



# Top-Down Design – FSMs to Controller Structure



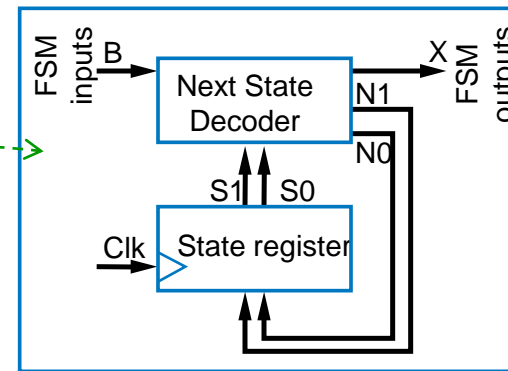
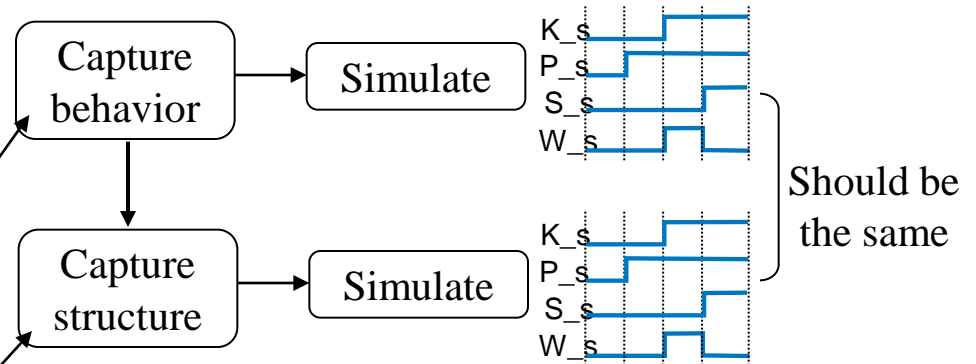
# Synthesis: Behavior to Structure

- It is easy to forget the desired end result of the Verilog description of the FSM is a digital logic circuit.
- The previous behavioral description for the FSM must be interpreted by the synthesizer to create the circuit structure



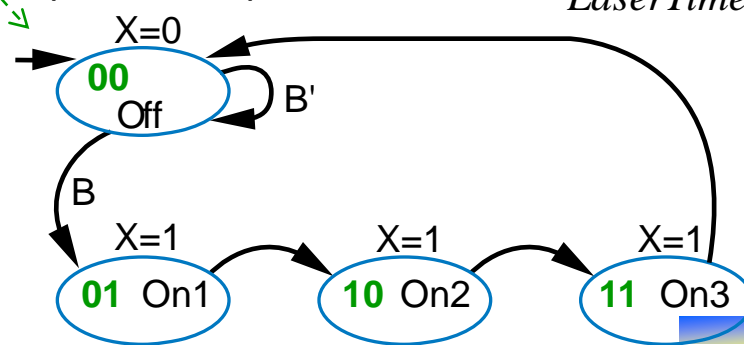
# Top-Down Design – FSMs to Controller Structure

- Recall from Chapter 2
  - **Top-down design**
    - *Capture behavior*, and simulate
    - *Capture structure (circuit)*, simulate again
    - Gets behavior right first, unfettered by complexity of creating structure
- Capture behavior: FSM
- Capture structure: Controller
  - Create architecture (state register and combinational logic)
  - Encode states
  - Create stable table (describes combinational logic)
  - Implement combinational logic



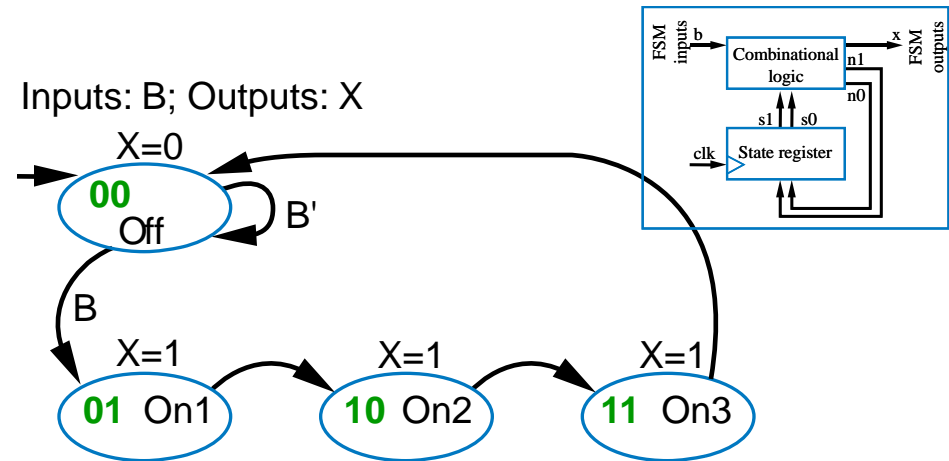
Inputs: B; Outputs: X

*LaserTimer example*



# Top-Down Design – FSMs to Controller Structure

- Recall from Chapter 2
  - **Top-down design**
    - Capture behavior, and simulate
    - Capture structure (circuit), simulate again
    - Gets behavior right first, unfettered by complexity of creating structure
- Capture behavior: FSM
- Capture structure: Controller
  - Create architecture (state register and combinational logic)
  - Encode states
  - Create stable table (describes combinational logic)
  - Implement combinational logic

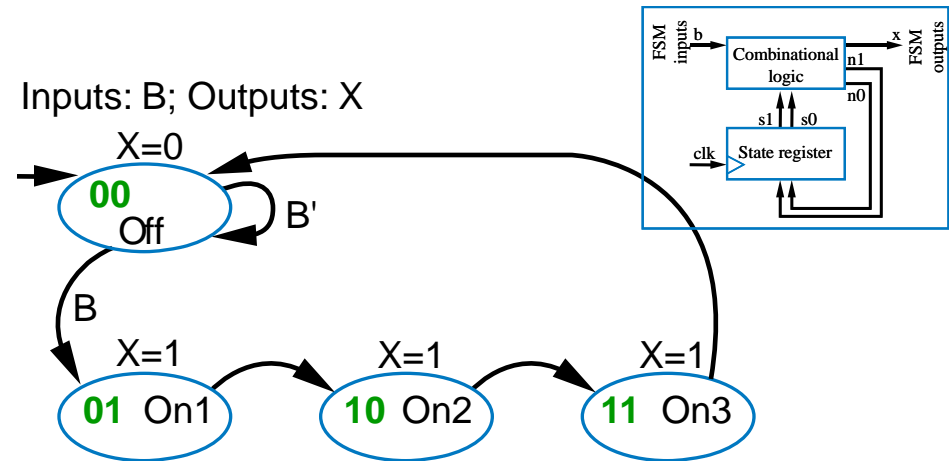


	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0



# Top-Down Design – FSMs to Controller Structure

- Recall from Chapter 2
  - **Top-down design**
    - Capture behavior, and simulate
    - Capture structure (circuit), simulate again
    - Gets behavior right first, unfettered by complexity of creating structure
- Capture behavior: FSM
- Capture structure: Controller
  - Create architecture (state register and combinational logic)
  - Encode states
  - Create stable table (describes combinational logic)
  - Implement combinational logic



	Inputs			Outputs		
	s1	s0	b	x	n1	n0
Off	0	0	0	0	0	0
	0	0	1	0	0	1
On1	0	1	0	1	1	0
	0	1	1	1	1	0
On2	1	0	0	1	1	1
	1	0	1	1	1	1
On3	1	1	0	1	0	0
	1	1	1	1	0	0

$$X = S1 + S0$$

$$N1 = S1'S0 + S1S0'$$

$$N0 = S1'S0'B + S1S0'$$



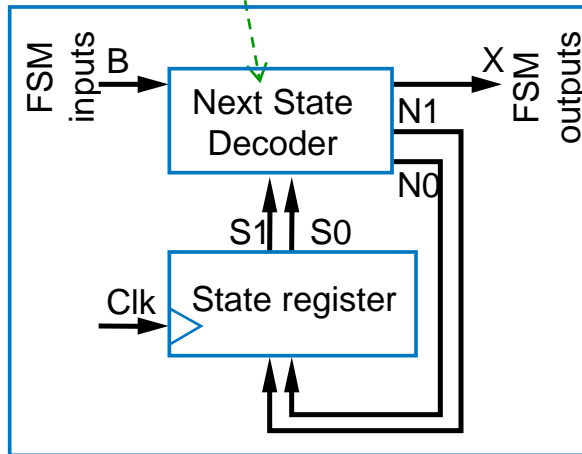
# Controller Structure

- Structural description
- Test with LaserTimerTB
  - Same results

$$X = S1 + S0$$

$$N1 = S1'S0 + S1S0' \rightarrow$$

$$N0 = S1'S0'B + S1S0'$$



```

`timescale 1 ns/1 ns

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

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

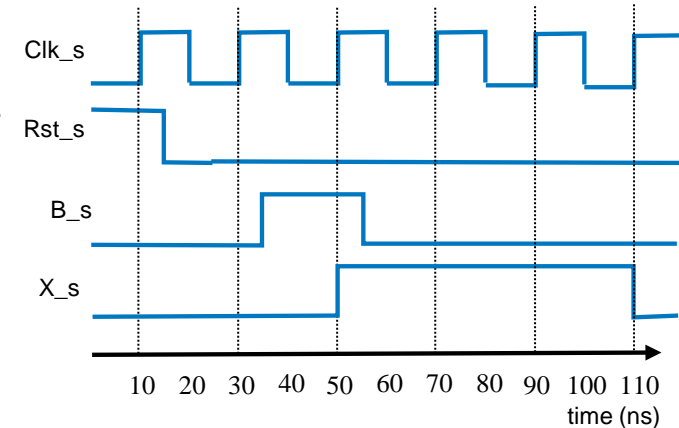
    parameter S_Off = 2'b00;

    reg [1:0] State, StateNext;
    // State encodings:
    //     S_Off 00, S_On1 01, S_On2 10, S_On3 11

    // CombLogic
    always @(State, B) begin
        X <= State[1] | State[0];
        StateNext[1] <= (~State[1] & State[0])
            | (State[1] & ~State[0]);
        StateNext[0] <= (~State[1] & ~State[0] & B)
            | (State[1] & ~State[0]);
    end

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

```



# Controller Structure

- Initial state is S\_Off
  - Encoded as "00" →
  - State register set to S\_Off during FSM reset
- Note that CombLogic uses equations, not case statement
  - Actually CombLogic is still behavioral
  - Do top-down design again, this time on CombLogic, to get structure

```
`timescale 1 ns/1 ns

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

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

    parameter S_Off = 2'b00;

    reg [1:0] State, StateNext;
    // State encodings:
    //     S_Off 00, S_On1 01, S_On2 10, S_On3 11

    // CombLogic
    always @(State, B) begin
        X <= State[1] | State[0];
        StateNext[1] <= (~State[1] & State[0])
                        | (State[1] & ~State[0]);
        StateNext[0] <= (~State[1] & ~State[0] & B)
                        | (State[1] & ~State[0]);
    end

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



# Common Pitfall: Not Assigning Every Output in Every State

- FSM outputs should be combinational function of current state (for Moore FSM)
- Not assigning output in given state means previous value is remembered
  - Output has memory
  - Behavior is not an FSM
- Solution 1
  - Be sure to assign every output in every state
- Solution 2
  - Assign default values before case statement
  - Later assignment in state overwrites default

```
// CombLogic
always @(State, B) begin
    X <= 0;
    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
```

*Could delete this without changing behavior (but probably clearer to keep it)*



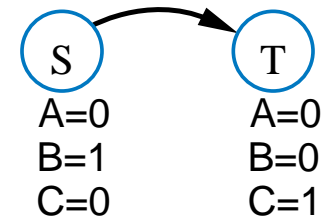


# Common Pitfall: Not Assigning Every Output in Every State

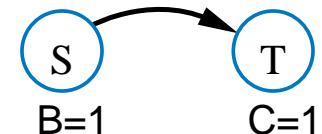
- Solution 2

- Assign default values before case statement
- Later assignment in state overwrites default
- Helps clarify which actions are important in which state
- Corresponds directly to the common simplifying FSM diagram notation of implicitly setting unassigned outputs to 0

```
case State
  S: begin
    A <= 0;
    B <= 1;
    C <= 0;
  end
  T: begin
    A <= 0;
    B <= 0;
    C <= 1;
  end
endcase
```



```
A <= 0;
B <= 0;
C <= 0;
case State
  S: begin
    B <= 1;
  end
  T: begin
    C <= 1;
  end
endcase
```





## More Simulation Concepts



# The Simulation Cycle

- Instructive to consider how an HDL simulator works
  - HDL simulation is complex; we'll introduce simplified form
- Consider example SimEx1
  - Three reg variables – Q, Clk, S
  - Three procedures – P1, P2, P3
- Simulator's job: Determine **values for nets and variables** over time
  - Repeatedly *executes* and *suspends* procedures
    - Note: Actually considers more objects, known collectively as *processes*, but we'll keep matters simple here to get just the basic idea of simulation
  - Maintains a simulation time *Time*

```
`timescale 1 ns/1 ns

module SimEx1(Q);

    output reg Q;
    reg Clk, S;

    // P1
    always begin
        Clk <= 0;
        #10;
        Clk <= 1;
        #10;
    end

    // P2
    always @(S) begin
        Q <= ~S;
    end

    // P3
    initial begin
        @ (posedge Clk);
        S <= 1;
        @ (posedge Clk);
        S <= 0;
    end

endmodule
```



# The Simulation Cycle

- Start of simulation

- Simulation time *Time* is 0
- Bit variables/nets initialized to the **unknown value x**
- Execute each procedure
  - In any order, until stops at a delay or event control

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

```
module SimEx1(Q);
```

```
    output reg Q;
```

```
    reg Clk, S;
```

```
    // P1
```

```
    always begin
```

```
        Clk <= 0;
```

```
        #10;
```

```
        Clk <= 1;
```

```
        #10;
```

```
    end
```

```
    // P2
```

```
    always @(S) begin
```

```
        Q <= ~S;
```

```
    end
```

```
    // P3
```

```
    initial begin
```

```
        @ (posedge Clk);
```

```
        S <= 1;
```

```
        @ (posedge Clk);
```

```
        S <= 0;
```

```
    end
```

```
endmodule
```

Procedures

P1 Clk <= 0, then stop.  
Activate when Time is 0+10=10 ns.

P2 No actions, then stop.  
Activate when S changes.

P3 No actions, then stop.  
Activate when Clk changes to 1

We'll use arrow  
to show where a  
procedure stops

Time (ns): Start 0

Variables

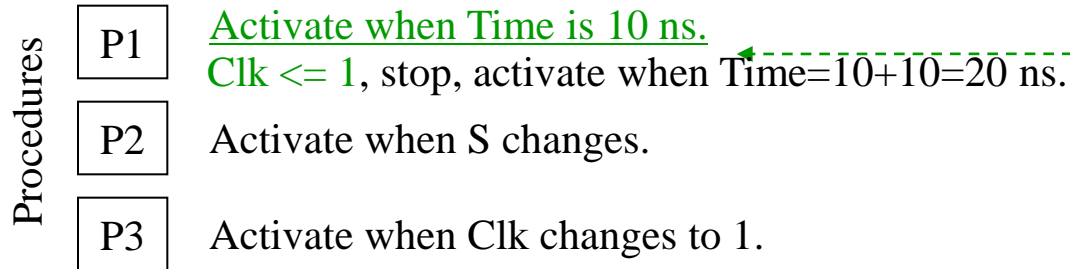
Q	x	x
Clk	x	0
S	x	x



# The Simulation Cycle

## Simulation cycle

- Set time to next time at which a procedure activates (note: could be same as current time)
  - In this case, **time = 10 ns** (P1 activates)
- Execute active procedures (in any order) until stops



Time (ns):		Start	0	10
Variables	Q	x	x	x
	Clk	x	0	1
	S	x	x	x

```

`timescale 1 ns/1 ns

module SimEx1(Q);

    output reg Q;
    reg Clk, S;

    // P1
    always begin
        Clk <= 0;
        #10;
        Clk <= 1;
        #10;
    end

    // P2
    always @(S) begin
        Q <= ~S;
    end

    // P3
    initial begin
        @ (posedge Clk);
        S <= 1;
        @ (posedge Clk);
        S <= 0;
    end

endmodule
    
```



# The Simulation Cycle

## Simulation cycle

- Set time to next time at which a procedure activates
  - Still 10 ns; Clk just changed to 1 (P3 activates)
- Execute active procedures (in any order) until stops

Procedures

- P1 Activate when Time is 20 ns.
- P2 Activate when S changes.
- P3 Activate when Clk changes to 1  
S <= 1, stop, activate when Clk changes to 1 again

Time (ns):		Start	0	10	10
Variables	Q	x	x	x	x
	Clk	x	0	1	1
	S	x	x	x	1

```

`timescale 1 ns/1 ns

module SimEx1(Q);

    output reg Q;
    reg Clk, S;

    // P1
    always begin
        Clk <= 0;
        #10;
        Clk <= 1;
        #10;
    end

    // P2
    always @(S) begin
        Q <= ~S;
    end

    // P3
    initial begin
        @ (posedge Clk);
        S <= 1;
        @ (posedge Clk);
        S <= 0;
    end

endmodule
    
```



# The Simulation Cycle

## Simulation cycle

- Set time to next time at which a procedure activates
  - Still 10 ns; S just changed (P2 activates)
- Execute active procedures until stops

Procedures

- P1 Activate when Time is 20 ns.
- P2 Activate when S changes.  
Q <= 0 (~S), stop, activate when S changes.
- P3 Activate when change on Clk to 1.

Time (ns):		Start	0	10	10	10
Variables	Q	x	x	x	x	0
	Clk	x	0	1	1	1
	S	x	x	x	1	1

```

`timescale 1 ns/1 ns

module SimEx1(Q);

    output reg Q;
    reg Clk, S;

    // P1
    always begin
        Clk <= 0;
        #10;
        Clk <= 1;
        #10;
    end

    // P2
    always @(S) begin
        Q <= ~S;
    end

    // P3
    initial begin
        @ (posedge Clk);
        S <= 1;
        @ (posedge Clk);
        S <= 0;
    end

endmodule
    
```



# The Simulation Cycle

## Simulation cycle

- Set time to next time at which a procedure activates
  - In this case, set Time = 20 ns (P1 activates)
- Execute active procedures until stops

Procedures

- P1 Activate when Time is 20 ns.  
 Clk <= 0, stop, activate when T=20+10=30ns.
- P2 Activate when S changes.
- P3 Activate when change on Clk to 1.

Time (ns):		Init	0	10	10	10	20
Variables	Q	x	x	x	x	0	0
	Clk	x	0	1	1	1	0
	S	x	x	x	1	1	1

```

`timescale 1 ns/1 ns

module SimEx1(Q);

    output reg Q;
    reg Clk, S;

    // P1
    always begin
        Clk <= 0;
        #10;
        Clk <= 1;
        #10;
    end

    // P2
    always @(S) begin
        Q <= ~S;
    end

    // P3
    initial begin
        @ (posedge Clk);
        S <= 1;
        @ (posedge Clk);
        S <= 0;
    end

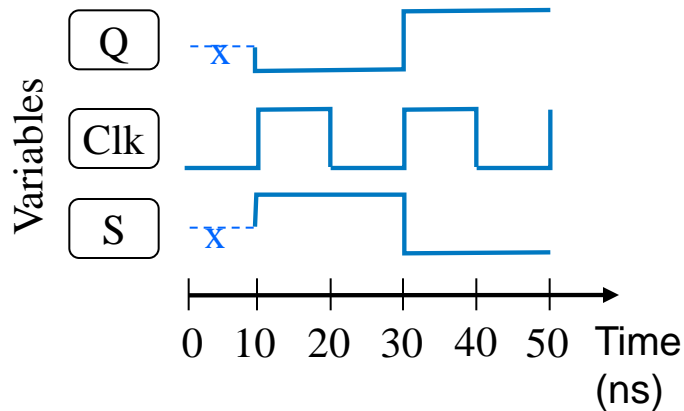
endmodule
    
```





# The Simulation Cycle

- Simulation ends when user-specified time is reached
- Variable/net values translate to waveforms



Time (ns):	Init	0	10	10	10	20	30	30	30	40	50
Q	x	x	x	x	0	0	0	0	1	1	1
Clk	x	0	1	1	1	0	1	1	1	0	1
S	x	x	x	1	1	1	1	0	0	0	0

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

```
module SimEx1(Q);
```

```
    output reg Q;
```

```
    reg Clk, S;
```

```
    // P1
```

```
    always begin
```

```
        Clk <= 0;
```

```
        #10;
```

```
        Clk <= 1;
```

```
        #10;
```

```
    end
```

```
    // P2
```

```
    always @(S) begin
```

```
        Q <= ~S;
```

```
    end
```

```
    // P3
```

```
    initial begin
```

```
        @ (posedge Clk);
```

```
        S <= 1;
```

```
        @ (posedge Clk);
```

```
        S <= 0;
```

```
    end
```

```
endmodule
```



# Variable Updates

- Assignment using "<=" ("non blocking assignment") doesn't change variable's value immediately
  - Instead, *schedules* a change of value by placing an *event* on an event queue
  - Scheduled changes occur at end of simulation cycle
- Important implications
  - Procedure execution order in a simulation cycle doesn't matter
    - Assume procedures 1 and 2 are both active
      - Proc1 schedules B to be 1, *but does not change the present value of B*. B is still 0.
      - Proc2 schedules A to be 0 (the present value of B).
      - At end of simulation cycle, B is updated to 1 and A to 0
  - Order of assignments to different variables in a procedure doesn't matter
    - Assume C was 0. Scheduled values will be C=1 and D=0, for either Proc3a or Proc3b.
  - Later assignment in procedure effectively overwrites earlier assignment
    - E will be updated with 0, but then by 1; so E is 1 at the end of the simulation cycle.

## Simulation cycle (revised)

- Set time to next time at which a procedure resumes
- Execute active procedures
- Update variables with schedule values

Assume B is 0.

Proc1:

B <= ~B;

Proc2:

A <= B;

A will be 0, not 1.

Proc3a: ←	Same	→	Proc3b:
C <= ~C;			D <= C;
D <= C;			C <= ~C;

Proc4:

E <= 0;

...

E <= 1;

←  
Recall FSM output assignment example,  
in which default assignments were added  
before the case statement.

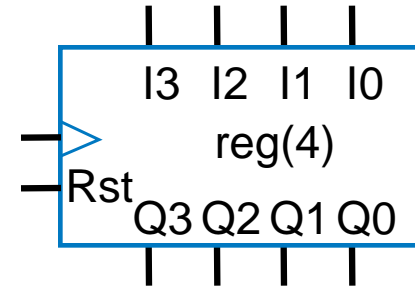


# Resets



# Resets

- **Reset** – Behavior of a register when a reset input is asserted
- Good practice dictates having defined reset behavior for every register
- Reset behavior should always have priority over normal register behavior
- Reset behavior
  - Usually clears register to 0s
  - May initialize to other value
    - e.g., state register of a controller may be initialized to encoding of initial state of FSM
- Reset usually asserted externally at start of sequential circuit operation, but also to restart due to failure, user request, or other reason



```
`timescale 1 ns/1 ns

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

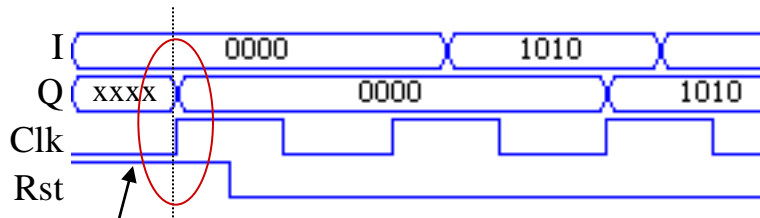
    input [3:0] I;
    output [3:0] Q;
    reg [3:0] Q;
    input Clk, Rst;

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

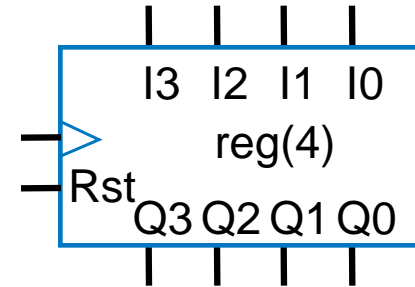


# Synchronous Reset

- Previous examples used **synchronous resets**
  - **Rst** input only considered during rising clock



Rst=1 has no effect until rising clock



```
`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
```

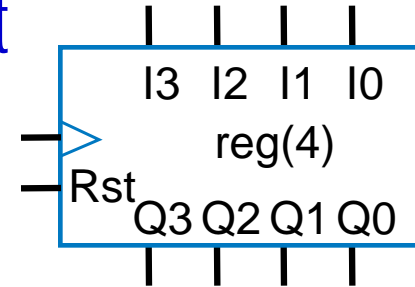
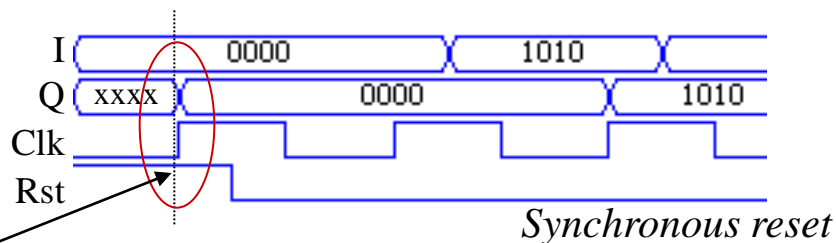
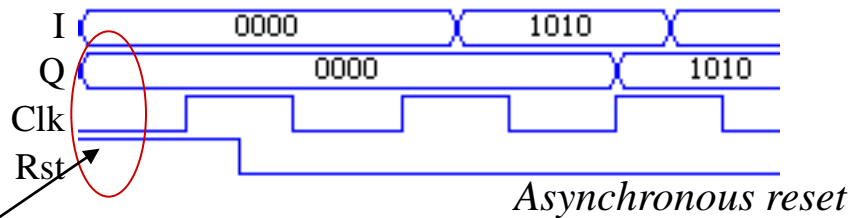


# Asynchronous Reset

- Can also use **asynchronous reset**

– **Rst** input considered independently from clock

- Add "posedge Rst" to sensitivity list



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



# Asynchronous Reset

- Could have used asynchronous reset for FSM state register too

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

Synchronous

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

Asynchronous



# Synchronous versus Asynchronous Resets

- Which is better – synchronous or asynchronous reset?
  - Hotly debated in design community
    - Each has pros and cons
      - e.g., asynchronous can still reset even if clock is not functioning, synchronous avoids timing analysis problems sometimes accompanying asynchronous designs
    - We won't try to settle the debate here
  - What's important is to be *consistent* throughout a design
    - All registers should have defined reset behavior that takes priority over normal register behavior
    - That behavior should all be synchronous reset or all be asynchronous reset
  - We will use synchronous resets in all of our remaining examples





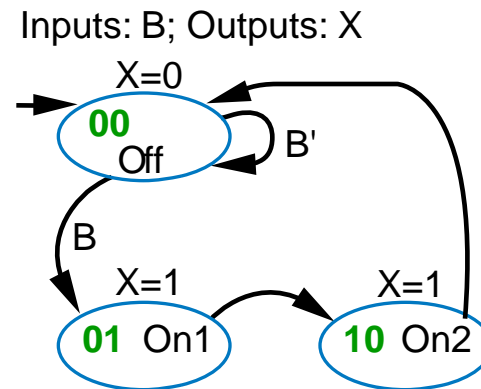


# Describing Safe FSMs



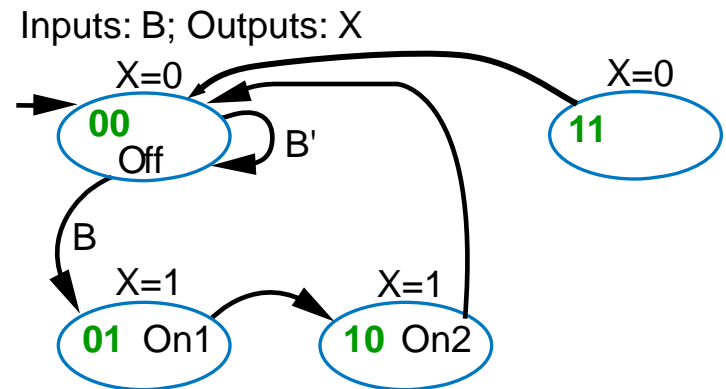
# Describing Safe FSMs

- **Safe FSM** – If enters illegal state, transitions to a legal state
- **Example**
  - Suppose example has only three states
  - Two-bit encoding has illegal state encoding "11"
    - Also known as "unreachable" state
    - Not possible to enter that state under normal FSM operation
    - But actually possible to enter that state due to circuit error – e.g., electrical noise that causes state register bit to switch



# Describing Safe FSMs

- **Safe FSM** – If enters illegal state, transitions to a legal state
- **Example**
  - Suppose example has only three states
  - Two-bit encoding has illegal state encoding "11"
  - Safe implementation
    - Transition to appropriate legal state
    - Even though that undefined state appears to be unreachable
    - Thus, FSM recovers from the error



# Describing Safe FSMs in Verilog

- Unsafe FSM description
  - Only describes legal states, ignores illegal states
- Some synthesis tools support "safe" option during synthesis
  - Automatically creates safe FSM from an unsafe FSM description

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

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_Off;
        end
    endcase
end
...
```



# Describing Safe FSMs in Verilog

- Explicitly describing a safe FSM
  - Include case item(s) to describe illegal states
  - Can use "default" case item
    - Executes if State equals anything other than S\_Off, S\_On1, or S\_On2
- Note: Use of *default* is wise regardless of number of states
  - Even if number is power of two, because state encoding may use more than minimum number of bits
    - e.g., one-hot encoding has many more illegal states than legal states
- Note: If synthesis tool support "safe" option, *use it*
  - Otherwise, tool may automatically optimize away unreachable states to improve performance and size, but making state machine unsafe

```
...  
reg [1:0] State, StateNext;  
  
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_Off;  
        end  
        default: begin  
            X <= 0;  
            StateNext <= S_Off;  
        end  
    endcase  
end  
...
```

