

WEEK 8-2017

Verilog HDL



Hardware Description Language (HDL)

- A hardware description language (HDL) is a computer based language that describes the hardware of digital systems in textual form.
- It is similar to the ordinary programming language such as C, but specifically oriented for describing the structure and behavior of logic circuits.
- Unlike ordinary programming language where serial execution occurs, HDL employs extensive parallel operations.

Why HDL?

- To facilitate the design and testing of digital circuits.
- Designing and testing of practical digital circuits involves large number of circuit components, and computer aided tools need to be used.
- Computer based design tools are required to leverage the creativity and effort of a designer and reduce the risk of producing a flawed design.
- CAD design tools rely on HDL.
- Creates a common platform that both circuit designers and computers can understand.

When HDL?

- ***Design entry:*** the functionality of the circuit to be implemented can be described using HDL.
- ***Logic simulation:*** applying input stimulus and observing the outputs in order to make sure that the logic implementation or the behavior works as it is intended. HDL is written to create a test bench - the stimulus that tests the functionality of the design.
- ***Logic synthesis:*** physical components and their interconnections (called a net list) can be generated for a model of a digital system described in an HDL.

When HDL?

- ***Timing verification:*** to confirm that the digital system will operate at a specified speed.
- ***Fault simulation:*** to compare the ideal circuit behavior from the circuit behavior that contain process induced-flaws.

Standard HDLs

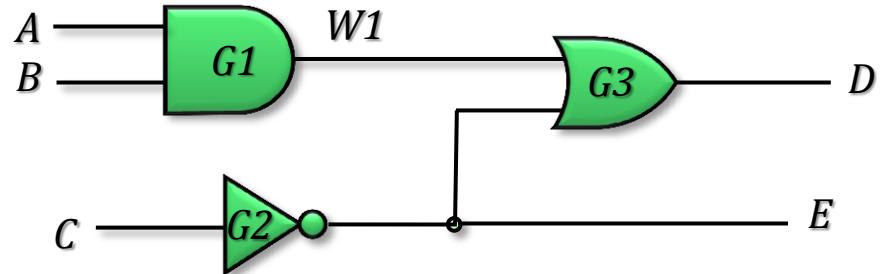
- **Companies that design integrated circuits use proprietary and public HDLS.**
- **In public domain, there are two standard HDLs that are supported by the IEEE standard: VHDL and Verilog HDL**
- **VHDL stands for Very High Speed Integrated Circuit (VHSIC) HDL**
- **Both HDLs are equally popular among the industries (50/50 share)**
- **In this course, Verilog HDL will be introduced.**

Module Declaration

- Verilog model is composed of text using *keywords*.
- Examples of keywords are
module, endmodule, input, output, wire, and, or, not ...
- Keywords are defined in lowercase.
- Any text between two forward slashes (//) and the end of the line is interpreted as a comment
- Multiline comments begin with /* and terminate with */
- Blank spaces are ignored, but they may not appear with in the text of a keyword, user-specified identifier, an operator, or the representation of a number.
- Verilog is a case sensitive. not is not the same as NOT.
- A module is fundamental descriptive unit in the Verilog language, and it is declared by the keyword **module** and must always be terminated by the keyword **endmodule**

Module Declaration

- Verilog HDL for the following combinational logic



//Verilog model of circuit shown on the side

```
module Simple_Circuit (A,B,C,D,E);
output D,E;
input A,B,C;
wire w1;

and G1(w1,A,B); // Optional gate instance name
not G2(E,C);
or G3(D,w1,E);
endmodule
```

Module Declaration

- Syntax for module

```
module identifier (port, port ...);
```

```
endmodule
```

- Identifiers are composed of alphanumeric characters and the underscore (_), and are case sensitive. It must start with an alphabetic character or an underscore, but *they can not start with a number*.
- The port list is enclosed in parentheses, and commas are used to separate elements of list. The parentheses should be terminated with a semicolon(;).
- The port list of a module are the inputs and outputs of the circuit

Module Declaration

- **Keywords for defining input and output ports are *input* and *output*.**
- **Syntax for output and input keywords**

input port, port, ...,port;

output port, port,...,port;

- **Internal connection is identified by a keyword *wire***
 - **Syntax for wire keyword**
- wire identifier, identifier ...identifier;**
-
- **The gates that form the circuit are specified by a list of predefined primitive gates. The Verilog HDL includes 12 basic gates as predefined primitives.**
 - **The primitive gates are identified by keywords: and, or, not, nor, nand, xor, xnor,buf.**

Module Declaration

- The elements of the primitive gate list are referred as instantiations of a gate. Each of which is referred as a gate instance.
- Syntax for primitive gates keywords

instance instancename(output, input, input ...);

eg.

and **G1(C,A,B);**

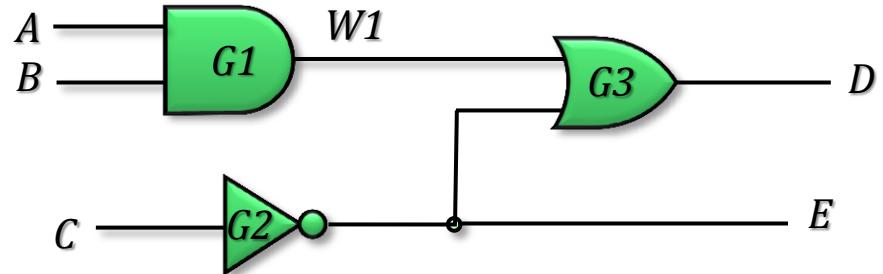


Declaration versus instantiation

- A Verilog module is declared. Its declaration specifies the input and output behavior of the hardware that it represents.
- Predefined primitives are not declared, but instantiated.
- Once a module has been declared, it may be instantiated within a design.

Module Declaration

- Verilog HDL for the following combinational logic



//Verilog model of circuit shown on the side

```
module Simple_Circuit_Prop_delay(A,B,C,D,E);
output D,E;
input A,B,C;
wire w1;

and #(30) G1(w1,A,B); // Optional gate instance name
not #(10) G2(E,C);
or  #(20) G3(D,w1,E);
endmodule
```

Module Declaration

- In Verilog, the propagation delay of a gate is specified in terms of time units and by the symbol #.
 - ‘ timescale 1ns/100ps
- time directive can be declared before the module. The first number is the unit for time delay and the second number specifies the precision for which the delays are rounded off.
- If no timescale is specified, a simulator may display dimensionless values or default to a certain time unit usually in 1ns.

Module Declaration

- **Identifiers can have multiple bit widths called vectors**
 - output [0:3] D; an output vector with four bits
 - wire [7:0] SUM; a wire vector sum with eight bits
- **The first (leftmost) listed in the square bracket is always the most significant bit of the vector.**
- **Each bit or group of bits in the vector can be addressed:**
 - D[2] specifies bit 2 of D.
 - SUM[2:0] specifies the three least significant bits of vector SUM.

// Structural description of 2-to-4 line decoder

```
module decoder_2x4_gates (A,B,E,D);
  output [0:3] D;
  input A,B,E;
  wire A_b, B_b, E_b;

  not G1(A_b,A), G2(B_b,B),G3(E_b,E);
  nand G4(D[0],B_b,E_b,A_b), G5(D[1],A_b,B,E_b),G6(D[2],B_b,A,E_b),
        G7(D[3],B,A,E_b);
endmodule
```

Test bench

- In order to simulate a circuit with an HDL, it is necessary to apply inputs to the circuit so that the simulator will generate an output response.
- An HDL description that provides the stimulus to a design is called a test bench.
- A test bench consists of a signal generator and an instantiation of the module that is to be verified.

```
module t_Simple_Circuit_Prop_delay;
    wire D,E;
    reg A,B,C;
    Simple_Circuit_Prop_delay M1(A,B,C,D,E);

    initial
        begin
            A=1'b0; B=1'b0;C=1'b0;
            #100 A=1'b1; B=1'b1; C=1'b1;
        end
    initial #200 $finish;

endmodule
```

Test bench

- A test bench has no input or output ports as it does not interact with its environment.
- Within the test bench, the inputs to the circuit are declared with keyword *reg* and the outputs are declared with keyword *wire*.
- initial statements are used to describe waveforms in a test bench. The set of statements to be executed is called a block statement and enclosed by keyword *begin ... end*
- The simulation is terminated by specifying \$finish system task.

Test bench

- Inputs specified by a three-bit truth table can be generated in initial block

```
initial
begin
D=3'b000;
repeat(7)
#10 D= D + 3'b001;
end
```

- A stimulus model (a test bench) has the following form:

```
module test_module_name;
// declare local reg and wire identifiers
// instantiate the design module under test
// specify a stopwatch using $finish to terminate the simulation
// generate stimulus using initial and always statements
// display the output response (text or graphics)
endmodule
```

Test bench

- The response to the stimulus will appear in text format as standard output and as waveforms.
- Numerical outputs are displayed by using Verilog system tasks. These are built-in system functions that are recognized by keywords that begin with symbol \$

\$display – display a one-time value of variables or strings with an end-of-line return

\$write – same as \$display, but without going to nextline

\$monitor – display variables whenever a value changes during a simulation

\$time - display the simulation time

\$finish – terminate the simulation

- The syntax for \$display, \$write and \$monitor is of the form

Taskname (format specification, argument list);

Eg. \$display ("%d %b %b", C, A,B); specifies the display of C in decimal and of A and B in binary.

\$display ("time = %d A=%b B=%b", \$time, A, B);

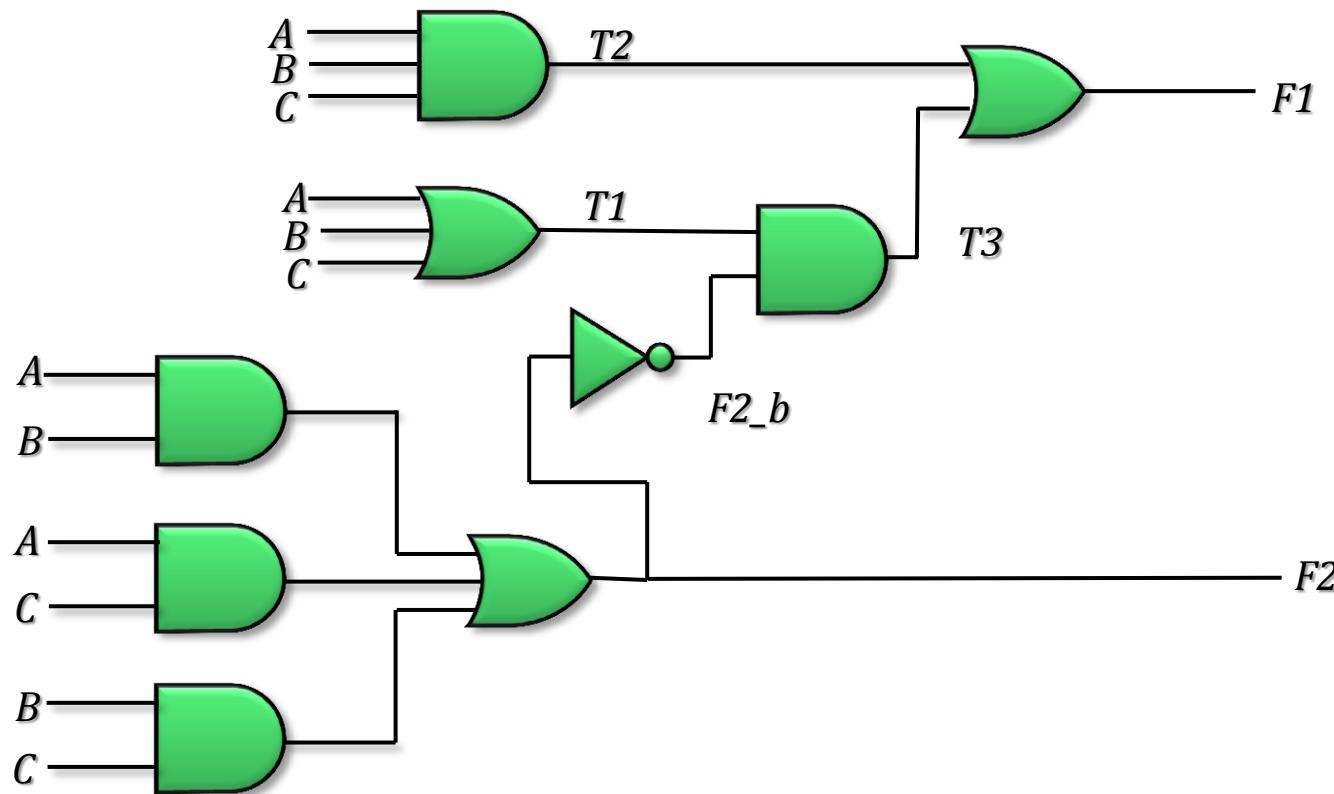
Test bench

- Two types of verification: functional and timing
- In functional verification – based on truth table
- In timing verification – based on waveforms and including effect of delays through the gates.
- An HDL test bench to produce the truth table of combinational circuit.

```
// Gate-level description of circuit
module Circuit_below(A,B,C,F1,F2);
output F1,F2;
input A,B,C;
wire T1,T2,T3,F2-b,E1,E2,E3;
or g1(T1,A,B,C);
and g2(T2,A,B,C);
and g3(E1,A,B);
and g4(E2,A,C);
and g5(E3,B,C);
or g6(F2,E1,E2,E3);
not g7(F2-b,F2);
and g8(T3,T1,F2-b);
or g9(F1,T2,T3);
endmodule
```

```
// Gate-level description of circuit
module test_circuit;
reg[0:2] D;
wire F1,F2;
Circuit_below M2(D[2],D[1],D[0],F1,F2);
initial
begin
D=3'b000;
repeat(7) #10 D= D+ 3'b001
end
initial
$monitor ("ABC=%b F1= %b F2 = %b", D, F1, F2)
endmodule
```

Test bench



Dataflow modeling

- Combinational logic can be expressed using Boolean equations.
- Boolean equations are specified in Verilog with a continuous assignment statement consisting of the keyword *assign* followed by a Boolean expression.
- The following symbols are used for logical operators in Verilog: AND(&), OR(|), and NOT(~).
- HDL models for a circuit specified with the following Boolean expressions

$$E = A + BC + B'D \quad \text{and} \quad F = B'C + BC'D'$$

//Verilog model: Circuit with Boolean expressions

```
module Circuit_Boolean_CA(E,F,A,B,C,D);
    output E,F;
    input A,B,C,D;
```

```
    assign E = A|(B&C)|(~B&D);
    assign F = (~B&C)|(B&~C&~D);
endmodule
```

Dataflow modeling

- Verilog HDL provides about 30 different operators

Symbol	Operation
+	binary addition
-	Binary subtraction
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
==	Equality
>	Greater than
<	Less than
{ }	Concatenate
? :	Conditional

```
//module decoder_2x4_df(  
    output [0:3] D,  
    input      A,B,enable);  
  
assign D[0] = ~A&~B&enable,  
       D[1] = ~A&B&enable,  
       D[2] = A&~B&enable,  
       D[3] = A&B&enable;  
  
endmodule  
  
//module binary_adder(  
    output [3:0] sum,  
    output      C_out,  
    input [3:0]  A,B  
    input      C_in);  
  
assign {C_out,Sum} = A+B+ C_in;  
  
endmodule
```

Dataflow modeling

- HDL for module with two four bit inputs A and B and three outputs.
- One output ($A_{lt}B$) is logic 1 if A is less than B, a second output ($A_{eq}B$) is logic 1 if A is equal to B. A third output ($A_{gt}B$) is logic 1 if A is greater than B.

```
module mag_compare(  
    output A_lt_B, A_eq_B, A_gt_B,  
    input [3:0]      A,B);  
  
    assign A_lt_B = (A<B);  
    assign A_gt_B = (A>B);  
    assign A_eq_B = (A==B);  
  
endmodule
```

Dataflow modeling

- Conditional operator (`? :`) takes three operands:
`condition? true-expression: false-expression;`

The condition is evaluated. If the result is logic 1, the true expression is evaluated. If the result is logic 0, the false expression is evaluated.

`assign OUT = select ? A:B;`

Specifies the condition that `OUT = A` if `select` is 1 else `OUT=B`.

```
module mux_2X1_df(m_out, A, B,select)
    output m_out;
    input   A,B, select;
    assign   m_out = (select) ? A:B;
endmodule
```

Behavioral Modeling

- **Behavioral Modeling:** HDL for digital circuits at a *functional* and *algorithmic* level.
- **Mostly used for sequential circuits; but can be used for combinational circuits as well.**
- **It uses the keyword *always***

Syntax for always

always @ (the event control expression)

list of statements (procedural assignment statements)

- **The target output of procedural statements must be of the reg data type because a reg data type retains its value until a new value is assigned.**
- **This is contrary to the wire data type (output data type), where the target is continuously update.**

Behavioral Modeling

//Behavioral description of two-to-one line multiplexer

```
module mux_2X1_beh(m_out, A, B,Select);
output m_out;
input A,B,select;
reg m_out;

always @ (A or B or select) //always @(A,B,select)

if (select == 1) m_out = A;
else m_out = B;

endmodule
```

Note that there is no semicolon (;) at the end of the always statement

Behavioral Modeling

// Behavioral description of four-to-one line multiplexer

```
module mux_2X1_beh
(output reg m_out,
 input     in_0,in_1,in_2,in_3,
 input [1:0] select);

always @ (in_0, in_1,in_2,in_3,select)
case(select)
  2'b00:  m_out = in_0;
  2'b01:  m_out = in_1;
  2'b10:  m_out = in_2;
  2'b11:  m_out = in_3;
endcase

endmodule
```

- **Binary numbers in Verilog are specified with letter b preceded by a prime. The size of the number is written first and then its value.**

Behavioral Modeling

- Represent the functionality of digital circuit without specifying its hardware.
- Sequential circuits are described in HDL using Behavioral modeling.
- Two kinds of abstract behaviors in the Verilog HDL:
 - Behavior declared using keyword *initial*
 - Behavior declared using keyword *always*
- A module can have many behaviors declared by initial or always. They execute concurrently with respect to each other starting at time 0 and may interact through common variable
- In simulating a sequential circuit, clock is required for triggering flip-flops and can be generated in the following ways

Behavioral Modeling

```
initial  
begin  
    clock = 1'b0;  
    repeat(30)  
        #10 clock = ~clock;  
    end
```

```
initial  
begin  
    clock = 1'b0;  
end  
initial #300 $finish  
always  
    #10 clock = ~clock;
```

```
initial  
begin  
    clock = 1'b0;  
forever  
    #10 clock = ~clock;  
end
```

- The third one is for a free-running clock
- The delay control operator (#) suspends execution of statements until a specified time has elapsed.
- Another operator (@) is called event control operator and is used to suspend activity until an event occurs.
- An event can be a change in signal value (@ A) or a specified transition of a signal value (e.g @ (posedge clock))

Behavioral Modeling

- **Signal level events occur in combinational circuits and latches**
 - always @ (A or B or C) always (A,B,C)
- **Signal transition events are used in synchronous sequential circuits.**
 - always @ (posedge clock or negedge reset)
 - always @ (posedge clock, negedge reset)
- **Assignments of a logic value to a variable within an initial or always statement are called procedural assignment**
- **Continuous assignments are made through keyword assign and have an implicit level-sensitivity.**

Behavioral Modeling

- There are two kind of procedural statements:
 - Blocking
 - Use the symbol $=$ as the assignment operator
 - Executed sequentially
 - assume $A=0$ and $B=1$
 - $B = A; C = B+1; B=0$ and $C = 1$
 - Sequential ordering and cyclic behavior with level sensitive
 - Combinational logic
 - Non-blocking
 - Use the symbol $<=$ as the assignment operator
 - Executed concurrently (assume $A = 0$ and $B = 1$)
 - $B <= A; C <= B+1; B=0$ and $C = 2$
 - When modeling concurrent execution and modeling latched behavior
 - Concurrent operation synchronized by a common clock

Flip-flops and latches

// Description of D latch

```
module D_latch (Q,D,enable);
output Q;
input D, enable;
reg Q;

always @ (enable or D)
    if (enable) Q<=D ;

endmodule
```

// Description of D latch

```
module D_latch
(output reg Q,
input enable,D);

always @ (enable or D)
    if (enable) Q<=D ;

endmodule
```

Flip-flops and latches

// Description of D flip-flop

```
module D_FF (Q,D,clk);
output Q;
input D, clk;
reg Q;
```

always @ (posedge clk)

```
    Q<=D ;
```

endmodule

// Description of D flip-flop
with asynchronous rest

```
module DFF
(output reg Q,
input clk,D, rst);
```

always @ ()

endmodule

Flip-flops and latches

- Constructing T flip-flop and JK from D flip-flop

// Description of T flip flop

```
module TFF (Q,T,clk,rst);
    output Q;
    input T, clk,rst;
    wire DT;
    assign DT= Q^T;
    DFF TF1(Q,DT,clk,rst);
endmodule
```

// Description of JK flip-flop
from D flip-flop

```
module JKFF (
    output Q,
    input J,K,clk, rst);
    wire JK;
```

```
assign JK= (J&~Q)|(~K&Q);
DFF JK1(Q,JK,clk,rst);
```

endmodule

Flip-flops and latches

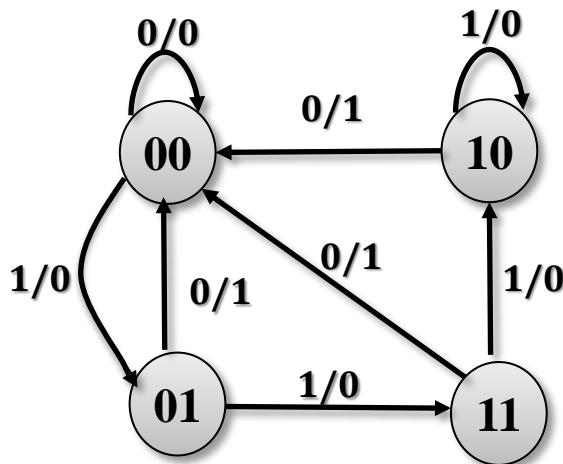
- **HDL model for JK flip-flop from characteristic**

// Description of JK FF from characteristic table

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

HDL model for state diagram

- HDL model for the operation of sequential circuit can be based on the state diagram



HDL model for state diagram

- HDL model for the operation of sequential circuit can be based on the state diagram

// Mealy FSM zero detector

```
module Mealy_Zero_Detector(
    output reg y_out,
    input  X_in, clock, reset
);

reg[1:0] state,next_state;
parameter S0=2'b00, S1=2'b01,S2=2'b10,S3=2'b11;

always @ (posedge clk, negedge reset)

    if (reset == 0) state <= S0;
    else state <= next_state;
always @(state, x_in)
    case(state)
        S0: if (x_in) next_state = S1; else next_state = S0;
        S1: if (x_in) next_state = S3; else next_state = S0;
        S2: if (x_in) next_state = S0; else next_state = S2;
        S3: if (x_in) next_state = S2; else next_state = S0;
    endcase
endmodule
```

HDL model for state diagram

//continue from previous slide

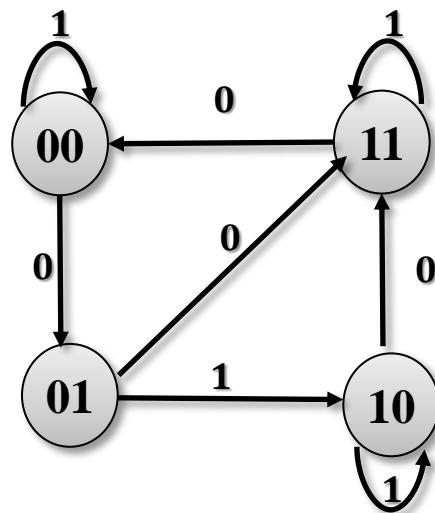
```
always @ (state, x_in)
  case(state)
    S0:      y_out = 0;
    S1,S2,S3: y_out = ~x_in;
  endcase

endmodule
```

- Three always blocks that execute concurrently and interact through common variables.
- The first always statement resets the circuit and specifies the synchronous clocked operation.
- The second always block determines the value of the next state transition as a function of present state and input.
- The third always block specifies the output as a function of present state and the input.

HDL model for state diagram

- HDL model for the operation of sequential circuit can be based on state diagram: Moore model



HDL model for state diagram

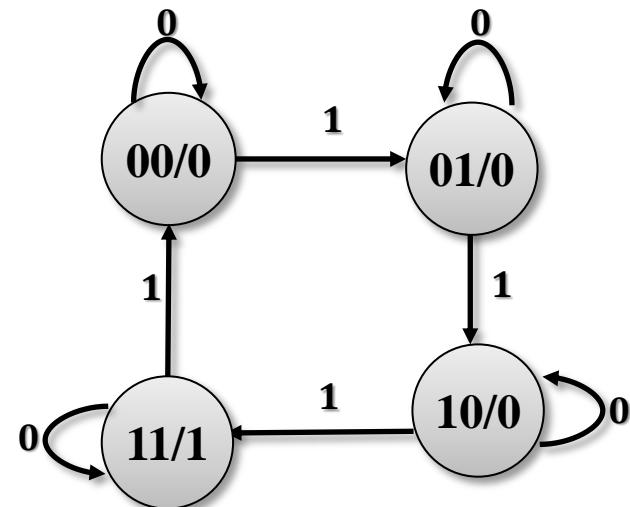
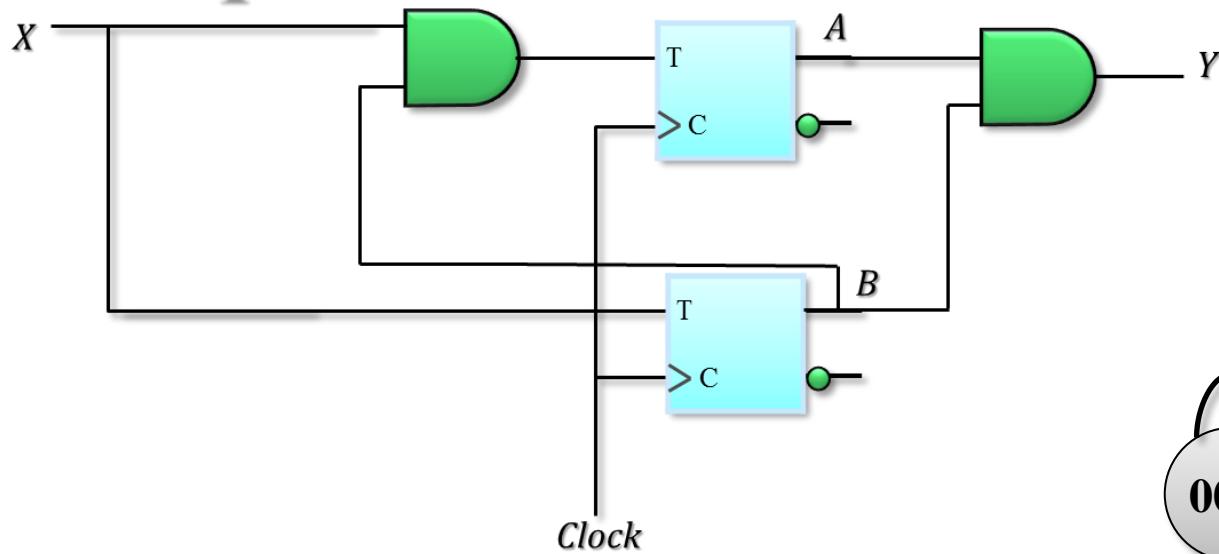
```
// Moore model FSM
```

```
module Moore_Model(  
    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 clk, negedge reset)  
  
    if (reset == 0) 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;  
  
endmodule
```

Structural description: clocked Sequential circuits

- **Sequential circuits are composed of combinational logic and flip-flops.**
- **Combinational part can be described with assign statements and Boolean equations.**
- **The flip-flops are described with an always statement.**
- **The separate modules can be combined to form a structural model by instantiation within a module.**

Structural description: clocked sequential circuits



Structural description: clocked sequential circuits

// structural model

```
module Moore_Model_One(  
    output      y_out,A,B,  
    input       X_in, clock, reset  
);  
  
wire   TA,TB;  
  
assign TA = X_in&B;  
assign TB = X_in;  
  
assign y_out = A&B;  
  
TFF MA(A,TA,clock,reset);  
TFF MB(B,TB,clock,reset);  
  
end module
```

// structural model

```
module TFF(Q,T,CLK,RST_b);  
    output      Q;  
    input       T, CLK,RST_b;  
    reg        Q;  
  
always @ (posedge CLK, negedge RST_b)  
if (RST_b == 0) Q<= 1'b0;  
else if (T) Q <= ~Q;  
  
end module
```