5-6

Synthesizable HDL Models of Sequential Circuits

J.J. Shann

84

Synthesizable HDL Models of Sequential Circuits

- ***Behavioral Modeling*** of Sequential Circuit
- HDL Models of Flip-Flops and Latches
- ***State Diagram-Based*** HDL Models
- ***Structural*** Description of Clocked Sequential Circuits

J.J. Shann 5-85

85

A. Behavioral Modeling

- Behavioral models:
  - are abstract representations of the *functionality* of digital hardware.
  - describe how a ckt behaves, but don't specify the internal details of the ckt.

J.J. Shann HDL-86

86

- Two kinds of abstract behaviors in the Verilog HDL:
  - ***single-pass behavior*** : declared by the keyword **initial**, for prescribe stimulus signals in a ***test bench***
    - specifies a single statement or a block statement (by **begin … end** or **fork … join** keyword pair)
    - expires after the associated statement executes.
    - * **Never use single-pass behavior to model the behavior of a ckt!**
  - ***cyclic behavior***: declared by the keyword **always**
    - executes and re-executes indefinitely, until the simulation is stopped.
- A module may contain an arbitrary # of **initial** or **always** behavioral statements.
  - They execute ***concurrently*** w.r.t. each other, starting at time 0, and may interact through common variables.

J.J. Shann HDL-87

87

initial statement

- **initial**: single-pass behavior, used in ***test bench***
  - E.g.: free-running clock

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

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

* The 3 behavioral statements can be written in any order.

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

an indefinite loop

J.J. Shann HDL-88

always statement

- The general form of **always** statement:

```
always @ (event control expression) begin
  //Procedural assignment statements that execute
    when the condition is met
end
```

  - The variables in the left-hand side of the procedural statements must be declared as the **reg** data type.
  - *event control expression*: *sensitivity list*
    - i. ***level sensitive events*** (in comb ckts and in latches)
      - E.g.: always @(A or B or C) //@(A, B, C)
    - ii. ***edge sensitive events*** (in flip-flops): **posedge**, **negedge**
      - E.g.: always @(posedge clock or negedge reset) //Verilog 1995
        always @(posedge clock, negedge reset) //Verilog 2001

J.J. Shann HDL-89

Procedural vs. *Continuous* Assignment

- ***Continuous*** assignment: **assign**
  - The updating of a continuous assignment is triggered whenever an event occurs in any variable included on the right-hand side of its expression.
- ***Procedural*** assignment: **initial** or **always**
  - is an assignment of a logic value to a variable within an **initial** or **always** statement.
  - A procedural assignment is made only when an assignment statement is executed and assigns value to it within a behavioral statement.
  - A variable having type **reg** remains unchanged until a procedural assignment is made to give it a new value.

J.J. Shann HDL-90

■90

Procedural Assignments

- Two kinds of procedural assignments:
  - ***blocking*** assignment: **=** (for ***combinational logic***)
    - Blocking assignment statements are *executed sequentially* in the order they are listed in a block of statements.
    - E.g.: B = A;
      C = B + 1; //C = A + 1
  - ***nonblocking*** assignment: **<=**
    (for ***sync seq ckt*** and *latched behavioral*)
    - Nonblocking assignment statements are *executed concurrently* by evaluating the set of expressions on the right-hand side of the list of statements.
    - They do not make assignments to their left-hand sides until all of the expressions are evaluated.
    - E.g.: B <= A;
      C <= B + 1; //C will contain the original value of B, plus 1.

J.J. Shann HDL-91

■91

- General rule:
  - ***Blocking*** assignment, = : Use blocking assignments when *sequential ordering* is imperative and in *cyclic behavior* that is *level sensitive* (i.e., in **combinational logic**).
  - ***Nonblocking*** assignment, <= : Use nonblocking assignments when modeling *concurrent execution* (e.g., *edge-sensitive behavior* such as **synchronous**, concurrent register transfers) and when modeling *latched behavior*.

* Prevent conditions that lead synthesis tools astray and create mismatches b/t the behavior of a model and the behavior of physical hardware produced by a synthesis tool.

J.J. Shann HDL-92

92

Example

- Example: Full Adder

$p = a \oplus b$ … carry propagate function
$q = a \cdot b$ … carry generate function
$s = a \oplus b \oplus cin = p \oplus cin$
$cout = a \cdot b + (a \oplus b) \cdot cin = q + p \cdot cin$

@(a, b, cin)

```
always @ (*)
  begin
    p = a ^ b;  //blocking
    q = a & b;
    s = p ^ cin;
    cout = q | (p & cin);
  end
```

✓

```
always @ (*)
  begin
    p <= a ^ b;  //nonblocking
    q <= a & b;
    s <= p ^ cin;
    cout <= q | (p & cin);
  end
```

@(a, b, cin, p, q)

* For simulation & synthesis tools

J.J. Shann HDL-93

93

Guidelines of **Blocking** and **Nonblocking** Assignments

- Use *continuous assignments* **assign** to model simple ***combinational*** logic.
- Use **always @ (*)** and *blocking assignments* to model more complicated ***combinational*** logic where the **always** statement is helpful.
- Use **always @ (posedge clk)** and *nonblocking assignments* to model synchronous ***sequential*** logic.
- Do not make assignments to the same signal in more than one **always** statement or continuous assignment statement.

```
assign y = s ? d1 : d2;
```

```
always @ (*)
  begin
    p = a ^ b;  //blocking
    q = a & b;  //blocking
    s = p ^ cin;
    cout = q | (p & cin);
  end
```

```
always @ (posedge clk)
  begin
    n1 <= d;  //nonblocking
    q <= n1;  //nonblocking
  end
```

J.J. Shann HDL-94

B. HDL Models of Flip-Flops and Latches

- Latch w/ control input:
  - responds to a change in data input w/ a change in the output as long as the enable input is asserted, i.e., in the active level.

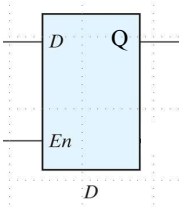
    ⇒ The latch is controlled by the "level" of its control input.
- Flip-Flop:
  - responses only to a "transition" of a triggering input called the "clock."
  - Positive-edge trigger: $0 \rightarrow 1$

    Negative-edge trigger: $1 \rightarrow 0$

J.J. Shann HDL-95

HDL Example 5.1: D-Latch

- HDL Example 5.1: D-Latch (behavioral)



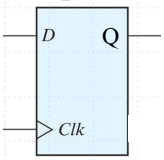

```
// Description of D latch (transparent latch), Fig. 5-6
//module D_latch (Q, D, enable);
//    output Q;
//    input   D, enable;
//    reg     Q;

//Verilog 2001, 2005

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

   always @ (enable, D)        //(enable or D)
      if (enable) Q <= D;      //(enable == 1)
endmodule
```

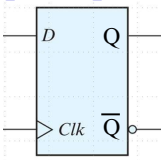

HDL Example 5.2: D-Type Flip-Flop

- HDL Example 5.2(a): D-Type Flip-Flop w/o Reset

D Flip-Flop

D	Q(t + 1)
0	0
1	1

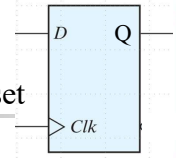





```
module D_flip_flop (Q, D, Clk);
   output    Q;
   input     D, Clk;
   reg       Q;

   always @ (posedge Clk)
      Q <= D;
endmodule
```

```
module D_flip_flop_b (Q, Q_b, D, Clk);
   output    Q, Q_b;
   input     D, Clk;
   reg       Q;

   assign    Q_b = ~Q;
   always @ (posedge Clk)
      Q <= D;
endmodule
```

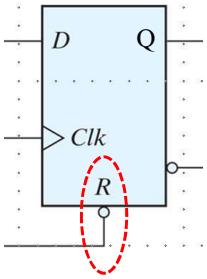
D flip-flop
w/o async reset




```
module D_flip_flop (Q, D, Clk);
   output   Q;
   input    D, Clk;
   reg      Q;

   always @ (posedge Clk)
      Q <= D;
endmodule
```

- HDL Example 5.2(b):

D Flip-Flop w/ **Active-Low *Asynchronous*** Reset

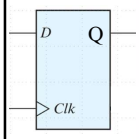



```
// Description of D flip-flop
// with active-low asynchronous reset

module D_flip_flop_AR (Q, D, Clk);
   output   Q;
   input    D, Clk;
   reg      Q;

   always @ (posedge Clk)

      Q <= D;
endmodule
```

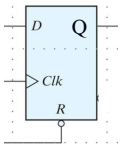
J.J. Shann HDL-98

98

- Testbench of Example 5.2(a)(b):




```
module D_flip_flop (Q, D, Clk);
   output   Q;
   input    D, Clk;
   reg      Q;

   always @ (posedge Clk)
      Q <= D;
endmodule
```




```
module D_flip_flop_AR (Q, D, Clk, rst);
   output   Q;
   input    D, Clk, rst;
   reg      Q;

   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;    //if (rst == 0)
      else Q <= D;
endmodule
```

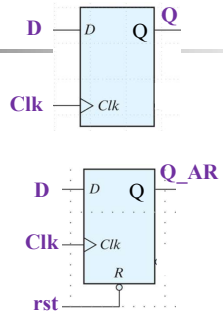
```
module t_D_flip_flops;
   wire  Q, Q_AR;
   reg   D, Clk, rst;

   D_flip_flop M0 (Q, D, Clk);
   D_flip_flop_AR M1 (Q_AR, D, Clk, rst);

   initial #100 $finish;
   initial begin Clk = 0; forever #5 Clk = ~Clk; end
   initial fork
      D = 1;
      rst = 1;
      #20 D = 0;
      #40 D = 1;
      #50 D = 0;
      #60 D = 1;
      #70 D = 0;
      #42 rst = 0;
      #72 rst = 1;
   join
endmodule
```

* Statements within the **fork … join** block execute in parallel.

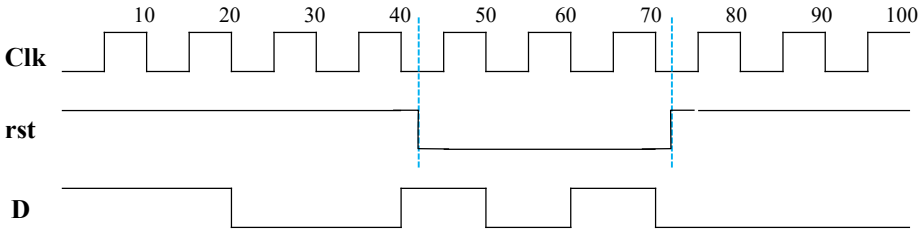
J.J. Shann HDL-99

99

- Waveform of the input stimuli generated by the testbench:




```
module t_D_flip_flops;
  wire Q, Q_AR;
  reg  D, Clk, rst;

  D_flip_flop M0 (Q, D, Clk);
  D_flip_flop_AR M1 (Q_AR, D, Clk, rst);

  initial #100 $finish;
  initial begin Clk = 0; forever #5 Clk = ~Clk; end
  initial fork
    D = 1;
    rst = 1;
    #20 D = 0;
    #40 D = 1;
    #50 D = 0;
    #60 D = 1;
    #70 D = 0;
    #42 rst = 0;
    #72 rst = 1;
  join
endmodule
```



J.J. Shann HDL-100

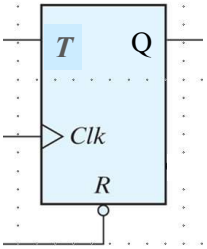
■100

HDL Example 5.3 (a)(b): T Flip-Flop Models

- HDL Example 5.3(a): T flip-flop w/ Asynchronous Reset

```
module D_flip_flop_AR (Q, D, Clk, rst);
  output  Q;
  input   D, Clk, rst;
  reg     Q;

  always @ (posedge Clk, negedge rst)
    if (!rst) Q <= 1'b0;    //if (rst == 0)
    else Q <= D;
endmodule
```

D f-f w/ active-low async reset



T	Q(t + 1)
0	$Q(t)$
1	$Q'(t)$

```
// Description of Toggle (T) flip-flop
module Toggle_flip_flop_1 (Q, T, Clk, rst);
  output   Q;
  input    T, Clk, rst;
  reg      Q;

  always @ (posedge Clk, negedge rst)
    if (!rst) Q <= 1'b0;
    else  if (T) Q <= !Q;
endmodule
```

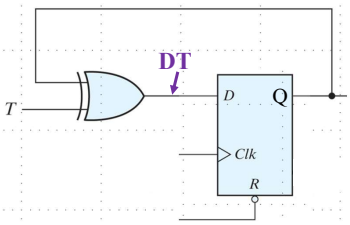
J.J. Shann HDL-101

■101

```
module D_flip_flop_AR (Q, D, Clk, rst);
  output  Q;
  input   D, Clk, rst;
  reg     Q;

  always @ (posedge Clk, negedge rst)
    if (!rst) Q <= 1'b0;    //if (rst == 0)
    else Q <= D;
endmodule
```

D f-f w/ active-low async reset

- HDL Example 5.3(b):
  T flip-flop from D flip-flop and gates




```
//T flip-flop from D flip-flop and gates
module T_flip_flop_2 (Q, T, Clk, rst);
  output   Q;
  input    T, Clk, rst;
  wire     DT;

  assign  DT = T ^ Q;
  D_flip_flop_AR M0 (Q, DT, Clk, rst);
endmodule
```

$DT = T \oplus Q$

Q

J.J. Shann HDL-102

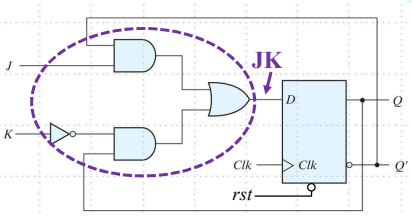
102

```
module D_flip_flop_AR (Q, D, Clk, rst);
  output  Q;
  input   D, Clk, rst;
  reg     Q;

  always @ (posedge Clk, negedge rst)
    if (!rst) Q <= 1'b0;    //if (rst == 0)
    else Q <= D;
endmodule
```

D f-f w/ active-low async reset

HDL Example 5.3 (c): JK Flip-Flop Models

- HDL Example 5.3(c):
  JK flip-flop from D flip-flop and gates




```
//JK flip-flop from D flip-flop and gates
module JK_flip_flop (Q, J, K, Clk, rst);
  output   Q;
  input    J, K, Clk, rst;
  wire     JK;

  assign  JK = (J & ~Q) | (~K & Q);
  D_flip_flop_AR JK1 (Q, JK, Clk, rst);
endmodule
```

$JK = JQ' + K'Q$

J.J. Shann HDL-103

103

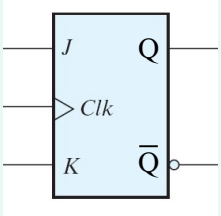
HDL Example 5.4: JK Flip-Flop

- HDL Example 5.4: JK flip-flop
  - *function description* of JK flip-flop

JK Flip-Flop

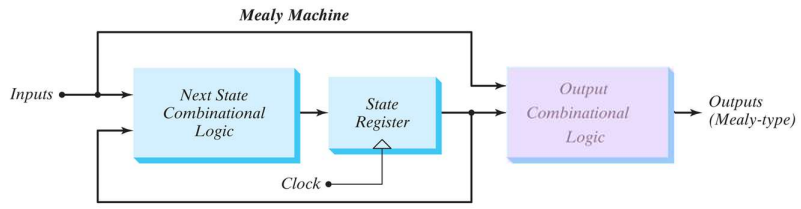
J	K	Q(t + 1)
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$

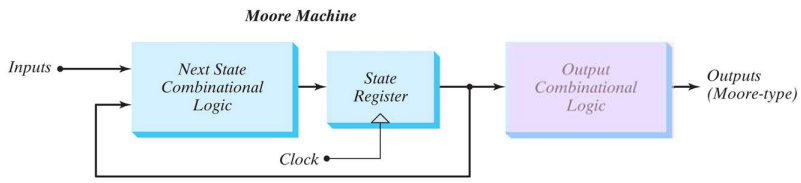
```
//Function description of JK flip-flop
module JK_flip_flop_2 (Q, Q_not, J, K, Clk);
  output    Q, Q_not;
  input     J, K, Clk;
  reg       Q;

  assign  Q_not = ~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
```



C. State Diagram-Based HDL Models of Synchronous Sequential Circuits

- Mealy model
- Moore model

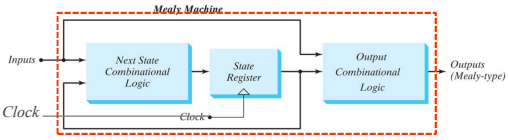


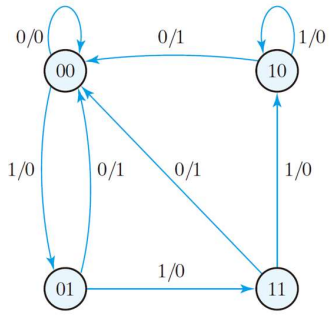


J.J. Shann HDL-105

HDL Example 5.5: Mealy FSM Machine, Zero Detector

- HDL Example 5.5: Mealy Machine, Zero Detector
  - p.281, Fig 5.16
  - p.280, Table 5.3
  - *state diagram-based* model

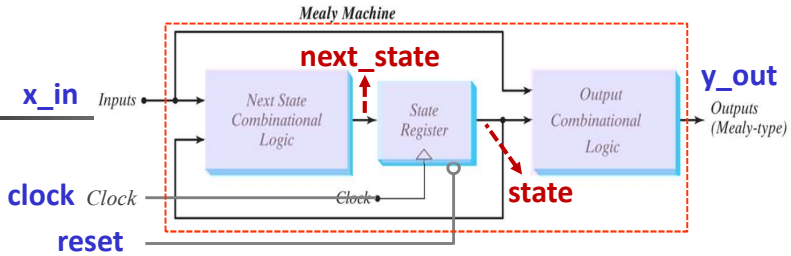


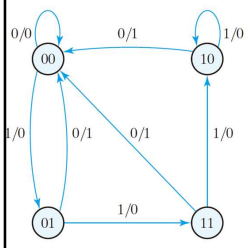


Present State		Next State				Output	
		x = 0		x = 1		x = 0	x = 1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

J.J. Shann HDL-106

106

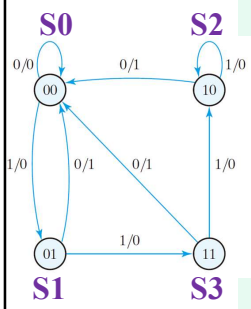




Present State		Next State				Output	
		x = 0		x = 1		x = 0	x = 1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

```
module Mealy_Zero_Detector (output  reg  y_out,
                            input        x_in, clock, reset);
  reg [1: 0]    state, next_state;
  ...
  //Form state transition-- State Register
  always @(posedge clock, negedge reset)
  ...
  // Form the next state-- Next State Combinational Logic
  always @ (state, x_in)
  ...
  // Form the output-- Output Combinational Logic
  always @ (state, x_in)
  ...
endmodule
```

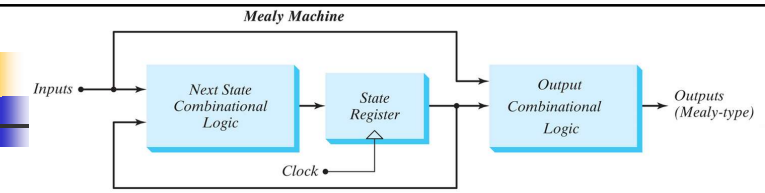
107



Present State		Next State				Output	
		x = 0		x = 1		x = 0	x = 1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

```
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 clock, negedge reset) //state transition
     if (reset == 0) state <= S0;      //Active-LOW reset
     else state <= next_state;

  always @ (state, x_in)            // Form the next state
     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

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

■108




```
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 clock, negedge reset) //state transition
     if (reset == 0) state <= S0;      //Active-LOW reset
     else state <= next_state;

  always @ (state, x_in)            // Form the next state
     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

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

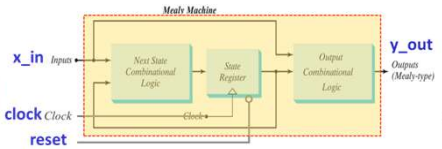
state register

next state combinational logic

output combinational logic

J.J. Shann HDL-109

■109



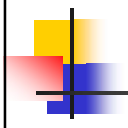
- Testbench of HDL Example 5.5: p.304

```
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 clock, negedge reset) //state transition
    if (reset == 0) state <= S0;
    else state <= next_state;
  always @ (state, x_in)          // Form the next state
    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
  always @ (state, x_in)          // Form the output
    case (state)
      S0:  y_out = 0;
      S1, S2, S3: y_out = ~x_in;
    endcase
endmodule
```

```
module t_Mealy_Zero_Detector;
  wire t_y_out;
  reg   t_x_in, t_clock, t_reset;

  Mealy_Zero_Detector M0 (t_y_out,
   t_x_in, t_clock, t_reset);

  initial #200 $finish;

  //Generate stimulus for t_clock
  initial
     begin
        t_clock = 0;
        forever #5 t_clock = ~t_clock;
     end

  //Generate stimulus for t_x_in, t-reset
  initial fork
     …
     join

endmodule
```

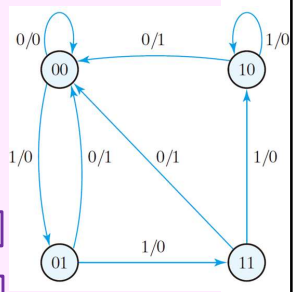
J.J. Shann HDL-110

■110



```
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 clock, negedge reset) //state transition
    if (reset == 0) state <= S0;
    else state <= next_state;
  always @ (state, x_in)          // Form the next st
    case (state)
      S0: if (x_in)  next_state = S1; else next_state
      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
  always @ (state, x_in)          // Form the output
    case (state)
      S0:  y_out = 0;
      S1, S2, S3: y_out = ~x_in;
    endcase
endmodule
```

```
//Generate stimulus for t_x_in, t-reset
initial fork
        t_reset = 0;
#2      t_reset = 1;
#87     t_reset = 0;
#89     t_reset = 1;
#10     t_x_in = 1;
#30     t_x_in = 0;
#40     t_x_in = 1;
#50     t_x_in = 0;
#52     t_x_in = 1;
#54     t_x_in = 0;
#70     t_x_in = 1;
#80     t_x_in = 1;
#70     t_x_in = 0;
#90     t_x_in = 1;
#100    t_x_in = 0;
#120    t_x_in = 1;
#160    t_x_in = 0;
#170    t_x_in = 1;
join
endmodule
```

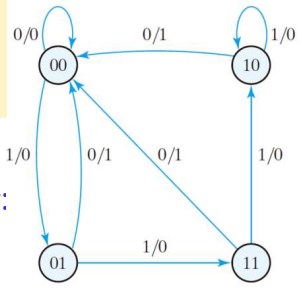


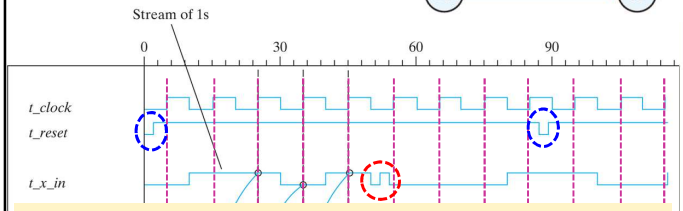
111

■111

* Testbench:
  write stimuli that will test a ckt thoroughly

- Simulation output of Mealy_Zero_Detector:
  p.305, Fig 5.22

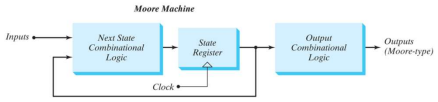


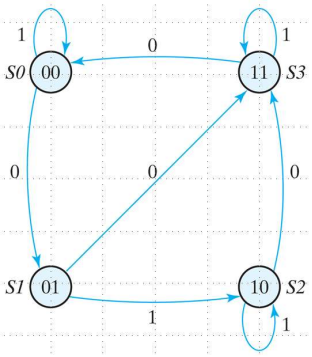



```
module t_Mealy_Zero_Detector;
  wire t_y_out;
  reg  t_x_in, t_clock, t_reset;
  Mealy_Zero_Detector M0 (t_y_out,
   t_x_in, t_clock, t_reset);
  initial #200 $finish;
//Generate stimulus for t_clock
  initial
    begin
      t_clock = 0;
      forever #5 t_clock = ~t_clock;
    end
  //Generate stimulus for t_x_in, t-reset
  initial fork
        t_reset = 0;
  #2    t_reset = 1;
  #87   t_reset = 0;
  #89   t_reset = 1;
  #10   t_x_in = 1;
  #30   t_x_in = 0;
  #40   t_x_in = 1;
  #50   t_x_in = 0;
  #52   t_x_in = 1;
  #54   t_x_in = 0;
  #70   t_x_in = 1;
  #80   t_x_in = 1;
  #70   t_x_in = 0;
  #90   t_x_in = 1;
  #100  t_x_in = 0;
  #120  t_x_in = 1;
  #160  t_x_in = 0;
  #170  t_x_in = 1;
  join
endmodule
```


HDL Example 5.6: Moore FSM Machine

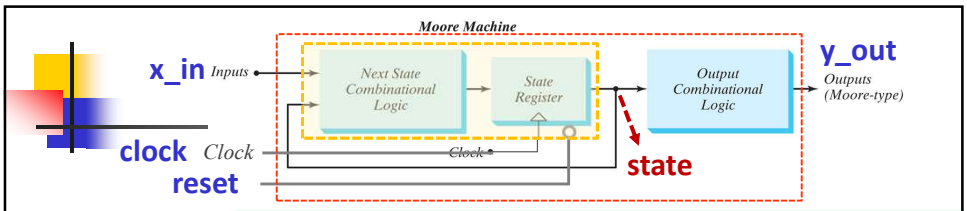
- HDL Example 5.6: Moore Machine
  - p.287, Fig 5.19
  - p.285, Table 5.4
  - *state diagram-based* model

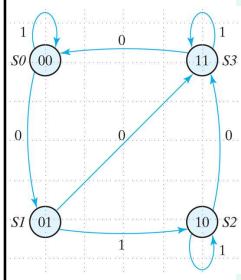




Present State		Input	Next State	
A	*B*	*x*	*A*	*B*
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

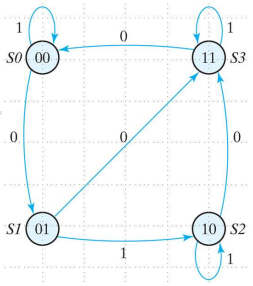
J.J. Shann HDL-113

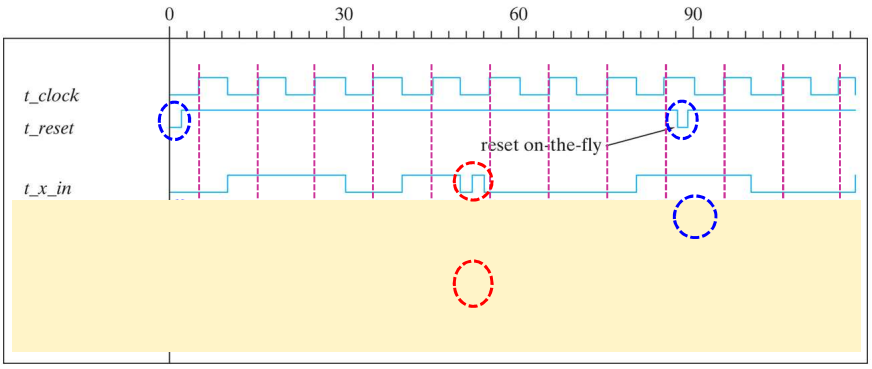





```
module Moore_Model_Fig_5_19 (output [1: 0] y_out,
                                    input  x_in, clock, reset);
  reg [1: 0] state;
  parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @ (posedge clock, negedge reset)
     if (reset == 0) state <= S0;   //Initialize to 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
```


- Simulation output of Moore_Model_Fig_5_19:
  - p.309, Fig 5.23



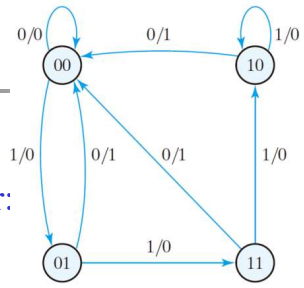


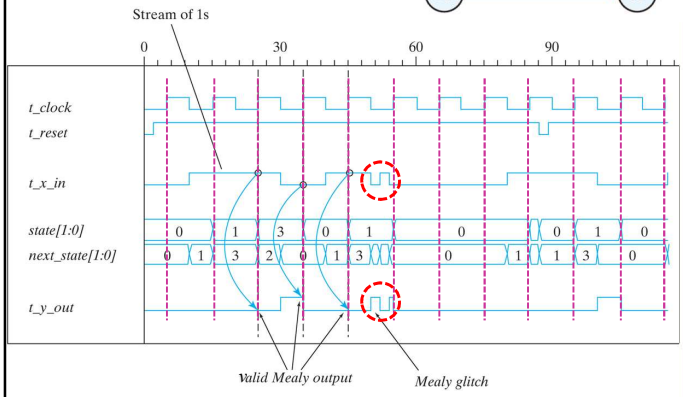
* No glitch

J.J. Shann HDL-115

- Simulation output of Mealy_Zero_Detector:

  p.305, Fig 5.22






```
module t_Mealy_Zero_Detector;
  wire t_y_out;
  reg  t_x_in, t_clock, t_reset;

  Mealy_Zero_Detector M0 (t_y_out,
   t_x_in, t_clock, t_reset);

  initial #200 $finish;

//Generate stimulus for t_clock
  initial
    begin
      t_clock = 0;
      forever #5 t_clock = ~t_clock;
    end

  //Generate stimulus for t_x_in, t-reset
  initial fork
        t_reset = 0;
  #2    t_reset = 1;
  #87   t_reset = 0;
  #89   t_reset = 1;
  #10   t_x_in = 1;
  #30   t_x_in = 0;
  #40   t_x_in = 1;
  #50   t_x_in = 0;
  #52   t_x_in = 1;
  #54   t_x_in = 0;
  #70   t_x_in = 1;
  #80   t_x_in = 1;
  #70   t_x_in = 0;
  #90   t_x_in = 1;
  #100  t_x_in = 0;
  #120  t_x_in = 1;
  #160  t_x_in = 0;
  #170  t_x_in = 1;
  join
endmodule
```


D. Structural Description of Clocked (Synchronous) Sequential Circuits

- Combinational logic ckts:
  - Theirs HDL models can be described
    - by a *connection of gates* (*primitives* and *UDPs*),
    - by *data flow statements* (*continuous assignments*), or
    - by *level sensitive cyclic behaviors* (**always** blocks)
- Sequential ckts: Combinational logic + flip-flops
  - Theirs HDL models use *sequential UDPs* and *behavioral statements* (*edge-sensitive cyclic behaviors*) to describe the operation of flip-flops.
    - The flip-flops are described w/ an **always** statement.
    - The combinational part can be described w/ **assign** statements and Boolean equations.

J.J. Shann HDL-117

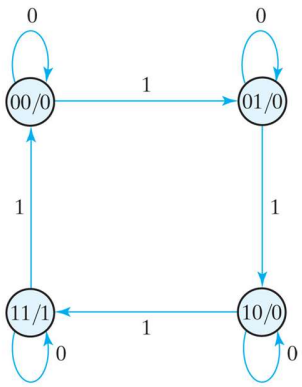
- HDL models of sequential circuits:
  - State-diagram-based model (C.)
  - Structural model (D.)
- Structural model: combine separate modules by instantiation within a **module**.

J.J. Shann HDL-118

118

HDL Example 5.7: Binary Counter (Moore Machine)

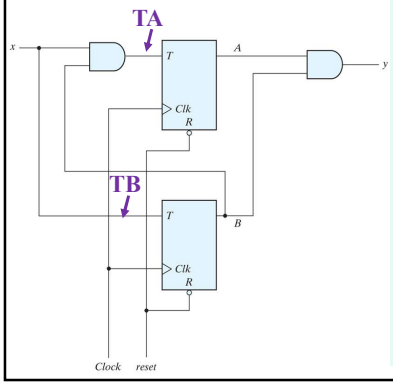
- HDL Example 5.7(a)
  - *State-diagram-based* model
  - p.288, Fig 5.20(b)




```
module Moore_Model_Fig_5_20 (
   output  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 S0
      else case (state)
         S0: if (x_in) state <= S1; else state <= S0;
         S1: if (x_in) state <= S2; else state <= S1;
         S2: if (x_in) state <= S3; else state <= S2;
         S3: if (x_in)   state <= S0; else state <= S3;
      endcase

   assign y_out = (state == S3);    //Output of f-fs
endmodule
```

119

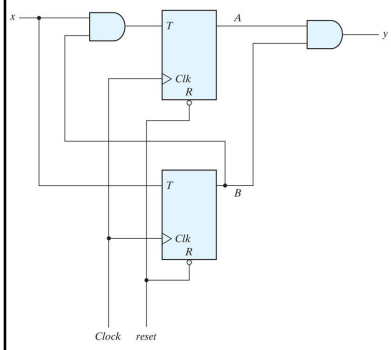
- HDL Example 5.7(b):
  - *Structural* model
  - p.288, Fig 5.20(a)




```
module Moore_Model_STR_Fig_5_20 (
   output  y_out, A, B,
   input   x_in, clock, reset
);
   wire TA, TB;
// Flip-flop input equations
   assign TA = x_in & B;
   assign TB = x_in;
//output equation
   assign y_out = A & B;
// Instantiate Toggle flip-flops
   Toggle_flip_flop_3 M_A (A, TA, clock, reset);
   Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule
```


- T flip-flop module for the circuit

```
module Moore_Model_STR_Fig_5_20 (
   output  y_out, A, B,
   input   x_in, clock, reset
);
   wire TA, TB;
// Flip-flop input equations
   assign TA = x_in & B;
   assign TB = x_in;
//output equation
   assign y_out = A & B;
// Instantiate Toggle flip-flops
   Toggle_flip_flop_3 M_A (A, TA, clock, reset);
   Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule
```




```
module Toggle_flip_flop_3 (Q, T, Clk, rst);
   output   Q;
   input    T, Clk, rst;
   reg      Q;
   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;
      else      Q <= Q ^ T;
endmodule
```

J.J. Shann HDL-121

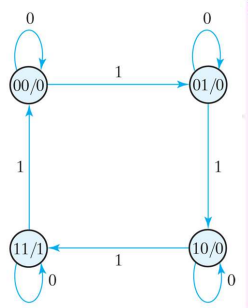
- Testbench of HDL Example 5.7(a)(b):

```
module Moore_Model_Fig_5_20 (
   output  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 S0
     else case (state)
        S0: if (x_in) state <= S1; else state <= S0;
        S1: if (x_in) state <= S2; else state <= S1;
        S2: if (x_in) state <= S3; else state <= S2;
        S3: if (x_in)   state <= S0; else state <= S3;
     endcase
   assign y_out = (state == S3);   //Output of f-fs
endmodule
```

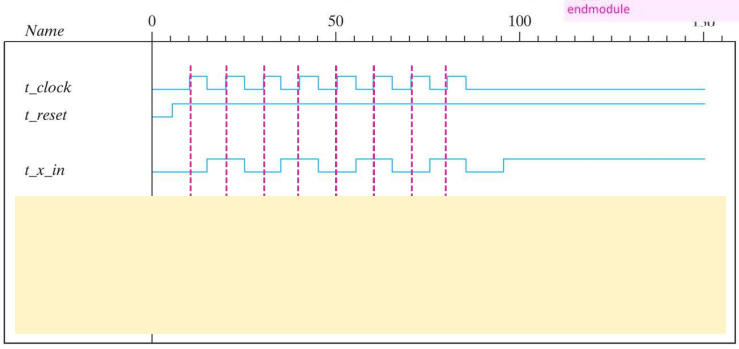
```
module Moore_Model_STR_Fig_5_20 (
   output  y_out, A, B,
   input   x_in, clock, reset
);
   wire TA, TB;
// Flip-flop input equations
   assign TA = x_in & B;
   assign TB = x_in;
//output equation
   assign y_out = A & B;
// Instantiate Toggle flip-flops
   Toggle_flip_flop_3 M_A (A, TA, clock, reset);
   Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule
```

```
module t_Moore_Fig_5_20;
   wire    t_y_out_2, t_y_out_1;
   reg     t_x_in, t_clock, t_reset;

   Moore_Model_Fig_5_20 M1
      (t_y_out_1, t_x_in, t_clock, t_reset);
   Moore_Model_STR_Fig_5_20 M2
      (t_y_out_2, A, B, t_x_in, t_clock, t_reset);

   initial #200 $finish;

   initial begin
      t_reset = 0;
      t_clock = 0;
      #5  t_reset = 1;
      repeat (16)
         #5  t_clock = ~t_clock;
   end

   initial begin
      t_x_in = 0;
      #15 t_x_in = 1;
      repeat (8)
         #10 t_x_in = ~t_x_in;
   end
endmodule
```


- Simulation output of HDL Example 5.7:
  - p.312, Fig 5.24




```
module t_Moore_Fig_5_20;
   wire    t_y_out_2, t_y_out_1;
   reg     t_x_in, t_clock, t_reset;

   Moore_Model_Fig_5_20 M1
      (t_y_out_1, t_x_in, t_clock, t_reset);
   Moore_Model_STR_Fig_5_20 M2
      (t_y_out_2, A, B, t_x_in, t_clock, t_reset);

   initial #200 $finish;

   initial begin
      t_reset = 0;
      t_clock = 0;
      #5  t_reset = 1;
      repeat (16)
         #5  t_clock = ~t_clock;
   end

   initial begin
      t_x_in = 0;
      #15 t_x_in = 1;
      repeat (8)
         #10 t_x_in = ~t_x_in;
   end
endmodule
```



* Test the ckt thoroughly!

J.J. Shann HDL-123