Digital Design & Computer Arch.

Lecture 5a: Sequential Logic Design II
Finite State Machines

Prof. Onur Mutlu

ETH Zürich

Spring 2023

9 March 2023

First, We Will Complete Sequential Logic Design

We Covered A Lot of Sequential Logic

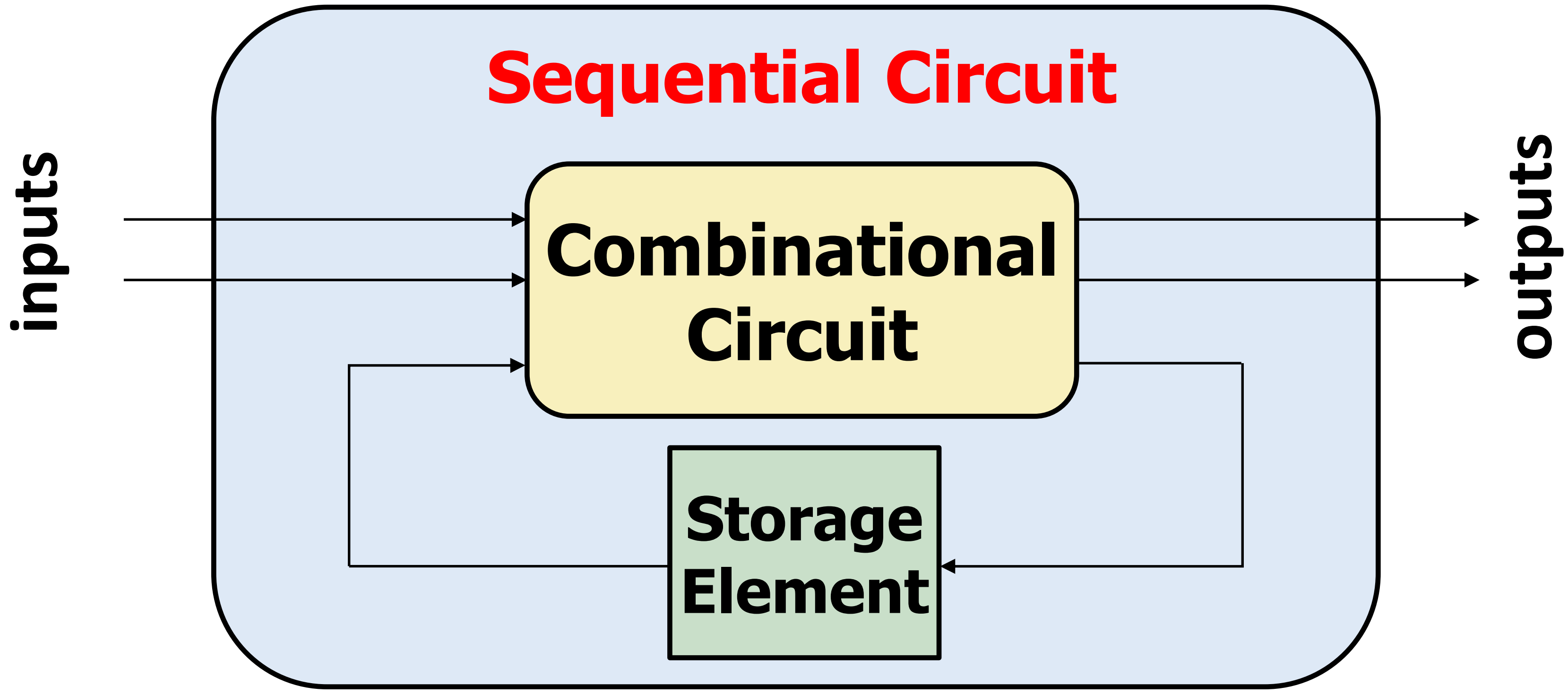
- **Circuits that can store information**
  - Cross-coupled inverter
  - R-S Latch
  - Gated D Latch
  - D Flip-Flop
  - Register
  - Memory

- **Sequential logic circuits**
  - State & Clock
  - Asynchronous vs. Synchronous

- **Finite State Machines (FSM)**
  - How to design FSMs

Recall: Sequential Circuits

- Circuits that produce output depending on **current** and **past** input values – circuits with **memory**



Recall: Sequential Logic Circuits

Combinational
Only depends on current inputs

Sequential
Opens depending on past inputs

https://www.easykeys.com/228_ESP_Combination_Lock.aspx
https://www.fosmon.com/product/tsa-approved-lock-4-dial-combo

Recall: State Diagram of Our Sequential Lock

- Completely describes the operation of the sequential lock

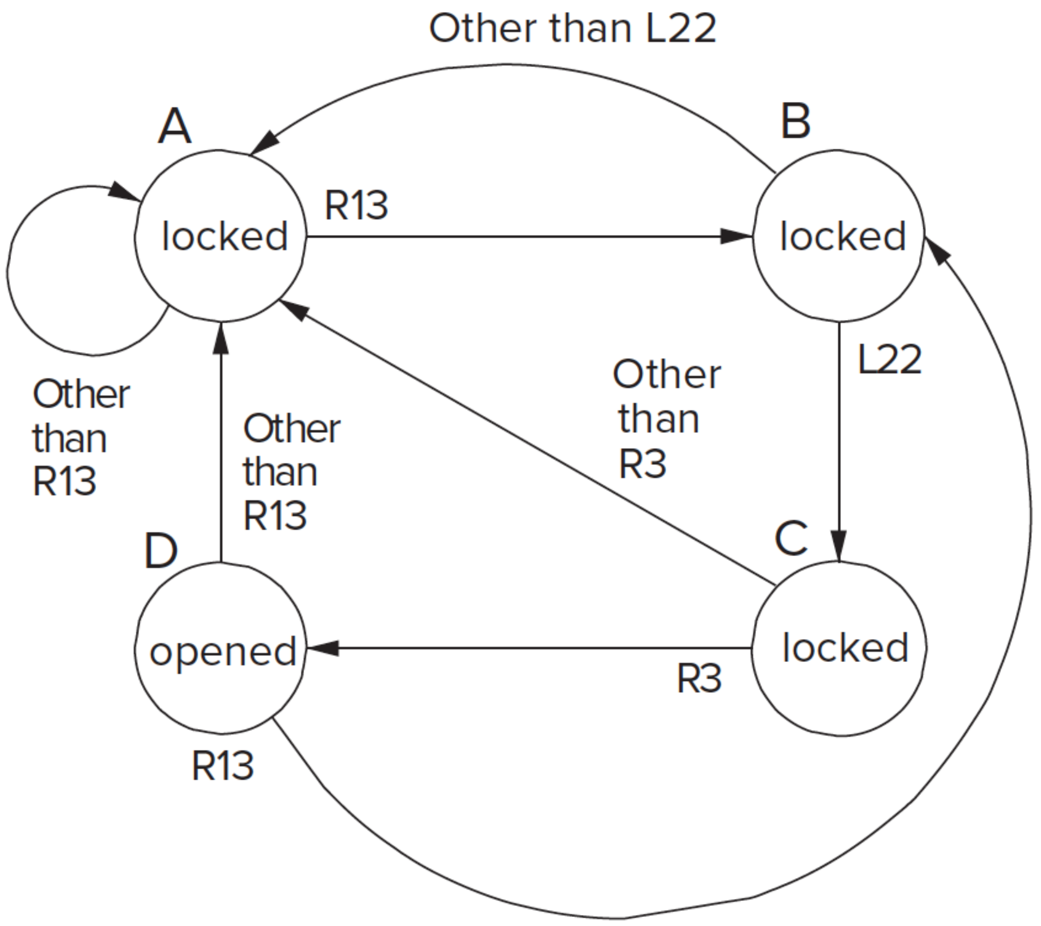


Image source: Patt and Patel, “Introduction to Computing Systems”, 2nd ed., page 76.

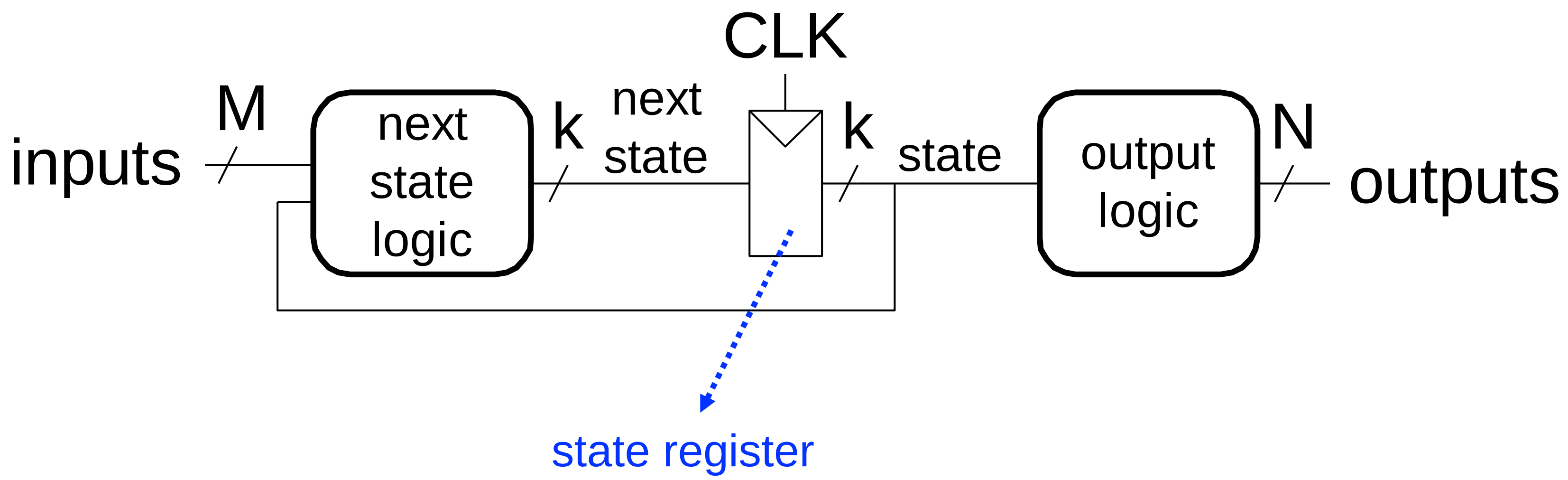
Recall: Finite State Machines (FSMs) Consist of:

- **Five elements:**
  1. A **finite** number of states
     - ***State***: snapshot of all relevant elements of the system at the time of the snapshot
  2. A **finite** number of external inputs
  3. A **finite** number of external outputs
  4. An explicit specification of all state transitions
     - How to get from one state to another
  5. An explicit specification of what determines each external output value

FSM: A discrete-time model of a stateful system

Recall: Finite State Machines (FSMs)

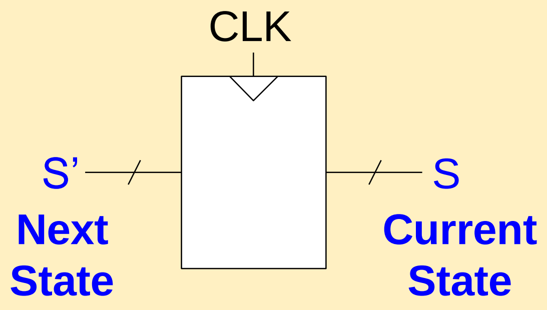
- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic



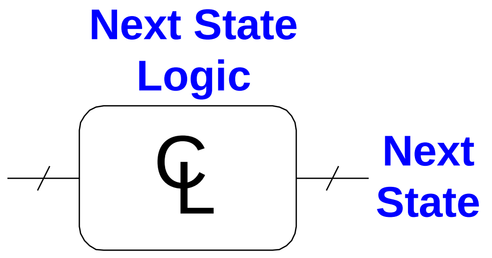
At the beginning of the clock cycle, next state is latched into the state register

Recall: Finite State Machines (FSMs) Consist of:

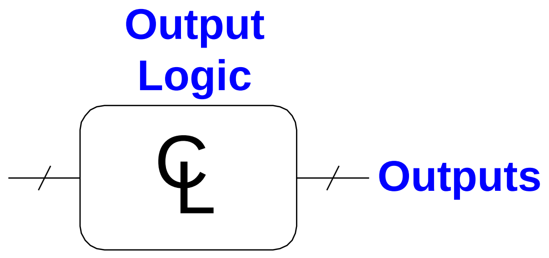
- **Sequential Circuits**
  - State register(s)
    - Store the current state and
    - Provide the next state at the clock edge



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



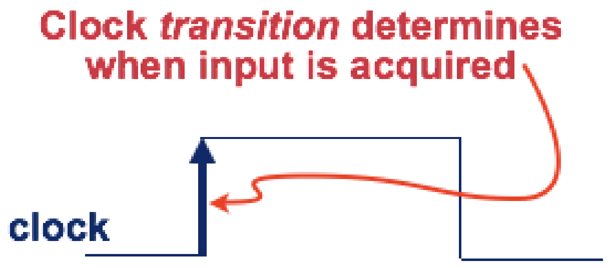
  - Output logic
    - Generates the outputs



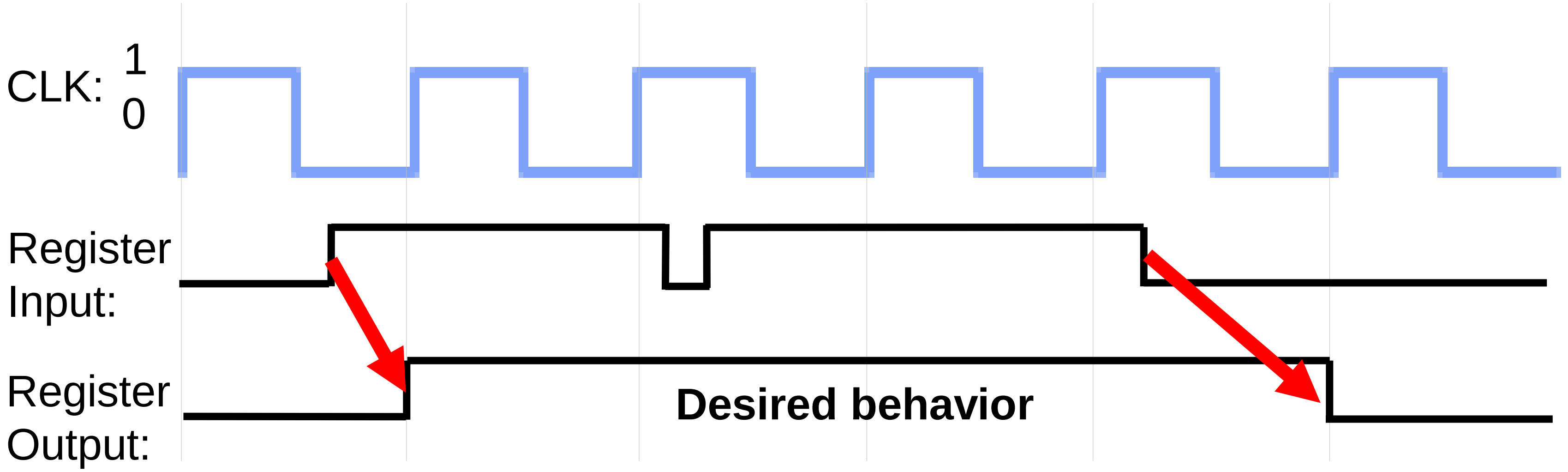
Recall: State Register Implementation

- How can we implement a **state register**? Two properties:

1. We need to store data at the **beginning** of every clock cycle

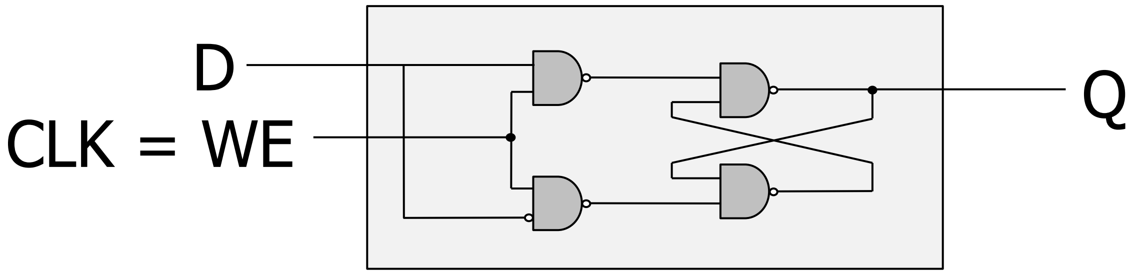


2. The data must be **available** during the **entire clock cycle**



Recall: The Problem with Latches: Transparency

Recall the
Gated D Latch



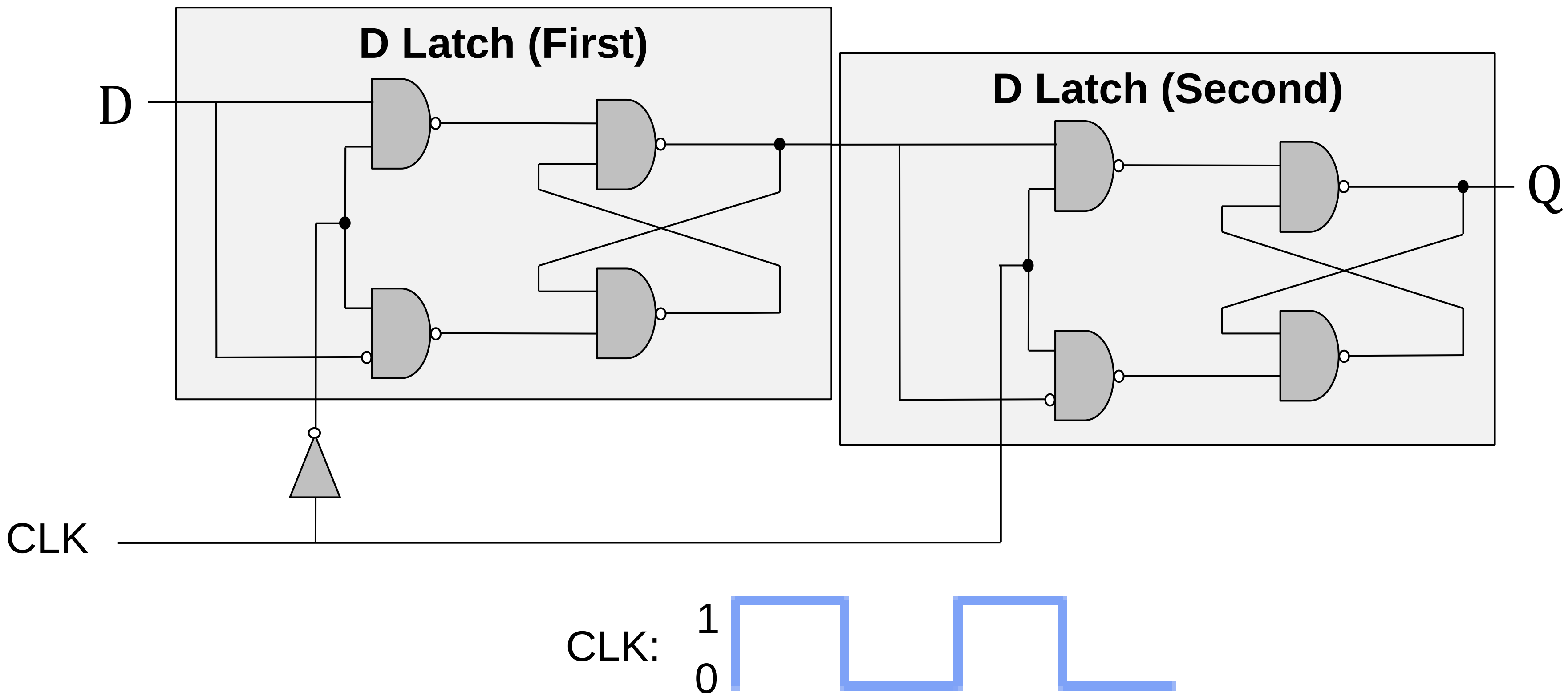
How can we change the latch, so that

1) D (input) is **observable** at **Q** (output) **only** at the **beginning of next** clock cycle?

2) Q is **available for the full clock cycle**

Recall: The D Flip-Flop

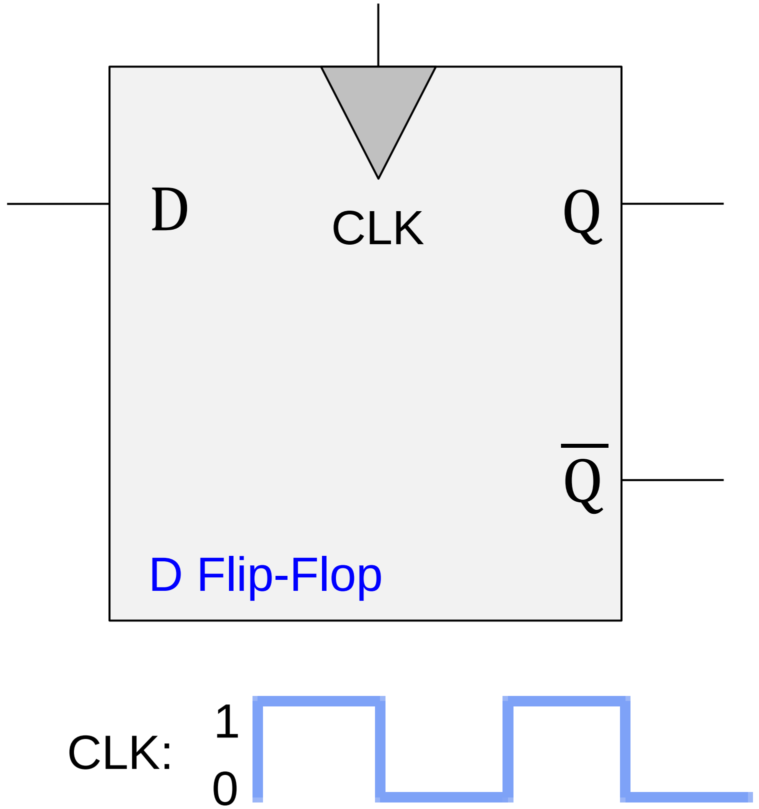
- 1) state change on clock edge, 2) data available for full cycle



- When the clock is low, 1st latch propagates **D** to the input of the 2nd (Q unchanged)
- Only when the clock is high, 2nd latch latches **D (Q stores D)**
  - At the rising edge of clock (clock going from 0->1), Q gets assigned D

Recall: The D Flip-Flop

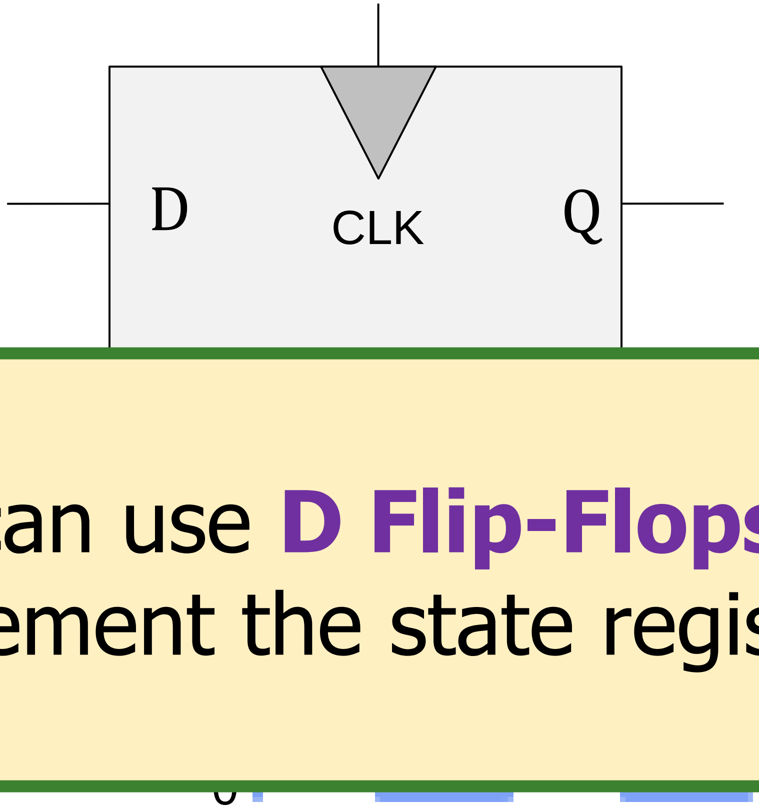
- 1) state change on clock edge, 2) data available for full cycle



- At the rising edge of clock (clock going from 0->1), **Q** gets assigned **D**
- At all other times, Q is unchanged

Recall: The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle



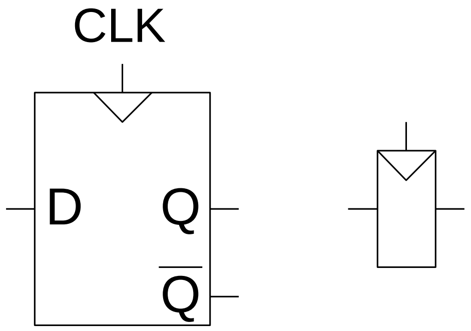
We can use **D Flip-Flops**
to implement the state register

- At the rising edge of clock (clock going from 0->1), **Q** gets assigned **D**
- At all other times, Q is unchanged

Recall: Rising-Clock-Edge Triggered Flip-Flop

- **Two inputs**: CLK, D

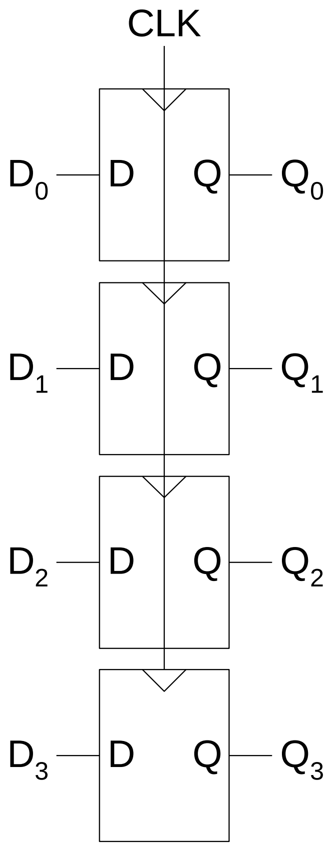
- **Function**
  - The flip-flop "samples" **D** on the rising edge of CLK (**positive edge**)
  - When CLK rises from 0 to 1, **D** passes through to **Q**
  - Otherwise, **Q** holds its previous value
  - **Q** changes **only** on the rising edge of CLK

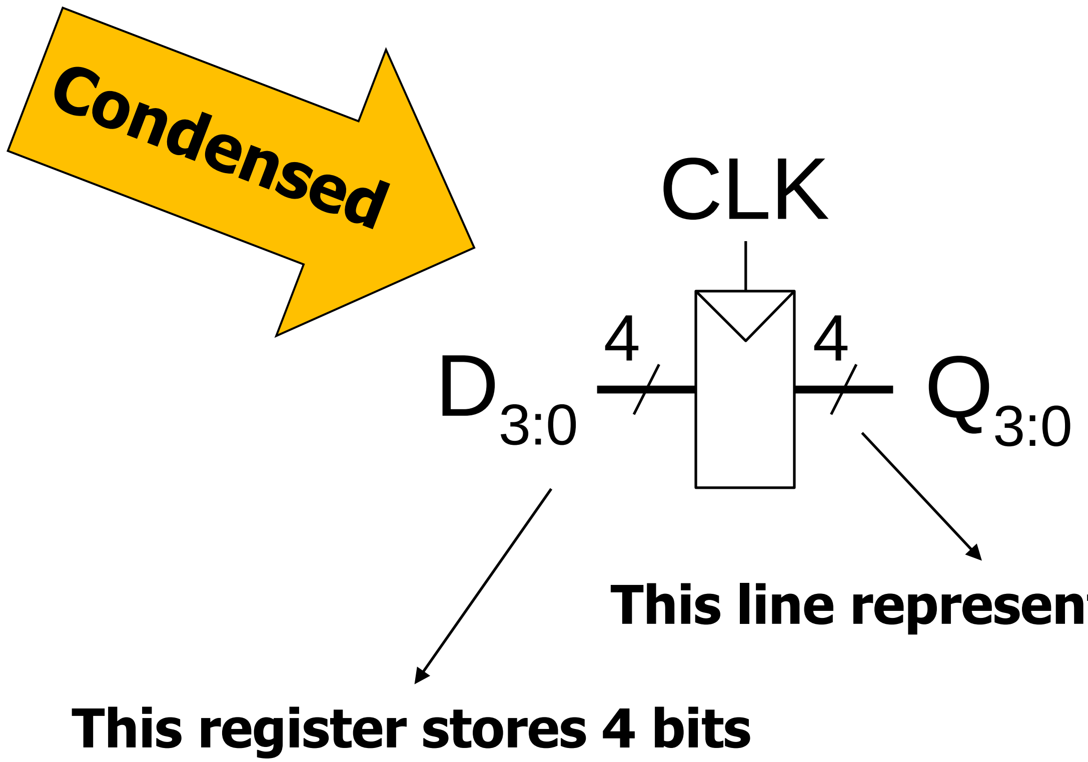


- A flip-flop is called an **edge-triggered state element** because it captures data on the clock edge
  - A latch is a **level-triggered** state element

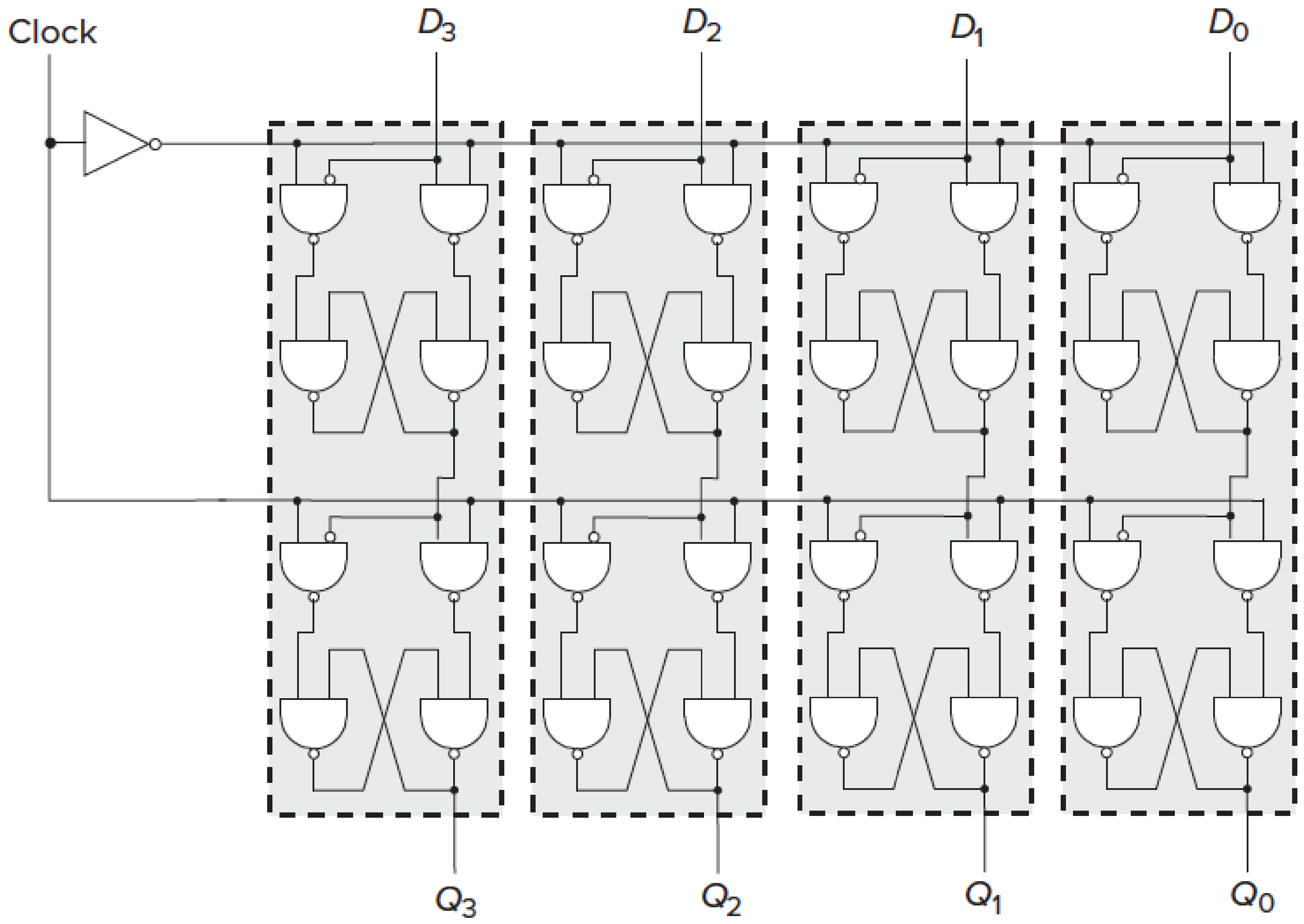
D Flip-Flop Based Register

- Multiple parallel D flip-flops, each of which storing 1 bit





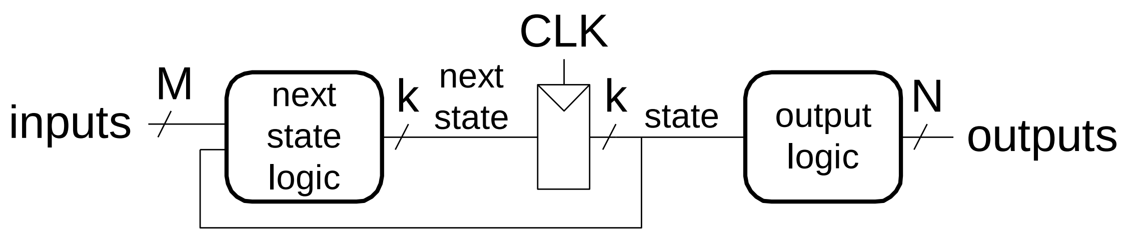
A 4-Bit D-Flip-Flop-Based Register (Internally)



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

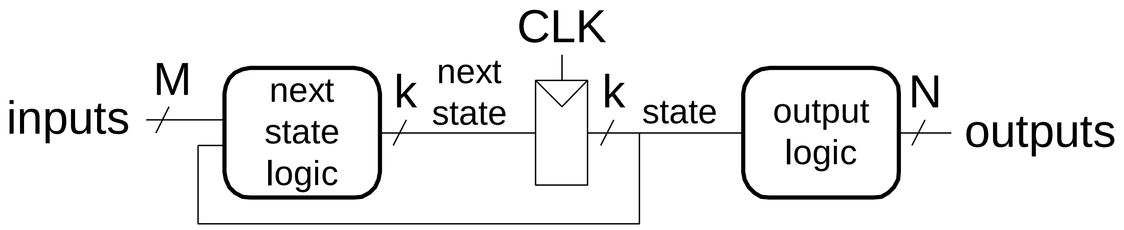
Moore FSM



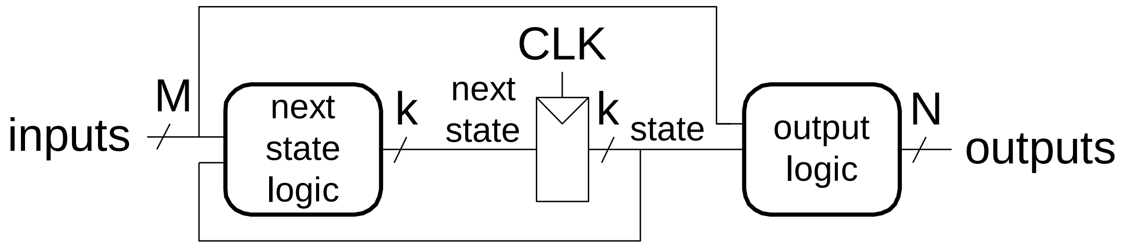
Finite State Machines (FSMs)

- 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

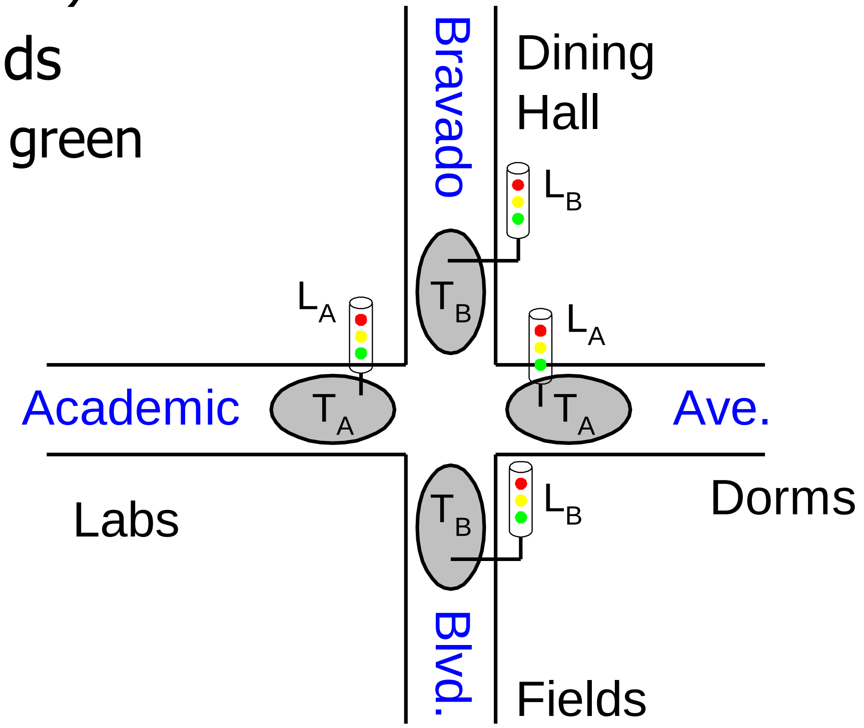


Mealy FSM



Finite State Machine Example

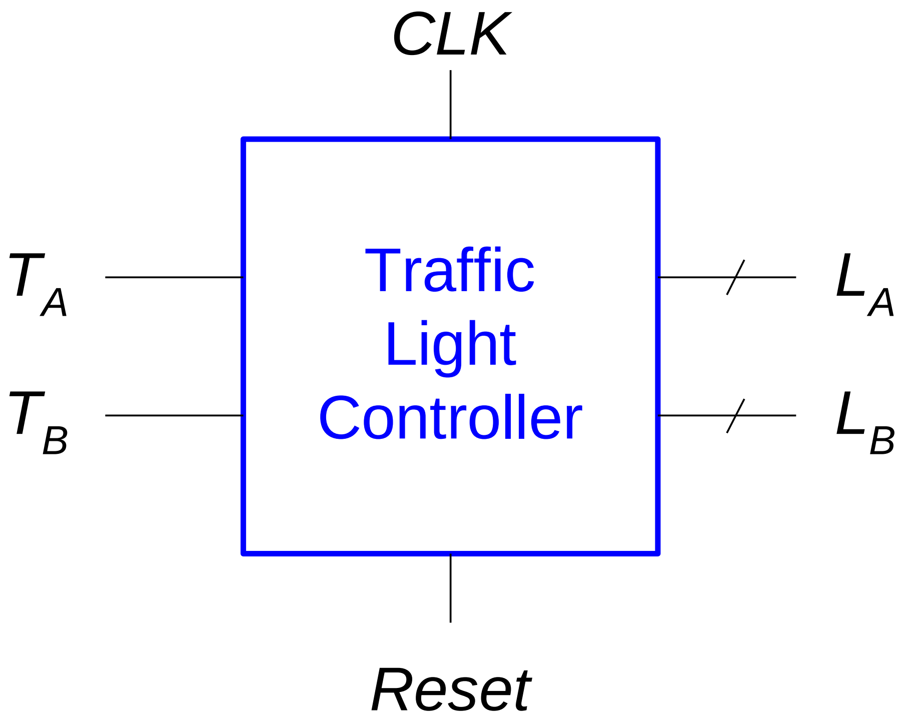
- "Smart" traffic light controller
  - **2 inputs**:
    - Traffic sensors: T_A , T_B (TRUE when there's traffic)
  - **2 outputs**:
    - Lights: L_A , L_B (Red, Yellow, Green)
  - State can change every 5 seconds
    - Except if green and traffic, stay green



From H&H Section 3.4.1

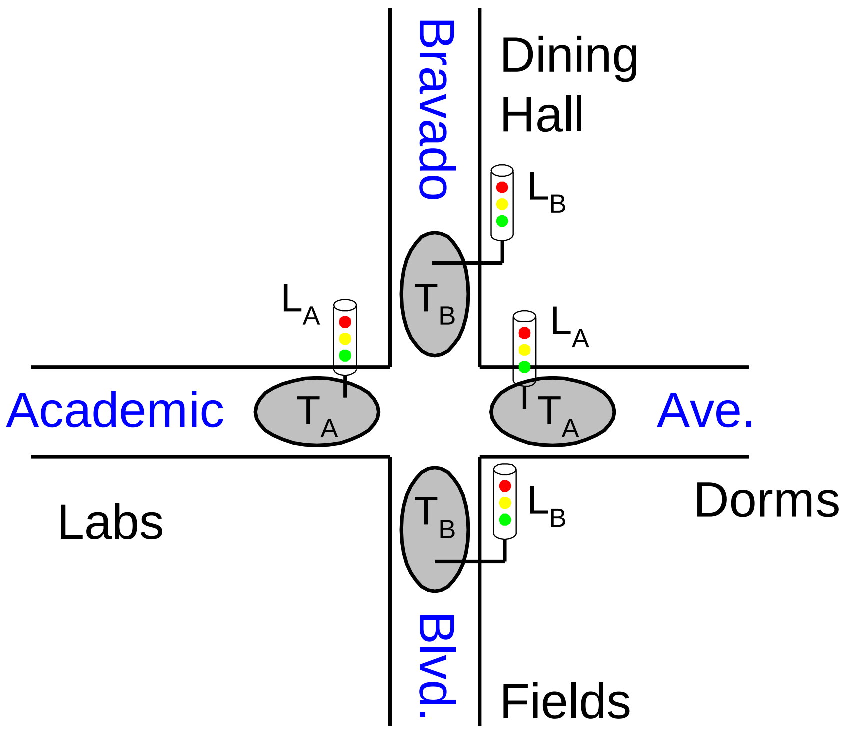
Finite State Machine Black Box

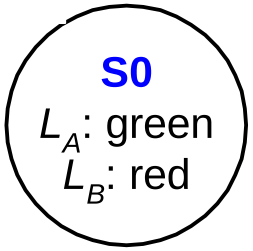
- **Inputs:** CLK, Reset, T_A , T_B
- **Outputs:** L_A , L_B



Finite State Machine Transition Diagram

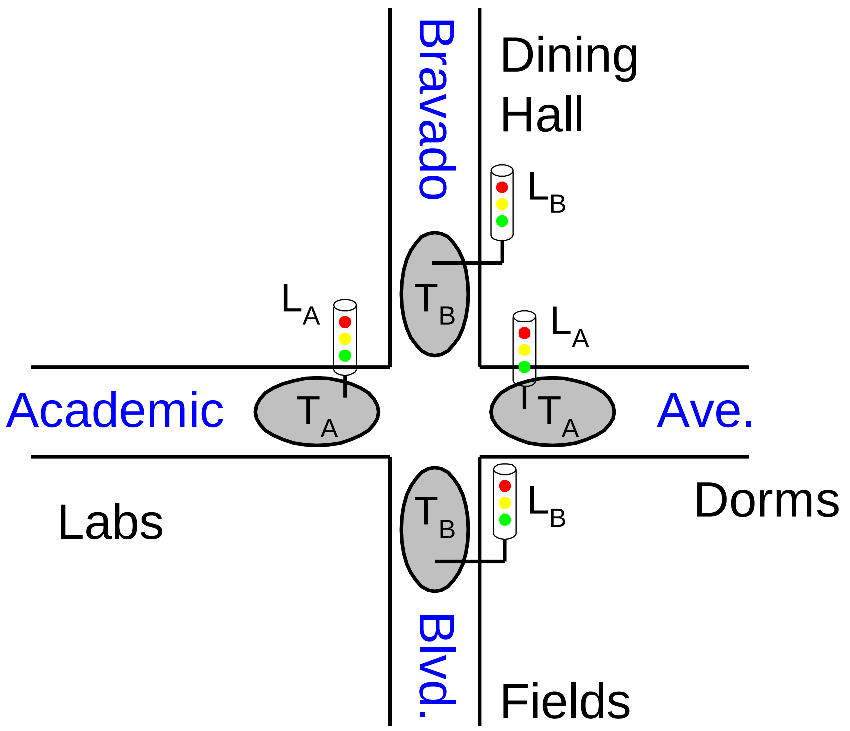
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

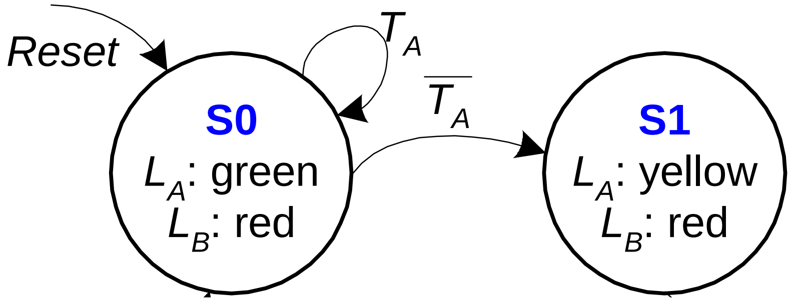




Finite State Machine Transition Diagram

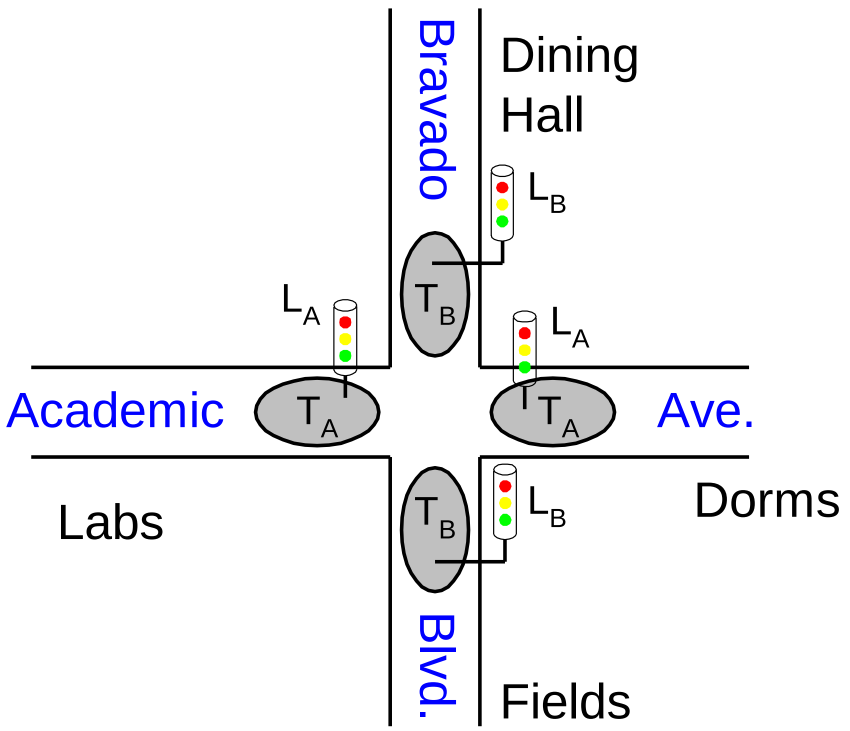
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs





Finite State Machine Transition Diagram

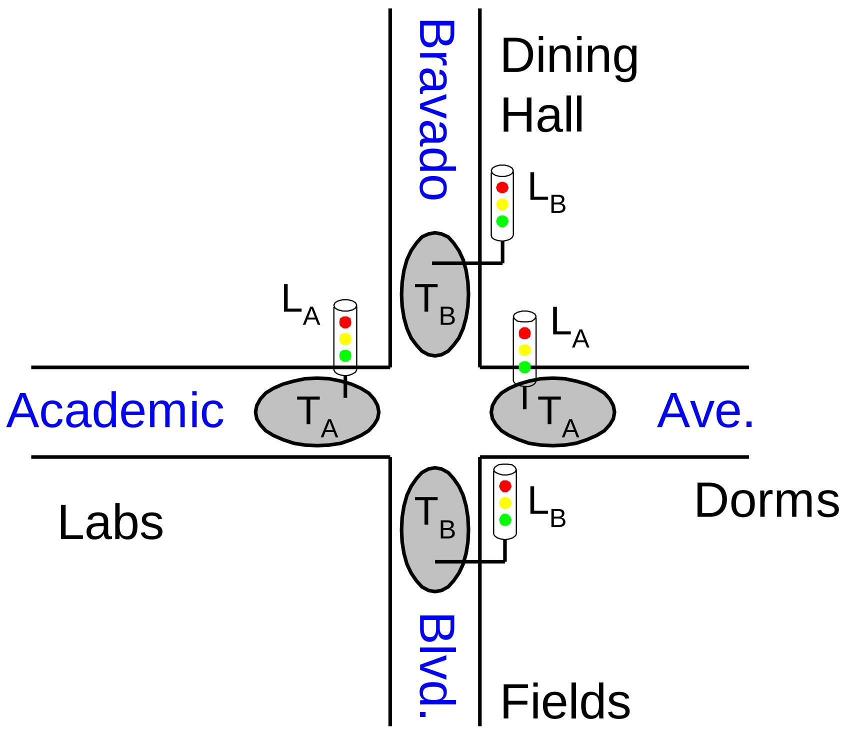
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

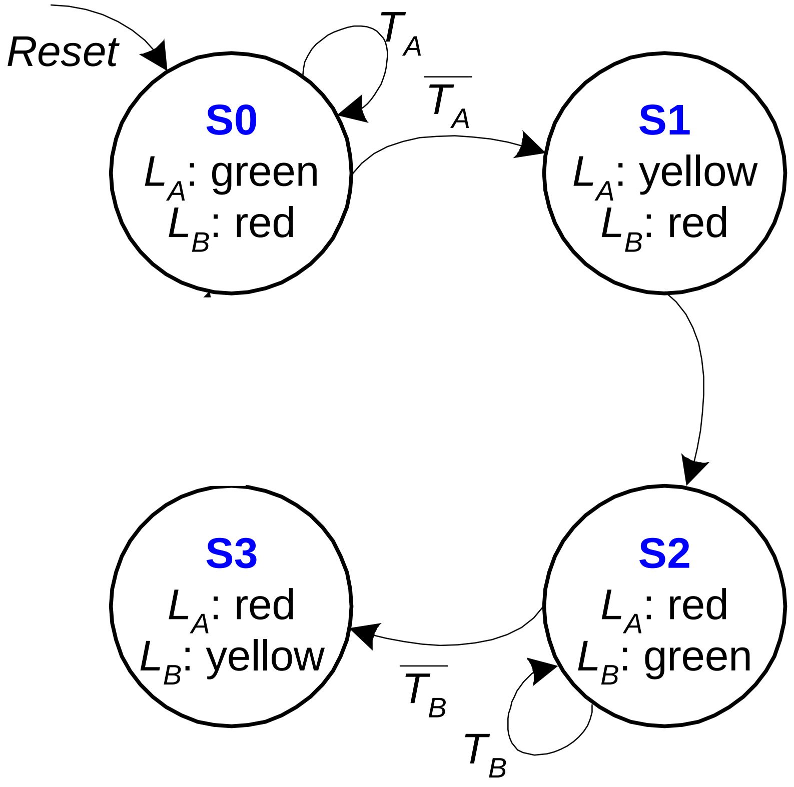


$$\begin{tikzcd}
\arrow[r, "Reset"] & \boxed{\begin{array}{c}\mathbf{S0}\\ L_A\text{: green}\\ L_B\text{: red}\end{array}} \arrow[loop above, "T_A"] \arrow[r, bend left, "\overline{T_A}"] & \boxed{\begin{array}{c}\mathbf{S1}\\ L_A\text{: yellow}\\ L_B\text{: red}\end{array}} \arrow[d] \\
& & \boxed{\begin{array}{c}\mathbf{S2}\\ L_A\text{: red}\\ L_B\text{: green}\end{array}} \arrow[loop below, "T_B"]
\end{tikzcd}$$

Finite State Machine Transition Diagram

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

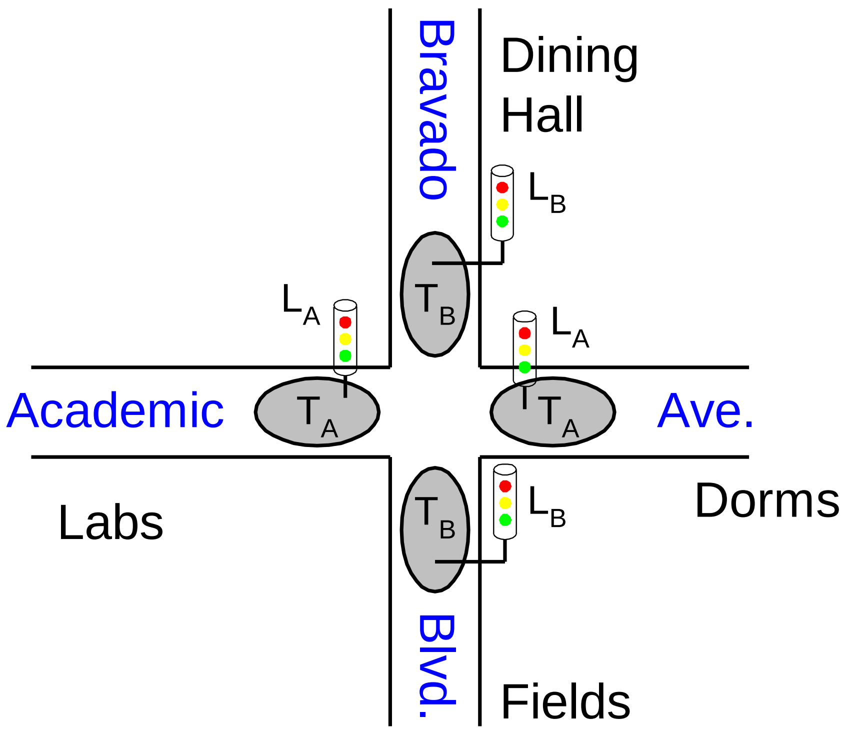


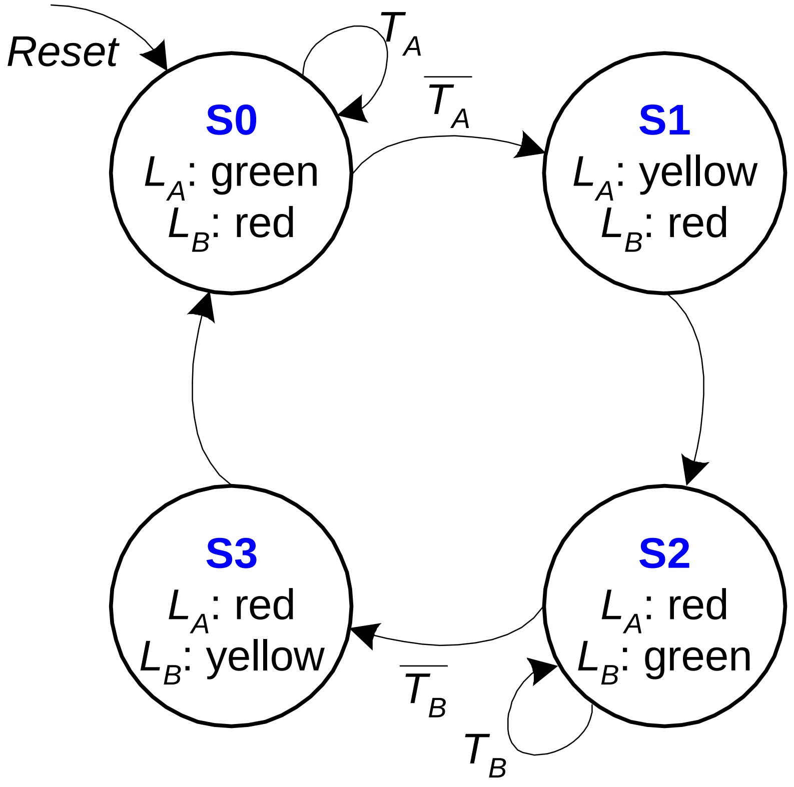


Reset
T_A
$\overline{T_A}$
S0 L_A: green L_B: red
S1 L_A: yellow L_B: red
S2 L_A: red L_B: green
S3 L_A: red L_B: yellow
$\overline{T_B}$
T_B

Finite State Machine Transition Diagram

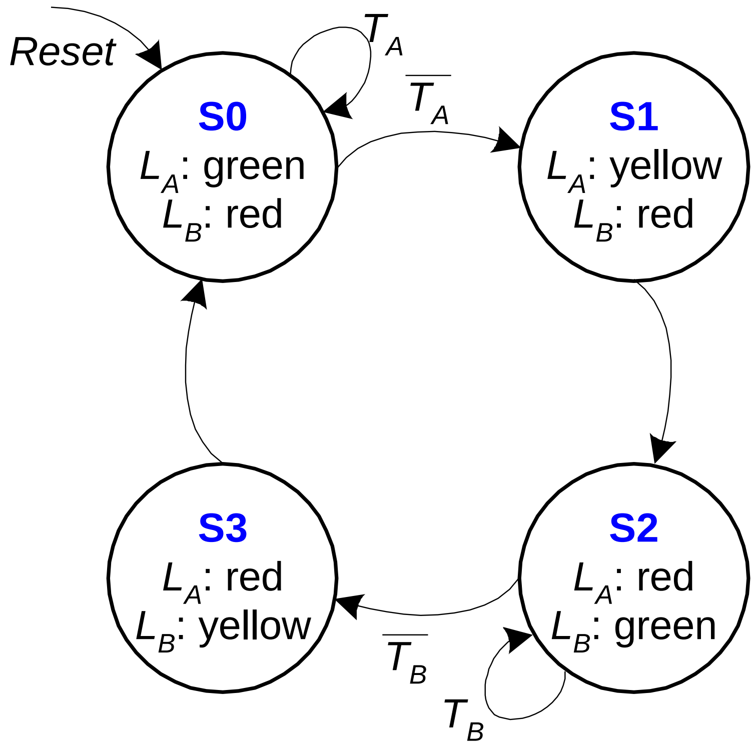
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs





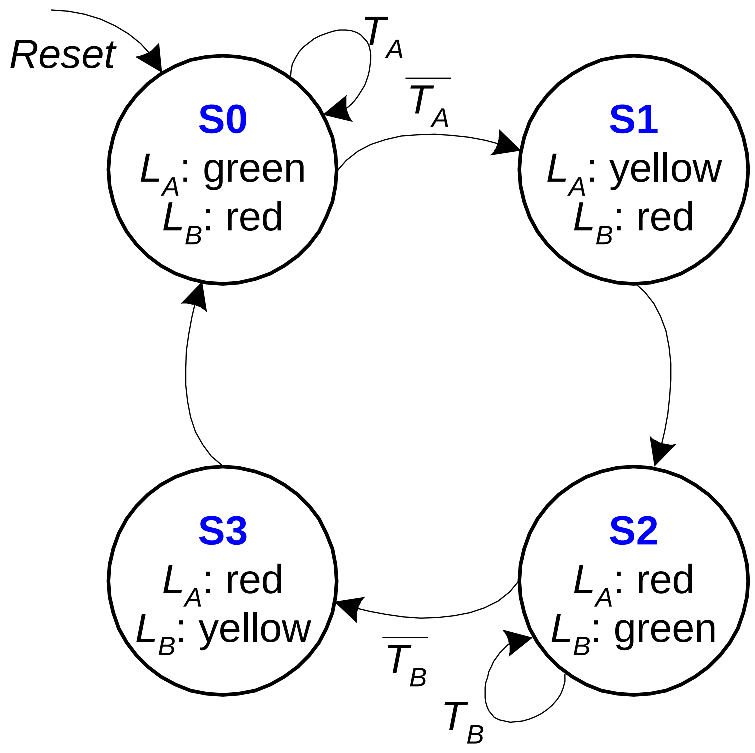
Finite State Machine: State Transition Table

FSM State Transition Table



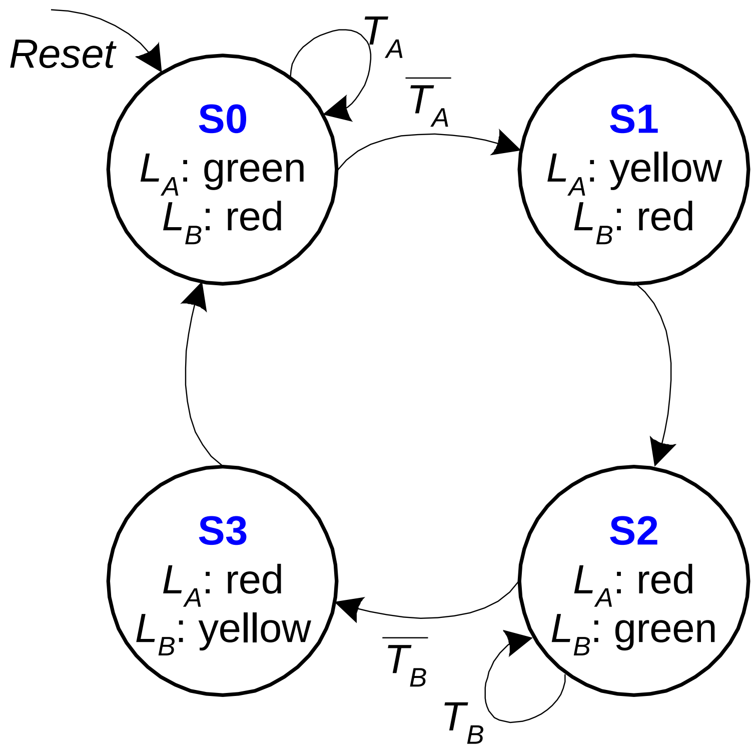
Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	
S0	1	X	
S1	X	X	
S2	X	0	
S2	X	1	
S3	X	X	

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

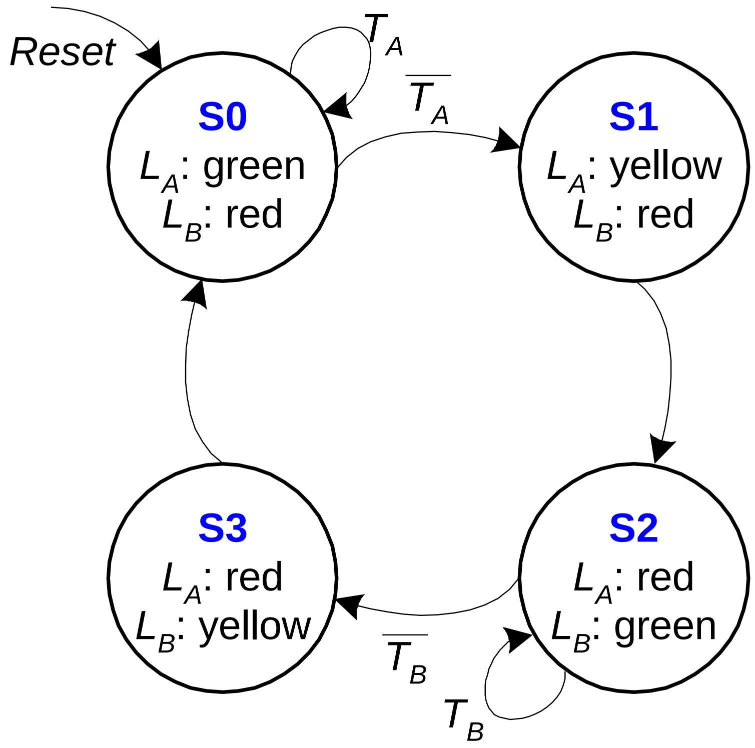
FSM State Transition Table



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

State	Encoding
S0	00
S1	01
S2	10
S3	11

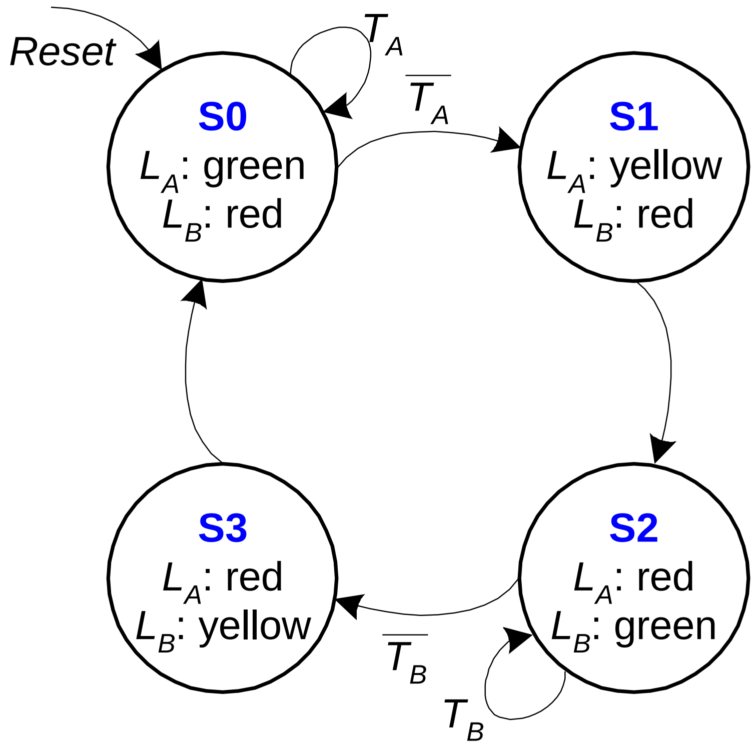
FSM State Transition Table



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 State Transition Table

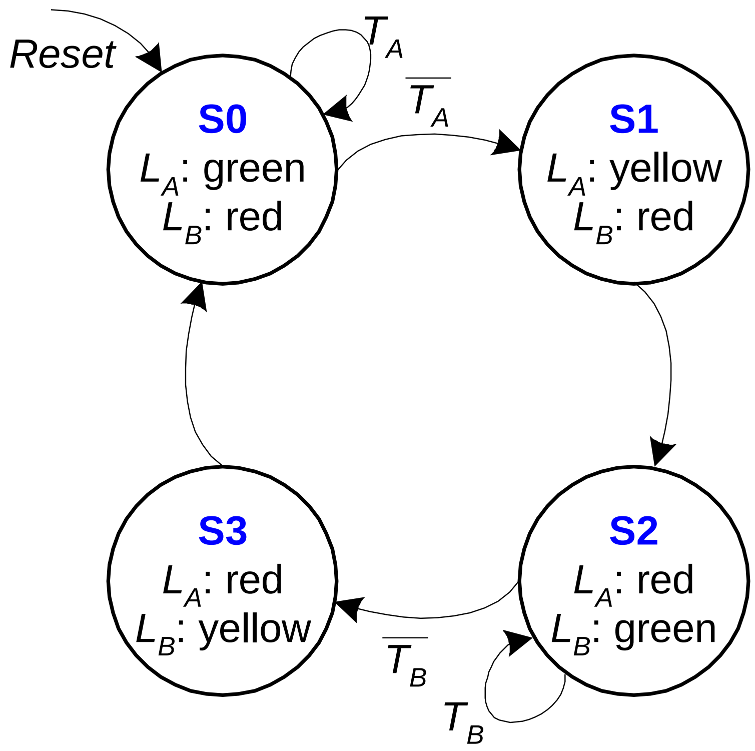


Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$S'_1 = ?$

FSM State Transition Table

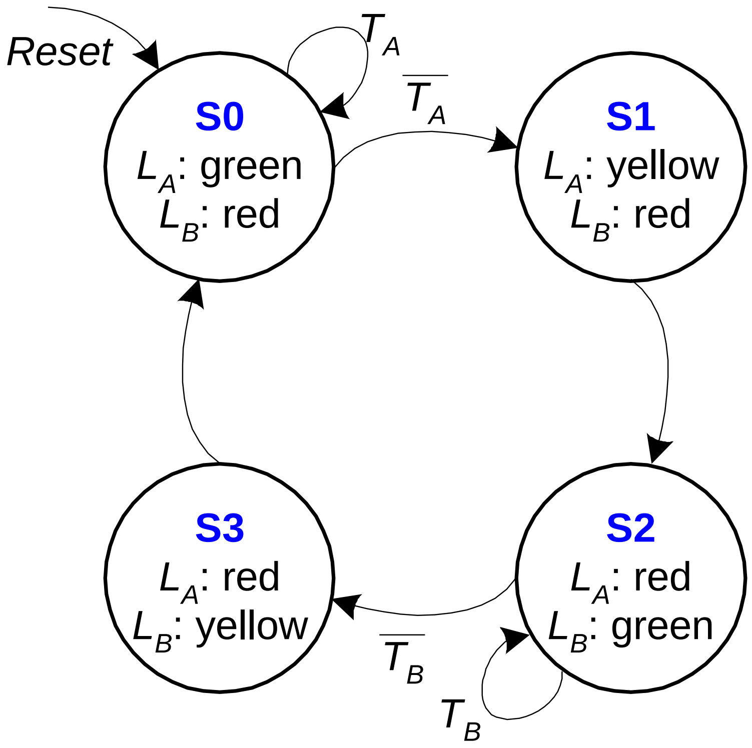


Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S'_1 = (\overline{S_1} \cdot S_0) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B}) + (S_1 \cdot \overline{S_0} \cdot T_B)$$

FSM State Transition Table



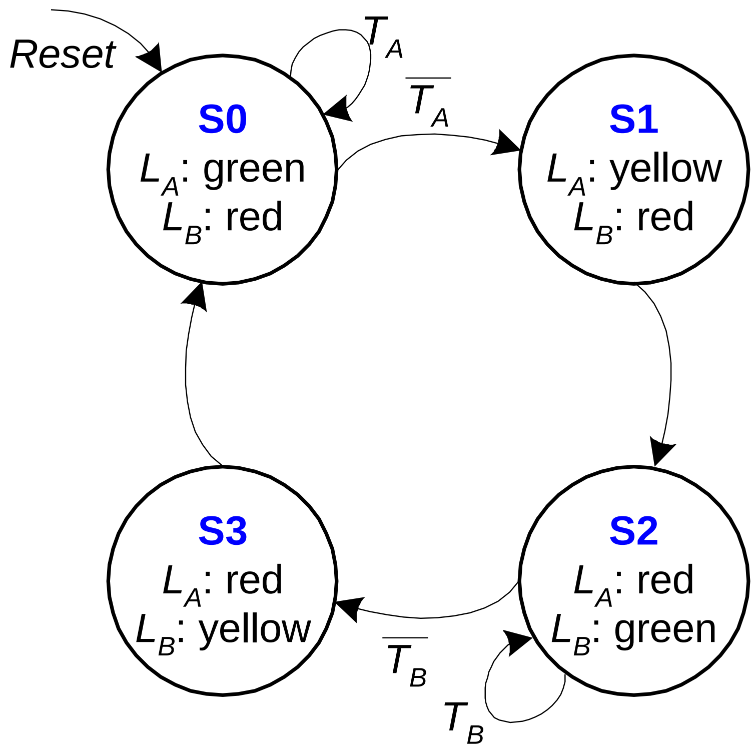
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S'_1 = (\overline{S_1} \cdot S_0) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B}) + (S_1 \cdot \overline{S_0} \cdot T_B)$$

$$S'_0 = ?$$

FSM State Transition Table



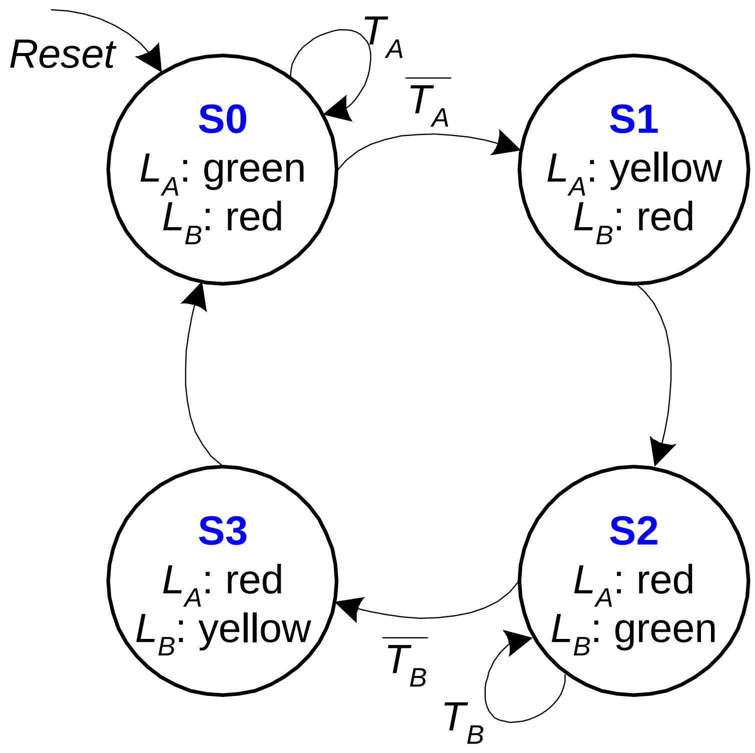
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

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 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

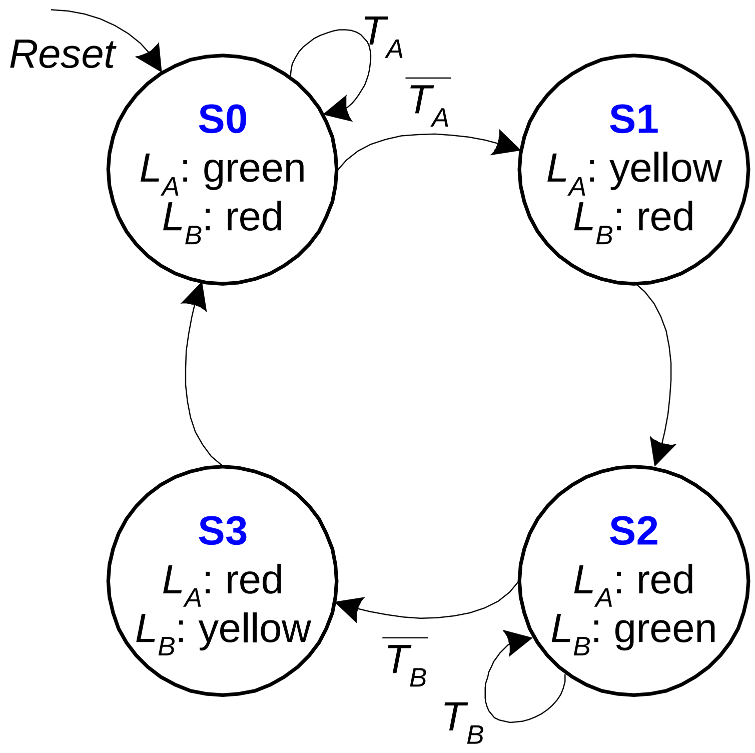
State	Encoding
S0	00
S1	01
S2	10
S3	11

$S'_1 = S_1 \text{ xor } S_0$ **(Simplified)**

$S'_0 = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$

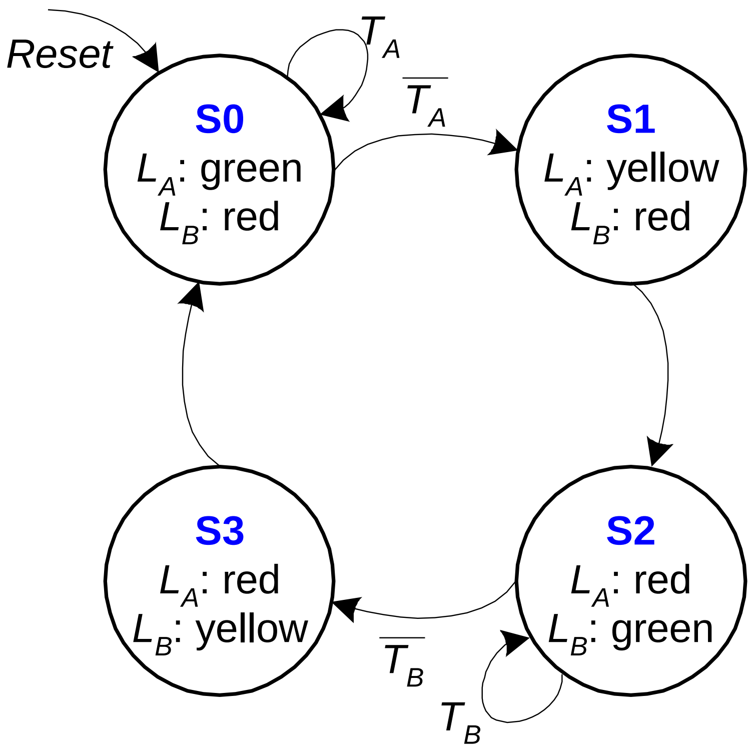
Finite State Machine: Output Table

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

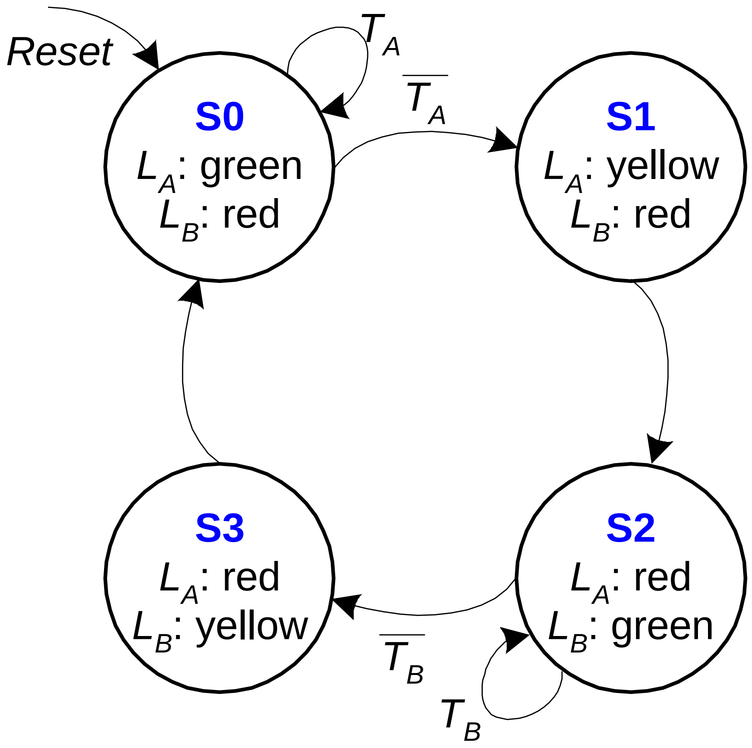
FSM Output Table



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

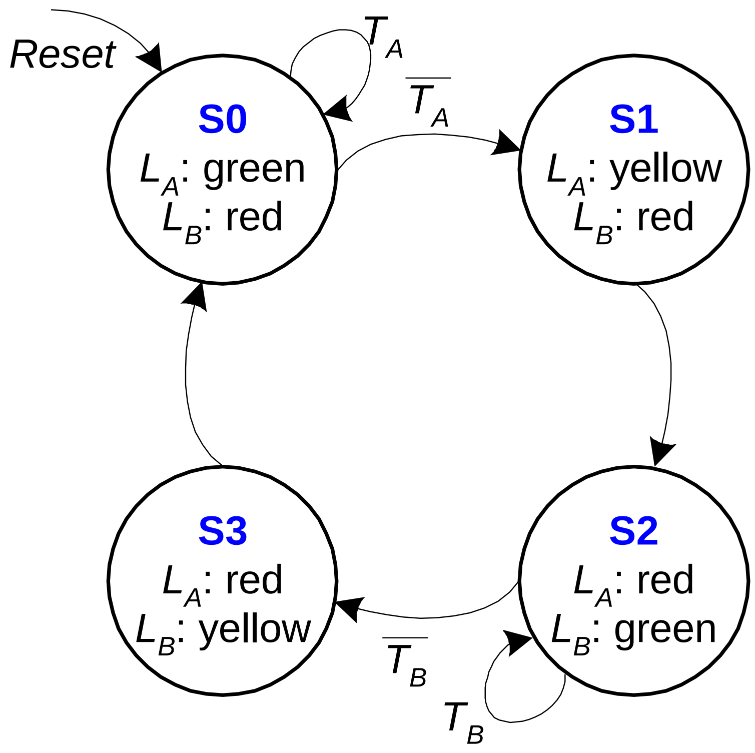


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$L_{A1} = S_1$$

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table

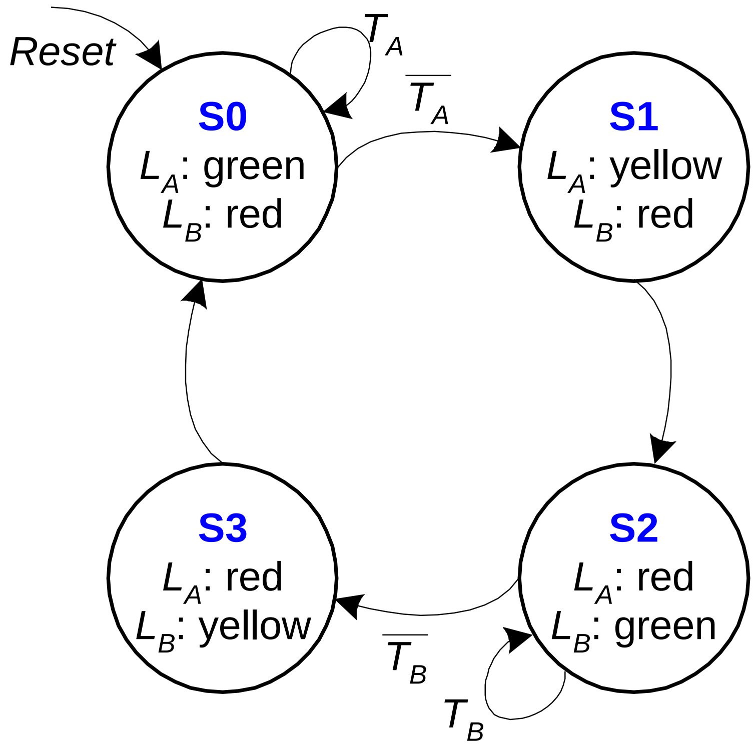


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

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

FSM Output Table

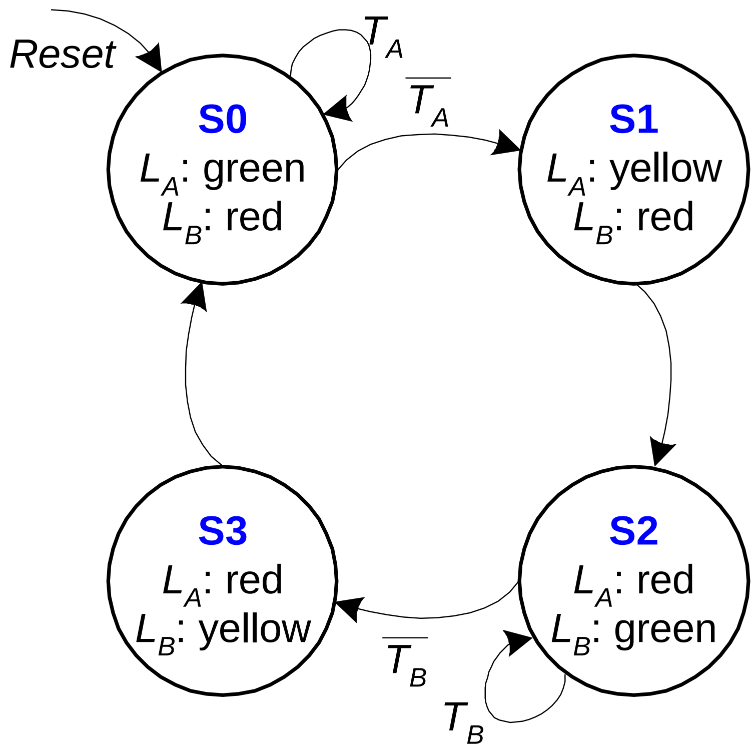


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

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

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table



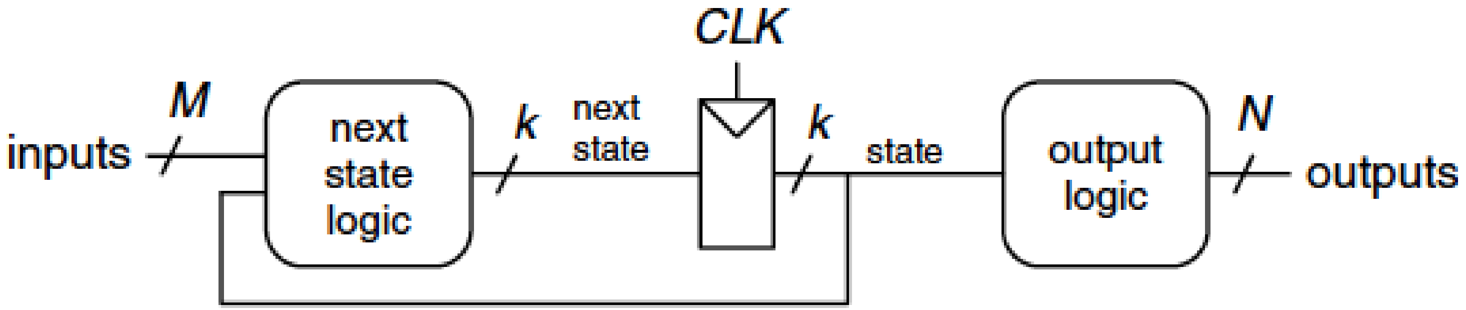
Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

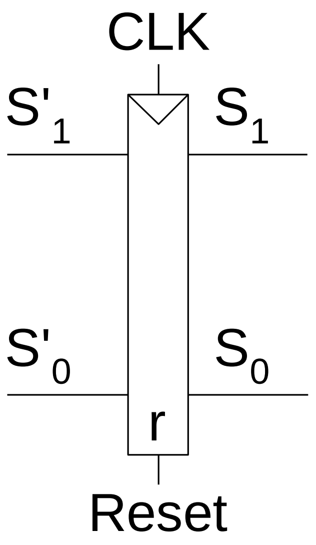
$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} \cdot S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 \cdot S_0$$

Finite State Machine: Schematic

FSM Schematic: State Register

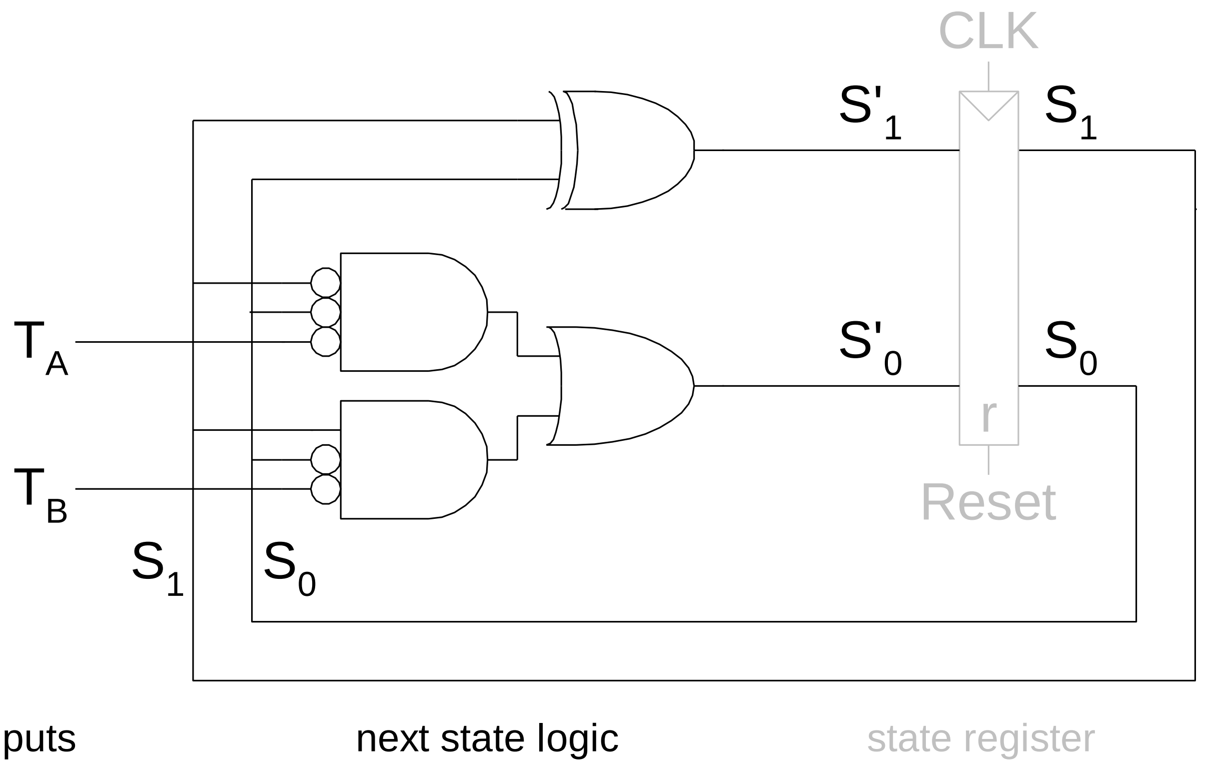


FSM Schematic: State Register



state register

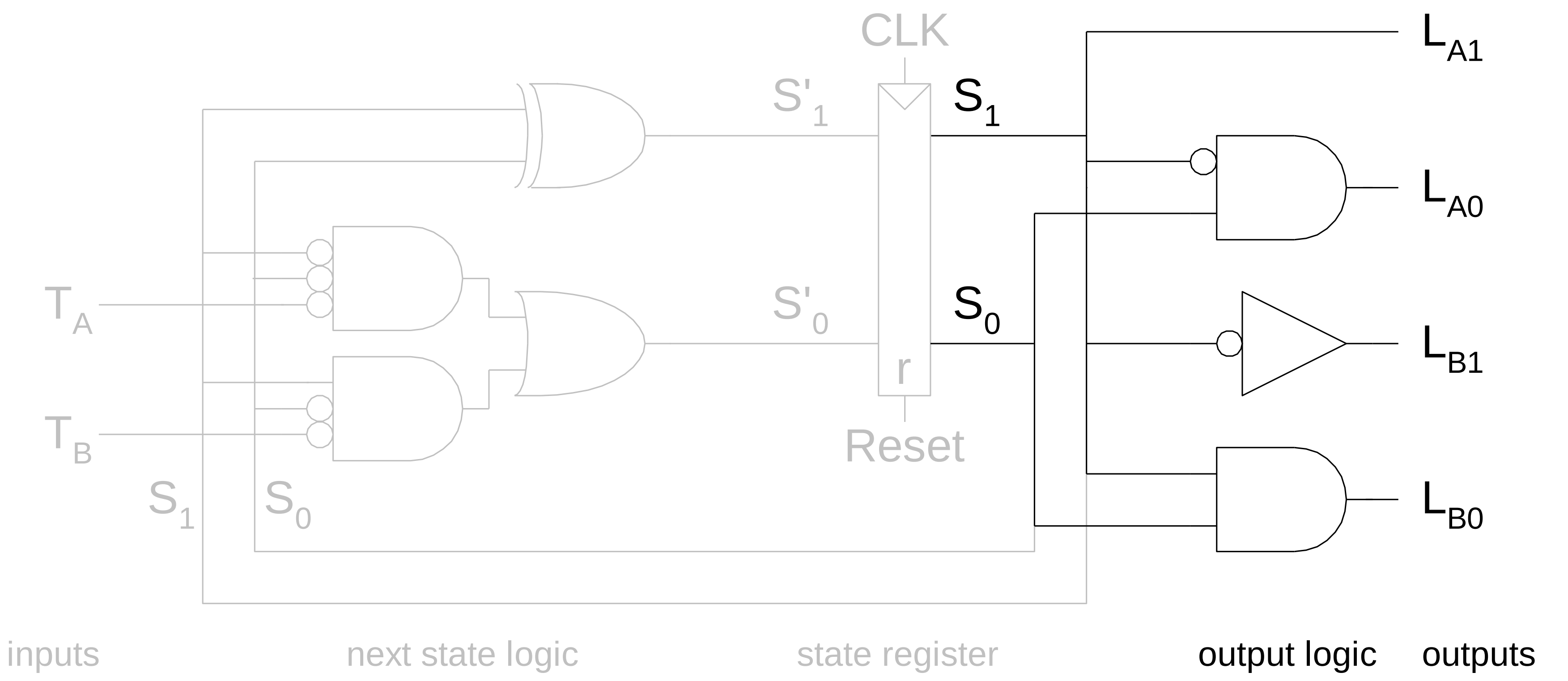
FSM Schematic: Next State Logic



$$S'_1 = S_1 \text{ xor } S_0$$

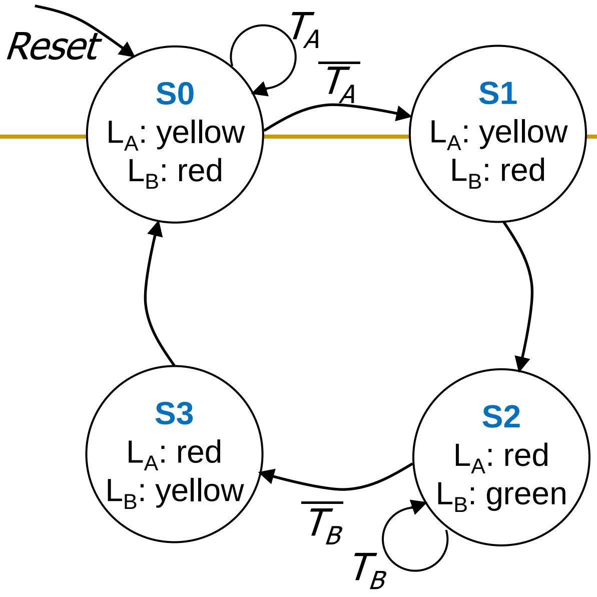
$$S'_0 = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$$

FSM Schematic: Output Logic

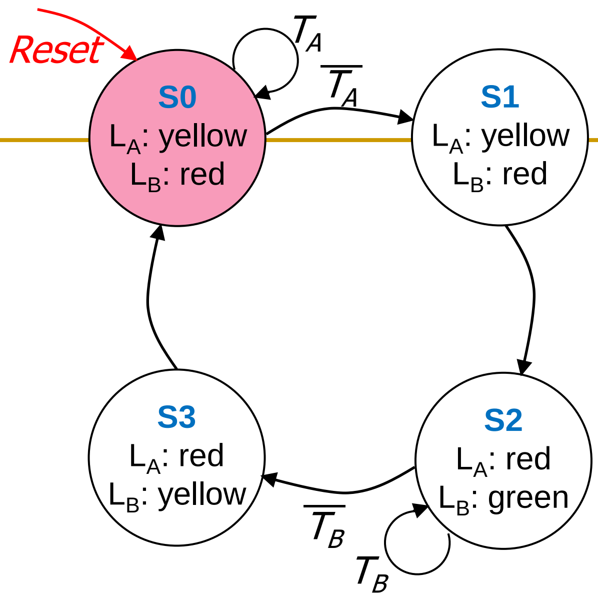


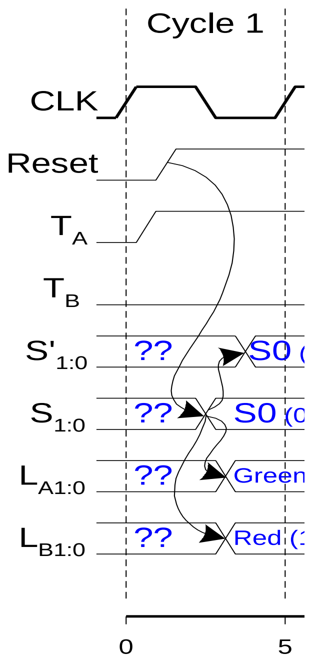
$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} \cdot S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 \cdot S_0$$

FSM Timing Diagram

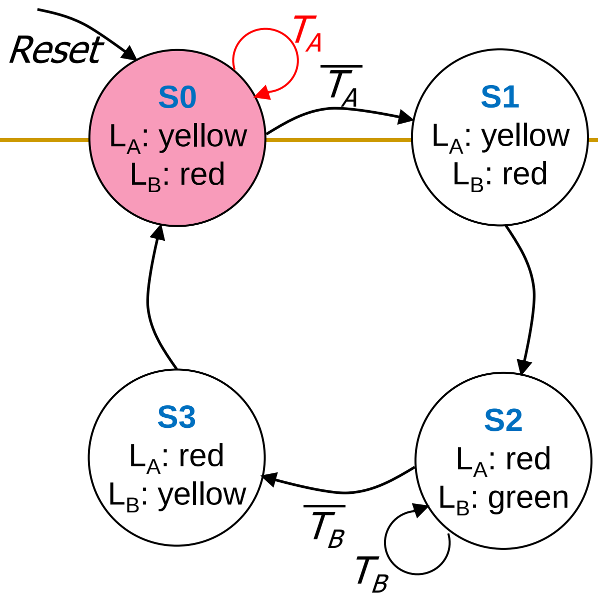


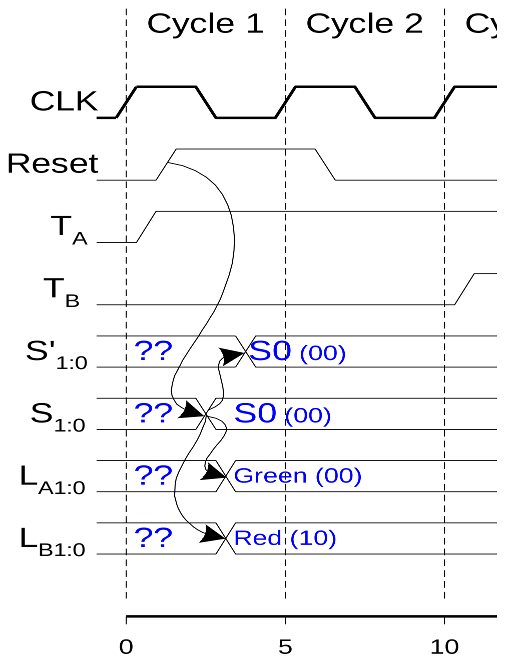
FSM Timing Diagram



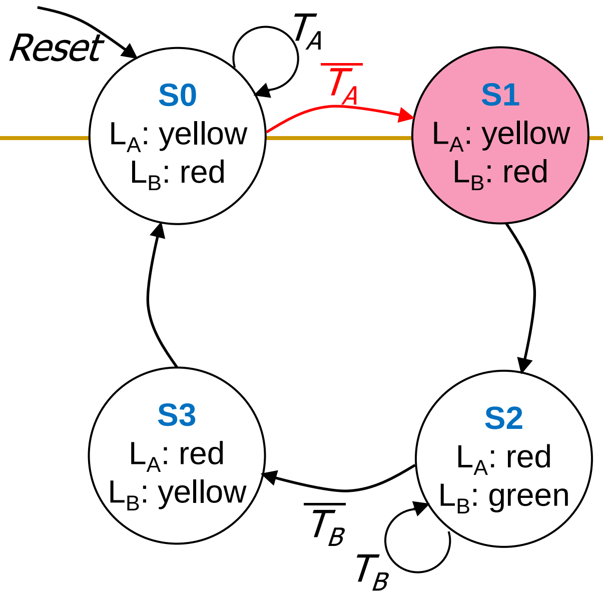


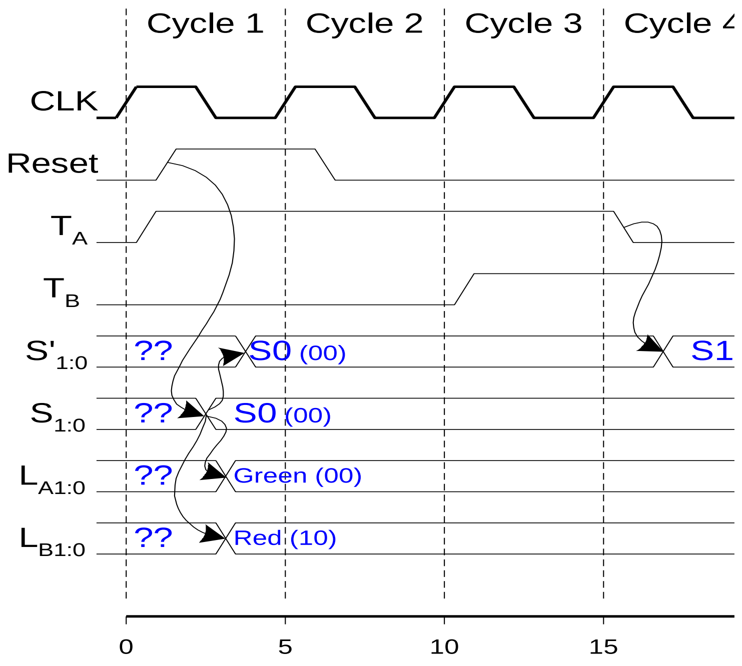
FSM Timing Diagram



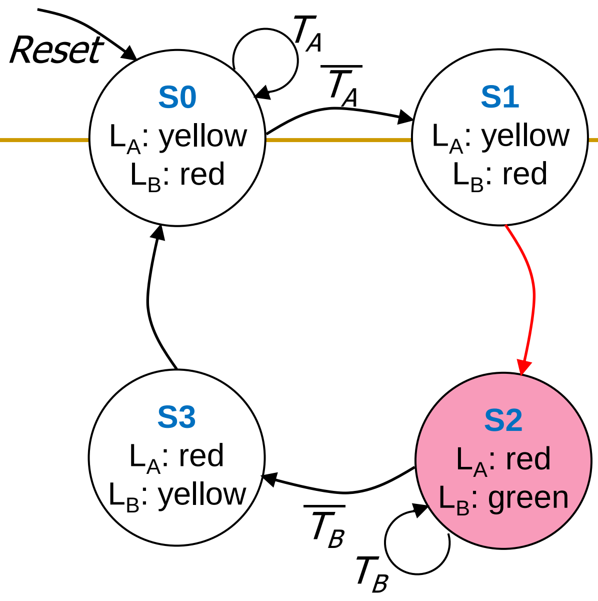


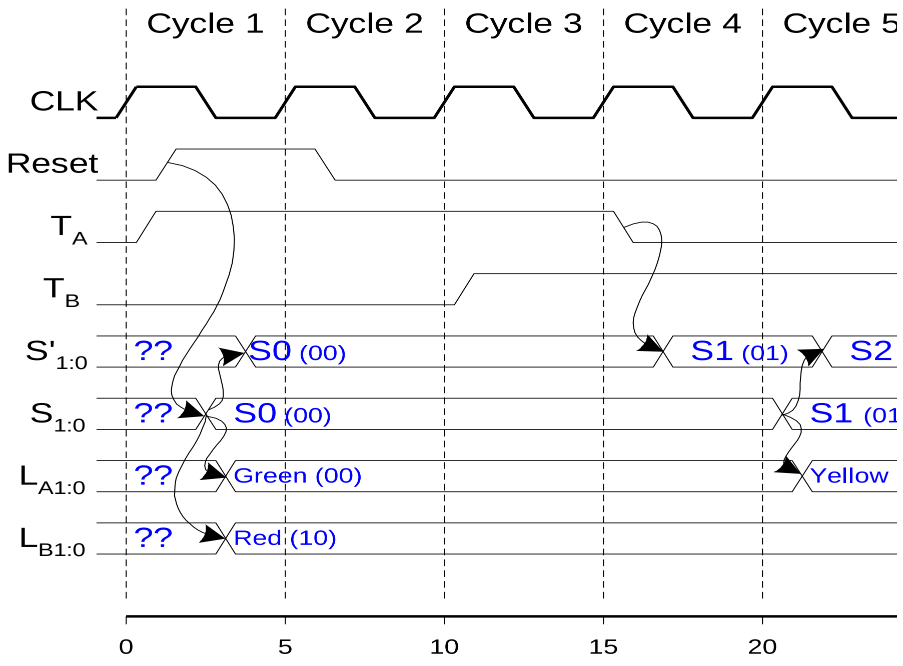
FSM Timing Diagram



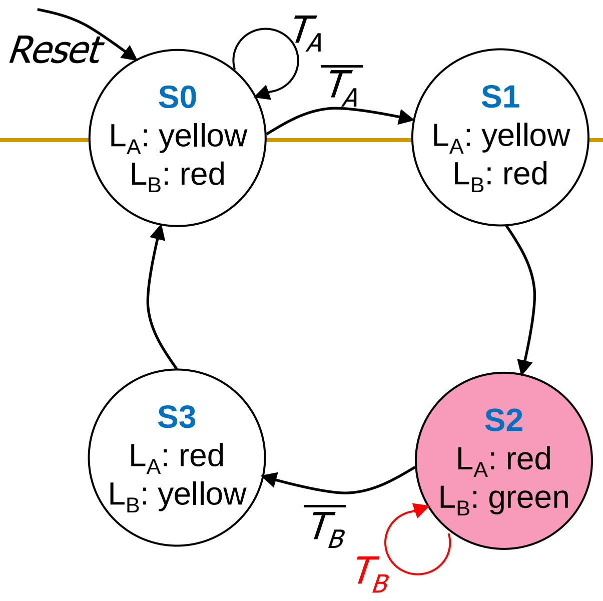


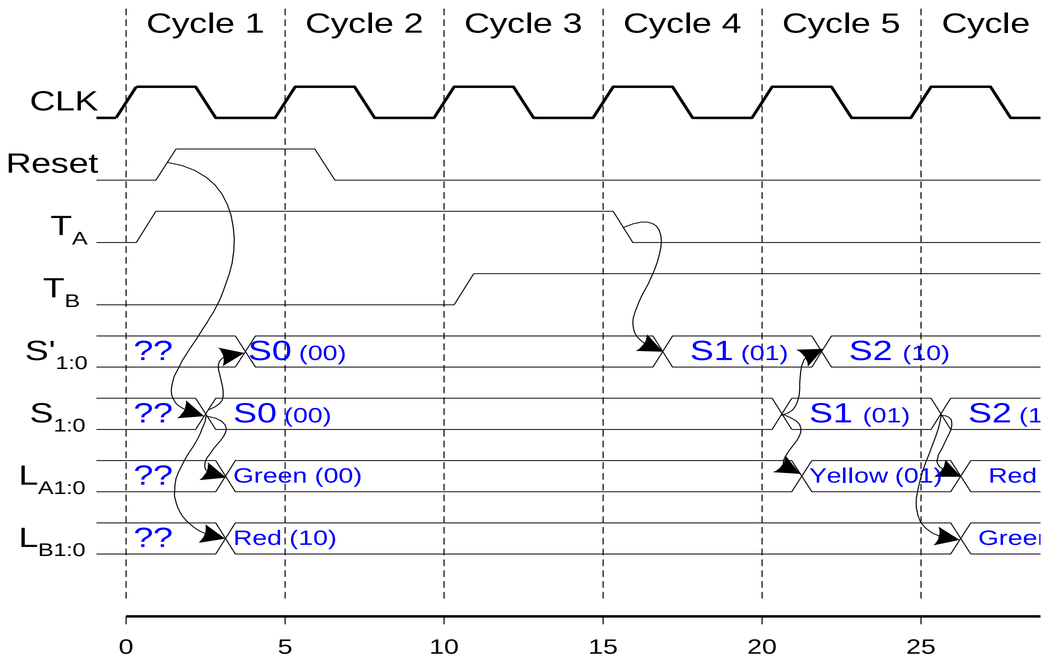
FSM Timing Diagram



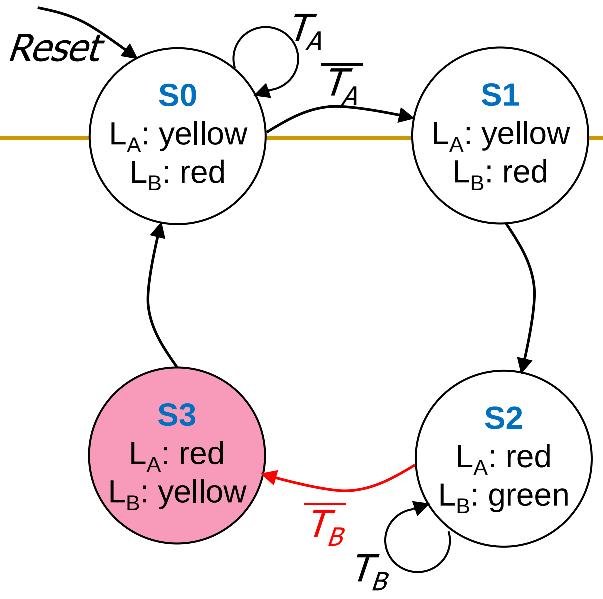


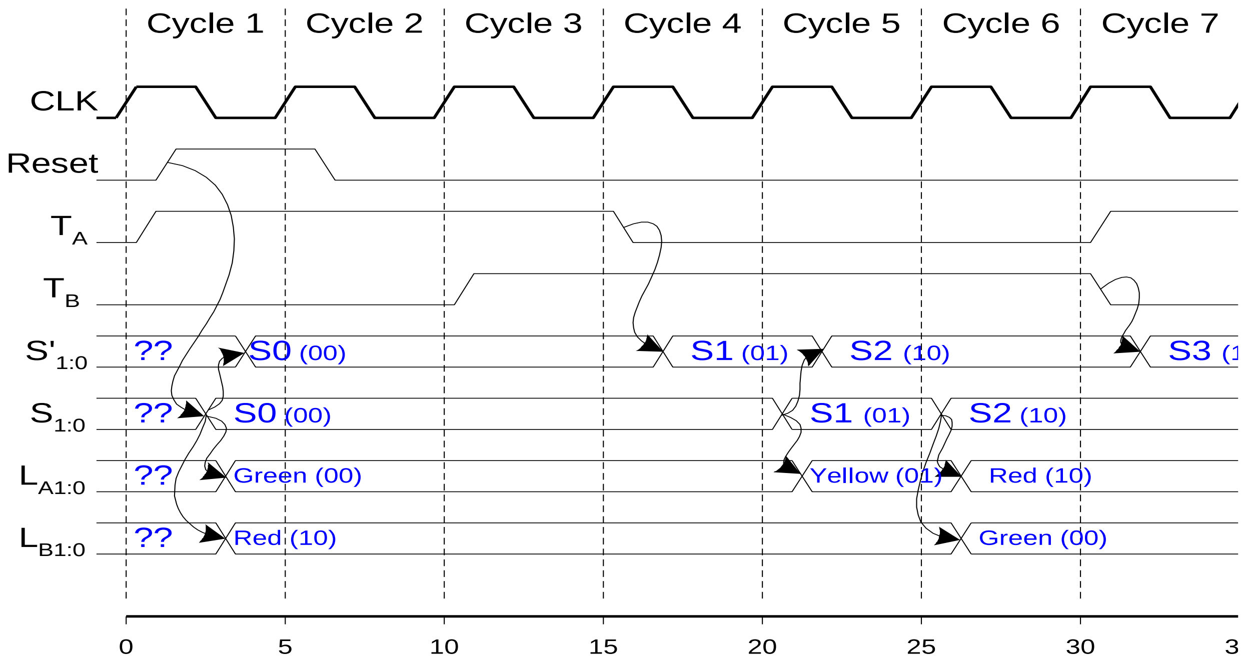
FSM Timing Diagram



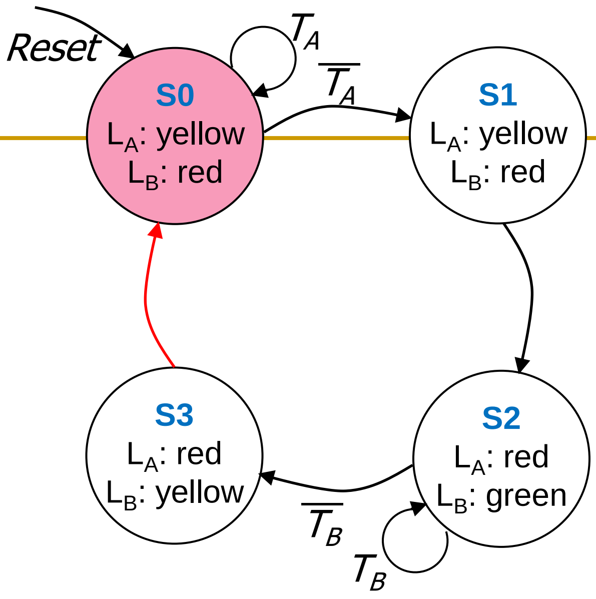


FSM Timing Diagram

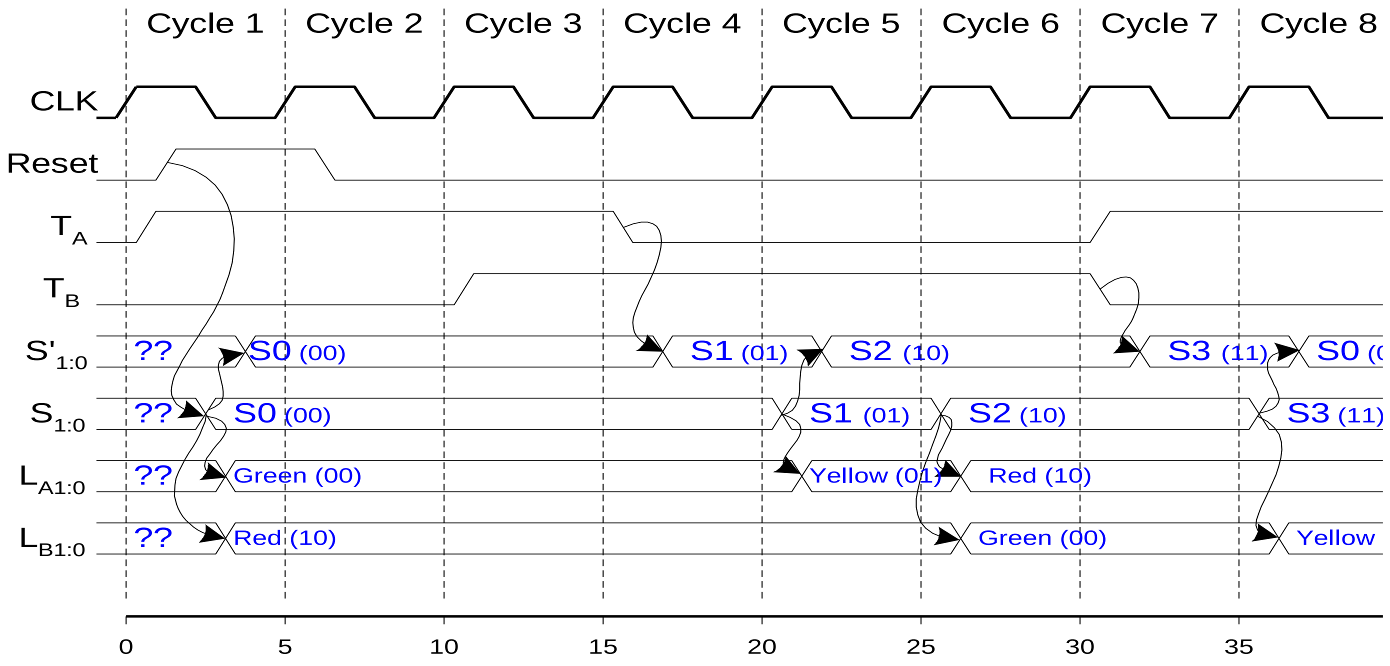




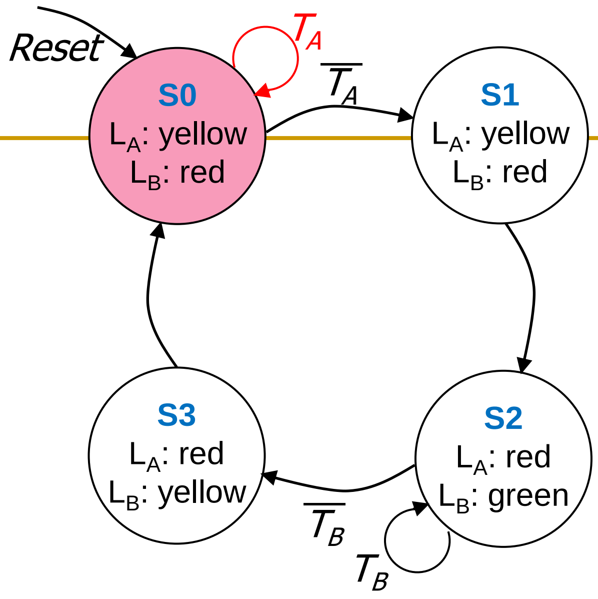
FSM Timing Diagram

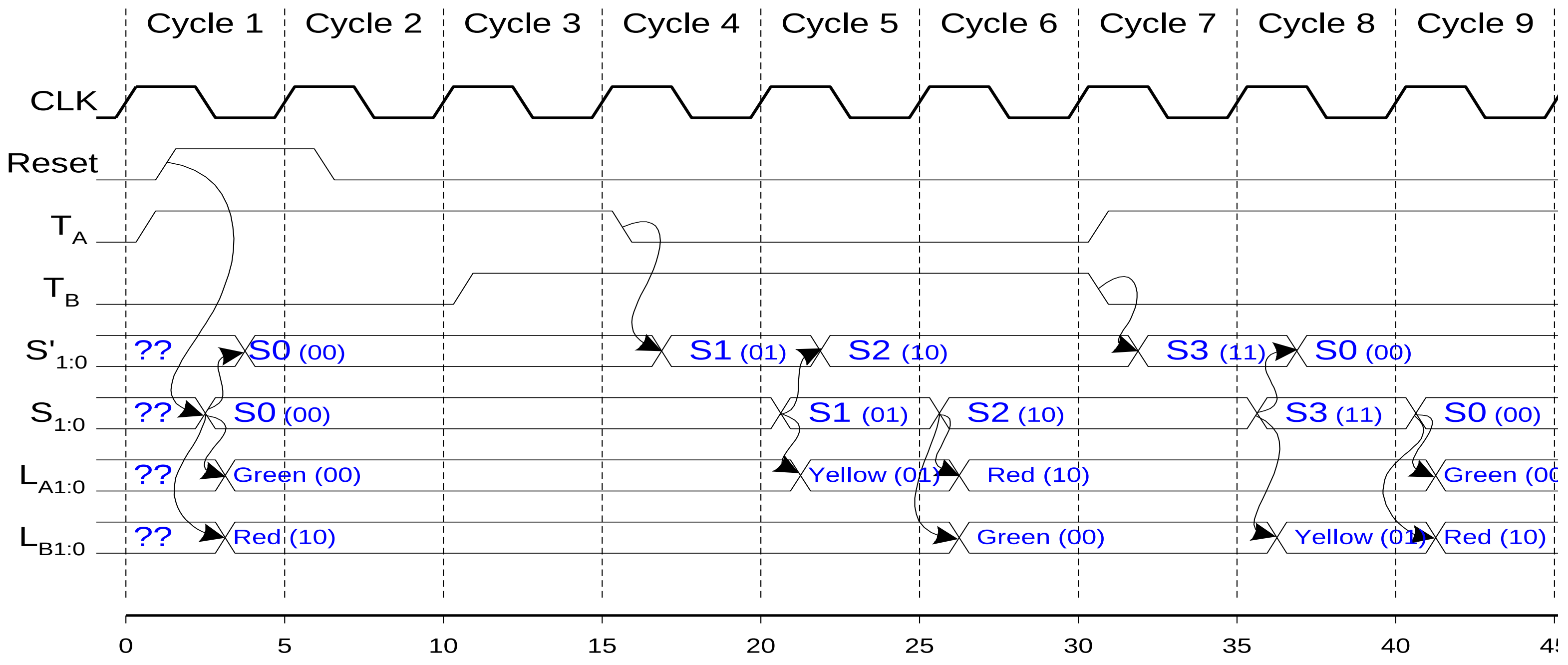


This is from H&H Section 3.4.1

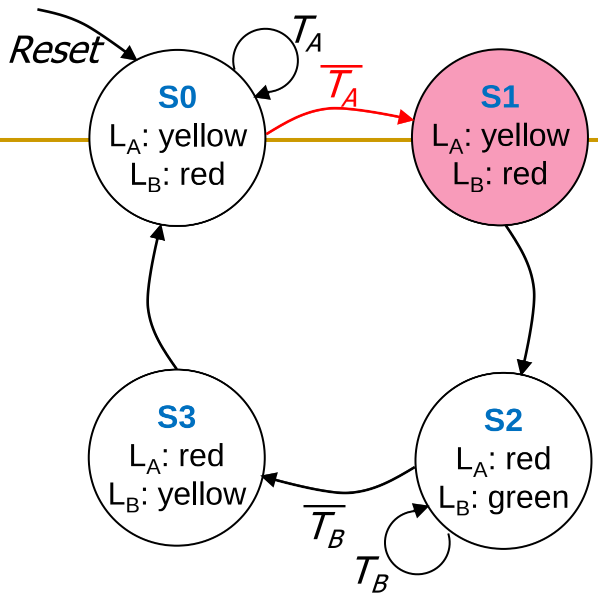


FSM Timing Diagram

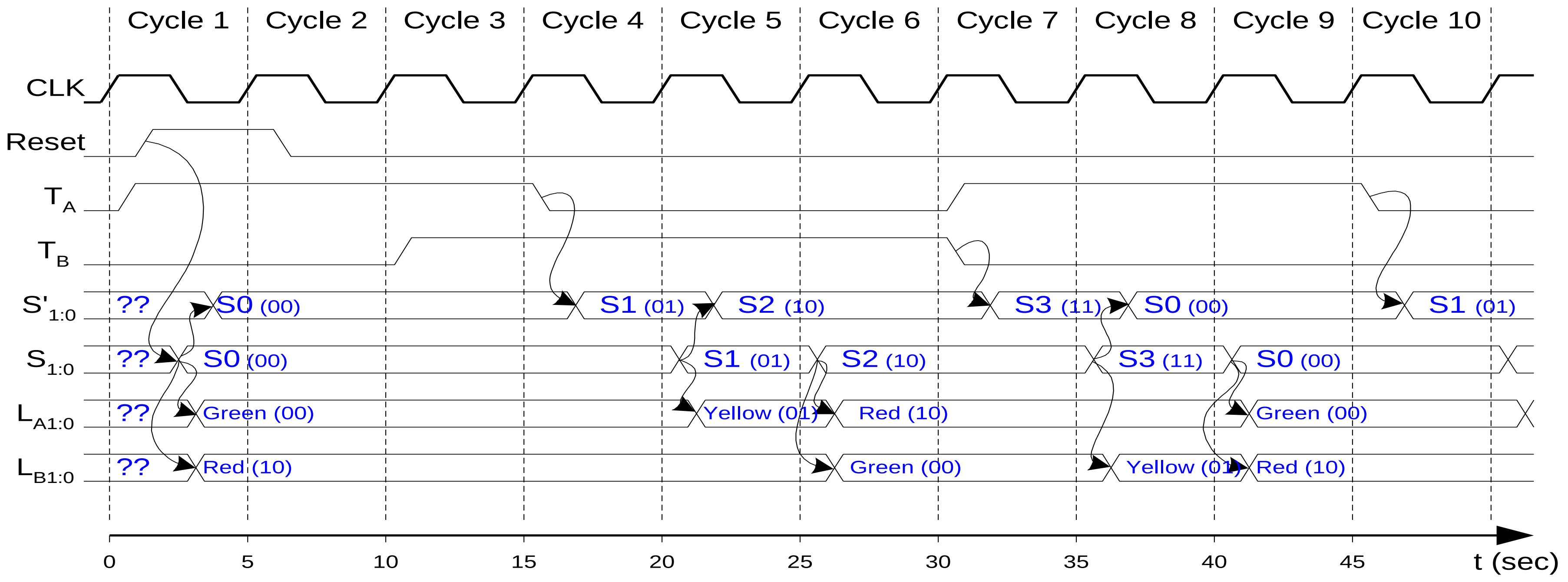




FSM Timing Diagram



See H&H Chapter 3.4

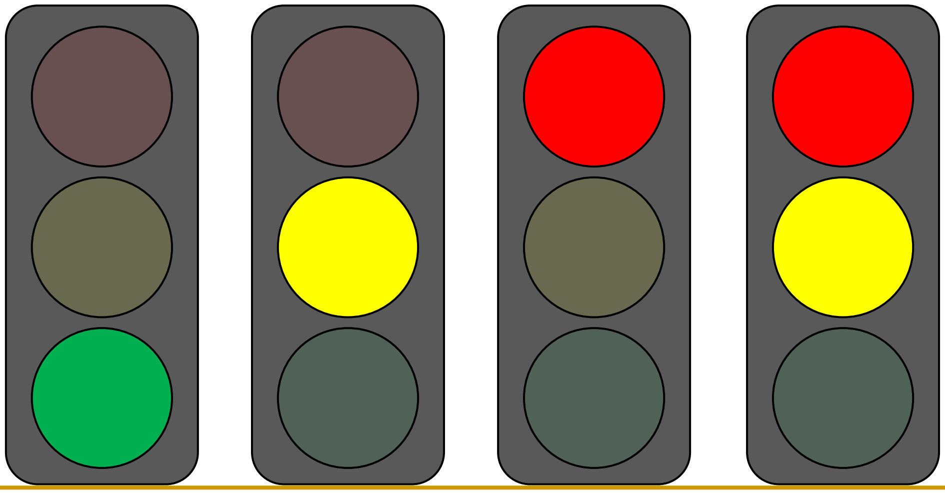


Finite State Machine: State Encoding

FSM State Encoding

- How do we encode the state bits?
  - Three common state binary encodings with different tradeoffs
    1. **Fully Encoded**
    2. **1-Hot Encoded**
    3. **Output Encoded**

- Let's see an example **Swiss** traffic light with 4 states
  - Green, Yellow, Red, Yellow+Red



FSM State Encoding (II)

1. **Binary Encoding (Full Encoding):**
   - Use the minimum possible number of bits
     - Use *log_2(num_states)* bits to represent the states
   - *Example state encodings:* 00, 01, 10, 11
   - **Minimizes** # flip-flops, but not necessarily output logic or next state logic

2. **One-Hot Encoding:**
   - Each bit encodes a different state
     - Uses *num_states* bits to represent the states
     - Exactly 1 bit is "hot" for a given state
   - *Example state encodings:* 0001, 0010, 0100, 1000
   - **Simplest design process** – very automatable
   - **Maximizes** # flip-flops, **minimizes** next state logic

FSM State Encoding (III)

3. **Output Encoding:**
   - Outputs are **directly accessible** in the state encoding
   - For example, since we have **3 outputs** (light color), encode state with **3 bits**, where each bit represents a color
   - *Example states:* 001, 010, 100, 110
     - Bit_0 encodes **green** light output,
     - Bit_1 encodes **yellow** light output
     - Bit_2 encodes **red** light output
   - **Minimizes** output logic
   - Only works for Moore Machines (output function of state)

FSM State Encoding (III)

3. **Output Encoding:**
   - Outputs are **directly accessible** in the state encoding

The **designer** must **carefully** choose an encoding scheme to **optimize** the design under given constraints

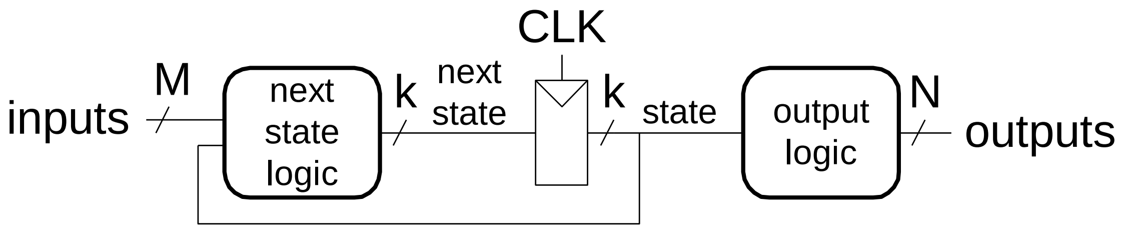
- **Minimizes** output logic
- Only works for Moore Machines (output depends only on state)

Moore vs. Mealy Machines

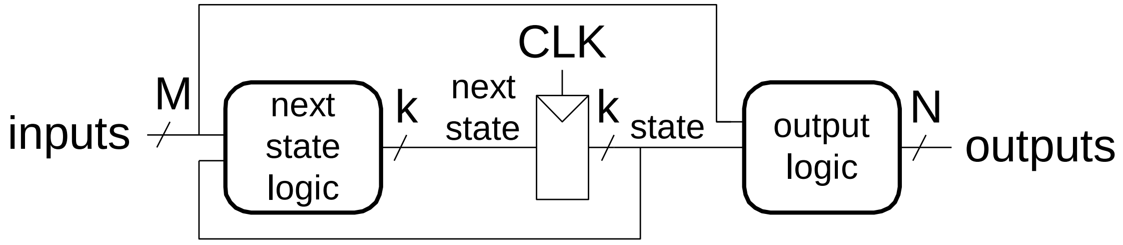
Recall: Moore vs. Mealy FSMs

- Next state is determined by the current state and the inputs
- Two types of FSMs 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

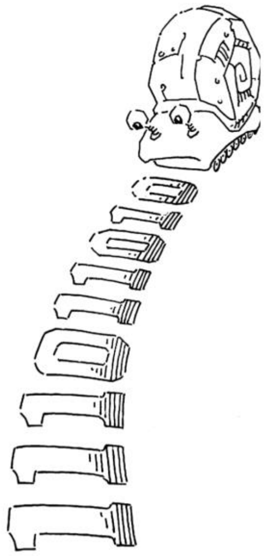


Mealy FSM

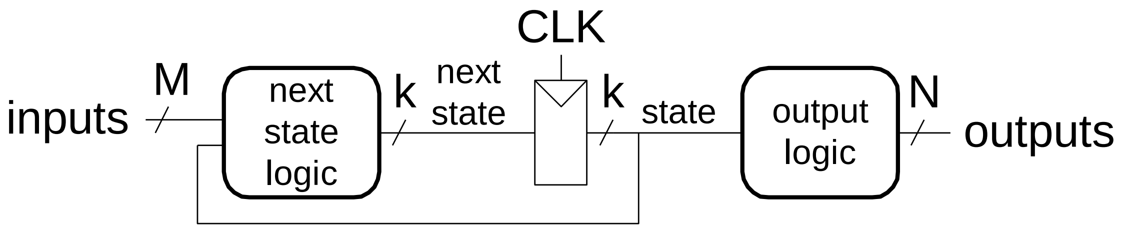


Moore vs. Mealy FSM Examples

- 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.

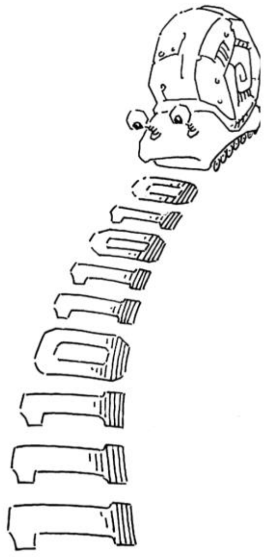


Moore FSM

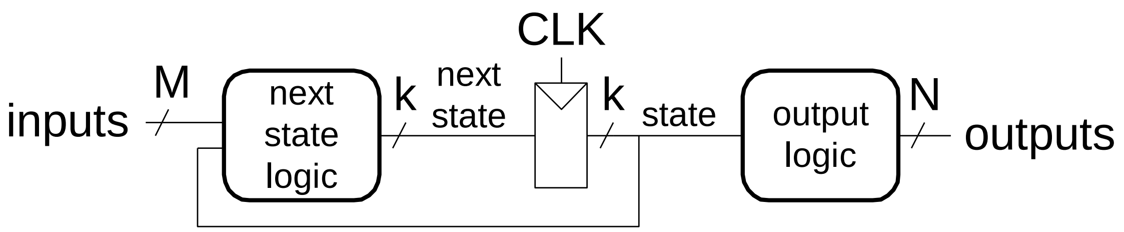


Moore vs. Mealy FSM Examples

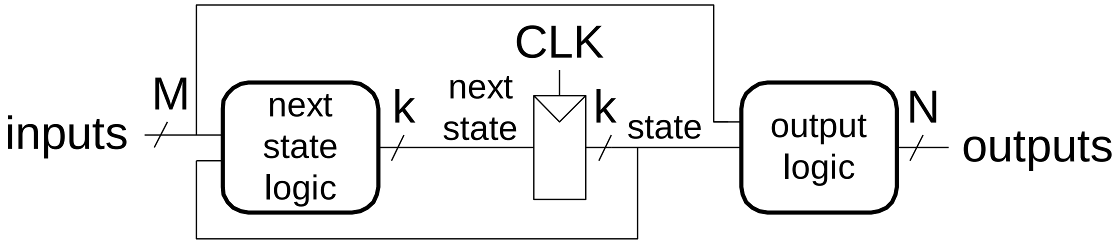
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are 1101.
- Design Moore and Mealy FSMs of the snail's brain.



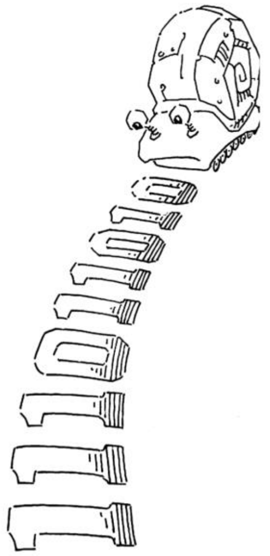
Moore FSM



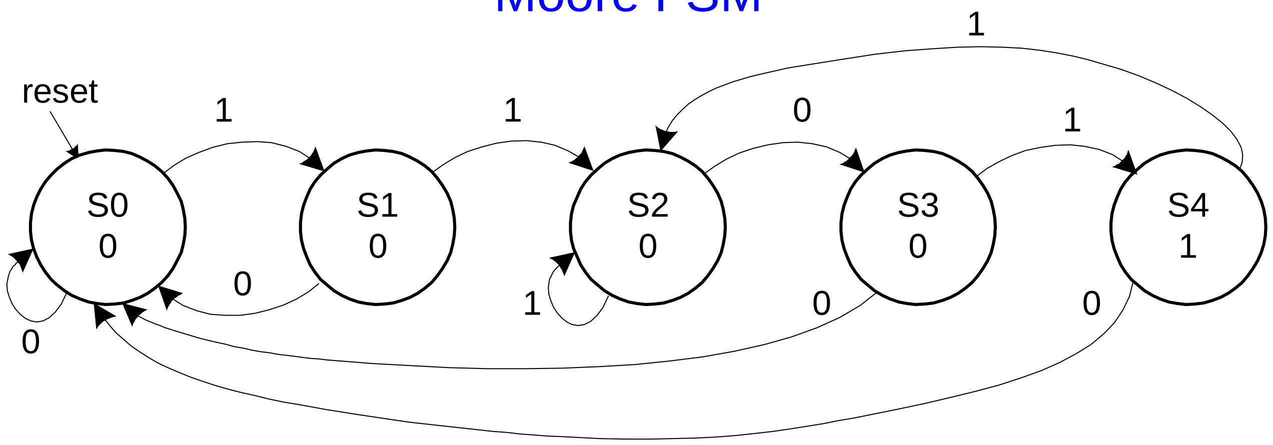
Mealy FSM



State Transition Diagrams

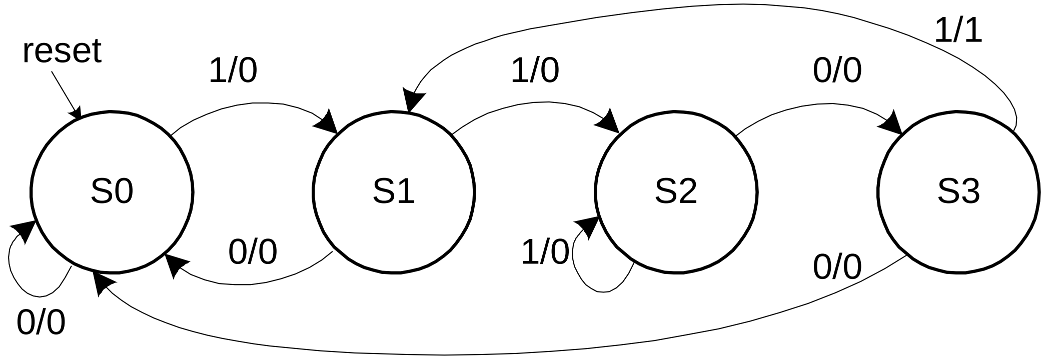


Moore FSM



What are the tradeoffs?

Mealy FSM



FSM Design Procedure

- **Determine** all possible states of your machine

- **Develop** a **state transition diagram**
  - Generally this is done from a textual description
  - You need to 1) determine the **inputs** and **outputs** for each **state** and 2) figure out how to get from one state to another

- **Approach**
  - Start by defining the **reset state** and what happens from it – this is typically an easy point to start from
  - Then continue to add **transitions** and **states**
  - Picking **good state names** is very important
  - Building an FSM is **like** programming (but it *is not* programming!)
    - An FSM has a sequential "control-flow" like a program with conditionals and goto's
    - The if-then-else construct is controlled by one or more inputs
    - The outputs are controlled by the state or the inputs
  - In hardware, we typically have many concurrent FSMs

What is to Come: LC-3 Processor

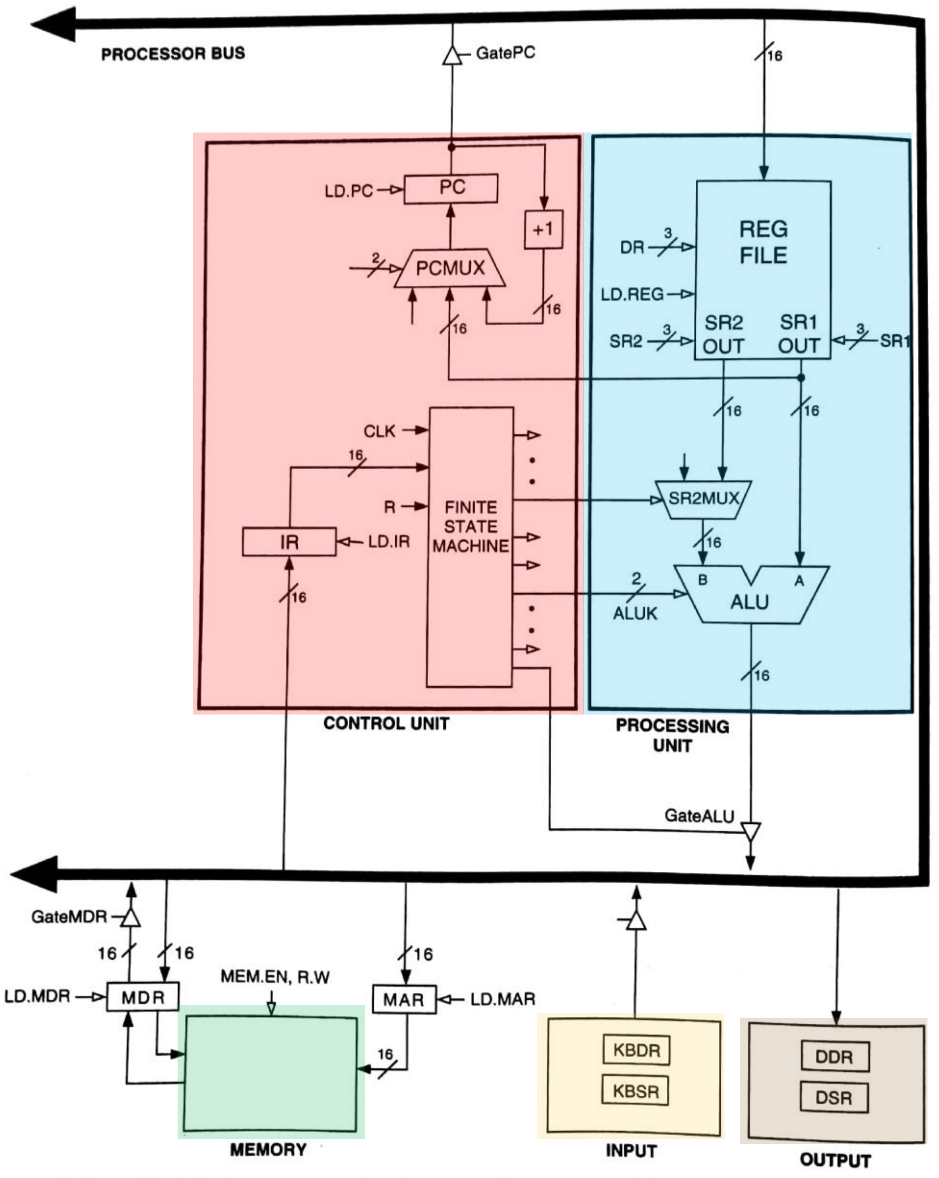
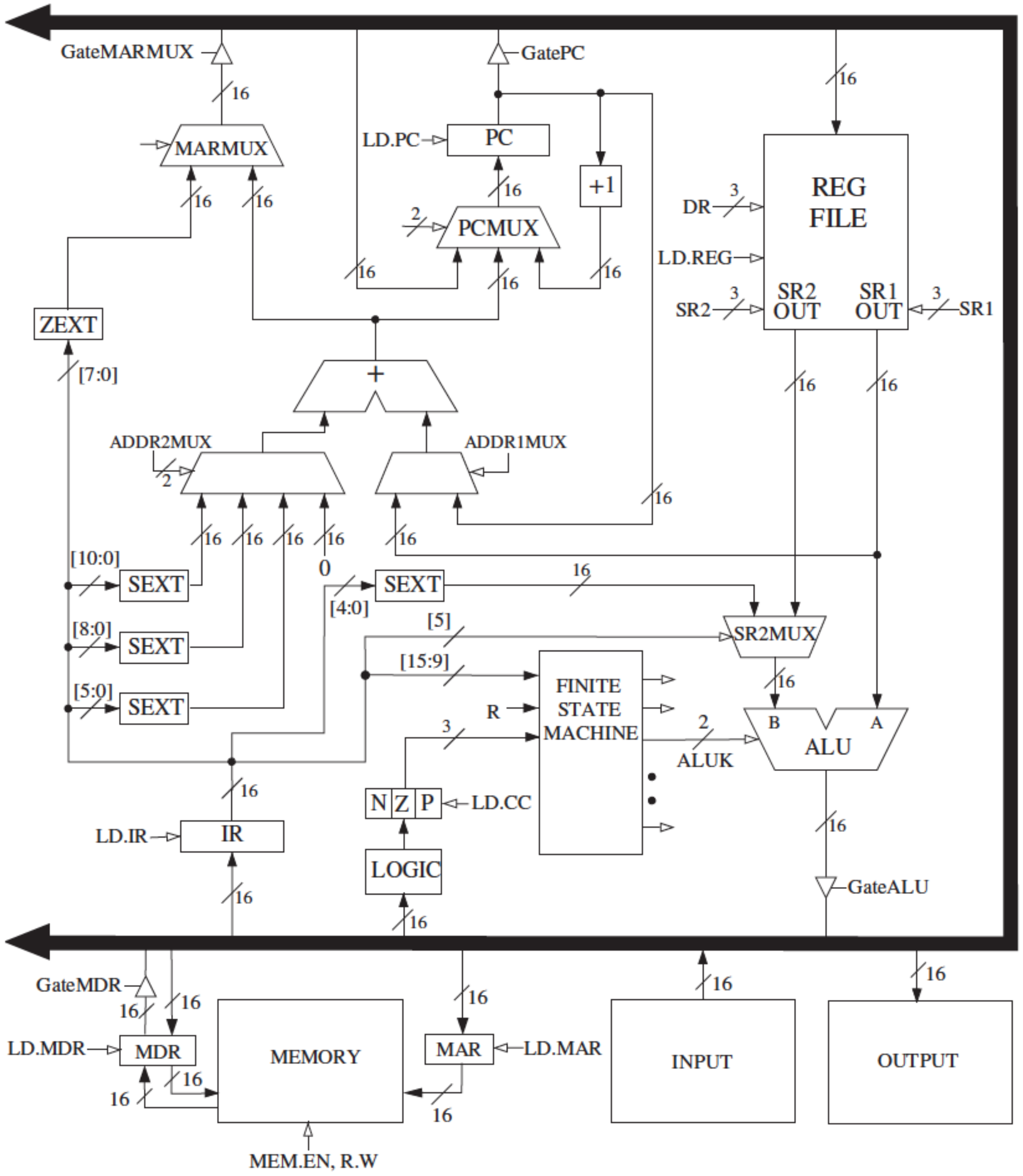


Figure 4.3 The LC-3 as an example of the von Neumann model

What is to Come: LC-3 Datapath



Digital Design & Computer Arch.

Lecture 5a: Sequential Logic Design II Finite State Machines

Prof. Onur Mutlu

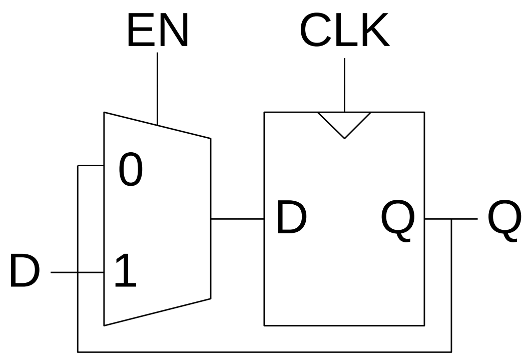
ETH Zürich
Spring 2023
9 March 2023

Backup Slides: Different Flip-Flop Types

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

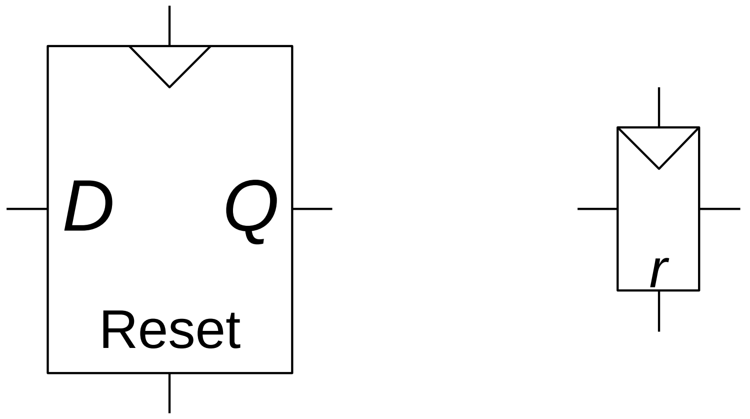
Internal
Circuit



Resettable Flip-Flop

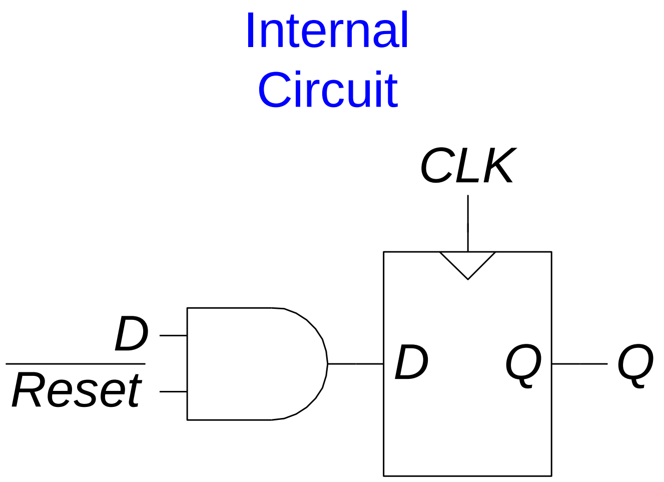
- **Inputs:** CLK, D, Reset
  - The Reset is used to set the output to 0.
- **Function:**
  - ***Reset = 1:*** Q is forced to 0
  - ***Reset = 0:*** the flip-flop behaves like an ordinary D flip-flop

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?



Settable Flip-Flop

- **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