

Digital Design & Computer Arch.

Lecture 6: Sequential Logic Design

Prof. Onur Mutlu

ETH Zürich

Spring 2020

6 March 2020

We Are Almost 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

- Wrap up Combinational Logic and Circuit Minimization
- 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 Architectures**
- **Required Assignment**
 - **Watch** Prof. Mutlu's inaugural lecture at ETH and understand it
 - <https://www.youtube.com/watch?v=kgiZISOcGFM>
- **Optional Assignment – for 1% extra credit**
 - **Write a 1-page summary** of the lecture and email us
 - What are your key takeaways?
 - What did you learn?
 - What did you like or dislike?
 - Submit your summary to [Moodle](#) – Deadline: March 25

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 1

- 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**

Wrap-Up Combinational Logic Circuits and Design

Recall: Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire

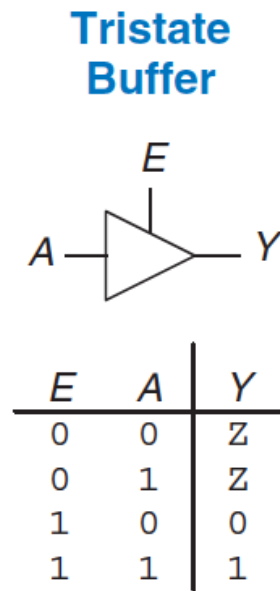


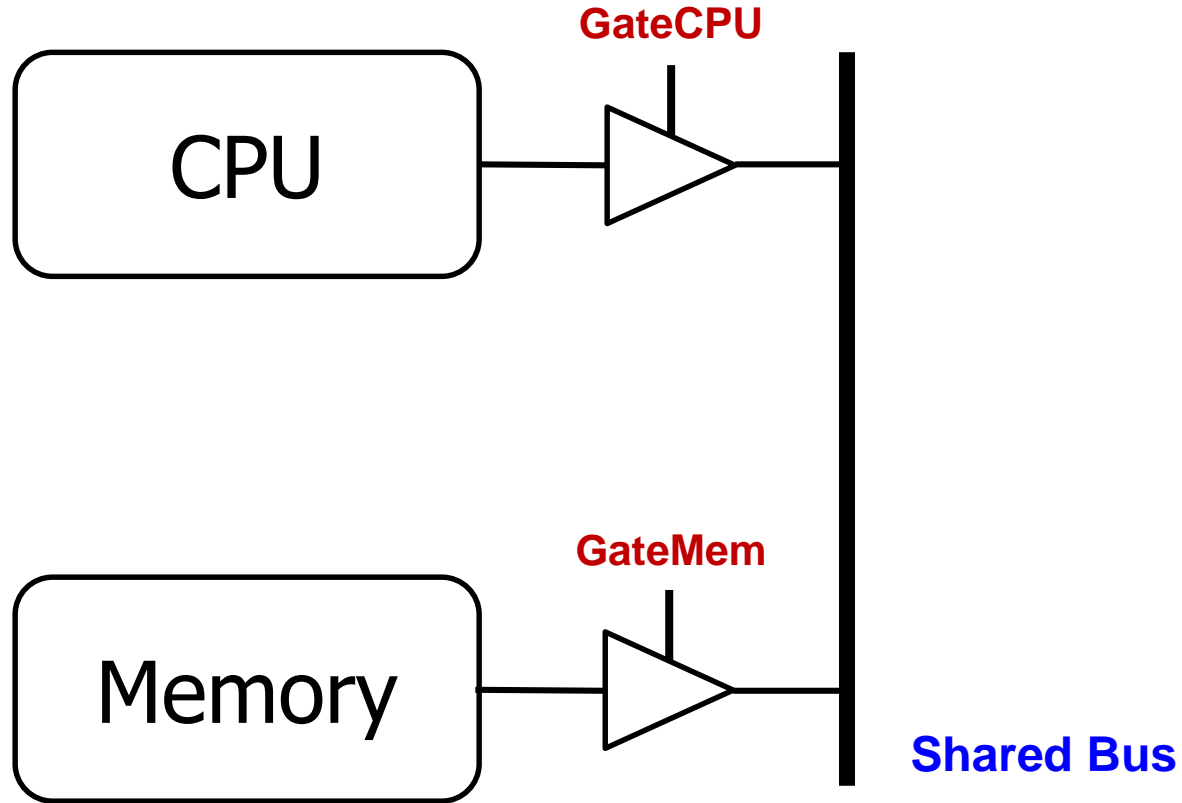
Figure 2.40 Tristate buffer

- **Floating signal (Z):** Signal that is not driven by any circuit
 - Open circuit, floating wire

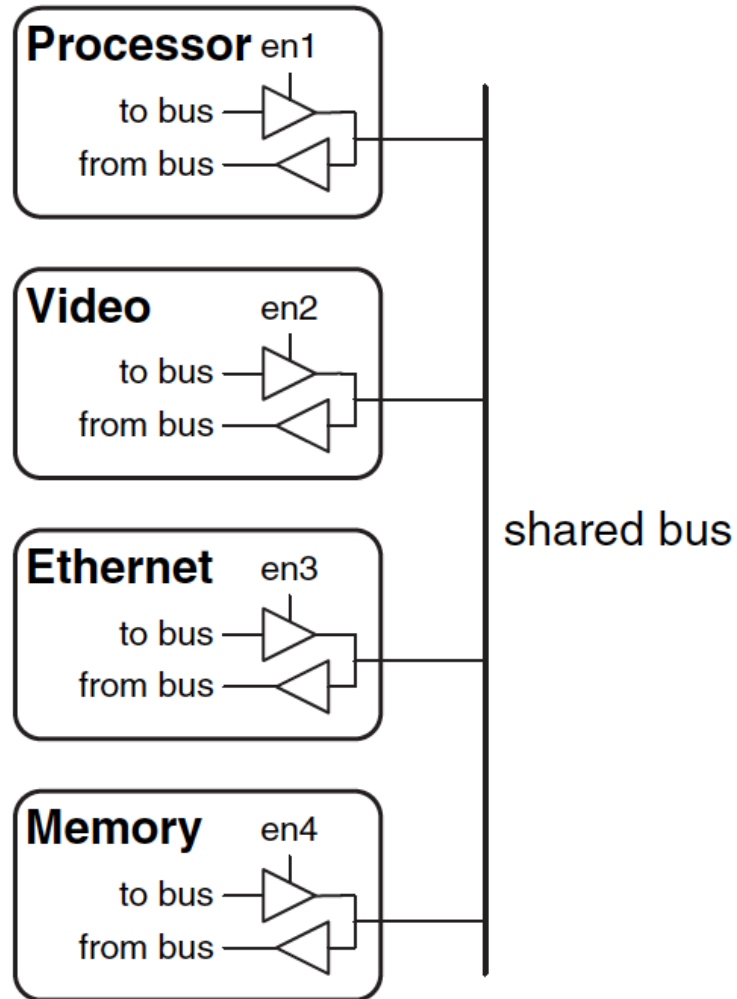
Recall: Example: Use of Tri-State Buffers

- Imagine a wire connecting the CPU and memory
 - At any time only the CPU or the memory can place a value on the wire, both not both
 - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time

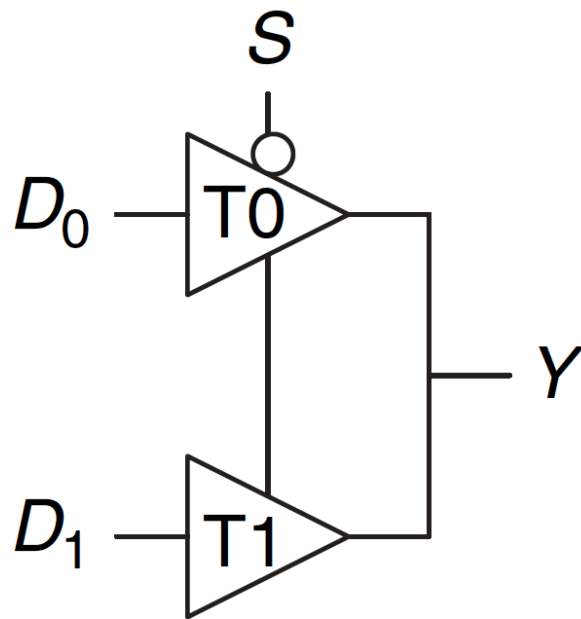
Recall: Example Design with Tri-State Buffers



Recall: Another Example

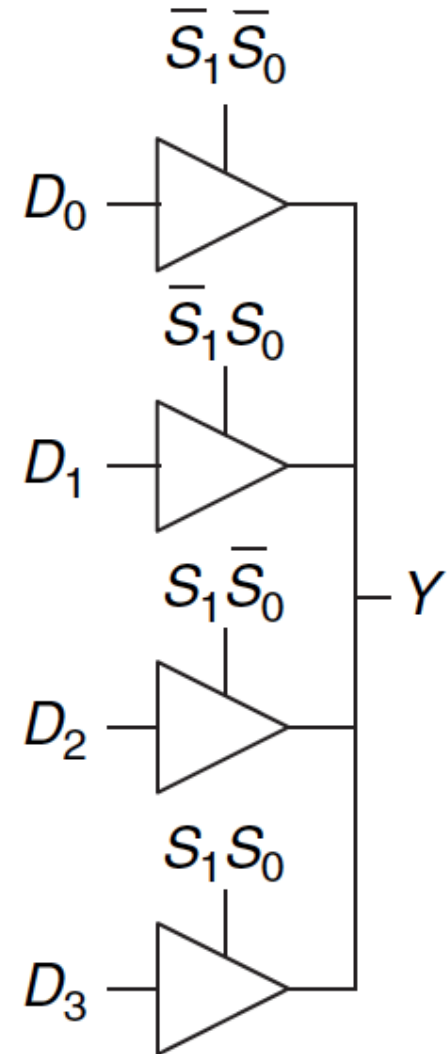


Multiplexer Using Tri-State Buffers



$$Y = D_0 \bar{S} + D_1 S$$

Figure 2.56 Multiplexer using tristate buffers



Aside: Logic Using Multiplexers

- Multiplexers can be used as lookup tables to perform logic functions

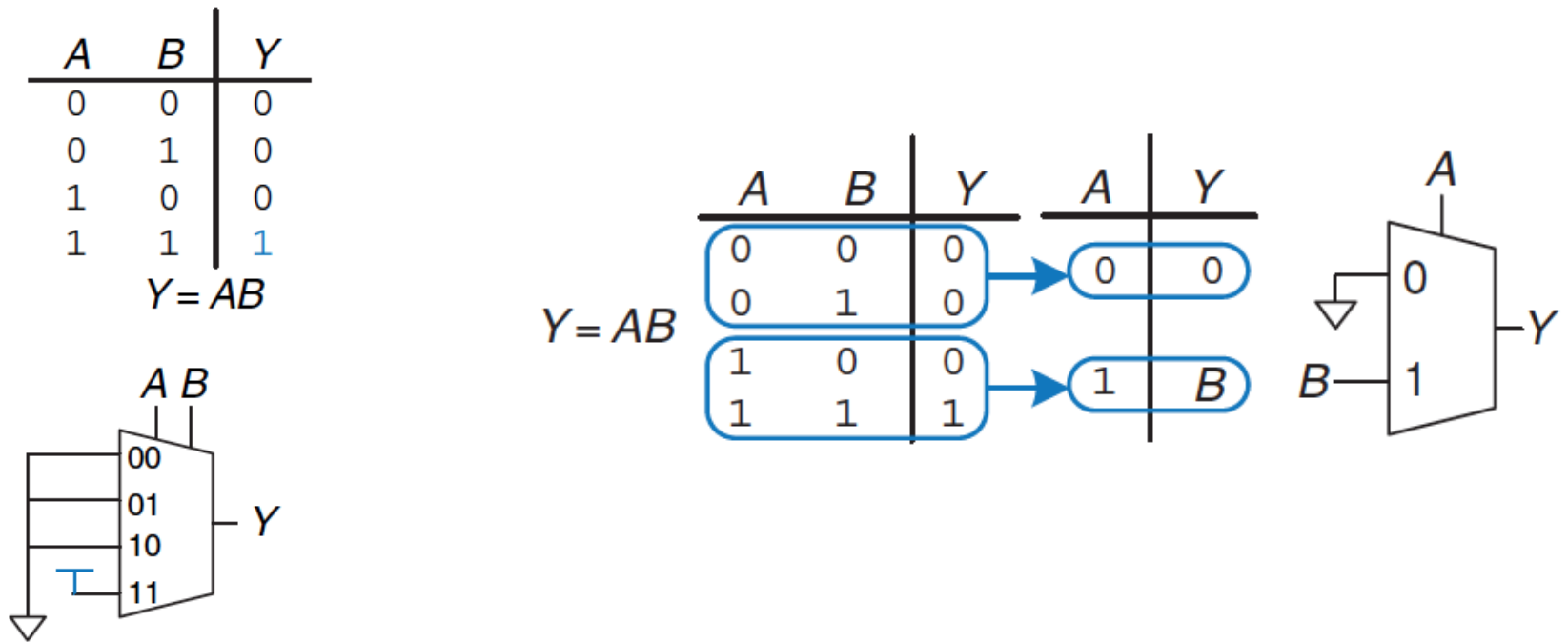
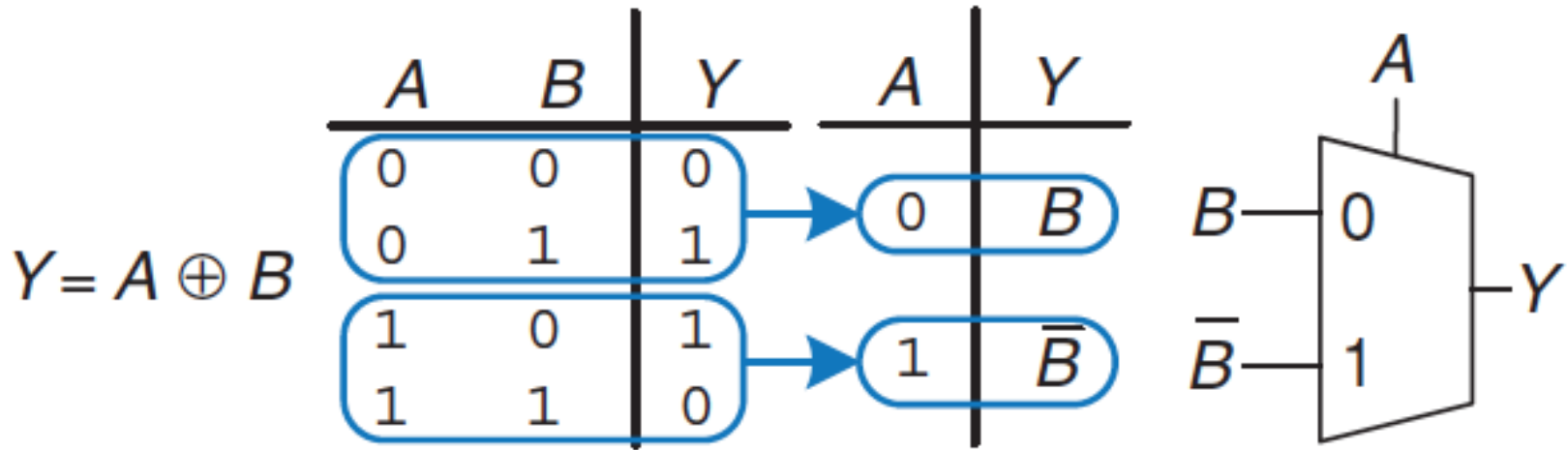


Figure 2.59 4:1 multiplexer implementation of two-input AND function

Aside: Logic Using Multiplexers (II)

- Multiplexers can be used as lookup tables to perform logic functions

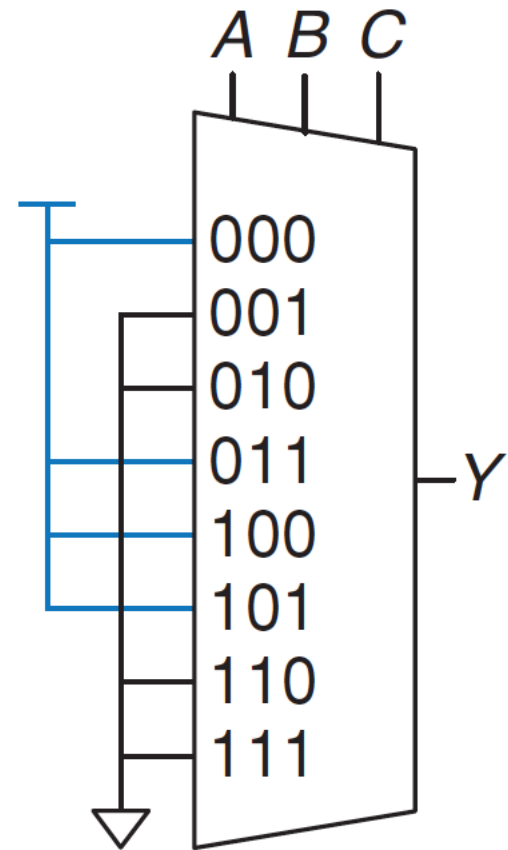


Aside: Logic Using Multiplexers (III)

- Multiplexers can be used as lookup tables to perform logic functions

| <i>A</i> | <i>B</i> | <i>C</i> | <i>Y</i> |
|----------|----------|----------|----------|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

$$Y = A\bar{B} + \bar{B}\bar{C} + \bar{A}BC$$



Aside: Logic Using Decoders (I)

- Decoders can be combined with OR gates to build logic functions.

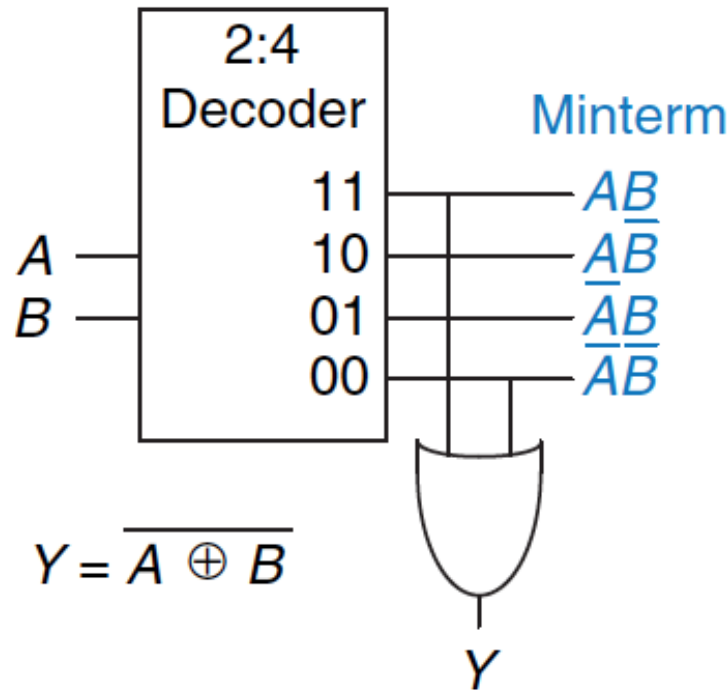
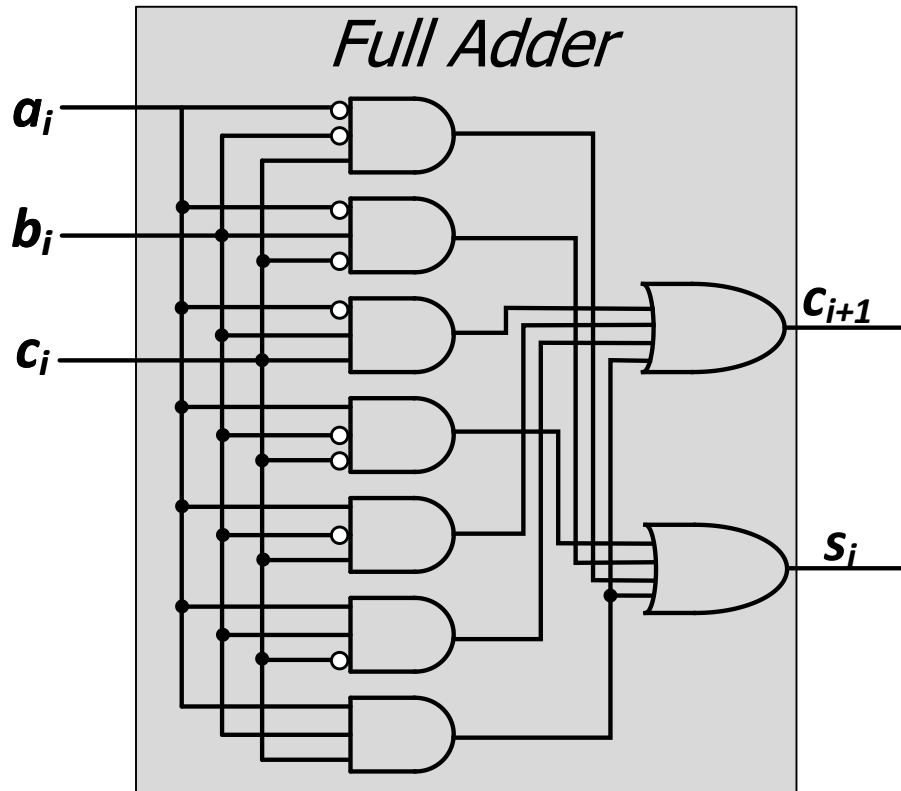


Figure 2.65 Logic function using decoder

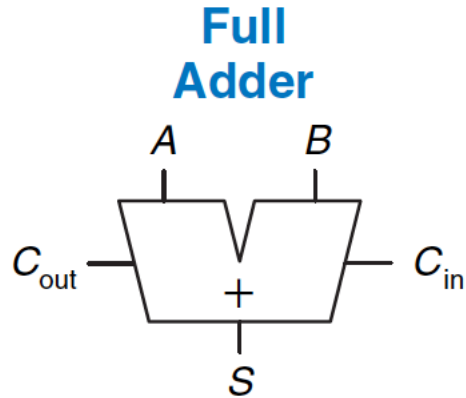
Logic Simplification using Boolean Algebra Rules

Recall: Full Adder in SOP Form Logic



| a_i | b_i | $carry_i$ | $carry_{i+1}$ | S_i |
|-------|-------|-----------|---------------|-------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Goal: Simplified Full Adder



$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

| C_{in} | A | B | C_{out} | S |
|----------|---|---|-----------|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

How do we simplify Boolean logic?

Quick Recap on Logic Simplification

- The original Boolean expression (i.e., logic circuit) may not be optimal

$$F = \sim A(A + B) + (B + AA)(A + \sim B)$$

- Can we reduce a given Boolean expression to an equivalent expression **with fewer terms?**

$$F = A + B$$

- The **goal** of logic simplification:
 - **Reduce** the number of gates/inputs
 - **Reduce** implementation cost

A basis for what the automated design tools are doing today

Logic Simplification

- Systematic techniques for simplifications

- amenable to automation

Key Tool: The Uniting Theorem — $F = A\bar{B} + AB$

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

$$F = A\bar{B} + AB = A(\bar{B} + B) = A(1) = A$$

Essence of Simplification:

Find two element subsets of the ON-set where only one variable changes its value. This single varying variable *can be eliminated!*

value is not needed

→ *B is eliminated, A remains*

| A | B | G |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$$G = \bar{A}\bar{B} + A\bar{B} = (\bar{A} + A)\bar{B} = \bar{B}$$

B's value stays the same within the ON-set rows

A's value changes within the ON-set rows

→ *A is eliminated, B remains*

Logic Simplification: Karnaugh Maps (K-Maps)

Karnaugh Maps are Fun...

- A pictorial way of minimizing circuits by visualizing opportunities for simplification
- They are for you to **study on your own...**

- See Backup Slides
- Read H&H Section 2.7
- Watch videos of Lectures 5 and 6 from 2019 Digitech course:
 - <https://youtu.be/0ks0PeaOUjE?list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&t=4570>
 - <https://youtu.be/ozs18ARNG6s?list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&t=220>

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

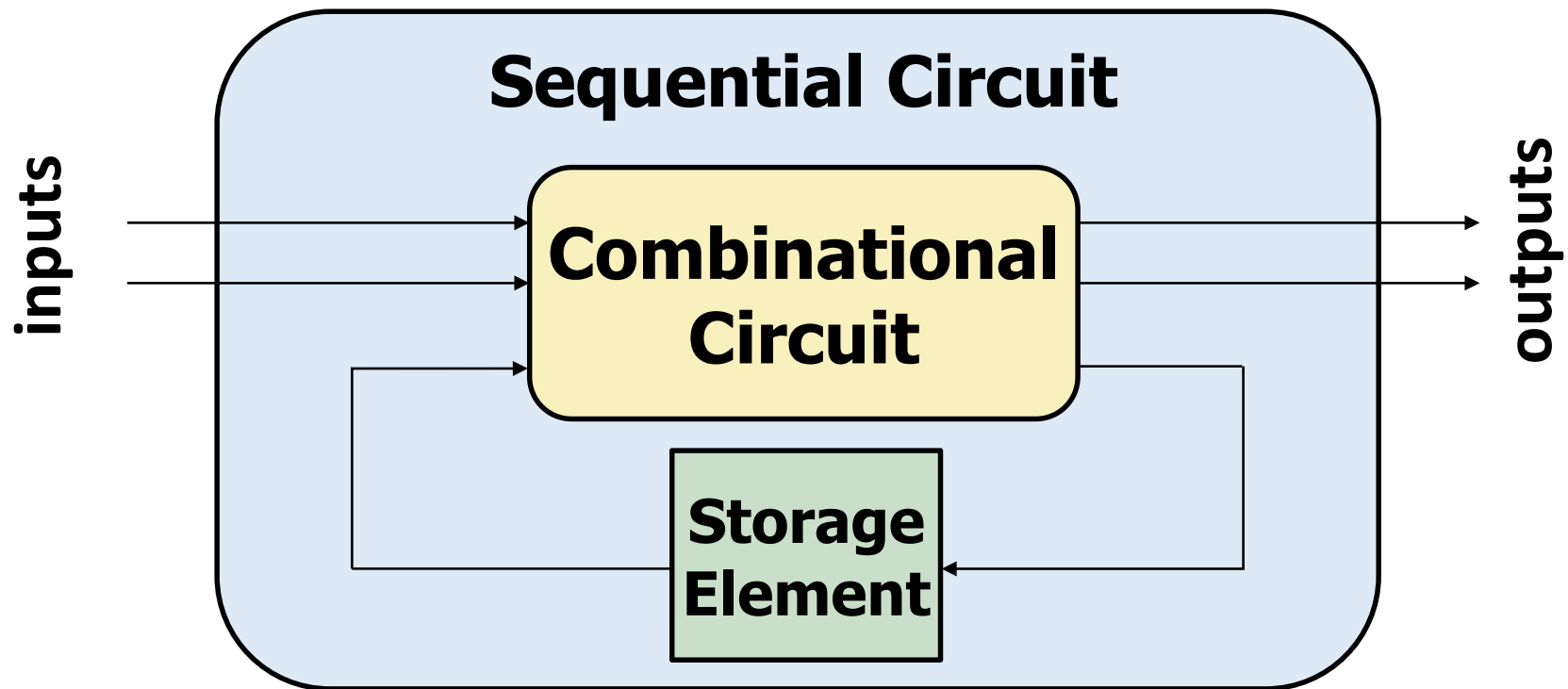
- Finite State Machines (FSM)
 - Moore Machine
 - Mealy Machine

- Verilog implementations of sequential circuits (next week)

Circuits that Can Store Information

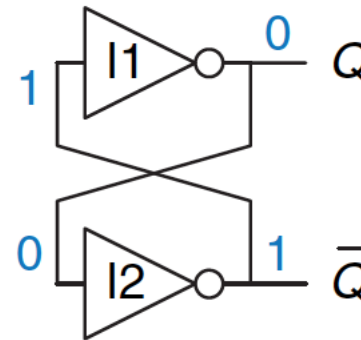
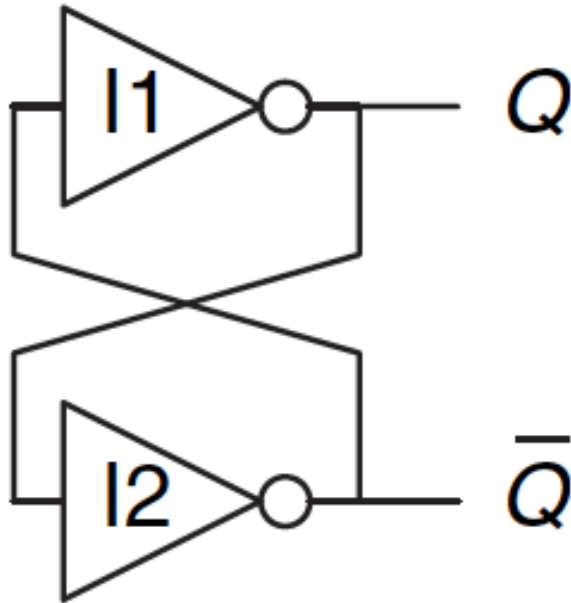
Introduction

- 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**?

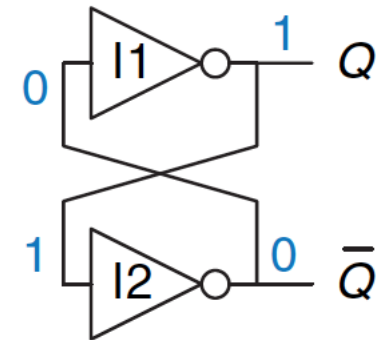


Capturing Data

Basic Element: Cross-Coupled Inverters



(a)

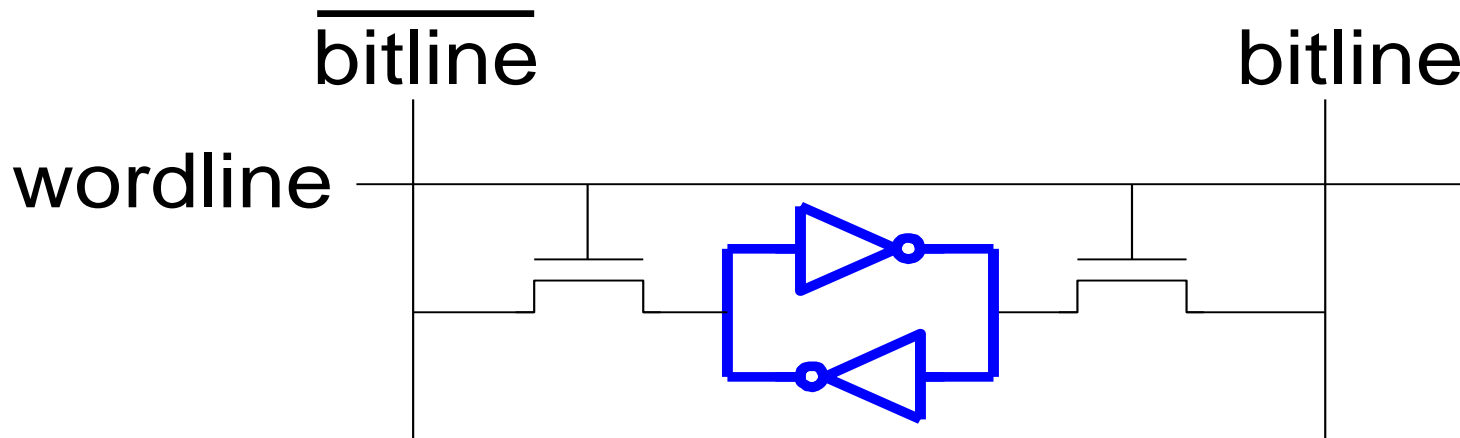


(b)

- 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**

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

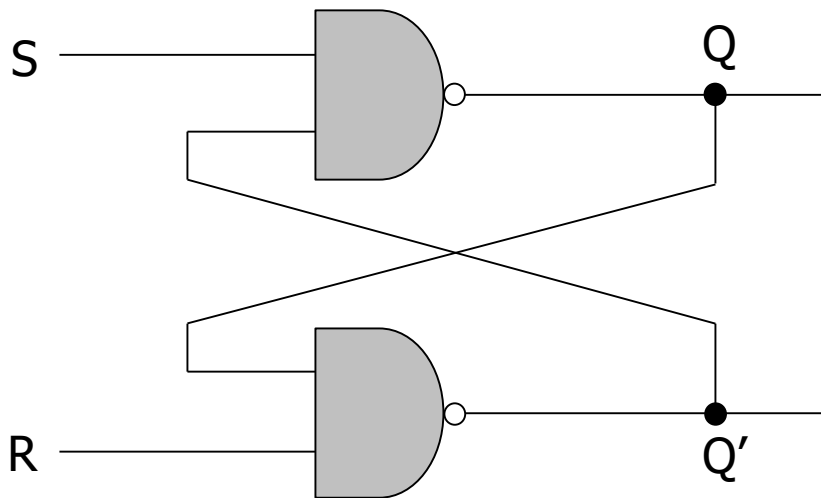
The Big Picture: Storage Elements

- 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

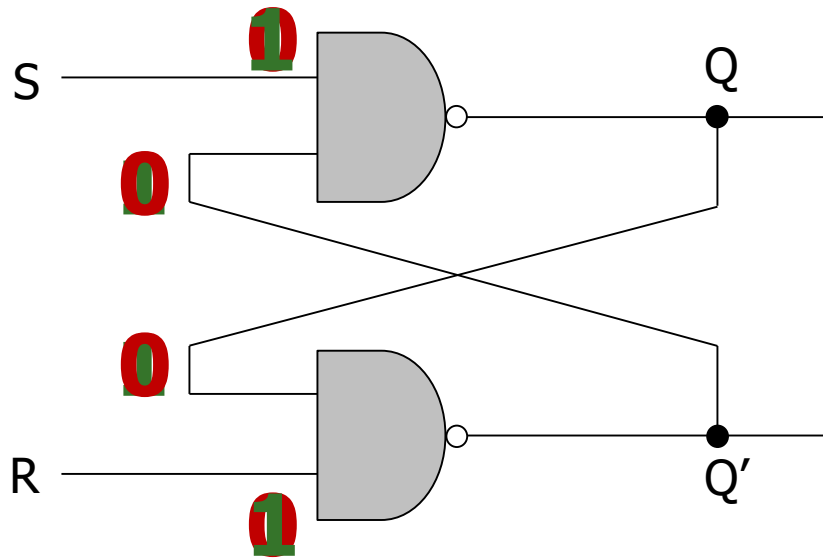
The R-S (Reset-Set) Latch

- 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 $R=S=0$?



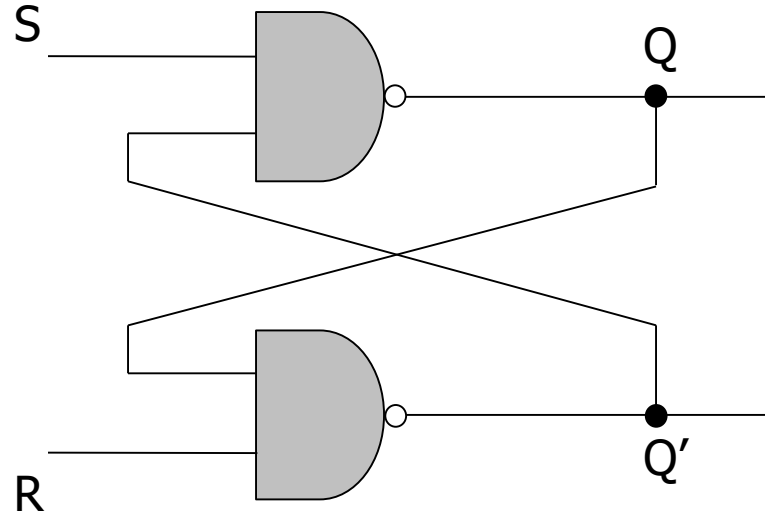
| Input | | Output |
|-------|---|-------------------|
| R | S | Q |
| 1 | 1 | Q_{prev} |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 0 | Forbidden |

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

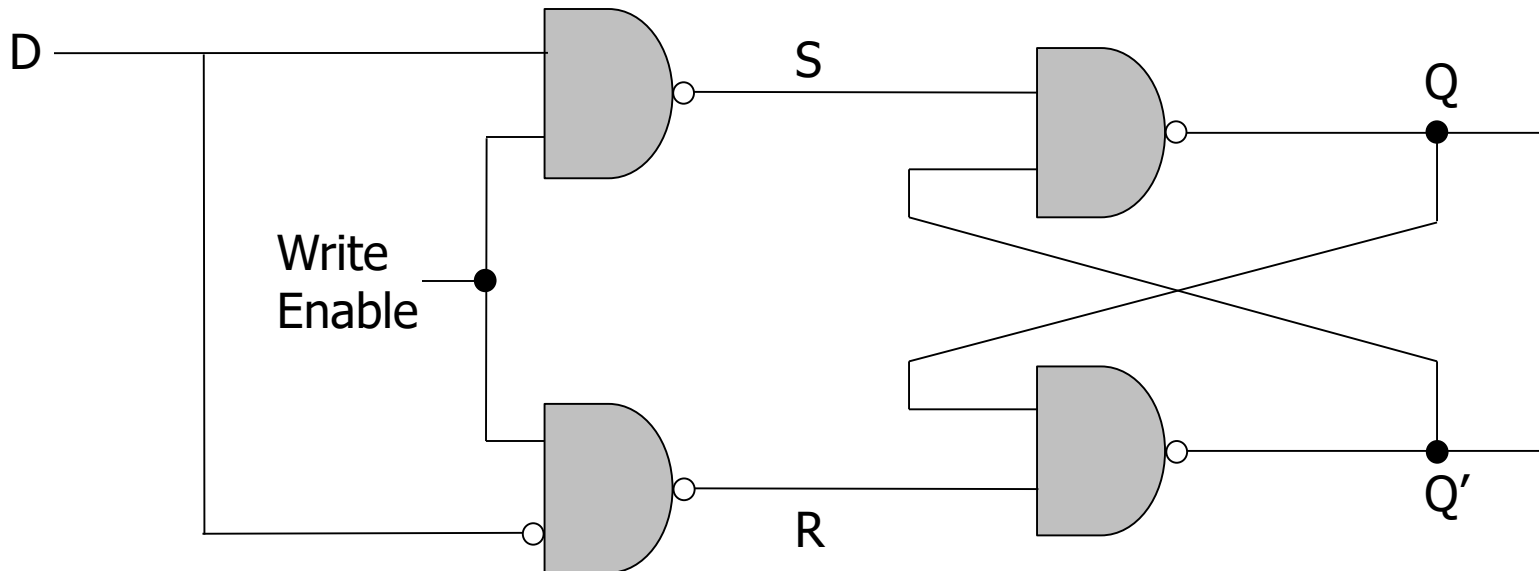
The Gated D Latch

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



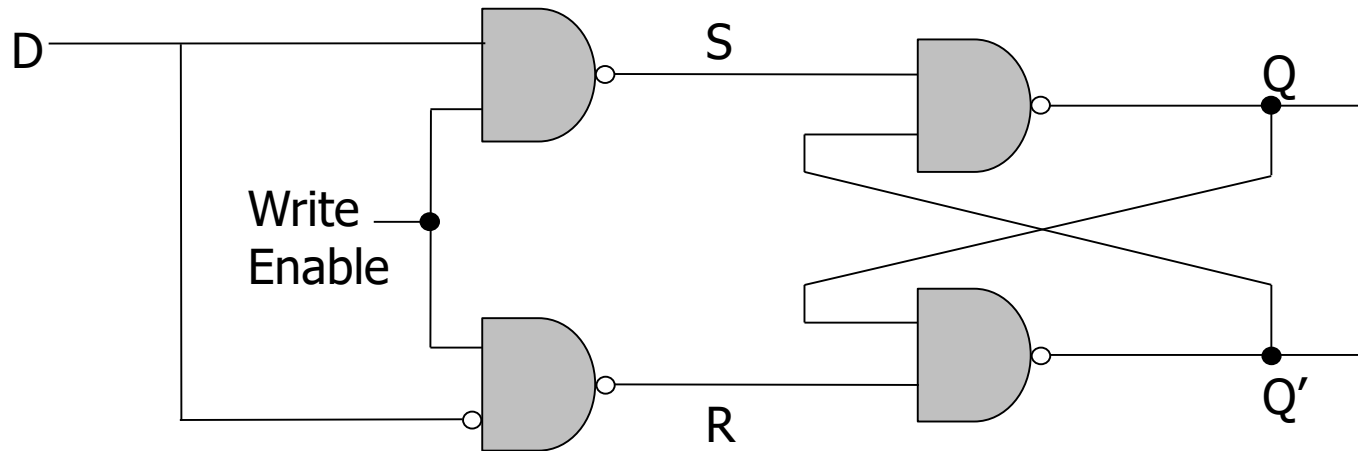
The Gated D Latch

- 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



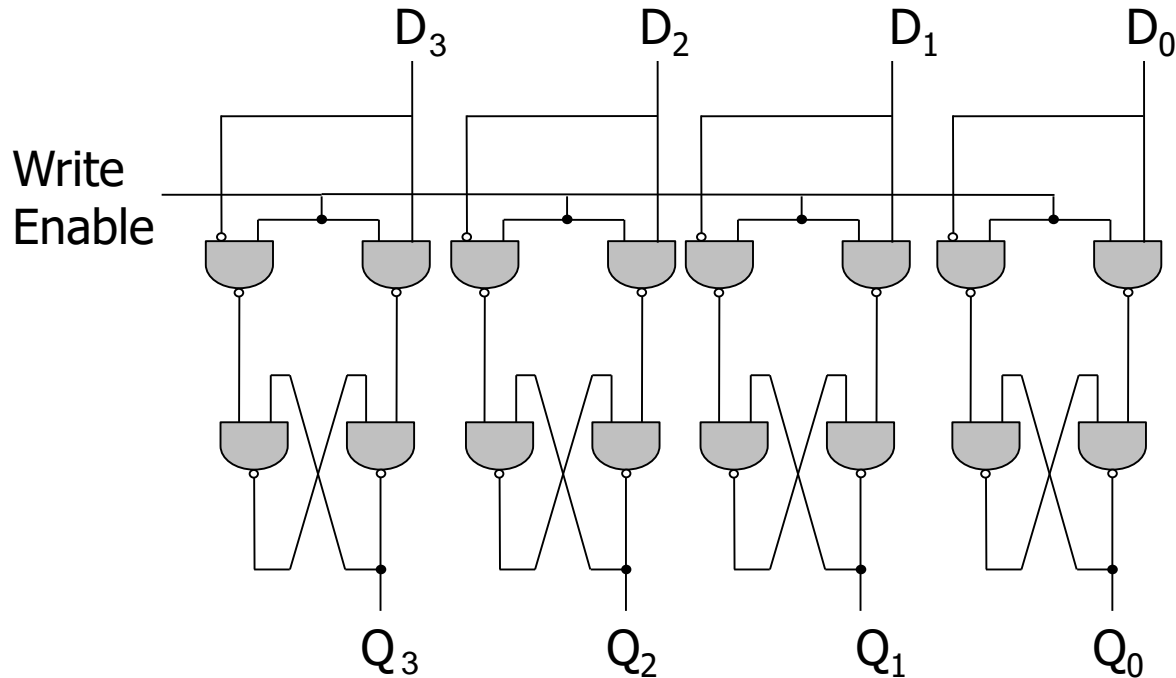
| Input | | Output |
|-------|---|------------|
| WE | D | Q |
| 0 | 0 | Q_{prev} |
| 0 | 1 | Q_{prev} |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

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



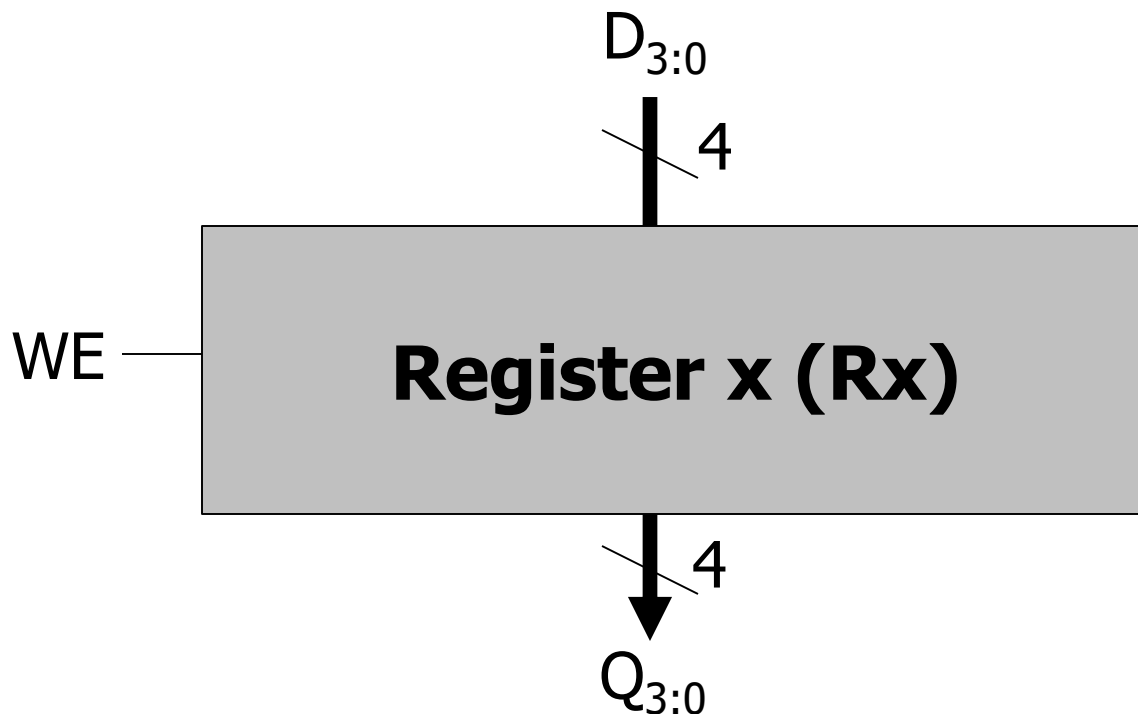
Here we have a **register**, or a structure that stores more than one bit and can be read from and written to

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 |

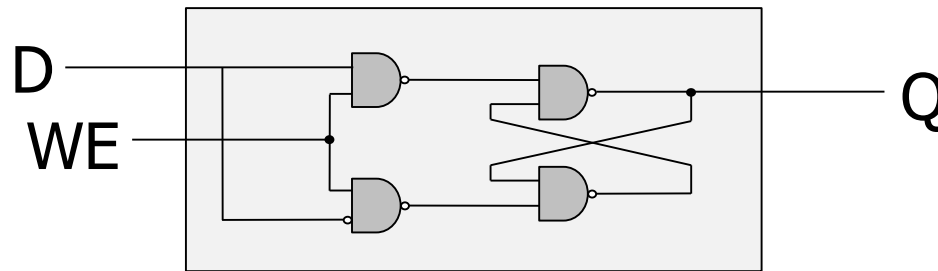
- Every unique location in memory is indexed with a unique **address**. 4 locations require 2 address bits ($\log[\#locations]$).
- **Addressability**: the number of bits of information stored in each location. This example: addressability is 8 bits.
- The entire set of unique locations in memory is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



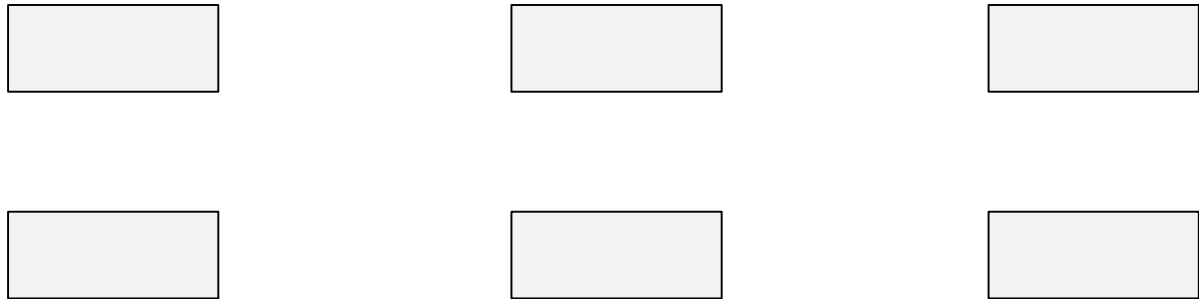
6-Bit Memory Array

| | | | |
|----------------|------------------|------------------|------------------|
| Addr(0) | Bit ₂ | Bit ₁ | Bit ₀ |
| Addr(1) | Bit ₂ | Bit ₁ | Bit ₀ |

Reading from Memory

How can we select the address to read?

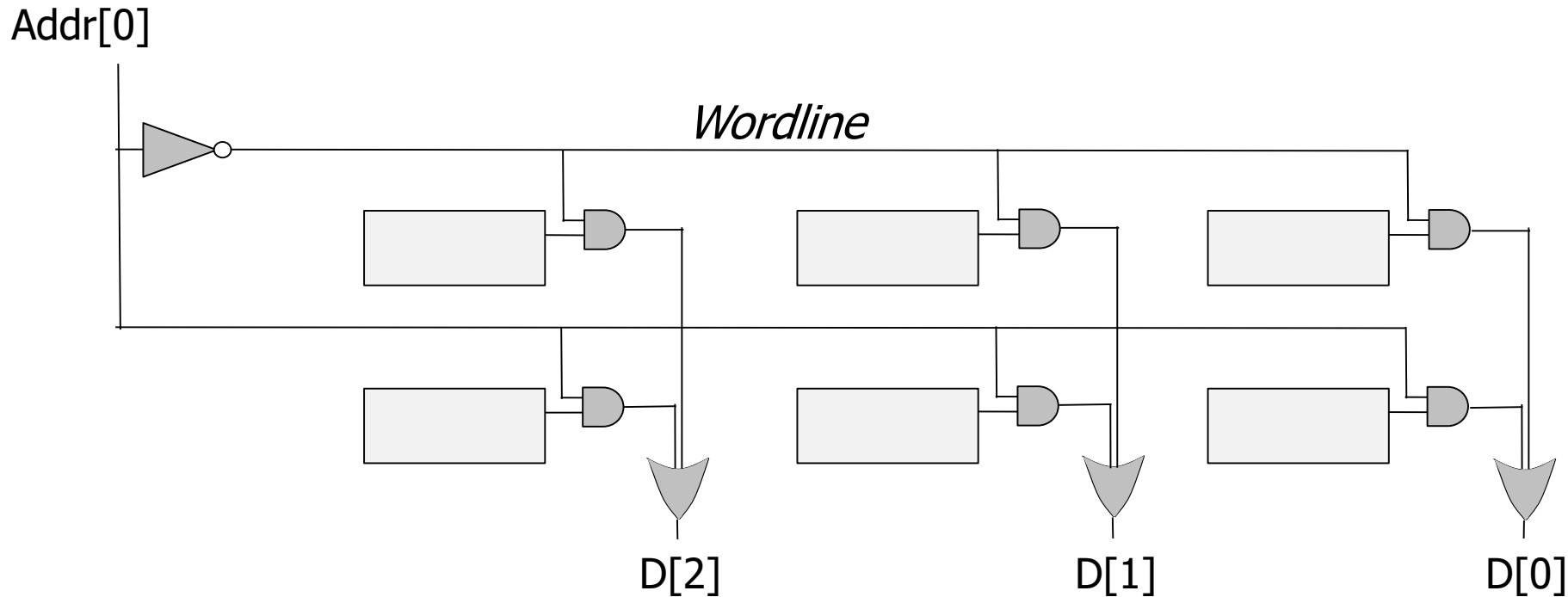
- Because there are 2 addresses, address size is $\log(2)=1$ bit



Reading from Memory

How can we select an address to read?

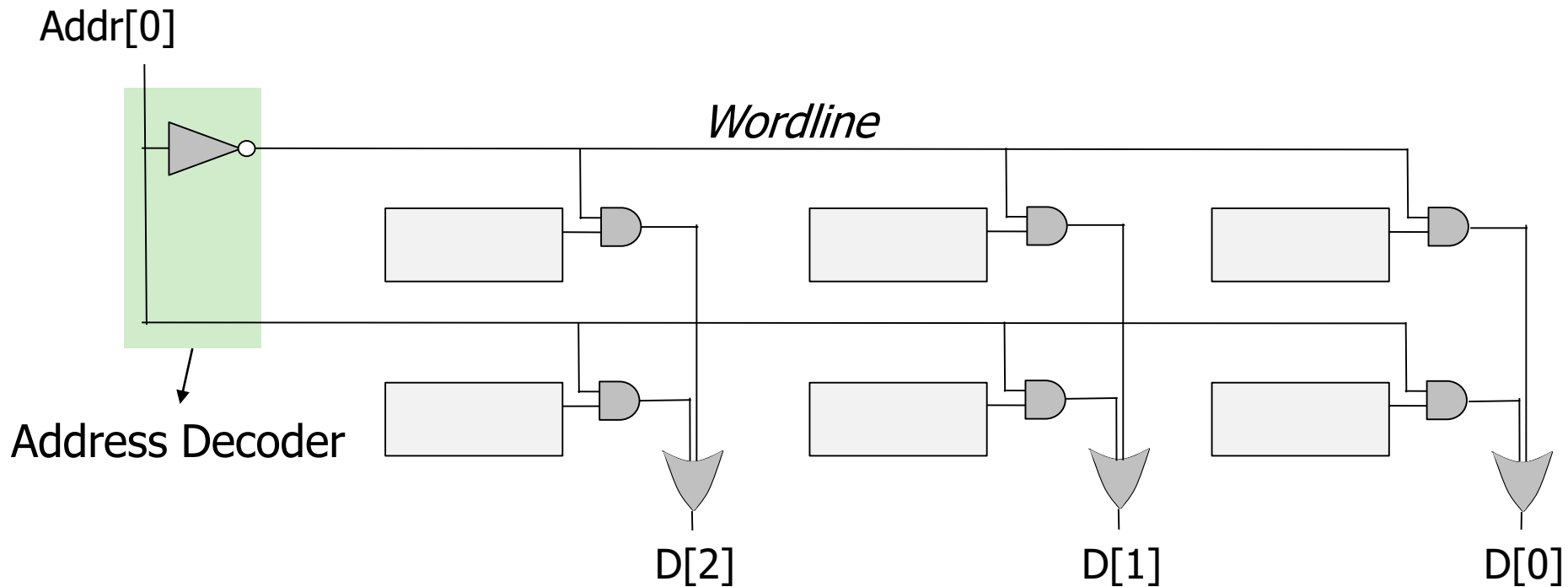
- Because there are 2 addresses, address size is $\log(2)=1$ bit



Reading from Memory

How can we select an address to read?

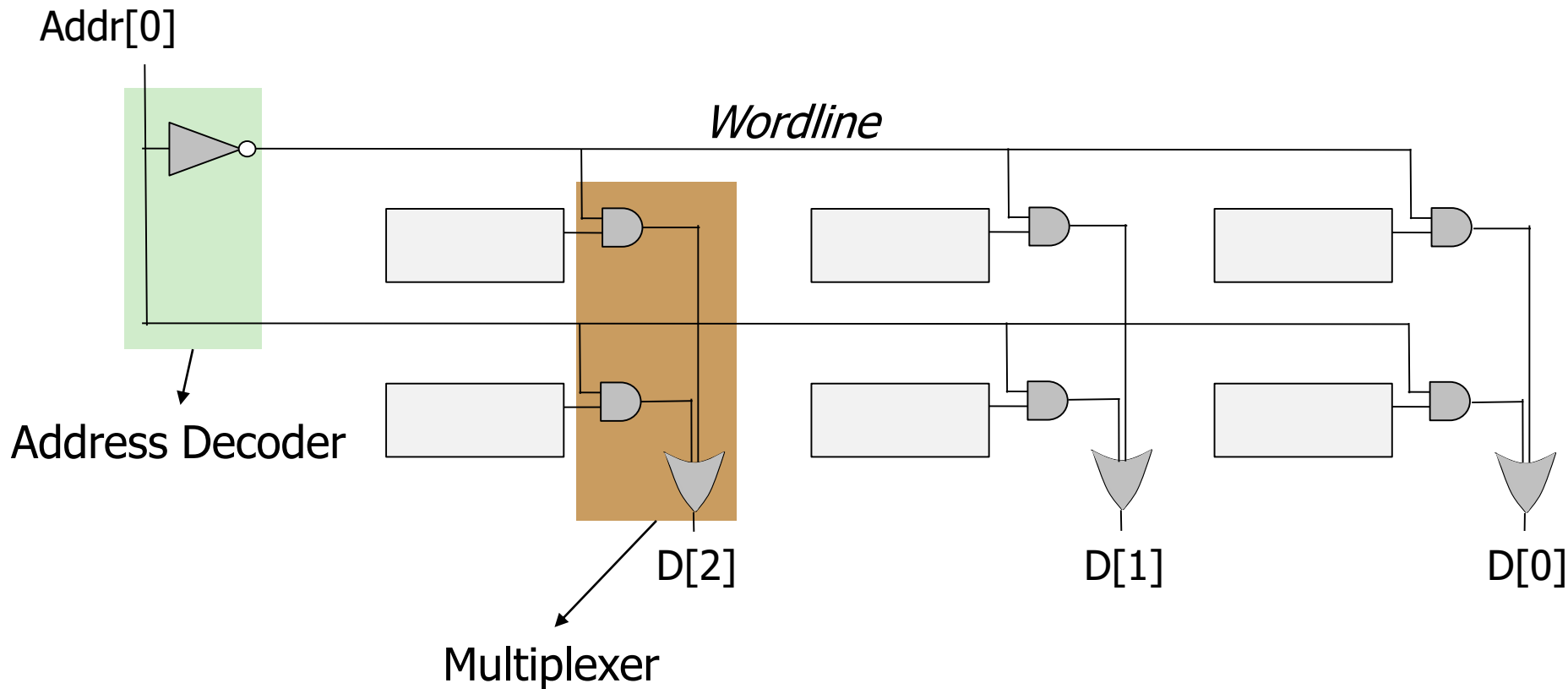
- Because there are 2 addresses, address size is $\log(2)=1$ bit



Reading from Memory

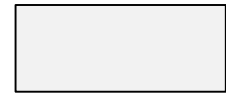
How can we select an address to read?

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



Writing to Memory

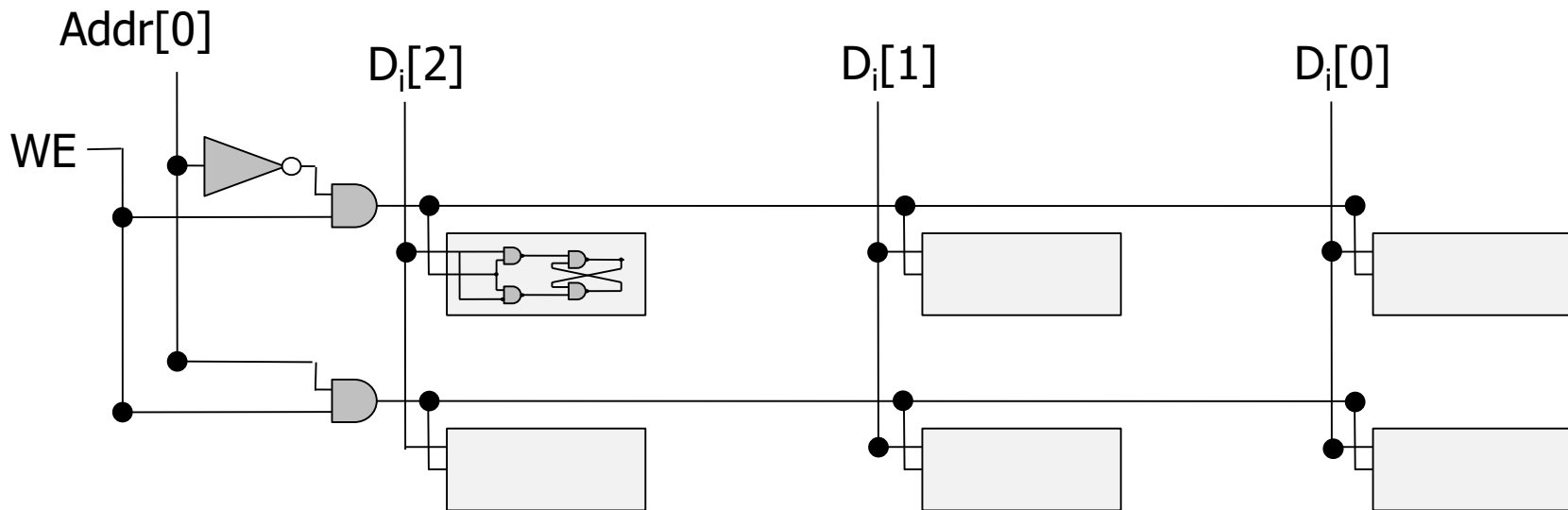
How can we select an address and write to it?



Writing to Memory

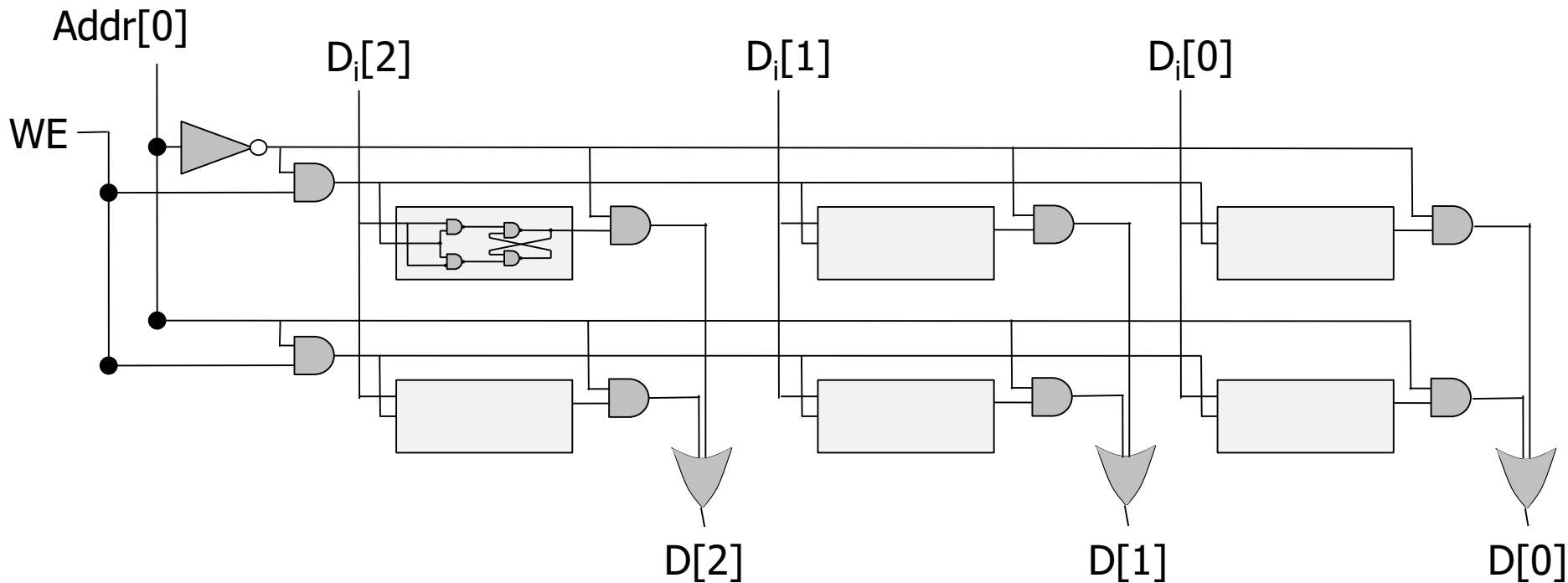
How can we select an address and write to it?

- Input is indicated with D_i

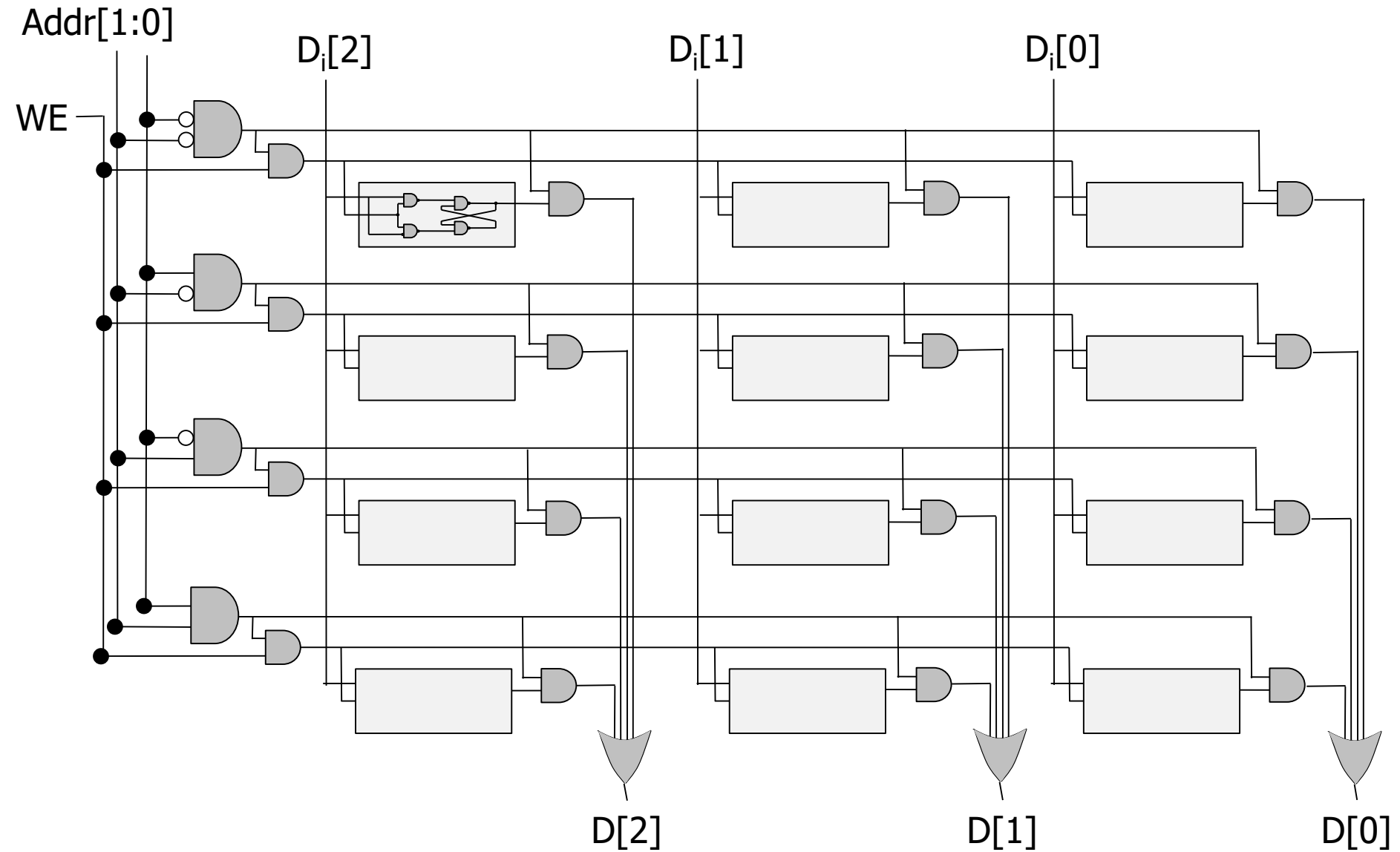


Putting it all Together

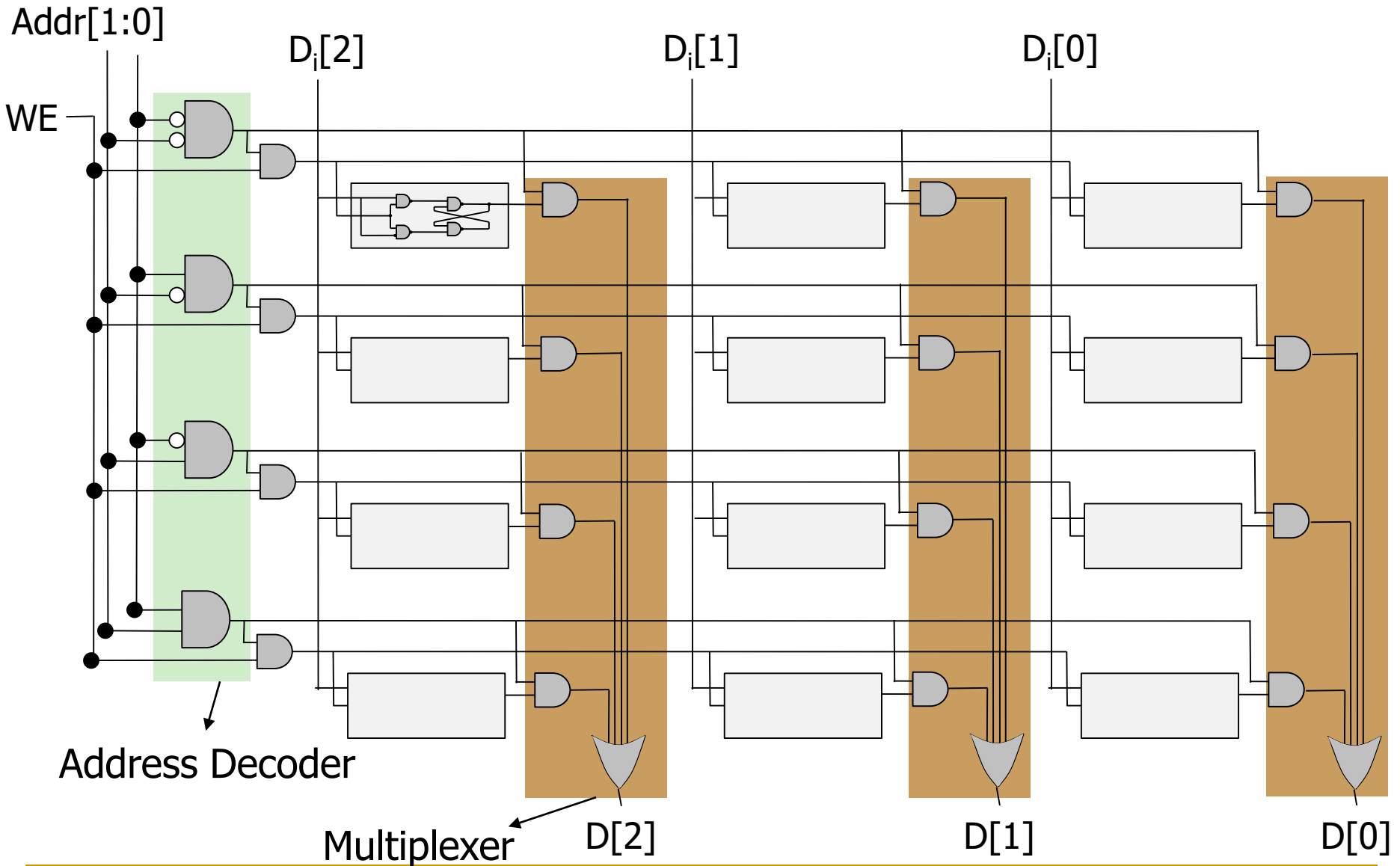
Let's enable reading and writing to a memory array



A Bigger Memory Array



A Bigger Memory Array



Sequential Logic Circuits

Sequential Logic Circuits

- We have looked at 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

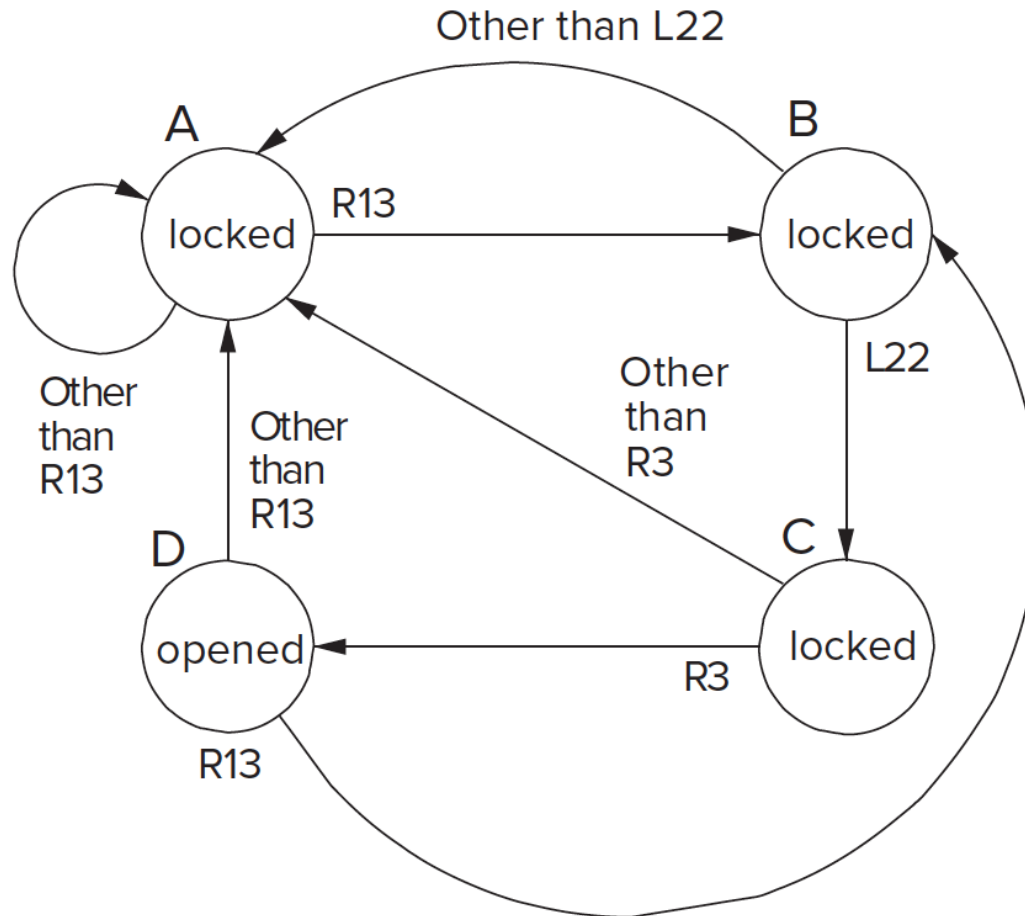
State

- 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

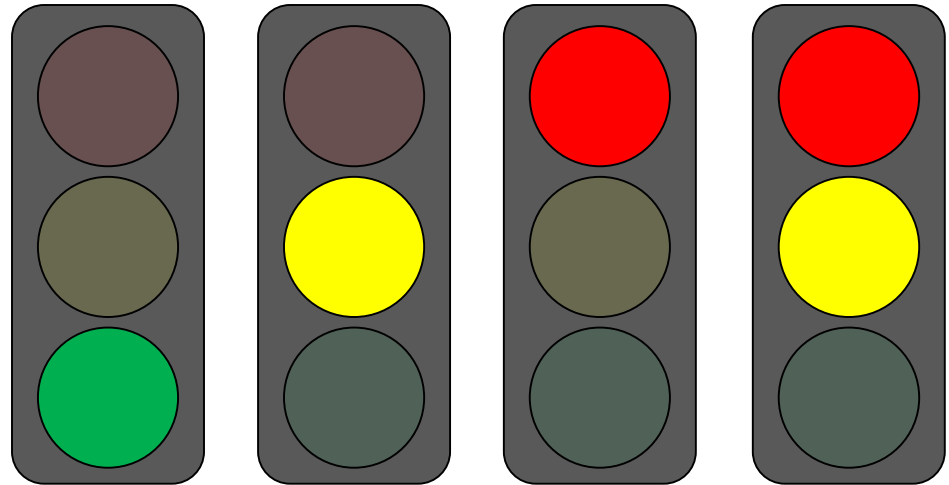


- We will understand "state diagrams" fully later today

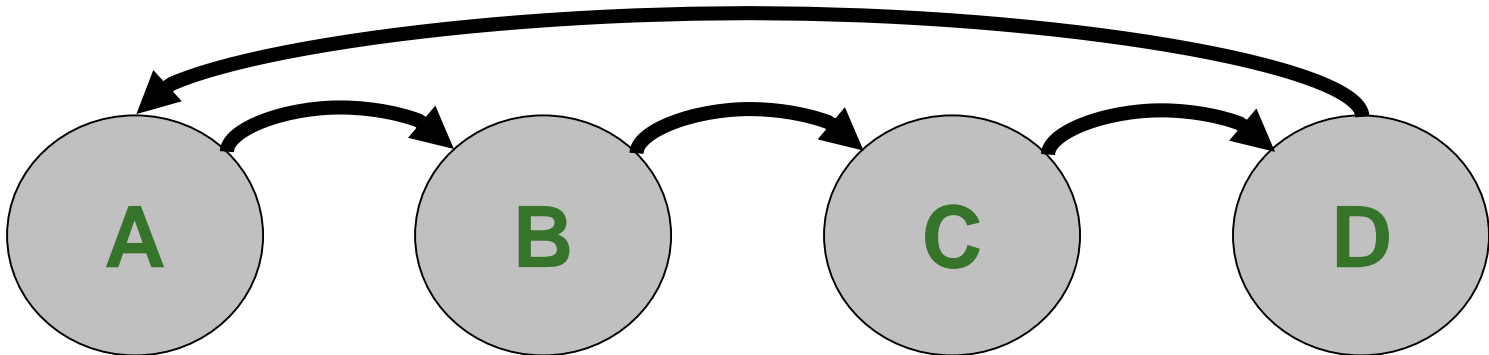
Another Simple Example of State

- A standard Swiss traffic light has **4 states**

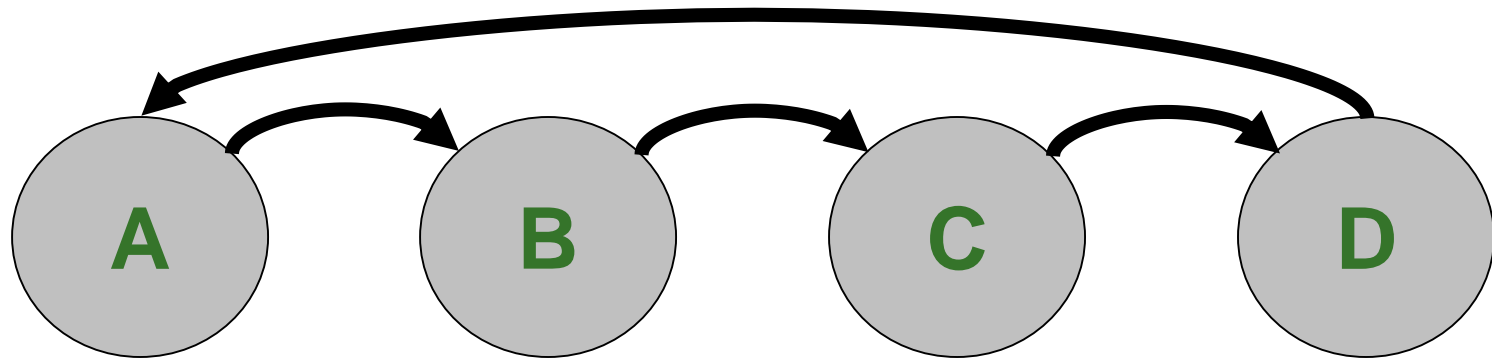
- A. Green
- B. Yellow
- C. Red
- D. Red and Yellow



- 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

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**

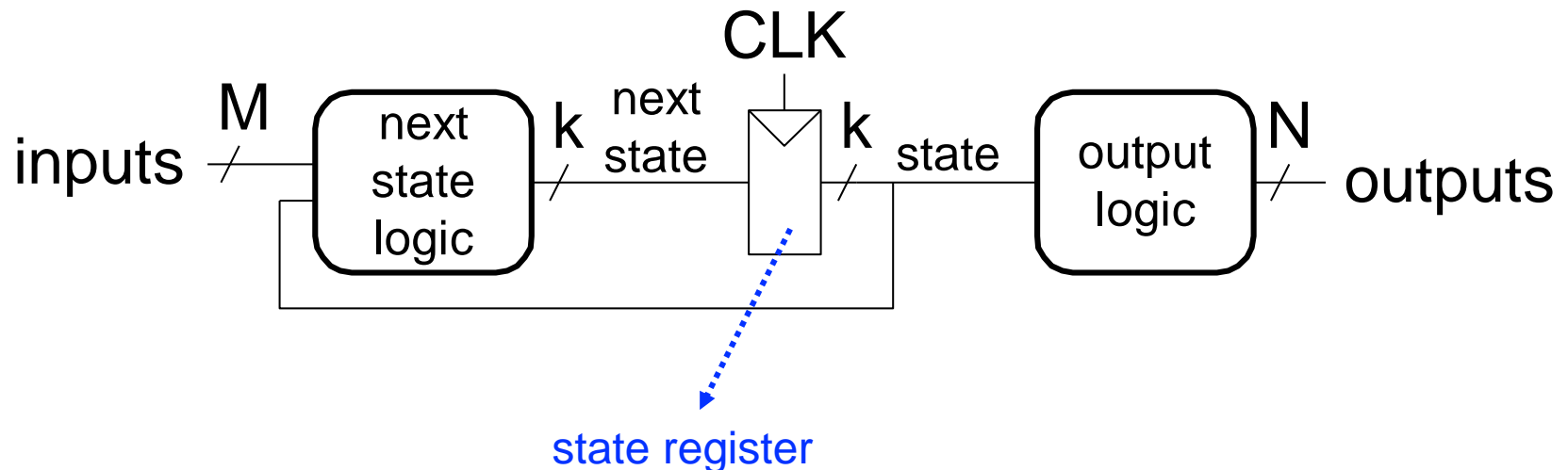
Finite State Machines (FSMs) Consist of:

■ **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

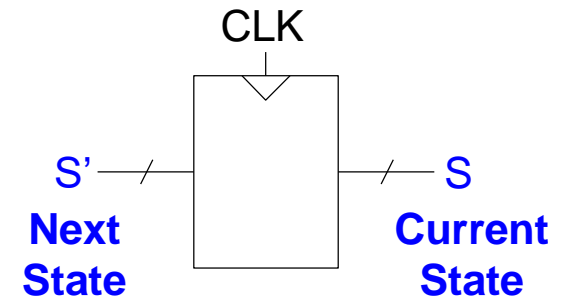


At the beginning of the clock cycle, next state is latched into the state register

Finite State Machines (FSMs) Consist of:

■ Sequential circuits

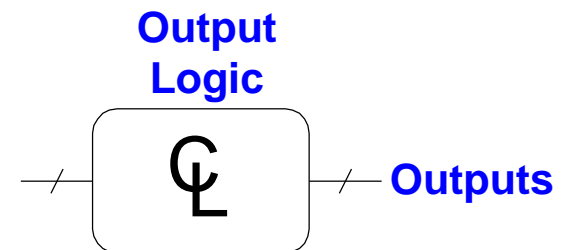
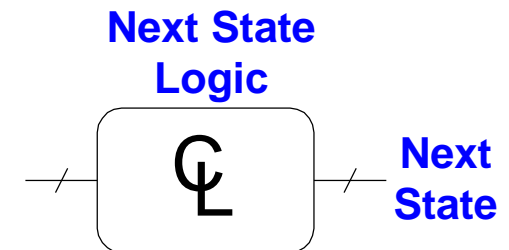
- State register(s)
 - Store the current state and
 - Load the next state at the clock edge



■ Combinational Circuits

- Next state logic
 - Determines what the next state will be

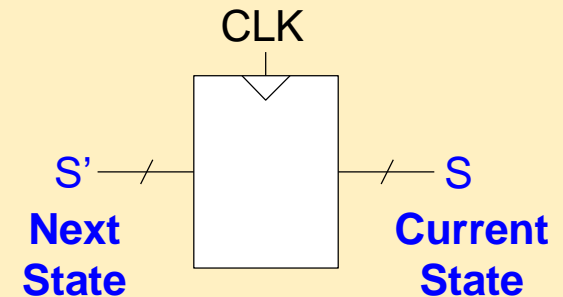
- Output logic
 - Generates the outputs



Finite State Machines (FSMs) Consist of:

■ Sequential circuits

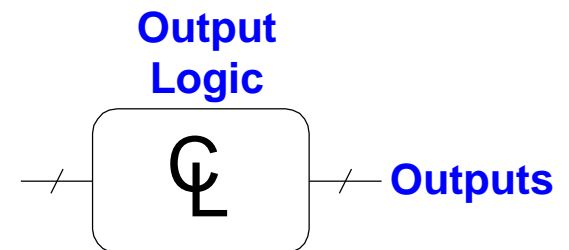
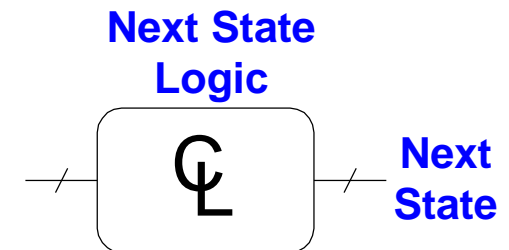
- State register(s)
 - Store the current state and
 - Load the next state at the clock edge



■ Combinational Circuits

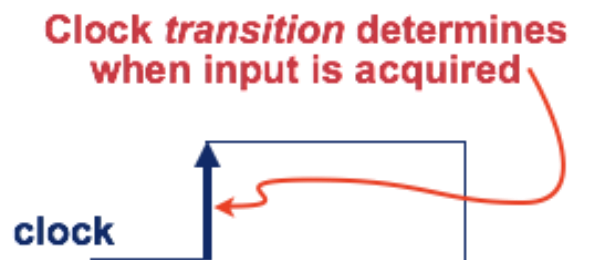
- Next state logic
 - Determines what the next state will be

- Output logic
 - Generates the outputs

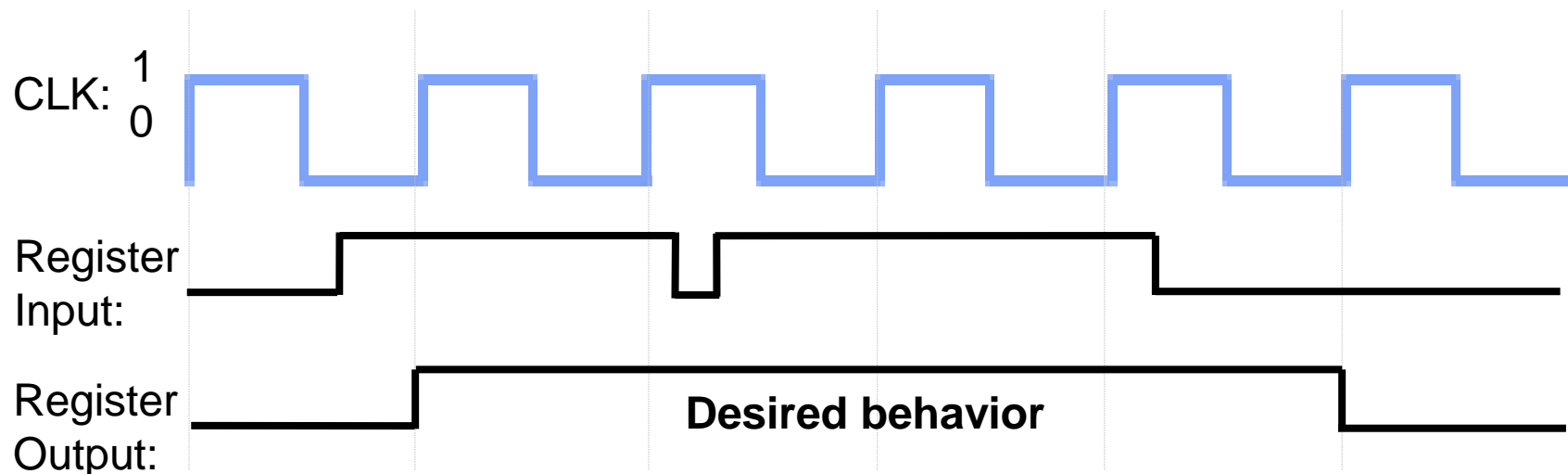


State Register Implementation

- How can we implement a **state register**? Two properties:
 - We need to store data at the **beginning** of every clock cycle

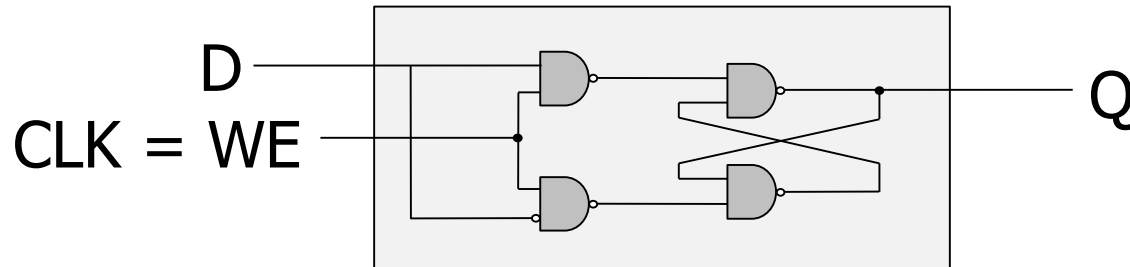


- The data must be **available** during the entire clock cycle

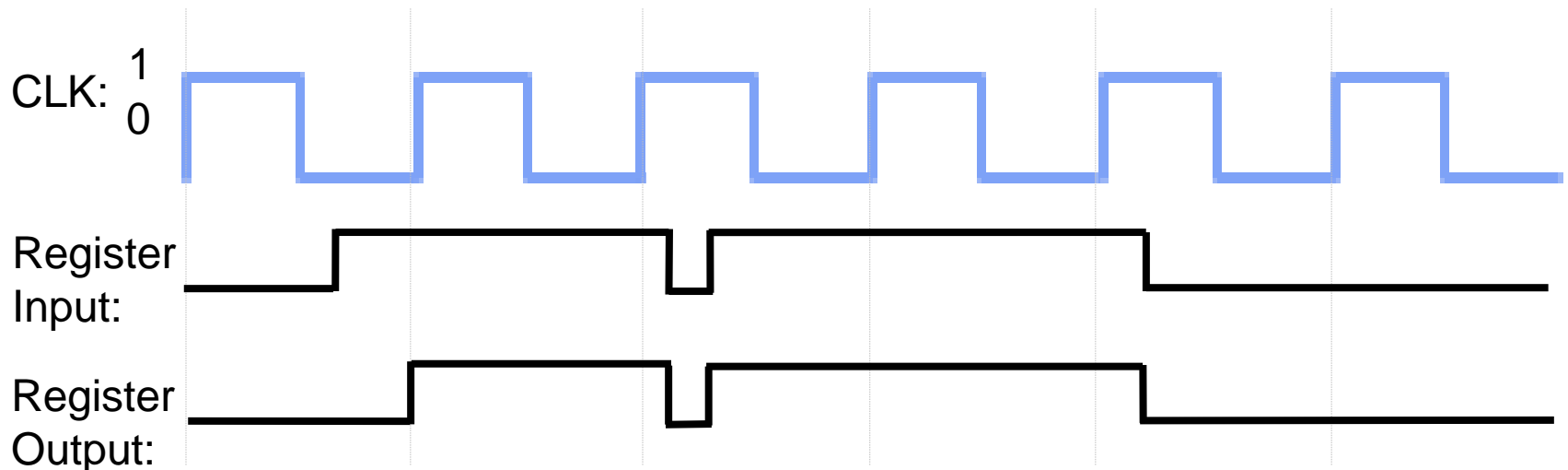


The Problem with Latches

Recall the
Gated D Latch

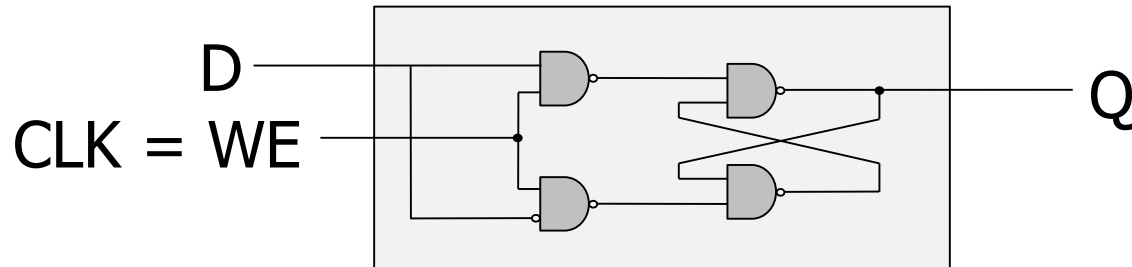


- Currently, we **cannot** simply wire a clock to WE of a latch
 - ❑ **Whenever the clock is high**, the latch propagates **D** to **Q**
 - ❑ **The latch is transparent**

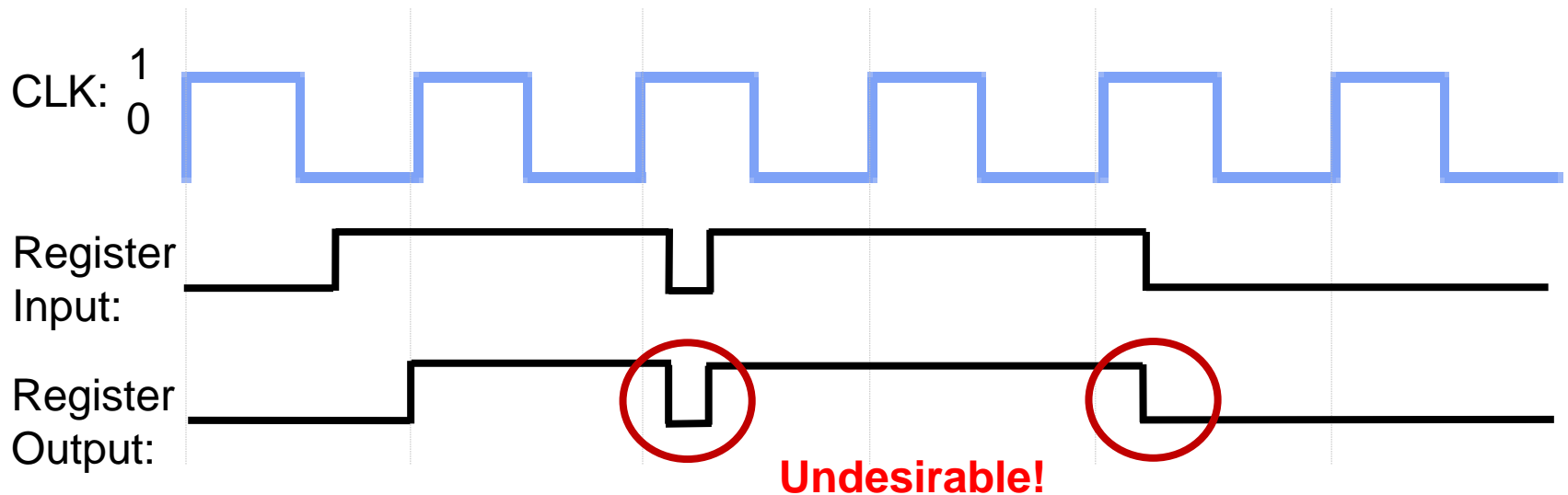


The Problem with Latches

Recall the
Gated D Latch

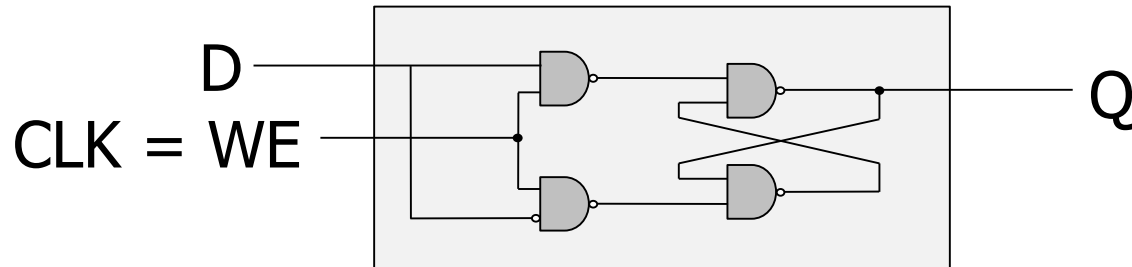


- Currently, we **cannot** simply wire a clock to WE of a latch
 - ❑ **Whenever the clock is high**, the latch propagates **D** to **Q**
 - ❑ **The latch is transparent**



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

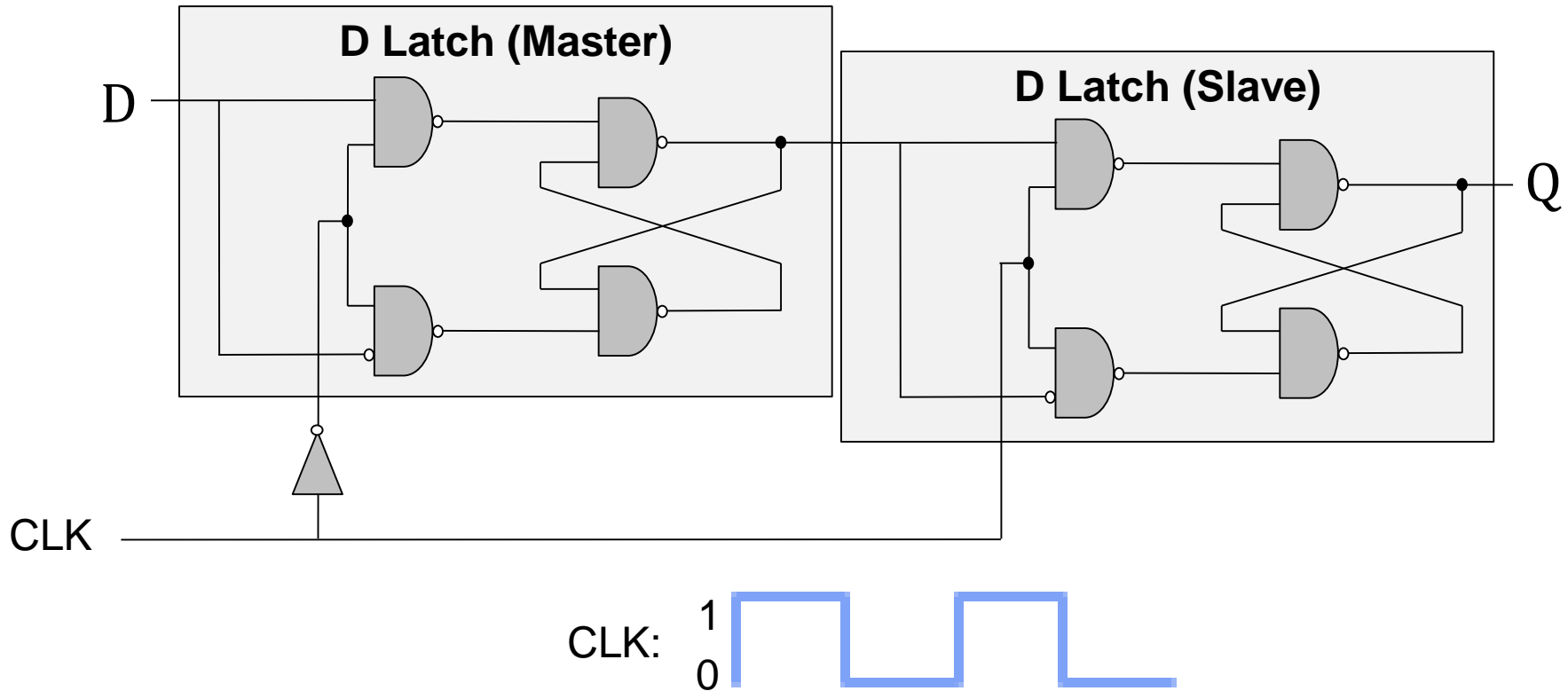
- To design viable FSMs
- We need storage elements that allow us
 - to read the **current state** throughout the **current clock cycle**

AND

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

The D Flip-Flop

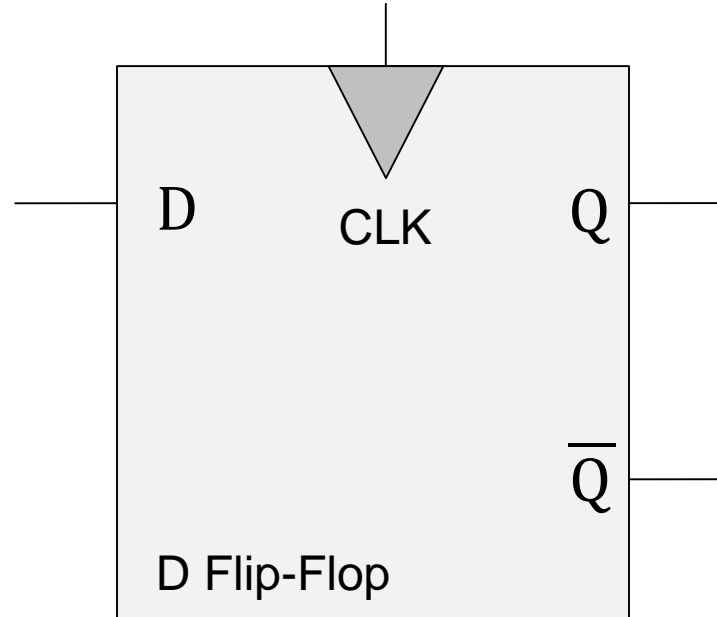
- 1) state change on clock edge, 2) data available for full cycle



- 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- \rightarrow 1), **Q** gets assigned **D**

The D Flip-Flop

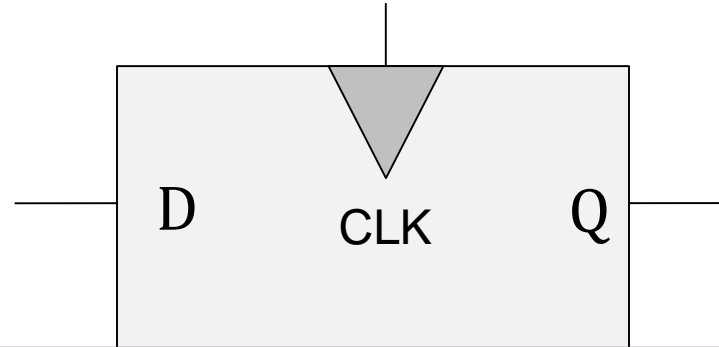
- 1) state change on clock edge, 2) data available for full cycle



- 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

- How do we implement this?



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

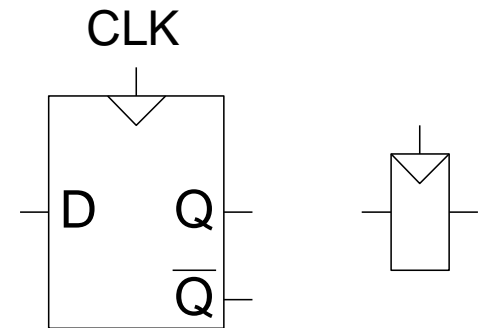
- At the rising edge of clock (clock going from 0->1), **Q** gets assigned **D**
- At all other times, Q is unchanged

Rising-Clock-Edge Triggered Flip-Flop

- **Two inputs:** CLK, D

- **Function**

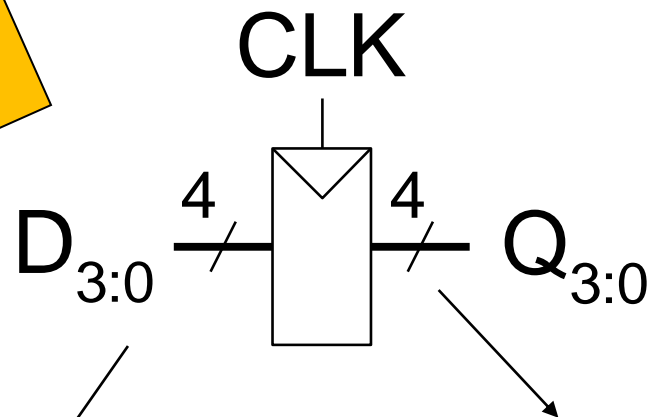
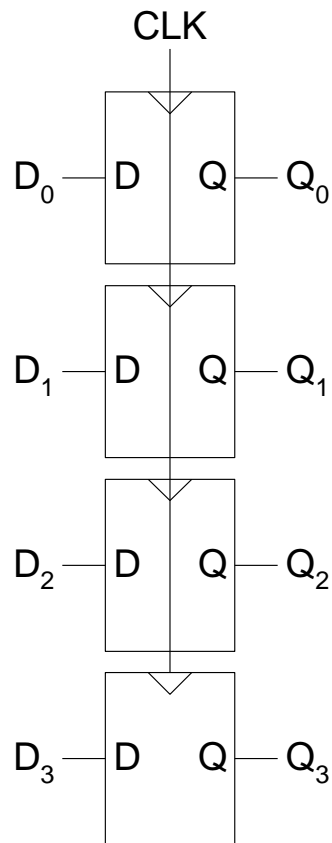
- The flip-flop “samples” **D** on the rising edge of CLK (**positive edge**)
- When CLK rises from 0 to 1, **D** passes through to **Q**
- Otherwise, **Q** holds its previous value
- **Q** changes **only** on the rising edge of CLK



- 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

Register

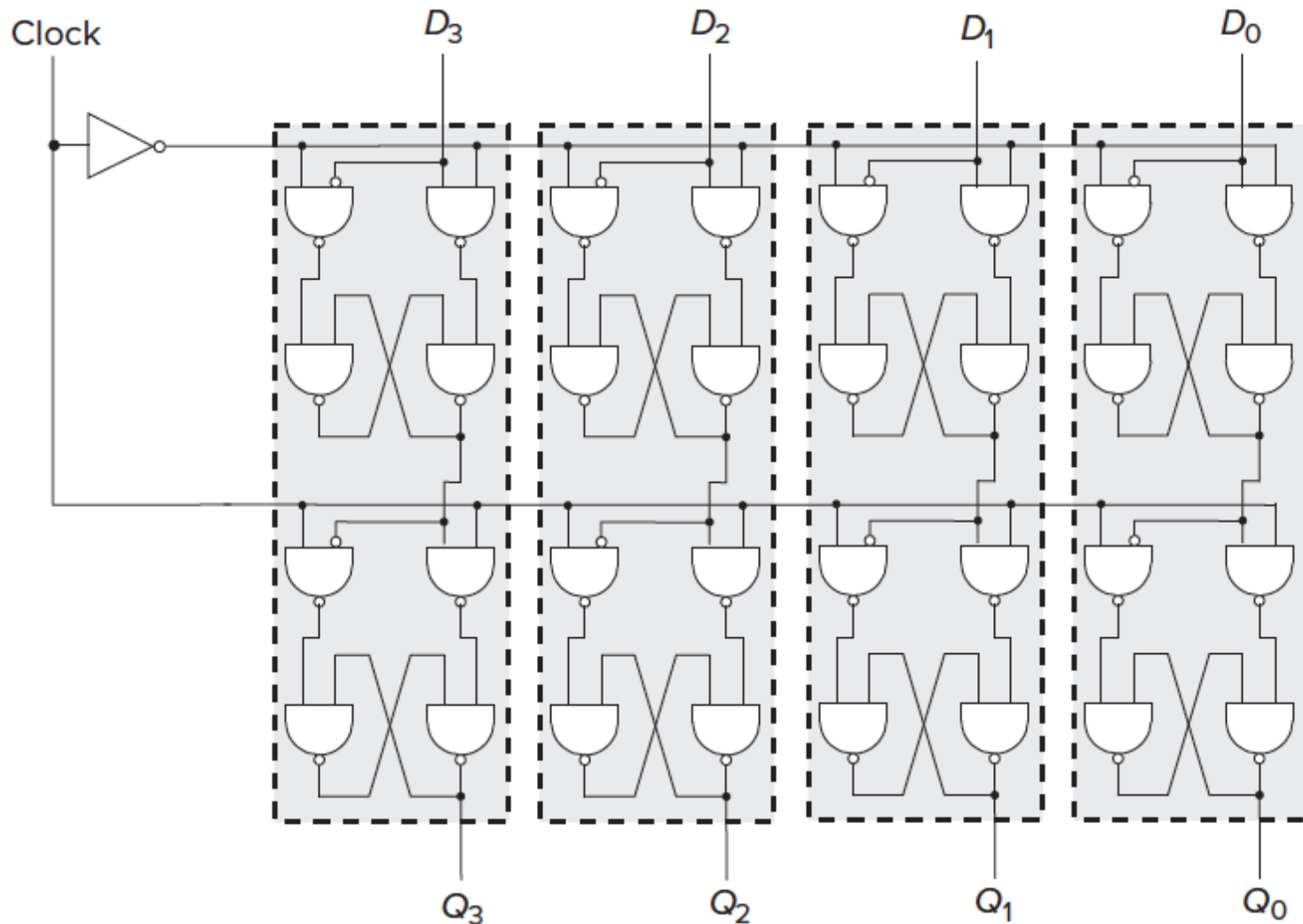
- Multiple parallel flip-flops, each of which storing 1 bit



This line represents 4 wires

This register stores 4 bits

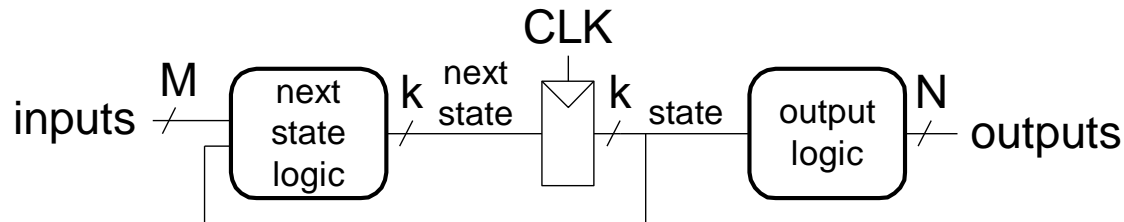
A 4-Bit D-Flip-Flop-Based Register (Internally)



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

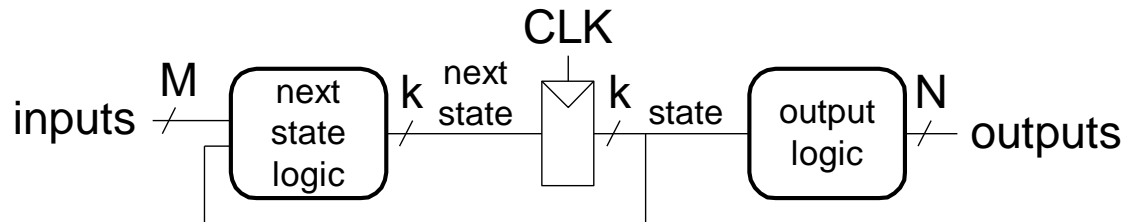
Moore FSM



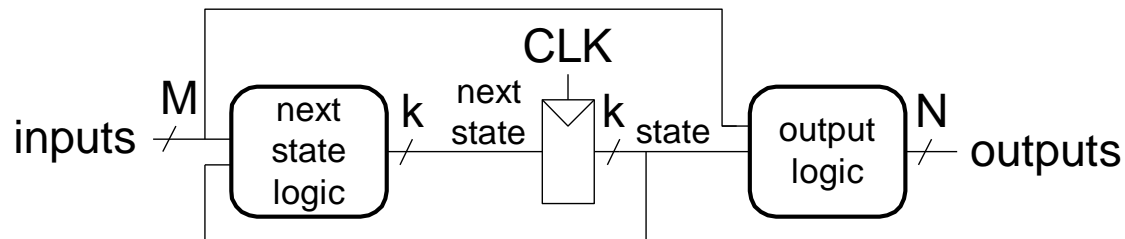
Finite State Machines (FSMs)

- 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

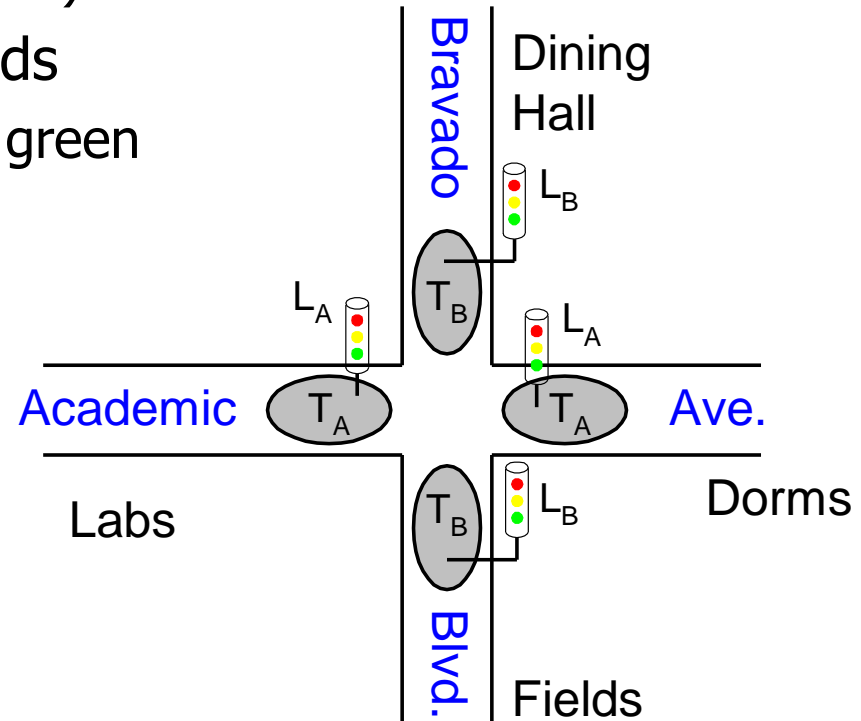


Mealy FSM



Finite State Machine Example

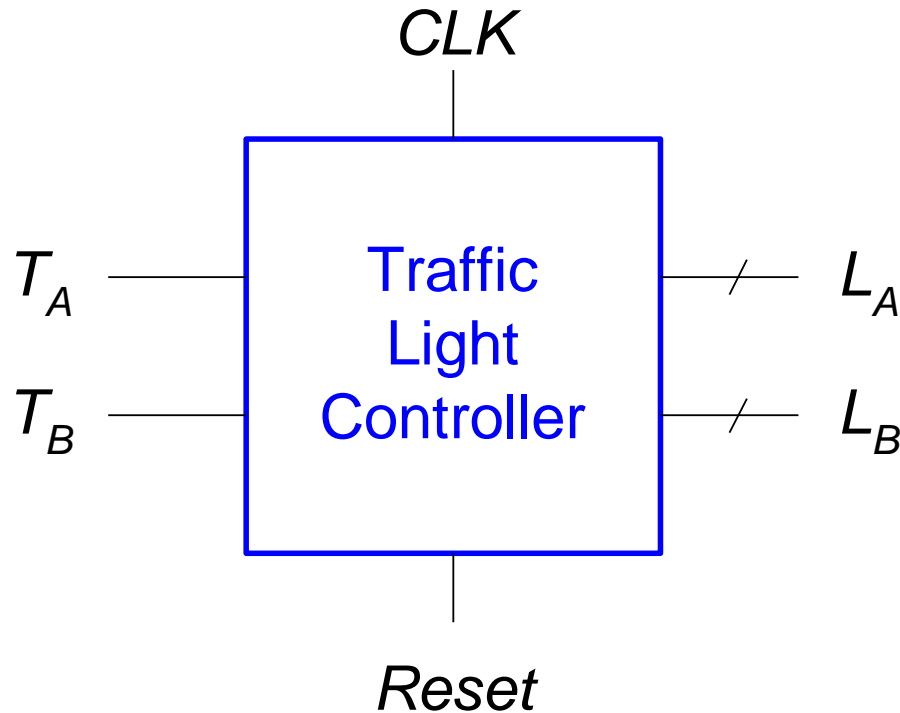
- “Smart” traffic light controller
 - **2 inputs:**
 - Traffic sensors: T_A , T_B (TRUE when there’s traffic)
 - **2 outputs:**
 - Lights: L_A , L_B (Red, Yellow, Green)
 - State can change every 5 seconds
 - Except if green and traffic, stay green



From H&H Section 3.4.1

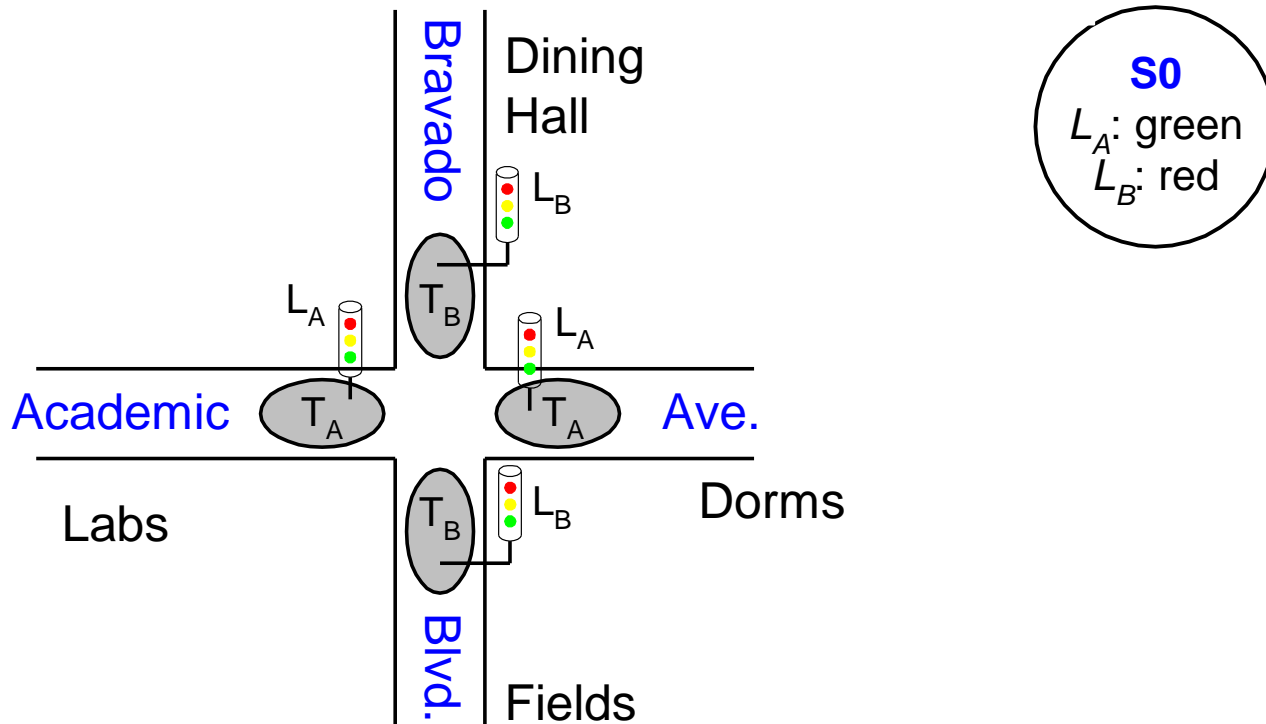
Finite State Machine Black Box

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



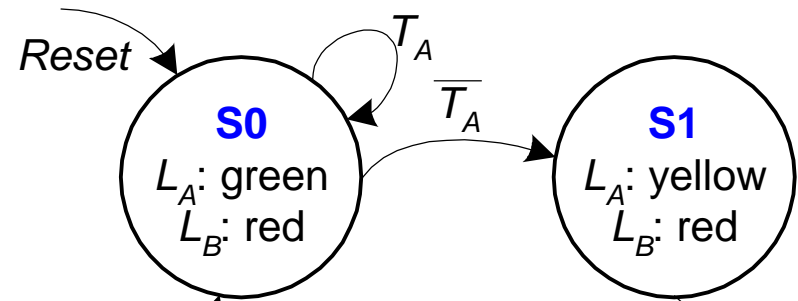
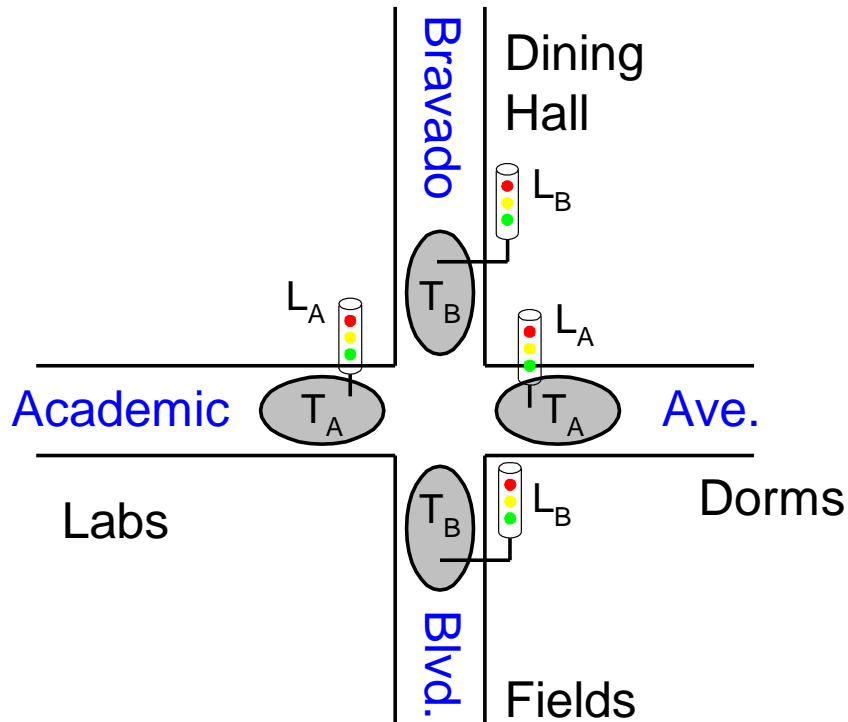
Finite State Machine Transition Diagram

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



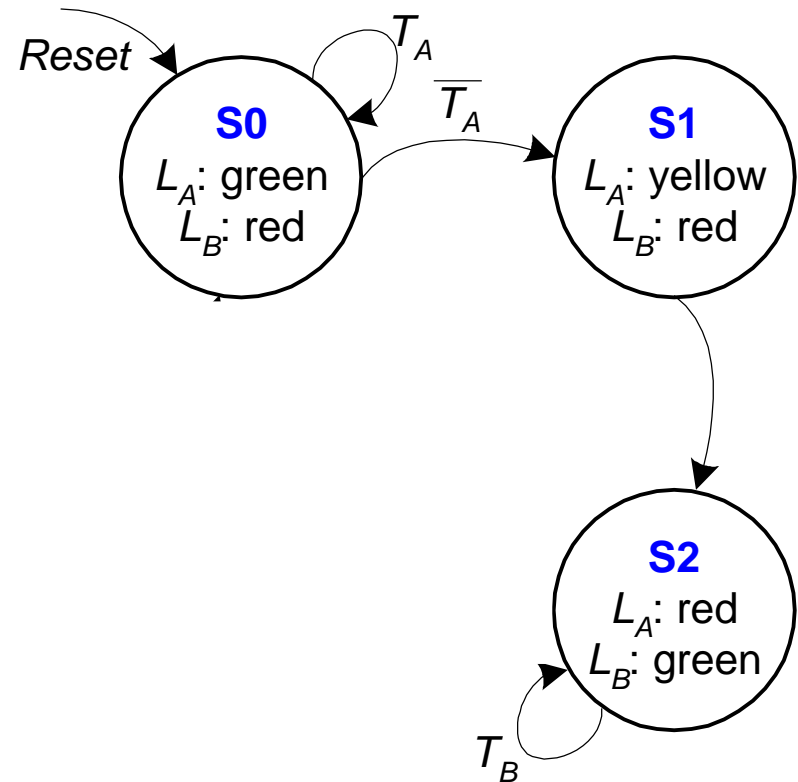
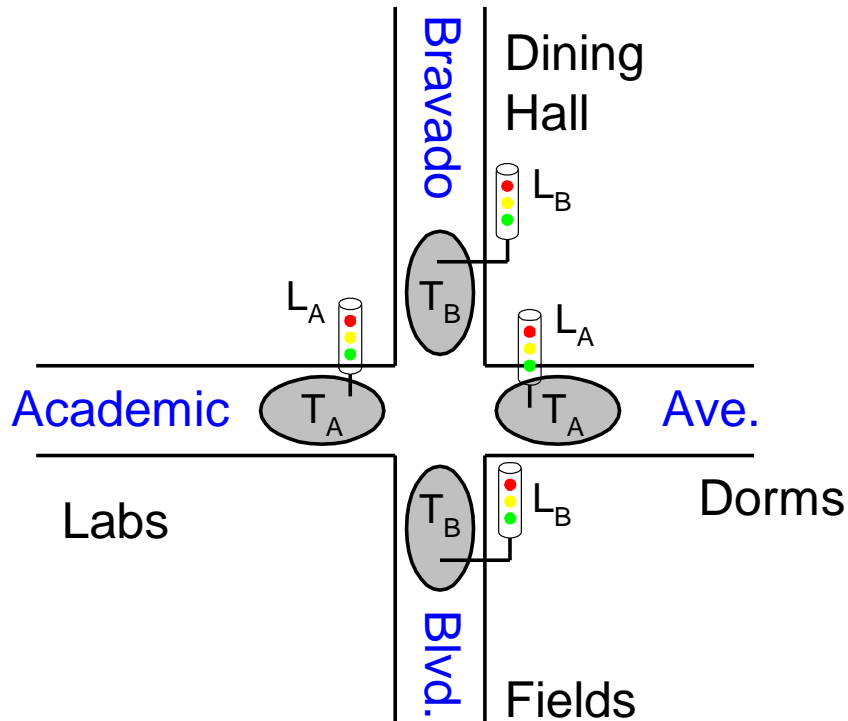
Finite State Machine Transition Diagram

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



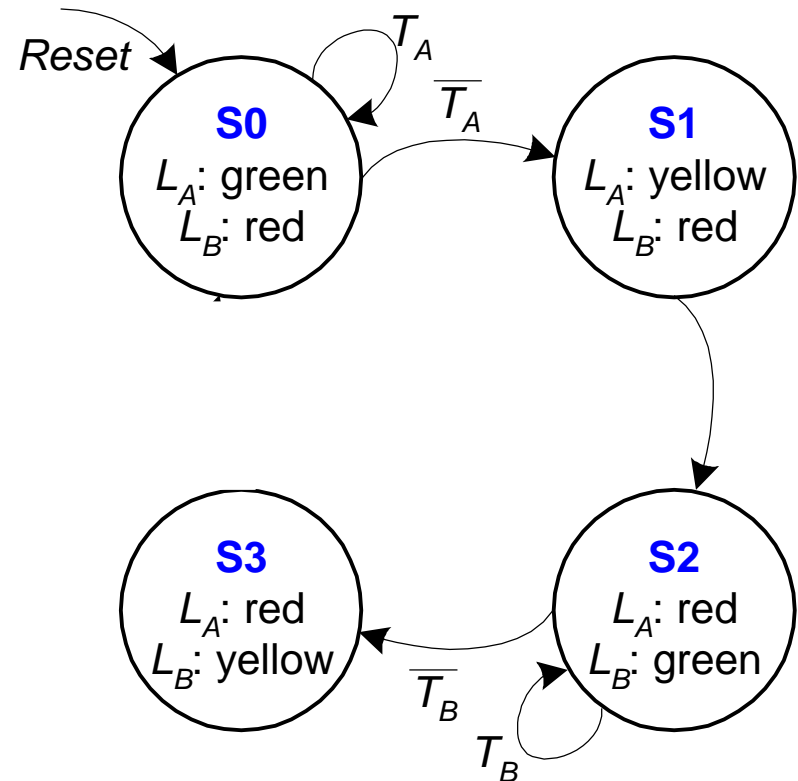
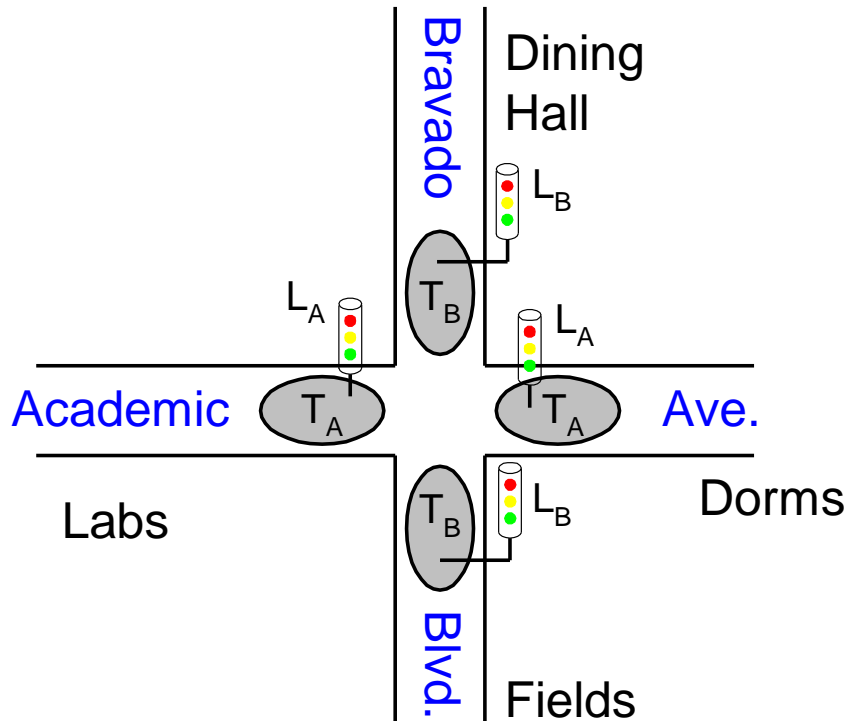
Finite State Machine Transition Diagram

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



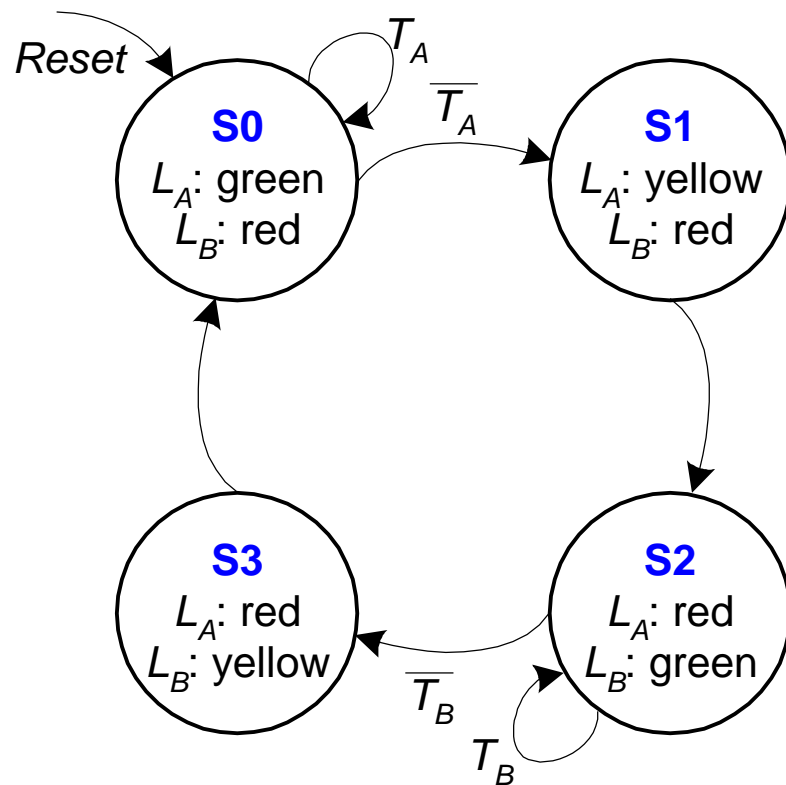
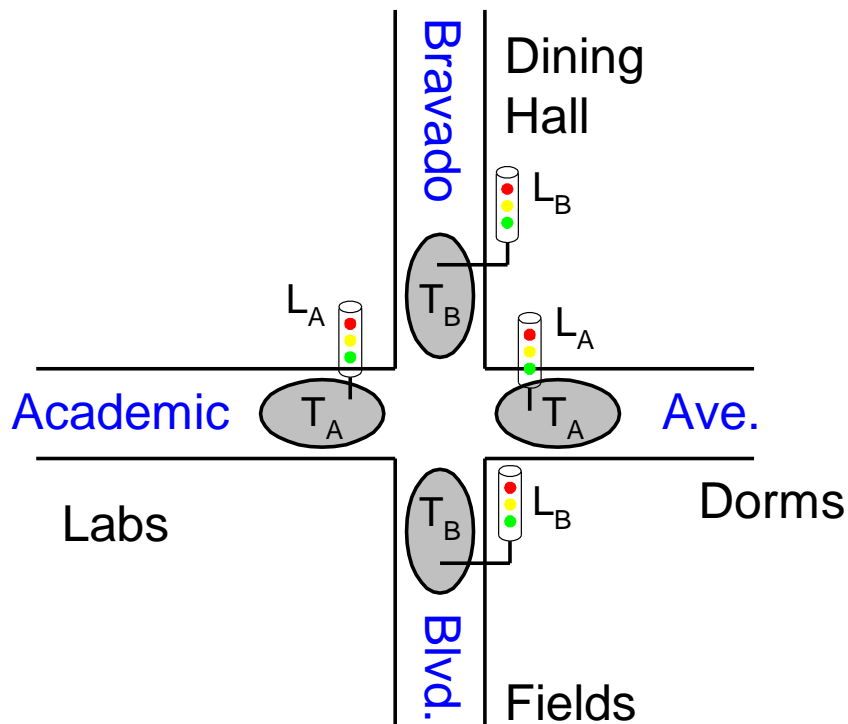
Finite State Machine Transition Diagram

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



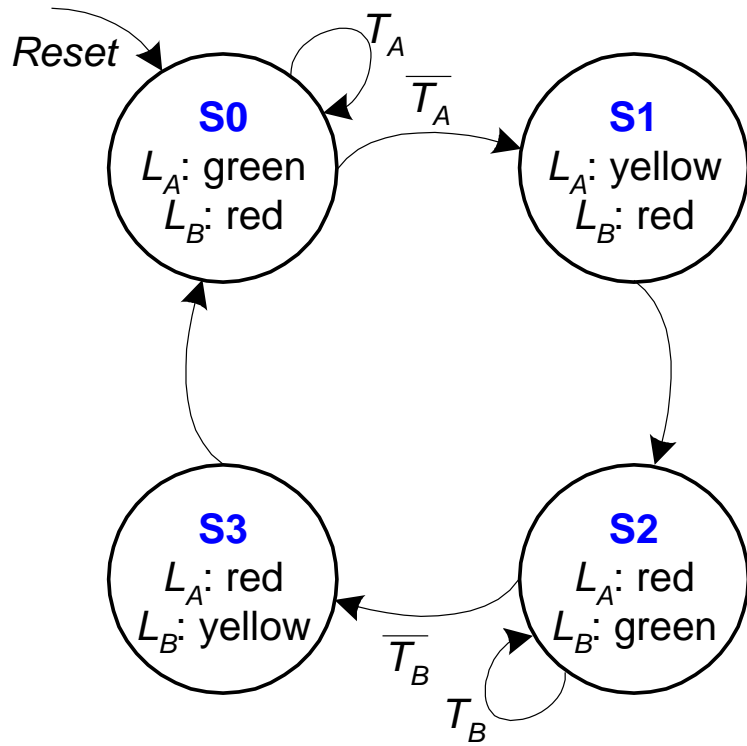
Finite State Machine Transition Diagram

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



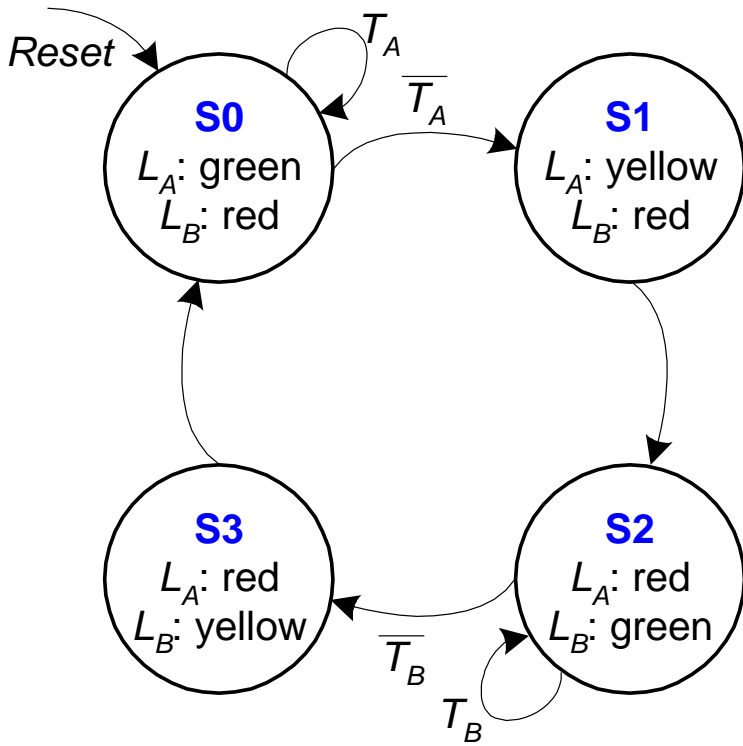
Finite State Machine: State Transition Table

FSM State Transition Table



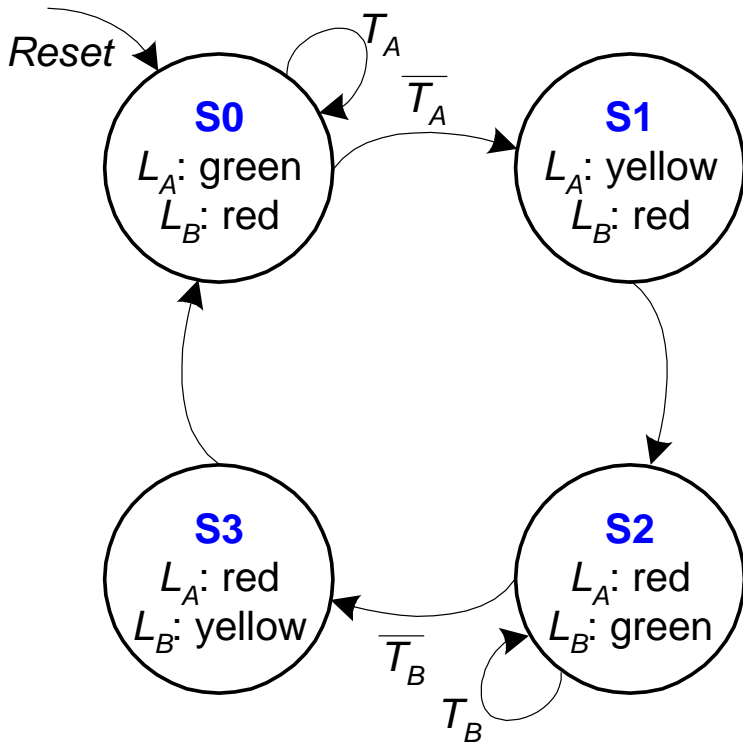
| Current State | Inputs | | Next State |
|---------------|--------|-------|------------|
| | T_A | T_B | |
| S | | | S' |
| S0 | 0 | X | |
| S0 | 1 | X | |
| S1 | X | X | |
| S2 | X | 0 | |
| S2 | X | 1 | |
| S3 | X | X | |

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 |

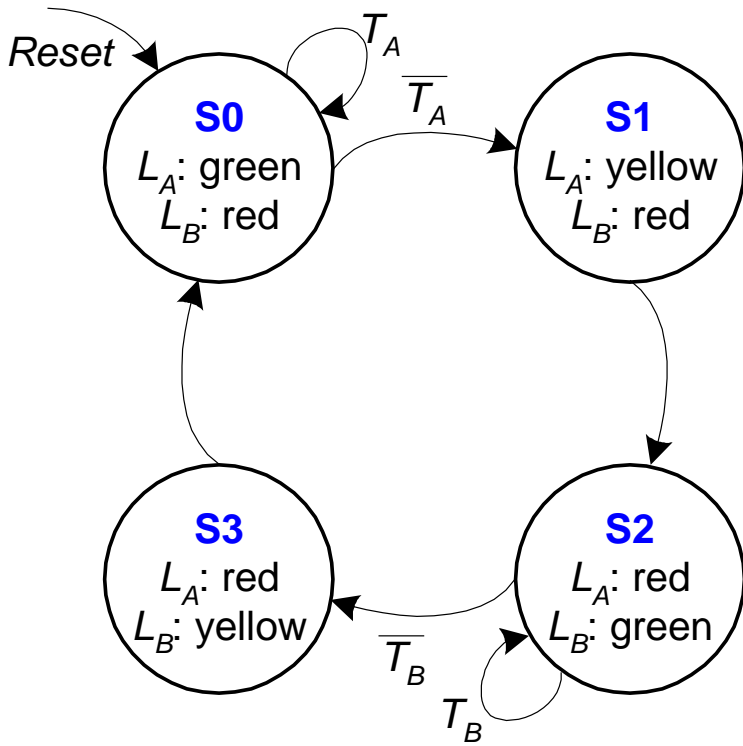
FSM State Transition Table



| Current State | Inputs | | Next State |
|---------------|--------|-------|------------|
| | T_A | T_B | |
| 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 |

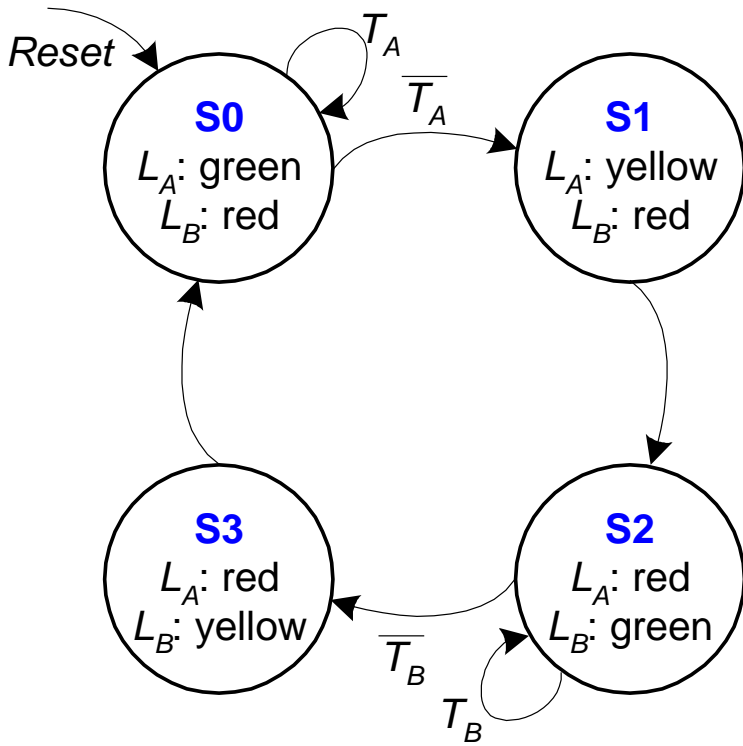
FSM State Transition Table



| 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

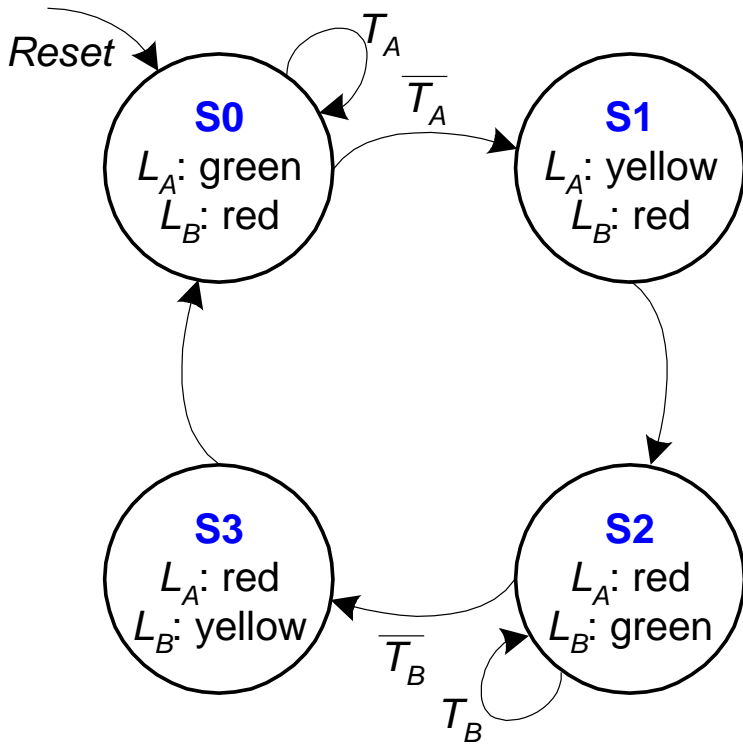


| Current State | | Inputs | | Next State | |
|---------------|-------|--------|-------|------------|--------|
| S_1 | S_0 | T_A | T_B | S'_1 | S'_0 |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0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

| State | Encoding |
|-------|----------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |
| S3 | 11 |

$S'_1 = ?$

FSM State Transition Table

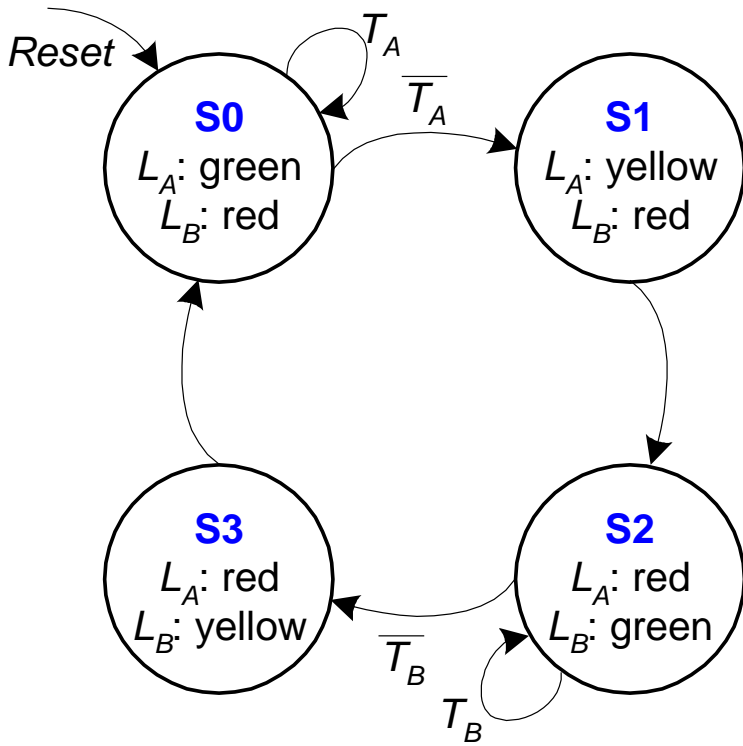


| Current State | | Inputs | | Next State | |
|----------------|----------------|----------------|----------------|-----------------|-----------------|
| S ₁ | S ₀ | T _A | T _B | S' ₁ | S' ₀ |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 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



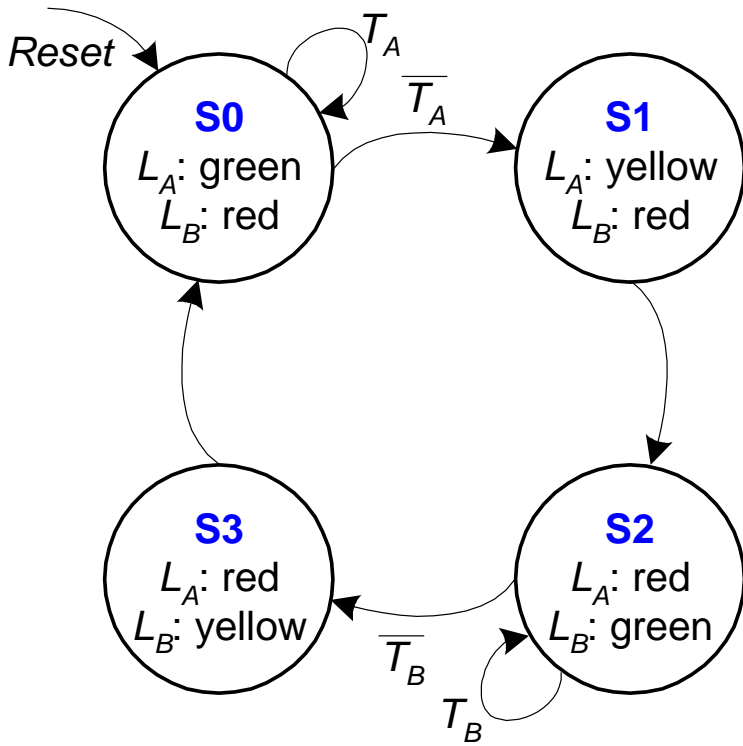
| Current State | | Inputs | | Next State | |
|----------------|----------------|----------------|----------------|-----------------|-----------------|
| S ₁ | S ₀ | T _A | T _B | S' ₁ | S' ₀ |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

| State | Encoding |
|-------|----------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |
| S3 | 11 |

$$S'_1 = (\bar{S}_1 \cdot S_0) + (S_1 \cdot \bar{S}_0 \cdot \bar{T}_B) + (S_1 \cdot \bar{S}_0 \cdot T_B)$$

$$S'_0 = ?$$

FSM State Transition Table



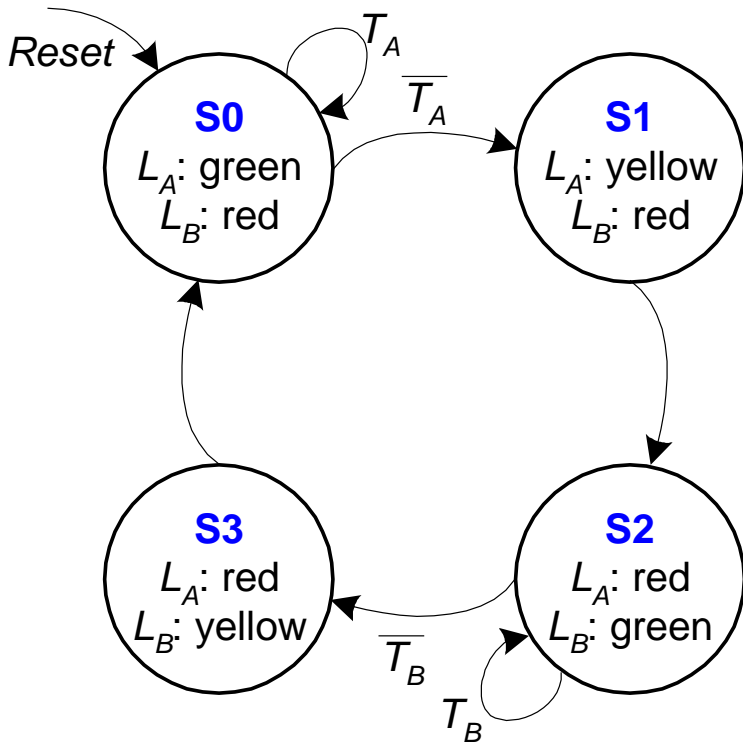
| Current State | | Inputs | | Next State | |
|----------------|----------------|----------------|----------------|-----------------|-----------------|
| S ₁ | S ₀ | T _A | T _B | S' ₁ | S' ₀ |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

| State | Encoding |
|-------|----------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |
| S3 | 11 |

$$S'_1 = (\bar{S}_1 \cdot S_0) + (S_1 \cdot \bar{S}_0 \cdot \bar{T}_B) + (S_1 \cdot \bar{S}_0 \cdot T_B)$$

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

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 |

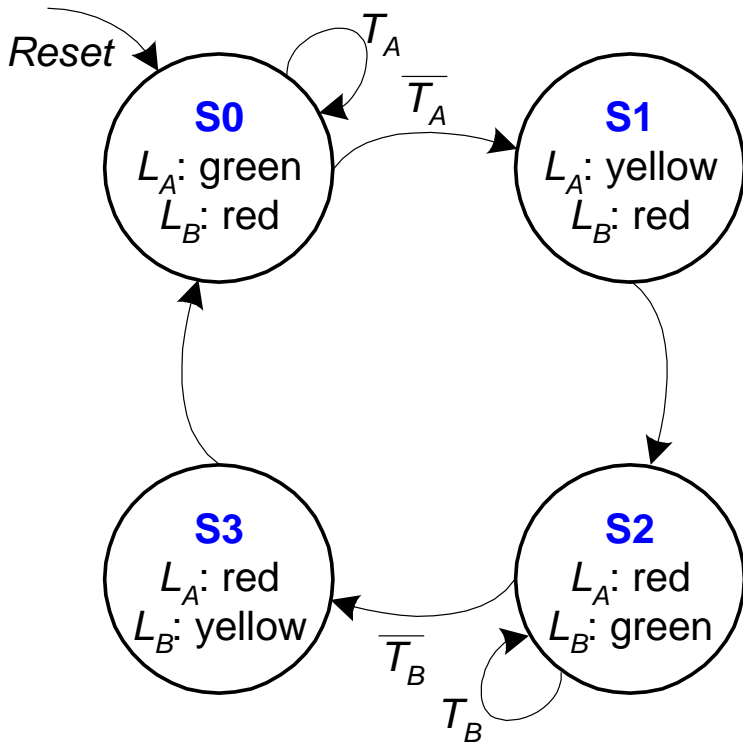
| State | Encoding |
|-------|----------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |
| S3 | 11 |

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

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

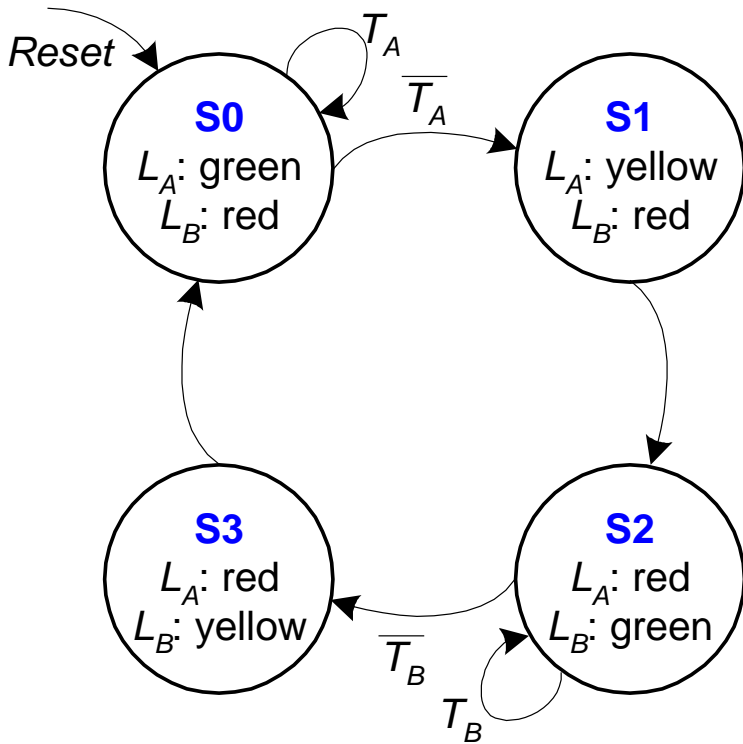
Finite State Machine: Output Table

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 |

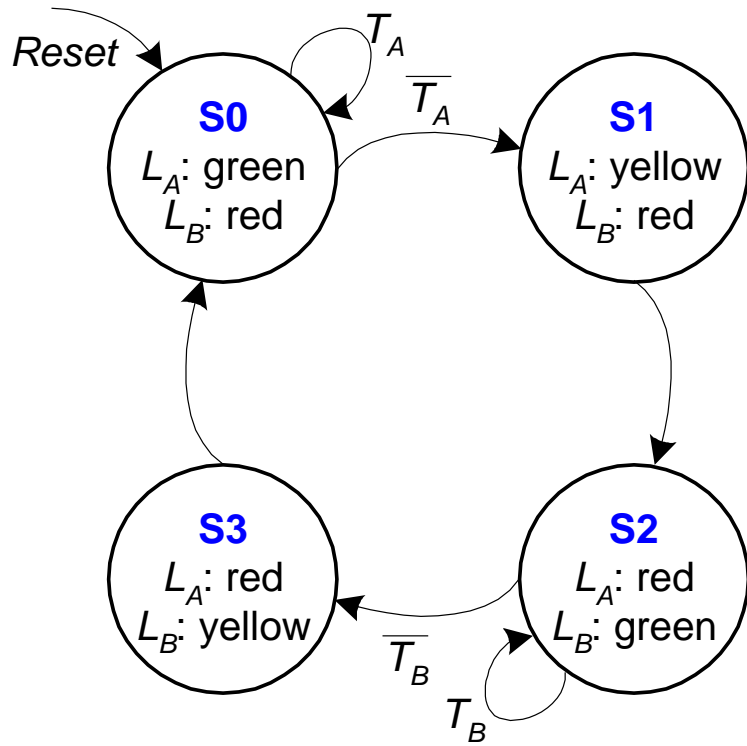
FSM Output Table



| 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

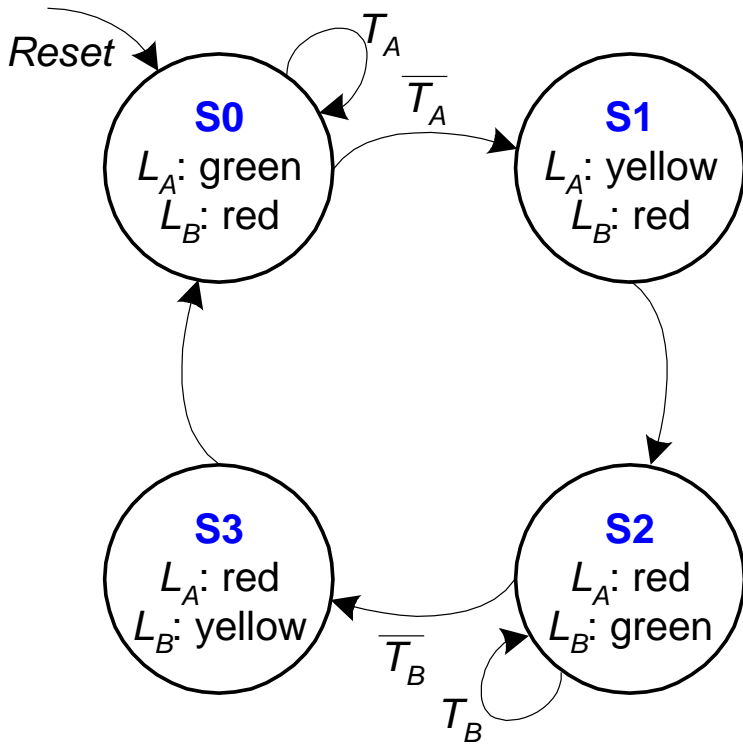


$$L_{A1} = S_1$$

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

| Output | Encoding |
|--------|----------|
| green | 00 |
| yellow | 01 |
| red | 10 |

FSM Output Table



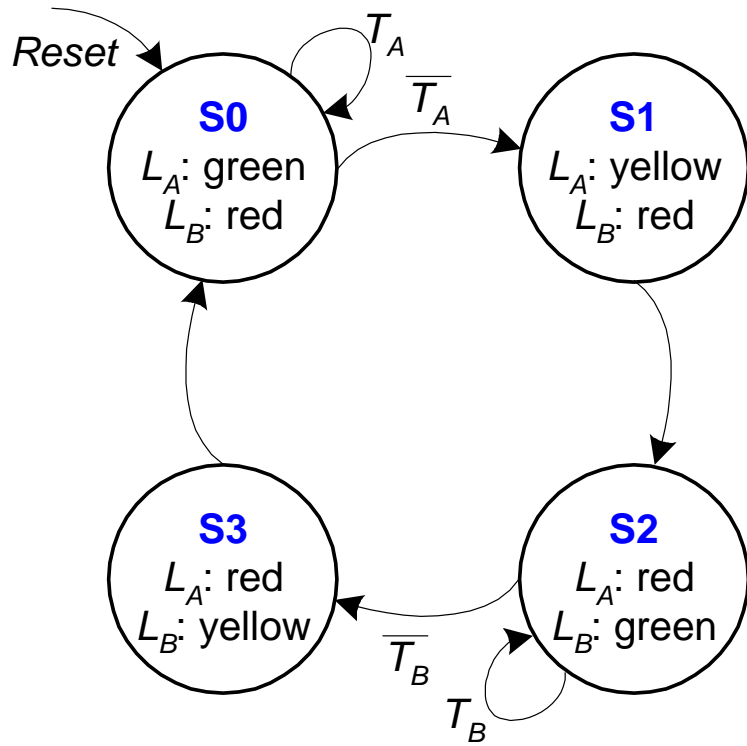
$$L_{A1} = S_1$$

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

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

| Output | Encoding |
|--------|----------|
| green | 00 |
| yellow | 01 |
| red | 10 |

FSM Output Table



| Current State | | Outputs | | | |
|----------------|----------------|-----------------|-----------------|-----------------|-----------------|
| S ₁ | S ₀ | L _{A1} | L _{A0} | L _{B1} | L _{B0} |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

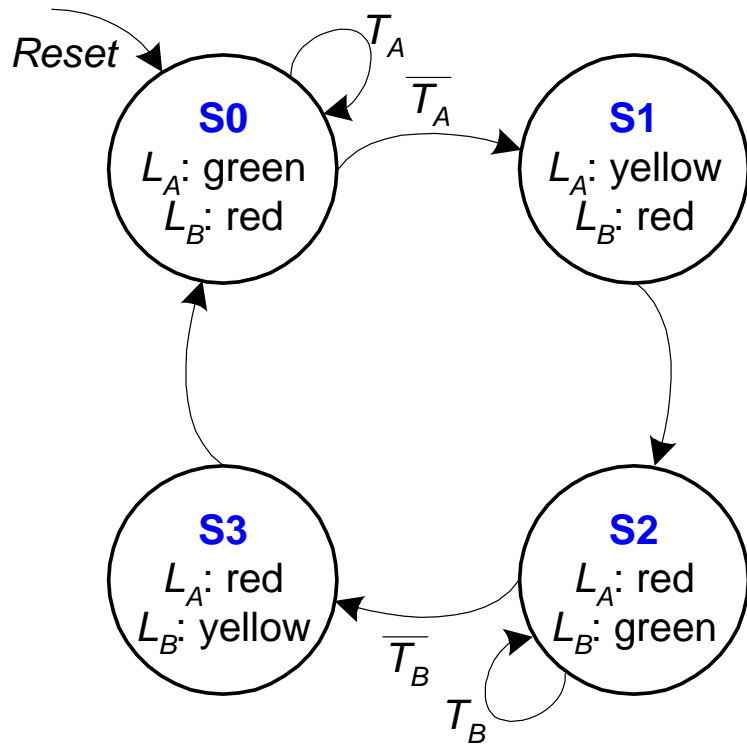
$$L_{A1} = S_1$$

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

$$L_{B1} = \overline{S_1}$$

| 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 |

$$L_{A1} = S_1$$

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

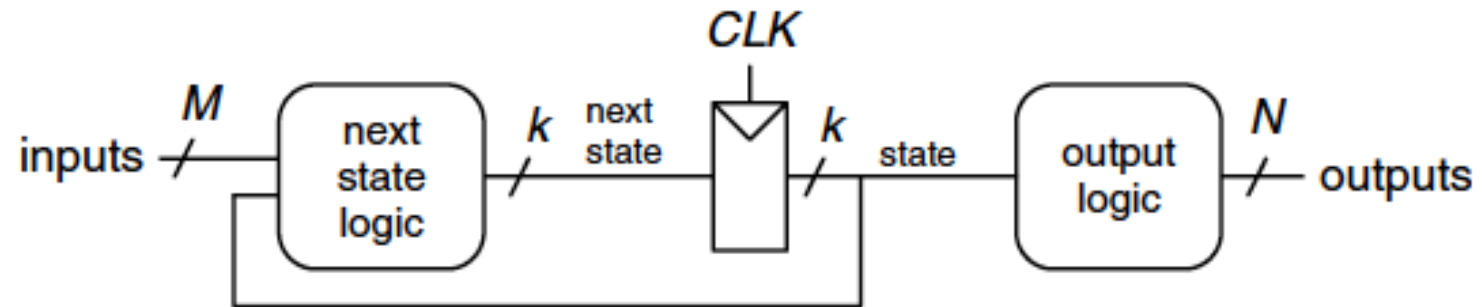
$$L_{B1} = \overline{S_1}$$

$$L_{B0} = S_1 \cdot S_0$$

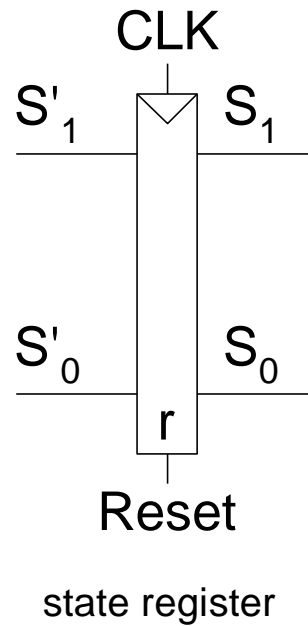
| Output | Encoding |
|--------|----------|
| green | 00 |
| yellow | 01 |
| red | 10 |

Finite State Machine: Schematic

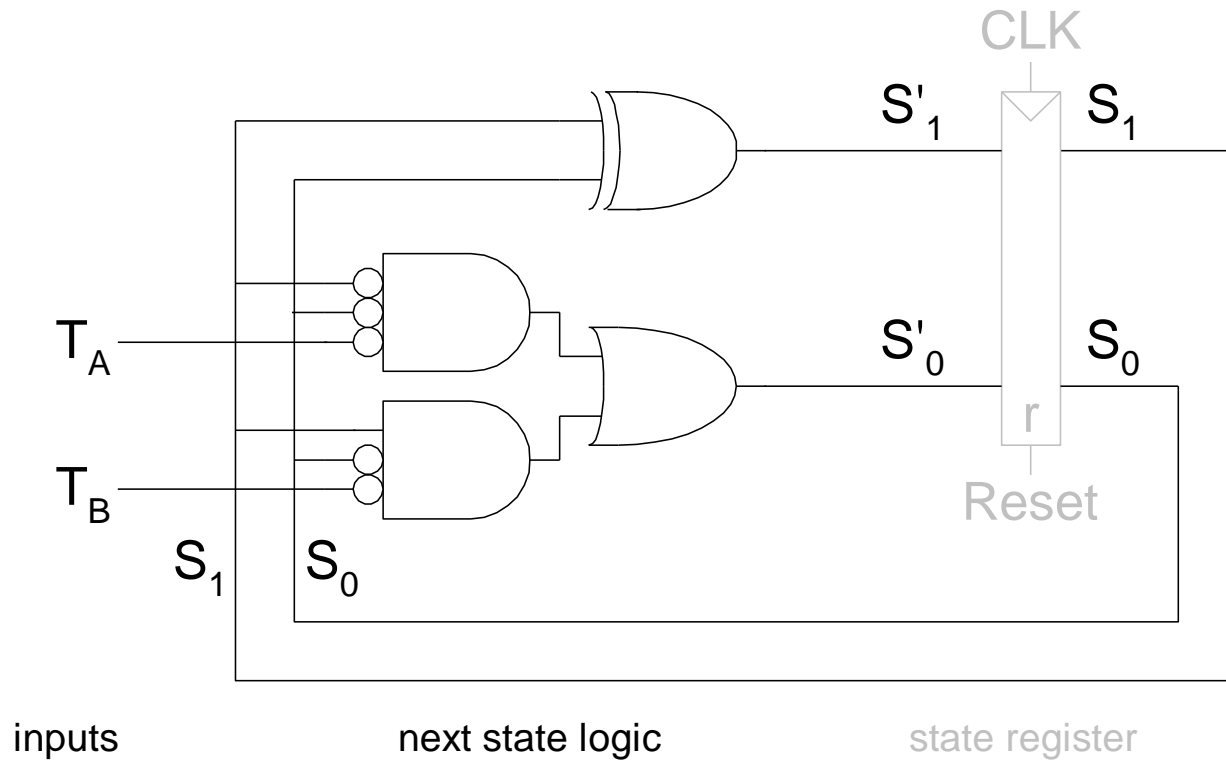
FSM Schematic: State Register



FSM Schematic: State Register



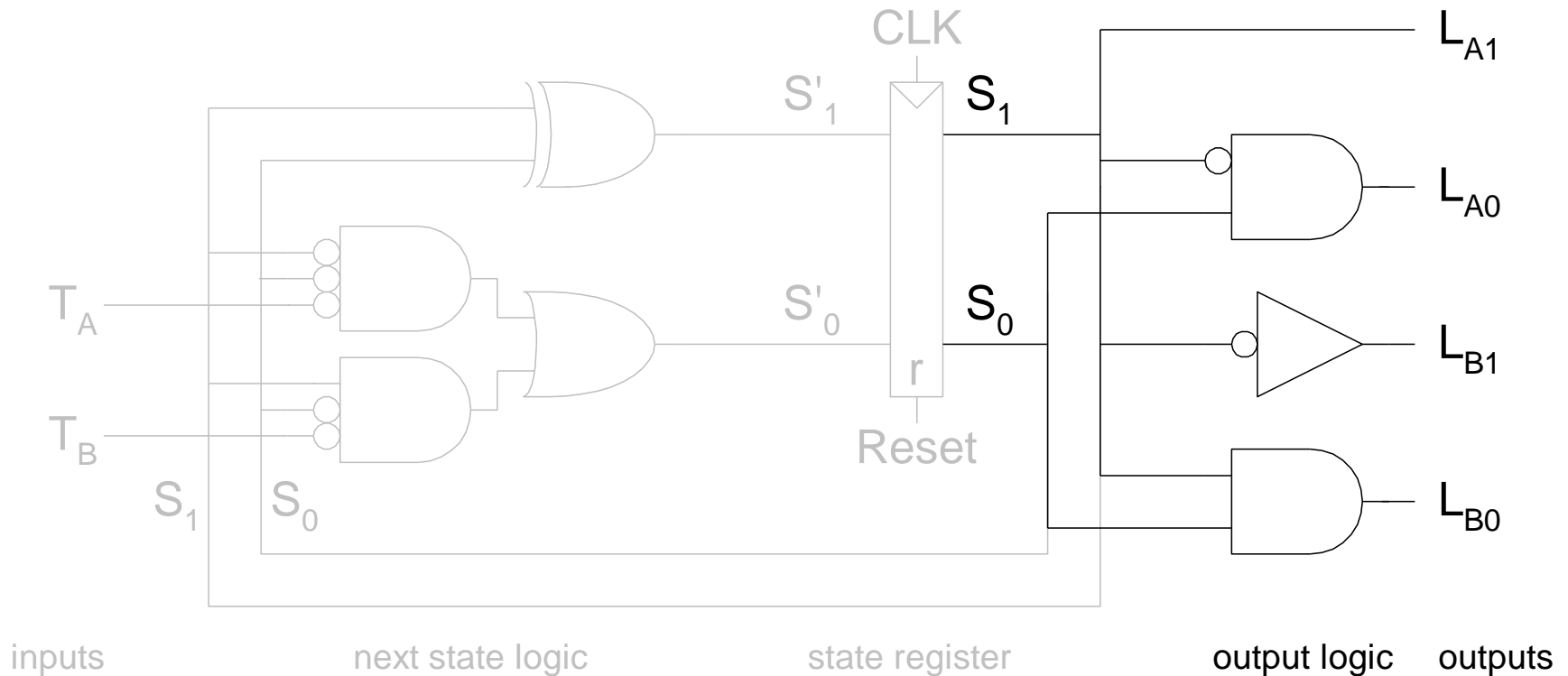
FSM Schematic: Next State Logic



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

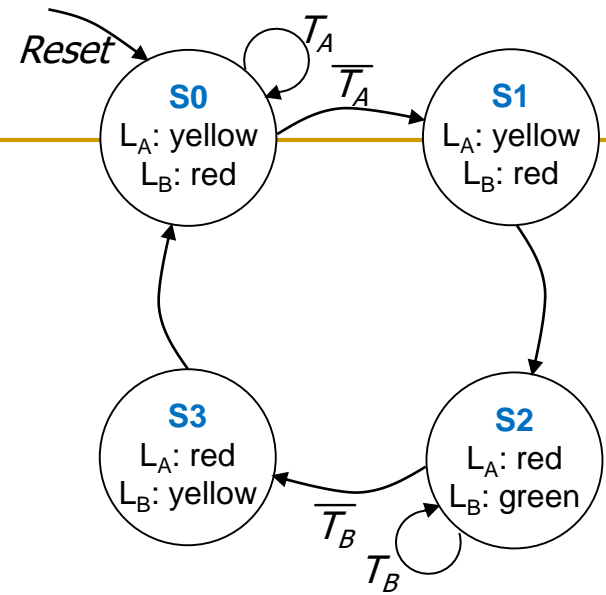
$$S'_0 = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$$

FSM Schematic: Output Logic



$$\begin{aligned}L_{A1} &= \overline{S_1} \\L_{A0} &= \overline{S_1} \cdot S_0 \\L_{B1} &= \overline{S_1} \\L_{B0} &= S_1 \cdot S_0\end{aligned}$$

FSM Timing Diagram



CLK_

Reset_

T_A _

T_B _

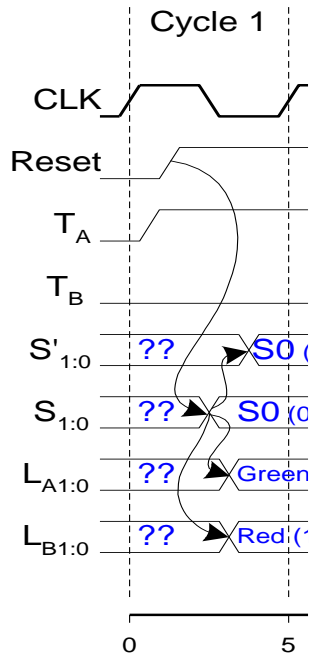
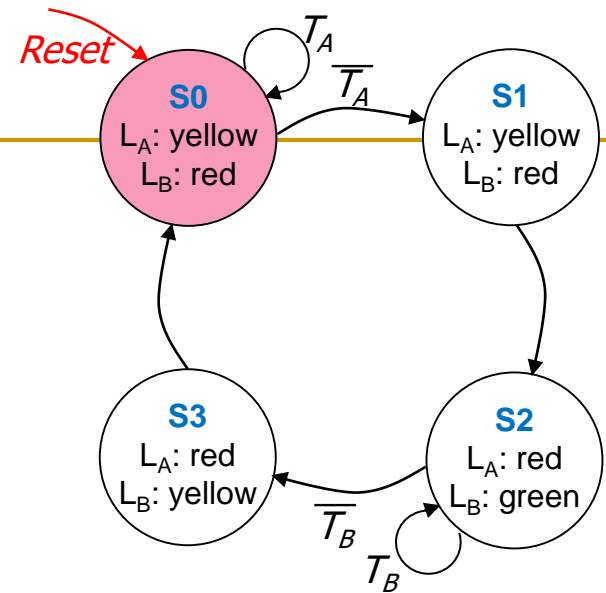
S'_{1:0} _

S_{1:0} _

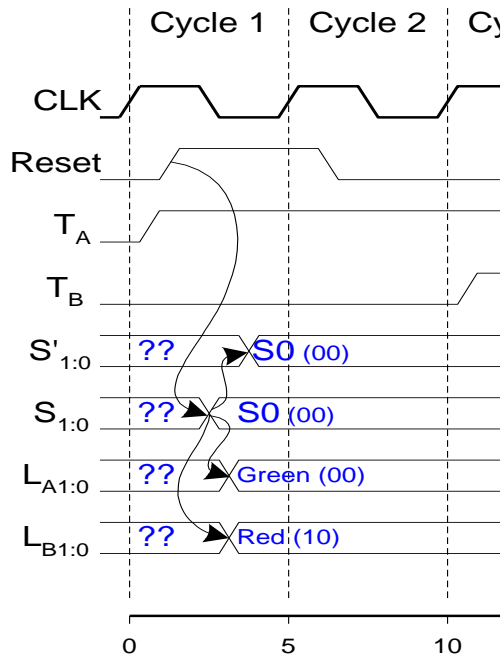
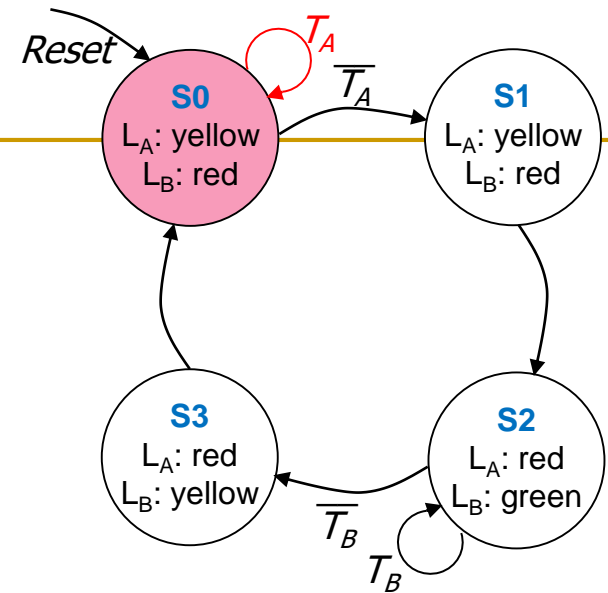
L_{A1:0} _

L_{B1:0} _

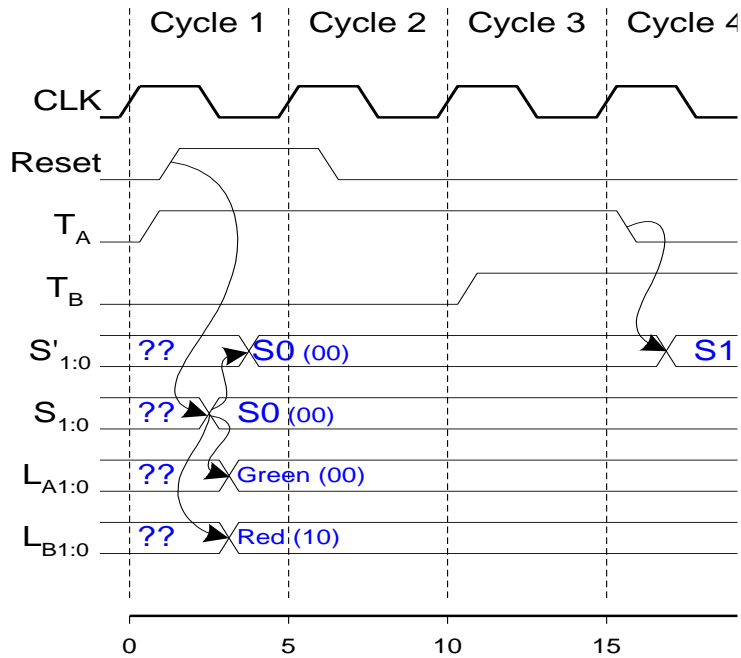
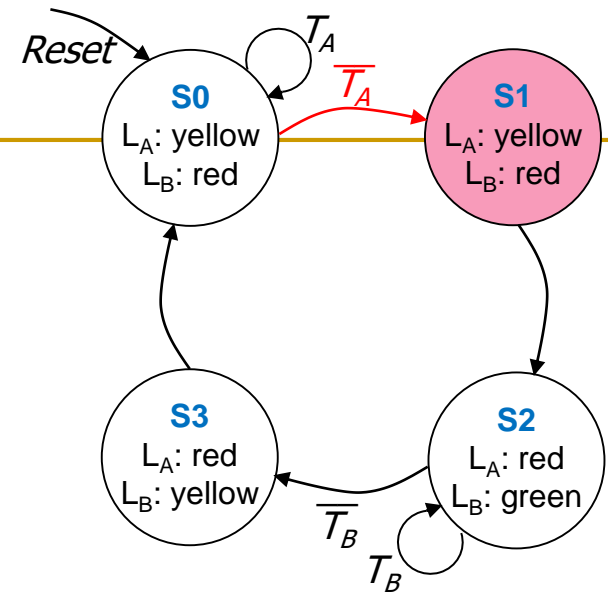
FSM Timing Diagram



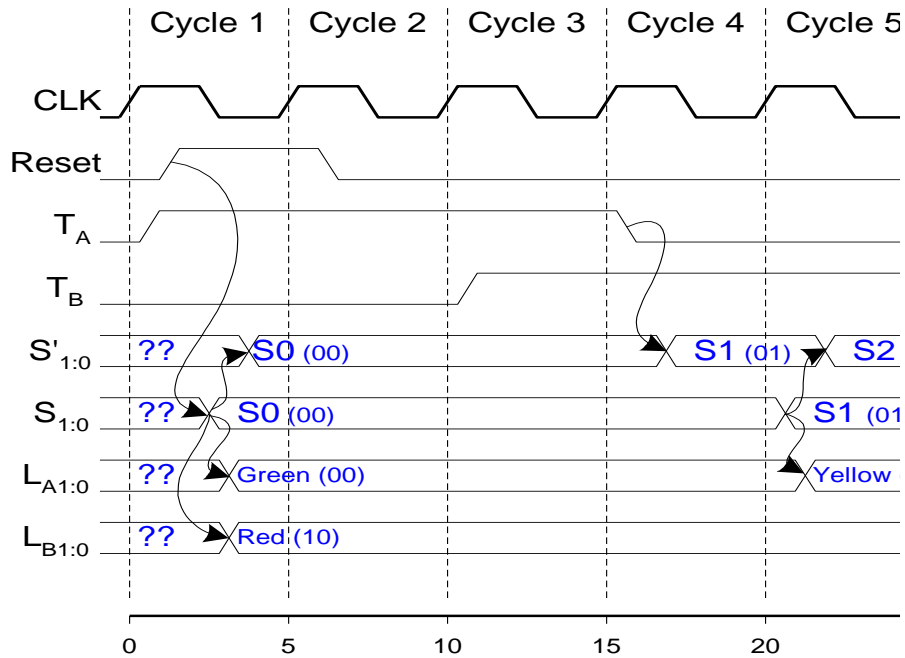
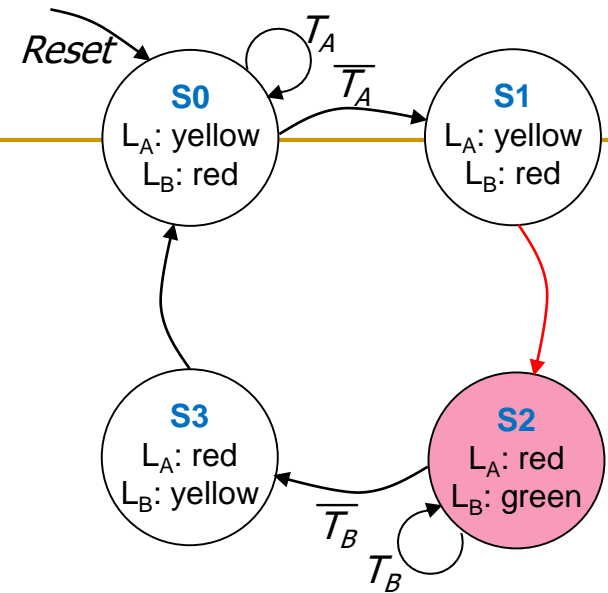
FSM Timing Diagram



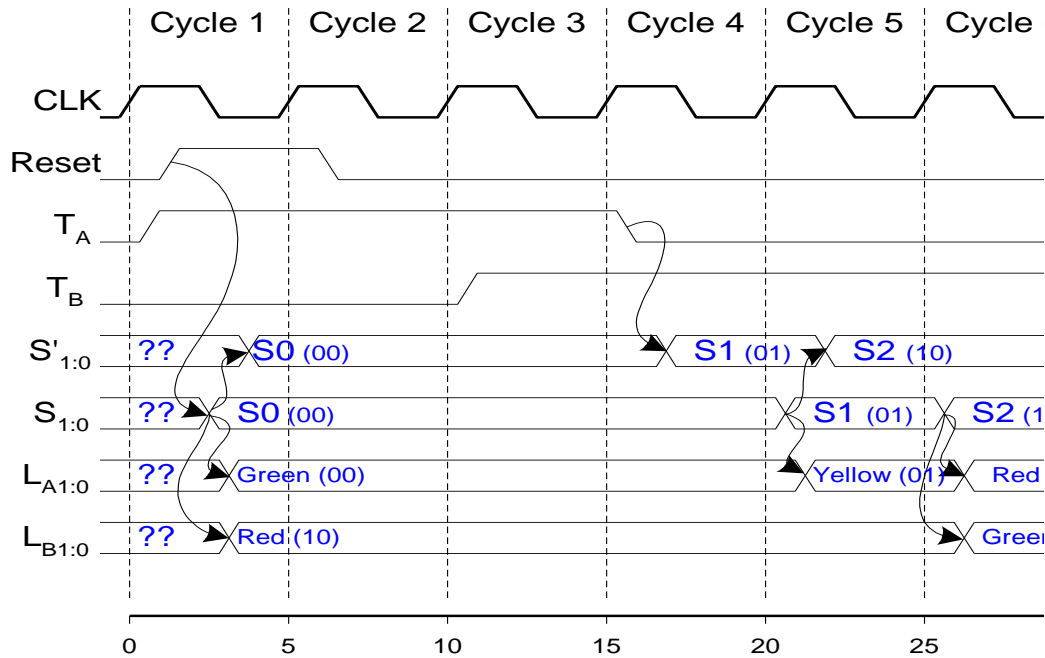
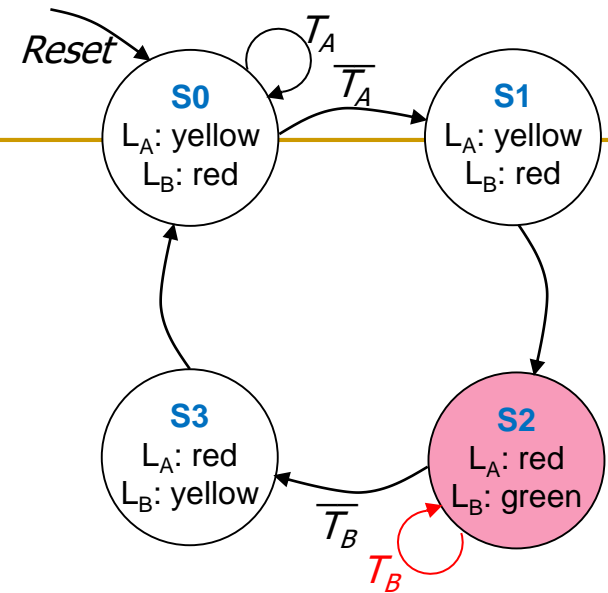
FSM Timing Diagram



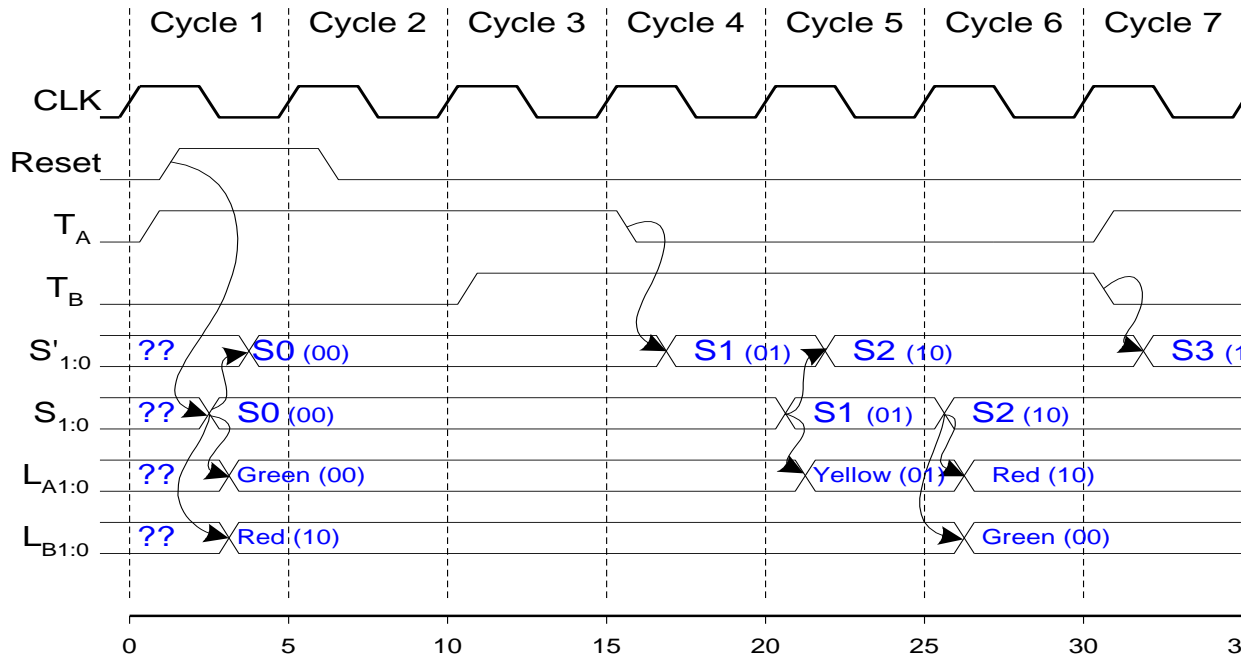
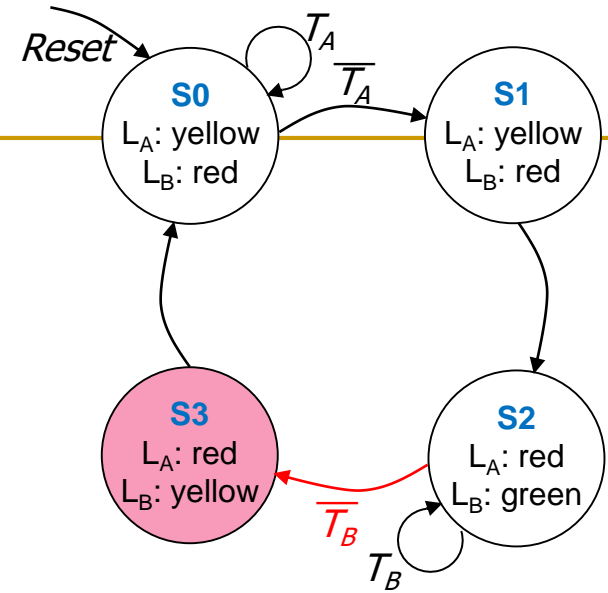
FSM Timing Diagram



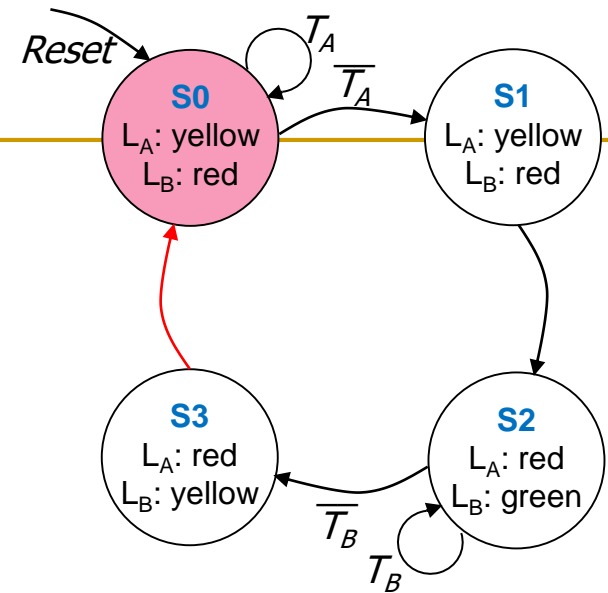
FSM Timing Diagram



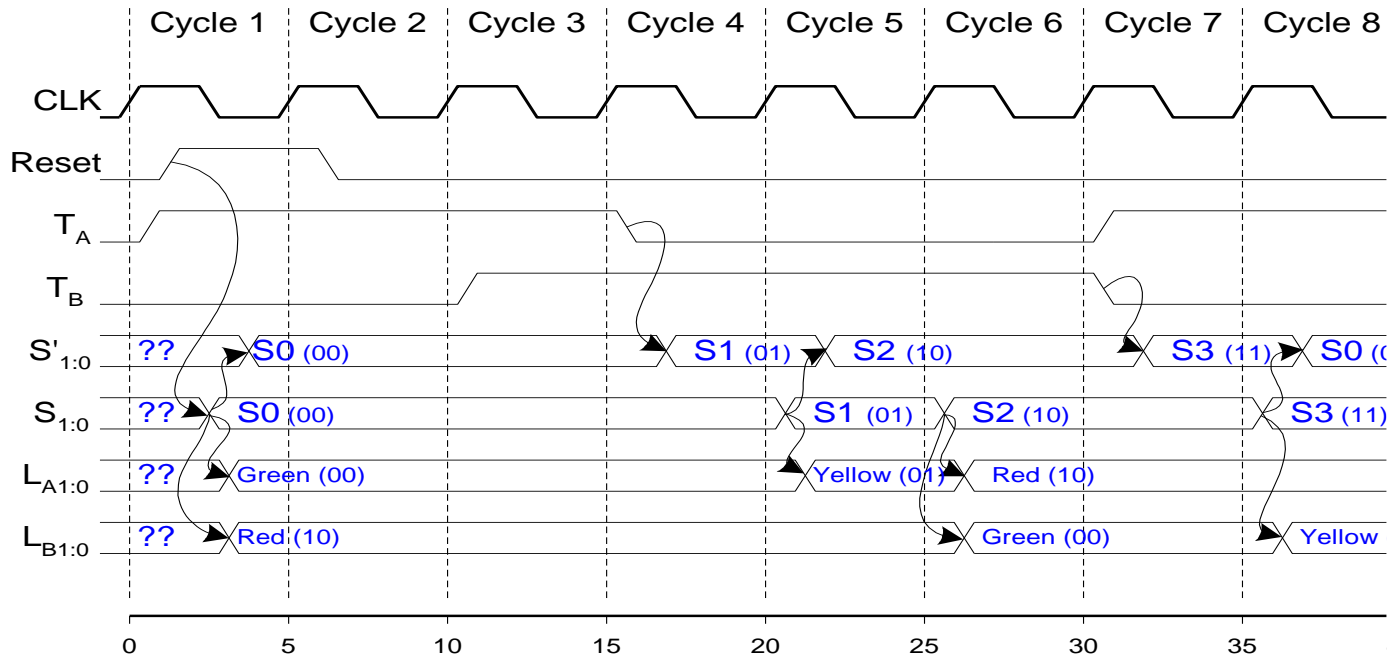
FSM Timing Diagram



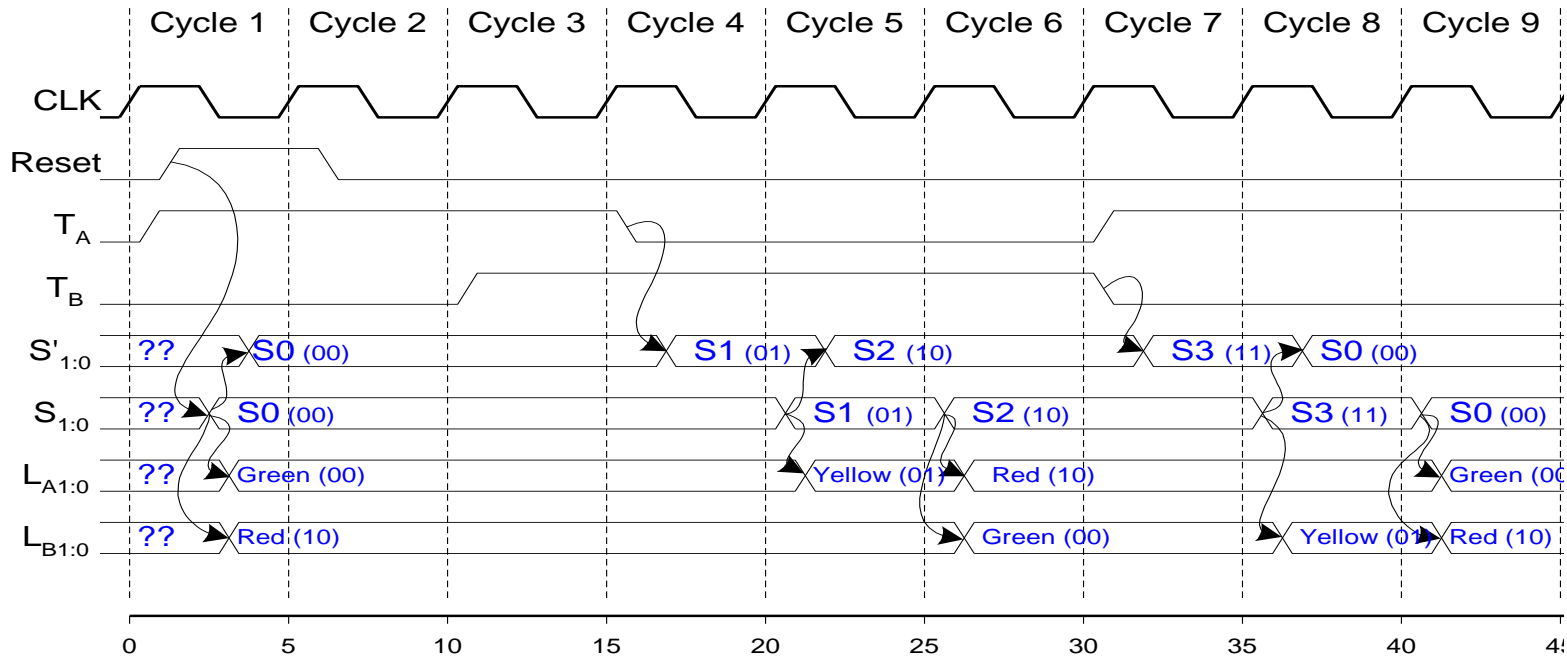
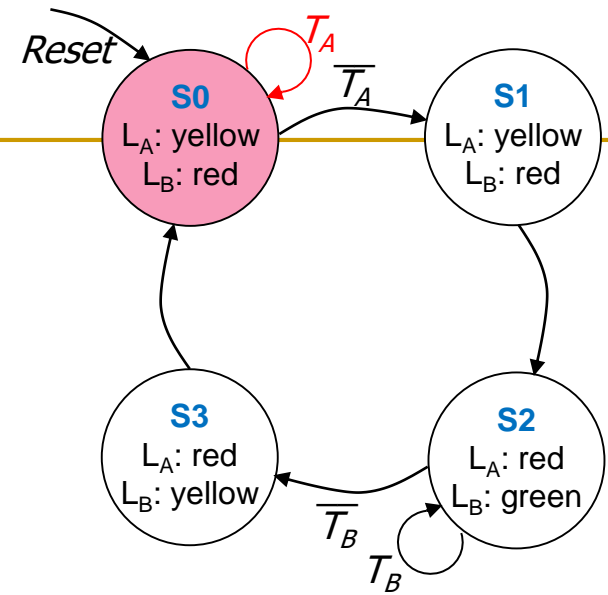
FSM Timing Diagram



This is from H&H Section 3.4.1

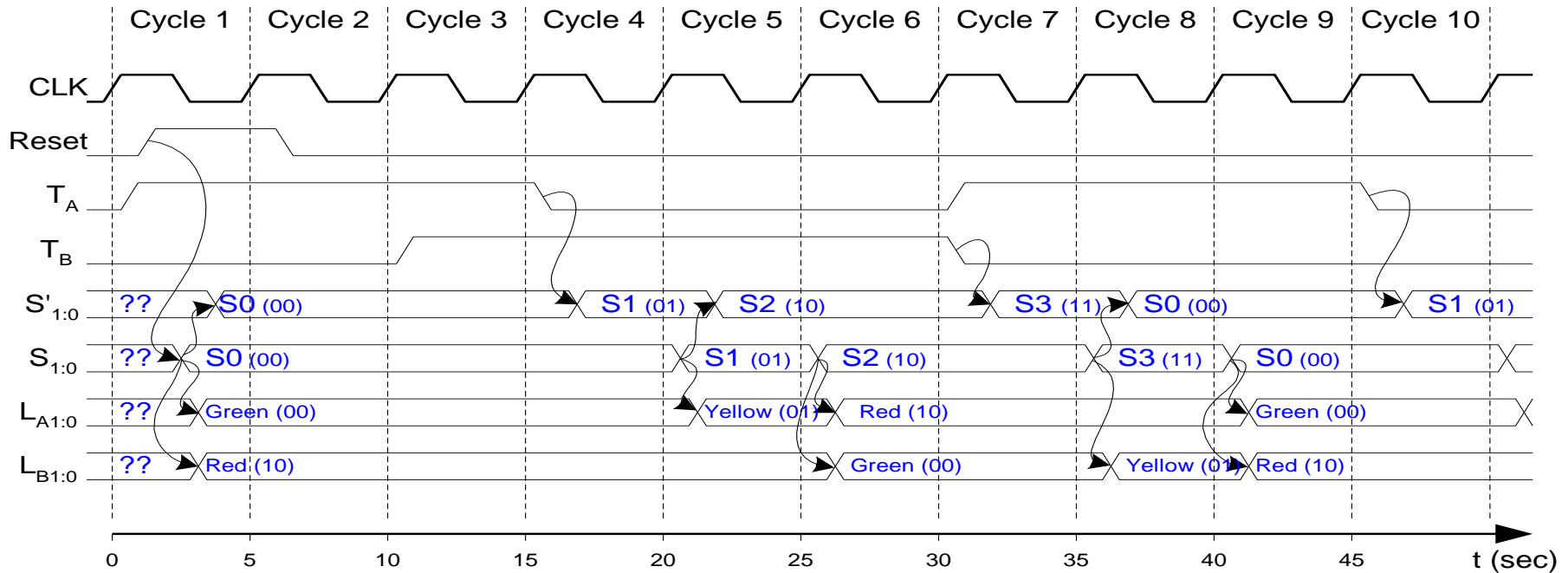
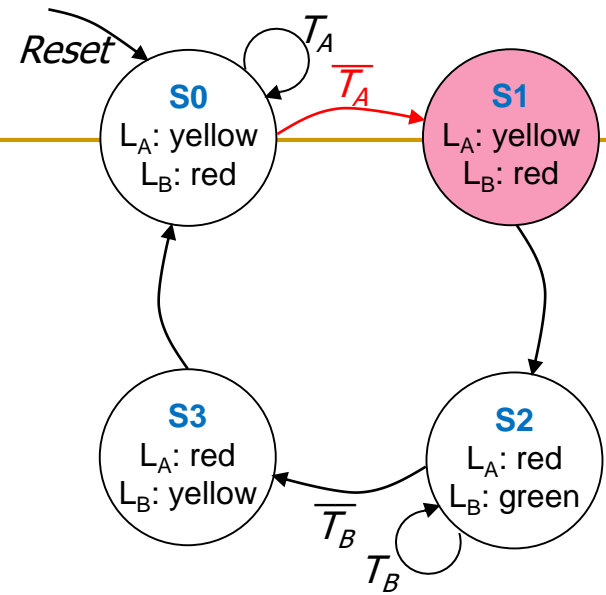


FSM Timing Diagram



FSM Timing Diagram

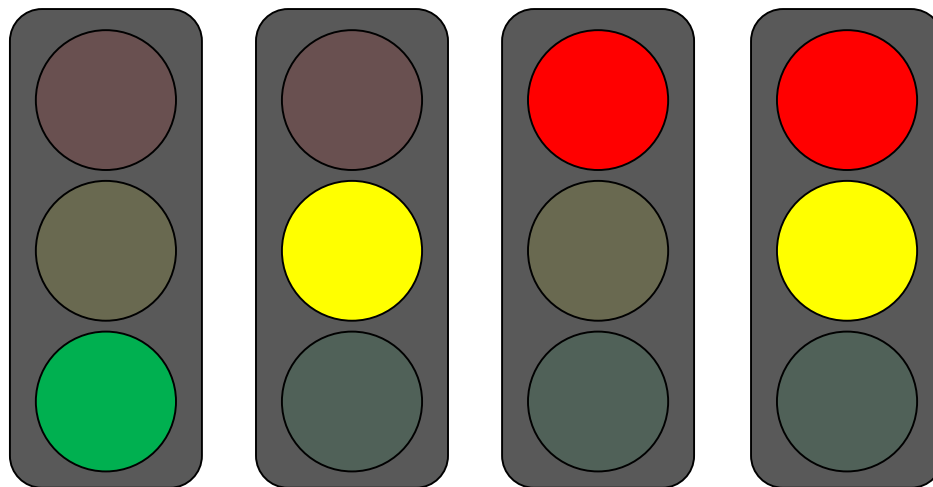
See H&H Chapter 3.4



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
 - Bit₀ encodes **green** light output,
 - Bit₁ encodes **yellow** light output
 - Bit₂ 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

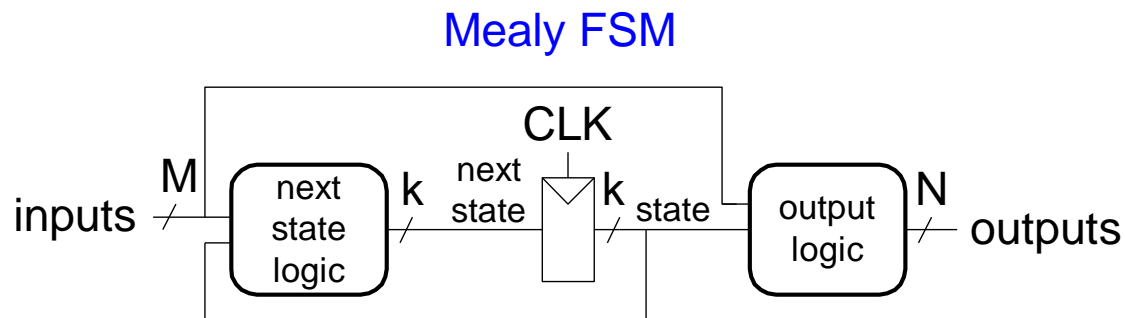
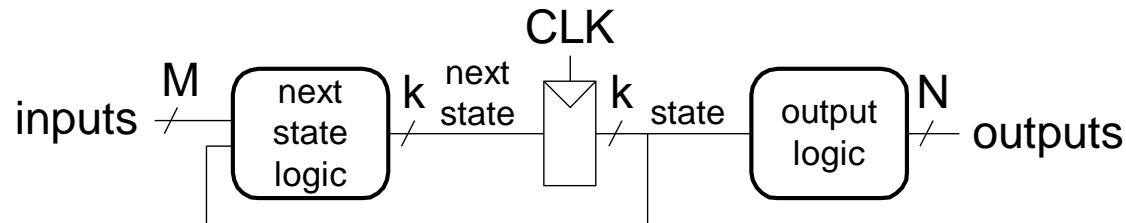
The **designer** must **carefully** choose an encoding scheme to **optimize** the design under given constraints

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

Moore vs. Mealy Machines

Recall: Moore vs. Mealy FSMs

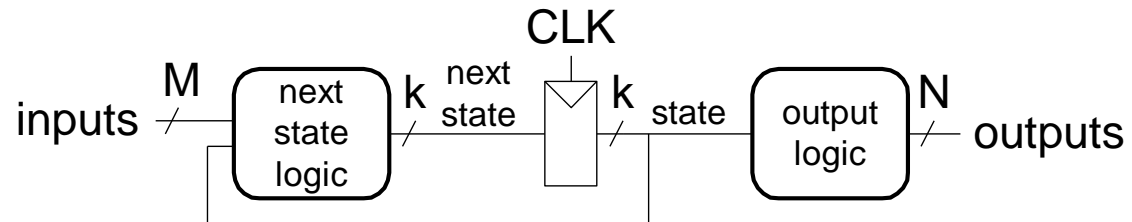
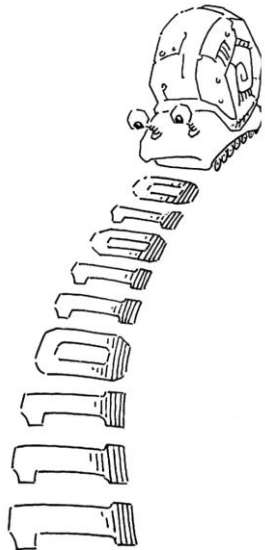
- Next state is determined by the current state and the inputs
- Two types of finite state machines differ in the **output logic**:
 - **Moore FSM**: outputs depend only on the current state
 - **Mealy FSM**: outputs depend on the current state and the inputs



Moore vs. Mealy FSM Examples

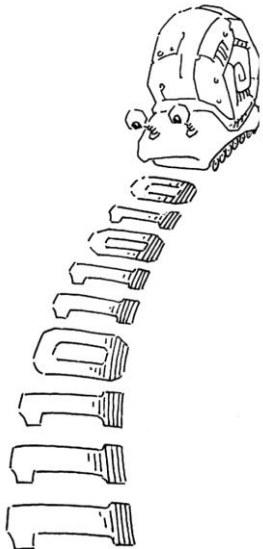
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are **1101**.
- Design Moore and Mealy FSMs of the snail's brain.

Moore FSM

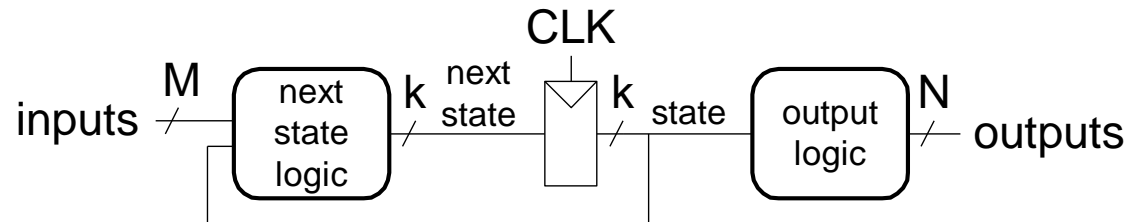


Moore vs. Mealy FSM Examples

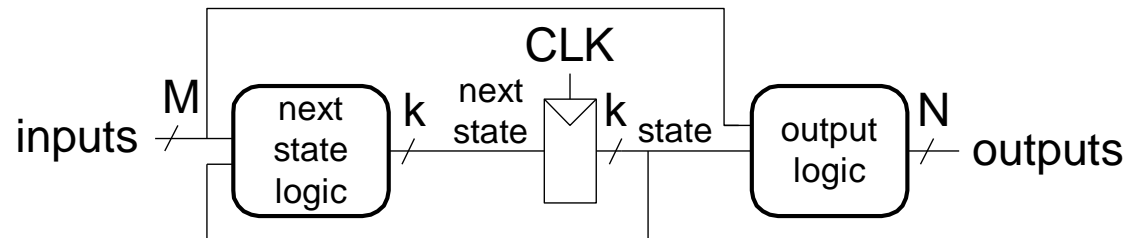
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are **1101**.
- Design Moore and Mealy FSMs of the snail's brain.



Moore FSM

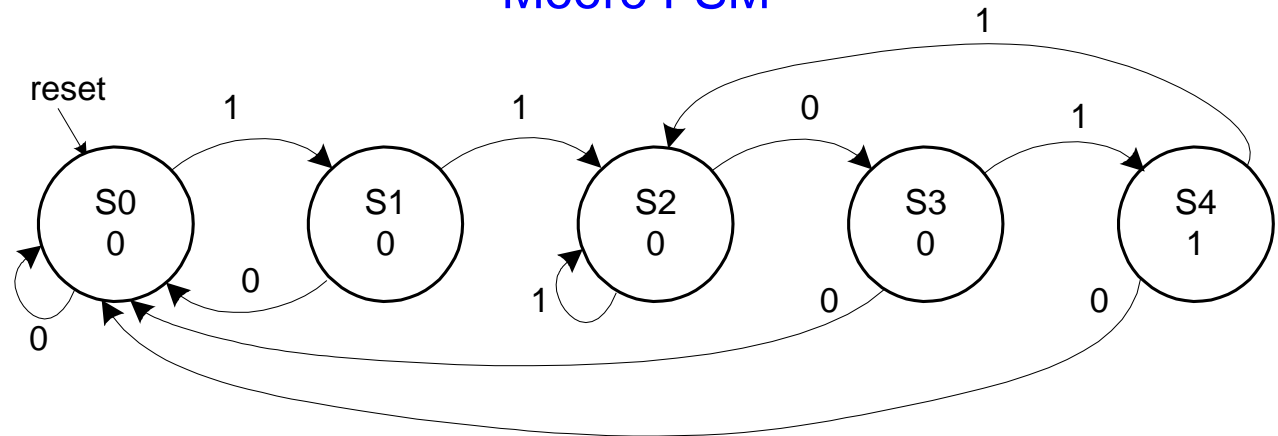


Mealy FSM



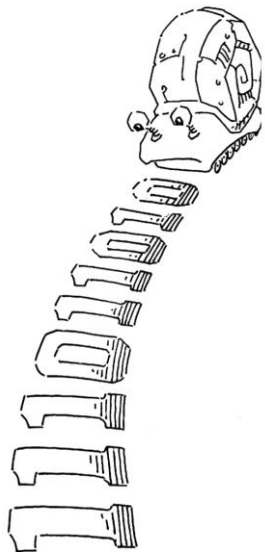
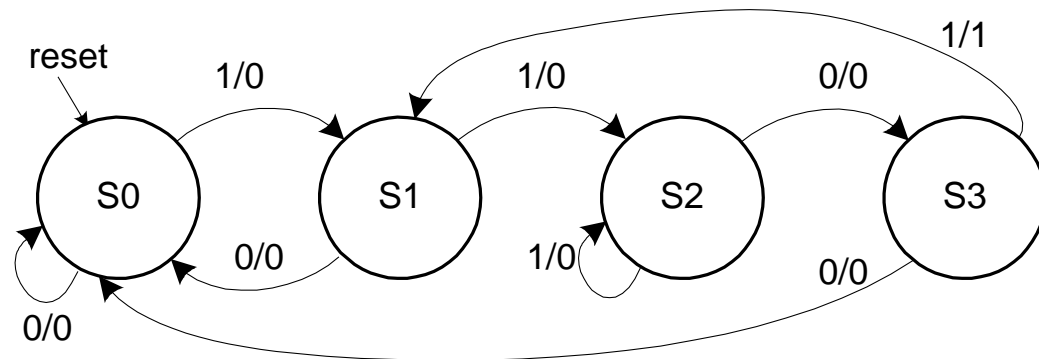
State Transition Diagrams

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

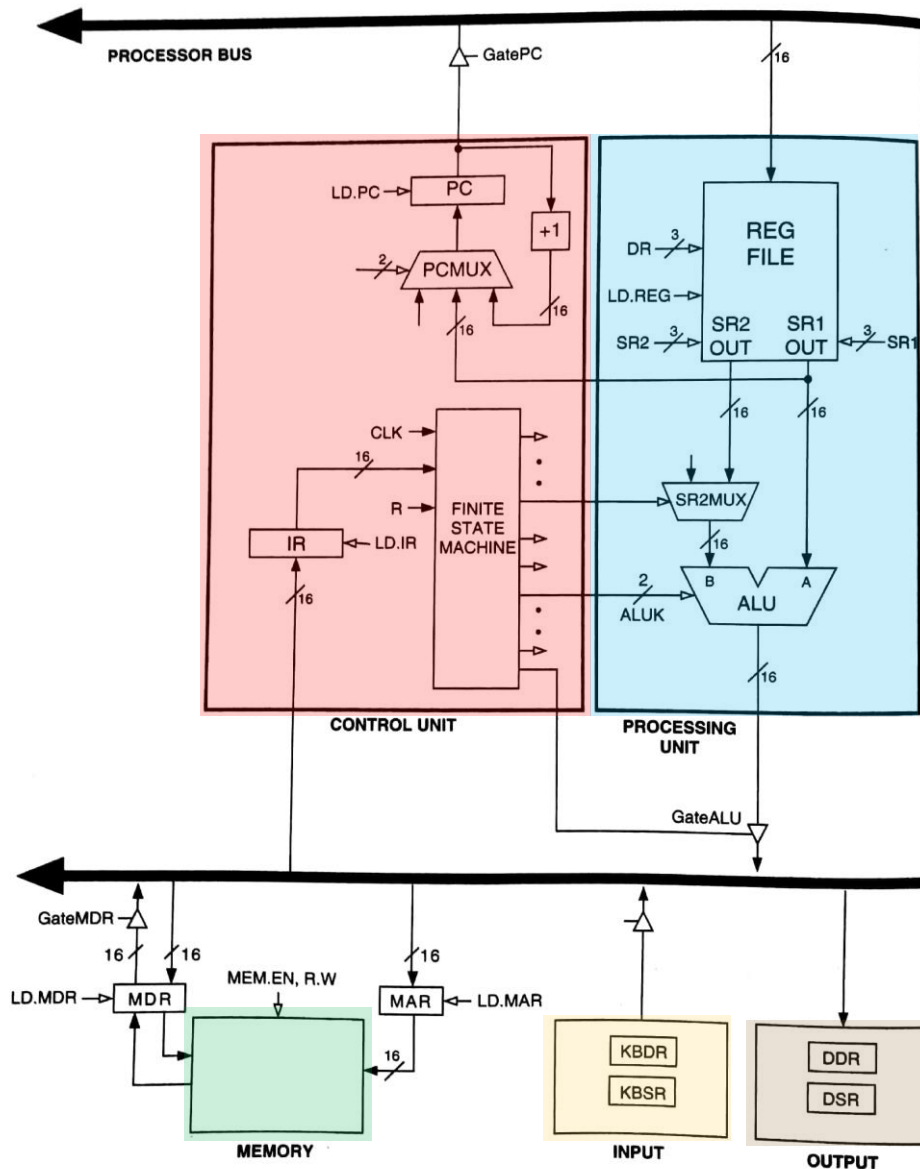


Figure 4.3 The LC-3 as an example of the von Neumann model

Digital Design & Computer Arch.

Lecture 6: Sequential Logic Design

Prof. Onur Mutlu

ETH Zürich

Spring 2020

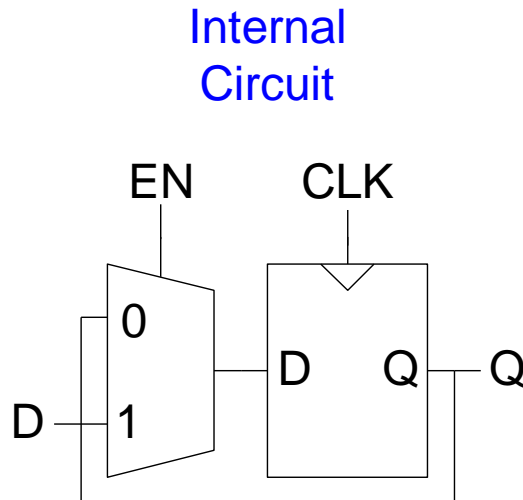
6 March 2020

Backup Slides:

Different Types of Flip Flops

Enabled Flip-Flops

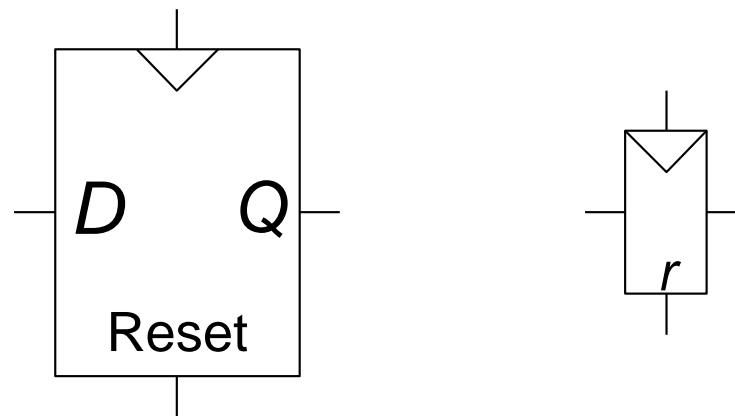
- **Inputs:** CLK, D, EN
 - The enable input (EN) controls when new data (D) is stored
- **Function:**
 - **EN = 1:** D passes through to Q on the clock edge
 - **EN = 0:** the flip-flop retains its previous state



Resettable Flip-Flop

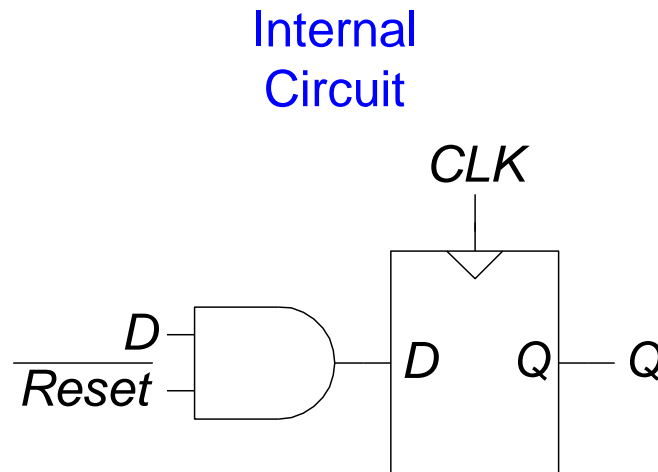
- **Inputs:** CLK, D, Reset
 - The Reset is used to set the output to 0.
- **Function:**
 - **Reset = 1:** Q is forced to 0
 - **Reset = 0:** the flip-flop behaves like an ordinary D flip-flop

Symbols



Resettable Flip-Flops

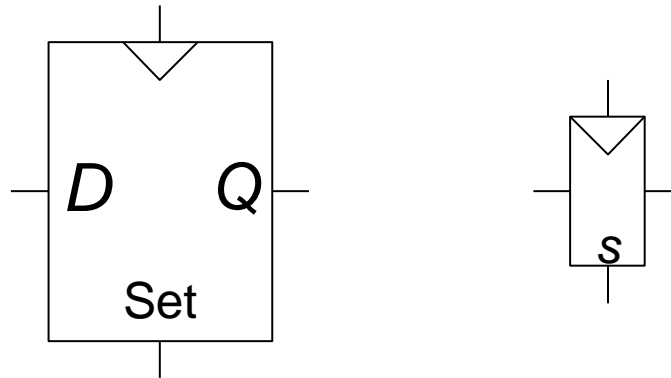
- Two types:
 - **Synchronous:** resets at the clock edge only
 - **Asynchronous:** resets immediately when $\text{Reset} = 1$
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop (see Exercise 3.10)
- Synchronously resettable flip-flop?



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



Logic Simplification: Karnaugh Maps (K-Maps)

Logic Simplification

- Systematic techniques for simplifications

- amenable to automation

Key Tool: The Uniting Theorem — $F = A\bar{B} + AB$

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

$$F = A\bar{B} + AB = A(\bar{B} + B) = A(1) = A$$

Essence of Simplification:

Find two element subsets of the ON-set where only one variable changes its value. This single varying variable *can be eliminated!*

value is not needed

→ B is eliminated, A remains

| A | B | G |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$$G = \bar{A}\bar{B} + A\bar{B} = (\bar{A} + A)\bar{B} = \bar{B}$$

B's value stays the same within the ON-set rows

A's value changes within the ON-set rows

→ A is eliminated, B remains

Complex Cases

■ One example

$$C_{out} = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

■ Problem

- **Easy** to see how to apply Uniting Theorem...
- **Hard** to know if you applied it in all the right places...
- ...especially in a function of many more variables

■ Question

- Is there an easier way to find potential simplifications?
- i.e., potential applications of Uniting Theorem...?

■ Answer

- Need an intrinsically *geometric* representation for Boolean $f()$
- Something we can draw, see...

Karnaugh Map

- Karnaugh Map (K-map) method
 - K-map is an alternative method of representing the **truth table** that helps **visualize adjacencies** in up to 6 dimensions
 - Physical adjacency \leftrightarrow Logical adjacency

2-variable K-map

| | | |
|---------------------|----------|----------|
| A \ B | 0 | 1 |
| 0 | 00 | 01 |
| 1 | 10 | 11 |

3-variable K-map

| | | | | |
|----------------------|-----------|-----------|-----------|-----------|
| A \ BC | 00 | 01 | 11 | 10 |
| 0 | 000 | 001 | 011 | 010 |
| 1 | 100 | 101 | 111 | 110 |

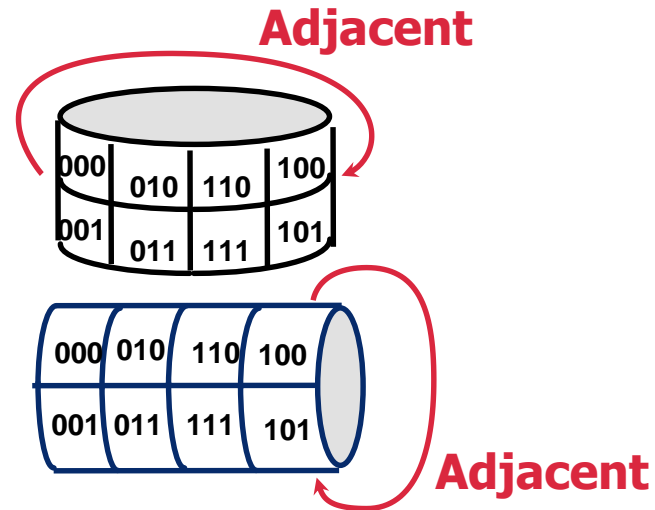
4-variable K-map

| | | | | |
|-----------------------|-----------|-----------|-----------|-----------|
| AB \ CD | 00 | 01 | 11 | 10 |
| 00 | 0000 | 0001 | 0011 | 0010 |
| 01 | 0100 | 0101 | 0111 | 0110 |
| 11 | 1100 | 1101 | 1111 | 1110 |
| 10 | 1000 | 1001 | 1011 | 1010 |

Numbering Scheme: 00, 01, 11, 10 is called a “Gray Code” — only a *single bit (variable)* changes from one code word and the next code word

Karnaugh Map Methods

| $A \backslash BC$ | 00 | 01 | 11 | 10 |
|-------------------|-----|-----|-----|-----|
| 0 | 000 | 001 | 011 | 010 |
| 1 | 100 | 101 | 111 | 110 |



K-map adjacencies go "around the edges"
Wrap around from first to last column
Wrap around from top row to bottom row

K-map Cover - 4 Input Variables

| CD \ AB | 00 | 01 | 11 | 10 |
|---------|----|----|----|----|
| 00 | 1 | 0 | 0 | 1 |
| 01 | 0 | 1 | 0 | 0 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 |

$$F(A, B, C, D) = \sum m(0, 2, 5, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$F = A + \bar{B}\bar{D} + B\bar{C}D$$

Strategy for "circling" rectangles on Kmap:

Biggest "oops!" that people forget:

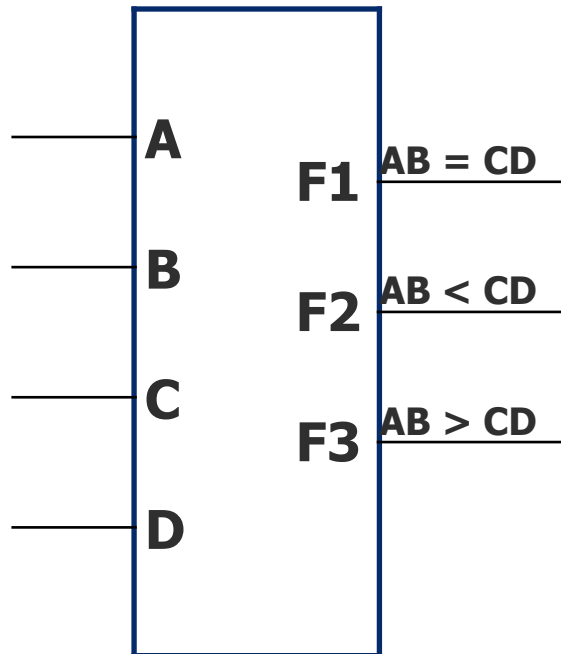
Logic Minimization Using K-Maps

- Very simple guideline:
 - Circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles
 - Each circle should be as large as possible
 - Read off the implicants that were circled
- More formally:
 - A Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants
 - Each circle on the K-map represents an implicant
 - The largest possible circles are prime implicants

K-map Rules

- **What can be legally combined (circled) in the K-map?**
 - Rectangular groups of size 2^k for any integer k
 - Each cell has the same value (1, for now)
 - All values must be adjacent
 - Wrap-around edge is okay
- **How does a group become a term in an expression?**
 - Determine which literals are constant, and which vary across group
 - Eliminate varying literals, then AND the constant literals
 - constant 1 → use X , constant 0 → use \bar{X}
- **What is a good solution?**
 - Biggest groupings → eliminate more variables (literals) in each term
 - Fewest groupings → fewer terms (gates) all together
 - OR together all AND terms you create from individual groups

K-map Example: Two-bit Comparator



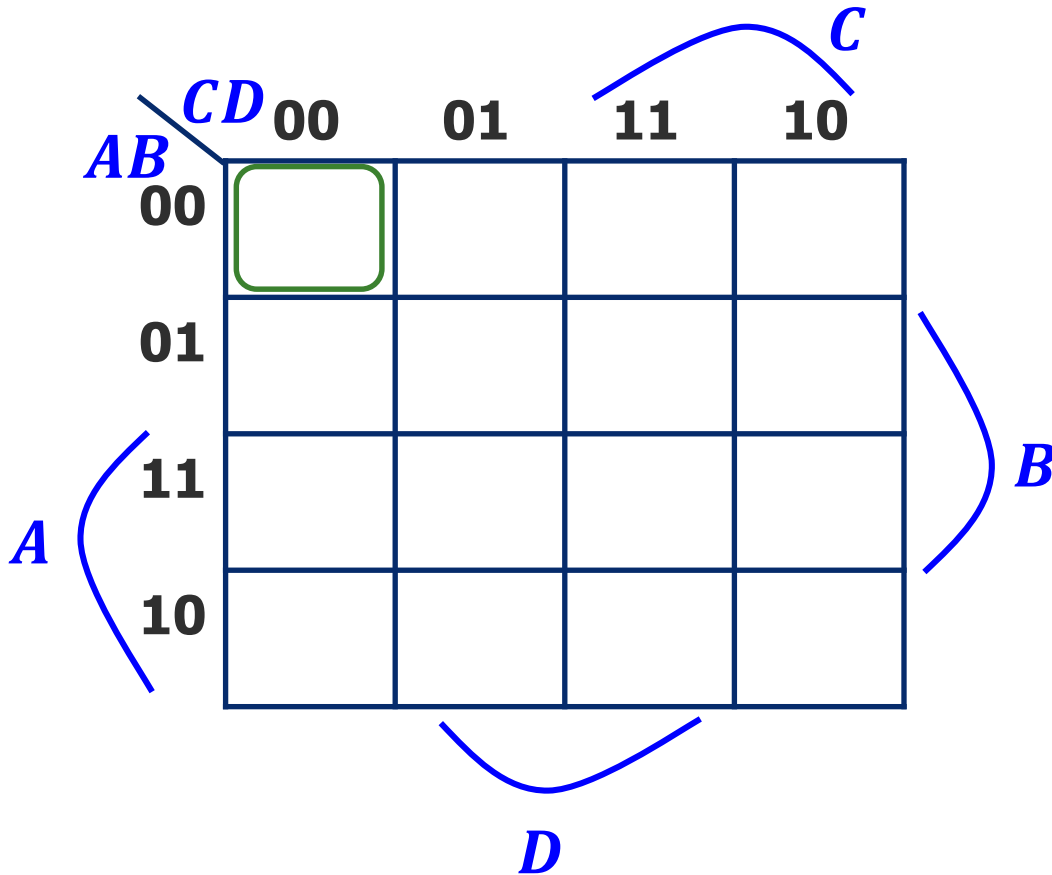
Design Approach:

Write a 4-Variable K-map
for each of the 3
output functions

| A | B | C | D | F1 | F2 | F3 |
|---|---|---|---|----|----|----|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |

K-map Example: Two-bit Comparator (2)

K-map for F1

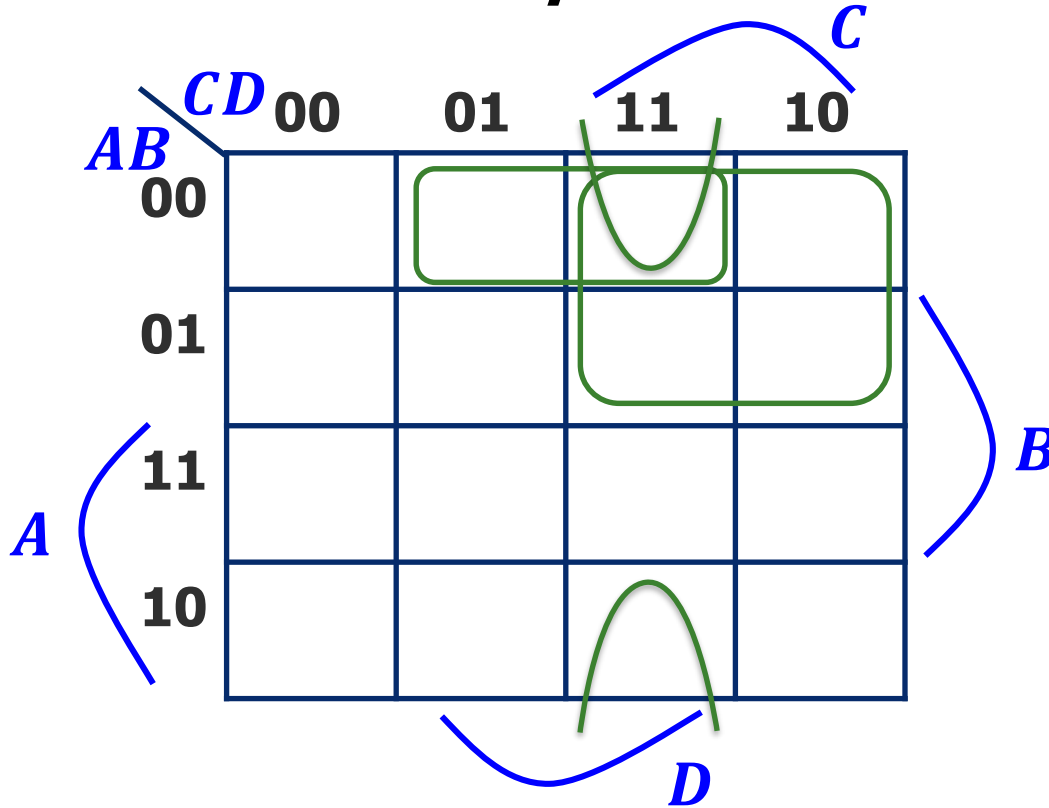


F1 =

| A | B | C | D | F1 | F2 | F3 |
|---|---|---|---|----|----|----|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |

K-map Example: Two-bit Comparator (3)

K-map for F2



F2 =

F3 = ? (Exercise for you)

| A | B | C | D | F1 | F2 | F3 |
|---|---|---|---|----|----|----|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |

K-maps with “Don’t Care”

- **Don’t Care** really means *I don’t care what my circuit outputs if this appears as input*
 - You have an engineering choice to use DON’T CARE patterns intelligently as 1 or 0 to better **simplify** the circuit

| A | B | C | D | F | G |
|-----|---|---|---|---|---|
| ... | | | | | |
| 0 | 1 | 1 | 0 | X | X |
| 0 | 1 | 1 | 1 | | |
| 1 | 0 | 0 | 0 | X | X |
| 1 | 0 | 0 | 1 | | |
| ... | | | | | |

I can pick 00, 01, 10, 11 independently of below

I can pick 00, 01, 10, 11 independently of above

Example: BCD Increment Function

- BCD (Binary Coded Decimal) digits
 - Encode decimal digits 0 - 9 with bit patterns 0000_2 — 1001_2
 - When **incremented**, the decimal sequence is 0, 1, ..., 8, 9, 0, 1

| A | B | C | D | W | X | Y | Z |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | X | X | X | X |
| 1 | 0 | 1 | 1 | X | X | X | X |
| 1 | 1 | 0 | 0 | X | X | X | X |
| 1 | 1 | 0 | 1 | X | X | X | X |
| 1 | 1 | 1 | 0 | X | X | X | X |
| 1 | 1 | 1 | 1 | X | X | X | X |

These input patterns **should never be encountered** in practice (hey -- it's a BCD number!) So, associated output values are **"Don't Cares"**

K-map for BCD Increment Function

A B

Z (without don't cares) =

+

Z (with don't cares) =

W X

| | | | | |
|-----------|----------|--|----------|----------|
| 10 | 1 | | X | X |
|-----------|----------|--|----------|----------|

| | | | | |
|-----------|--|--|----------|----------|
| 10 | | | X | X |
|-----------|--|--|----------|----------|

Y

| | | | | |
|-----------|-----------|-----------|-----------|-----------|
| CD | 00 | 01 | 11 | 10 |
| AB | | | | |
| 00 | | 1 | | 1 |
| 01 | | 1 | | 1 |
| 11 | X | X | X | X |
| 10 | | | X | X |

Z

| | | | | |
|-----------|-----------|-----------|-----------|-----------|
| CD | 00 | 01 | 11 | 10 |
| AB | | | | |
| 00 | 1 | | | 1 |
| 01 | 1 | | | 1 |
| 11 | X | X | X | X |
| 10 | 1 | | X | X |

(Note: In the original image, the 1s in the Z K-map are highlighted with purple shading and grouped with blue arcs labeled A, B, C, and D.)

K-map Summary

- **Karnaugh maps** as a formal systematic approach for logic simplification
- 2-, 3-, 4-variable K-maps
- K-maps with “**Don't Care**” outputs
- H&H Section 2.7