

Modeling Sequential Circuits in Verilog

COE 202

Digital Logic Design

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

- ❖ Modeling Latches and Flip-Flops
- ❖ Blocking versus Non-Blocking Assignments
- ❖ Modeling Sequential Circuit Diagrams
- ❖ Modeling Mealy and Moore State Diagrams
- ❖ Writing Test Benches for Sequential Circuits
- ❖ Modeling Registers and Counters

Recall: Sensitivity List of always block

❖ Syntax:

```
always @(sensitivity list) begin
```

```
    procedural statements
```

```
end
```

- ❖ Sensitivity list is a list of signals: @(signal1, signal2, ...)
- ❖ The sensitivity list triggers the execution of the **always** block

When there is a ***change of value in any listed signal***

Otherwise, the **always** block does nothing until another change occurs on a signal in the sensitivity list

Guidelines for Sensitivity List

- ❖ For combinational logic, the sensitivity list must include **ALL** the signals **that are read** inside the **a1ways** block
Combinational logic can also use: `@(*)` or `@*`
- ❖ For sequential logic, the sensitivity list may not include all the signals that are read inside the **a1ways** block
- ❖ For **edge-triggered** sequential logic use:
a1ways `@(posedge signal1, negedge signal2, ...)`
- ❖ The **positive edge** or **negative edge** of each signal can be specified in the sensitivity list

Modeling a D Latch with Enable

```
// Modeling a D Latch with Enable and output Q
// Output Q must be of type reg
// Notice that the if statement does NOT have else
// If Enable is 0, then value of Q does not change
// The D_latch stores the old value of Q

module D_latch (input D, Enable, output reg Q);
    always @(D, Enable)
        if (Enable) Q <= D; // Non-blocking assignment
        // No else means a latch (Q does not change)
endmodule
```

Modeling a D-type Flip-Flop

```
// Modeling a D Flip-Flop with outputs Q and Qbar
module D_FF (input D, Clk, output reg Q, Qbar);
    // Q and Qbar change at the positive edge of Clk
    // Notice that always is NOT sensitive to D input
    always @(posedge Clk)
    begin
        Q <= D;           // Non-blocking assignment
        Qbar <= ~D;       // Non-blocking assignment
    end
endmodule
```

Negative-Edge Triggered D-type Flip-Flop

```
// Modeling a Negative-Edge Triggered D Flip-Flop
// The only difference is the negative edge of Clk
module D_FF2 (input D, Clk, output reg Q, Qbar);
    // Q and Qbar change at the negative edge of Clk
    always @(negedge Clk)
    begin
        Q <= D;           // Non-blocking assignment
        Qbar <= ~D;       // Non-blocking assignment
    end
endmodule
```

D-type Flip-Flop with Synchronous Reset

```
// Modeling a D Flip-Flop with Synchronous Reset input
module D_FF3 (input D, Clk, Reset, output reg Q, Qbar);
    // always block is NOT sensitive to Reset or D
    // Updates happen only at positive edge of Clk
    // Reset is Synchronized with Clk
    always @(posedge Clk)
        if (Reset)
            {Q, Qbar} <= 2'b01;    // Non-blocking assignment
        else
            {Q, Qbar} <= {D, ~D}; // Non-blocking assignment
endmodule
```


D-type Flip-Flop with Asynchronous Reset

```
// Modeling a D Flip-Flop with Asynchronous Reset input
module D_FF4 (input D, Clk, Reset, output reg Q, Qbar);
    // Q and Qbar change at the positive edge of Clk
    // Or, at the positive edge of Reset
    // Reset is NOT synchronized with Clk
    always @(posedge Clk, posedge Reset)
        if (Reset)
            {Q, Qbar} <= 2'b01;    // Non-blocking assignment
        else
            {Q, Qbar} <= {D, ~D}; // Non-blocking assignment
endmodule
```

Procedural Assignment

- ❖ Procedural assignment is used inside a **procedural block** only
- ❖ Two types of procedural assignments:

- ❖ **Blocking assignment:**

variable = *expression*; // = operator

Variable is updated **before** executing next statement

Similar to an assignment statement in programming languages

- ❖ **Non-Blocking assignment:**

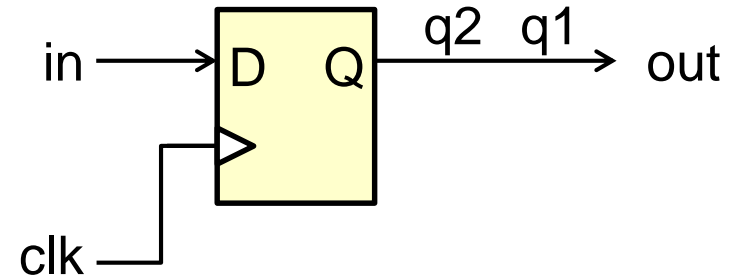
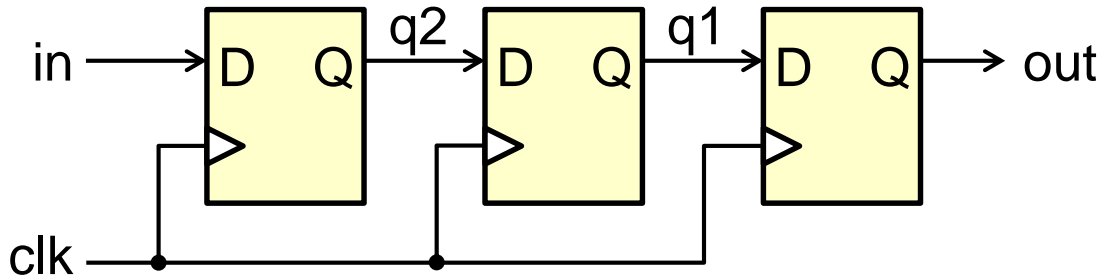
variable <= *expression*; // <= operator

Variable is updated **at the end** of the procedural block

Does not block the execution of next statements

Blocking versus Non-Blocking Assignment

Guideline: Use Non-Blocking Assignment for Sequential Logic



```
module nonblocking
(input in, clk, output reg out);
reg q1, q2;
always @ (posedge clk) begin
    q2 <= in;
    q1 <= q2;
    out <= q1;
end
endmodule
```

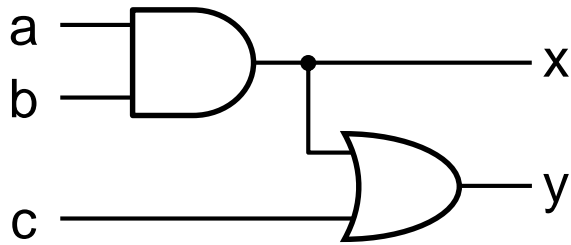
Read: in, q2, q1

Parallel Assignment at the end

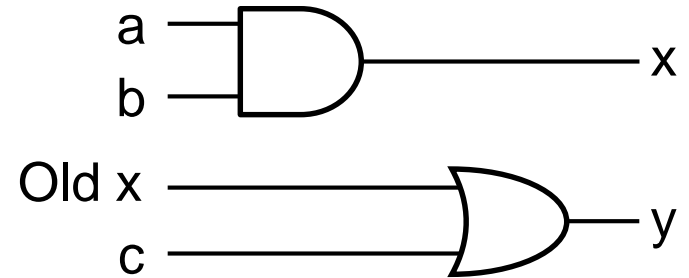
```
module blocking
(input in, clk, output reg out);
reg q1, q2;
always @ (posedge clk) begin
    q2 = in;
    q1 = q2; // q1 = in
    out = q1; // out = in
end
endmodule
```

Blocking versus Non-Blocking Assignment

Guideline: Use Blocking Assignment for Combinational Logic



Old x is Latched



```
module blocking
(input a,b,c, output reg x,y);
always @ (a, b, c) begin
x = a & b; // update x
y = x | c; // y = a&b | c;
end
endmodule
```

```
module nonblocking
(input a,b,c, output reg x,y);
always @ (a, b, c) begin
x <= a & b;
y <= x | c;
end
endmodule
```

Evaluate all expressions

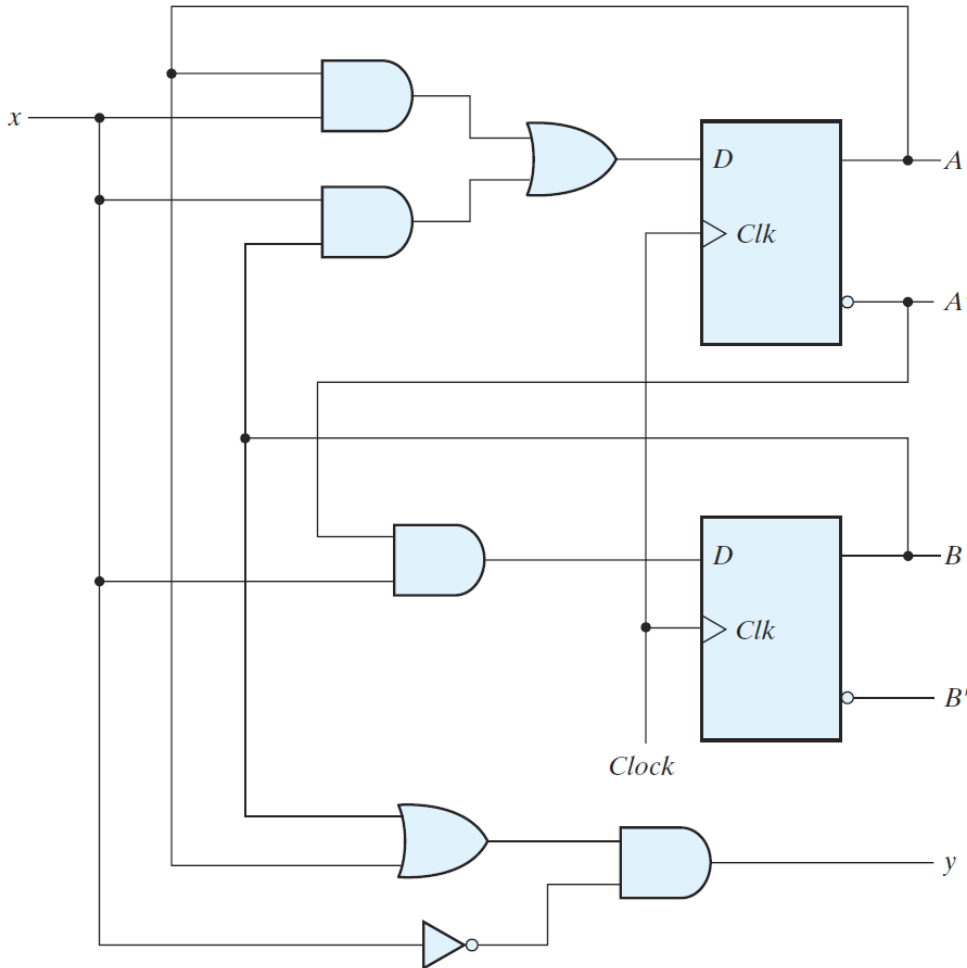
Parallel Assignment at the end

Verilog Coding Guidelines

1. When modeling **combinational** logic, use **blocking** assignments
2. When modeling **sequential** logic, use **non-blocking** assignments
3. When modeling **both sequential** and **combinational logic** within the same always block, use **non-blocking** assignments
4. Do **NOT** mix blocking with non-blocking assignments in the same always block
5. Do **NOT** make assignments to the same variable from more than one always block

Structural Modeling of Sequential Circuit

Modeling the Circuit Structure



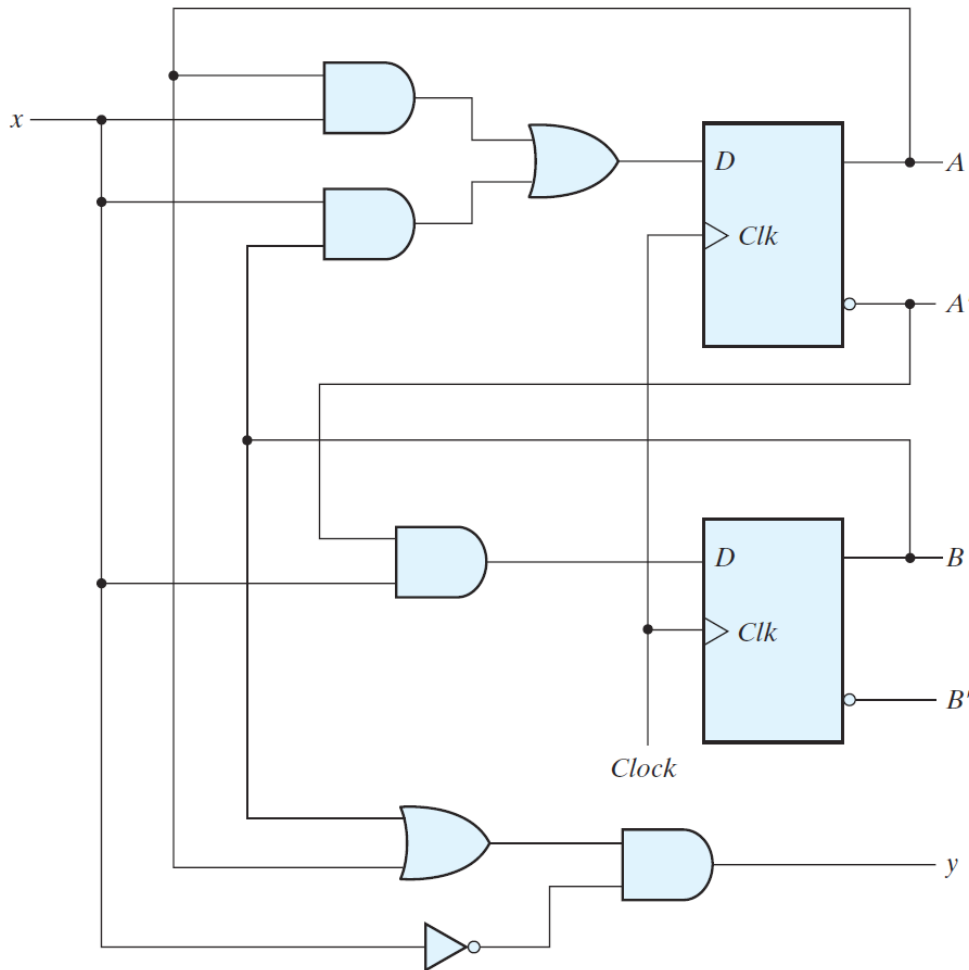
```
// Mixed Structural and assign
module Seq_Circuit_Structure
  (input x, Clock, output y);
  wire DA, DB, A, Ab, B, Bb;

  // Instantiate two D Flip-Flops
  D_FF FFA(DA, Clock, A, Ab);
  D_FF FFB(DB, Clock, B, Bb);

  // Next state and output logic
  assign DA = (A & x) | (B & x);
  assign DB = Ab & x;
  assign y = (A | B) & ~x;
endmodule
```

Behavioral Modeling of Sequential Circuit

Modeling the Circuit Behavior



```
module Seq_Circuit_Behavior
  (input x, Clock, output y);
  // reg A, B for the Flip-Flops
  reg A, B;
  wire DA, DB;
  // Modeling D FFs with always
  // Update A, B at positive edge
  // Non-Blocking assignment
  always @(posedge Clock)
    {A, B} <= {DA, DB};
  // Next state and output logic
  assign DA = (A & x) | (B & x);
  assign DB = ~A & x;
  assign y = (A | B) & ~x;
endmodule
```

Verifying Structural and Behavioral Models

```
module Seq_Circuit_TB; // Test Bench
    reg x, clk;
    wire y, z;

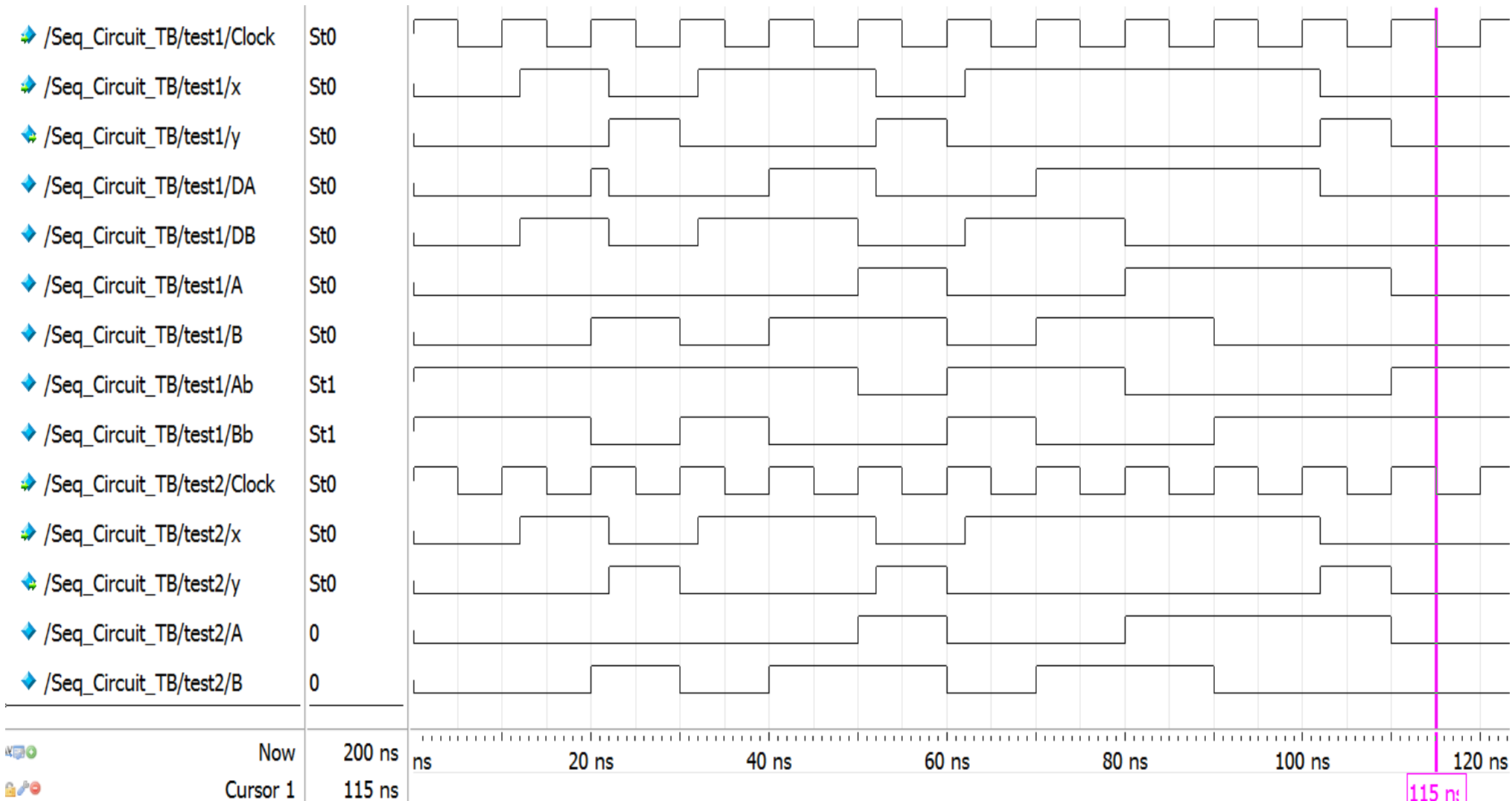
    // Instantiate structural and behavioral sequential circuits
    // Same inputs x and clk, but different outputs y and z
    Seq_Circuit_Structure test1 (x, clk, y);
    Seq_Circuit_Behavior test2 (x, clk, z);

    // Generate a clock with period = 10
    initial clk = 1;
    always #5 clk = ~clk;

    // Test sequence: x = 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, . . .
    initial begin
        x=0; #12 x=1; #10 x=0; #10 x=1; #20 x=0; #10 x=1; #40 x=0;
    end
endmodule
```

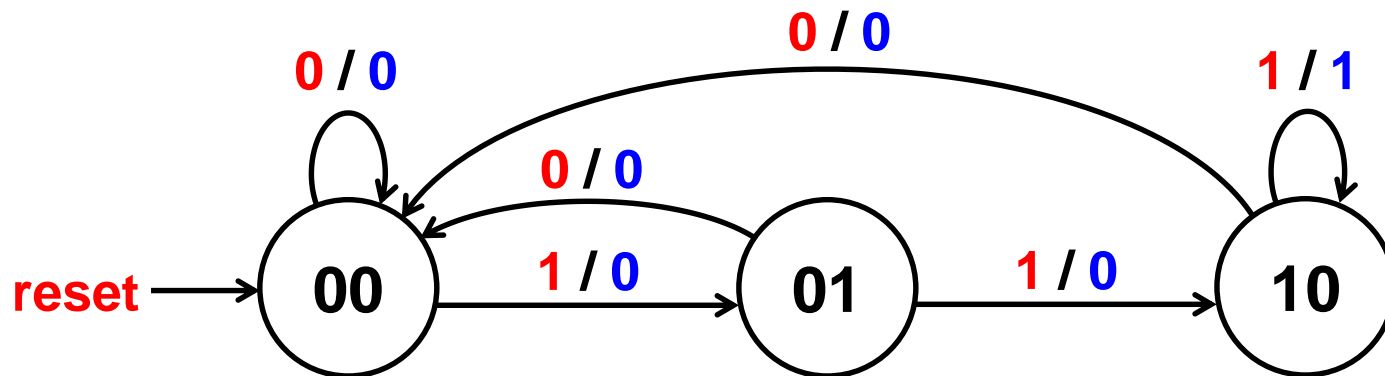

Simulation Waveforms

Structural and behavioral descriptions have identical waveforms



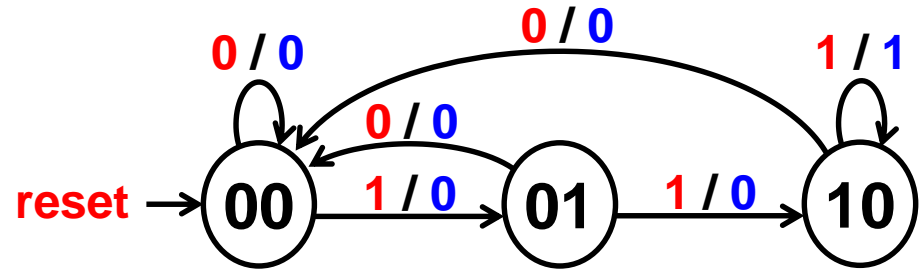
Modeling a State Diagram

- ❖ A state diagram can be modeled directly in Verilog
- ❖ Without the need of having the circuit implementation
- ❖ An example of a Mealy state diagram is shown below
- ❖ This is the state diagram of the **111** sequence detector
- ❖ State assignment: $S_0 = 00$, $S_1 = 01$, and $S_2 = 10$



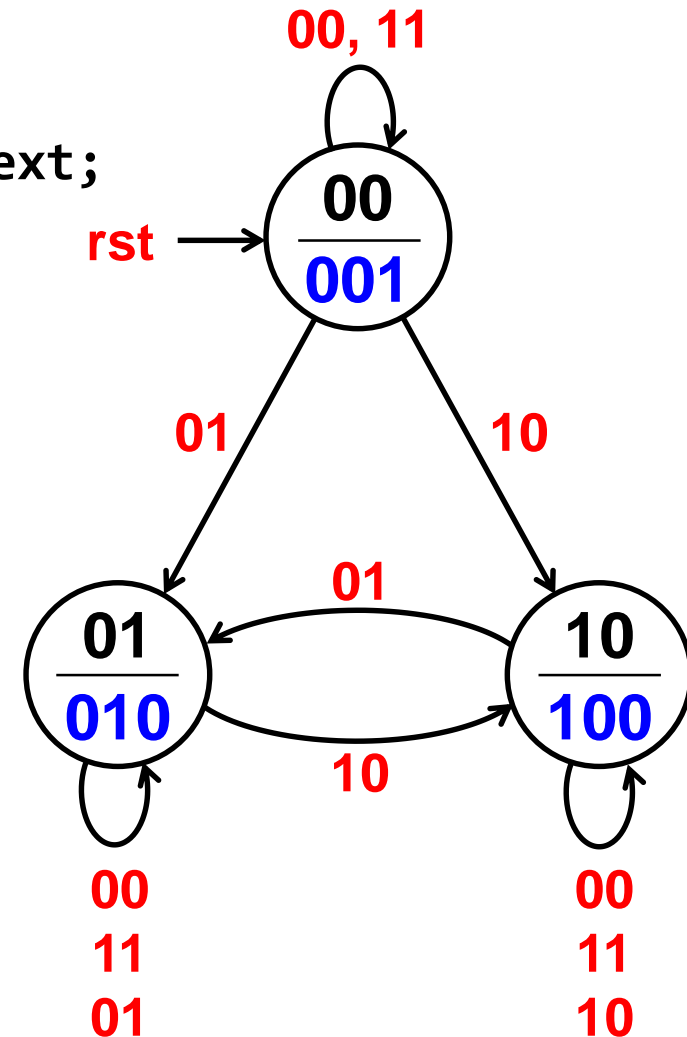
Modeling a Mealy State Diagram

```
module Mealy_111_detector (input x, clock, reset, output z);
  reg [1:0] state, next_state;
  // Flip-flops with synchronous reset
  always @(posedge clock)
    if (reset) state <= 2'b00;
    else state <= next_state;
  // Next state logic
  always @(*)
    case(state)
      2'b00: next_state = (x==1) ? 2'b01 : 2'b00;
      2'b01: next_state = (x==1) ? 2'b10 : 2'b00;
      2'b10: next_state = (x==1) ? 2'b10 : 2'b00;
      default: next_state = 2'b00; // Unused state
    endcase
  // Output logic depends on present state and input x
  assign z = (state == 2'b10) & x;
endmodule
```



Modeling a Moore State Diagram

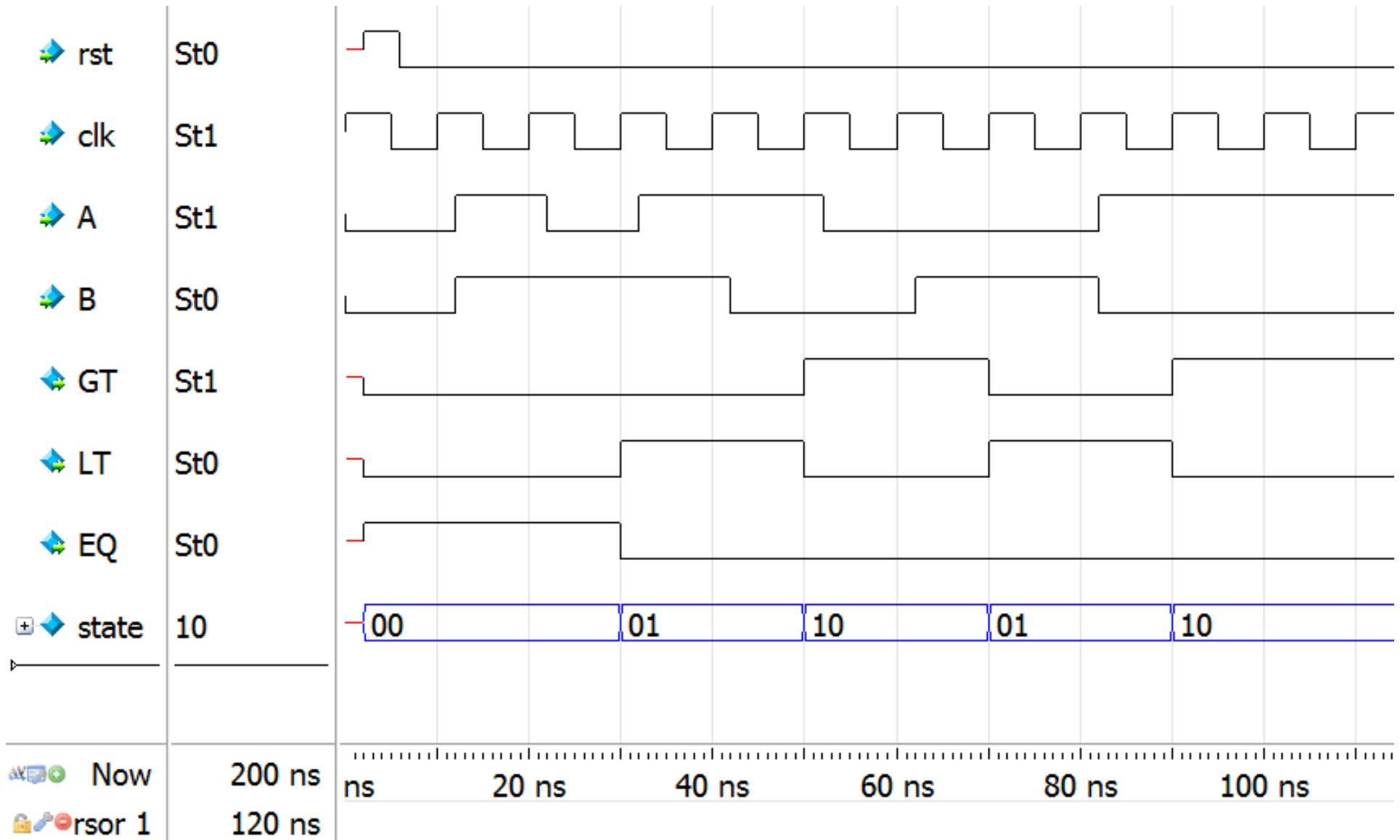
```
module Moore_Comparator (input A, B, clk, rst, output GT, LT, EQ);
  reg [1:0] state, next;
  // Flip-flops with asynchronous reset
  always @(posedge clk, posedge rst)
    if (rst) state <= 'b00; else state <= next;
  // Next state logic (combinational)
  always @(*)
  case (state)
    'b00: next = ({A,B}=='b01)?'b01:
              ({A,B}=='b10)?'b10:'b00;
    'b01: next = ({A,B}=='b10)?'b10:'b01;
    'b10: next = ({A,B}=='b01)?'b01:'b10;
    default: next = 'b00; // Unused state
  endcase
  // Output logic
  assign {GT, LT} = state;
  assign EQ = ~(GT|LT);
endmodule
```



Test Bench for the Moore Comparator

```
module Moore_Comparator_TB; // Test Bench
    reg A, B, clk, rst;
    wire GT, LT, EQ;
    Moore_Comparator test (A, B, clk, rst, GT, LT, EQ);
    // Reset pulse
    initial begin #2 rst = 1; #4 rst = 0; end
    // Generate clock with period = 10
    initial clk = 1;
    always #5 clk = ~clk;
    // Generate input test sequence
    initial begin
        {A,B}='b00; #12 {A,B}='b11; #10 {A,B}='b01; #10 {A,B}='b11;
        #10 {A,B}='b10; #10 {A,B}='b00; #10 {A,B}='b01; #20 {A,B}='b10;
    end
endmodule
```

Moore Comparator Waveforms

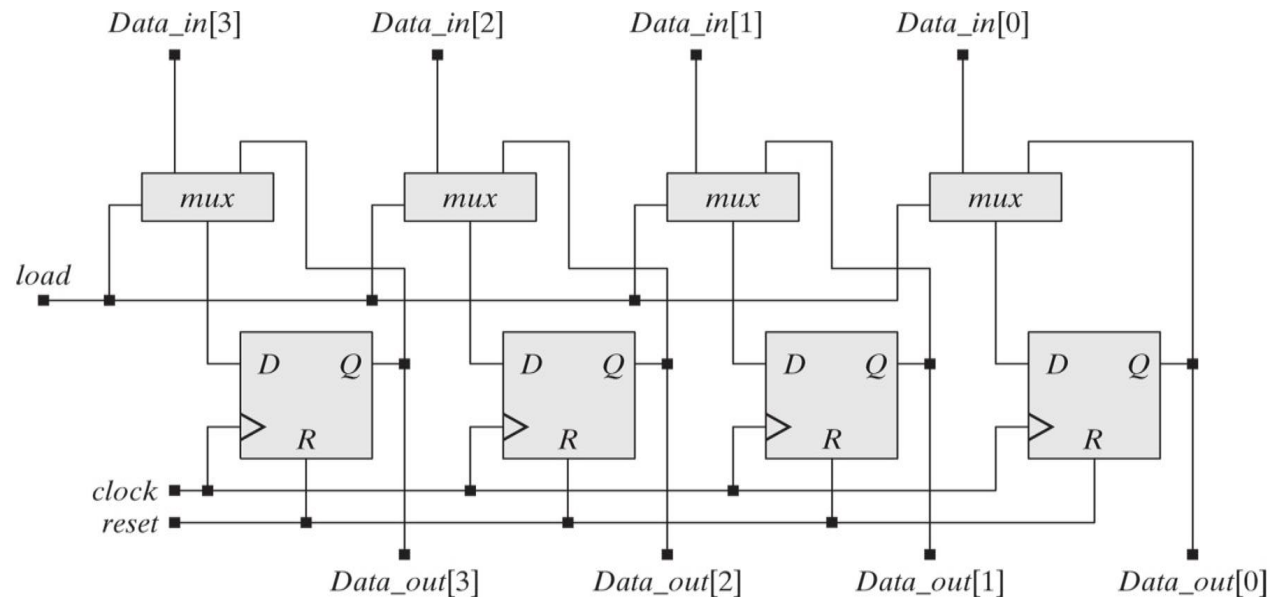


Modeling a Register with Parallel Load

```
module Register #(parameter n = 4)
  (input [n-1:0] Data_in, input load, clock, reset,
   output reg [n-1:0] Data_out);

  always @(posedge clock, posedge reset) // Asynchronous reset
    if (reset) Data_out <= 0;
    else if (load) Data_out <= Data_in;

endmodule
```



Modeling a Shift Register

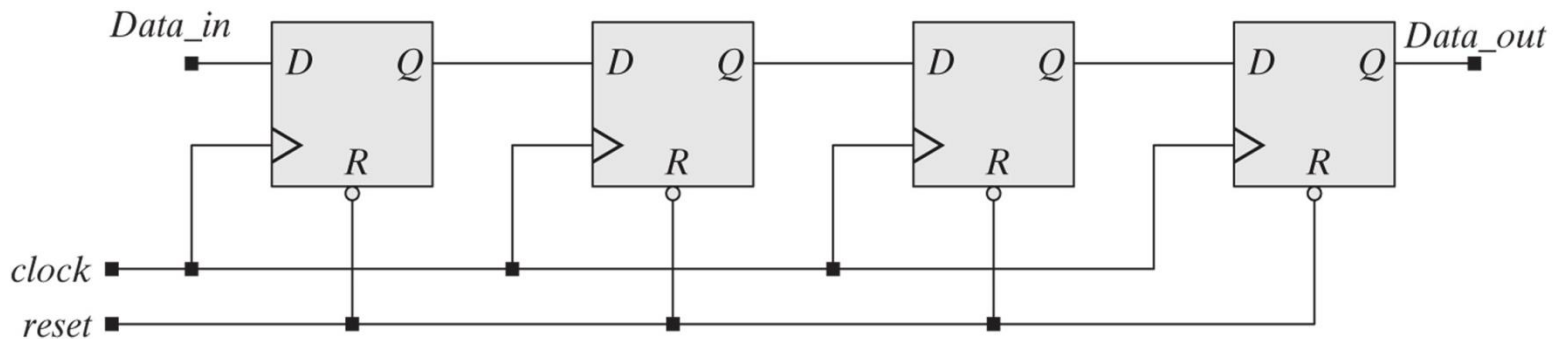
```
module Shift_Register #(parameter n = 4)
  (input Data_in, clock, reset, output Data_out);

  reg [n-1:0] Q;

  assign Data_out = Q[0];           // Serial Output

  always @(posedge clock, negedge reset) // Asynchronous reset
    if (!reset) Q <= 0;           // Active Low reset
    else Q <= {Data_in, Q[n-1:1]}; // Shifts to the right

endmodule
```



Modeling a Counter with Parallel Load

```
module Counter_with_Load #(parameter n = 4) // n-bit counter
  ( input [n-1:0] D, input Load, EN, clock,
    output reg [n-1:0] Q, output Cout);
```

```
  assign Cout = (&Q) & EN;
```

```
  // Sensitive to Positive-edge
```

```
  always @(posedge clock)
```

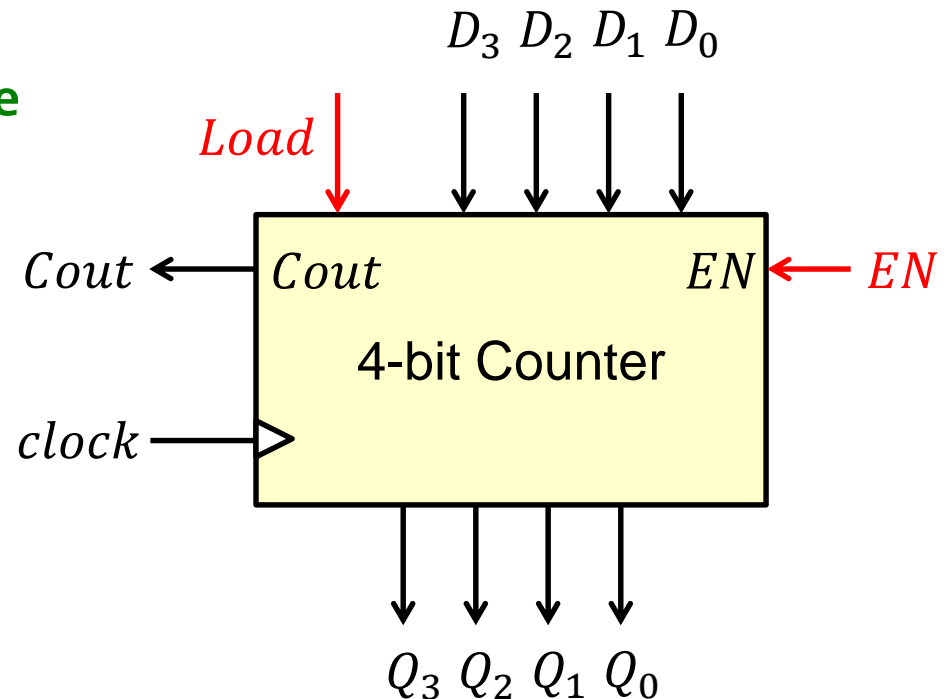
```
    if (Load)
```

```
      Q <= D;
```

```
    else if (EN)
```

```
      Q <= Q + 1;
```

```
  endmodule
```



Modeling a Generic Up-Down Counter

```
module Up_Down_Counter #(parameter n = 16) // n-bit counter
```

```
( input [n-1:0] Data_in,  
  input [1:0] f, input reset, clock,  
  output reg [n-1:0] Count );
```

f = 0 → Disable counter
f = 1 → Count up
f = 2 → Count down
f = 3 → Load counter

```
// Asynchronous reset
```

```
always @(posedge clock, posedge reset)
```

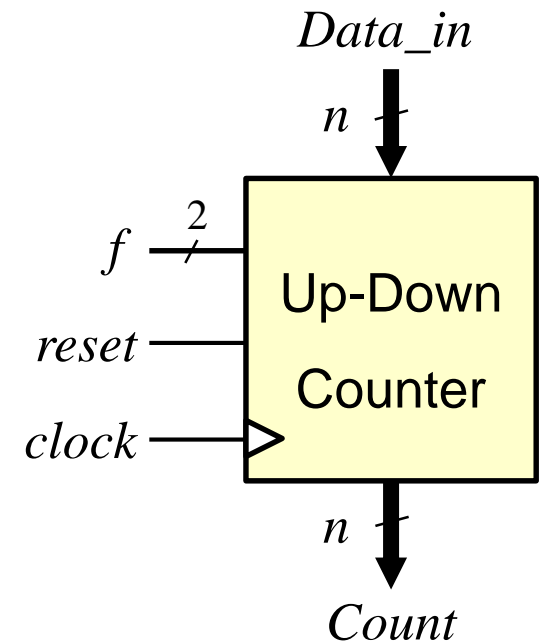
```
  if (reset) Count <= 0;
```

```
  else if (f == 1) Count <= Count + 1;
```

```
  else if (f == 2) Count <= Count - 1;
```

```
  else if (f == 3) Count <= Data_in;
```

```
endmodule
```



Test Bench for the Up-Down Counter

```
module Up_Down_Counter_TB; // Test Bench
    reg [7:0] Data_in; reg [1:0] f; reg rst, clk; wire [7:0] Count;
    // Instantiate an 8-bit test counter
    Up_Down_Counter #(8) test (Data_in, f, rst, clk, Count);
    // Initialize Data_in (in hexadecimal)
    initial Data_in = 8'h2A;
    // Generate reset pulse
    initial begin #2 rst=1; #4 rst=0; end
    // Generate clock (cycle = 10)
    initial clk = 1; always #5 clk = ~clk;
    // Generate function sequence
    initial begin
        #2 f=3; #10 f=1; #30 f=2; #10 f=0; #10 f=1;
    end
endmodule
```

Up-Down Counter Waveforms

