

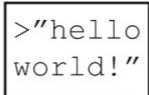


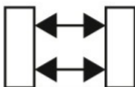
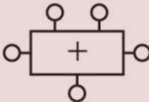

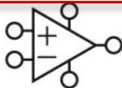


COMP2300-COMP6300-ENGN2219

Computer Organization & Program Execution

Convener: Shoaib Akram
shoaib.akram@anu.edu.au



Australian
National
University

Application Software		Programs
Operating Systems		Device Drivers
Architecture		Instructions Registers
Micro-architecture		Datapaths Controllers
Logic		Adders Memories
Digital Circuits		AND Gates NOT Gates
Analog Circuits		Amplifiers Filters
Devices		Transistors Diodes
Physics		Electrons

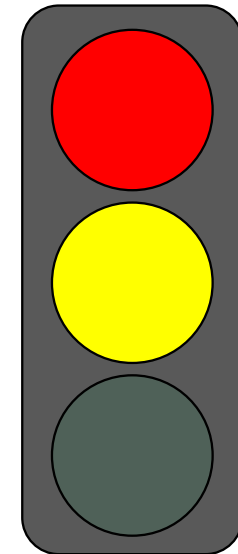
**Broadening our horizon
"one layer at a time"**

We Covered Combinational Blocks

- Computation
 - Adders
 - ALU
 - Comparator
- Control
 - Multiplexer
 - Decoder
 - Tri-state Buffer
- Standard form (SOP)
- Boolean equation to 2-level implementation

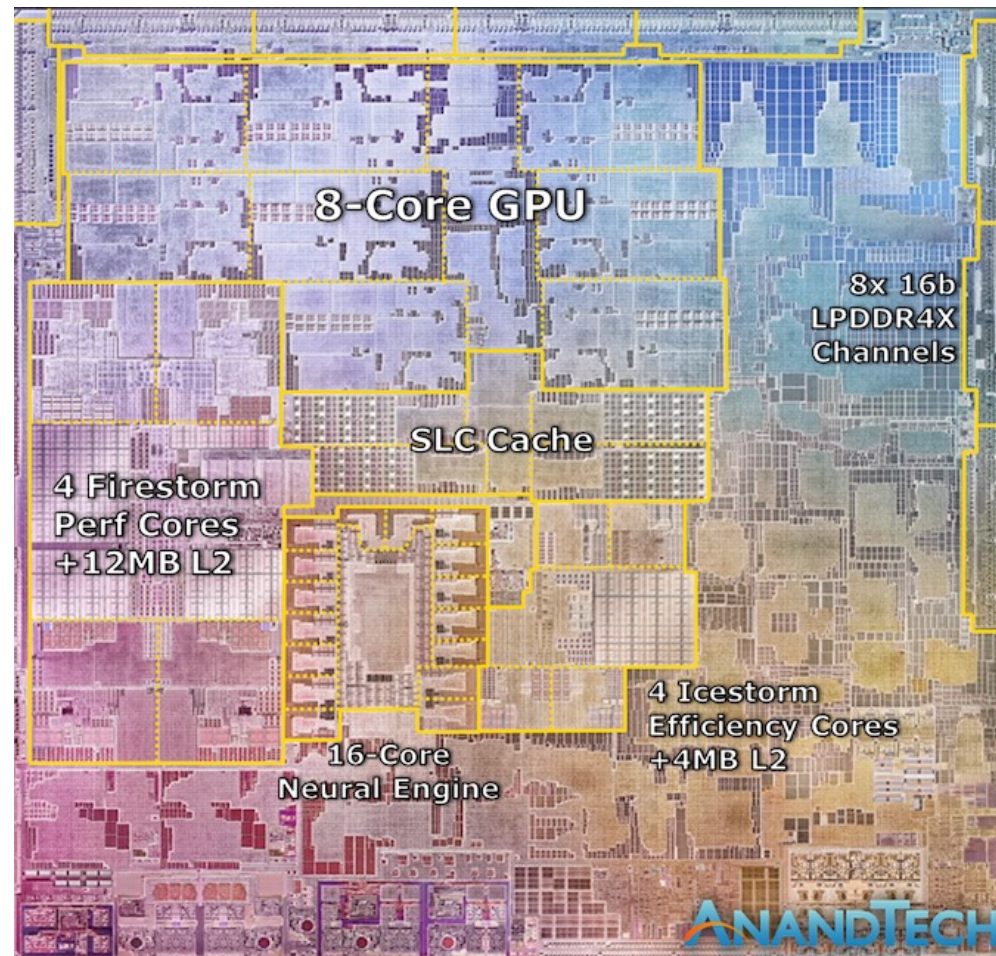
What will we learn this week?

- Circuits that can store information
 - State and clock
 - Cross-coupled inverter
 - SR latch
 - D latch
 - D flip-flop
 - Register & Memory
- Synchronous sequential circuits
 - Finite state machines
- Synchronous vs. Asynchronous sequential circuits



Circuits that Store Information

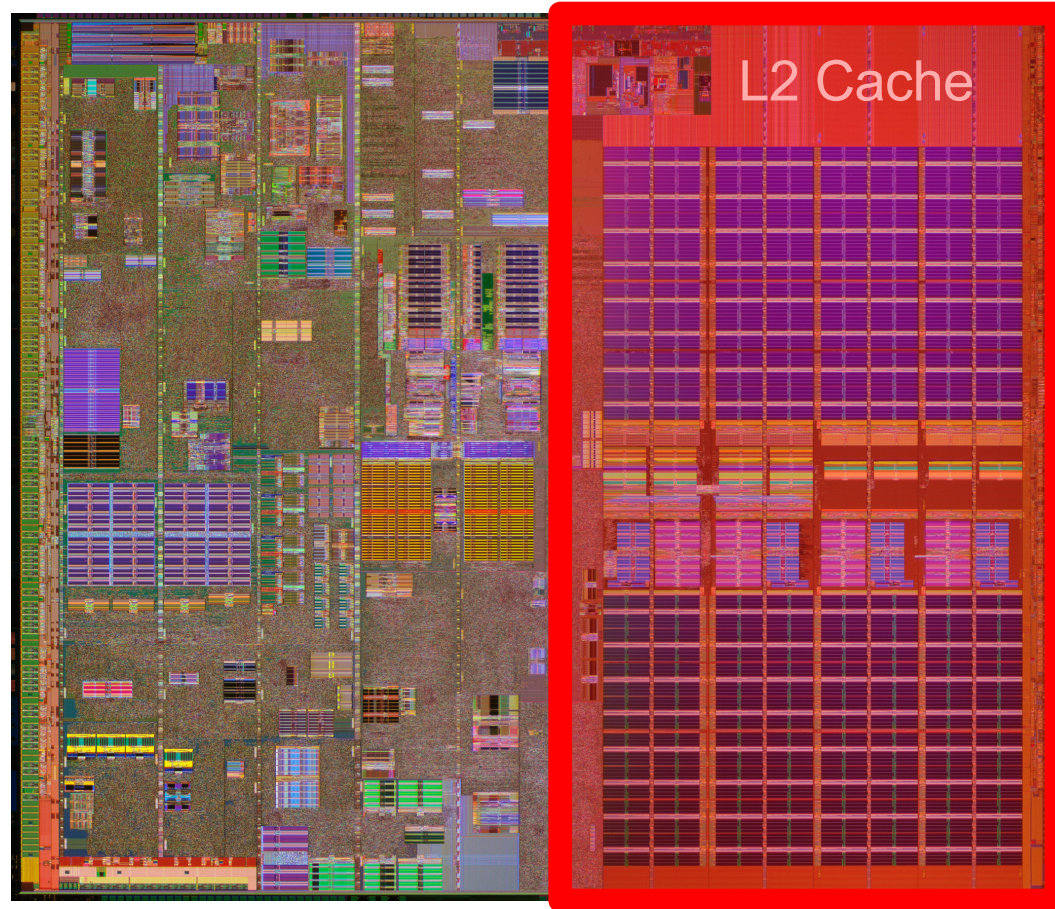
All Computers Need Memory to Work



Apple M1,
2021

Source: <https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested>

Large Portion of a System is Memory



https://download.intel.com/newsroom/kits/40thanniversary/gallery/images/Pentium_4_6xx-die.jpg

Intel Pentium 4, 2000

Large Portion of a System is Memory

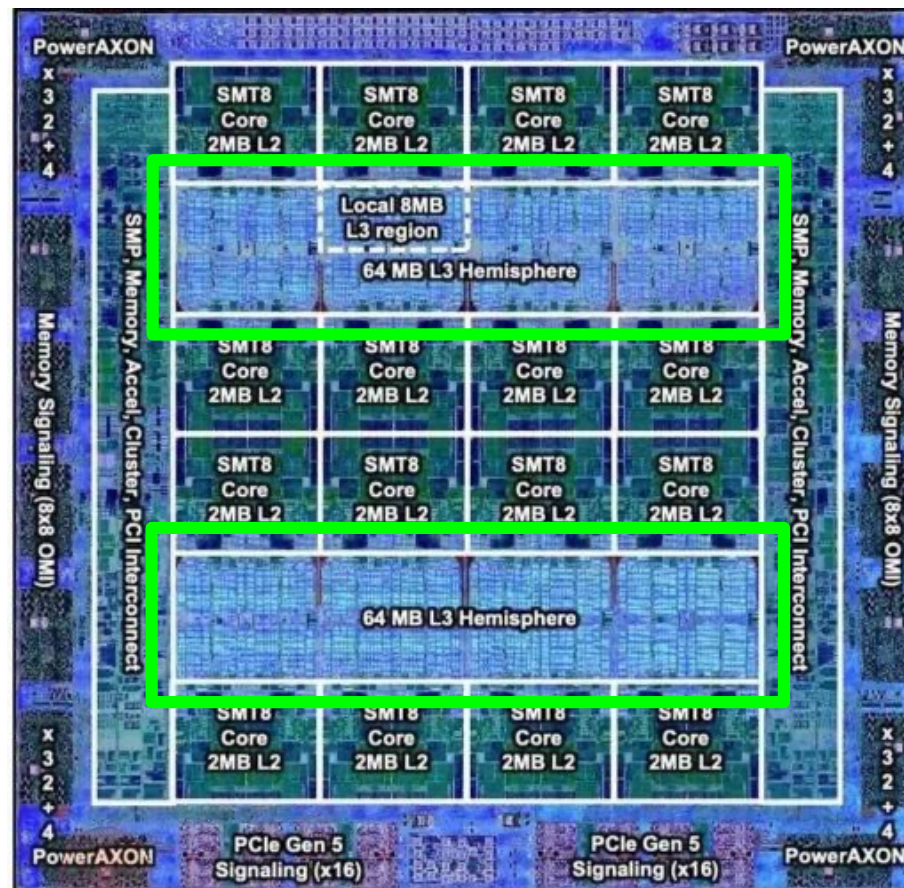
IBM POWER10,
2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:
2 MB per core

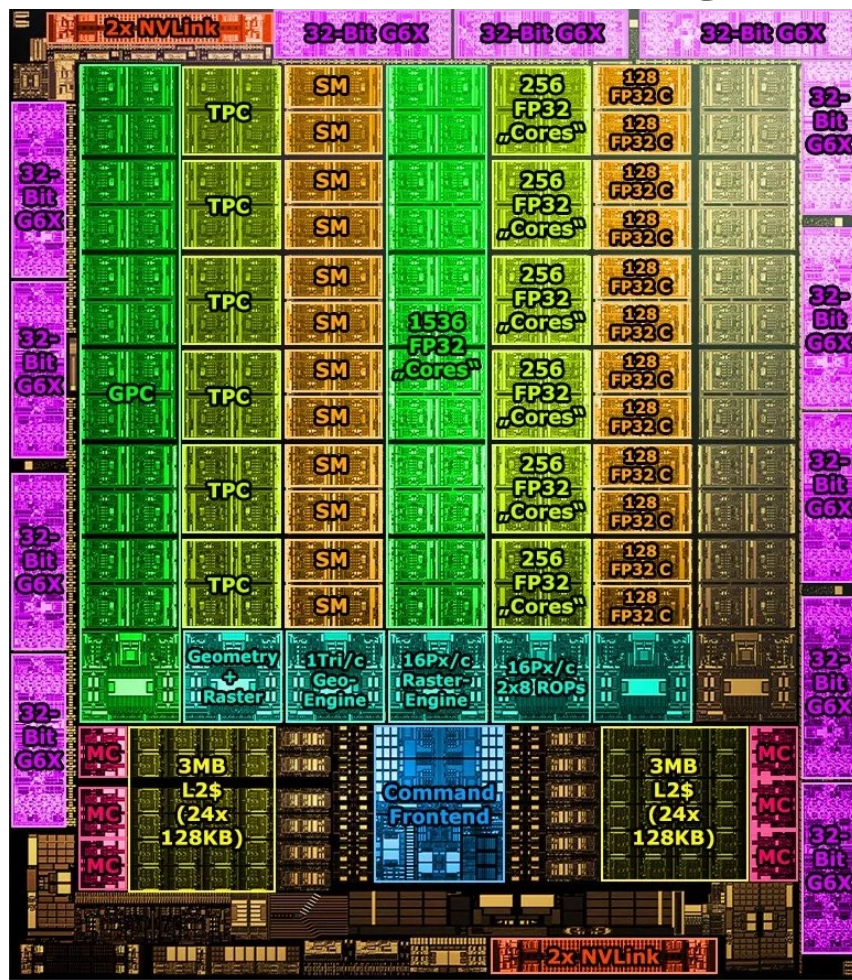
L3 Cache:
120 MB shared



<https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/>

Large Portion of a System is Memory

Nvidia Ampere,
2020



Cores:

128 Streaming Multiprocessors

L1 Cache or Scratchpad:

192KB per SM

Can be used as L1 Cache and/or
Scratchpad

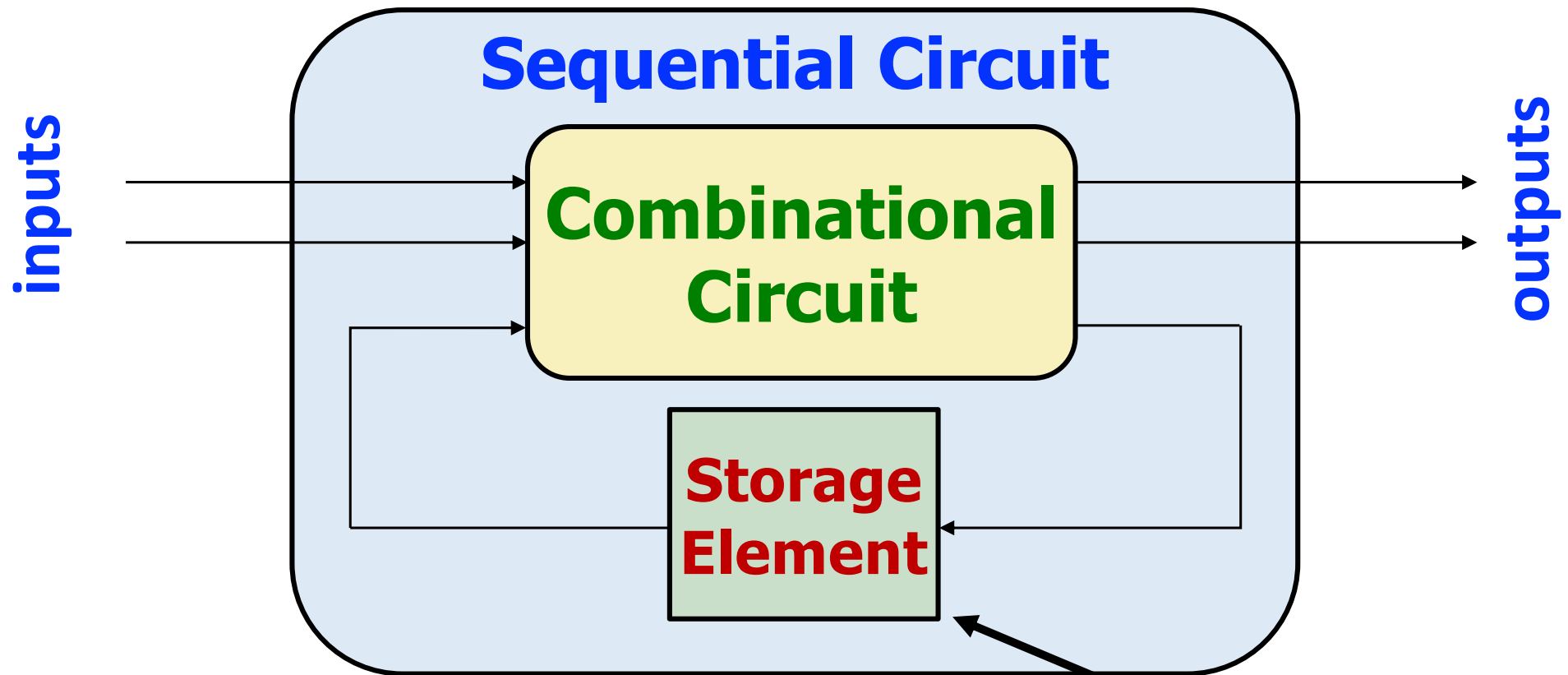
L2 Cache:

40 MB shared

Motivation

- **Combinational** circuit output depends **only** on **current** input **combinations**
- 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?**

Sequential Circuit



The output depends on the input combinations and the state of the system

What is State?

- What is **state**?
 - A **set of bits** that **summarize** past behavior of system
 - The **set of bits** are called **state variables**
 - Each bit is stored in a state or memory element
- The state of a system ***informs us everything about the past we need to know to predict the future***
- The output of sequential logic depends on both the current input values and the **state** of the system

Example of State

- Consider a **controller** for a traffic light



- To set the **next** state, the controller needs to remember the **current** state
- The past input values (signals from traffic sensors) are summarized by the “**current**” state stored as a **2-bit** value

Another Example

- We remember the meaning of the **words** in time **t** since
 - we remembered them in time **t - 1**
 - all the way back to the point of time
 - when we first committed them to **memory**

Capturing State

- To **remember** or to store the **state** of the system (current state of traffic light), we need a **logic element** with memory

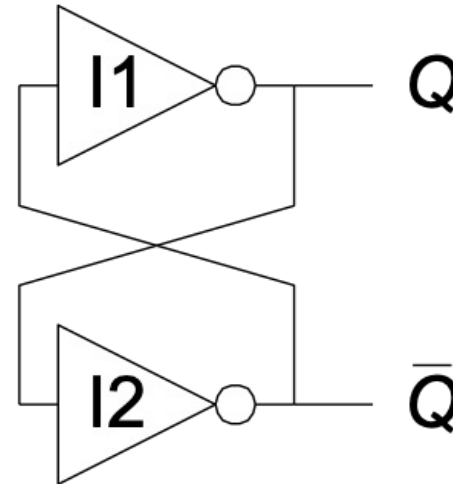
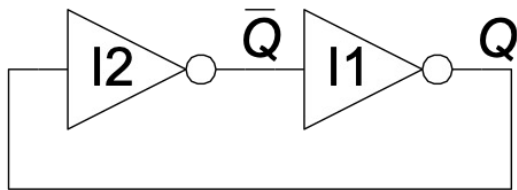
- **How can we capture input data and persist it until the next input?**

Capturing and Storing One Bit

- **One bit of information represents two possible states**
- To store one bit, we need
 - An element with two stable states (**bistable** element)
 - The ability to change the state
- First, we will find the bistable element
 - And then focus on the *ability to change state*

Bistable element

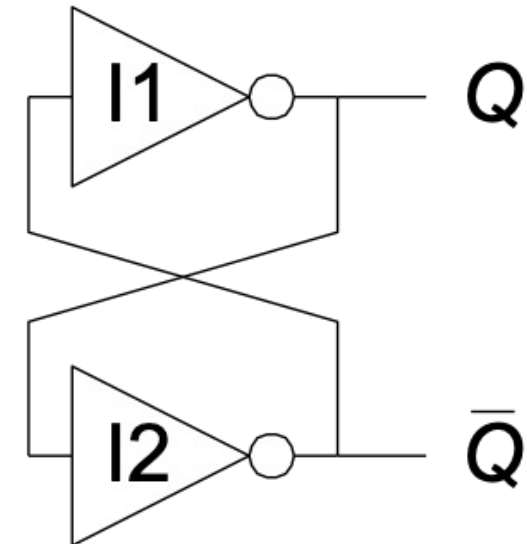
- The fundamental building block of **memory** is a **bistable element** with two stable states



Cross-coupled inverters: A pair of inverters connected in a loop

Analysis of Bistable Element

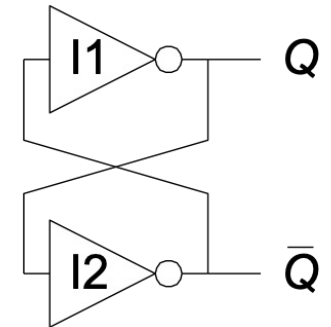
- **How should we analyze circuits with cyclic paths?**
 - When the circuit is switched on, Q is either **0** or **1** (**scenarios**)
 - **Show:** Output is stable (**consequence – A**)
 - **Show:** Q and Q' are complements of each other (**consequence – B**)
- **Scenario # 1: $Q = 0$ (FALSE)**
 - I_2 receives **0**, $Q' = 1$: **B** is satisfied
 - $Q' = 1$, $Q = 0$, **A** is satisfied
 - Consistent with original assumption
- **Scenario # 2: $Q = 1$ (TRUE)**
 - I_2 receives **1**, $Q' = 0$: **B** is satisfied
 - $Q' = 0$, $Q = 1$, **A** is satisfied



Section 3.2 of H&H

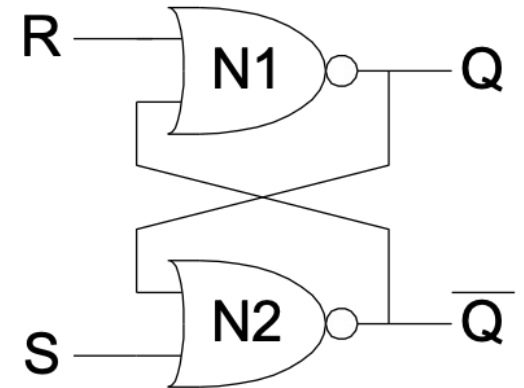
Bistable Element: Observations

- **Bistable element has two stable states**
 - Stores one bit of information
- **Q** reveals about past, necessary to explain the future
 - If **Q = 0**, it will remain **0** forever
 - If **Q = 1**, it will remain **1** forever
- The (**initial**) state of the bistable element is unpredictable when **powered on**
 - Bistable element is *not practical* because the user lacks inputs to control its state
 - **We need something else!**



SR Latch

- Two **cross-coupled NOR** gates
- The **state** can be **controlled** with S/R inputs
 - **S** = Set to **1**
 - **R** = Reset to **0**



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

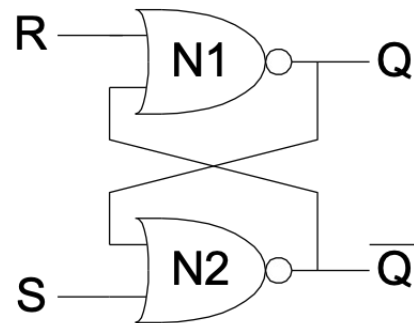
- NOR gate revision
 - When both inputs are **0**, the output is **1**
 - If any of the inputs is **1**, the output is **0**

SR Latch: Analysis

- SR latch has inputs *unlike the cross-coupled inverter*
- Four scenarios in the truth table (Sim = Simultaneous)

Scenario	S	R	Q	Q'
Sim-0	0	0		
Reset	0	1		
Set	1	0		
Sim-1	1	1		

Whiteboard: SR Latch



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

S	R	Q
0	0	
0	1	
1	0	
1	1	

Section 3.2.1 of H&H

SR Latch: Analysis

- **Scenario # 1 (Reset):** $R = 1, S = 0$
 - N1 sees at least one **TRUE (1)** input
 - $Q = \text{FALSE (0)}$
 - N2 sees both Q and S **FALSE**
 - $Q' = \text{TRUE (1)}$
- **Scenario # 2 (Set):** $R = 0, S = 1$
 - N1 sees **0** and Q'
 - What is Q' ?
 - N2 sees at least one **TRUE** input
 - $Q' = \text{FALSE (0)}$
 - Revisit N1 ($R = 0$ and $Q' = 0$)
 - $Q = \text{TRUE (1)}$

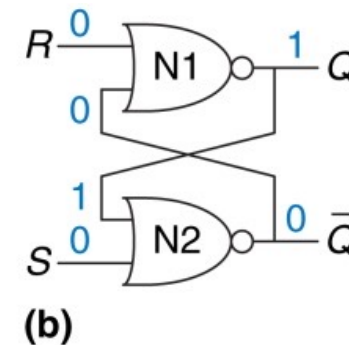
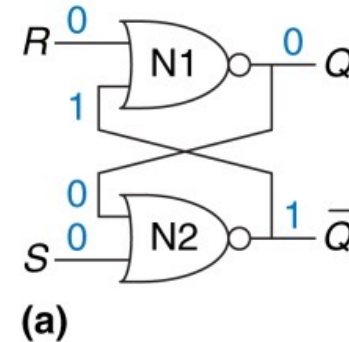
SR Latch: Analysis

- **Scenario # 3 (Sim-1): R = 1, S = 1**
 - N1 sees at least one **TRUE (1)** input
 - **Q = FALSE (0)**
 - N2 sees at least one **TRUE (1)** input
 - **Q' = FALSE (0)**

- **Scenario # 4 (Sim-0): R = 0, S = 0**
 - N1 sees **0** and **Q'**
 - What is **Q'**?
 - N2 sees **0** and **Q**
 - What is **Q**?
 - **We are stuck!**
 - Wait: remember the cross-coupled inverter? **Q** can be **0** or **1** 😊

SR Latch: Analysis

- Scenario # 4-A (Sim-0): $R = 0, S = 0, Q = 0$
 - N2 sees $S = 0$ and $Q = 0$
 - $Q' = 1$
 - N1 sees one TRUE (1) input
 - $Q = 0$ (hindsight: Q is indeed 0 as assumed)
- Scenario # 4-B (Sim-0): $R = 0, S = 0, Q = 1$
 - N2 sees $Q = 1$
 - $Q' = \text{FALSE}$
 - N1 receives two FALSE inputs
 - $Q = 1$ (hindsight: Q is indeed 1 as assumed)



SR Latch: Analysis Summary

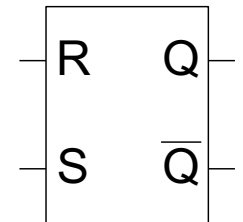
- SR latch has inputs **unlike the cross-coupled inverters**
- Four scenarios in the truth table (**Sim** = **Simultaneous**)

Scenario	S	R	Q	Q'
Sim-0	0	0	Q _{prev}	Q' _{prev}
Reset	0	1	0	1
Set	1	0	1	0
Sim-1	1	1	0	0

SR Latch: Observations

- SR latch is a **bistable element** but its state can be controlled
 - To **set** a bit means to make it **TRUE**
 - To **reset** is to make it **FALSE**
- **Q** accounts for the entire history of past inputs
 - All *prior set/reset patterns* are irrelevant
 - The *most recent set/reset* event predicts the future behavior of the **SR** latch
- Asserting both **set/reset** to **1** does not make sense
 - Neither intuitively nor physically
 - The circuit outputs **0** on **Q** and **Q'** which is inconsistent
 - **We need something else!**

SR Latch
Symbol



Next: D Latch

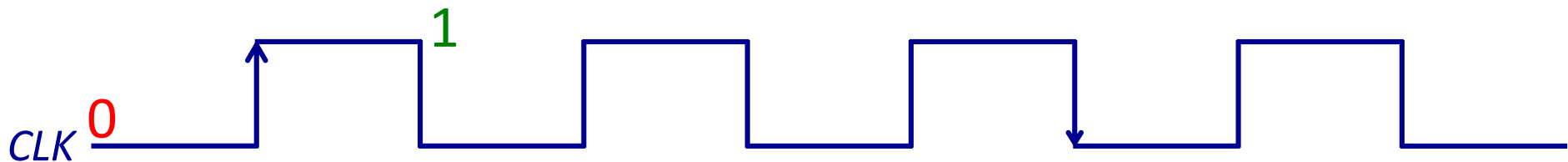
First, We Will Learn to Model the
Progression of Time

Representing Time

- Physicist and philosopher's view
 - Relentless arrow continuously progressing forward
 - Changes in the world can be infinitesimally small

Arrow of time
Changes continuously

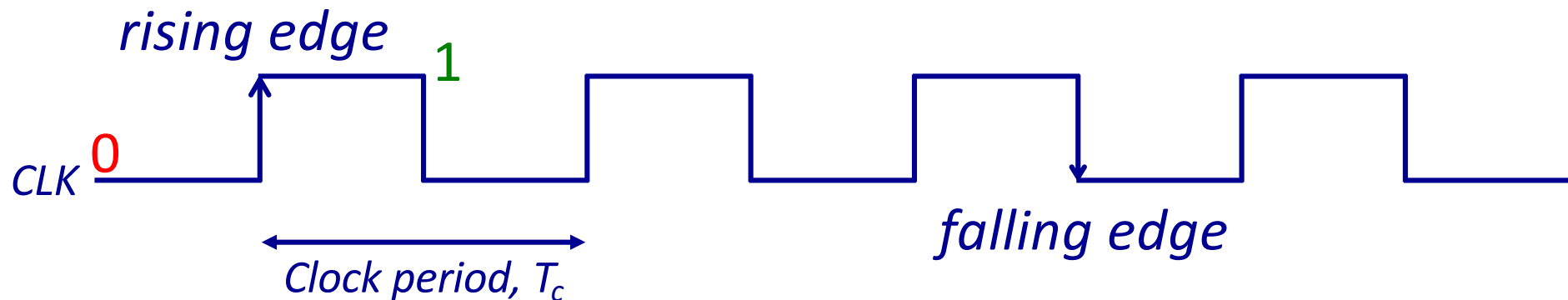
- **Too mysterious and deep for computer scientists**
 - We prefer a **discrete** representation
 - Break time into **fixed-length** intervals called **cycles**
 - cycle 1, cycle 2, cycle 3, and so on **Cycles are indivisible and atomic**



Changes in the world occur only during cycle transitions; within cycles, the world stands still

Time in Digital Systems

- Periodic **square wave (clock)** generated by a special circuit
- Used to *synchronize* state updates everywhere in the system



$$\text{Frequency} = 1/T_c$$

Changes in the world occur only during cycle transitions. Within cycles, the world stands still

Time in Digital Systems

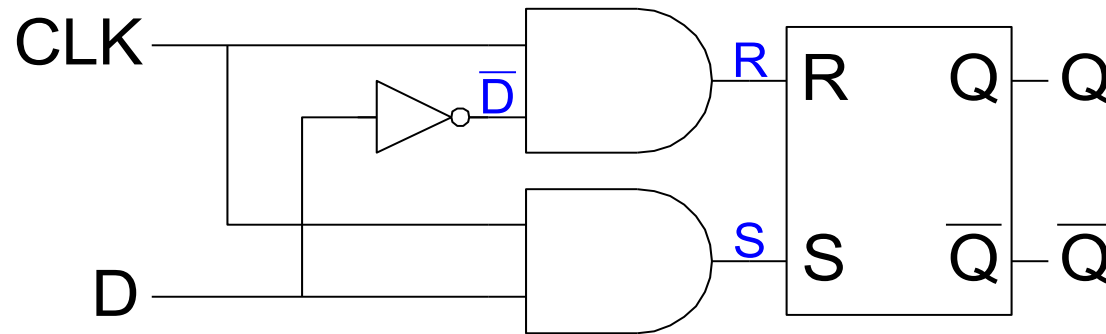
- Our goal is to endow logic circuits with the ability to
 - **Maintain state over time**
 - **Respond to time changes**

D Latch

- **Two drawbacks of SR latch**
 - Strange behavior when **S = 1** and **R = 1**
 - **S** and **R** inputs serve two roles: *what* the state is and *when* the state changes
- Designing sequential circuits is **easier** when we have control over *when* the state changes
- The **D latch** **overcomes** the two drawbacks
 - A data input (**D**) *controls what* the next state should be
 - The clock input (**CLK**) controls *when* the state should change

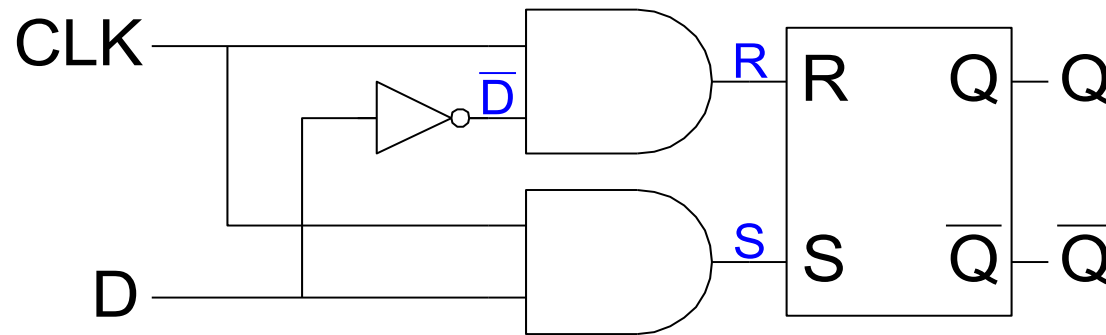
Section 3.2.2 of H&H

D Latch



- **Scenario # 1:** $CLK = 0, S = 0, R = 0, D = X$
 - The value of **D** is irrelevant
 - $Q = Q_{prev}$ (remember the old value)
 - **Scenario # 2:** $CLK = 1, D = 0, S = 0, R = 1$
 - The latch is **reset**
 - **Scenario # 3:** $CLK = 1, D = 1, S = 1, R = 0$
 - The latch is **set**
- Latch is opaque (blocks new data from flowing to Q)
- Latch is transparent (acts like a buffer)

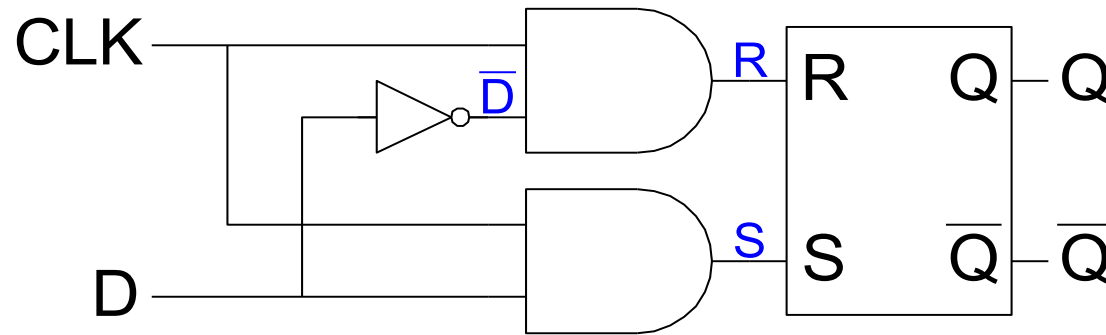
Whiteboard: D Latch



S	R	Q
0	0	Q_{prev}
0	1	0
1	0	1
1	1	0

Section 3.2.2 of H&H

D Latch: Truth Table



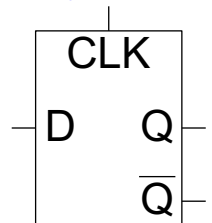
Scenario	CLK	D	Q	Q'
Opaque	0	X	Q_{prev}	Q'_{prev}
Transparent/0	1	0	0	1
Transparent/1	1	1	1	0

Section 3.2.2 of H&H

D Latch: Observations

- D latch is a **level-triggered** or a **level-sensitive** circuit
 - Reacts to the level (0 or 1) of the CLK input
- D latch avoids the **awkward** case of both **S/R** asserted
- D latch **changes** its state **continuously** when **CLK = 1**
- Designing **correct & efficient** sequential circuits is easier when
 - the state changes only at a **specific instant** in time instead of **changing continuously**
 - **We need something else!**

D Latch
Symbol



Summary

- **Cross-coupled inverter**
 - Two stable states *but* no **inputs** to control the state
- **SR Latch**
 - Two control inputs: **S** for **Set** (**1**) and **R** for **Reset** (**0**)
 - Awkward when **S** = **1** and **R** = **1**
 - The state changes instantly (*need greater control over when it changes*)
- **D Latch**
 - One **D**ata input (**D**) and one control input (**CLK**)
 - Level-sensitive: **CLK** = **1** (**Transparent**), **CLK** = **0** (**Opaque**)
 - Output (**Q**) changes continuously when **CLK** = **1**

D Flip-Flop

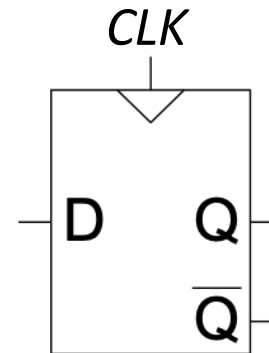
- **Problem with D Latch:** The output changes *continuously* when **CLK = 1**
 - This behavior leads to timing bugs and circuits that are difficult to analyze
- **D flip-flop** is an **edge-triggered** circuit
 - Sets its **state** to the **data input** when **CLK** changes from **0** to **1**
 - At all other times, the D flip-flop remembers its **state**

Flip-flop samples the input at the edge of the clock

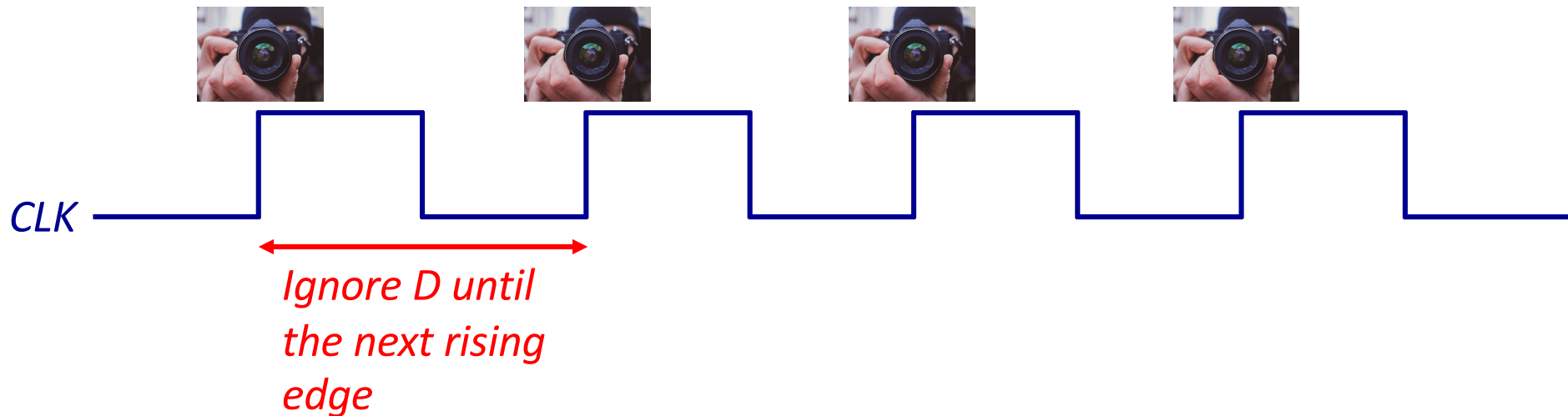


Analogy: Photography

- Input: **D**, Output/State: **Q**
- **CLK** = Clock (periodic square wave)

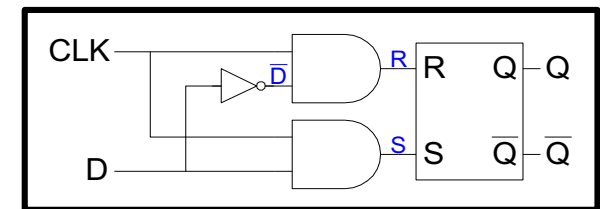
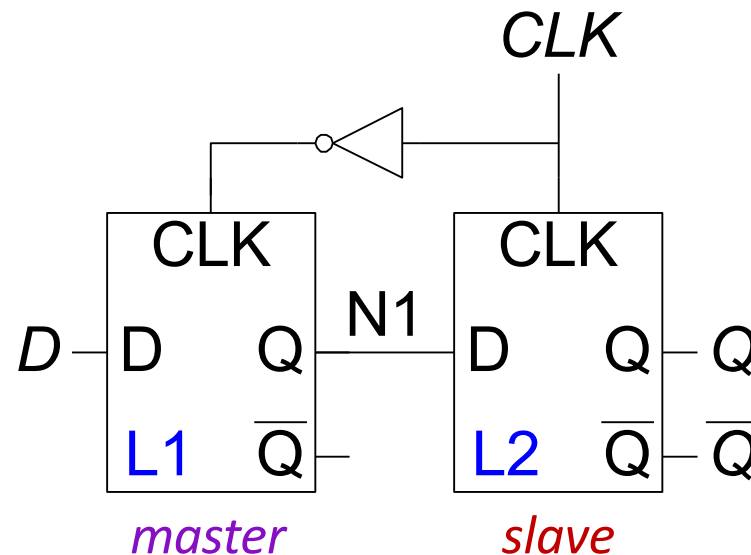


Sample the input just at the rising edge (copy D to Q)



D Flip-Flop

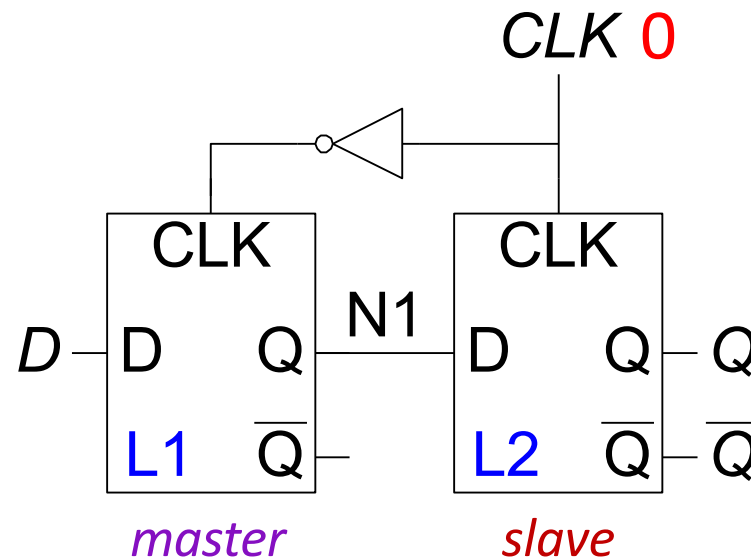
- Two back-to-back D latches controlled by complementary clocks
- The **L1** latch is the master and **L2** is the slave latch
 - Remember that, when **CLK = 0**, D latch is *opaque*
 - And, when **CLK = 1**, D latch is *transparent*



D Latch

D Flip-Flop: Analysis

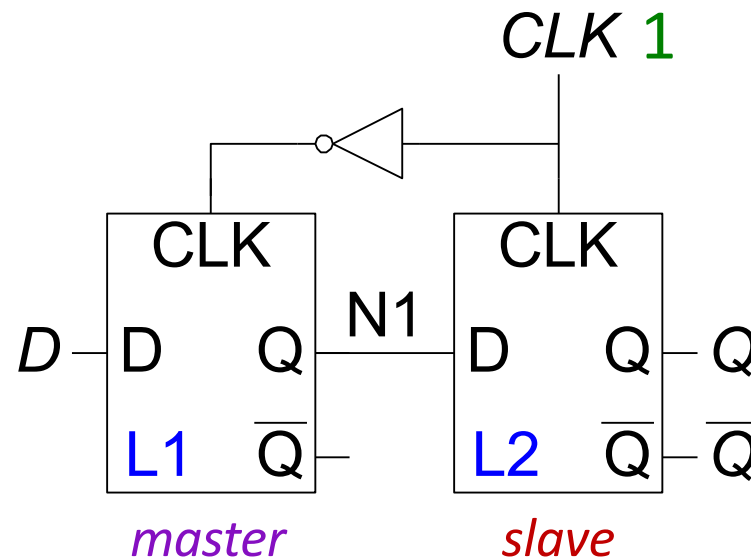
- **CLK = 0**
 - master: transparent
 - slave: opaque



The value at D propagates through to N1, but Q is cut off from N1

D Flip-Flop: Analysis

- **CLK = 1**
 - master: opaque
 - slave: transparent



The value at N1 propagates through to Q, but N1 is cut off from D

D Flip-Flop: Observations

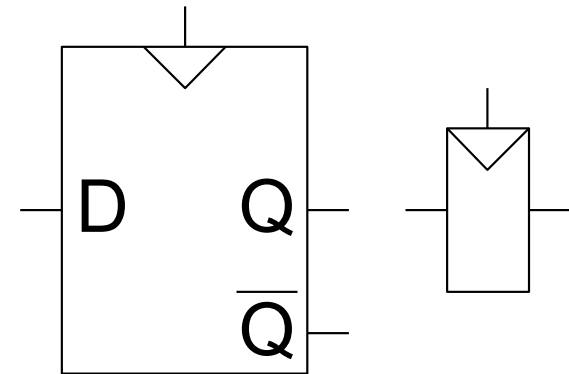
- The value at **D** immediately before the **CLK** rises from **0** to **1** get copied to **Q** immediately after **CLK** rises
- At all other times, **Q** retains its **old** value
 - There is an *opaque* latch blocking **D** from flowing to **Q**

A **D** flip-flop copies **D** to **Q** on the *rising/falling edge* of the **CLK**, and remembers its state at all other times

Other Names

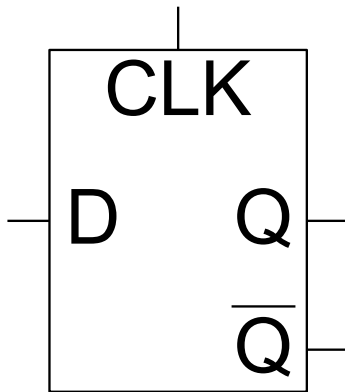
- All names means the same
 - Master-slave flip-flop
 - Edge-triggered flip-flop
 - Positive edge-triggered flip-flop

D Flip-Flop Symbols

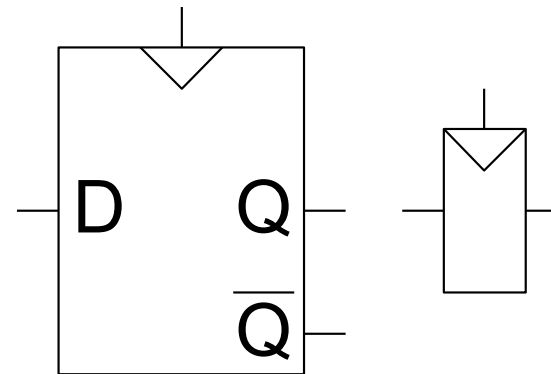


Remember the Symbols

D Latch
Symbol



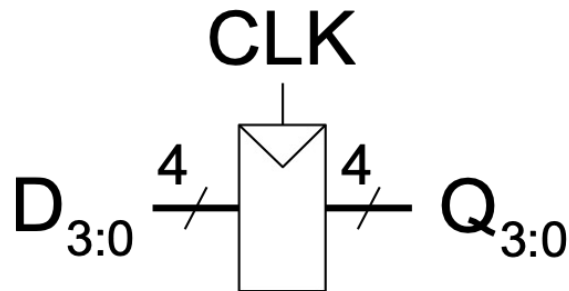
D Flip-Flop
Symbols



Register

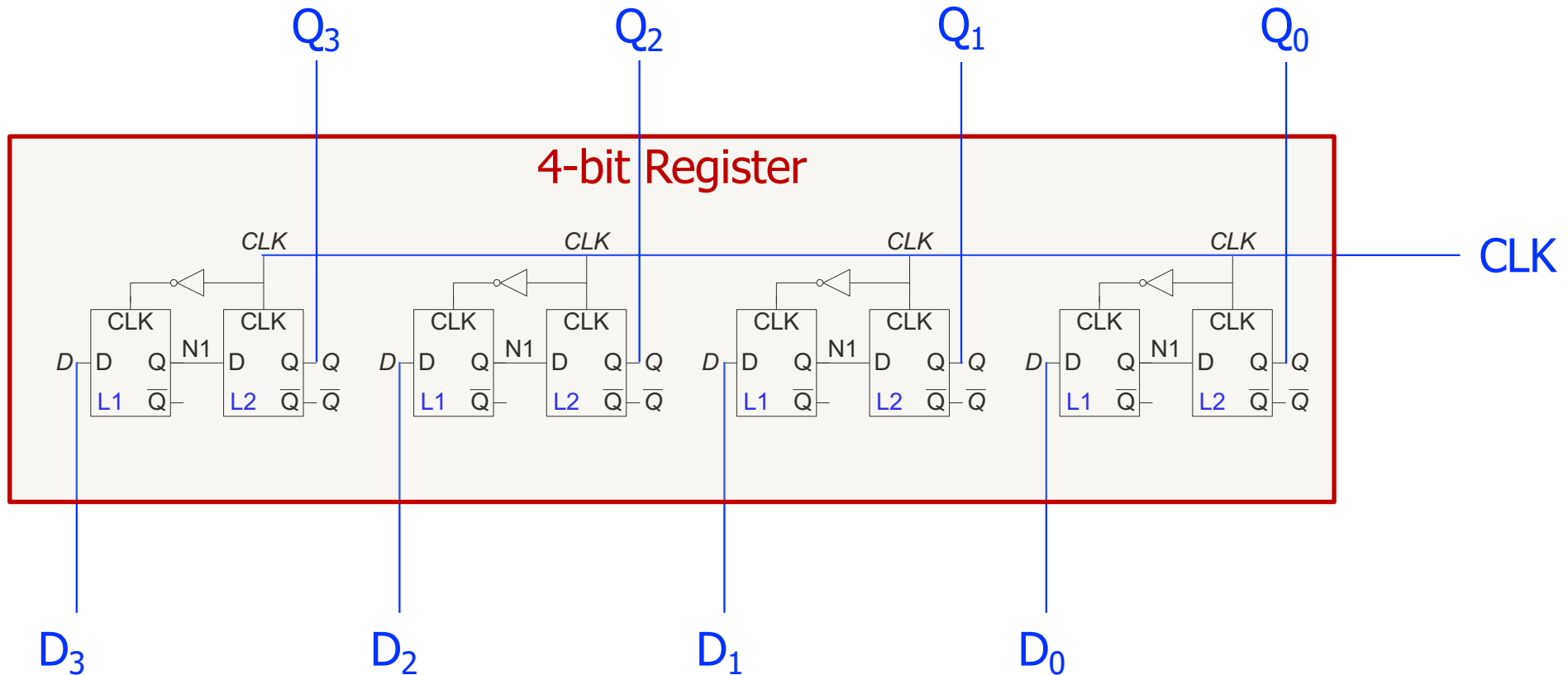
Register

- How can we use flipflops to store more than one bit?
 - Principle of **modularity**: Use more flipflops!
 - A single **CLK** to simultaneously write to all flipflops



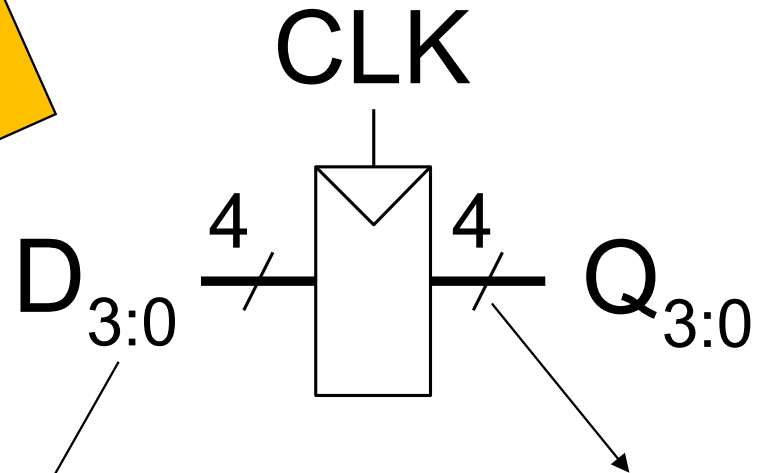
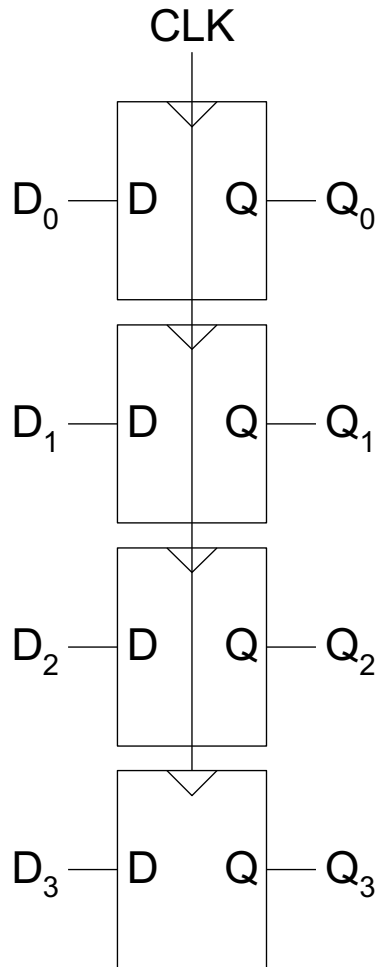
- **Register**: A structure that stores more than **one bit** of information and can be **read from** and **written to**
- This **register** holds **4 bits**, and its data is referenced as **Q[3:0]**

4-bit Register



To build an **N-bit** register, use a bank of **N** flipflops with a shared **CLK**

4-bit Register



This line represents 4 wires

This register stores 4 bits

Register Width

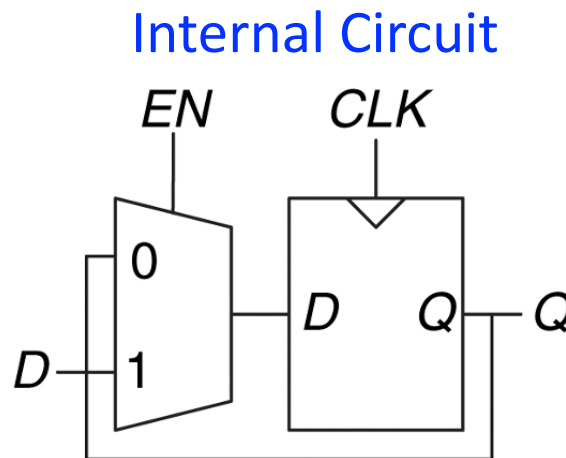
- Register width is the number of flipflops in a register
 - Typical register widths: 8-bit, 16-bit, 32-bit, 64-bit, ..., 512+ bits today
 - Register width is an important parameter of any computer
- **Why do we need wide registers with large widths?**
 - Solving **large/important** problems require manipulating large numbers
 - Large numbers need large number of bits, hence more flipflops
 - Floating point numbers (**section 5.3**) with decimal point require extra bits
- **Question:** Do we need 512-bit registers in microwave controller, automobile engine control unit (ECU), computer used in financial tech, cockpit domain controller in cars?

Now, We will See

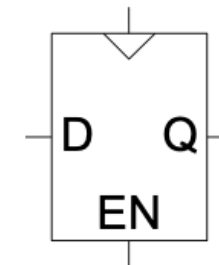
Different Types of Flipflops

Enabled Flip-Flops

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

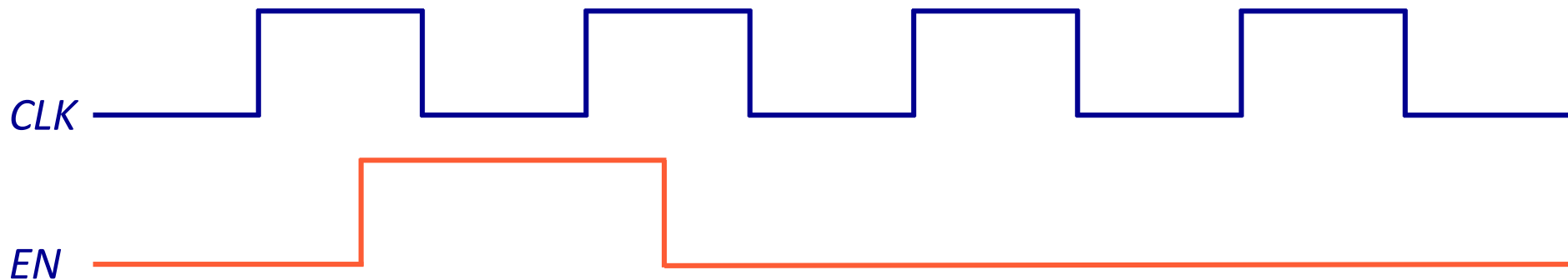
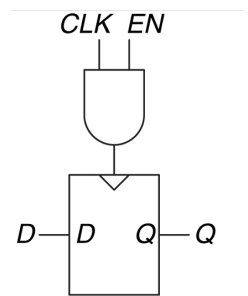


Symbol



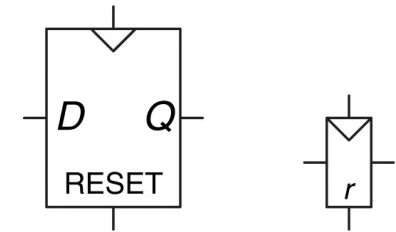
Clock Gating

- Doing an **AND** of **CLK** with **EN** is called **clock gating**
- **Caution:** Performing logic on clock is best avoided
 - AND gate delays the clock
 - If **EN** changes while **CLK** is **1**, the flipflop sees a **glitch** (switch at an incorrect time)



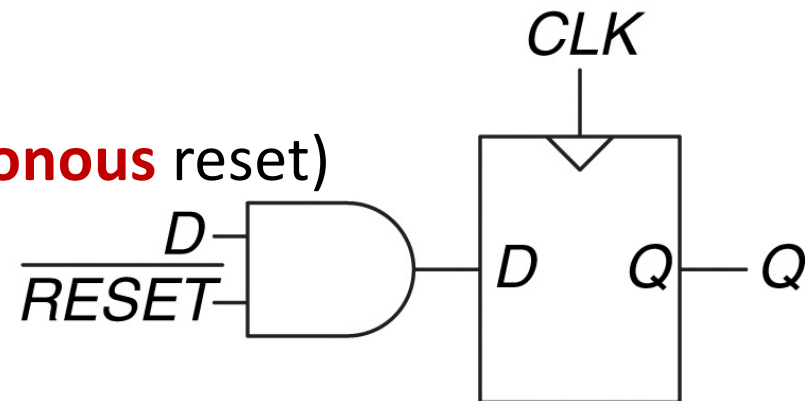
Resettable Flip-Flop

- Add another input (**RESET**)
- **Function:**
 - **RESET** = 1 (ignore **D** and reset output to 0)
 - **RESET** = 0 (ordinary D flip-flop)



- **Two types**

- Reset on clock edge only (**synchronous** reset)

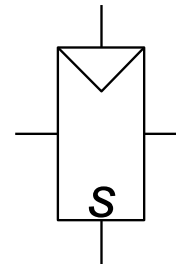
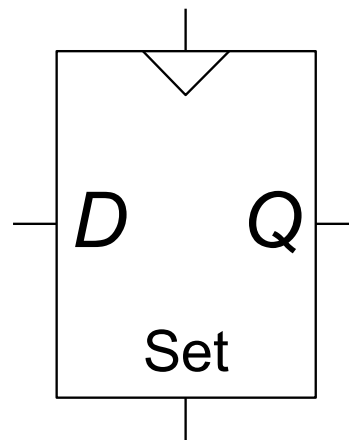


- Reset immediately (**asynchronous** reset) **Homework exercise!**

Settable Flip-Flop

- Add another input (**SET**)
- **Function:**
 - **SET** = 1 (**Q** is set to 1)
 - **SET** = 0 (ordinary D flip-flop)

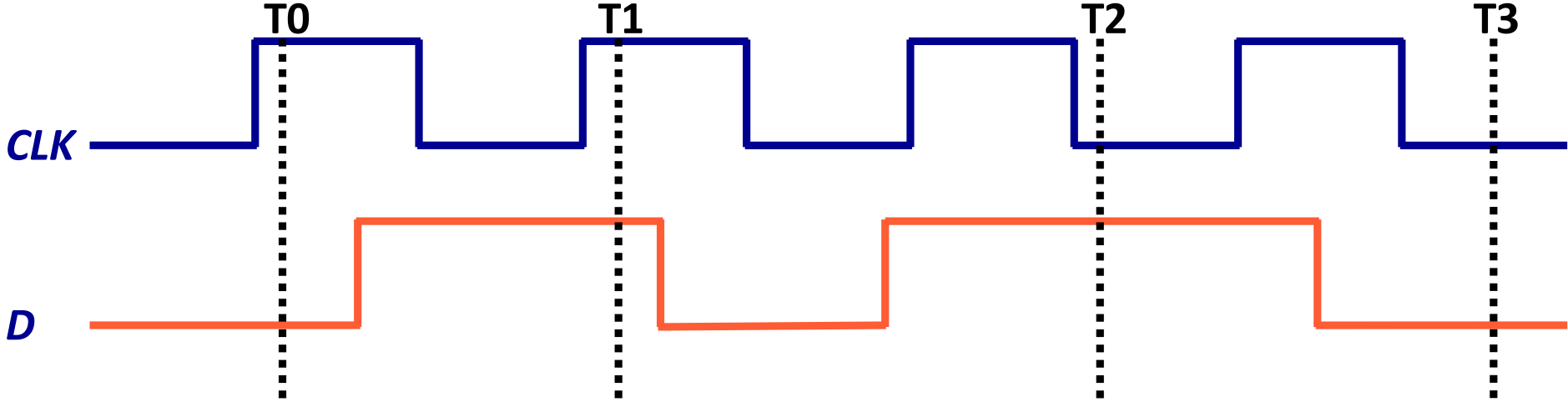
Symbols



Examples:

Latch vs. Flipflop

Example - I



Q (Latch)

Q (Flipflop)

T0

T0

T1

T1

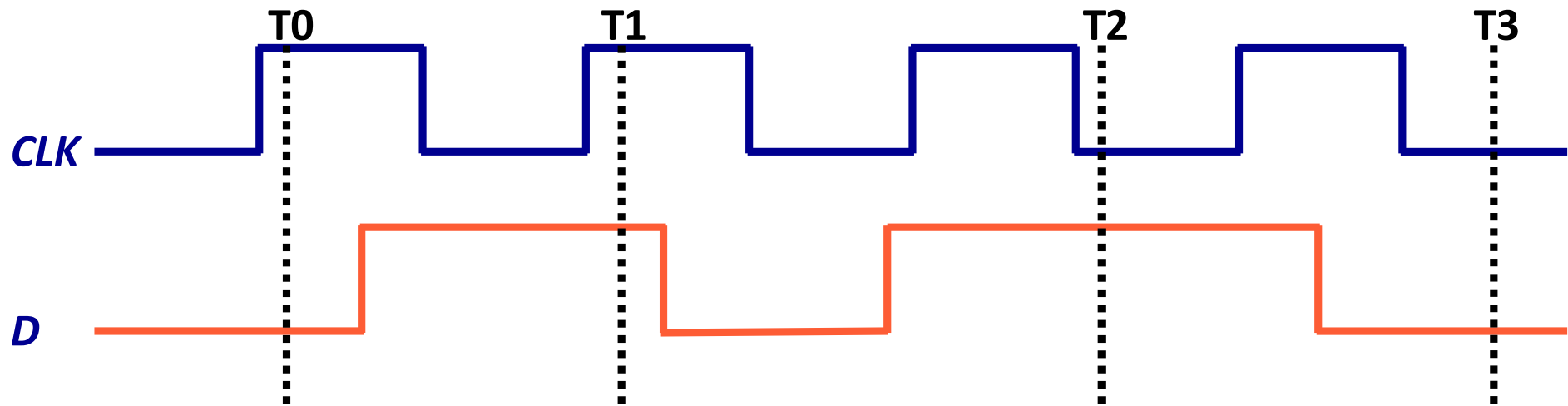
T2

T2

T3

T3

Example - I



Q (Latch)

T0 0 FALSE

T1

T2

T3

Q (Flipflop)

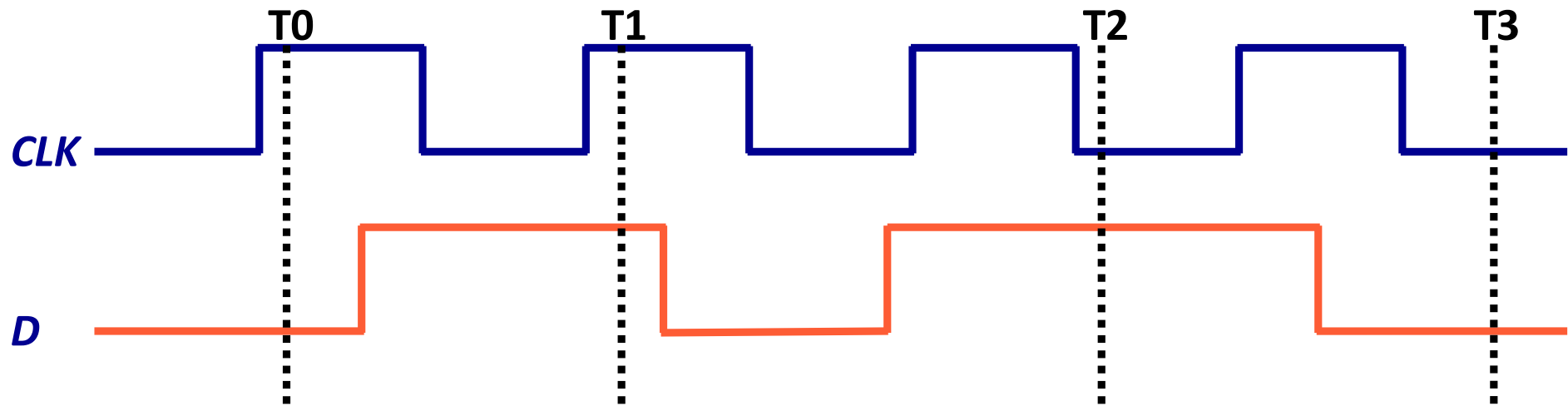
T0 0 FALSE

T1

T2

T3

Example - I



Q (Latch)

T0 0 FALSE

T1 1 TRUE

T2

T3

Q (Flipflop)

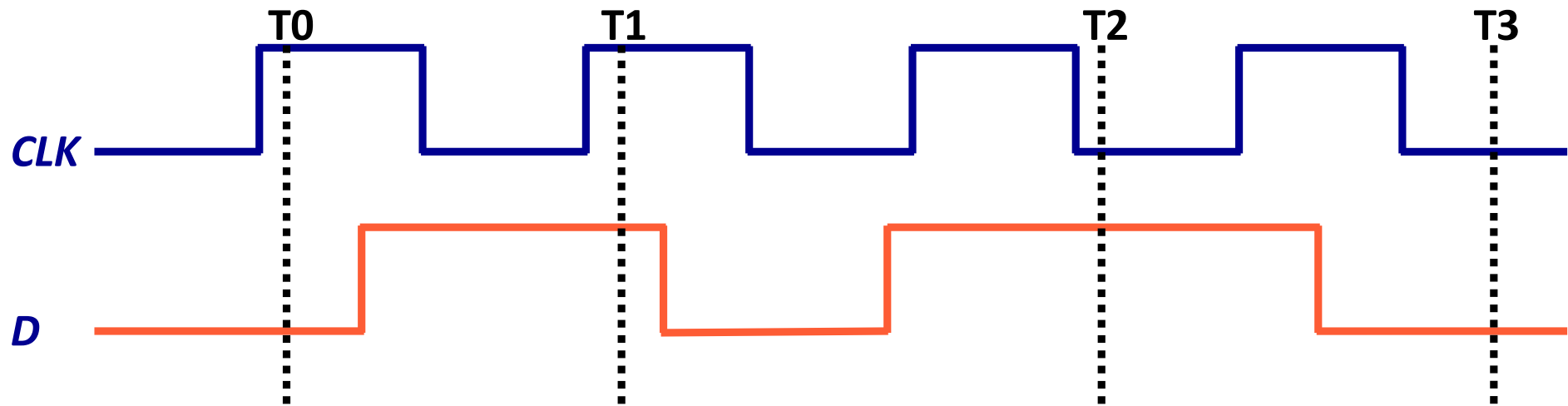
T0 0 FALSE

T1 1 TRUE

T2

T3

Example - I



Q (Latch)

T0 0 FALSE

T1 1 TRUE

T2 1 TRUE

T3

Q (Flipflop)

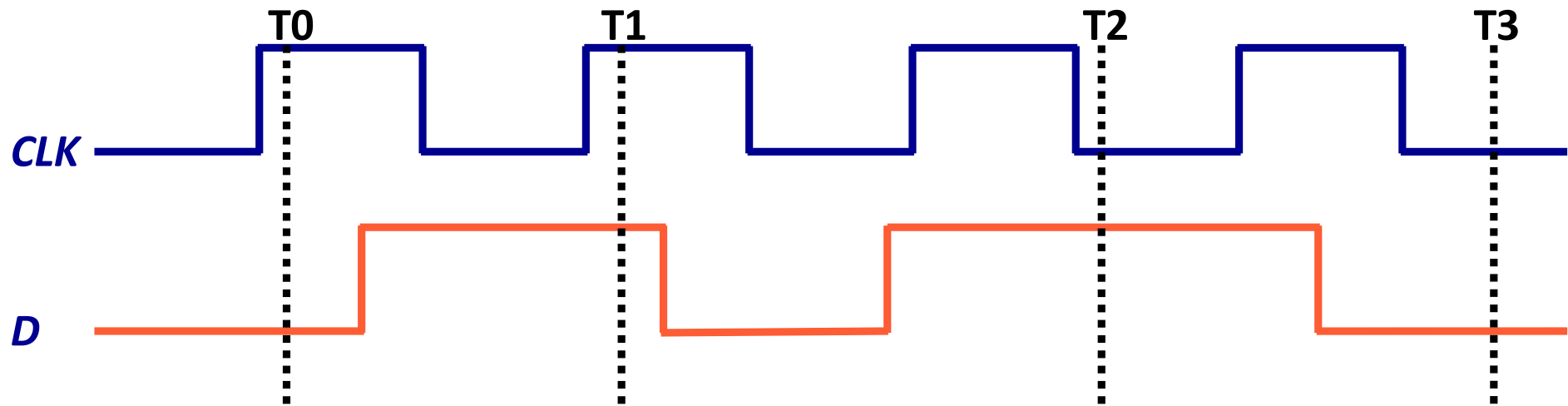
T0 0 FALSE

T1 1 TRUE

T2 1 TRUE

T3

Example - I



Q (Latch)

T0 0 FALSE

T1 1 TRUE

T2 1 TRUE

T3 0 FALSE

Q (Flipflop)

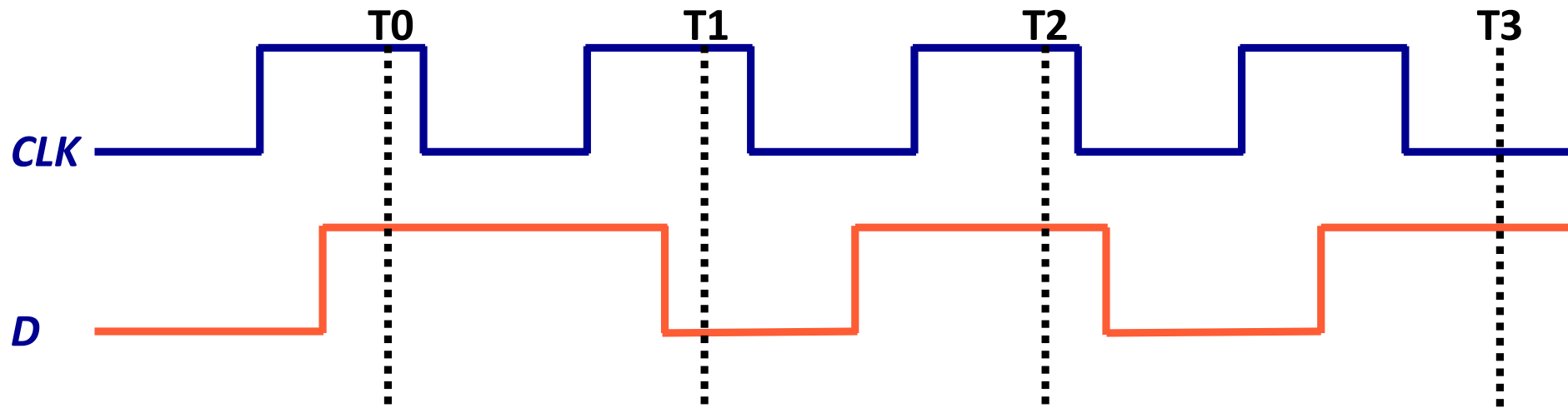
T0 0 FALSE

T1 1 TRUE

T2 1 TRUE

T3 1 TRUE

Example - II



Q (Latch)

Q (Flipflop)

T0

T0

T1

T1

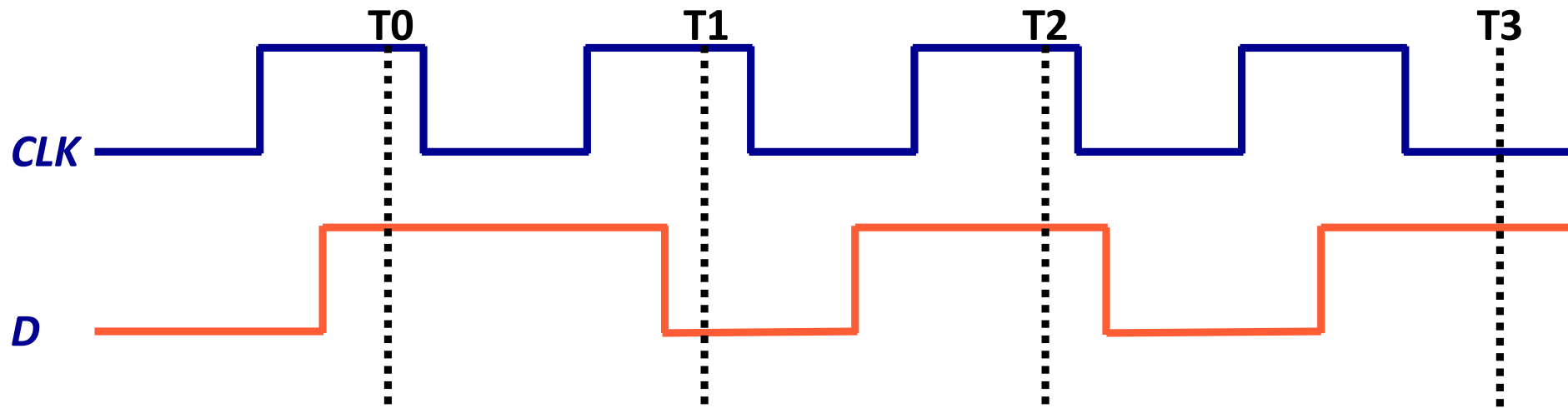
T2

T2

T3

T3

Example - II



Q (Latch)

T0 1 TRUE

T1

T2

T3

Q (Flipflop)

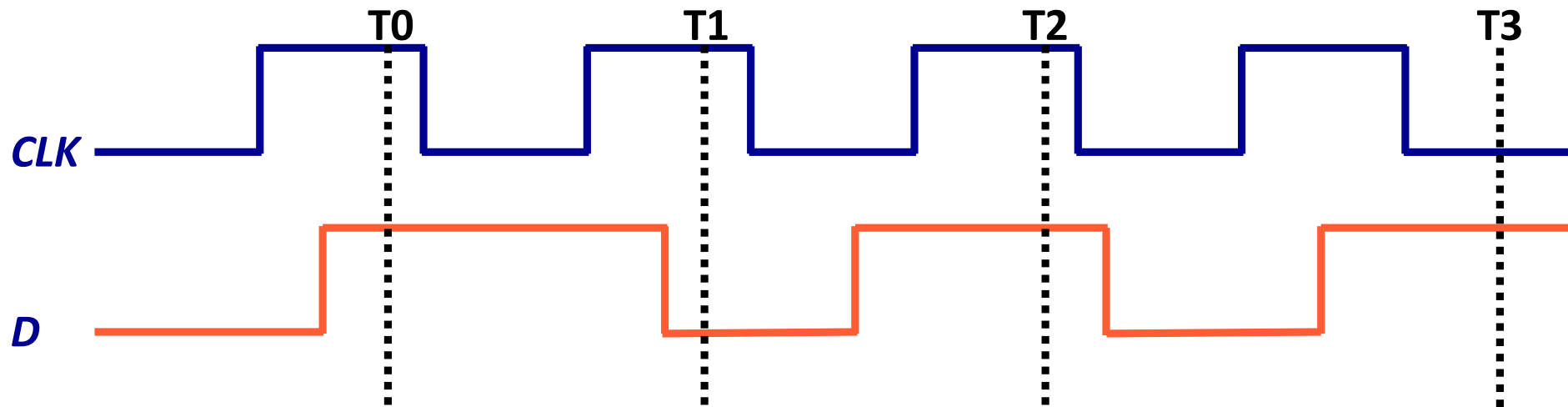
T0 0 FALSE

T1

T2

T3

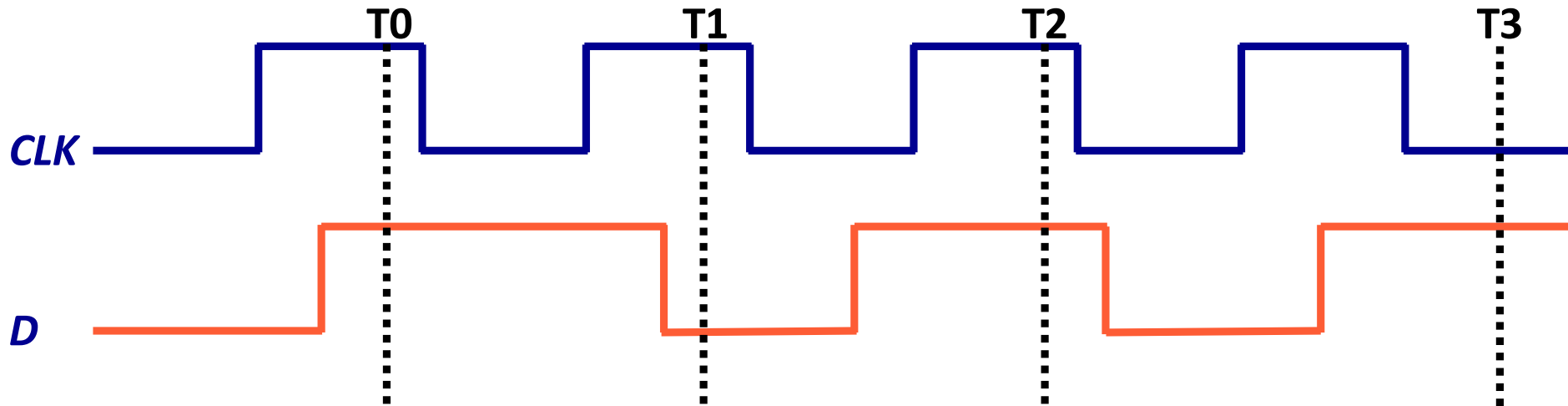
Example - II



	<u>Q (Latch)</u>
T0	1 TRUE
T1	0 FALSE
T2	
T3	

	<u>Q (Flipflop)</u>
T0	0 FALSE
T1	1 TRUE
T2	
T3	

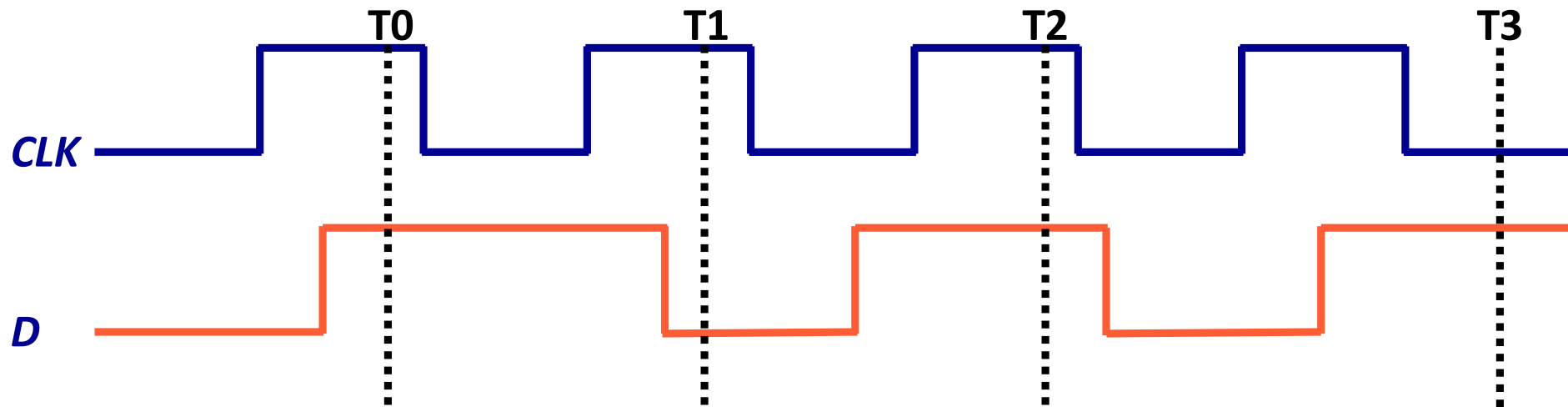
Example - II



	<u>Q (Latch)</u>
T0	1 TRUE
T1	0 FALSE
T2	1 TRUE
T3	

	<u>Q (Flipflop)</u>
T0	0 FALSE
T1	1 TRUE
T2	1 TRUE
T3	

Example - II



	<u>Q (Latch)</u>
T0	1 TRUE
T1	0 FALSE
T2	1 TRUE
T3	1 TRUE

	<u>Q (Flipflop)</u>
T0	0 FALSE
T1	1 TRUE
T2	1 TRUE
T3	0 FALSE

Memory

Memory

- Houses have **unique addresses**
 - In computers, everything is binary **0** or **1** so **addresses** are in binary as well
- Houses have stuff inside them
 - In computers, the **“only”** type of stuff is binary data
 - Binary data can be program variables, photos, videos, emails, word documents



Memory

A huge number of **uniquely identifiable** *locations* each **storing** a piece of **information**

Memory

Address: Naming or identification scheme

VS.

Data: Information stored inside a location

Memory

How many **locations**?

How big is the **piece of information** at each location?

Which **technology** is used to store information?

Memory

- **Memory** is made up 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

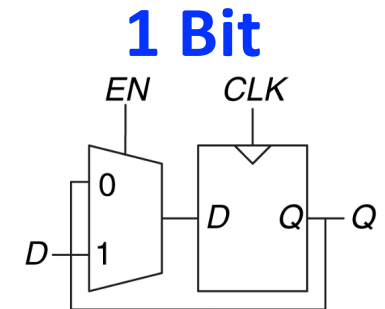
- Each unique memory location is indexed with a unique **address**. 4 locations require 2 address bits ($\log[\#locations]$)
- **Addressability**: the number of **bits** of information stored **in each location**. This example: addressability is 8 bits
- The entire set of **unique locations** in memory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

Memory

- **Memory** in almost all computers today is *byte addressable* due to historical reasons (later)
- Typical memory is **MUCH** larger consisting of billions of locations
- Two billion locations and **8-bit (1 byte)** addressability
 - Address space is **2 GB** or **2 billion 1-byte** locations
- Uniquely identifying each byte of memory *allows individual bytes of stored information* to be changed easily

Addressing Memory

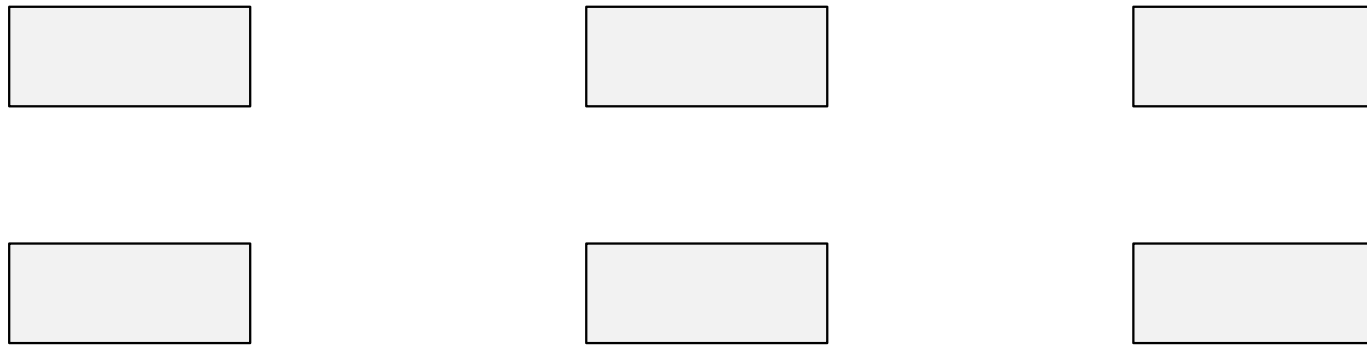
- **Let's implement a simple memory array with:**
 - 3-bit addressability & address space of 2 (total of 6 bits)
- **How can we select an address to read?**
 - Two addresses so address size is $\log(2) = 1$ bit



6-Bit Memory Array

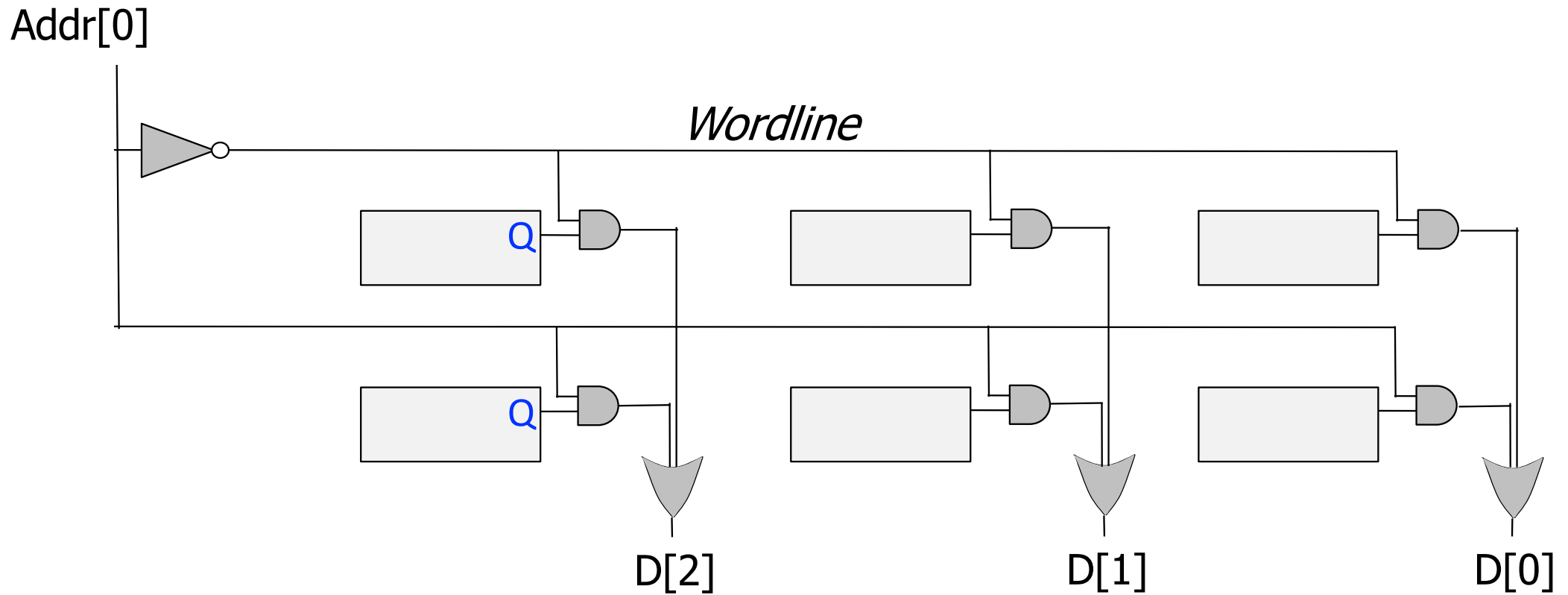
Addr(0)	Bit₂	Bit₁	Bit₀
Addr(1)	Bit₂	Bit₁	Bit₀

Reading from Memory

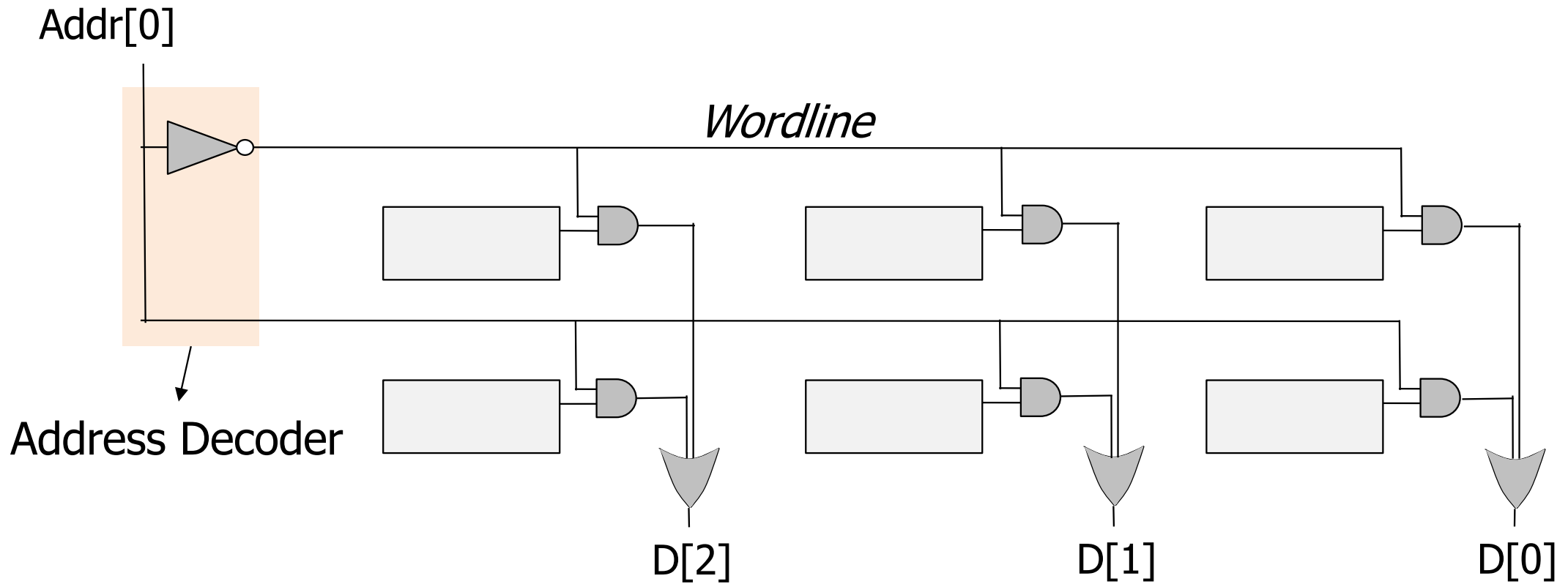


- Each “box” is a **bit cell** storing one bit
 - Can be enabled (like the enabled flip-flop with **EN**)
 - Takes an **input** (D_i) and produces an output (**D**)

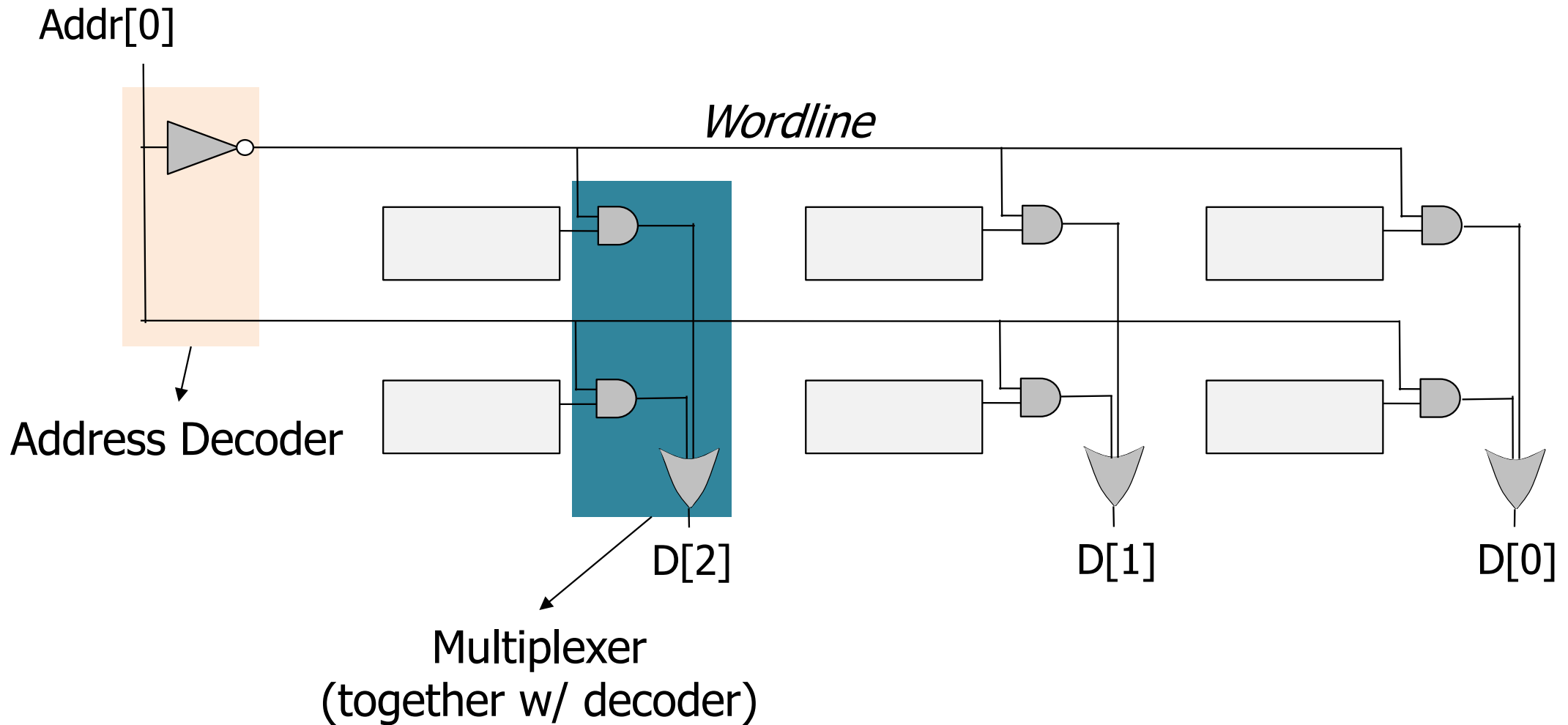
Reading from Memory



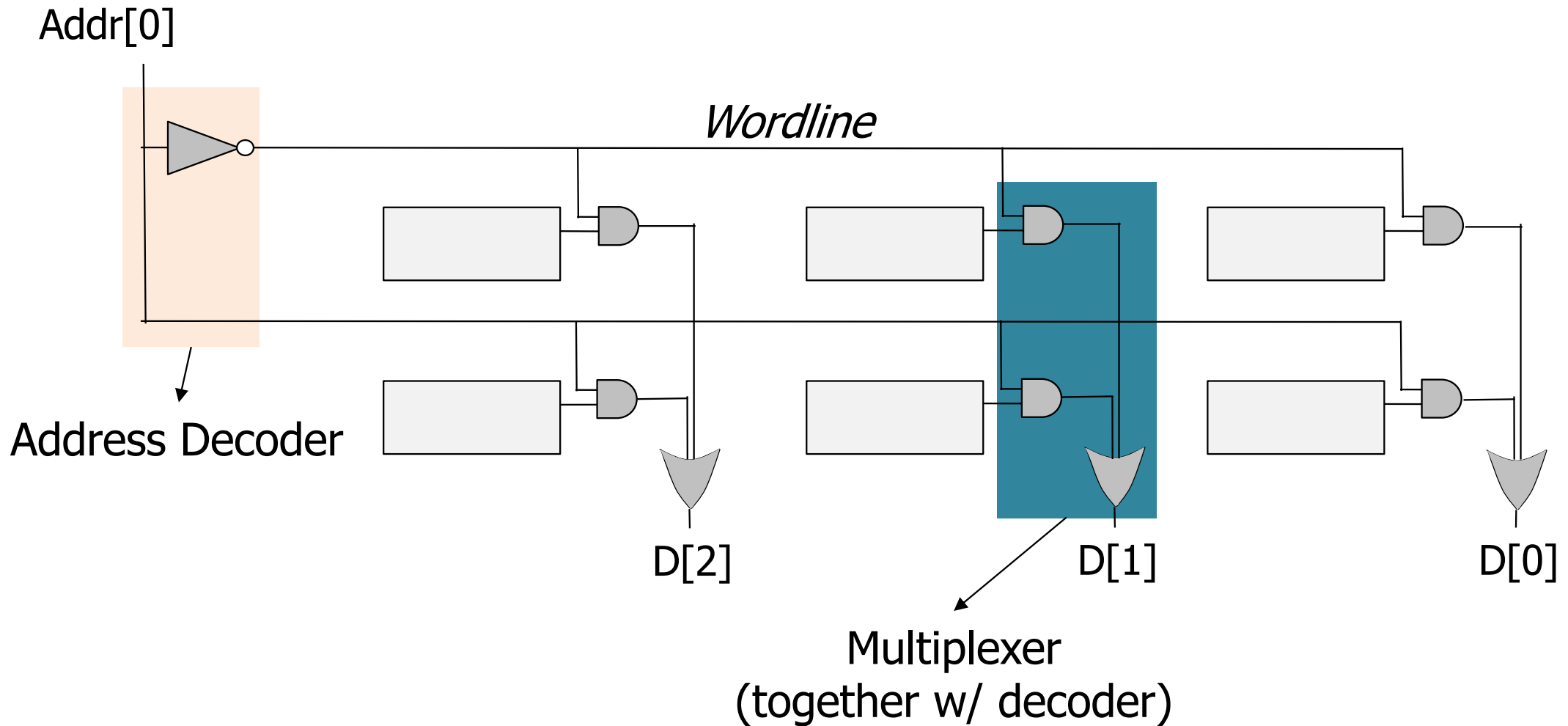
Reading from Memory



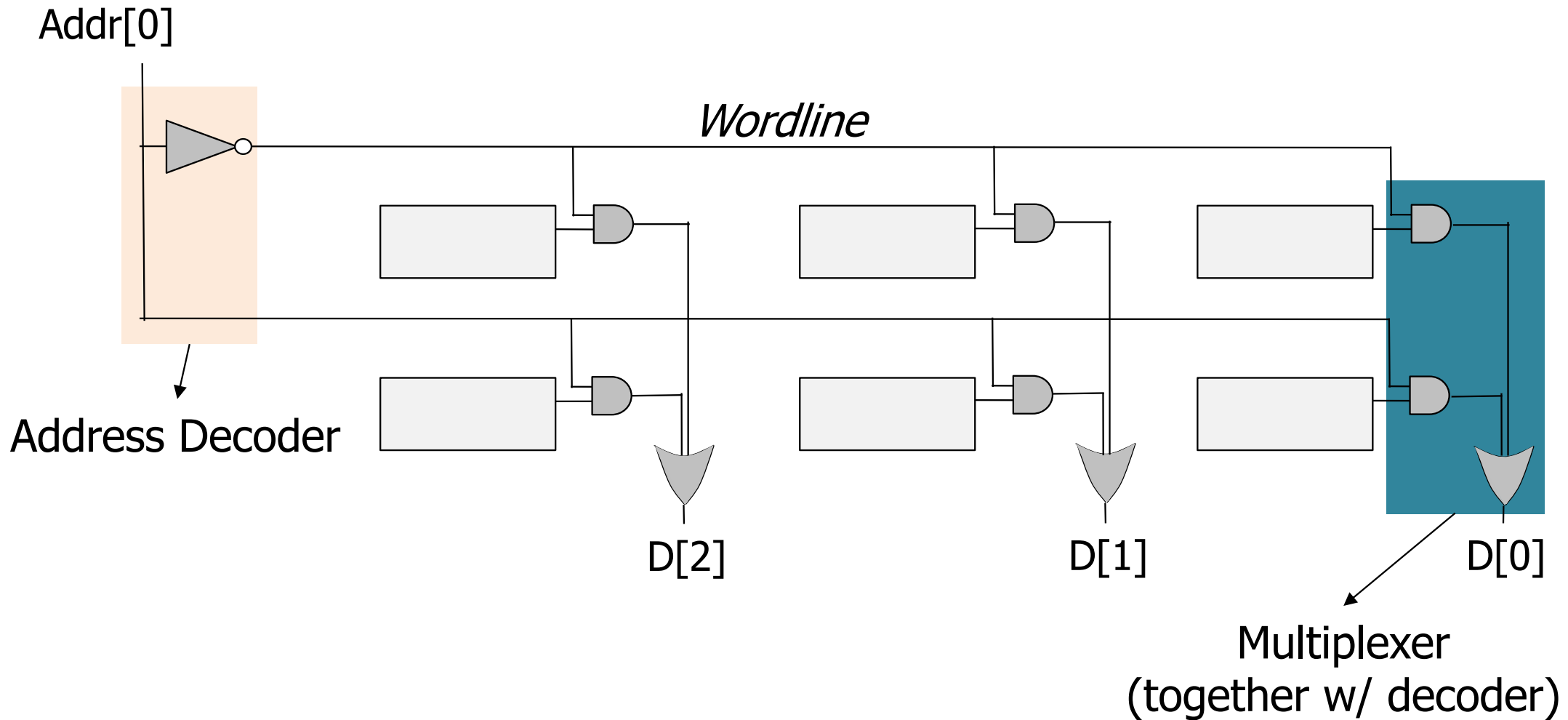
Reading from Memory



Reading from Memory



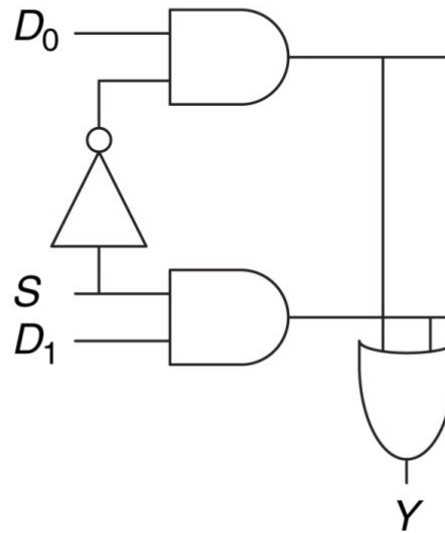
Reading from Memory



Recall: Multiplexer (MUX), or Selector

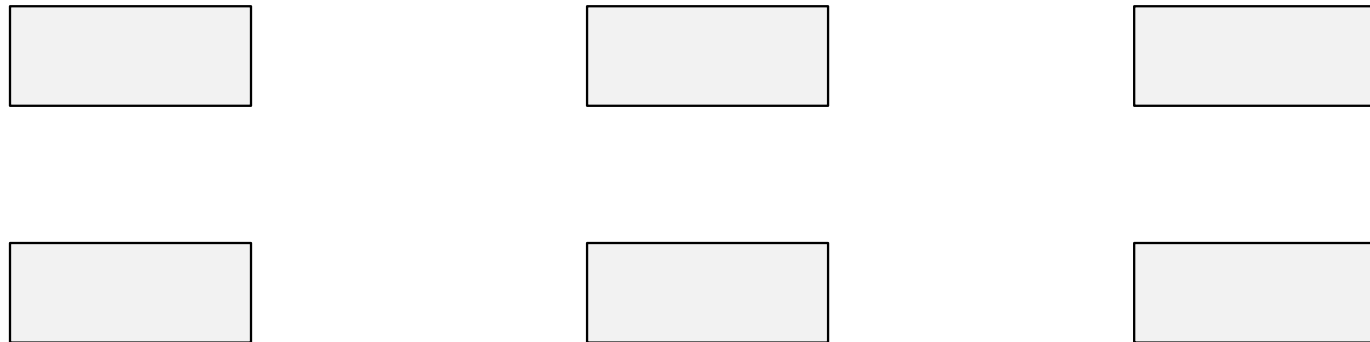
- **Selects** one of the N inputs to connect it to the output
 - based on the value of $\log_2 N$ -bit control input called **select**
- Example: 2-to-1 MUX

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



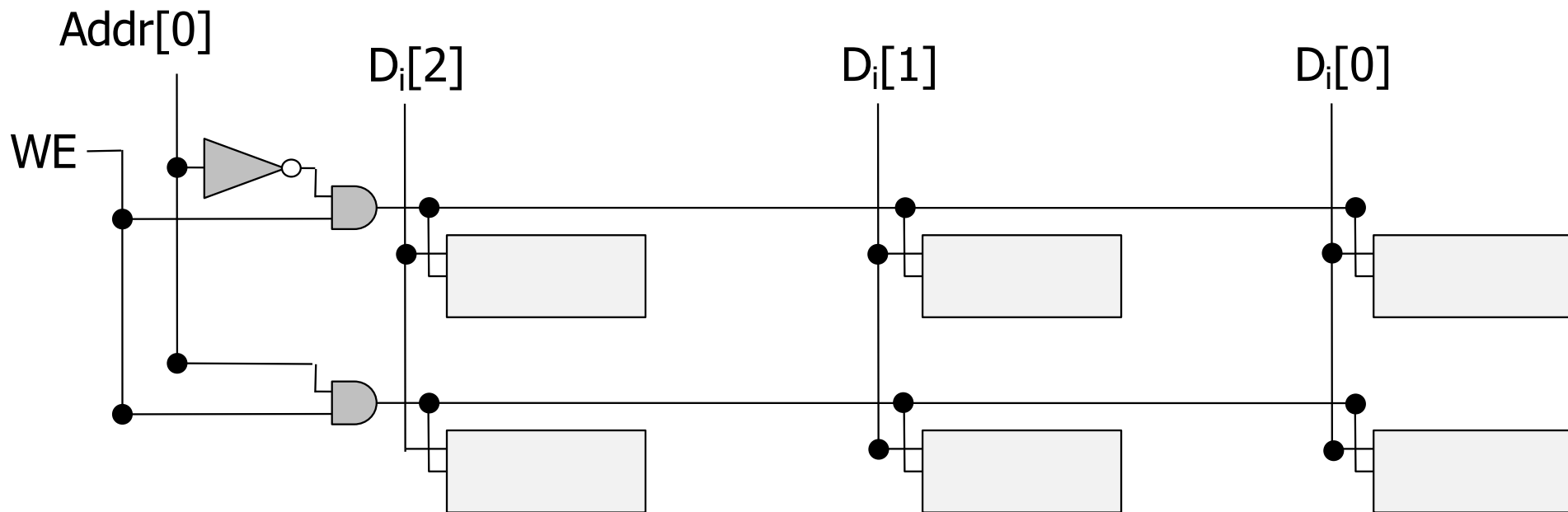
Writing to Memory

- **How can we select an address to write to it?**
 - How should we tell memory our intention to write?
 - We assert the **EN** input of the enabled flip-flop
 - How should we tell memory what data we need to write?
 - We use the **D_i** input of the flip-flop



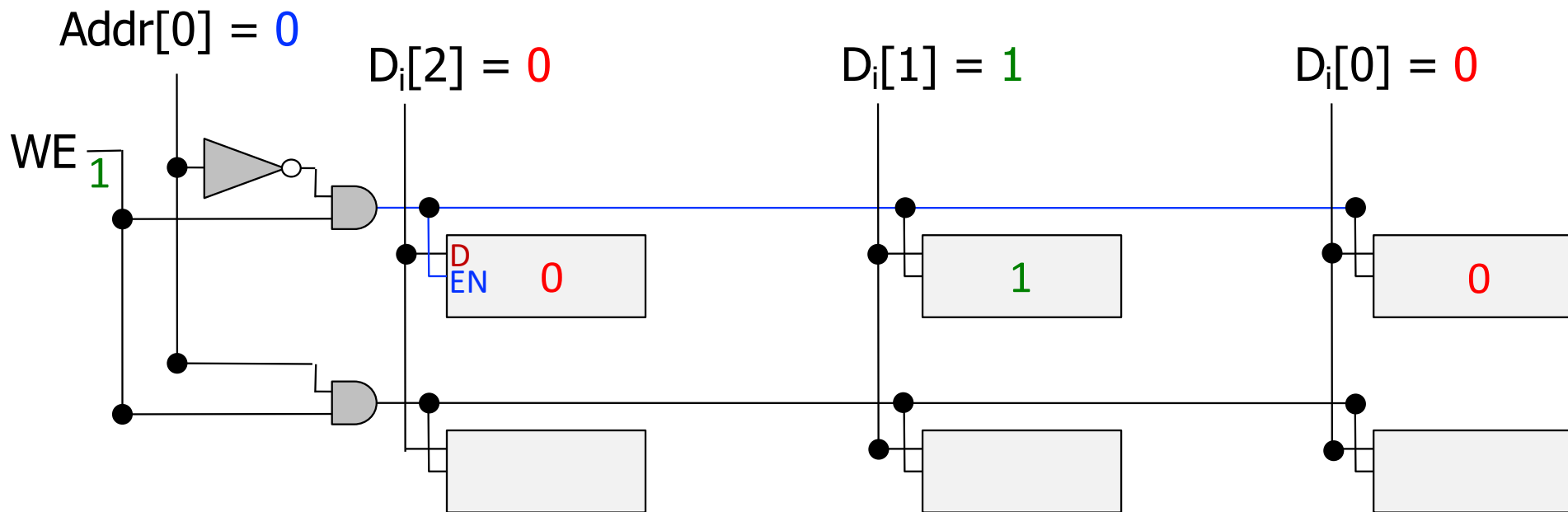
Writing to Memory

- How can we select an address to write to it?
 - Input is indicated with D_i



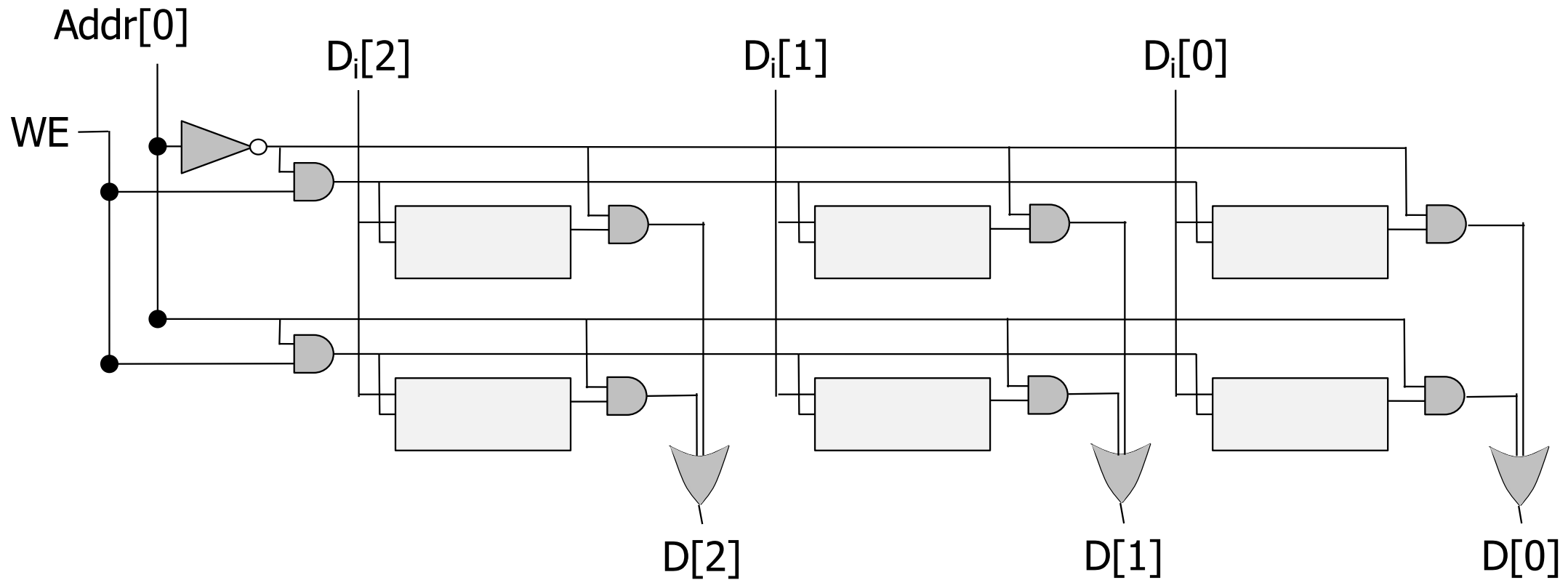
Writing to Memory

- How can we select an address to write to it?
 - Input is indicated with D_i



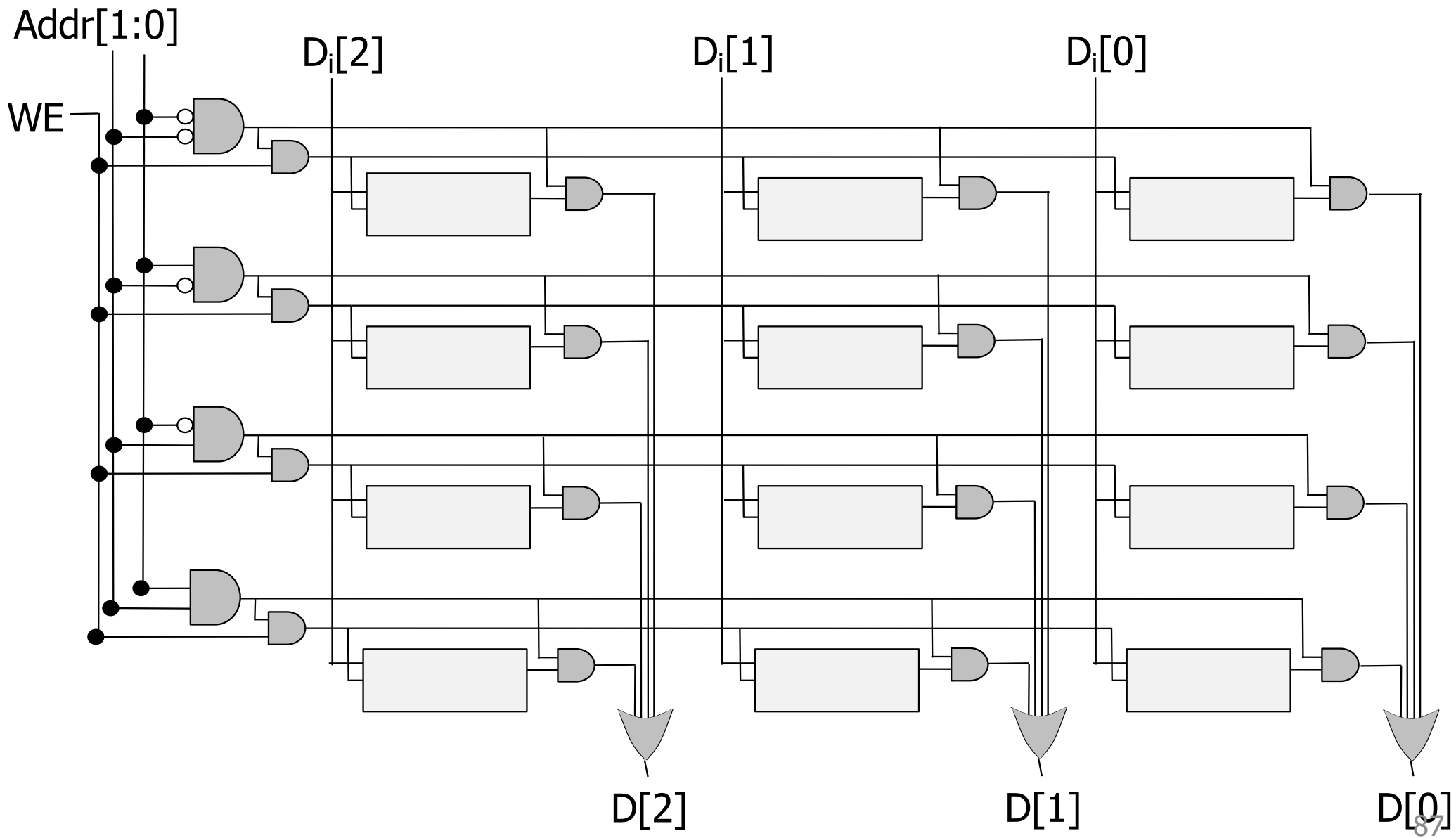
Full Memory Array

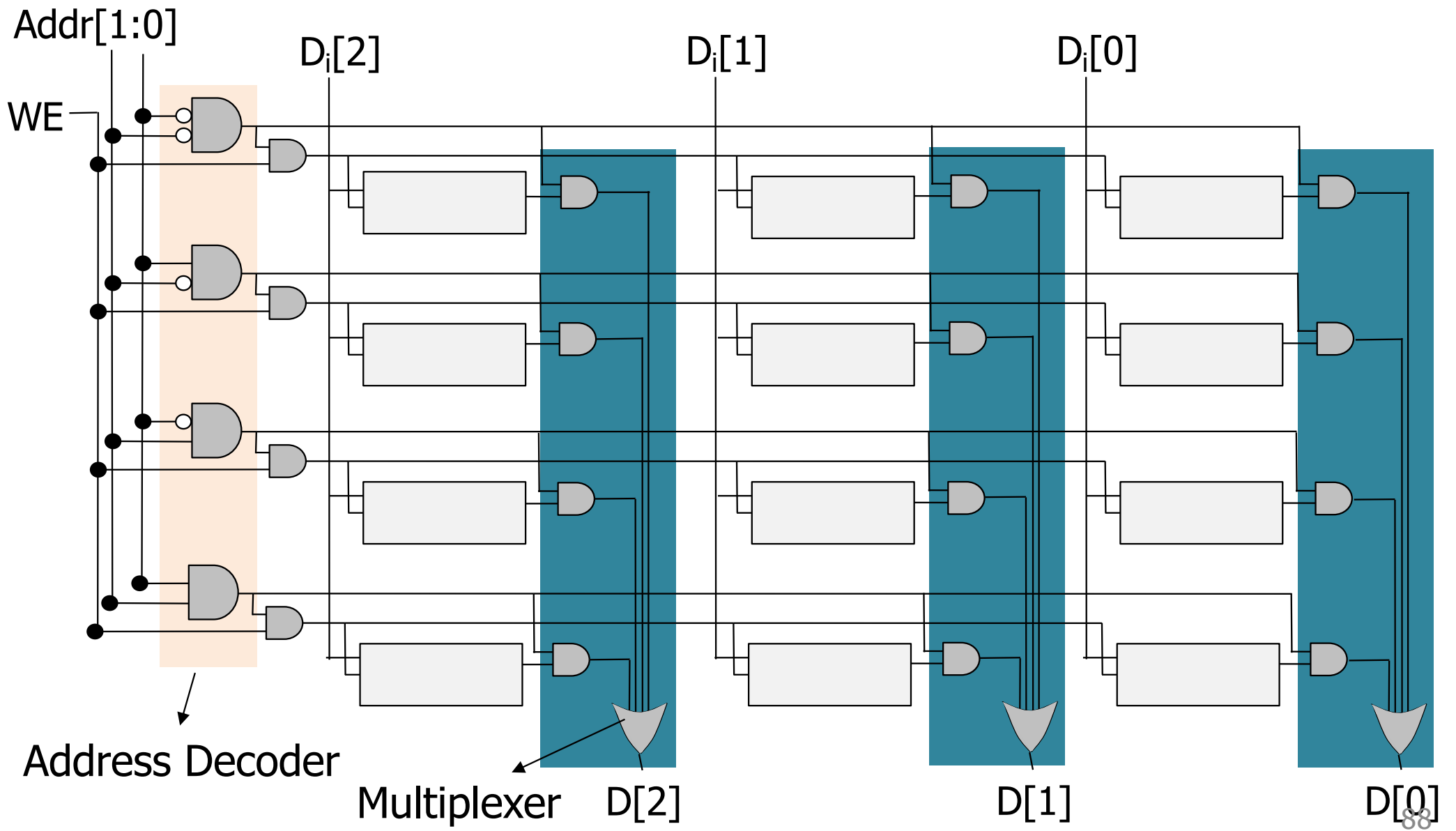
- Both reading from and writing to a memory array



A Bigger Memory Array

4 locations X 3 bits





Example: Reading Location 3

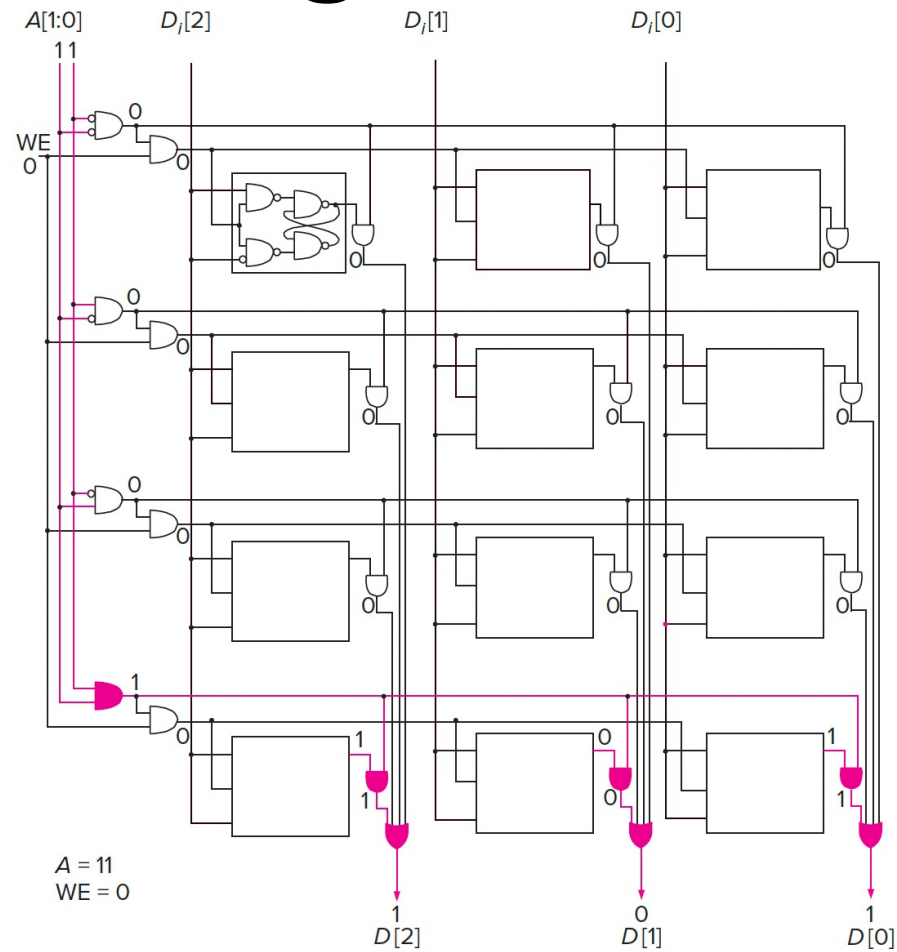
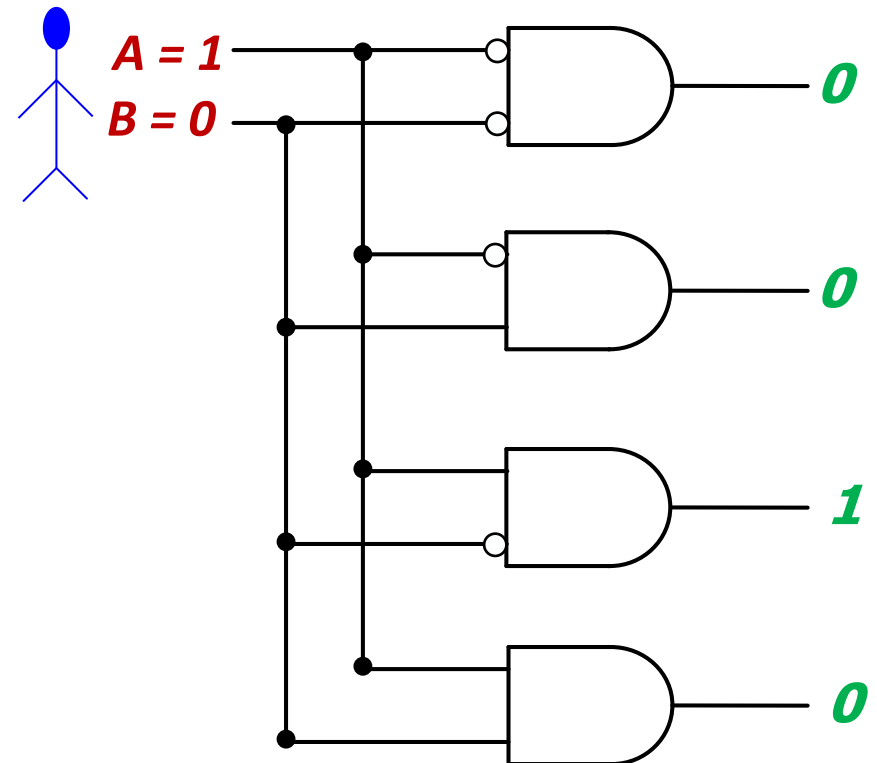


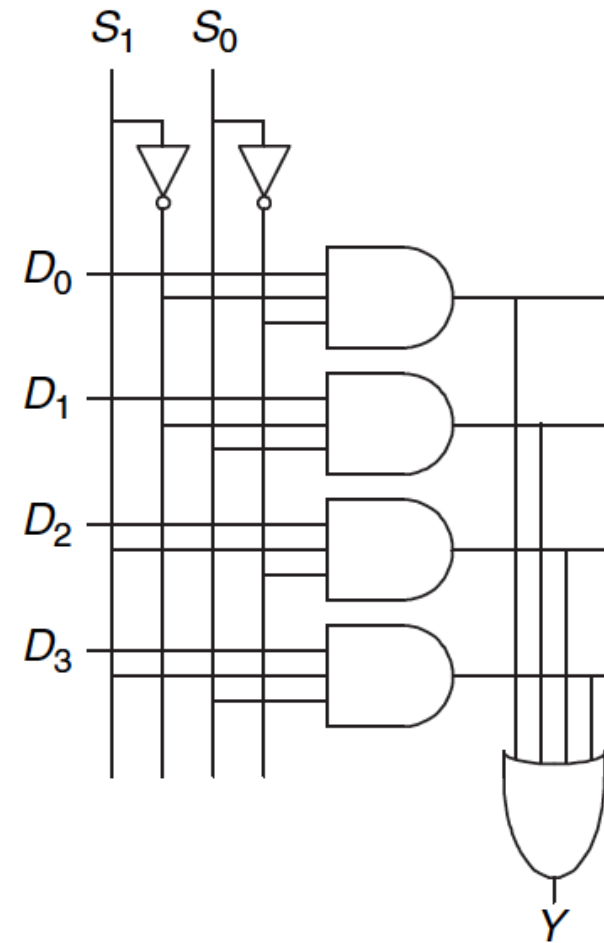
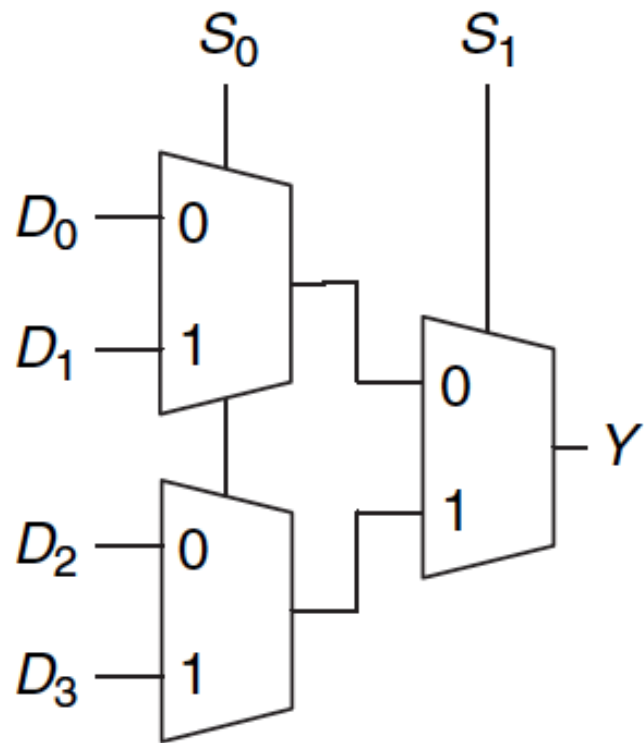
Figure 3.21 Reading location 3 in our 2²-by-3-bit memory.

Recall: Decoder

- The decoder is useful in determining how to interpret a bit pattern
 - **It could be the address of a location in memory, that the processor intends to read from**
 - **It could be an instruction in the program and the processor needs to decide what action to take (based on *instruction opcode*)**

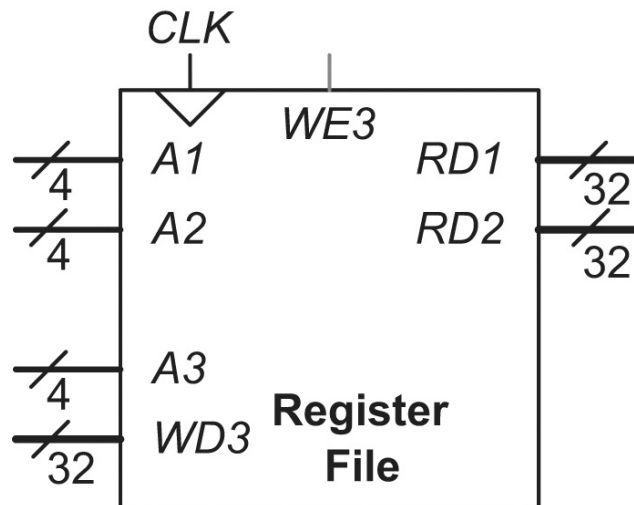


Recall: A 4-to-1 Multiplexer



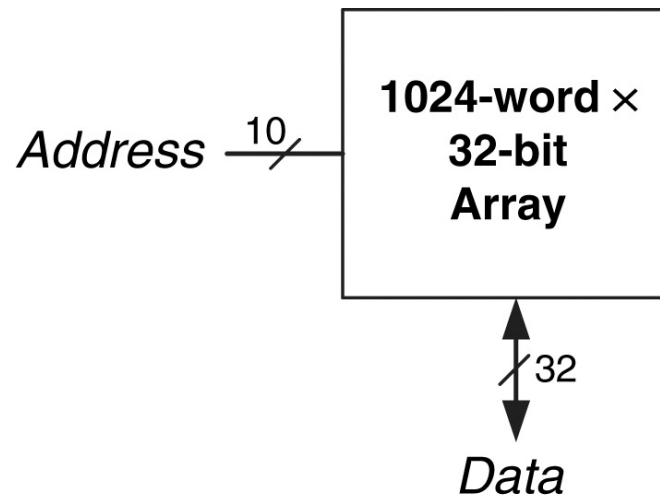
Multi-Ported Memory

- Each port gives read/write access to one memory address
- Typically, we need to access many addresses simultaneously
 - Example with two **read** ports and one **write** port



Another Representation of Memory

- What is the memory's **addressability**?
- How big is the **address space**?



Sequential Logic Circuits

- **Week 1 & 2:** Combinational circuits that process information
- **Week 3:** Boolean algebra, & circuits that can store information, & basic storage elements, & memory
- **Now,** digital logic structures that can **both** process information (i.e., make decisions) **and** store information
 - **The decision is based on both input combinations and their history**

Sequential Logic Circuits

- We have discovered 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

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



Sequential

Opens depending on past inputs

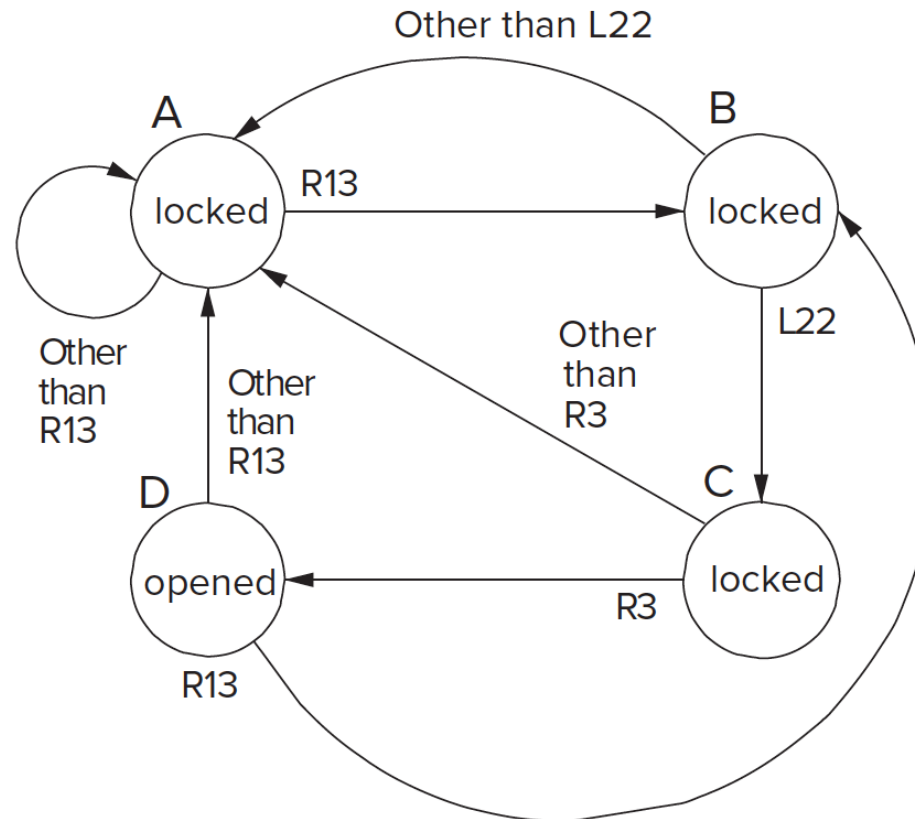
State

- In order for this lock to work, it has to **remember** past events
- If passcode is **R13-L22-R3**, *sequence of states* to unlock
 - A. Locked and no operations have been performed
 - B. Locked but **R13** completed
 - C. Locked but **R13 – L22** completed
 - D. Unlocked and **R13 – L22 – R3** completed
- 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 (L5 or L6), lock **returns** to state A



State Diagram of Sequential Lock

- Completely describes the operation of the sequential lock



Another Example: Soft Drink Machine

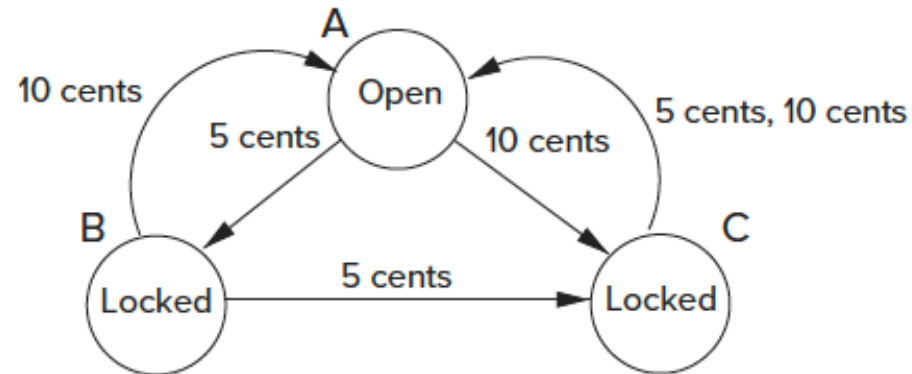


Figure 3.27 State diagram of the soft drink machine.

- There are only **three** possible states
 - A. The lock is open, so a bottle can be (or has been) removed
 - B. The lock is not open, but 5 cents have been inserted.
 - C. The lock is not open, but 10 cents have been inserted.

Another Example: Soft Drink Machine

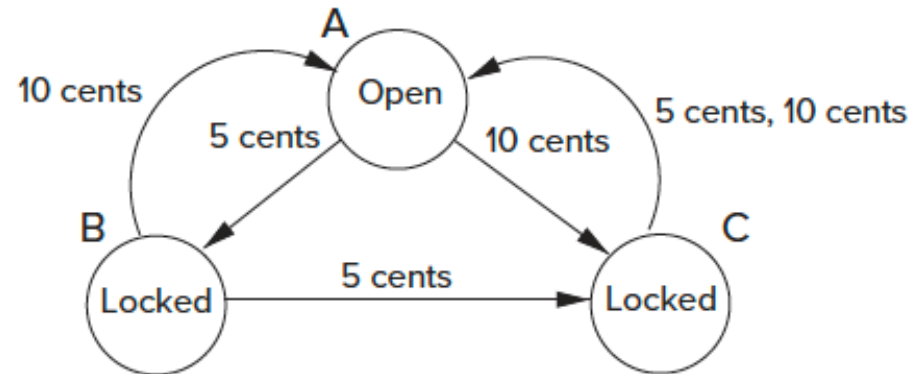


Figure 3.27 State diagram of the soft drink machine.

- One possible sequence of states is as follows

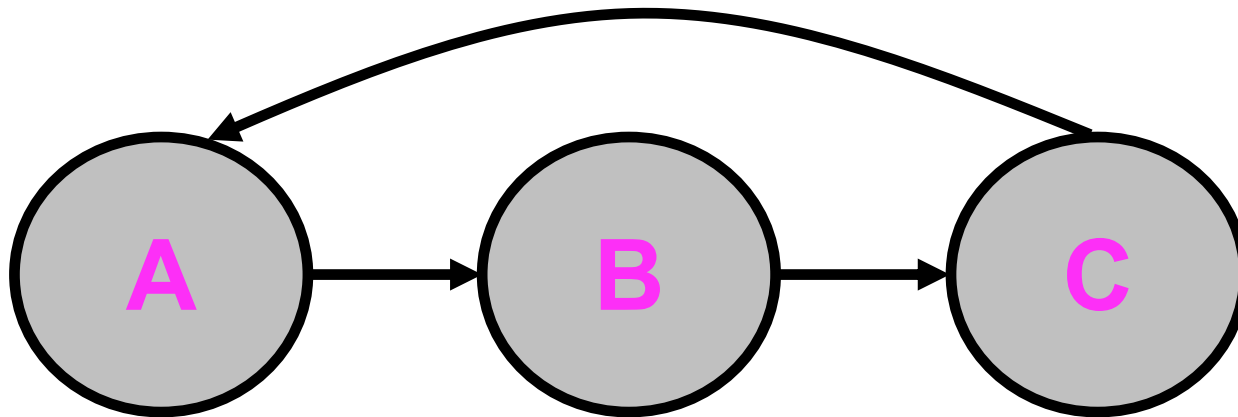


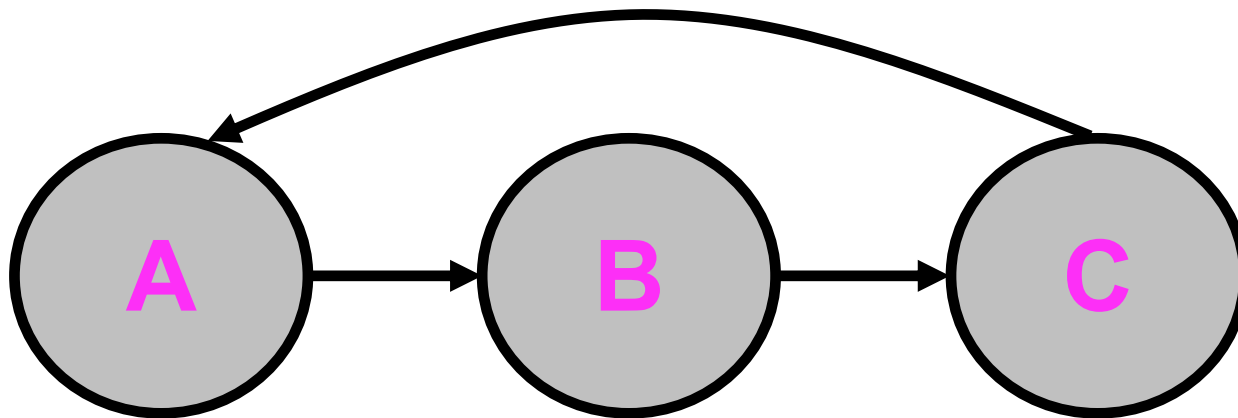
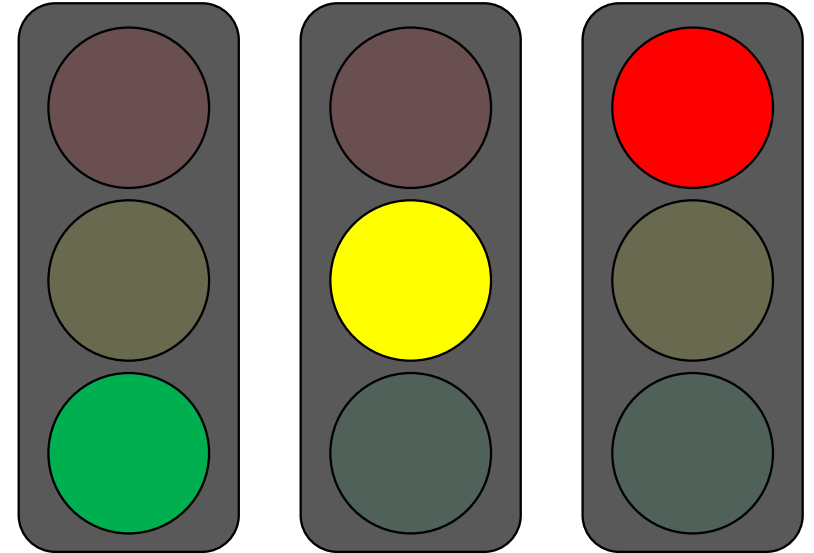
Image source: Patt and Patel, "Introduction to Computing Systems", 2nd ed., page 84.

Another Example: Traffic Light

- There are only three possible states:

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

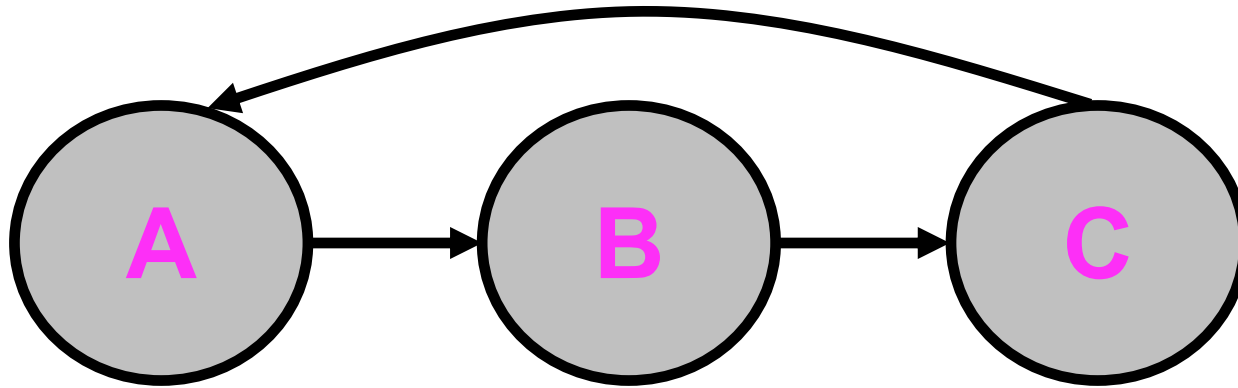
- The sequence of states is as follows



Asynchronous vs. Synchronous State Changes

- Sequential lock we saw is an **asynchronous** machine
 - State transition occur when they occur
 - There is nothing that *synchronizes* when each state transition must occur
- Most modern computers are **synchronous** machines
 - State transitions take place after fixed units of time
 - Controlled by the clock that dictates when transitions occur
- These are two different design paradigms, with **tradeoffs**

Changing State & The Notion of Clock




- When should the vending machine change state from **A** to **B**?
- When should the traffic light change from one state to another?

Changing State & The Notion of Clock

- When should the machine change state from **A** to **B**?
- We need a **clock** to dictate when to change state
 - Clock alternates between **0** & **1**



- At the start of a clock cycle (), system state changes
 - During a clock cycle, the **state stays constant**
 - The machine stays in a specific state an **equal amount of time**

Changing State & The Notion of Clock

- **Clock** is a general mechanism that **triggers transition from one state to another** in a (**synchronous**) 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 delay

Asynchronous vs. Synchronous State Changes

- Sequential lock we saw is an **asynchronous** machine
 - State transition occur when they occur
 - There is nothing that synchronizes when each state transition must occur
- Most modern computers are **synchronous** machines
 - State transitions take place after fixed units of time
 - Controlled in part by a clock, as we will see soon
- These are two different design paradigms, with **tradeoffs**
 - Synchronous control can be easier to get correct when the system consists of **many components and many states**
 - Asynchronous control can be more efficient (no clock **overheads**)

We will assume synchronous systems in this course

"the art of directing the **simultaneous** performance of several players or singers by the use of gesture." Wikipedia, Conducting



We will assume synchronous systems in this course

Why are Arbitrary Sequential Circuits a Bad Idea?

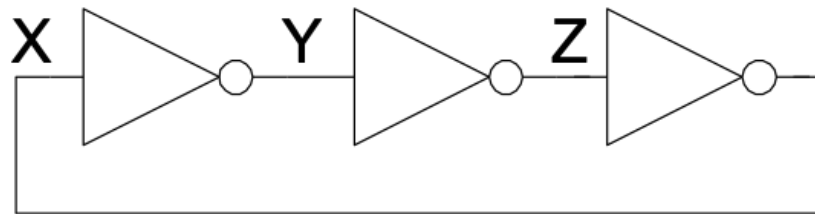
Reading: Section 3.4 of H&H

Arbitrary Sequential Circuits

- **Important:** We need to **discipline** ourselves and only build synchronous sequential circuits
- State is **synchronized** at the clock edges
- Let's examine two arbitrary sequential circuits

Ring Oscillator

- What does the circuit below do?
 - One output, No inputs**



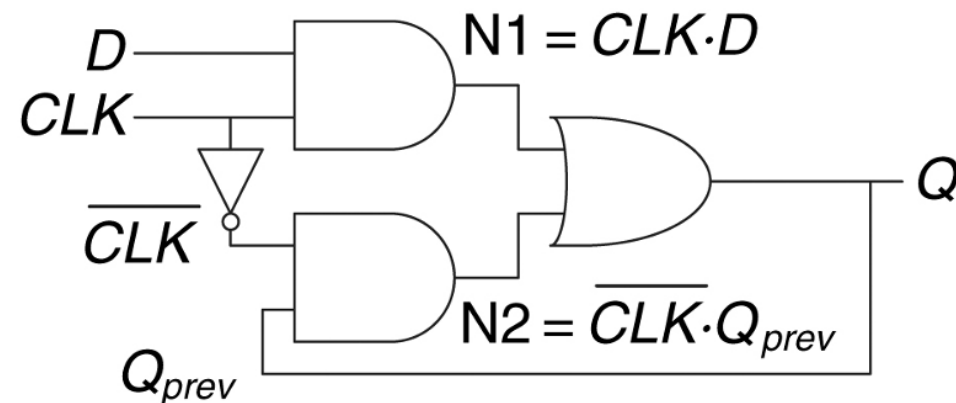
- X **oscillates** b/w **0** and **1** (prove it assuming $X = 0$)
 - No stable state (**Unstable** or **Astable**)
- If the propagation delay of each inverter is **1 ns**, then the clock period is 6 ns

Example 3.3 of H&H

Asynchronous D Latch

- What does the circuit below do?
 - CLK = D = 1 (*transparent*, Q = 1)
 - CLK = 0 (Q = Q_{prev})

$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$



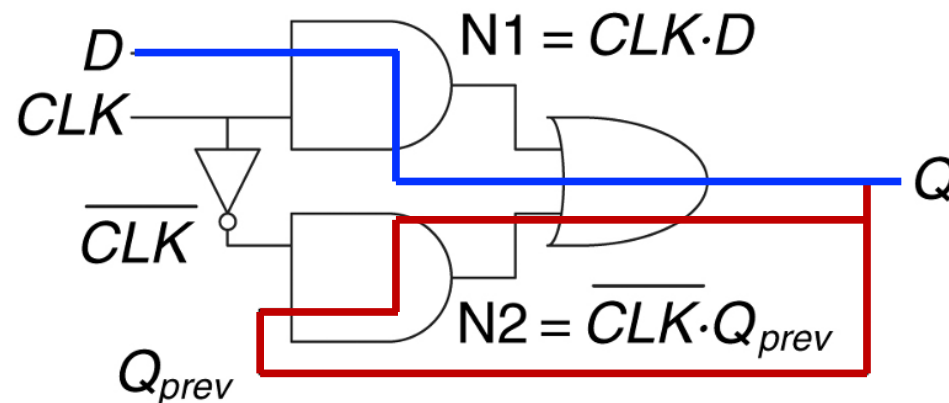
Example 3.4 (page 119) of H&H

Asynchronous D Latch

- What does the circuit below do?
 - CLK = D = 1 (*transparent*, Q = 1)
 - CLK = 0 (Q = Q_{prev})

CLK	D	Q _{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

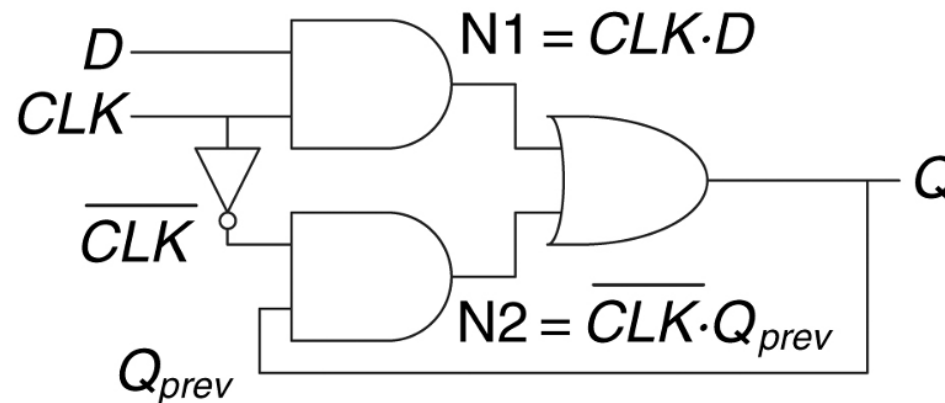
$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$



Example 3.4 (page 119) of H&H

Asynchronous D Latch

- What if $t_{INV} \gg t_{AND}$ and $t_{INV} \gg t_{OR}$?
 - Node N1 and Q may both fall before \overline{CLK} changes
 - Q gets **stuck** at 0
 - Race condition:** The path through **CLK to Q** is **faster** than \overline{CLK} to Q



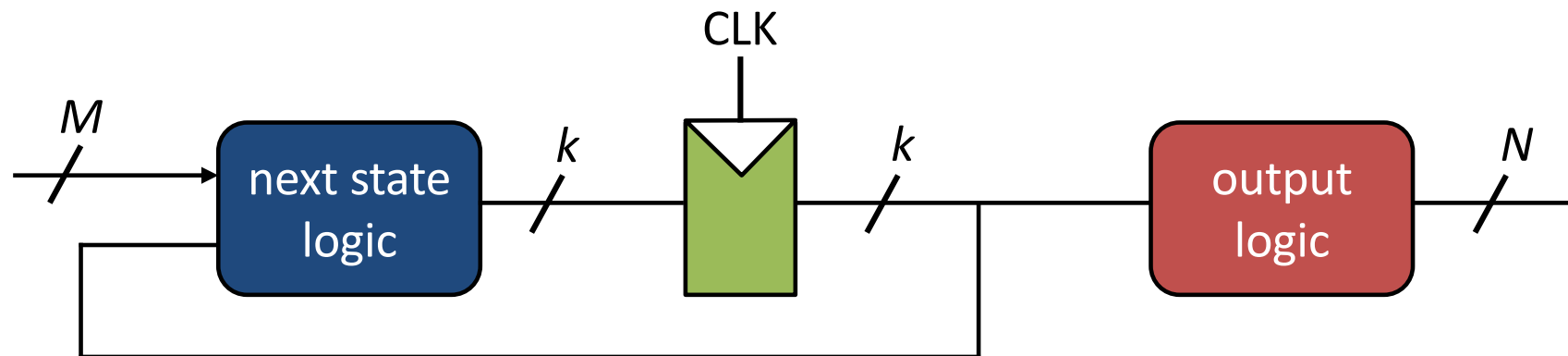
Example 3.4 (page 119) of H&H

Takeaways

- When outputs are fed back directly back to inputs, these are called **cyclic** paths
- Combinational logic has **NO** cyclic paths
 - Outputs settle after **propagation** delay
- Circuits with cyclic paths are called **asynchronous** circuits
 - **Difficult to analyze**
 - **Timing issues (race conditions, oscillations)**
 - **May work in one set of conditions but not in another**

Synchronous Sequential Circuits

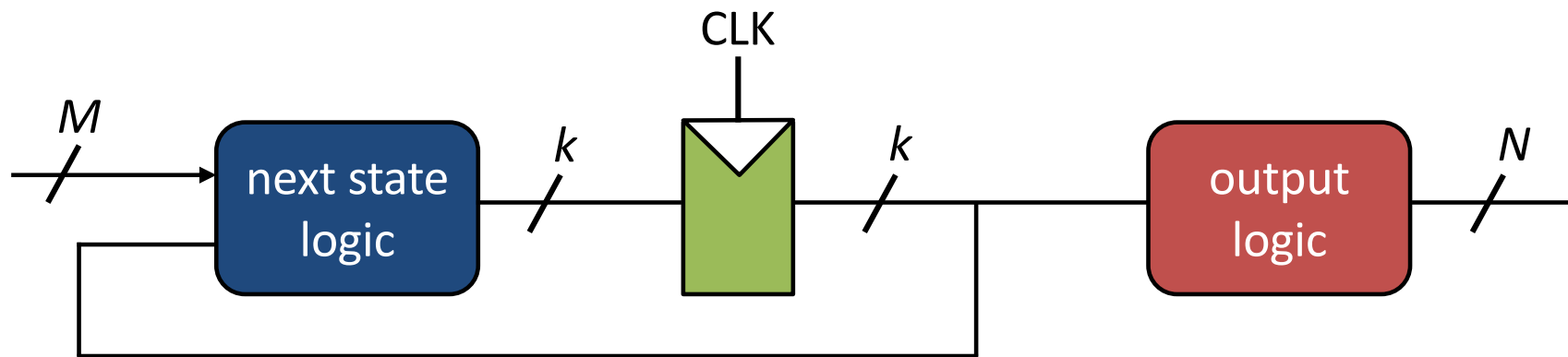
- What is the problem with asynchronous circuits?
 - *Cyclic paths lead to races and unstable behavior*
- **Solution:** Break the cyclic paths by **inserting** registers somewhere in the path
- Registers contain **state**, and **synchronized** to the clock



Section 3.3.2 of H&H

Synchronous Sequential Circuits

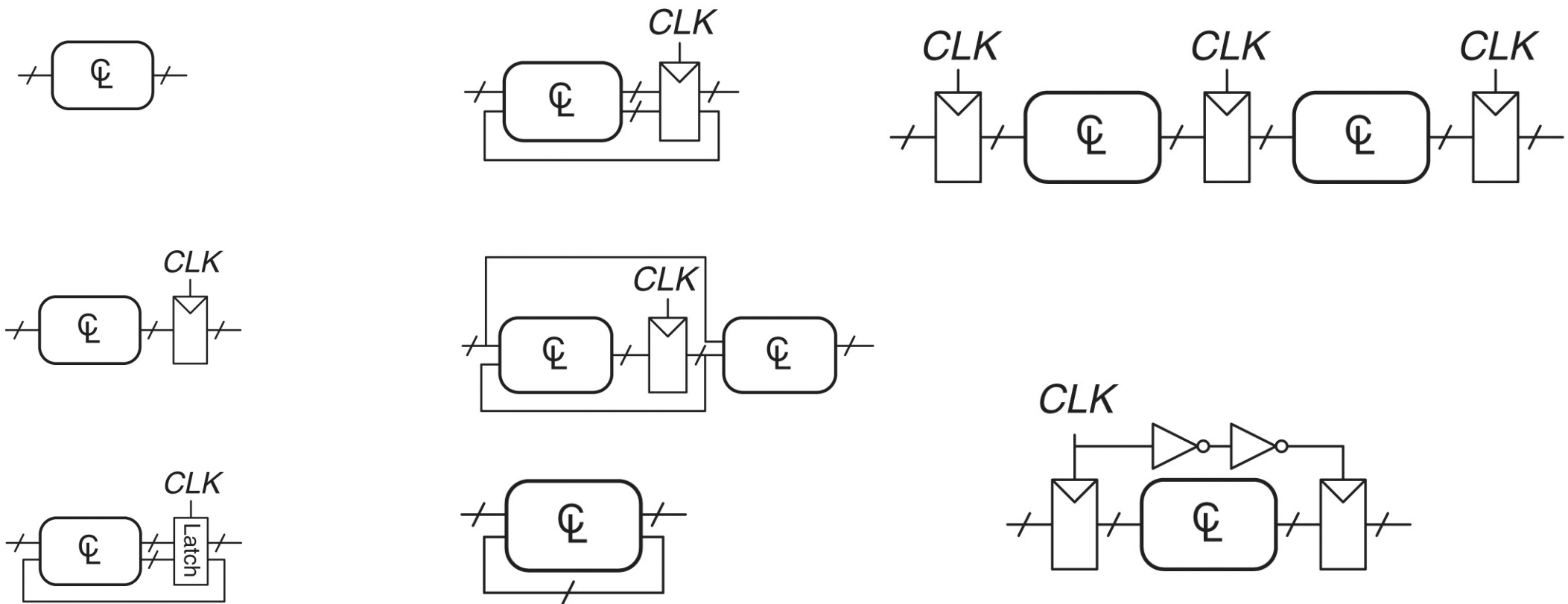
General Rule: *If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated*



Composition Rules

- Every circuit element is either a register or a combinational element
- At least one element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register

Which circuits are synchronous sequential?



Example 3.5 of H&H

Finite State Machines

Compulsory Reading: Section 3.4 of H&H

Finite State Machines

- What is a **Finite State Machine** (FSM)?
 - **A discrete-time model** of a stateful system
 - A finite set of states a system can be in
- An **FSM** pictorially shows
 - The set of all possible states that a system can be in
 - How the system transitions from one state to another
- An **FSM** can model
 - A traffic light, an elevator, microwave, microprocessor, fan speed, car lock

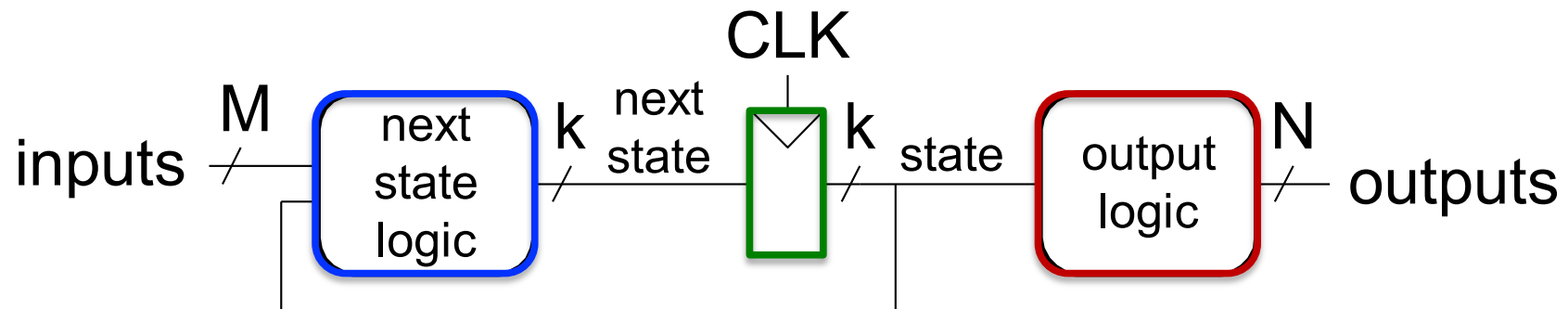
FSMs Consist of:

■ Five elements:

- A **finite** number of **states**
 - **State**: snapshot of all relevant elements of the system at the time of the snapshot
- A **finite** number of external **inputs**
- A **finite** number of external **outputs**
- An explicit **specification of all state transitions**
 - **How to get from one state to another**
- An explicit **specification of what determines each external output value**
 - If state is A, then output is 10

Finite State Machines (FSMs)

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

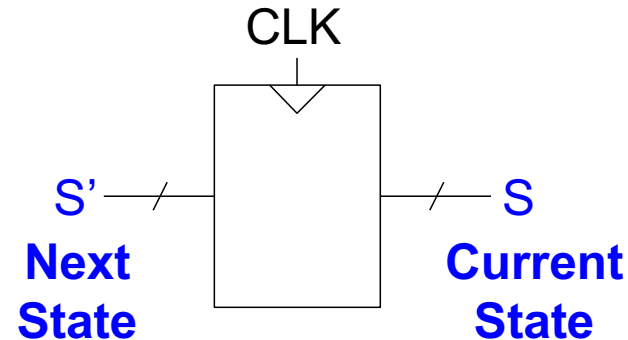


FSMs Consist of:

- **Sequential circuits**

- **State register(s)**

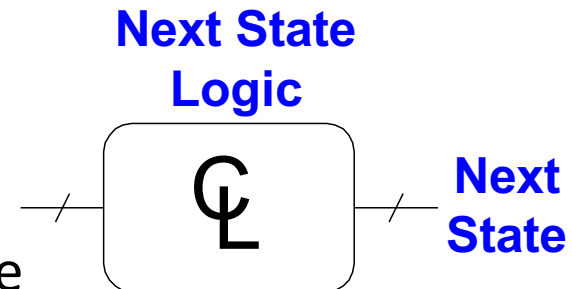
- Store the current state and
 - Provide the next state at the clock edge



- **Combinational Circuits**

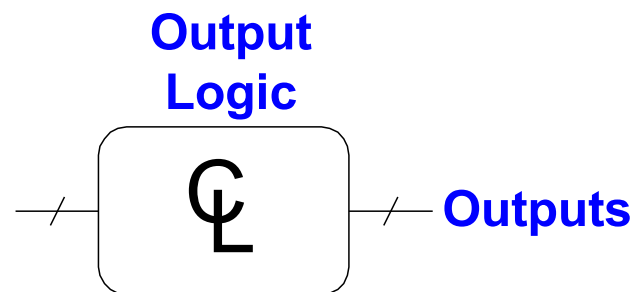
- **Next state logic**

- Determines what the next state will be



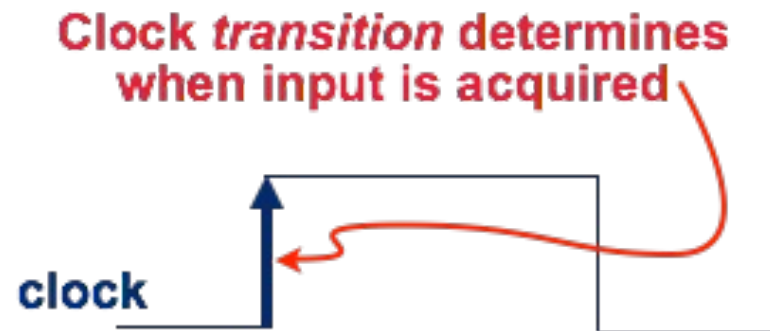
- **Output logic**

- Generates the outputs



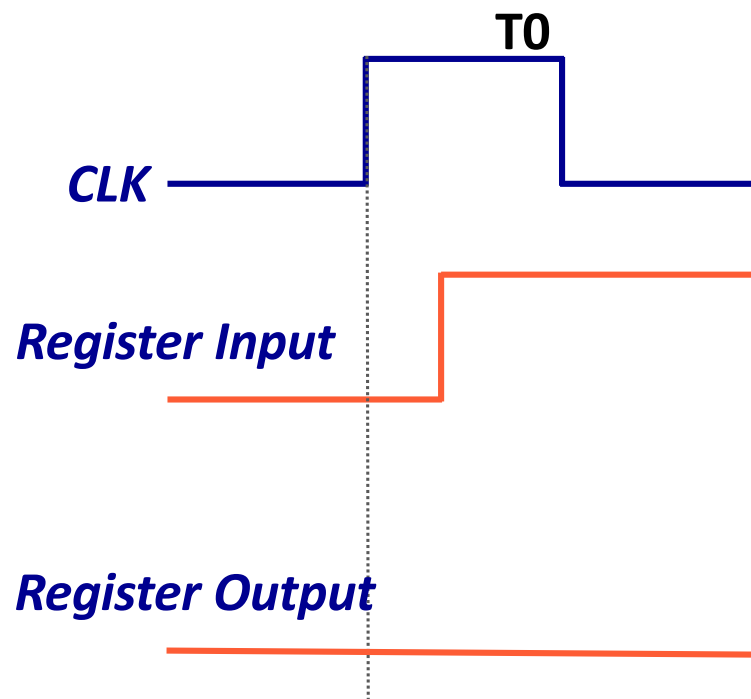
State Register

- Why do we need flip-flops (**NOT latches**) to implement a **state register**
- Properties of state register
 - We need to store data at the **beginning** of every clock cycle



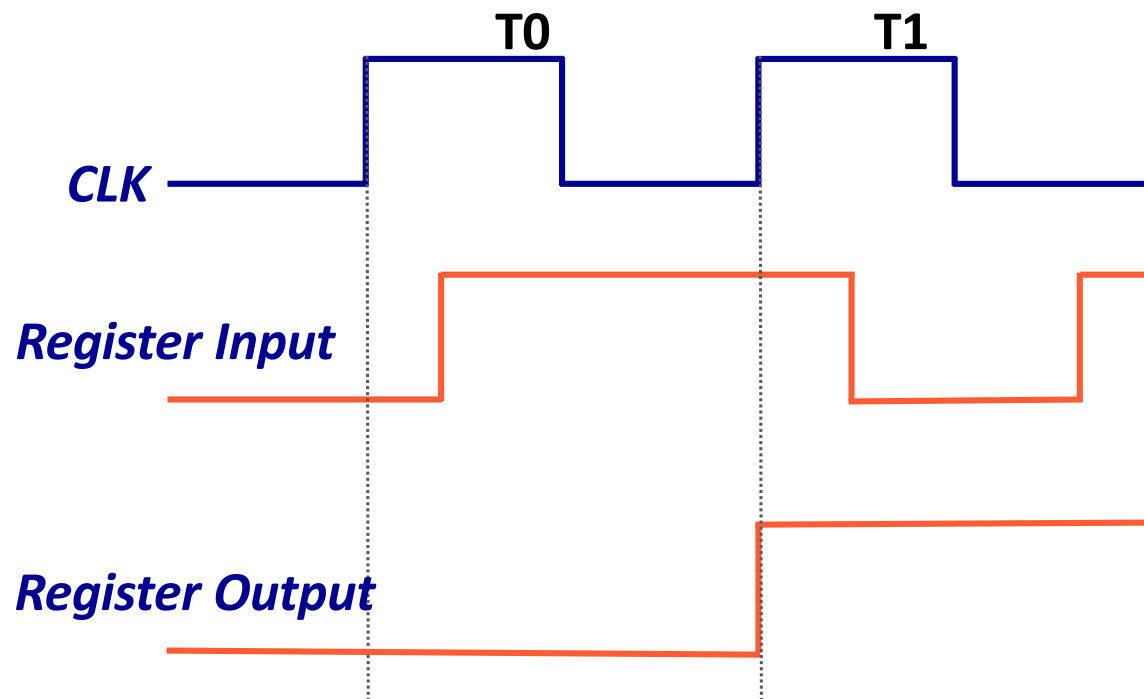
State Register

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



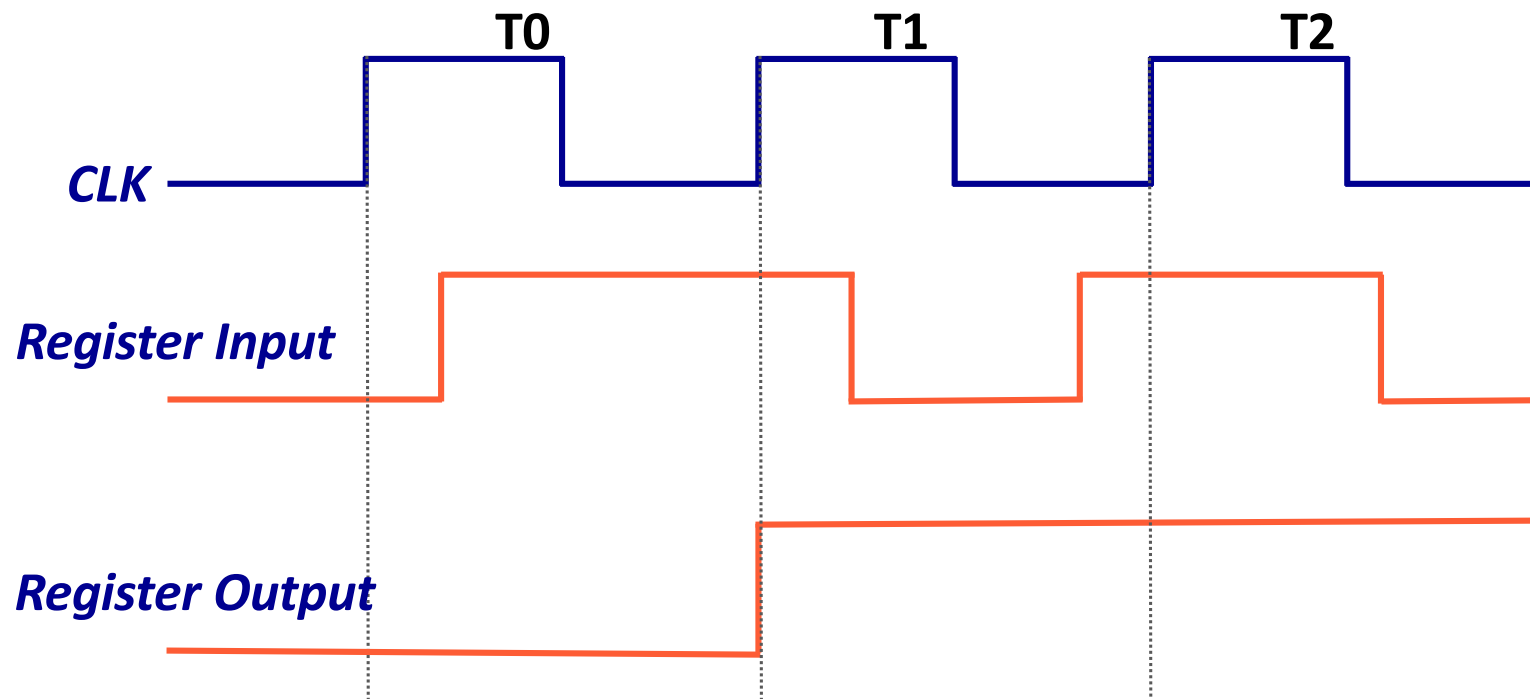
State Register

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



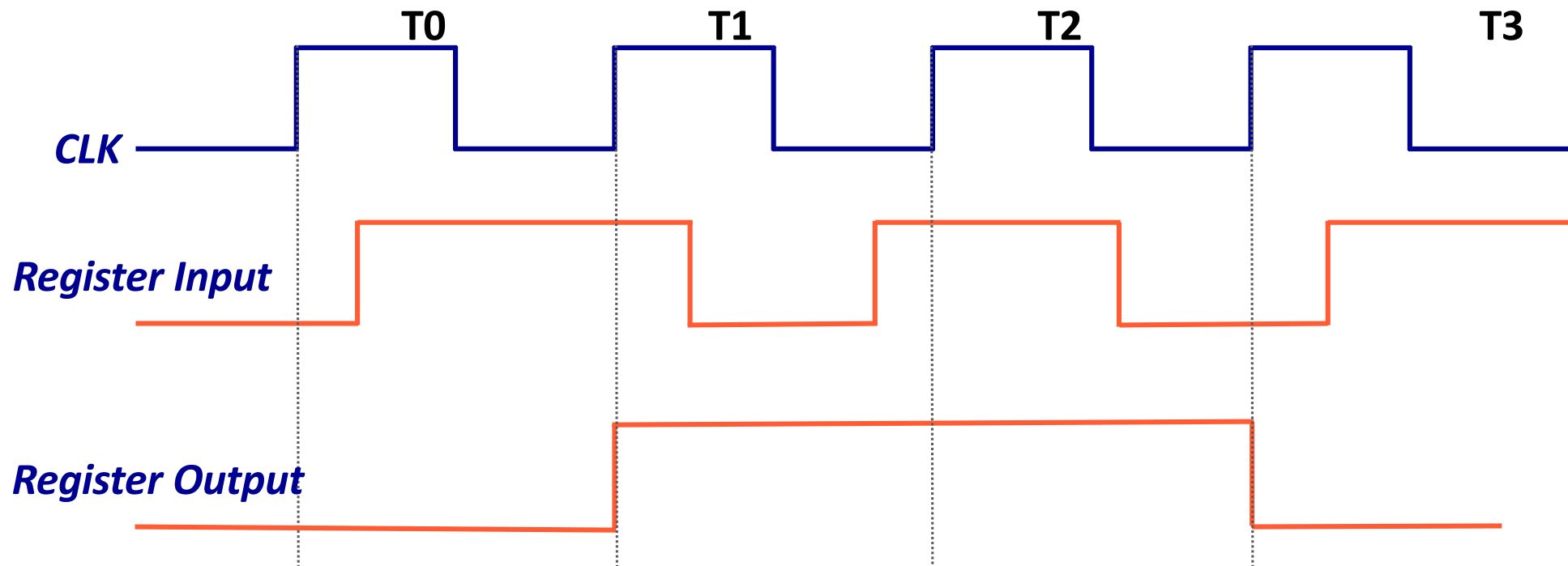
State Register

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



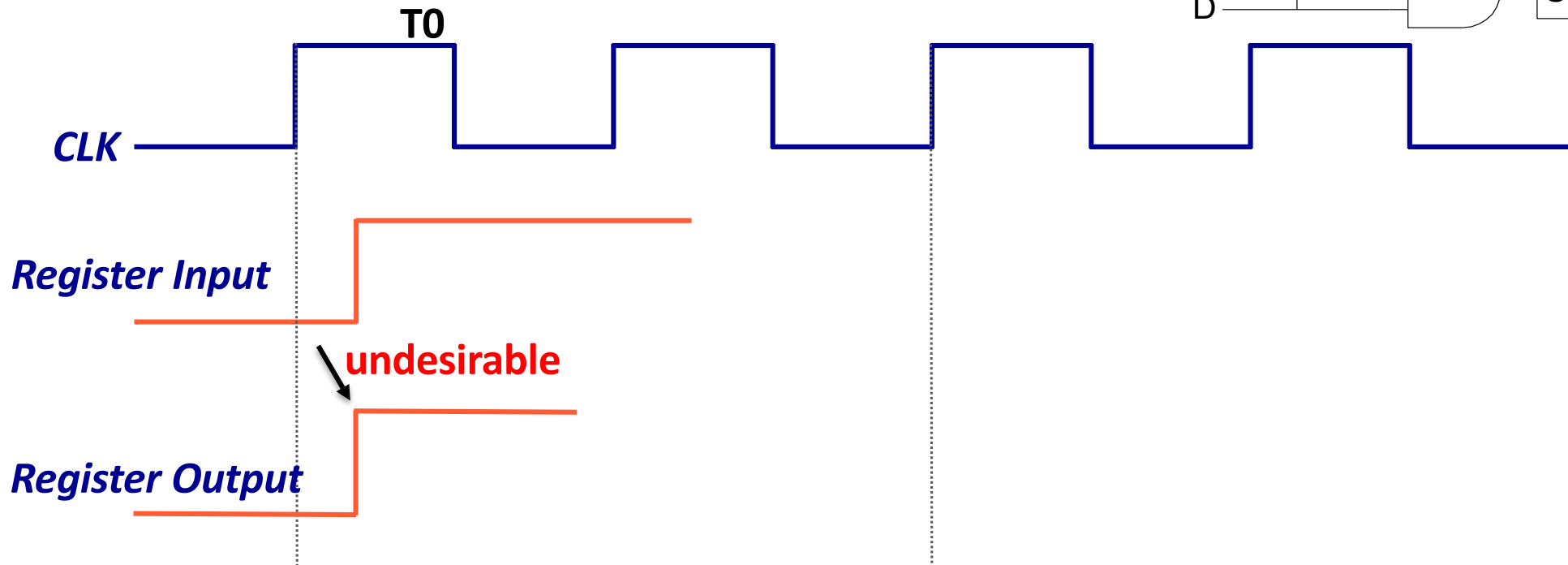
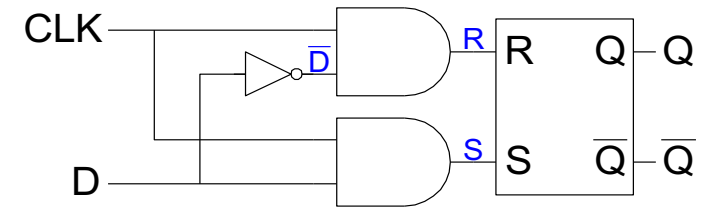
State Register

- The data must be **available** during the **entire clock cycle**
 - So combinational elements have enough time to process input combinations



The Problem with Latches

- We cannot simply wire a clock to **CLK** input of a latch
 - Whenever the clock is **HIGH**, the latch propagates **D** to **Q**
 - **The latch is transparent**



State Register uses Flip-Flops

- D (input) is **observable** at Q (output) **only** at the **beginning of the next** clock cycle
- Q is **available for the full clock cycle**

Implementing FSMs

Traffic Light Controller

The Next Example is from H & H: Section 3.4

Acknowledgement: *Selection of Slides from Digital Design and Computer Architecture, Onur Mutlu, ETH Zurich, Spring 2022*

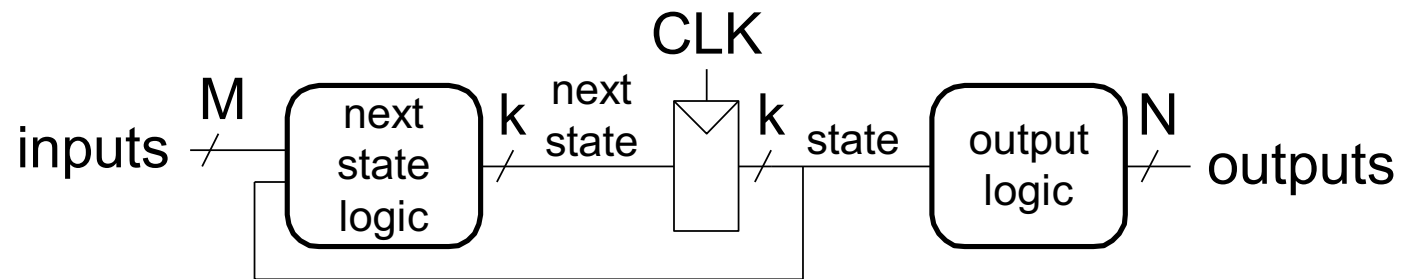
<https://safari.ethz.ch/digitaltechnik/spring2022/doku.php?id=schedule>

Finite State Machines (FSMs)

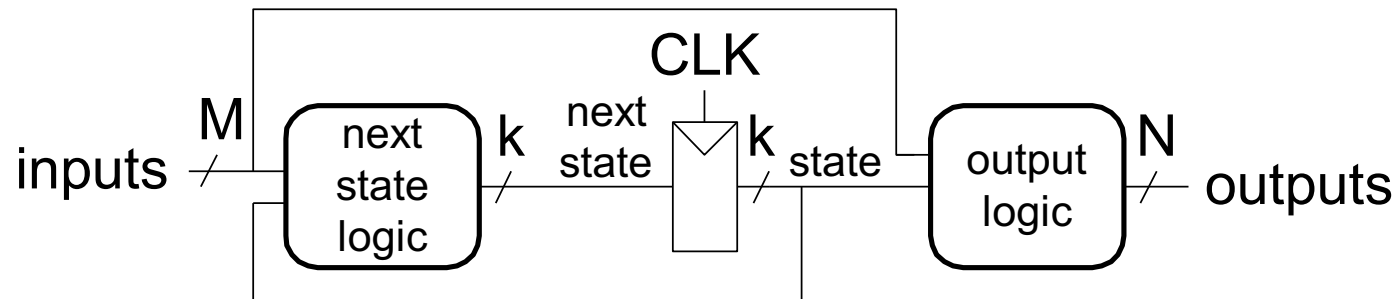
- Next state is determined by the current state and the inputs
- Two types of finite state machines differ in the **output logic**:
 - **Moore FSM**: outputs depend only on the current state
 - **Mealy FSM**: outputs depend on the current state and inputs

Moore & Mealy FSM

Moore FSM

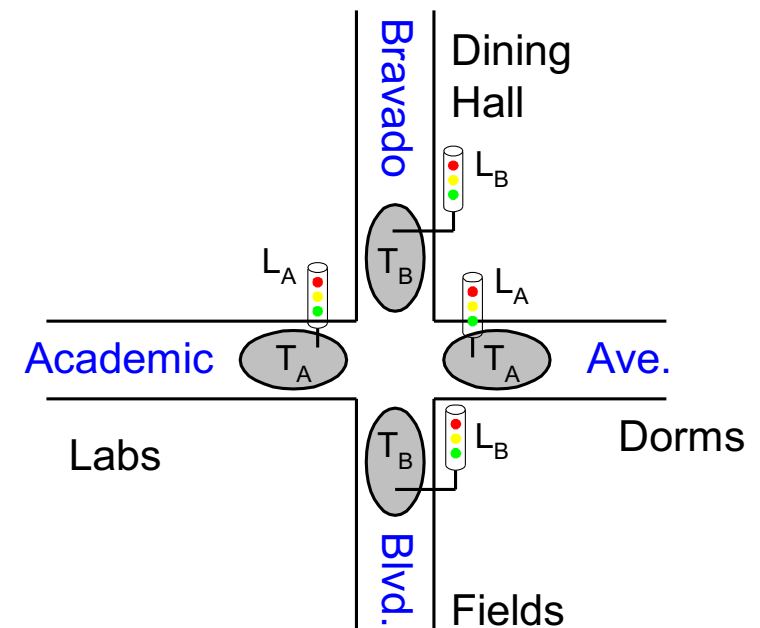


Mealy FSM



Finite State Machine Example

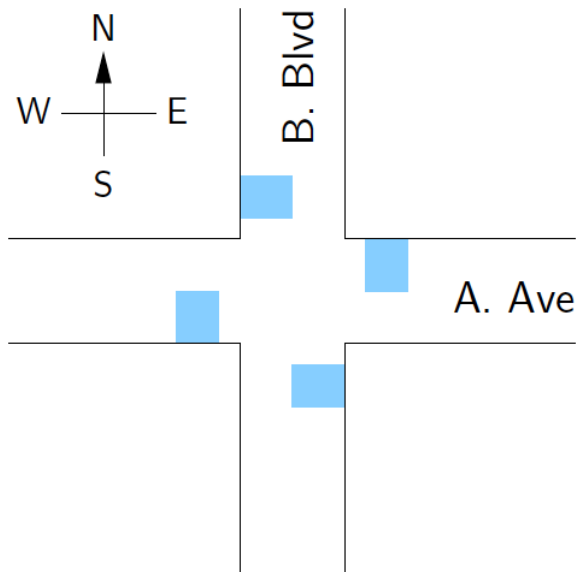
- “Smart” traffic light controller
 - **2 inputs:**
 - Traffic sensors: T_A , T_B (TRUE when there’s traffic)
 - **2 outputs:**
 - Lights: L_A , L_B (Red, Yellow, Green)
- State can change every 5 seconds
 - Except if **green** and traffic, stay **green**



From H&H Section 3.4.1

Finite State Machine Example

- Traffic sensors are built into the road
- Each sensor indicates if a street is empty or there are vehicles nearby

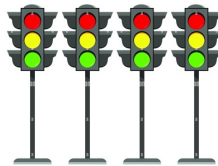


$T_A = (\text{eastbound traffic on A}) \text{ OR } (\text{westbound traffic on A})$

$T_B = (\text{northbound traffic on B}) \text{ OR } (\text{southbound traffic on B})$

Finite State Machine Example

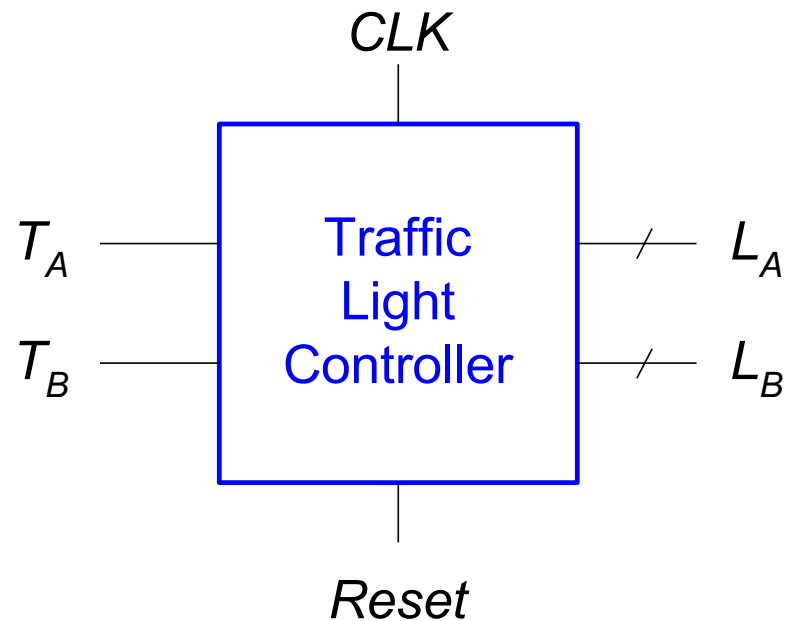
- **Inputs T_A and T_B**
 - Returns **TRUE** if there are cars on the road
 - Returns **FALSE** if the road is empty
- **Outputs $L_{A1:0}$ and $L_{B1:0}$**
 - Each set of lights receive 2-bit digital inputs from the traffic light controller specifying whether it should be: **RED**, **YELLOW**, **GREEN**



Output	Encoding
GREEN	00
YELLOW	01
RED	10

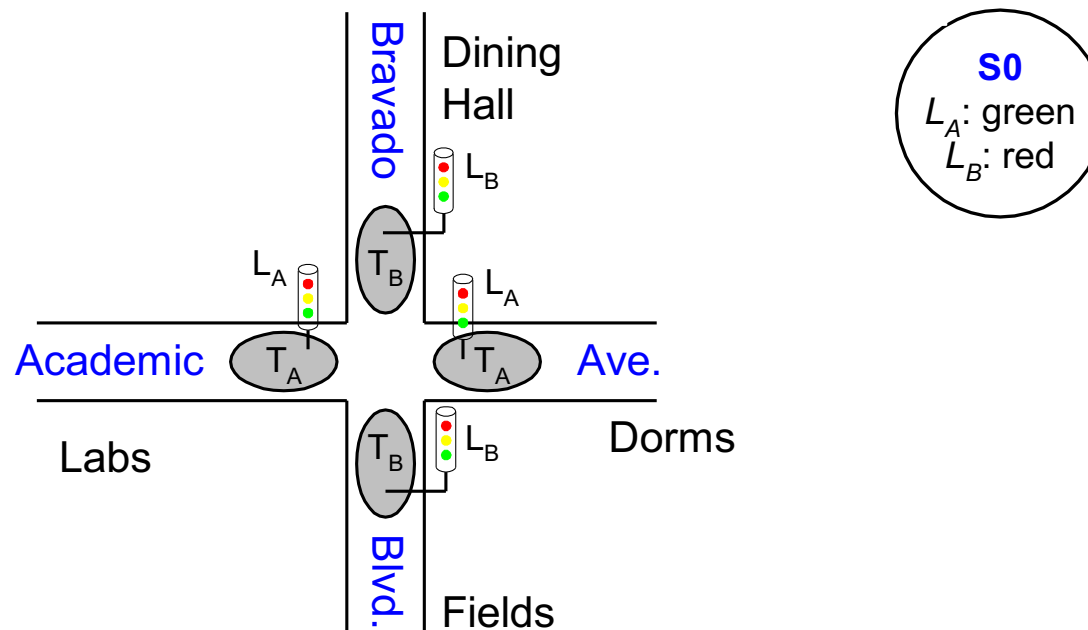
Finite State Machine Blackbox

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



Finite State Machine Diagram

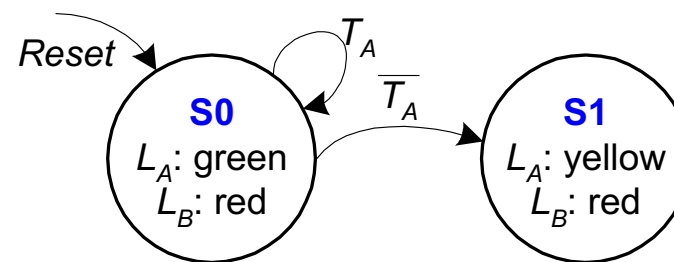
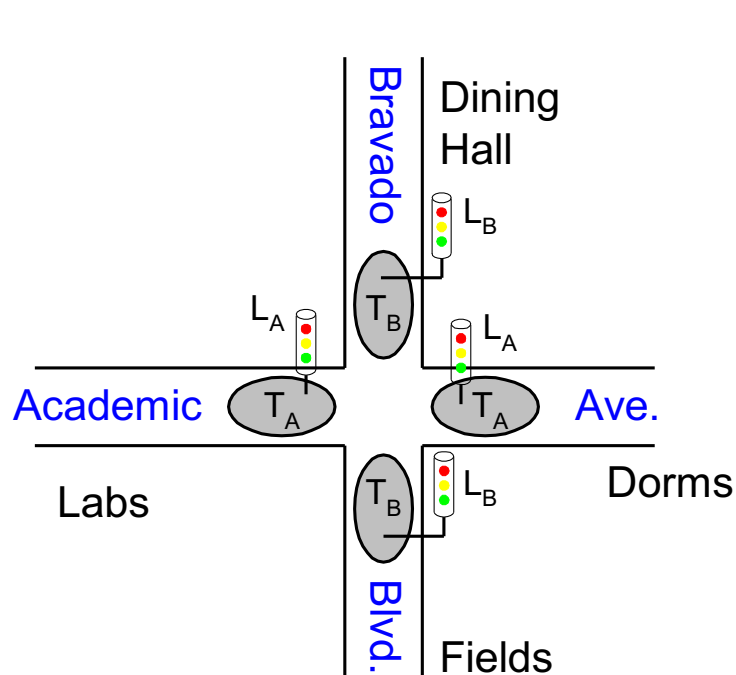
- **Moore FSM:** outputs labeled in each state
 - **States:** Circles
 - **Transitions:** Arrows (Arcs)



Finite State Machine Diagram

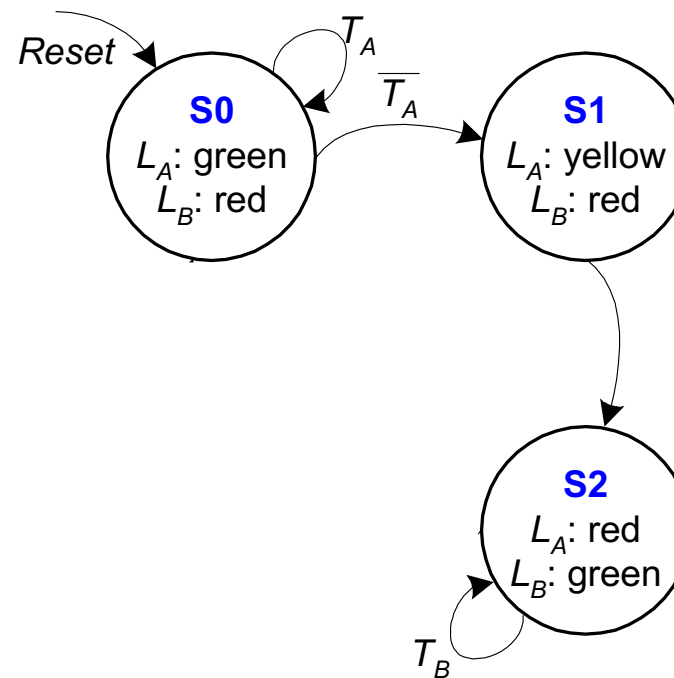
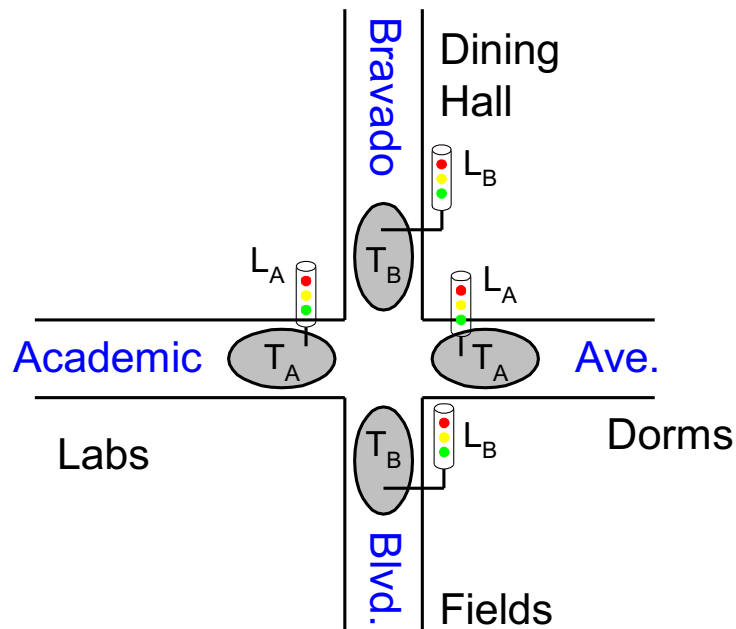
- Moore FSM: outputs labeled in each state
 - States: Circles
 - Transitions: Arrows (Arcs)

→ From “current state” **S0** to “next state” **S1**



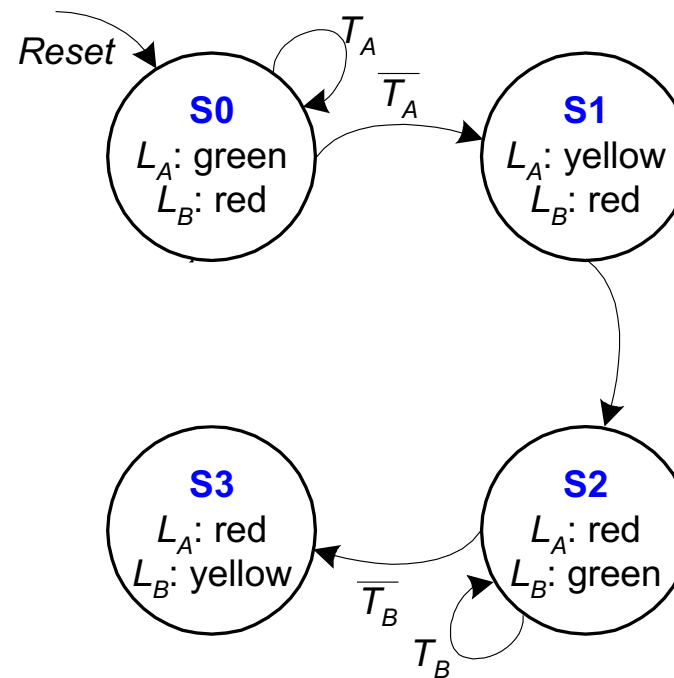
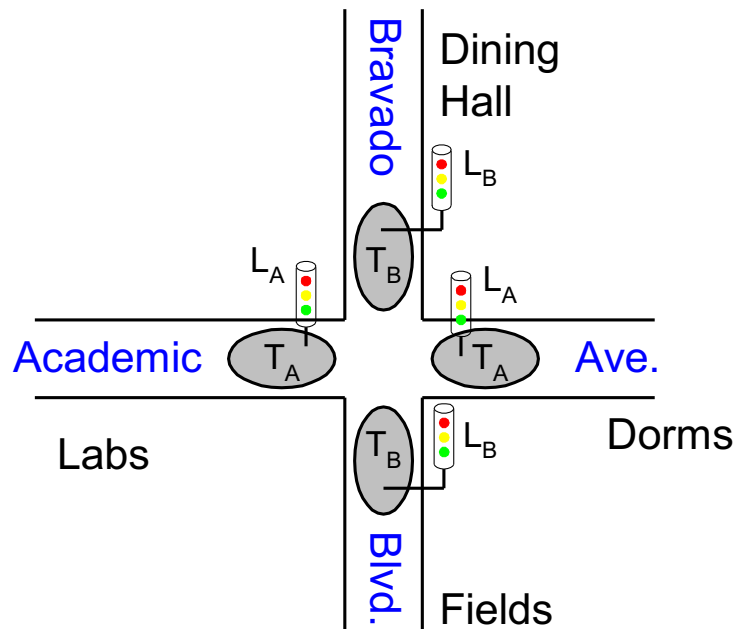
Finite State Machine Diagram

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



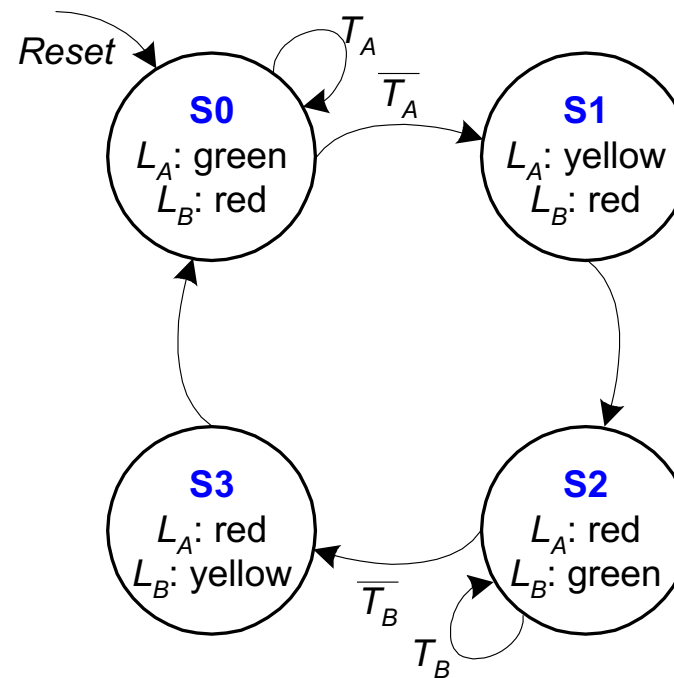
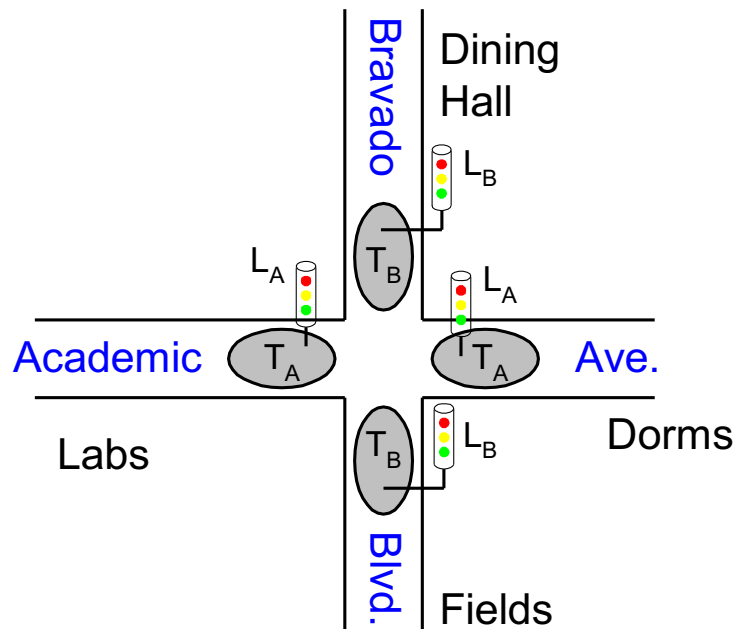
Finite State Machine Diagram

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



Finite State Machine Diagram

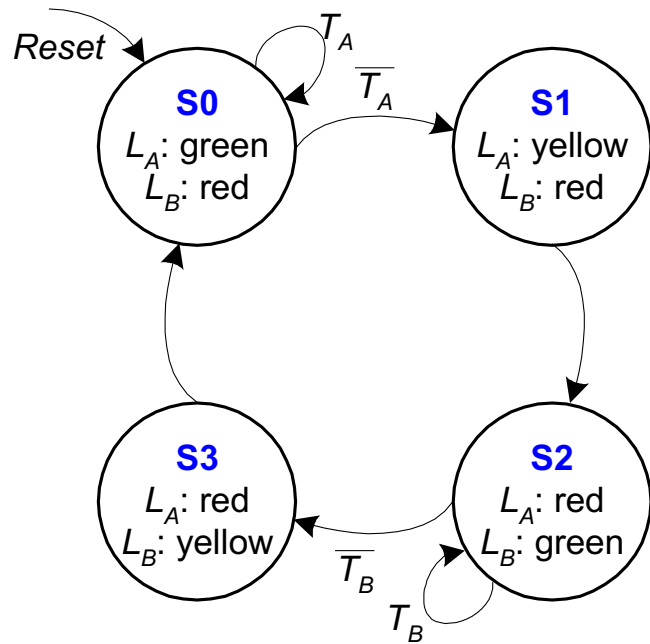
- Moore FSM: outputs labeled in each state
 - States: Circles
 - Transitions: Arrows (Arcs)



Finite State Machine:

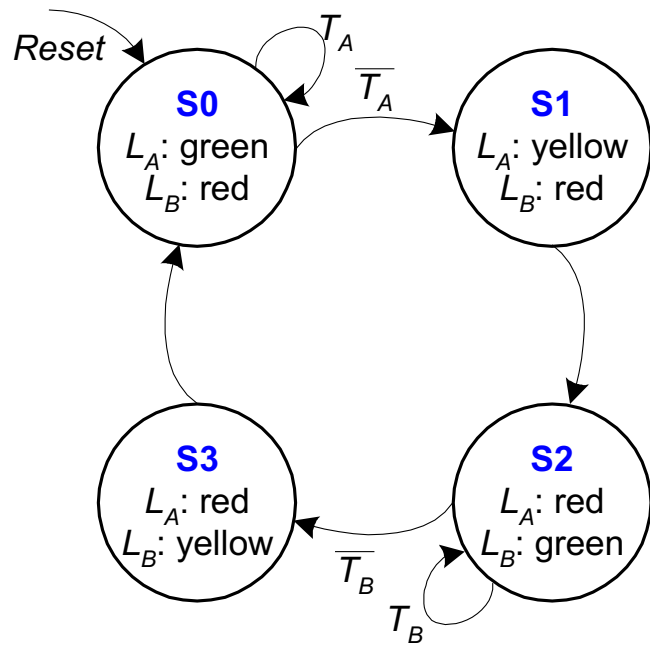
State Transition Table

FSM State Transition Table



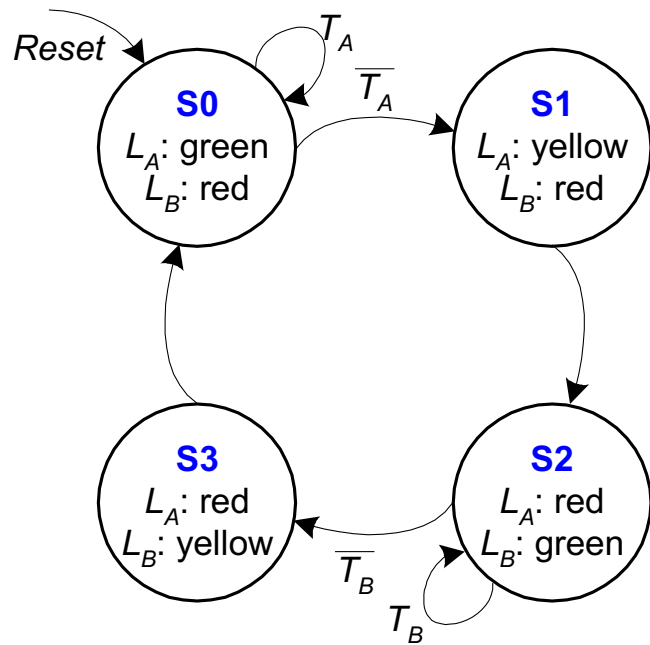
Current State	Inputs		Next State
	T_A	T_B	
S	T_A	T_B	S'
S0	0	X	
S0	1	X	
S1	X	X	
S2	X	0	
S2	X	1	
S3	X	X	

FSM State Transition Table



Current State	Inputs		Next State
	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

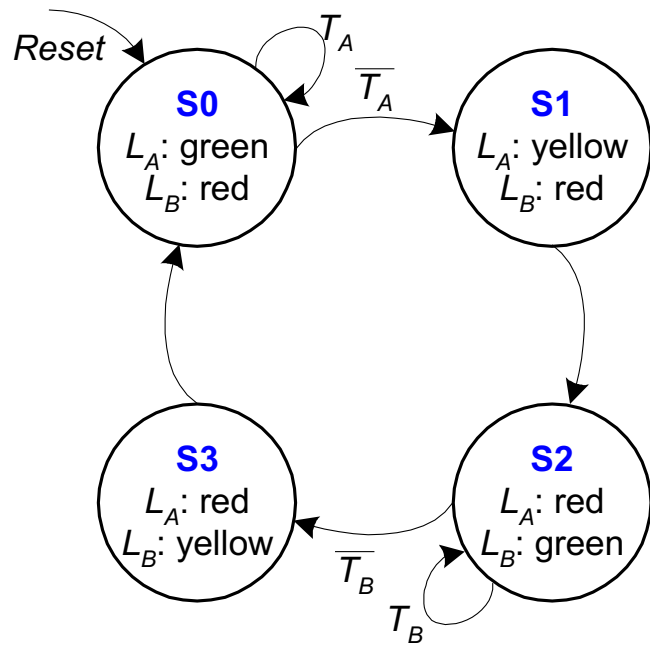
FSM State Transition Table



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

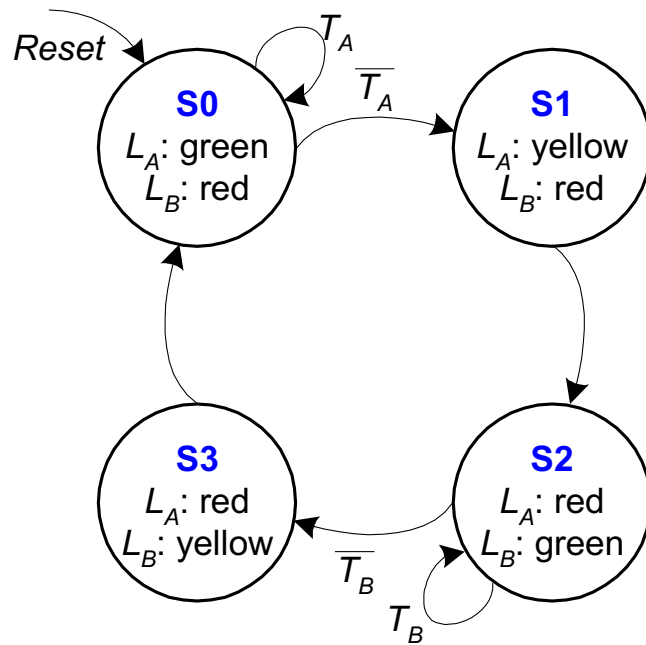
FSM State Transition Table



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

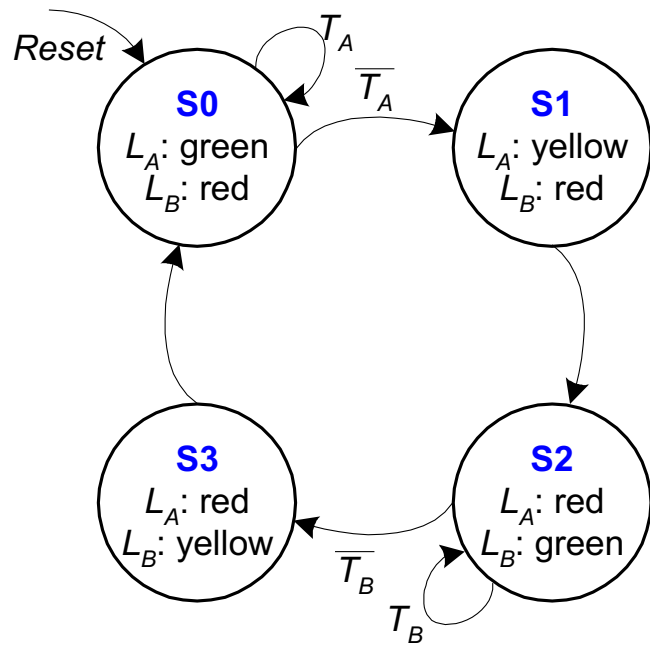


Current State		Inputs		Next State	
S ₁	S ₀	T _A	T _B	S' ₁	S' ₀
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

S'₁ = ?

FSM State Transition Table

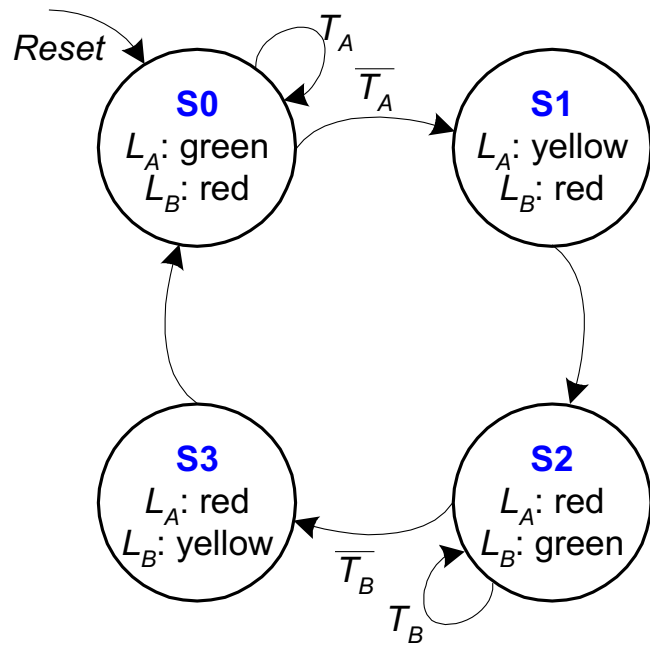


Current State		Inputs		Next State	
S ₁	S ₀	T _A	T _B	S' ₁	S' ₀
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

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

FSM State Transition Table



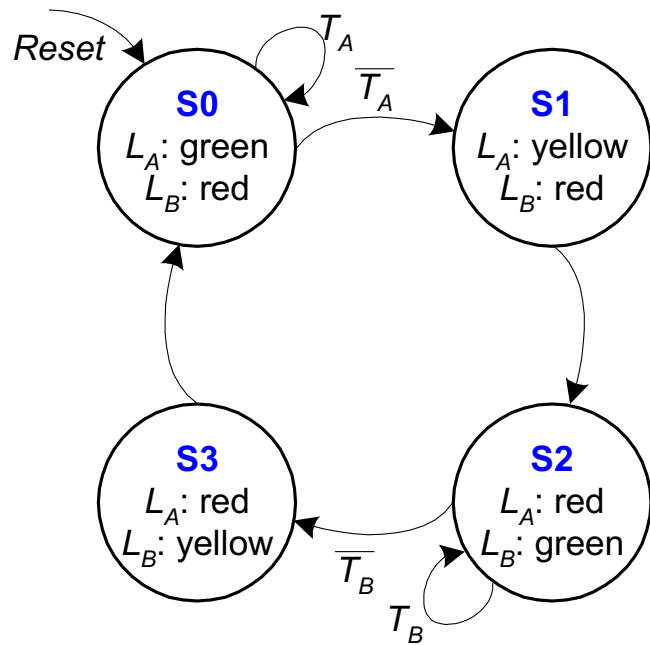
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

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



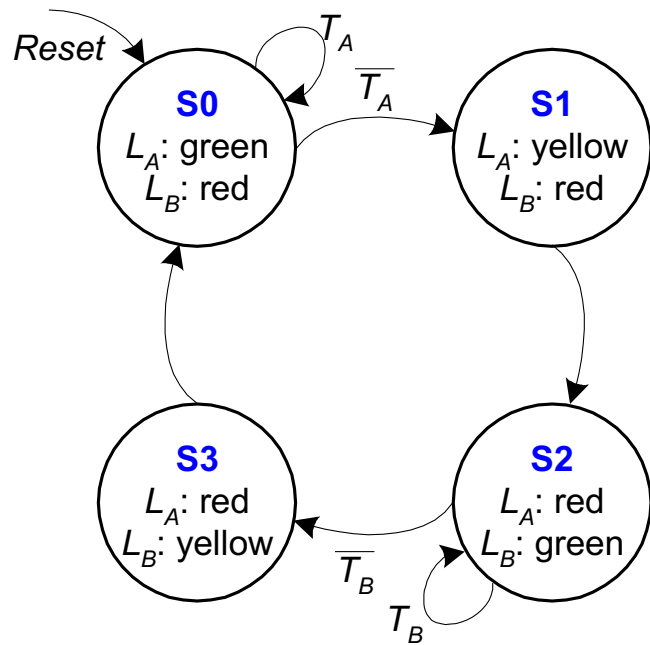
Current State		Inputs		Next State	
S ₁	S ₀	T _A	T _B	S' ₁	S' ₀
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

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

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

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

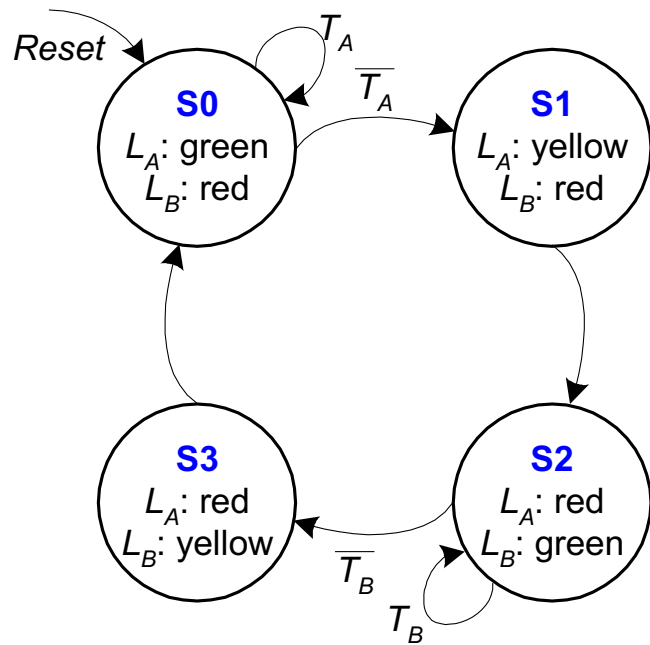
$$S'_1 = S_1 \text{ xor } S_0 \quad \text{(Simplified)}$$

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

Finite State Machine:

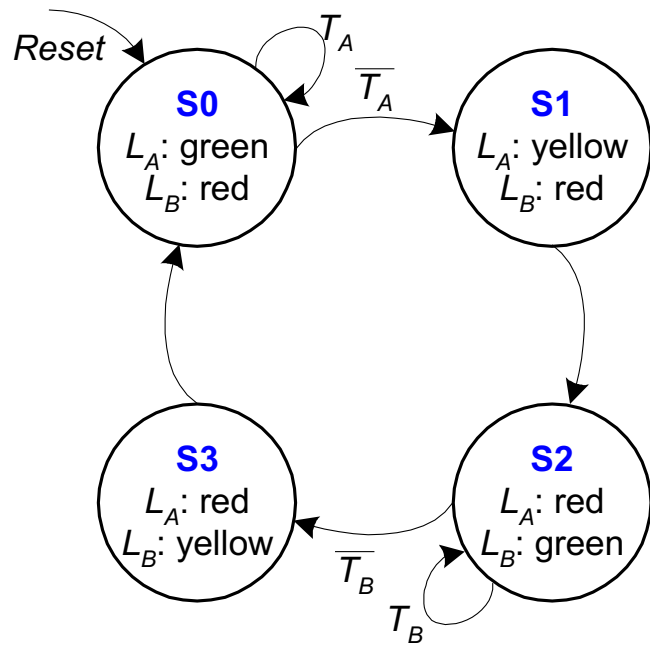
Output Table

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

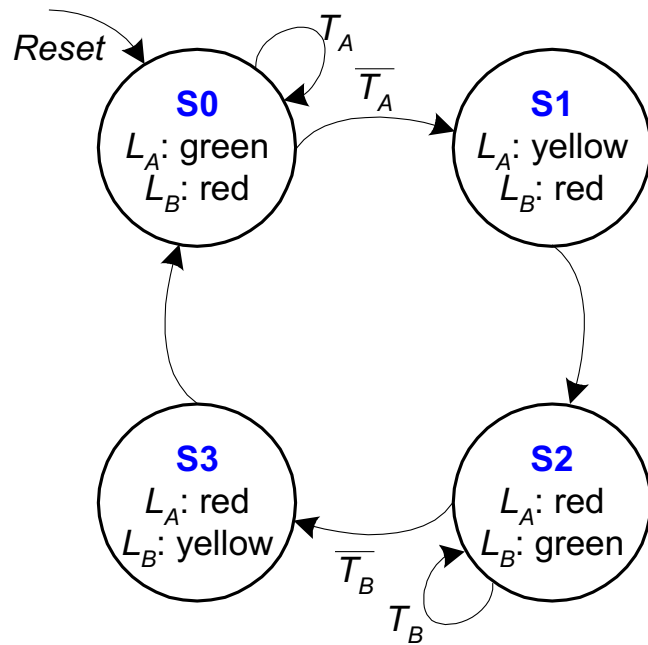
FSM Output Table



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

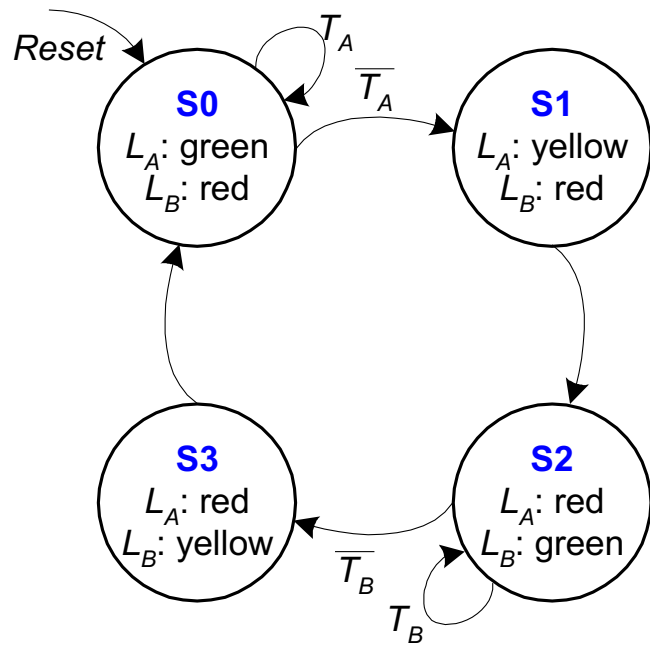


$$L_{A1} = S_1$$

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



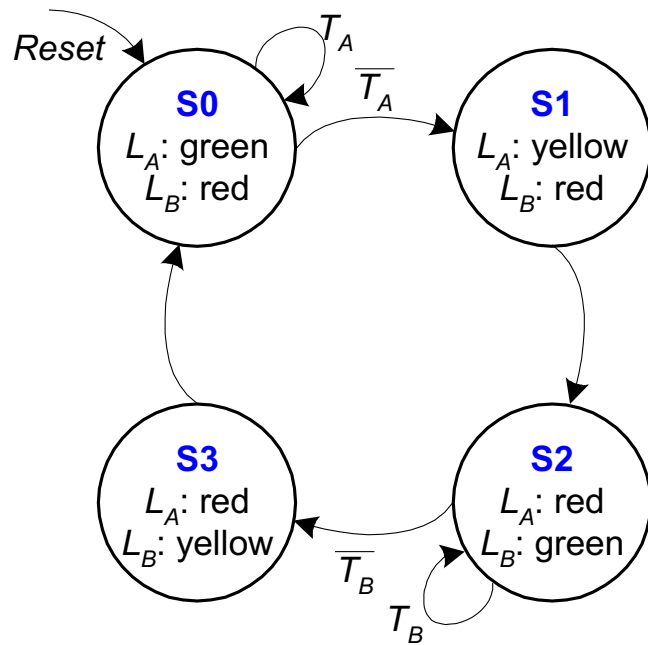
$$L_{A1} = \overline{S_1}$$

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

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



$$L_{A1} = \overline{S_1}$$

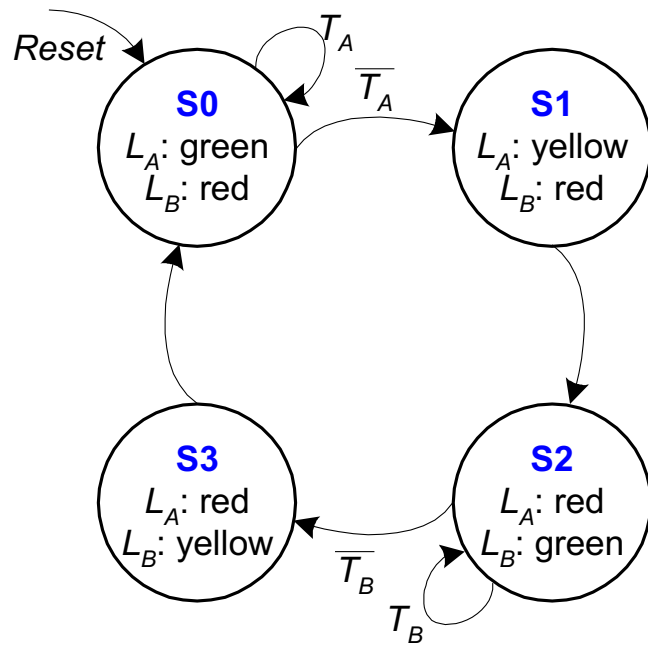
$$L_{A0} = \overline{S_1} \cdot S_0$$

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

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



$$L_{A1} = \overline{S_1}$$

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

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

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

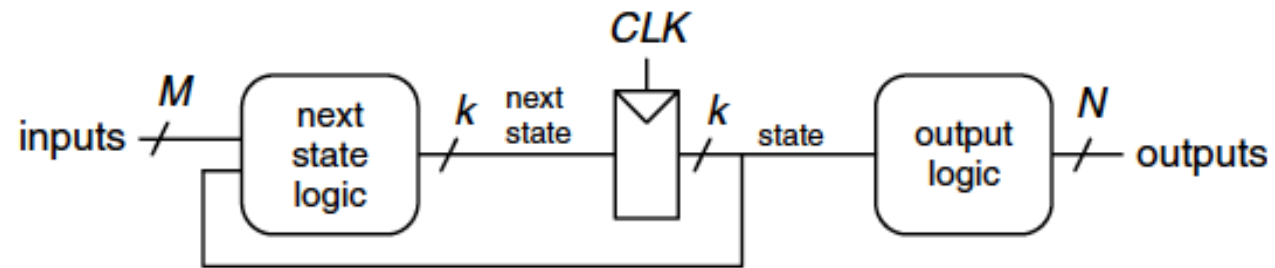
Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

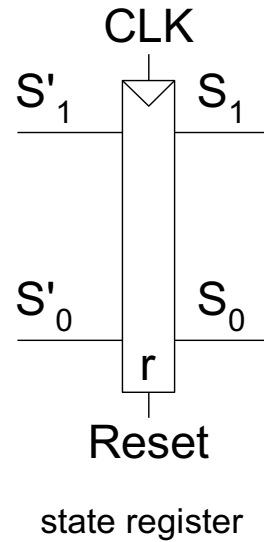
Finite State Machine:

Schematic

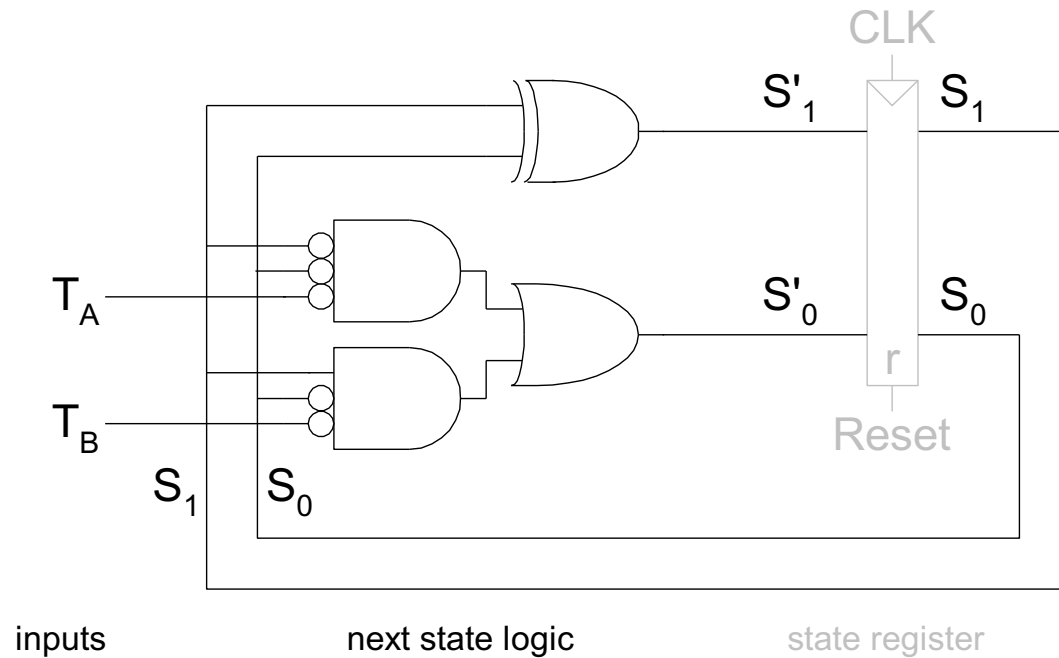
FSM Overview



FSM Schematic: State Register



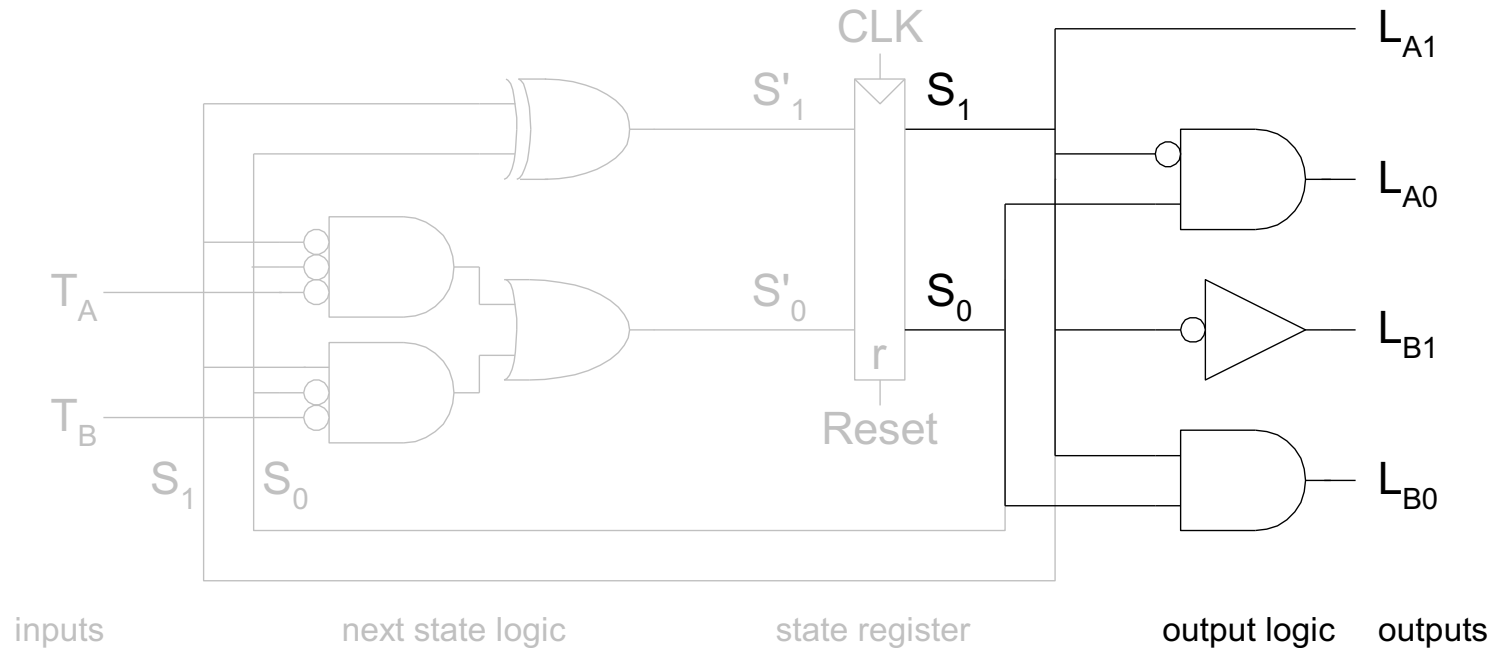
FSM Schematic: Next State Logic



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

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

FSM Schematic: Output Logic



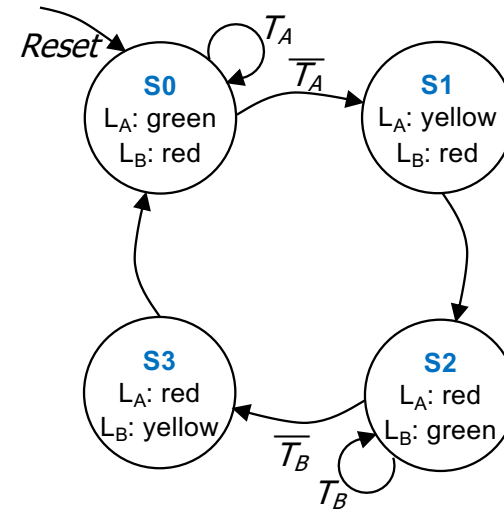
$$L_{A1} = S_1$$

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

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

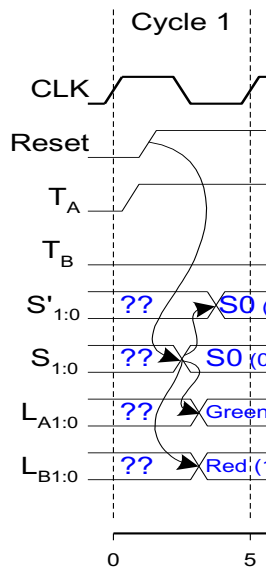
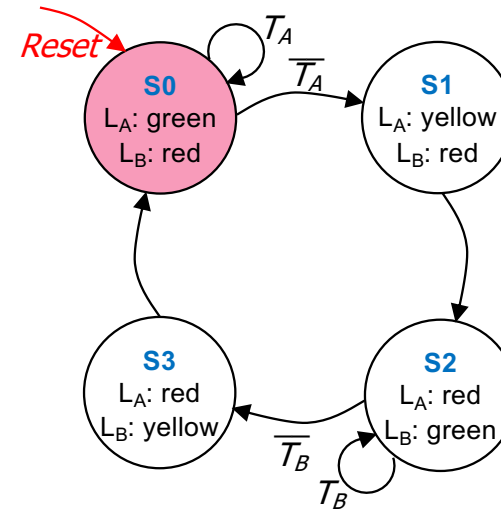
$$L_{B0} = S_1 \cdot S_0$$

FSM Timing Diagram

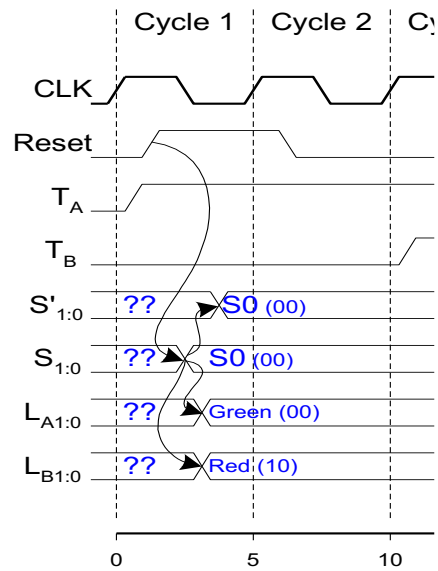
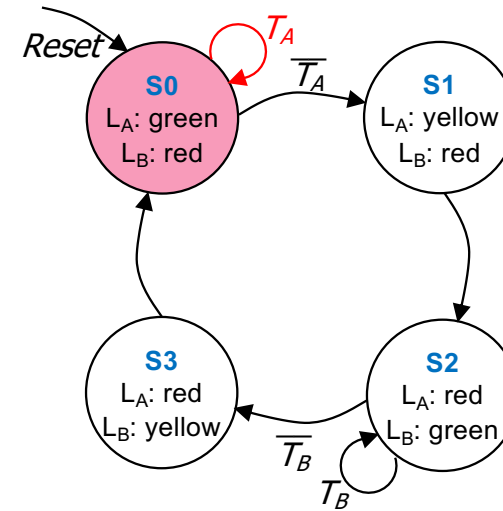


CLK_
Reset_
T_A_
T_B_
S'_{1:0}_
S_{1:0}_
L_{A1:0}_
L_{B1:0}_

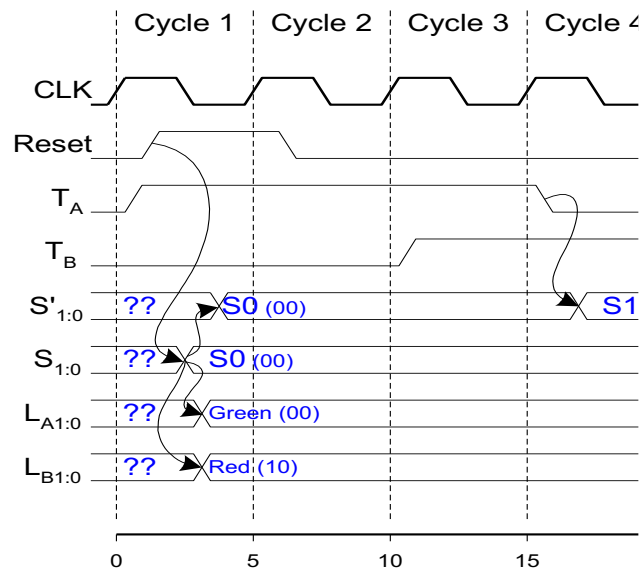
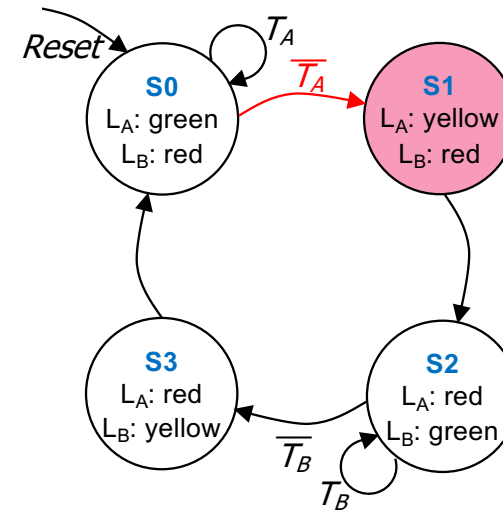
FSM Timing Diagram



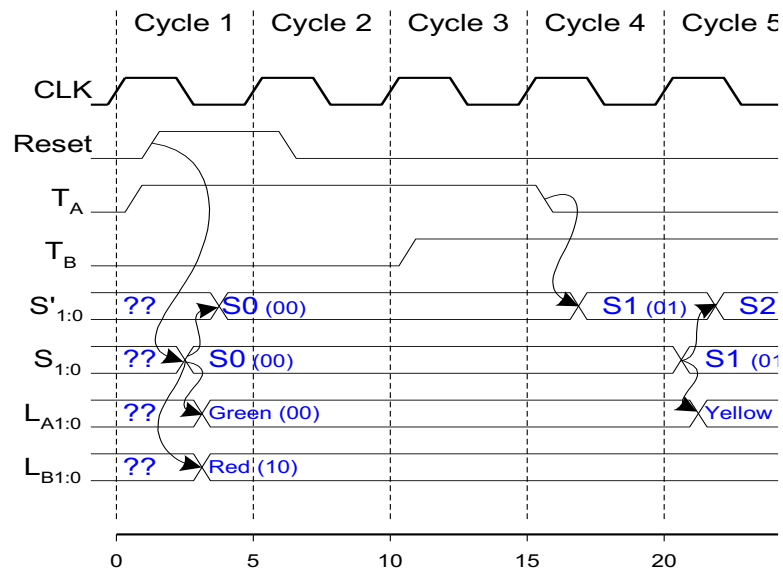
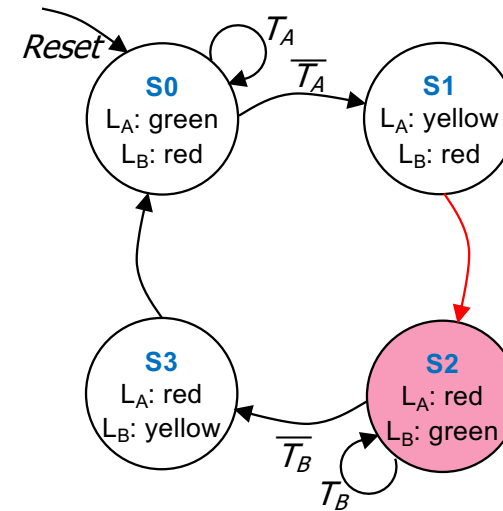
FSM Timing Diagram



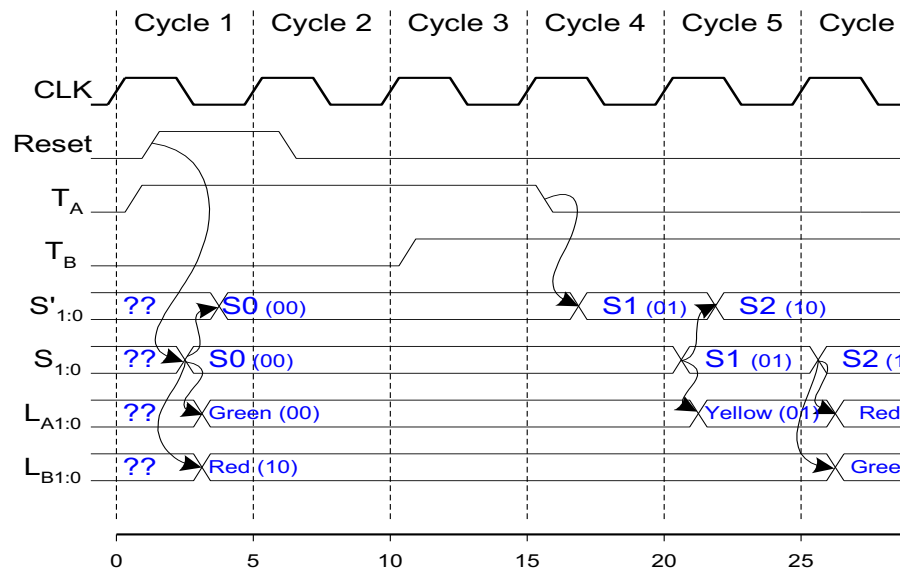
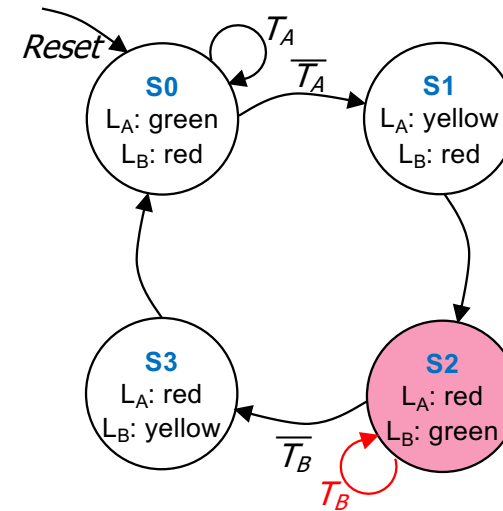
FSM Timing Diagram



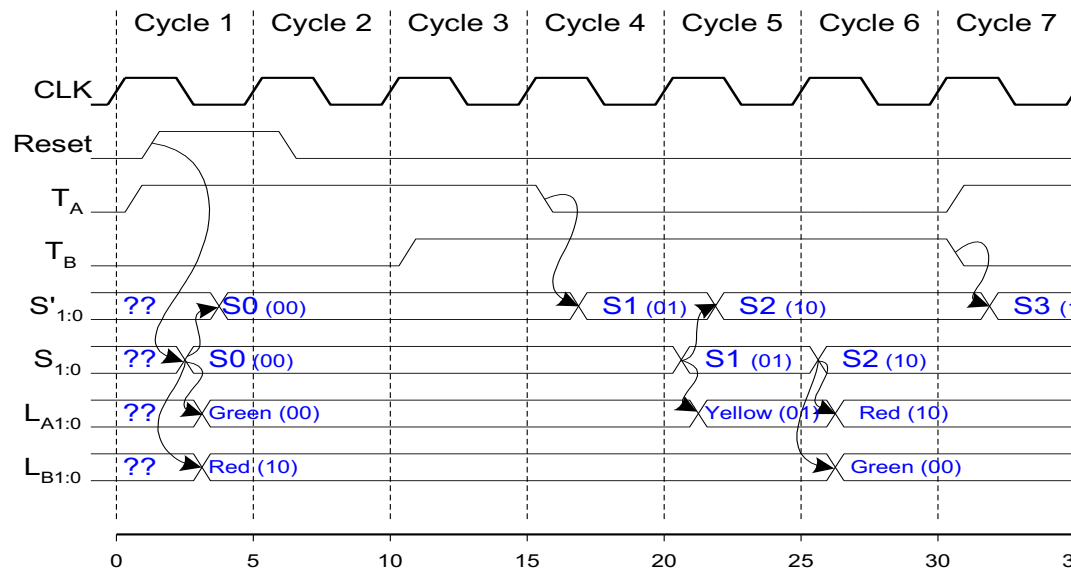
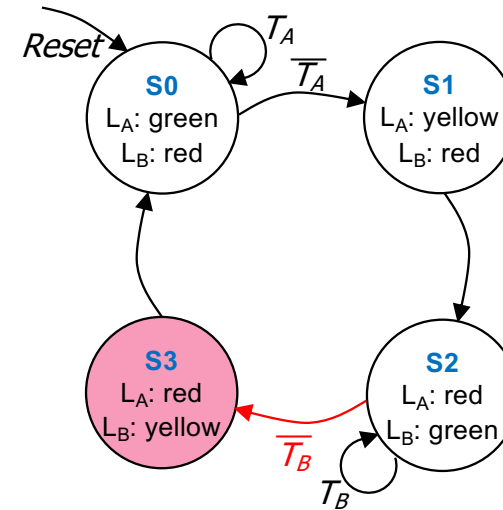
FSM Timing Diagram



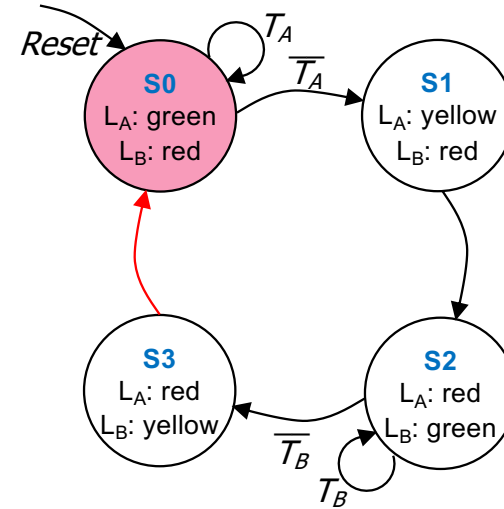
FSM Timing Diagram



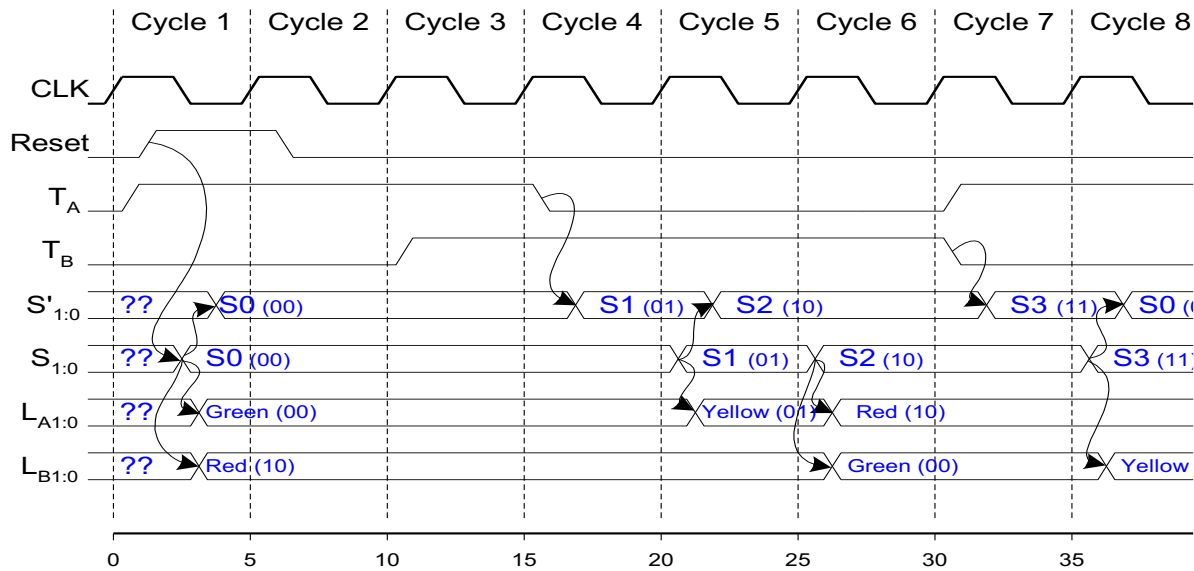
FSM Timing Diagram



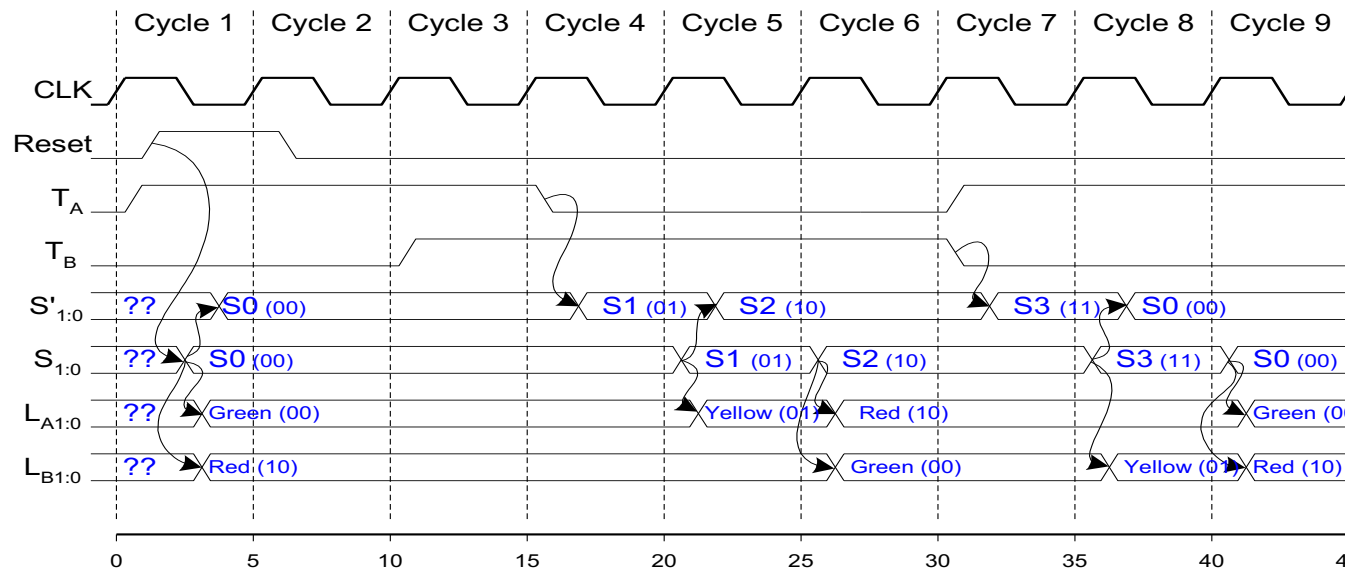
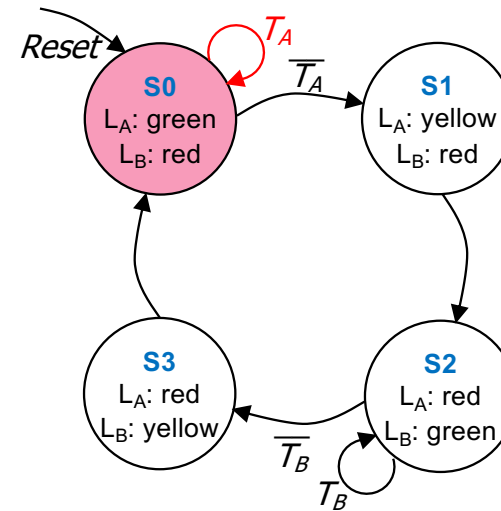
FSM Timing Diagram



This is from H&H Section 3.4.1

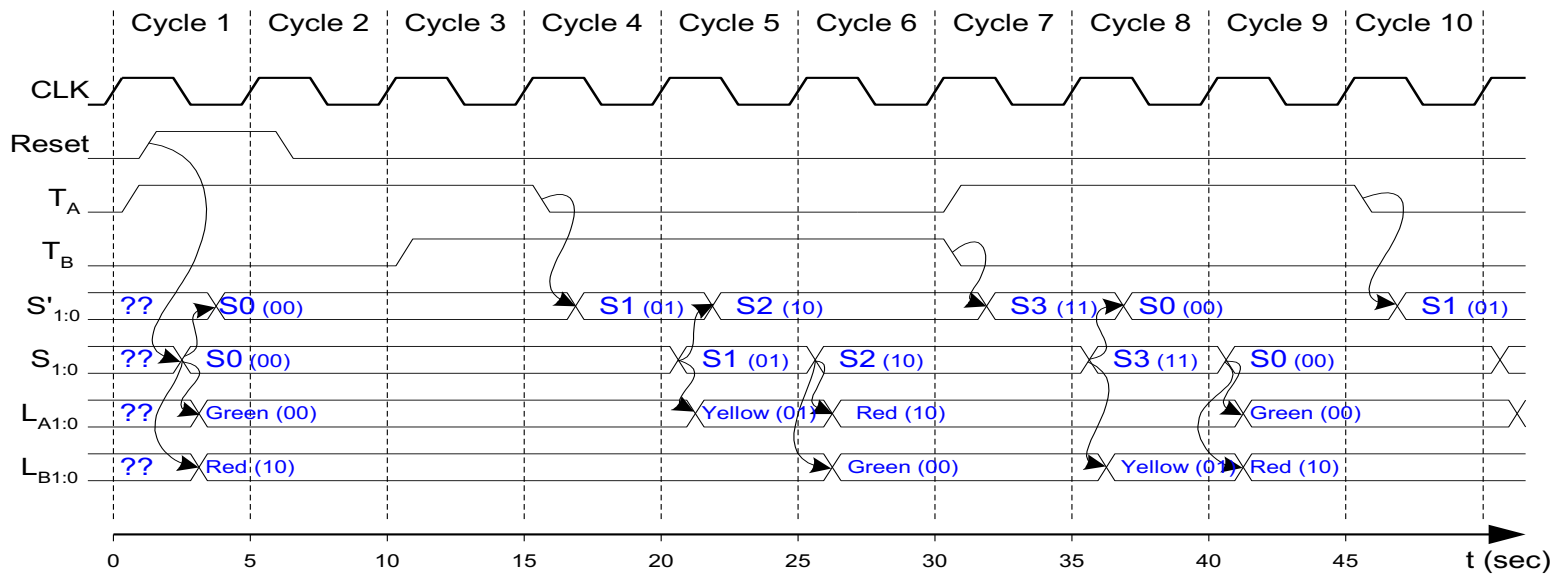
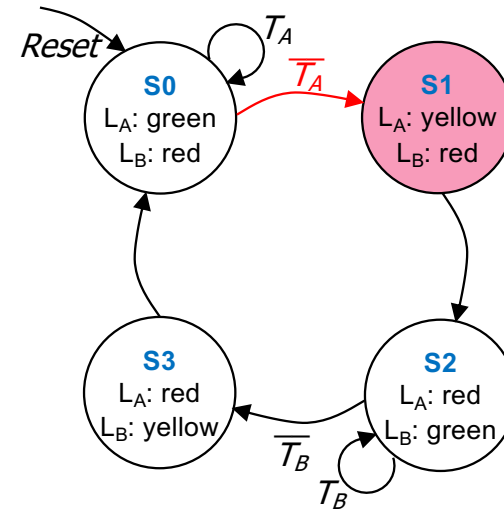


FSM Timing Diagram



FSM Timing Diagram

See H&H Chapter 3.4

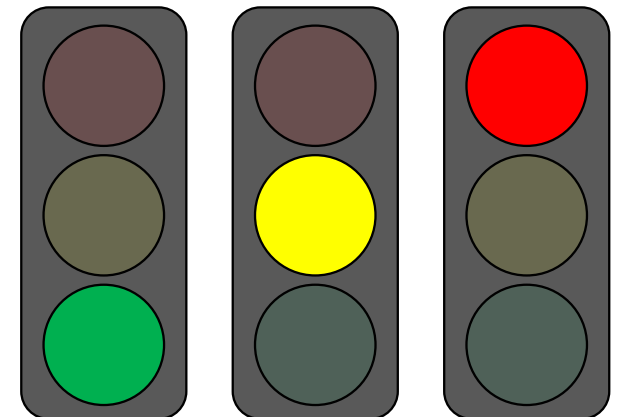


Finite State Machine:

State Encoding

FSM State Encoding

- How do we encode the state bits?
 - Three common state binary encodings with different tradeoffs
 - **Fully Encoded**
 - **1-Hot Encoded**
- Let's see an example traffic light with 3 states
 - Green, Yellow, Red



FSM State Encoding

1. Binary Encoding (Full Encoding):

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

2. One-Hot Encoding:

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

FSM State Encoding

1. Binary Encoding (Full Encoding):

- Use the minimum possible number of bits

- Use

- *Example*

- **Minim**

2. One-H

- Each b

- Use

- Exac

- *Example state encodings: 0001, 0010, 0100, 1000*

- **Simplest design process** – very automatable

- **Maximizes** # flip-flops, **minimizes** next state logic

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

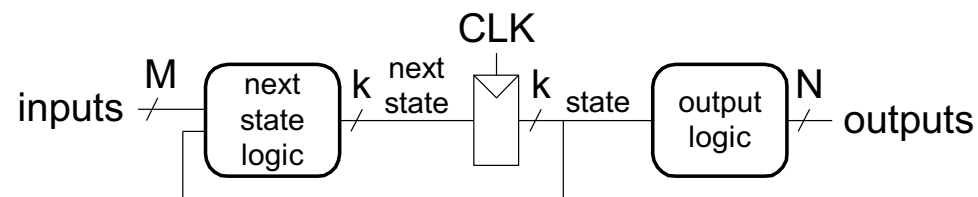
logic

Moore vs. Mealy Machines

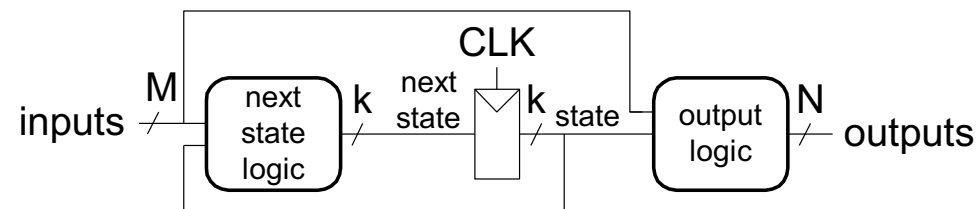
Recall: Moore vs. Mealy Machines

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

Moore FSM



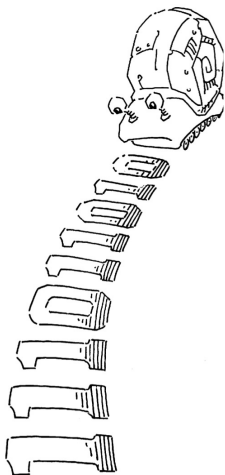
Mealy FSM



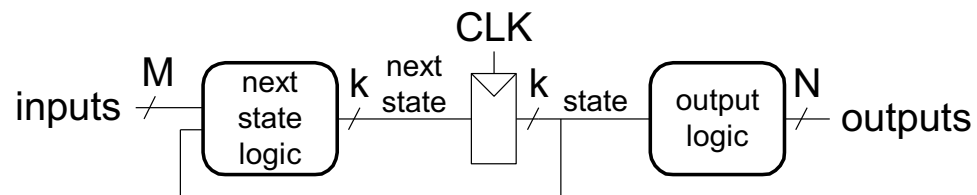
Section 3.4.3 of H&H

Moore vs. Mealy Examples

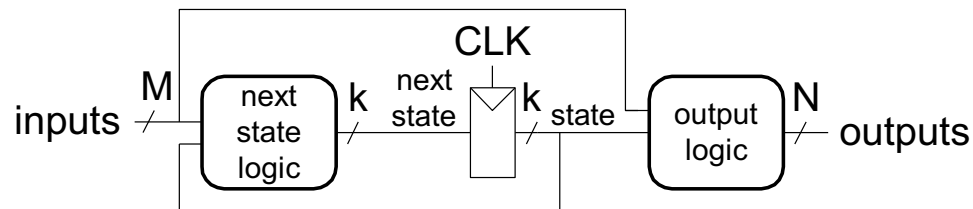
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are **1101**.
- Design Moore and Mealy FSMs of the snail's brain.



Moore FSM

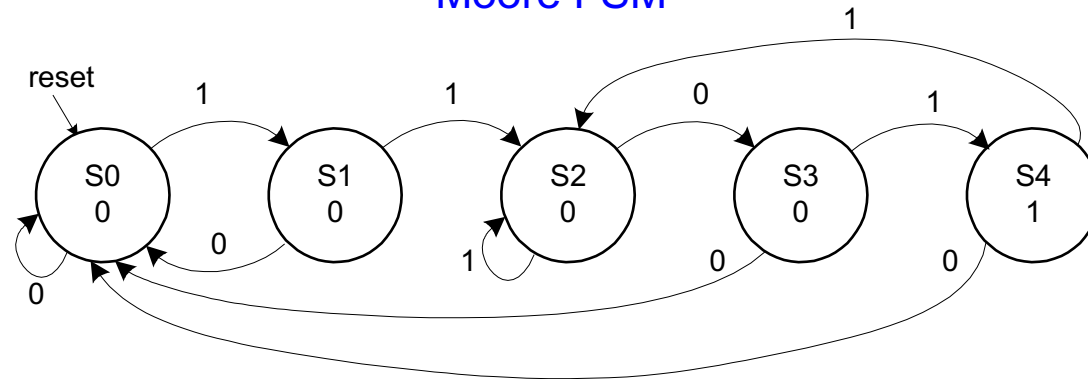


Mealy FSM



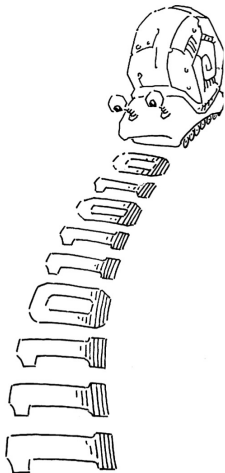
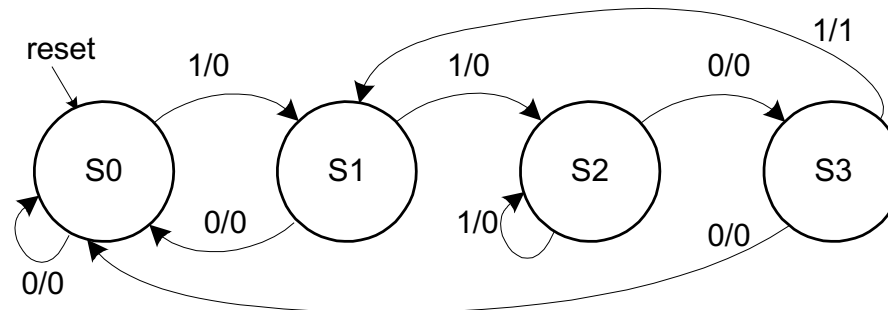
State Transition Diagrams

Moore FSM



What are the tradeoffs?

Mealy FSM



FSM Design Procedure (I)

Step # 1: State transition diagram

- Formalize the specification and remove ambiguity

Step # 2: Derive the next state logic

- Binary encoding for states
- State transition (truth) table
- Minimized Boolean equations for next state logic

Step # 3: Derive the output logic

- Binary encoding for outputs
- Output table & Boolean equations

Step # 4: Turn the Boolean equations into logic gate implementation

- Next state logic & output logic

FSM Design Procedure (II)

- **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**

Sync. Seq. Circuits: What We Covered Until Now

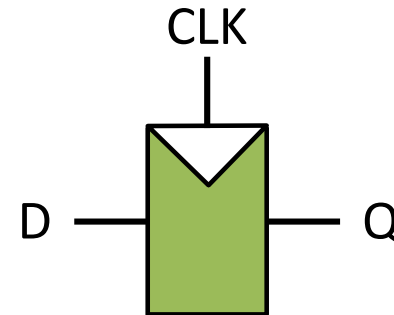
- The concept of state
- State diagrams
- Asynchronous vs. synchronous state changes
- Synchronous sequential circuits
- FSMs
 - Components, transition diagram, tables, equations, schematic
 - State transition diagrams
 - State encoding
 - Moore vs. Mealy
 - Design procedure

Timing Issues in Sequential Circuits

Reading: Section 3.5 of H&H (More detailed than what we need in this course) 186

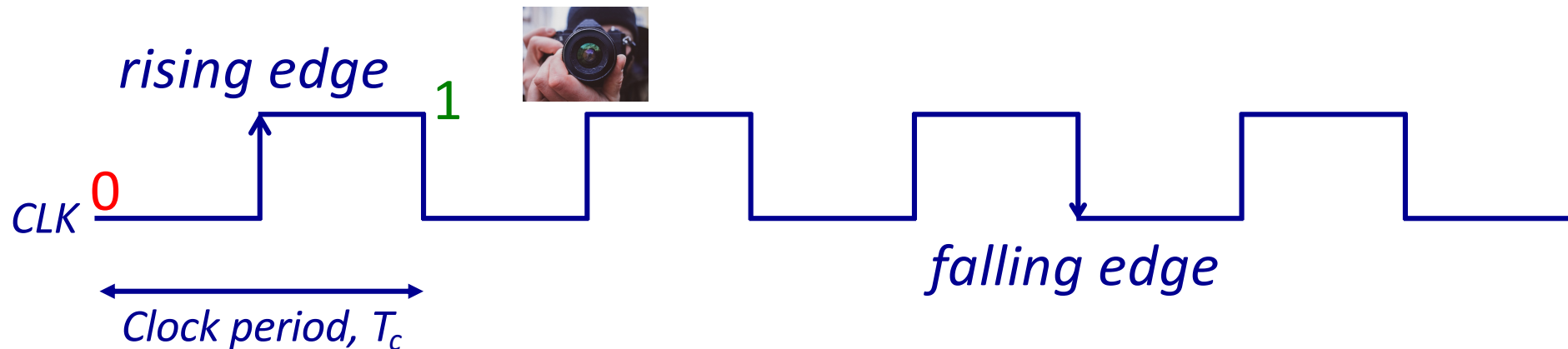
Timing in Sequential Circuits

- We need to understand three aspects of timing specification
 - Clock-to-Q propagation delay
 - Setup time
 - Hold time



Recall the Clock

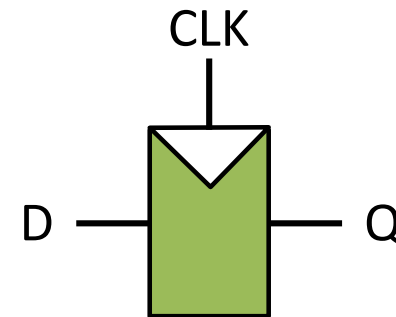
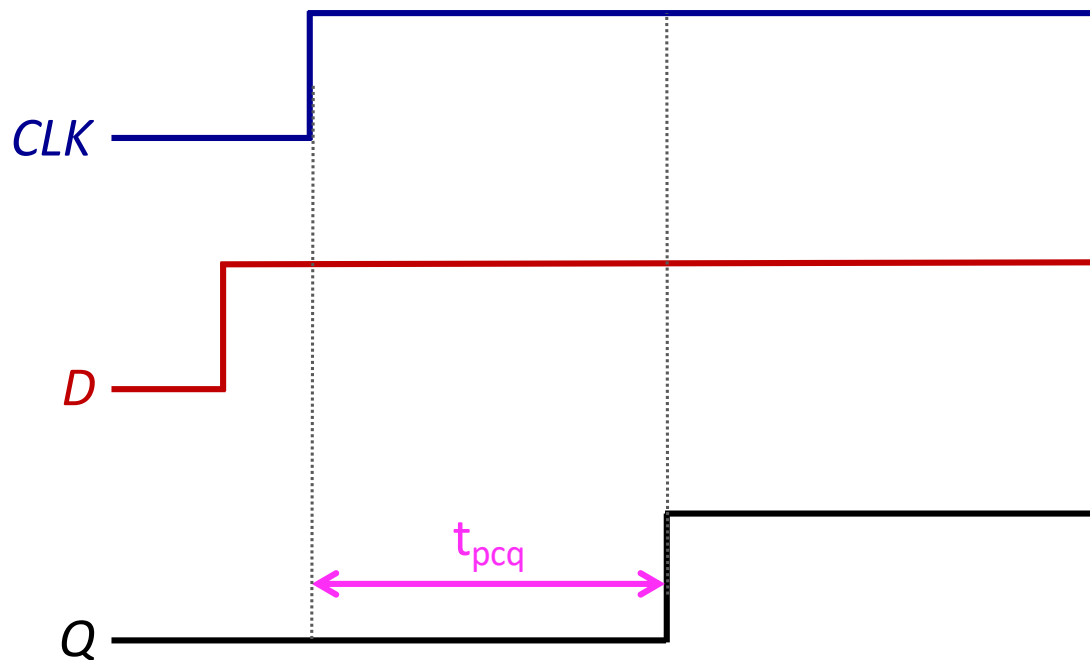
- Output does not change instantly when the **rising edge** arrives
- Input needs to stay **stable** for some time for the flip-flop to take a **reliable** photograph



$$\text{Frequency} = 1/T_c$$

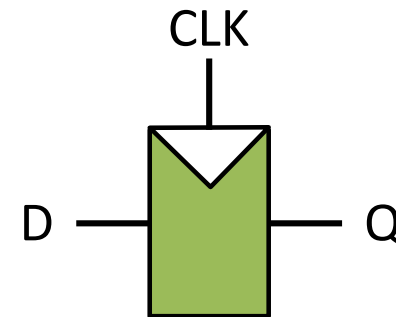
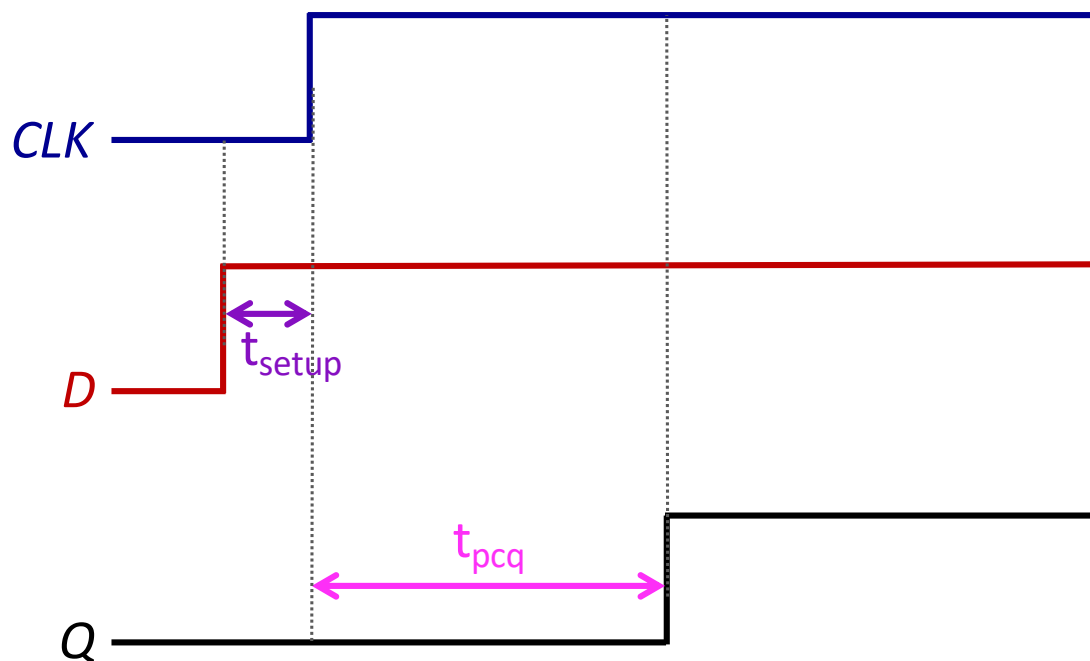
Clock-to-Q Propagation Delay

- When the clock rises, the time it takes for the output to **settle** to the final value (t_{pcq})



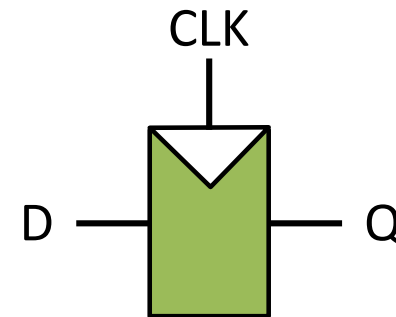
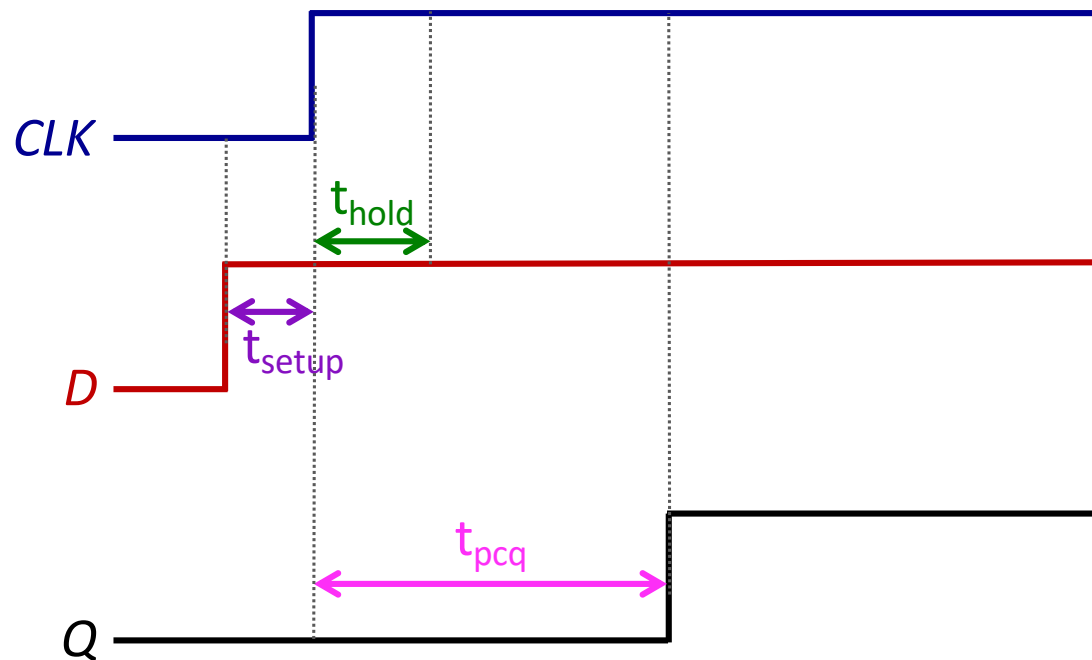
Setup Time

- For the circuit to **sample** its input correctly, the input must have **stabilized** at least some setup time, t_{setup} , before the rising edge of the clock



Hold Time

- The input must remain **stable** for at least some hold time (t_{hold}) after the rising edge of the clock



Aperture Time

- The sum of the **setup** and **hold** times is called the aperture time of the circuit
- It is the **total time for which the input must remain stable**



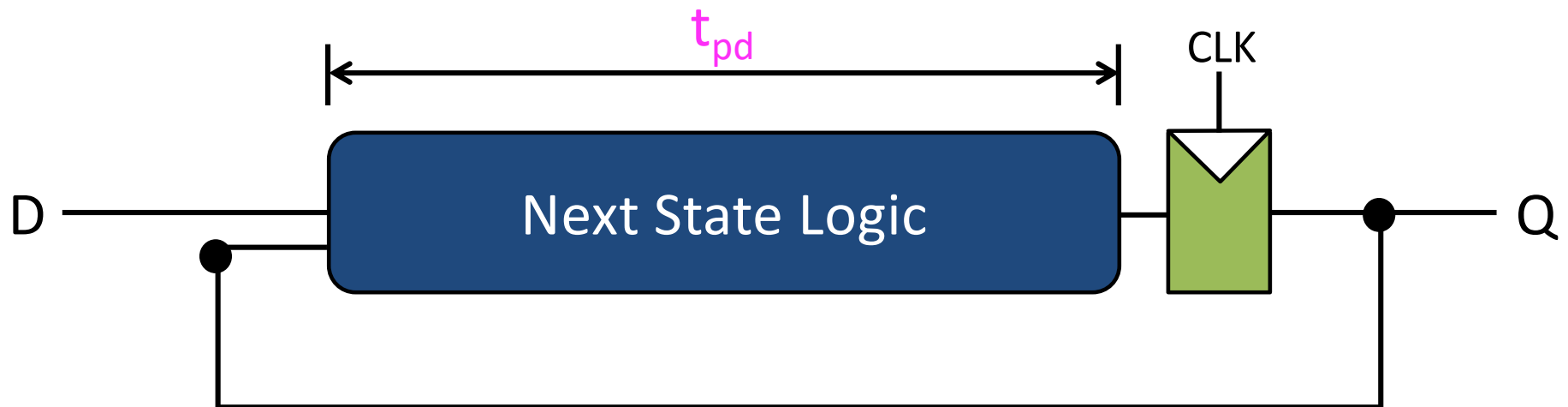
Technology and Hold Time

- It is a reasonable assumption that modern flip-flops have a **hold time** close to **zero**
- We can ignore hold time in subsequent discussions

Example: Calculating Clock Period

- What is the **clock period** for the circuit below for it to work correctly in terms of t_{pd} , t_{setup} , and t_{pcq} ?

$$T_c = t_{pcq} + t_{pd} + t_{setup}$$



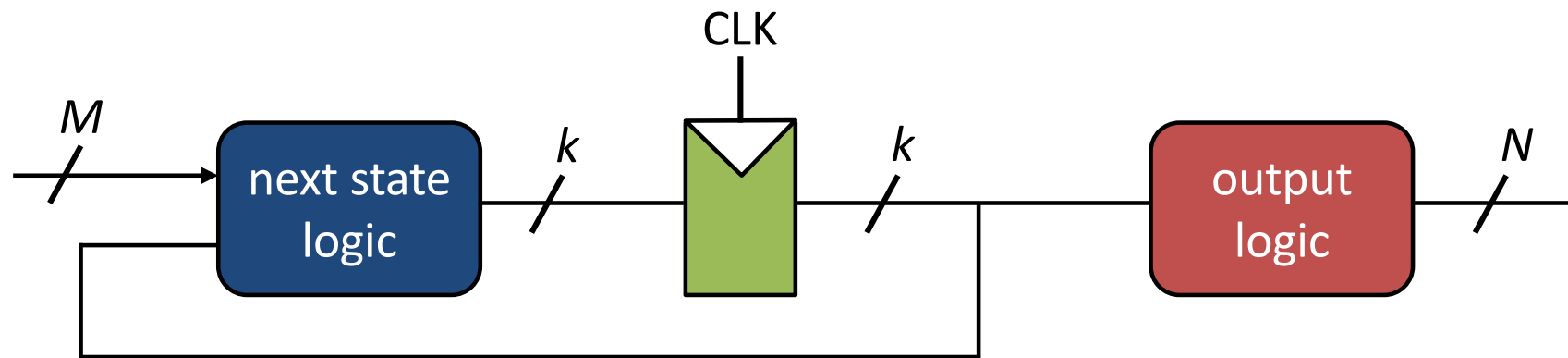
Sequencing Overhead

- $t_{pcq} + t_{setup}$ is called the **sequencing** overhead of the flipflop

- $T_c = t_{pd} + t_{pcq} + t_{setup}$
 - Ideally, the entire **clock period** should be spent doing useful work (**processing done by the combinational circuit**)
 - The **sequencing** overhead of the flip-flop **cuts** into this time

Recall: Synchronous Sequential Circuits

General Rule: *If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated*



Different Types of Storage Elements

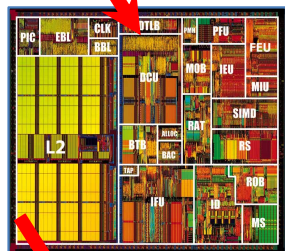
Diff. Storage Elements – Motivation

- **Flip-flops, registers, and flip-flop-based register files make up synchronous sequential circuits**
 - They process information based on current and past combinations of inputs
 - They have arithmetic circuits or logic elements to process inputs and history *stored as state*
 - Processing happens during a clock cycle, while state changes happen at the edge of the clock (**synchronous**)
- **Sometimes, we need to store lots of information**
 - Computer main memory, USB drives, flash drives, disks, SSDs
 - We can build cheaper storage elements that store information asynchronously – and processing happens somewhere else synchronously

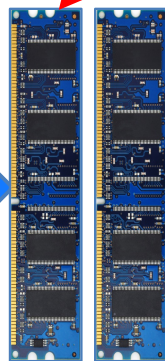
Example

Composed using ALUs, registers, muxes, and decoder to process one instruction every cycle

Just rows and rows of storage elements, and no cycle-by-cycle processing Just read and write words

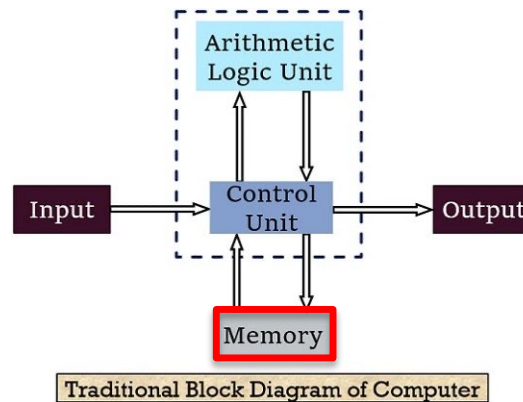


CPU

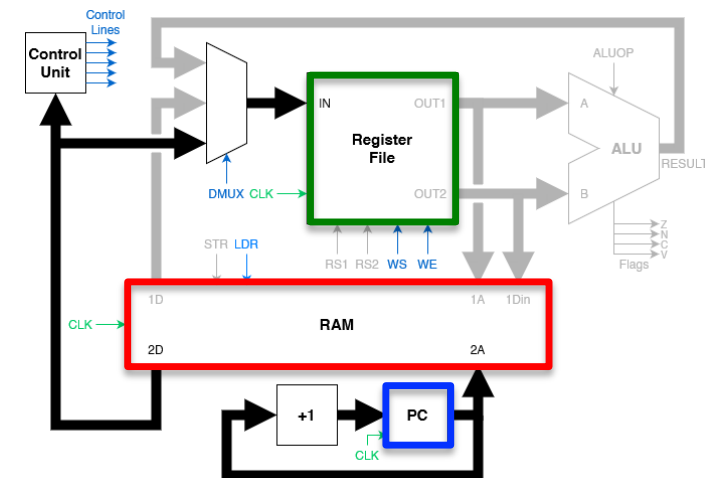


Main Memory

- On-chip fast memory called a L1/L2/L3 cache
- Subset of recently read data from main memory



Traditional Block Diagram of Computer



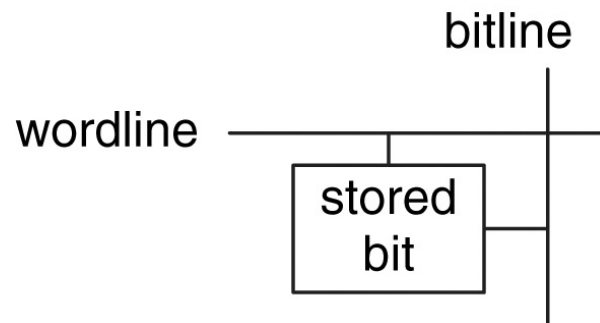
Storage Elements

- **Latches and flip-flops**
 - **Very fast**
 - **Very expensive** (one bit costs tens of transistors)
- **Static RAM (SRAM)**
 - **Relatively fast**
 - **Expensive** (one bit costs **6** transistors)
- **Dynamic RAM (DRAM)**
 - **Slower than latches, flip-flops, and SRAM**
 - **Reading destroys content**, requiring a **refresh** operation to recharge the capacitor
 - **It needs a special process for manufacturing**
 - **Cheap** (one bit **costs only one transistor plus one transistor**)

Section 5.5 of H&H

Generalized Storage Element

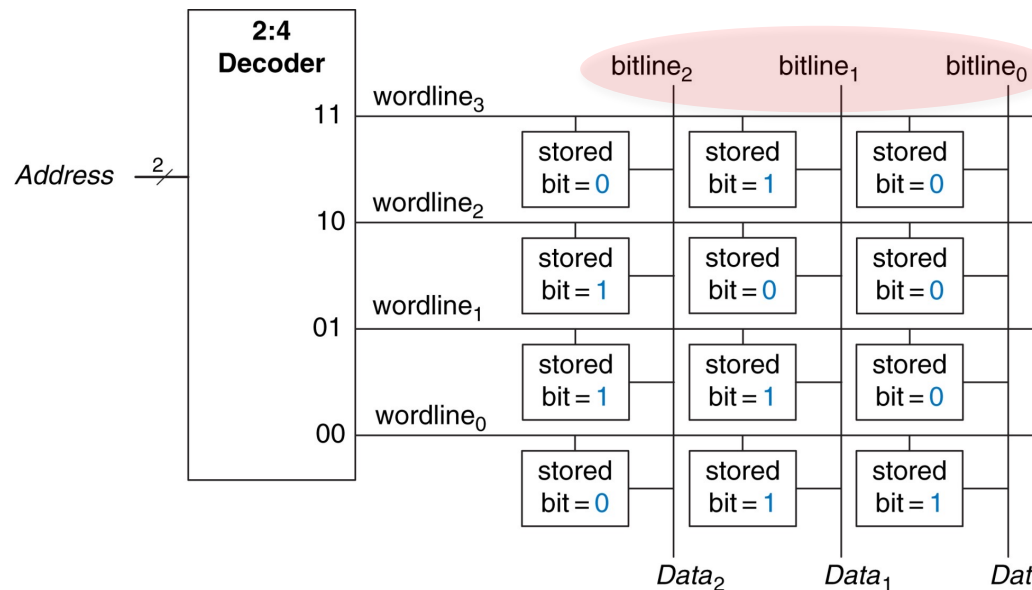
- Stores **one** bit (**bit cell**)
 - **Wordline enables** (**selects**) the storage element or bit cell
 - Bit line is used to **read** the stored bit or **write** a new bit



- Why is it called **word line**?
 - Word line **addresses** an entire (**unique**) word in a row of words
 - We **read** from and **write** to an entire row

Generalized Memory Organization

- A **decoder** to decode the address
- A **multiplexer** to select an output (**not shown below**)
 - Can also use **tri-state buffers** instead of AND/OR multiplexer



called bitlines because they produce the stored bits

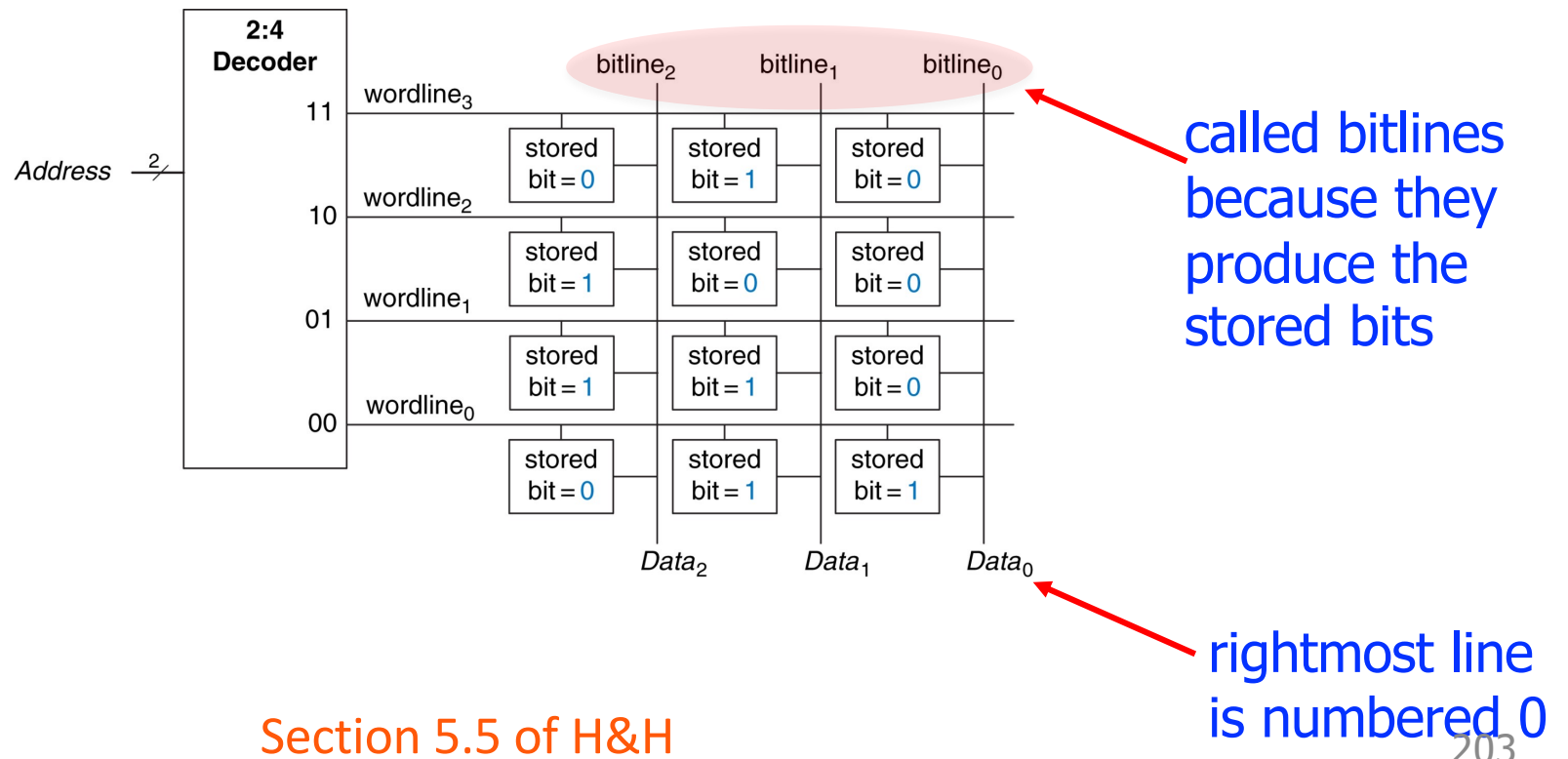
- Storage element can be of any “type”
- **Address/Data** makes up a **single port**

Section 5.5 of H&H

rightmost line is numbered 0

Generalized Memory Organization

- Bitlines are used for both reading the stored bit and writing a new bit (circuit details of how it is done is outside scope)



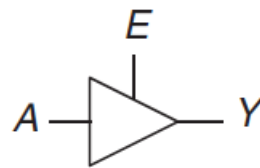
Read/Write Procedure in Detail

- **READ**
 - A **wordline** is asserted, and the corresponding row of bits cells drives the bitlines **HIGH** or **LOW**
- **WRITE**
 - The bitlines are driven **HIGH** or **LOW** first and then a **wordline** is **asserted**, allowing the **bitlines** values to be stored in that row of bit cells

Recall a Tri-State Buffer

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

Tristate Buffer



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

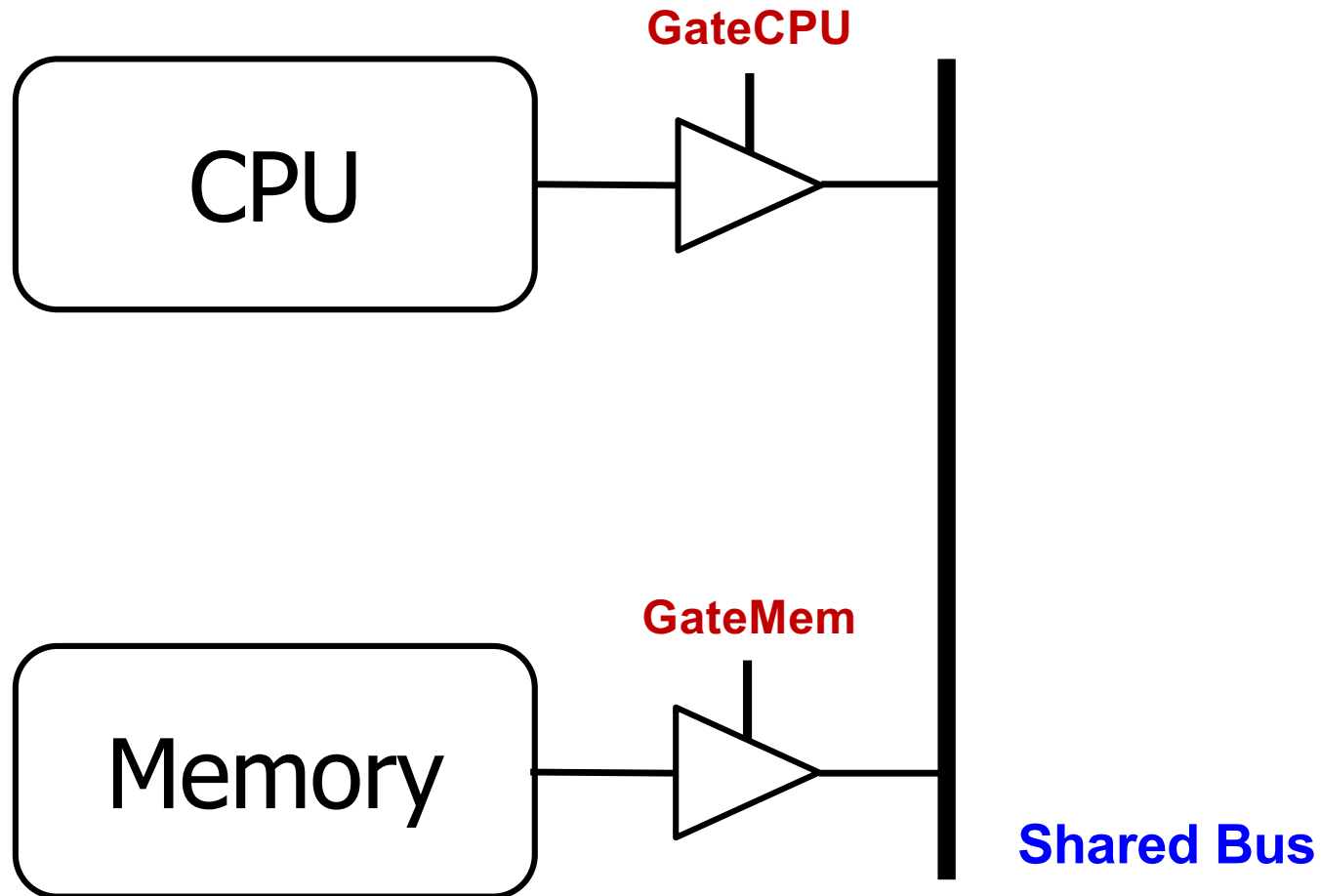
A tri-state buffer acts like a switch

Figure 2.40 Tristate buffer

- **When E is LOW, output is a floating signal (Z)**
- **Floating:** Signal not driven by any circuit (open circuit, floating wire)

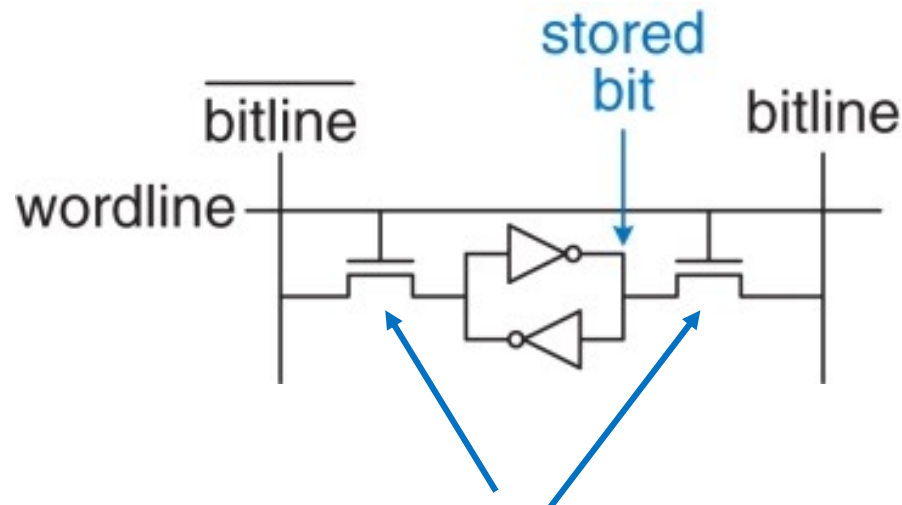
Memory Port with Tri-State Buffer

Homework Exercise!



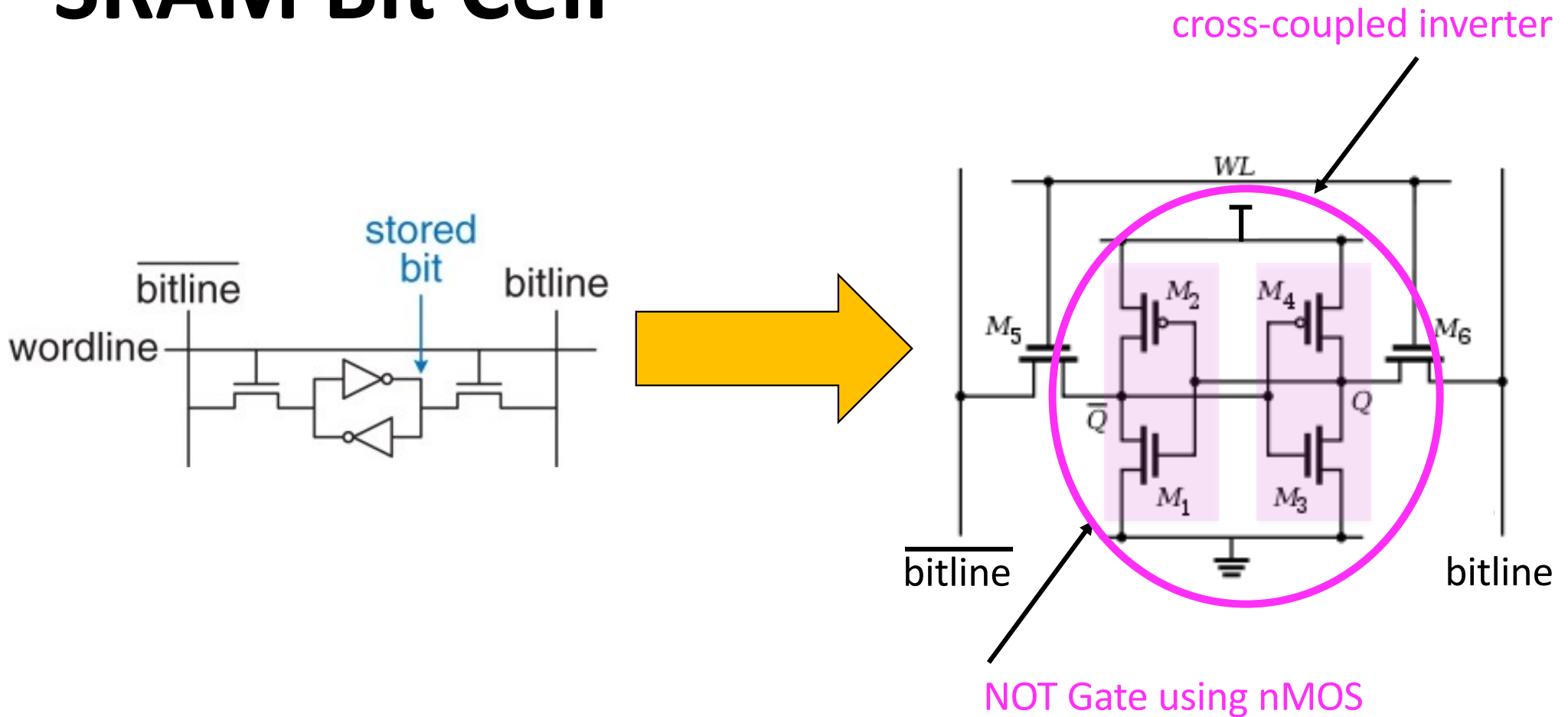
SRAM Bit Cell

- **SRAM**: Static Random Access Memory
 - Data bit is stored in a cross-coupled inverter
 - Each cell has two outputs: bitline and **bitline** and **bitline**



- When the wordline is asserted, both nMOS transistors are turned **ON**, and data values are **transferred** to or from the **bitlines**

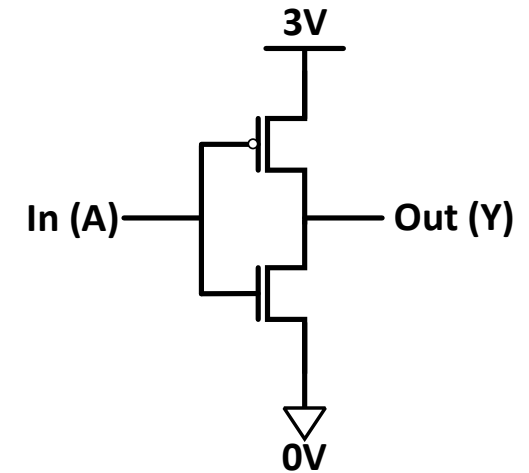
SRAM Bit Cell



source: <https://electronics.stackexchange.com/questions/162466/how-do-the-access-transistors-in-an-sram-cell-work>

Recall: MOS NOT Gate (Inverter)

- We have seen a NOT gate at the transistor level
 - If A = **0V** then Y = **3V**
 - If A = **3V** then Y = **0V**
- Interpretation of voltage levels
 - Interpret **0V** as logical (binary) **0** value
 - Interpret **3V** as logical (binary) **1** value



A	P	N	Y
0	ON	OFF	1
1	OFF	ON	0

$$Y = \bar{A}$$

Random Access Memory (RAM)

- **RAM**: Random Access Memory
 - Each byte has an address (byte-addressable)
 - Any data word (*consisting of multiple bytes*) can be accessed independently of the other
 - **Any data word is accessed with the same delay as any other**

Another Device for Storing Information

- What storage medium is this?



- A length of magnetic tape in a plastic enclosure with one or two reels for controlling the motion of the tape
- **Cassette used in tape recorders:** nearby data is accessed more quickly than faraway data
- **Inherently sequential device:** they must be rewound and read from the start to load data

Yet Another Device for Storing Information

- What storage medium is this?



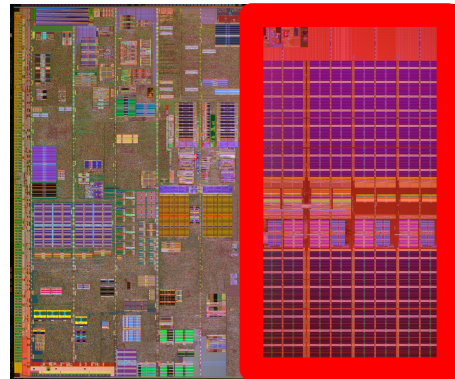
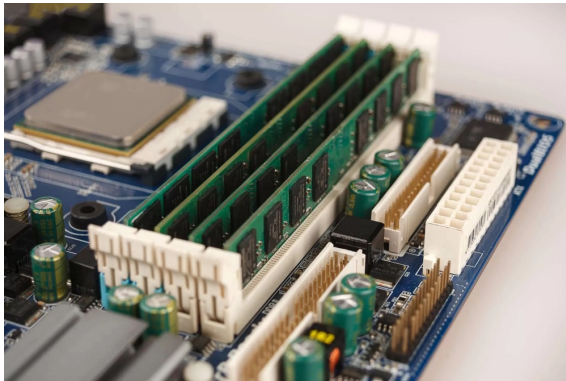
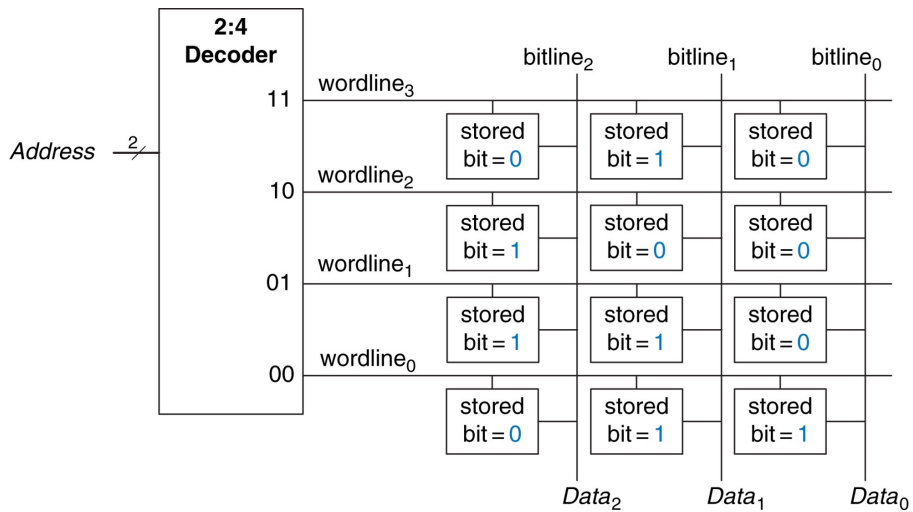
- A rotational platter with magnetic coating and arm assembly inside a casing
- **Hard disk used in all computers:** nearby data is accessed more quickly than faraway data
- **Inherently sequential device:** a disk must rotate under an arm for data to be read

Random vs. Sequential access devices

These

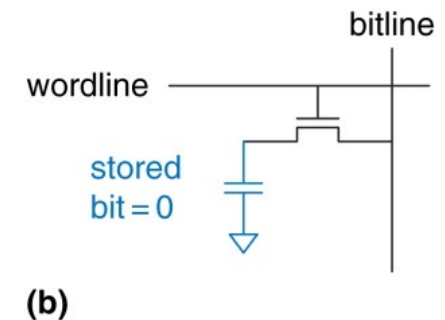
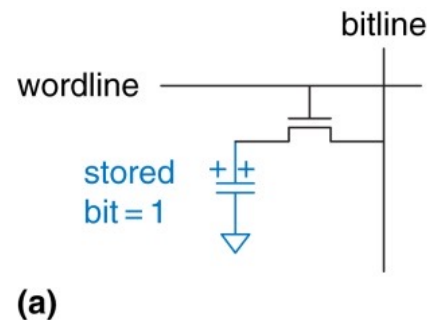
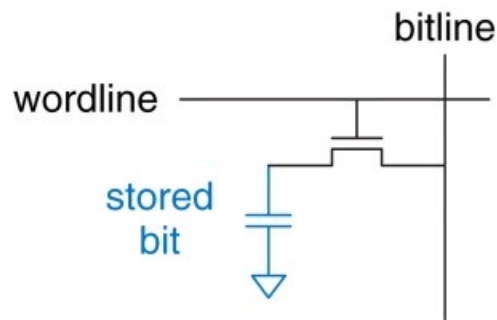
vs.

These



DRAM Bit Cell

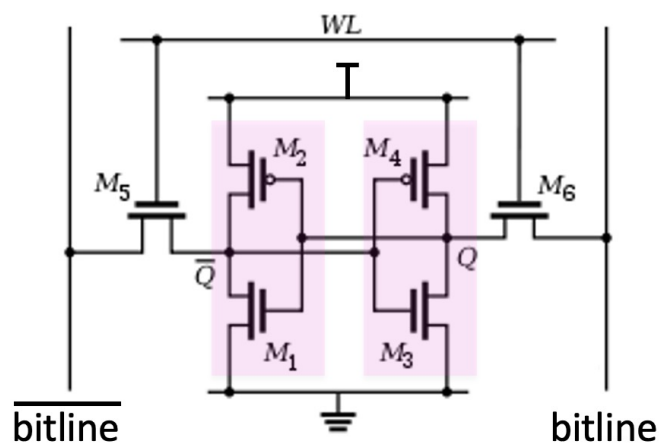
- **DRAM**: **D**ynamic **R**andom **A**ccess **M**emory
 - Stores a **bit** as the presence or absence of a **charge** on a capacitor
 - The **nMOS** transistor behaves as a **switch** that either connects or disconnects the capacitor from the **bitline**



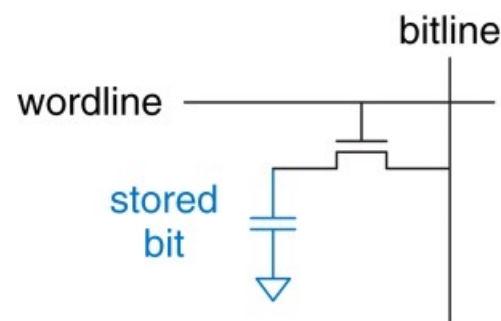
- When the wordline is **asserted**, the **nMOS transistor** turns **ON**, and the stored bit value **transfers to or from** the **bitline**

Static vs. Dynamic

- Recall the **SRAM** cell: The **nMOS transistors** (M5 and M6) are driven by cross-coupled inverter connected to **supply voltage**
- **DRAM**: The capacitor node is not actively driven **HIGH** or **LOW** by a transistor tied to **supply voltage**



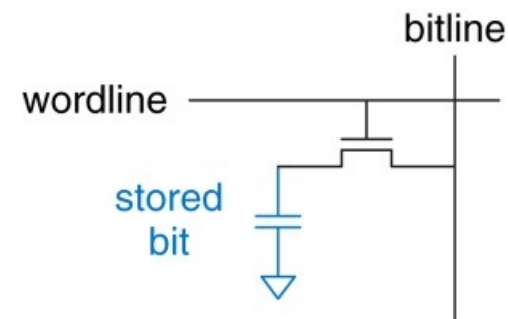
SRAM Cell



DRAM Cell

DRAM Reads and Writes

- **Read:** Upon a **read**, data values are **transferred** from the capacitor to the bitline
 - **Reading destroys** the bit value stored on the capacitor, so the data word must be **restored (rewritten)** after each **read**
 - **Refresh:** Even when DRAM is not **read**, the contents must be **refreshed** (read and rewritten) every **few milliseconds**, because the charge on the capacitor gradually **leaks** away



- **Write:** Upon a **write**, data values are **transferred** from the bitline to the capacitor

DRAM Refresh

- **Refresh:** Even when DRAM is not **read**, the contents must be **refreshed** (read and rewritten) every **few milliseconds** because *the charge on the capacitor* gradually **leaks** away
- **Refreshing consumes** extra energy
- **Refreshing** requires costly circuitry
 - It is a **critical drawback** of DRAM technology, especially at large capacities (many gigabytes, for example)

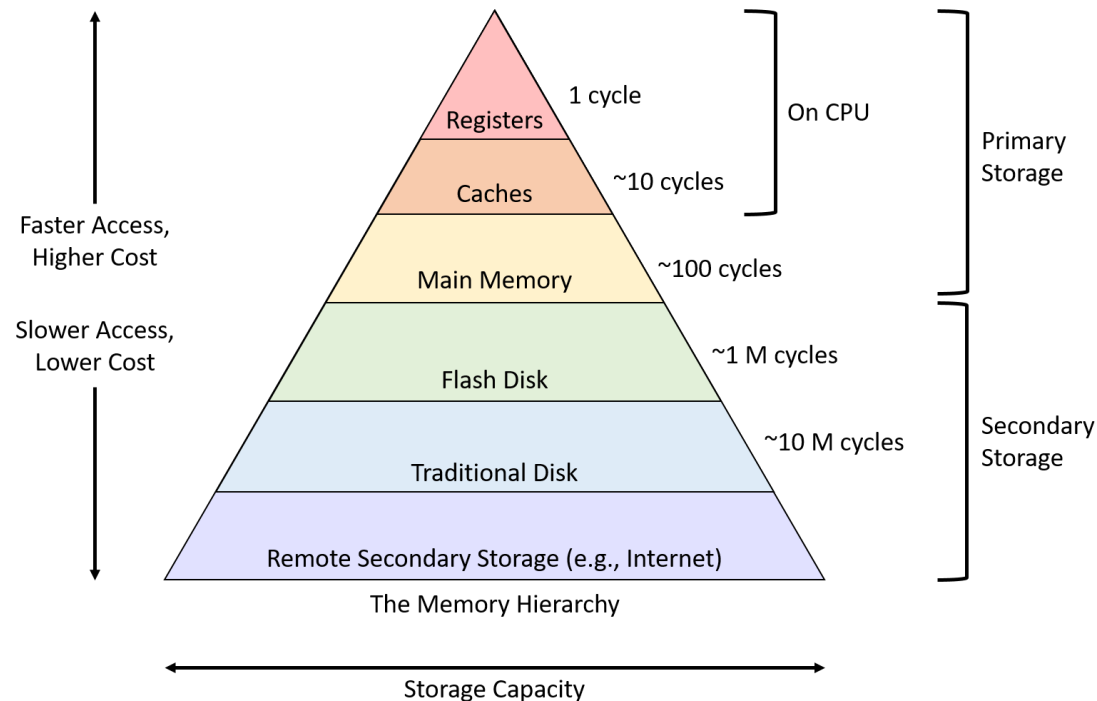
Memory Comparison

- **Latches and flip-flops (20 transistors)**
 - **Very fast**
 - **Very expensive** (one bit costs tens of transistors)
- **Static RAM (SRAM) (6 transistors)**
 - **Relatively fast**
 - **Expensive** (one bit costs 6 transistors)
- **Dynamic RAM (DRAM) (1 transistor)**
 - **Slower than latches, flip-flops, and SRAM**
 - **Reading destroys content**, requiring a **refresh** operation to recharge the capacitor
 - **It needs a special process for manufacturing**
 - **Cheap** (one bit **costs only one transistor plus one transistor**)

Section 5.5 of H&H

Memory Hierarchy

- Large memories use **SRAM** and **DRAM**
- Registers inside the **CPU** use **flip-flops**



- Flip-flops are used in **synchronous sequential circuits**

