

Blocking versus Nonblocking Assignment

- Blocking: Evaluates right-hand side (RHS) then makes left-hand side (LHS) assignment before moving on to next statement

```
always @(posedge clk)
begin
    B = A;
    C = B;
    D = C; // Result: D = A
end
```

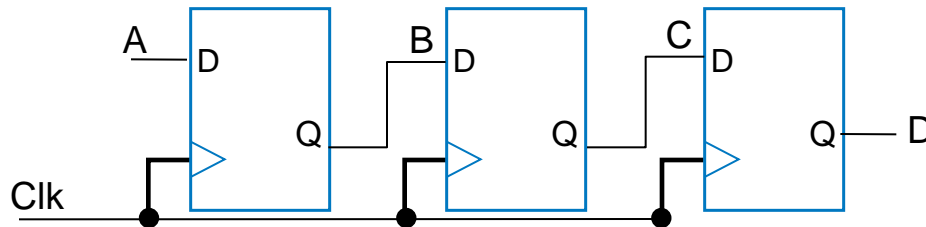
- Nonblocking: Evaluates each right-hand side without waiting to make left-hand side assignment. Then makes all LHS assignments concurrently

```
always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C; // Result: D = value of C just before the clock pulse
            // it will take two more clock pulses before D = A
end
```

– see circuit on next slide

Nonblocking Realization

```
always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C; // Result: D = value of C just before the clock pulse
            // it will take two more clock pulses before D = A
end
```



Blocking versus Nonblocking Example

```
always @(posedge clk)
begin
    B = A;
    C = B;
    D <= C;
end
```

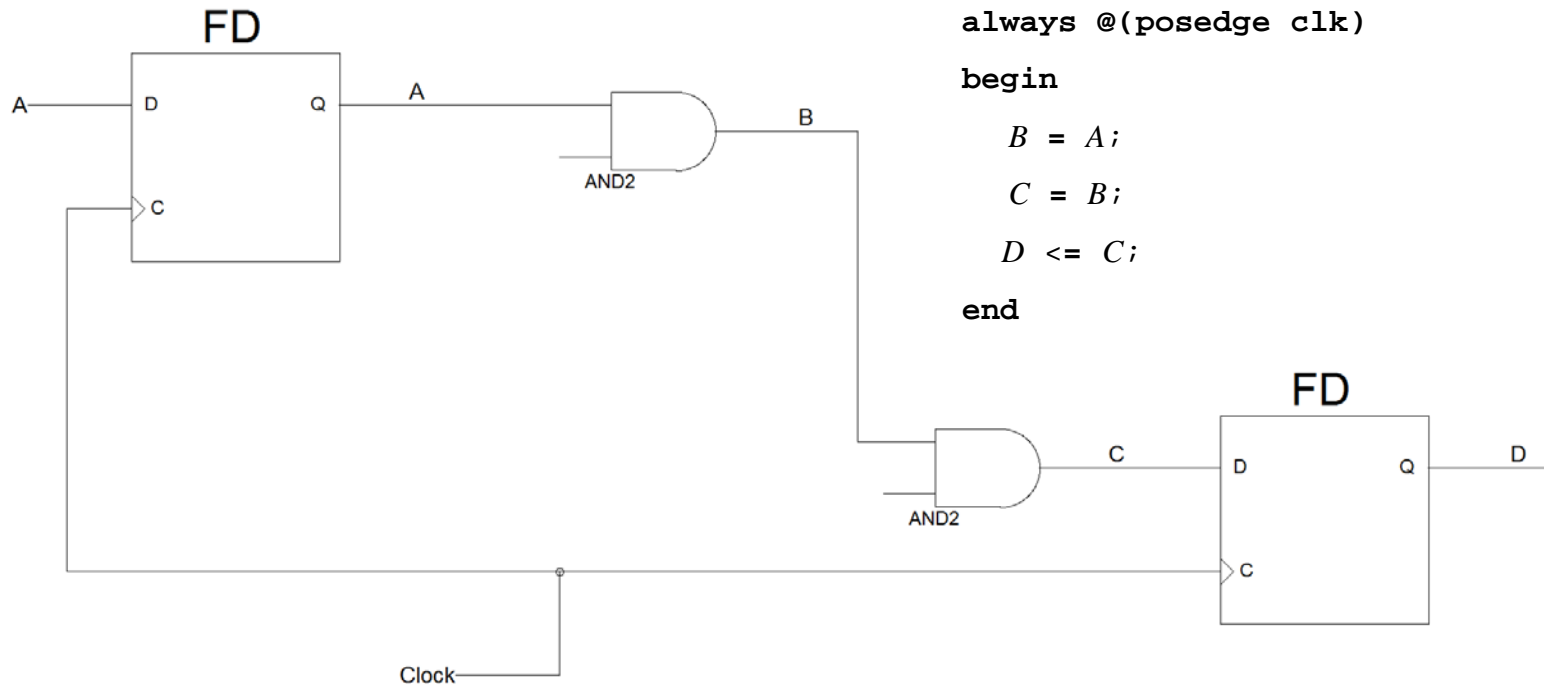
Just prior to the clock edge, A = 5, B = 10, C = 15, D = 20

After the clock edge, B = 5, C = 5, D = 20

see next slide for circuit



Circuit for Previous Slide

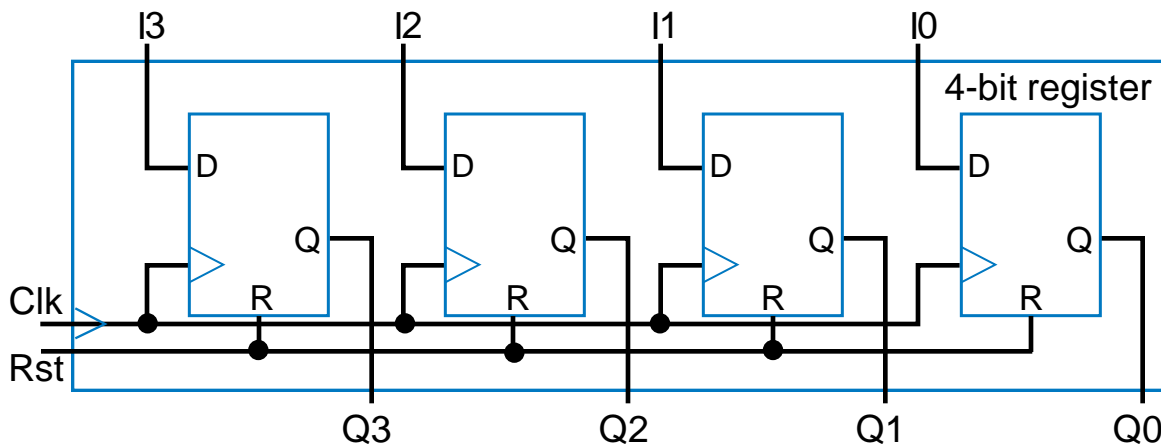
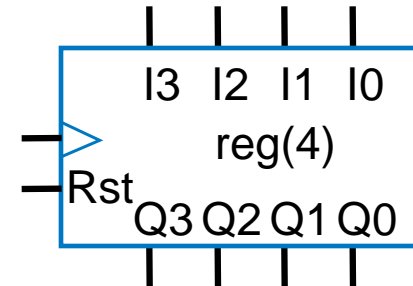


Blocking and Nonblocking Tips

- A variable assigned to a value by a nonblocking (\leq) assignment must be declared as a registered (reg) variable.
- It is strongly recommended that
 - edge-sensitive operations, i.e. synchronous, use nonblocking assignments
 - combinational logic be described by blocking assignments.

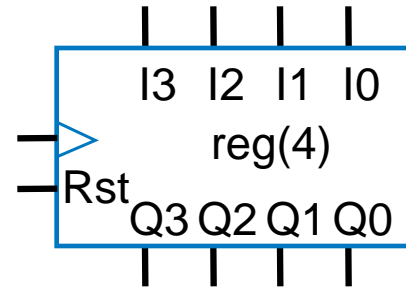
Recall Register Structure

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



Vectors (arrays) in Verilog

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



```
`timescale 1 ns/1 ns

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

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

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

```
module Reg4(I3,I2,I1,I0,Q3,...);
    input I3, I2, I1, I0;

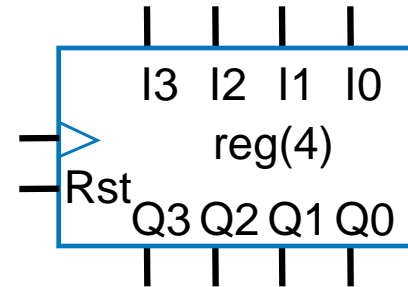
    I3 I2 I1 I0

    module Reg4(I, Q, ...);
        input [3:0] I;

        I: [ ][ ][ ][ ]
        I[3]I[2]I[1]I[0]
```

Numbers in Verilog

- Binary constant
 - 4'b0000
 - 4: size, in number of bits
 - 'b: binary base
 - 0000: binary value
- Other constant bases possible
 - d: decimal base, o: octal base, h: hexadecimal base
 - 12'hFA2
 - 'h: hexadecimal base
 - 12: 3 hex digits require 12 bits
 - FA2: hex value
 - Size is always in bits, and optional
 - 'hFA2 is OK
 - For decimal constant, size and 'd optional
 - 8'd255 or just 255
 - In previous uses like "A <= 1;" 1 and 0 are actually decimal numbers. 'b1 and 'b0 would explicitly represent bits
- Underscores may be inserted into value for readability
 - 12'b1111_1010_0010
 - 8_000_000



```
`timescale 1 ns/1 ns

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

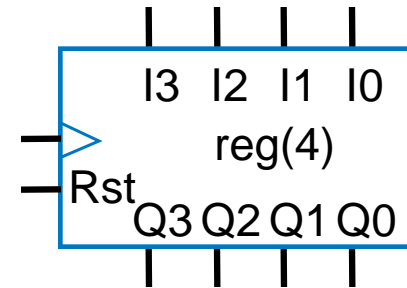
    input [3:0] I;
    output [3:0] Q;
    reg [3:0] Q;
    input Clk, Rst;

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



Edge Triggered Behavior

- Procedure's event control involves Clk input
 - Note I input is not in event list, thus, synchronous
 - "posedge Clk"
 - Event is not just any change on Clk, but specifically change from 0 to 1 (positive edge)
 - negedge also possible
- Process has synchronous reset
 - Resets output Q only on rising edge of Clk
- Process writes output Q
 - Q declared as reg variable, thus stores value, too



```
`timescale 1 ns/1 ns

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

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

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



Testbench

- reg/wire declarations and module instantiation similar to previous testbenches
- Module uses two procedures
 - One generates 20 ns clock⁽¹⁾
 - 0 for 10 ns, 1 for 10 ns
 - Note: always procedure repeats
 - Other provides values for inputs Rst and I (i.e., vectors)
 - initial procedure executes just once, does not repeat
 - (more on next slide)

¹Note, the time delay is behavioral and not structural, i.e., it can't be synthesized.

```
`timescale 1 ns/1 ns //time unit / precision
```

```
module Testbench();
```

```
    reg [3:0] I_s;    // inputs
    reg Clk_s, Rst_s; // inputs
    wire [3:0] Q_s;   // outputs
```

```
    Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);
```

```
    // Clock Procedure
```

```
    always begin
```

```
        Clk_s <= 0;
```

```
        #10; // after 10 time increments(1 ns)
```

```
        Clk_s <= 1;
```

```
        #10;
```

```
    end // Note: Procedure repeats
```

```
    // Vector Procedure
```

```
    initial begin // run once
```

```
        Rst_s <= 1; // initialize Rst_s
```

```
        I_s <= 4'b0000;
```

```
        @(posedge Clk_s); //wait for clock
```

```
        #5 Rst_s <= 0; // wait 5ns, clear Rst_s
```

```
        I_s <= 4'b0000; // set up D inputs
```

```
        @(posedge Clk_s);
```

```
        #5 Rst_s <= 0;
```

```
        I_s <= 4'b1010;
```

```
        @(posedge Clk_s);
```

```
        #5 Rst_s <= 0;
```

```
        I_s <= 4'b1111;
```

```
    end
```

```
endmodule
```



Register Behavior

Testbench

- Variables/nets can be shared between procedures
 - Only one procedure should write to variable
 - Variable can be read by many procedures
 - Clock procedure writes to Clk_s
 - Vector procedures reads Clk_s
- Event control "@(posedge Clk_s)"
 - May be prepended to statement to synchronize execution with event occurrence
 - Statement may be just ";" as in example
 - In previous examples, the "statement" was a sequential block (begin-end)
 - Test vectors thus don't include the clock's period hard coded
- Care taken to change input values away from clock edges

```

`timescale 1 ns/1 ns //

module Testbench();

    reg [3:0] I_s;
    reg Clk_s, Rst_s;
    wire [3:0] Q_s;

    Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);

    // Clock Procedure
    always begin
        Clk_s <= 0;
        #10;
        Clk_s <= 1;
        #10;
    end // Note: Procedure repeats

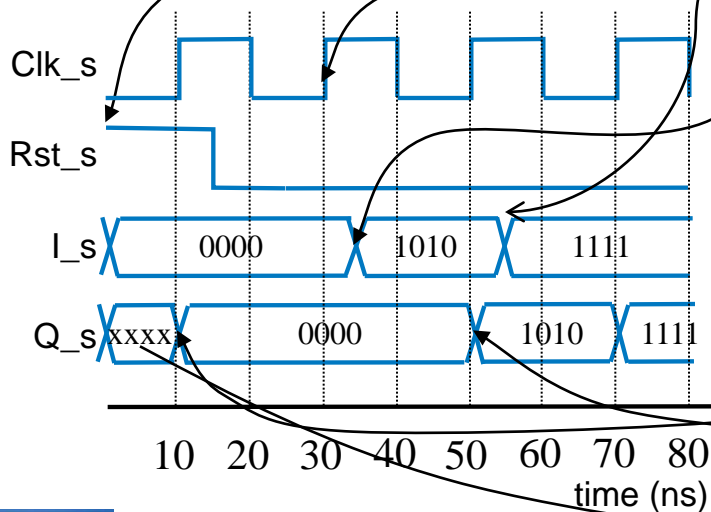
    // Vector Procedure
    initial begin
        Rst_s <= 1;
        I_s <= 4'b0000;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        I_s <= 4'b0000;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        I_s <= 4'b1010;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        I_s <= 4'b1111;
    end
endmodule

```



Testbench waveforms

- Simulation results
 - Note that Q_s updated only on rising clock edges
 - Note Q_s thus unknown until first clock edge
 - Q_s is reset to "0000" on first clock edge



```

...
// Vector Procedure
initial begin
    Rst_s <= 1;
    I_s <= 4'b0000;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b0000;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b1010;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b1111;
end

```

note 5 unit delay

```

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

```

Remember that Q_s is connected to Q, and I_s to I, in the testbench

Initial value of a bit is the unknown value x

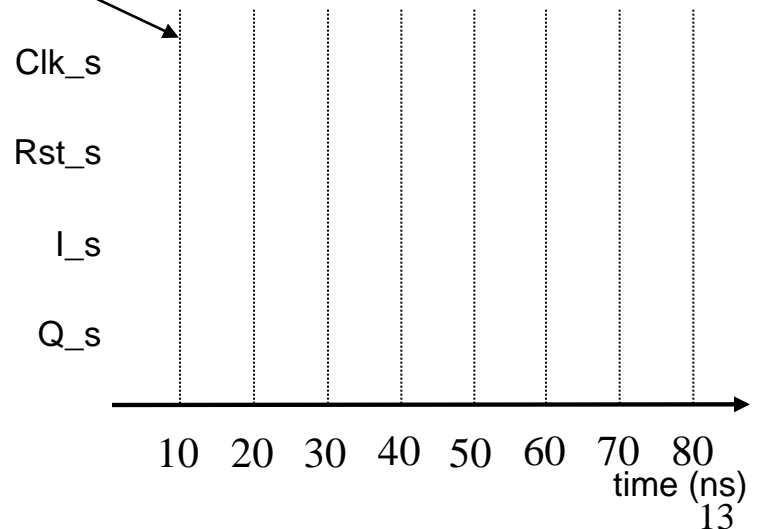


Common Pitfalls

- Using "always" instead of "initial" procedure
 - Causes endless execution
- Not including any delay control or event control in an always procedure
 - May cause infinite loop in the simulator
 - Simulator executes those statements over and over, never executing statements of another procedure
 - Simulation time can never advance
 - Symptom – Simulator appears to just hang, generating no waveforms


```
// Vector Procedure
always begin
    Rst_s <= 1;
    I_s <= 4'b0000;
    @(posedge Clk_s);
    ...
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b1111;
end
```

```
// Vector Procedure
always begin
    Rst_s <= 1;
    I_s <= 4'b0000;
end
```



Common Pitfalls

- Not initializing all module inputs
 - May cause undefined outputs
 - Or simulator may initialize to default value. Switching simulators may cause design to fail.
 - *Tip:* Immediately initialize all module inputs when first writing procedure



```
// Vector Procedure
always begin
    Rst_s <= 1;
    I_s <= 4'b0000;
    @(posedge Clk_s);
    ...
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b1111;
end
```



Common Pitfalls

- Forgetting to explicitly declare an identifier as a wire in a port connection
 - e.g., Q_s
 - Verilog **implicitly declares** identifier as a net of the default net type, typically a *one-bit* wire
 - Intended as shortcut to save typing for large circuits
 - May not give warning message during compilation
 - Works fine if a one-bit wire was desired
 - But may be mismatch – in this example, the wire should have been four bits, not one bit
 - Unexpected simulation results
 - Always explicitly declare wires
 - Best to avoid use of Verilog's implicit declaration shortcut

```
`timescale 1 ns/1 ns

module Testbench();

    reg [3:0] I_s;
    reg Clk_s, Rst_s;
    wire [3:0] Q_s;

    Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);

    ...
endmodule
```





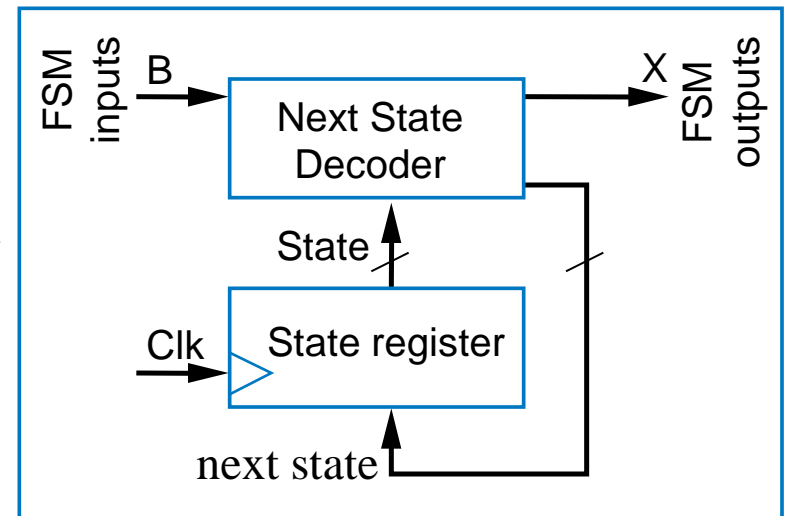
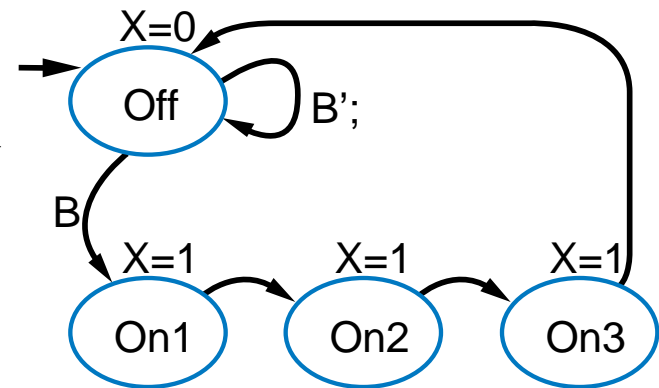
Finite-State Machines (FSMs)—Sequential Behavior Model



Finite-State Machines (FSMs)— Review

- **Finite-state machine (FSM)** is a common model of sequential behavior
 - Example: If $B=1$, hold $X=1$ for 3 clock cycles
 - Note: Transitions implicitly ANDed with rising clock edge
 - Implementation model has two parts:
 - State register
 - Combinational logic
 - HDL model will reflect those two parts

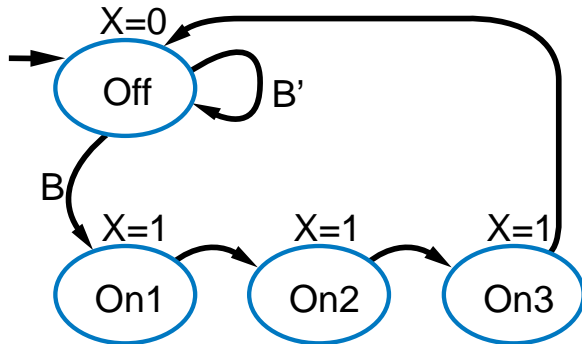
Inputs: B; Outputs: X



Finite-State Machines (FSMs)—Sequential Behavior

Modules with Multiple Procedures and Shared Variables

Inputs: B; Outputs: X



```
`timescale 1 ns/1 ns
```

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

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

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

```
    reg [1:0] State, StateNext;
```

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

```
    ...
    S_On1: begin
        X <= 1;
        StateNext <= S_On2;
    end
    S_On2: begin
        X <= 1;
        StateNext <= S_On3;
    end
    S_On3: begin
        X <= 1;
        StateNext <= S_Off;
    end
endcase
end

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

- Code will be explained on following slides



Parameters

- **parameter** declaration
 - Not a variable or net, but rather a *constant*
 - A constant is a value that must be initialized, and that cannot be changed within the module's definition
 - Example:
 - *S_Off*, *S_On1*, *S_On2*, *S_On3*
 - Correspond to FSM's states
 - Should be initialized to unique values

```
`timescale 1 ns/1 ns

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

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

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

    reg [1:0] State, StateNext;

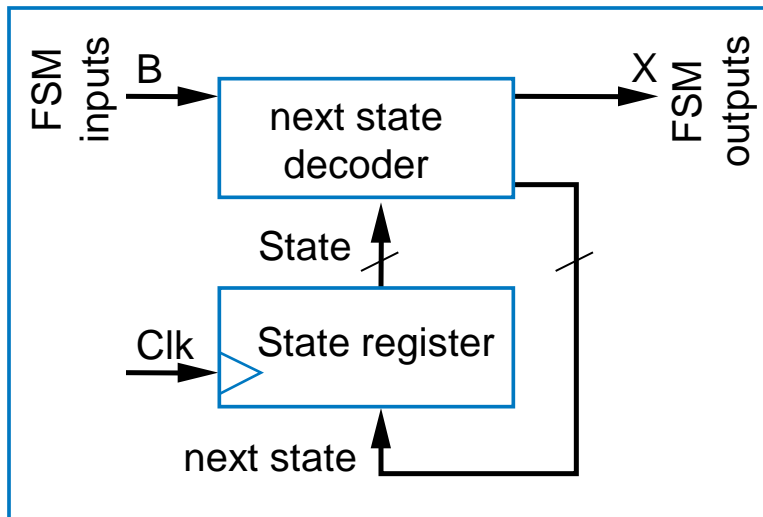
    // CombLogic
    always @(State, B) begin
        ...
    end

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



Finite-State Machines (FSMs)—Sequential Behavior

- Module has two procedures
 - One procedure for combinational logic (next state decoder)
 - One procedure for state register
 - But it's still a behavioral description



```
`timescale 1 ns/1 ns

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

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

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

    reg [1:0] State, StateNext;

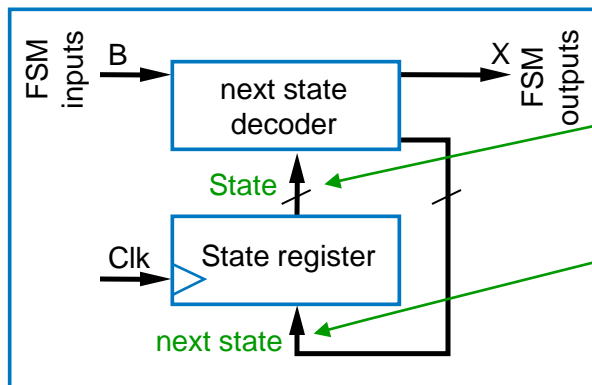
    // CombLogic
    always @(State, B) begin
        ...
    end

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



Finite-State Machines (FSMs)—Sequential Behavior

- Module declares two reg variables
 - *State*, *StateNext* (holds value till clock edge)
 - Each is 2-bit vector (need two bits to represent four unique state values 0 to 3)
 - Variables are shared between CombLogic and StateReg procedures
- CombLogic procedure
 - Event control sensitive to *State* and input *B*
 - *B* is external, *State* is from internal StateReg
 - Will output *StateNext* and *X*
- StateReg procedure
 - Sensitive to *Clk* input
 - Will output *State*, which it stores



```
`timescale 1 ns/1 ns

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

    input B;
    output reg X;
    input Clk, Rst;
    // state assignments
    parameter S_Off = 0, S_On1 = 1,
              S_On2 = 2, S_On3 = 3;

    reg [1:0] State, StateNext;

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

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



Procedures with Case Statements

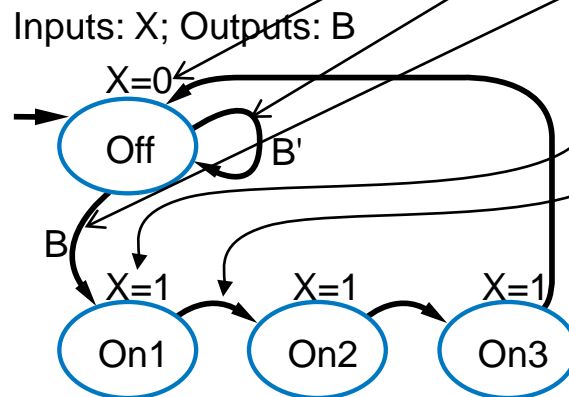
- Procedure may use **case** statement
 - Preferred over if-else-if when just one expression determines which statement to execute
 - **case** (expression)
 - Execute statement whose case item expression value matches case expression
 - *case item expression : statement*
 - *statement* is commonly a begin-end block, as in example
 - First case item expression that matches executes; remaining case items ignored
 - If no item matches, nothing executes
 - Last item may be "**default : statement**"
 - Statement executes if none of the previous items matched

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



Next State Decoder and Output Decoder

- FSM's CombLogic procedure
 - Case statement describes states
 - case (State)
 - Executes corresponding statement (often a begin-end block) based on State's current value
 - A state's statements consist of
 - Output decoder
 - Setting of next state (transitions)
- Ex: State is S_On1
 - Executes statements for state On1, jumps to endcase



```

reg [1:0] State, StateNext;

// CombLogic
always @(State, B) begin
    case (State)
        s_Off: begin
            X <= 0;
            if (B == 0)
                StateNext <= s_Off;
            else
                StateNext <= s_On1;
        end
        s_On1: begin
            X <= 1;
            StateNext <= s_On2;
        end
        s_On2: begin
            X <= 1;
            StateNext <= s_On3;
        end
        s_On3: begin
            X <= 1;
            StateNext <= s_Off;
        end
    endcase
end
    
```



Finite-State Machines (FSMs)—Sequential Behavior

- FSM StateReg Procedure

- Similar to 4-bit register

- Register for State is 2-bit vector reg variable

- Procedure has synchronous reset

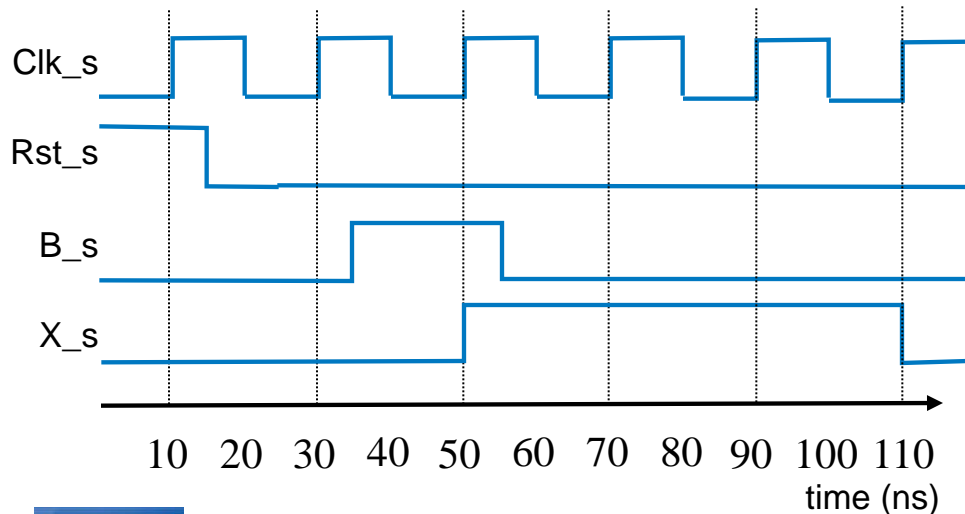
- Resets State to FSM's initial state, S_Off

```
...  
    parameter S_Off = 0, S_On1 = 1,  
              S_On2 = 2, S_On3 = 3;  
  
    reg [1:0] State, StateNext;  
  
    ...  
  
    // StateReg  
    always @(posedge Clk) begin  
        if (Rst == 1 )  
            State <= S_Off;  
        else  
            State <= StateNext;  
        end  
  
    ...
```



FSM Test Bench

- FSM testbench
 - First part of file (variable/net declarations, module instantiations) similar to before
 - Vector Procedure
 - Resets FSM
 - Sets FSM's input values (“test vectors”)
 - Waits for specific clock cycles
 - We observe the resulting waveforms to determine if FSM behaves correctly



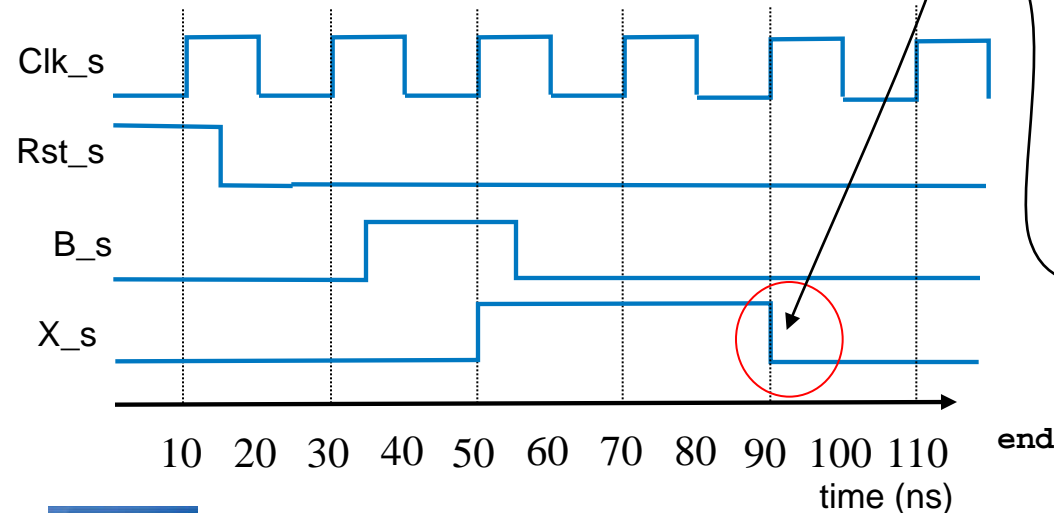
```
...
// Clock Procedure
always begin
    Clk_s <= 0;
    #10;
    Clk_s <= 1;
    #10;
end // Note: Procedure repeats

// Vector Procedure
initial begin
    Rst_s <= 1;
    B_s <= 0;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    @(posedge Clk_s);
    #5 B_s <= 1;
    @(posedge Clk_s);
    #5 B_s <= 0;
    @(posedge Clk_s);
    @(posedge Clk_s);
    @(posedge Clk_s);
end
endmodule
```



Self-Checking Testbench

- Reading waveforms is error-prone
- Create *self-checking testbench*
 - Use *if* statements to check for expected values
 - If a check fails, print error message
 - Ex: if X_s fell to 0 one cycle too early, simulation might output:
 - 95: Third X=1 failed



```
// Vector Procedure
initial begin
    Rst_s <= 1;
    B_s <= 0;
    @(posedge Clk_s);
    #5 if (X_s != 0)
        $display("%t: Reset failed", $time);
    Rst_s <= 0;
    @(posedge Clk_s);
    #5 B_s <= 1;
    @(posedge Clk_s);
    #5 B_s <= 0;
    if (X_s != 1)
        $display("%t: First X=1 failed", $time);
    @(posedge Clk_s);
    #5 if (X_s != 1)
        $display("%t: Second X=1 failed", $time);
    @(posedge Clk_s);
    #5 if (X_s != 1)
        $display("%t: Third X=1 failed", $time);
    @(posedge Clk_s);
    #5 if (X_s != 0)
        $display("%t: Final X=0 failed", $time);
end
```



Simulation \$Display System Procedure

- **\$display** – built-in Verilog system procedure for printing information to display during simulation
 - A **system procedure** interacts with the simulator and/or host computer system
 - To write to a display, read a file, get the current simulation time, etc.
 - Starts with **\$** to distinguish from regular procedures
- String argument is printed literally...
 - `$display("Hello")` will print "Hello"
 - Automatically adds newline character
- ...except when special sequences appear
 - **%t**: Display a time expression
 - Time expression must be next argument
 - **\$time** – Built-in system procedure that returns the current simulation time
 - *95: Third X=1 failed*

```
// Vector Procedure
initial begin
    Rst_s <= 1;
    B_s <= 0;
    @(posedge Clk_s);
    #5 if (X_s != 0)
        $display("%t: Reset failed", $time);
    Rst_s <= 0;
    @(posedge Clk_s);
    #5 B_s <= 1;
    @(posedge Clk_s);
    #5 B_s <= 0;
    if (X_s != 1)
        $display("%t: First X=1 failed", $time);
    @(posedge Clk_s);
    #5 if (X_s != 1)
        $display("%t: Second X=1 failed", $time);
    @(posedge Clk_s);
    #5 if (X_s != 1)
        $display("%t: Third X=1 failed", $time);
    @(posedge Clk_s);
    #5 if (X_s != 0)
        $display("%t: Final X=0 failed", $time);
end
```

