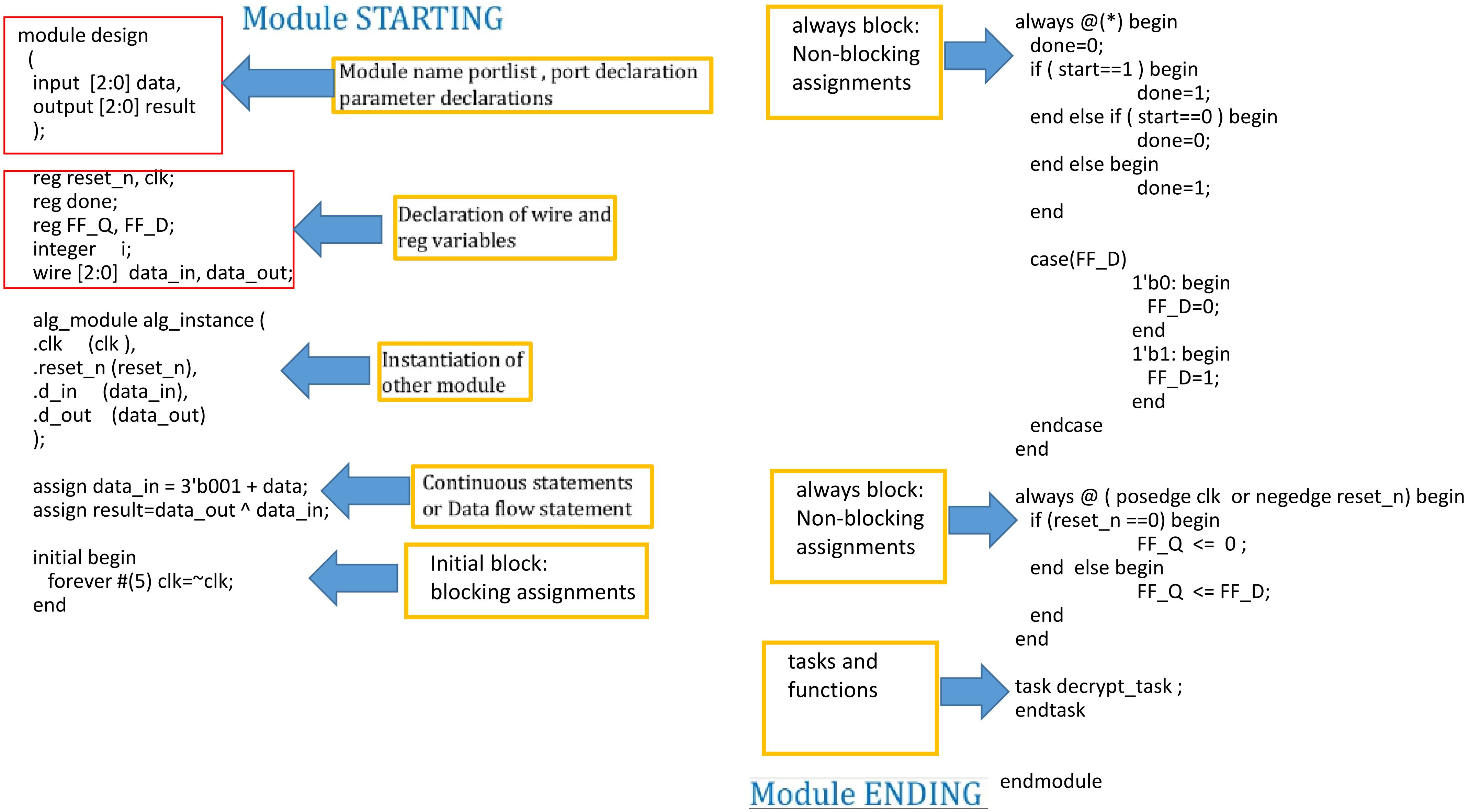
# Verilog :

## Expressions, Operators

# Verilog Module Structure



## Module STARTING

```
module design
 (
  input  [2:0] data,
  output [2:0] result
  );
```

Module name portlist , port declaration parameter declarations

```
  reg reset_n, clk;
  reg done;
  reg FF_Q, FF_D;
  integer     i;
  wire [2:0]  data_in, data_out;
```

Declaration of wire and reg variables

```
  alg_module alg_instance (
  .clk     (clk ),
  .reset_n (reset_n),
  .d_in     (data_in),
  .d_out    (data_out)
  );
```

Instantiation of other module

```
  assign data_in = 3'b001 + data;
  assign result=data_out ^ data_in;
```

Continuous statements or Data flow statement

```
  initial begin
    forever #(5) clk=~clk;
  end
```

Initial block: blocking assignments

always block: Non-blocking assignments

```
always @(*) begin
  done=0;
  if ( start==1 ) begin
                done=1;
  end else if ( start==0 ) begin
                done=0;
  end else begin
                done=1;
  end

  case(FF_D)
               1'b0: begin
                 FF_D=0;
               end
               1'b1: begin
                 FF_D=1;
               end
  endcase
end
```

always block: Non-blocking assignments

```
always @ ( posedge clk  or negedge reset_n) begin
  if (reset_n ==0) begin
                FF_Q  <=  0 ;
  end  else begin
                FF_Q  <= FF_D;
  end
end
```

tasks and functions

```
task decrypt_task ;
endtask
```

## Module ENDING

```
endmodule
```

# Expressions

- Two types:

    1)Unary expressions: operator operand (e.g. –a , +b)

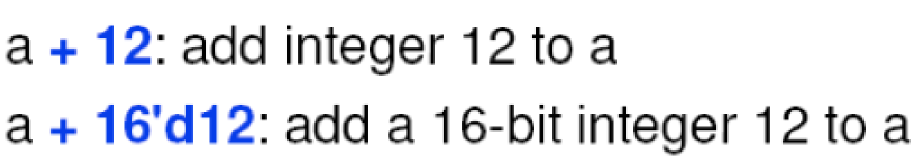
    2)Binary expressions: operand operator operand (e.g. a+b)

- The operands may be a net (or wire)
- Logic expression returns: 1 (true), 0 (false), X (unknown)

# Arithmetic Operators

- There are two types of operators: **Binary and Unary**

- Binary operators:
- add(+), subtract(-), multiply(*), divide(/), power(**), modulus(%)
- Specif



a + 12: add integer 12 to a

a + 16'd12: add a 16-bit integer 12 to a

- If any operand bit has a value ”x”, the result of the expression is all ”x”.

# Arithmetic Operators

//suppose that: **a = 4’b0011**;

// **b = 4’b0100**;

**// d = 6; e = 4; f = 2**;

```
a + b         //add a and b; evaluates to 4’b0111
b - a                        //subtract a from b; evaluates to 4’b0001
a * b                         //multiply a and b; evaluates to 4’b1100
d / e         //divide d by e, evaluates to 4’b0001. Truncates fractional part
b % a        // modulus of b/a, evaluates to 4’b0001.
e ** f        //raises e to the power f, evaluates to 4’b1111

//divide, modulo and power operators are most likely not synthesizable.
```

# Arithmetic Operators

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

```
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

- Unary operators
- Operators ”+” and ”-” can act as unary operators
- They indicate the sign of an operand

```
        i.e., -4      // negative four
              +5          // positive five
```

!!! Negative numbers are represented as 2’s compliment numbers !!!

!!! Use negative numbers only as type integer or real !!!

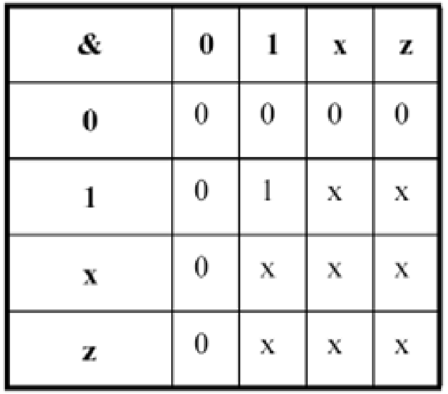
!!! Avoid the use of <sss>’<base><number >in expressions !!!

!!! These are converted to unsigned 2’s compliment numbers !!!

# Bitwise Operators

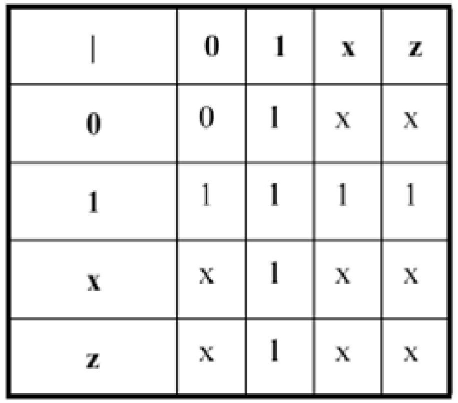
- negation (~), and(&), or(|), xor(^), xnor(^- , -^)
- Perform bit-by-bit operation on two operands (except ~)
- Mismatched length operands are zero extended
- x and z treated the same

AND



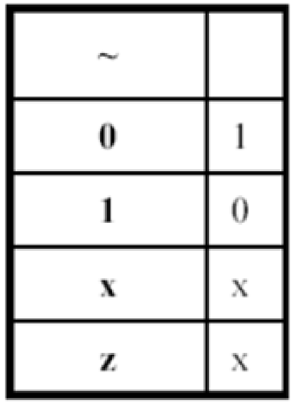
| & | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

OR



| \| | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 1 | x | x |
| 1 | 1 | 1 | 1 | 1 |
| x | x | 1 | x | x |
| z | x | 1 | x | x |

NOT



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

# Bitwise Operators

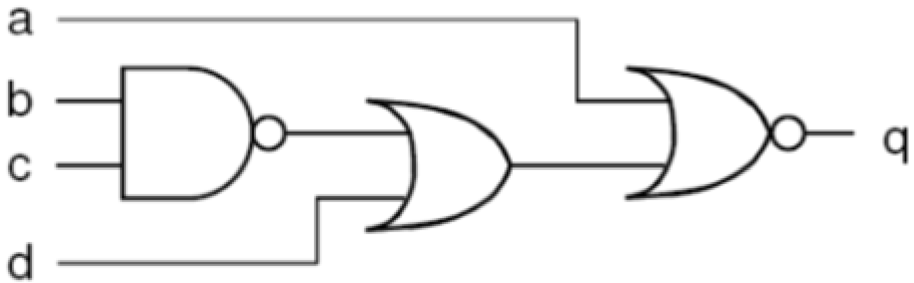
- Logical operators result in logical 1, 0 or x
- Bitwise operators results in a bit-by-bit value

```
//let x = 4’b1010, y = 4’b0000
          x | y                  //bitwise OR, result is 4’b1010
          x || y                  //logical OR, result is 1
```

# Bitwise Operators

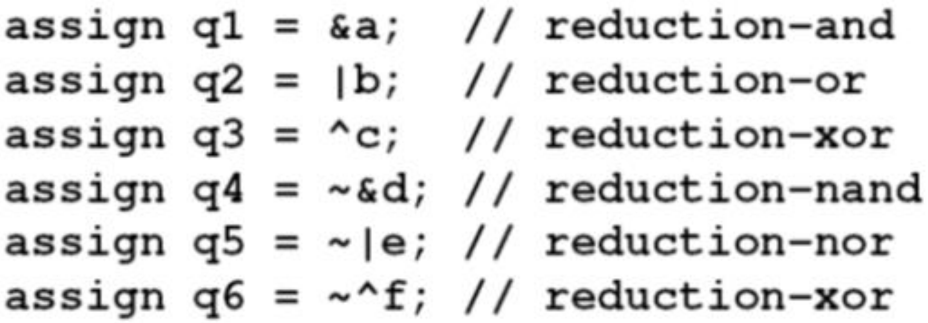
- Bitwise operators give bit-by-bit results

```
module cct(q,a,b,c,d);
input a,b,c,d;
output q;
assign q =~(a | ( d | ~ ( b & c ) ) );
endmodule
```



# Reduction Operators

- and(&), nand(~&), or(|), nor(~|) xor(^), xnor(^~,~^)
- operates on only one operand



```
assign q1 = &a;   // reduction-and
assign q2 = |b;   // reduction-or
assign q3 = ^c;   // reduction-xor
assign q4 = ~&d;  // reduction-nand
assign q5 = ~|e;  // reduction-nor
assign q6 = ~^f;  // reduction-xor
```

- Performs a bitwise operation on all bits of the operand
- Returns a 1-bit result
- Works from right to left, bit by bit

```
//let 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
```

# Logical Operators

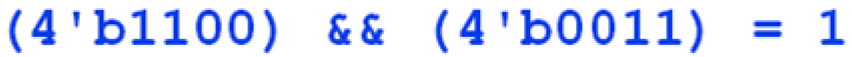
- logical-and(&&)     //binary operator
- logical-or(||)      //binary operator
- logical-not(!)              //unary operator

```
//suppose that: a = 3 and b = 0, then...
(a && b)            //evaluates to zero
(b || a)             //evaluates to one
(!a)                        //evaluates to 0
(!b)                         //evaluates to 1

//with unknowns: a = 2’b0x; b = 2’b10;
(a && b) // evaluates to x

//with expressions...
(a == 2) && (b == 3) //evaluates to 1 only if both comparisons are true
```

# Logical Operators

- Logical operators evaluate to a 1 bit value
- 0 (false), 1 (true), or x (ambiguous)
- Operands not equal to zero are equivalent to one
- Logical operators take variables or expressions as operators
  - 
    ```
    (4'b1100) && (4'b0011) = 1
    ```

Let a = 4'b1100; b = 4'b0000;

!a // 0 - false

!b // 1 - true

a && b // 0 - false

a || b // 1 - true

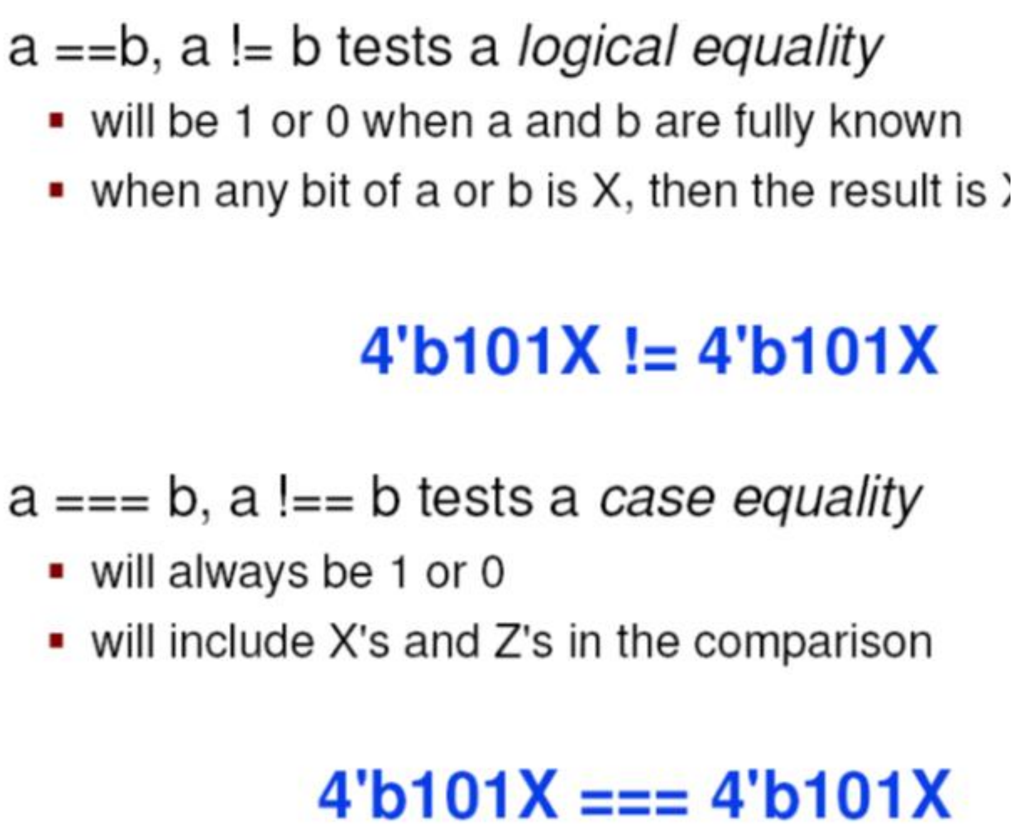
# Relational Operators

- greater-than (>)
- less-than (<)
- greater-than-or-equal-to (>=)
- less-than-or-equal-to (<=)
- Relational operators return logical 1 if expression is true, 0 if false or x if one of the operand is x (unknown).

```
//let a = 4, b = 3, and...
//x = 4'b1010, y = 4'b1101, z = 4'b1xxx

a <= b              //evaluates to logical zero (false)
a > b               //evaluates to logical one (true)
y >= x              //evaluates to logical 1
y < z               //evaluates to x
```

# Equality Operators



a ==b, a != b tests a *logical equality*

- will be 1 or 0 when a and b are fully known
- when any bit of a or b is X, then the result is

**4'b101X != 4'b101X**

a === b, a !== b tests a *case equality*

- will always be 1 or 0
- will include X's and Z's in the comparison

**4'b101X === 4'b101X**

# Equality Operators

- logical equality (== )
- logical inequality (!= )
- logical case equality (===)
- logical case inequality (!==)
- Equality operators return logical 1 if expression is true, else 0
- Operands are compared bit by bit
- Zero filling is done if operands are of unequal length (Warning!)
- Logical case inequality allows for checking of x and z values

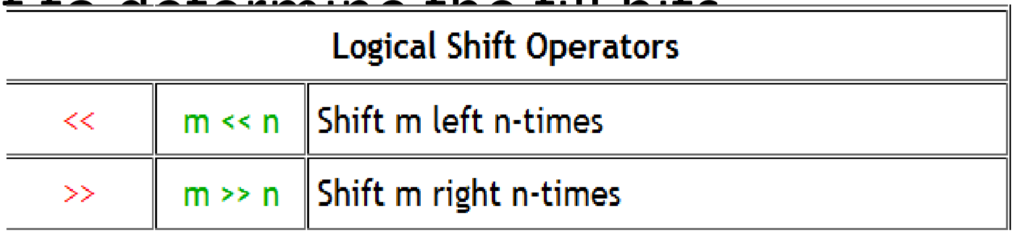
```
//let a = 4'b1100, b = 4'b101x
a == 4'b1100 // true - 1
a != 4'b1100 // false – 0

b == 4'b101x // unknown - x
b != 4'b101x // unknown - x
b === 4'b101x // true - 1
b !== 4'b101x // false - 0
```

# Shift Operators

- right shift (>>)
- left shift (<<)
- arithmetic right shift (>>>)
- arithmetic left shift (<<<)
- Shift operator shifts a vector operand left or right by a specified number of bits, filling vacant bit positions with zeros.
- Shifts do not wrap around.
- Arithmetic shift uses context to determine the fill bits.

```
// let x = 4'b1100
y = x >> 1; // y is 4'b0110
y = x << 1; // y is 4'b1000
y = x << 2; // y is 4'b0000
```



| Logical Shift Operators | | |
|---|---|---|
| << | m << n | Shift m left n-times |
| >> | m >> n | Shift m right n-times |

# Shift Operators

- arithmetic right shift (>>>)
- Shift right specified number of bits, fill with value of sign bit if expression is signed, otherwise fill with zero.
- arithmetic left shift (<<<)
- Shift left specified number of bits, filling with zero.

```
// let  a = 5'b10100;
              b = a <<< 2;       //b is 5'b10000
              c = a >>> 2;       //c is 5'b11101
```

# Concatenation Operator {,}

- Provides a way to append busses or wires to make busses
- I The operands must be sized
- I 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
```

# Conditional Operator ?:

- Operates like the C statement
- conditional expression ? true expression : false expression ;
- Result=(a>=b) ? 3 : 5
- The conditional expression is first evaluated
  - If the result is true, true expression is evaluated
  - If the result is false, false expression is evaluated
  - If the result is x:
    - both true and false expressions are evaluated,...
    - their results compared bit by bit,...
    - returns a value of x if bits differ, OR...
    - the value of the bits if they are the same.

# Conditional Operator ?:

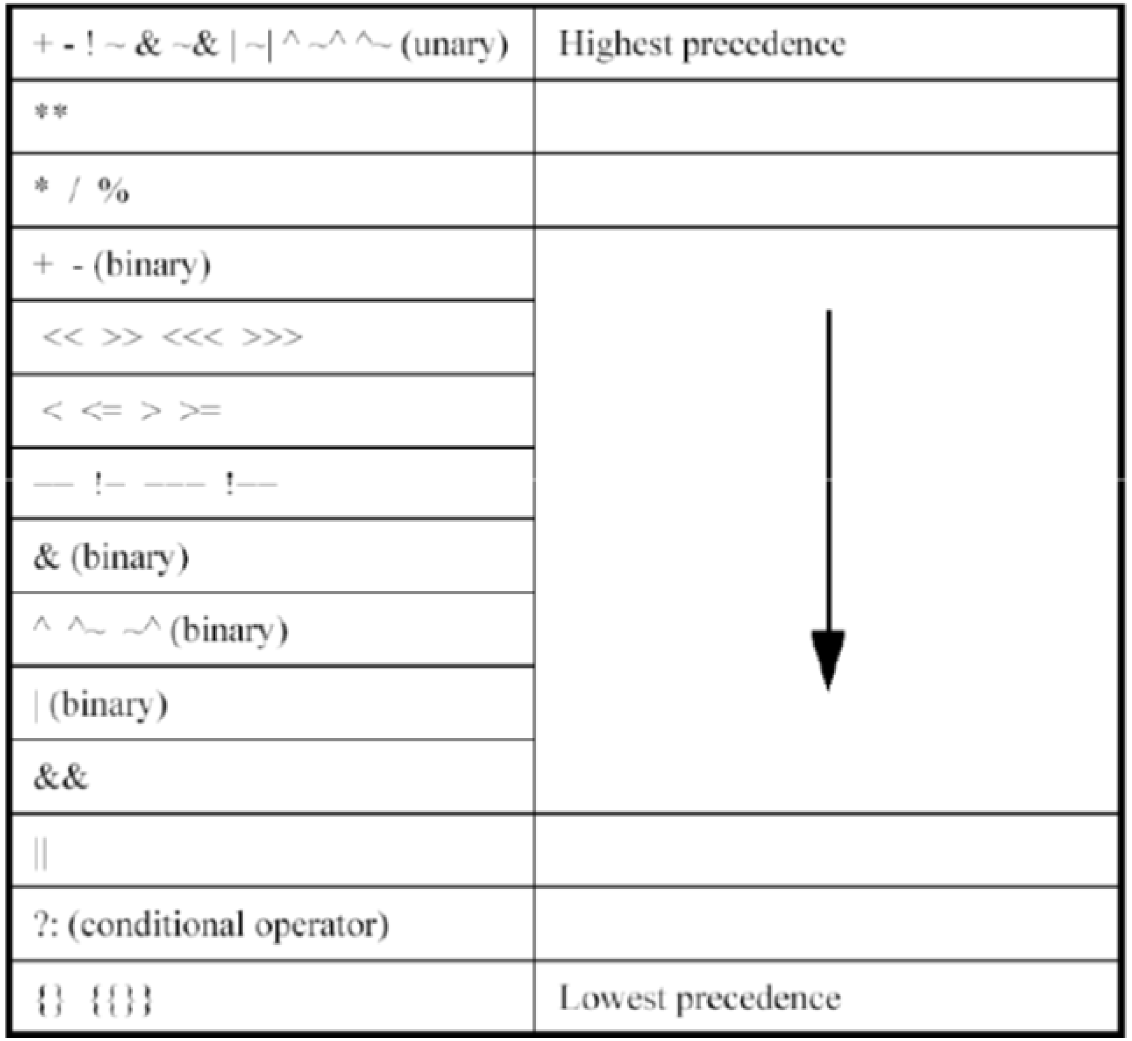
```
assign q = c ? a : b;
```

c is or-reduced.

- If result is 1, then q = a.
- If result is 0, then q = b.
- If result is x, then q is combined, bit by bit, from a and b.

| | |
|---|---|
| a | 1110011 |
| b | 0000001 |
| c | xxx00x1 |

# Operator precedence



| | |
|---|---|
| + - ! ~ & ~& \| ~\| ^ ~^ ^~ (unary) | Highest precedence |
| ** | |
| * / % | |
| + - (binary) | |
| << >> <<< >>> | |
| < <= > >= | |
| == != === !== | |
| & (binary) | |
| ^ ^~ ~^ (binary) | |
| \| (binary) | |
| && | |
| \|\| | |
| ?: (conditional operator) | |
| {} {{}} | Lowest precedence |