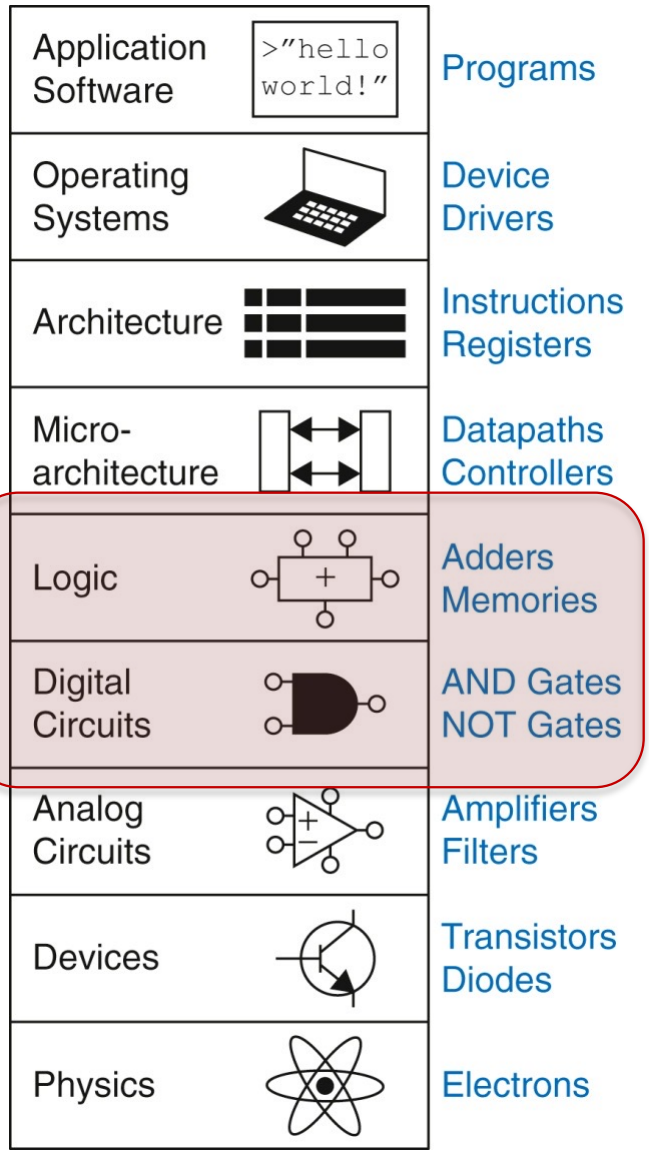
COMP2300-COMP6300-ENGN2219
Computer Organization &
Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

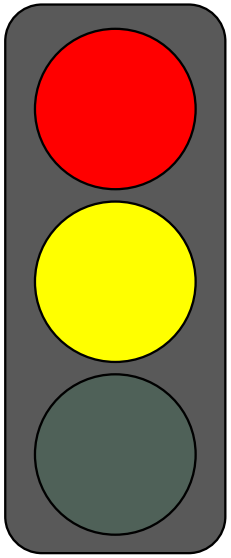
Australian
National
University



Broadening our horizon "one layer at a time"

What will we learn this week?

- Synchronous sequential circuits
  - Finite state machines
  - State and Clock
- Synchronous vs. Asynchronous
- Timing Issues



- Architecture
- ARM ISA

First, We Will Finish

Sequential Logic

Where we ended last time

- **Static RAM (SRAM):** No need to refresh periodically. Transistors always connected to active power
- **Dynamic RAM (DRAM):** Capacitor requires periodic refreshing (hence dynamic)
- **RAM: R**andom **A**ccess **M**emory: Take an equal amount of time to access (read/write) any location

Sequential Logic Circuits

- **Week 1 & 2:** Circuits that process information
- **Week 3:** Circuits that can store information, basic storage elements and memory
- **Now:** Digital logic structures that can **both** process information (i.e., make decisions) **and** store information
  - **Decision is based on both input combinations and past inputs**

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

Sequential

Opens depending on past inputs

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

State

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

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

State Diagram of Sequential Lock

- Completely describes the operation of the sequential lock

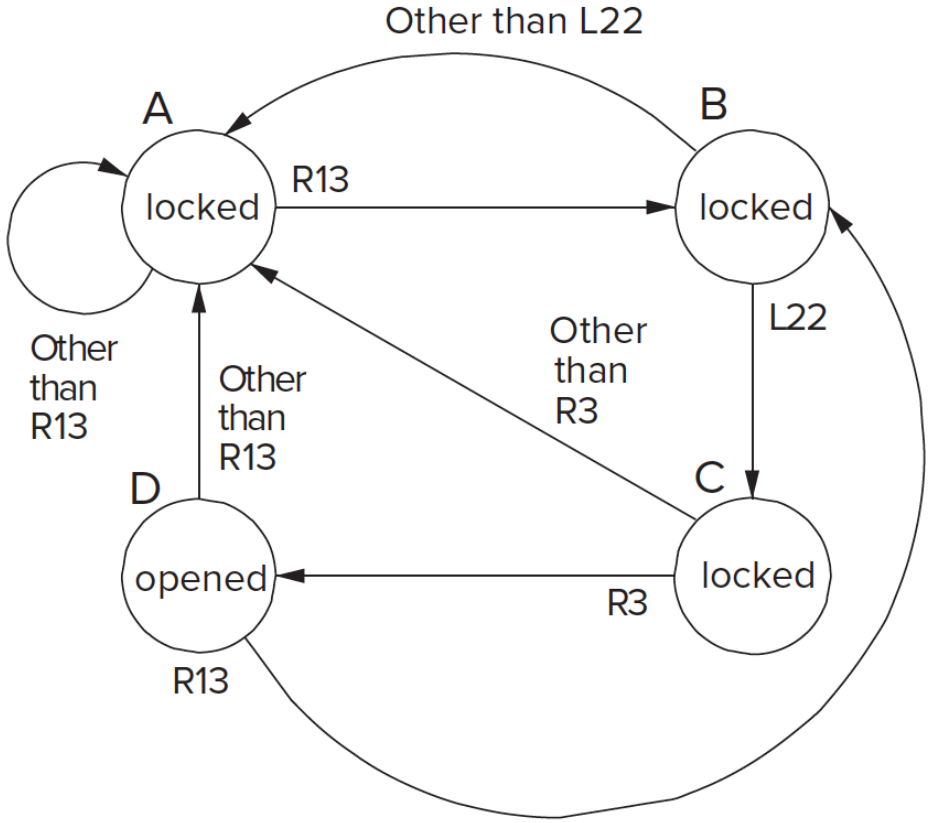


Image source: Patt and Patel, "Introduction to Computing Systems", 2nd ed., page 76.

Another Example: Soft Drink Machine

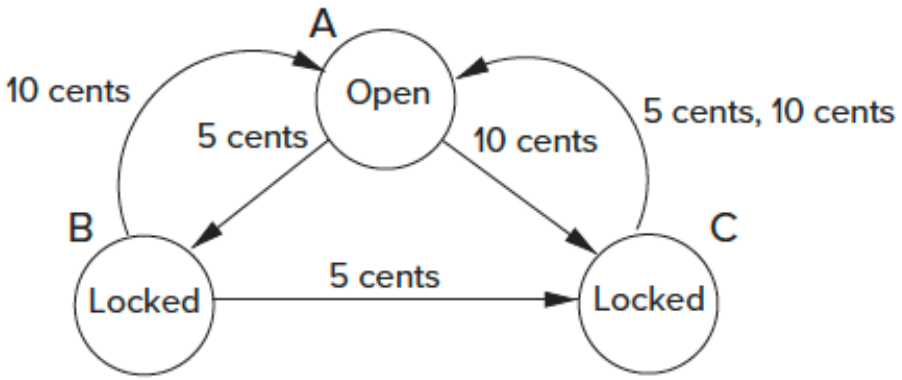


Figure 3.27 State diagram of the soft drink machine.

- 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.

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

Another Example: Soft Drink Machine

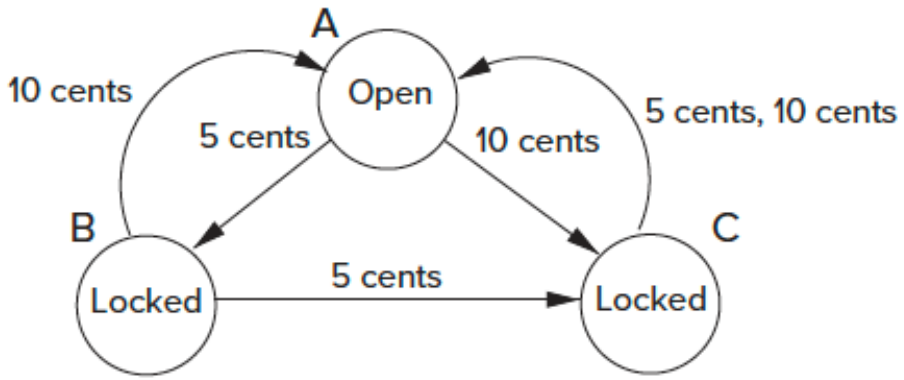


Figure 3.27 State diagram of the soft drink machine.

- One possible sequence of states is as follows

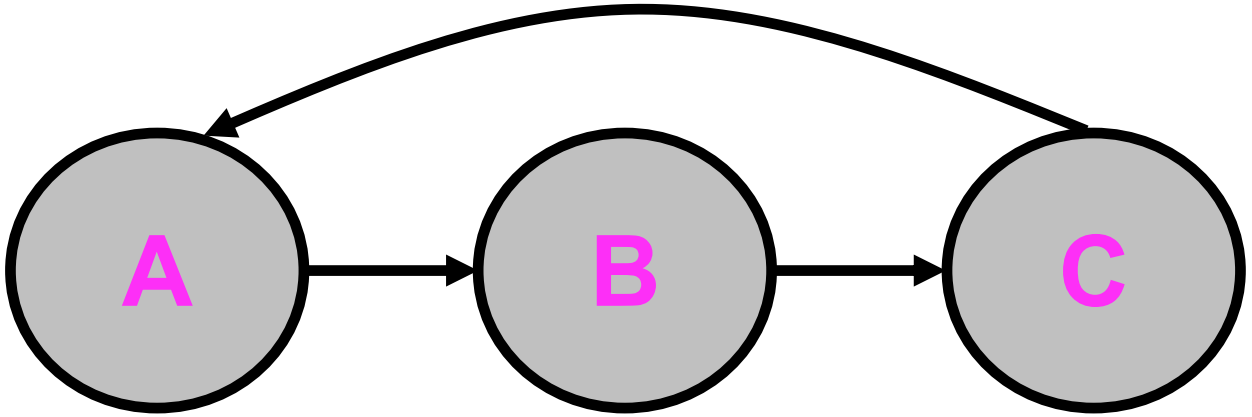
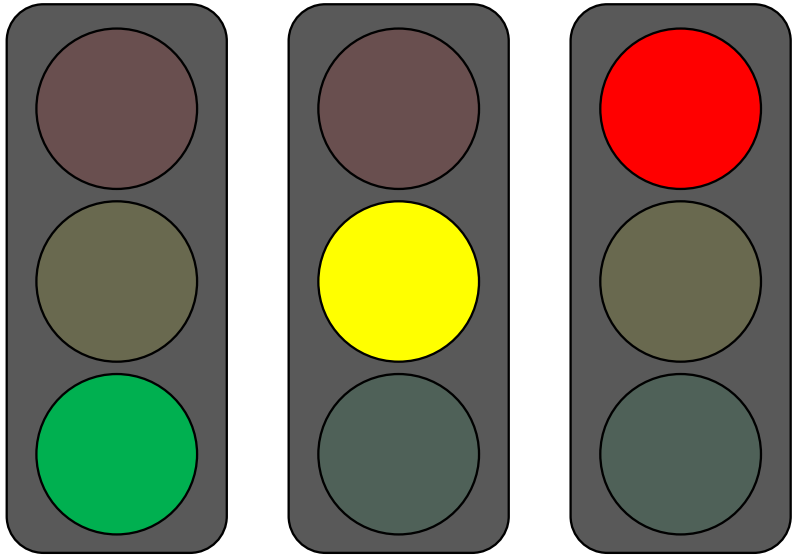


Image source: Patt and Patel, "Introduction to Computing Systems", 2nd ed., page 84.

Another Example: Traffic Light

- There are only three possible states:
  A. Green
  B. Yellow
  C. Red



- The sequence of states is as follows

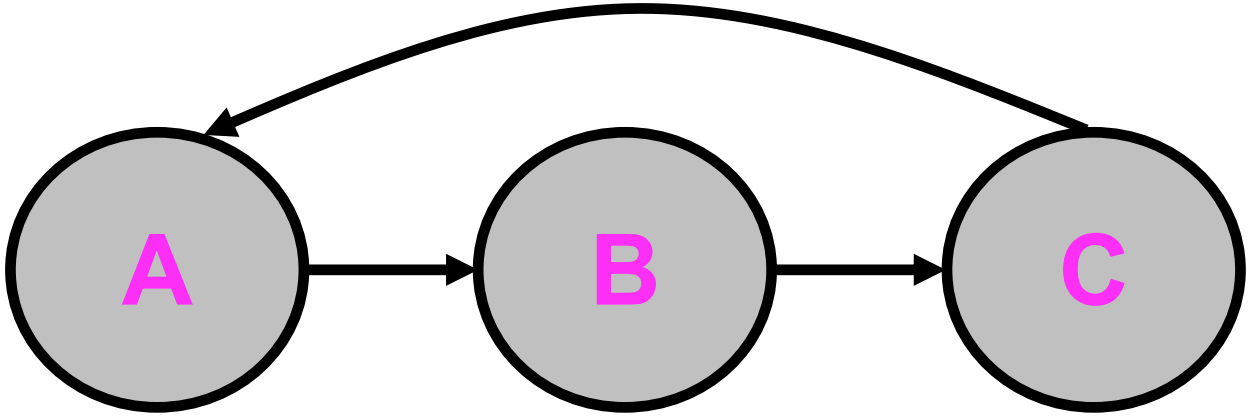
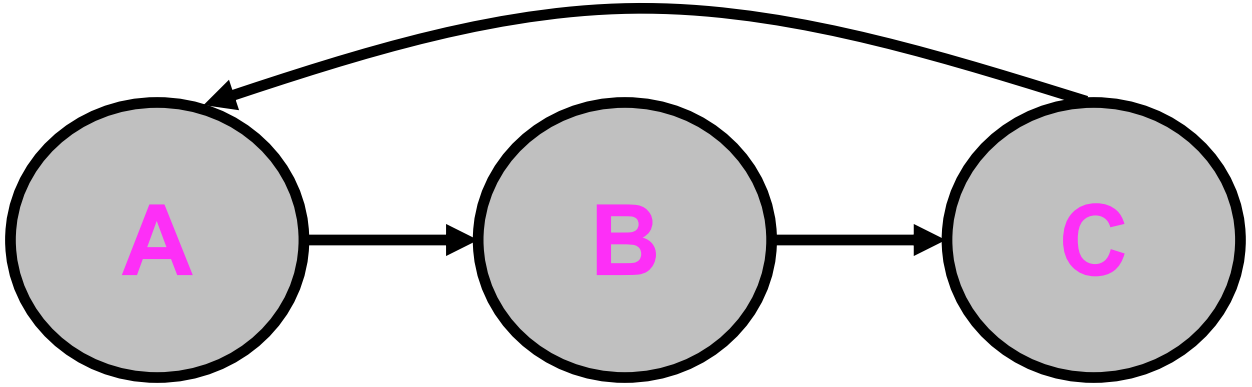


Image source: Patt and Patel, "Introduction to Computing Systems", 2nd ed., page 84.

Asynchronous vs. Synchronous State Changes

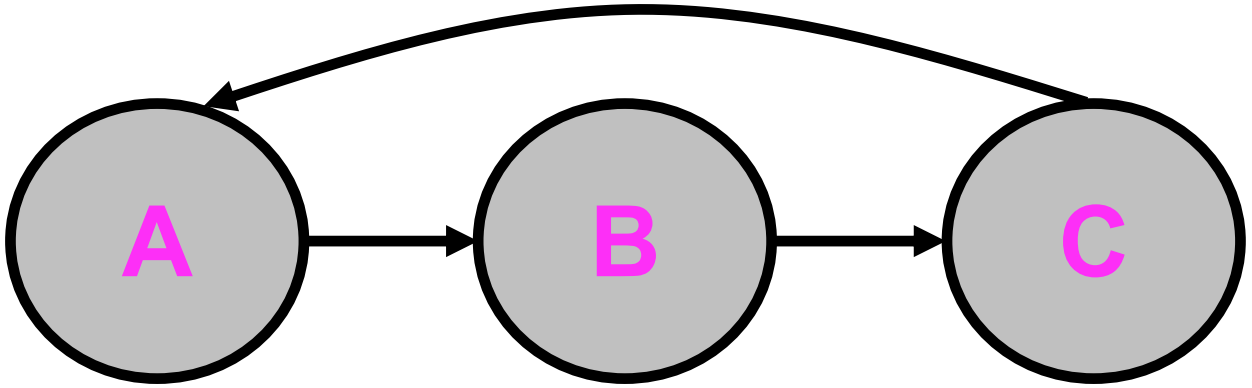
- Sequential lock we saw is an asynchronous machine
  - State transition occur when they occur
  - There is nothing that synchronizes when each state transition must occur
- Most modern computers are synchronous machines
  - State transitions take place after fixed units of time
  - Controlled in part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

Changing State: The Notion of Clock (I)

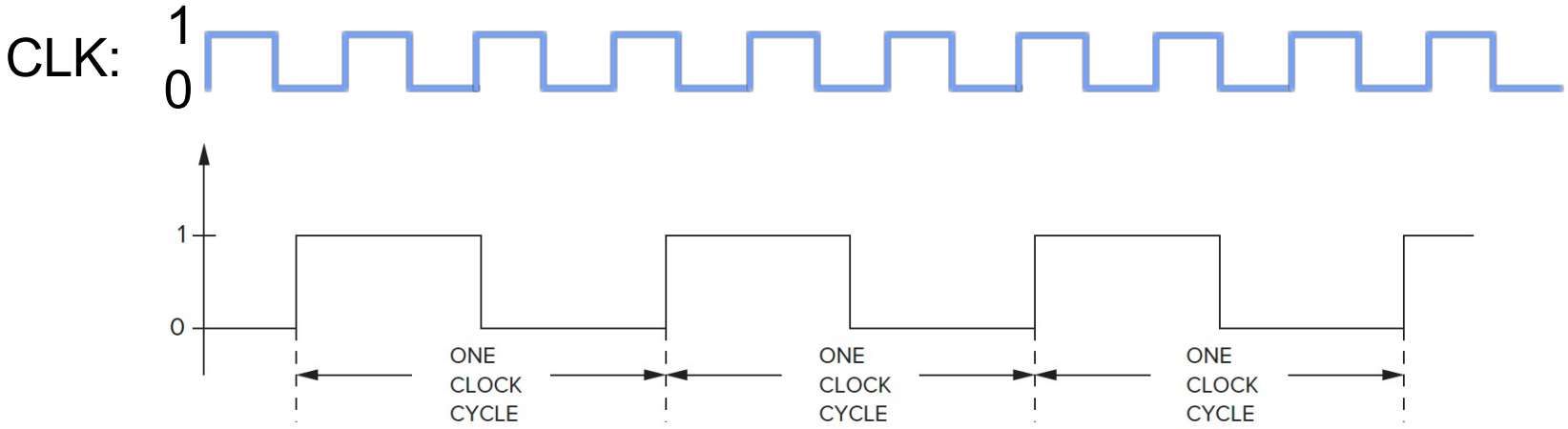


- When should the vending machine change state from **A** to **B**?
- When should the traffic light change from one state to another?

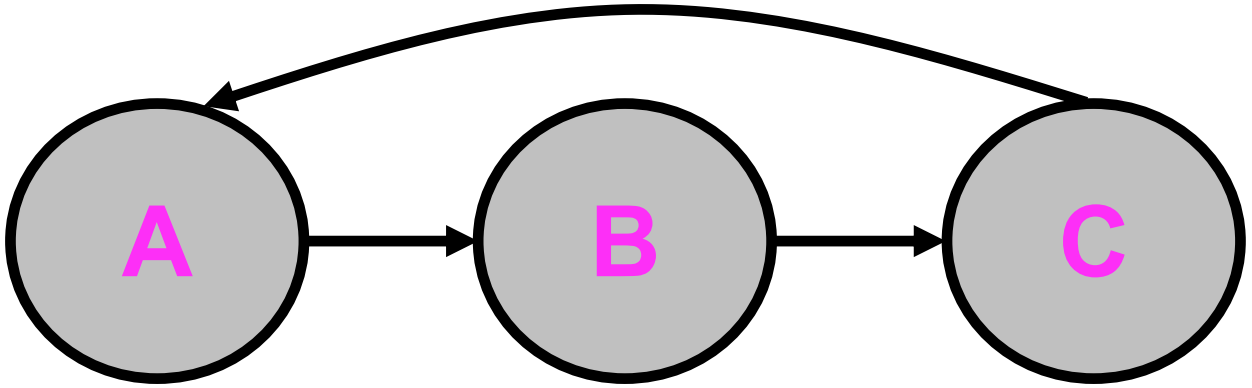
Changing State: The Notion of Clock (I)



- 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



Changing State: The Notion of Clock (I)



- 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 (II)

- **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

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
  - Each state of the system at a given time

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

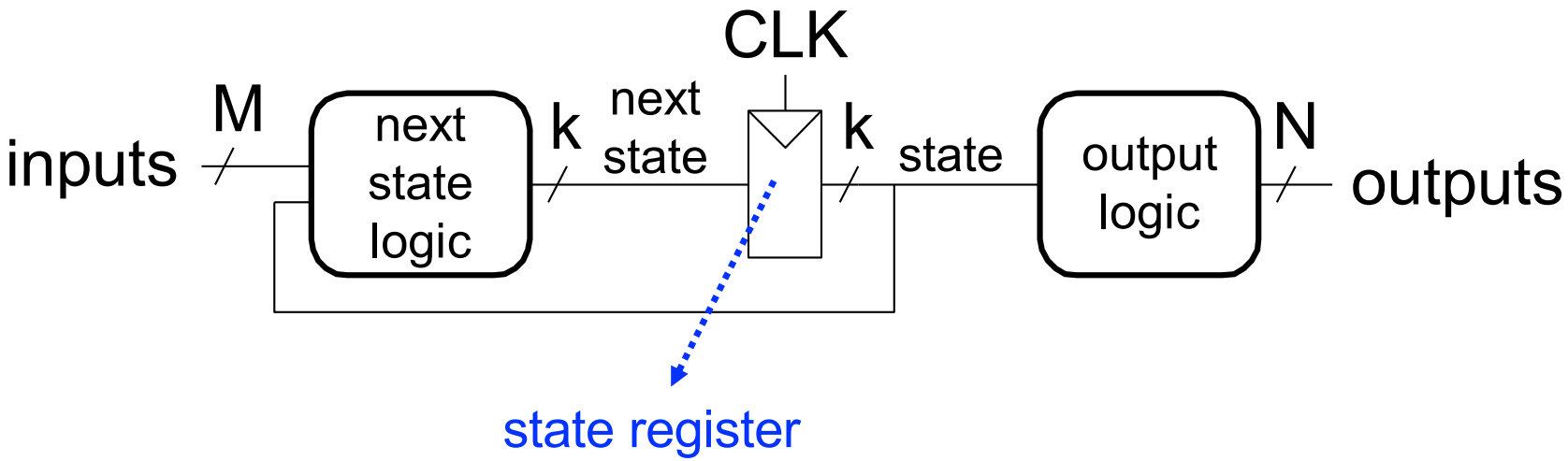
- An **FSM** can model
  - A traffic light, an elevator, microwave, microprocessor, fan speed, car lock

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
    - E.g., If state is A, then output is RED; If state is B, output is YELLOW

Finite State Machines (FSMs)

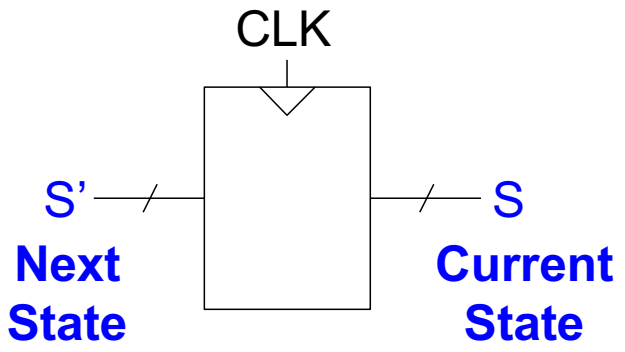
- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic



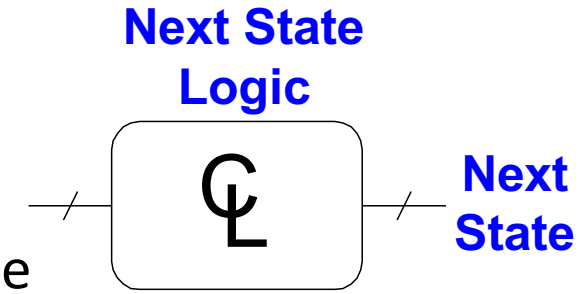
Section 3.4 of H&H

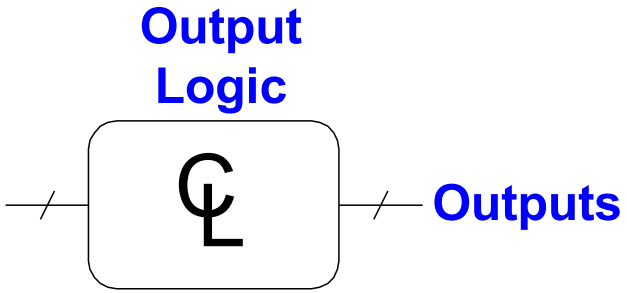
FSMs Consist of:

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



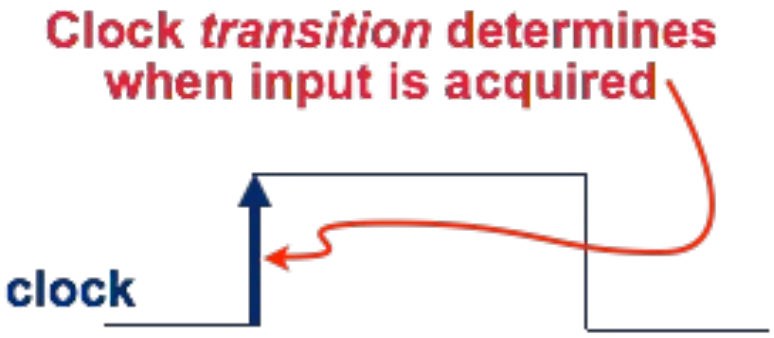
- **Combinational Circuits**
  - Next state logic
    - Determines what the next state will be
  - Output logic
    - Generates the outputs





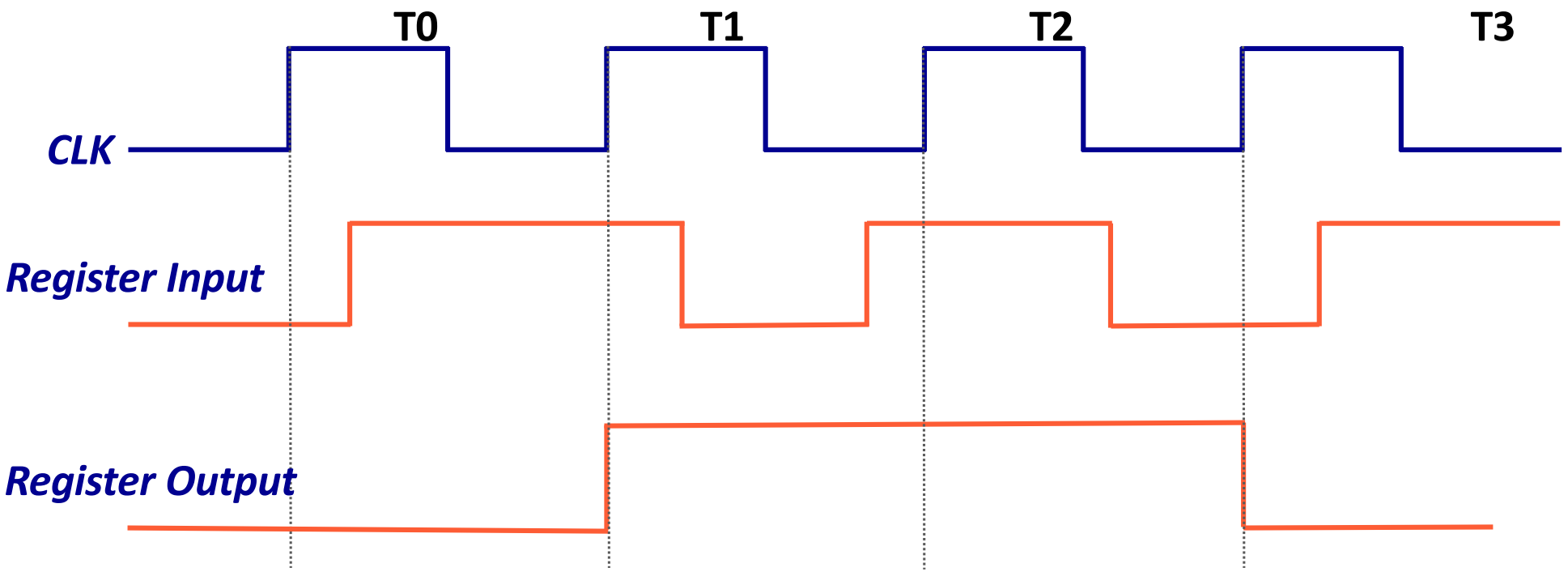
State Register

- We need flip-flops (not latches) to implement state register. **Why?**
- Properties of state register
  - We need to store data at the **beginning** of every clock cycle



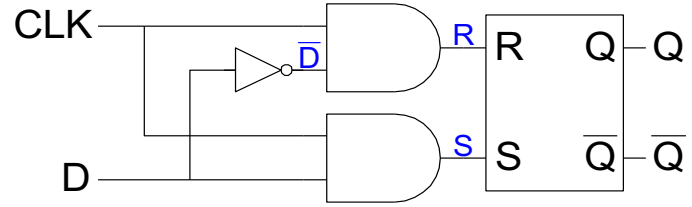
State Register

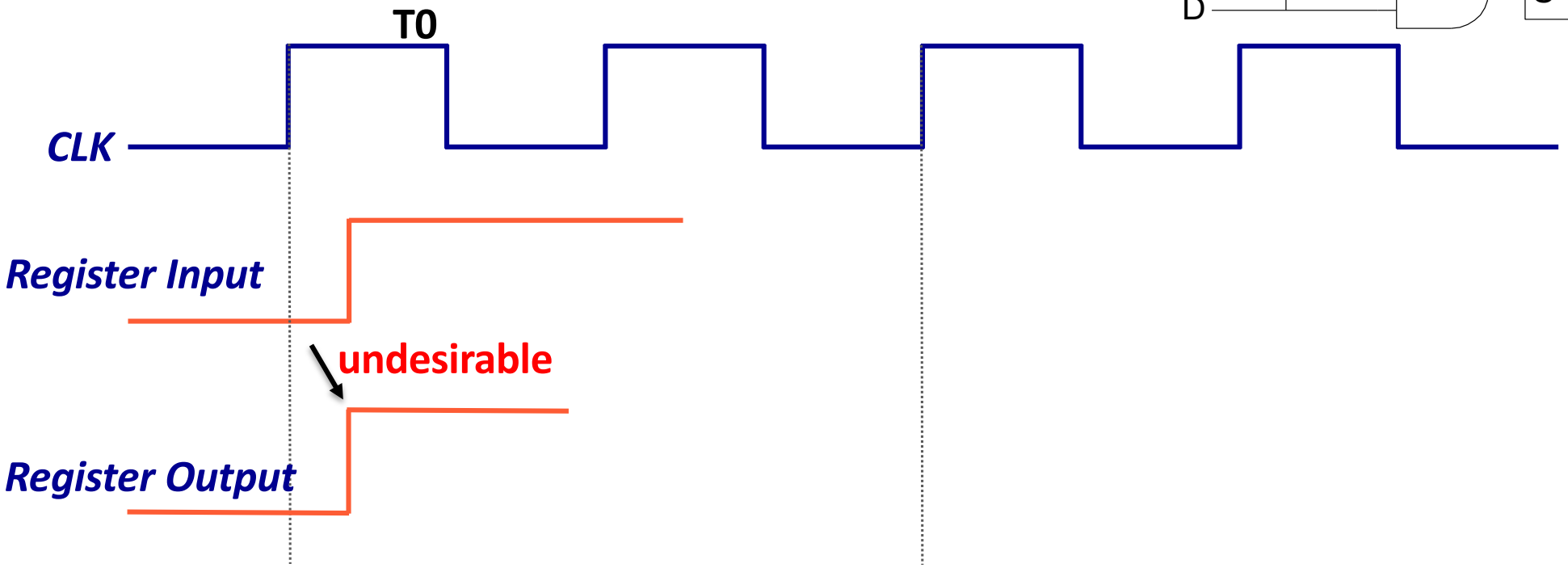
- We need flip-flops (not latches) to implement state register. **Why?**
- Properties of state register
  - The data must be **available** during the **entire clock cycle**



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

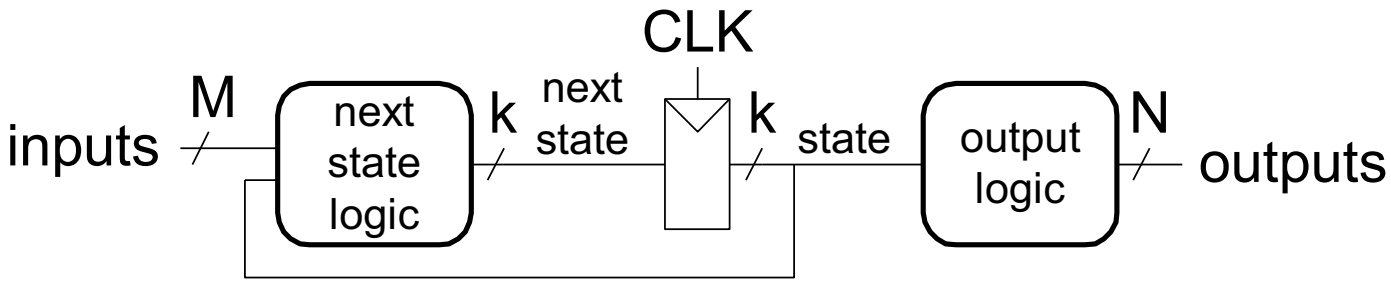
Acknowledgement: *Selection of Slides from Digital Design and Computer Architecture, Onur Mutlu, ETH Zurich, Spring 2022*
https://safari.ethz.ch/digitaltechnik/spring2022/doku.php?id=schedule

Finite State Machines (FSMs)

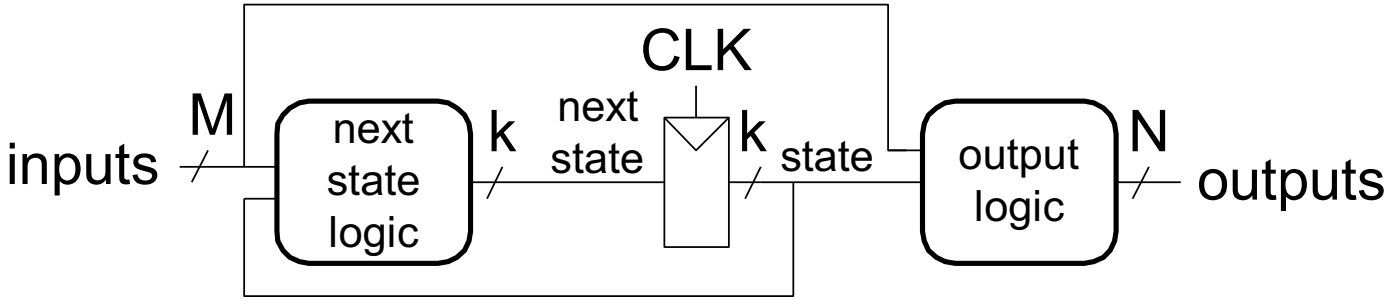
- Next state is determined by the current state and the inputs
- Two types of finite state machines differ in the **output logic**:
  - **Moore FSM**: outputs depend only on the current state
  - **Mealy FSM**: outputs depend on the current state and inputs

Finite State Machines (FSMs)

Moore FSM

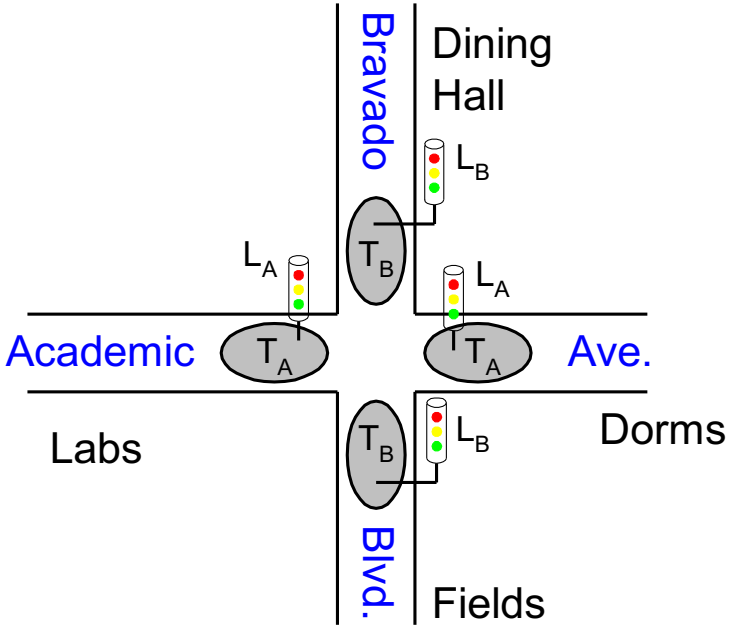


Mealy FSM



Finite State Machine Example

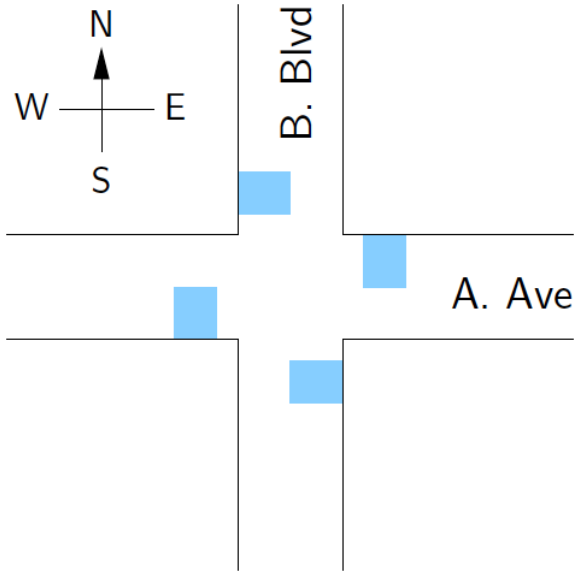
- "Smart" traffic light controller
  - **2 inputs**:
    - Traffic sensors: T_A , T_B (TRUE when there's traffic)
  - **2 outputs**:
    - Lights: L_A , L_B (Red, Yellow, Green)
  - State can change every 5 seconds
    - Except if green and traffic, stay green



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

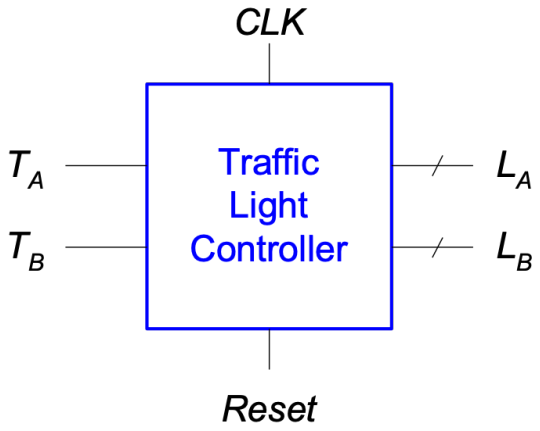


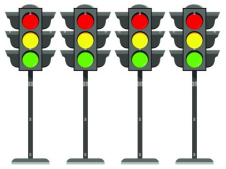
T_A = (eastbound traffic on A) OR (westbound traffic on A)

T_B = (northbound traffic on B) OR (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

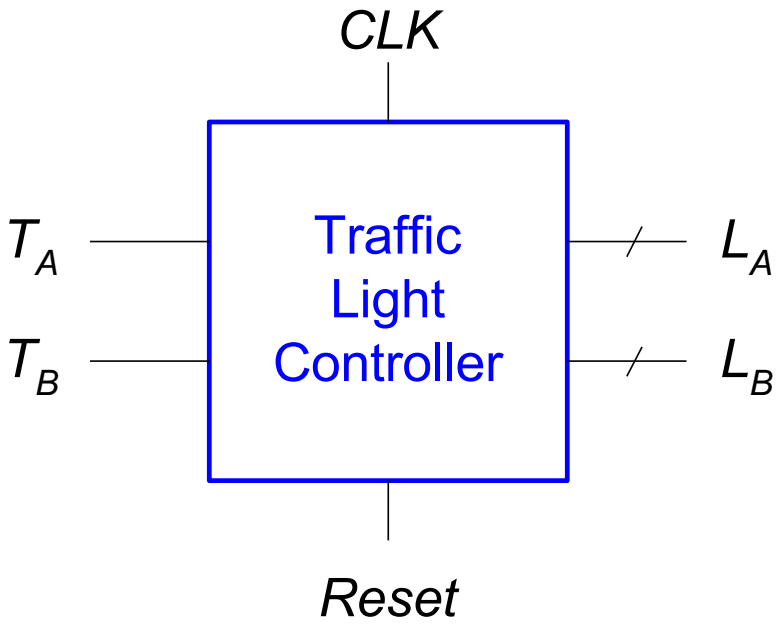




Output	Encoding
GREEN	00
YELLOW	01
RED	10

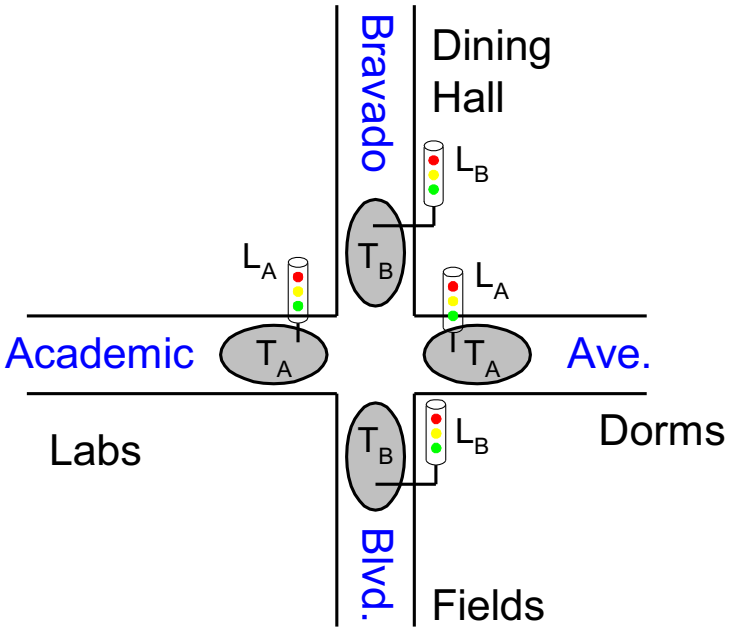
Finite State Machine Blackbox

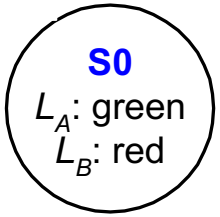
- **Inputs:** CLK, Reset, T_A , T_B
- **Outputs:** L_A , L_B



Finite State Machine Diagram

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

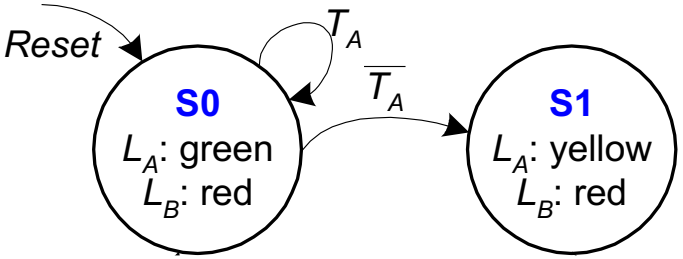


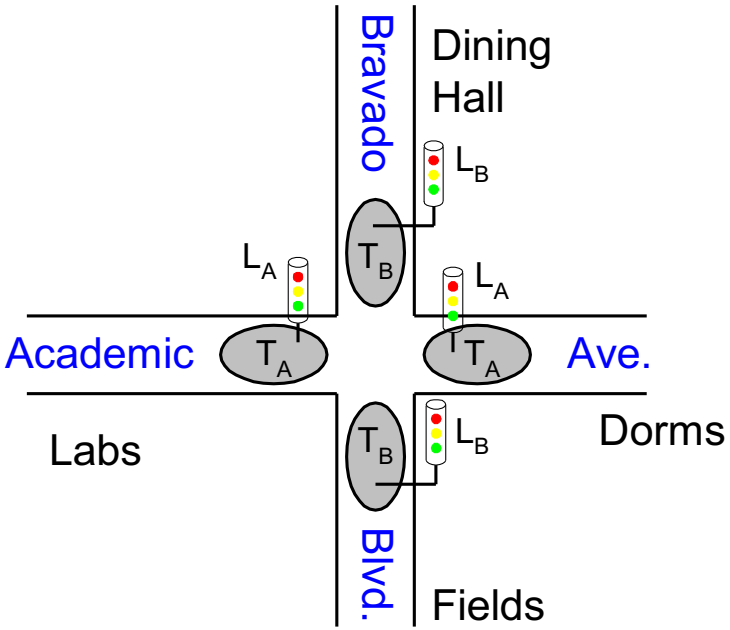


Finite State Machine Diagram

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

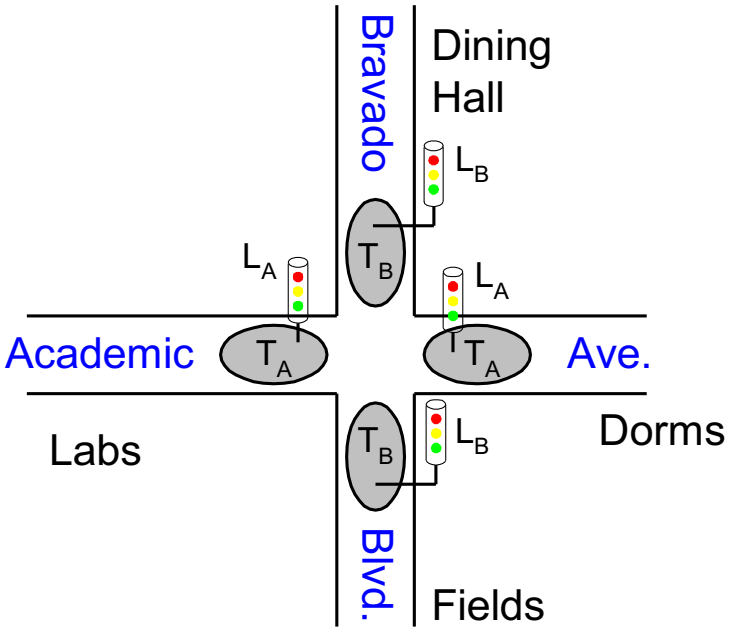
→ From "current state" **S0** to "next state" **S1**

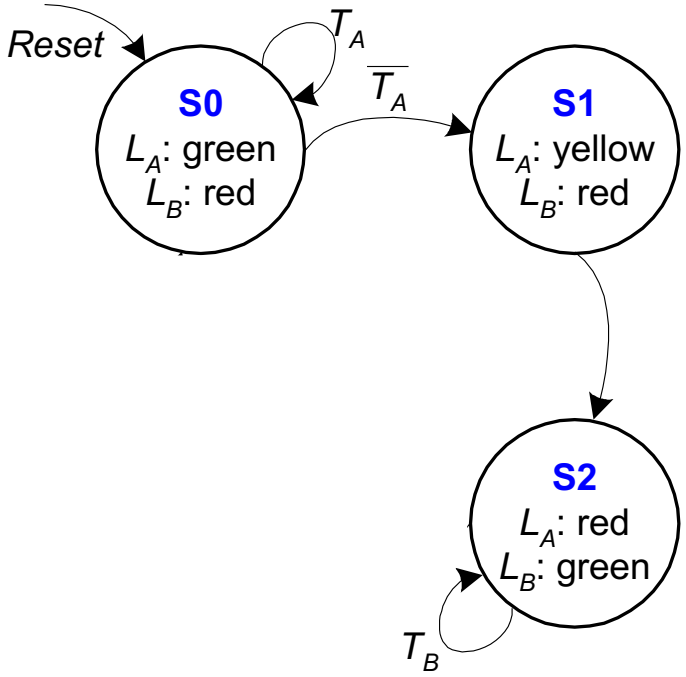




Finite State Machine Diagram

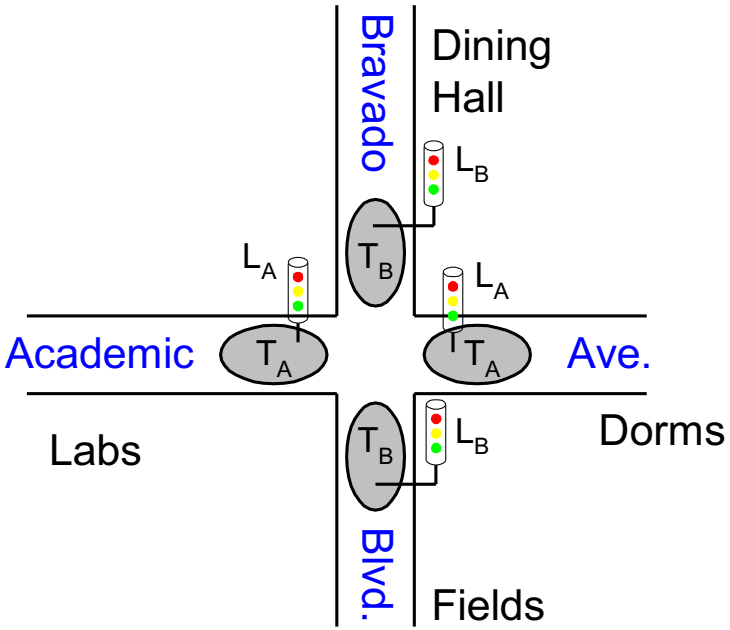
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arrows (Arcs)

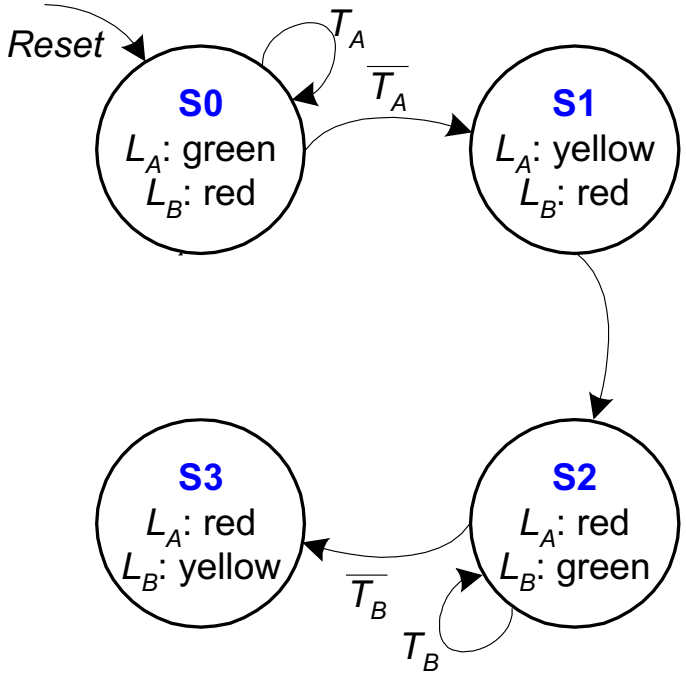




Finite State Machine Diagram

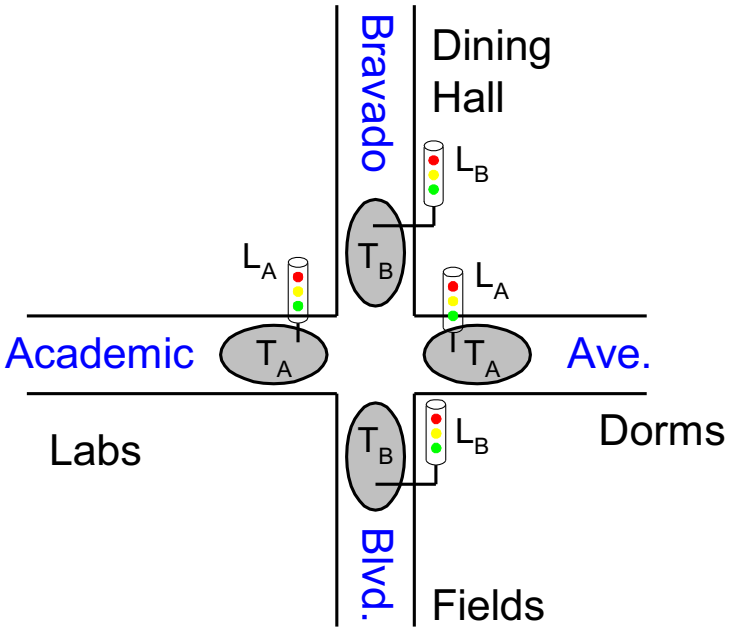
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arrows (Arcs)

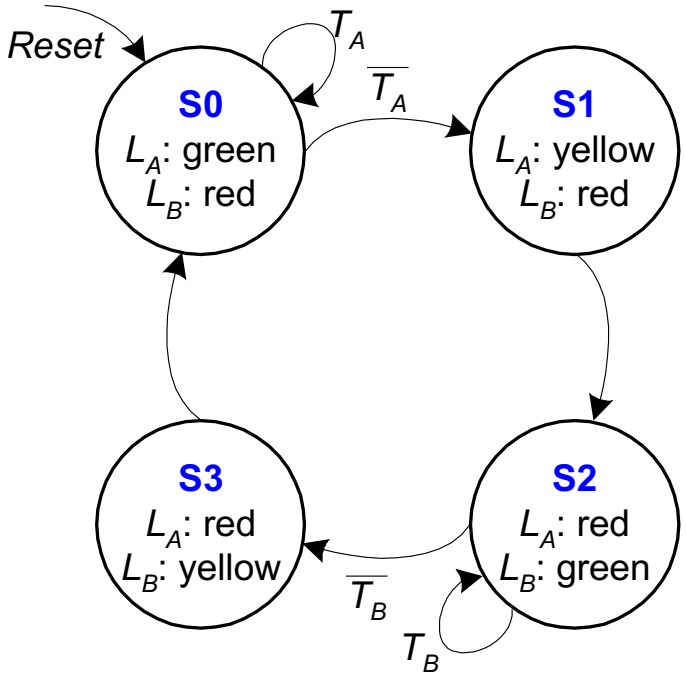




Finite State Machine Diagram

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

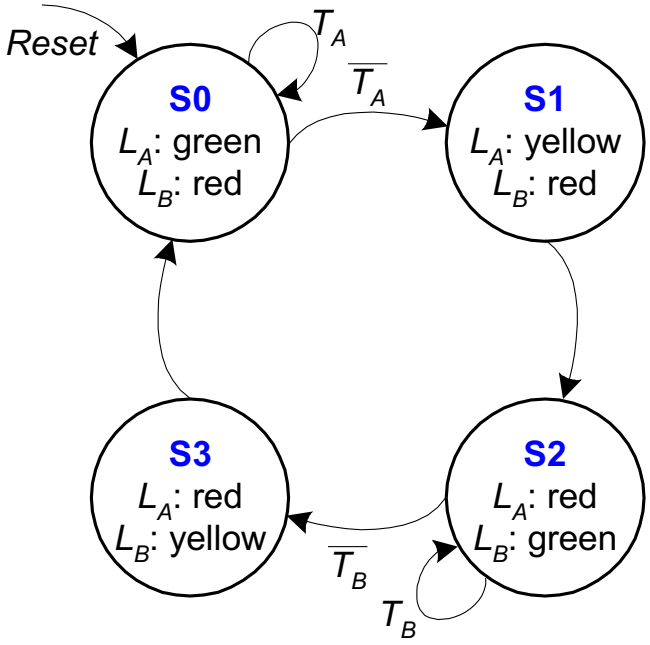




Finite State Machine:

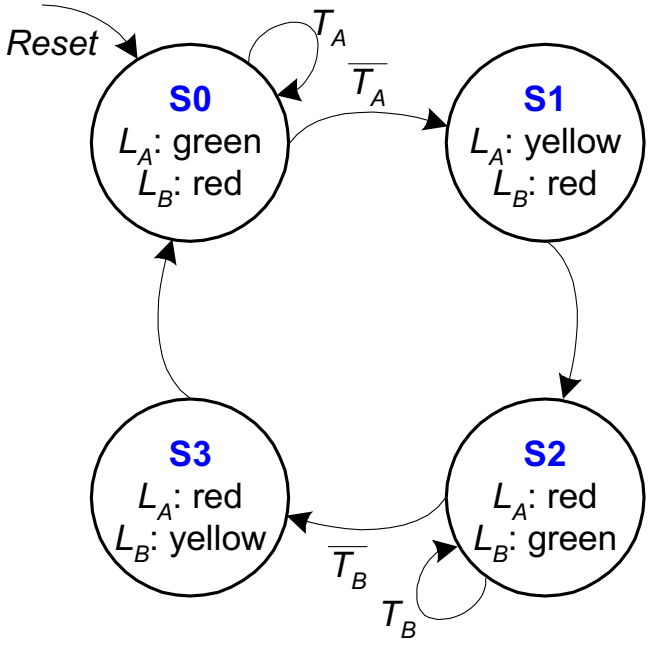
State Transition Table

FSM State Transition Table



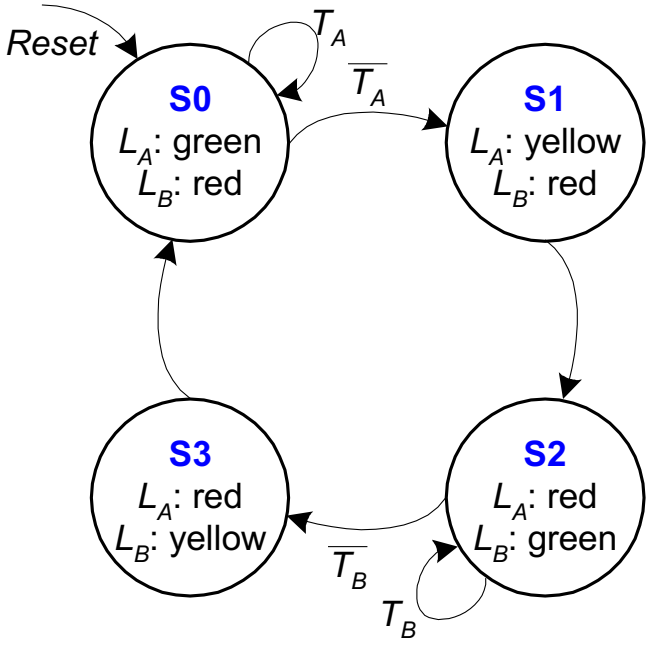
Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	
S0	1	X	
S1	X	X	
S2	X	0	
S2	X	1	
S3	X	X	

FSM State Transition Table



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

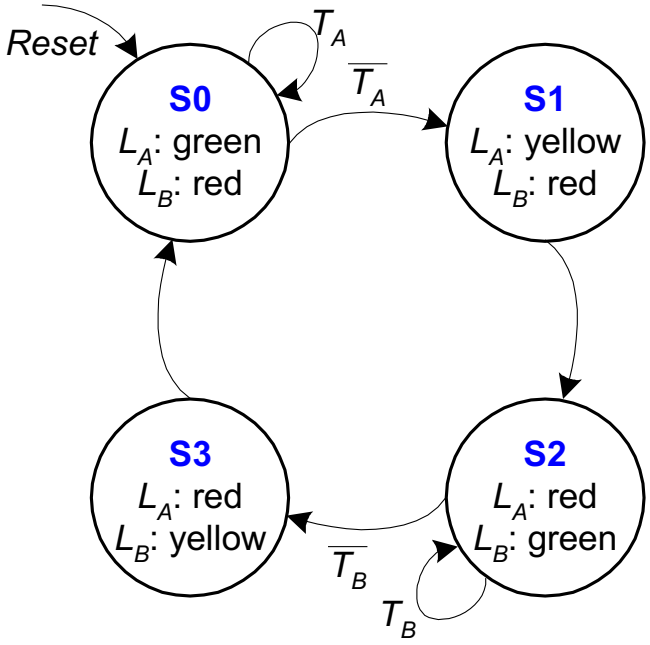
FSM State Transition Table



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

State	Encoding
S0	00
S1	01
S2	10
S3	11

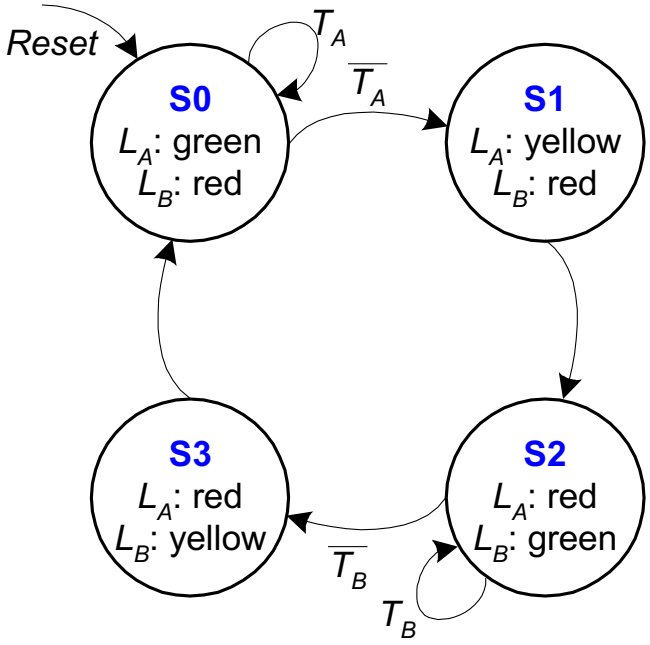
FSM State Transition Table



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

State	Encoding
S0	00
S1	01
S2	10
S3	11

FSM State Transition Table

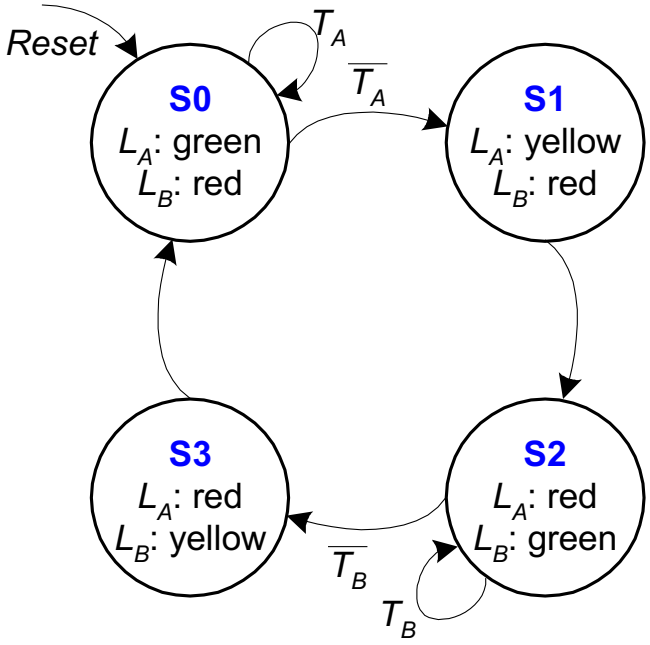


Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$S'_1 = ?$

FSM State Transition Table

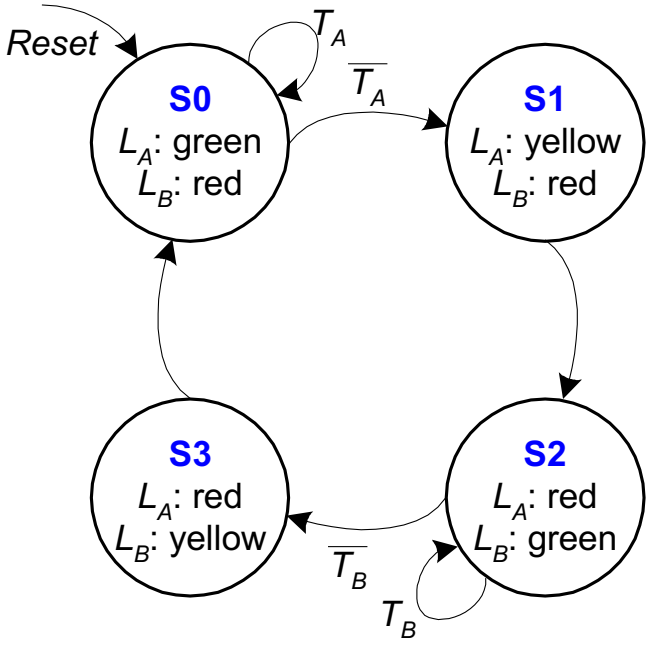


Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

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

FSM State Transition Table



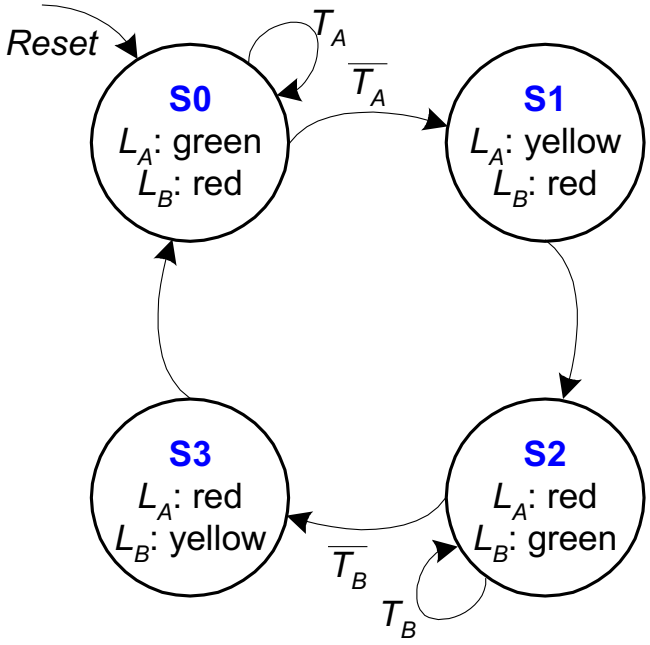
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

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

$S'_0 = ?$

FSM State Transition Table



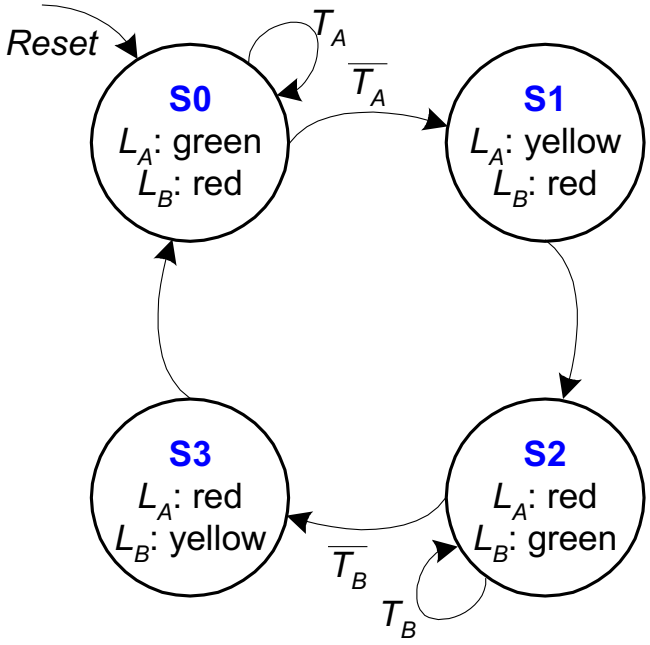
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

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

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

FSM State Transition Table



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

State	Encoding
S0	00
S1	01
S2	10
S3	11

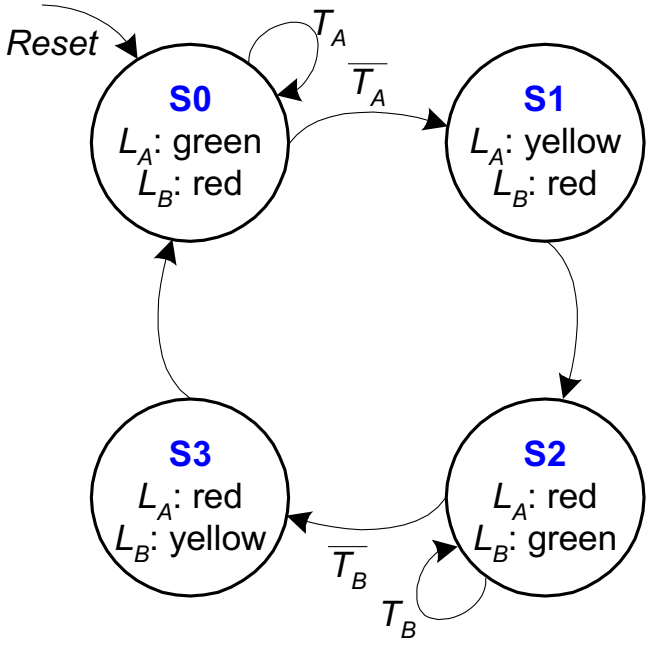
$$S'_1 = S_1 \text{ xor } S_0 \quad \textbf{(Simplified)}$$

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

Finite State Machine:

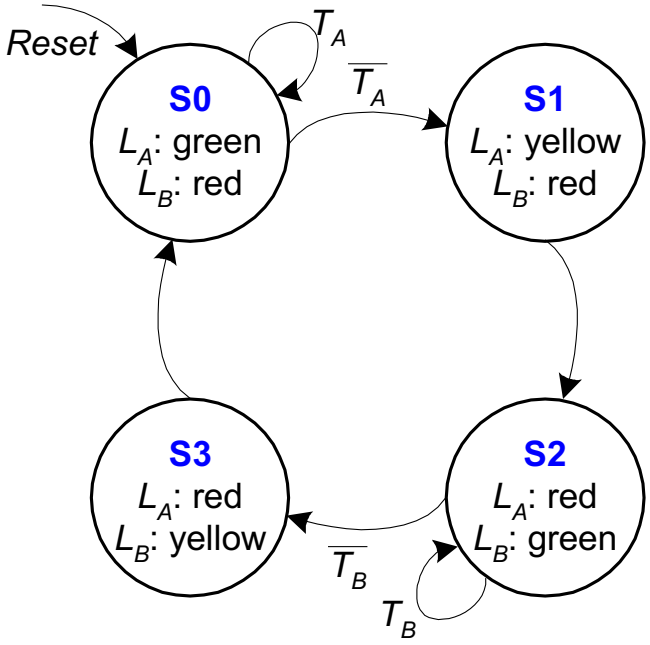
Output Table

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

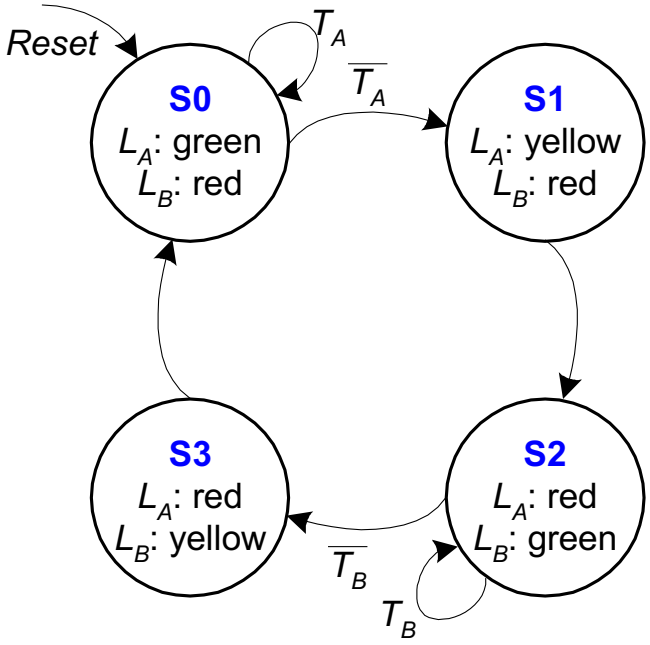
FSM Output Table



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

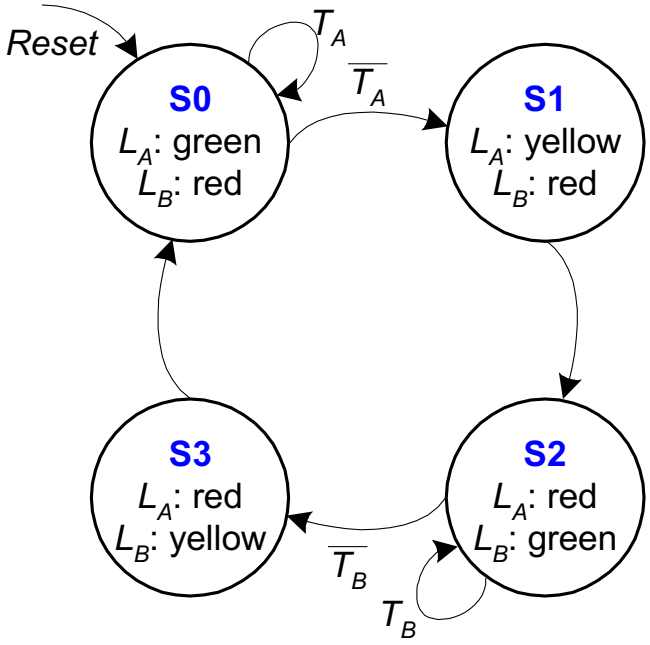


$L_{A1} = S_1$

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

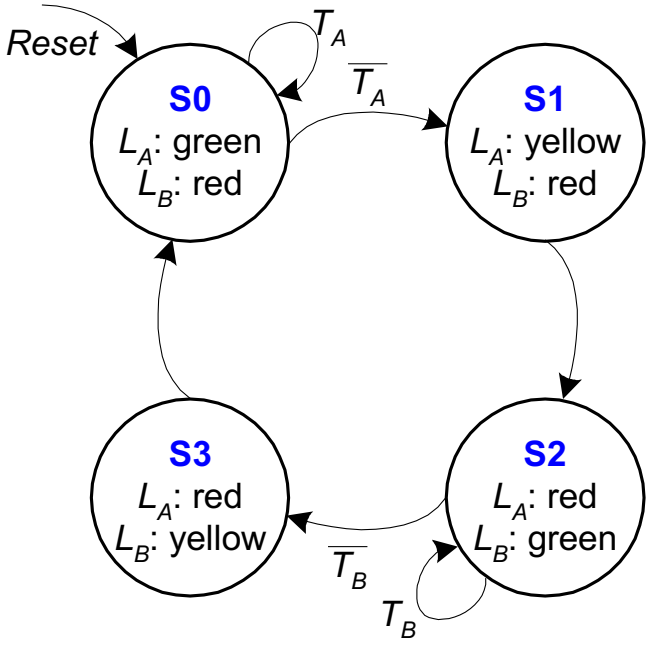


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

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

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table

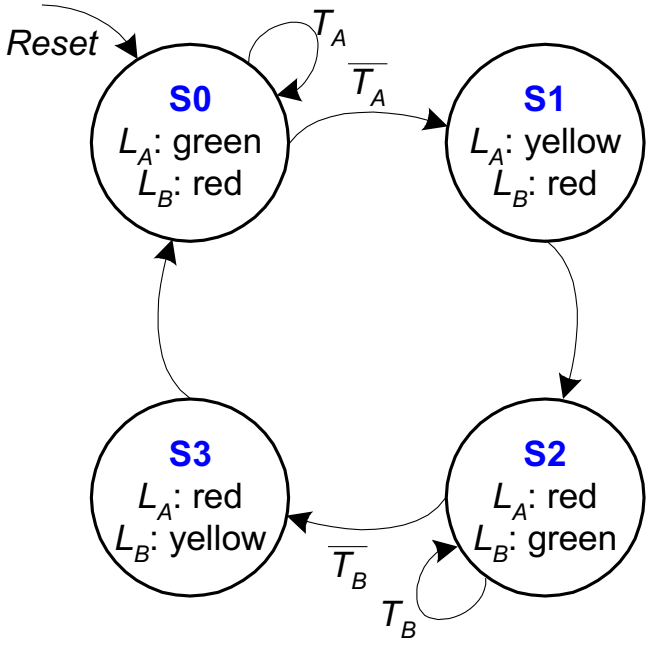


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

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

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table



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

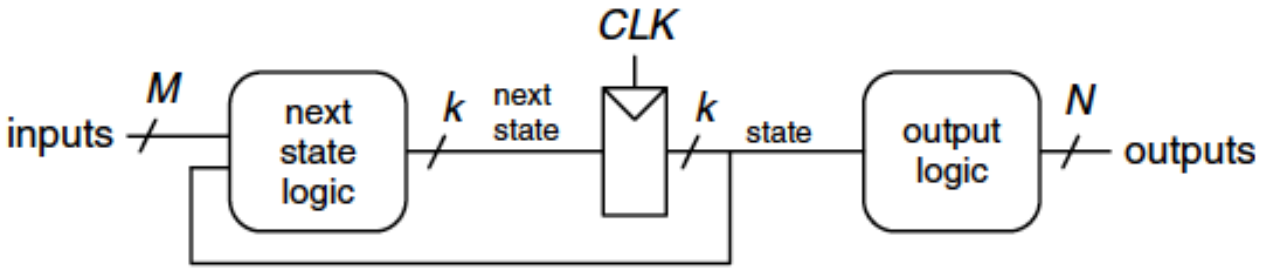
$$L_{A1} = \overline{S_1}$$
$$L_{A0} = \overline{S_1} \cdot S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 \cdot S_0$$

Output	Encoding
green	00
yellow	01
red	10

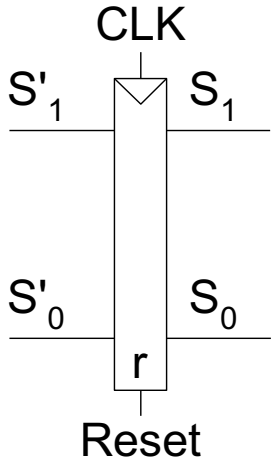
Finite State Machine:

Schematic

FSM Overview

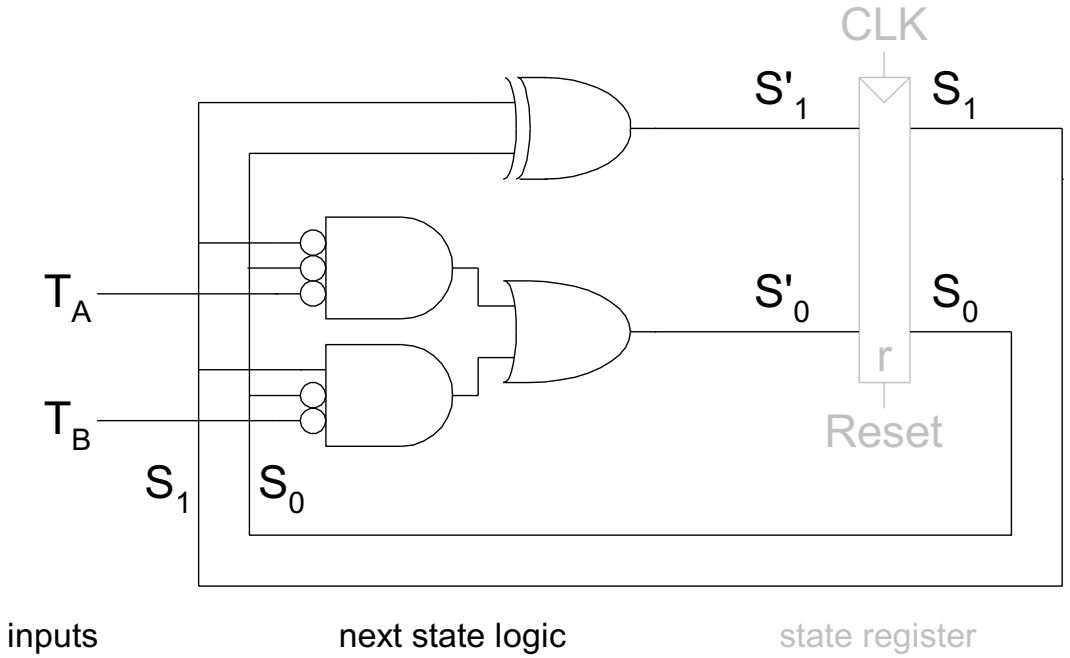


FSM Schematic: State Register



state register

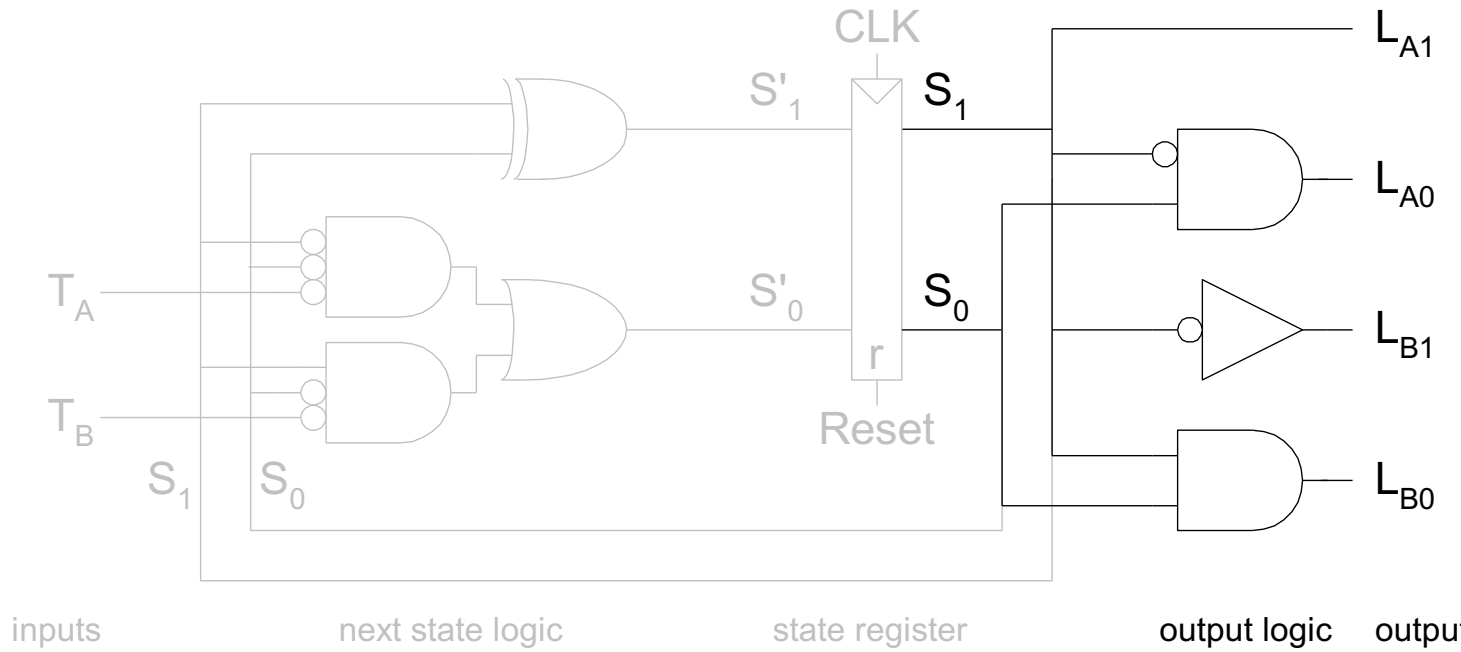
FSM Schematic: Next State Logic



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

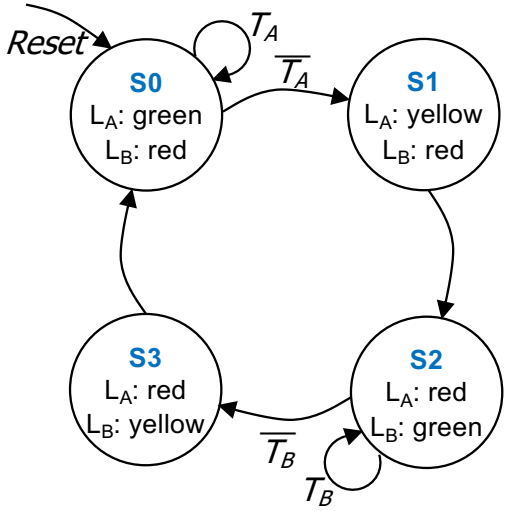
$$S'_0 = (\overline{S}_1 \cdot \overline{S}_0 \cdot \overline{T}_A) + (S_1 \cdot \overline{S}_0 \cdot \overline{T}_B)$$

FSM Schematic: Output Logic



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

FSM Timing Diagram



CLK

Reset

T_A

T_B

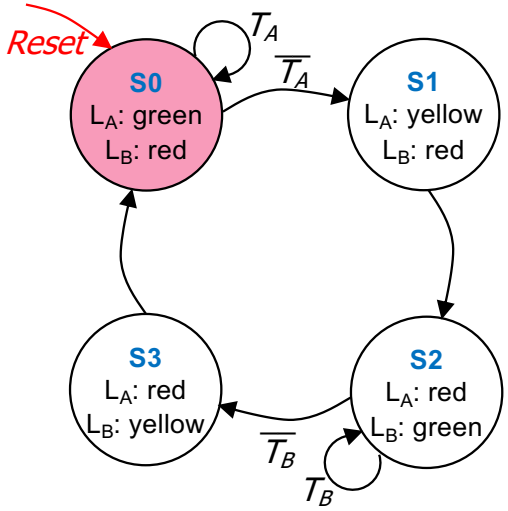
$S'_{1:0}$

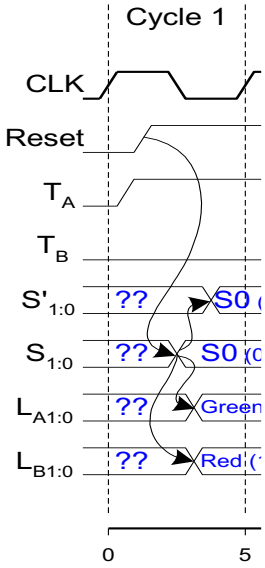
$S_{1:0}$

$L_{A1:0}$

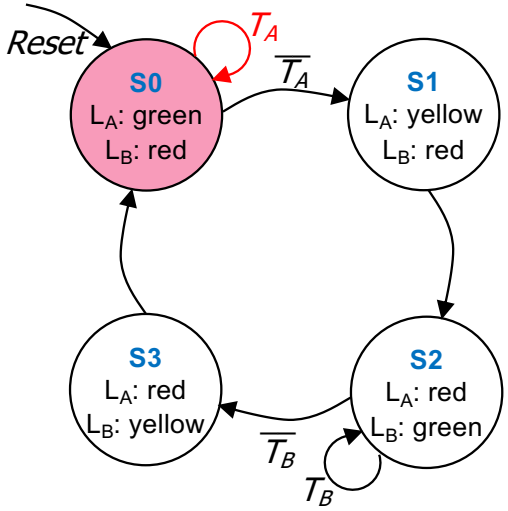
$L_{B1:0}$

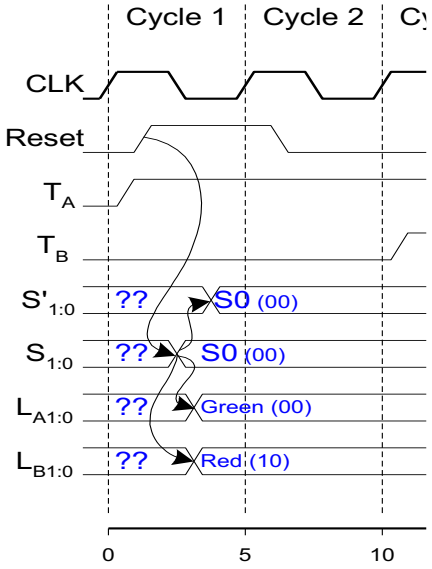
FSM Timing Diagram



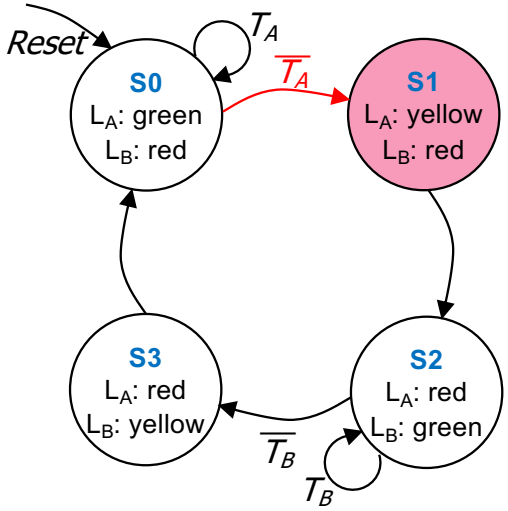


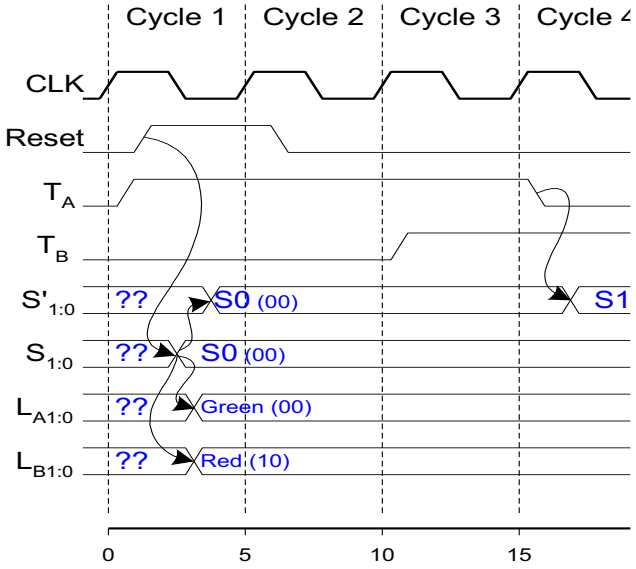
FSM Timing Diagram



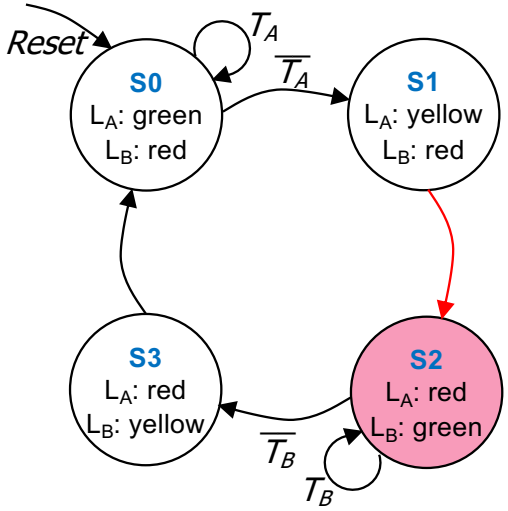


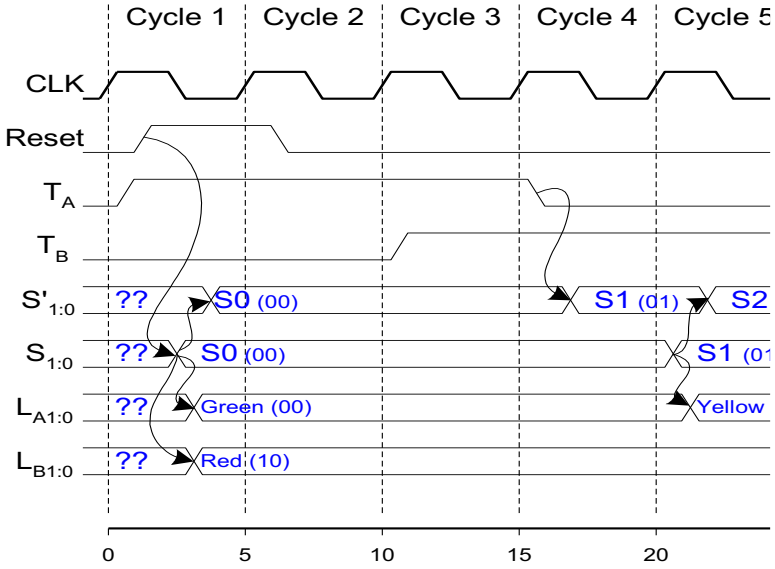
FSM Timing Diagram



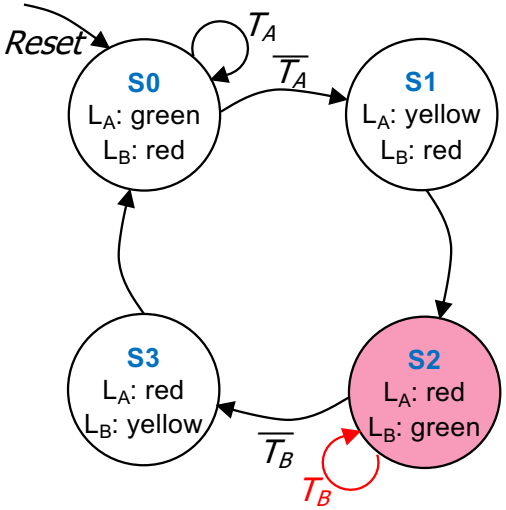


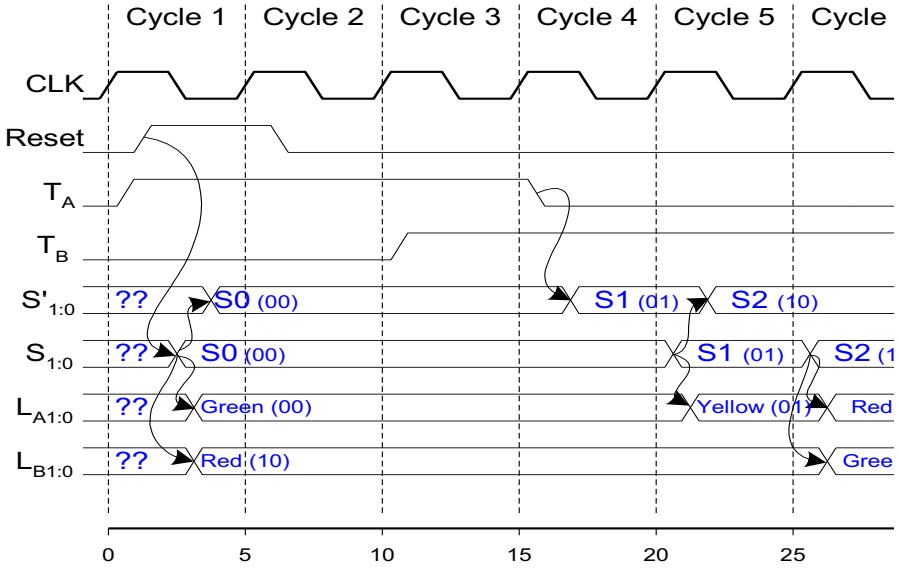
FSM Timing Diagram



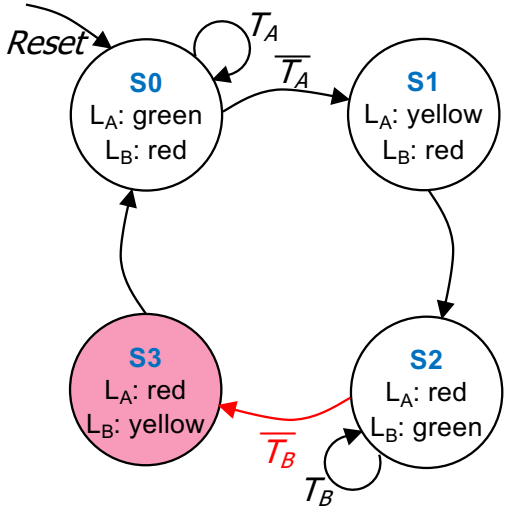


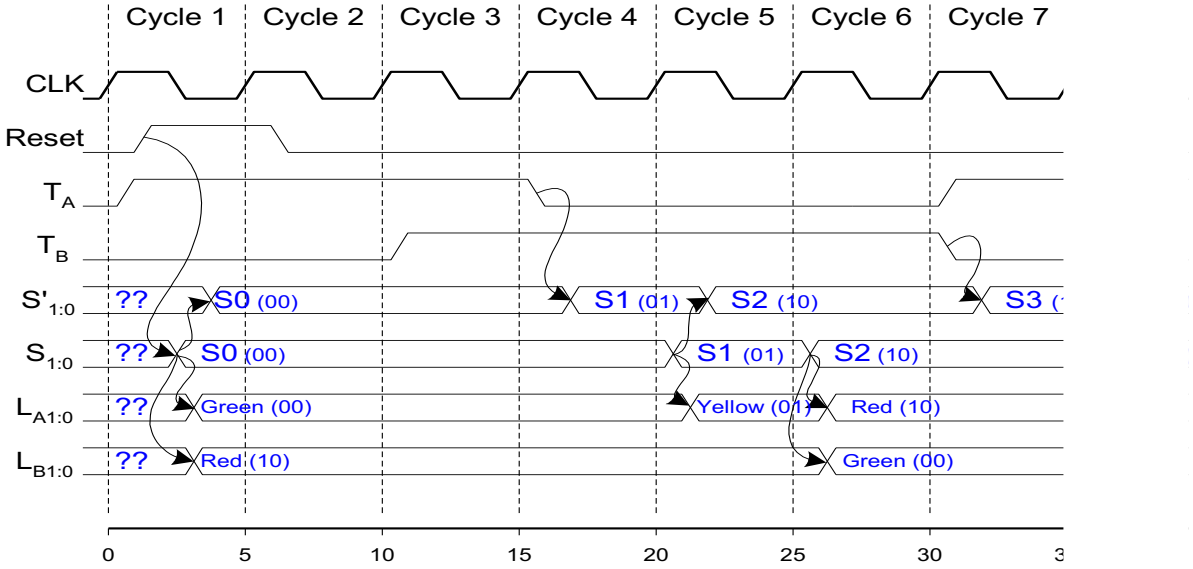
FSM Timing Diagram



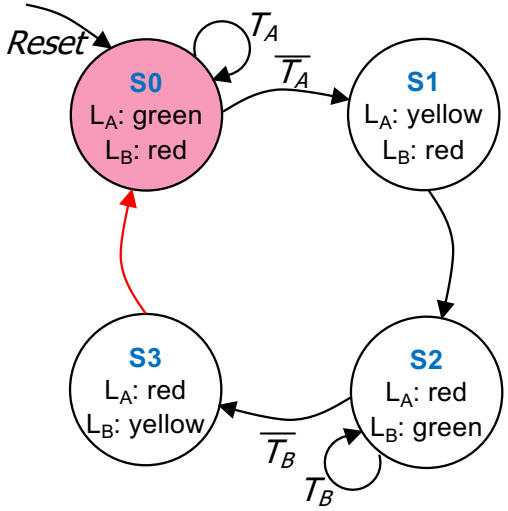


FSM Timing Diagram

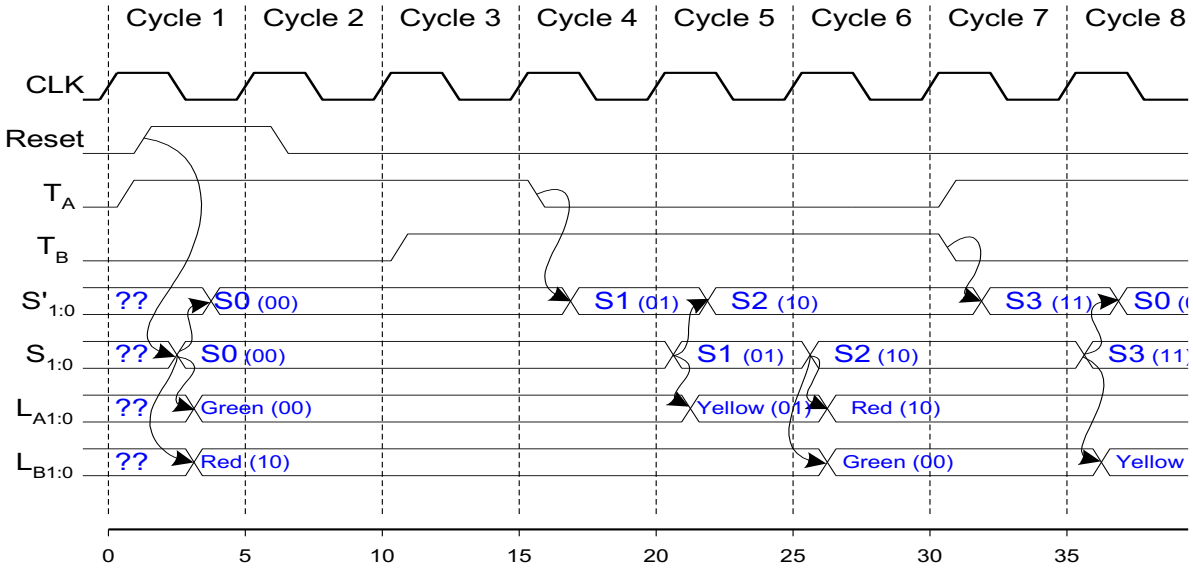




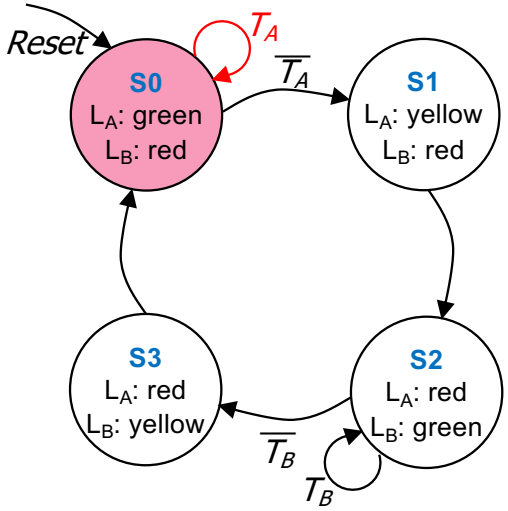
FSM Timing Diagram

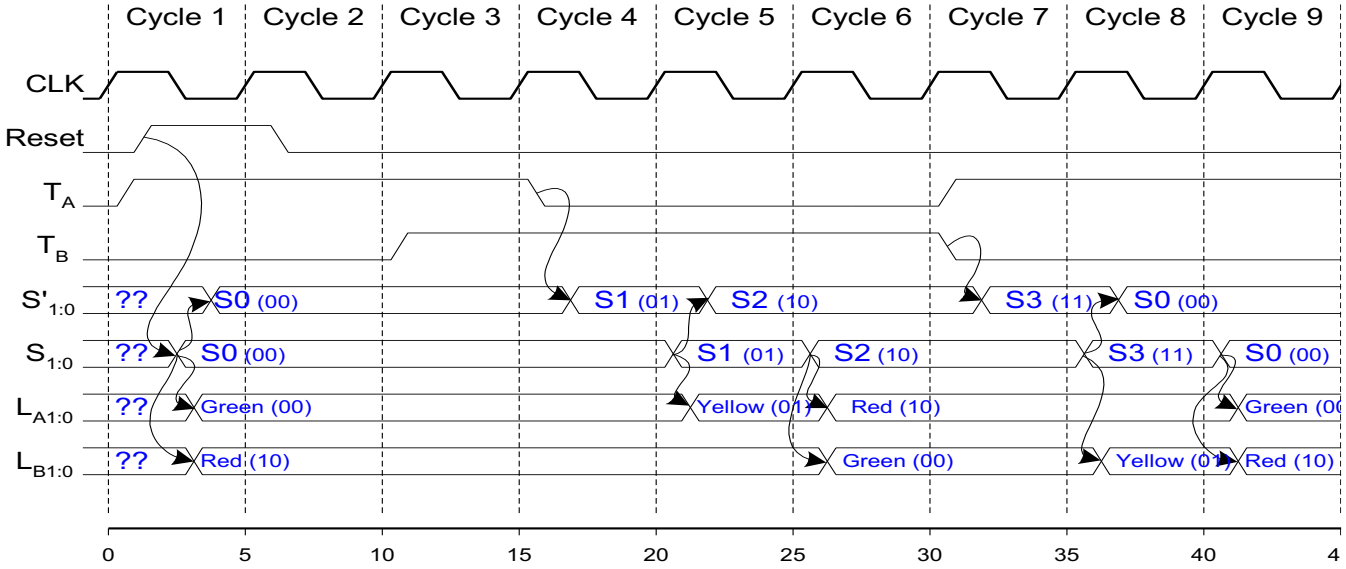


This is from H&H Section 3.4.1

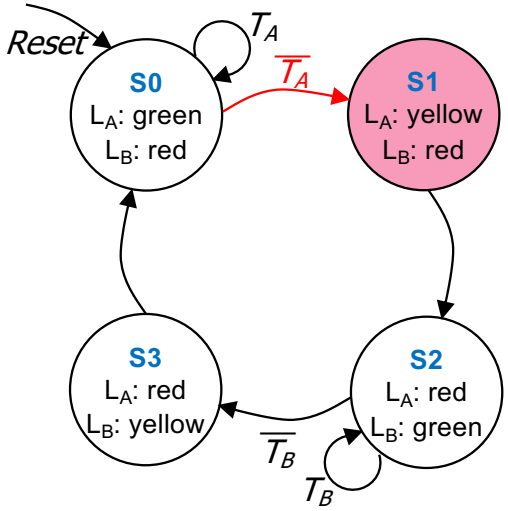


FSM Timing Diagram

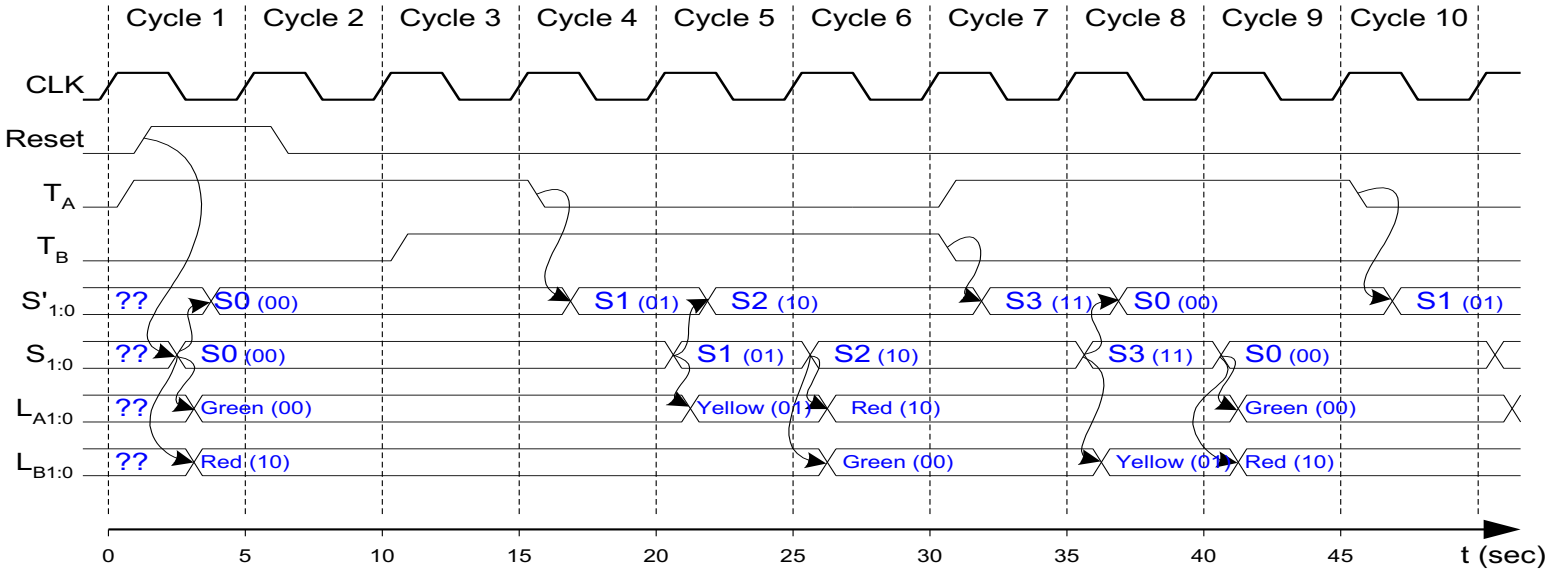




FSM Timing Diagram



See H&H Chapter 3.4



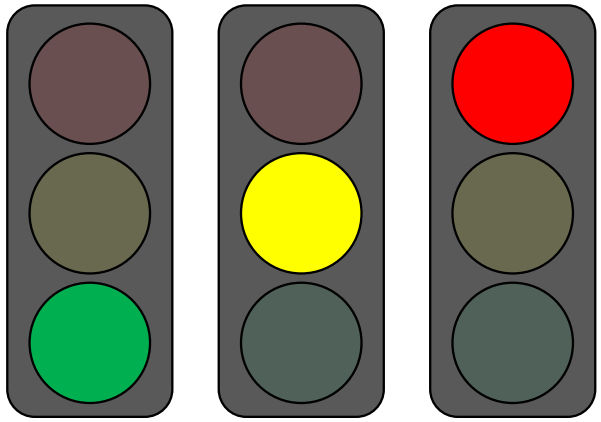
Finite State Machine:

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**
    - **Output 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
   - *Examp*
   - **Minim** logic
2. **One-H**
   - Each b
     - Use
     - Exa
   - *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

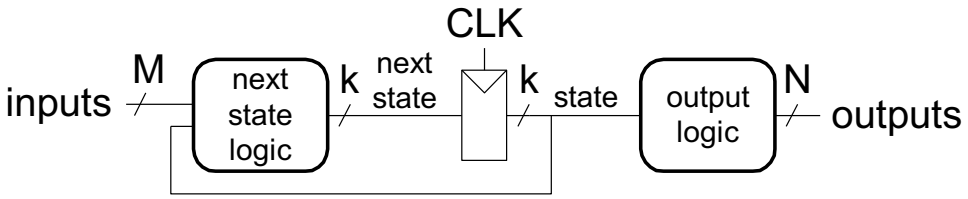
Moore vs. Mealy Machines

Section 3.4.3 of H&H

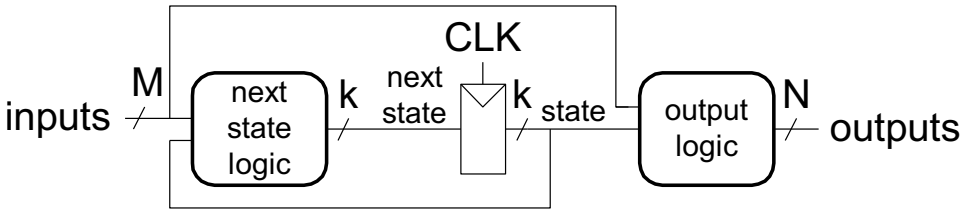
Recall: Moore vs. Mealy Machines

- 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



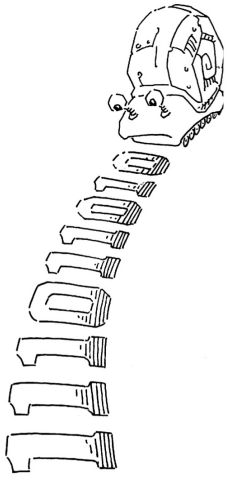
Mealy FSM



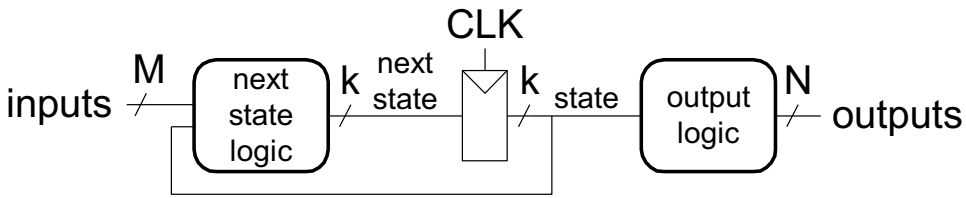
Section 3.4.3 of H&H

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.



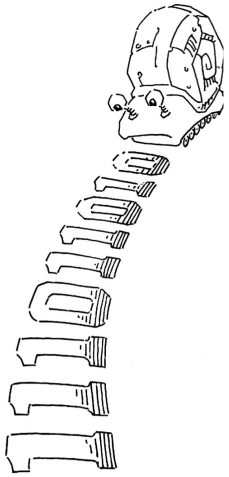
Moore FSM



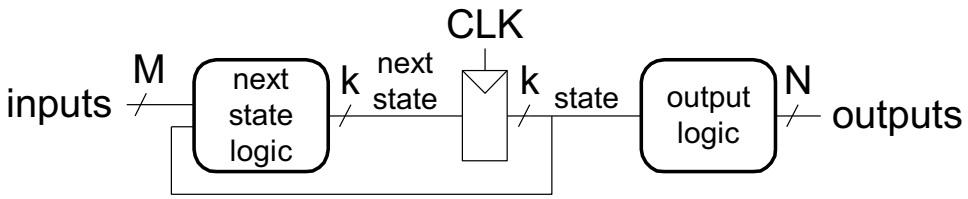
Example 3.7 of H&H

Moore vs. Mealy Examples

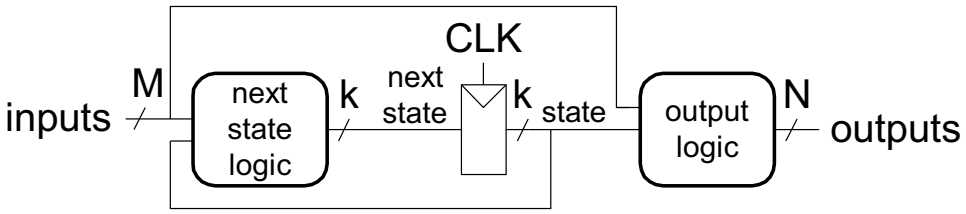
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are 1101.
- Design Moore and Mealy FSMs of the snail's brain.



Moore FSM

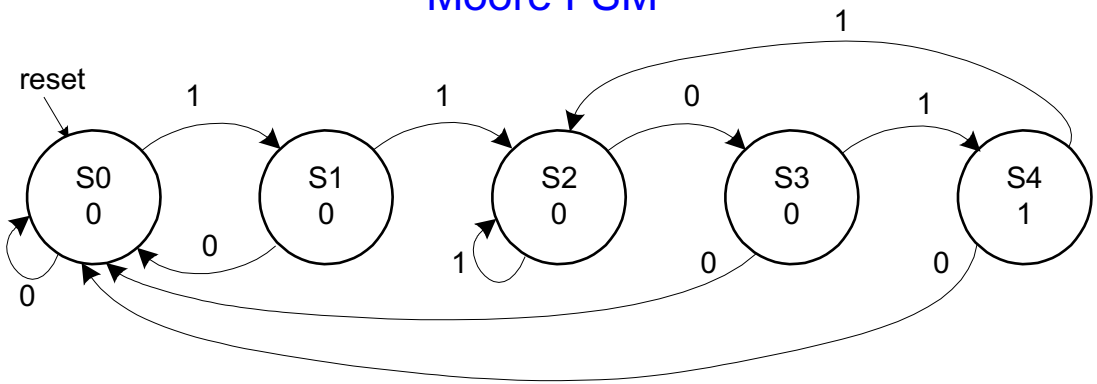


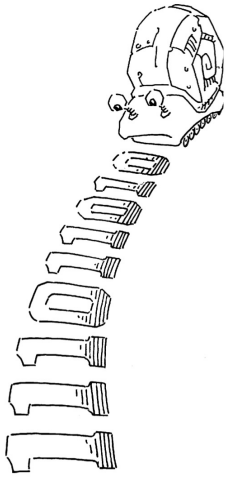
Mealy FSM



State Transition Diagrams

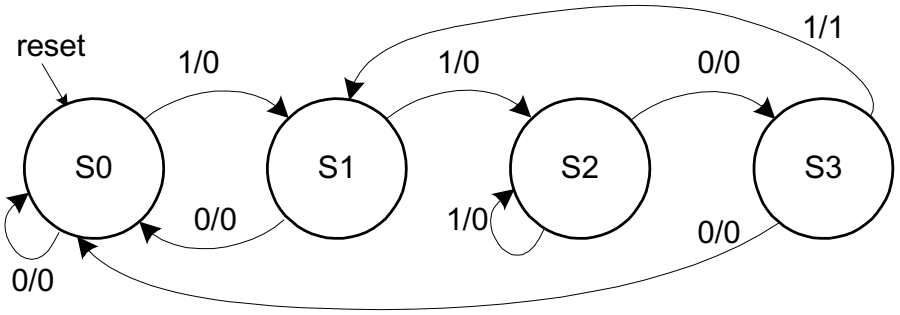
Moore FSM





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

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

Why are Arbitrary Sequential Circuits a Bad Idea?

Reading: Section **3.4** of H&H

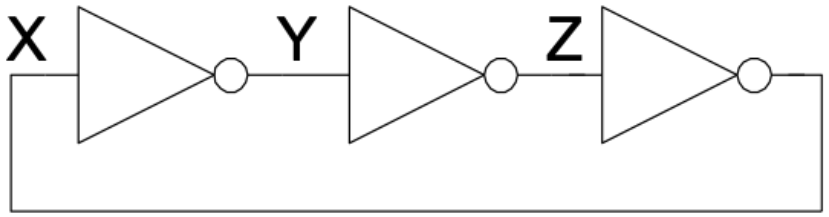
Arbitrary Sequential Circuits

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

Section 3.3 of H&H

Ring Oscillator

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



Assume X = 0
- Y = 1, Z = 0, X = 1
- Inconsistent with original assumption
- **Oscillates** between 0 and 1

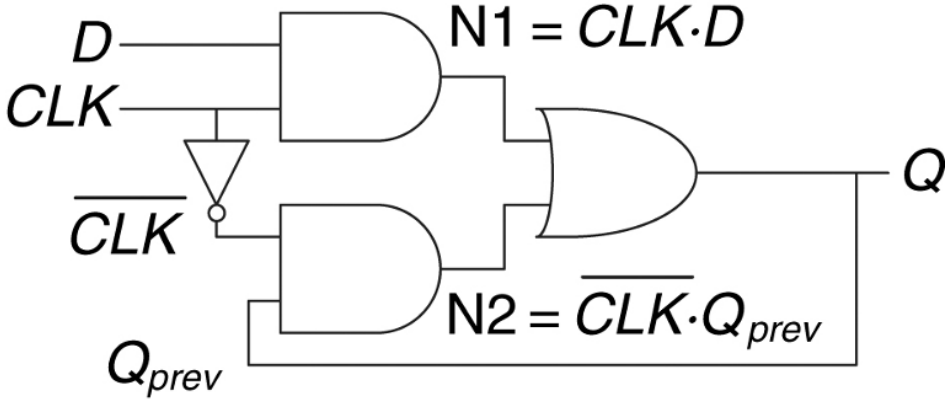
- No stable state (**Unstable** or **Astable**)
- X **oscillates** b/w **0** and **1**
- If the propagation delay of each inverter is **1 ns**, then can you plot X? What is the clock period?
- What is the period (repetition time)?

Example 3.3 of H&H

Cooked-up D Latch Optional Self-Study

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

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



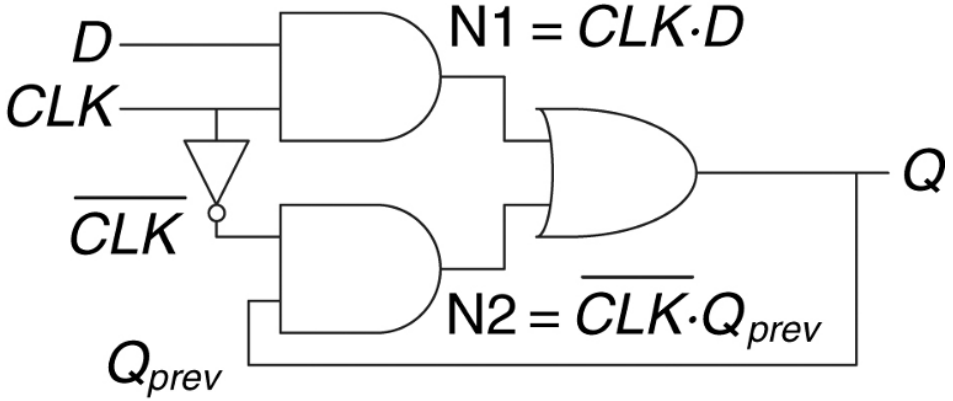
Example 3.4 (page 119) of H&H

Cooked-up D Latch Optional Self-Study

- 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

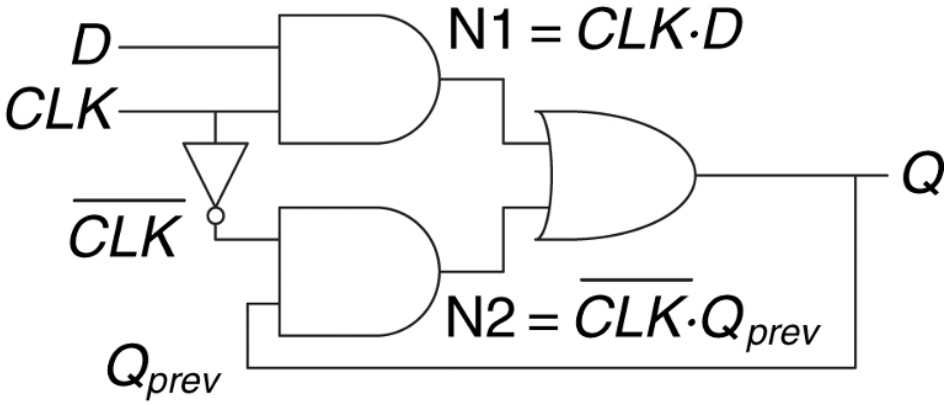
$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$



Example 3.4 (page 119) of H&H

Cooked-up D Latch Optional Self-Study

- What if $t_{INV} >> t_{AND}$ and $t_{INV} >> t_{OR}$?
  - Node N1 and Q may both fall before $\overline{CLK}$ changes/rises
  - Q gets stuck at 0
  - This is known as a *race condition: The path through CLK to Q is faster than $\overline{CLK}$ to Q*



Example 3.4 (page 119) of H&H

Takeaways

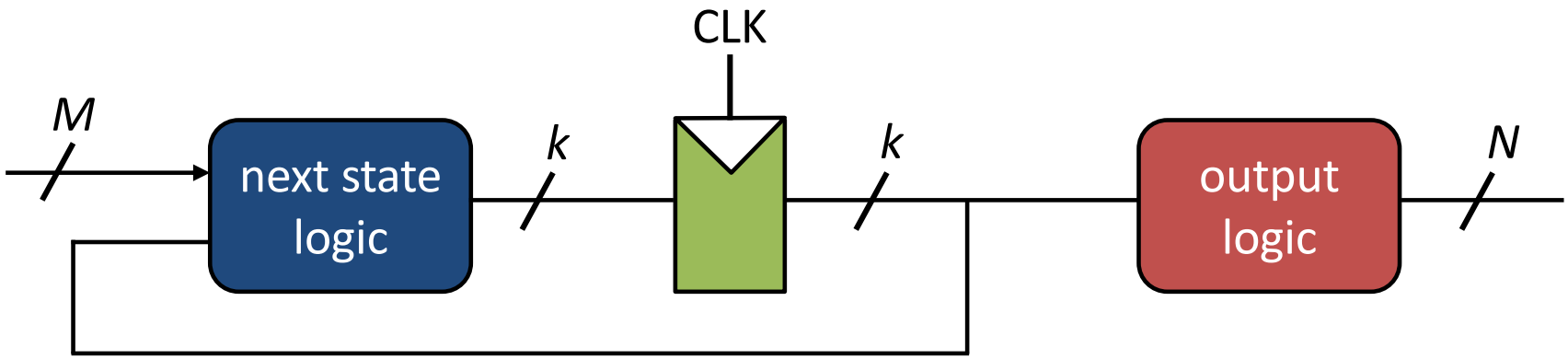
- When outputs are fed back directly back to inputs, these are called **cyclic** paths

- 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 (e.g., temperature) but not in another

Synchronous Sequential Circuits

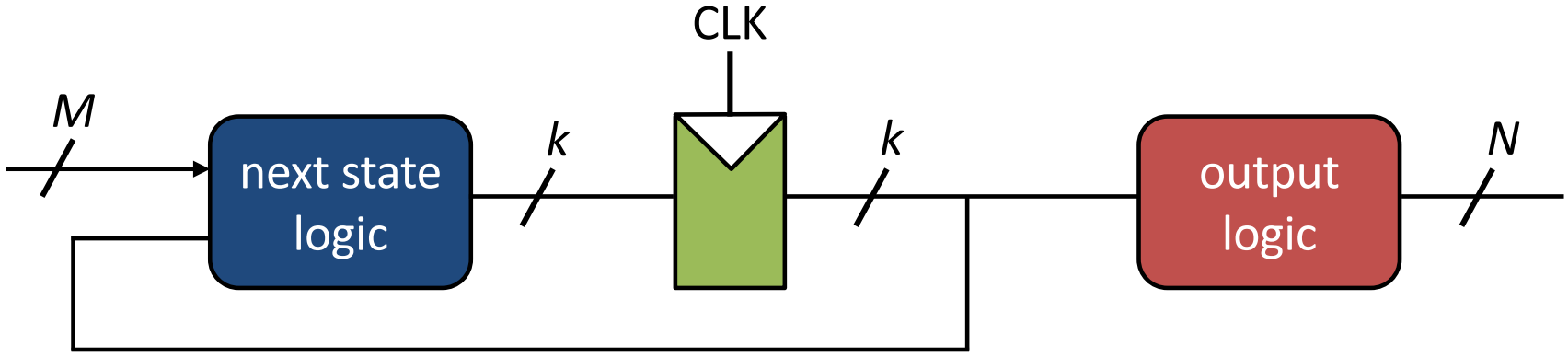
- What is the problem with asynchronous circuits?
  - *Cyclic paths lead to races and unstable behavior*
- **Solution:** Break the cyclic paths by **inserting** registers somewhere in the path
- Registers contain **state**, and **synchronized** to the clock



Section 3.3.2 of H&H

Synchronous Sequential Circuits

- 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

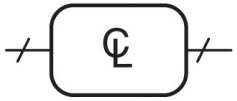


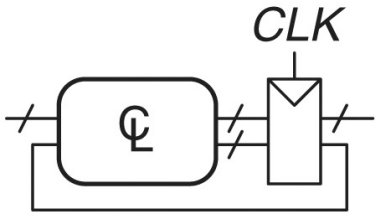
General Rule: *If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated*

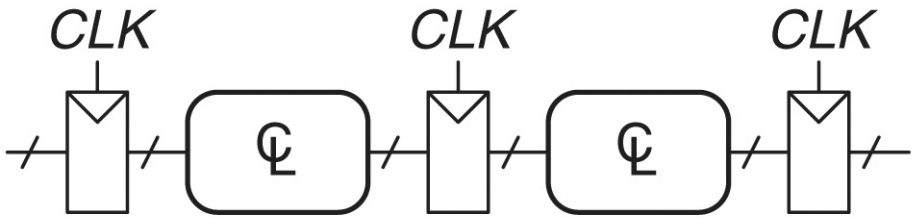
Composition Rules

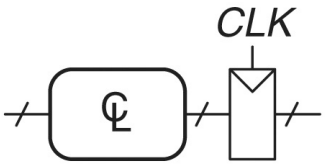
- Every circuit element is either a register or a combinational element
- At least one element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register

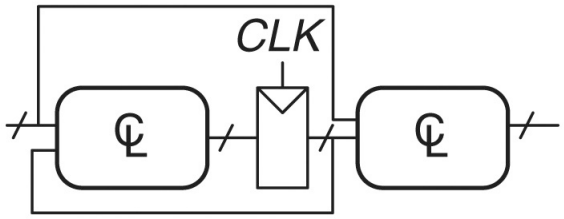
Which circuits are synchronous sequential?

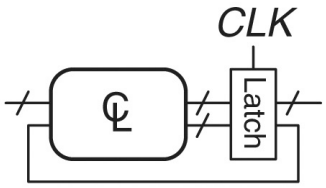


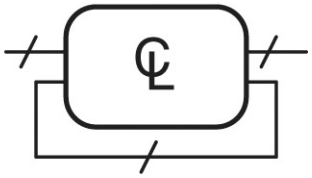


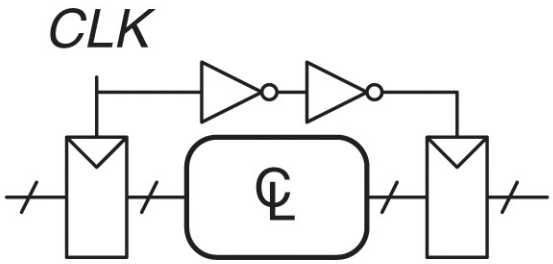












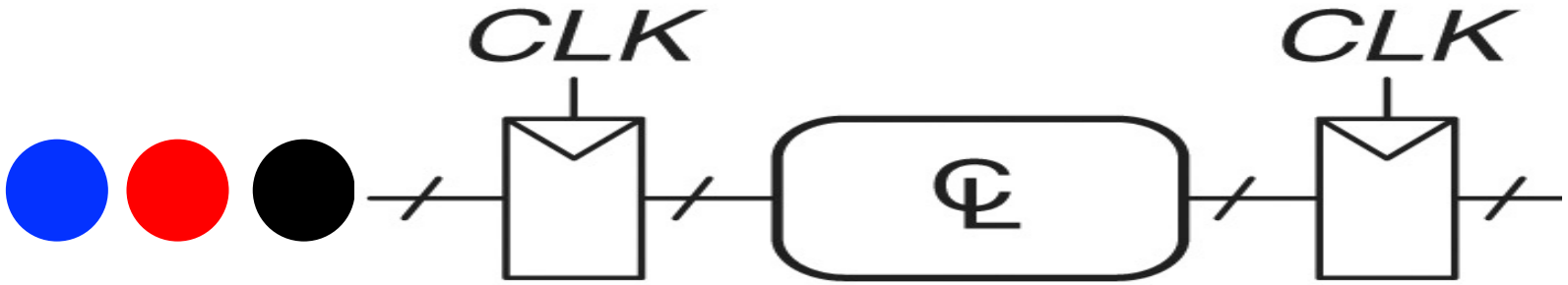
Example 3.5 of H&H

General Idea of

Pipelining

Another Synchronous Sequential Circuit Example: Pipelining

- Each circle is a **task** to process
  - Consider a task as a group of inputs processed by combinational logic

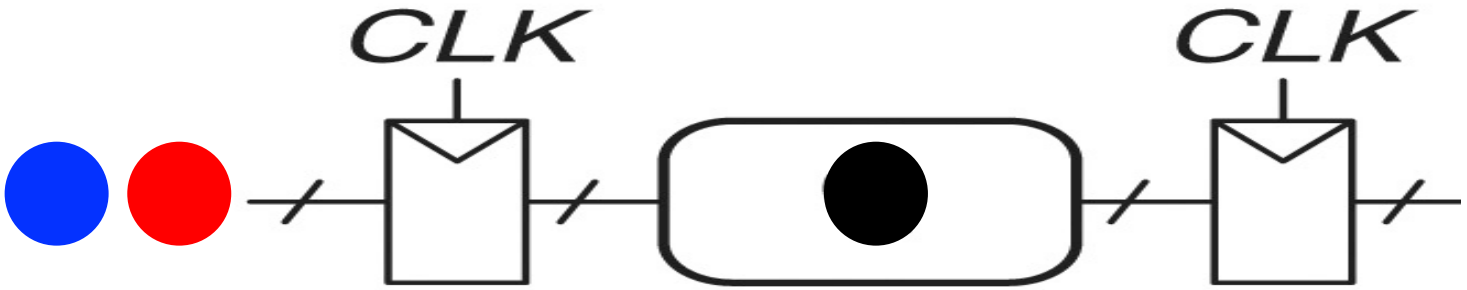


- Three clock cycles to process the three input groups
- One task per clock cycle

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

- Each circle is a **task** to process
  - Consider a task as a group of inputs processed by combinational logic

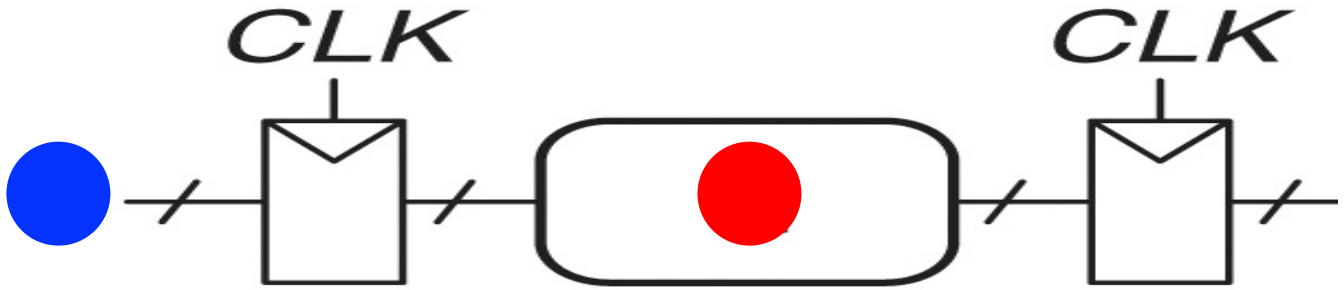


- Three clock cycles to process the three input groups
- One task per clock cycle

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

- Each circle is a **task** to process
  - Consider a task as a group of inputs processed by combinational logic

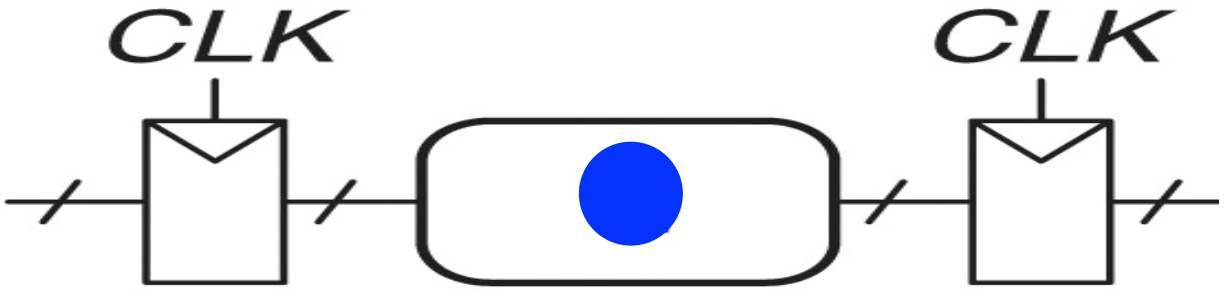


- Three clock cycles to process the three input groups
- One task per clock cycle

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

- Each circle is a **task** to process
  - Consider a task as a group of inputs processed by combinational logic

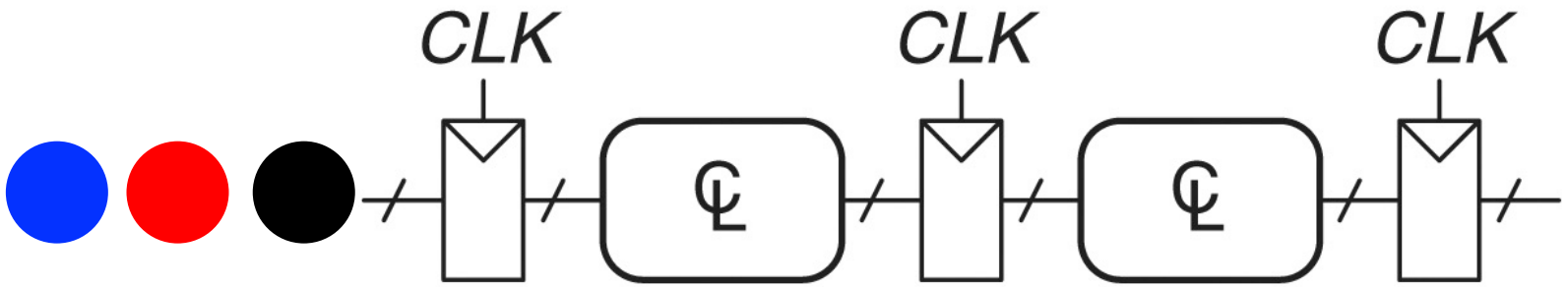


- Three clock cycles to process the three input groups
- One task per clock cycle

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

- **Pipelining:** Break the task into multiple stages
  - Each stage performs a **sub-set** of the task

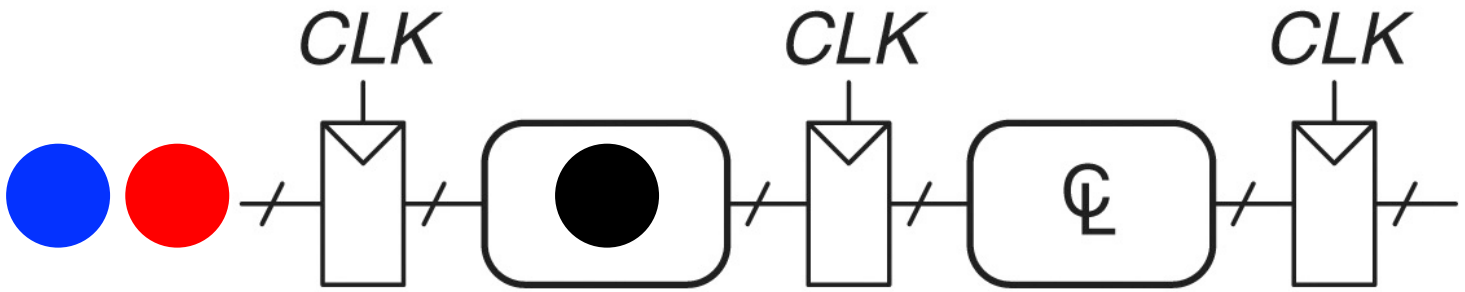


- More clock cycles to process the three input groups
- But combinational logic is less complex. We can **shorten** the clock cycle time

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

- **Cycle 1**
  - Start processing black token, the other tokens wait



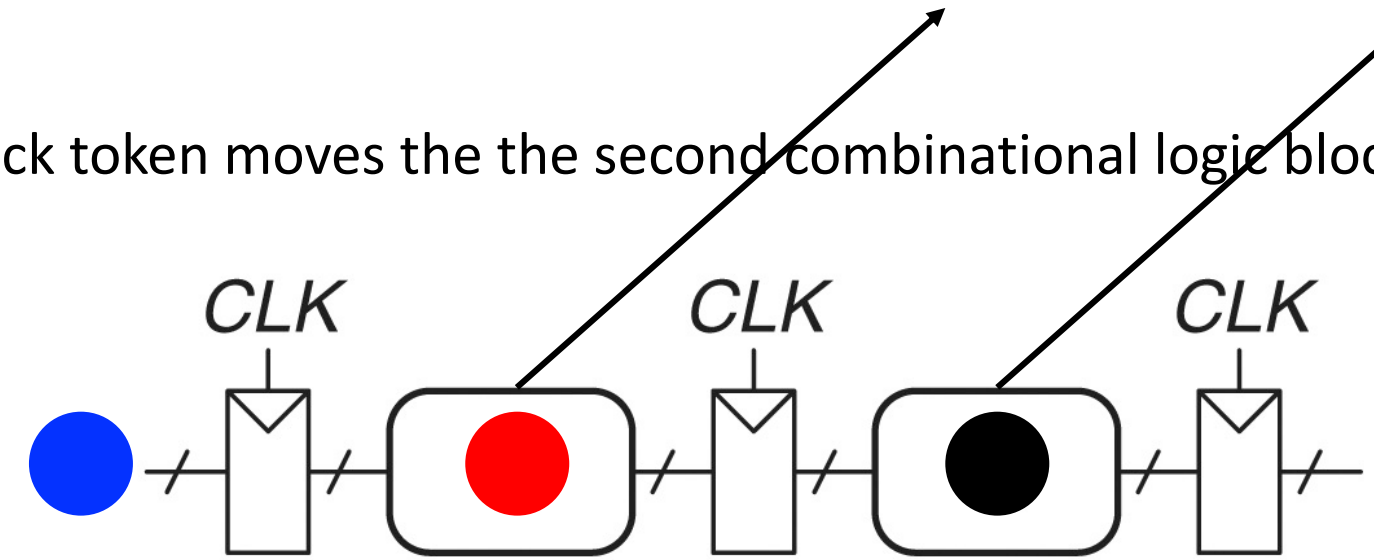
- The first combinational circuit performs the sub-set of task (processing)

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

Parallelism in time

- Cycle 2
  - Black token moves the the second combinational logic block

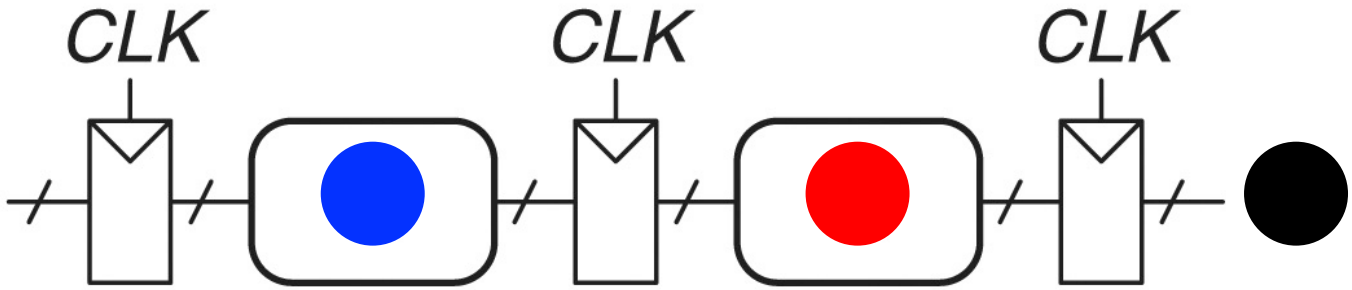


- The first combinational circuit starts processing the red task (token)

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

- **Cycle 3**
  - Black token has been processed

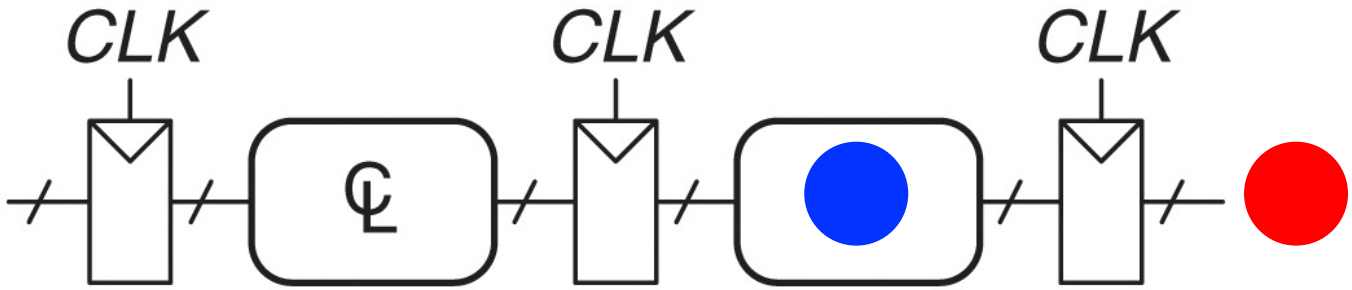


- Blue token is being processed in first block, and red in second block
  - We have **parallelism**!

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

- **Cycle 4**
  - Red token has been processed

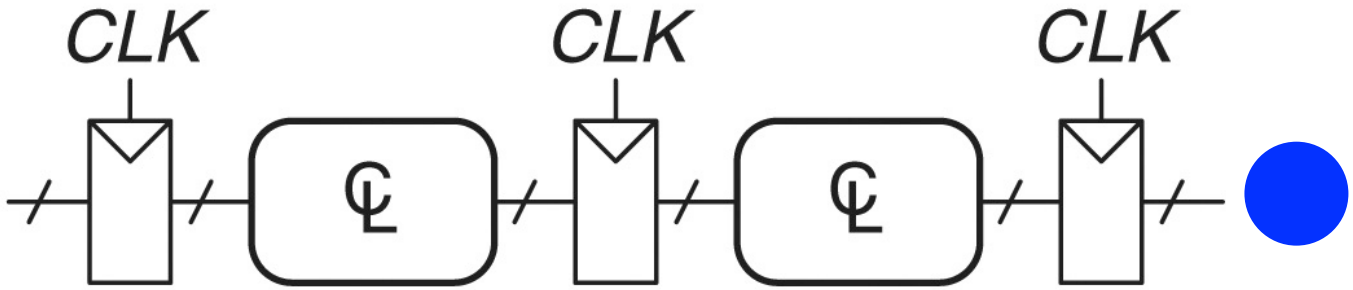


- Blue token is being processed in first block, and red in second block
  - We have **parallelism**!

Example 3.5 of H&H

Another Synchronous Sequential Circuit Example: Pipelining

- **Cycle 5**
  - Blue token has been processed

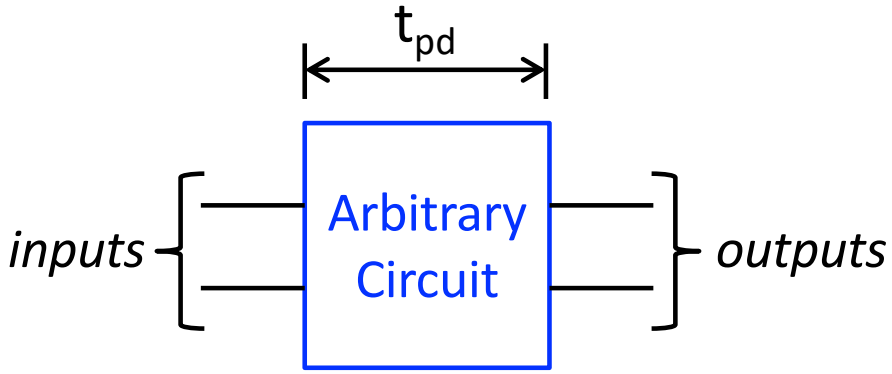


- Blue token is being processed in first block, and red in second block
  - We have **parallelism**!

Example 3.5 of H&H

Speed of a Circuit

- At a high level, an arbitrary digital circuit processes a group of inputs and produces a group of outputs



- We needs **metrics** to quantify the speed with which we can process inputs to produce outputs (i.e., the performance of a circuit)
  - **Latency:** The time required to produce one group of outputs once the inputs arrive (propagation delay, end-to-end latency)
  - **Throughput:** The number of input groups processed per unit of time

Example: Latency/Throughput

- What is the latency and throughput for a tray of cookies?

  - Step 1: **Roll** cookies (5 minutes)
  - Step 2: **Bake** in the oven (15 minutes)
    - Once cookies are baked, start another tray

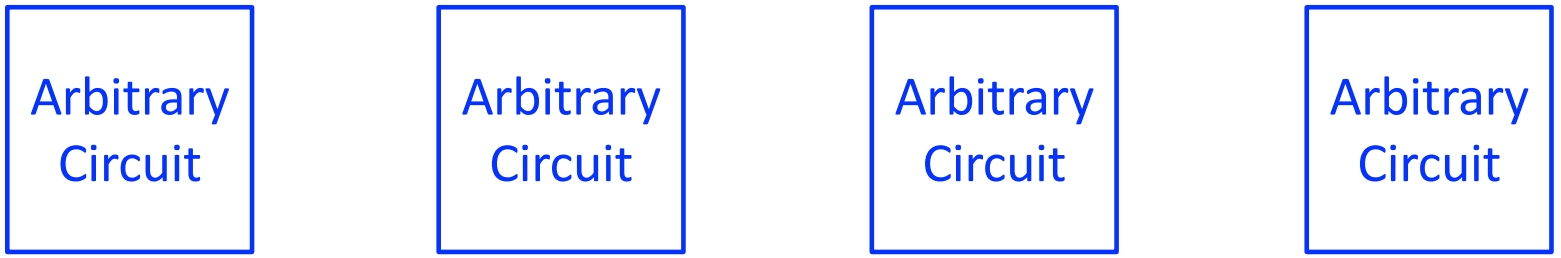
- Latency (hours/tray):
- Throughput (trays/hour):

Parallelism

- Many scenarios in the real-world requires us to increase the throughput of the digital system
  - # add operations per second (ALU)
  - # instructions per second (CPU)
- **Parallelism** is a key technique for increasing throughput and processing several inputs at the same time

Spatial Parallelism

- **Spatial Parallelism:** Use multiple copies of hardware (circuit) to get multiple tasks done at the same time



- Suppose a task has a latency of L seconds
  - **No spatial parallelism:** Throughput is 1/L (one task per L seconds)
  - **N copies of hardware:** Throughput is N/L (N tasks per L seconds)
  - Gain in throughput (speedup) = N

Spatial Parallelism does not reduce the latency of the circuit. We can finish more tasks per unit of time. But each task still takes L seconds

Temporal Parallelism

- **Temporal Parallelism (pipelining):**
  - Break down a circuit into stages
  - Each task passes through all stages
  - Multiple tasks are spread through stages

Automotive Pipeline

Pipelining

- If a task of latency L is broken into N stages, and all stages are of equal length, then the throughput is N/L

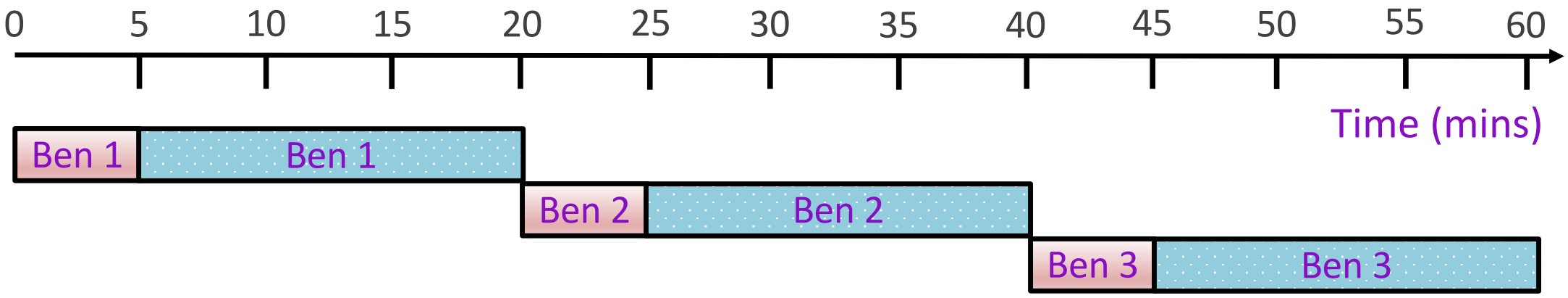
- The challenge of pipelining is to find stages of equal length

- Let's go back to baking cookies

Cookie Parallelism

- Ben and Jon are making cookies. Let's study the latency and throughput of rolling and baking many cookie trays with
  - No parallelism
  - Spatial parallelism
  - Pipelining
  - Spatial parallelism + pipelining

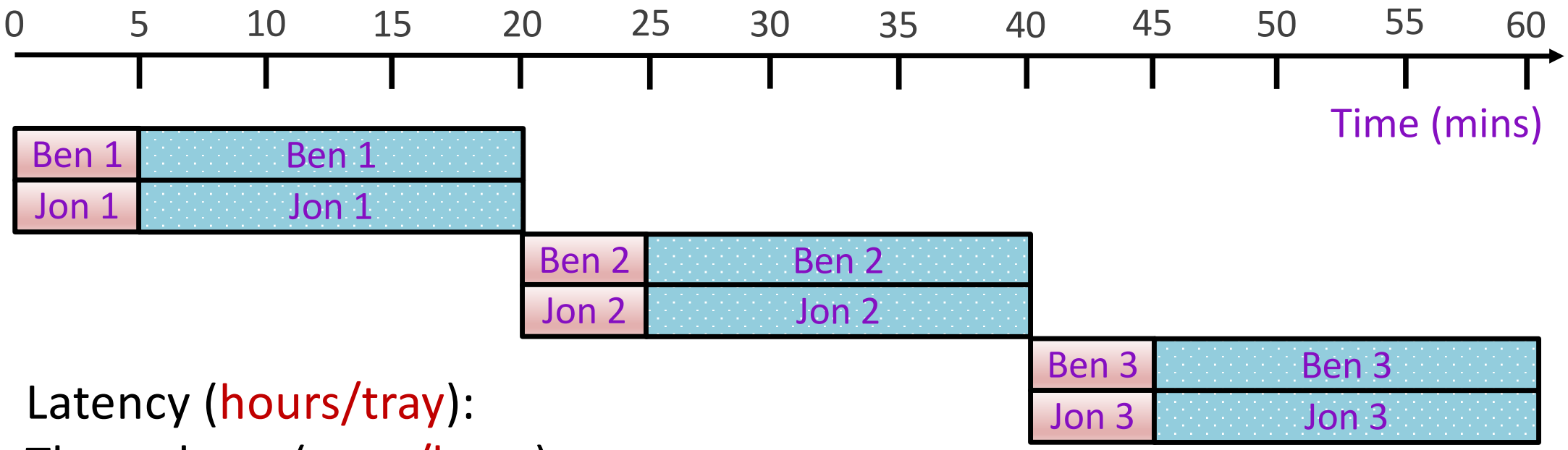
No Parallelism (Ben Only)



Latency (hours/tray):

Throughput (trays/hour):

Spatial Parallelism (Ben & Jon)

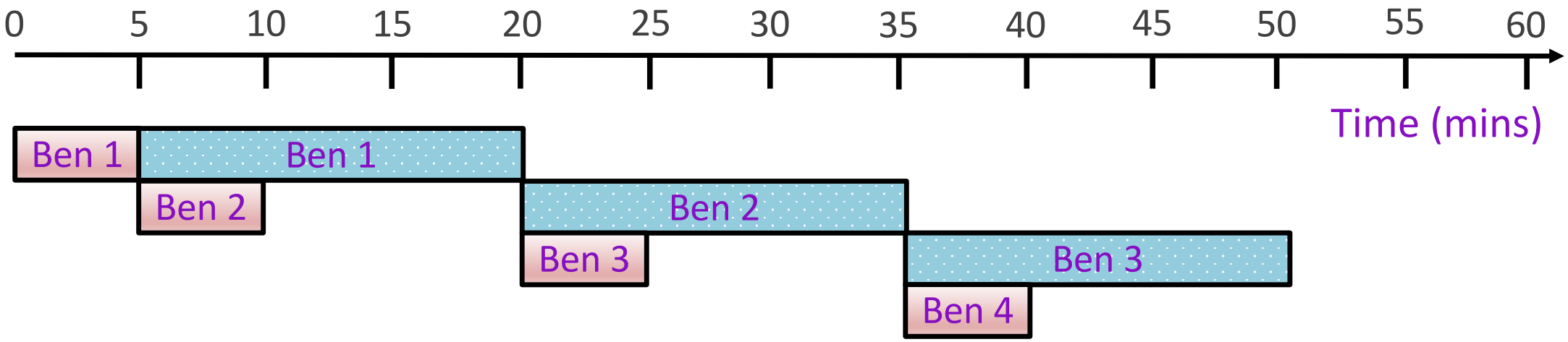


Latency (hours/tray):

Throughput (trays/hour):

Note: Jon owns a tray and oven (hardware duplication)

Pipelining (Ben Only)

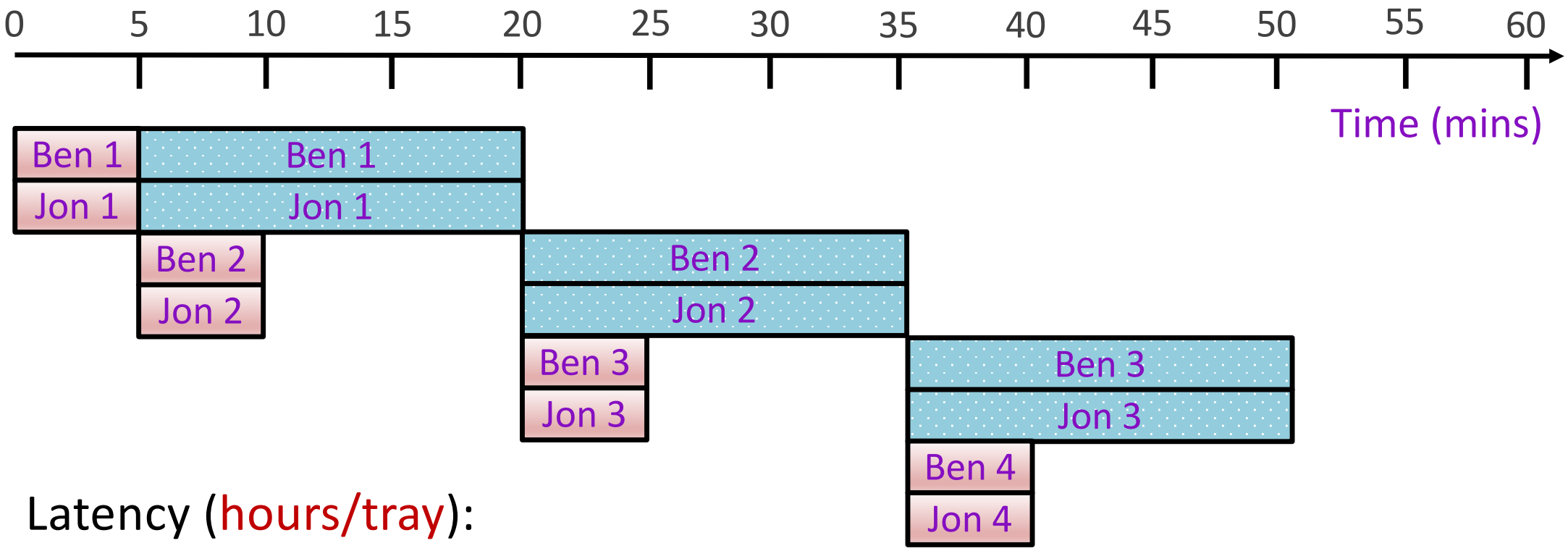


Latency (hours/tray):

Throughput (trays/hour):

Note: Ben decides not to waste a separate tray and oven

Spatial + Temporal Parallelism



Latency (hours/tray):

Throughput (trays/hour):

Answers Explained

- **No parallelism**
  - Latency is clearly 20 minutes (1/3 hours/tray)
  - Throughput is 3 trays per hour
- **Spatial parallelism**
  - Latency remains unchanged as it still takes 20 mins to finish a tray
  - Throughput is doubled via duplication: 6 trays per hour
- **Pipelining**
  - Latency for a single tray remains unchanged
  - Throughput: Ben puts a new tray in the oven every 15 minutes, so the throughput is 4 trays per hour
  - Note that in the first hour, Ben loses 5 minutes to fill the pipeline
- **Spatial parallelism + pipelining**
  - Latency remains unchanged
  - Throughput: Ben & Jon combo puts two trays in the oven every 15 minutes, so the throughput is 8 trays per hour

Pipelining Circuits

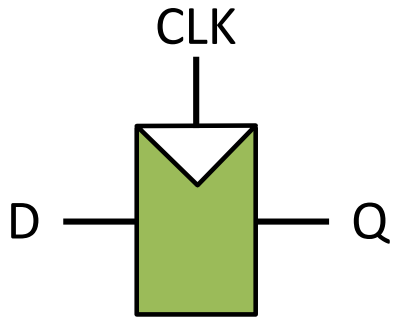
- Divide a large combinational circuit into shorter stages
- Insert registers between the stages
  - The outputs of one stage are copied into a register and communicated to the next stage
- Run the **pipelined** circuit at a **higher** clock frequency
  - Each clock cycle, data flows through the pipeline from left to the right
  - Multiple tasks can be spread across the pipeline

Timing Issues in Sequential Circuits

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

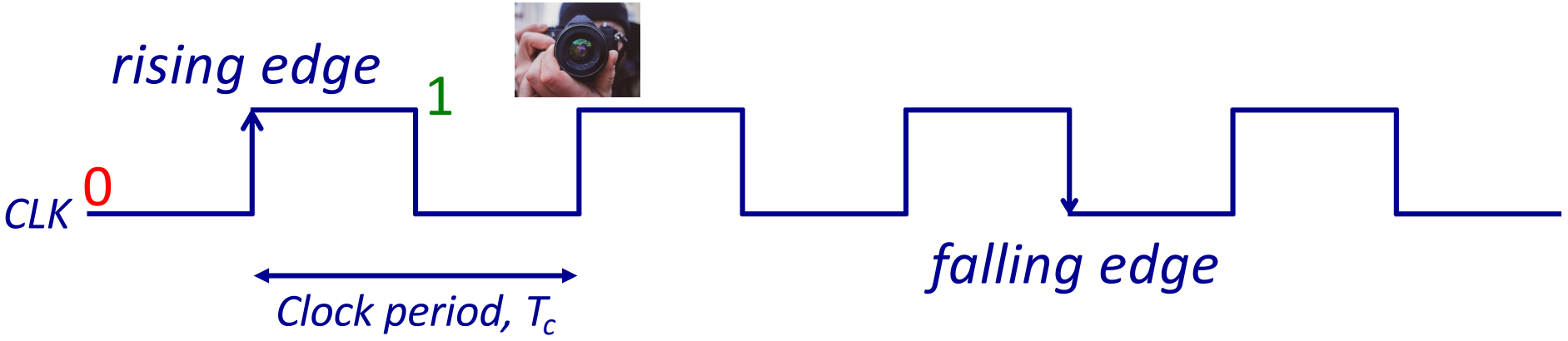
Timing in Sequential Circuits

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



Recall the Clock

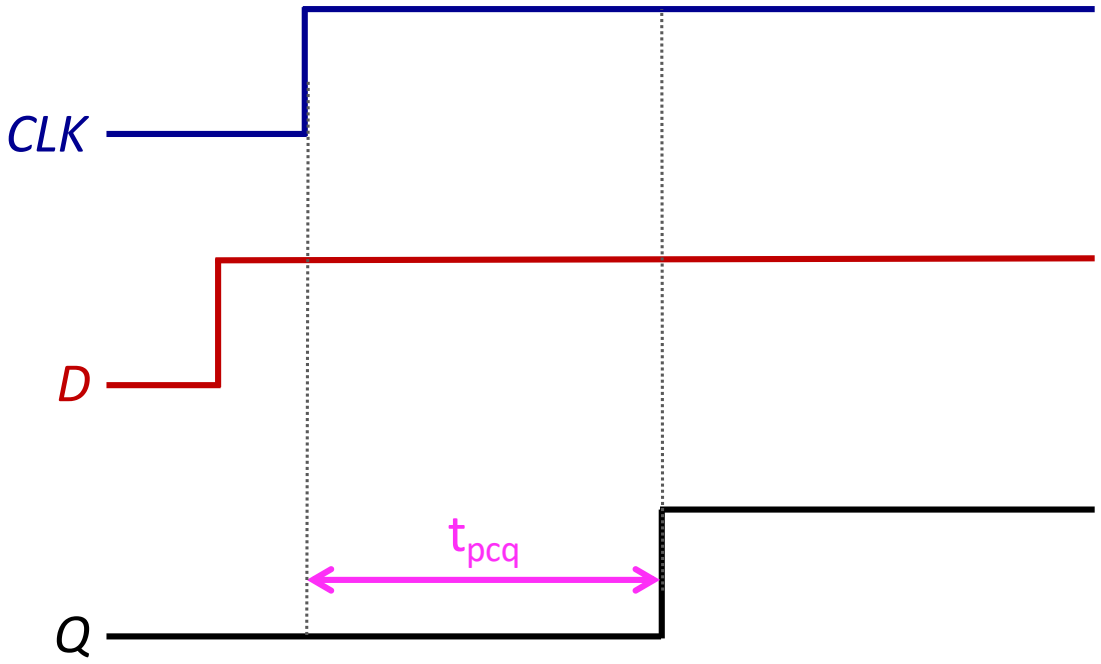
- Output does not change instantly when the **rising edge** arrives
- Input need to stay **stable** for some time period for the flip-flop to take a **reliable** photograph

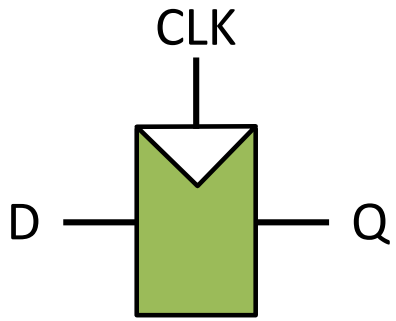


Frequency = $1/T_c$

Clock-to-Q Propagation Delay

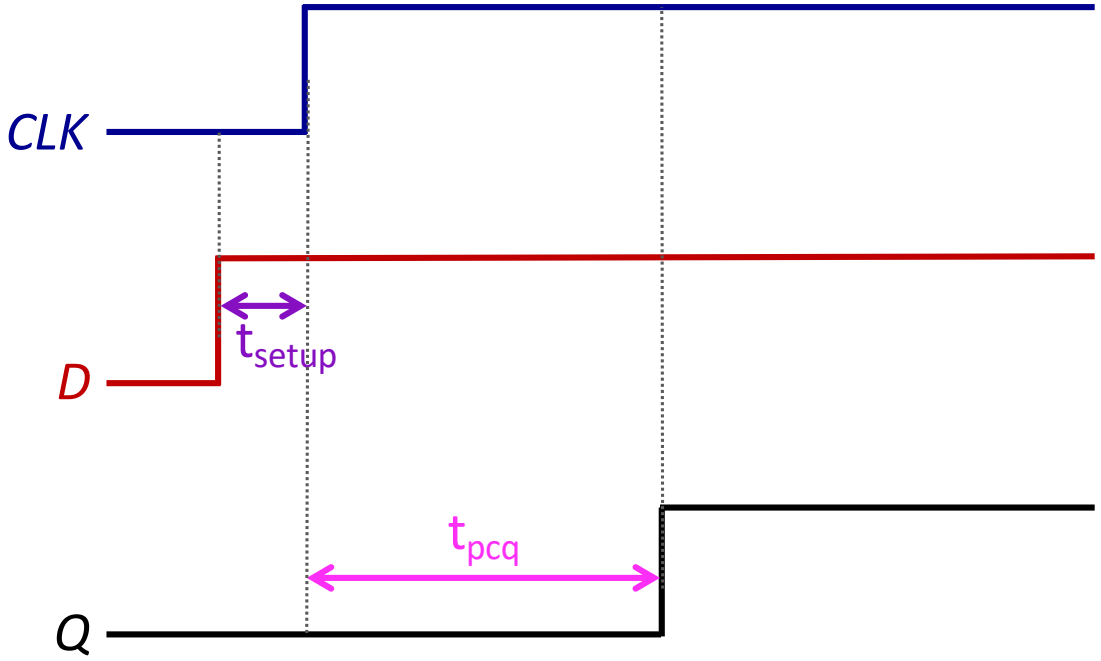
- When the clock rises, the time it takes for the output to **settle** to the final value (t_{pcq})

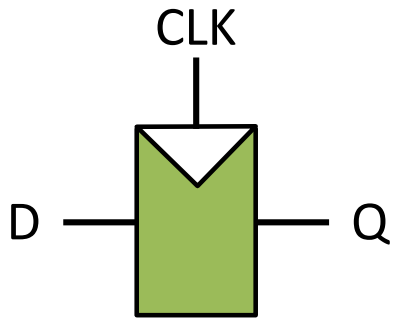




Setup Time

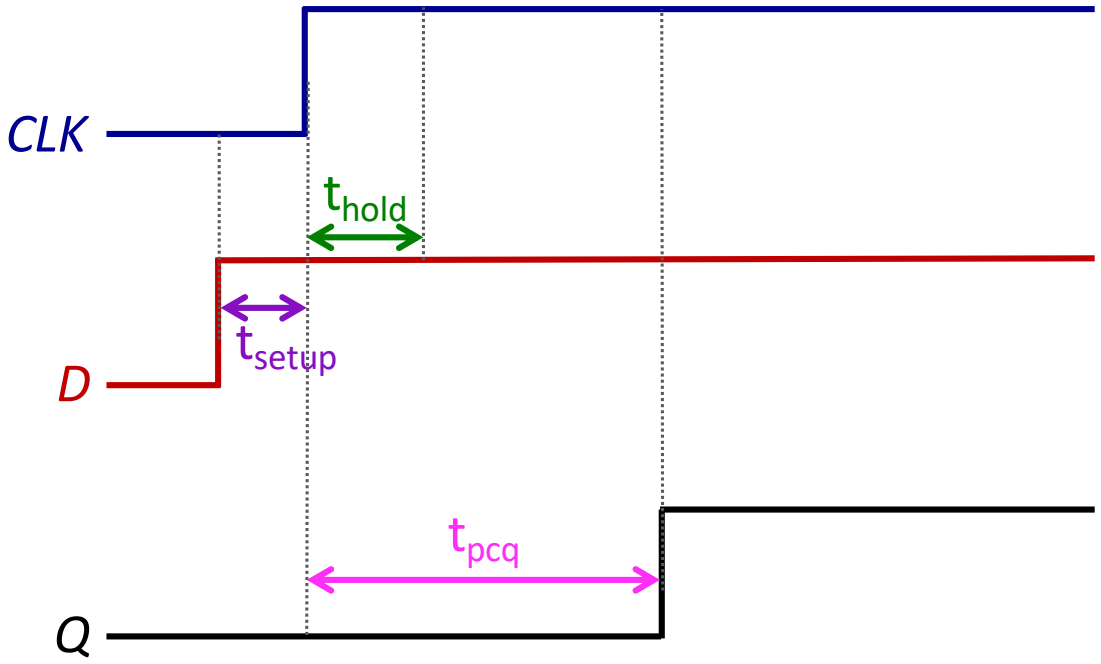
- For the circuit to sample its input correctly, the input must have **stabilized** at least some setup time, t_{setup}, before the rising edge of the clock

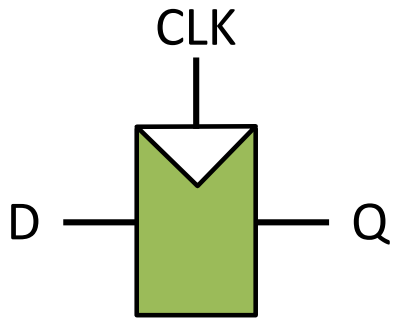




Hold Time

- 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

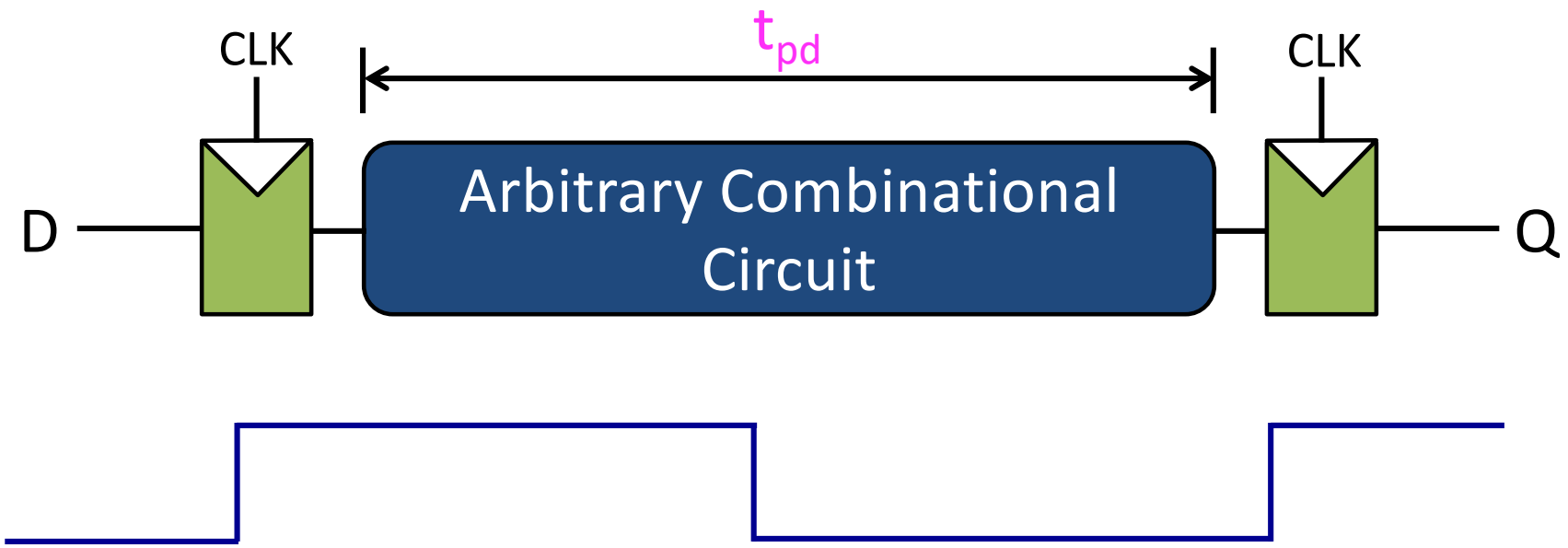
Sequencing Overhead

- $t_{pcq} + t_{setup}$ 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 (i.e., processing done by the combinational circuit)
  - The **sequencing** overhead of the flip-flop **cuts** into this time

Example

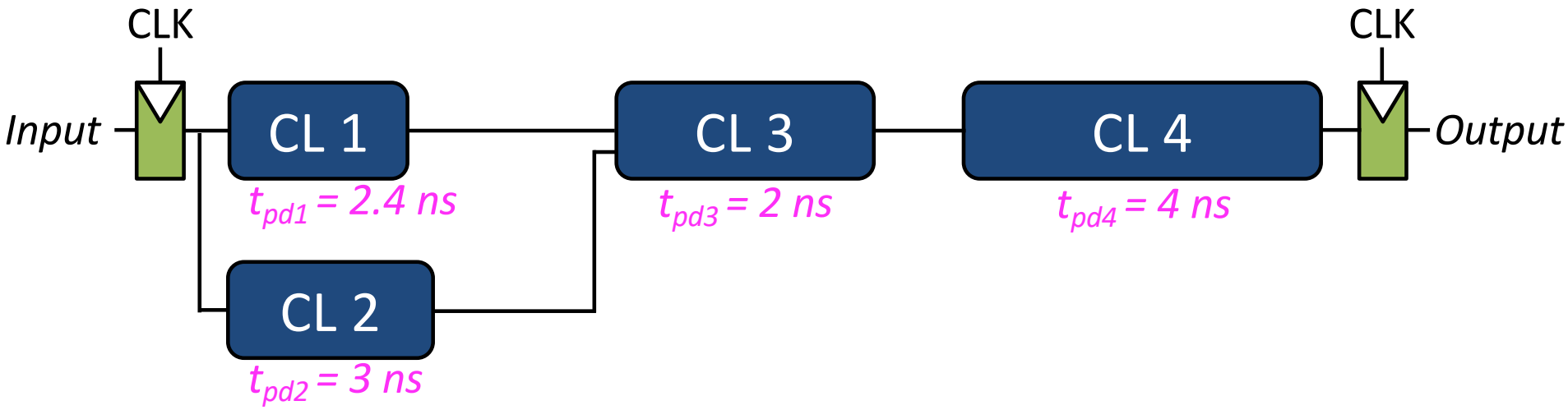
- What is the **clock period** for the circuit below for it to work correctly?
  - t_{pd}
  - $t_{pd} + t_{pcq}$
  - $t_{pd} + t_{pcq} + t_{setup}$



Exercise (Self-Study): Pipelining

Clock-to-Q propagation delay: 0.3 ns

Setup time : 0.2 ns

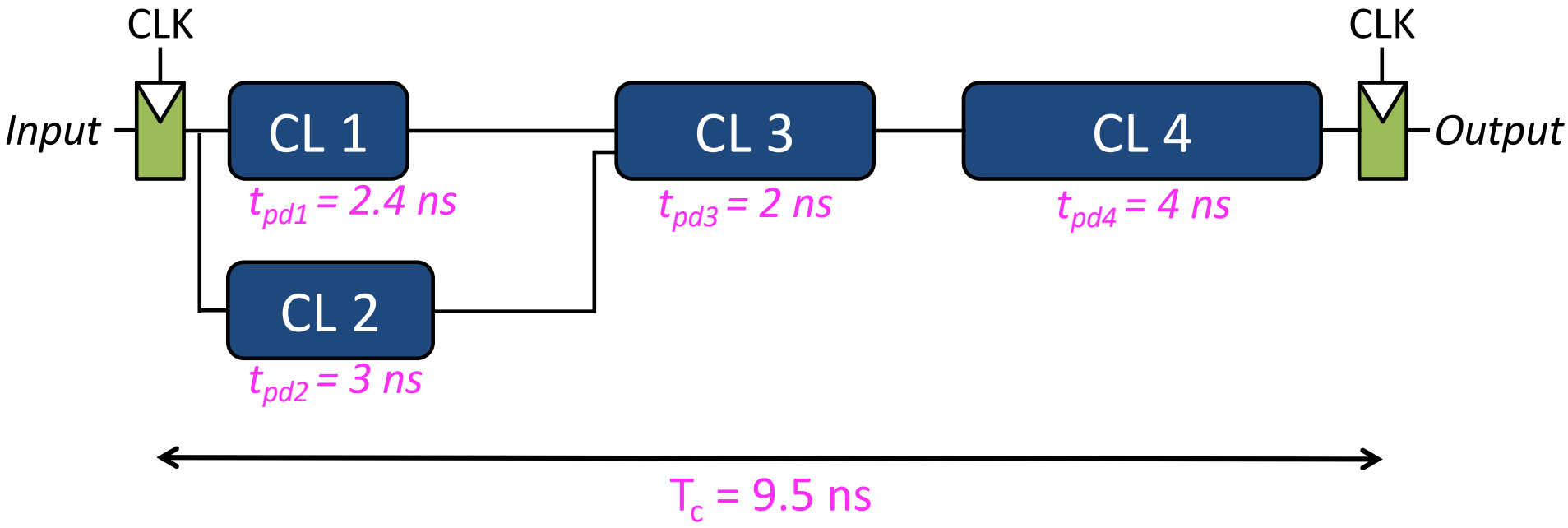


Consider the circuit above. What is the clock period?

Exercise (Self-Study): Pipelining

Clock-to-Q propagation delay: 0.3 ns

Setup time : 0.2 ns

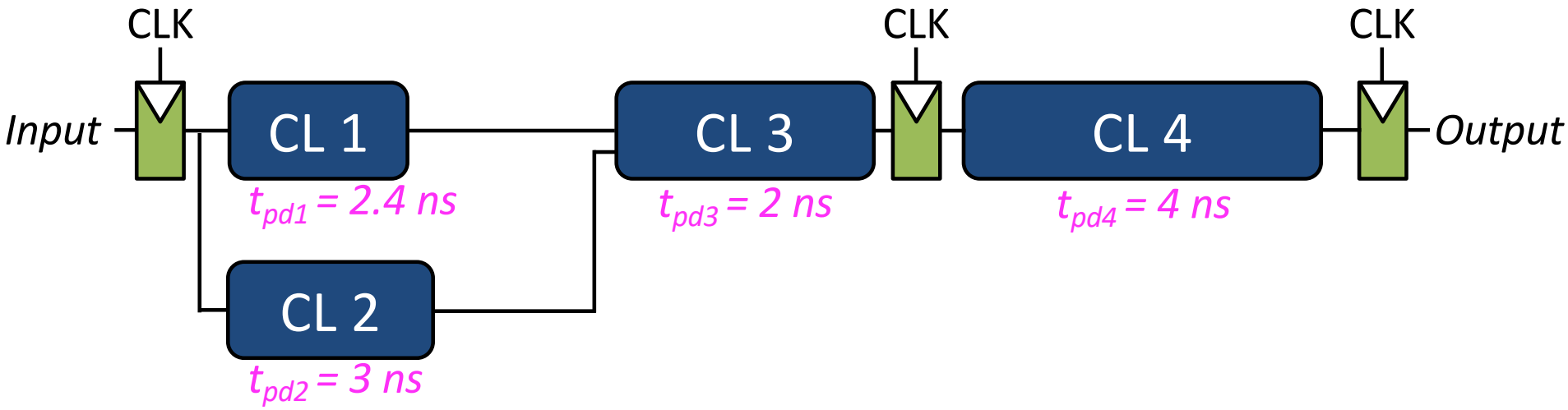


Latency = 9.5 ns

Frequency = 1/9.5 ns = 105 MHz

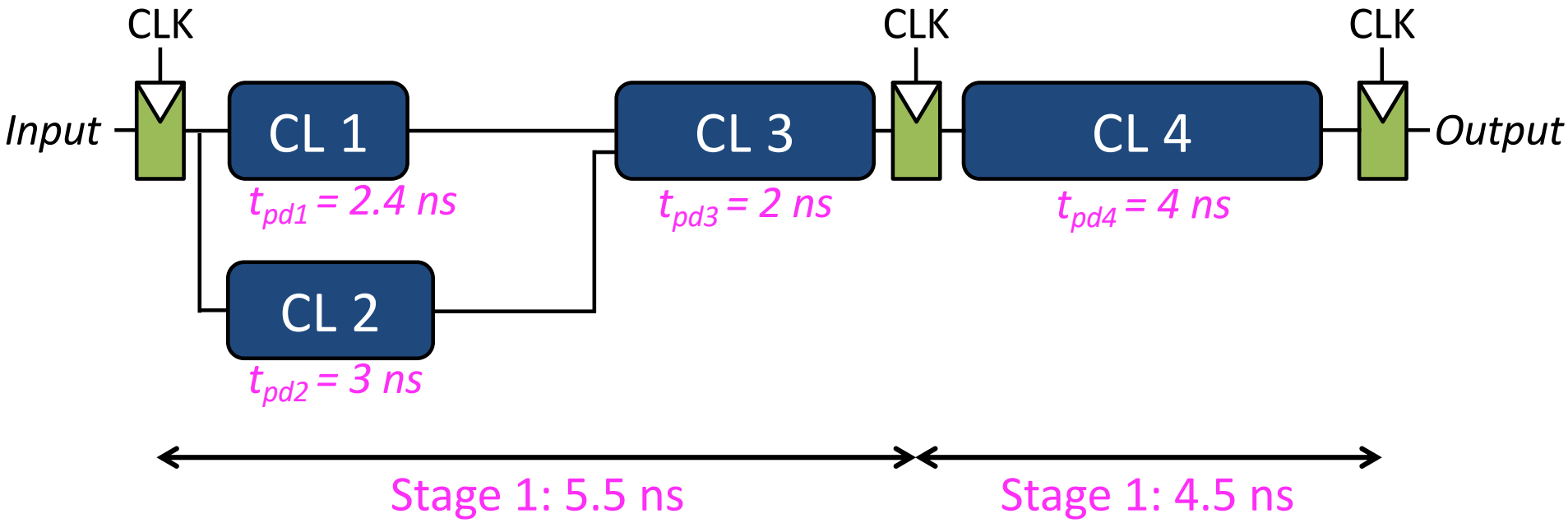
Exercise: 2-stage pipeline

Each task takes *two* clock cycles, but cycle time is reduced



Exercise: 2-stage pipeline

Each task takes *two* clock cycles, but cycle time is reduced

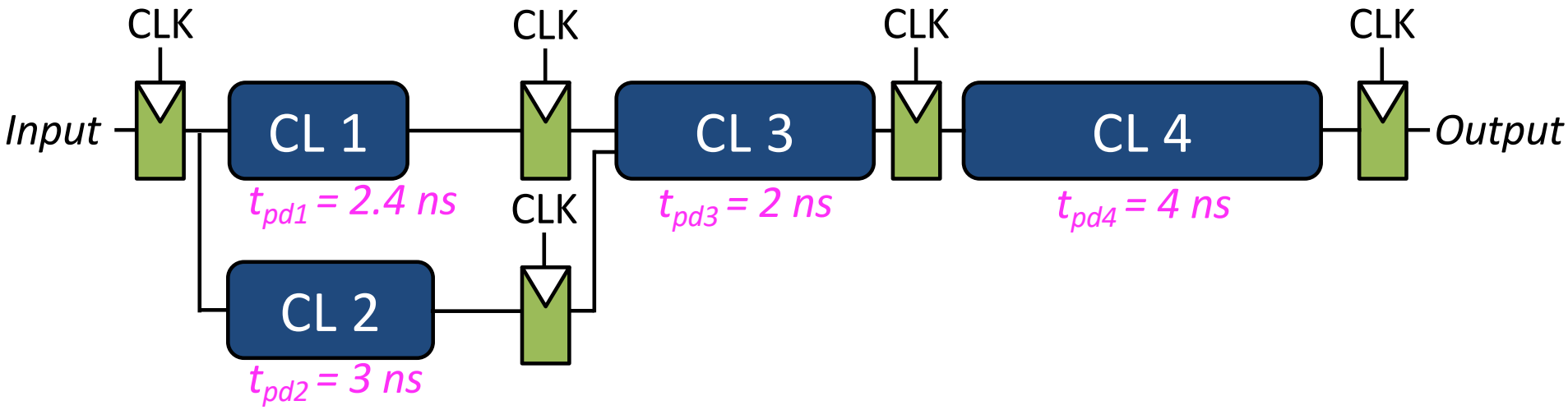


Latency = 2 X 5.5 ns = 11 ns

Frequency = 1/5.5 ns = 182 MHz

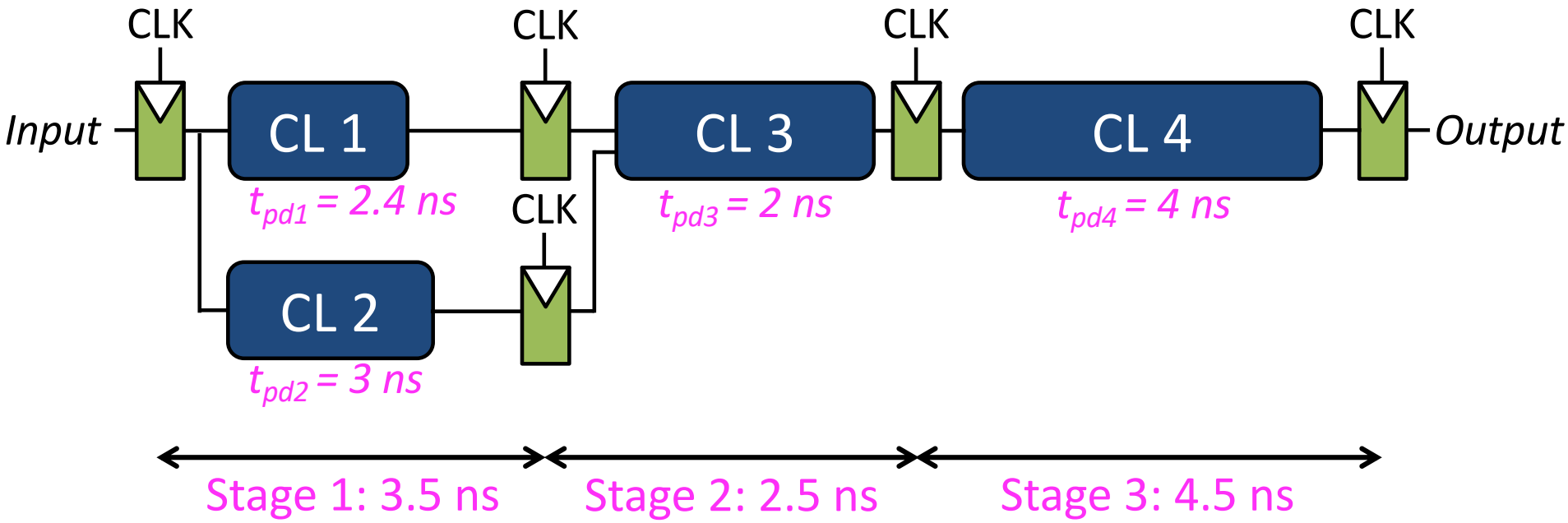
Exercise: 3-stage pipeline

Each task takes *three* clock cycles, but cycle time is further reduced



Exercise: 3-stage pipeline

Each task takes *three* clock cycles, but cycle time is further reduced



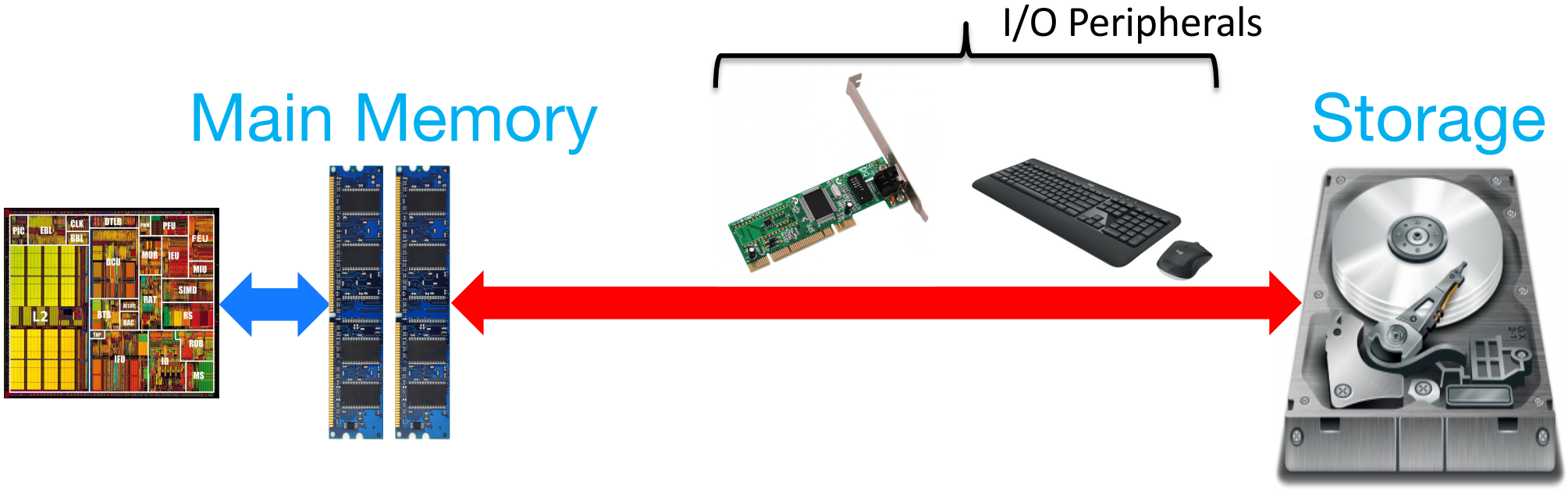
Latency = 3 X 4.5 ns = 13.5 ns

Frequency = 1/4.5 ns = 222 MHz

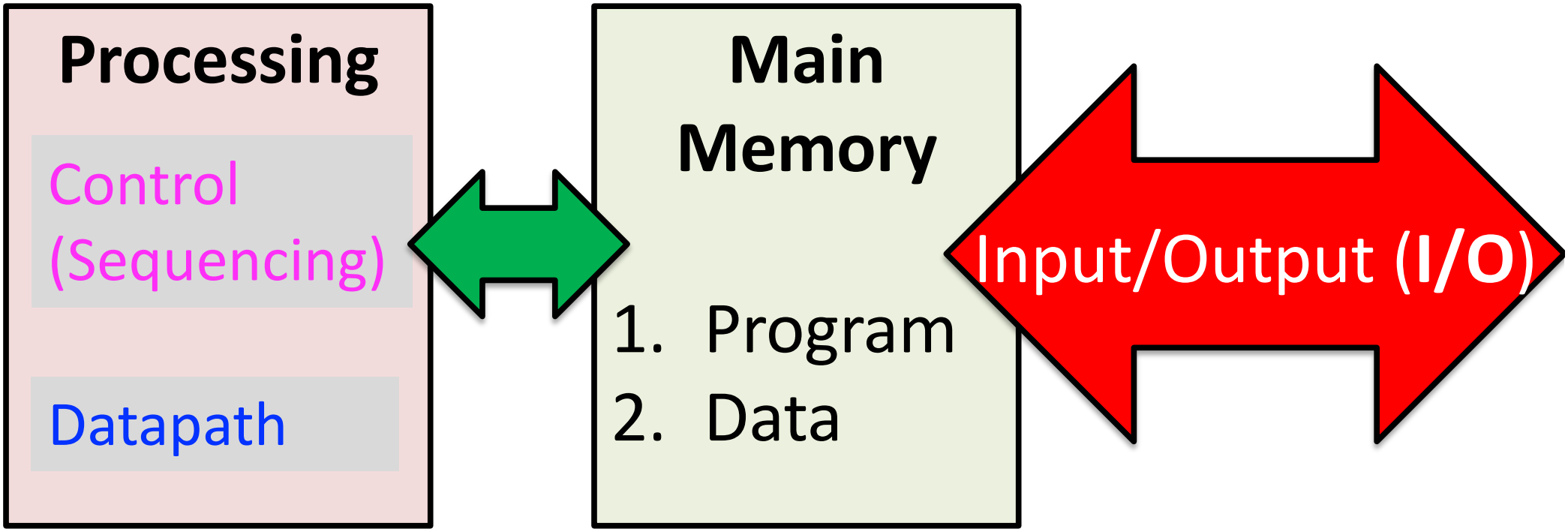
Von Neumann Model

Recall: A Computer System

- **Key resources:** CPU, memory, and Input/Output (I/O) devices
  - CPU (processor) does the actual processing (computation)
  - Memory stores temporary data and forms a hierarchy (Registers, SRAM, DRAM, ...)
  - Some fast (small capacity) memory is close to the CPU; rest is far
  - Storage disk is an I/O device (much slower than memory, stores persistent data)
  - Memory is volatile, while disk is non-volatile



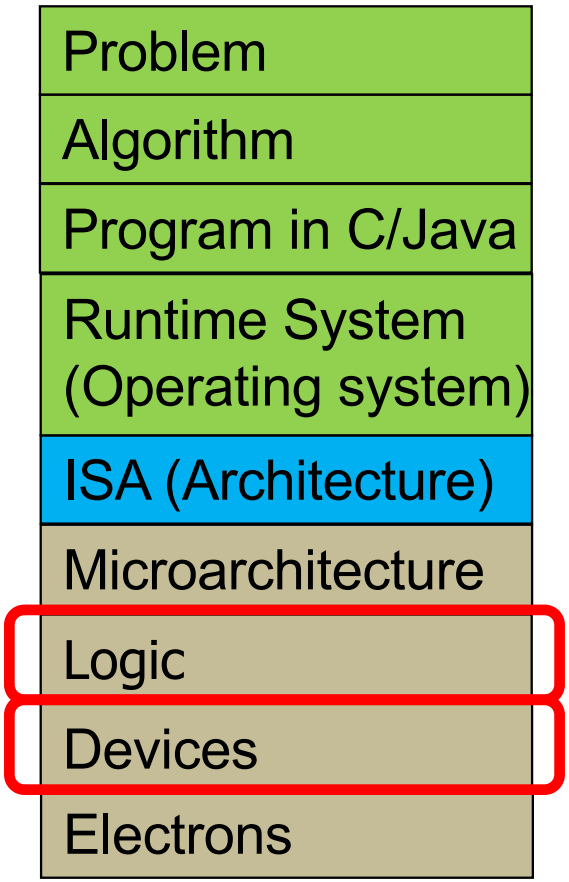
What is a Computer?



- We will cover all three components

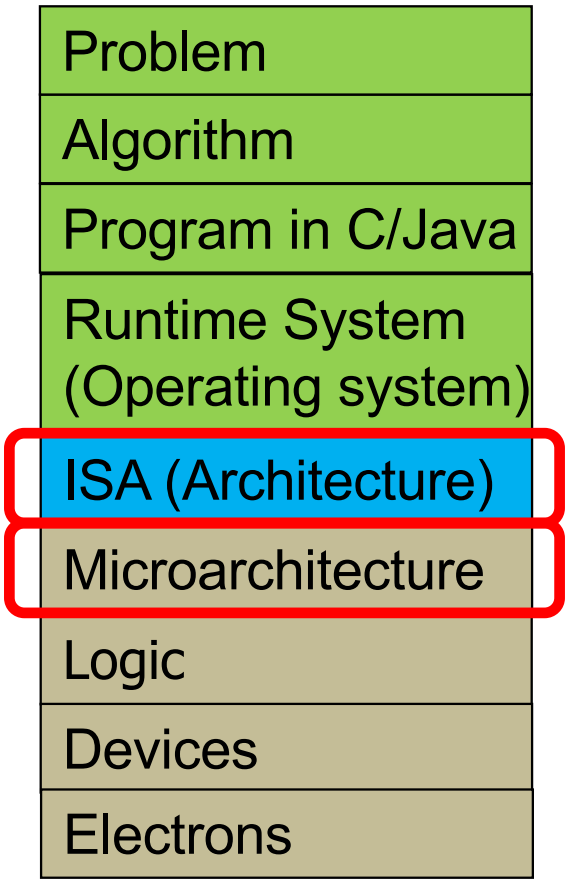
Building up a Basic Computer Model

- In past lectures, we learned how to design
  - Combinational logic structures
  - Sequential logic structures
- With logic structures, we can build
  - Execution units
  - Decision units
  - Memory/storage units
  - Communication units
- All are basic elements of a computer
  - We will raise our abstraction level today
  - Use logic structures to construct a basic computer model



Building up a Basic Computer Model

- **ISA:** Specification of the instructions computer can perform
- **Microarchitecture:** Circuit implementation of the specification



What is a Computer?

- To get a task done by a (general-purpose) computer, we need
  - A computer program
    - That specifies what the computer must do
  - The computer itself
    - To carry out the specified task

- Program: A set of instructions
  - Each instruction specifies a well-defined piece of work for the computer to carry out
  - Instruction: the smallest piece of specified work in a program

- Instruction set: All possible instructions that a computer is designed to be able to carry out

The Von Neumann Model

- In order to build a computer, we need an execution model for processing computer programs

- John von Neumann proposed a fundamental model in 1946

- The von Neumann Model consists of 5 components
  - Memory (stores the program and data)
  - Processing unit
  - Input
  - Output
  - Control unit (controls the order in which instructions are carried out)

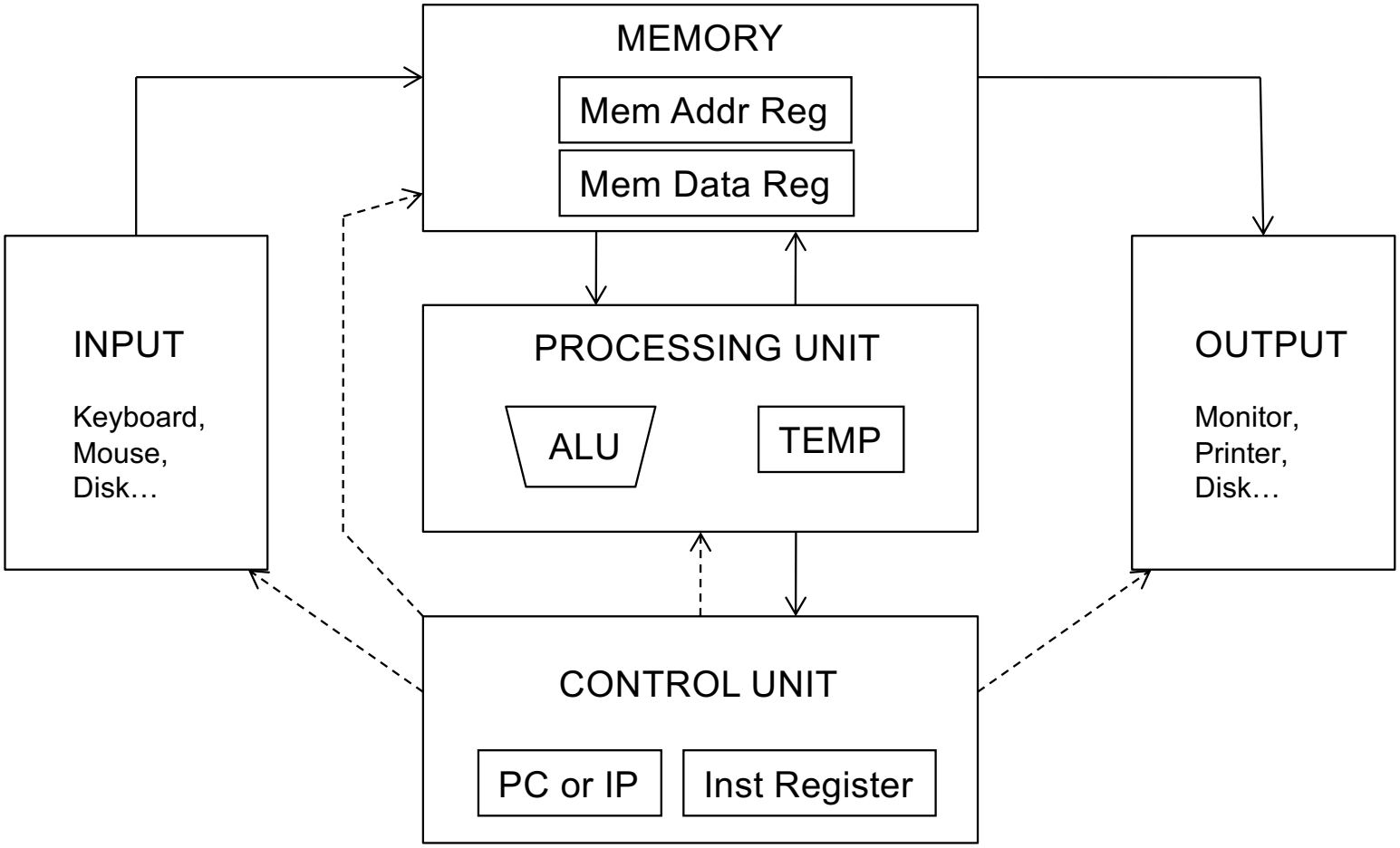
Burks, Goldstein, von Neumann,
"Preliminary discussion of the logical design of an electronic computing instrument," 1946.

All general-purpose computers today use the von Neumann model

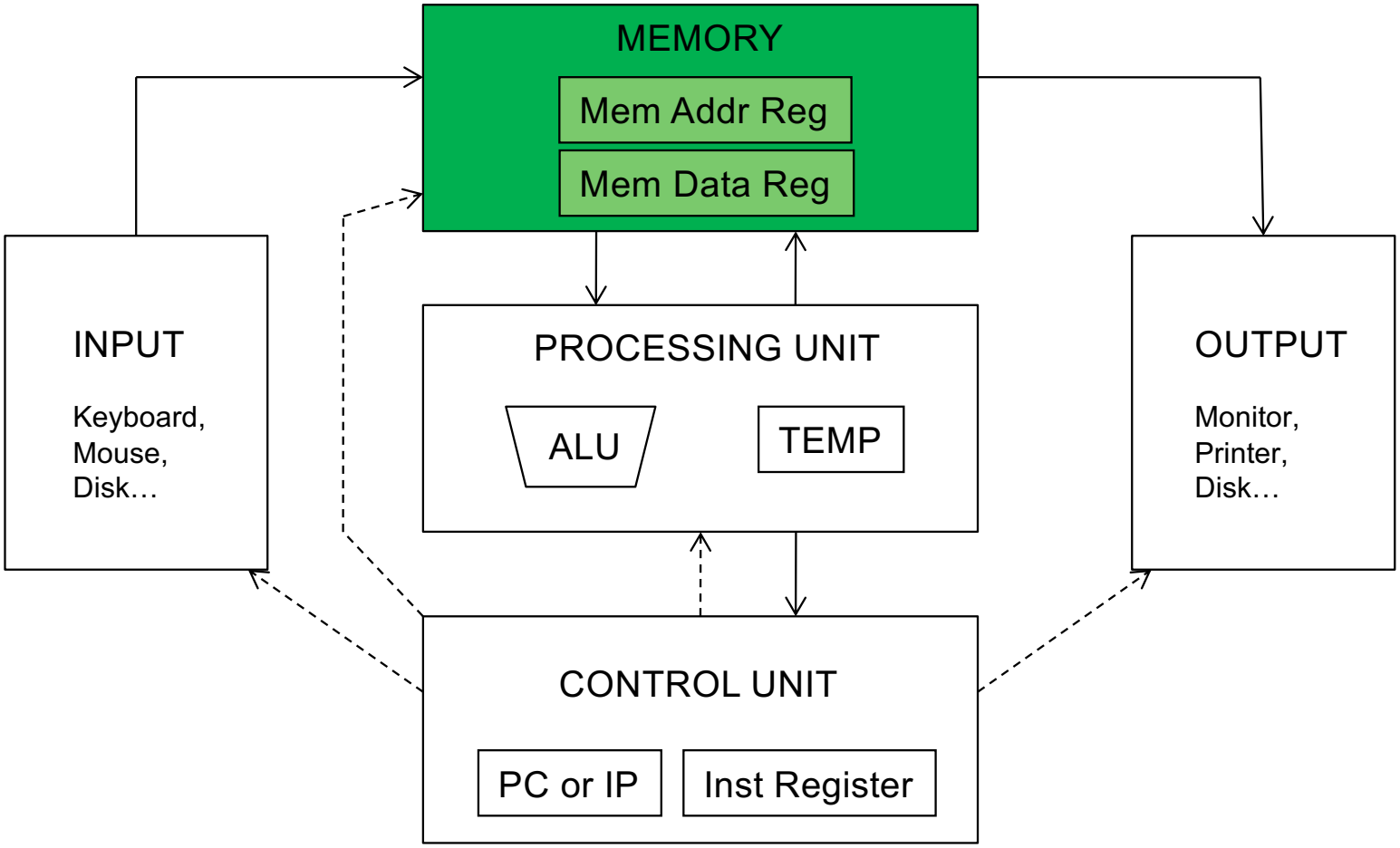
The Von Neumann Model

- Throughout this lecture, we will examine two examples of the von Neumann model
  - ARM
  - MIPS

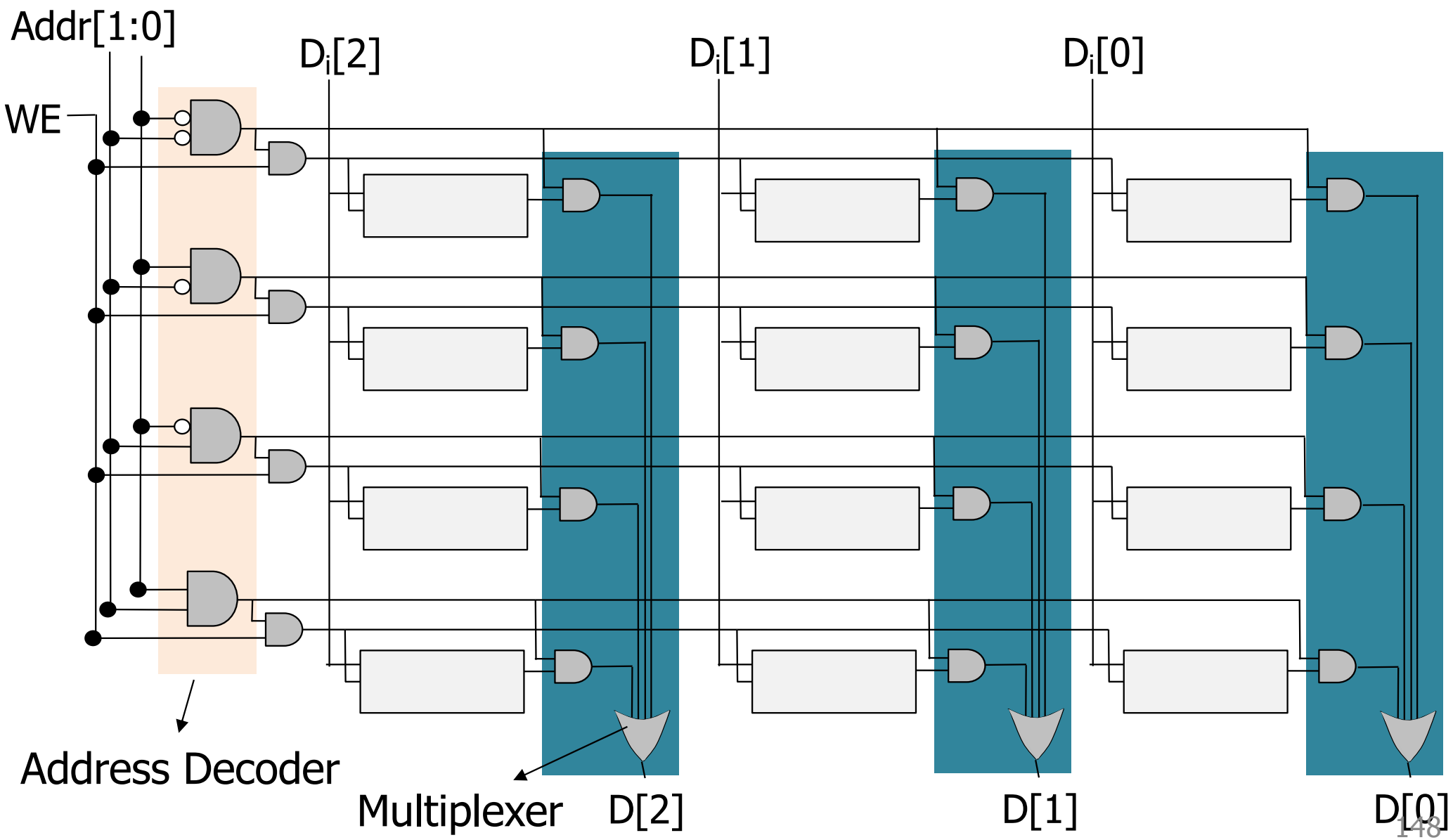
The Von Neumann Model



The Von Neumann Model

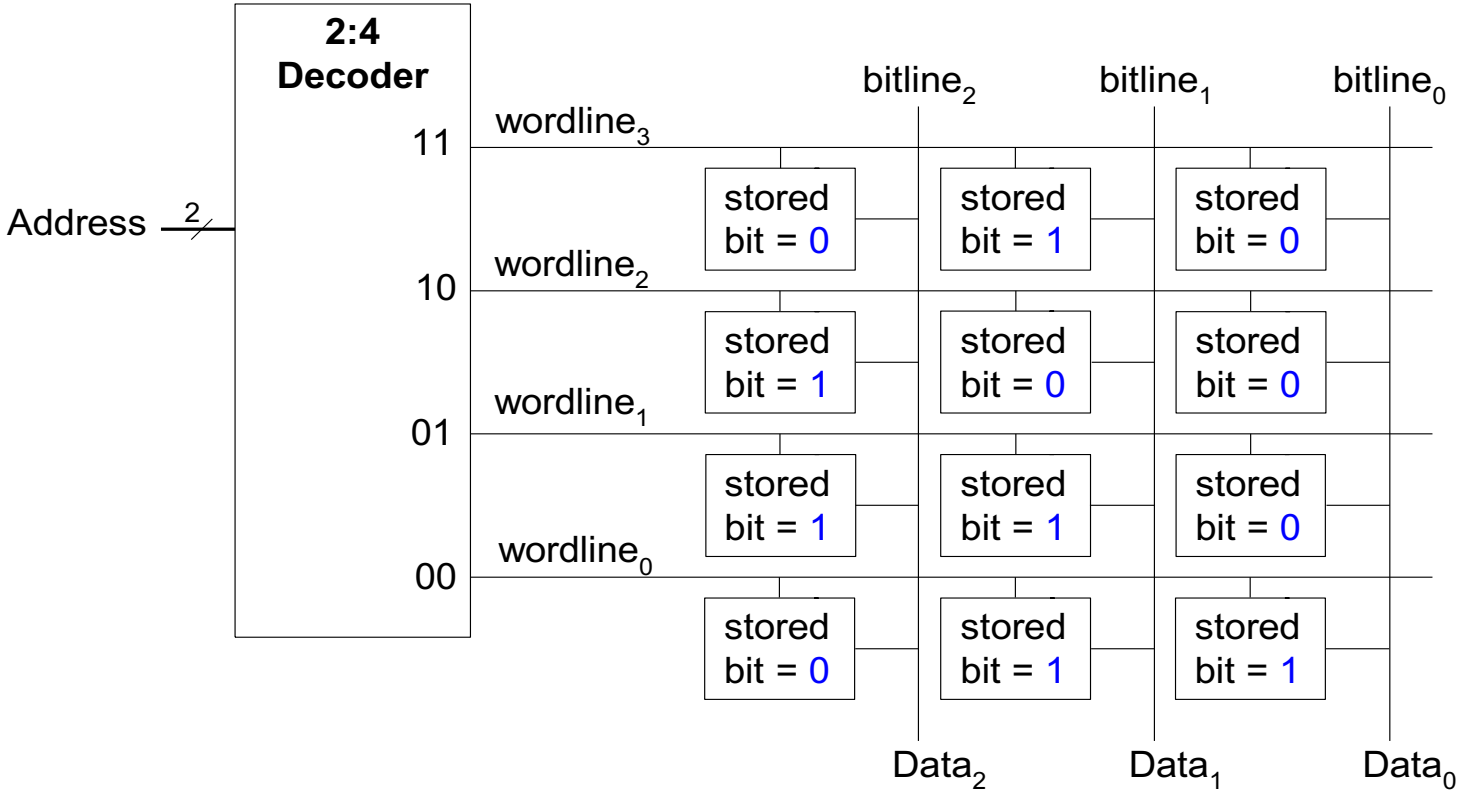


Recall: A Memory Array (4 locations X 3 bits)


Addr[1:0]
WE
$D_i[2]$
$D_i[1]$
$D_i[0]$
Address Decoder
Multiplexer
D[2]
D[1]
D[0]

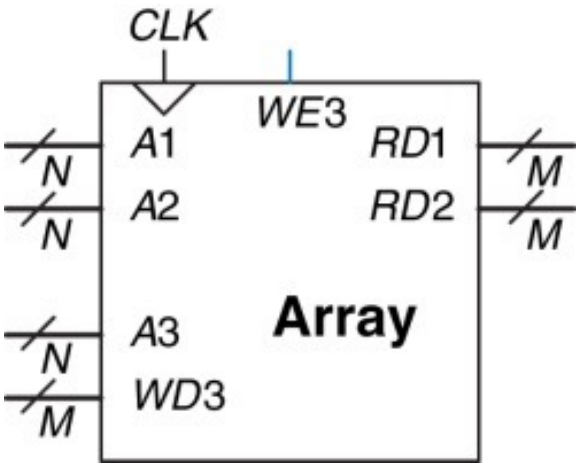
Recall: Memory Array Organization

- Decoder drives the wordline HIGH based on the address
- Data on the selected row appears on the bitlines



Recall: Memory Ports

- Each memory port gives **read** or **write** access to one memory address
- Multiported memories can access **multiple** addresses **simultaneously**
- **Example of three-ported memory**
  - Port 1 reads the data from address A1 onto the read data output RD1
  - Port 2 reads the data from address A2 onto the read data output RD2
  - Port 3 writes the data from the write data input WD3 into address A3 on the rising clock edge if WE3 is **TRUE**



Memory

- Memory stores
  - Programs
  - Data

address space

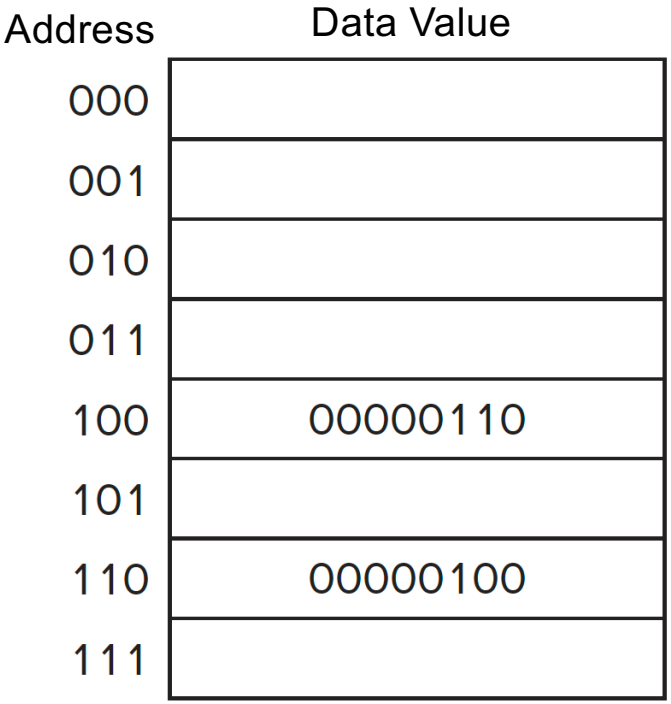
- Memory contains bits
  - Bits are logically grouped into bytes (8 bits) and words (e.g., 8, 16, 32 bits)

- Address space: Total number of uniquely identifiable locations
  - In MIPS, the address space is 2^{32}
    - 32-bit addresses
  - In ARM, the address space is 2^{32}
    - 32-bit addresses
  - In x86-64, the address space is (up to) 2^{48}
    - 48-bit addresses

- Addressability: How many bits are stored in each location (address)
  - E.g., 8-bit addressable (or byte-addressable)
  - E.g., word-addressable
  - A given instruction can operate on a byte or a word

A Simple Example

- A representation of memory with 8 locations
- Each location contains 8 bits (one byte)
  - Byte addressable memory; address space of 8
  - Value 6 is stored in address 4 & value 4 is stored in address 6

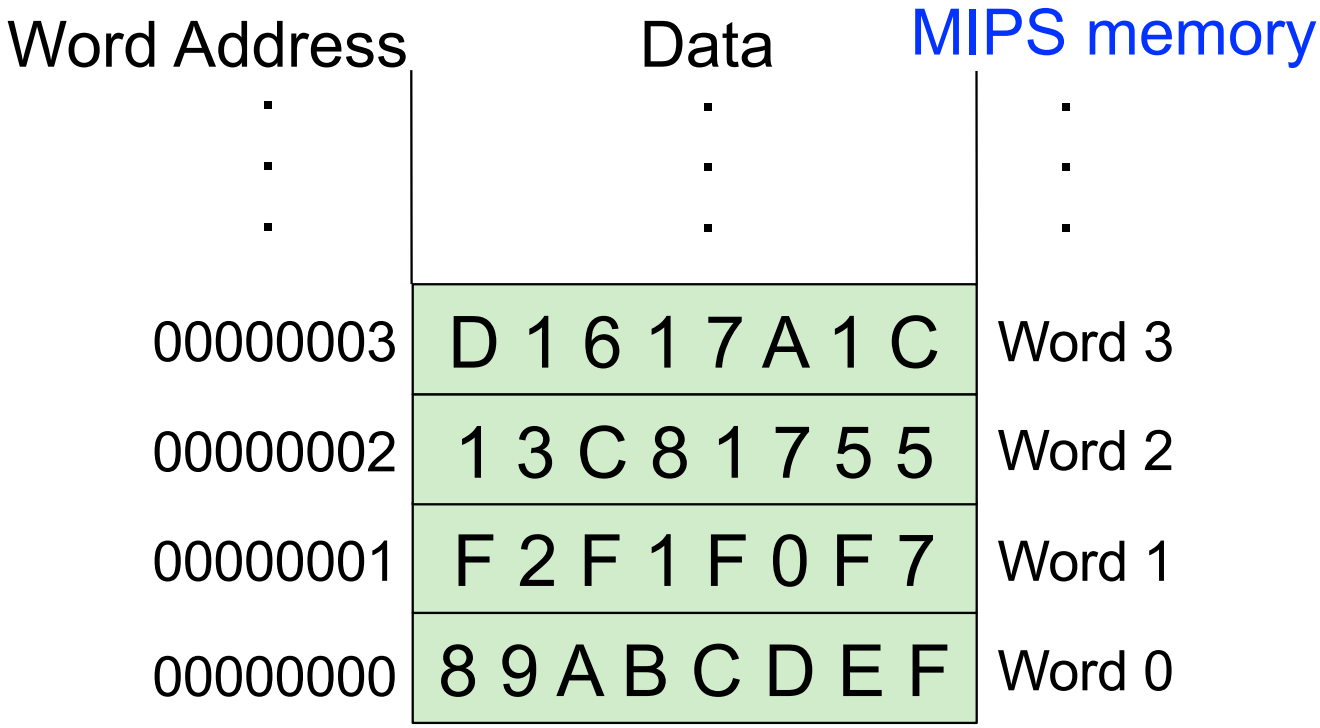


Question:
How can we make same-size memory bit addressable?

Answer:
64 locations
Each location stores 1 bit

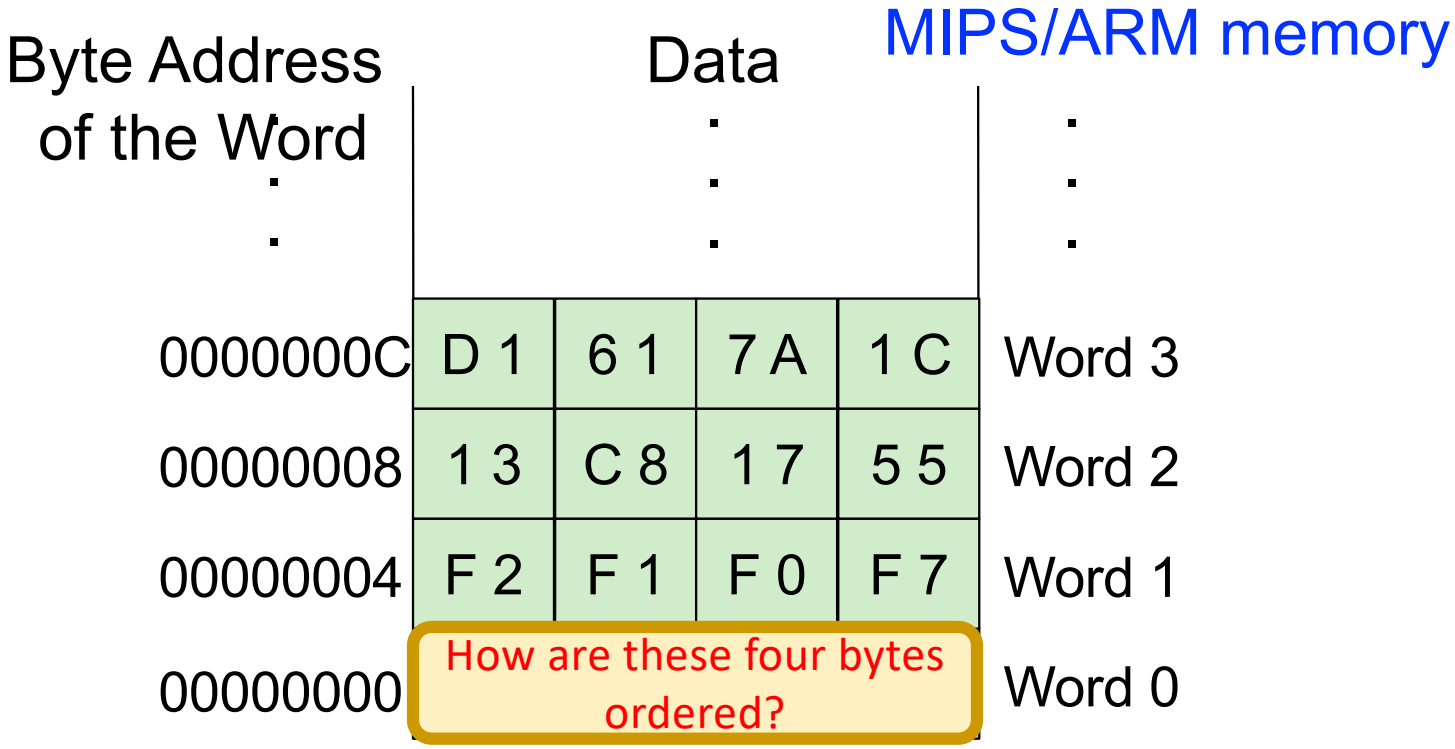
Word-Addressable Memory

- Each data word has a unique address
  - In MIPS, a unique address for each 32-bit data word
  - In **QuAC**, a unique address for each 16-bit data word



Byte-Addressable Memory

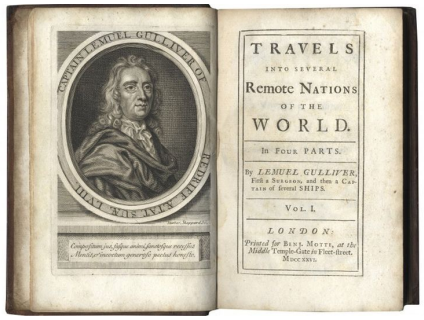
- Each byte has a unique address
  - MIPS is actually byte-addressable
  - ARM is also byte-addressable

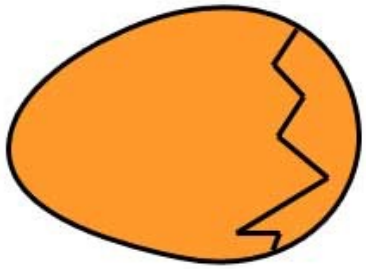


Which of the four bytes is most vs. least significant?

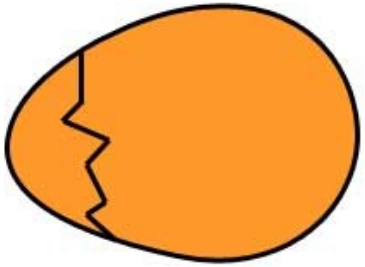
Big Endian vs. Little Endian

- Jonathan Swift's Gulliver's Travels
  - Big Endians broke their eggs on the big end of the egg
  - Little Endians broke their eggs on the little end of the egg



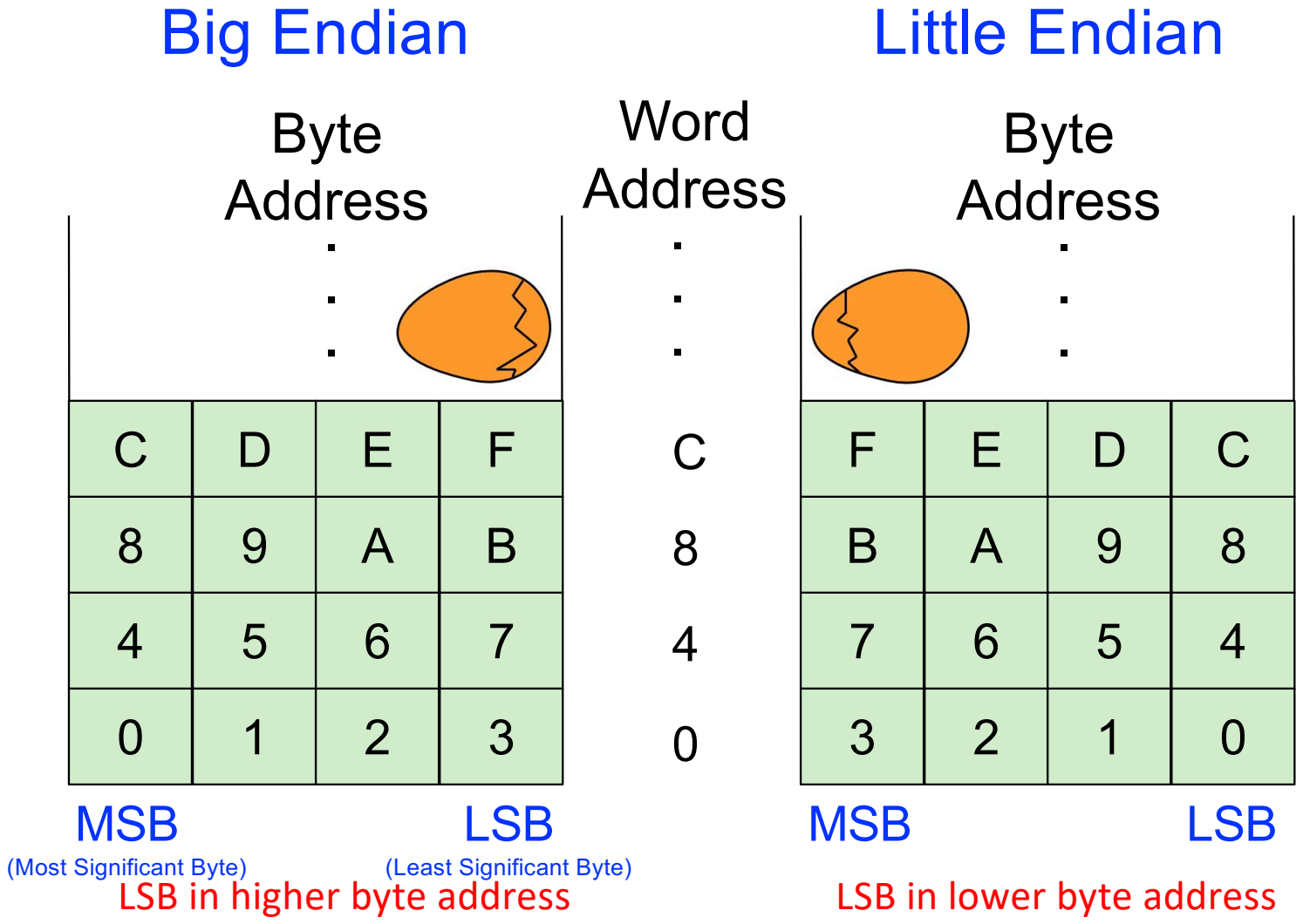


BIG ENDIAN - The way people always broke their eggs in the Lilliput land

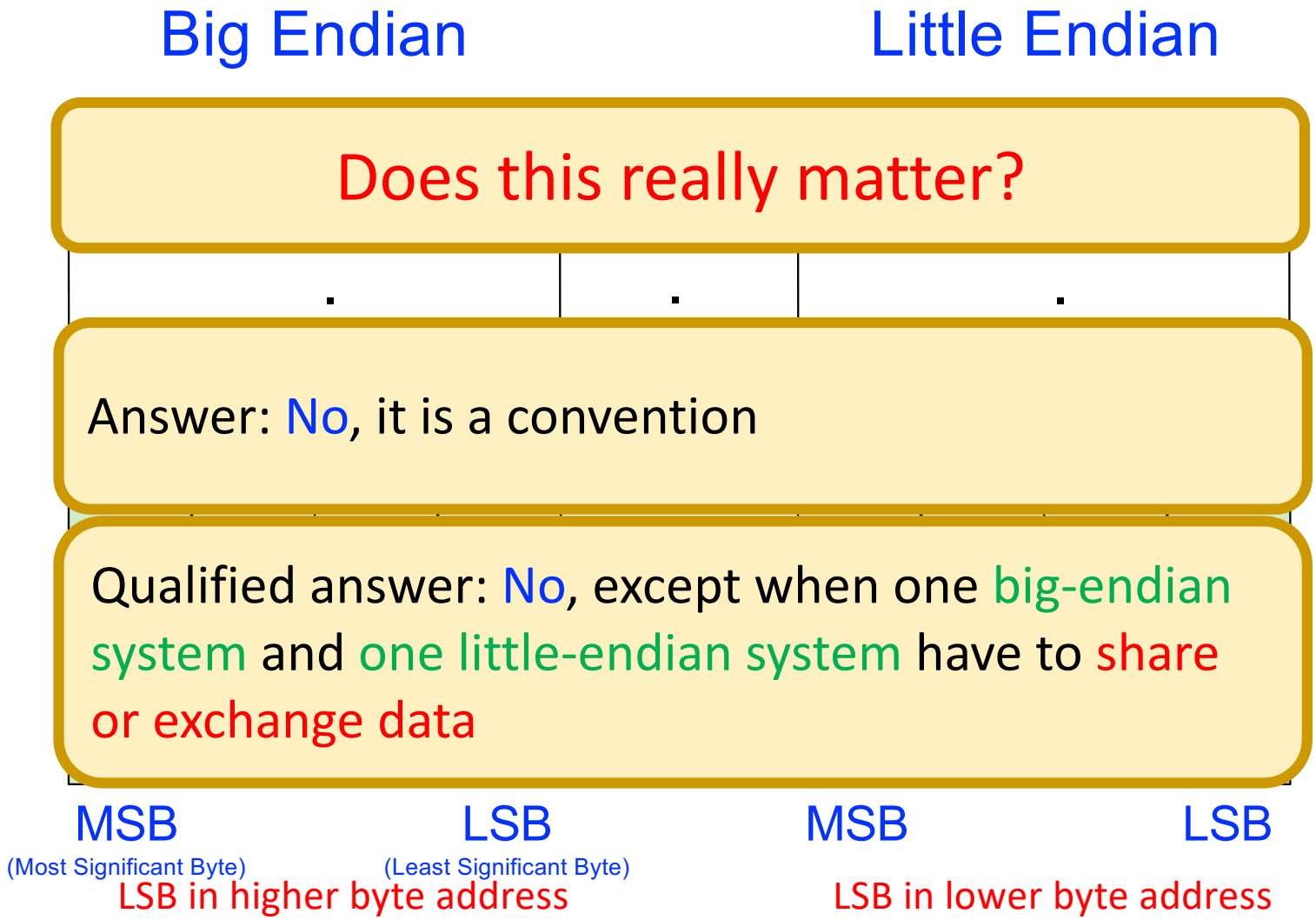


LITTLE ENDIAN - The way the king then ordered the people to break their eggs

Big Endian vs. Little Endian

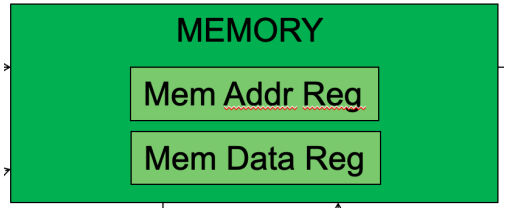


Big Endian vs. Little Endian



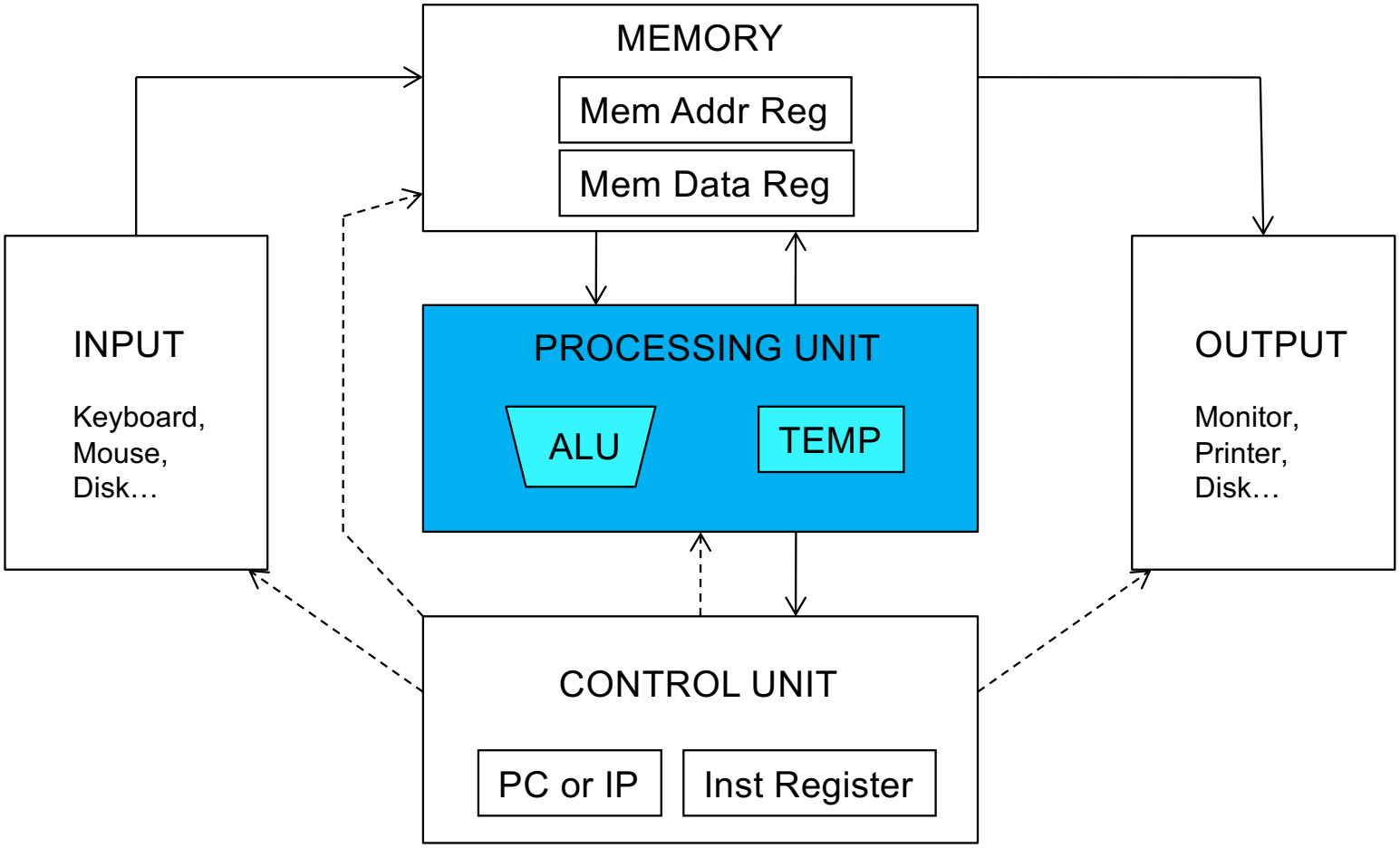
Accessing Memory: MAR and MDR

- There are two ways of accessing memory
  - Reading or loading data from a memory location
  - Writing or storing data to a memory location
- Two registers are usually used to access memory
  - Memory Address Register (MAR)
  - Memory Data Register (MDR)



- To read
  - Step 1: Load the MAR with the address we wish to read from
  - Step 2: Data in the corresponding location gets placed in MDR
- To write
  - Step 1: Load the MAR with the address and the MDR with the data we wish to write
  - Step 2: Activate Write Enable signal → value in MDR is written to address specified by MAR

The Von Neumann Model



Processing Unit

- Performs the actual computation(s)
- The processing unit can consist of many functional units
- We start with a simple Arithmetic and Logic Unit (ALU), which executes computation and logic operations
  - ARM: ADD, AND, NOT, SUB
  - MIPS: add, sub, mult, and, nor, sll, slr, slt…
- The ALU processes quantities that are referred to as words
  - Word length in ARM**v4** is 32 bits (**v8** is 64 bits)
  - Word length in MIPS is 32 bits

Recall: Arithmetic & Logic Unit (ALU)

- Combines a variety of arithmetic and logical operations into a single unit (that performs only one function at a time)
- Usually denoted with this symbol:

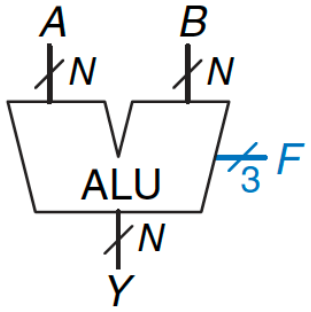


Figure 5.14 ALU symbol

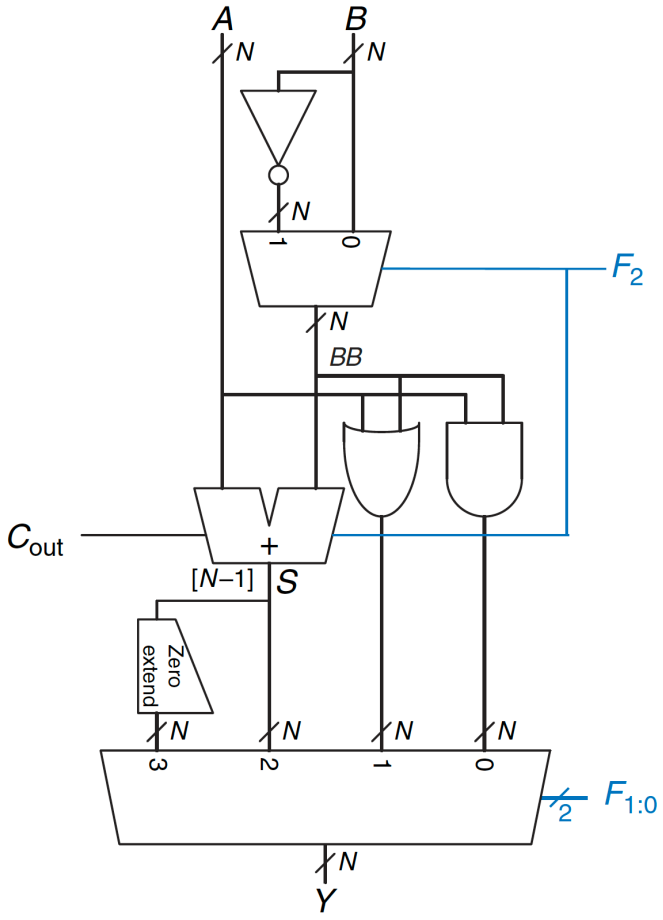
Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

Recall: Arithmetic & Logic Unit (ALU)

Table 5.1 ALU operations

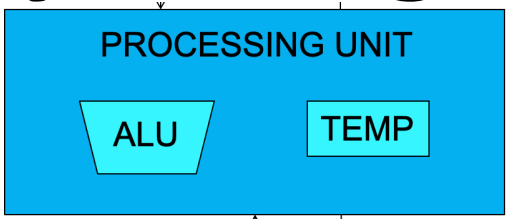
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



Processing Unit: Fast Temporary Storage

- It is almost always the case that a computer provides a small amount of storage very close to ALU
  - Purpose: to store temporary values and quickly access them later
- E.g., to calculate ((A+B)*C)/D, the intermediate result of A+B can be stored in temporary storage
  - Why? It is too slow to store each ALU result in memory & then retrieve it again for future use
    - A memory access is much slower than an addition, multiplication or division
  - Ditto for the intermediate result of ((A+B)*C)
- This temporary storage is usually a set of registers
  - Called Register File

Registers: Fast Temporary Storage



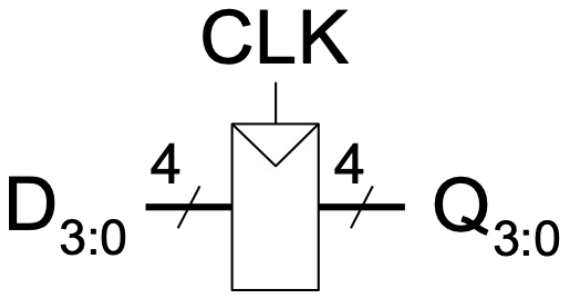
- Memory is large but slow
- Registers in the Processing Unit
  - Ensure fast access to values to be processed in the ALU
  - Typically one register contains one word (same as word length)

ARM GPRS

- Register Set or Register File
  - Set of registers that can be manipulated by instructions
  - ARM has 16 general purpose registers (GPRs)
    - R0 to R15: 4-bit register number
    - Register size = Word length = 32 bits
  - MIPS has 32 general purpose registers
    - R0 to R31: 5-bit register number (or Register ID)
    - Register size = Word length = 32 bits

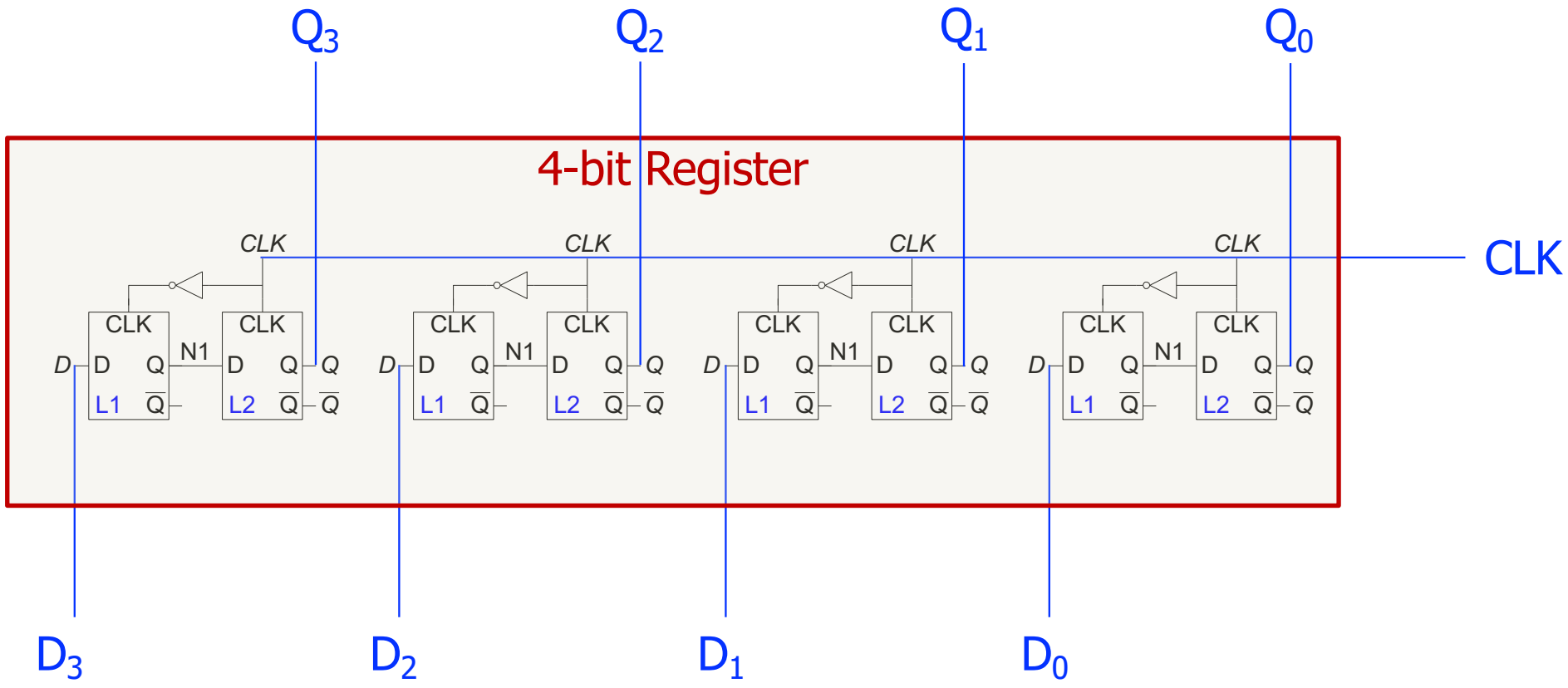
Recall: Register

- **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



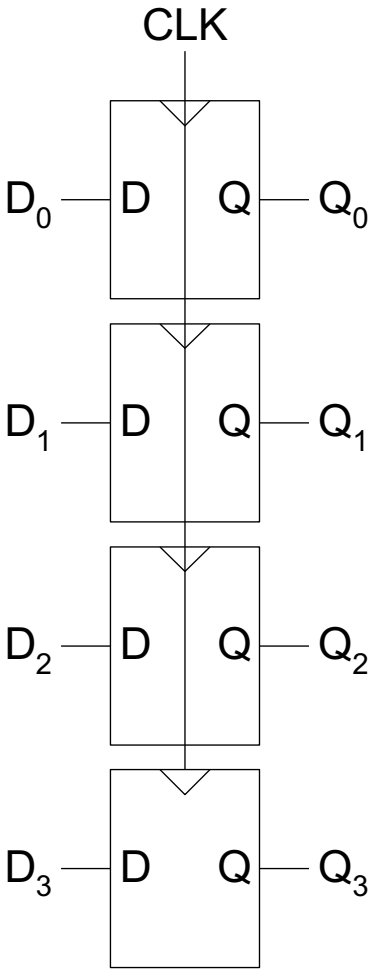
- **Register:** A structure that stores more than **one bit** of information and can be read from and written to
- This **register** holds 4 bits, and its data is referenced as Q[3:0]

Recall: 4-bit Register

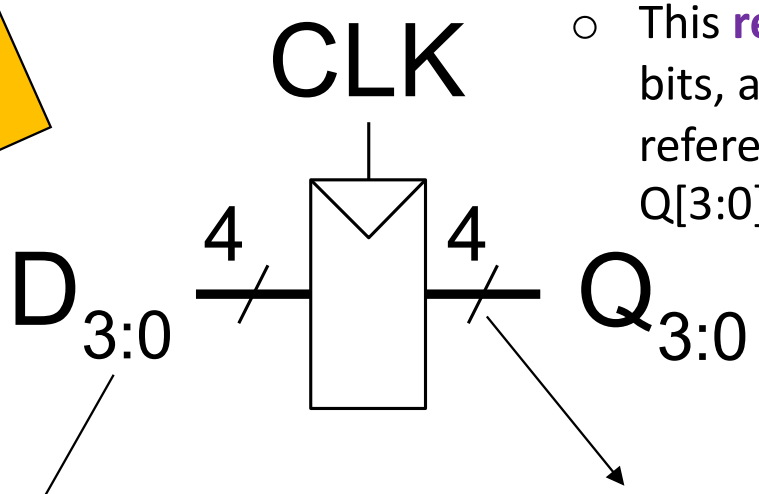


To build an **N-bit** register, use a bank of **N** flipflops with a shared **CLK**

Recall: 4-bit Register



Condensed



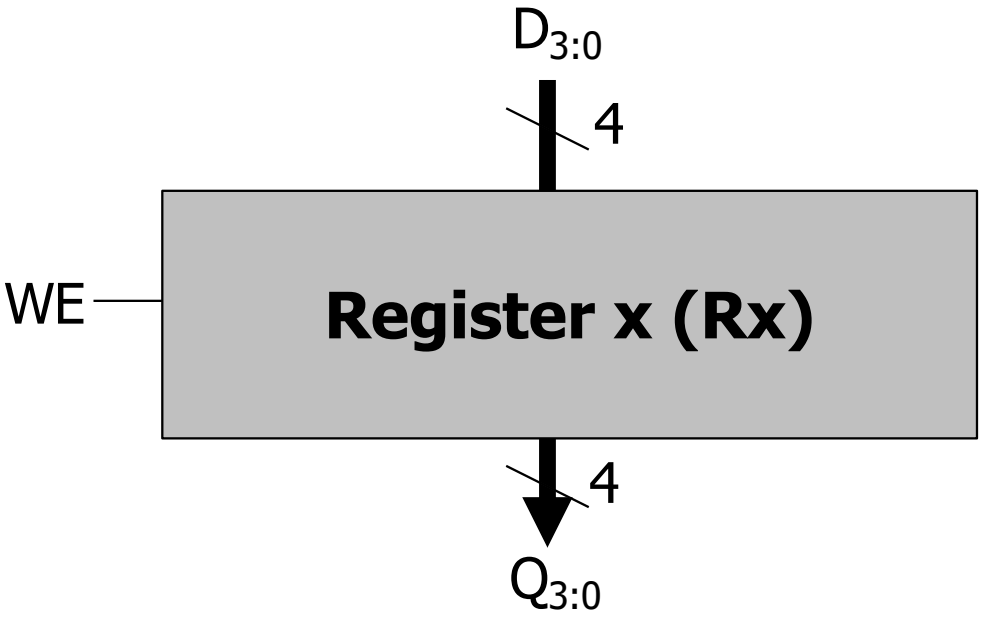
This line represents 4 wires

This register stores 4 bits

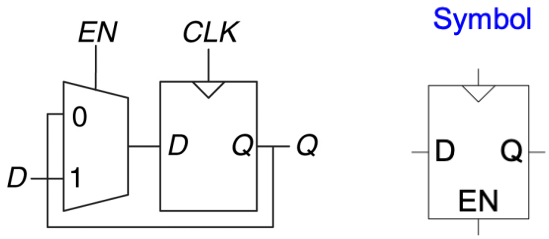
- Here we have a **register,** or a structure that stores more than one bit and can be read from and written to
- This **register** holds 4 bits, and its data is referenced as Q[3:0]

More Realistic Register

- A single WE signal for all flip-flops for simultaneous writes



Enabled Flip-Flop



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

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

MIPS Register File

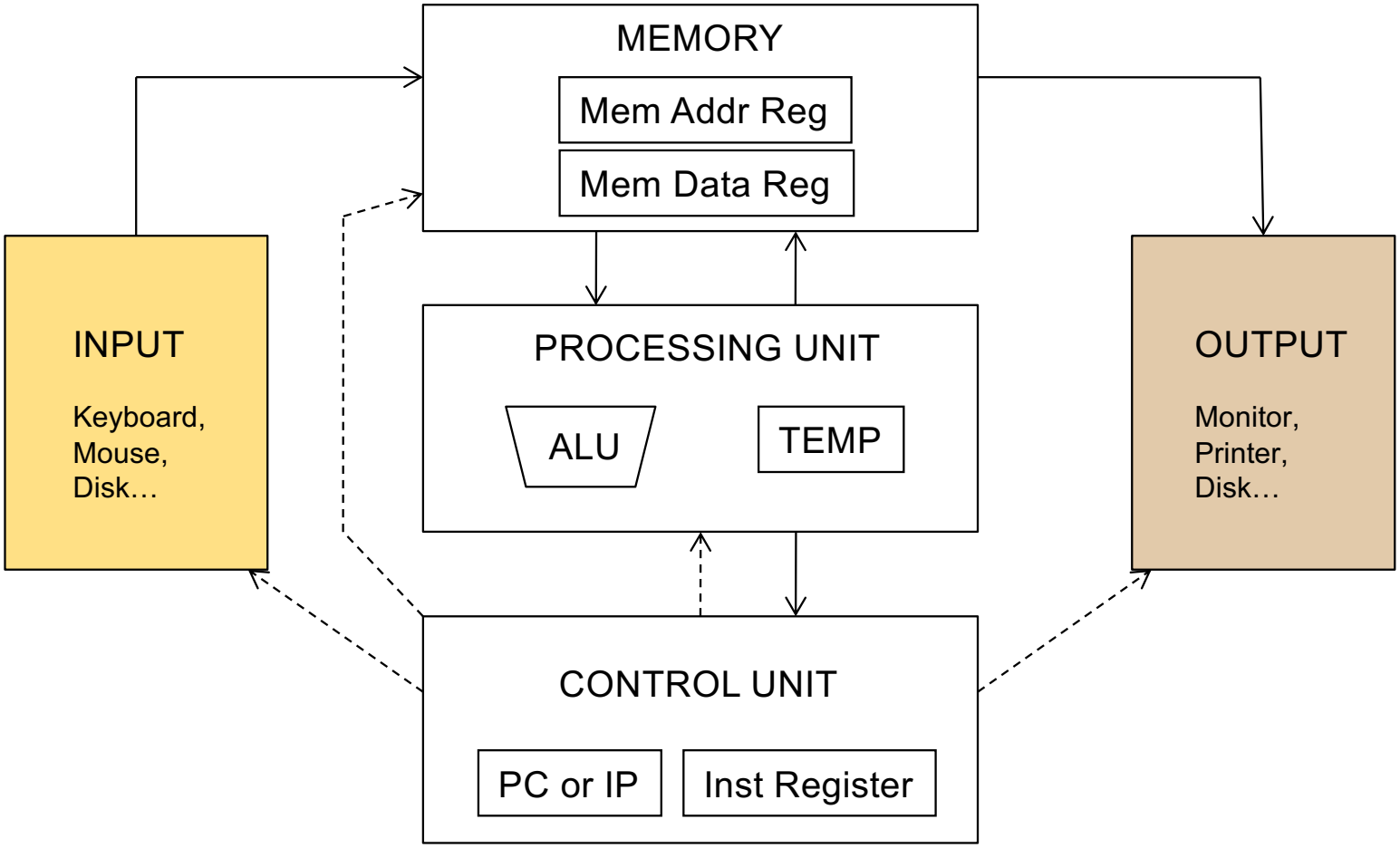
Name	Register Number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	function return value
`$a0-$a3`	4-7	function arguments
`$t0-$t7`	8-15	temporary variables
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	temporary variables
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	function return address

ARM Register File

Table 6.1 ARM register set

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

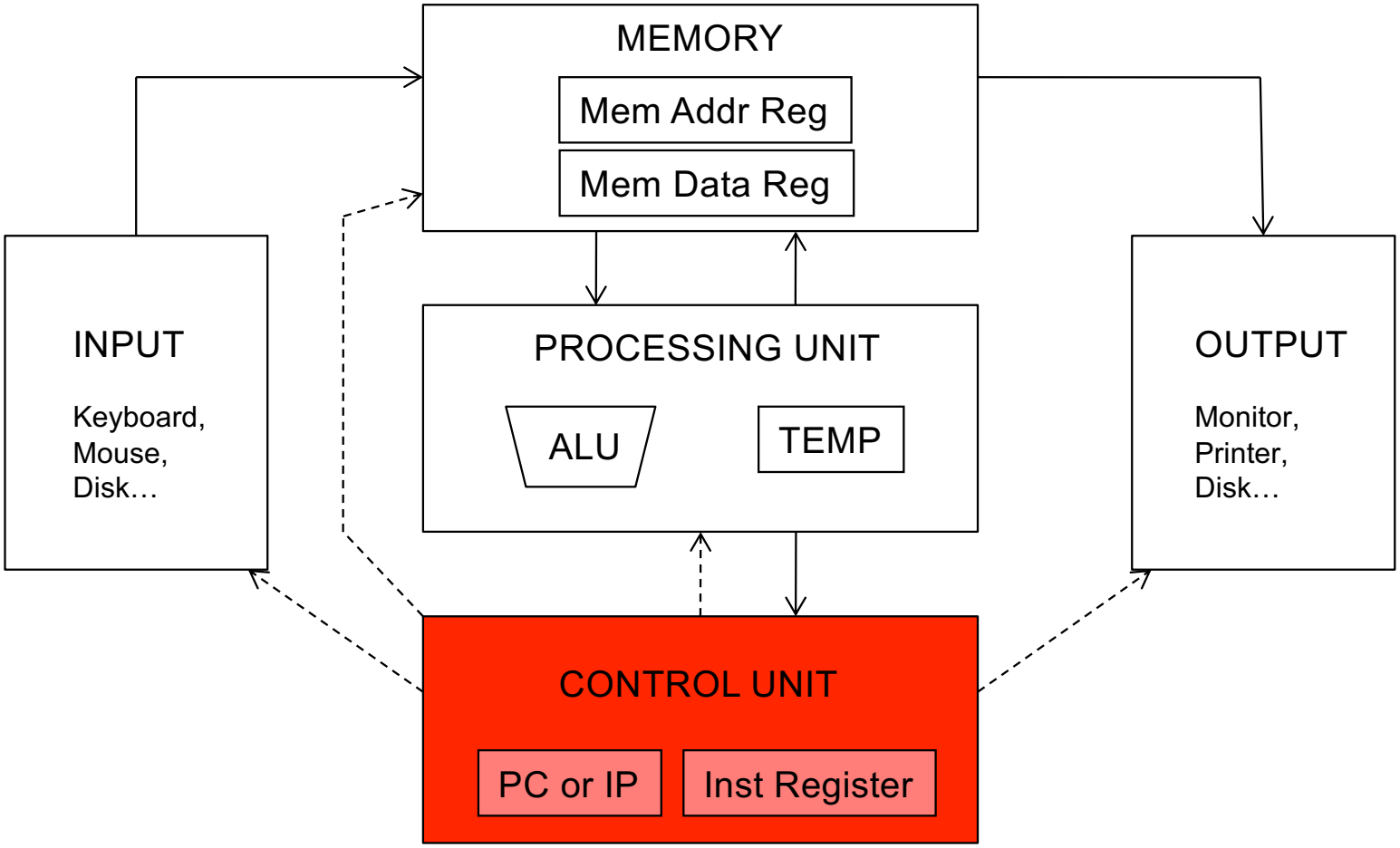
The Von Neumann Model



Input and Output

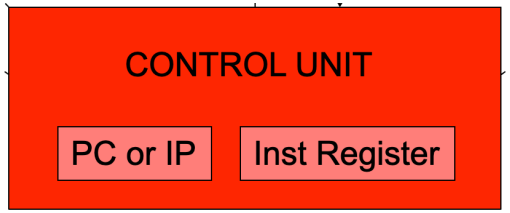
- Enable information to get into and out of a computer
- Many devices can be used for input and output
- They are called peripherals
  - Input
    - Keyboard
    - Mouse
    - Scanner
    - Disks
    - Etc.
  - Output
    - Monitor
    - Printer
    - Disks
    - Etc.

The Von Neumann Model



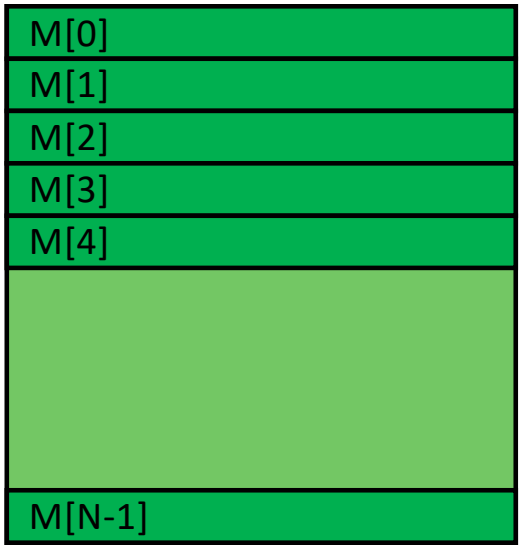
Control Unit

- The control unit is like the conductor of an orchestra
- It conducts the step-by-step process of executing (every instruction in) a program



- It keeps track of which instruction being processed, via
  - Instruction Register (IR), which contains the instruction
- It also keeps track of which instruction to process next, via
  - Program Counter (PC) or Instruction Pointer (IP), another register that contains the address of the (next) instruction to process

Programmer Visible (Architectural) State



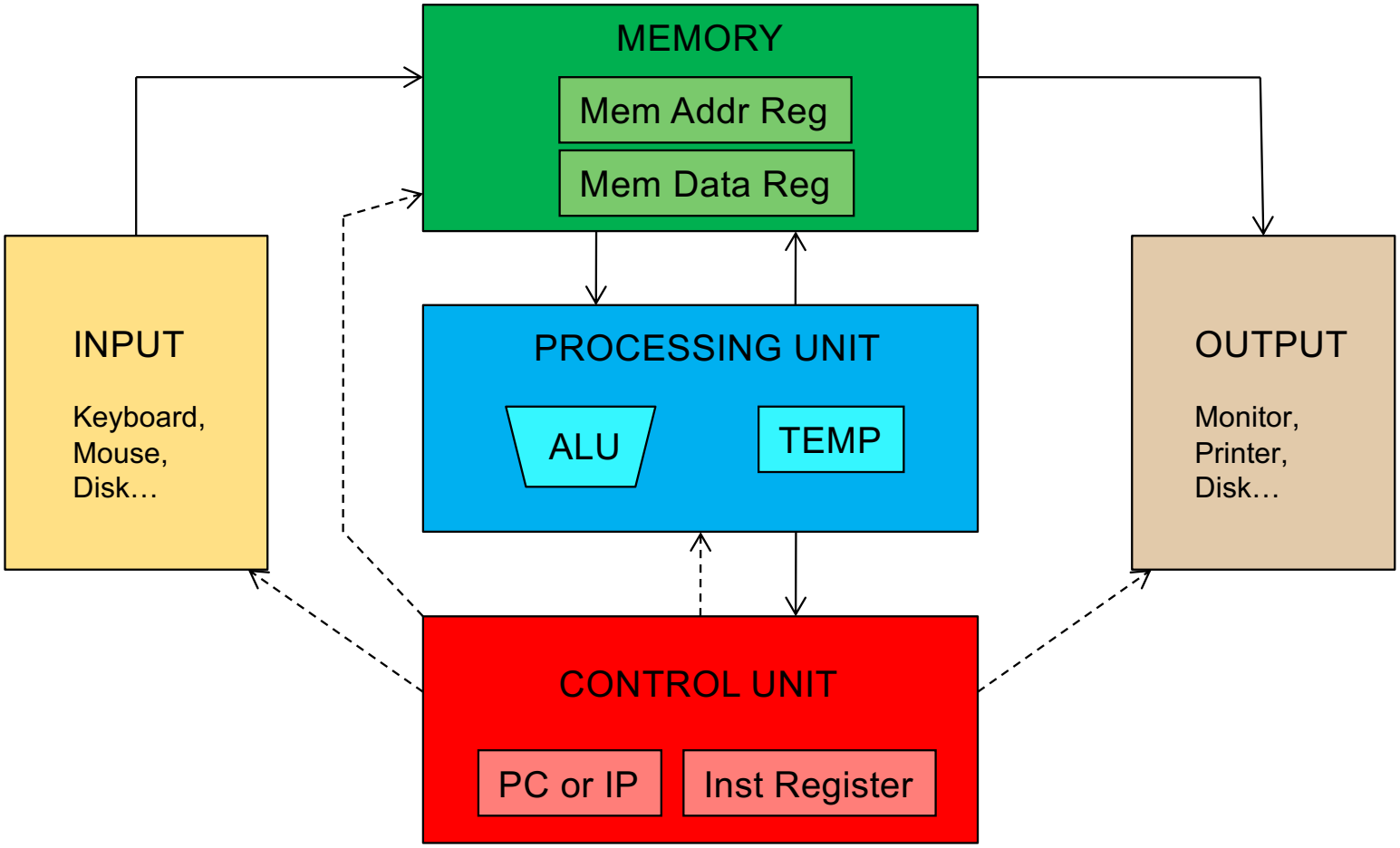
Memory
array of storage locations
indexed by an address

Registers
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Program Counter
memory address
of the current (or next) instruction

Instructions (and programs) specify how to transform
the values of programmer visible state

The Von Neumann Model



Von Neumann Model: Two Key Properties

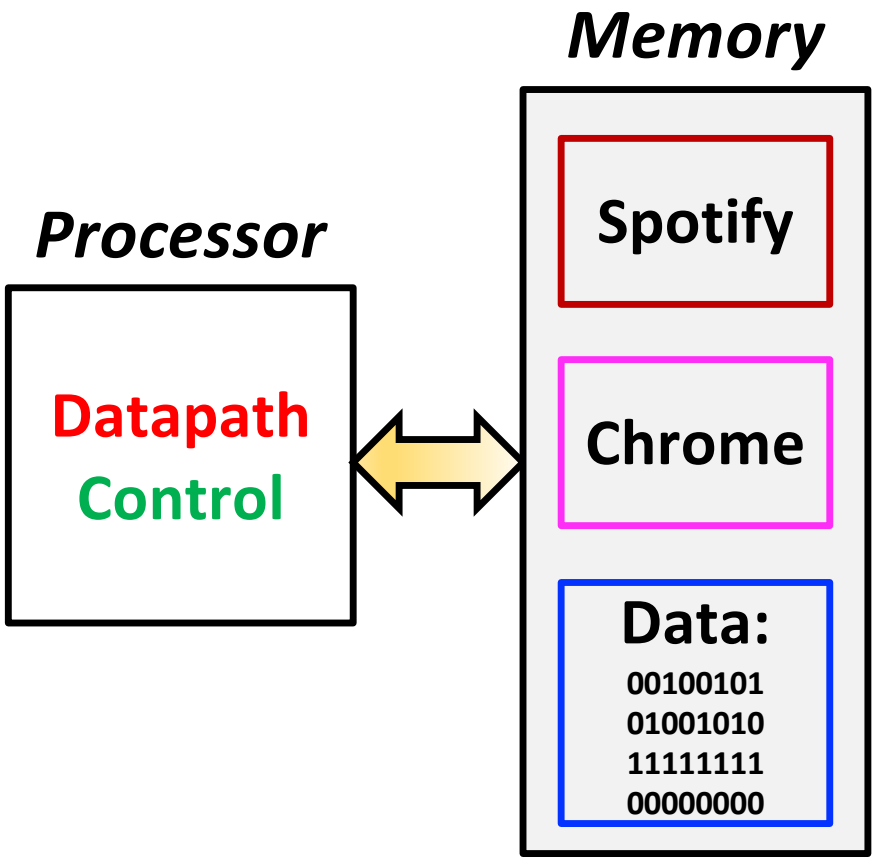
- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

- Stored program
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - The interpretation of a stored value depends on the control signals

- Sequential instruction processing
  - One instruction processed (fetched, executed, completed) at a time
  - Program counter (instruction pointer) identifies the current instruction
  - Program counter is advanced sequentially except for control transfer instructions

Von Neumann Model: Two Key Properties

- **Spotify** and **Chrome** are stored as instructions in memory
- Some *arbitrary* data is also stored in memory
- Both instructions and data appear as a sequence of 0's and 1's in memory



Examples of

von Neumann Machines

LC-3: A von Neumann Machine

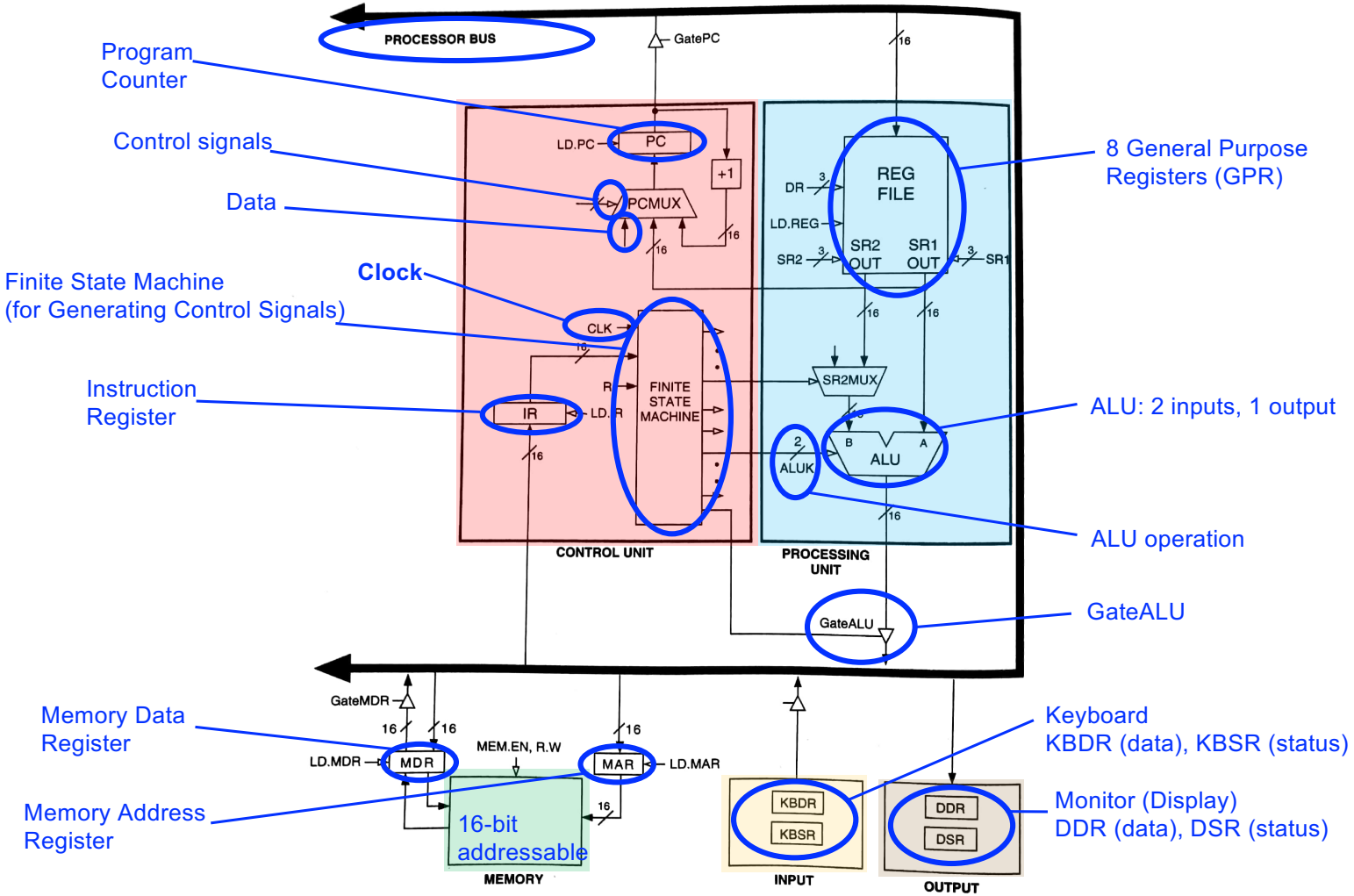
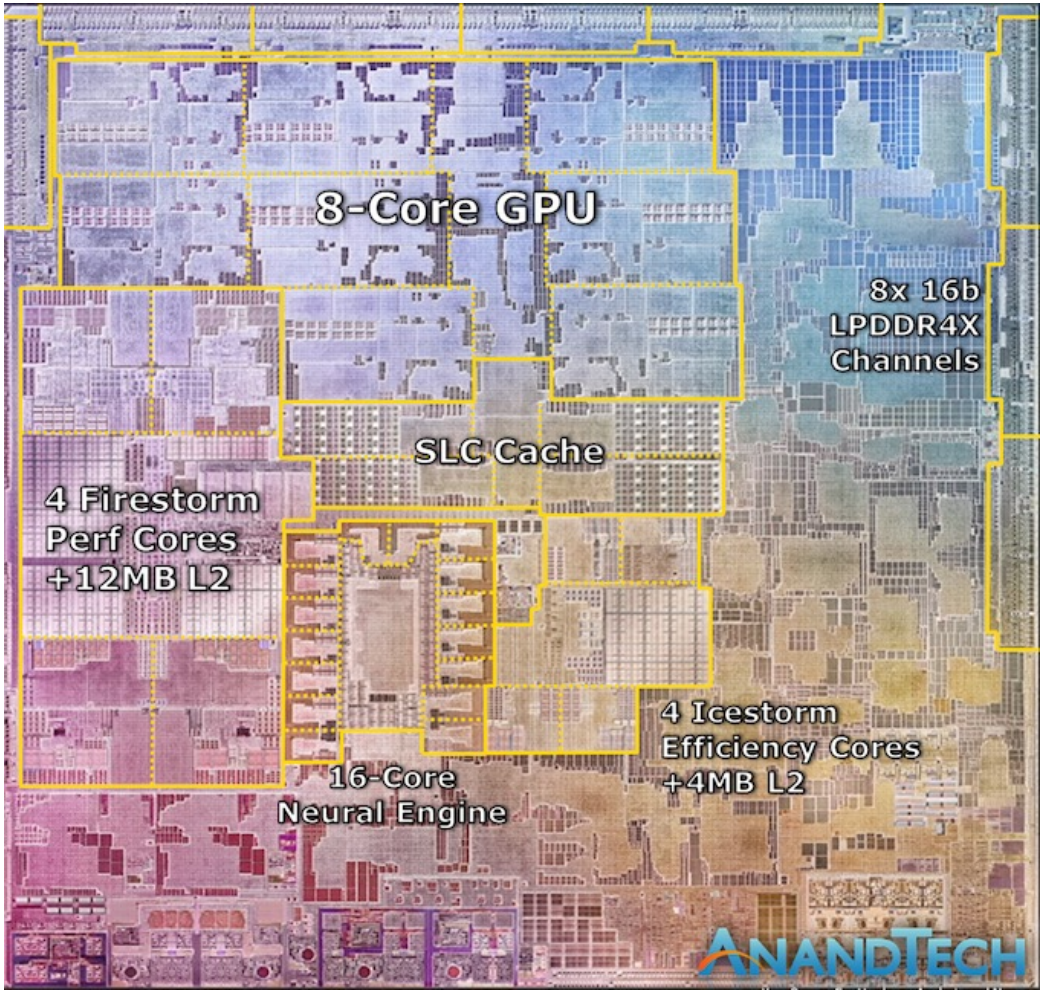


Figure 4.3 The LC-3 as an example of the von Neumann model

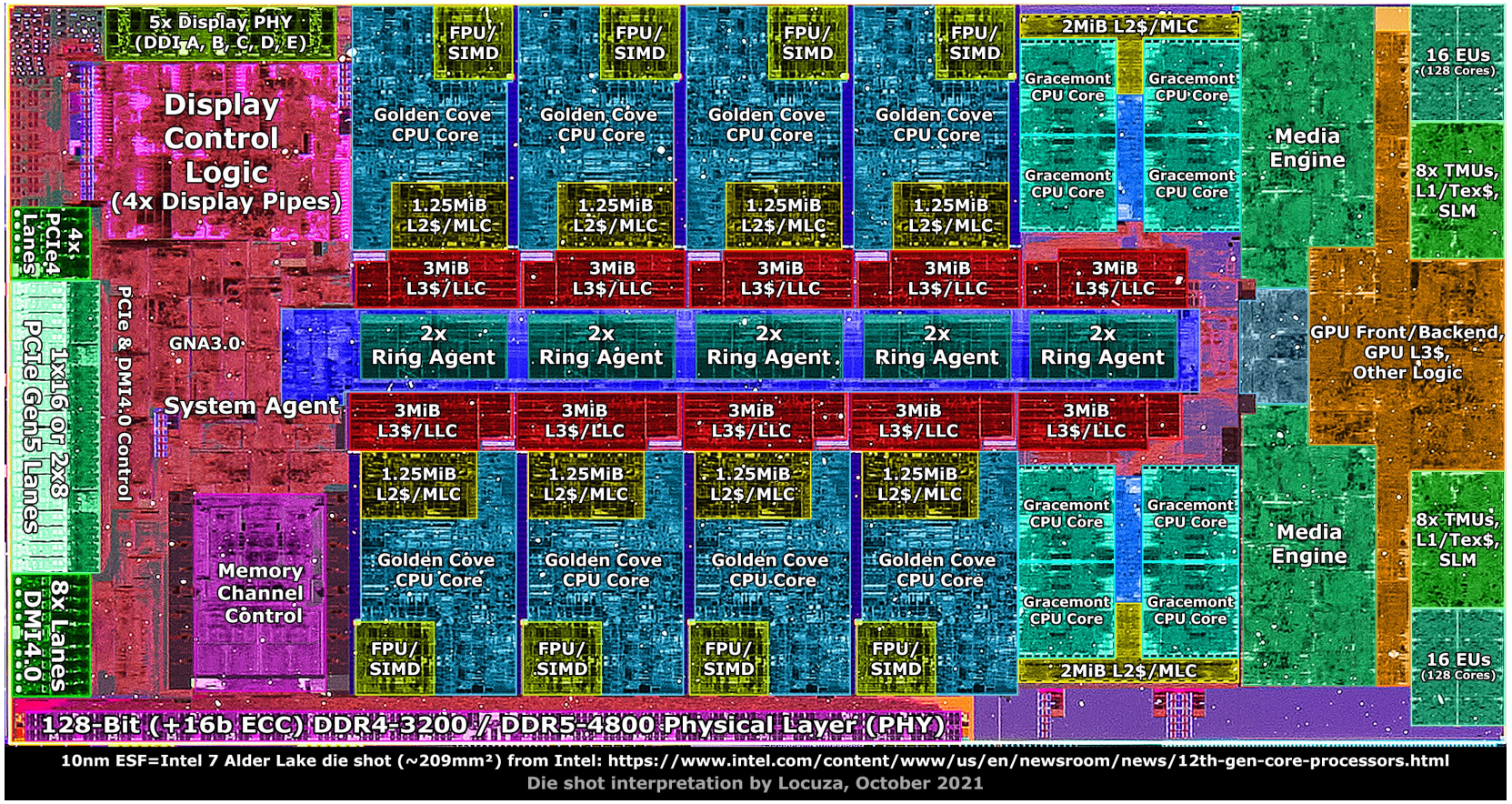
Another Von Neumann Machine



Apple M1, 2021

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

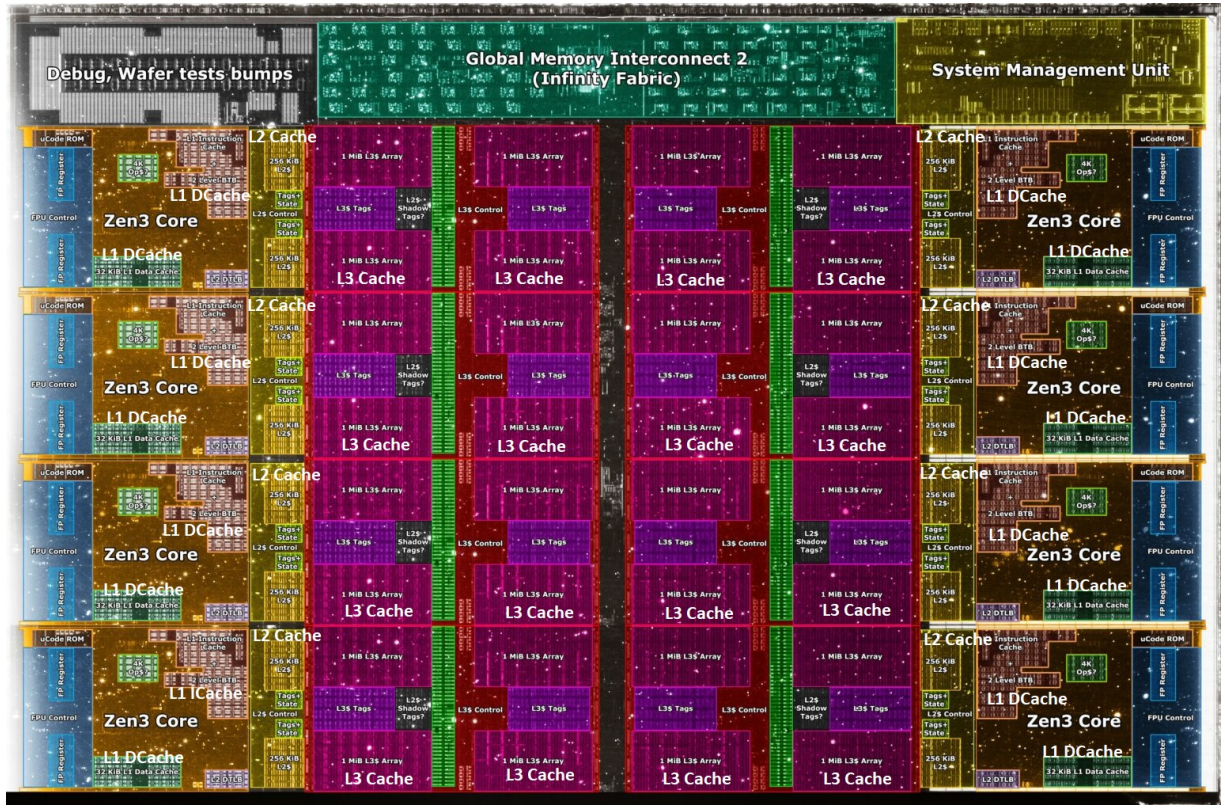
Another Von Neumann Machine



Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652

Another Von Neumann Machine



AMD Ryzen 5000, 2020

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

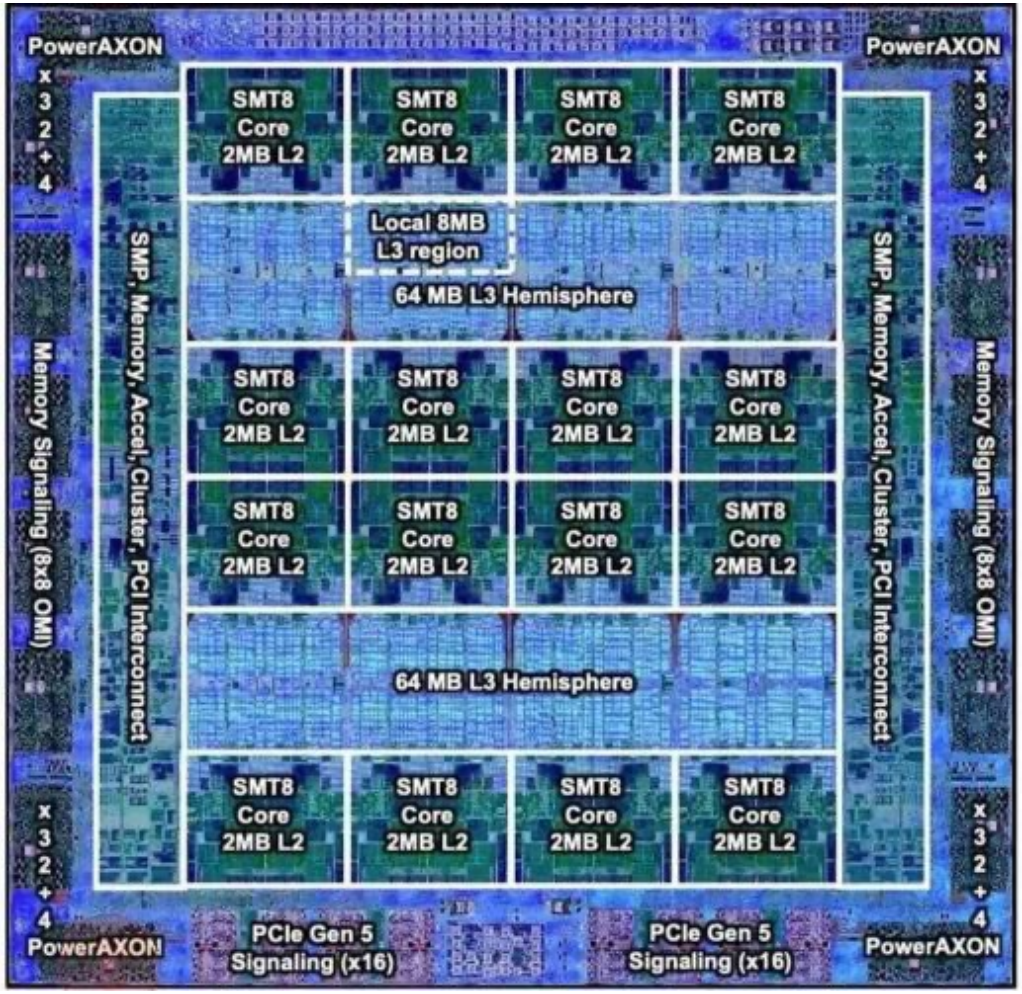
Core Count:
8 cores/16 threads

L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

Another Von Neumann Machine



IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:

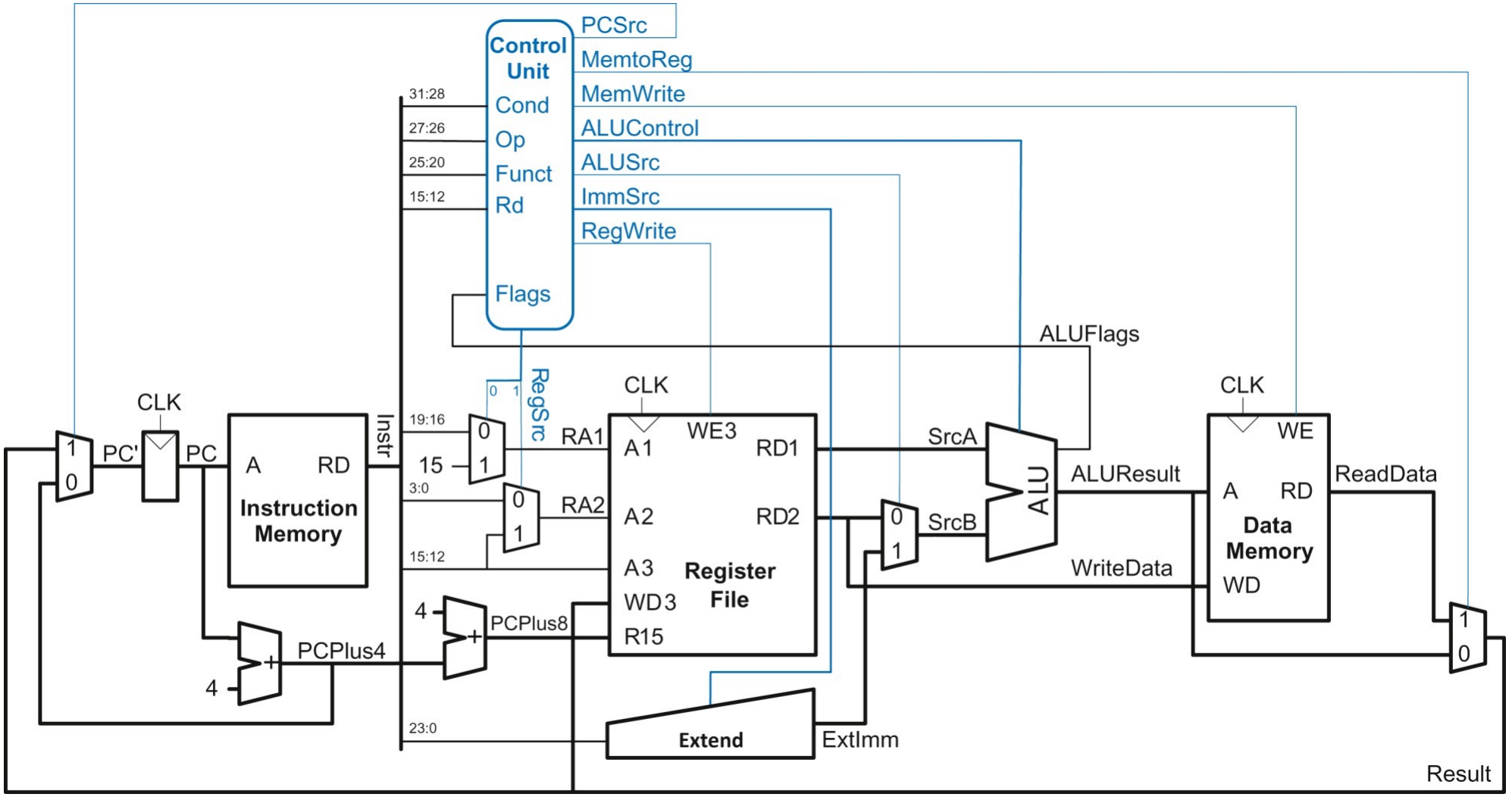
2 MB per core

L3 Cache:

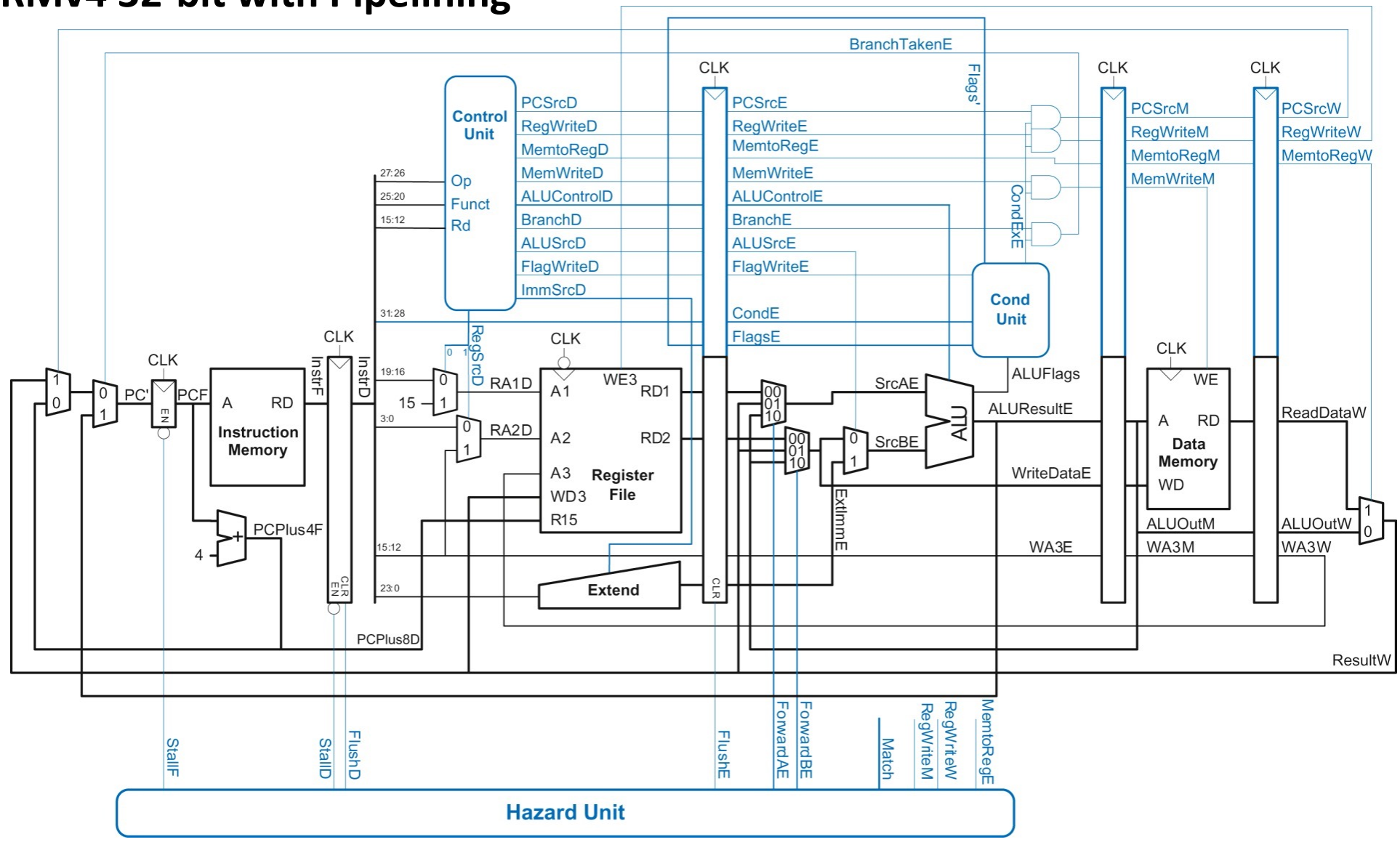
120 MB shared

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

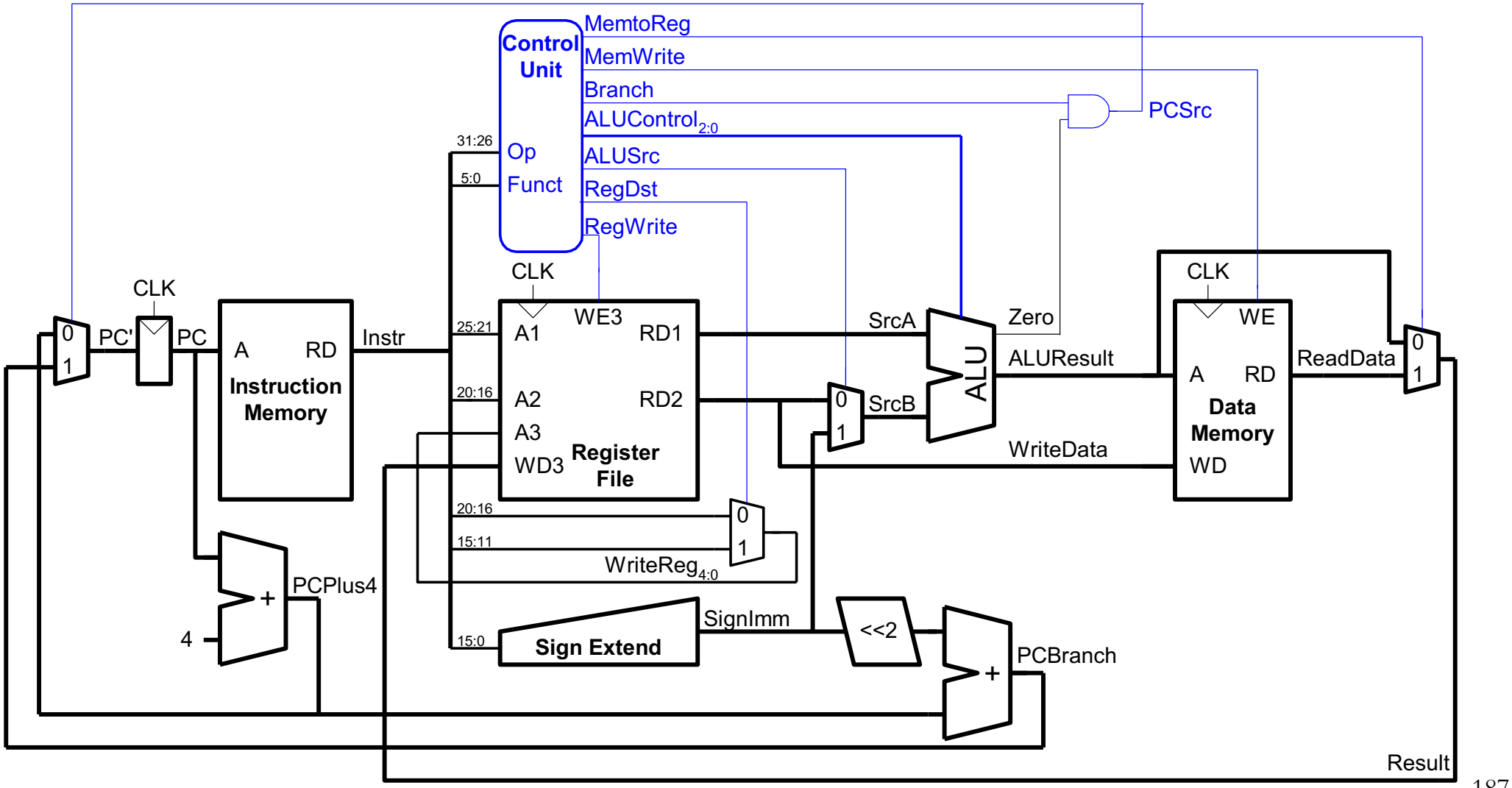
ARMv4 32-bit


Control Unit
Cond
Op
Funct
Rd
Flags
PCSrc
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
ALUFlags
31:28
27:26
25:20
15:12
19:16
15
3:0
15:12
23:0
RegSrc
CLK
PC'
PC
A
RD
Instruction Memory
Instr
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
ALUResult
WE
Data Memory
WD
ReadData
WriteData
4
PCPlus4
PCPlus8
Extend
ExtImm
Result

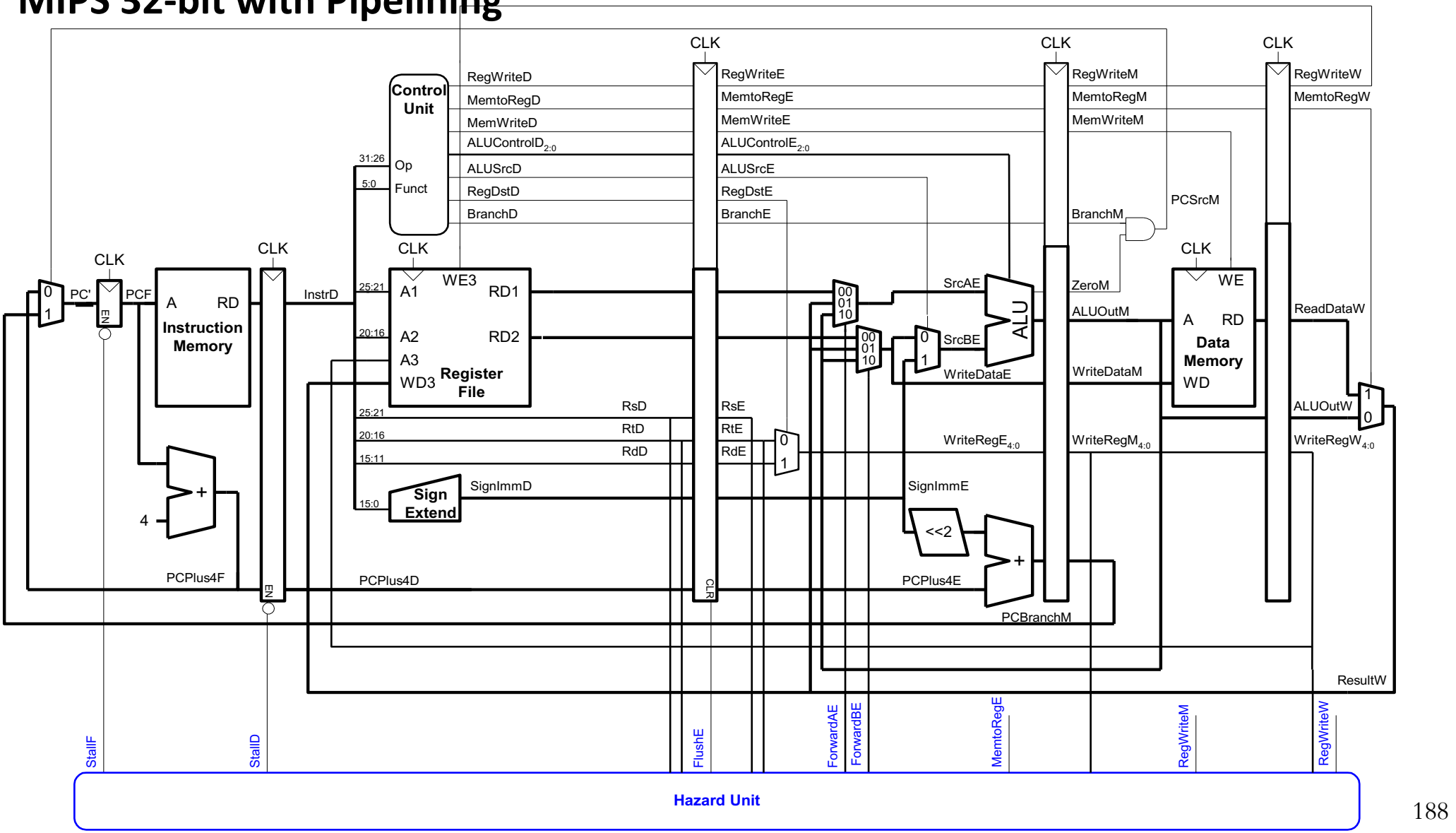
ARMv4 32-bit with Pipelining



MIPS 32-bit


Control Unit
MemtoReg
MemWrite
Branch
ALUControl2:0
ALUSrc
RegDst
RegWrite
Op
Funct
31:26
5:0
PCSrc
CLK
PC'
PC
0
1
A
RD
Instruction Memory
Instr
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
WE
Data Memory
WD
ReadData
20:16
15:11
WriteReg4:0
PCPlus4
4
+
15:0
Sign Extend
SignImm
<<2
PCBranch
Result

MIPS 32-bit with Pipelining



Stored Program and Sequential Execution

- Instructions and data are stored in memory
  - Typically the instruction length is the word length
- The processor fetches instructions from memory sequentially
  - Fetches one instruction
  - Decodes and executes the instruction
  - Continues with the next instruction
- The address of the current instruction is stored in the program counter (PC)
  - If word-addressable memory, the processor increments the PC by 1 (in QuAC)
  - If byte-addressable memory, the processor increments the PC by the instruction length in bytes (4 in MIPS and ARM)
    - In MIPS the OS typically sets the PC to 0x00400000 (start of a program)

A sample program stored in memory

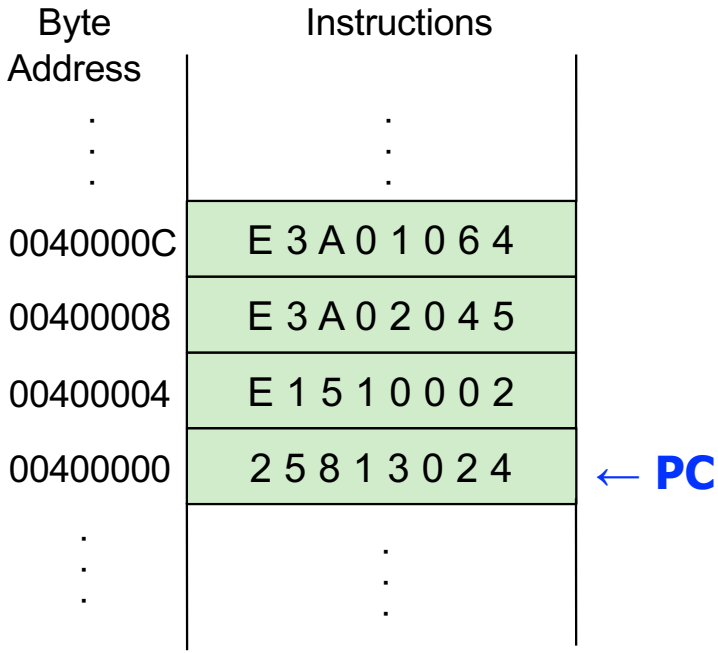
- A sample ARM program
  - 4 instructions stored in consecutive words in memory
    - No need to understand the program now. We will get back to it

ARM assembly code

```
MOV   R1,  #100
MOV   R2,  #69
CMP   R1,  R2
STRHS R3,  [R1, #0x24]
```

Machine code (encoded instructions)

```
0xE3A01064
0xE3A02045
0xE1510002
0x25813024
```



A sample program stored in memory

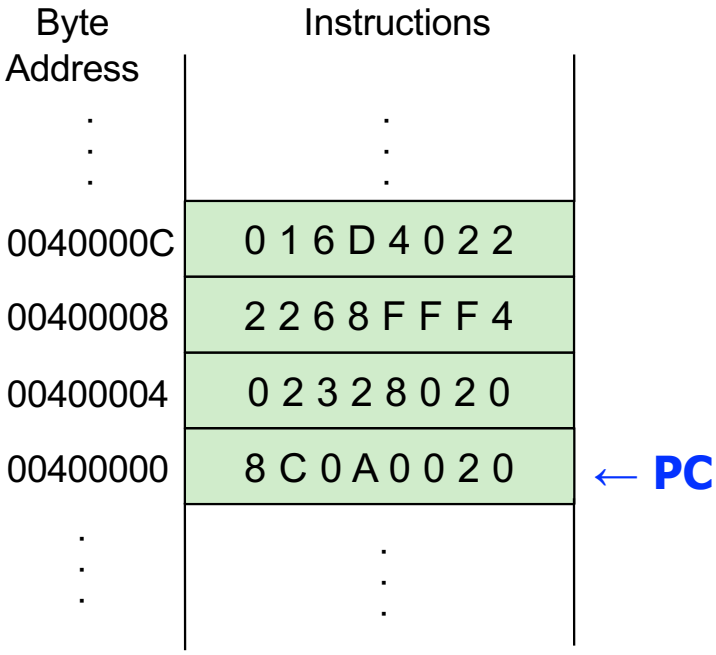
- A sample ARM program
  - 4 instructions stored in consecutive words in memory
    - No need to understand the program now. We will get back to it

MIPS assembly

```
lw   $t2, 32($0)
add  $s0, $s1, $s2
addi $t0, $s3, -12
sub  $t0, $t3, $t5
```

Machine code (encoded instructions)

```
0x8C0A0020
0x02328020
0x2268FFF4
0x016D4022
```



The Instruction

- An instruction is the most basic unit of computer processing
  - Instructions are words in the language of a computer
  - Instruction Set Architecture (ISA) is the vocabulary

- The language of the computer can be written as

  - Machine language: Computer-readable representation (that is, 0's and 1's)

  - Assembly language: Human-readable representation

- We will study ARM and (some) MIPS instructions
  - Principles are similar in all ISAs (x86, SPARC, RISC-V, …)

The Instruction: Opcode & Operands

- An instruction is made up of two parts
  - Opcode and Operands

- Opcode specifies what the instruction does
- Operands specify who the instruction is to do it to

- Both are specified in instruction format (or instr. encoding)
  - An MIPS and ARM instructions consists of 32 bits (bits [31:0])
  - MIPS example below: Bits [31:26] specify the opcode → up to 64 distinct opcodes
  - Bits [25:11] are used to figure out where the operands are

R-Type

Instruction Types

- There are three main types of instructions
- Operate (data processing) instructions
  - Execute operations in the ALU
- Data movement (memory) instructions
  - Read from or write to memory
- Control flow (branch/jump) instructions
  - Change the sequence of execution (decision making)
- Let us start with some example instructions

An Example Operate Instruction

- Addition

High-level code

```
a = b + c;
```

Assembly

```
ADD a, b, c
```

- ADD: mnemonic to indicate the operation to perform
- b, c: source operands
- a: destination operand
- a ← b + c

Registers

- We map variables to registers

Assembly

```
add a, b, c
```

ARM registers

```
b = R1
c = R2
a = R0
```

MIPS registers

```
b = $s1
c = $s2
a = $s0
```

From Assembly to Machine Code

- Addition

ARM assembly

```
ADD   R0, R1, R2
```

Instruction Fields

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op	I	cmd	S	Rn	Rd	Src2

Machine Code (Instruction Encoding)

31:28	27:26	25	24:21	20	19:16	15:12	11:0
1110	00	0	0100	1	0001	0000	000000000010

Machine Code (In short, hexadecimal)

0x E 0 9 1 0 0 0 2

Instruction Format (or Encoding)

- op = opcode (what the instruction does?)
  - 00 means operate instruction; cmd = 0100 means ADD
  - Some bits are pre-set (details later)

- Format (Encoding)

ADD R0, R1, R3

ADD Rd, Rn, Rm

Semantics:
Rd = Rn + Rm

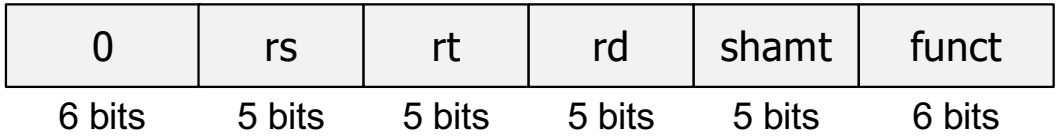
31:28	27:26	25	24:21	20	19:16	15:12	11:4								3:0
1110	op	0	cmd	S	Rn	Rd	0	0	0	0	0	0	0	0	Rm

- Rn and Rm are source registers
- Rd is destination register

DP – Format Instructions

Instruction Format: R Type in MIPS

- MIPS R-type Instruction Format
  - 3 register operands



  - 0 = opcode
  - rs, rt = source registers
  - rd = destination register
  - shamt = shift amount (only shift operations)
  - funct = operation in R-type instructions

Read Operands from Memory

- With operate instructions, such as addition, we tell the computer to execute arithmetic (or logic) computations in the ALU

- We also need instructions to access the operands from memory
  - Load them from memory to registers
  - Store them from registers to memory

- Next, we see how to read (or load) from memory

- Writing (or storing) is performed in a similar way, but we will talk about that later

Load Word in ARM

- ARM assembly (assuming word-addressable)

High-level code

```
a = A[2];
```

ARM assembly

```
LDR  R3, [R0, #2]
```

R3 ← Memory[R0 + 2]

- MIPS assembly (assuming word-addressable)

High-level code

```
a = A[2];
```

MIPS assembly

```
lw   $s3, 2($s0)
```

$s3 ← Memory[$s0 + 2]

These instructions use a particular addressing mode
(i.e., the way the address is calculated), called base+offset

Load Word in Byte-Addressable MIPS

- MIPS assembly

High-level code

```
a = A[2];
```

MIPS assembly

```
lw   $s3, 8($s0)
```

$s3 ← Memory[$s0 + 8]

- Byte address is calculated as: word_address * bytes/word
  - 4 bytes/word in MIPS

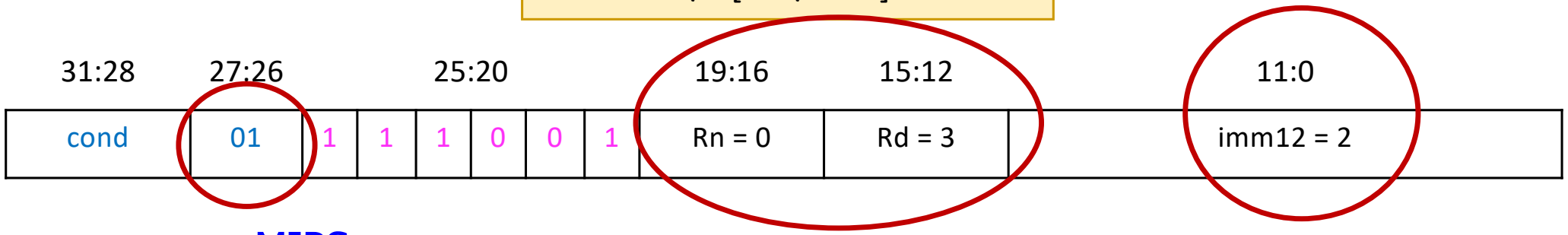
Instruction Format with Immediate

Memory – Format Instructions

- ARM

ARM assembly

```
LDR   R3, [R0, #2]
```

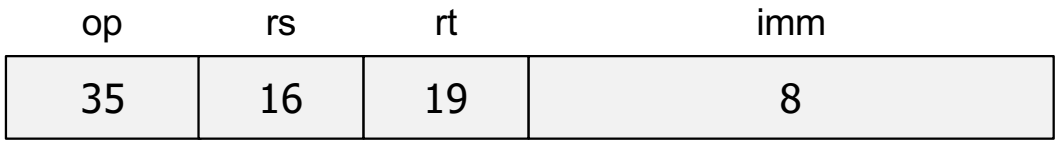


- MIPS

MIPS assembly

```
lw   $s3, 8($s0)
```

Field Values



I-Type

Instruction (Processing) Cycle

How are these instructions executed?

- By using instructions, we can speak the language of the computer
- Thus, we now know how to tell the computer to
  - Execute computations in the ALU by using, for instance, an addition
  - Access operands from memory by using the load word instruction
- But, how are these instructions executed on the computer?
  - The process of executing an instruction is called is the instruction cycle (or, instruction processing cycle)

The Instruction Cycle

- The instruction cycle is a sequence of steps or phases, that an instruction goes through to be executed
  - FETCH
  - DECODE
  - EVALUATE ADDRESS
  - FETCH OPERANDS
  - EXECUTE
  - STORE RESULT

- Not all instructions require the six phases
  - LDR does **not** require EXECUTE
  - ADD does **not** require EVALUATE ADDRESS
  - Intel x86 instruction ADD [eax], edx is an example of instruction with six phases

After STORE RESULT, a NEW FETCH

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

FETCH

- The FETCH phase obtains the instruction from memory and loads it into the Instruction Register (IR)
- This phase is common to every instruction type
- Complete description
  - Step 1: Load the MAR with the contents of the PC, and simultaneously increment the PC
  - Step 2: Interrogate memory. This results in the instruction being placed in the MDR by memory
  - Step 3: Load the IR with the contents of the MDR

DECODE

- The DECODE phase identifies the instruction
  - Also generates the set of control signals to process the identified instruction in later phases of the instruction cycle
- Recall the decoder
  - A 4-to-16 decoder identifies which of the 16 opcodes is going to be processed
- The input is the four bits IR[15:12]
- The remaining 12 bits identify what else is needed to process the instruction

EVALUATE ADDRESS

- The EVALUATE ADDRESS phase computes the address of the memory location that is needed to process the instruction
- This phase is necessary in LDR
  - It computes the address of the data word that is to be read from memory
  - By adding an offset to the content of a register
- But not necessary in ADD

FETCH OPERANDS

- The FETCH OPERANDS phase obtains the source operands needed to process the instruction

- In LDR
  - Step 1: Load MAR with the address calculated in EVALUATE ADDRESS
  - Step 2: Read memory, placing source operand in MDR

- In ADD
  - Obtain the source operands from the register file
  - In some microprocessors, operand fetch from register file can be done at the same time the instruction is being decoded

EXECUTE

- The EXECUTE phase executes the instruction
  - In ADD, it performs addition in the ALU
  - In XOR, it performs bitwise XOR in the ALU
  - …

STORE RESULT

- The STORE RESULT phase writes the result to the designated destination

- Once STORE RESULT is completed, a new instruction cycle starts (with the FETCH phase)

The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

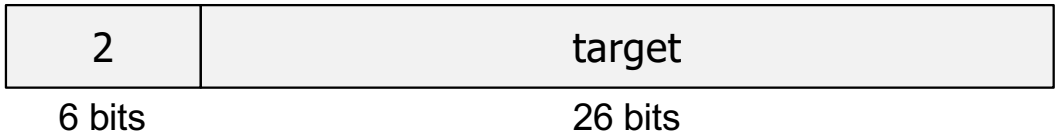
Changing the Sequence of Execution

- A computer program executes in sequence (i.e., in program order)
  - First instruction, second instruction, third instruction and so on
- Unless we change the sequence of execution
- Control instructions allow a program to execute out of sequence
  - They can change the PC by loading it during the EXECUTE phase
  - That wipes out the incremented PC (loaded during the FETCH phase)

Jump in ARM

- Unconditional branch or jump
- MIPS

```
j   target
```



J-Type

  - 2 = opcode
  - target = target address
  - PC ← PC† [31:28] | sign-extend(target) * 4
  - Variations
    - jal: jump and link (function calls)
    - jr: jump register

```
jr   $s0
```

j uses pseudo-direct addressing mode

jr uses register addressing mode

† This is the incremented PC

The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

The Instruction Cycle

- We have described a typical microarchitecture “plan”
  - Not all instructions need all phases
  - The ordering of phases in not set in stone
  - Some phases can be grouped as one
  - Some structured may not be needed
    - Microarchitecture style dictates many details (week 5 and 6)

ARM Instruction Set Architecture (ISA)

The Instruction Set

- It defines opcodes, data types, and addressing modes

- ADD and LDR have been our first examples

The Instruction Set Architecture

- The ISA is the interface between what the software commands and what the hardware carries out

- The ISA specifies
  - The memory organization
    - Address space (ARM: 2^{32}, MIPS: 2^{32})
    - Addressability (ARM: 8 bits, MIPS: 8 bits, QuAC: 16 bits)
    - Word- or Byte-addressable
  - The register set
    - R0 to R15 in ARM
    - 32 registers in MIPS
  - The instruction set
    - Opcodes
    - Data types
    - Addressing modes
    - Length and format of instructions