

# Verilog

- A **Hardware Description Language (HDL )** is a machine readable and human readable language for describing hardware.
- **Verilog** and **VHDL** are HDLs.
- HDLs are used to (i) simulate and (ii) synthesize a designed digital circuit or system.
- In order to test a circuit before its implementation, it can be **simulated** by a simulator.
- A digital circuit HDL description can be **synthesized** as an integrated circuit by a synthesis tool.
- Synthesis tools are used to synthesize the design as an Application Specific Integrated Circuit (ASIC) or as a **Field Programmable Gate Array (FPGA)**.
- In CS223 laboratory, Verilog HDL will be used to describe the digital systems.
- The system designed can be simulated and implemented on an FPGA.

Verilog description of digital circuit starts with the keyword **module** and ends with keyword **endmodule**. Module statement specifies the name of the module and its input ports and output ports.

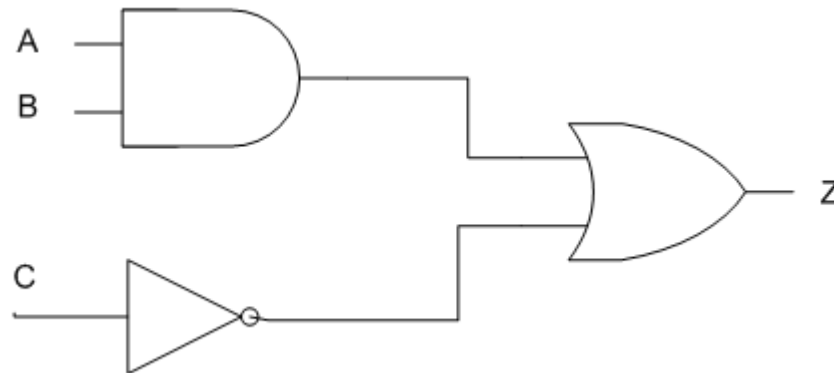
```
module mydesign (input A, B, C,  
                output Z);  
    ...  
    ...  
    ...  
endmodule
```

or

```
module mydesign (A, B, C, Z);  
    input A, B, C;  
    output Z;  
    ...  
    ...  
    ...  
endmodule
```

input ports

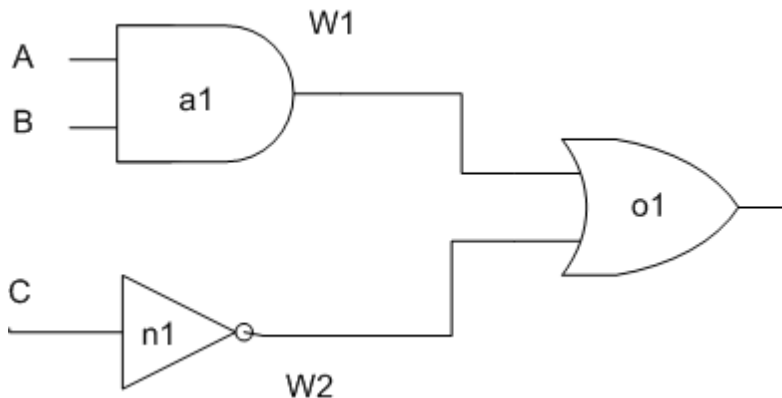
output port



The circuit can be described in several forms:

**1. Structural Description:** Specify each hardware component and connections between components. Logic gates (AND, OR, NOT) are primitive components. These primitives can be instantiated where the first parameter is always the output and input(s) follow.

and(output, input1, input2);  
or(output, input1, input2);  
not(output, input);



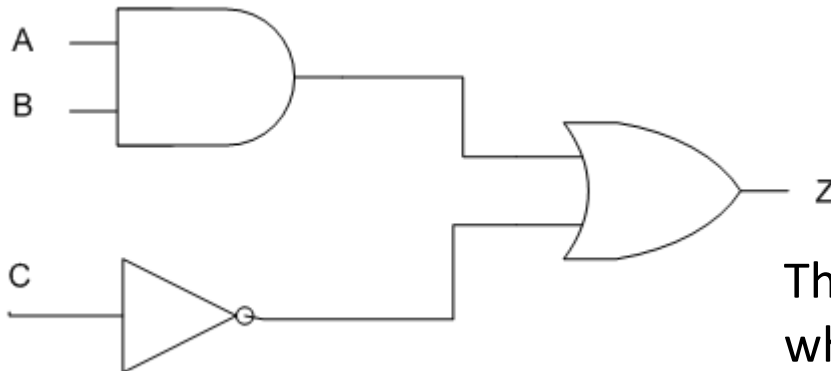
```
module mycircuit (input A, B, C,  
                  output Z );  
    wire W1, W2;           //wires  
  
    and a1(W1, A, B);       // W1 = A & B  
    not n1(W2, C);          // W2 = ~C  
    or o1(Z, W1, W2);      // Z = W1 | W2  
endmodule
```

**2. Data Flow Description:** The logic equation of the circuit is specified using Verilog logical operators.

& and  
| or  
~ not  
^ xor

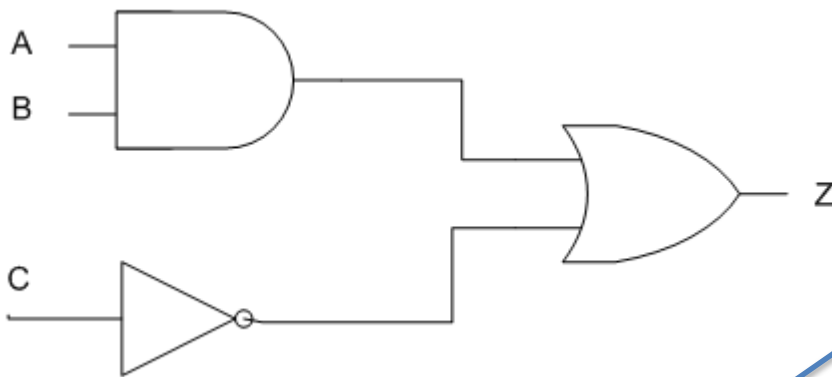
Continuous assignment

```
module mycircuit (input A, B, C,  
                  output Z );  
    assign Z = (A & B) | ~C;  
endmodule
```



The value of output port Z is calculated whenever the values (A, B, C) at input ports change.

**3. Behavioral Description:** An always procedure can be used to describe the behaviour of the circuit. The keyword **always** specifies a loop which will be executed whenever the variables in the event control part of the always statement change.

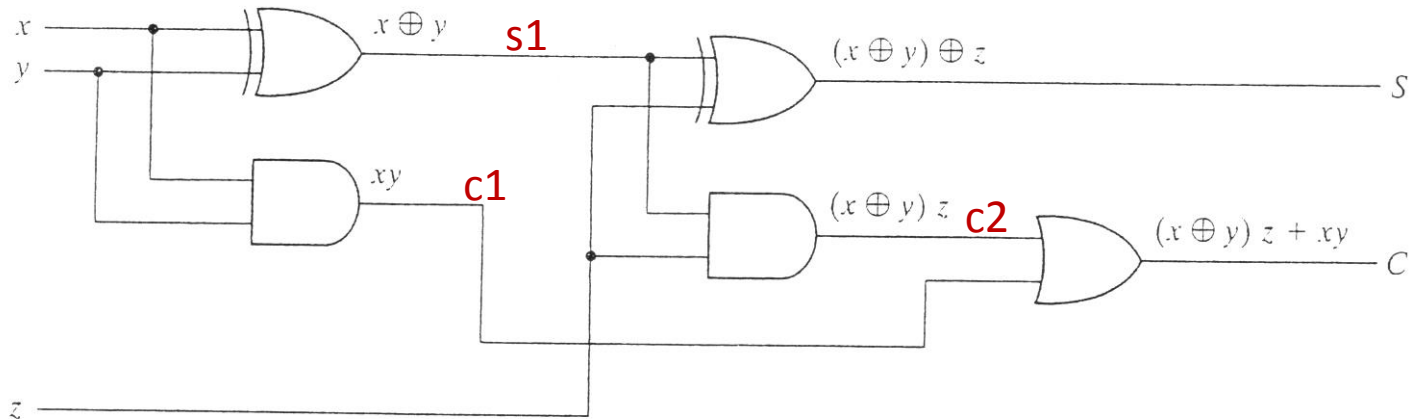


A variable assigned a value inside an always procedure must be of type **reg**.

```
module mycircuit (input A, B, C,
                  output Z );
    reg Z;
    always @(A, B, C)
    begin
        Z <= (A & B) | ~C;
    end
endmodule
```

The expression @(A, B, C) is called the event control part of the procedure. A change of value of a listed item is considered an event and the list is called **sensitivity list**. Whenever an event occurs the statements between begin and end are executed.

## Example: Description of a Full Adder circuit



```
module full_adder (x, y, z, S, C);  
  input x, y, z;  
  output S, C;  
  
  assign S = x ^ y ^ z;  
  assign C = (x&y) | (x^y)&z;  
endmodule
```

*Data Flow Description*

```
module full_adder (x, y, z, S, C);  
  input x, y, z;  
  output S, C;  
  wire s1, c1, c2;  
  
  xor xor1(s1, x, y);  
  xor xor2(S, s1, z);  
  and and1(c1, x, y);  
  and and2(c2, s1, z);  
  or or1(C, c1, c2);  
endmodule
```

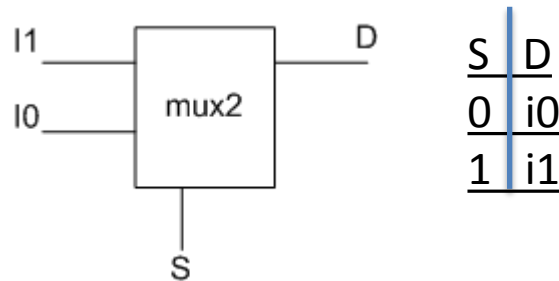
*Structural Description*

## **Example:** Behavioral description of a Full Adder circuit

```
module full_adder (x, y, z, S, C);  
  input x, y, z;  
  output S, C;  
  reg S, C;  
  
  always @(x, y, z)  
  begin  
    S <= x ^ y ^ z;  
    C <= x&y | x&z | y&z;  
  end  
endmodule
```

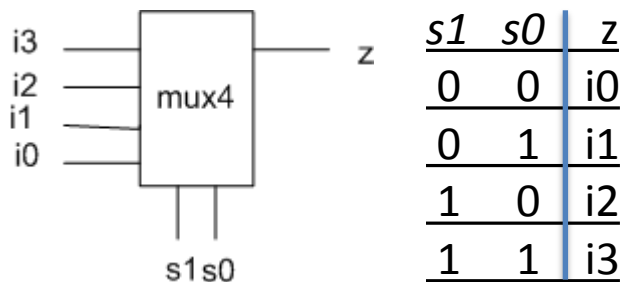
## Module Instantiations

and, or, inv are instantiations of the Verilog primitives and, or, inv.  
A verilog description can use previously defined module instances.



```
module mux2 (i1, i0, s, d);  
  input i1, i0, s;  
  output d;  
  
  assign d = ~s&i0 | s&i1;  
endmodule
```

mux2 description can be used to describe a mux4



```
// Description of mux4 (structural)  
module mux4 (i3, i2, i1, i0, s1, s0, z);  
  input i3, i2, i1, i0, s1, s0;  
  output z;  
  wire d1, d0;  
  
  mux2 m1(i3, i2, s0, d1); // instance1  
  mux2 m2(i1, i0, s0, d0); // instance2  
  mux2 m3(d1, d0, s1, z); // instance3  
endmodule
```

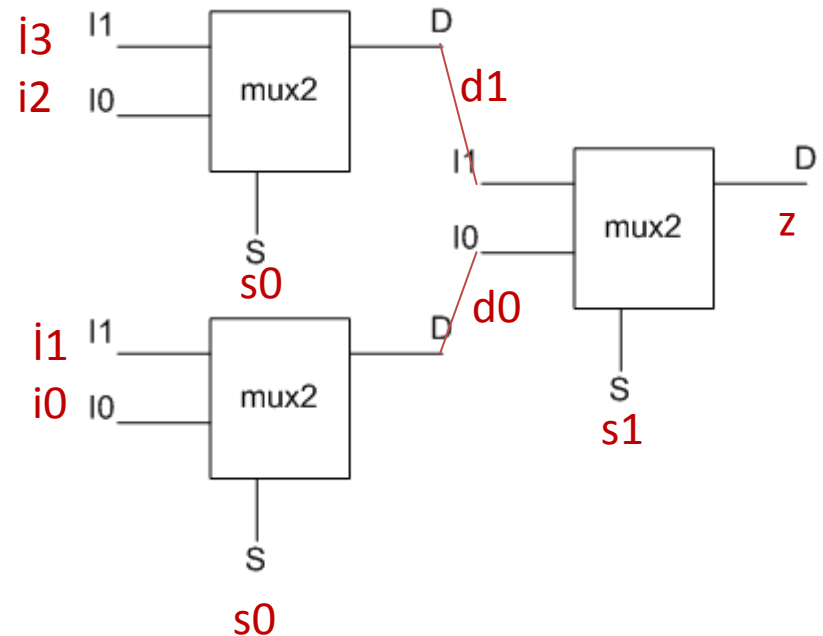


Structural description uses instances of already defined mux2 to define mux4. It is also possible to design mux2 by specifying each gate in the module. It will be a longer structural description. Data flow description of mux4 uses assign statements.

#### // Description of mux4 (Data flow)

```
module mux4 (i3, i2, i1, i0, s1, s0, z);
  input i3, i2, i1, i0, s1, s0;
  output z;
  wire d1, d0;

  assign d1 = ~s0 & i2 | s0 & i3;
  assign d0 = ~s0 & i0 | s0 & i1;
  assign z = ~s1 & d0 | s1 & d1;
endmodule
```



Behavioural Description of mux2 and mux4 are also possible

```
module mux2(i1, i0, s, d);  
  input i1, i0, s;  
  output d;  
  reg d;  
  
  always @(i1, i0, s)  
  begin  
    if(s==0) d <= i0;  
    else d <= i1;  
  end  
endmodule
```

```
module mux4(i3, i2, i1, i0, s1, s0, z);  
  input i3, i2, i1, i0, s1, s0;  
  output z;  
  reg z;  
  
  always @(i3, i2, i1, i0, s1, s0)  
  begin  
    if(s0==0)  
      if(s1==0) z <= i0;  
      else z <= i2;  
    else if(s1==0) z <= i1;  
    else z <= i3;  
  
  end  
endmodule
```

Simulation is a process to see how a module behaves. Simulation is performed by a tool known as a **simulator**. Simulator generates output values of a given module for given input values. The module to be simulated is called Unit Under Test (**UUT**) and the input values are called the **test vector**. The outputs may be printed or shown graphically as **waveforms**.

In order to test Verilog module the test vectors can be generated by another Verilog module called the **testbench**. The testbench is a Verilog module which will apply the test vectors sequentially to the UUT. At the end outputs of the UUT will be shown as waveforms or printed or both printed and shown as waveforms depending on the simulator tool used.

```
module testmux2;
```

```
// Inputs
```

```
reg i1; reg i0; reg s;
```

```
// Outputs
```

```
wire d;
```

```
// Instantiate the Unit Under Test (UUT)
```

```
mux2 uut (.i1(i1), .i0(i0), .s(s), .d(d) );
```

```
initial begin
```

```
    #100; // Wait
```

```
    // Initialize Inputs
```

```
    #10 s=0; i1= 0; i0=0;
```

```
    #10 s=0; i1=0;i0=1;
```

```
    #10 s=0; i1=1;i0=0;
```

```
    #10 s=0; i1=1;i0=1;
```

```
    #10 s=1; i1=0;i0=1;
```

```
    #10 s=1; i1=0;i0=1;
```

```
    #10 s=1; i1=1;i0=0;
```

```
    #10 s=1; i1=1;i0=1;
```

```
end
```

```
initial $monitor($time,,"s=%d,i1=%d,i0=%d,d=%d",s,i1,i0,d);
```

```
endmodule
```

## A testbench for mux2 and output of the simulator

### Printed output

Simulator is doing circuit initialization process. Finished circuit initialization process.

0 s=x,i1=x,i0=x,d=x

110 s=0,i1=0,i0=0,d=0

120 s=0,i1=0,i0=1,d=1

130 s=0,i1=1,i0=0,d=0

140 s=0,i1=1,i0=1,d=1

150 s=1,i1=0,i0=1,d=0

170 s=1,i1=1,i0=0,d=1

180 s=1,i1=1,i0=1,d=1

The testbench does not have input or outputs. i1, i0, s are variables which are connected to input ports of UUT. Since they are assigned values in the testbench they have to be reg type.

The connections of variables to ports of mux2 is shown as .i1(i1), .i0(i0), .s(s), .d(d)

Keyword **initial** specifies that initially (time=0) The statements between begin ... end are to be executed sequentially.  
# symbol indicates time delay. 100 units of time is for initialization of this simulator. Input values are applied to the ports of mux2 every 10 units of time until the all 8 different combinations are input.

\$monitor is for printing the inputs and the corresponding output values whenever inputs change.

```
module testmux2;
    // Inputs
    reg i1; reg i0; reg s;
    // Outputs
    wire dt;
    // Instantiate the Unit Under Test (UUT)
    mux2 uut (.i1(i1), .i0(i0), .s(s), .d(d) );
    initial begin
        #100; // Wait
        // Initialize Inputs
        #10 s = 0; i1 = 0; i0 = 0;
        #10 s=0; i1=0;i0=1;
        #10 s=0; i1=1;i0=0;
        #10 s=0; i1=1;i0=1;
        #10 s=1; i1=0;i0=1;
        #10 s=1; i1=0;i0=1;
        #10 s=1; i1=1;i0=0;
        #10 s=1; i1=1;i0=1;
    end
    initial $monitor($time, "s=%d, i1=%d, i0=%d, d=%d", s, i1, i0, d);
endmodule
```

```
module Mux4(input i3, i2, i1, i0, s1, s0,  
            output Z);
```

```
    reg Z  
    always @(s1,s0)  
    begin  
        if(s1==0 && s0==0)  
            z<=i0;  
        else if(s1==0 && s0==1)  
            z<=i1;  
        else if(s1==1 && s0==0)  
            z<=i2;  
        else if(s1==1 && s0==1)  
            z<=i3;  
        else  
            z<=i3;  
    end  
endmodule
```

What is wrong with this  
Mux4 description ?

```
module Mux4(input i3, i2, i1, i0, s1, s0,  
            output Z);
```

```
    reg Z  
    always @(s1,s0) ← Missing i3-i0 in sensitivity list  
    begin  
        if(s1==0 && s0==0)  
            z<=i0;  
        else if(s1==0 && s0==1)  
            z<=i1;  
        else if(s1==1 && s0==0)  
            z<=i2;  
        else if(s1==1 && s0==1)  
            z<=i3;  
        else  
            z<=i3;  
    end  
endmodule
```

```

module decode2x4(input i1, input i0,
                  output z3,output z2, output z1,output z0);
reg z3, z2, z1, z0;
always @(i1,i0)
begin
    if(i1==0 && i0==0)
    begin
        z3<=0; z2<=0;
        z1<=0; z0<=1;
    end
    else if(i1==0 && i0==1)
    begin
        z3<=0; z2<=0;
        z1<=1; z0<=0;
    end
    else if(i1==1 && i0==0)
    begin
        z3<=0; z2<=1;
        z1<=0; z0<=0;
    end
    else if(i1==1 && i0==1)
    begin
        z3<=1;
    end
endmodule

```

What is wrong with this  
decoder description ?




```

module decode2x4(input i1, input i0,
                  output z3,output z2, output z1,output z0);
reg z3, z2, z1, z0;
always @(i1,i0)
begin
    if(i1==0 && i0==0)
    begin
        z3<=0; z2<=0;
        z1<=0; z0<=1;
    end
    else if(i1==0 && i0==1)
    begin
        z3<=0; z2<=0;
        z1<=1; z0<=0;
    end
    else if(i1==1 && i0==0)
    begin
        z3<=0; z2<=1;
        z1<=0; z0<=0;
    end
    else if(i1==1 && i0==1)
    begin
        z3<=1;
    end
end
endmodule

```

Missing assignments to z2, z1, z0



## Constants in Verilog

4'b000 is a 4-bit constant number with all bits 0.

1'b1 is a one bit constant 2'b10 two bit constant

8'b00001010 denotes an 8 bit number which is decimal 10.

It can be specified as 8'd10 where d indicates following value is decimal constant expressed as an 8 bit constant.

12'h2AF specifies a 12 bit hexadecimal constant 0010 1010 1111

## Vectors in Verilog

A vector data type defines a collection bits when it is more convenient than declaring each bit separately.

V[3:0] defines a four bit vector. Each bit can be accessed by V[2] or V[3].

A range of bits may also be selected V[3:2] which selects leftmost 2 bits of V.

```

module dec3to8(input [2:0] a,
               output reg[7:0] y);
    always @*
    case (a)
        3'b000: y<= 8'b00000001;
        3'b001: y<= 8'b00000010;
        3'b010: y<= 8'b00000100;
        3'b011: y<= 8'b00001000;
        3'b100: y<= 8'b00010000;
        3'b101: y<= 8'b00100000;
        3'b110: y<= 8'b01000000;
        3'b111: y<= 8'b10000000;
    endcase
endmodule

```

```

module dec38test;
    // Inputs
    reg [2:0] a;
    reg [2:0] testvec;
    // Outputs
    wire [7:0] y;
    // Instantiate the Unit Under Test (UUT)
    dec3to8 uut (.a(a), .y(y) );
    initial begin
        // Initialize Inputs
        testvec[2:0]<= 3'b0;
        // Wait 100 ns for global reset to finish
        #100;
        repeat(8)
            begin
                a[2:0]<=testvec[2:0];
                testvec<=testvec+1;
                #10;
            end
        $stop;
    end
endmodule

```

Module instantiations

```

[module-name] [instance-name]
(
    . [port-name] ( [signal-name] ) ,
    . [port-name] ([signal-name]),
    ...

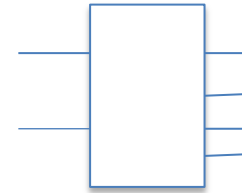
```

The first portion of the statement specifies which component

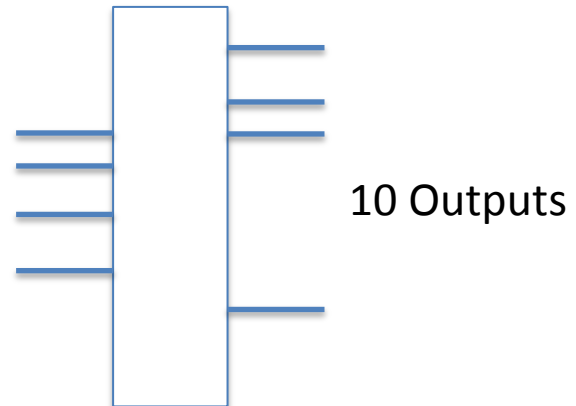
## Generalized Decoder Description

```
// a - binary input (n bits wide)
// b - one hot output (m bits wide)
module Dec(a, b) ;
parameter n=2 ;
parameter m=4 ;
input [n-1:0] a ;
output [m-1:0] b ;
wire [m-1:0] b = 1«a ;
endmodule
```

2-to-4 Decoder  
 $n=2, m=4$



4-to-10 Decoder  
 $n=4, m=10$



Output is determined by the value of  $a$  since  $a$  indicates number of shifts to the left.  
i.e if  $a=00$   $b[0]=1$ , if  $a=01$   $b[1]=1$ , if  $a=10$   $b[2]=1$  and if  $a=11$   $b[3]=1$

# Sequential Circuits

## SR Latch (cross coupled NOR gates) with asynchronous reset

```
module RSlatch(  
    input S,  
    input R,  
    input C,  
    input CLR,  
    output Q);  
  
    wire S1,R1,CLRC,QN;  
  
    not(CLRC,CLR);  
    and(S1,S,C,CLRC);  
    and(R1,R,C);  
    nor N1(QN,S1,Q);  
    nor N2(Q,R1,QN,CLR);  
endmodule
```

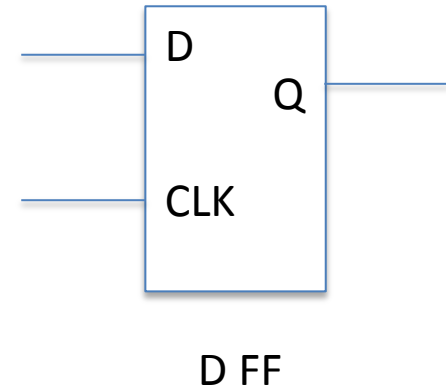
```
module Rslatch(  
    input S,R,C,clr,  
    output Q);  
  
    wire S1,R1,QN;  
  
    assign S1=S&C&CLR;  
    assign R1=R&C&|clr;  
    nor N1(QN,S1,Q);  
    nor N2(Q,R1,QN);  
endmodule
```

## D-Latch definition using latch definition with asynchronous reset

```
module Dlatch(  
    input d,  
    input clk,  
    input clr,  
    output q);  
  
    wire dn;  
  
    not n1(dn,d);  
    RSlatch latch1(d,dn,clk,clr,q);  
endmodule
```

## D Flip-flop Verilog Description

```
module DFF(Q,D,Clk);  
    input D,CLK;  
    output Q;  
    reg Q;  
  
    always @(posedge Clk)  
        Q<=D;  
endmodule
```



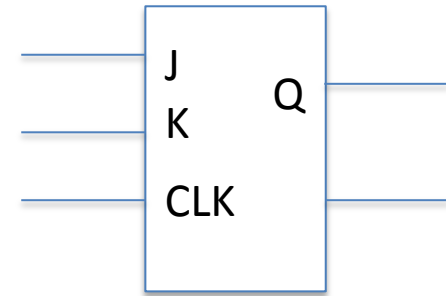


## D-Flip Flop with asynchronous reset (clr) input

```
module DFF(  
    input d,clk,clr,  
    output reg q);  
  
    always @(posedge clk, posedge clr)  
        if(clr) q<=1'b0;  
        else q<=d;  
endmodule
```

## JK Flip Flop Verilog Description

```
module JKFF(J,K,Clk,Q,Qc);  
    input J,K,Clk;  
    output Q,Qc;  
    reg Q;  
  
    assign Qc=~Q;  
    always @(posedge Clk)  
    case({J,K})  
        2'b00:Q<=Q;  
        2'b01:Q<=1'b0;  
        2'b10:Q<=1'b1;  
        2'b11:Q<=1~Q;  
    endcase  
endmodule
```



JK FF

## JK Flip-Flop from D Flip-flop Description

```
module JKFF(output reg Q,input J,K,Clk,rst);  
    wire JK;  
    assign JK=(J & ~Q)|(~K & Q);  
    //instantiate the DFF  
    DFF JK1 (Q,JK,CLK,rst);  
endmodule
```

```

module StateDiagramExample_1( output [1: 0] y_out,
    input x_in, clock, reset );
    reg [1: 0] state;
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
    always @ (posedge clock, negedge reset)
    if (reset == 0) state <= S0; // Initialize to state S0
    else case (state)
    S0: if (x_in) state <= S1; else state <= S0;
    S1: if (x_in) state <= S2; else state <= S3;
    S2: if (x_in) state <= S3; else state <= S2;
    S3: if (x_in) state <= S0; else state <= S3;
    endcase
    assign y_out = state; // Output of flip-flops
endmodule

```

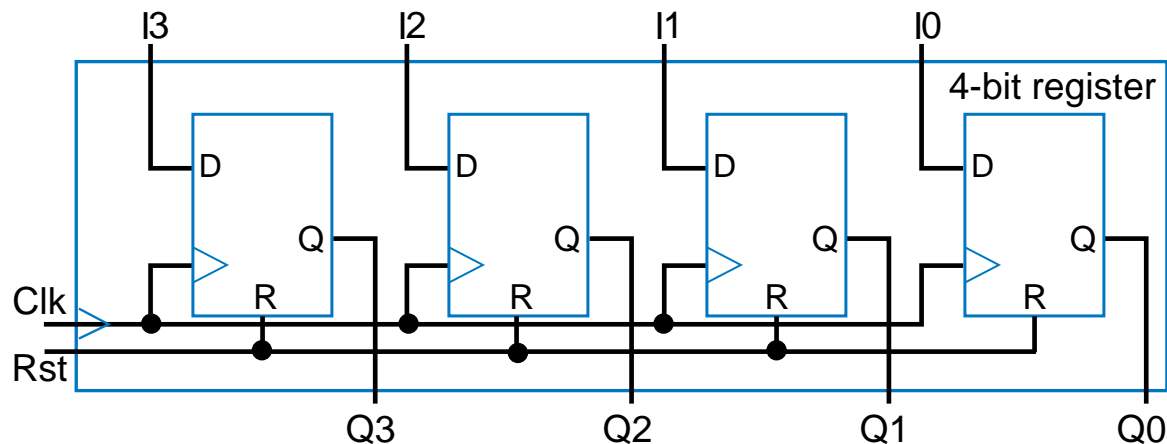
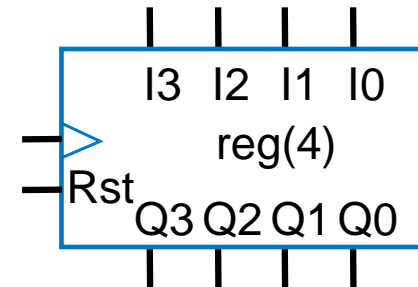
```

module DFFenable(
    input  clk, reset, en, d,
    output reg q);
    //signaldeclaration
    reg r, rnext;
    // body
    // D FF
    always @(posedge clk , posedge reset)
    i f (reset) r<= 1'b0;
    else r<= rnext;
    //next-statelogic
    always @*
    i f (en) rnext <= d;
    else rnext <= r;
    //outputlogic
    always @*
    q <= r;
endmodule

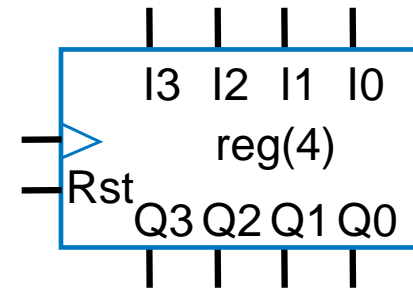
```

# Register Behavior

- Sequential circuits have storage
- **Register**: most common storage component
  - N-bit register stores N bits
  - Structure may consist of connected flip-flops



- Typically just describe register behaviorally
  - Declare output Q as reg variable to achieve storage
- Uses **vector** types
  - Collection of bits
    - More convenient than declaring separate bits like I3, I2, I1, I0
  - Vector's bits are numbered
    - Options: [0:3], [1:4], etc.
    - [3:0]
      - Most-significant bit is on left



```

`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

    input [3:0] I;
    output [3:0] Q;
    reg [3:0] Q;
    input Clk, Rst;

    always @(posedge Clk) begin
        if (Rst == 1 )
            Q <= 4'b0000;
        else
            Q <= I;
        end
    end
endmodule

```

Event control by the clock →

Synchronous reset →

Binary constant →

# Finite-State Machines (FSMs)

Finite-state machine (FSM) is a common model of sequential behavior

Example: If  $B=1$ , hold  $X=1$  for 3 clock cycles

Note: Transitions implicitly AND ed with rising clock edge

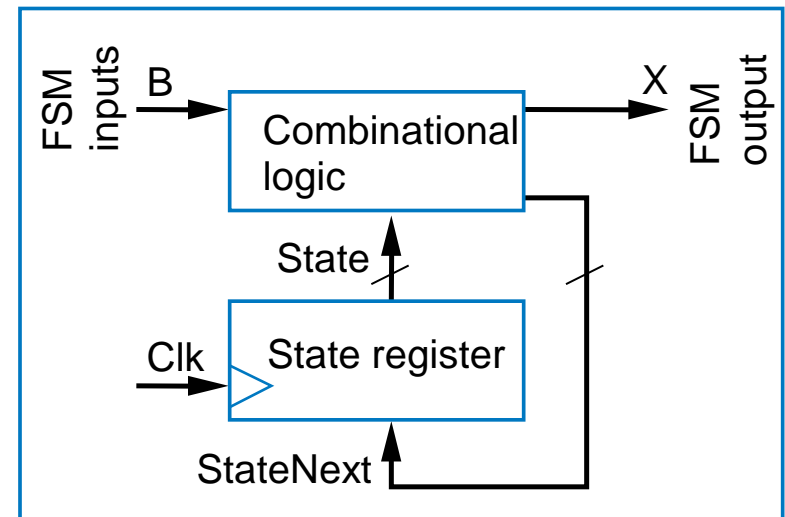
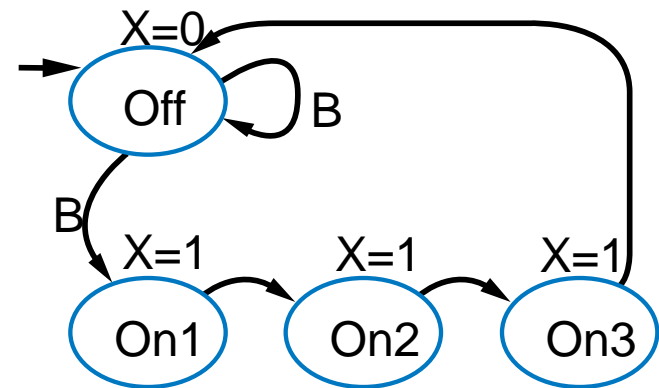
Implementation model has two parts:

State register

Combinational logic

HDL model will reflect those two parts

Inputs:  $B$ ; Outputs:  $X$





## Finite-State Machines (FSMs)—Sequential Behavior Modules with Multiple Procedures and Shared Variables

```

timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

    input B;
    output reg X;
    input Clk, Rst;

    parameter S_Off = 0, S_On1 = 1,
              S_On2 = 2, S_On3 = 3;

    reg [1:0] State, StateNext;

    // CombLogic
    always @(State, B) begin
        case (State)
            S_Off: begin
                X <= 0;
                if (B == 0)
                    StateNext <= S_Off;
                else
                    StateNext <= S_On1;
            end
            S_On1: begin
                X <= 1;
                StateNext <= S_On2;
            end
            S_On2: begin
                X <= 1;
                StateNext <= S_On3;
            end
            S_On3: begin
                X <= 1;
                StateNext <= S_Off;
            end
        endcase
    end

    // StateReg
    always @(posedge Clk) begin
        if (Rst == 1)
            State <= S_Off;
        else
            State <= StateNext;
        end
    end
endmodule

```

Modules has two procedures  
One procedure for  
combinational logic  
One procedure for state  
register  
But it's still a behavioral  
description

```
timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

    input B;
    output reg X;
    input Clk, Rst;

    parameter S_Off = 0, S_On1 = 1,
              S_On2 = 2, S_On3 = 3;

    reg [1:0] State, StateNext;

    // CombLogic
    always @(State, B) begin
        ...
    end

    // StateReg
    always @(posedge Clk) begin
        ...
    end
endmodule
```

## FSM testbench

First part of file (variable/net declarations,  
module instantiations) similar to before

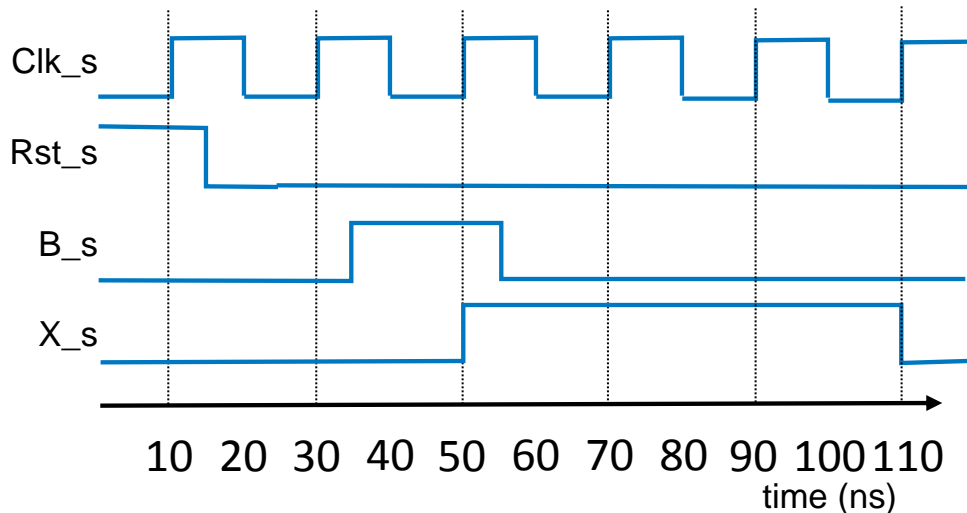
Vector Procedure

- Resets FSM

- Sets FSM's input values ("test vectors")

- Waits for specific clock cycles

We observe the resulting waveforms to  
determine if FSM behaves correctly



```
...  
// Clock Procedure  
always begin  
    Clk_s <= 0;  
    #10;  
    Clk_s <= 1;  
    #10;  
end // Note: Procedure repeats  
  
// Vector Procedure  
initial begin  
    Rst_s <= 1;  
    B_s <= 0;  
    @(posedge Clk_s);  
    #5 Rst_s <= 0;  
    @(posedge Clk_s);  
    #5 B_s <= 1;  
    @(posedge Clk_s);  
    #5 B_s <= 0;  
    @(posedge Clk_s);  
    @(posedge Clk_s);  
    @(posedge Clk_s);  
end  
endmodule
```

## Traffic Light Controller

(p125 Harris&Harris)

```
module TLC(input TA,TB,clk,rst,
output reg [1:0] LA,LB);
parameter s0=2'b00,s1=2'b01, s2=2'b10,
s3=2'b11;
parameter Green=2'b00, Red=2'b01,
Yellow=2'b10;
reg [1:0] state,nextstate;
//state register logic
always @(posedge clk,posedge rst)
if(rst==1) state <=s0;
else state<=nextstate;
// combinational circuit
always @(state,TA,TB)
case (state)
s0:if(TA) nextstate<=s0;
else nextstate<=s1;
s1:nextstate<=s2;
s2.if(TB) next state <=s2;
else nextstate<=s3;
```

```
s3: nextstate<=s0;
endcase
// output logic
always @(state)
case(state)
s0: begin LA<=Green; LB<= Red;end
s1 : begin LA<=Yellow; LB<= Red;end
s2 : begin LA<=Red; LB<= Green;end
s3 : begin LA<=Red; LB<= Yellow;end
endcase
endmodule
```

### Example 3

```
module FSM ( input clk, reset,  
             input a,b,  
             output y1);  
//symbolic state declaration  
localparam [1:0] s0= 2'b00, s1= 2'b01,  
s2= 2'b10;  
//signal declaration  
reg[1:0] statereg,statenext;  
//state register  
always @(posedge clk,posedge reset)  
if(reset)  
statereg<=s0;  
else  
statereg<=statenext;
```

```
//Combinational logic next-state logic  
always @*  
case(statereg)  
s0:if(a)  
    if(b)statenext=s2;  
    else  
        statenext=s1;  
else  
    statenext=s0;  
s1:if(a)  
    statenext=s0;  
else  
    statenext=s1;  
s2: statenext=s0;  
default:statenext=s0;  
endcase  
//output logic  
assign y1=(state-reg==s0) || (state-reg==s1);  
endmodule
```