Verilog Modeling

1. Modeling Style
2. Verilog Modeling
    - Multiplexer
    - Decoder
    - Encoder

Verilog Modeling Styles

- Dataflow modelling – description of input-output relation of a circuit using continuous assignment statements.

- Behavioural modelling – description of a circuit behaviour using procedural statements.

- Structural modelling – description of a circuit using a set of interconnected components/modules.

Dataflow modeling

- Given a Boolean equation $f = \overline{x1} \cdot \overline{x2} \cdot \overline{x3} + x1 \cdot \overline{x2} \cdot \overline{x3} + x1 \cdot \overline{x2} \cdot x3 + x1 \cdot x2 \cdot \overline{x3}$

```
module func1 (x1,x2,x3, f);
        input x1,x2,x3;
        output f;
        assign f= (~x1&~x2&~x3) | (x1&~x2&~x3) | (x1&~x2&x3) | (x1&x2&~x3);
endmodule
```

- how about your 2-bit comparator?

Structural Modeling

Module name

Port list

```
module xor_logic ( out, a, b );
input a, b;
output out;
wire aBar, bBar, t1, t2;

not invA (aBar, a);
not invB (bBar, b);
and and1 (t1, a, bBar);
and and2 (.o(t2), .i1(b), .i2(aBar));
or or1 (out, t1, t2);
endmodule
```

Port declaration

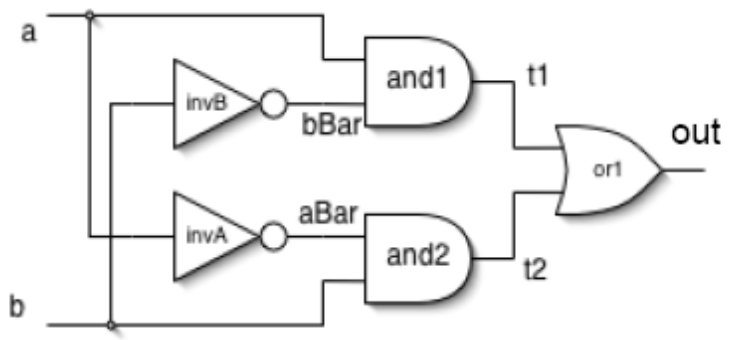
Internal signal declaration

Positional port connection

Named port connection

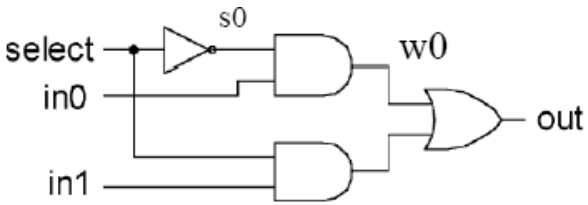
Instances of built-in components

Module instantiation



Example

```
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
input in0,in1,select;
output out;
wire s0,w0,w1;

not (s0, select);
and (w0, s0, in0),
      (w1, select, in1);
or (out, w0, w1);

Endmodule          // mux2
```



Multiple instances can share the same "master" name.

Better specify explicitly.

Behavioural Modeling

```
module OR2(x,y,z);
input x,y;
output z;
reg z;

always@(x or y)
begin
        if((x==0) && (y==0)) z=0;
        else z=1;
end
endmodule
```

```
module mux4 (in0, in1, in2, in3, select, out);
input in0,in1,in2,in3;
input [1:0] select;
output out;
reg out;
  always @ (in0 in1 in2 in3 select)
        case (select)
        2'b00: out=in0;
        2'b01: out=in1;
        2'b10: out=in2;
        2'b11: out=in3;
        endcase
endmodule // mux4
```

Example

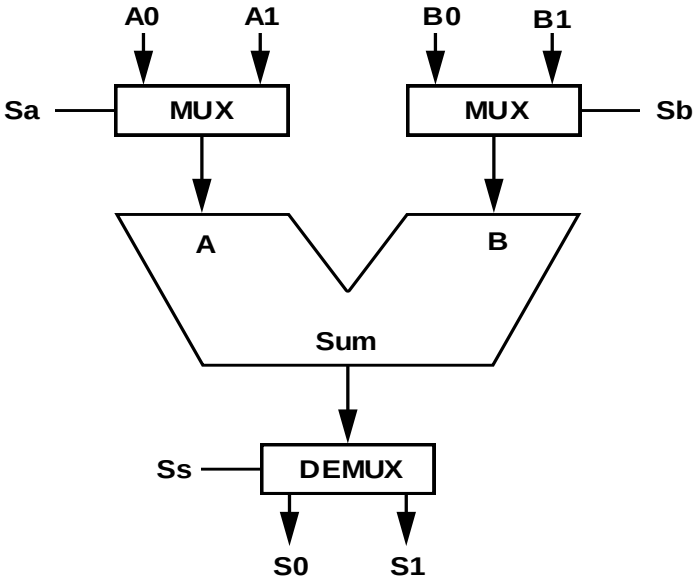
```
module 2mux(in, select, out);
input [7:0] in;
input [1:0] select;
output [1:0] out;

mux4 mux4a( in[0], in[1], in[2], in[3], select, out[0] );
mux4 mux4b( .select(select), .in0(in[4]), .in1(in[5]),
in2(in[6]), in3(in[7]),        .out(out[1])  );

endmodule
```

Verilog Modeling

1. Modeling Style
2. Verilog Modeling
   - Multiplexer
   - Decoder
   - Encoder

Multiplexers/Selectors

Multiple input sources



Multiple output destinations

Multiplexers/Selectors: General Concept

- 2^n data inputs, n control inputs, 1 output
- used to connect 2^n points to a single point
- control signal pattern form binary index of input connected to output

$Z = A' I_0 + A I_1$

A	Z
0	I_0
1	I_1

Functional form

Logical form

I_1	I_0	A	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Two alternative forms for a 2:1 Mux Truth Table

Mux in verilog

Dataflow Modeling

```
module mux2to1 (i1, i0, a, z);

input i1, i0, a;

output z;


assign z= (~a & i0) | (a & i1);


endmodule
```

Behavioral Modeling

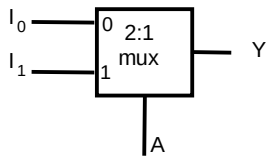
```
module mux2to1 (i1, i0, a, z);

input i1, i0, a;

output reg z;

always @(i1, i0, a) begin

    if(a) z=i1;

     else z=i0;

end

endmodule
```

if (condition)
branch1
else branch2

Structural Modeling

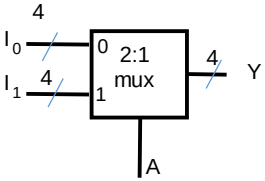
```
module mux2to1 (i1, i0, a, z);
input i1, i0, a;
output z;
wire w1,w2, w3;
not  inv1(w1,a);
and and1(w2,w1,i0);
and and2(w3,a,i1);
or or1(z, w2,w3);
endmodule
```

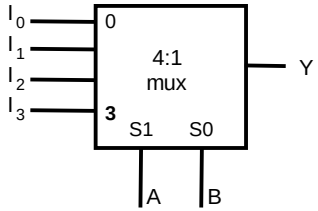
Multiplexers/Selectors



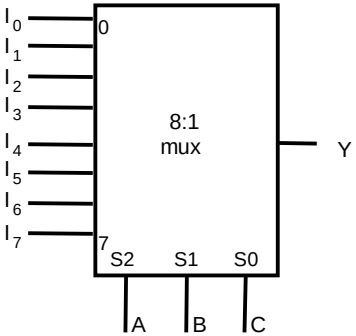
$$Y = A'\, I_0 \quad + \quad A\, I_1$$

How about this?





$$Y = A'\, B'\, I_0 + A'\, B\, I_1 + A\, B'\, I_2 + A\, B\, I_3$$



$$Y = A'\, B'\, C'\, I_0 + A'\, B'\, C\, I_1 + A'\, B\, C'\, I_2 + A'\, B\, C\, I_3 + A\, B'\, C'\, I_4 + A\, B'\, C\, I_5 + A\, B\, C'\, I_6 + A\, B\, C\, I_7$$

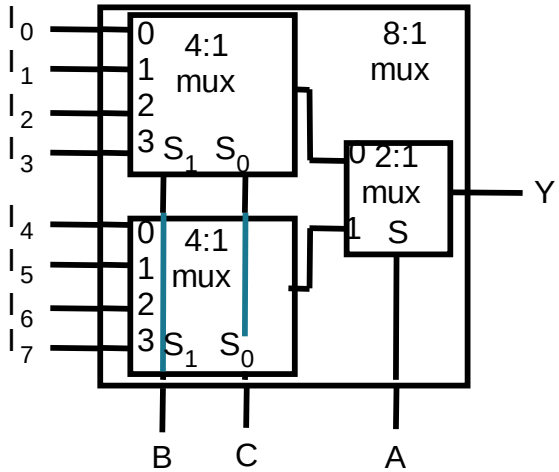
MUX in verilog

```
module mux4to1 (s, in, z);

input [1:0] s;

input [3:0] in; //e.g. 4-bit bus

output reg z;

always @(s, in)

case(s)

2'b00: z=in[0];

2'b01: z=in[1];

2'b10: z=in[2];

2'b11: z=in[3];

endcase

endmodule
```

```
module mux2to1 (i1, i0, a, z);
input i1, i0, a;
output z;
wire w1,w2, w3;
```

```
module mux4to1 (s, in, z);

input [1:0] s;

input [3:0] in;

output reg z;

wire w1, w2;

mux2to1 mux1(i[3], i[2], s[0], w1 );

mux2to1 mux2(i[1], i[0], s[0], w2 );

mux2to1 mux3(w1, w2, s[1], z);

endmodule
```

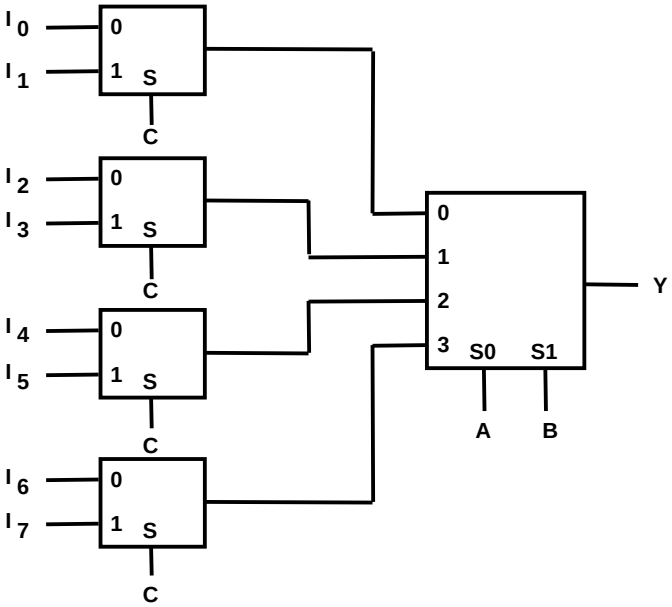
Multiplexer/Selector: Expansion



Alternative 8:1 Mux Implementation

Control signals B and C simultaneously choose one of I_0-I_3 and I_4-I_7

Control signal A chooses which of the upper or lower MUX's output to gate to Y



Issue : implied memory

- **reg ：**

variable retains old value when the variable is not assigned a new value under some condition.

- **Example:**

```
module mux4to1(en,x,z);
input en,x;
output reg z;
always @(en or x)
if(en)
  z=x;
endmodule
```

Multiplexer/Selector: Implementing logic functions

2^{n-1} :1 multiplexer can implement any function of n variables

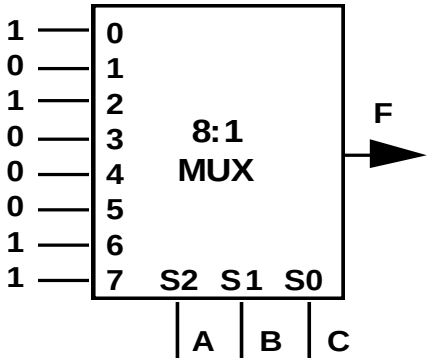
n-1 control variables; remaining variable is a data input to the mux

Example:

$F(A,B,C) = m_0 + m_2 + m_6 + m_7$

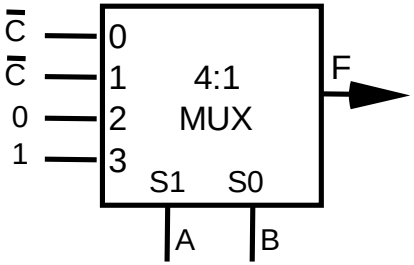
$= A' B' C' + A' B C' + A B C' + A B C$

$= A' B' (C') + A' B (C') + A B' (0) + A B (1)$



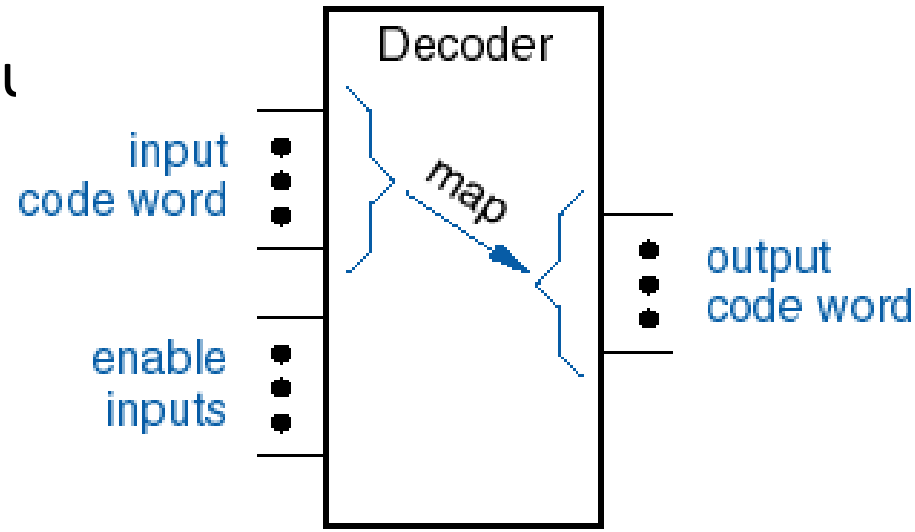
"Lookup Table"

A	B	C	F	
0	0	0	1	$\overline{C}$
0	0	1	0	
0	1	0	1	$\overline{C}$
0	1	1	0	
1	0	0	0	0
1	0	1	0	
1	1	0	1	1
1	1	1	1	



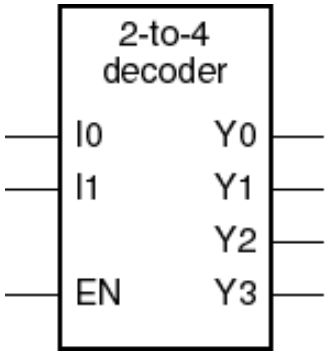
Decoders

- General decoder structu

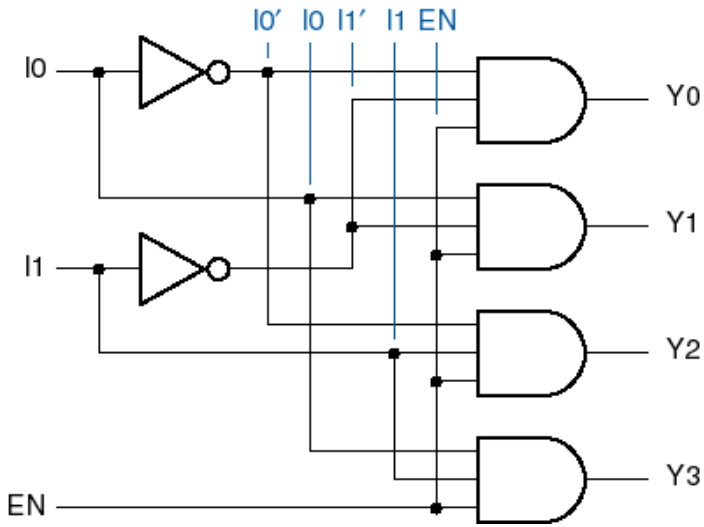


- Typically *n* inputs, 2^n outputs
  - 2-to-4, 3-to-8, 4-to-16, etc
- Control inputs (called select S) represent Binary index of output to which the input is connected
- Data input usually called "enable" (G).

Binary 2-to-4 decoder



Inputs			Outputs			
EN	I1	I0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0



Note “x” (don’t care) notation.

Decoder in Verilog #1

```
module decoder2by4_using_case (i, y, en);
input [1:0] i  ;
input  en ;
output reg [3:0] y ;

always @ (en or i)
 begin
  y = 0;
  if (en) begin
   case (i)
     2'h0 : y = 4'h1;          // or [y3 y2 y1 y0] = 0001
     2'h1 : y = 4'h2;          // or [y3 y2 y1 y0] = 0010
     2'h2 : y = 4'h4;          // or [y3 y2 y1 y0] = 0100
     2'h3 : y = 4'h8;          // or [y3 y2 y1 y0] = 1000
   endcase
  end
 end
endmodule
```

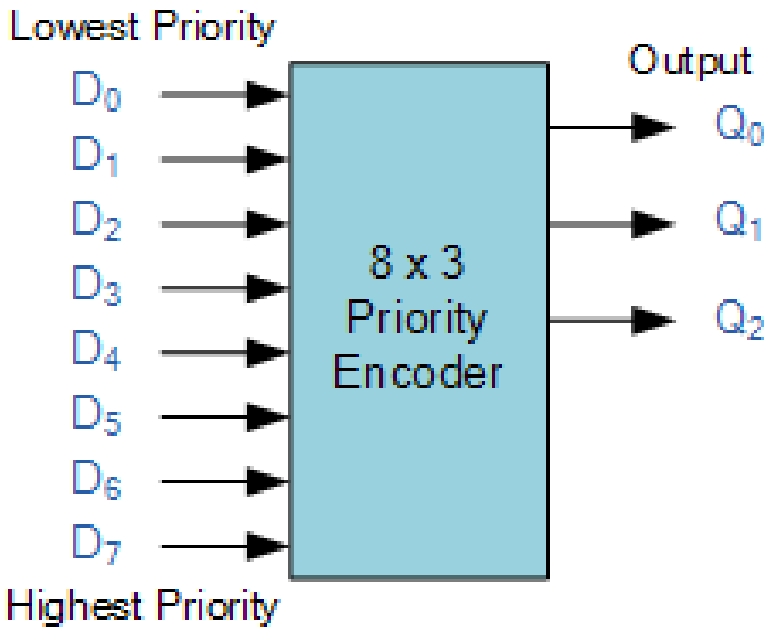
Watch out the pairs !
- begin - end
- case – endcase
- module – endmodule ,,,

Decoder in Verilog #2

```
1   // Implements a circuit that can display five characters on a 7-segment
2   // display.
3   // inputs:  SW2-0 selects the letter to display. The characters are:
4   //     SW 2 1 0    Char
5   //     ----------------
6   //        0 0 0    'H'
7   //        0 0 1    'E'
8   //        0 1 0    'L'
9   //        0 1 1    'O'
10  //        1 0 0    ' ' Blank
11  //        1 0 1    ' ' Blank
12  //        1 1 0    ' ' Blank
13  //        1 1 1    ' ' Blank
14  //
15  // outputs: LEDR2-0 show the states of the switches
16  //          HEX0 displays the selected character
17  module pro3 (SW, LEDR, HEX0);
18     input [2:0] SW;            // toggle switches
19     output [2:0] LEDR;         // red LEDs
20     output [0:6] HEX0;         // 7-seg display
21
22     wire [2:0] C;
23
24     assign LEDR = SW;
25     assign C[2:0] = SW[2:0];
```

```
22     wire [2:0] C;
23
24     assign LEDR = SW;
25     assign C[2:0] = SW[2:0];
26
27     /*
28      *       0
29      *      ---
30      *     |   |
31      *    5|   |1
32      *     | 6 |
33      *      ---
34      *     |   |
35      *    4|   |2
36      *     |   |
37      *      ---
38      *       3
39      */
40     // the following equations describe HEX0[0-6] in cannonical SOP form
41     assign HEX0[0] = ~((~C[2] & ~C[1] & C[0]) | (~C[2] & C[1] & C[0]));
42     assign HEX0[1] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
43     assign HEX0[2] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
44     assign HEX0[3] = ~((~C[2] & ~C[1] & C[0]) | (~C[2] & C[1] & ~C[0]) |
45        (~C[2] & C[1] & C[0]));
46     assign HEX0[4] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]) |
47        (~C[2] & C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
48     assign HEX0[5] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]) |
49        (~C[2] & C[1] & ~C[0]) | (~C[2] & C[1] & C[0]));
50     assign HEX0[6] = ~((~C[2] & ~C[1] & ~C[0]) | (~C[2] & ~C[1] & C[0]));
51  endmodule
```

8-to-3 Bit Priority Encoder



Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

X = dont care

The priority encoders output corresponds to the currently active input which has the highest priority.
If input lines “D2“, “D3” and “D5” are applied simultaneously the output code would be for input “D5” (“101”) as this has the highest order out of the 3 inputs.

Outputs are calculated as:

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

X = dont care

Output Q_0

$$Q_0 = \Sigma(1, 3, 5, 7)$$
$$= \Sigma\left(\bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4\bar{D}_3\bar{D}_2D_1 + \bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4D_3 + \bar{D}_7\bar{D}_6D_5 + D_7\right)$$
$$= \Sigma\left(\bar{D}_6\bar{D}_4\bar{D}_2D_1 + \bar{D}_6\bar{D}_4D_3 + \bar{D}_6D_5 + D_7\right)$$
$$= \Sigma\left(\bar{D}_6\left(\bar{D}_4\bar{D}_2D_1 + \bar{D}_4D_3 + D_5\right) + D_7\right)$$

Output Q_1

$$Q_1 = \Sigma(2, 3, 6, 7)$$
$$= \Sigma\left(\bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4\bar{D}_3D_2 + \bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4D_3 + \bar{D}_7D_6 + D_7\right)$$
$$= \Sigma\left(\bar{D}_5\bar{D}_4D_2 + \bar{D}_5\bar{D}_4D_3 + D_6 + D_7\right)$$
$$= \Sigma\left(\bar{D}_5\bar{D}_4(D_2 + D_3) + D_6 + D_7\right)$$

Output Q_2

$$Q_2 = \Sigma(4, 5, 6, 7)$$
$$= \Sigma\left(\bar{D}_7\bar{D}_6\bar{D}_5D_4 + \bar{D}_7\bar{D}_6D_5 + \bar{D}_7D_6 + D_7\right)$$
$$= \Sigma(D_4 + D_5 + D_6 + D_7)$$

$$Q_0 = \Sigma\left(\bar{D}_6\left(\bar{D}_4\bar{D}_2D_1 + \bar{D}_4D_3 + D_5\right) + D_7\right)$$
$$Q_1 = \Sigma\left(\bar{D}_5\bar{D}_4(D_2 + D_3) + D_6 + D_7\right)$$
$$Q_2 = \Sigma(D_4 + D_5 + D_6 + D_7)$$

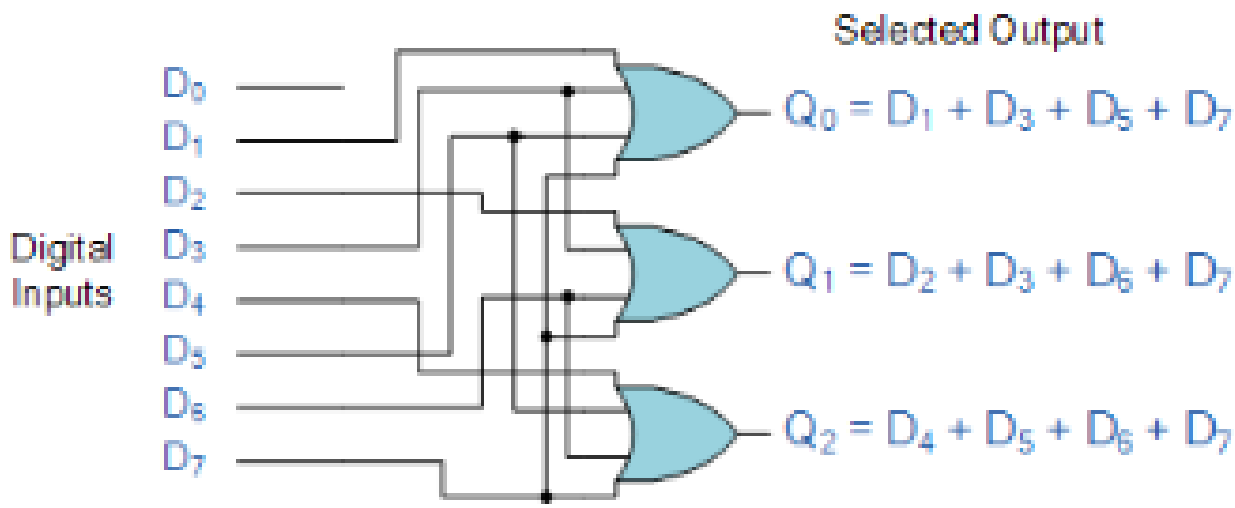
8-to-3 Bit Priority Encoder

$$Q_0 = \sum\left(\overline{D}_6\left(\overline{D}_4\overline{D}_2D_1 + \overline{D}_4D_3 + D_5\right) + D_7\right)$$

$$Q_1 = \sum\left(\overline{D}_5\overline{D}_4\left(D_2 + D_3\right) + D_6 + D_7\right)$$

$$Q_2 = \sum\left(D_4 + D_5 + D_6 + D_7\right)$$

These zero inputs would be ignored allowing the implementation of the final Boolean expression for the outputs of the 8-to-3 priority encoder.



8-to-3 Bit Priority Encoder in other modeling

```
// Priority encoder
module encode (A, valid, Y);
input  [7:0] A;                  // 8-bit input vector
output [2:0] Y;                  // 3-bit encoded output
output valid;                    // Asserted when an input is not all 0's
reg    [2:0] Y;                  // target of assignment
reg    valid;

  always @(A) begin
    valid = 1;
    casex (A)
      8'bXXXXXXX1: Y = 0;
      8'bXXXXXX10: Y = 1;
      8'bXXXXX100: Y = 2;
      8'bXXXX1000: Y = 3;
      8'bXXX10000: Y = 4;
      8'bXX100000: Y = 5;
      8'bX1000000: Y = 6;
      8'b10000000: Y = 7;
      default:  begin
         valid = 0;
         Y = 3'bX;   // Don't care when input is all 0's
      end
    endcase
  end
endmodule
```

Priority Encoder using if -else

```
Module
pri_encoder ( a, i,
en );
 output reg [2:0] a ;
 input  en ;
 input [7:0] i ;

 always @ (en or i)
 begin
   a = 0;
  if (en) begin
    if (i[1] == 1)
       a = 1;
    else if (i[2] == 1)
       a = 2;
    else if (i[3] == 1)
              a = 3;
```

```
else if (i[4] == 1)
       a = 4;
    else if (i[5] == 1)
       a = 5;
    else if(i[6] ==1)
       a = 6;
    else if(i[7]==1)
       a=7;
  end
end
endmodule
```

8-to-3 Bit Priority Encoder in yet another modeling

```
module p_encoder (D, Q);
    input [7:0] D;
    output [2:0] Q;

    assign Q = (D[7]==1)? 3'b111:
                (D[6]==1) ? 3'b110:
                (D[5]==1) ? 3'b101:
                (D[4]==1) ? 3'b100:
                (D[3]==1) ? 3'b011:
                (D[2]==1) ? 3'b010:
                (D[1]==1) ? 3'b001:
                (D[0]==1) ? 3'b000;
endmodule
```

This modeling uses conditional operator.

(_____) ? A : C

You can do using “ if, else “, too.

Revision of Verilog: if-else Statements

if statements allows the tool to decide, depending on the conditions specified, a statement is to be executed or not.

```
if( condition )
    statement;
```

```
if( reset )
   counter = 0;
else
   counter = counter + 1;
```

If there are more than one statements within an if block, we can combine them using begin -- end. We can also nest if-else statements as in these examples

```
if( reset )
begin
   counter <= 0;
   over_flow <= 0;
end
else if ( counter == 15 )
begin
   counter <= 0;
   over_flow <= 1;
end
else
begin
   counter <= counter + 1;
   over_flow <= 0;
end
```

Case statement

```
case (case_expression)
  case_item1   :  case_item_statement1;
  case_item2   :  begin
                  case_item_statement2a;
                  case_item_statement2b;
                  case_item_statement2c;
                  end
  case_item3   :  case_item_statement3;
  default      :  case_item_statement5;
endcase
```

Never foreget Default !

Example

```
case (SEL2bit)
    00:SELO <= IN_A;
    01:SELO <= IN_B;
   default :
       SELO <= IN_C;
endcase
```

```
module MUX4 (A, S, F) ;
    input        [3:0]  A ;
    input    [1:0]  S ;
    output          F ;
    reg      F ;
        always @( A or S ) begin
          case S
              2'b00 : F = A [0] ;
              2'b01 : F = A [1] ;
              2'b10 : F = A [2] ;
              default : F = A [3] ;
        endcase
    end
endmodule
```

Case statement

Compare !

```
module MUX4 (A, S, F) ;
    input    [3:0]  A ;
    input    [1:0]  S ;
    output          F ;
    reg      F ;
      always @( A or S ) begin
          if ( S ==2'b00 )
              F = A [0] ;
          else if ( S ==2'b01 )
              F = A [1] ;
          else if ( S ==2'b10 )
              F = A [2] ;
          else
              F = A [3] ;
          end
endmodule
```

If statement

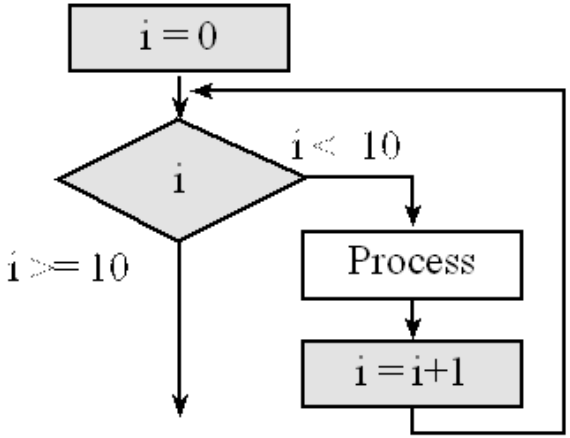
for - loop

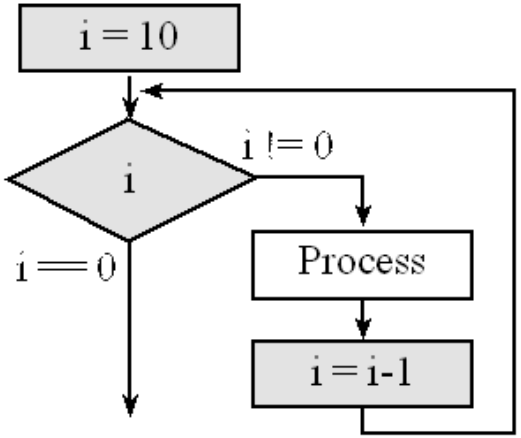
The <init> code is executed once.
The <test> code returns a true/false and is tested before each iteration.
The <body> and <end> codes are executed each iteration.

example

```
for (i = 0 ; i < 4 ; i = i + 1)begin
DATA_B[i] = DATA_A[i];
end
```

```
for(<init>; <test>; <end>)
   <body>;
```





Procedural assignment

begin ~ end

```
always @( posedge CK or posedge RES ) begin
    if ( RES==1'b1 ) begin
        a_reg <= 1'b0;
        b_reg <= 1'b0;
    end
    else if ( ENBL==1'b1 ) begin
        a_reg <= a_in;
        b_reg <= b_in;
    end
end
```

Processed in the order of statements in code.

Blocking and non-blocking assignments

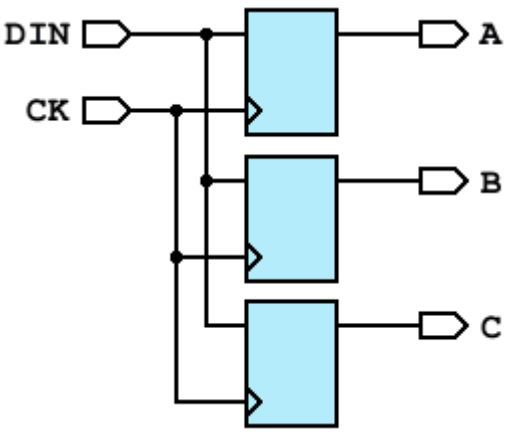
Blocking (operation Similar to S/W)

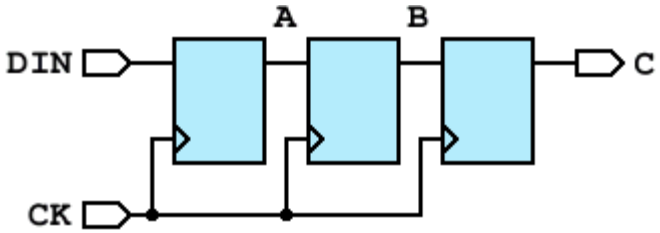
```
always @( posedge CK )
begin
    A = DIN;
    B = A;
    C = B;
end
```

Non-blocking

```
always @( posedge CK )
begin
    A <= DIN;
    B <= A;
    C <= B;
end
```

After compilation…





Strongly recommend using <= instead of =.

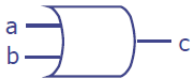
Assignment statement: simple combinational logic

- continuously operating (according to the change of inputs) logic
- <u>for wire type signals only → do not keep the value</u>

AND

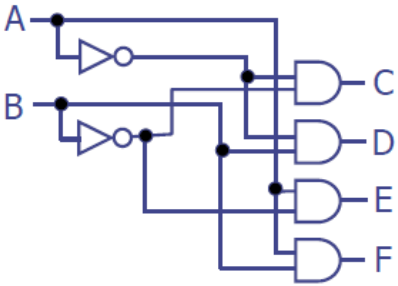

```
//AND
module test(a,b,c);
input  a,b;
output  c;

assign  c = a & b;

endmodule
```

OR




```
//OR
module test(a,b,c);
input  a,b;
output  c;

assign c = a | b;

endmodule
```

```
//Test回路
module test(A,B,C,D,E,F);
input   A,B;
output   C,D,E,F;

  assign C = ~A & ~B;
  assign D = ~A & B;
  assign E = A & ~B;
  assign F = A & B;

endmodule
```



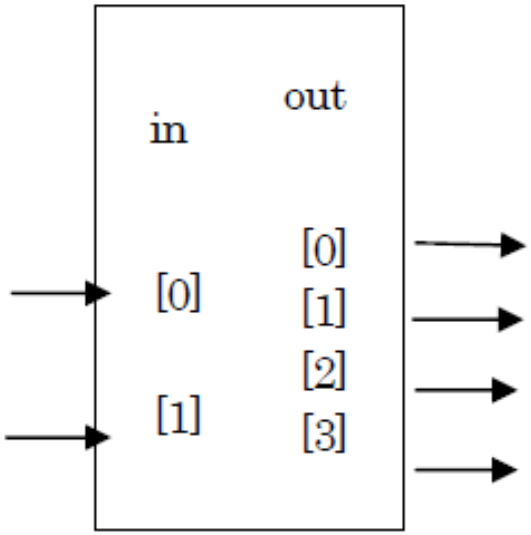
A	B	C	D	E	F
L	L	H	L	L	L
L	H	L	H	L	L
H	L	L	L	H	L
H	H	L	L	L	H

function statement:
for more complicated combinational logic

if statements and case statements can be used.

example: 2 to 4 decoder

```
module dec2to4 ( in, out ) ;
            input [1:0] in ;
            output [3:0] out ;
      function [3:0] dec;
      input [1:0] in;
      begin
            case ( in )
                0: dec = 4'b0001;
                1: dec = 4'b0010;
                2: dec = 4'b0100;
                3: dec = 4'b1000;
            endcase
      end
       endfunction
       assign out = dec ( in ) ;
endmodule
```



always statement: for sequential circuit

- Sequential logic triggered by clock, reset,,, uses always statement, for repeated operations in particular.
- In always blocks, reg are used. Assignment to wire is a grammatical error.
- Assignment to reg should use <= . （= is likely to lead to mistakes.）

```
//加算演算子による4ﾋﾞｯﾄ･ｶｳﾝﾀ
module counter(CLK,RESET,Q);
input CLK,RESET;
output [3:0] Q;
reg [3:0] Q;
always @ (posedge CLK or posedge RESET) begin
    if (RESET==1'b1)
Q <= 4'h0;
    else
Q <= Q + 4'h1;
    end
endmodule
```

always @（event）:

posedge:

At the risng edge of CLK or RESET,

"begin – end" is executed always. !

negedge for falling edge