0	$Y = A \oplus B$
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
2-input EX-NOR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1	$Y = \overline{A \oplus B}$
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Example: Half-adder implementation



a	b	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Adder Verilog Code: Gate level Modelling

```
module half_adder(sum, carry,  
a, b);  
input a, b;  
output sum, carry;  
xor sum1(sum, a, b);  
and carry1(carry, a, b);  
endmodule
```

- “module” is the basic building block in Verilog.
- In Verilog, a module is declared by the keyword module.
- A corresponding keyword endmodule must appear at the end of the module definition.
- Each module must have a module_name, which is the identifier for the module, and a port list, which describes the input and output terminals of the module

The module name

- The module name, formally called an identifier should best describe what the system is doing. Each identifier in Verilog, including module names must follow these rules:
- It can be composed of letters, digits, dollar sign (\$), and underscore characters (_) only.
- It must start with a letter or underscore.
- No spaces are allowed inside an identifier.
- Upper and lower case characters are distinguished (Verilog is case sensitive)
- Reserved keywords cannot be used as identifiers.

Testbench for half-adder

```
module half_adder_testbench;  
reg a,b;  
wire sum,cout;  
half_adder h1(sum, carry, a, b);
```

```
initial begin  
a = 1'b0;  
b = 1'b0;  
#20;
```

```
a = 1'b0;  
b = 1'b1;  
#20;
```

NOTE: Testbench is same for all types of modelling

```
a = 1'b1;  
b = 1'b0;  
#20;
```

```
a = 1'b1;  
b = 1'b1;  
#20;
```

```
$finish;  
end  
endmodule
```

Verilog data-type **reg** can be used to model hardware registers since it can hold values between assignments.

A **wire** represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block.

```
1  initial  
2      [single statement]  
3  
4  initial begin  
5      [multiple statements]  
6  end
```

Half Adder Verilog Code: Dataflow Modelling

```
module half_adder(sum, carry,  
a, b);  
input a,b;  
output sum,carry; // sum and  
carry  
  
assign sum = a^b;  
assign carry = a&b ;  
  
endmodule
```

- assign is used for driving wire/net type declarations.
- Since wires change values according to the value driving them, whenever the operands on the RHS changes, the value is evaluated and assigned to LHS(thereby simulating a wire).

Half Adder Verilog Code: Behavioral Modelling

```
module half_adder(sum, carry, a, b);  
input a,b;  
output reg sum,carry; // sum and carry  
always @(*)  
begin  
case ({a,b})  
2'b00: begin sum = 0;carry=0;end  
2'b01: begin sum = 1;carry=0;end  
2'b10: begin sum = 1;carry=0; end  
2'b11: begin sum = 0;carry=1; end  
default : begin sum = 0;carry=0; end  
endcase  
end  
endmodule
```

- always is a procedural block is used for modelling registers and combinational logic. always block contains sensitivity list, that is, the event list, upon which the logic inside the block must be evaluated.
- always(@ posedge clk) triggers the the logic inside the block at every positive edge.