INTRODUCTION TO HDL

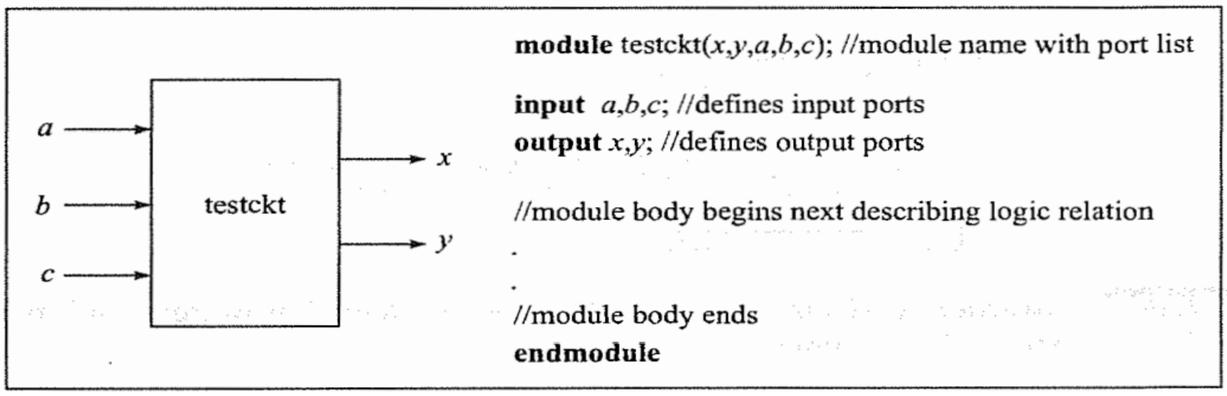
This is textual description of a digital circuit

Currently, there are two widely used HDLs-Verilog and VHDL (Very high-speed integrated circuit Hardware Description Language).

Verilog is considered simpler and is more popular.

General structure of Verilog language

To design any circuit that has say, three inputs a, b, c and two outputs say, x, y as shown in Fig. the corresponding Verilog code can be written as shown next.



```
module testckt(x,y,a,b,c); //module name with port list

input  a,b,c; //defines input ports
output x,y; //defines output ports

//module body begins next describing logic relation
.
.
//module body ends
endmodule
```

Verilog supports around 100 keywords; they are predefined lowercase identifiers. Example **module, endmodule, input, output, wire**, **and, or** and **not.** Identifiers in Verilog are case sensitive, begin with a letter or underscore and can be of any length. Consists of Alphabets[a-zA-Z], digits[0-9], dollar signs ($), and the underscore (_) symbol. Verilog is case sensitive,

A module has name **testckt**, which can be any valid identifier) followed by list of input output ports. The symbol '//' is a single line comment, / *……..*/ multiline comment.

Module is a fundamental descriptive unit in the Verilog language. It describes the logic within the black box. It is declared by the keyword **module** and must always be terminated by the keyword **endmodule.**

HDL Models of Combinational Logic Circuits

1. **Gate Level Modelling** – uses **gates**
2. **Data Flow Modelling** – uses keyword **assign.**
3. **Behavioural Modelling** – uses **always.**

Gate Level Modelling – uses predefined and user defined primitive gates

Verilog supports predefined gate level primitives such as **and, or, not, nand, nor, xor, xnor** etc.

The syntax for 2 input OR gate it is as shown below,

or (output, input 1, input 2)

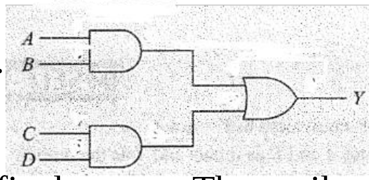
For   NOT gate,   **not** (output, input)

Note that, Verilog can take up to **12** inputs for logic gates.

Example 1: Two-input OR gate

module or_gate (Y, A, B);
input A, B;   *//defines two input port*
output Y;   *// defines one output port*
or g1(Y,A,B);   /*Gate *declaration with predefined keyword or representing logic OR, g1 is optional userdefined* gate identifier. */
endmodule

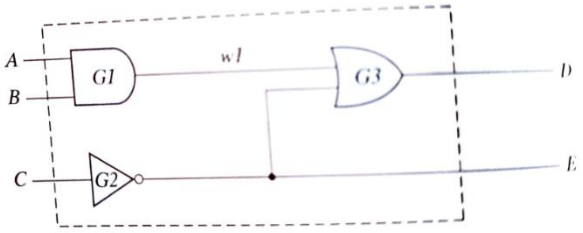
Example 2:



Let us now look at description of a logic circuit shown in Fig. that has 4 inputs A, B, C and D and 1 output Y.
The inputs are fed to two 2-input AND gate.
AND gate outputs are fed to a 2-input OR gate to generate final output. The verilog code for this is given below. Note that, we define two intermediate variables W1 and W2 representing two AND gate outputs through keyword **wire.** Wire represents a physical wire in a circuit, used for internal connection.

module simpleckt1(Y, A,B,C,D);
input A,B,C,D;
output Y;
wire W1,W2;   //Internal connection
and Gl(W1,A,B);   //g1 represents upper AND gate
and G2(W2,C,D);   *// g2 represents lower AND gate*
or G3(Y, W1, W2); // *g3 represents the OR gate*
endmodule

Example 3:



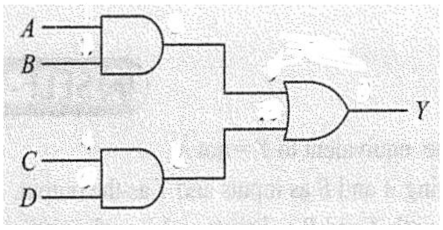
module simpleckt2(D,E,A,B,C);
    input A,B,C;
    output D,E;
    wire w1;   //Internal connection
    and G1(w1,A,B);
    not G2(E,C);
    or G3(D, w1,E);
endmodule

Data Flow model - uses assign and operators

Example 1: Two-input OR gate

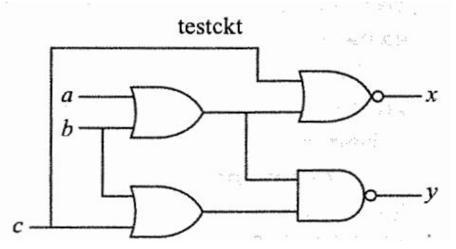
module or_gate (Y, A, B);
input A, B;
output Y;
assign Y=A|B;
endmodule

Example 2:



module andor(Y,A,B,C,D);
    input A,B,C,D;
    output Y;
    assign Y=(A&B) | (C&D);
endmodule

Example 3:



module testckt(x,y,a,b,c);
    input a,b,c;
    output x,y;
    assign x=~((a|b) | c);   //NOR thru NOT-OR
    assign y=~((a|b) & (b|c)); //NAND by NOT AND
endmodule

Example 4:

$Y=\overline{C}+\overline{A}\,\overline{D}+A\overline{B}\,\overline{D}$

```
module andor(Y,A,B,C,D);
        input A,B,C,D;
        output Y;
        assign Y=~C | (~A&~D)|(A&~B&~D);
endmodule
```

Example 5: Full adder

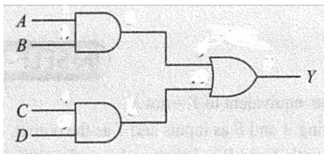
```
module fadder(a,b,c,sum,carry);
            input a,b,c;
            output sum,carry;
            assign sum=(a^b^c);
            assign carry=(a&b)|(b&c)|(a&c);
endmodule
```

Example 6: 2:1 Mux **$Y=D_0S^1+D_1S$**

```
module mux2to1(S,D0,D1,Y);
        input S,D0,D1;
        output Y;
        assign Y=(S^1&D0) | (S&D1);
endmodule
```

Behavioural Model - statements are executed sequentially, It is ideally suited to describe a sequential logic circuit. However, it is also possible to describe combinational circuits.
It always uses **always** keyword followed by a sensitivity list. It uses **reg** in place of **wire**.

Example:1



From the diagram Y = AB + CD,
i.e. Y = 1
if AB = 11
or
if CD = 11,
otherwise Y = 0.

We can use if ... else if ... else construct to describe this circuit.

```
module andor(A,B,C,D,Y);
input A,B,C,D;
output Y;
reg Y;                    //Y is o/p after procedural(sequential) assignment
always@(A or B or C or D)    // A,B,C,D are form sensitivity list, or is keyword
if((A==1)&&(B==1))
     Y=1;
else if ((C==1)&&(D==1))
     Y=1;
else
     Y=0;  // for all other combinations of A,B,C,D
endmodule
```

Example 2 : Full Adder

```
module fadder(a,b,c,sum,carry);
input a,b,c;
output sum,carry;
reg sum,carry;
always @(a or b or c)
      sum=a^b^c;
      carry=(a&b)|(b&c)|(a&c);
endmodule
```

Example 3: **2:1 Mux** $Y=D_0S^1+D_1S$

```
module mux2to1(S,D0,D1,Y);
input S,D0,D1;
output Y;
reg Y;
always@(S or D0 or D1)
if (S==1)
         Y=D1;
else
         Y=D0;
endmodule
```

Or

```
module mux2tol(S,D0,Dl,Y);
input S,D0,Dl;
output Y;
reg Y;
always@ (S or D0 or Dl)
     case (S)
             0 : Y=D0;
             1 : Y=Dl;
     endcase
endmodle
```

Example 4: **4:1 Mux**

```
module mux4tol(S1,S0,D0,D1,D2,D3,Y);
input S1, S0, D0, Dl, D2,D3;
output Y;
reg Y;
always @ (S1 or S0 or D0 or D1 or D2 or D3)
   case ({S1,S0})     //Concatenation of S1 or S0 , two binary digit can generate ,
                      //4 different values 0,1,2,and 3
        0 : Y=D0;     // 00, 01,10,11 respectively.
        1 : Y=D1;
        2 : Y=D2;
        3 : Y=D3;
   endcase
endmodule
```

BUS or Vector Representation in HDL

BUS or vector representation in HDL description through design of a 1 to 4 demultiplexer that can also serve as a 2 to 4 decoder. The data input of former is treated as enable input of later. We consider S as a select input defined by two binary digits S[l] and S[O]. Output Y is 4 bit long, one of which goes high for a particular combination of select inputs if data (enable) input is high. The Verilog code for this demultiplexer/decoder is given below:

Example 1: **1 : 4 Demux**

```
module demuxlto4(S,D,Y);
input   [1:0] S;
input   D;
output [3 : 0] Y;
reg     [3:0] Y;
always @ (S or D)
 case ( {D, S})                 //Concatenation of D and S to give 3 bits, D is MSB
     3'b100 : Y= 4'b0001; //Binary Representation, if D=1, S=00, Y=0001
     3'b101 : Y= 4'b0010;  // if D=1, S=01, Y=0010
     3'b110 : Y= 4'b0100;  // if D=1, S=01, Y=0100
     3'b111 : Y= 4'b1000;  // if D=1, S=01, Y=1100
   default : Y= 4'b0000;   //for D=0, then    Y=0000
  endcase
endmodule
```

Example 2: **8:1 Mux**

```
module mux8(S,D,Y);
input   [2:0] S;
input   [7:0] D;
output  Y;
reg      Y;
always @ (S or D)
          if(s==0)
                    Y=D[0];
          else if(s==1)
                    Y=D[1];
          else if(s==2)
                    Y=D[2];
          else if(s==3)
                    Y=D[3];
          else if(s==4)
                    Y=D[4];
          else if(s==5)
                    Y=D[5];
          else if(s==6)
                    Y=D[6];
          else if(s==7)
                    Y=D[7];
endmodule
```

1:8 Demultiplexer

```
module Demultiplexer(in, s0,s1,s2,d0,d1,d2,d3,d4,d5,d6,d7);
input in,s0,s1,s2;
output d0,d1,d2,d3,d4,d5,d6,d7;
assign d0=(in & ~s2 & ~s1 &~s0),
            d1=(in & ~s2 & ~s1 &s0),
            d2=(in & ~s2 & s1 &~s0),
            d3=(in & ~s2 & s1 &s0),
            d4=(in & s2 & ~s1 &~s0),
            d5=(in & s2 & ~s1 &s0),
            d6=(in & s2 & s1 &~s0),
            d7=(in & s2 & s1 &s0);
endmodule
```

3:8 Decoder

```
module Decoder(a,b,c,d0,d1,d2,d3,d4,d5,d6,d7);
input a,b,c;
output d0,d1,d2,d3,d4,d5,d6,d7;
assign d0=(~a&~b&~c),
           d1=(~a&~b&c),
           d2=(~a&b&~c),
           d3=(~a&b&c),
           d4=(a&~b&~c),
           d5=(a&~b&c),
           d6=(a&b&~c),
           d7=(a&b&c);
  endmodule
```

or

```
module decoder_3_8(
   output reg [7:0] y,
    input [2:0] i
    );
always@(*)
begin
    case(i)
        3'b000: y= 8'b0000_0001;
        3'b001: y= 8'b0000_0010;
        3'b010: y= 8'b0000_0100;
        3'b011: y= 8'b0000_1000;
        3'b100: y= 8'b0001_0000;
        3'b101: y= 8'b0010_0000;
        3'b110: y= 8'b0100_0000;
        3'b111: y= 8'b1000_0000;
  default:  y = 8'bxxxx_xxxx;
```

```
    endcase
end
endmodule
```

D Latch

```
module dlatch(D, En, Q);
input  D, En;
output  Q;
reg       Q;
always @ (En or D)
if(En)
    Q=D;
endmodule
```

D FlipFlop

```
          module dlatch(D, clk, Q);
          input  D, clk;
output  Q;
reg       Q;
          always  @  (posedge  clk)        or       always
@(negedge clk)

          if(En)
     Q=D;
endmodule
```

SR Latch

```
module srlatch(S,R, En, Q);
input  S,R, En;
output  Q;
reg       Q;
always @ (En or S or R)
          if(En)
             Q=S | (~R&Q);
endmodule
```

PIPO Shift register

```
module pipo(clk, pi,po);
input  clk;
input [3:0]pi;
output [3:0]po;
reg      [3:0]po;
always @ (posedge clk)
begin
          po=pi;
end
endmodule
```

3bit Up counter

```
module upcounter(clk, clr,Q);
input  clk, clr;
output [2:0]Q;
reg     [2:0] Q;
always @ (posedge clk)
        if(~clr)
                  Q<=3`b0;
           else
                Q=Q+1;
```

```
endmodule
```

3bit Down counter

```
module upcounter(clk, clr,Q);
input  clk, clr;
output [2:0]Q;
reg     [2:0] Q;
always @ (posedge clk)
        if(~clr)
                  Q<=3`b1;
           else
                Q=Q-1;
endmodule
```