Digital Design & Computer Arch.

Lecture 6: Sequential Logic Design

Prof. Onur Mutlu

ETH Zürich

Spring 2021

12 March 2021

We Are Done with This

- Building blocks of modern computers
  - Transistors
  - Logic gates
- Combinational circuits
- Boolean algebra
- How to use Boolean algebra to represent combinational circuits
- Minimizing logic circuits

Agenda for Today and Next Week

- Today
  - Start (and finish) Sequential Logic
- Next week
  - Hardware Description Languages and Verilog
    - Combinational Logic
    - Sequential Logic
  - Timing and Verification

Assignment: Required Lecture Video

- Why study computer architecture? Why is it important?
- Future Computing Platforms: Challenges & Opportunities

- **Required Assignment**
  - **Watch one of** Prof. Mutlu's lectures and analyze either (or both)
  - https://www.youtube.com/watch?v=kgiZlSOcGFM (May 2017)
  - https://www.youtube.com/watch?v=mskTeNnf-i0 (Feb 2021)

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page summary** of one of the lectures and email us
    - What are your key takeaways?
    - What did you learn?
    - What did you like or dislike?
    - Submit your summary to Moodle – Deadline: April 5

Extra Assignment: Moore's Law (I)

- **Paper review**
- G.E. Moore. "Cramming more components onto integrated circuits," Electronics magazine, 1965

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page review**
  - Upload PDF file to Moodle – Deadline: April 5

- I strongly recommend that you follow my guidelines for (paper) review (see next slide)

Extra Assignment 2: Moore's Law (II)

- **Guidelines on how to review papers critically**
  - Guideline slides: pdf ppt
  - Video: https://www.youtube.com/watch?v=tOL6FANAJ8c
  - Example reviews on "Main Memory Scaling: Challenges and Solution Directions" (link to the paper)
    - Review 1
    - Review 2
  - Example review on "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems" (link to the paper)
    - Review 1

Assignment: Required Readings

- Combinational Logic
  - P&P Chapter 3 until 3.3 + H&H Chapter 2
- Sequential Logic
  - P&P Chapter 3.4 until end + H&H Chapter 3 in full
- Hardware Description Languages and Verilog
  - H&H Chapter 4 in full
- Timing and Verification
  - H&H Chapters 2.9 and 3.5 + (start Chapter 5)

- By the end of next week, make sure you are done with
  - **P&P Chapters 1-3 + H&H Chapters 1-4**

Required Readings (for Next Week)

- Hardware Description Languages and Verilog
  - H&H Chapter 4 in full

- Timing and Verification
  - H&H Chapters 2.9 and 3.5 + (start Chapter 5)

- By tomorrow, make sure you are done with
  - **P&P Chapters 1-3 + H&H Chapters 1-4**

Required Readings (for Next Next Week)

- Von Neumann Model, LC-3, and MIPS
  - P&P, Chapters 4, 5
  - H&H, Chapter 6
  - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
  - H&H, Appendix B (MIPS instructions)

- Programming
  - P&P, Chapter 6

- **Recommended:** Digital Building Blocks
  - H&H, Chapter 5

Sequential Logic Circuits and Design

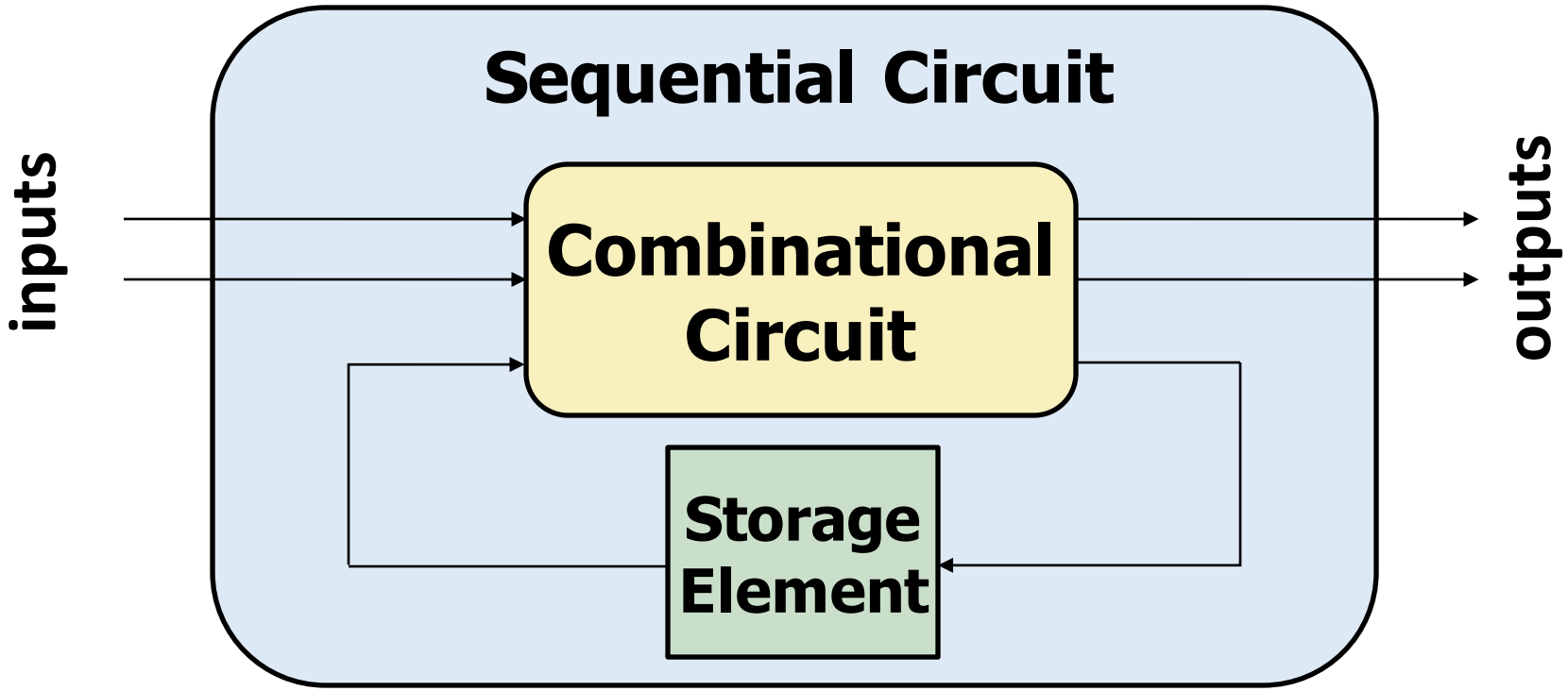
What We Will Learn Today

- Circuits that can store information
  - Cross-coupled inverter
  - R-S Latch
  - Gated D Latch
  - D Flip-Flop
  - Register
- Finite State Machines (FSM)
  - Moore Machine
  - Mealy Machine
- Verilog implementations of sequential circuits (next week)

Circuits that Can Store Information

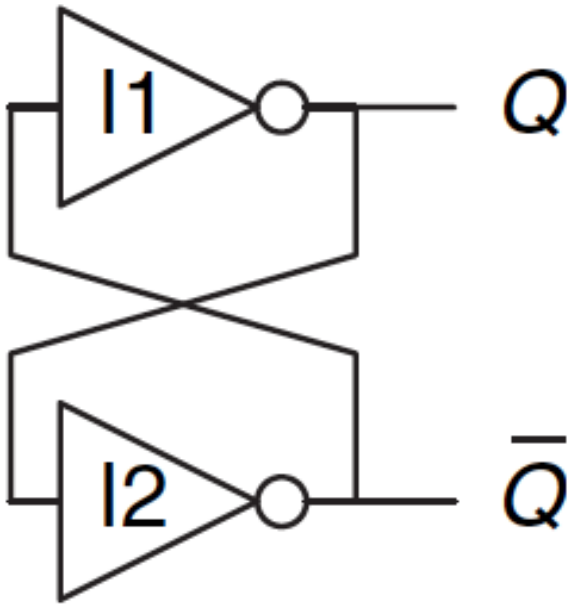
Introduction

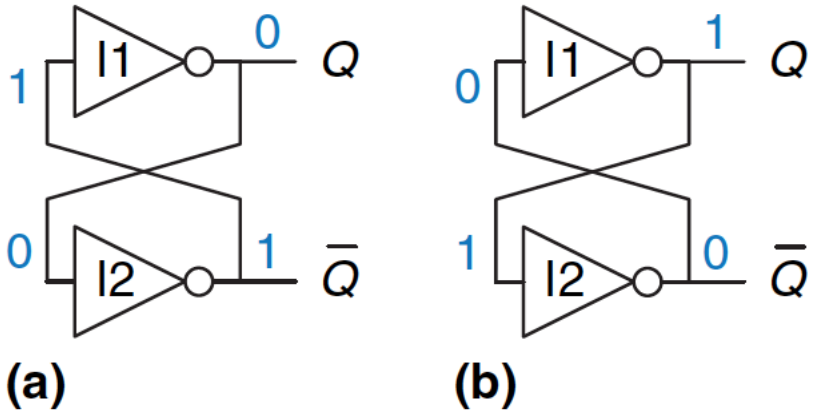
- Combinational circuit output depends **only** on current input
- We want circuits that produce output depending on **current** and **past** input values – circuits with **memory**
- How can we design a circuit that **stores information**?



Capturing Data

Basic Element: Cross-Coupled Inverters



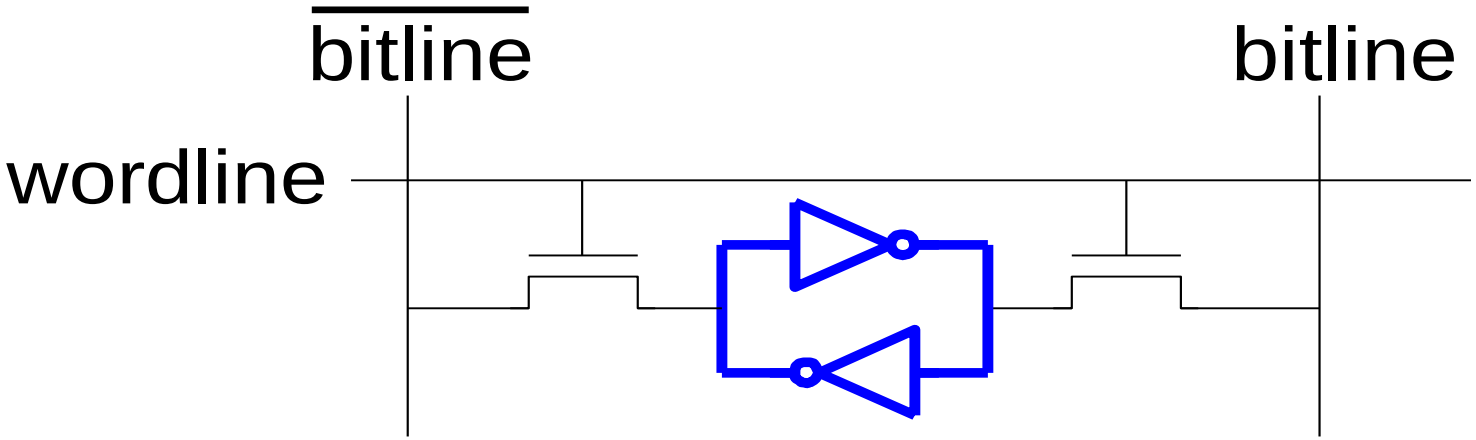


- Has two stable states: Q=1 or Q=0.
- Has a third possible "metastable" state with both outputs oscillating between 0 and 1 (we will see this later)
- Not useful without a *control mechanism* for setting Q

Image source: Harris and Harris, Digital Design and Computer Architecture, 2nd Ed., p.110.

More Realistic Storage Elements

- **Have a control mechanism for setting Q**
  - We will see the R-S latch soon
  - Let's look at an SRAM (static random access memory) cell first



SRAM cell

- We will get back to SRAM (and DRAM) later

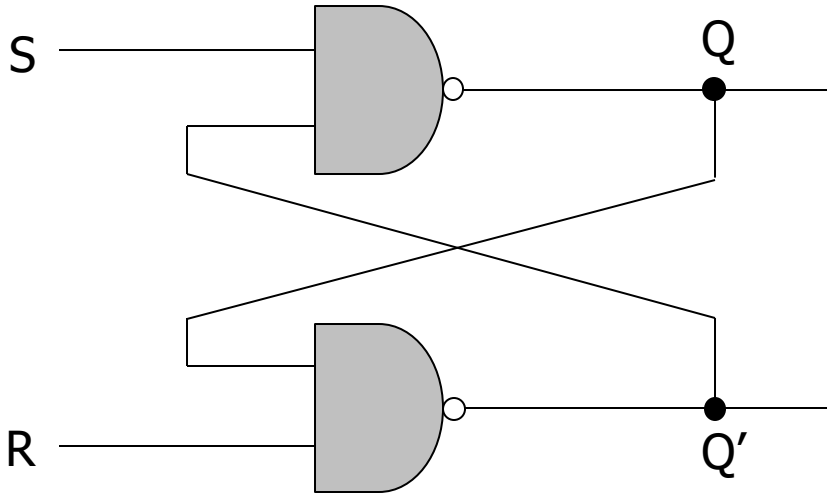
The Big Picture: Storage Elements

- Latches and Flip-Flops
  - Very fast, parallel access
  - Very expensive (one bit costs tens of transistors)
- Static RAM (SRAM)
  - Relatively fast, only one data word at a time
  - Expensive (one bit costs 6+ transistors)
- Dynamic RAM (DRAM)
  - Slower, one data word at a time, reading destroys content (refresh), needs special process for manufacturing
  - Cheap (one bit costs only one transistor plus one capacitor)
- Other storage technology (flash memory, hard disk, tape)
  - Much slower, access takes a long time, non-volatile
  - Very cheap

Basic Storage Element: The R-S Latch

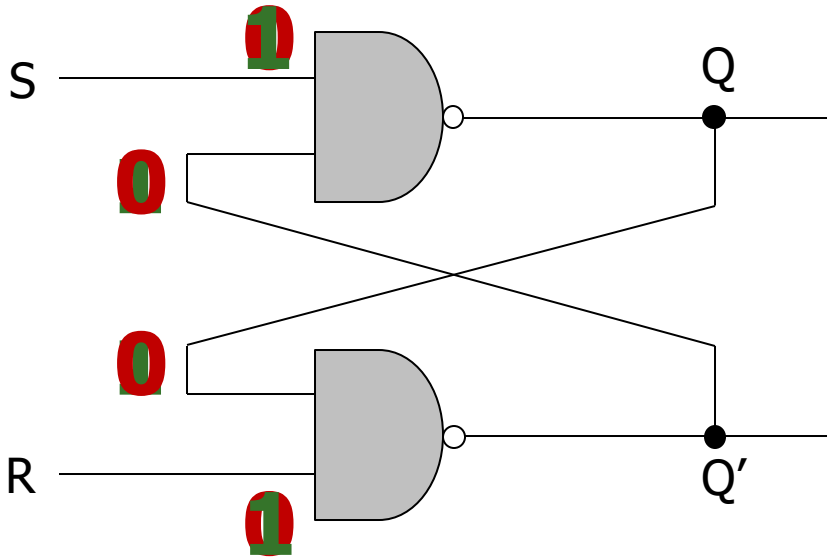
The R-S (Reset-Set) Latch

- Cross-coupled **NAND gates**
  - Data is stored at **Q** (inverse at **Q'**)
  - **S** and **R** are control inputs
    - In *quiescent* (*idle*) *state,* **both S and R are held at 1**
    - **S (set):** drive **S** to 0 (keeping **R** at 1) to change **Q** to 1
    - **R (reset):** drive **R** to 0 (keeping **S** at 1) to change **Q** to 0
- **S** and **R** should never **both** be 0 at the same time



Input		Output
R	S	Q
1	1	Q_{prev}
1	0	1
0	1	0
0	0	Forbidden

Why not R=S=0?



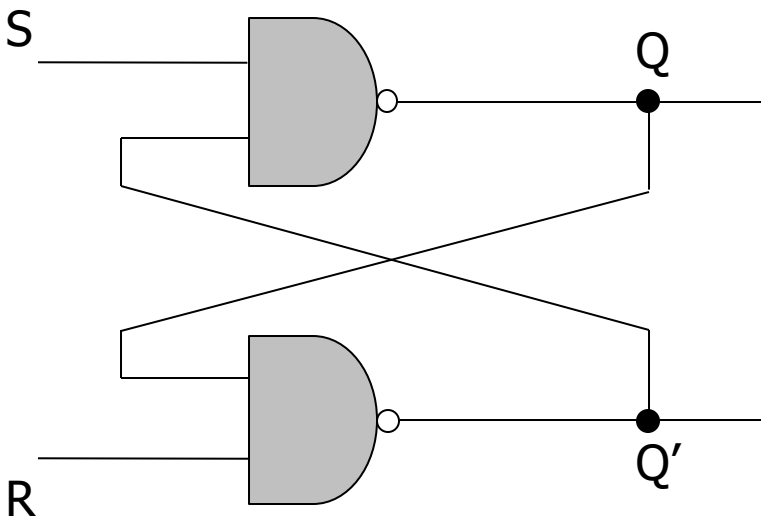
Input		Output
R	S	Q
1	1	Q_{prev}
1	0	1
0	1	0
0	0	Forbidden

1. If **R=S=0, Q** and **Q'** will both settle to 1, which **breaks** our invariant that **Q** = !**Q'**
2. If **S** and **R** transition back to 1 at the same time, **Q** and **Q'** begin to oscillate between 1 and 0 because their final values depend on each other (**metastability**)
   - This eventually settles depending on **variation in the circuits** (more **metastability** to come in **Lecture 8**)

The Gated D Latch

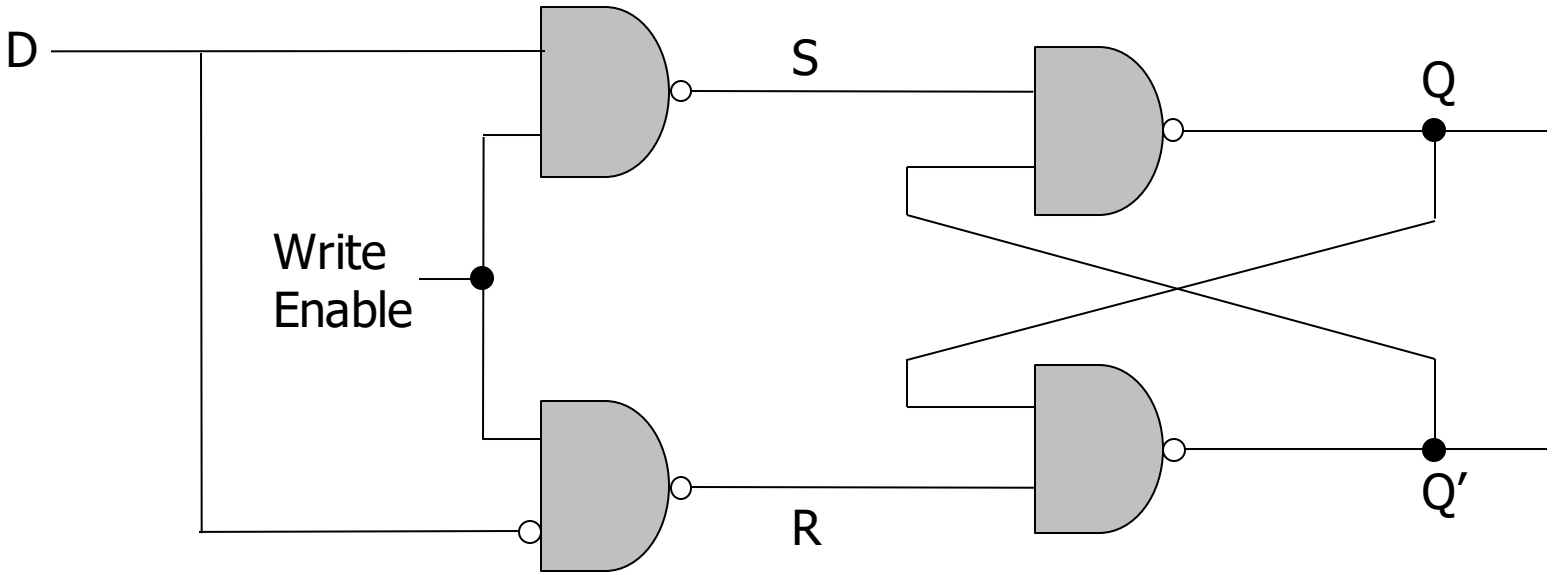
The Gated D Latch

- How do we **guarantee** correct operation of an R-S Latch?



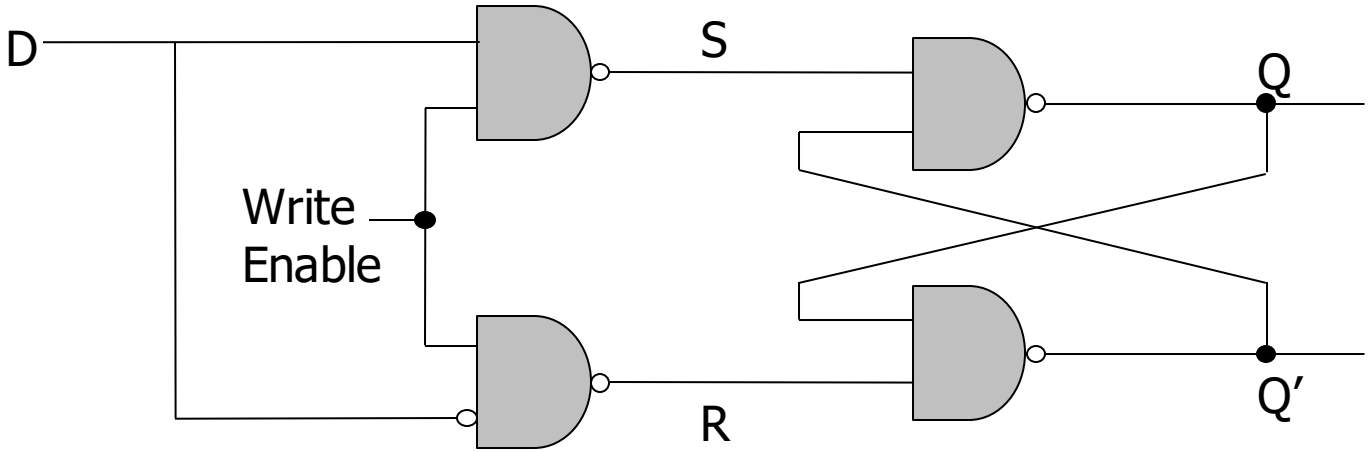
The Gated D Latch

- How do we **guarantee** correct operation of an R-S Latch?
  - Add two more NAND gates!



  - **Q** takes the value of **D**, when **write enable (WE)** is set to 1
  - **S** and **R** can never be 0 at the same time!

The Gated D Latch



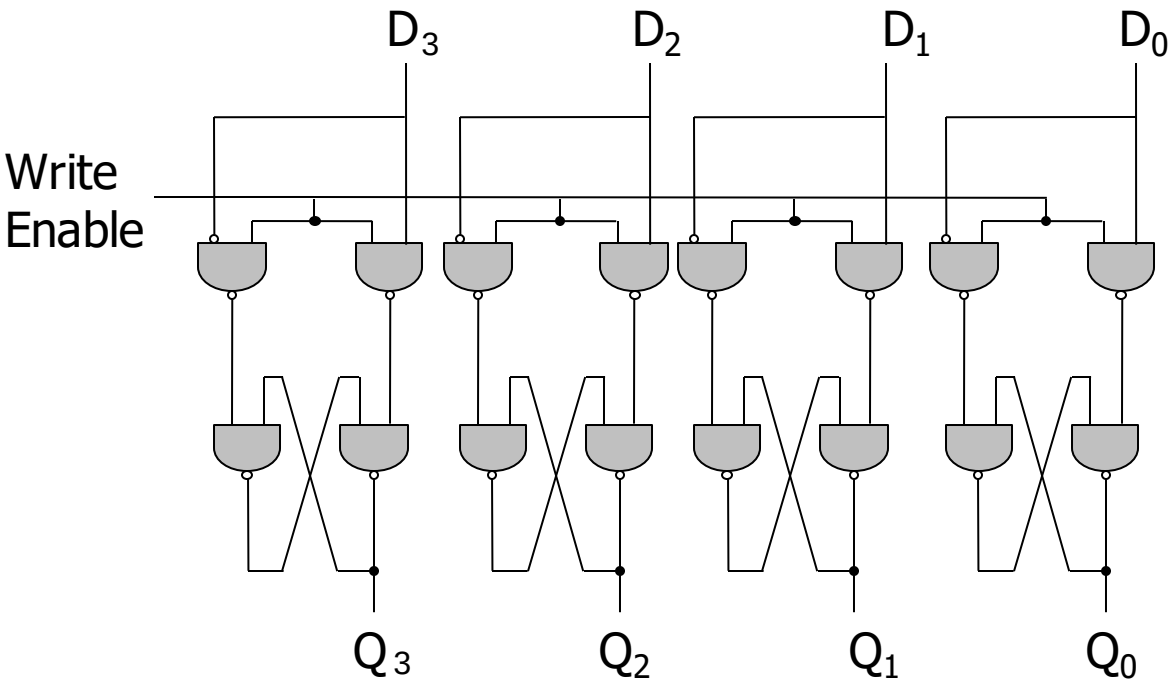
Input		Output
WE	D	Q
0	0	Q_{prev}
0	1	Q_{prev}
1	0	0
1	1	1

The Register

The Register

How can we use D latches to store **more** data?

- Use **more** D latches!
- A single WE signal for all latches for simultaneous writes



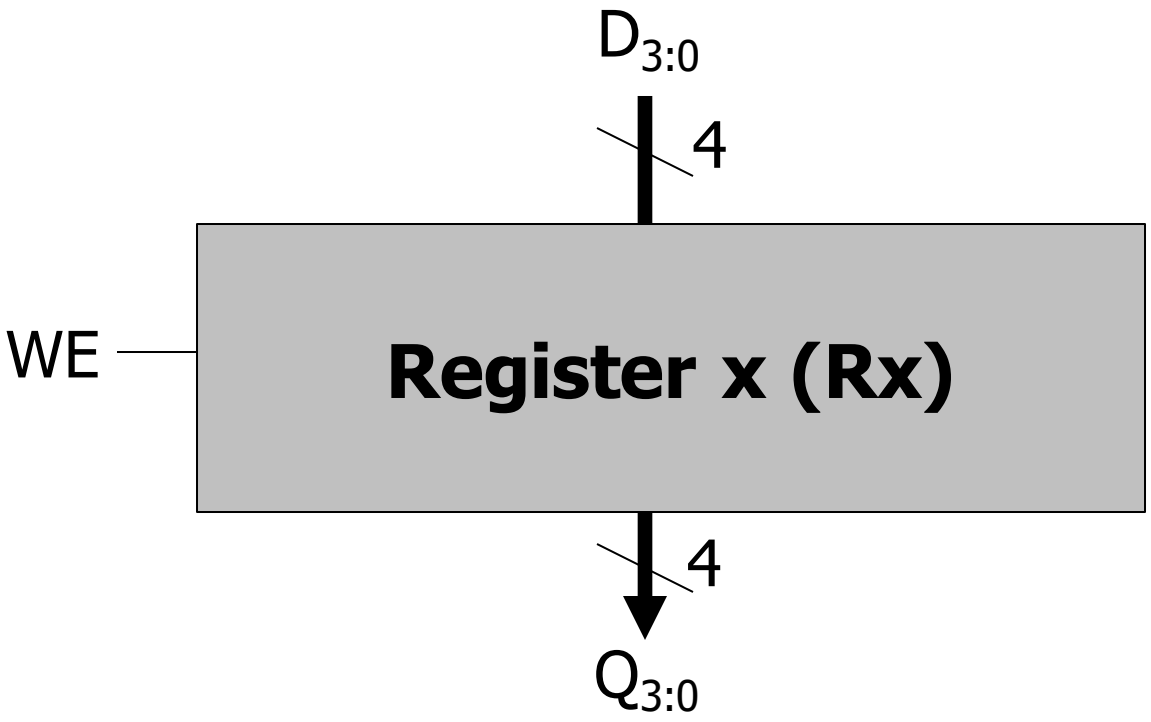
Here we have a **register,** or a structure that stores more than one bit and can be read from and written to

This **register** holds 4 bits, and its data is referenced as Q[3:0]

The Register

How can we use D latches to store **more** data?

- Use **more** D latches!
- A single WE signal for all latches for simultaneous writes



Here we have a **register,** or a structure that stores more than one bit and can be read from and written to

This **register** holds 4 bits, and its data is referenced as Q[3:0]

Memory

Memory

- **Memory** is comprised of locations that can be written to or read from. An example memory array with 4 locations:

Addr(00): 0100 1001	**Addr**(01): 0100 1011
Addr(10): 0010 0010	**Addr**(11): 1100 1001

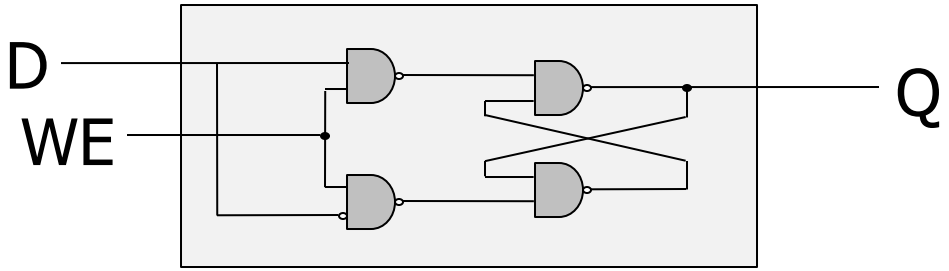
- Every unique location in memory is indexed with a unique **address.** 4 locations require 2 address bits (log[#locations]).
- **Addressability:** the number of **bits** of information stored **in each location**. This example: addressability is 8 bits.
- The entire set of **unique locations** in memory is referred to as the **address space.**
- Typical memory is **MUCH** larger (e.g., billions of locations)

Addressing Memory

Let's implement a simple memory array with:

- 3-bit addressability **&** address space size of 2 (total of 6 bits)

1 Bit



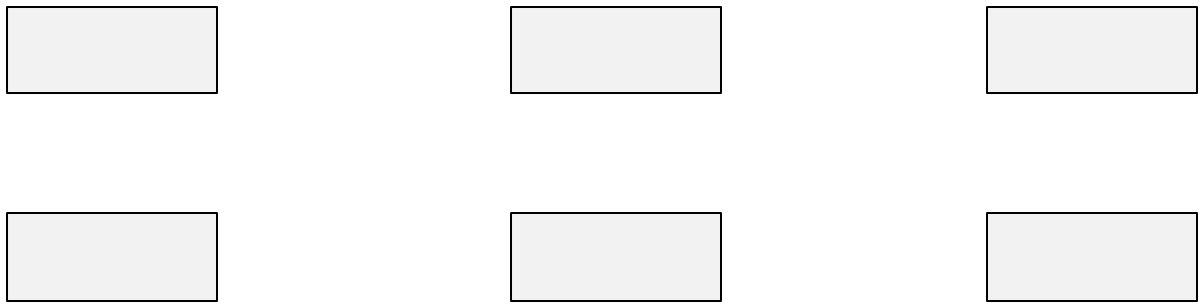
6-Bit Memory Array

Addr(0)	Bit_2	Bit_1	Bit_0
Addr(1)	Bit_2	Bit_1	Bit_0

Reading from Memory

How can we select an address to read?

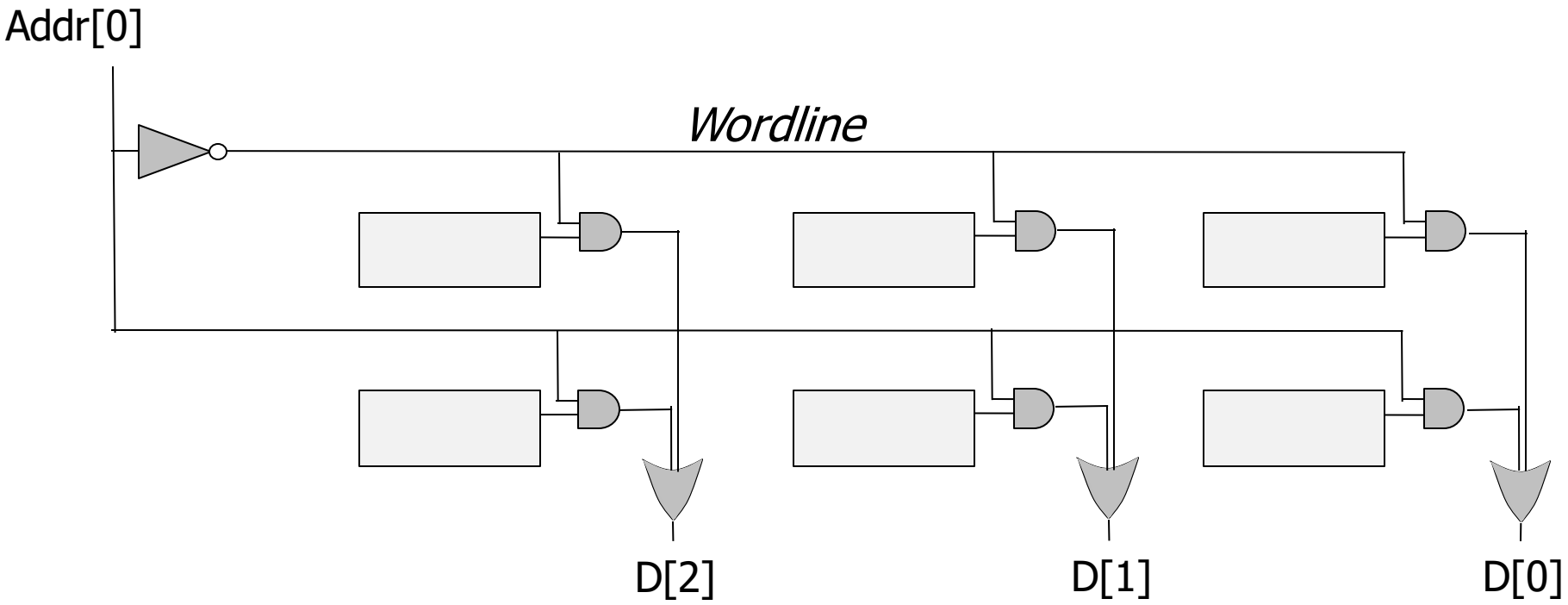
- Because there are 2 addresses, address size is log(2)=1 bit



Reading from Memory

How can we select an address to read?

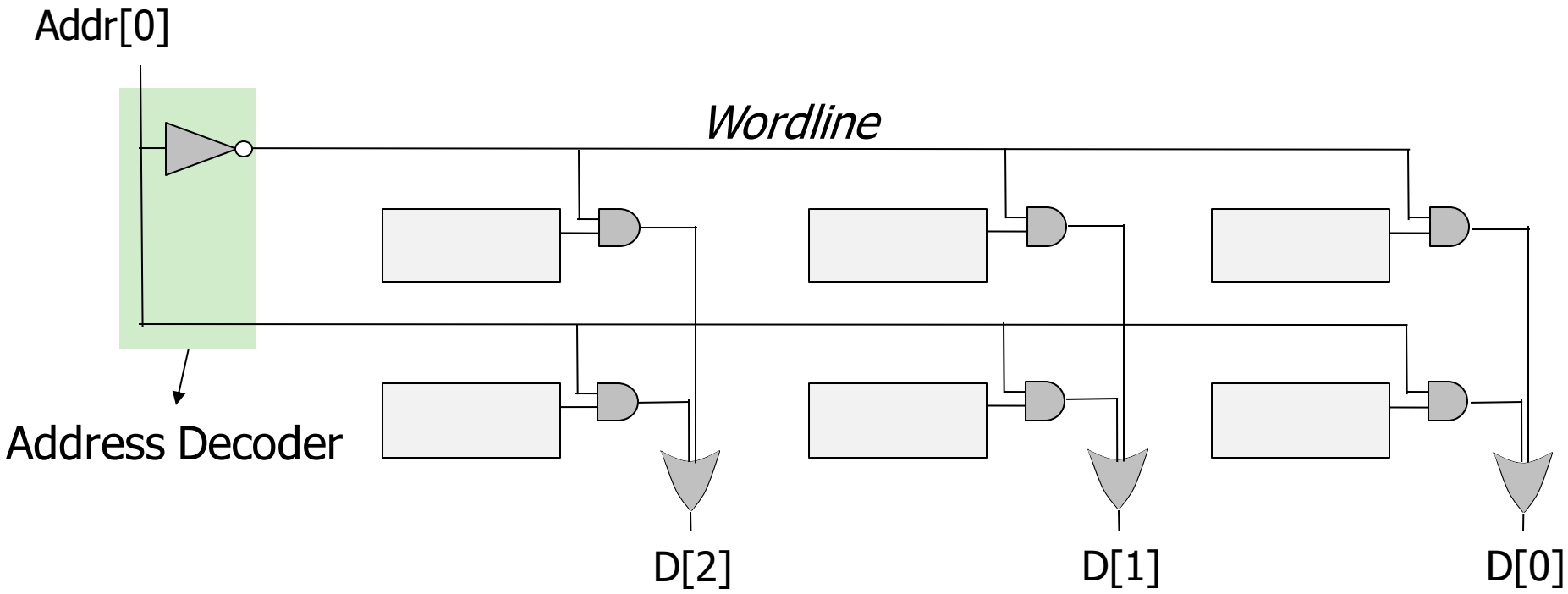
- Because there are 2 addresses, address size is log(2)=1 bit



Reading from Memory

How can we select an address to read?

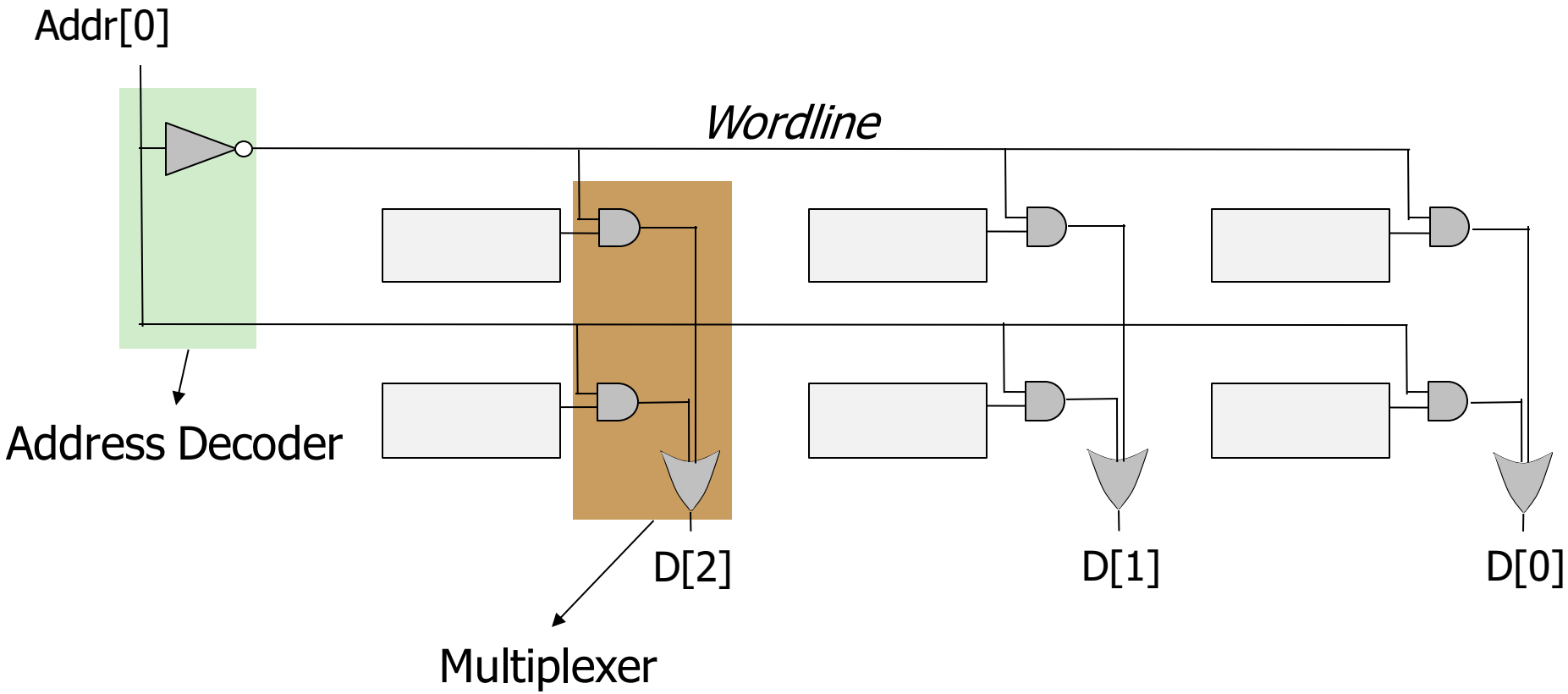
- Because there are 2 addresses, address size is log(2)=1 bit



Reading from Memory

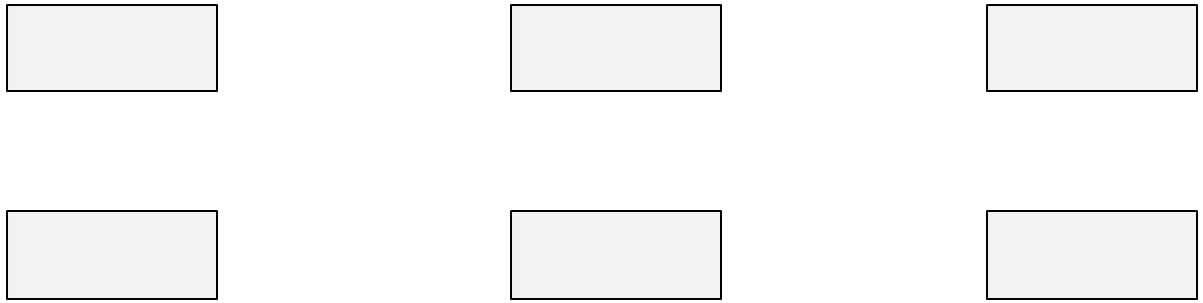
How can we select an address to read?

- Because there are 2 addresses, address size is log(2)=1 bit



Writing to Memory

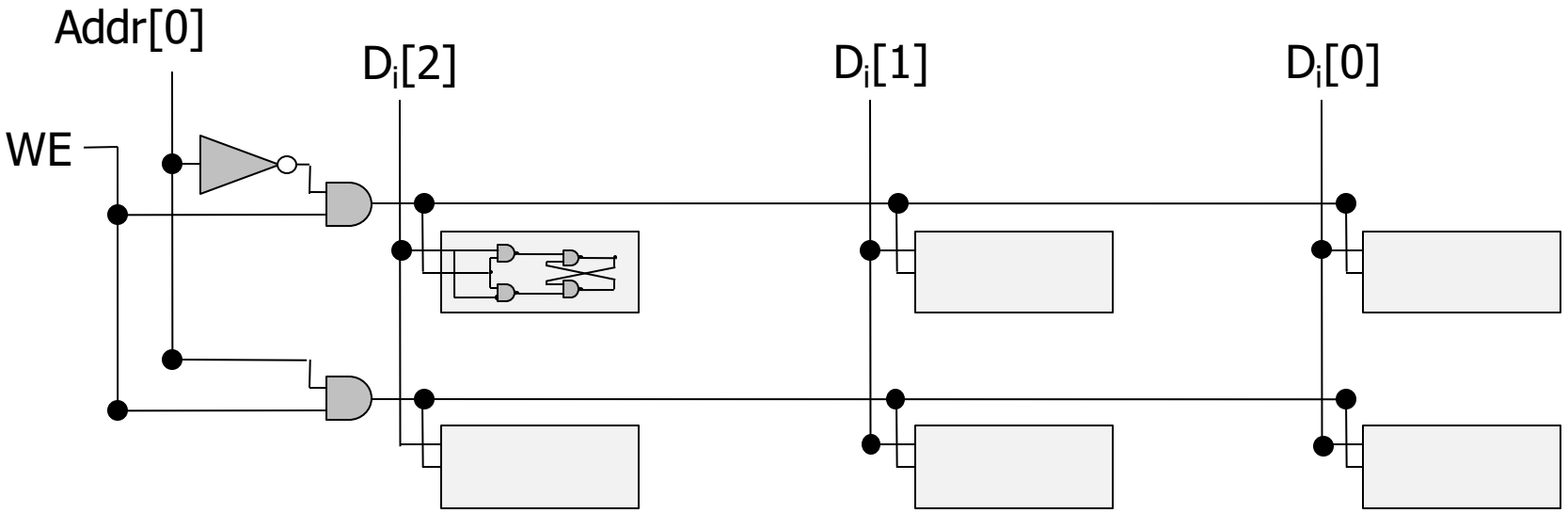
How can we select an address and write to it?



Writing to Memory

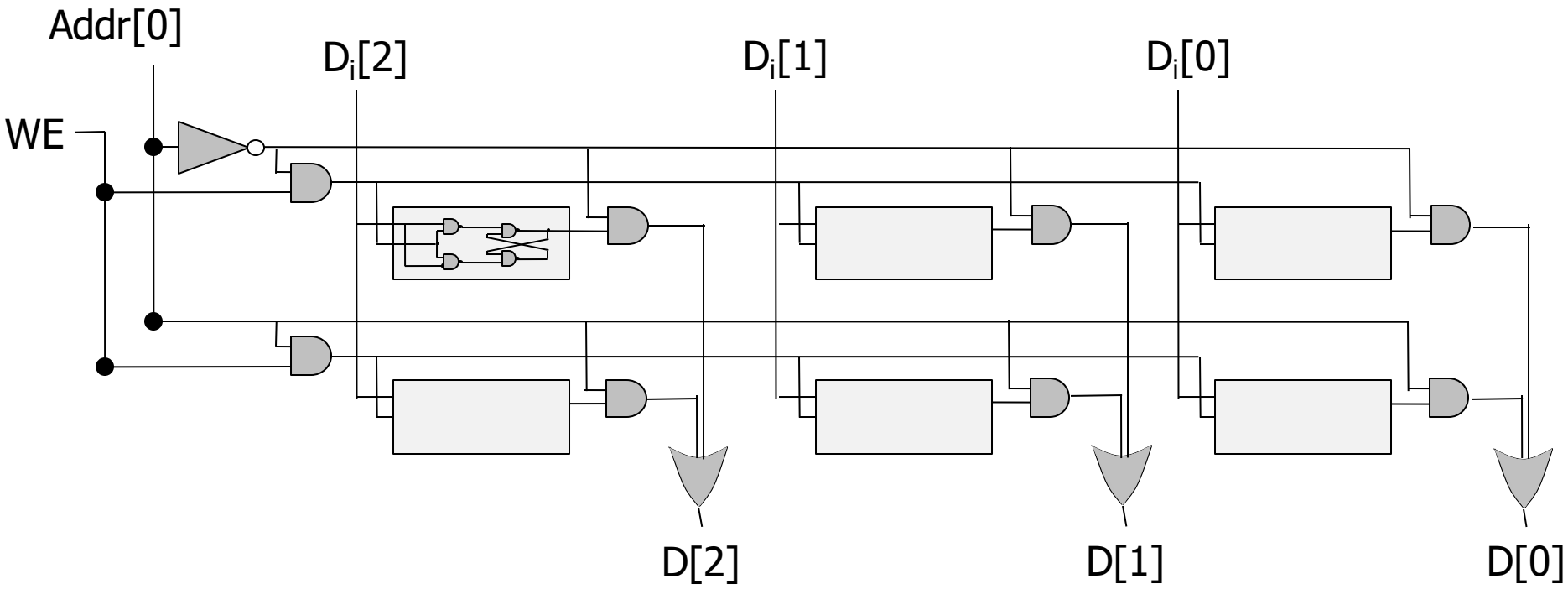
How can we select an address and write to it?

- Input is indicated with D_i

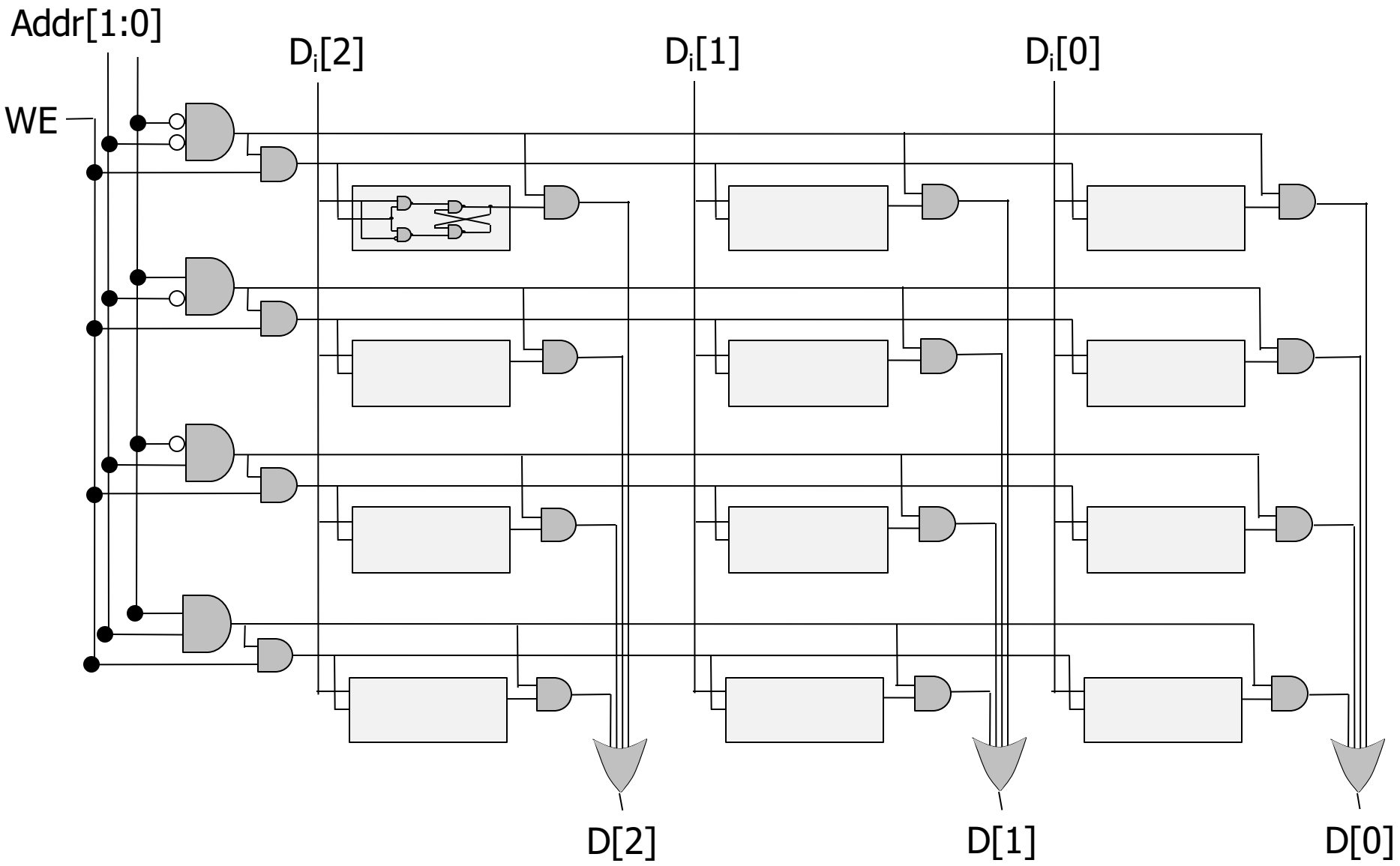


Putting it all Together

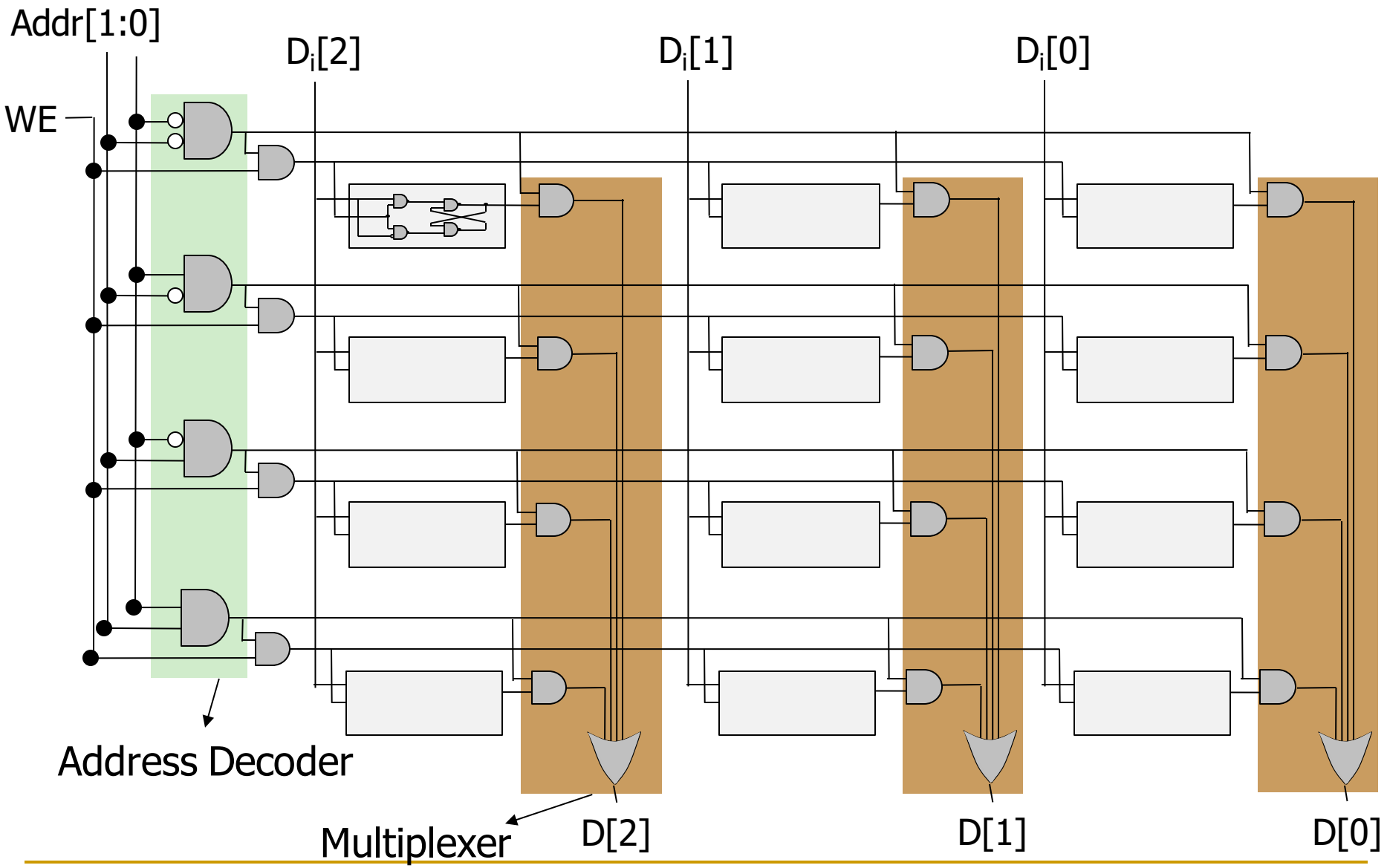
Let's enable reading and writing to a memory array



A Bigger Memory Array (4 locations X 3 bits)



A Bigger Memory Array (4 locations X 3 bits)



Aside: Implementing Logic Functions Using Memory

Memory-Based Lookup Table Example

- Memory arrays can also perform Boolean Logic functions
  - 2^N-location M-bit memory can perform any N-input, M-output function
  - Lookup Table (LUT): Memory array used to perform logic functions
  - Each address: row in truth table; each data bit: corresponding output value

Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

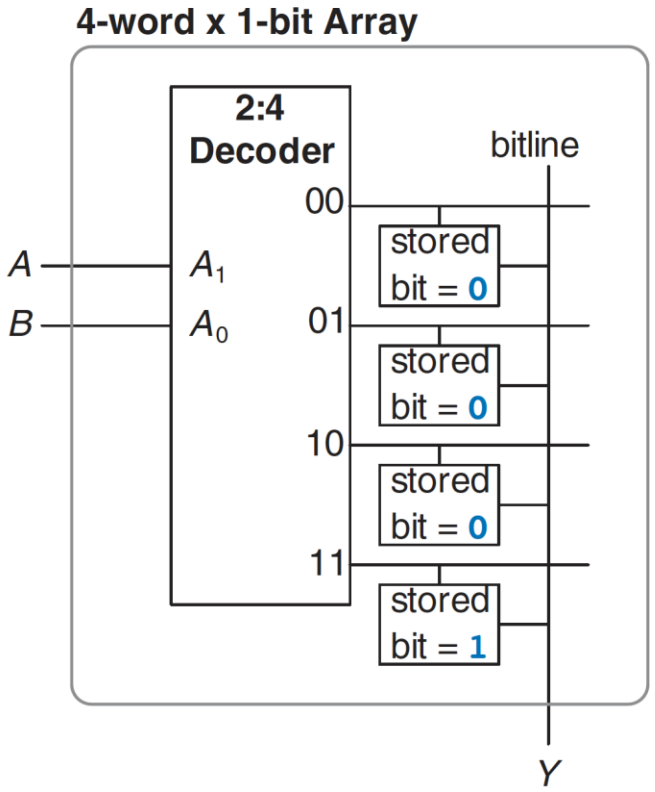


Figure 5.52 4-word × 1-bit memory array used as a lookup table

Lookup Tables (LUTs)

- LUTs are commonly used in FPGAs
  - To enable programmable/reconfigurable logic functions
  - To enable easy integration of combinational and sequential logic

(A) data 1	(B) data 2	(C) data 3	data 4	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(A) data 1	(B) data 2	data 3	data 4	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0

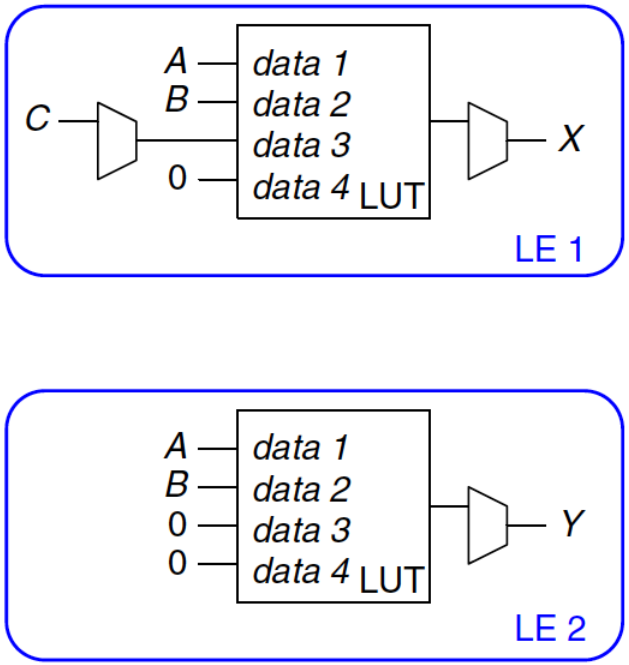


Figure 5.59 **LE configuration for two functions of up to four inputs each**

Read H&H Chapter 5.6.2

Sequential Logic Circuits

Sequential Logic Circuits

- We have examined designs of circuit elements that can **store information**
- Now, we will use these elements to build circuits that **remember** past inputs

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

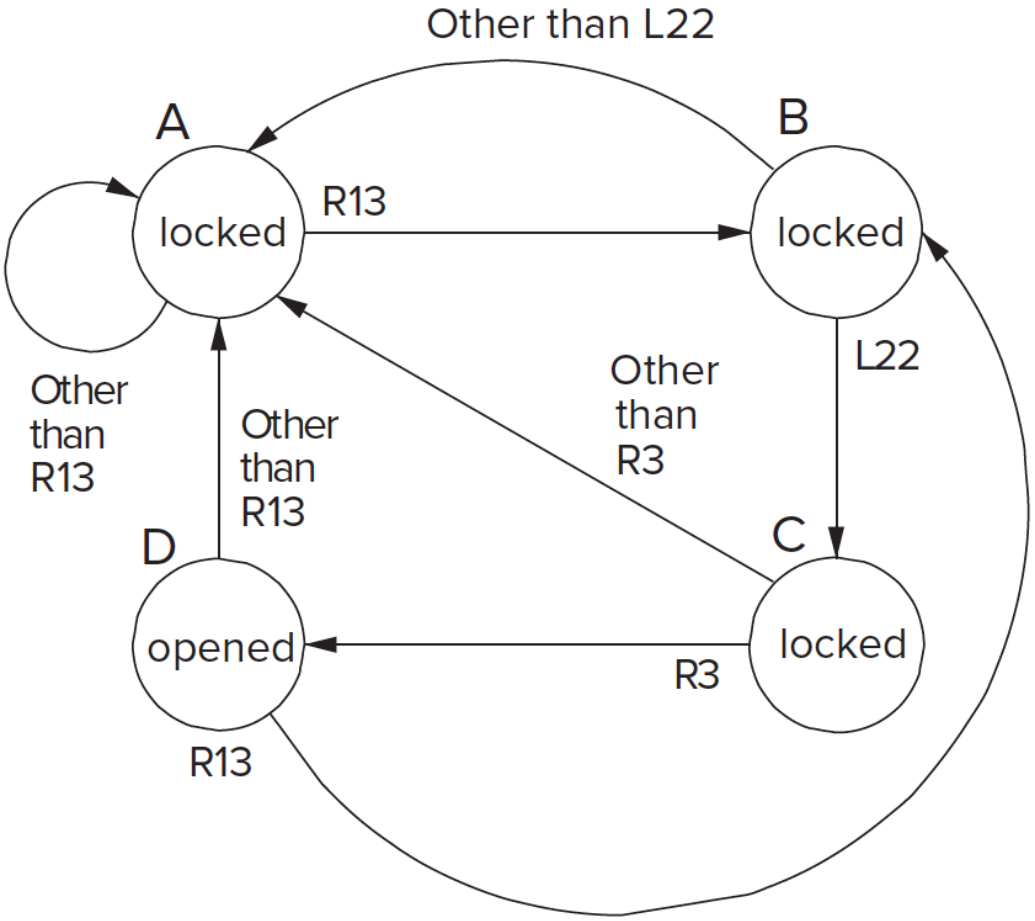
State

- In order for this lock to work, it has to keep track (**remember**) of the past events!
- If passcode is **R13-L22-R3**, sequence of **states** to unlock:
  A. The lock is not open (locked), and no relevant operations have been performed
  B. Locked but user has completed R13
  C. Locked but user has completed R13-L22
  D. Unlocked: user has completed R13-L22-R3

- The **state** of a system is a snapshot of all relevant elements of the system at the moment of the snapshot
  - To open the lock, **states A-D must be completed in order**
  - If anything else happens (e.g., L5), lock **returns** to state A

State Diagram of Our Sequential Lock

- Completely describes the operation of the sequential lock

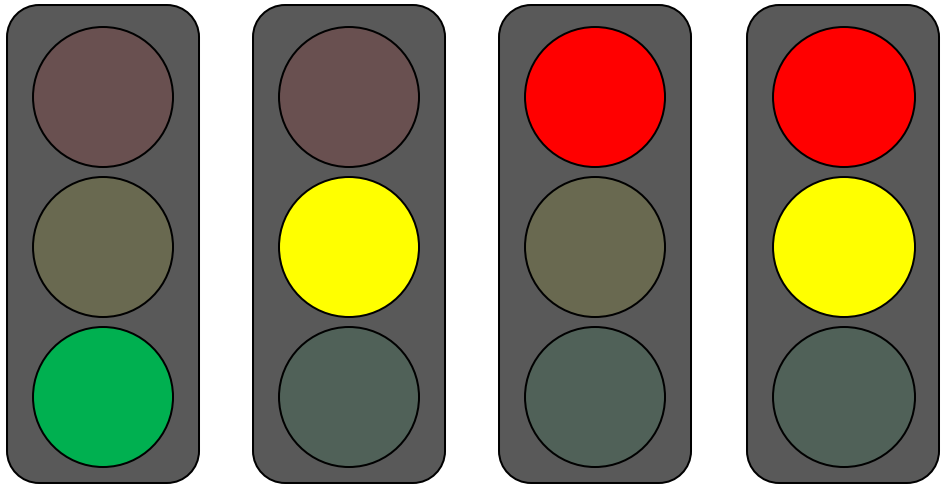


- We will understand "state diagrams" fully later today

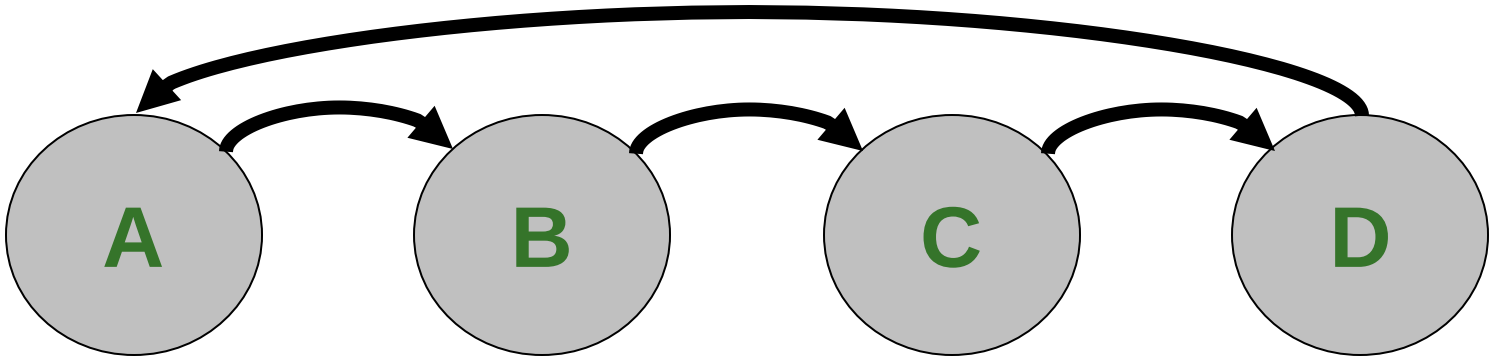
Image source: Patt and Patel, "Introduction to Computing Systems", 2nd ed., page 76.

Another Simple Example of State

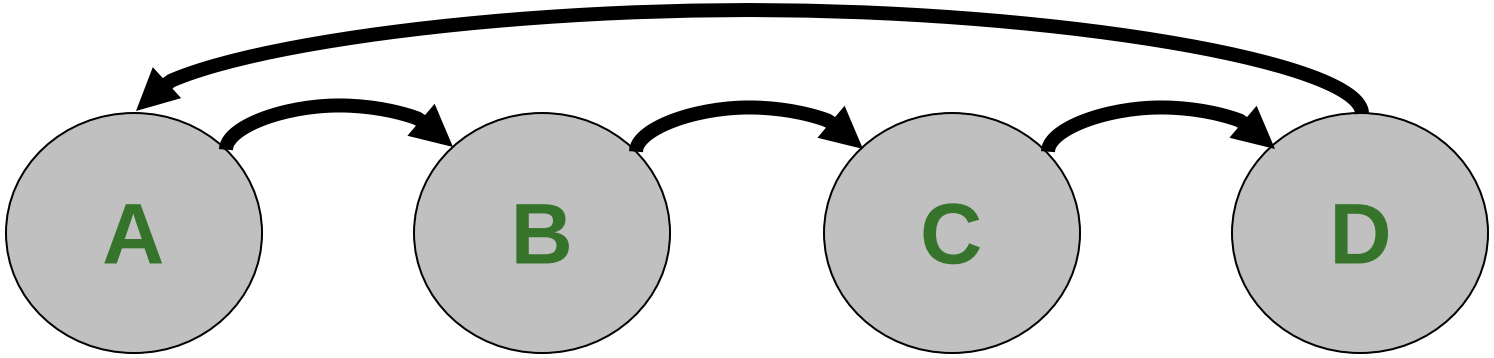
- A standard Swiss traffic light has **4 states**
  A. Green
  B. Yellow
  C. Red
  D. Red and Yellow



- The sequence of these states are always as follows



Changing State: The Notion of Clock (I)



- When should the light change from one state to another?
- We need a **clock** to dictate when to change state
  - Clock signal alternates between 0 & 1

- At the start of a clock cycle ( ), system state changes
  - During a clock cycle, the state stays constant
  - In this traffic light example, we are assuming the traffic light stays in each state an equal amount of time

Changing State: The Notion of Clock (II)

- **Clock** is a general mechanism that triggers transition from one state to another in a sequential circuit
- Clock synchronizes state changes across many sequential circuit elements
- Combinational logic evaluates for the length of the clock cycle
- Clock cycle should be chosen to accommodate maximum combinational circuit delay
  - More on this later, when we discuss timing (Lecture 8)

Finite State Machines

Finite State Machines

- What is a **Finite State Machine** (FSM)?
  - **A discrete-time model** of a stateful system
  - Each state represents a snapshot of the system at a given time

- An FSM pictorially shows
  1. the set of all possible **states** that a system can be in
  2. how the system transitions from one state to another

- An FSM can model
  - A traffic light, an elevator, fan speed, a microprocessor, etc.

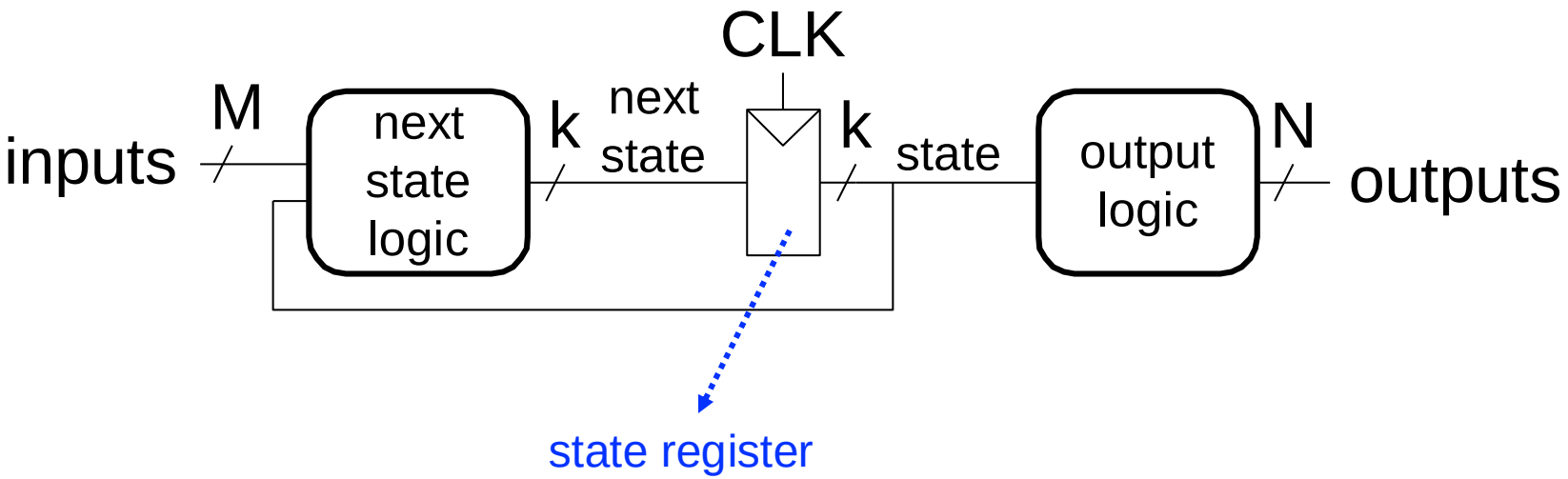
- **An FSM enables us to pictorially think of a stateful system using simple diagrams**

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

Finite State Machines (FSMs)

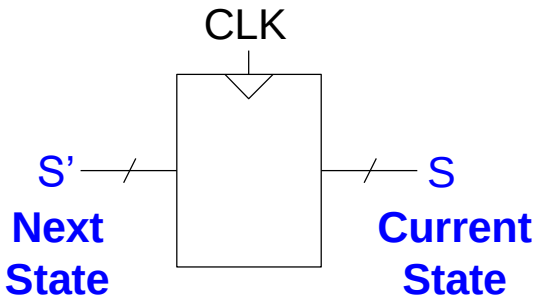
- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic



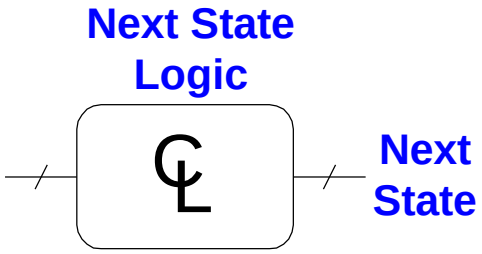
At the beginning of the clock cycle, next state is latched into the state register

Finite State Machines (FSMs) Consist of:

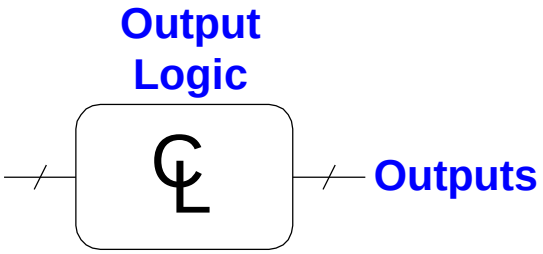
- **Sequential circuits**
  - State register(s)
    - Store the current state and
    - Load the next state at the clock edge



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

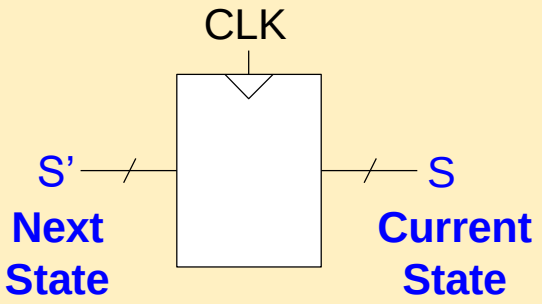


  - Output logic
    - Generates the outputs

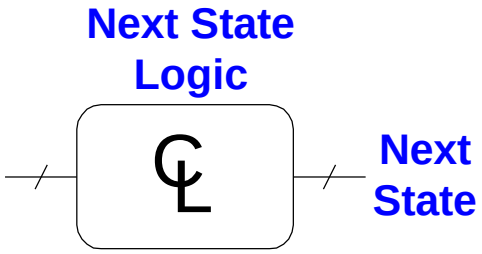


Finite State Machines (FSMs) Consist of:

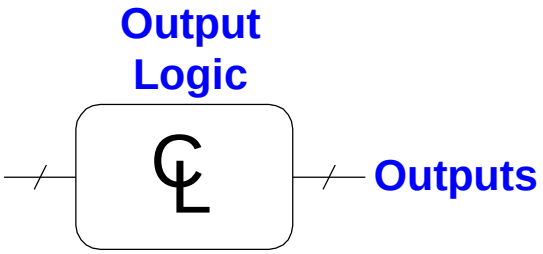
- **Sequential circuits**
  - State register(s)
    - Store the current state and
    - Load the next state at the clock edge



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



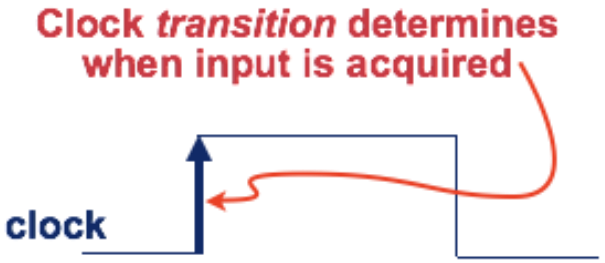
  - Output logic
    - Generates the outputs



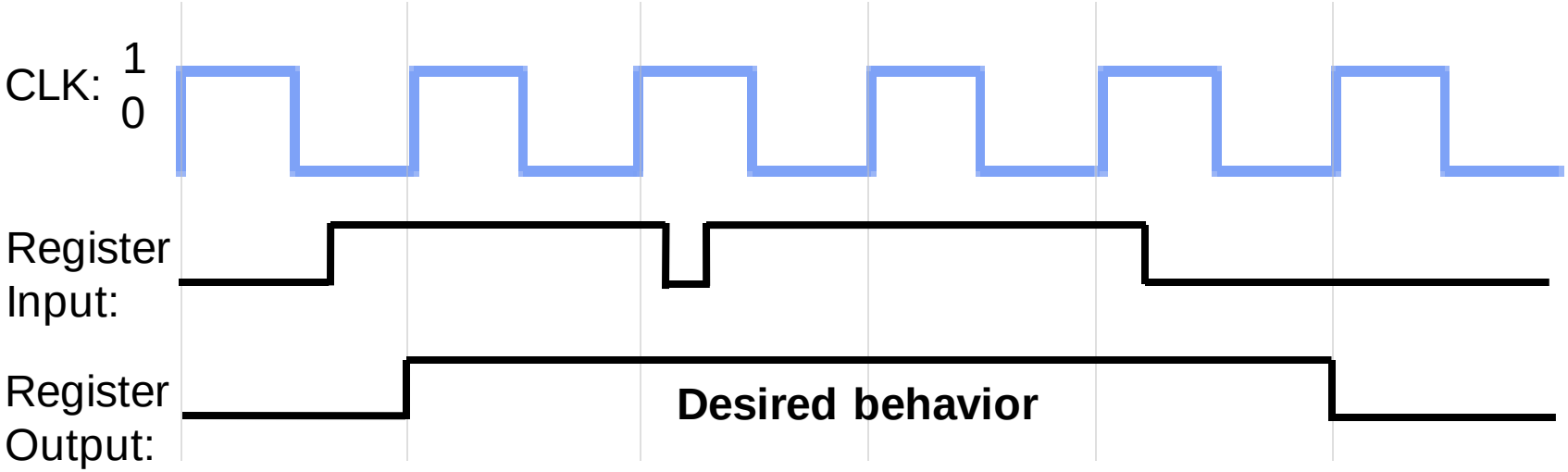
State Register Implementation

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

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

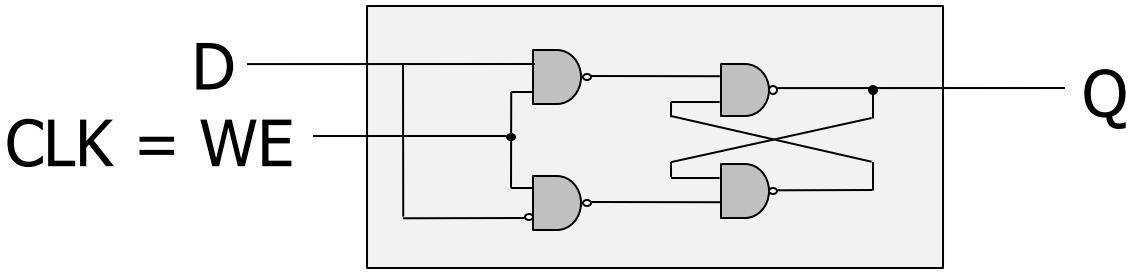


  2. The data must be **available** during the **entire clock cycle**

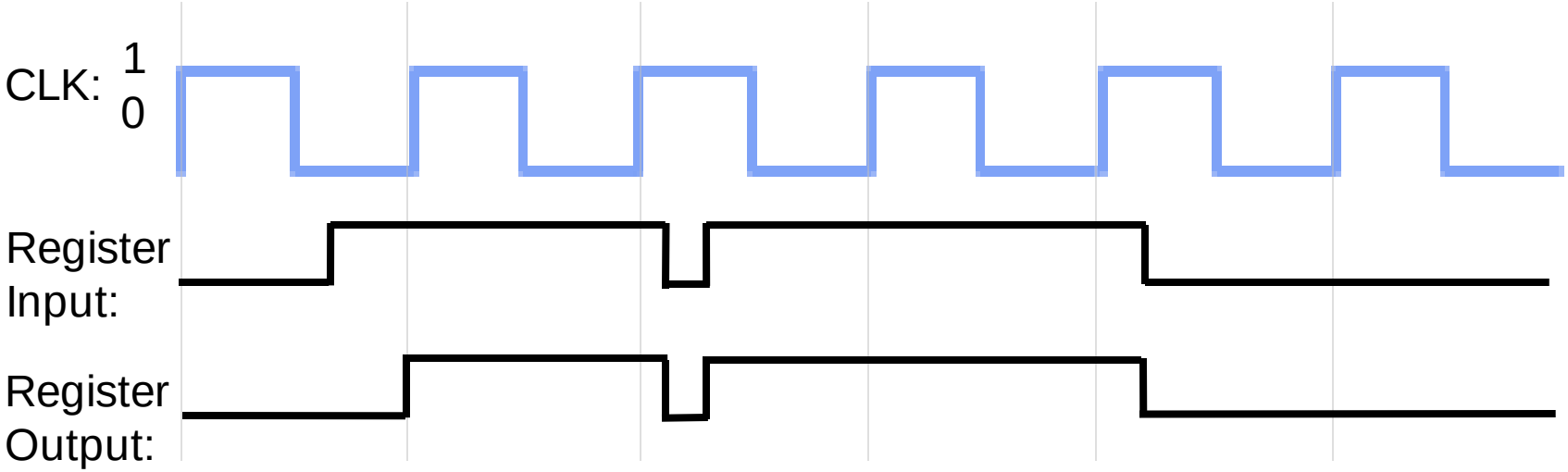


The Problem with Latches

Recall the Gated D Latch

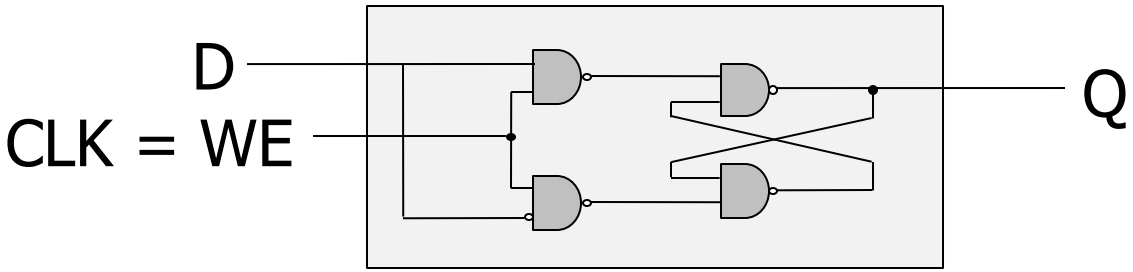


- Currently, we **cannot** simply wire a clock to WE of a latch
  - **Whenever the clock is high,** the latch propagates **D** to **Q**
  - **The latch is transparent**

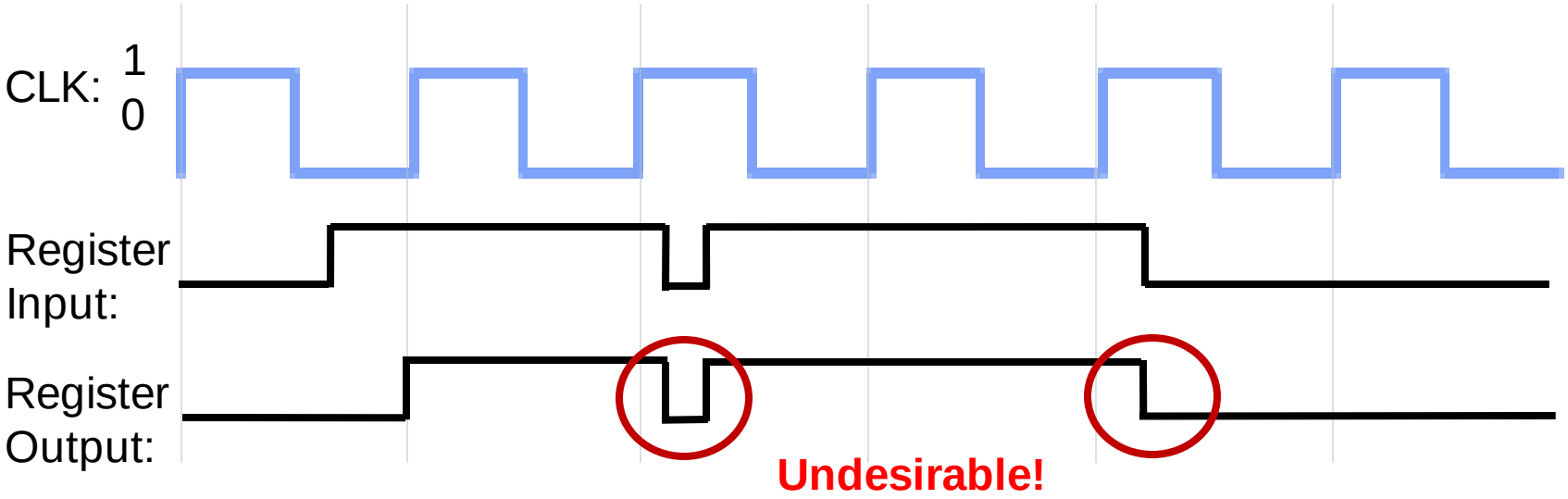


The Problem with Latches

Recall the Gated D Latch

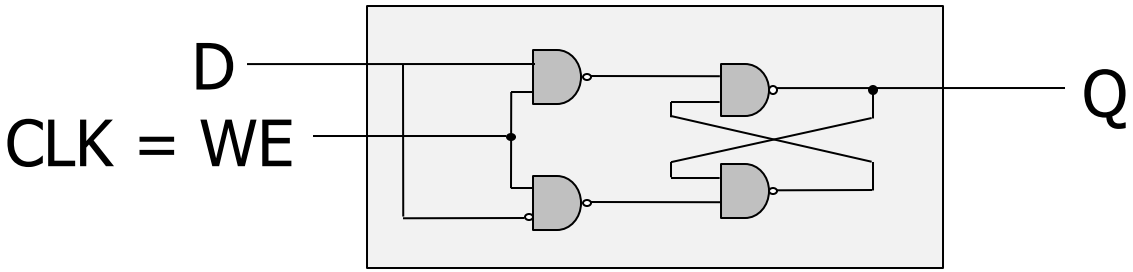


- Currently, we **cannot** simply wire a clock to WE of a latch
  - **Whenever the clock is high,** the latch propagates **D** to **Q**
  - **The latch is transparent**



The Problem with Latches

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**

The Need for a New Storage Element

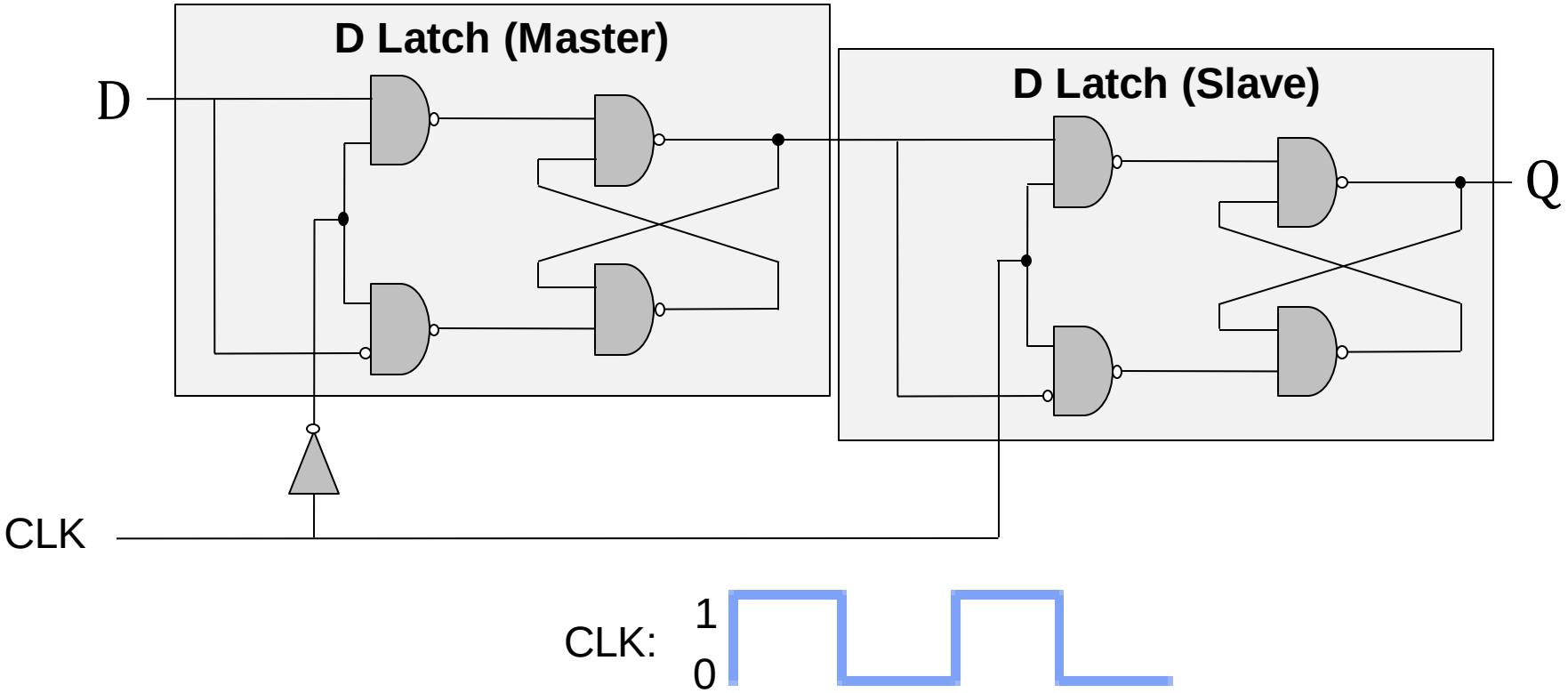
- To design viable FSMs
- We need storage elements that allow us
  - to read the **current state** throughout the **current clock cycle**

  AND

  - not write the **next state** values into the storage elements until the beginning of the **next clock cycle**.

The D Flip-Flop

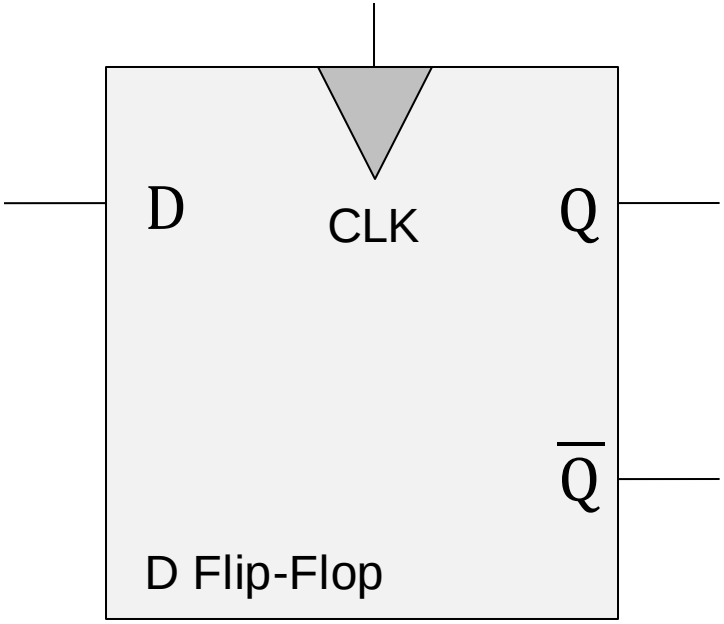
- 1) state change on clock edge, 2) data available for full cycle



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

The D Flip-Flop

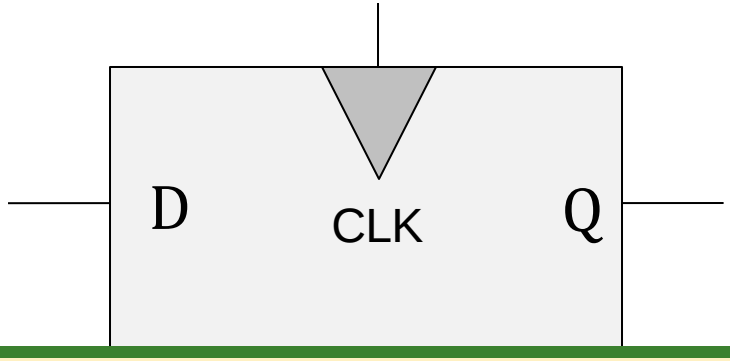
- 1) state change on clock edge, 2) data available for full cycle



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

The D Flip-Flop

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



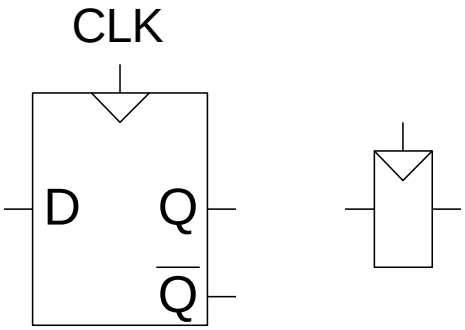
We can use these **Flip-Flops**
to implement the state register!

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

Rising-Clock-Edge Triggered Flip-Flop

- **Two inputs**: CLK, D

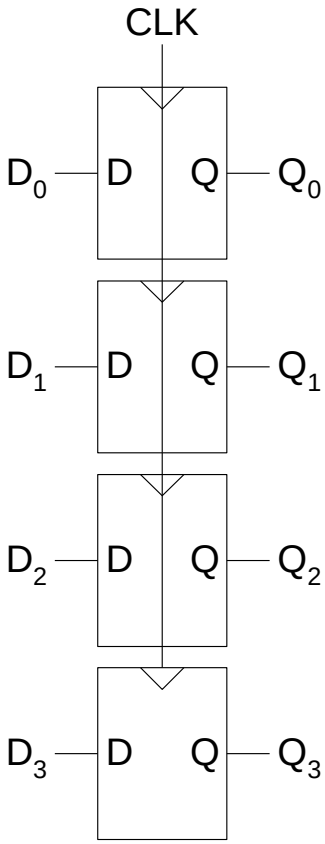
- **Function**
  - The flip-flop "samples" **D** on the rising edge of CLK (**positive edge**)
  - When CLK rises from 0 to 1, **D** passes through to **Q**
  - Otherwise, **Q** holds its previous value
  - **Q** changes **only** on the rising edge of CLK

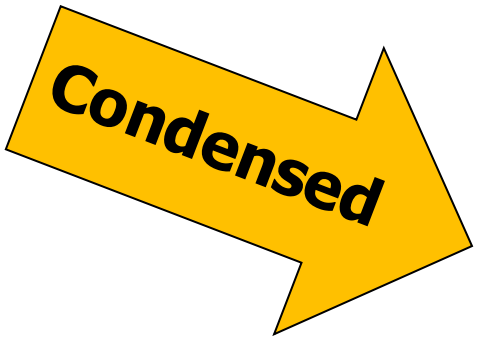


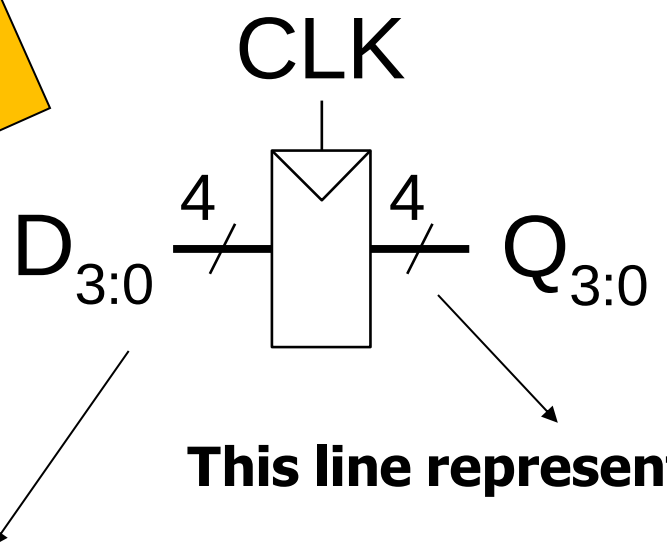
- A flip-flop is called an **edge-triggered state element** because it captures data on the clock edge
  - A latch is a **level-triggered** state element

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)

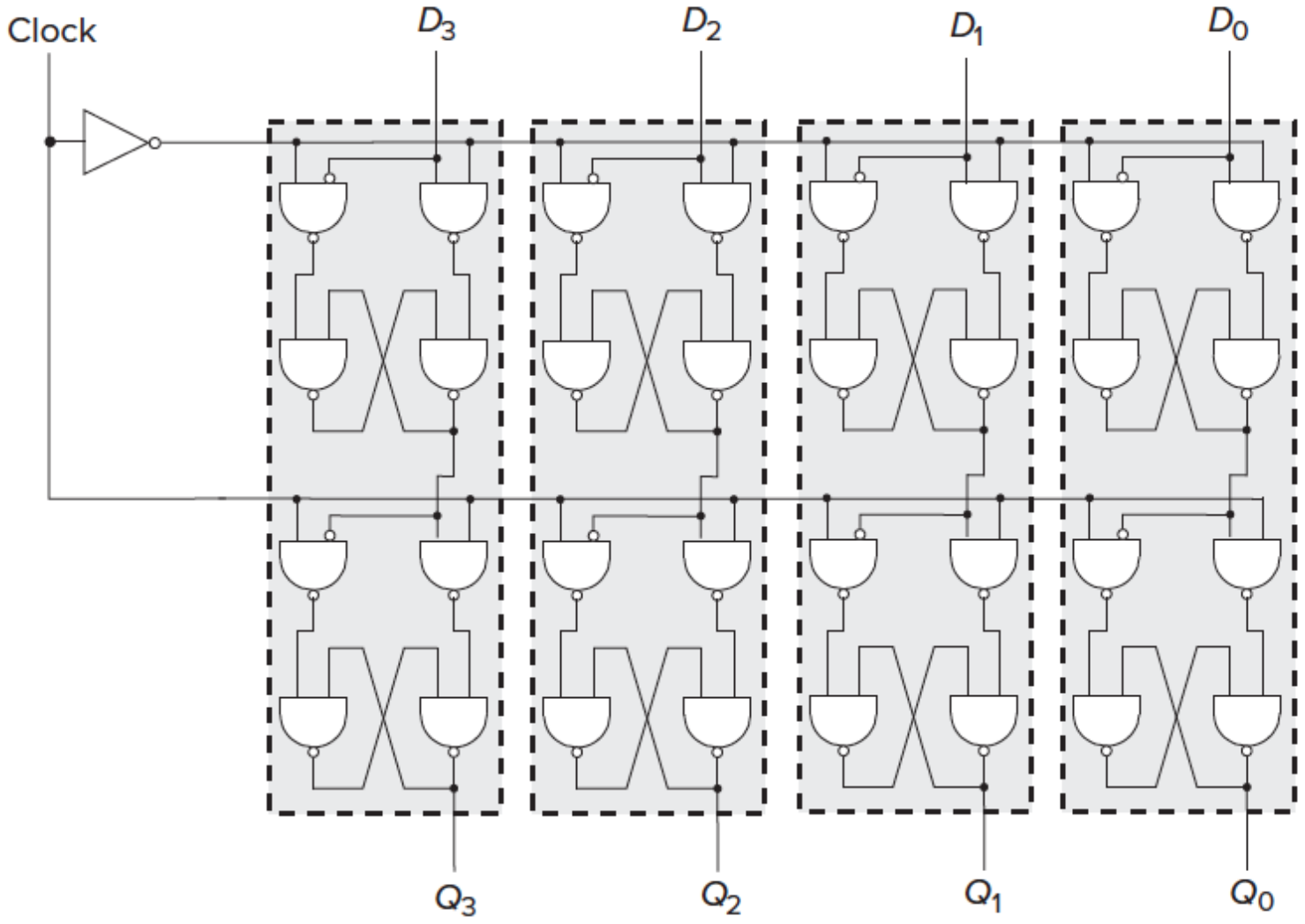
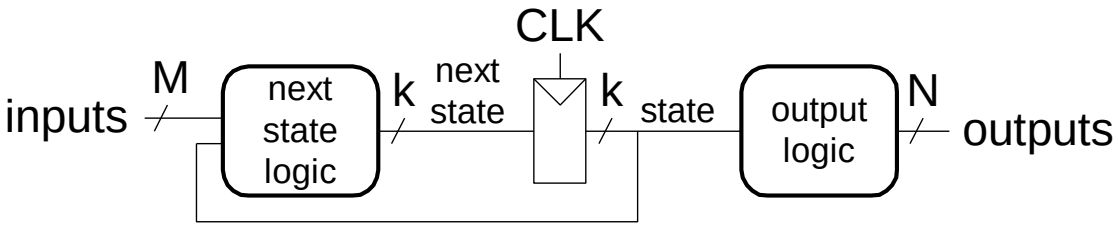


Image source: Patt and Patel, "Introduction to Computing Systems", 3rd ed., tentative page 95.

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

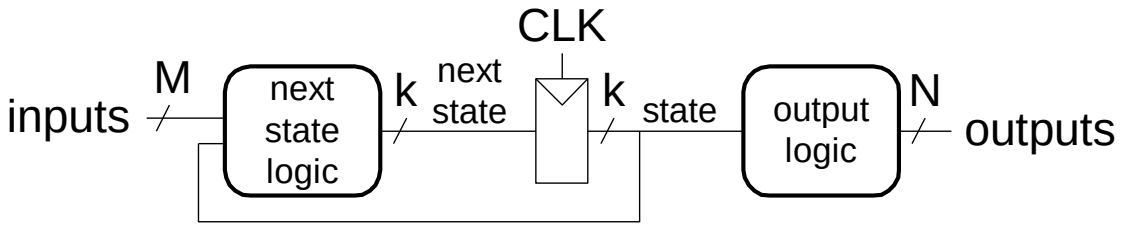
Moore FSM



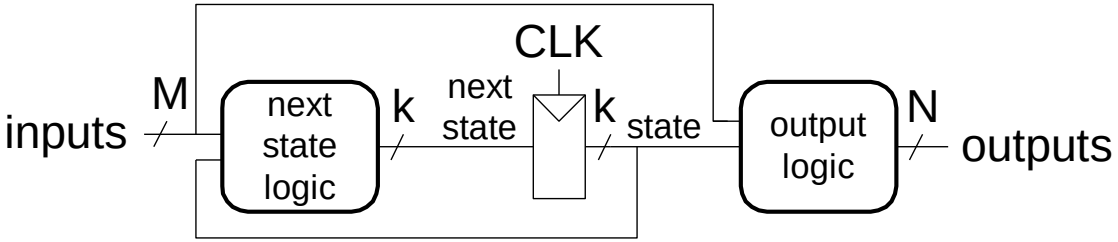
Finite State Machines (FSMs)

- 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

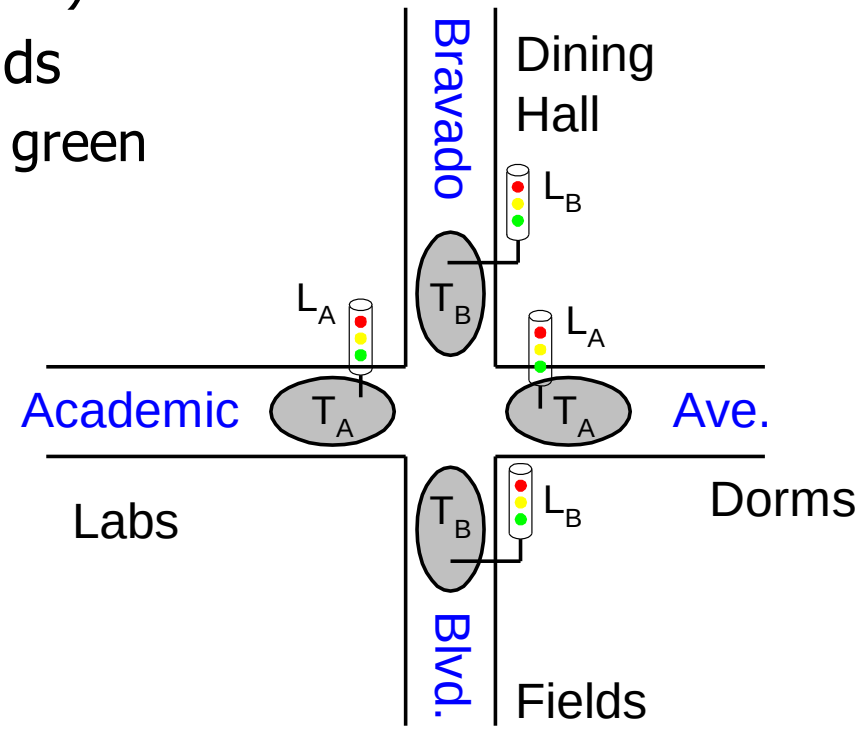


Mealy FSM



Finite State Machine Example

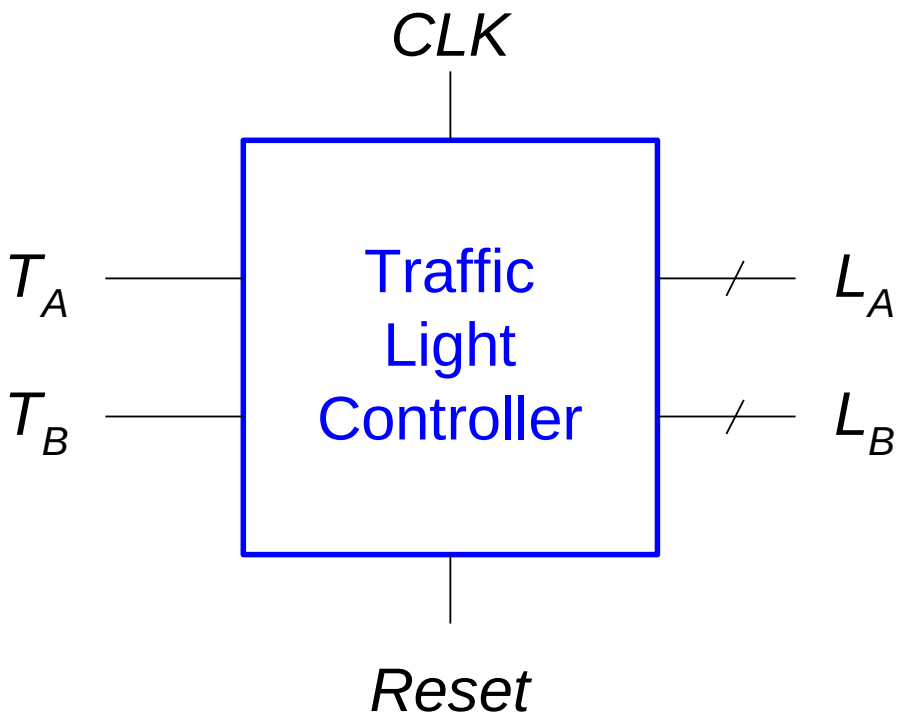
- "Smart" traffic light controller
  - **2 inputs**:
    - Traffic sensors: T_A , T_B (TRUE when there's traffic)
  - **2 outputs**:
    - Lights: L_A , L_B (Red, Yellow, Green)
  - State can change every 5 seconds
    - Except if green and traffic, stay green



From H&H Section 3.4.1

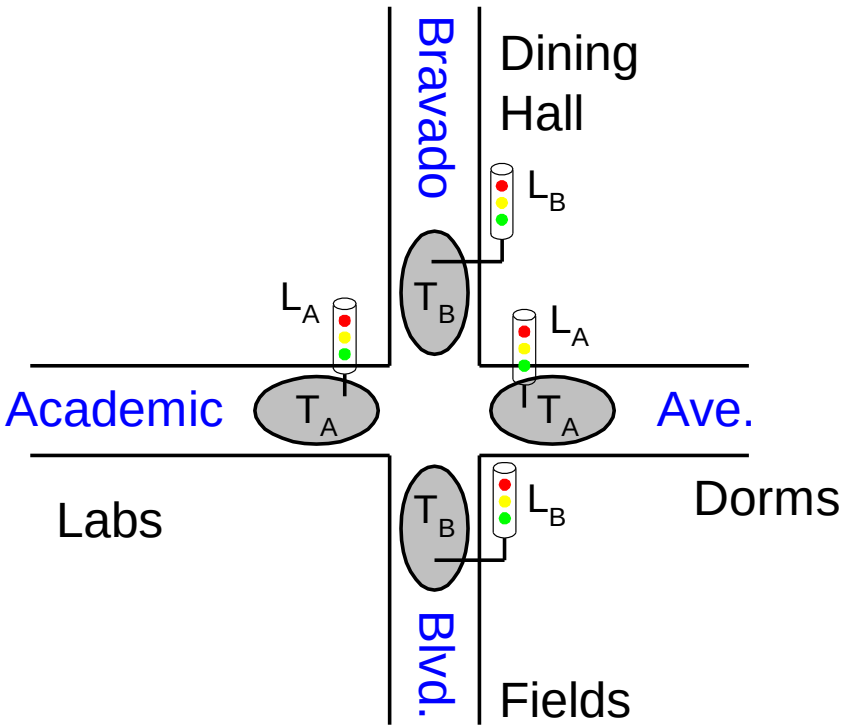
Finite State Machine Black Box

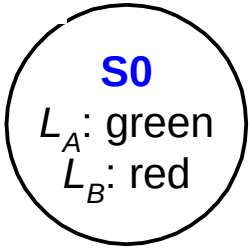
- **Inputs:** CLK, Reset, T_A , T_B
- **Outputs:** L_A , L_B



Finite State Machine Transition Diagram

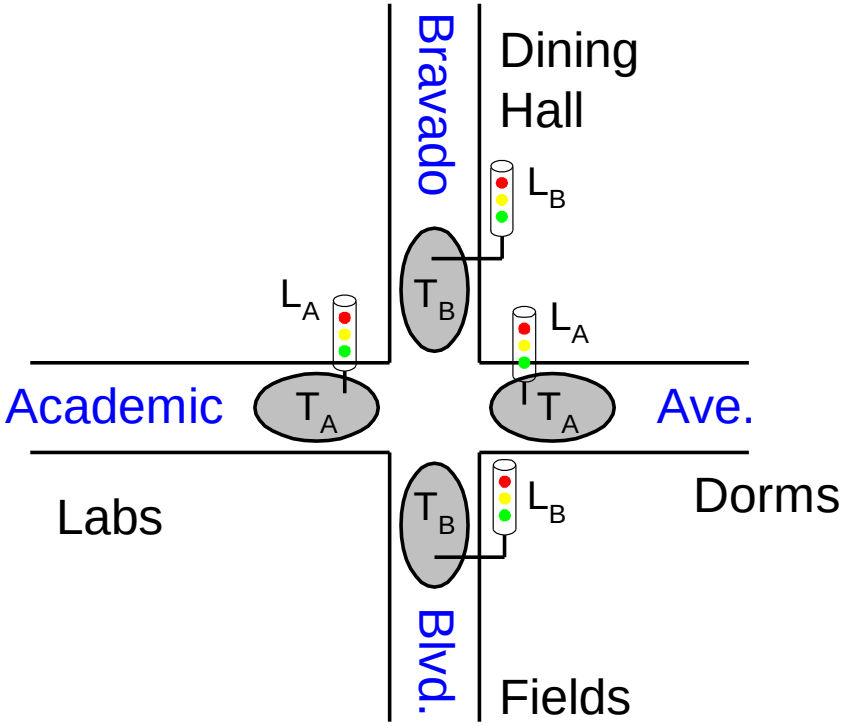
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

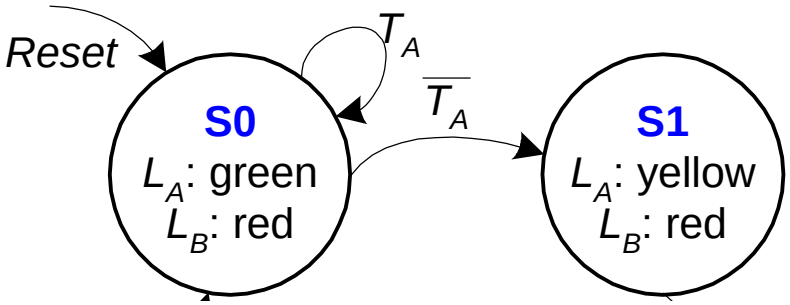




Finite State Machine Transition Diagram

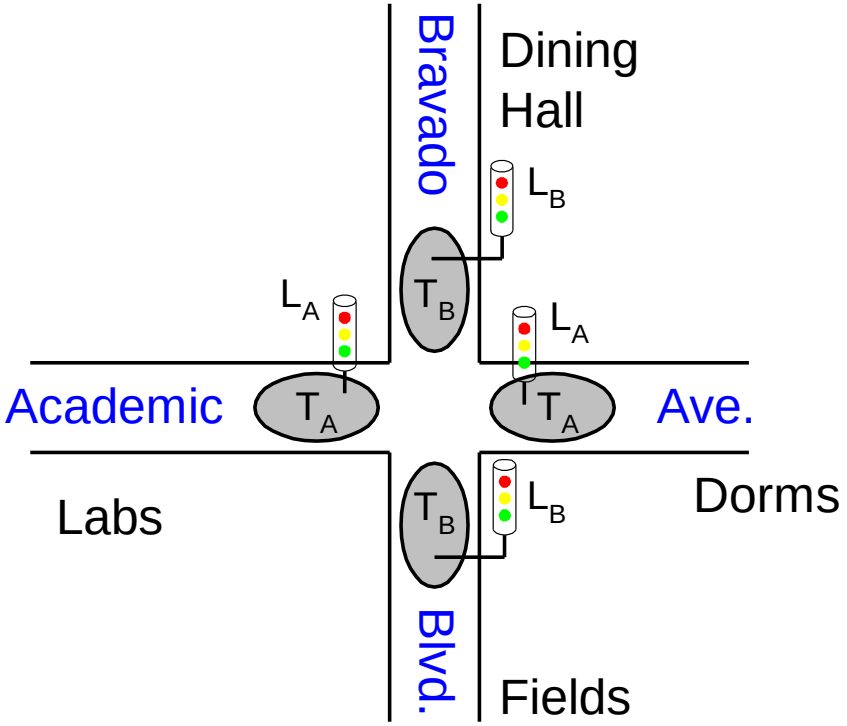
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

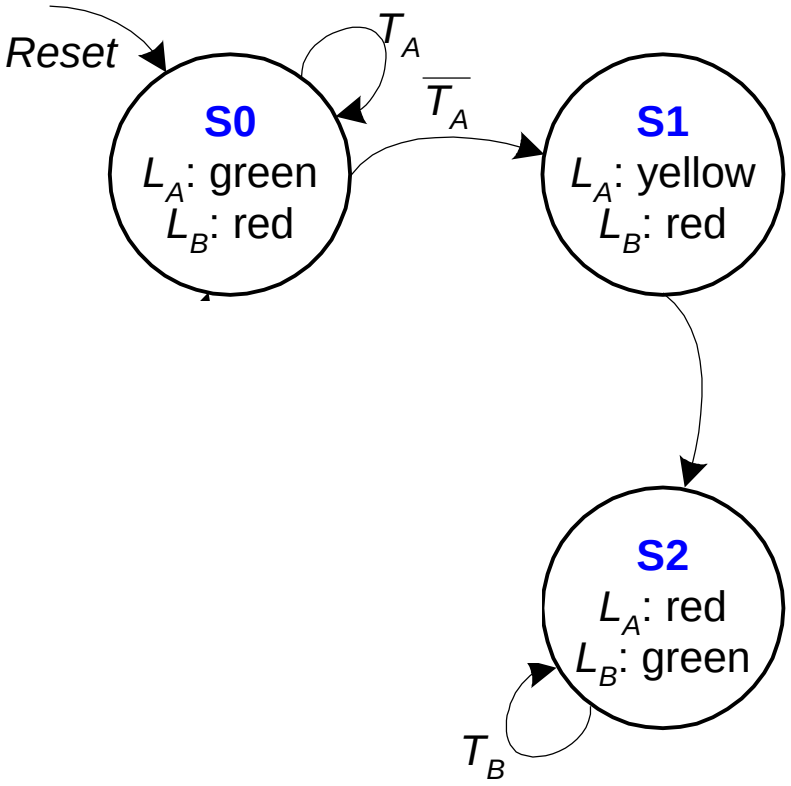




Finite State Machine Transition Diagram

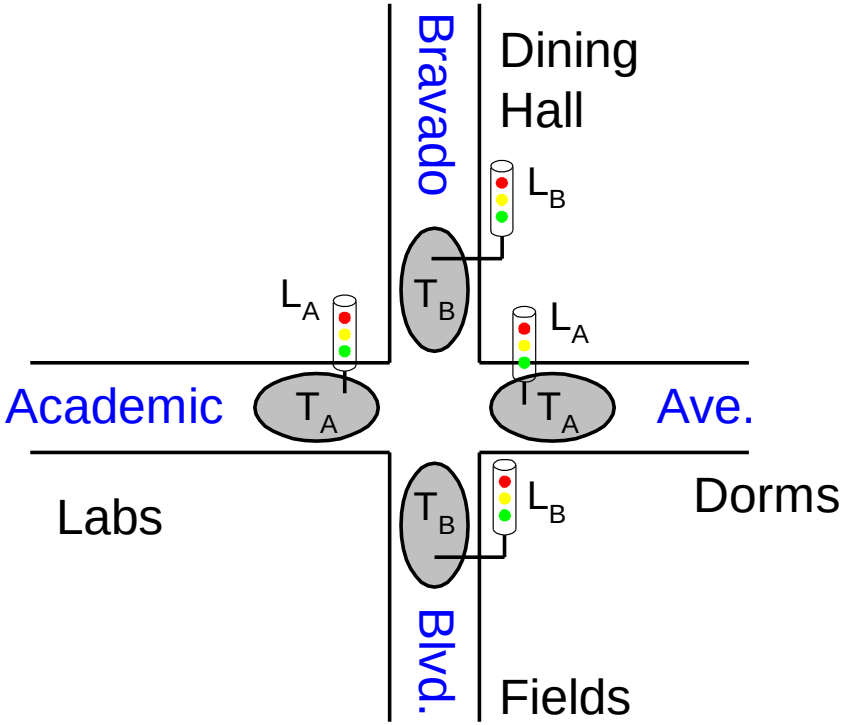
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

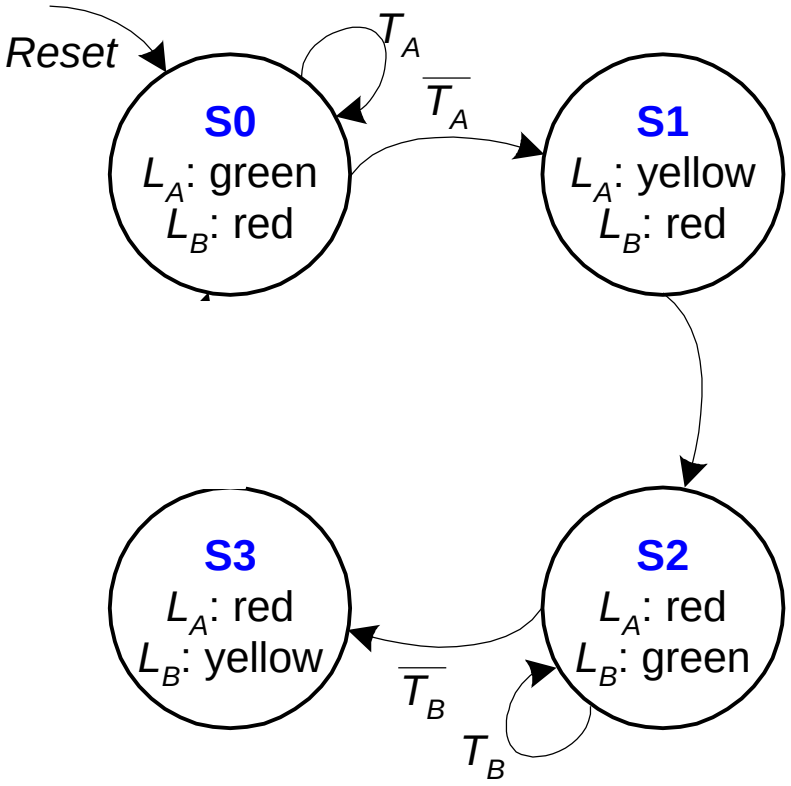




Finite State Machine Transition Diagram

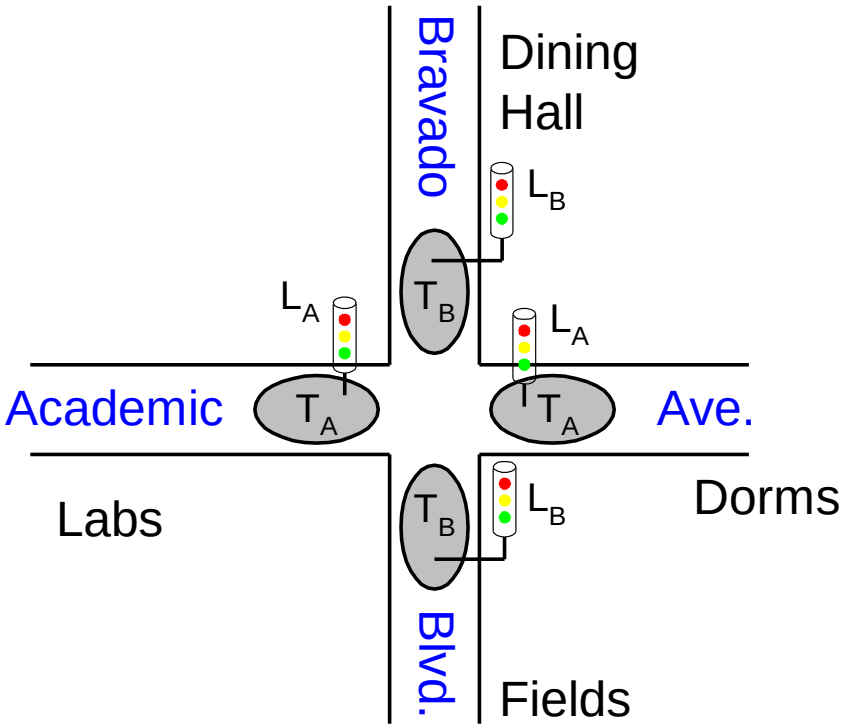
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

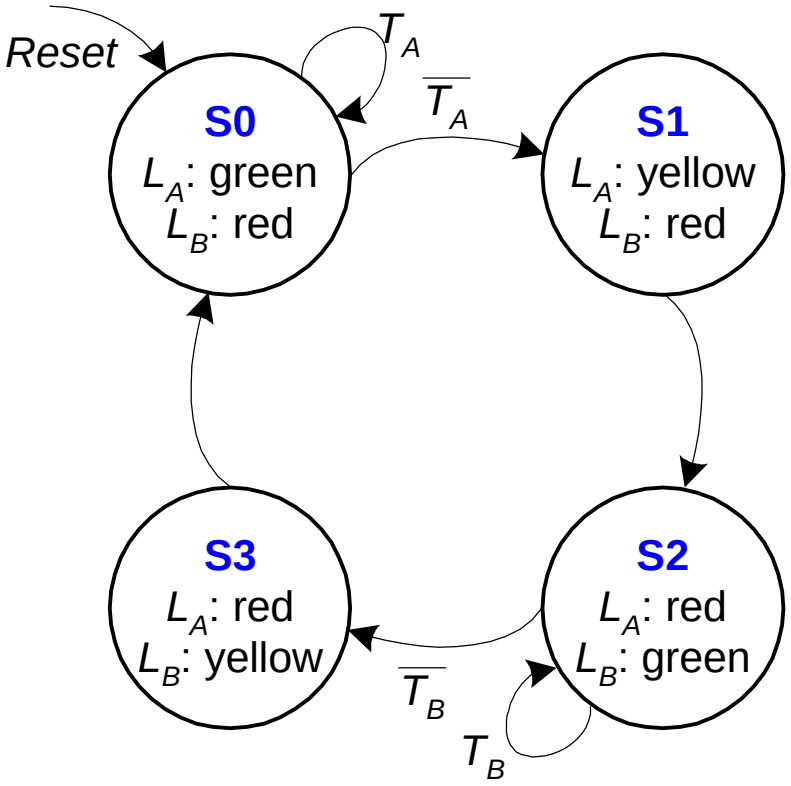




Finite State Machine Transition Diagram

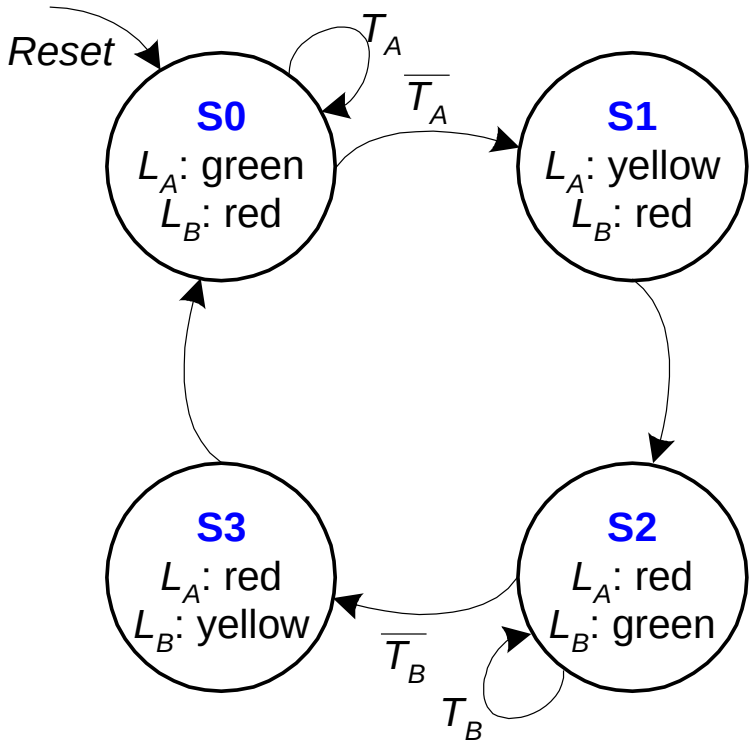
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs





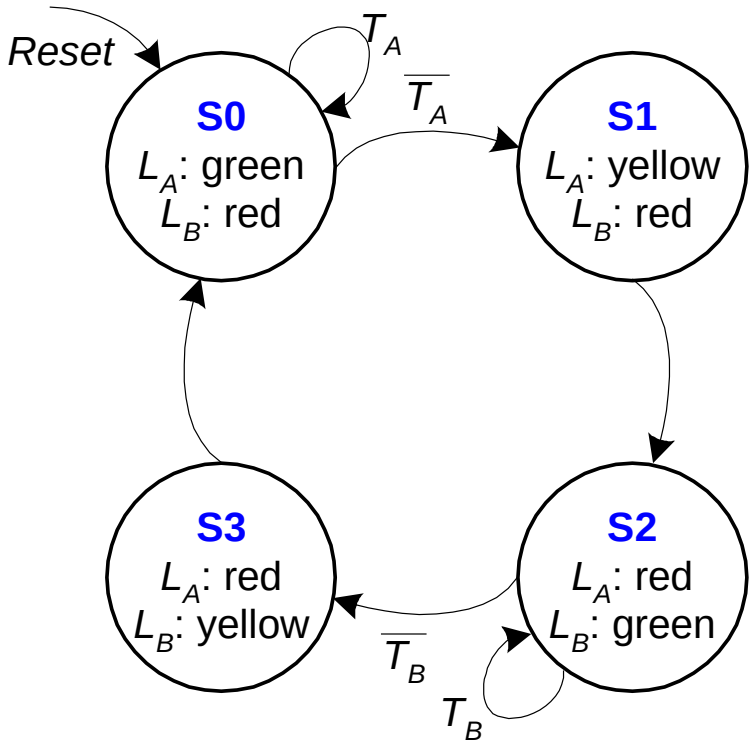
Finite State Machine: State Transition Table

FSM State Transition Table



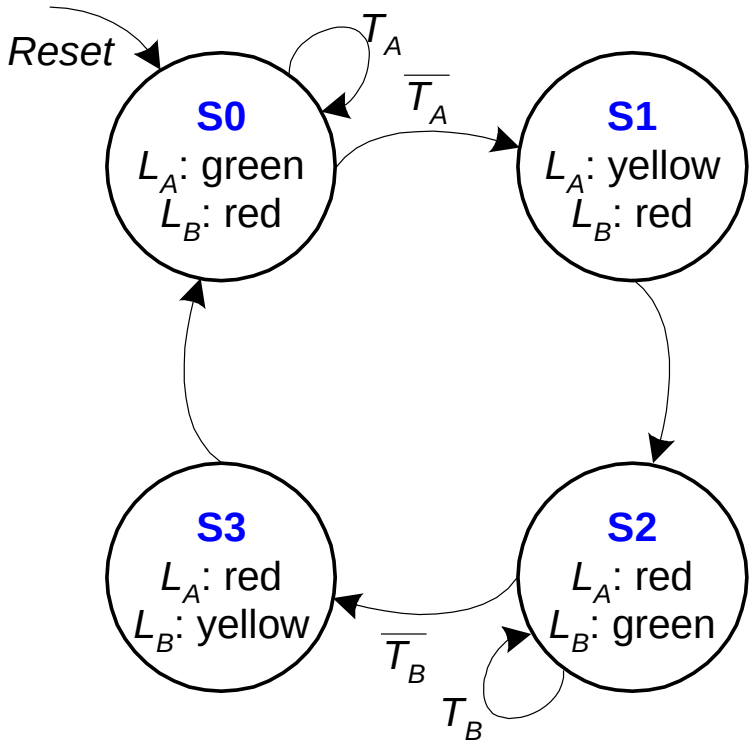
Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	
S0	1	X	
S1	X	X	
S2	X	0	
S2	X	1	
S3	X	X	

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

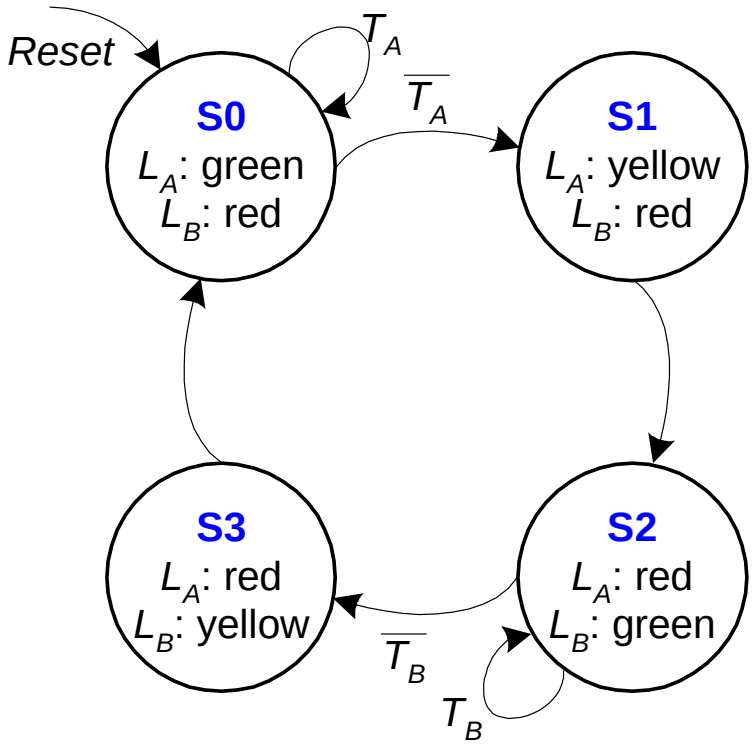
FSM State Transition Table



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

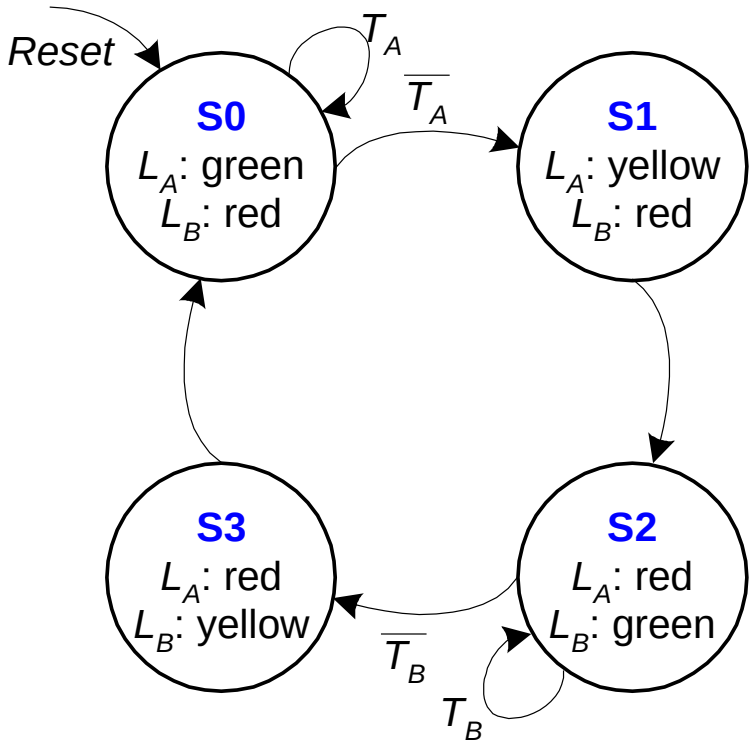
FSM State Transition Table



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

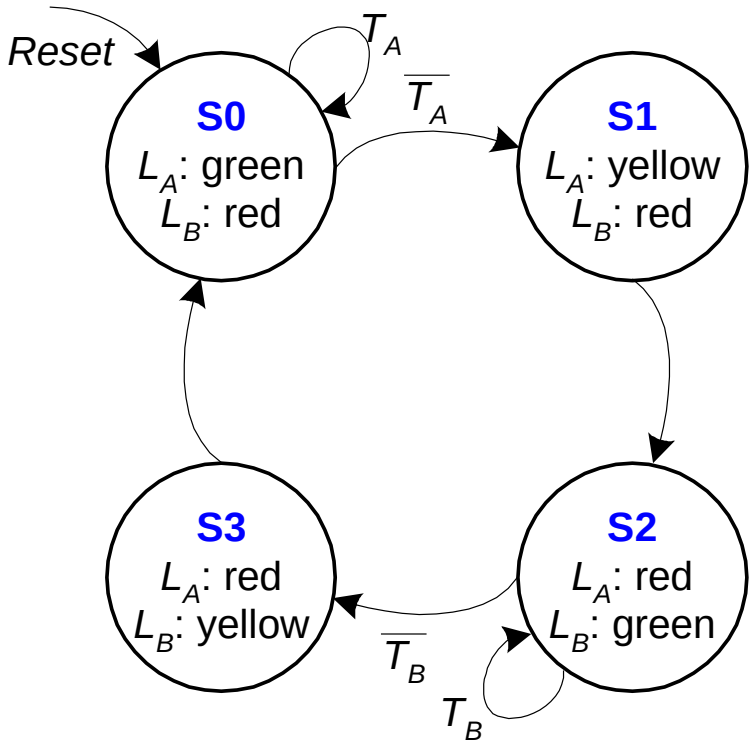


Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$S'_1 = ?$

FSM State Transition Table

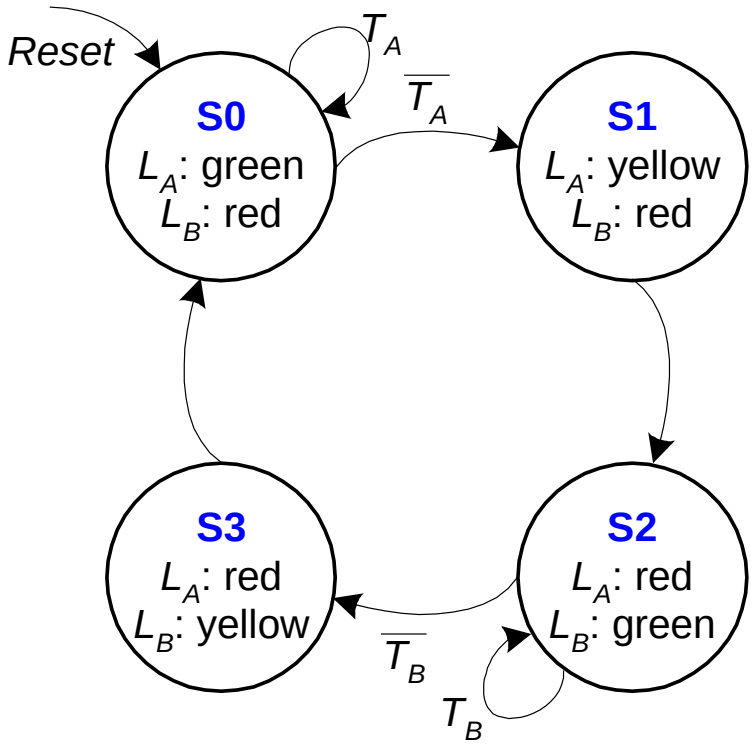


Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

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



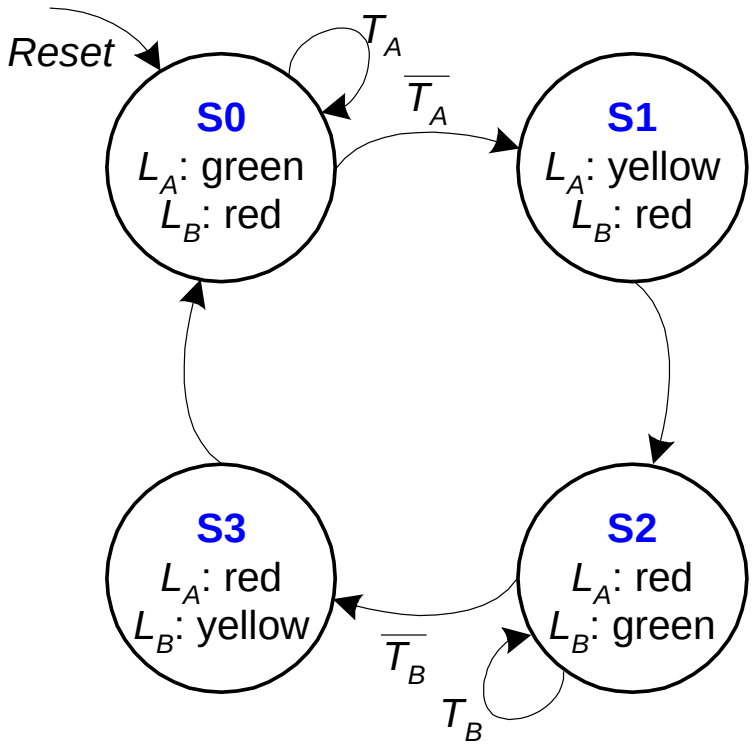
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

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



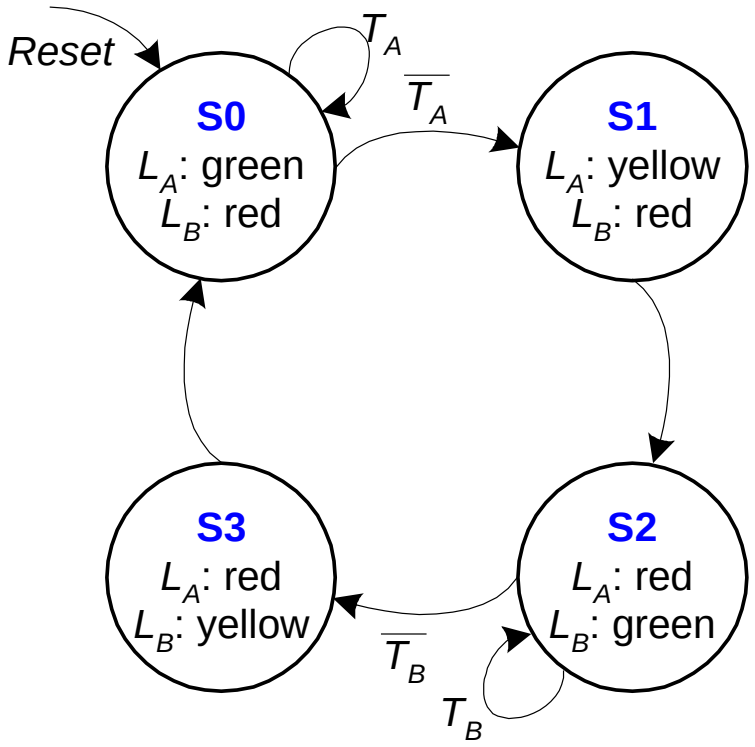
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

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

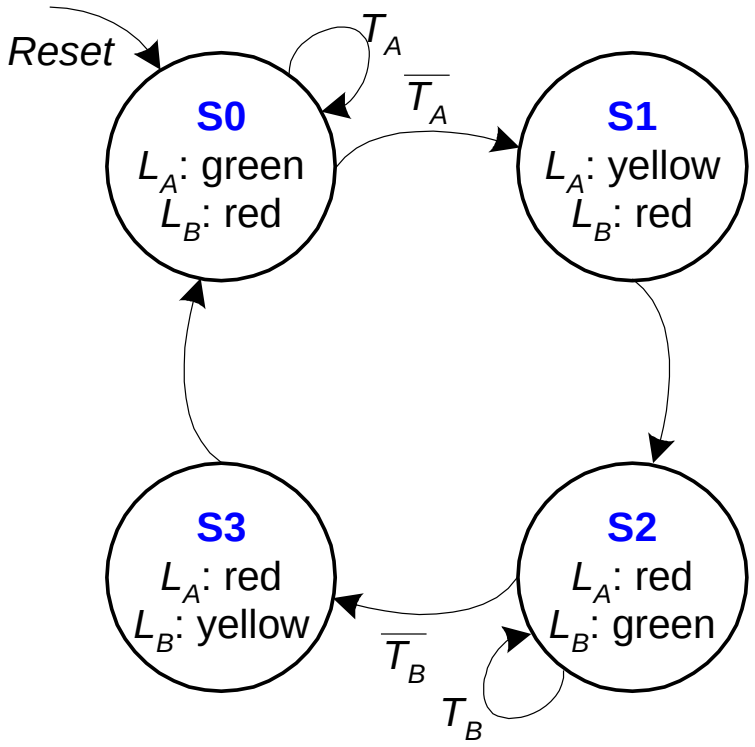
State	Encoding
S0	00
S1	01
S2	10
S3	11

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

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

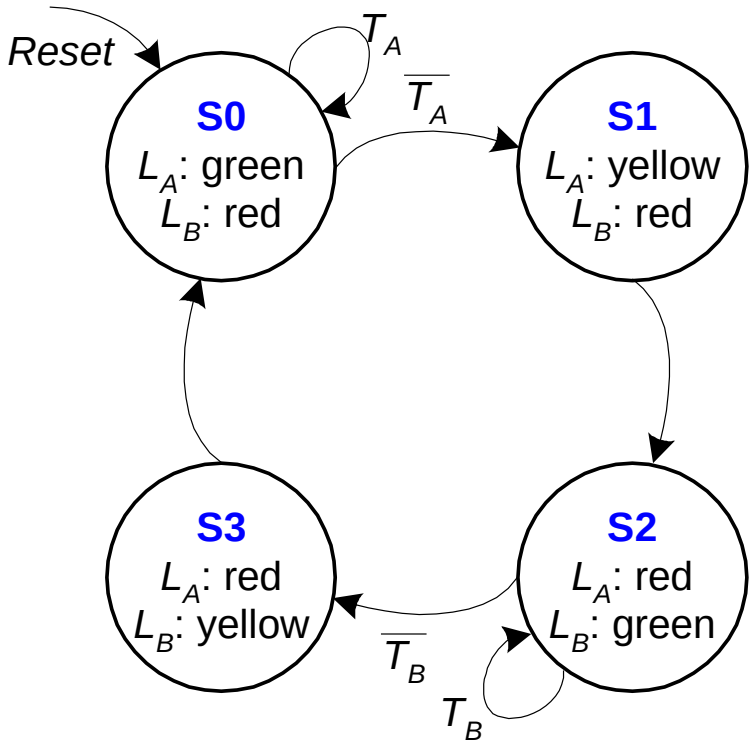
Finite State Machine: Output Table

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

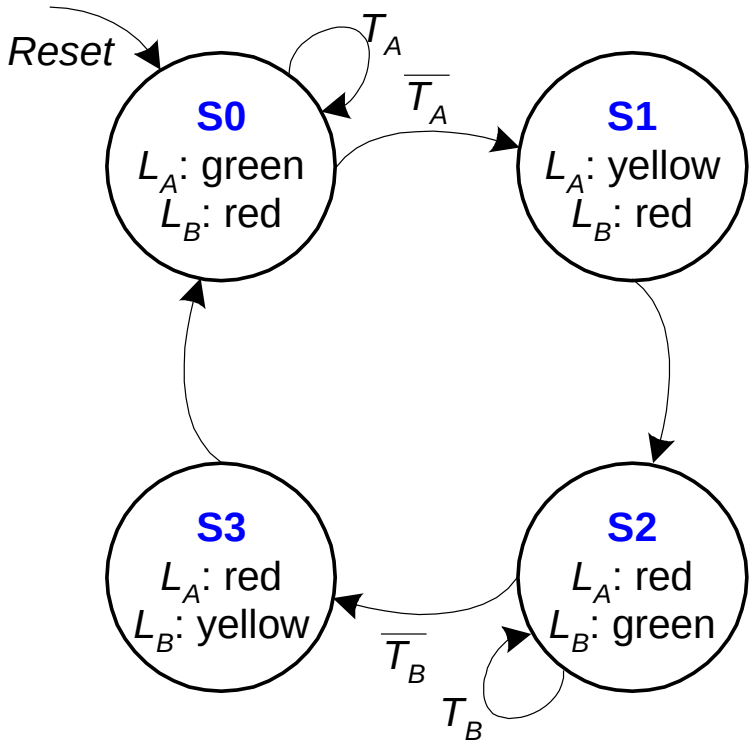
FSM Output Table



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

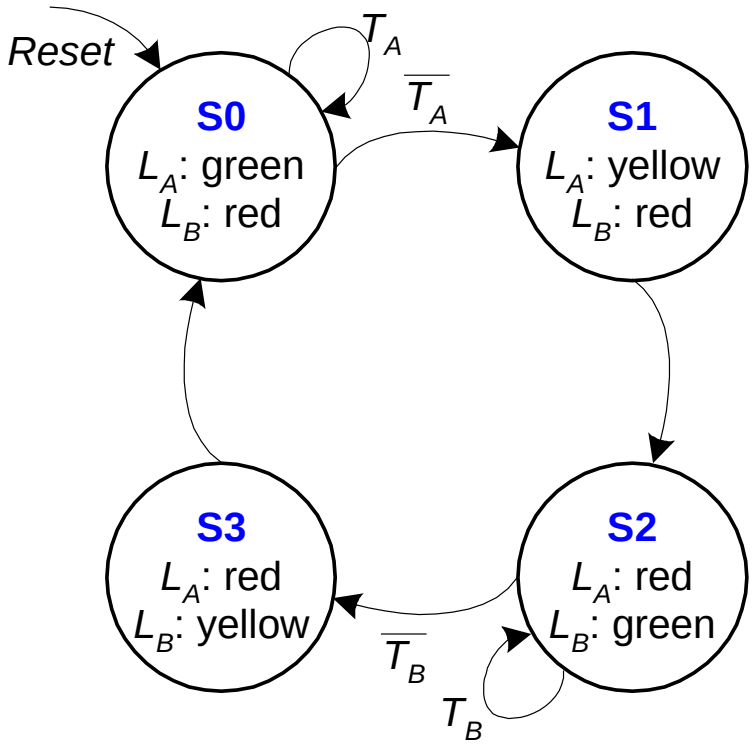


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$L_{A1} = S_1$

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table

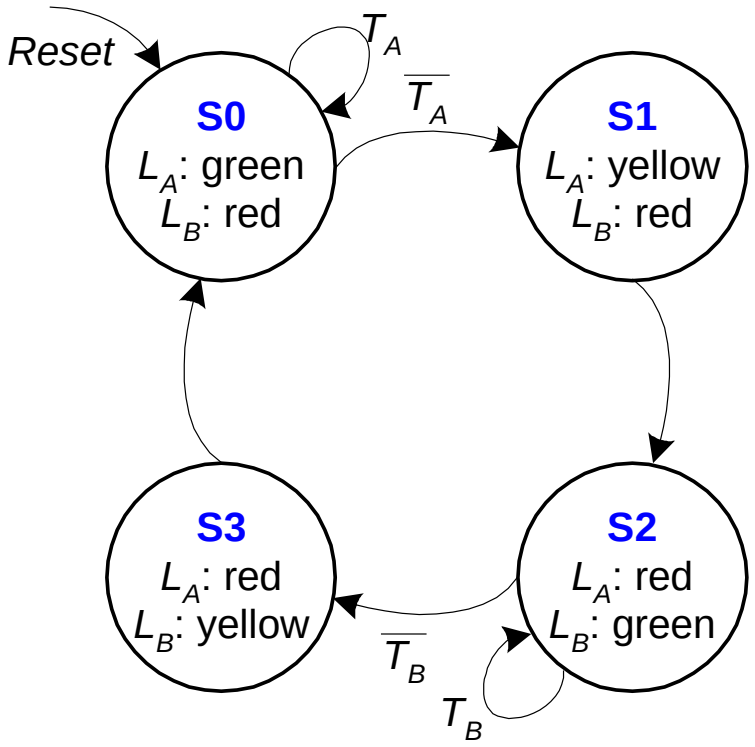


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

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

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table

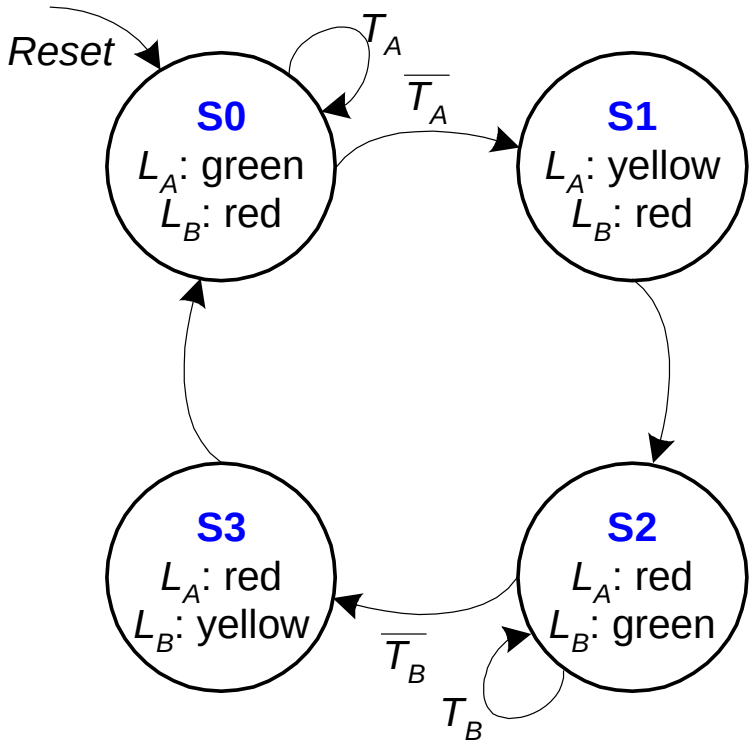


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

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

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table



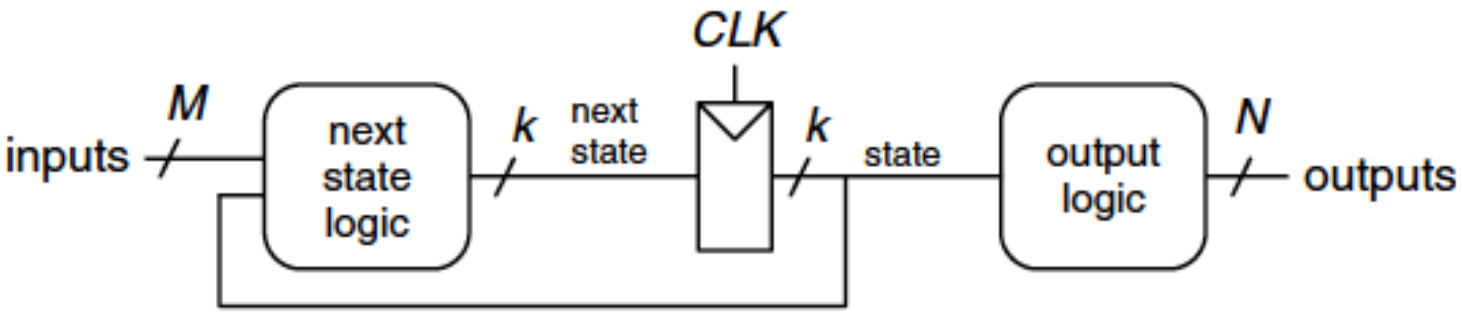
Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

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

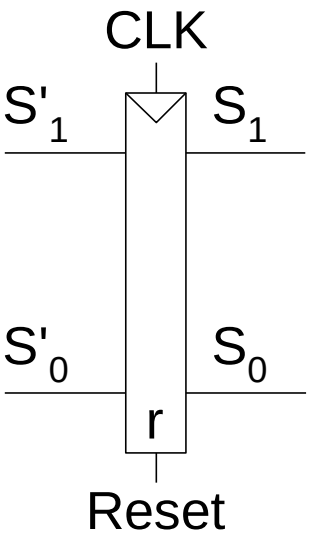
Output	Encoding
green	00
yellow	01
red	10

Finite State Machine: Schematic

FSM Schematic: State Register

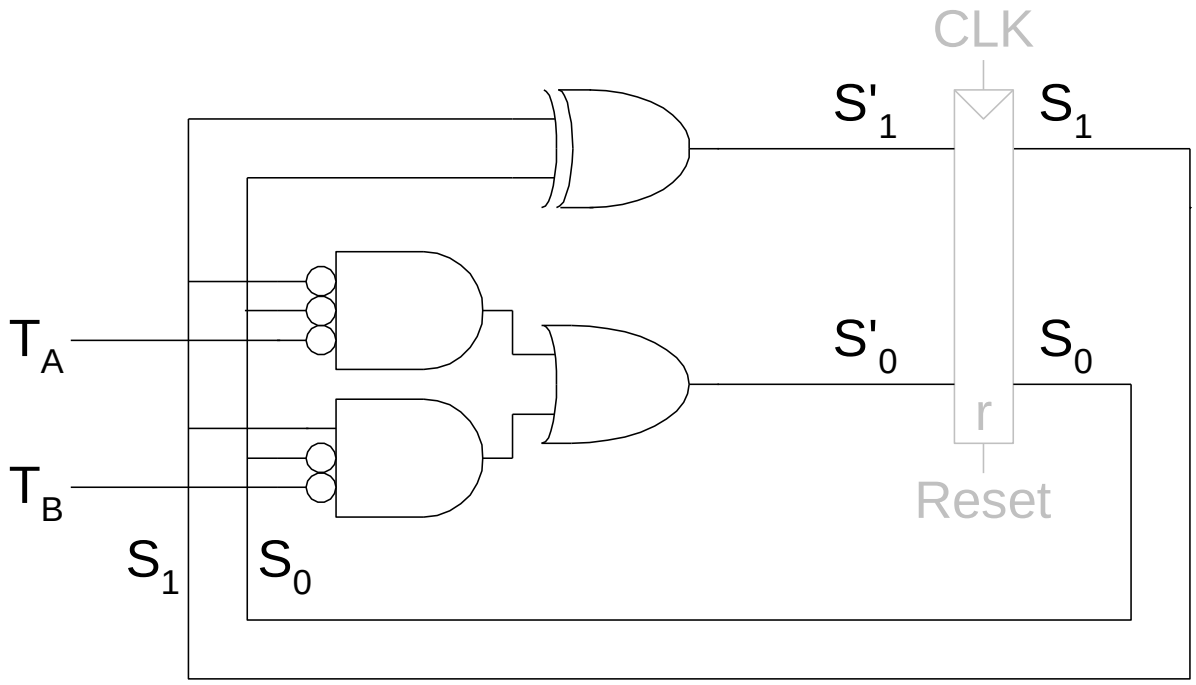


FSM Schematic: State Register



state register

FSM Schematic: Next State Logic

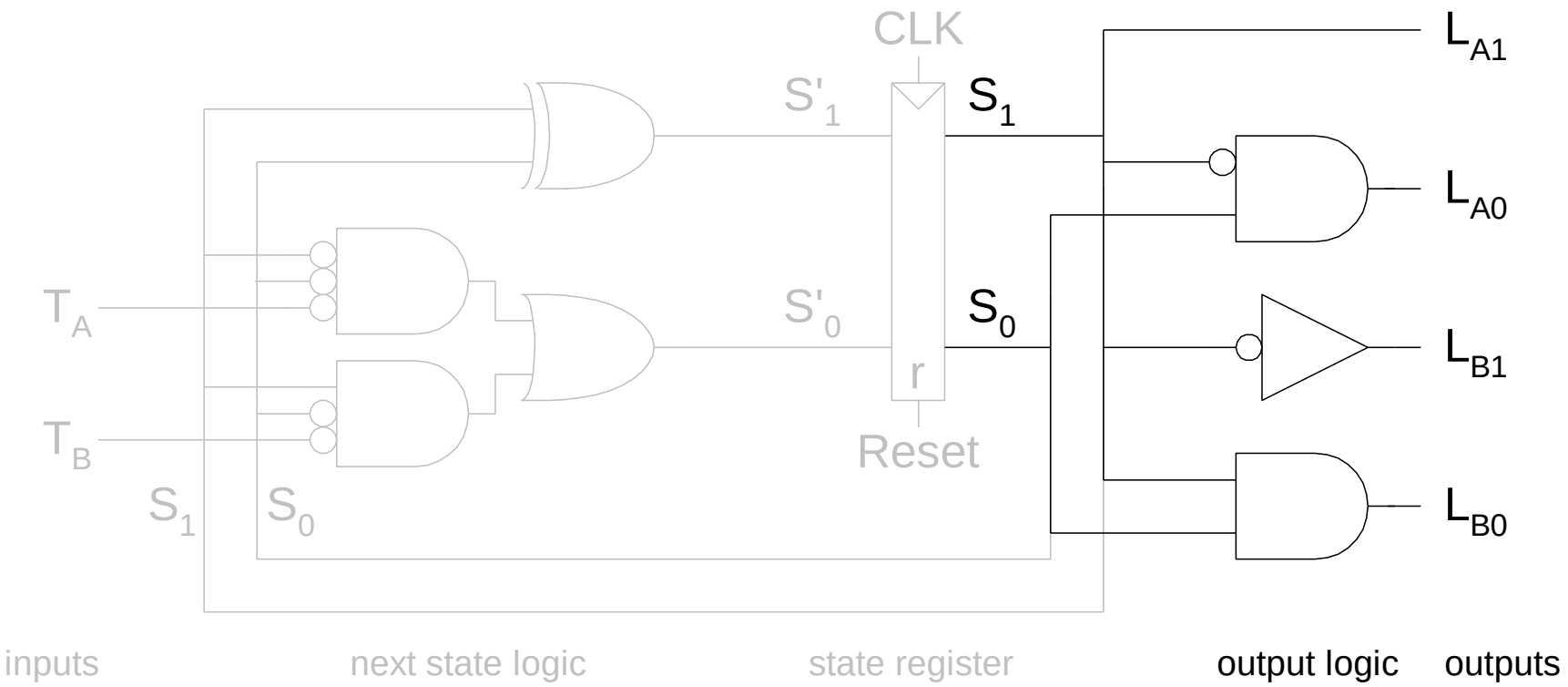


inputs next state logic state register

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

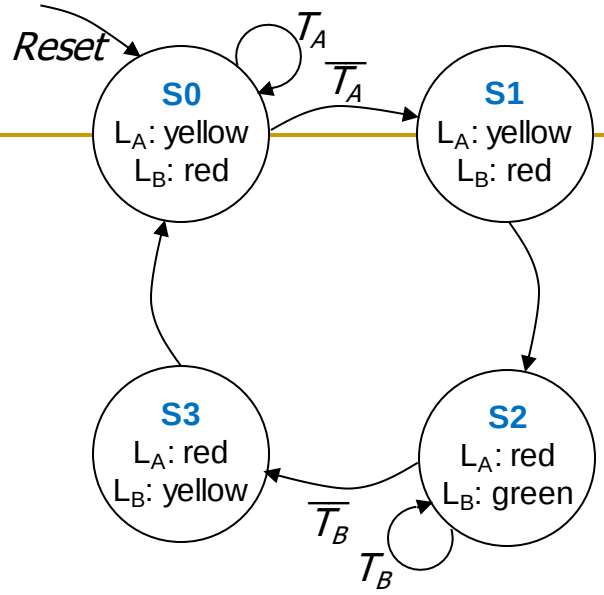
$$S'_0 = (\overline{S}_1 \cdot \overline{S}_0 \cdot \overline{T}_A) + (S_1 \cdot \overline{S}_0 \cdot \overline{T}_B)$$

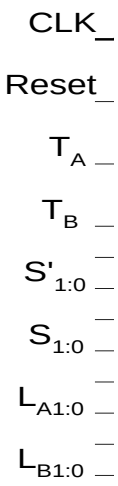
FSM Schematic: Output Logic



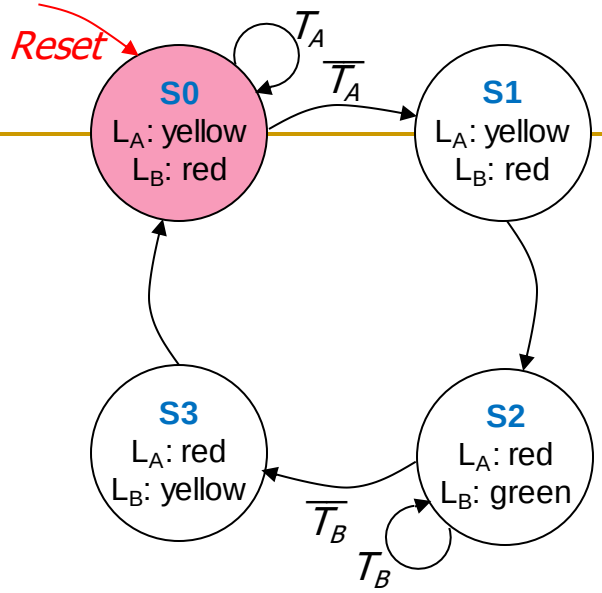
$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} \cdot S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 \cdot S_0$$

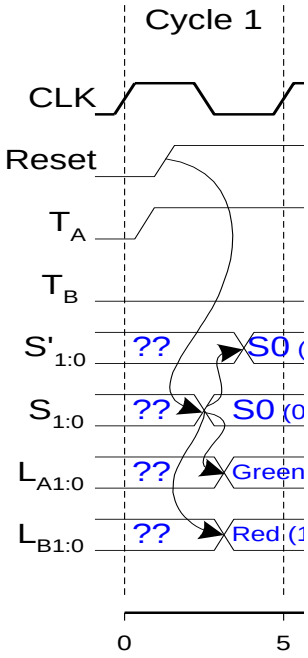
FSM Timing Diagram



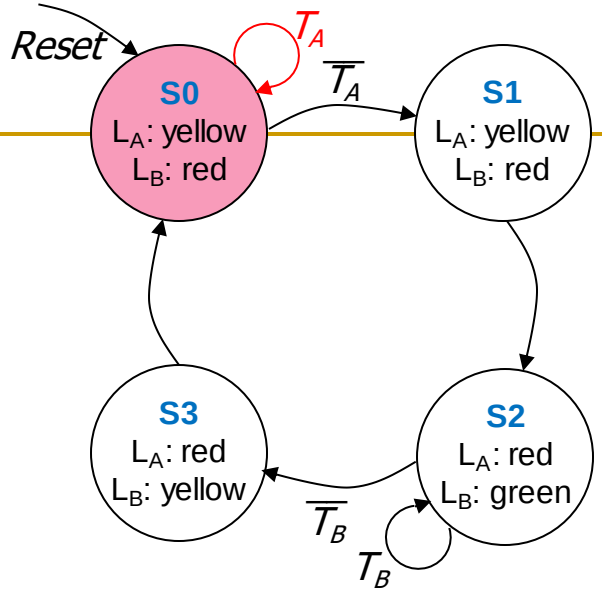


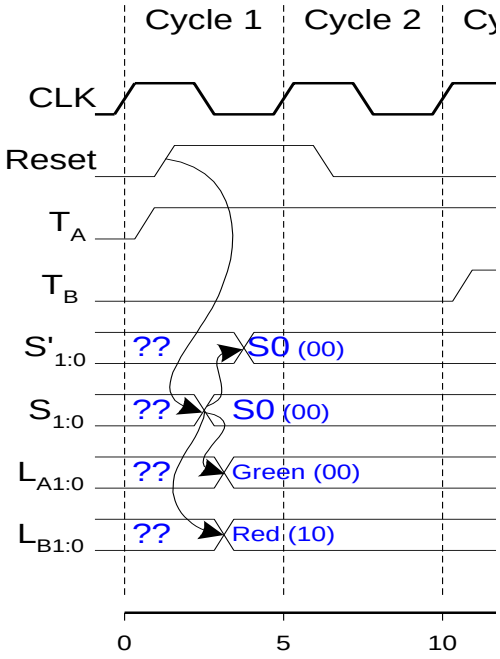
FSM Timing Diagram



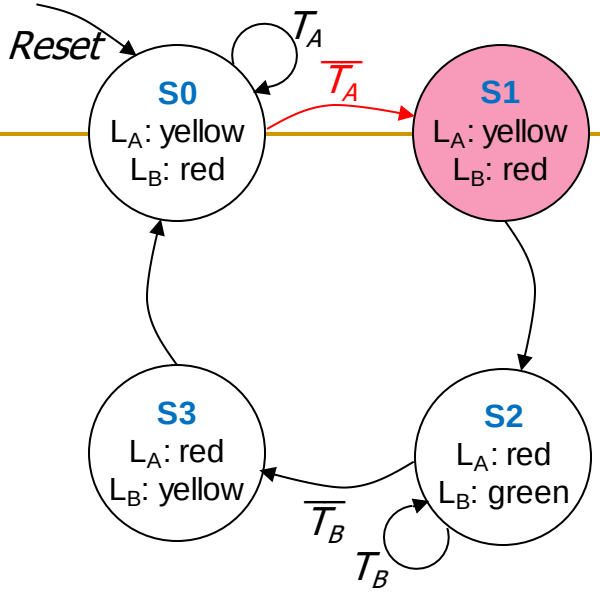


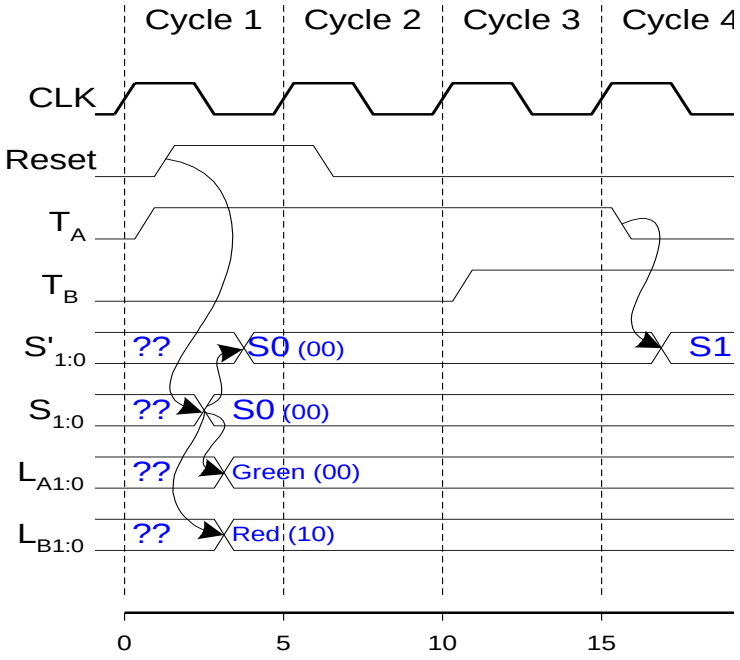
FSM Timing Diagram



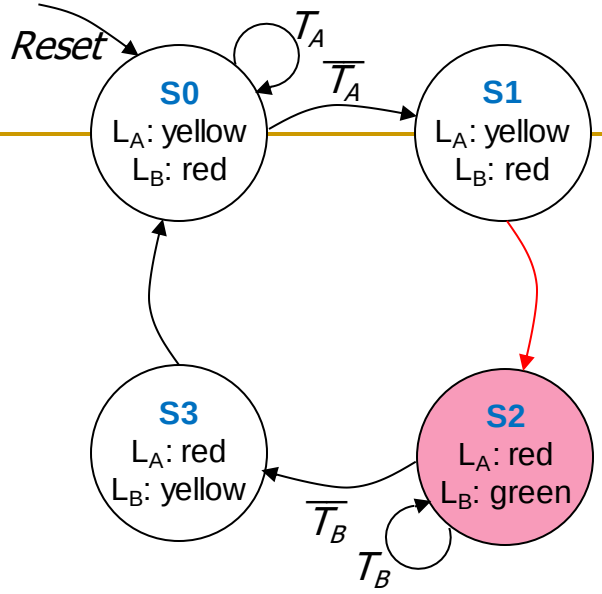


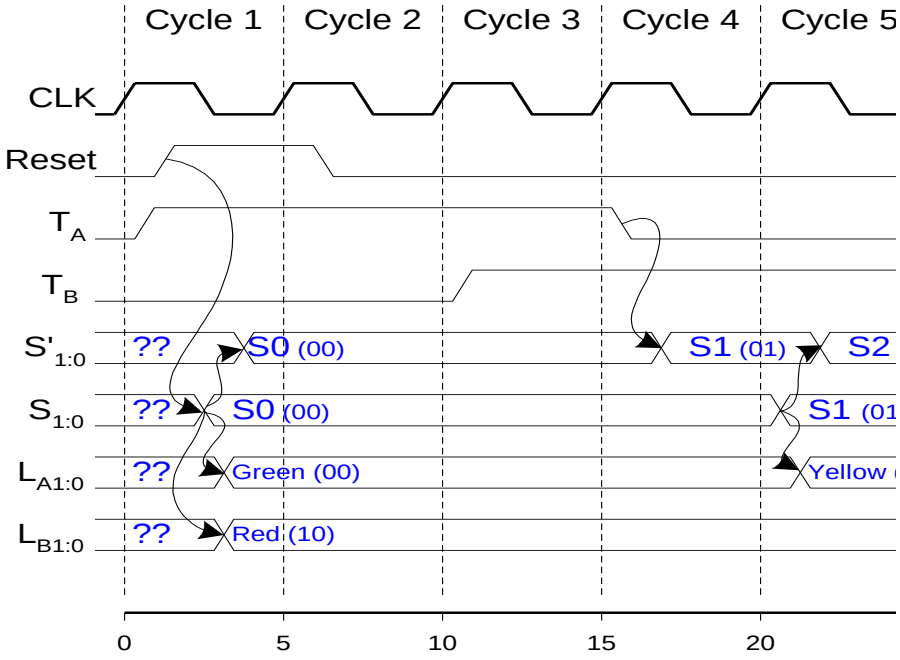
FSM Timing Diagram



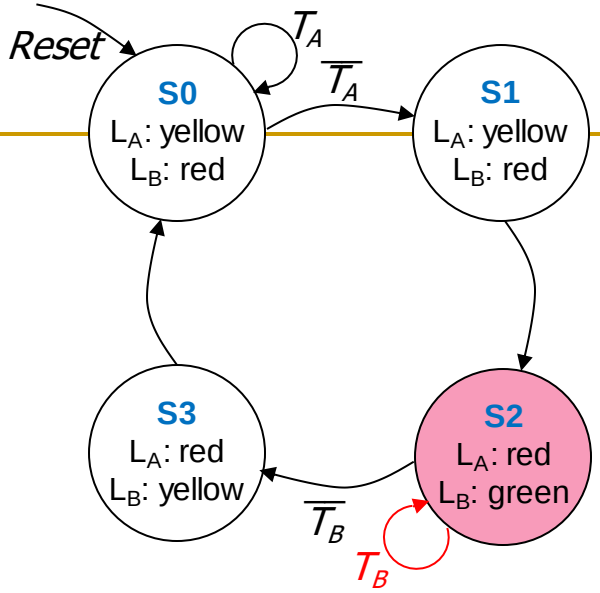


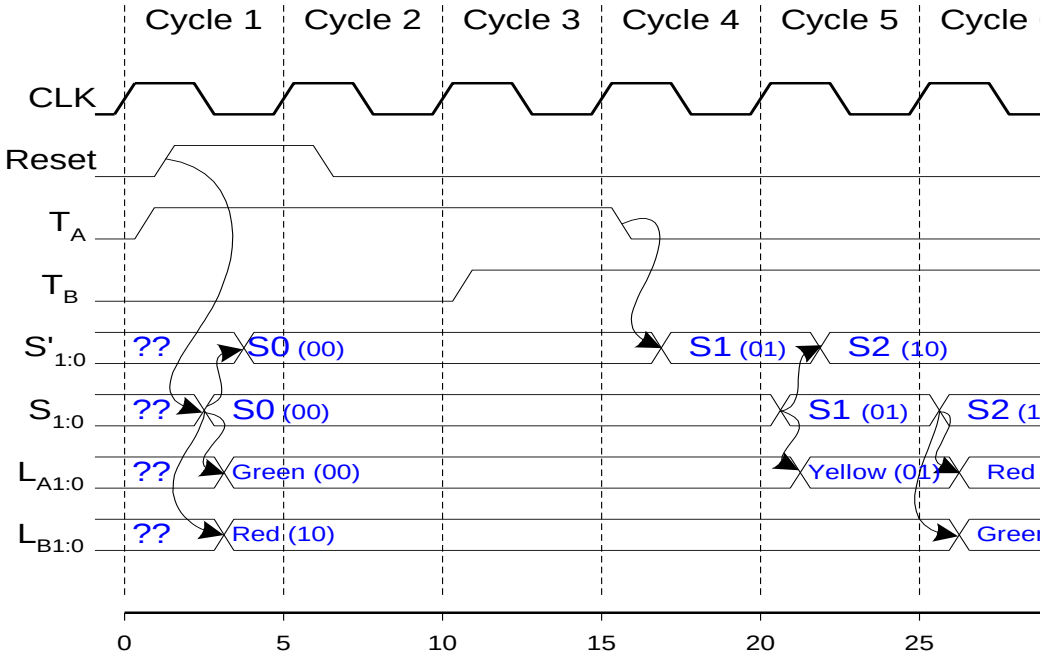
FSM Timing Diagram



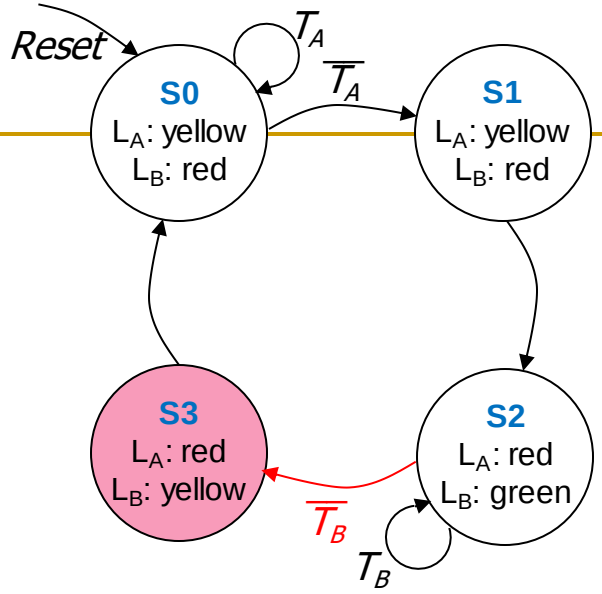


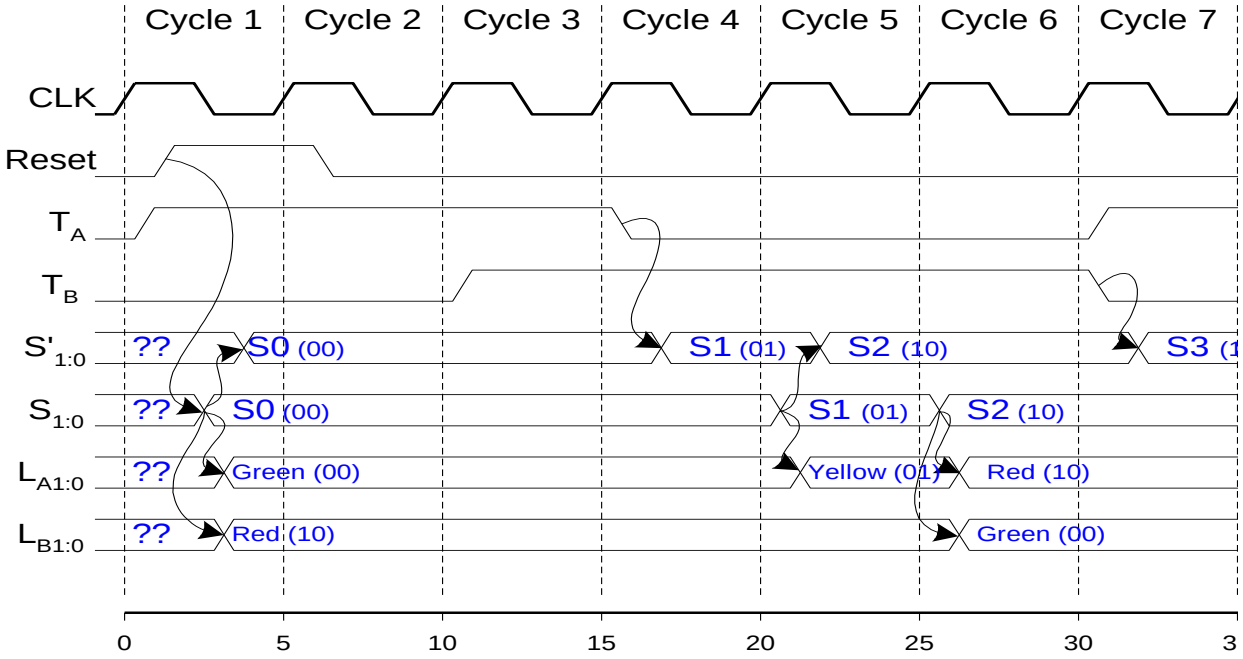
FSM Timing Diagram



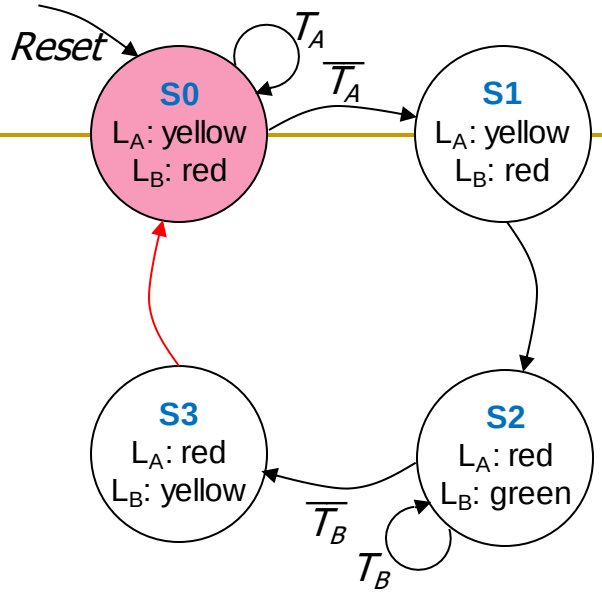


FSM Timing Diagram

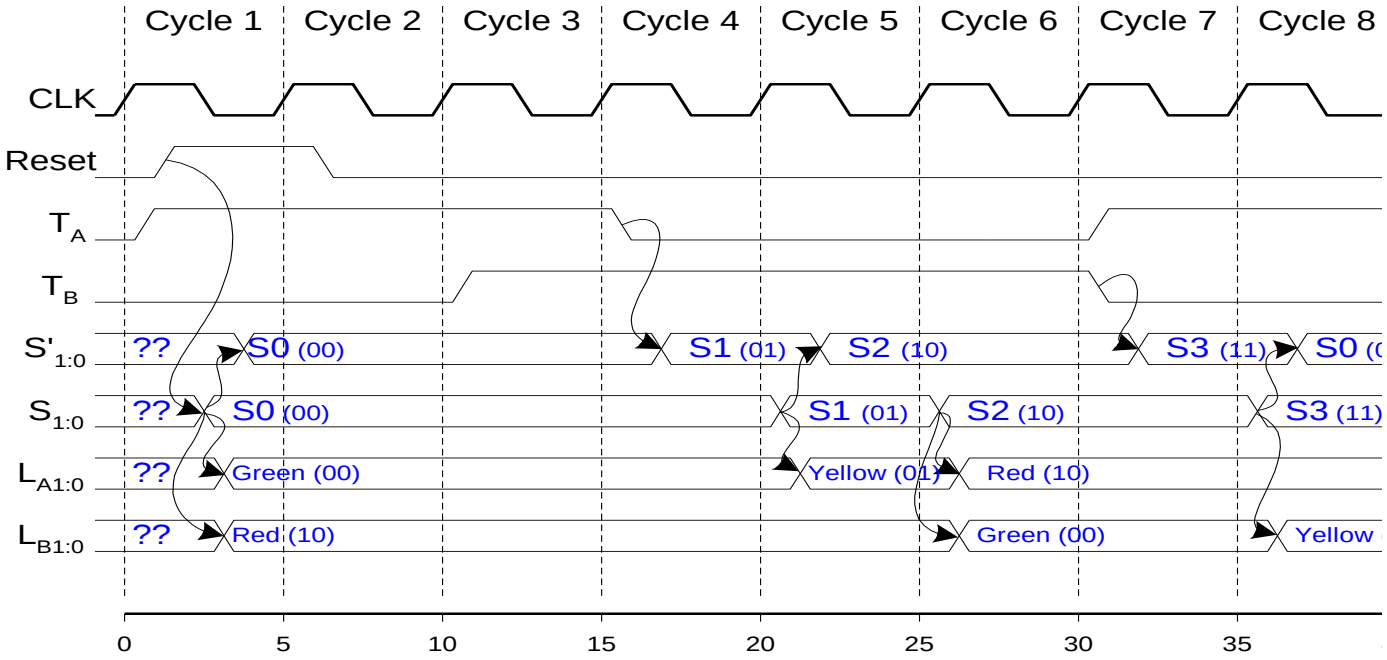




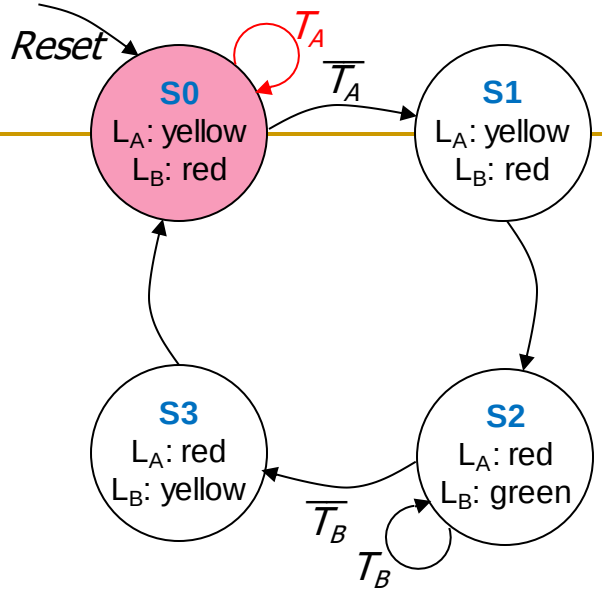
FSM Timing Diagram

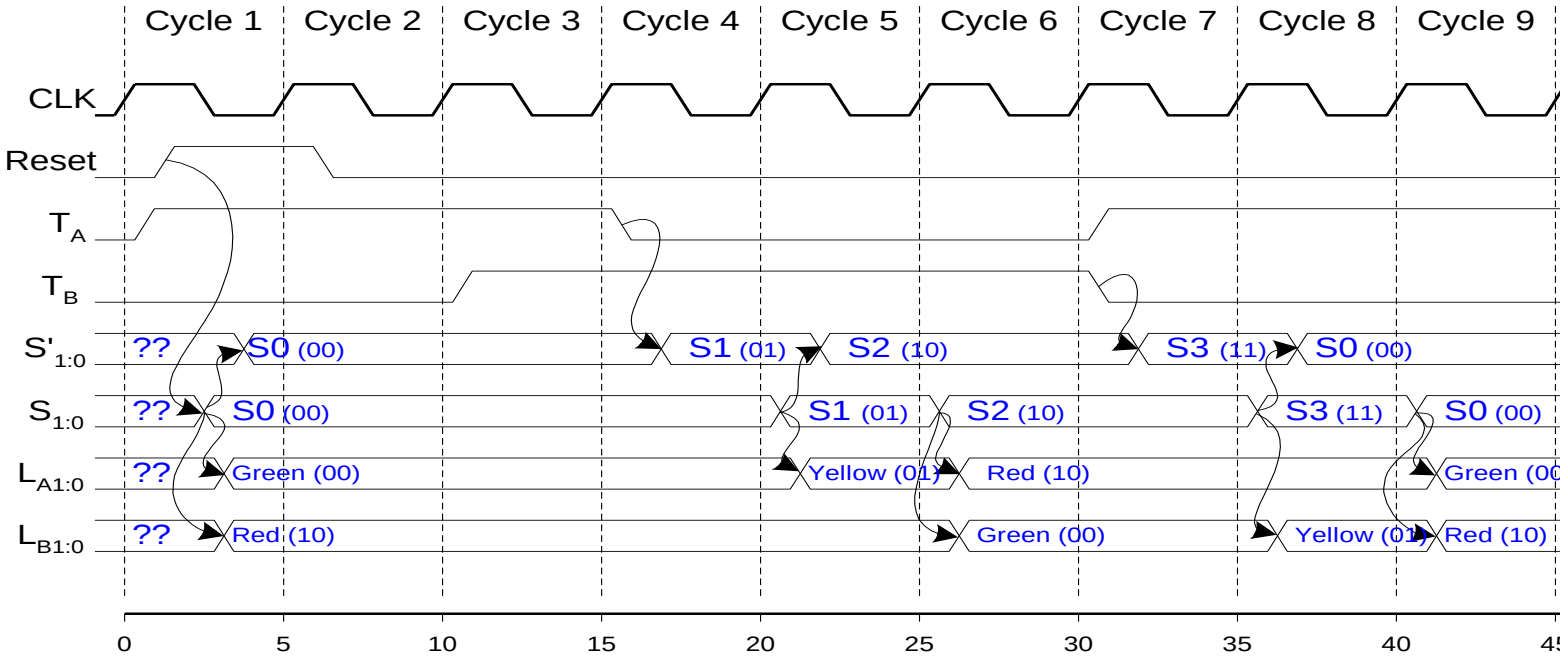


This is from H&H Section 3.4.1

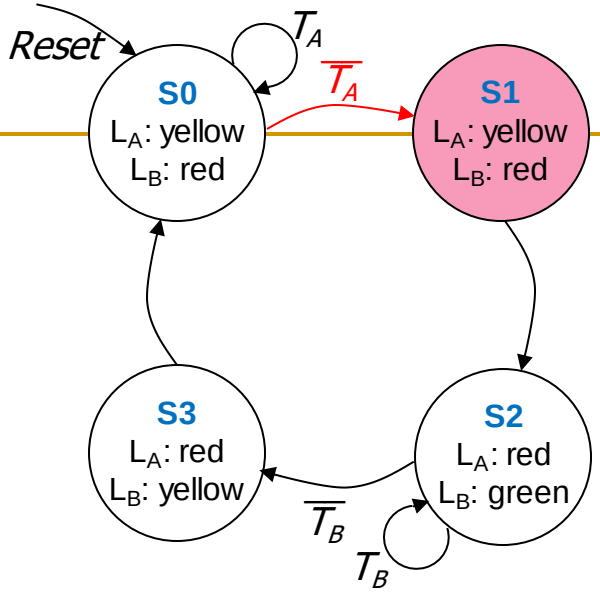


FSM Timing Diagram

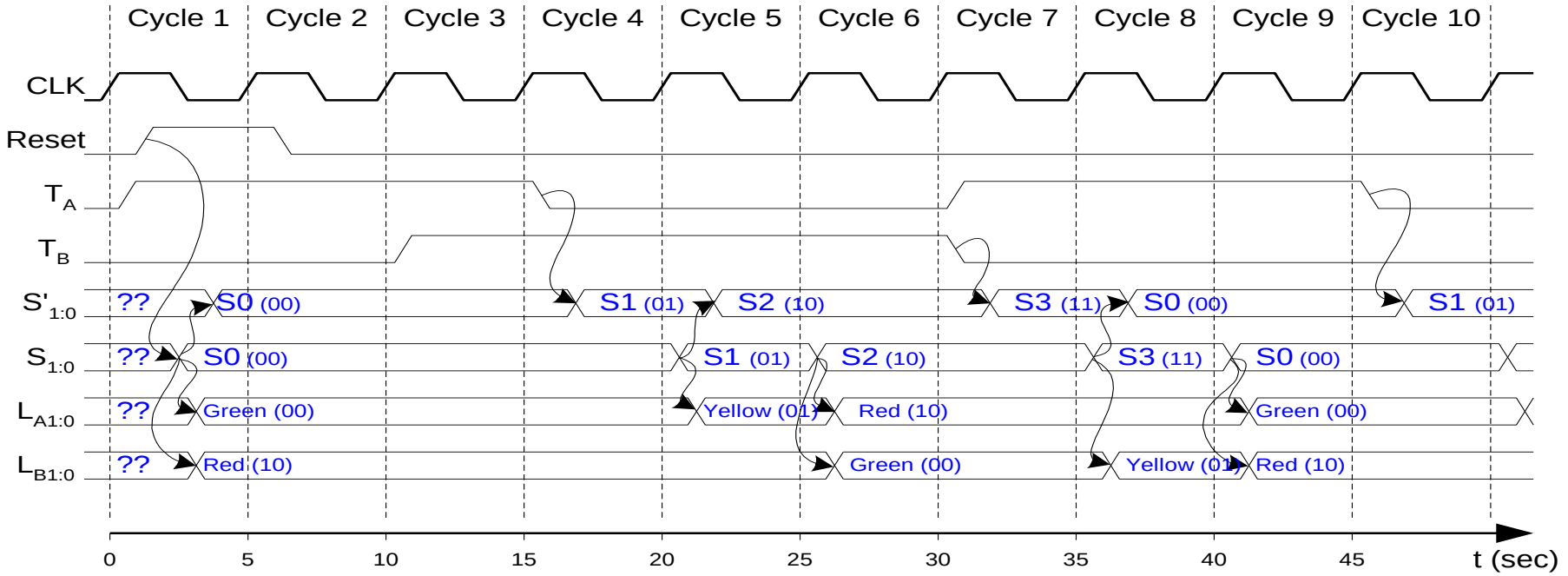




FSM Timing Diagram



See H&H Chapter 3.4

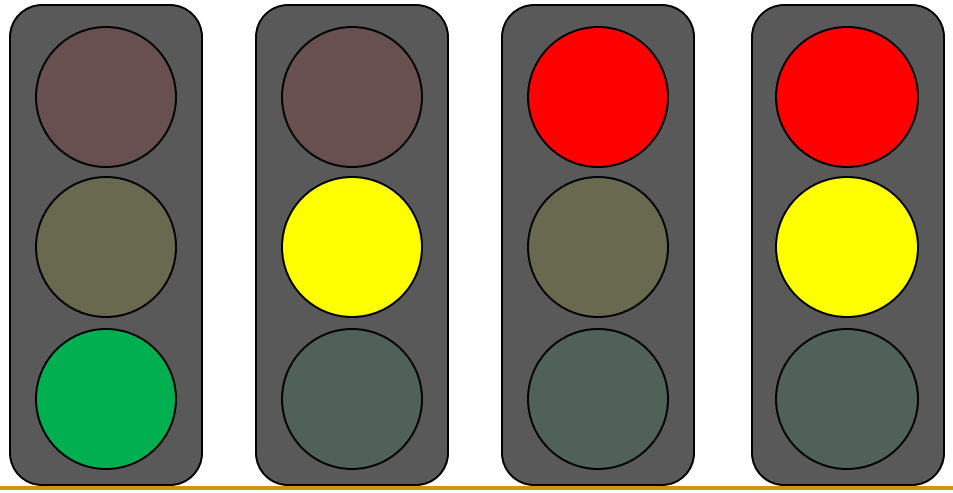


Finite State Machine: State Encoding

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 number of bits used to encode all states
     - Use $log_2(num\_states)$ bits to represent the states
   - *Example states:* 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 states:* 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

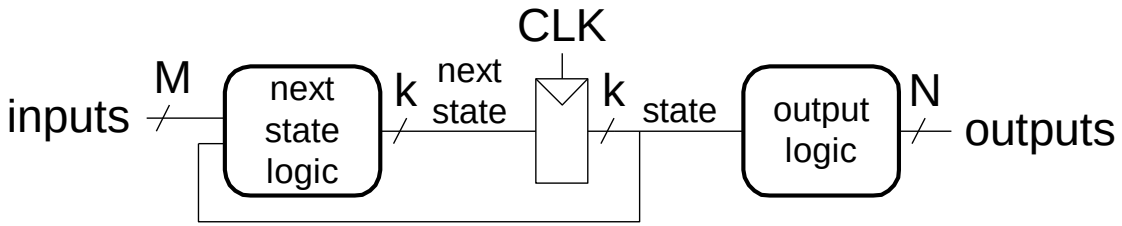
   - **Minimizes** output logic
   - Only works for Moore Machines (output function of state)

Moore vs. Mealy Machines

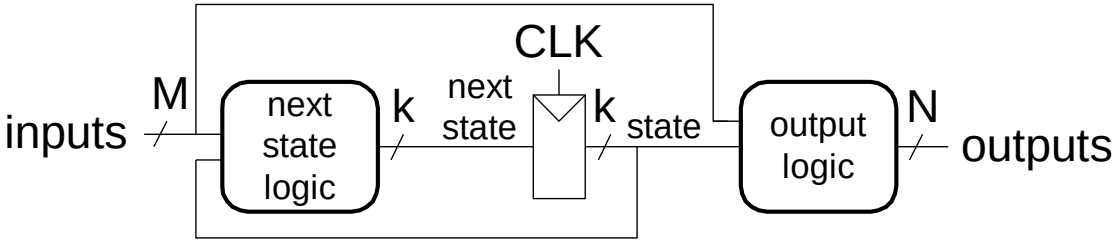
Recall: Moore vs. Mealy FSMs

- 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

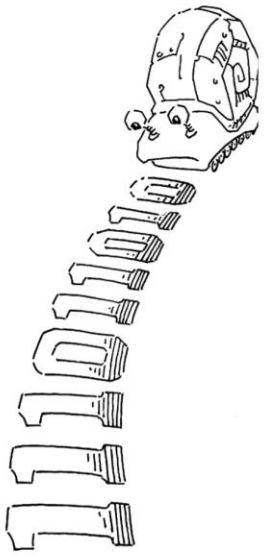


Mealy FSM

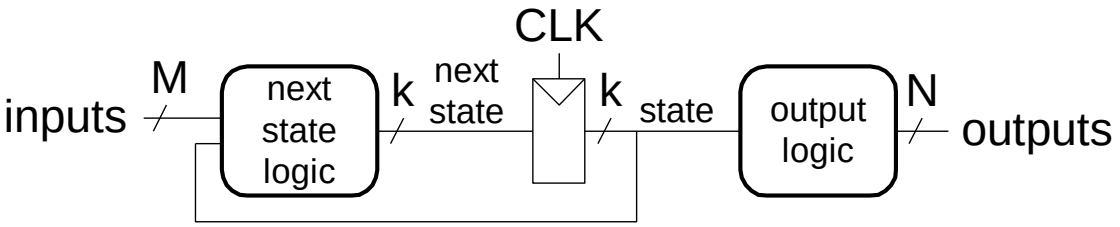


Moore vs. Mealy FSM Examples

- 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.

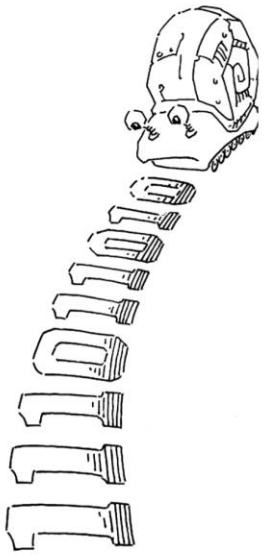


Moore FSM

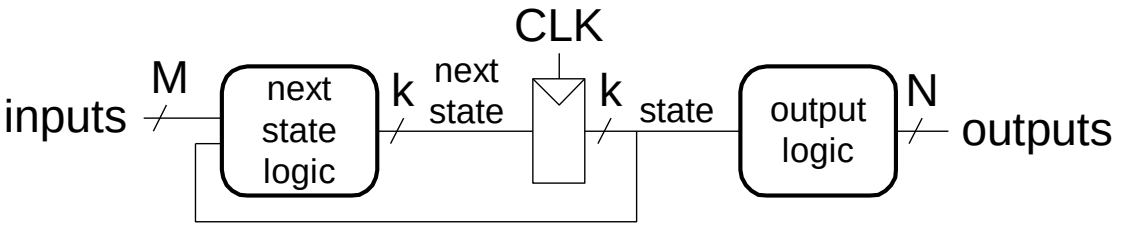


Moore vs. Mealy FSM Examples

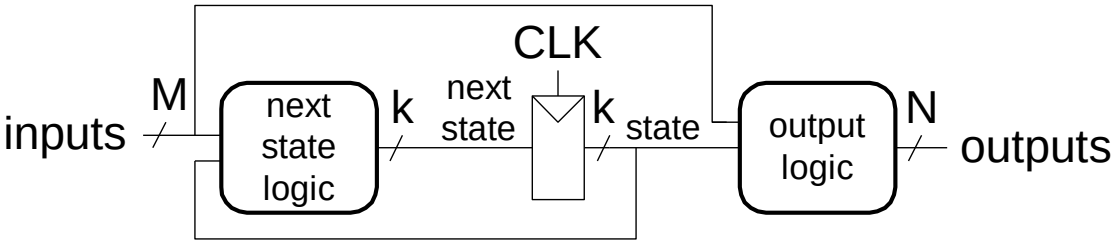
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are 1101.
- Design Moore and Mealy FSMs of the snail's brain.



Moore FSM

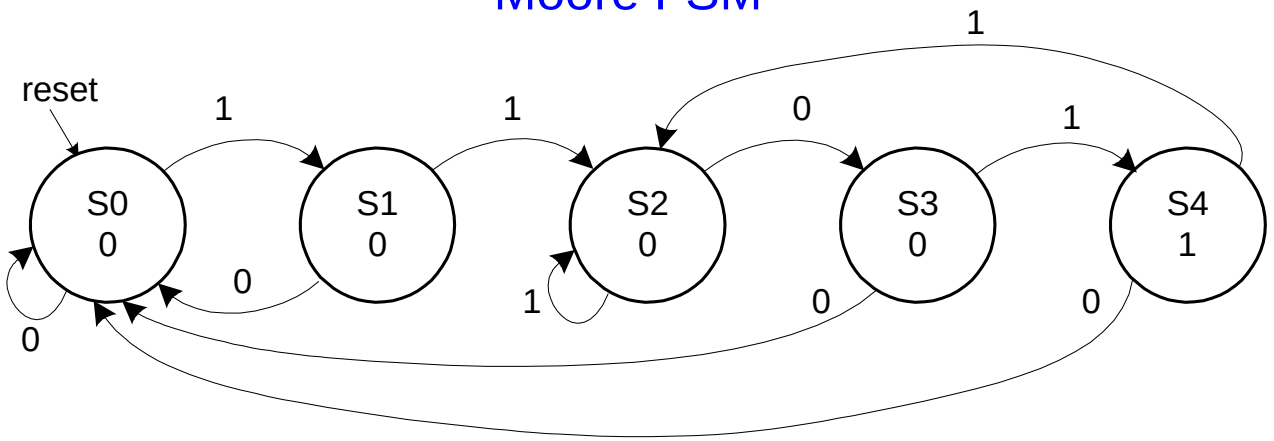


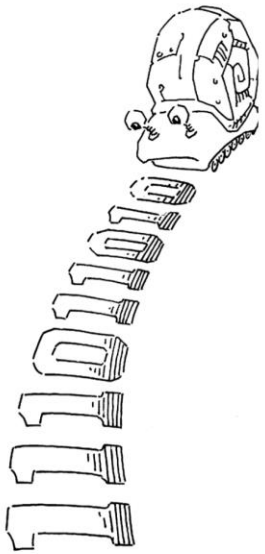
Mealy FSM



State Transition Diagrams

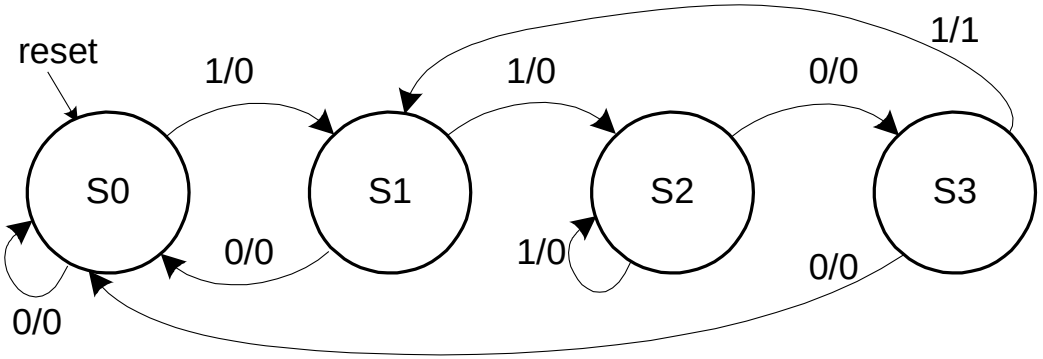
Moore FSM





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

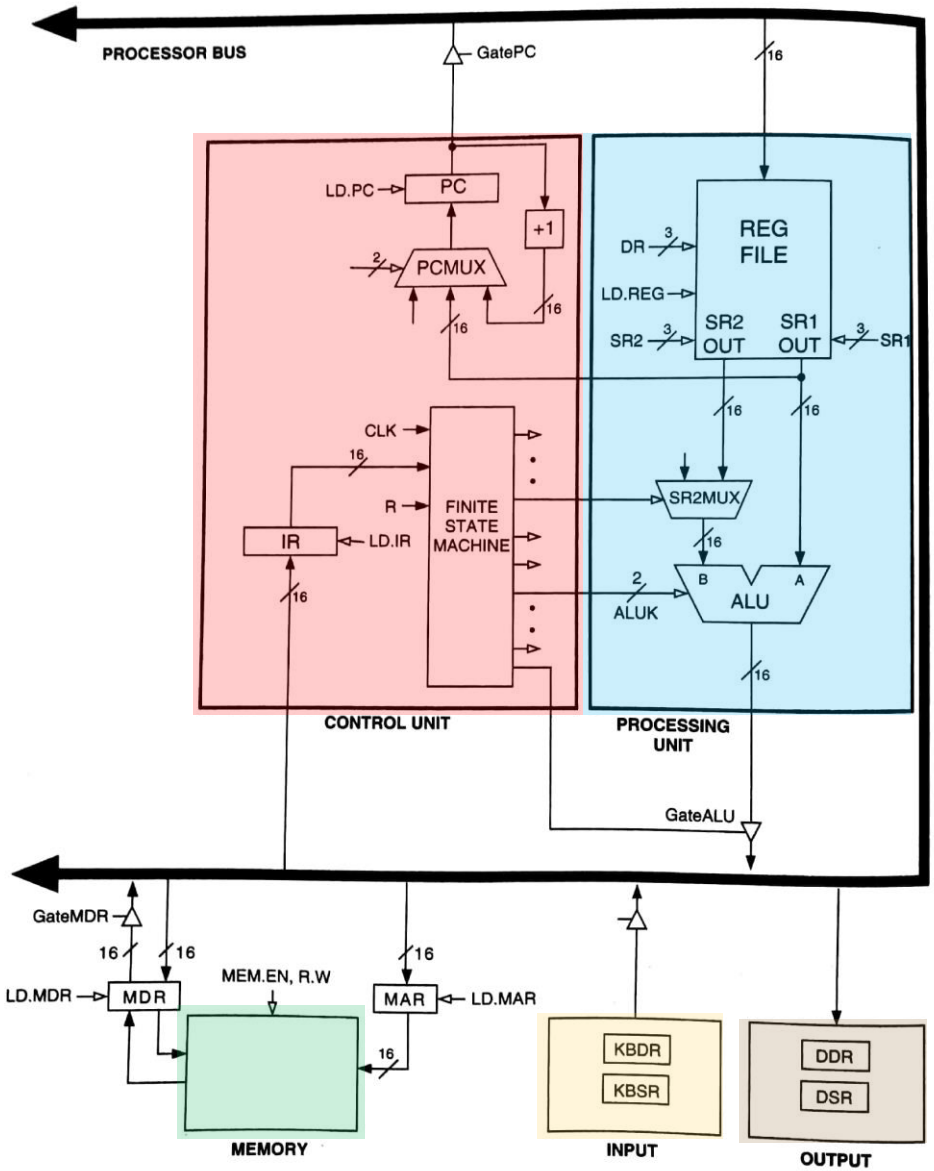
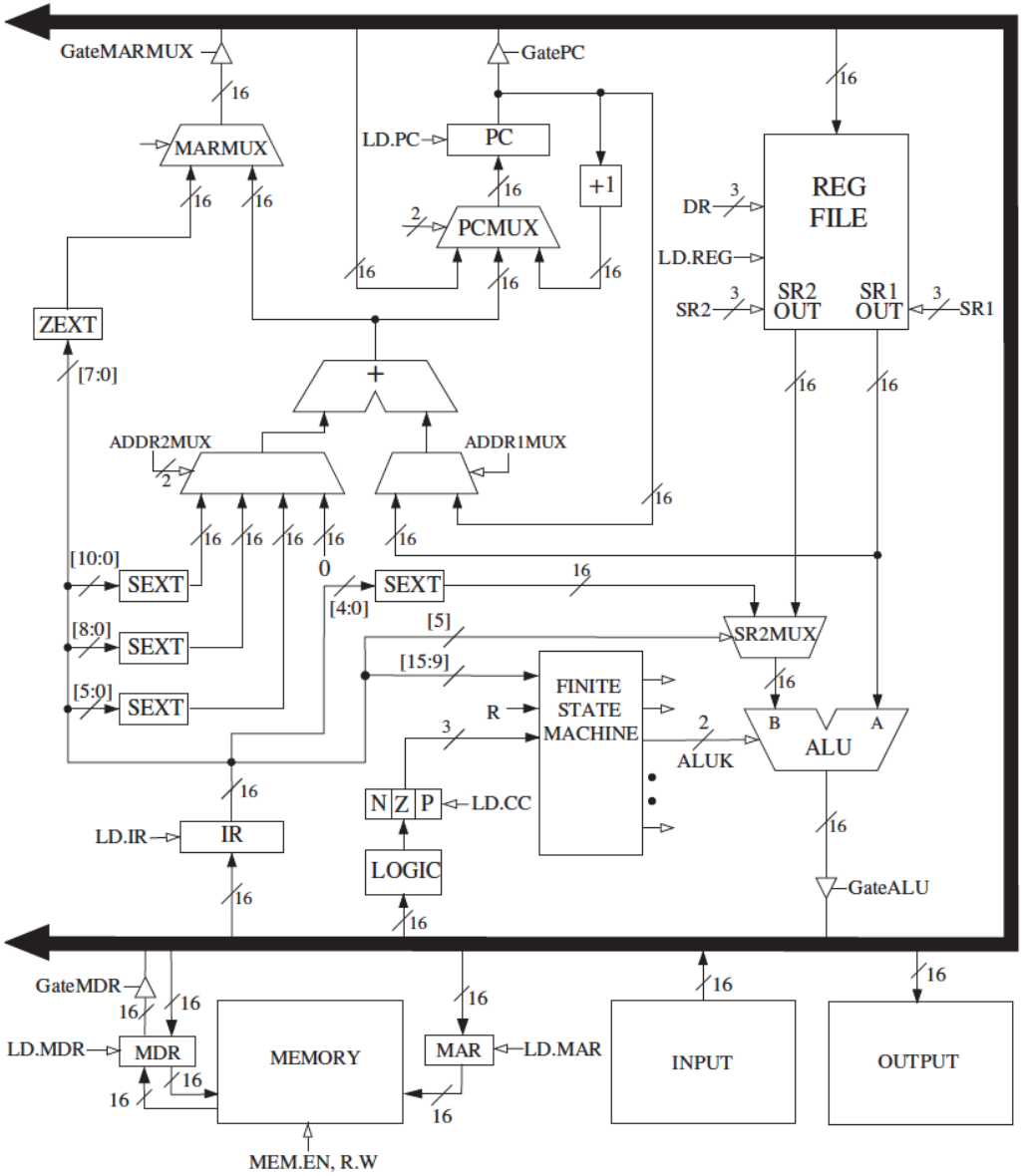


Figure 4.3 The LC-3 as an example of the von Neumann model

What is to Come: LC-3 Datapath



Digital Design & Computer Arch.

Lecture 6: Sequential Logic Design

Prof. Onur Mutlu

ETH Zürich

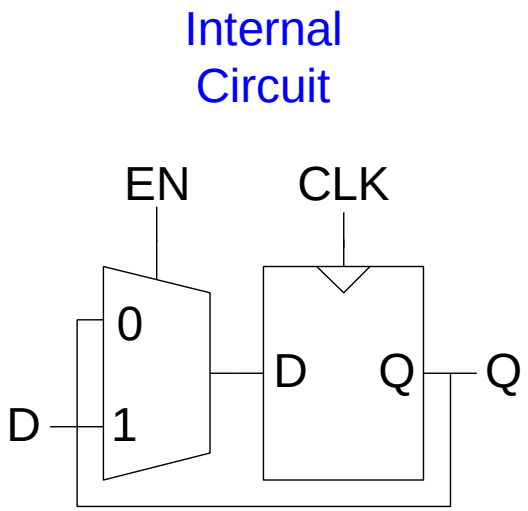
Spring 2021

12 March 2021

Backup Slides: Different Types of Flip Flops

Enabled Flip-Flops

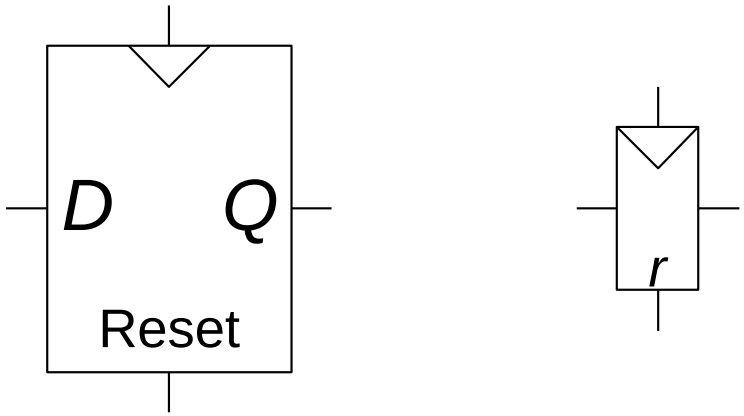
- **Inputs:** CLK, D, EN
  - The enable input (EN) controls when new data (D) is stored
- **Function:**
  - **EN = 1**: D passes through to Q on the clock edge
  - **EN = 0**: the flip-flop retains its previous state



Resettable Flip-Flop

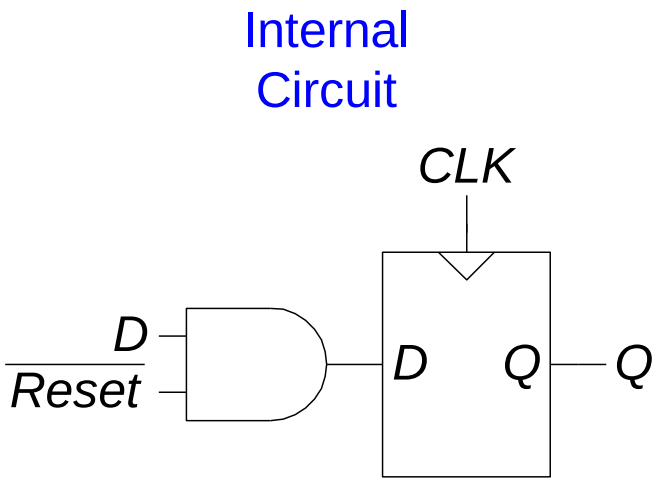
- **Inputs:** CLK, D, Reset
  - The Reset is used to set the output to 0.
- **Function:**
  - ***Reset = 1:*** Q is forced to 0
  - ***Reset = 0:*** the flip-flop behaves like an ordinary D flip-flop

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