# Verilog

## levels of abstraction

- switch level
- Gate level
- data flow level
- Behavioral level

Verilog is case sensitive

~~syntax~~: * switch level

syntax: -> mos_name instance_name (output, data, control)

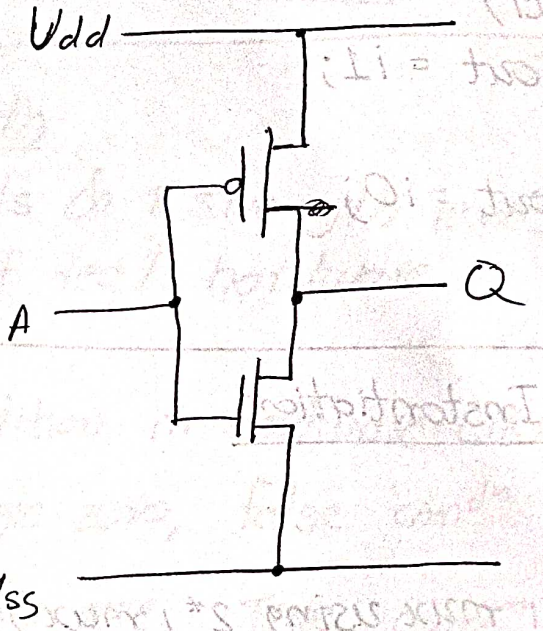
example

```
Module inverter (Q, A);
  input A;
  output Q;
  supply1 vdd;
  supply0 vss;

  Pmos  P(Q, vdd, A);

  nmos  n (Q, Vss, A);

endmodule
```

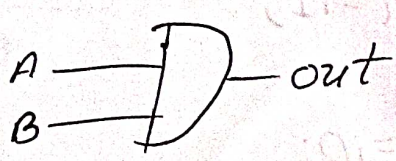



* Gate level

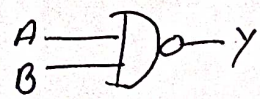
Syntax: Primitive_name instance_name (output, inputs)

Example

1. and G1(out, A, B);




2. nand G2 (Y, A, B);



## * Data Flow level

Example:

```
assign z = x & y;
assign p = q | r;
assign z = ~y;
```

## * Behavioral level

example (2*1 mux)

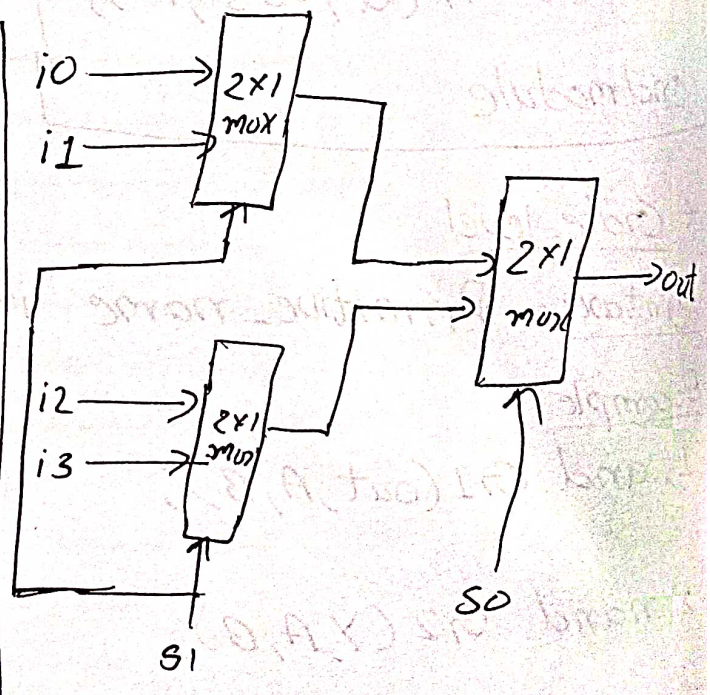
```
always @ (i0, i1, sel)
begin
    if (sel)
        out = i1;
    else
        out = i0;
end
```

---

## Module Instantiation

Example: (4*1 mux using 2*1 mux)

```
module mux_2to1(i0, i1, sel, out);
    input i0, i1, sel;
    output out;
    always @(i0, i1, sel)
    begin
        if (sel)
            out = i1;
        else
            out = i0;
    end
endmodule
```



```
module mux_4to1 (i0, i1, i2, i3, s1, s0, out);
  input i0, i1, i2, i3, s1, s0;
  output out;

  wire x1, x2;

  mux_2to1 m1(i0, i1, s1, x1);
  mux_2to1 m2(i2, i3, s1, x2);
  mux_2to1 m3(x1, x2, s0, out);
endmodule
```

* Values & signal strength

- verilog supports 4 values levels & 8 strength levels to model the functionality of real hardware.

Value levels - Condition in Hardware Circuits

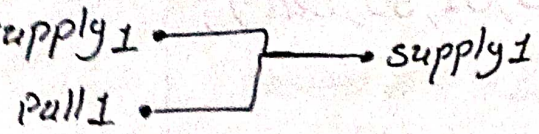
- 0 - Logic zero, false condn
- 1 - Logic one, True condn
- X - Unknown Logic value
- Z - High impedncee, floating state

Signal strength

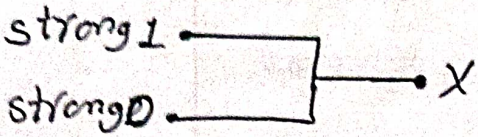
| strength level | Type | Degree |
|---|---|---|
| - Supply | - Driving | strong |
| - strong | - -11- | ↑ |
| - pull | - -11- | |
| - large | - storage | |
| - weak | - Driving | |
| - medium | - storage | |
| - small | --11- | |
| - highz | high impedence | weak |

example

(1)




(2)




⇒ If two signals of unequal strength are driven on a wire, the stronger signal prevails.

⇒ If two signals of equal strengths are driven on a wire, the result is unknown.

---

## * Port Assignment

⇒ Input – internally net, externally reg or net

⇒ output - internally reg or net, externally net

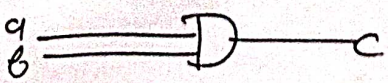
⇒ in out - only wire data type

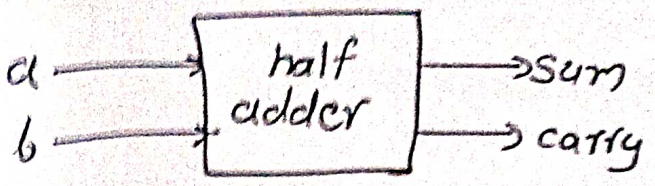
---

## * Net data type

- Nets represent connections b/w hardware elements.
- It must be continuously driven i.e. cannot be used to store the values.
- Nets are declared primarily with the keyword "wire"
- The default value of net is z
- Net represent a class of data types such as wire, wand, wor, tri, triand, trior, etc.

eg ⇒ consider the below figure, Net c is connected to the output of and gate





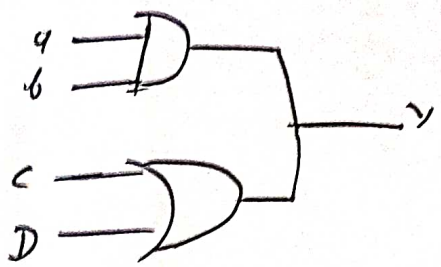

```
module half_add(a,b,sum,carry);
  input a,b;
  output sum,carry;
  wire sum,carry;

  assign sum= a^b;
  assign carry = a&b;
endmodule
```

assign is always used with net data type (wire) not reg datatype

## uses of wond

Eg




```
module use_of_wire(y,a,b,C,D);
  input A,B,C,D;
  output y;
  wire y;

  assign y = A&B;
  assign y = C|D;
endmodule
```

```
module use_of_wire(Y,A,B,C,D)
  input A,B,C,D;
  output y;
  wond y;

  assign y = A&B;
  assign y = C|D
endmodule
```

here due to two different answer from and gate & or gate output will be indeterminant

here by using wond, output from and gate & or gate will go through another & and operation so output will be determinant

## *Numbers System

Syntax :- N'BXX

- N → total no. of bits
- B → Base
  - Binary → b
  - Octal → o
  - Decimal → d
  - hexadecimal → h
- XX → No. of Bits → 0, 1, x, z

| verilog | stored number |
|---|---|
| => 1'b0 | => 0 |
| => 4'b0101 | => 0101 |
| => 8'b0001xxz | => 0001xxz |
| => 8'o12 | => 00001010 |
| => 4'd5 | => 0101 |
| => 4'h7 | => 0111 |
| => 12'hABC | => 1010 1011 1100 |

---

## * Register data type

verilog supports following register data types -

-> reg (most widely used)
-> integer (used for loop counting)
-> real (used to store floating point numbers)
-> time (keeps track of simulation time)

cgm =>

```
reg count;
//single bit register variable
reg [7:0] bus;
// 8 bit bus
```

```
reg reset;
initial
begin
reset = 1'b0;
#10 reset = 1'b1;
#10 reset = 1'b0;
end
```

* integer data type
- To store integer
- eg 0, 4, -17, -148 etc.
- Synthesizable in nature
-> Default value is X
-> Default width is 32 bits

eg

```
integer count, new_num;
initial
begin
count = 4;
new_num = -17;
end
```

* Real data type
- To store floating no.
- not synthesizable
- default value is 0.0

eg

```
real count, new_num;
initial
begin
count = 4.76;
new_num = -1.3;
end
```

```
integer a;
real b;
initial
begin
b = -6.5; // a will be assigned the -7
a = b;
end
```

## * Time data Type

- To store simulation time
- not synthesizable (useful for testbench)

eg

```
time new_sim_time;
initial
begin
new_sim_time = $time
//save the current simulation time
end
```

## Bitwise operators

| | operator symbols | Operation Perormed | no. of op$^{rs}$ |
|---|---|---|---|
| 1. | ~ | 1's complement | 1 |
| 2. | & | Bitwise AND | 2 |
| 3. | \| | Bitwise OR | 2 |
| 4. | ^ | Bitwise XOR | 2 |
| 5 | ~^ or ^~ | Bitwise XNOR | 2 |

ex => Let A[2:0], B[2:0], C[2:0] are three bit vectors &
f and w are two scalars

(i) C = ~A

A[2:0] => A[2] A[1] A[0]

C[2:0] => C[2] C[1] C[0]

This statement produces the result as C(2) = A(2)'

C(1) = A(1)'

C(0) = A(0)'

Let A = 110

Then C = ~A = 001

Vector A is of three bits so all the bits are complemented

Let w = 0,

Then f = ~w = ~0 = 1

Here f & w are scalars so only 1 bit result is produced

(ii) C = A & B

This statement produces the results as C(2) = A(2) . B(2)

C(1) = A(1) . B(1)

C(0) = A(0) . B(0)

Let A[2:0] = 110 and B[2:0] = 011

Then C = 010

(iii) C = A ^ B

C(2) = A(2) ^ B(2), C(1) = A(1) ^ B(1)

C(0) = A(0) ^ B(0)

This statement produces the results as ~~C(2) = A(2)', C(1) = A(1)'~~

~~C(0) = A(0)'~~

Let A[2:0] = 110 and B[2:0] = 001

Then C = 111

---

disable input

AND Gate

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

zero is disable input

means -> जब कोई input में zero आ रहा है output zero है, चाहे रहा है

OR Gate

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

one is disable input

# Things to remember for Bitwise operations:

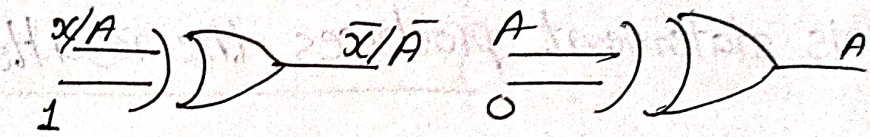
| & | 0 | 1 | X |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | X |
| X | 0 | X | X |

| \| | 0 | 1 | X |
|---|---|---|---|
| 0 | 0 | 1 | X |
| 1 | 1 | 1 | 1 |
| X | X | 1 | X |

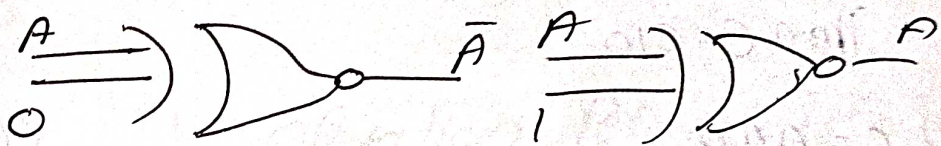
| ^ | 0 | 1 | X |
|---|---|---|---|
| 0 | 0 | 1 | X |
| 1 | 1 | 0 | X |
| X | X | X | X |

| ~^ | 0 | 1 | X |
|---|---|---|---|
| 0 | 1 | 0 | X |
| 1 | 0 | 1 | X |
| X | X | X | X |

XOR Gate




XNOR Gate




---

## Point to remember:

If f is scalar type & and A[2:0] & B[2:0] are two 3 bit vectors then

f = A & B

will produce results by performing operations on lower bits only

so, f = A[0] & B[0] will be the answer

Let A[2:0] = 110 and B[2:0] = 001

Then f = 0 & 1 = 0

# Logical operators

Logical operators operate on individual bits of operand

| | operator symbol | Operation Performed |
|---|---|---|
| 1. | ! | NOT |
| 2. | && | AND |
| 3. | \|\| | OR |

Ex => let A[2:0], B[2:0], C[2:0] are three bit vectors and f and w are two scalars.

## For Scalar operands

a) ! operator performs 1's complement and it has the same effect on operand as ~ in Bitwise

b) If w=0, then

f = !w = w' = 0' = 1

f = ~w = w') = 0' = 1

Both have some results

## For Vector operands

(i) f = !A = [A[2] + A[1] + A[0]]'

Let A[2:0] = 100

f = !A = [1 + 0 + 0]' = 1' = 0

(ii) $f = A \&\& B$

$f = [A[2] + A[1] + A[0]] . [B[2] + B[1] + B[0]]$

Let $A[2:0] = 100$ and $B[2:0] = 010$

$f = [1+0+0] . [0+1+0] = 1.1 = 1$

(iii) $f = A \| B$

$f = [A[2] + A[1] + A[0]] + [B[2] + B[1] + B[0]]$

Let $A[2:0] = 100$ and $B[2:0] = 010$

$f = [1+0+0] + [0+1+0] = 1+1 = 1$

---

## XOR & XNOR

| | XOR | XNOR | no. of 1's |
|---|---|---|---|
| 000 | 0 | 0 | 0 |
| 001 | 1 | 1 | 1 |
| 010 | 1 | 1 | 1 |
| 011 | 0 | 0 | 2 |
| 100 | 1 | 1 | 1 |
| 101 | 0 | 0 | 2 |
| 110 | 0 | 0 | 2 |
| 111 | 1 | 1 | 3 |

XOR => odd no. of 1's Detector

eg 10011010100 => no. of 1's => 5 => odd => 1

XNOR => Even no. of 1's detector for even no. of inputs / bits only

# Reduction operators

Reduction operator performs an operation on the bits of a single vector operand & produces 1 bit result.

| | operator symbols | operation performed | no. of operands |
|---|---|---|---|
| 1 | & | Reduction AND | 1 |
| 2. | ~& | Reduction NAND | 1 |
| 3. | \| | Reduction OR | 1 |
| 4. | ~\| | Reduction NOR | 1 |
| 5. | ^ | Reduction XOR | 1 |
| 6. | ~^ or ^~ | Reduction XNOR | 1 |

Eg ⇒ (i) Let A[2:0] = 110

Then f = &A

f = [A[2]. A[1]. A[0]] = [1.1.0] = 0. (Scalar result)

(ii) Let A[2:0] = 100

Then f = ~&A

f = [A[2]. A[1]. A[0]]' = [1.0.0]' = 1 (Scalar result)

(iii) Let A[2:0] = 111

Then f = |A

f = [A[2] + A[1] + A[0]] = [1+1+1] = 1 (Scalar result)

(iv) Let A[2:0] = 001

Then f = ~|A

f = [A[2] + A[1] + A[0]]' = [0+0+1]' = 0 (Scalar result)

## Arithmetic operators

They perform standard arithmetic operations

| Operator symbols | Operation performed | no. of operands |
|---|---|---|
| + | addition | 2 |
| − | Subtraction | 2 |
| − | 2's Complement | 1 |
| * | Multiplication | 2 |
| / | Division | 2 |

(i) C = A+B; Puts the sum of A and B in C

(ii) C = A-B; puts the difference of A and B in C

| A[2] | A[1] | A[0] |
|---|---|---|
| B[2] | B[1] | B[0] |
| C[2] | C[1] | C[0] |

(iii) C = -A; Puts the 2's complement of A in C

Let A = 001

C = -[001] = 111