Week 7 - T1 2020

Verilog HDL

ELEC2141: Digital Circuit Design

Summary

Moore and Mealy sequential circuits

State reduction

State assignment

Sequential circuit design

Circuit synthesis

Moore to Mealy/Mealy to Moore conversion

Overview

Hardware Description Languages

Verilog

Module declaration

Test bench

Dataflow modelling

Behavioural modelling

Sequential circuit implementation

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: by applying input stimulus and observing the outputs, the logic implementation or the behavior of the circuit design can be tested (called test bench) to see if functions as intended

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 IEEE standards: 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

Common HDL Tools

Text Editor – to write the HDL program

Compiler – parse the HDL program, find syntax errors and convert the description to an intermediate technology-neural digital design description language

Synthesizer – target the design to a specific hardware technology and build the physical implementation of the circuit

Simulator – simulate the HDL program using given input combination to verify correct behaviour of the program.

Verilog basics

Verilog models are composed of text using *keywords* defined in lowercase

Examples include *module, endmodule, input, output,wire, and, or, not* …

Comments are given between two forward slashes (//)

Multiline comments begin with /* and terminate with */

Blank spaces are ignored, but cannot appear within the text of a keyword, user-specified identifier, an operator, or representation of a number

Verilog is a case sensitive - not is not the same as NOT

Module Declaration

A *module* is fundamental descriptive unit (or component) in the Verilog language

It is used to define a circuit or a sub circuit

It is declared by the keyword *module* and must always be terminated by the keyword *endmodule*

The inputs and outputs of the module are termed *ports*

Objects such as gates, registers or functions used in a module are given *identifiers* so they can be referenced from other places

Module Declaration

Syntax for module

```
module   identifier(port, port …);
    …
endmodule
```

Identifiers are composed of alphanumeric characters and the underscore (_), and are case sensitive

- They 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 (;)

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 connections are identified by a keyword *wire*

Syntax for wire keyword

```
wire  identifier, identifier …identifier;
```

Module Declaration

The gates that form the circuit are specified from 8 predefined primitive gates

The primitive gates are identified by keywords: `and, or, not, nor, nand, xor, xnor, buf`

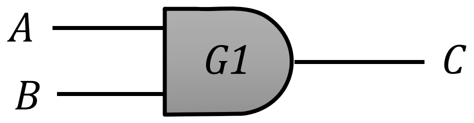
The elements of the primitive gate list are referred as *instantiation* of a gate

Syntax for primitive gates keywords

```
    instance  instancename(output, inputs);
```

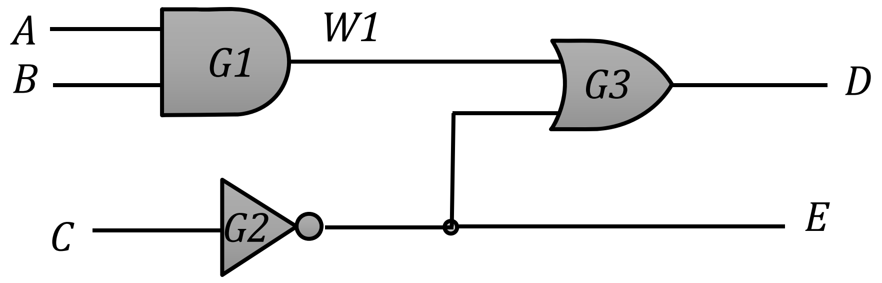
e.g.

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



Module Declaration

Verilog HDL for the following combinational logic



```
// Verilog model of circuit shown above
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

Declaration versus *instantiation*

- A Verilog module is declared with its declaration specifying 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
- Declaration defines a module, instantiation uses it

Module Declaration

The *time directive* or unit of time can be declared before the module

```
`timescale   time_unit base / precision base
```

- `time_unit` is the amount of time a delay of 1 represents. The time unit must be 1, 10, or 100
- `base` *is the time base for each unit:* `s ms us ns ps` **or** `fs`
- `precision` and `base` represent how many decimal points of precision to use relative to the time units.

Example:

```
`timescale   1ns/10ps
```

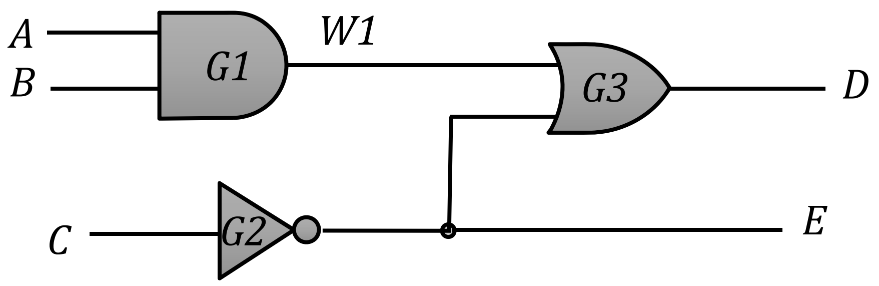
Indicates delays are in 1 nanosecond units with 2 decimal points of precision (10 ps is 0.01 ns).

Note: there is no default timescale in Verilog; delays are simply relative numbers until a timescale directive declares the units uand base the numbers represent.

Module Declaration

In Verilog, the propagation delay of a gate is specified in terms of time units and by #delay ; #(rise delay, fall delay)

For the following HDL Verilog code:



```
// Verilog model of circuit above
`timescale 1ns/10ps
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); // here propagation delay specified
    not        #10 G2(E,C);
    or         #20 G3(D,w1,E);
endmodule
```

Module Declaration

Signals can be grouped together in a vector

The vector definition syntax is:

```
[msb:lsb]
```

`msb` is the index number of the leftmost bit and `lsb` is the index of the rightmost one

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
```

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

Module Declaration

```
// Structural description of 2-to-4 line decoder
module  decoder_ 2x4_gates (A,B,D,EN);
   output   [0:3] D;
   input    A,B,EN;
   wire     A_b, B_b, EN_b;

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

Test bench

To simulate a circuit with an HDL, inputs are applied to the circuit so the simulator can generate an output response

A HDL description that provides the stimulus to a design is a *test bench* and consists of a signal generator and instantiation of the module to be verified

A test bench has no input or output ports as it does not interact with its environment

Initial statements are used to provides initial values and do not play a role in circuit synthesis

Test bench

The inputs to the circuit are declared with keyword `reg` and the outputs are declared with the keyword `wire`

The set of statements to be executed is called a block statement and enclosed by keywords `begin … end`

The simulation is terminated with `$finish` system task

```
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

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

A stimulus model (test bench) for that would be

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

 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 which are built-in system functions recognized by keywords that begin with the $ symbol

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

Taskname (format specification, argument list);

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

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

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

$write – same as $display, but without going to next line

$monitor – displays variables whenever a value changes during a simulation

$time - displays the simulation time

$finish – terminates the simulation

Test bench

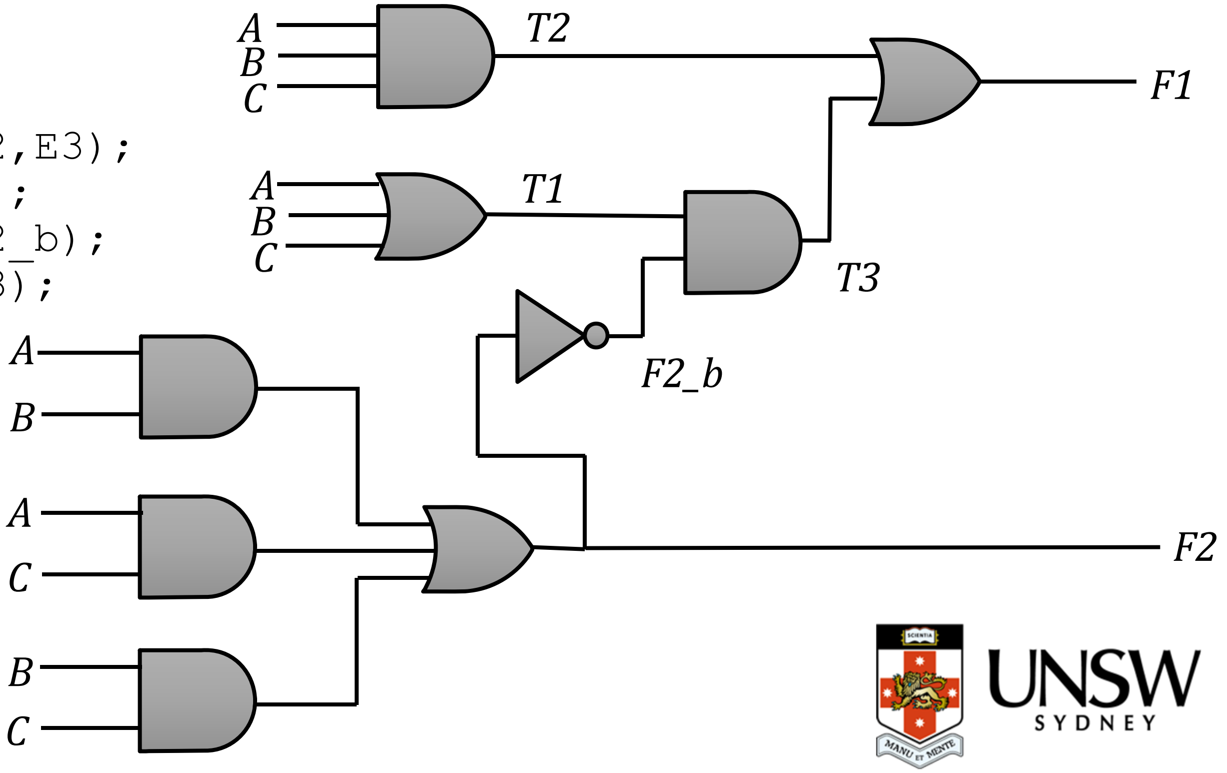
Two types of verification: functional and timing

Functional verification is based on a truth table

Timing verification is based on waveforms and includes effect of delays through the gates

Test bench

```
// 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
```



Test bench

```
// 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);
    initial #200 $finish;
endmodule
```

Circuit modeling

Circuits can be modelled using

- *Structural modeling*:– here a set of interconnected components representing the functionality of the circuit is used
- *Dataflow modeling*: here a set of concurrent assignment statements are used to assign values to signals and model the flow of data
- *Behavioural modeling*: here procedural blocks or processes are defined to represent the behaviour of circuits

Dataflow modeling

Combinational logic can be expressed in Verilog with Boolean equations

Done with a continuous assignment statement consisting of the keyword *assign* followed by a Boolean expression

The symbols for logical operators in Verilog are AND(&), OR(|), and NOT(~)

e.g. E= A + BC +B’D   and 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 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;
```

When the condition is evaluated, if the result is logic 1, the true expression is evaluated but if the result is logic 0, the false expression is evaluated

```
assign  OUT = select ? A : B;
```

Specifies 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
```

Behavioural modeling

Behavioural modeling performed in HDL for digital circuits at a *functional* and *algorithmic* level.

Mostly used for sequential circuits, but can be also used for combinational circuits

It uses the keyword *always*; always blocks loop to execute over and over again

Syntax for always

```
always @(the event control expression)
list of statements (procedural assignment statements)
```

The target output of procedural statements must be of reg data type because a reg data type retains its value until a new value assigned

This is contrary to the wire data type (output data type), where the target is continuously update

Behavioural 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 - no semicolon (;) at the end of the always statement

Behavioural modeling

The *case* statement compare an expression to a series of cases and executes the statement or statement group associated with the first matching case

It support single or multiple statements

Multiple statements are grouped using begin and end keywords

Syntax for case

```
case(expression)
case1 : statement
case 2: statement
…
default: statement
endcase
```

Behavioural modeling

```
// Behavioural 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 apostrophe. The size of the number is written first and then its value.

Behavioural 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:

- Behaviour declared using keyword *initial*
- Behaviour declared using keyword *always*

A module can have many behaviours declared by initial or always

They execute concurrently with respect to each other starting at time 0 and may interact through common variable

Behavioural 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
```

In simulating a sequential circuit, a clock is required to trigger flip-flops and can be generated in 3 different ways as above; the third one is for a free-running clock

The delay control operator (#) suspends execution of statements until a specified time has elapsed

Behavioural modeling

The 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))

Signal level events occur in combinational circuits and latches

```
    always @ (A or B or C)    always @ (A,B,C)
```

Signal transition events occur 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 assignments*

Continuous assignments are made through keyword assign and have an implicit level-sensitivity.

Behavioural 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 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 @ (posedge clk, negedge rst)

    if (rst==0)        Q<=1'b0;
    else               Q <=D;

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,clk,DT,rst);

endmodule
```

```
// Description of JK flip-flop

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

assign JK= (J&~Q)|(~K&Q);

DFF JK1(Q,clk,JK,rst);

endmodule
```

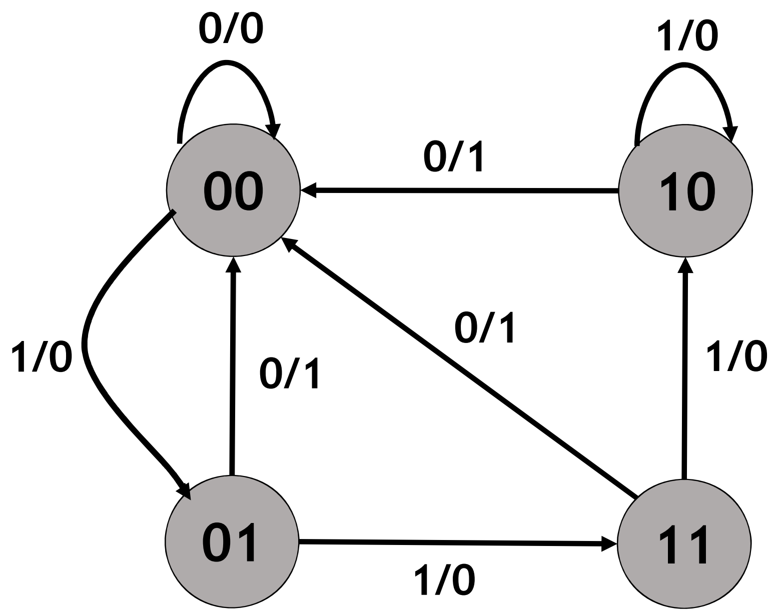
Flip-flops and latches

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 models for state diagram

HDL models 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 = S2; else next_state = S0;
       S3:   if (x_in) next_state = S2; else next_state = S0;
     endcase
```

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.

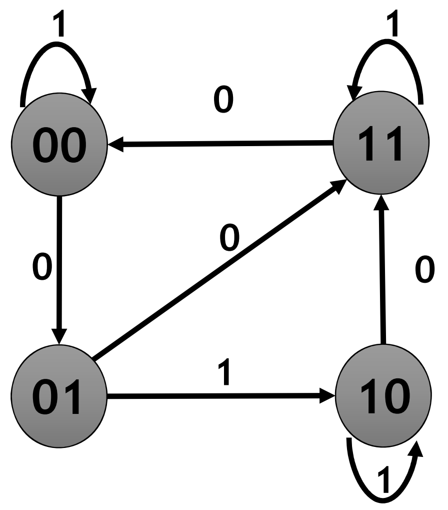
The first always statement resets the circuit and specifies the synchronous clocked operation.

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

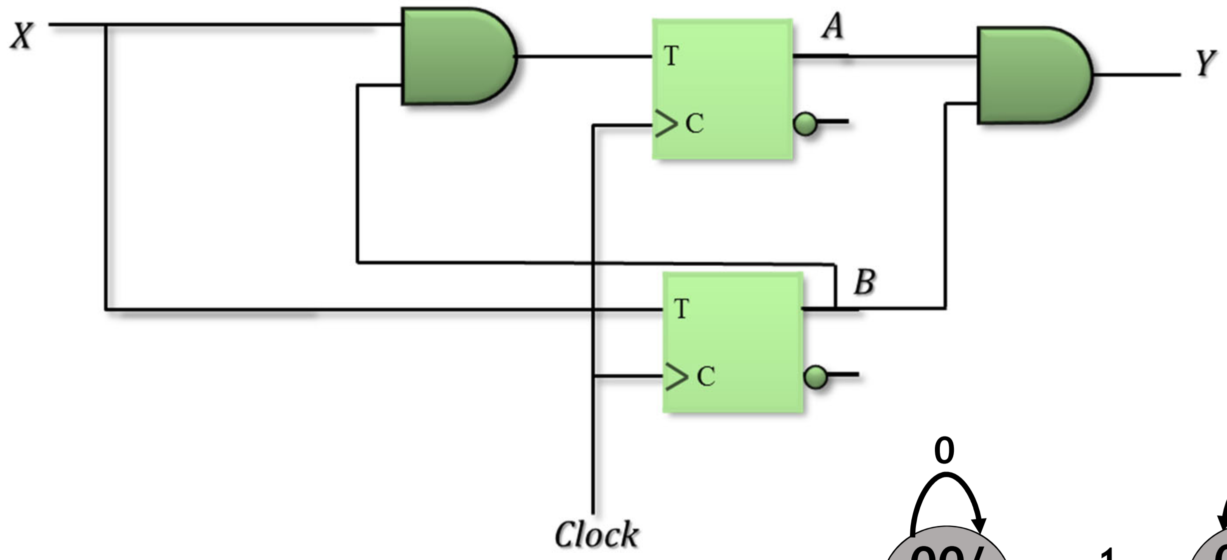
Sequential circuits are composed of combinational logic and flip-flops.

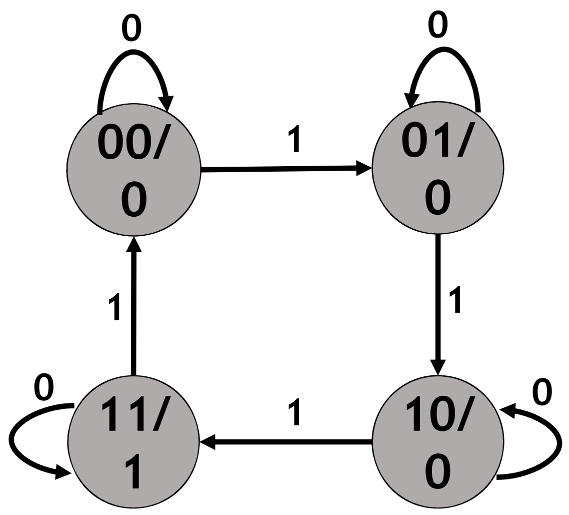
Combinational part can be described with assign statements and Boolean equations.

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
```