Unit-V

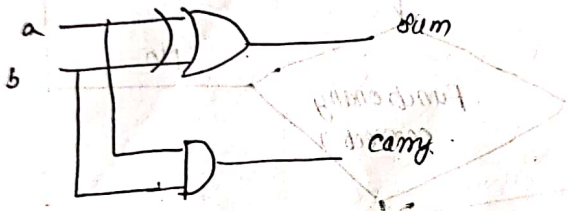
Verilog HDL

Levels of design description

(1) Gate level modeling

→ Design is carried out in terms of basic gates.

→ All basic gates are available as ready modules called primitives.




```
module half_adder (sum, carry, a, b);
output sum, carry;
input a, b;
xor X1 (sum, a, b);
and X2 (carry, a, b);
endmodule
```

(2) Data flow modeling

All operations on signals and variables are represented in terms of assignments.

The syntax for continuous assignments is

assign [delay] LHS_net = RHS_expression

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

```
assign #4 sum = a ^ b;
assign #3 carry = a & b;
end module
```

(3) Behavioral level modeling

The design is described using procedural constructs. These are

(1) Initial statement – This statement executes only once.
(2) always statement – This executes in a loop.

only register data type can be assigned a value in either of these 2 statements.

```
module beh_half_adder ( sum, carry, a, b);
output sum, carry;
input a, b;
always @(a or b)
begin
  sum = a ^ b;
  carry = a & b;
end
end module
```

Ports

Keyword	Type of port
Input	I/p port
output	o/p port
Inout	Bidirectional port.

Data types

(1) Net – connections b/w hardware elements
(2) Register – represents data storage elements.

Net types

wire, tri, wor, trior, wand, triand, trireg

→ Represents physical connection b/w structural elements.

→ Its value is determined from value of its drivers like continuous assignments or gate o/p.

→ Default value is 'z'.

Eg:- wire Reset;

Register types

→ Represents data storage elements.

→ It is assigned values only within an always statement or initial statement.

→ Default value is 'x'.

Eg:- reg clk;

```
initial
  begin
    clk = 1'b0;
    #10 clk = ~clk;
  end.
```

(1) Integer — contains integer variable

Eg:- integer count;
integer addr [3:0];

(2) Real — specifies the variable in decimal notation or in scientific notation.

Eg:- real pwd;

```
initial
  begin
    pwd = 2.13;
  end.
```

(3) Time

Value

Eg:-

$ d

$

(3) time — used to store and manipulate time values. stores simulation time.

Eg:- time curtime;
initial
curtime = $time; // saves current simulation time.
end.

system tasks

$display — display values of variables or strings or expressions

$time - returns current simulation time.

$monitor - monitors a signal when its value changes

$stop - puts simulation in interactive mode.

$finish - terminates simulation

Comments

(1) single line comment — starts with //. Verilog skips from that points to the end

(2) Multiple line comment — starts with /* and ends with */.

\b - blank space
\t - tabs
\n - new line

Eg:- monitor (time, "a = %b \t, b = %b \t, sum = %b \n", a, b, sum);

value set

0 - Logic zero
1 - Logic one
X - unknown logic value
Z - High impedance.

gate_type [instance name] [term1, term2, ... termn];

Multiple I/P

Eg:- and g1 (out, a, b);

nand g2 (out, a, b);

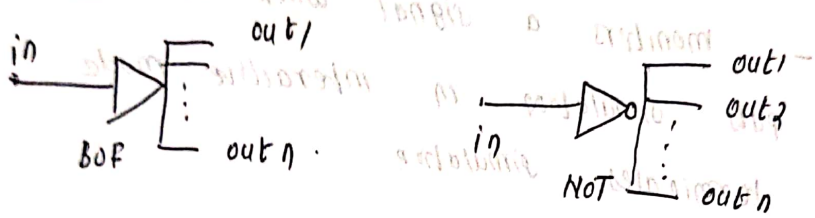
or g3 (out, a, b);

nor g4 (out, a, b);

xor g5 (out, a, b);

xnor g6 (out, a, b);

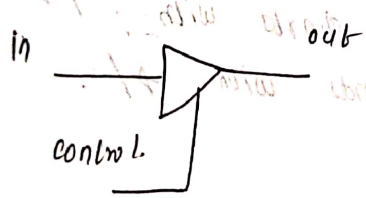
Multiple o/p



Eg:- buf g1 (fan(0), fan(1), fan(2), clk);

not g2 (status1, status2, ready);

Tristate gates



Eg:- bufif1 g1 (out, in, control);

↳ out = in if control = 1. else out = z.

bufif0 g1 (out, in, control);

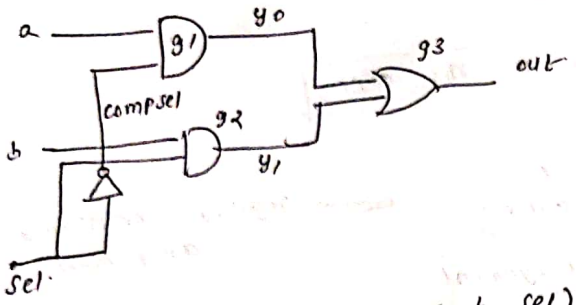
↳ out = in if control = 0, else out = z.

Illy,

notif0 g1 (out, in, control);

notif1 g1 (out, in, control);

Eg:- 2 to 1 mux



sel	out
0	a
1	b

sel	a	b	o/p
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

```
module mux (out, a, b, sel);
output out;
input a, b, sel;
wire compsel;
wire y0, y1;
not g0 (compsel, sel);
and g1 (y0, a, compsel);
and g2 (y1, b, sel);
or g3 (out, g1, g2);
end module
```

Note: delays with rise, fall and turn off

Eg:- and #(3,5) g1 (out, a, b) // rise = 3, fall = 5

and #(3,4,5) g1 (out, a, b) // rise = 3, fall = 4, turn off = 5

turn-off delay → transition to high impedance (z) from another value (0,1,x) is turn off delay.

Data flow modeling

Continuous assignments

```
assign lhs_target = rhs_expr;
```

Eg:-

```
wire c, a, b;
assign c = a + b;
```

← regular continuous assignment

Implicit continuous assignment

```
wire c = a & b;
```

Delays

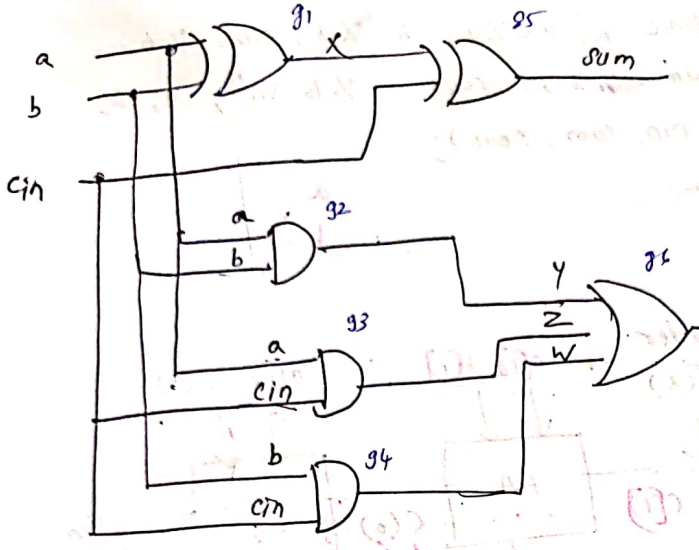
Regular assignment delay

```
wire c, a, b;
assign #2 c = a & b;
```

Assignment takes effect with time delay of 2 time steps. If a and b changes in value, the program waits for 2 time steps, computes value of c based on values of a & b at the time of computation, and assigns it to c.

Implicit continuous assignment delay

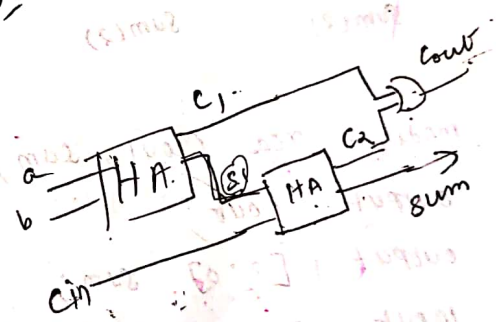
structural gate level modeling of Full adder



sum = A xor B xor cin

carry := (A and B) or (A and cin) or (B and cin).

```
module fa (cout, sum, a, b, cin);
output cout, sum;
input a, b, cin;
wire c1, c2, s1;
ha g1 (c1, s1, a, b);
ha g2 (c2, sum, s1, cin);
or g3 (cout, c1, c2);
end module
```



Test bench

```
module fa_tb;
wire cout, sum;
reg a, b, cin;
fa f1 (cout, sum, a, b, cin);
initial
begin
a = 1'b0; b = 1'b0; cin = 1'b0;
#5 a = 1'b0; b = 1'b0; cin = 1'b1;
#5 a = 1'b0; b = 1'b1; cin = 1'b0;
#5 a = 1'b0; b = 1'b1; cin = 1'b1;
#5 a = 1'b1; b = 1'b1; cin = 1'b1;
```

3 bit

a = 4'b0000
b 1010
cin 1111

1'b0;
1'b1;

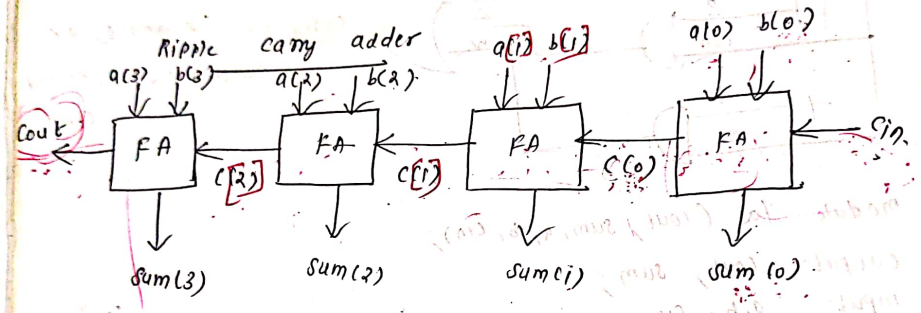
end

initial

$monitor ($time, " a=%b, b=%b, cin=%b, sum=%b, cout=%b \n", a, b, cin, sum, cout);

initial #40 $stop;

endmodule

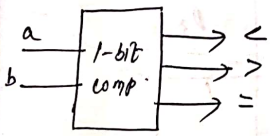



```
module rca (cout, sum, a, b, cin);
output cout;
output [3:0] sum;
input [3:0] a,b;
input cin;
wire [3:0] c;
fa g1 (c[0], sum[0], a[0], b[0], cin);
fa g2 (c[1], sum[1], a[1], b[1], c[0]);
fa g3 (c[2], sum[2], a[2], b[2], c[1]);
fa g4 (cout, sum[3], a[3], b[3], c[2]);
endmodule
```

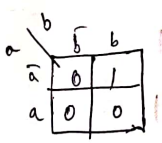
Test bench

```
module   rca_tb ;
wire   cout ;
wire   [3:0]  sum ;
 reg  [3:0]   a,b ;
 reg  cin ;
  rca   r1 (cout, sum, a, b, cin);
  initial
  begin
     a = 4'b1010 ;  b = 4'b1111 ;  cin = 1'b0 ;
  #10  a = 4'b1110 ;  b = 4'b1100 ;  cin = 1'b1 ;
  #10  a = 4'b0110 ;  b = 4'b1001 ;  cin = 1'b1 ;
  #10  a = 4'b1001 ;  b = 4'b1100 ;  cin = 1'b0 ;
  end
  initial
  $monitor ("$time, "a = %b , b = %b , cin = %b,
            sum = %b , cout = %b \n", a, b, cin, sum, cout);
  initial  #50  $stop ;
endmodule
```

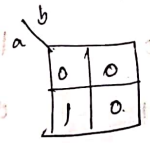
1- bit comparator



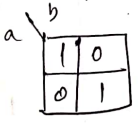
a	b	<	>	=
0	0	0	0	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	1



Lesser = $\bar{a}b$



greater = $a\bar{b}$



equal = $\bar{a}\bar{b} + ab$.

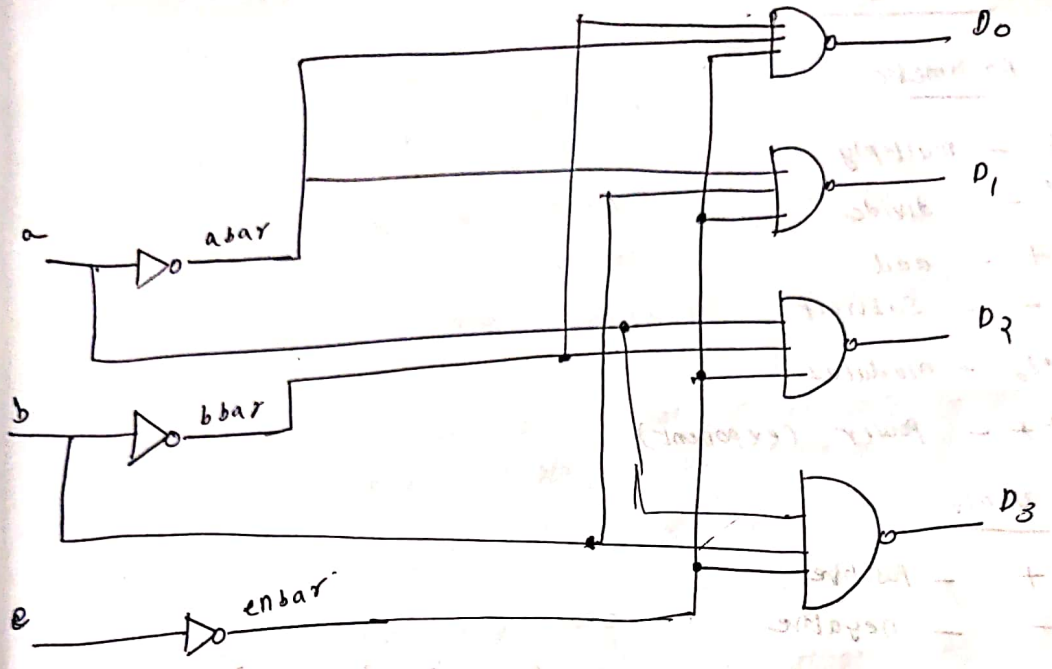
```
module comp_1_bit (a, b, lesser, greater, equal);
    input a;
    input b;
    output lesser;
    output greater;
    output equal;
    wire x, y, l, m;
    not g1 (x, a);
    not g2 (y, b);
    and g3 (greater, a, y);
    and g4 (lesser, x, b);
    and g5 (l, x, y);
    and g6 (m, a, b);
    or g7 (equal, l, m);
endmodule
```

Test bench

```
module comp_1_bit_tb ();
    reg a;
    reg b;
    wire lesser, greater, equal;
    comp_1_bit c1 (a, b, lesser, greater, equal);
    initial
    begin
        a = 1'b0; b = 1'b0;
    #10 a = 1'b0; b = 1'b1;
    #10 a = 1'b1; b = 1'b0;
    #10 a = 1'b1; b = 1'b1;
    end
    initial
    $monitor ($time, "a = %b, b = %b, lesser = %b, greater = %b, equal = %b \n", a, b, lesser, greater, equal);
    initial #40 $stop;
```

2 - to - 4 decoder

E	A	B	D_0	D_1	D_2	D_3
1	x	x	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0



3 to 8 decoder

assign

```
module   decoder ( d, a, b, e);
output   [3:0] d;
input    a, b, e;
wire     abar, bbar, ebar;
not      n1 (abar, a);
not      n2 (bbar, b);
not      n3 (ebar, e);
nand     n4 (d[0], abar, bbar, ebar);
nand     n5 (d[1], abar, b, ebar);
nand     n6 (d[2], a, bbar, ebar);
nand     n7 (d[3], a, b, ebar);
endmodule
```

Operator types

1) Arithmetic

* - multiply
/ - divide
+ - add
- - subtract
% - modulus
** - power (exponent)

2) Unary

+ - positive
- - negative

3) Logical

! - logical negation (performs complement)
&& - logical AND
|| - logical OR

4) Relational

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

5) Equality

== - equality
!= - in equality

6) Bitwise

~ - bitwise negation
& - bitwise AND
| - bitwise OR
^ - " XOR
^~ or ~^ - bitwise XNOR.

Shift

>> - Right shift << - Left shift
>>> - Arithmetic right shift <<< - Arithmetic left shift

concatenation - { }

conditional - ? :

Behavioral modeling

→ A module may contain arbitrary no. of initial or always statements

→ These execute concurrently with respect to each other.

→ All always & initial statements execute concurrently starting at time 0.

Initial statement

→ executes only once.

→ begins its execution at start of simulation at time 0.

Always statements

→ executes repeatedly

→ starts at time 0.

Eg:- always → executes loop indefinitely
clk = ~clk

Timing controls

① Delay control
② Event control
③ Level sensitive control.

① Delay control

Regular delay control

delay procedural_statement.

Eg:-
```
initial
begin
#3. a = 4'b1000;
#6  a = 4'b1010;
```

initial
// executes at time 0.
After 3 time units
1st assignment is executed.

Intra - assignment delay control

```
initial
begin
x=0; y=0;
z = #10 x*y;
end.
```

// The value of x and y is assigned at time = 0 and then evaluates x*y and then waits for 10 time units to assign value to z.

④ Event control

@ (negedge clk); → executes following block at -ve edge of the variable clk.

@ (posedge clk) — executes at +ve edge

@ clk — executes at both edges.

@ (posedge clk1 or negedge clk2)

↳ executes at 2 cases

① whenever clk1 changes from 0 to 1

② clk2 changes from 1 to 0.

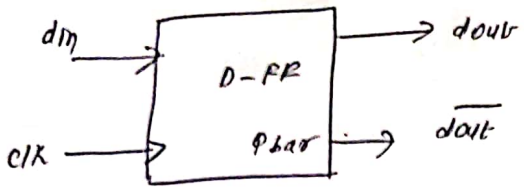
③ Level sensitive timing control

Eg:-

```
always
wait (enable)
# 20 cnt = cnt+1;
```

The event on enable is monitored continuously. If enable is 0, then statement will not be executed. If enable is 1, then statement count is executed after 20 time units.

D- flip flop (Behavioral modeling)




```
module dff (dout, din, clk);
output dout;
input clk, din;
reg dout;
initial
  dout = 1'b0;
  always @ (negedge clk)
    dout = din;
end module.
```

Block statements

① sequential block

→ st-atements in sequential block execute in sequence.

→ A delay value in each statement is relative to simulation time of execution of previous statement.

Syntax

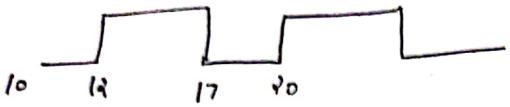
```
begin
  [ : block_id { declarations } ]
    procedural_statements
end.
```

→ A block can be labeled optionally. If labeled, reg. can be declared locally within the block.

→ The value of local reg remain valid throughout the entire simulation.

Eg:-
```
begin
#2  wave = 1;
#5  wave = 0;
#3  wave = 1;
end.
```

Assuming sequential block gets executed at 10 time units



② parallel block

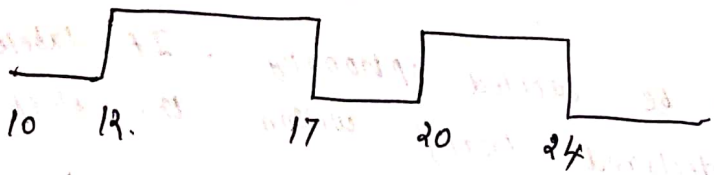
→ statements executes concurrently

→ has delimiters fork and join

Syntax

```
fork
[ : block_id { declarations } ]
    procedural statements
join.
```

Eg:-
```
fork.
#2  wave = 1;
#7  wave = 0;
#10 wave = 1;
join.
```



Procedural assignments

→ The assignment statement within an initial or an always statement is called procedural assignment.

```
always @ (clk)
begin
reg count, z;
count = count + 1;
z = count;
end
```

* types

(i) Blocking procedural assignments

* The assignment operator is '=' in blocking assignment.

* Blocking procedural assignment is executed before any of the statements that follow it are executed.

```
module fa (sum, cout, A, B, cin);
output sum, cout;
input A, B, cin;
always @ (A or B or cin)
begin
reg c1, c2, c3;
sum = A ^ B ^ cin;        // sum assignment occurs first, sum is computed then next statement executes and c1 is assigned.
c1 = A & B;
c2 = B & cin;
c3 = A & cin;
cout = c1 | c2 | c3;
end
endmodule.
```

(ii) Non blocking assignments

* Here the execution is concurrent with that of following assignment.
* symbol <= is used.

Eg:-

```
initial
begin
A = 1'b0;
B = 1'b1;
C = 1'b0;
end.
always @ (posedge clk)
begin
A <= B;
@ (negedge clk)
C <= B & (~C);
# 2 B <= C;
end.
```

At +ve edge of the clk

→ values A, B, C are read and stored, B & ~C are computed.

→ A is assigned the value of B i.e; 1

→ All these are done concurrently at the same time.

During -ve edge

→ C is assigned with the value of B and ~C.

→ After 2 units delay, B is assigned with the value of C stored earlier.

If statement

```
if (condn 1)
  procedural statement 1
else if (condn 2)
    statement 2
else
    statement 3.
```

Eg:-

```
module mux (out, a, b, sel);
output out;
input a, b, sel;
reg out;
always @(a, b)
  begin
    if (sel == 1'b0)
        out = a;
    else
        out = b;
  end
endmodule
```

case statement

```
case (case_expr)
  ==
  default :
endcase.
```

↳ covers all values that are not covered by case item expr.

Eg:- Design of ALU

```
module alu (out, a, b, opcode);
output [7:0] out;
input [3:0] a, b;
input [1:0] opcode;
reg [7:0] out;
```

↳ It is declared as reg because procedural statement var. can be assigned only as reg.

```
parameter
    ADD = 2'b00,
    SUB = 2'b01,
    MUL = 2'b10,
    DIV = 2'b11;

always @ (a, b, opcode)
  case (opcode):
    ADD : out = a+b;
    SUB : out = a-b;
    MUL : out = a*b;
    DIV : out = a/b;
  endcase
endmodule
```

Loop statement

1) forever loop → Repeated execution of a block in an endless manner

Syntax

```
forever
    Procedural_statement
```

* stopped using disable statement.

Eg:-

```
always @ (posedge en)
    forever #2 clk = ~clk;
```

2) Repeat loop

Syntax

```
repeat (loop_count)
    procedural statement;
```

Eg:-
```
always
  begin
    repeat (3)
      begin
        @ (negedge clk)
          m[i] = i*8;
          i = i + 1;
      end
```

(3) while loop statement

If boolean expr. evaluated is true, the statement is executed and expression is evaluated and checked.

If false, loop is terminated and following statement is taken for execution.

syntax Eg:- while (condition)
procedural statement

Eg:-
```
always
  begin
    while (i<8)
      @ (negedge clk)
        begin
        end
  end
```

(4) for-loop statement

It has 4 parts

(i) initial - assignment

(ii) condition

(iii) step - assignment

(iv) procedural statement

Syntax

for (initial-assign ; condn ; step-assign)

procedural statement

Eg:-

```
always
begin
  for (i=0; i<8; i=i+1)
  ==========
end.
```

Procedural continuous assignment

2 types

(i) assign and deassign procedural statements → assigns to reg

(ii) force and release " " → assigns to net also used for reg.

Assign - deassign

→ allows continuous assignments within a behavioral block.

Eg:-

```
module clr (a, n, clr, pr, clk);
output [7:0] a;
input [7:0] n;
input clr, pr, clk;
reg [7:0] a;
initial a = 8'h00;
always @ (posedge clk)
  a = a + 1'b1;
always @ (clr or pr)
  if (clr) assign a = 7'h00;
```

Eg:-

```
initial
  begin
  force s = a&b;
  #3;
  release s
  end
```

Scanned by CamScanner

```
else if (pr) assign a = n;
else deassign a;
endmodule
```

* If clr goes high, a is reset to 0.
  If preset goes high, a is set to a number specified as n.
* These assignments will remain as long as clr and preset are active.
* If both goes low, then value of a is incremented at every +ve edge of the clk.

Force - release

* similar to assign and deassign, which can be applied to nets and registers.

Gate level modeling of priority encoder

→ If 2 or more inputs are equal to 1 at same time, ip having highest priority will take precedence.

→ The valid bit V is set to 1 when one or more i/p are equal to 1. If all ip are 0, then valid bit is set to 0.

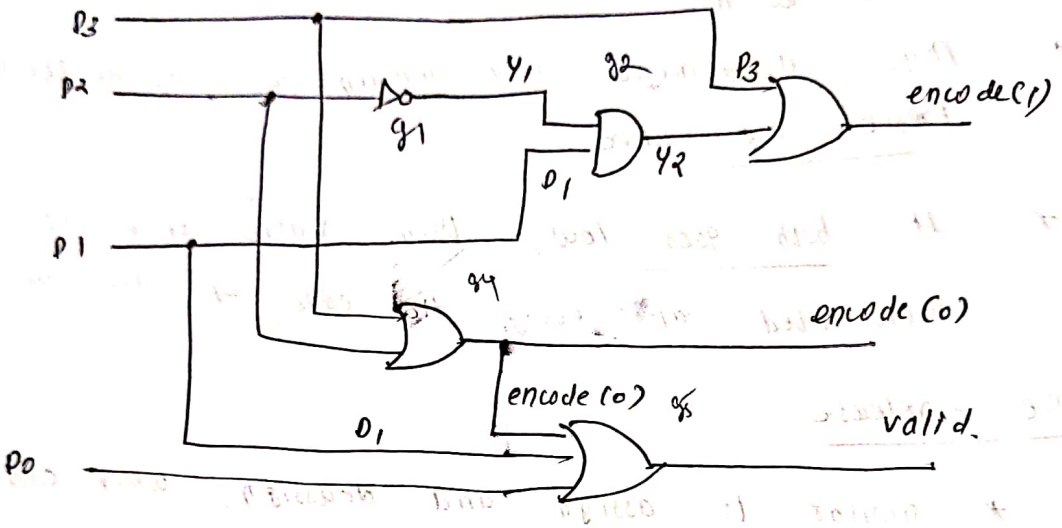
Inputs				outputs		
D_0	D_1	D_2	D_3	Encode o	encode i	valid(V)
0	0	0	0	x	x	0
1	0	0	0	0	0	1
0	1	0	0	0	1	1
0	0	1	0	1	0	1
0	0	0	1	1	1	1

encode (0) = $D_2 + D_3$

encode (1) = $D_3 + D_1 \overline{D_2}$

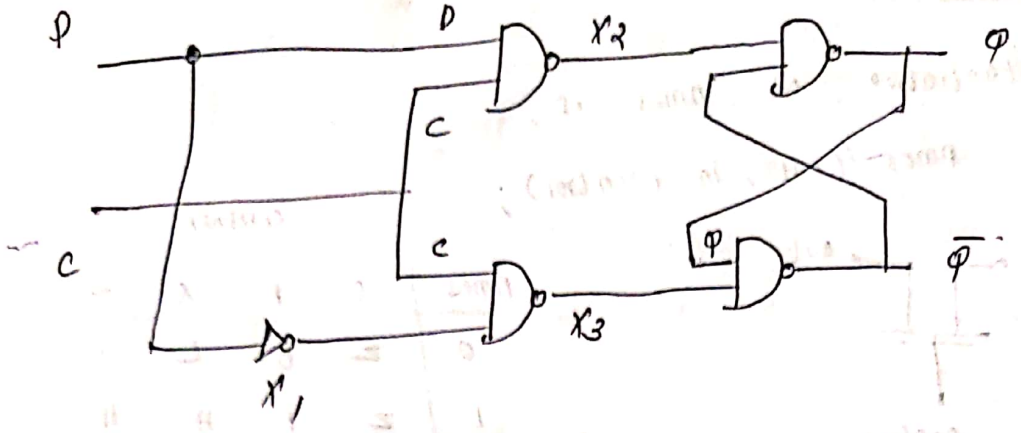
valid = $D_0 + D_1 + D_2 + D_3$

encode (0) = D_2.




```
module priority_encoder (encode0, encode1, valid, d0, d1, d2, d3);
output encode0, encode1, valid;
input d0, d1, d2, d3;
wire y1, y2;
not g1 (y1, d2);
and g2 (y2, y1, d1);
or g3 (encode1, d3, y2);
or g4 (encode0, d3, d2);
or g5 (valid, encode0, d1, d0);
endmodule
```

D-latch



control (C)	D	Next state of Q.
0	X	No change
1	0	Q = 0 Reset state
1	1	Q = 1 Set state

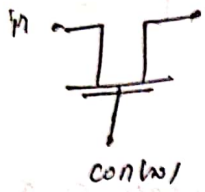
```
module D_latch (q, qbar, d, c);
wire x1, x2, x3;
output q, qbar;
input d, c;
wire x1, x2, x3;
not g1 (x1, d);
nand g2 (x2, q, c);
nand g3 (x3, x1, c);
nand g4 (q, x2, qbar);
nand g5 (qbar, x3, q);
endmodule.
```

Switch-level modeling

Mos switches

Instantiation of nmos is,

nmos (out, in, control);



nmos	control 0	1	X	Z
0	z	0	L	L
1	z	1	H	H
X	z	X	X	X
Z	z	z	z	z

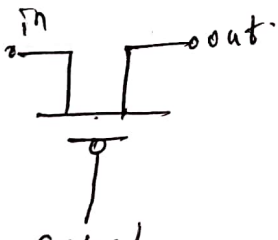
in

→ When control i/p is at high the switch is ON.

→ Connects the i/p lead to outside, and offers zero imp.

→ When control i/p is low, switch is off and o/p is floating (z).

pmos



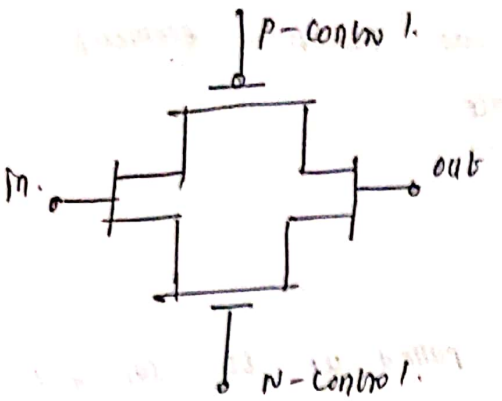
pmos	control i/p 0	1	X	Z
0	0	z	L	L
1	1	z	H	H
X	X	z	X	X
Z	z	z	z	z

in

→ When i/p is high, switch is OFF and o/p is floating (z).

→ When i/p is low, switch turns ON and o/p offers zero impedance.

cmos switch



→ N-control turns ON nmos and keeps it ON when it is in 1-state.

→ P-control turns ON pmos and keeps it ON when it is 0-state.

cmos is instantiated as,

```
cmos u1 (out, in, p-control, N-control);

module cmosswitch (out, in, n_ctrl, P_ctrl);
output out;
input in, P_ctrl, n_ctrl;
// cmos u1 (out, in, P_ctrl, n_ctrl);
nmos u1 (out, in, n_ctrl);
pmos u2 (out, in, P_ctrl);
endmodule
```

power and ground

power and ground sources are defined with keywords supply1 and supply0.

supply 1 are equivalent to Vdd.
supply 0 " " " Vss.

Eg-

```
supply1 vdd;
supply0 vss;
assign x = vdd;
assign y = vss;
```

pull up and pull down

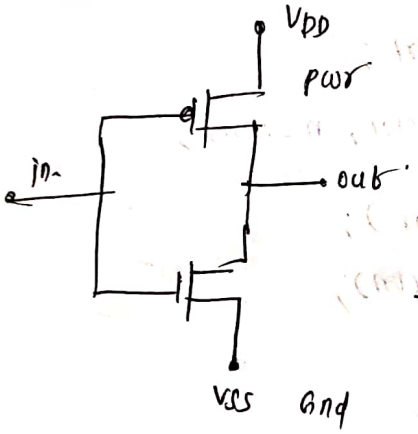
Mos transistor functions as resistive element when it is active state.

```
pullup (x);
pulldown (y);
```

→ net x is connected or pulled up to supply 1 through a resistance

→ net y is connected or pulled down to supply 0 through a resistance.

CMos inverter



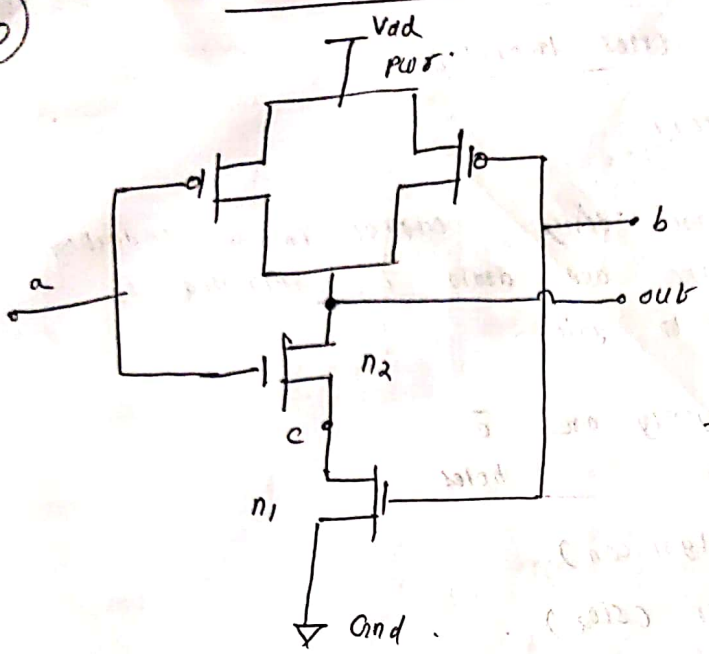
→ when i/p is low, nmos is off, but pmos is on. supply1 is connected to o/p. so o/p is high

→ when i/p is high, pmos is off. But nmos is on. supply0 ie; Gnd is connected to o/p. so o/p is low.

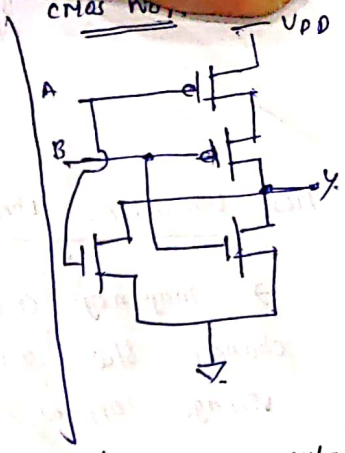
```
module cmos_inv (out, in);
  output out;
  input in;
  supply1 pwr;
  supply0 gnd;

  pmos u1 (out, pwr, in);
  nmos u2 (out, gnd, in);
endmodule
```

(16)

CMOS NAND gate



CMOS NOT



a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

→ When a & b is high, n_1 and n_2 are pulled down towards gnd, and o/p is zero.

→ When any one of I/p is low, P1 or P2 turns ON and terminals are pulled up towards the supply ie; Vdd and o/p is high.

```
module cmos_nand (out, a, b);
output out;
input a, b;
supply1 pwr;
supply0 gnd;
wire c;
nmos u1 (out, c, a);
nmos u2 (c, gnd, b);
pmos u3 (out, pwr, a);
pmos u4 (out, pwr, b);
endmodule
```