Digital Logic and Computer Architecture – CS322M

Satyajit Das

Assistant Professor, Dept CSE

Co-Founder – Revin Tech

satyajit.das@iitg.ac.in

What will we learn?

- Combinational and Sequential Circuits
- How can a circuit remember a value
- Different types of memorizing elements
- Finite State Machines

Introduction

- **Outputs of combinational circuit depend ONLY on current input values.**
- **Outputs of sequential logic depend on current *and* prior input values – it has memory.**
- **Some definitions:**
  - ***State***: all the information about a circuit necessary to explain its future behavior
  - ***Latches and flip-flops***: state elements that store one bit of state
  - ***Synchronous sequential circuits***: combinational logic followed by a bank of flip-flops

Sequential == Combinational + State

- **Largest part of a sequential circuit is combinational**
  - The only additional thing we need to learn is to store the state
  - Defining flip-flops (latches), registers should do the trick
- **Sequential circuits divide the operation into time slots**
  - At every time slot inputs (if there are any) are taken
  - Present state and inputs are used to calculate the next state
  - The next state is saved in the flip-flops (registers)
- **How fast we can finish the operation?**
  - The clock signal is used to move from one state to the next state
  - I.e. a 2 GHz clock has time steps of 500ps.
  - The work within a time slot is done by a combinational circuit.

Datapath vs Finite State Machine

RTL design

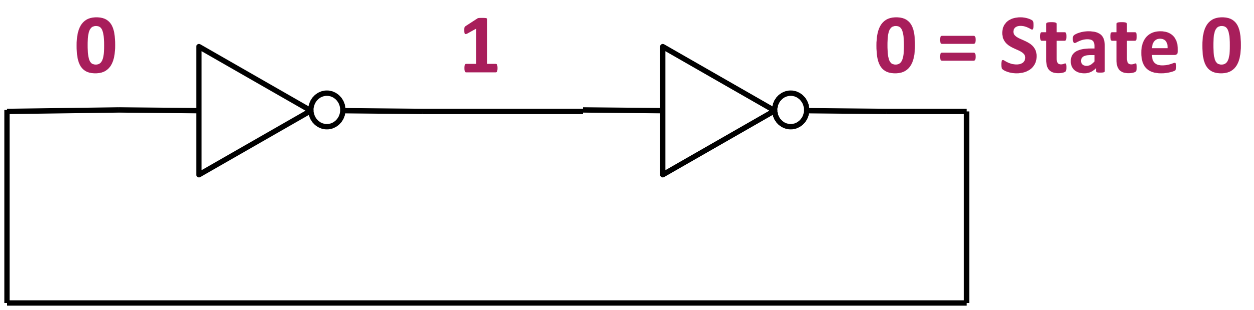
- **RTL is a generic way of defining digital circuits**
  - State is stored in registers
  - Every time slot, inputs and present state calculates the next state
  - Next state is stored in a register.
  - The clock moves the circuit from the present state to next state
- **RTL defines datapath circuits …**
  - They process data and do the main work
- **… and Finite State machines**
  - Generate control signals for the datapath
- **The distinction makes life easier**

State Elements

- **The state of a circuit influences its future behavior**
- **State elements store state**
- **Bistable circuits**
  - SR Latch
  - D Latch
  - D Flip-flop

How can a circuit remember?

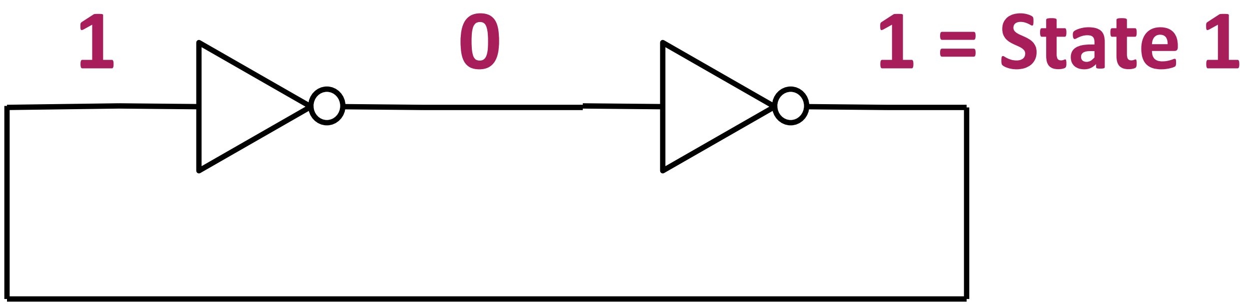
- **Bistable circuits can have two distinct states**
  - Once they are in one state, they will remain there.



- **The Loop keeps the state stable**

How can a circuit remember?

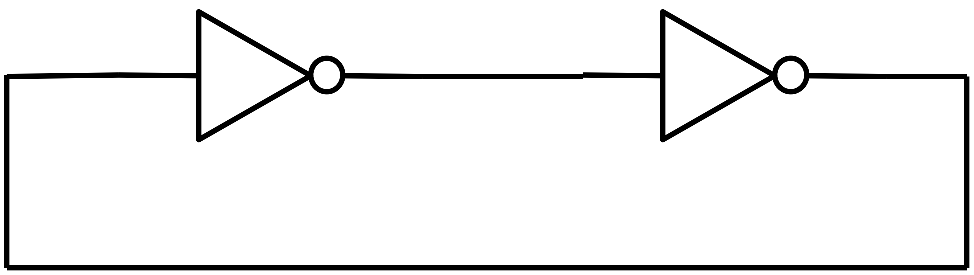
- **Bistable circuits can have two distinct states**
  - Once they are in one state, they will remain there.



- **The Loop keeps the state stable**

How can a circuit remember?

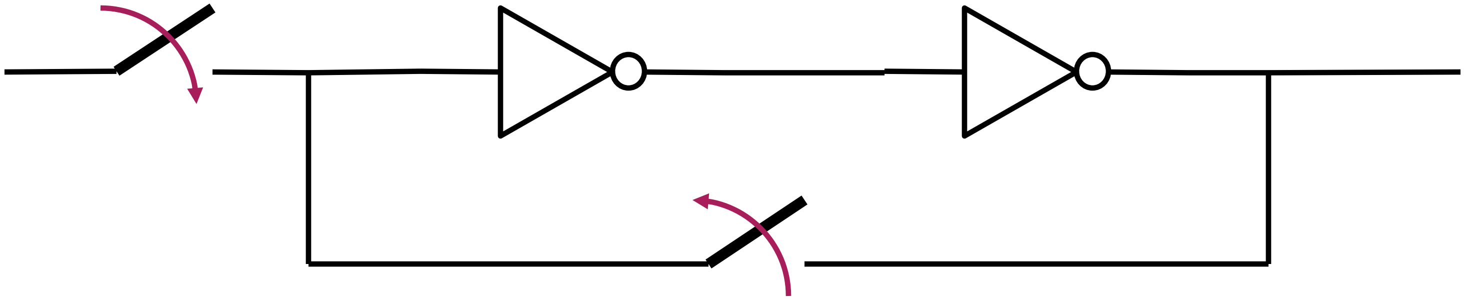
- **Bistable circuits can have two distinct states**
  - Once they are in one state, they will remain there.



- **But how can we move from one state to another?**

How can a circuit remember?

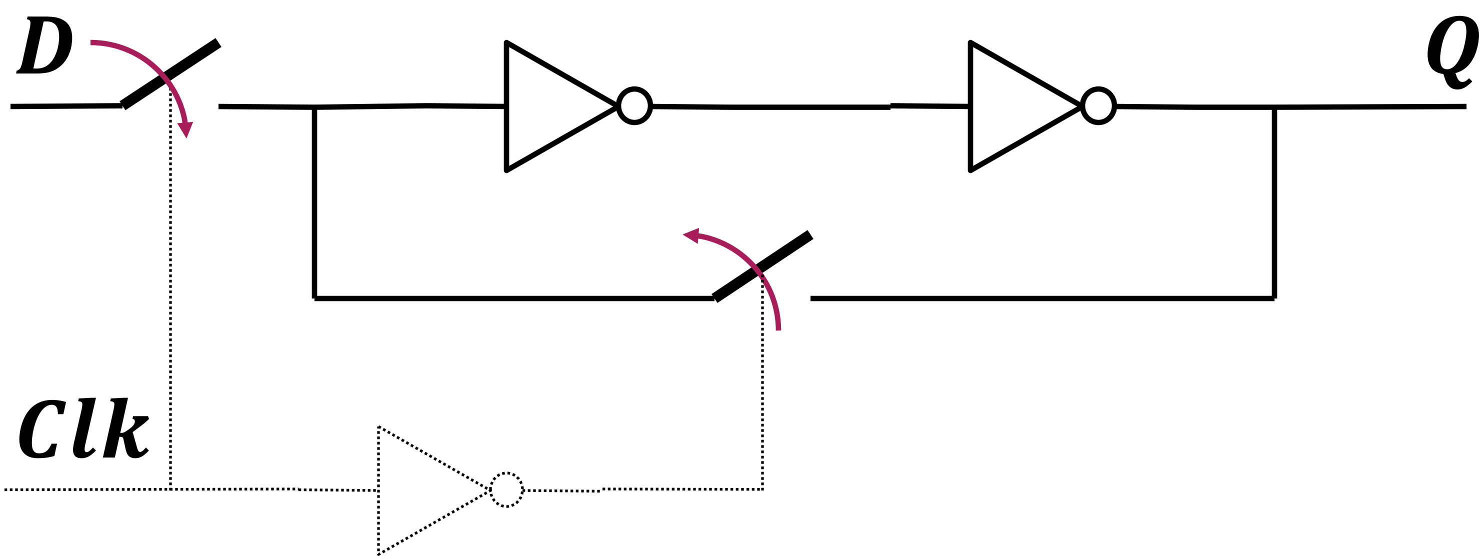
- **Bistable circuits can have two distinct states**
  - Once they are in one state, they will remain there.



- **But how can we move from one state to another?**
  - We add one switch to break the loop and at the same time add another switch that connects an input to the circuit

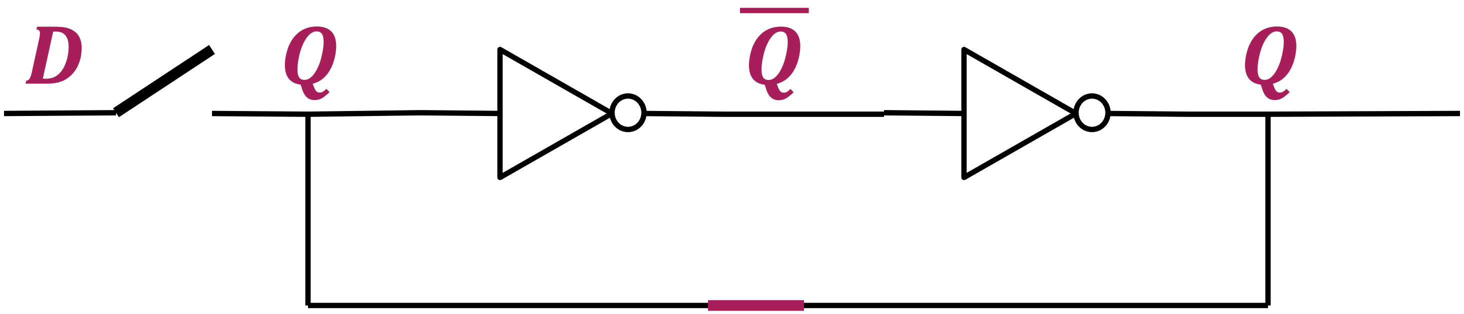
The D Latch

- **D Latch is the basic bi-stable circuit used in modern CMOS.**
  - The clock controls the switches. **Only one is active** at a time.
  - Traditionally the input is called D (Data) and the output Q



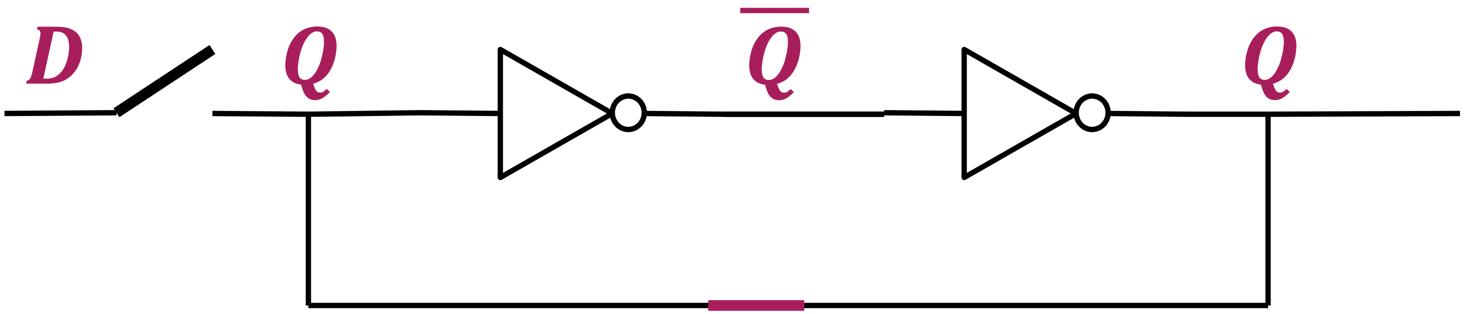
The D Latch has two modes

- **Latch mode, loop is active, input disconnected, keeps state**

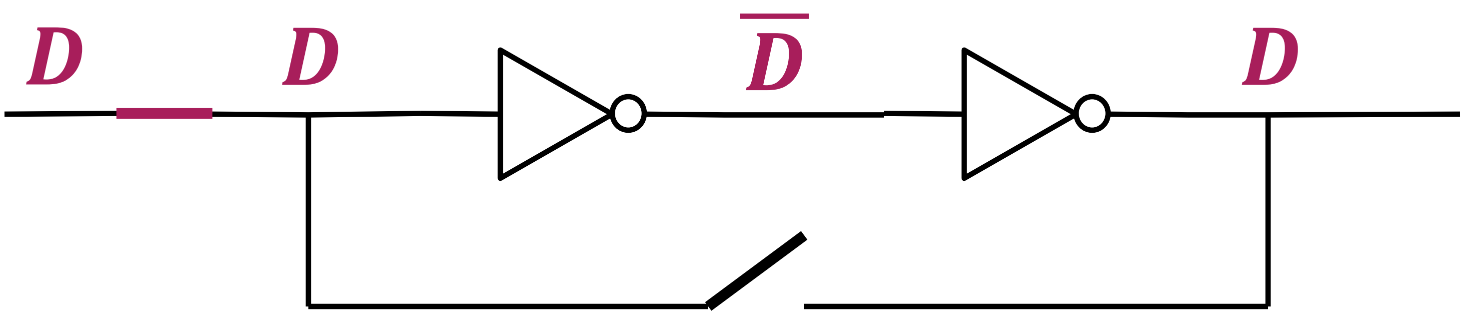


The D Latch has two modes

- Latch mode, loop is active, input disconnected, keeps state



- Transparent mode, loop is inactive, input is connected and propagates to output



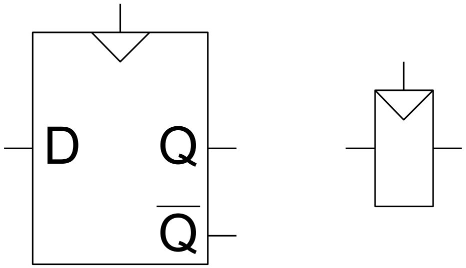
Summary D Latch

- **Simple bi-stable circuit**
  - Can be used to store a 0 or a 1.
- **Has two modes**
  - ***Transparent mode***: input propagates to output
  - ***Latch mode***: the output is stored (also called ***opaque mode***)
- **The clock controls the modes of operation.**
  - Depending on the type, it might be latch is transparent when Clk=1 or latch is transparent when Clk=0

Rising edge trigerred D Flip-Flop

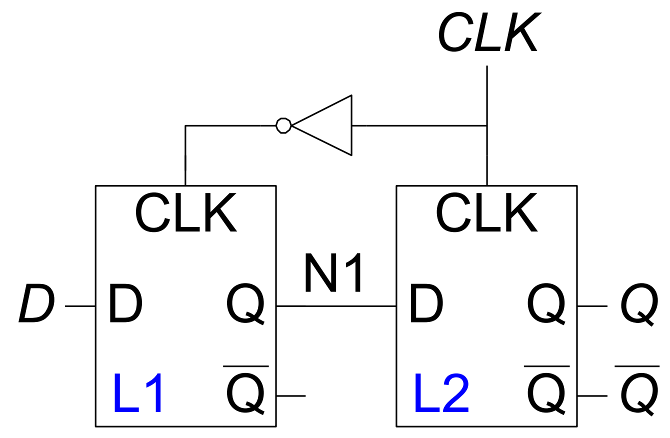
- **Two inputs: CLK, D**
- **Function**
  - The flip-flop "samples" D on the rising edge of CLK
  - When CLK rises from 0 to 1, D passes through to Q
  - Otherwise, Q holds its previous value
  - Q changes only on the rising edge of CLK
- **A flip-flop is called an edge-triggered device because it is activated on the clock edge**

D Flip-Flop
Symbols



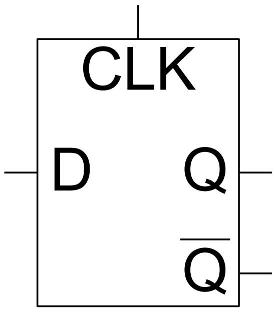
D Flip-Flop Internal Circuit

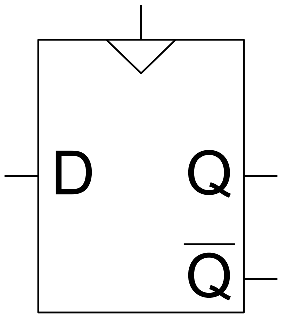
- **Two back-to-back latches (L1 and L2) controlled by complementary clocks**
  - ***When CLK = 0***
    - L1 is transparent
    - L2 is opaque
    - D passes through to N1
  - ***When CLK = 1***
    - L2 is transparent
    - L1 is opaque
    - N1 passes through to Q



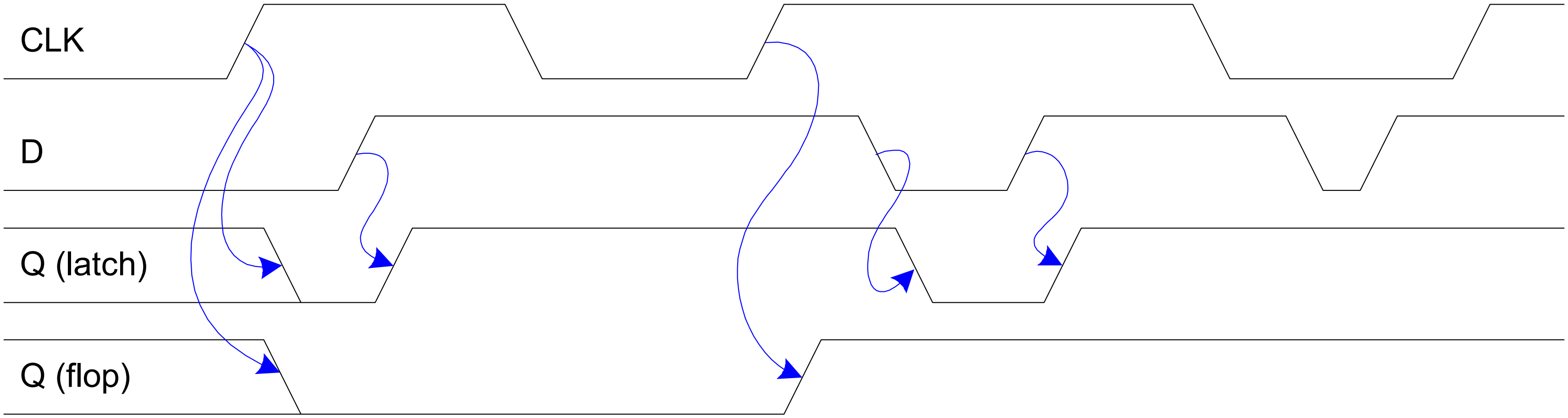
- **Thus, on the edge of the clock (when CLK rises from 0   1)**
- **D passes through to Q**

D Flip-Flop vs. D Latch



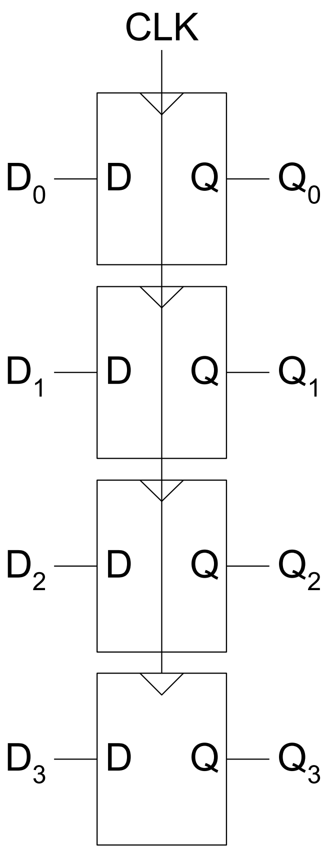


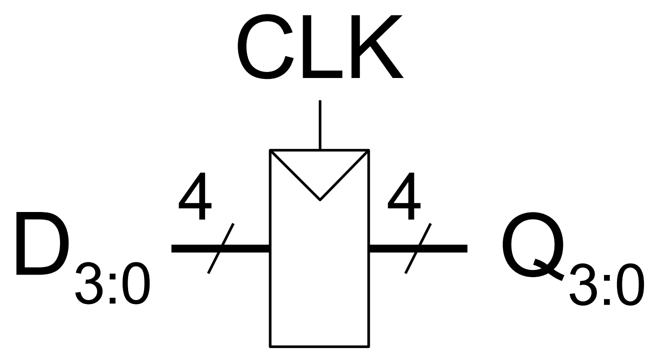
D Flip-Flop vs. D Latch



Registers

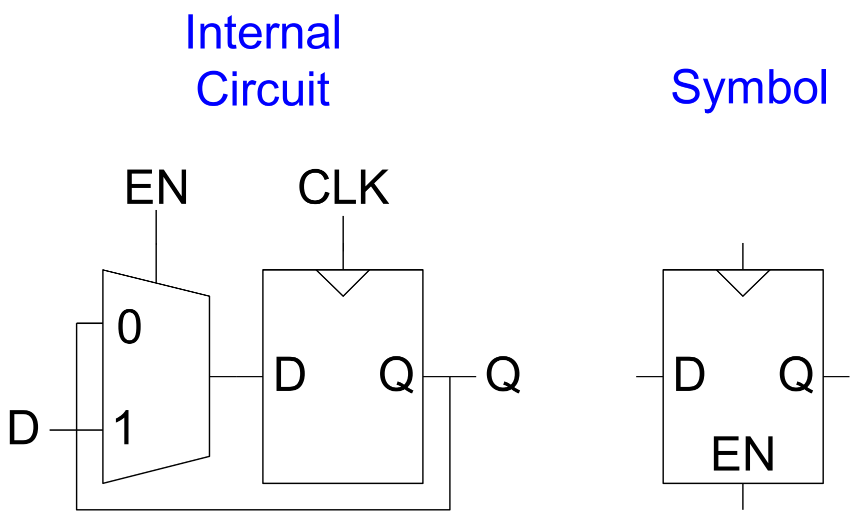
- **Multiple parallel flip-flops that store more than 1 bit**





Enabled Flip-Flops

- **Inputs: CLK, D, EN**
  - The enable input (EN) controls when new data (D) is stored
- **Function**
  - **EN = 1**: D passes through to Q on the clock edge
  - **EN = 0**: the flip-flop retains its previous state

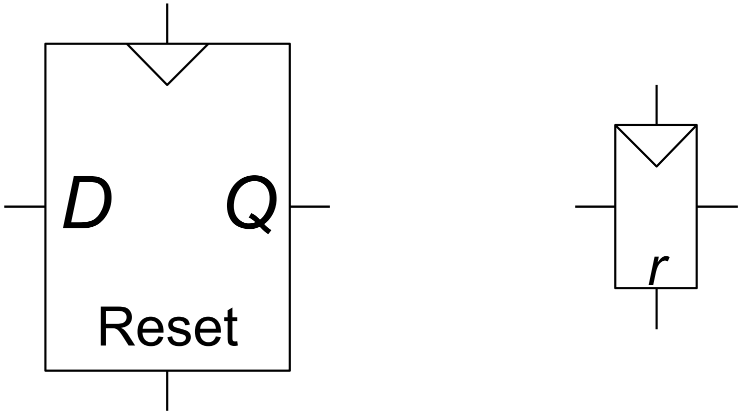


Resettable Flip-Flops

- **Inputs: CLK, D, Reset**
  - The Reset is used to set the output to 0.

- **Function:**
  - ***Reset = 1:*** Q is forced to 0
  - ***Reset = 0:*** the flip-flop behaves like an ordinary D flip-flop

Symbols

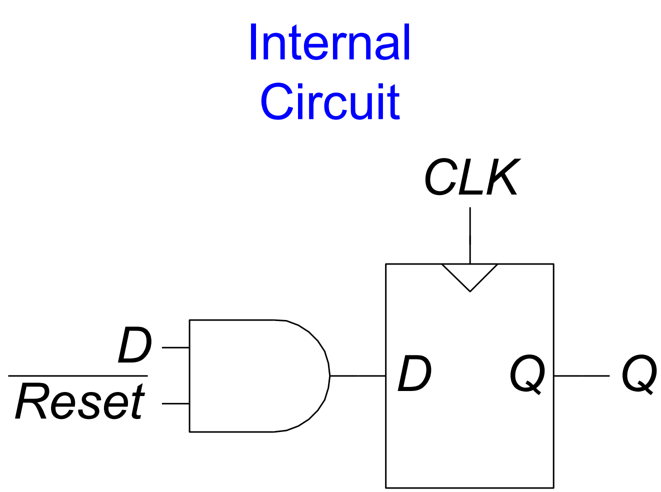


Resettable Flip-Flops

- **Two types:**
  - Synchronous: resets at the clock edge only
  - Asynchronous: resets immediately when Reset = 1
- **Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop (see Exercise 3.10)**
- **Synchronously resettable flip-flop?**

Resettable Flip-Flops

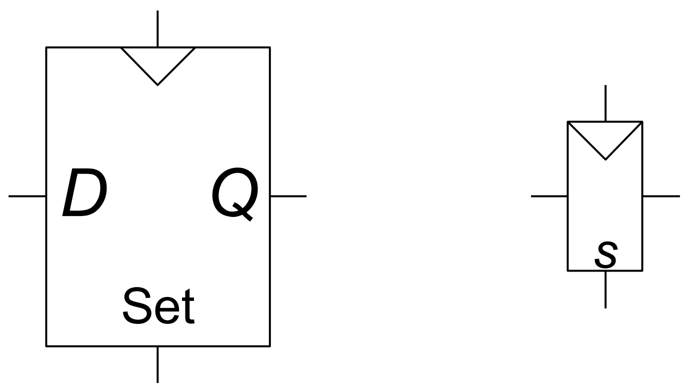
- **Two types:**
  - Synchronous: resets at the clock edge only
  - Asynchronous: resets immediately when Reset = 1
- **Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop (see Exercise 3.10)**
- **Synchronously resettable flip-flop?**



Settable Flip-Flops

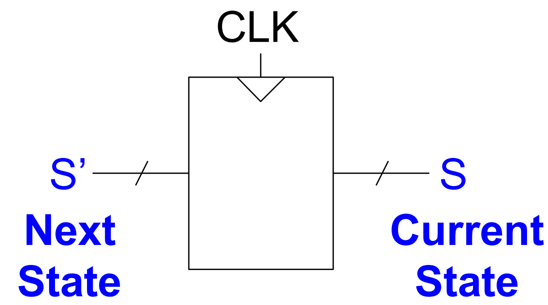
- **Inputs: CLK, D, Set**
- **Function:**
  - **Set = 1**: Q is set to 1
  - **Set = 0**: the flip-flop behaves like an ordinary D flip-flop

Symbols

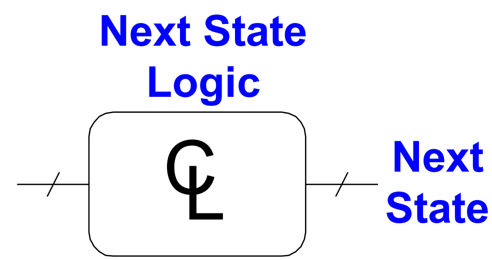


Finite State Machine (FSM) consists of:

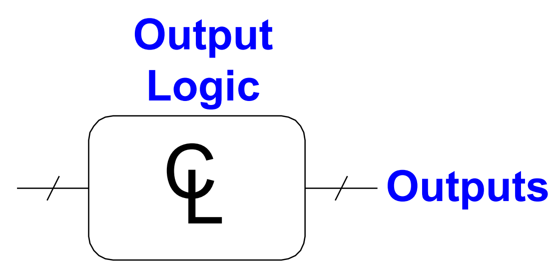
- **State register:**
  - Store the current state and
  - Load the next state at the clock edge
  - Sequential circuit



- **Next state logic**
  - Determines what the next state will be
  - Combinational circuit



- **Output logic**
  - Generates the outputs
  - Combinational Circuit



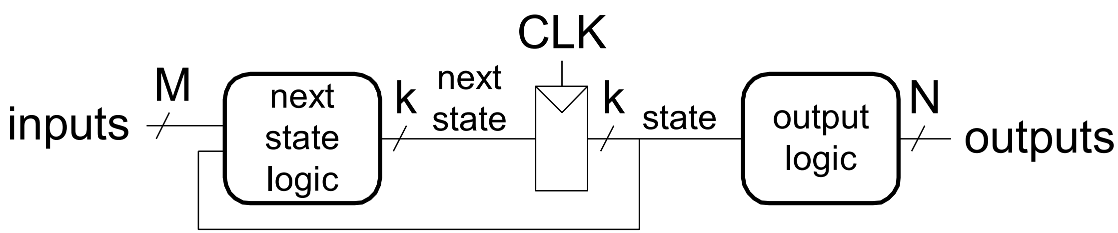
Finite State Machine (FSM)

- **FSMs get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states.**

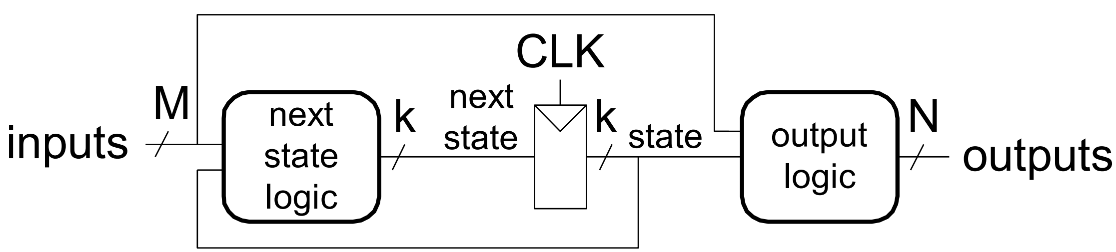
Finite State Machines (FSMs)

- **Next state is determined by the current state and the inputs**
- **Two types of finite state machines differ in the output logic:**
  - **Moore FSM**: outputs depend only on the current state
  - **Mealy FSM**: outputs depend on the current state and the inputs

Moore FSM

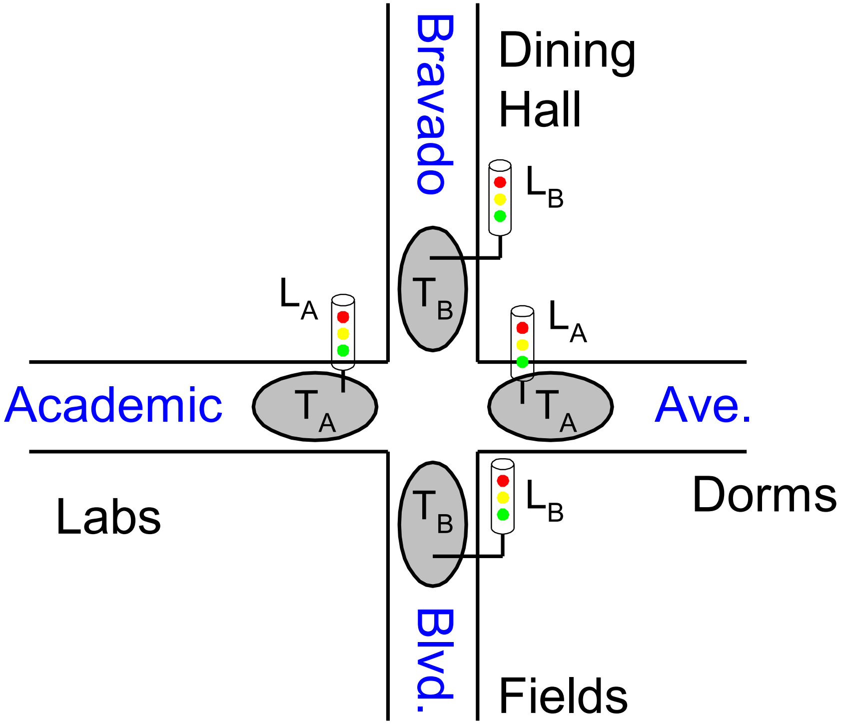


Mealy FSM



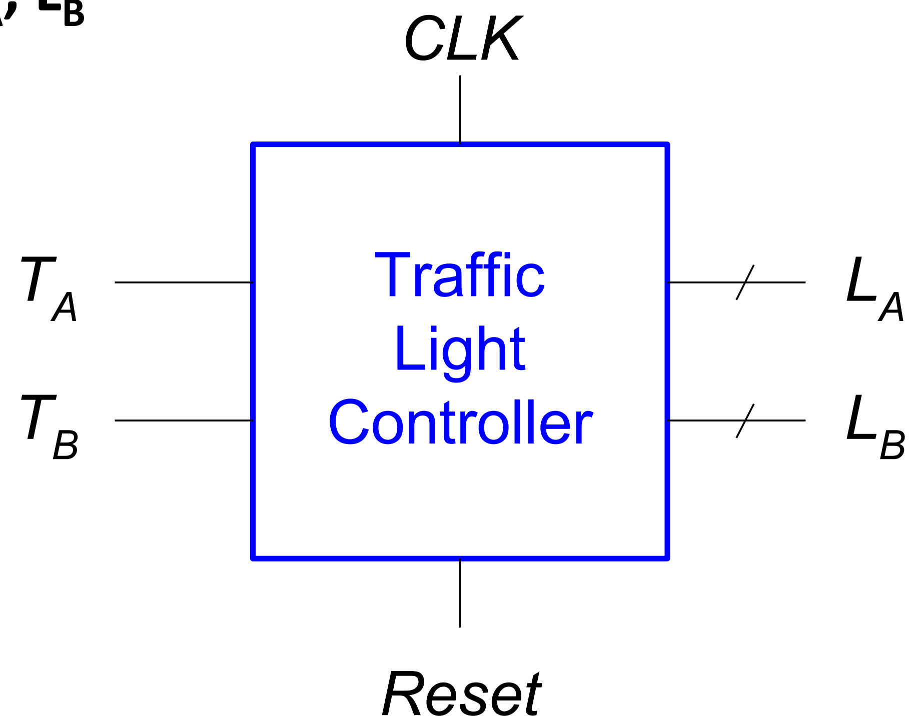
Finite State Machine Example

- **Traffic light controller**
  - **2 inputs**: Traffic sensors: T_A, T_B (TRUE when there's traffic)
  - **2 outputs**: Lights: L_A, L_B



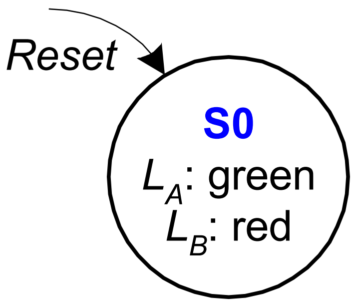
FSM Black Box

- **Inputs: CLK, Reset, T_A, T_B**
- **Outputs: L_A, L_B**



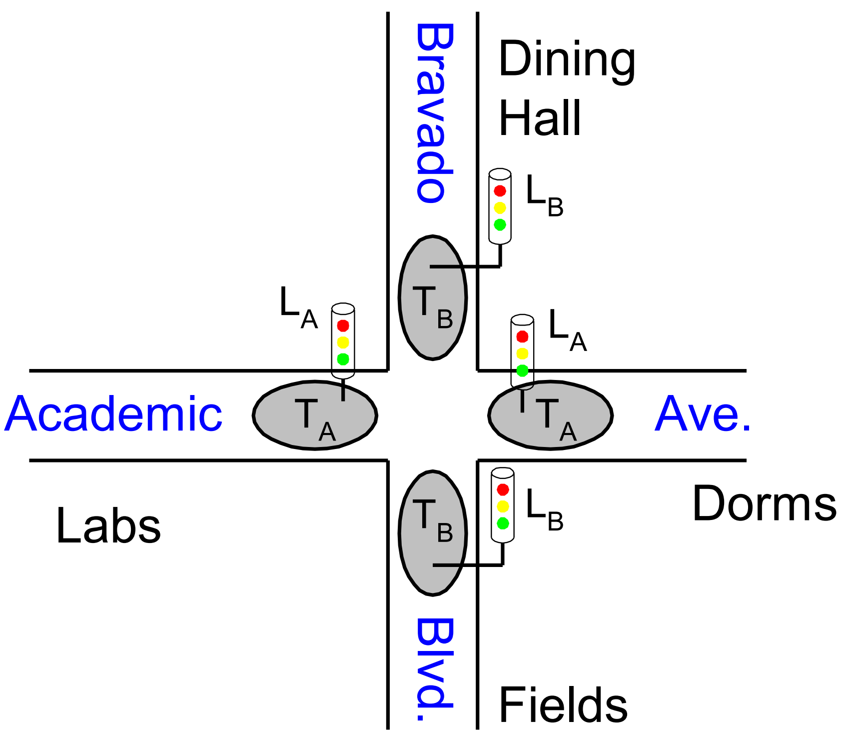
FSM State Transition Diagram

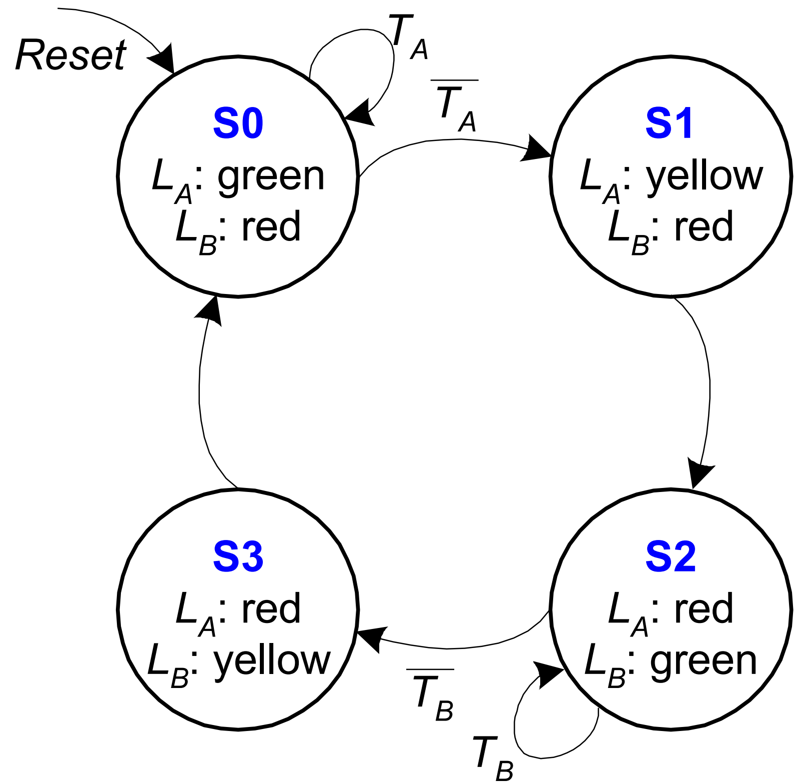
- **Moore FSM: outputs labeled in each state**
  - States: Circles
  - Transitions: Arcs



FSM State Transition Diagram

- **Moore FSM: outputs labeled in each state**
  - States: Circles
  - Transitions: Arcs





FSM State Transition Table

Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	
S0	1	X	
S1	X	X	
S2	X	0	
S2	X	1	
S3	X	X	

FSM State Transition Table

Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

FSM Encoded State Transition Table

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X		
0	0	1	X		
0	1	X	X		
1	0	X	0		
1	0	X	1		
1	1	X	X		

State	Encoding
S0	00
S1	01
S2	10
S3	11

FSM Encoded State Transition Table

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

FSM Encoded State Transition Table

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S_1' = (\overline{S_1} \cdot S_0) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B}) + (S_1 \cdot \overline{S_0} \cdot T_B)$$

$$S_0' = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$$

FSM Encoded State Transition Table

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$S_1' = S_1 \text{ xor } S_0$

Simplification (Inspection or K-Maps)

$S_0' = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$

FSM Output Table

Current State		Outputs	
S_1	S_0	L_A	L_B
0	0		
0	1		
1	0		
1	1		

FSM Output Table

Current State		Outputs	
S_1	S_0	L_A	L_B
0	0	green	red
0	1	yellow	red
1	0	red	green
1	1	red	yellow

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

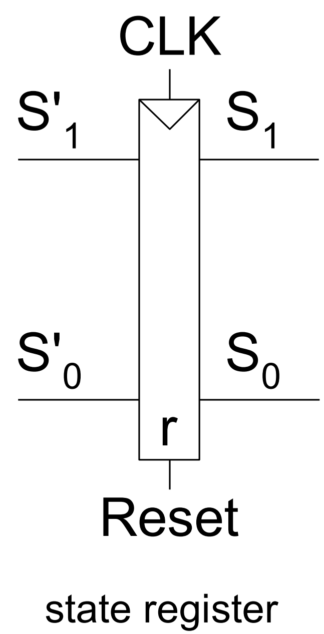
FSM Output Table

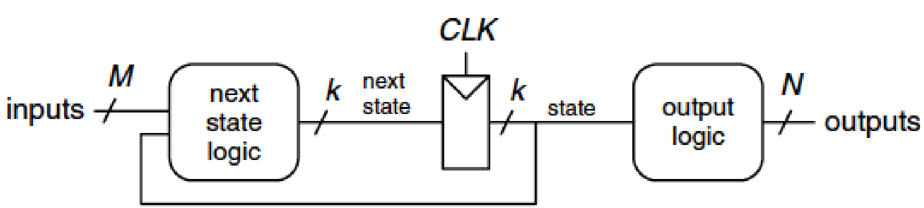
Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} \cdot S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 \cdot S_0$$

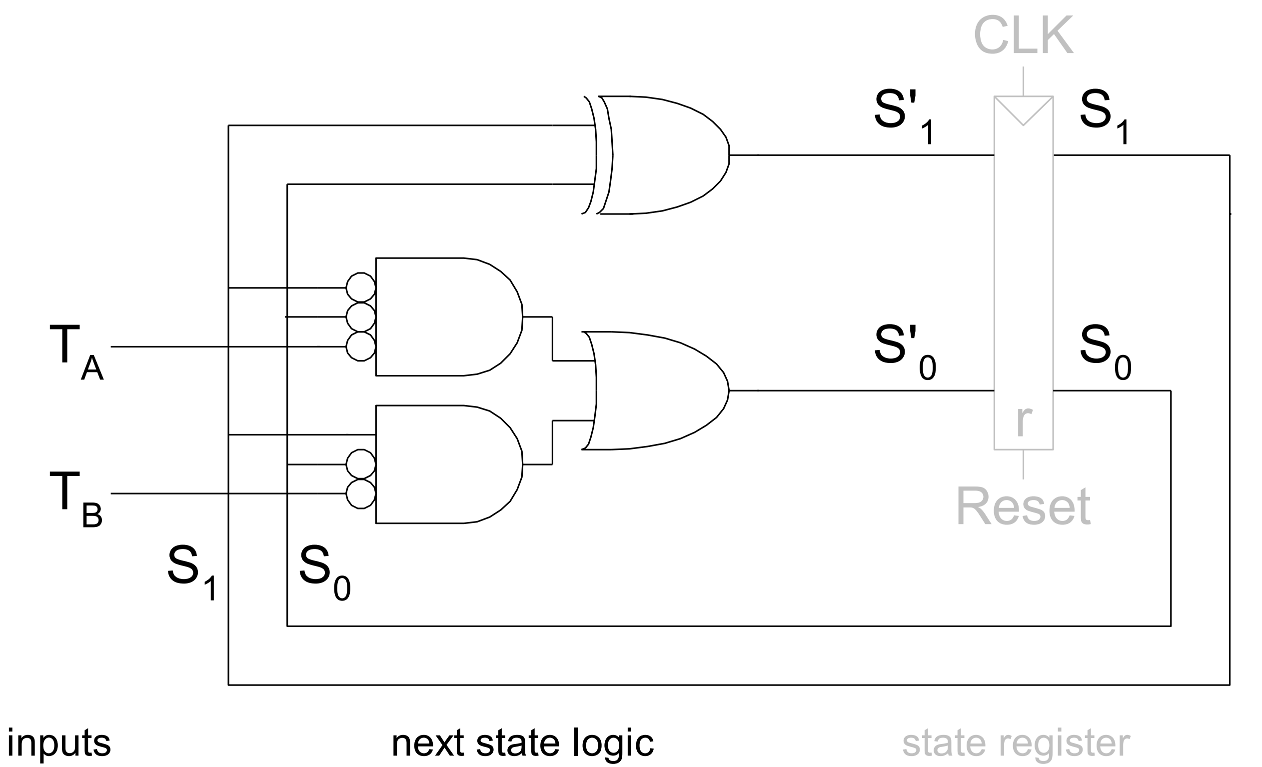
FSM Schematic: State Register

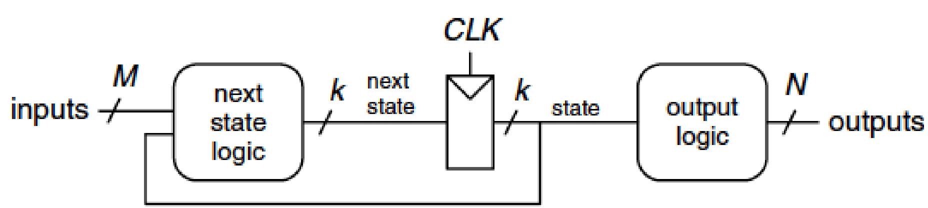




(a)

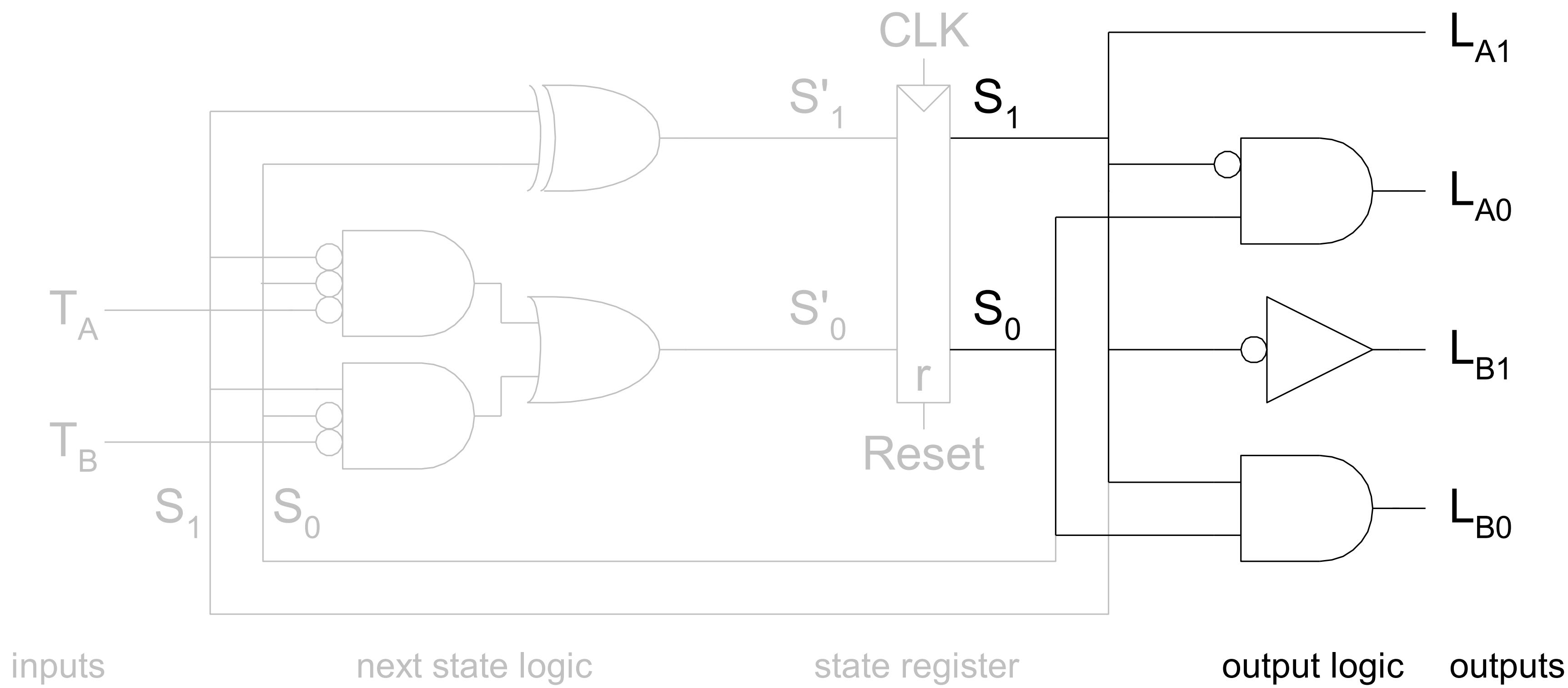
FSM Schematic: Next State Logic

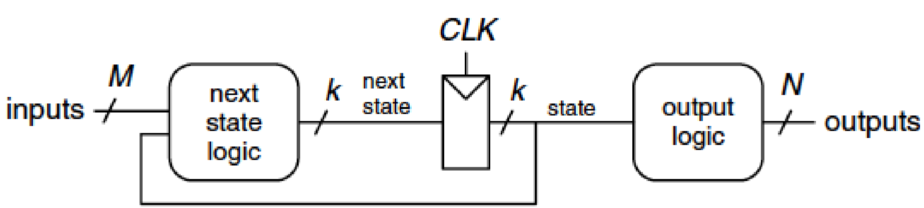




(a)

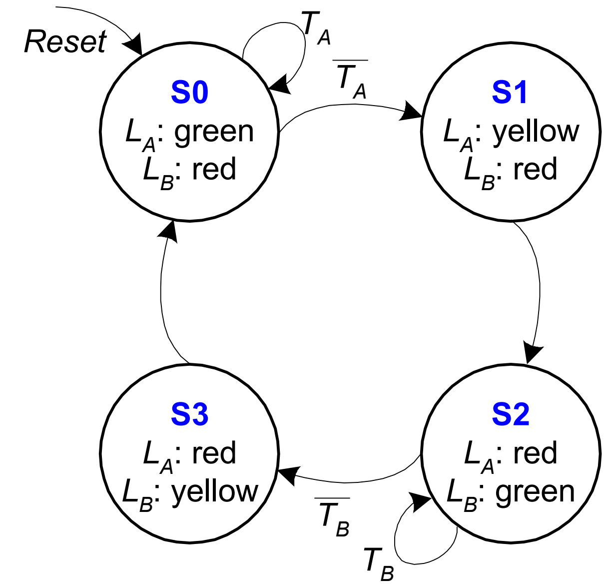
FSM Schematic: Output Logic

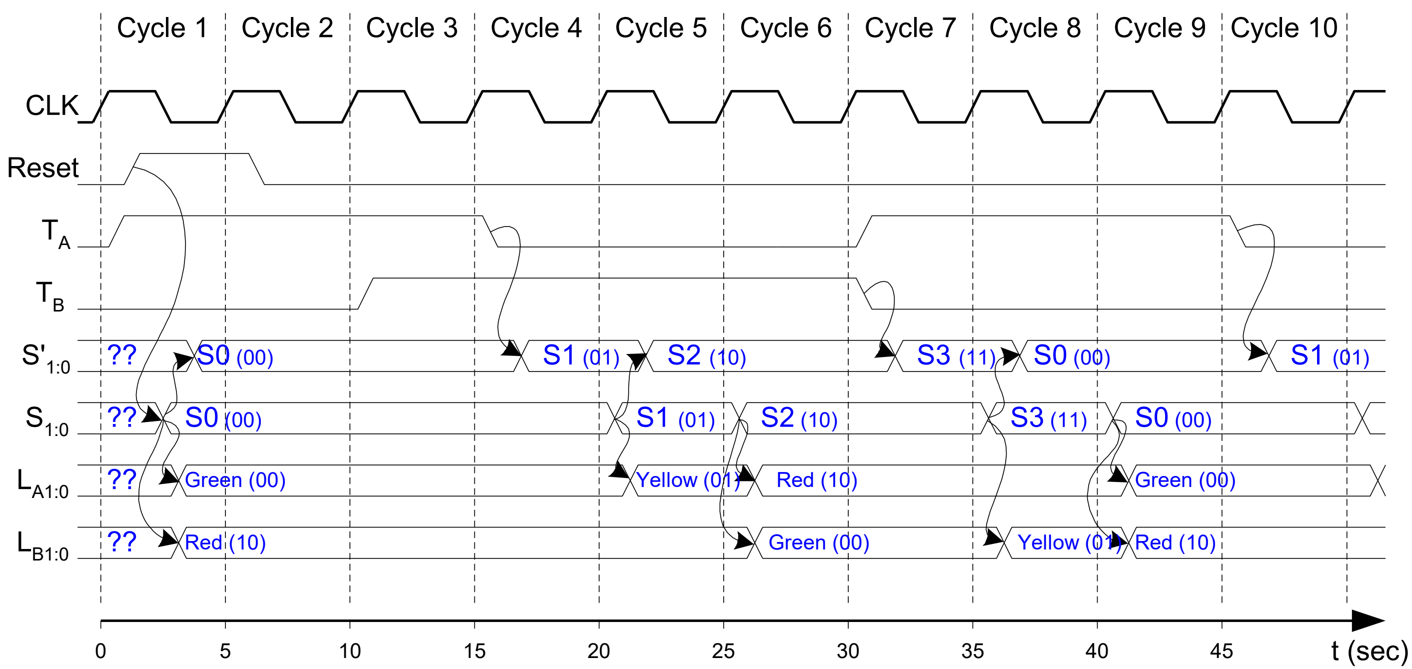




(a)

FSM Timing Diagram





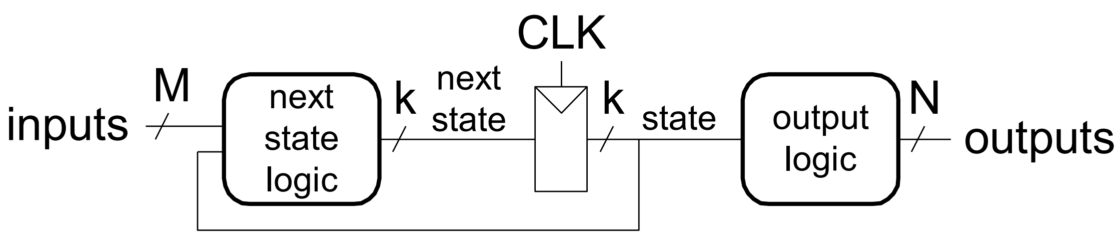
FSM State Encoding

- **Binary encoding: i.e., for four states, 00, 01, 10, 11**
- **One-hot encoding**
  - One state bit per state
  - Only one state bit is HIGH at once
  - I.e., for four states, 0001, 0010, 0100, 1000
  - Requires more flip-flops
  - Often next state and output logic is simpler

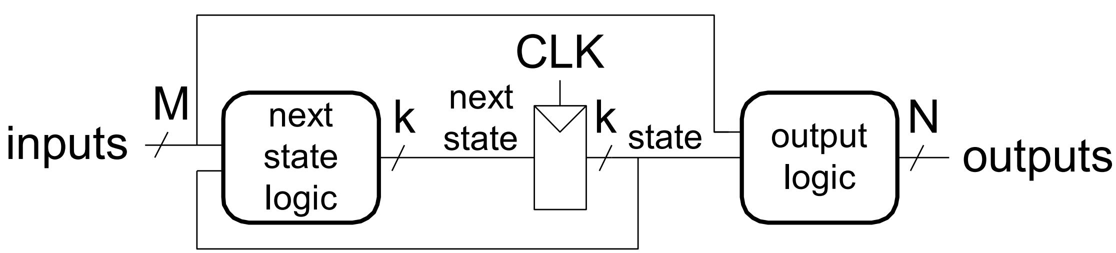
Moore vs. Mealy FSM

- **Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last four digits it has crawled over are 1101. Design Moore and Mealy FSMs of the snail's brain.**

Moore FSM

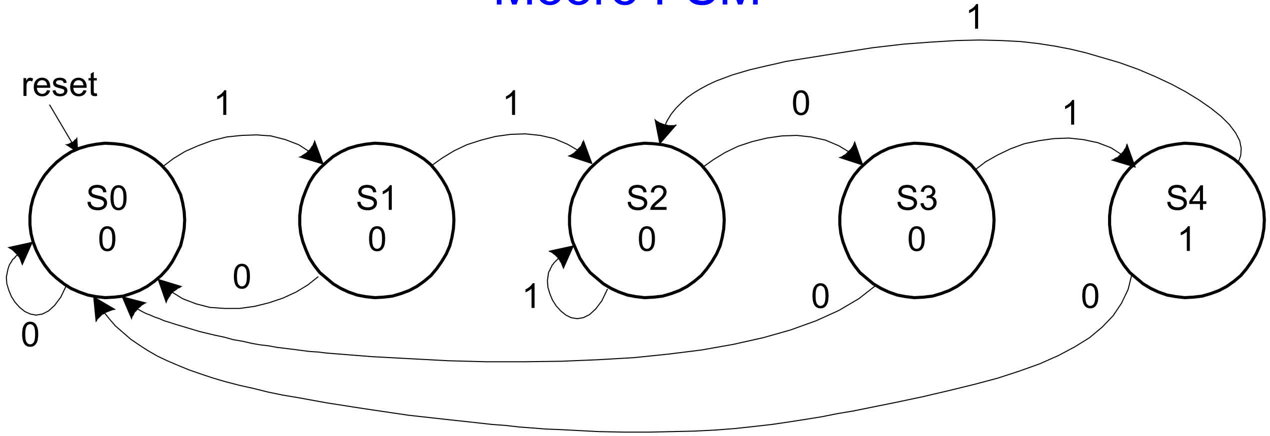


Mealy FSM



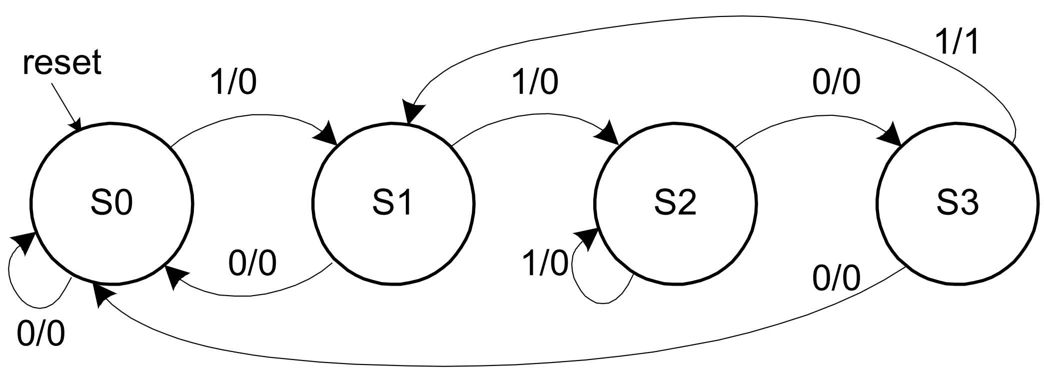
State Transition Diagrams (snail - 1101)

Moore FSM



- **Mealy FSM: arcs indicate input/output**

Mealy FSM



Moore FSM State Transition Table

Current State			Inputs	Next State		
S_2	S_1	S_0	A	S'_2	S'_1	S'_0
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1			
0	1	0	0			
0	1	0	1			
0	1	1	0			
0	1	1	1			
1	0	0	0			
1	0	0	1			

State	Encoding
S0	000
S1	001
S2	010
S3	011
S4	100

Moore FSM State Transition Table

Current State			Inputs	Next State		
S_2	S_1	S_0	A	S'_2	S'_1	S'_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	0	1	0
0	1	1	0	0	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	0	1	0	1	0

State	Encoding
S0	000
S1	001
S2	010
S3	011
S4	100

Moore FSM Output Table

Current State			Output
S_2	S_1	S_0	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	

Moore FSM Output Table

Current State			Output
S_2	S_1	S_0	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1

$$\mathbf{Y = S_2}$$

Mealy FSM State Transition and Output

Current State		Input	Next State		Output
S_1	S_0	A	S'_1	S'_0	Y
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

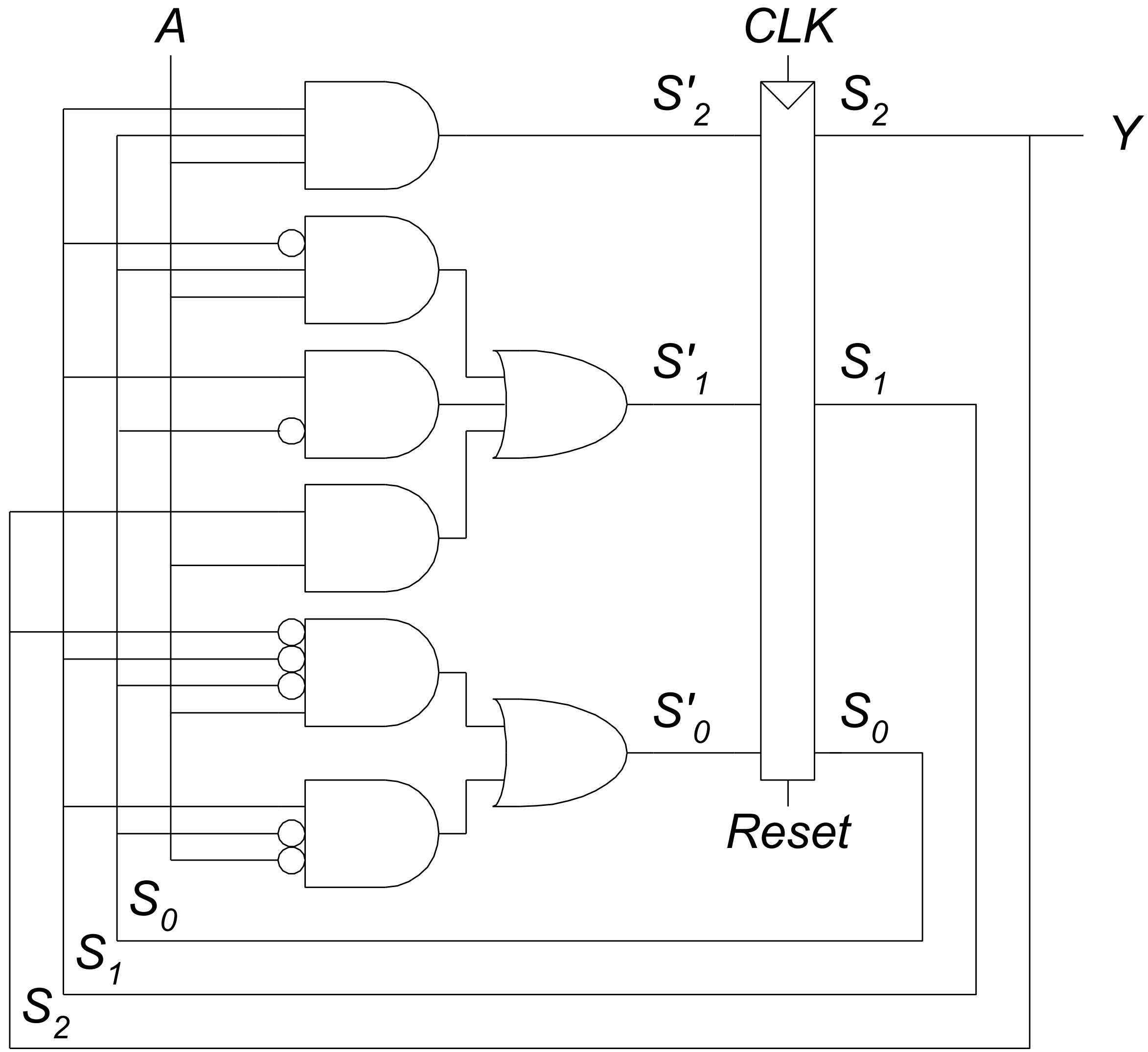
State	Encoding
S0	00
S1	01
S2	10
S3	11

Mealy FSM State Transition and Output

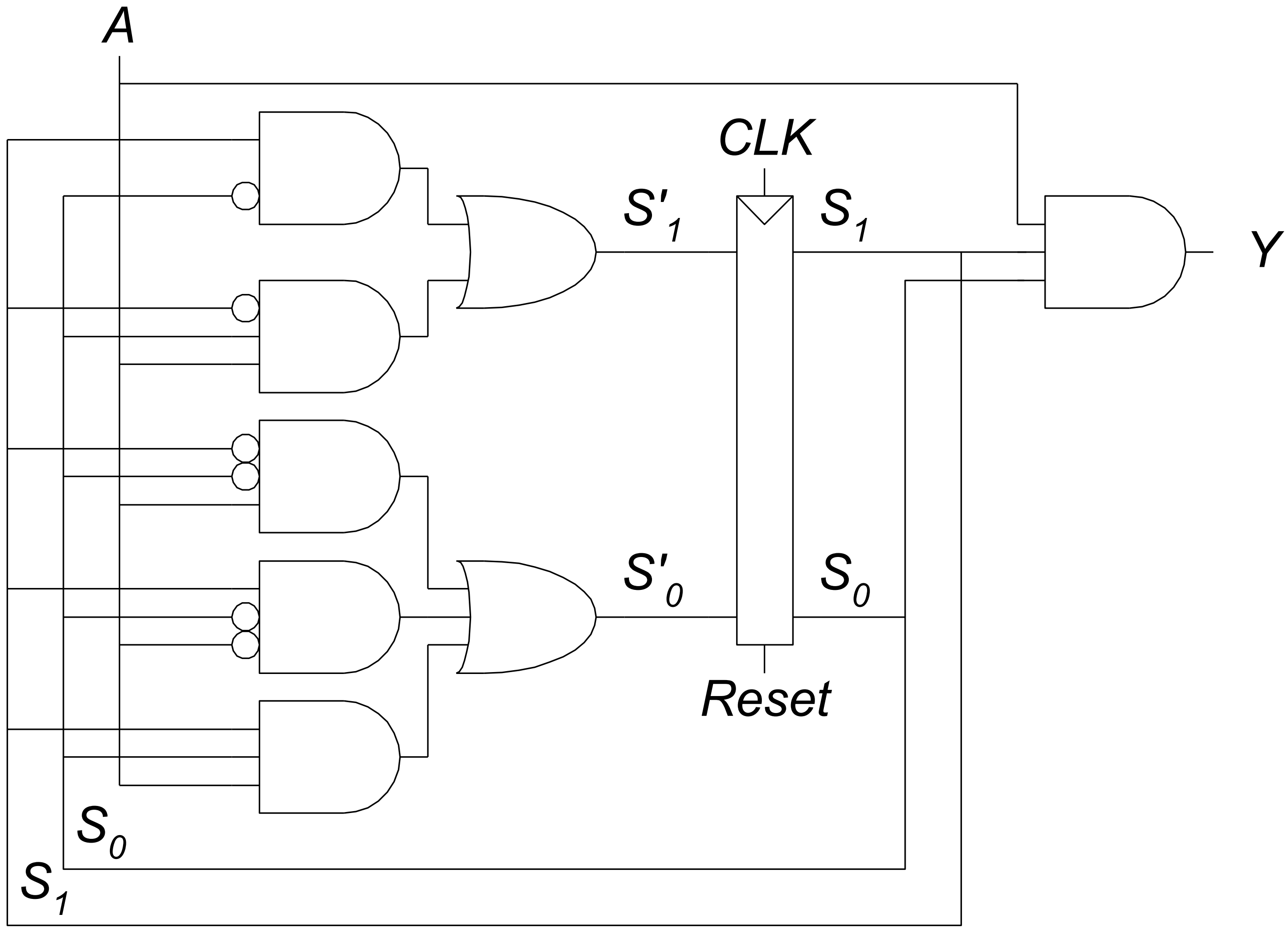
Current State		Input	Next State		Output
S_1	S_0	A	S'_1	S'_0	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

State	Encoding
S0	00
S1	01
S2	10
S3	11

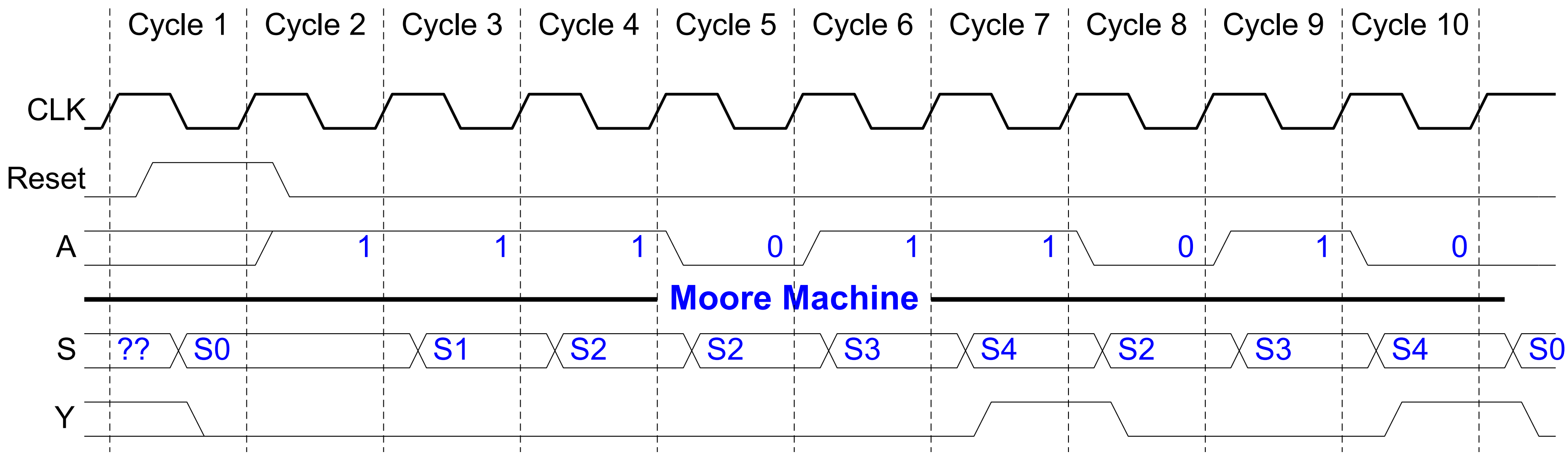
Moore FSM Schematic



Mealy FSM Schematic



Moore and Mealy Timing Diagram

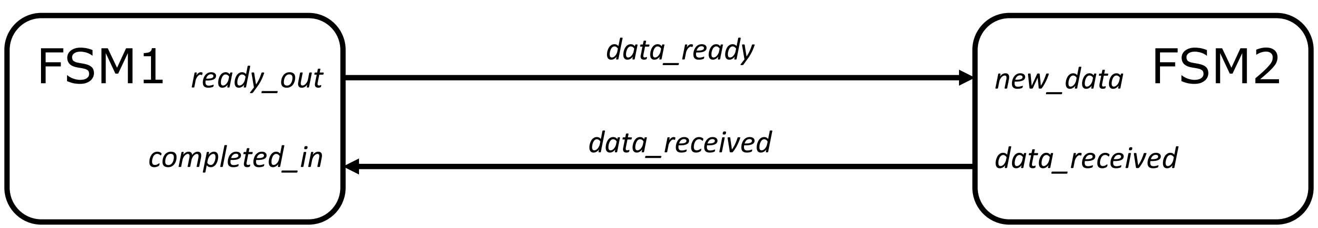


Why do we care if FSM is Moore/Mealy?

- **Remember combinational circuits**
  - You are not supposed to have loops

- **If two FSMs are connected, there will be NO loop if**
  - At least one of them is MOORE

- **There CAN BE a combinational loop if**
  - Both FSMs are MEALY type.



Factoring State Machines

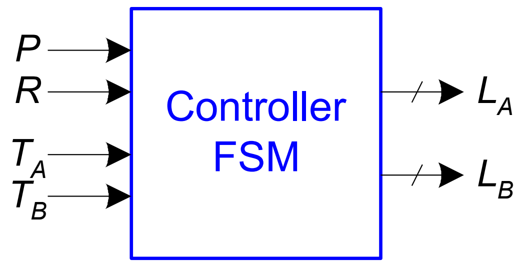
- **Break complex FSMs into smaller interacting FSMs**

- **Example: Modify the traffic light controller to have a Parade Mode.**
  - The FSM receives two more inputs: *P*, *R*
  - When *P* = 1, it enters Parade Mode and the Bravado Blvd. light stays green.
  - When *R* = 1, it leaves Parade Mode

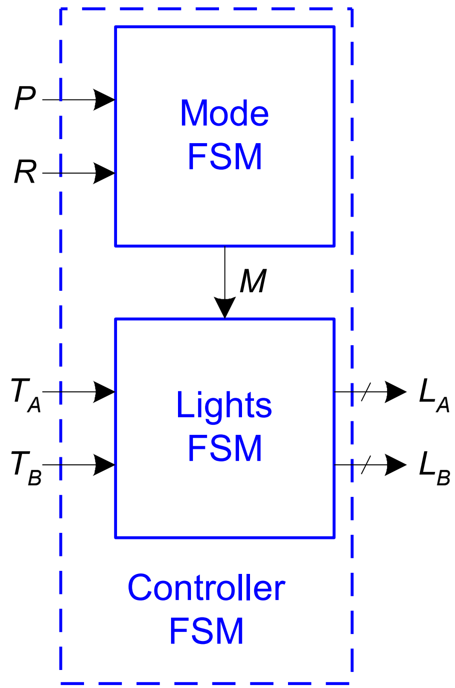
- **Designing complex FSMs is often easier if they can be broken down into multiple interacting simpler state machines such that the output of some machines is the input of others. This application of hierarchy and modularity is called factoring of state machines.**

Parade FSM

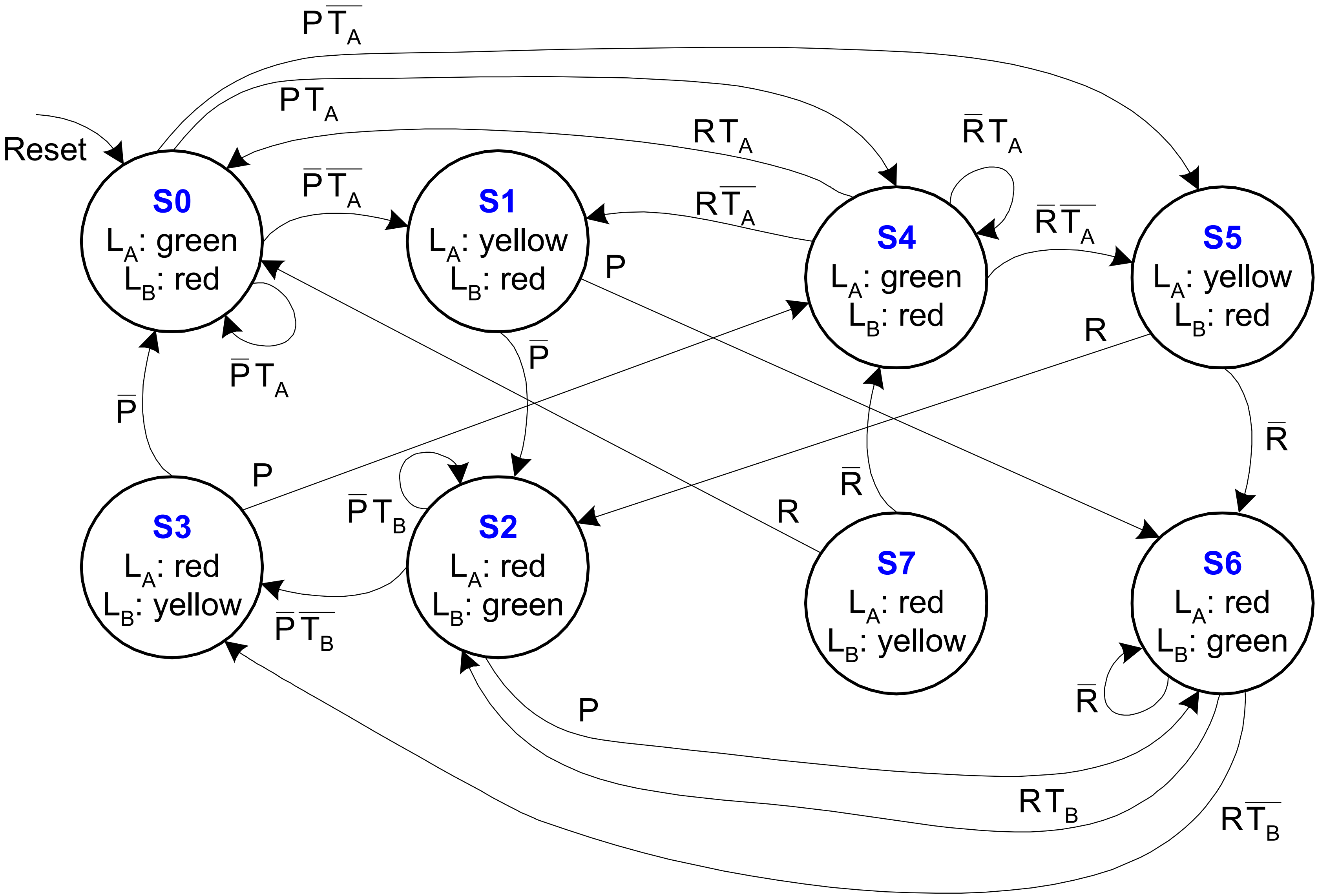
Unfactored FSM



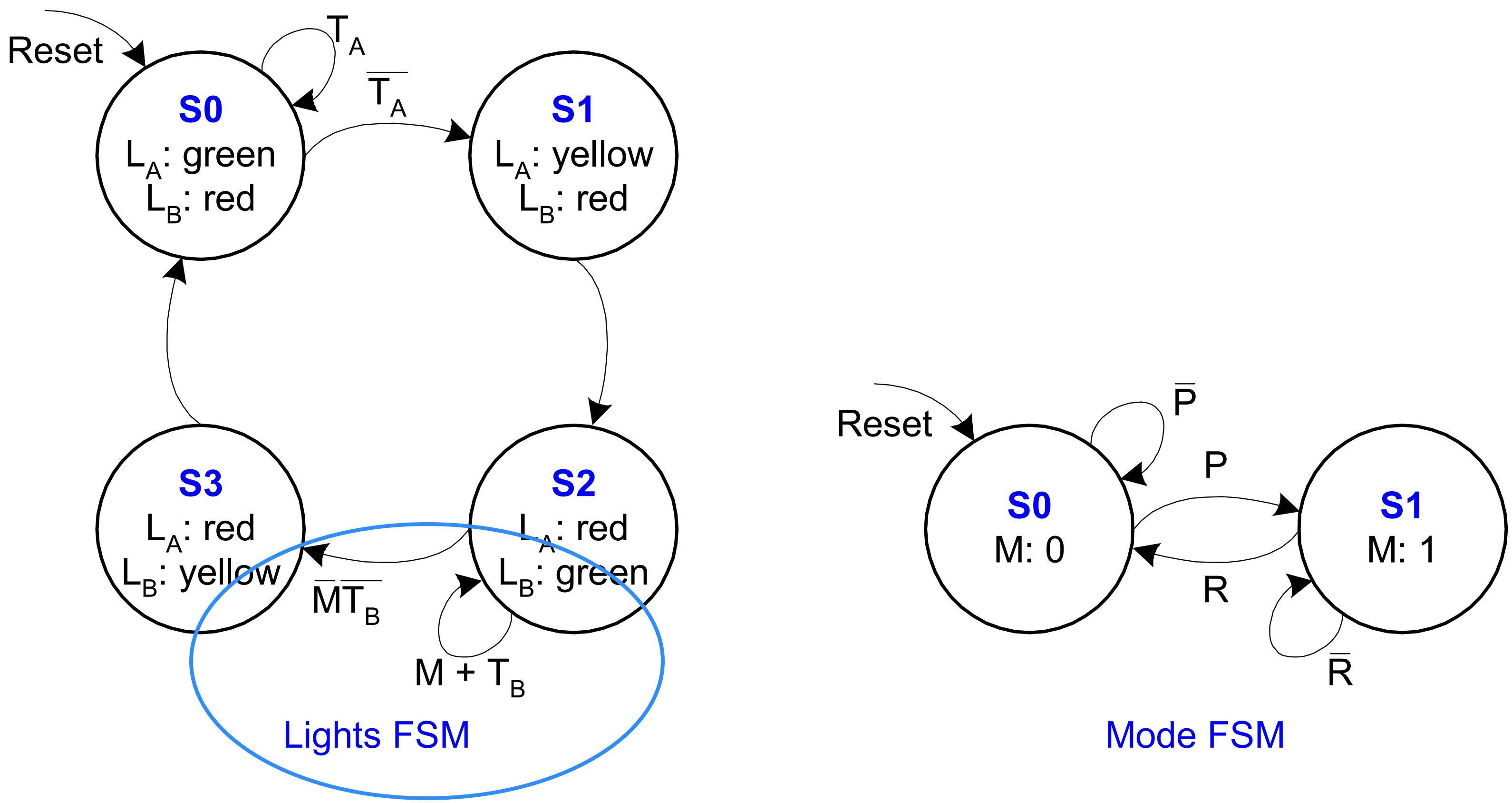
Factored FSM



Unfactored FSM State Transition Diagram



Factored FSM State Transition Diagram



FSM Design Procedure

- **Prepare**
  - Identify the inputs and outputs
  - Sketch a state transition diagram
  - Write a state transition table
  - Select state encodings
- **For a *Moore* machine**:
  - Rewrite the state transition table with the selected state encodings
  - Write the output table
- **For a *Mealy* machine:**
  - Rewrite the combined state transition and output table with the selected state encodings
- **Write Boolean equations for the next state and output logic**
- **Sketch the circuit schematic**

What Did We Learn?

- **D Latch is the basic memorizing element**
  - Transparent mode, copies input to output
  - Latch mode, keeps content
- **(Rising) Edge Triggered Flip-Flops are more practical**
  - Input is copied to output when the clock rises from 0 to 1
- **Finite State Machines**
  - ***Moore***, output depends on **only** the **current state**
  - ***Mealy***, output depends on **current state and the inputs**.
- **Three Aspects of an FSM**
  - Holds the present state
  - Calculate the next state
  - Determine the outputs

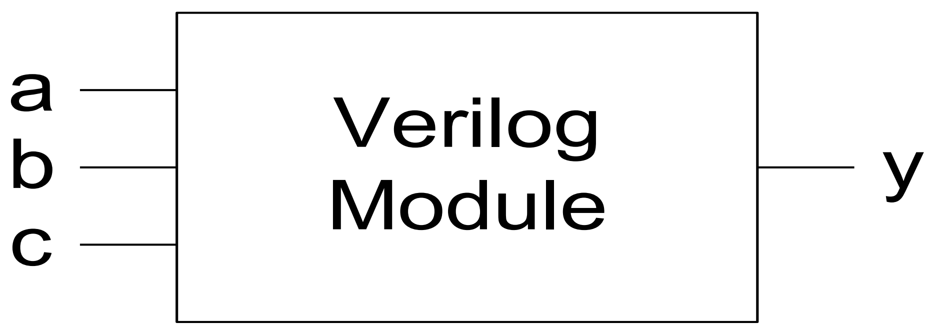
What will we learn?

- Short summary of Verilog Basics
- Sequential Logic in Verilog
- Using Sequential Constructs for Combinational Design
- Finite State Machines

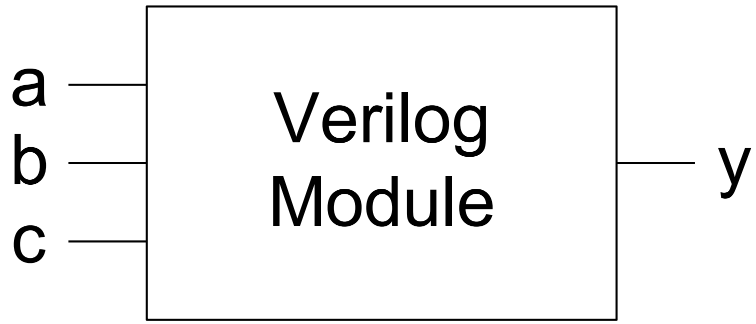
Summary: Defining a module

- **A module is the main building block in Verilog**

- **We first need to declare:**
  - Name of the module
  - Types of its connections (input, output)
  - Names of its connections



Summary: Defining a module




```
module example (a, b, c, y);
      input a;
      input b;
      input c;
      output y;

// here comes the circuit description

endmodule
```

Summary: What if we have busses ?

- **You can also define multi-bit busses.**
  - [range_start : range_end]

```
input  [31:0] a;  // a[31], a[30] .. a[0]
output [15:8] b1; // b1[15], b1[14] .. b1[8]
output [7:0]  b2; // b2[7], b2[6] .. b1[0]
input         clk;
```

Structural HDL Example

Short Instantiation

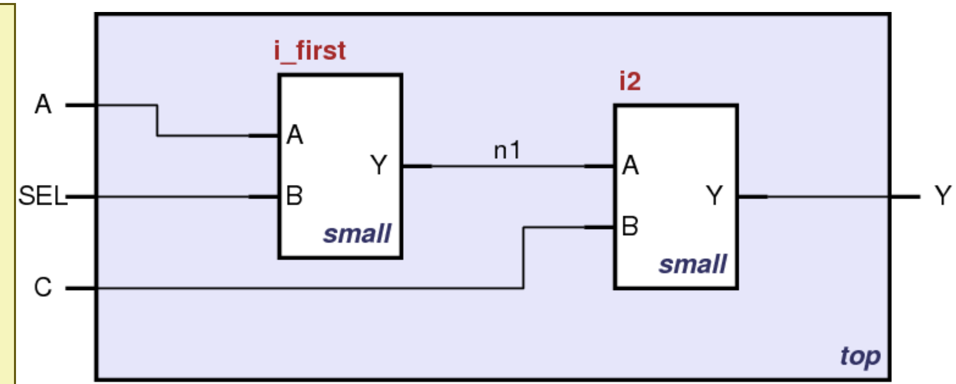
```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

// alternative
small i_first ( A, SEL, n1 );

/* Shorter instantiation,
   pin order very important */

// any pin order, safer choice
small i2 ( .B(C),
           .Y(Y),
           .A(n1) );

endmodule
```




```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

Summary: Bitwise Operators

```
module gates(input  [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);

   /* Five different two-input logic
      gates acting on 4 bit busses */

   assign y1 = a & b;    // AND
   assign y2 = a | b;    // OR
   assign y3 = a ^ b;    // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR

endmodule
```

Summary: Conditional Assignment

- **? : is also called a ternary operator because it operates on 3 inputs:**
  - s
  - d1
  - d0.

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

   assign y = s ? d1 : d0;
   // if (s) then y=d1 else y=d0;

endmodule
```

Summary: How to Express numbers ?

N’Bxx

`8’b0000_0001`

- **(N) Number of bits**
  - Expresses how many bits will be used to store the value
- **(B) Base**
  - Can be b (binary), h (hexadecimal), d (decimal), o (octal)
- **(xx) Number**
  - The value expressed in base, apart from numbers it can also have X and Z as values.
  - Underscore _ can be used to improve readability

Summary: Verilog Number Representation

Verilog	Stored Number	Verilog	Stored Number
4’b1001	1001	4’d5	0101
8’b1001	0000 1001	12’hFA3	1111 1010 0011
8’b0000_1001	0000 1001	8’o12	00 001 010
8’bxX0X1zZ1	XX0X 1ZZ1	4’h7	0111
‘b01	0000 .. 0001	12’h0	0000 0000 0000

Precedence of Operations in Verilog

<table>
<tr><td>Highest</td><td>~</td><td>NOT</td></tr>
<tr><td></td><td>*, /, %</td><td>mult, div, mod</td></tr>
<tr><td></td><td>+, -</td><td>add,sub</td></tr>
<tr><td></td><td><<, >></td><td>shift</td></tr>
<tr><td></td><td><<<, >>></td><td>arithmetic shift</td></tr>
<tr><td></td><td><, <=, >, >=</td><td>comparison</td></tr>
<tr><td></td><td>==, !=</td><td>equal, not equal</td></tr>
<tr><td></td><td>&, ~&</td><td>AND, NAND</td></tr>
<tr><td></td><td>^, ~^</td><td>XOR, XNOR</td></tr>
<tr><td></td><td>|, ~|</td><td>OR, NOR</td></tr>
<tr><td>Lowest</td><td>?:</td><td>ternary operator</td></tr>
</table>

Sequential Logic in Verilog

- **Define blocks that have memory**
  - Flip-Flops, Latches, Finite State Machines
- **Sequential Logic is triggered by a ‘CLOCK’ event**
  - Latches are sensitive to level of the signal
  - Flip-flops are sensitive to the transitioning of clock
- **Combinational constructs are not sufficient**
  - We need new constructs:
    - `always`
    - `initial`

always Statement, Defining Processes

```
always @ (sensitivity list)
      statement;
```

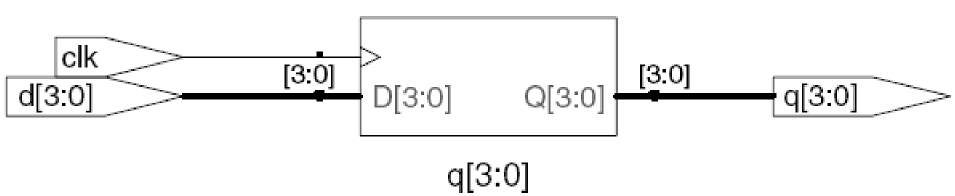
- **Whenever the event in the sensitivity list occurs, the statement is executed**

Example: D Flip-Flop

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```



Example: D Flip-Flop

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                // pronounced “q gets d”

endmodule
```

- The posedge defines a rising edge (transition from 0 to 1).
- This process will trigger only if the clk signal rises.
- Once the clk signal rises: the value of d will be copied to q

Example: D Flip-Flop

```
module flop(input             clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;               // pronounced “q gets d”

endmodule
```

- **‘assign’ statement is not used within always block**
- **The <= describes a ‘non-blocking’ assignment**
  - We will see the difference between ‘blocking assignment’ and ‘non-blocking’ assignment in a while

Example: D Flip-Flop

```
module flop(input              clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                 // pronounced “q gets d”

endmodule
```

- **Assigned variables need to be declared as reg**
- **The name reg does not necessarily mean that the value is a register. (It could be, it does not have to be).**
- **We will see examples later**

D Flip-Flop with Asynchronous Reset

```
module flop_ar (input               clk,
                input               reset,
                input         [3:0] d,
                output reg    [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
       if (reset == ‘0’) q <= 0;   // when reset
       else              q <= d;   // when clk
    end
endmodule
```

- **In this example: two events can trigger the process:**
  - A ***rising edge*** on clk
  - A ***falling edge*** on reset

D Flip-Flop with Asynchronous Reset

```
module flop_ar (input                  clk,
                input                  reset,
                input            [3:0] d,
                output reg       [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
      if (reset == '0') q <= 0;    // when reset
      else              q <= d;    // when clk
    end
endmodule
```

- **For longer statements a begin end pair can be used**
  - In this example it was not necessary
- **The always block is *highlighted***

D Flip-Flop with Asynchronous Reset

```
module flop_ar (input                clk,
                input                reset,
                input          [3:0] d,
                output reg     [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
      if (reset == ‘0’) q <= 0;    // when reset
      else              q <= d;    // when clk
    end
endmodule
```

- **First reset is checked, if reset is 0, q is set to 0.**
  - This is an ‘asynchronous’ reset as the reset does not care what happens with the clock

- **If there is no reset then normal assignment is made**

D Flip-Flop with Synchronous Reset

```
module flop_sr (input              clk,
                input              reset,
                input        [3:0] d,
                output reg   [3:0] q);

  always @ (posedge clk)
    begin
       if (reset == ‘0’) q <= 0;   // when reset
       else              q <= d;   // when clk
    end
endmodule
```

- **The process is only sensitive to clock**
  - Reset ***only happens*** when the ***clock rises***. This is a ‘synchronous’ reset

- **A small change, has a large impact on the outcome**

D Flip-Flop with Enable and Reset

```
module flop_ar (input              clk,
                input              reset,
                input              en,
                input        [3:0] d,
                output reg   [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
       if (reset == ‘0’) q <= 0;   // when reset
       else if (en)      q <= d;   // when en AND clk
    end
endmodule
```

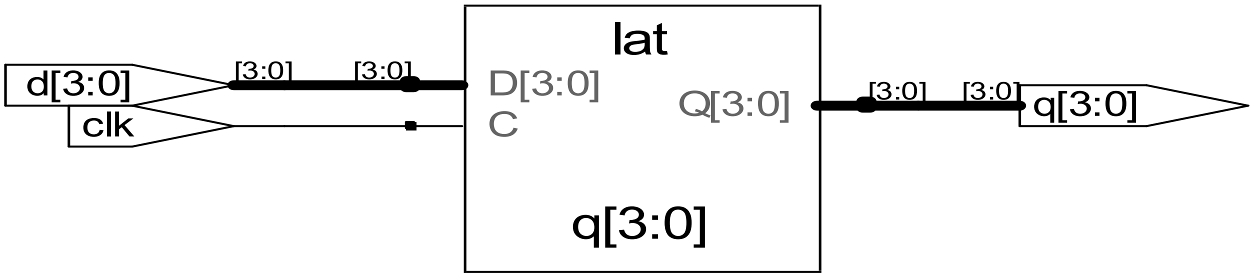
- **A flip-flop with enable and reset**
  - Note that the en signal is ***not*** in the sensitivity list

- **Only when “clk is rising” *AND* “en is 1” data is stored**

Example: D Latch

```
module latch (input             clk,
              input       [3:0] d,
              output reg [3:0] q);

  always @ (clk, d)
    if (clk) q <= d;       // latch is transparent when
                           // clock is 1
endmodule
```



Summary: Sequential Statements so far

- Sequential statements are within an ‘`always`’ block
- The sequential block is triggered with a change in the sensitivity list
- Signals assigned within an always must be declared as `reg`
- We use `<=` for (non-blocking) assignments and do not use ‘`assign`’ within the always block.

Summary: Basics of always Statements

```
module example (input            clk,
                input      [3:0] d,
                output reg [3:0] q);

  wire [3:0] normal;         // standard wire
  reg  [3:0] special;        // assigned in always

  always @ (posedge clk)
    special <= d;            // first FF array

  assign normal = ~ special; // simple assignment

  always @ (posedge clk)
    q <= normal;             // second FF array
endmodule
```

- You can have many always blocks

Summary: Basics of always Statements

```
module example (input            clk,
                input      [3:0] d,
                output reg [3:0] q);

  wire [3:0] normal;           // standard wire
  reg  [3:0] special;          // assigned in always

  always @ (posedge clk)
    special <= d;              // first FF array

  assign normal = ~ special;   // simple assignment

  always @ (posedge clk)
    q <= normal;               // second FF array
endmodule
```

- **Assignments are different within always blocks**

Why does an always Statement Memorize?

```
module flop (input              clk,
             input        [3:0] d,
             output reg [3:0] q);

  always @ (posedge clk)
    begin
        q <= d;   // when clk rises copy d to q
    end
endmodule
```

- This statement describes what happens to signal q
- … but what happens when clock is not rising?

Why does an always Statement Memorize?

```
module flop (input            clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @ (posedge clk)
    begin
        q <= d;   // when clk rises copy d to q
    end
endmodule
```

- **This statement describes what happens to signal q**
- **... but what happens when clock is not rising?**
- **The value of q is preserved (memorized)**

Why does an always Statement Memorize?

```
module comb (input              inv,
             input        [3:0] data,
             output reg [3:0] result);

  always @ (inv, data)       // trigger with inv, data
    if (inv) result <= ~data;// result is inverted data
    else     result <= data; // result is data

endmodule
```

- **This statement describes what happens to signal result**
  - When inv is 1, result is ~data
  - What happens when inv is ***not 1*** ?

Why does an always Statement Memorize?

```
module comb (input              inv,
             input       [3:0] data,
             output reg  [3:0] result);

  always @ (inv, data)        // trigger with inv, data
    if (inv) result <= ~data;// result is inverted data
    else     result <= data; // result is data

endmodule
```

- **This statement describes what happens to signal result**
  - When `inv` is 1, result is `~data`
  - When `inv` is not 1, result is `data`

- **Circuit is combinational (no memory)**
  - The output (`result`) is defined for all possible inputs (`inv data`)

always Blocks for Combinational Circuits

- **If the statements define the signals completely, nothing is memorized, block becomes combinational.**
  - Care must be taken, it is easy to make mistakes and unintentionally describe memorizing elements (latches).

- **Always blocks allow powerful statements**
  - `if .. then .. else`
  - `case`

- **Use always blocks only if it makes your job easier**

Always Statement is not Always Practical…

```
  reg  [31:0] result;
  wire [31:0] a, b, comb;
  wire        sel,

  always @ (a, b, sel)    // trigger with a, b, sel
    if (sel) result <= a; // result is a
    else     result <= b; // result is b

  assign comb = sel ? a : b;

endmodule
```

- **Both statements describe the same multiplexer**
- **In this case, the always block is more work**

Sometimes Always Statements are Great

```
module sevensegment (input      [3:0] data,
                     output reg [6:0] segments);

  always @ ( * )                  // * is short for all signals
    case (data)                   // case statement
      4'd0: segments = 7'b111_1110;  // when data is 0
      4'd1: segments = 7'b011_0000;  // when data is 1
      4'd2: segments = 7'b110_1101;
      4'd3: segments = 7'b111_1001;
      4'd4: segments = 7'b011_0011;
      4'd5: segments = 7'b101_1011;
      // etc etc
      default: segments = 7'b000_0000; // required
    endcase

endmodule
```

The case Statement

- **Like `if .. then .. else` can only be used in always blocks**
- **The result is combinational only if the output is defined for all cases**
  - Did we mention this before ?
- **Always use a `default` case to make sure you did not forget a case (which would infer a latch)**
- **Use `casez` statement to be able to check for don't cares**
  - See book page 202, example 4.28

Non-blocking and Blocking Statements

Non-blocking

```
always @ (a)
begin
   a <= 2'b01;
   b <= a;
// all assignments are made here
// b is not (yet) 2'b01
end
```

- **Values are assigned at the end of the block.**
- **All assignments are made in parallel, process flow is not-blocked.**

Blocking

```
always @ (a)
begin
   a = 2'b01;
// a is 2'b01
   b = a;
// b is now 2'b01 as well
end
```

- **Value is assigned immediately.**
- **Process waits until the first assignment is complete, it blocks progress.**

Why use (Non)-Blocking Statements

- **There are technical reasons why both are required**
  - It is out of the scope of this course to discuss these
- **Blocking statements allow sequential descriptions**
  - More like a programming language
- **If the sensitivity list is correct, blocks with non-blocking statements will always evaluate to the same result**
  - It may require some additional iterations

Example: Blocking Statements

- **Assume all inputs are initially '0'**

```
always @ ( * )
  begin
    p    = a ^ b ;          // p    = 0
    g    = a & b ;          // g    = 0
    s    = p ^ cin ;        // s    = 0
    cout = g | (p & cin) ;  // cout = 0
  end
```

Example: Blocking Statements

- Now a changes to ‘1’

```
always @ ( * )
  begin
    p    = a ^ b ;          // p    = 1
    g    = a & b ;          // g    = 0
    s    = p ^ cin ;        // s    = 1
    cout = g | (p & cin) ; // cout = 0
  end
```

- The process triggers
- All values are updated in order
- At the end, s = 1

Same Example: Non-Blocking Statements

- **Assume all inputs are initially '0'**

```
always @ ( * )
  begin
    p    <= a ^ b ;         // p    = 0
    g    <= a & b ;         // g    = 0
    s    <= p ^ cin ;       // s    = 0
    cout <= g | (p & cin) ; // cout = 0
  end
```

Same Example: Non-Blocking Statements

- Now a changes to ‘1’

```
always @ ( * )
  begin
    p    <= a ^ b ;          // p    = 1
    g    <= a & b ;          // g    = 0
    s    <= p ^ cin ;        // s    = 0
    cout <= g | (p & cin) ; // cout = 0
  end
```

- The process triggers
- All assignments are concurrent
- When s is being assigned, p is still 0, result is still 0

Same Example: Non-Blocking Statements

- After the first iteration **p** has changed to '1' as well

```
always @ ( * )
  begin
    p    <= a ^ b ;         // p    = 1
    g    <= a & b ;         // g    = 0
    s    <= p ^ cin ;       // s    = 1
    cout <= g | (p & cin) ; // cout = 0
  end
```

- Since there is a change in **p**, process triggers again
- This time **s** is calculated with **p=1**
- The result is correct after the second iteration

Rules for Signal Assignment

- Use `always @(posedge clk)` and non-blocking assignments (`<=`) to model synchronous sequential logic

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

- Use continuous assignments (`assign …`)to model simple combinational logic.

```
assign y = a & b;
```

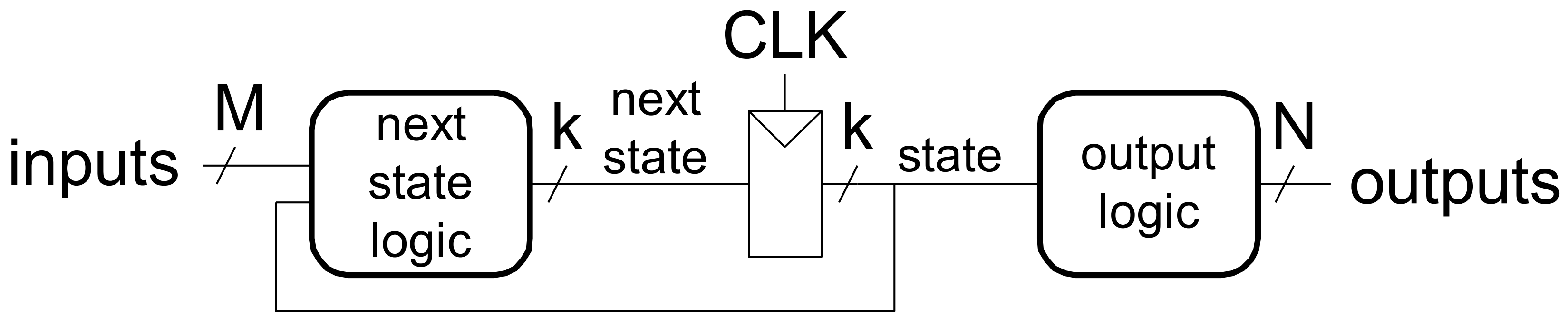
Rules for Signal Assignment (cont)

- Use `always @ (*)` and blocking assignments (`=`) to model more complicated combinational logic where the always statement is helpful.

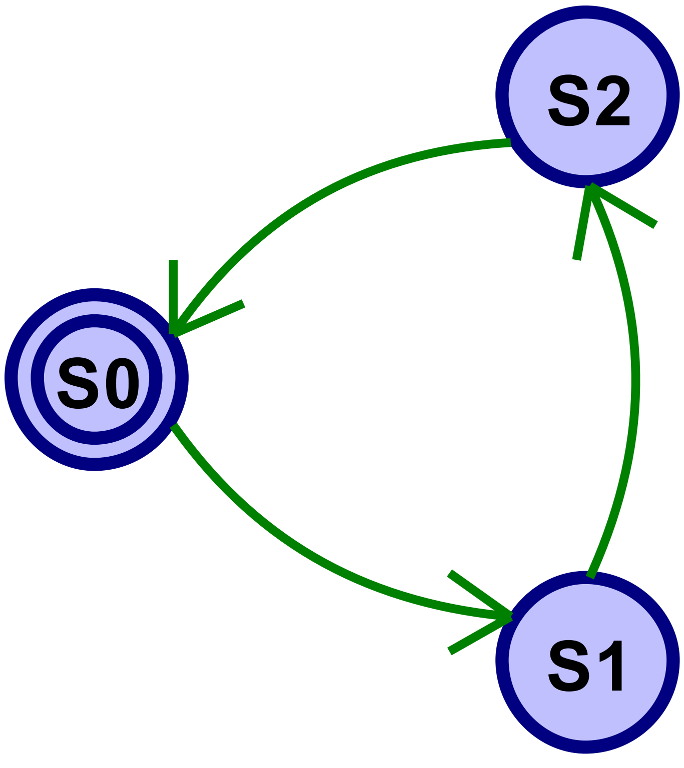
- Do not make assignments to the same signal in more than one always statement or continuous assignment statement

Finite State Machines (FSMs)

- **Each FSM consists of three separate parts:**
  - next state logic
  - state register
  - output logic



FSM Example: Divide by 3



FSM in Verilog, Definitions

```
module divideby3FSM (input clk,
                     input reset,
                     output q);

   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;
```

- We define `state` and `nextstate` as 2-bit `reg`
- The parameter descriptions are optional, it makes reading easier

FSM in Verilog, State Register

```
// state register
   always @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
```

- **This part defines the state register (memorizing process)**
- **Sensitive to only clk, reset**
- **In this example reset is active when ‘1’**

FSM in Verilog, Next State Calculation

```
// next state logic
  always @ (*)
     case (state)
        S0:      nextstate = S1;
        S1:      nextstate = S2;
        S2:      nextstate = S0;
        default: nextstate = S0;
     endcase
```

- **Based on the value of state we determine the value of `nextstate`**
- **An `always .. case` statement is used for simplicity.**

FSM in Verilog, Output Assignments

```
// output logic
   assign q = (state == S0);
```

- In this example, output depends only on state
  - Moore type FSM
- We used a simple combinational assign

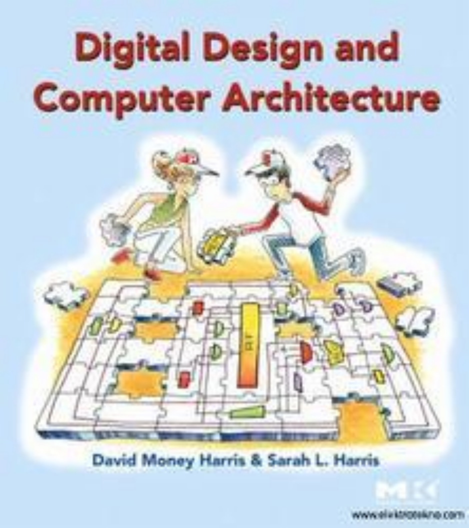
FSM in Verilog, Whole Code

```
module divideby3FSM (input clk, input reset, output q);
   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   always @ (posedge clk, posedge reset) // state register
      if (reset) state <= S0;
      else       state <= nextstate;
   always @ (*)                          // next state logic
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
   assign q = (state == S0);            // output logic
endmodule
```

A word about the examples



- **All examples in the slides are 1:1 from our book**
  - This should help you while studying
  - There is nothing wrong with the examples

- **We would just suggest to do things a bit differently**
  - Use sensible names for the states (not `S0`, `S1`, `S2`..)
  - Use `begin .. end` blocks for the `always` statements
  - Use a suffix to distinguish between next and present state
    - `state     = state_present     state_q`
    - `nextstate = state_next        state_d`

FSM in Verilog, Whole Code once again

```
module divideby3FSM (input clk, input reset, output q);
   reg  [1:0] state_present, state_next;

   parameter init = 2'b00;
   parameter one  = 2'b01;
   parameter two  = 2'b10;

   always @ (posedge clk, posedge reset) begin // state register
      if (reset) state_present <= init;        //   asynchronous reset
      else       state_present <= state_next;  //   move to next state
   end
   always @ (*) begin                          // next state logic
      case (state_present)                     //   based on current state
         init:    state_next = one;            //   decide what to do
         one:     state_next = two;
         two:     state_next = init;
         default: state_next = init;           //   add a default
      endcase
    end
    assign q = (state_present == init);        // output logic
endmodule
```

What Did We Learn?

- **Basics of Defining Sequential Circuits in Verilog**
- **Always statement**
  - Is needed for defining memorizing elements (flip-flops, latches)
  - Can also be used to define combinational circuits
- **Blocking vs Non-blocking statements**
  - = assigns the value immediately
  - <= assigns the value at the end of the block
- **Writing FSMs**
  - Next state calculation
  - Determining outputs
  - State assignment