

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoib Akram

shoib.akram@anu.edu.au



Australian
National
University

Shoaib Akram

Lecturer, School of Computing, (Jan 2020 –)

Ph.D., 2019

Teaching: Computer Microarchitecure (semester 2)

Interests: Hardware/software interaction and performance analysis

It is an interesting time to learn and research about computer systems. Traditional semiconductor laws are breaking down. But the society needs more compute power and storage capacity: Big Data, AI/ML, communication needs, among others.

My current focus is on enabling fast access to large amounts of information (data) using emerging memory and storage devices.

Logistics

Course webpage: <https://comp.anu.edu.au/courses/engn2219/>

Lectures (on the website)

- Lecture slides
- Lecture videos (I will do my own recording)
- Weekly problem sets (for your practice only, *Not Graded*)
- Key Ideas and summary

Policies (will be up shortly)

- General conduct, assignment groups/submissions, support, management, grading

Resources

- Frequently asked questions
- Writing design documents
- Stuff needed to finish the assignments

Piazza

I will use Piazza for all communication

- If you ignore Piazza, *you will miss key announcements*
 - Drop-in sessions, make-up lectures, problems, exercises, corrections, lecture timing (ENGN2218 conflict)
- Ask questions on Piazza first (most likely you will receive a response quickly)
- Post solutions to weekly problems, ask your classmates if you are on the right track
- *Ask private questions on Piazza to instructors*
- Students are added/dropped automatically

Tutorials/Labs

Labs are a critical component of this course (one every week)

Handout will be posted on the website “Labs” before each lab

First six labs

- In each lab, you will finish a sub-component of design assignment 1
- If you finish the first six labs, you will finish ~50% of the first assignment
- We only mark the week 2 – 3 labs to make sure you are making progress and to give you feedback (due Monday 6 pm week 4)
- Week 3 lab accounts towards 5 points (out of 100) for assignment 1

Week 7 lab (after the teaching break)

- Test your design project

Week 8 – 11 Labs

- C programming (*not really about C, but about learning key computer systems concepts, more on this later*)

Assessments

Two assignments (60%)

- CPU design assignment (30%)
 - 5% due on 6 pm, Monday of week 4
 - Full assignment due on 12 pm, Monday of week 8
- Programming assignment (30%)
 - Due date: Monday 6 pm, week 13

Final Exam (40%)

- I will release problem sets and exercises throughout the course for preparation
- Mock-up exams will be available during/after the break

Assignment Submission

Extensions will be granted on a per-request basis

- Via Email: Shoaib.Akram@anu.edu.au

Assignment submissions are handled via Gitlab

- You will learn about it in the labs
- Make a habit of using Git properly
- Push often, always pull the latest

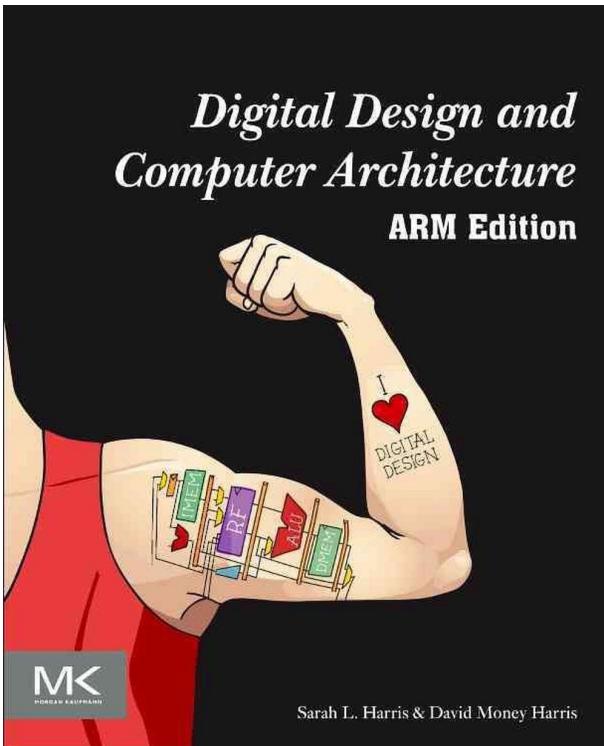
You can form group of up to two students to work
on the assignment (one submission per group)

Ink/Whiteboard in Lectures

Try to take notes

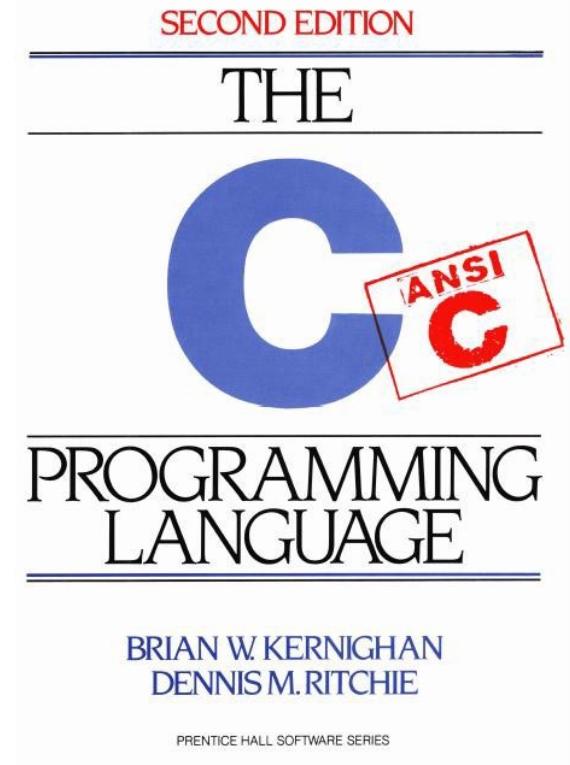
- Help you think and most of the time I will do this is to solve a problem
- Ink + Wacom (not very stable)

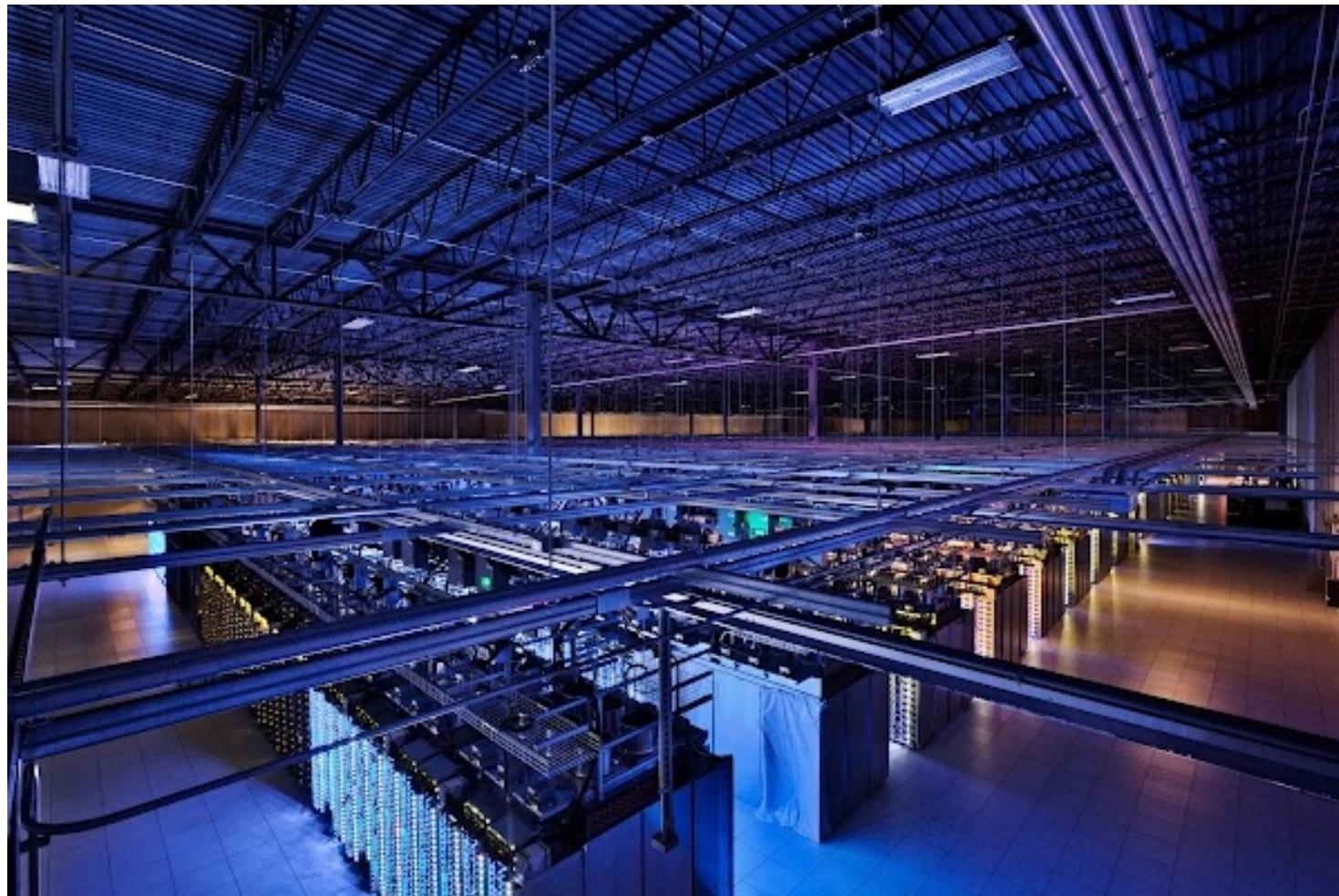
Textbook



- Freely available online (*check Piazza or course webpage*)
- *I will post the chapters/sections on the Lectures page after the lecture*

Kernighan & Ritchie, The C Programming Language, 2nd Edition
• “ANSI” (old-school) C





Council Bluffs, Iowa data center, Google (115, 000 sq. feet)



**Self-flying nano drone
94 milli-watts**

Research server for my
students with special
memory & storage
devices



Fundamentals are important

All computer systems, big or small, have a few fundamental components

- **Microprocessor** (processor or central processing unit or CPU) for doing computation
- **Main memory** for storing temporary information and program data close to the processor
- **Storage** devices (e.g., disks or SSDs) for storing long-term or persistent information
- **I/O devices** to communicate with the external environment
 - Sensors
 - Peripherals

Most computer systems can be viewed as below

- Three key resources: CPU, memory, storage
- CPU is the heart of a computer system
- Processor can access memory much faster than storage



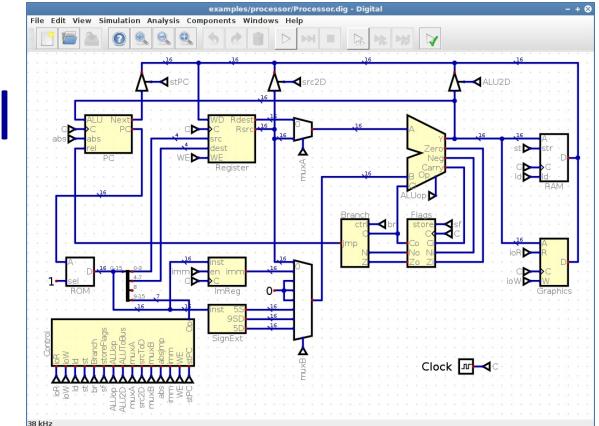
This Course is about ...

How does the general-purpose processor work? How do modern processors perform a wide variety of tasks?

How do processors interact with main memory and storage? How does the memory and storage system work?

The best way to learn how something works is to build one

- You will build a processor in Digital
- URL: <https://github.com/hneemann/Digital>



This Course is also about ...

A computer system is more than just hardware

- How does hardware and software interact?
- What should programmers know about hardware?
- C is a good vehicle for answering the above questions
- You can talk about hardware resources in high-level terms but still stay close to the hardware
- Key learning outcome of this course: *How can you shoot yourself in the foot when writing C programs?*
- **Remember:** It's not about C or Java or Python. It's about gaining a deep insight into computer systems!

A 5-Step Recipe for Failure

Stay out of the loop

- Do not check Piazza
- Do not ask questions
- Do not care what is going on in lectures

And do not learn to use Gitlab

Do not attempt end of week problem sets

- Okay, not every week

Do not come to labs OR do not read/attempt the lab handouts

- And start the assignment on your own the weekend before due date (no way!)

Do not seek help from tutors

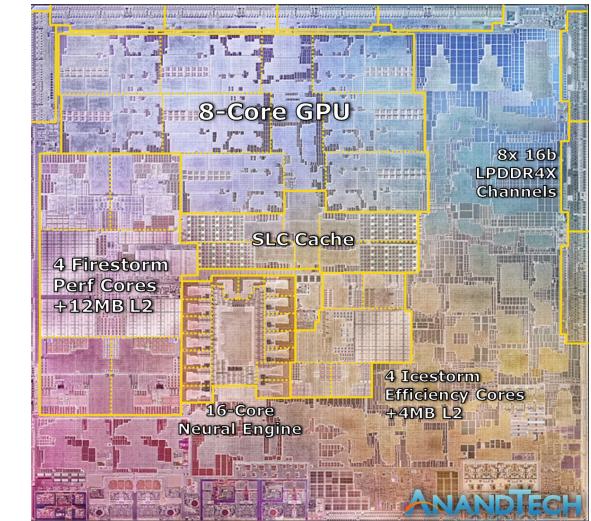
Do not communicate your problems to me!



How do engineers manage complexity?

- Look at components from a higher level
- Get into detail if necessary

No human (programmer) can track
10 billion elements. **Computer systems
work because of abstraction!**



Apple M1 Chip
Billions of transistors
All working in parallel

Transformation Hierarchy

- We think of problems in English
 - Sort students by their UIDs
- The actual work is done by electrons
 - Do electrons speak English?
- How do we make the electrons do the work?
 - *We use a systematic transformation hierarchy to transform the problem in English into electron movement*
 - *This transformation hierarchy is driven by our need to abstract away complexity*



Problem

Algorithm

Program

Architecture

micro-arch

circuits

devices



Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Programs

Device Drivers

Instructions
Registers

Datapaths
Controllers

Adders
Memories

AND Gates
NOT Gates

Amplifiers
Filters

Transistors
Diodes

Electrons

Definitions

Abstraction: Hiding details to view the system from a high level

Deconstruction: Going from abstraction back to its component parts (breaking abstraction)

Low-level language: Languages that are tied to the machine architecture. Each architecture supports at least one low-level language called assembly.

High-level language: Programming languages that are at distance higher than the architecture. They are machine-independent. E.g., C, Java, Python, Rust, Ruby, Go

Instruction Set Architecture: A specification of all the instructions a processor can perform. Each instruction is an arithmetic (add, sub) or a data movement (fetch from memory) operation.

Microarchitecture: ISA has no physical significance. Microarchitecture is the physical implementation of an ISA.

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Programs

Device Drivers

Instructions
Registers

Datapaths
Controllers

Adders
Memories

AND Gates
NOT Gates

Amplifiers
Filters

Transistors
Diodes

Electrons

Week 4

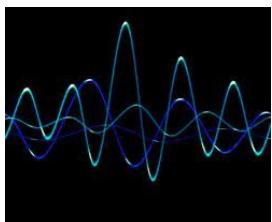
Week 5 – 6

Weeks 2 – 3

Week 1

Representing Information

Question: How many different values can each of these *physical* variables take?



Frequency of oscillation



Voltage on a wire



Temperature

Answer: Infinite

All these are continuous signals
They contain infinite amount of information

Representing Information

Digital Systems: Represent information with discrete-valued variables, i.e., variables with a finite # distinct values

Modern digital systems use a **binary (*two-valued*)** representation



0 1

0 1

0 1

Binary Representation

Digital systems internally use “voltages” for representing binary variables

- Low voltage means 0
- High voltage means 1

B I N A Y D I G I T

A **bit** is a unit of information. A *binary variable represents one bit of information. To represent discrete sets with more than two elements, we combine multiple bits into a binary code*

Binary Codes

Suppose we want to represent four colors: {red, blue, green, black}

- How many bits of information do I need?
- (00, 01, 10, 11)
- The assignment of the **2-bit binary code** to colors is *ad-hoc*
- Also legitimate is: (10, 11, 00, 01)

How many bits of information do I need to represent the alphabet set in English?

- For 26 alphabets, we need 5 bits

Information Content in a Binary Code

$$D = \log_2 N \text{ bits}$$

The color set has four states: $N = 4$, # bits = 2

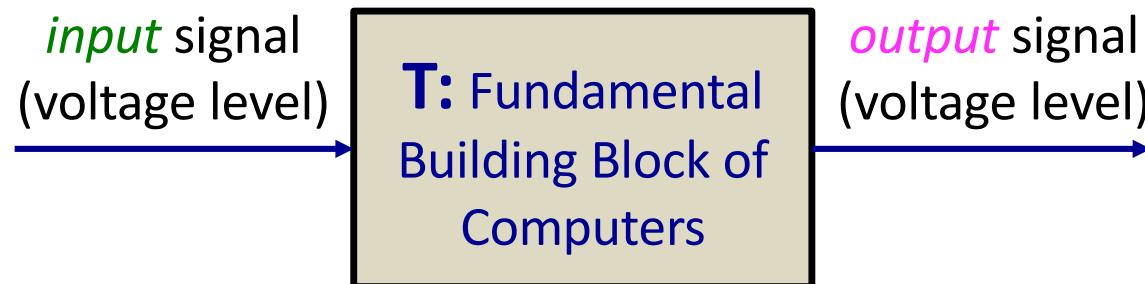
The alphabet set has 26 states: $N = 26$, # bits = 5

Conversely,

If D is 2, $N = 4$

If D is 5, $N = 32$

Why do computers use binary?

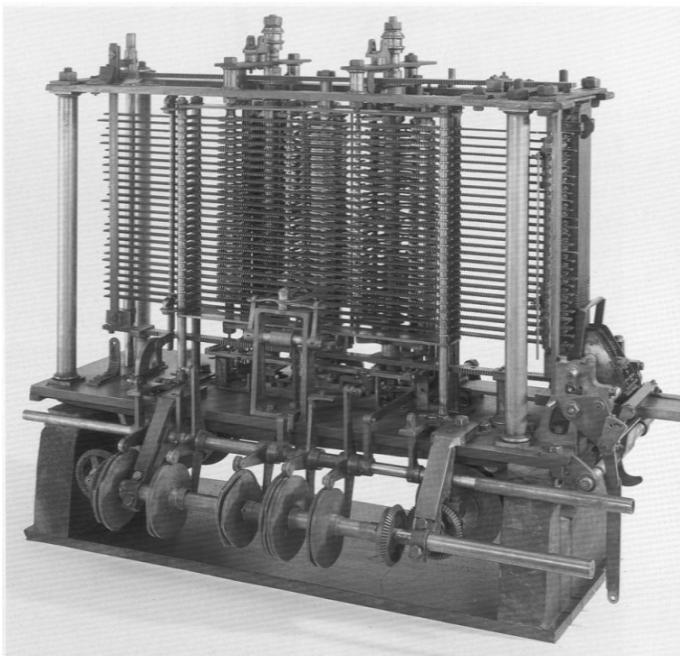


We can divide a continuous voltage range into ten levels to represent 0 – 9, but that would make **T** very complex

The fundamental building block of all computers is a transistor. A transistor can only distinguish two voltage levels. We call these voltage levels 1 and 0

Voltages and Transistors, Why?

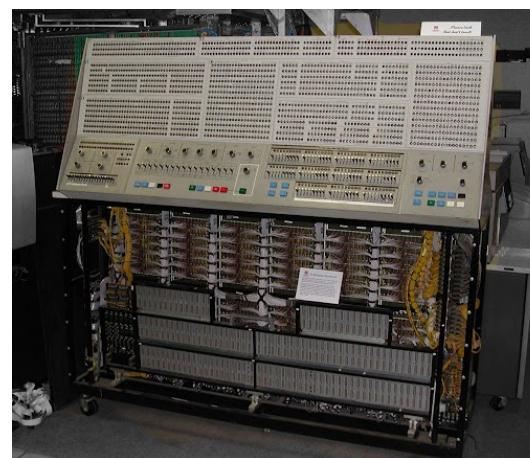
Mechanical parts: Not easy to scale to do large computations



The Analytical Engine
Charles Babbage
1834 – 1871



CDC 6600, 1964, \$ 2.5 M
Slower than my phone



IBM 360, 1964

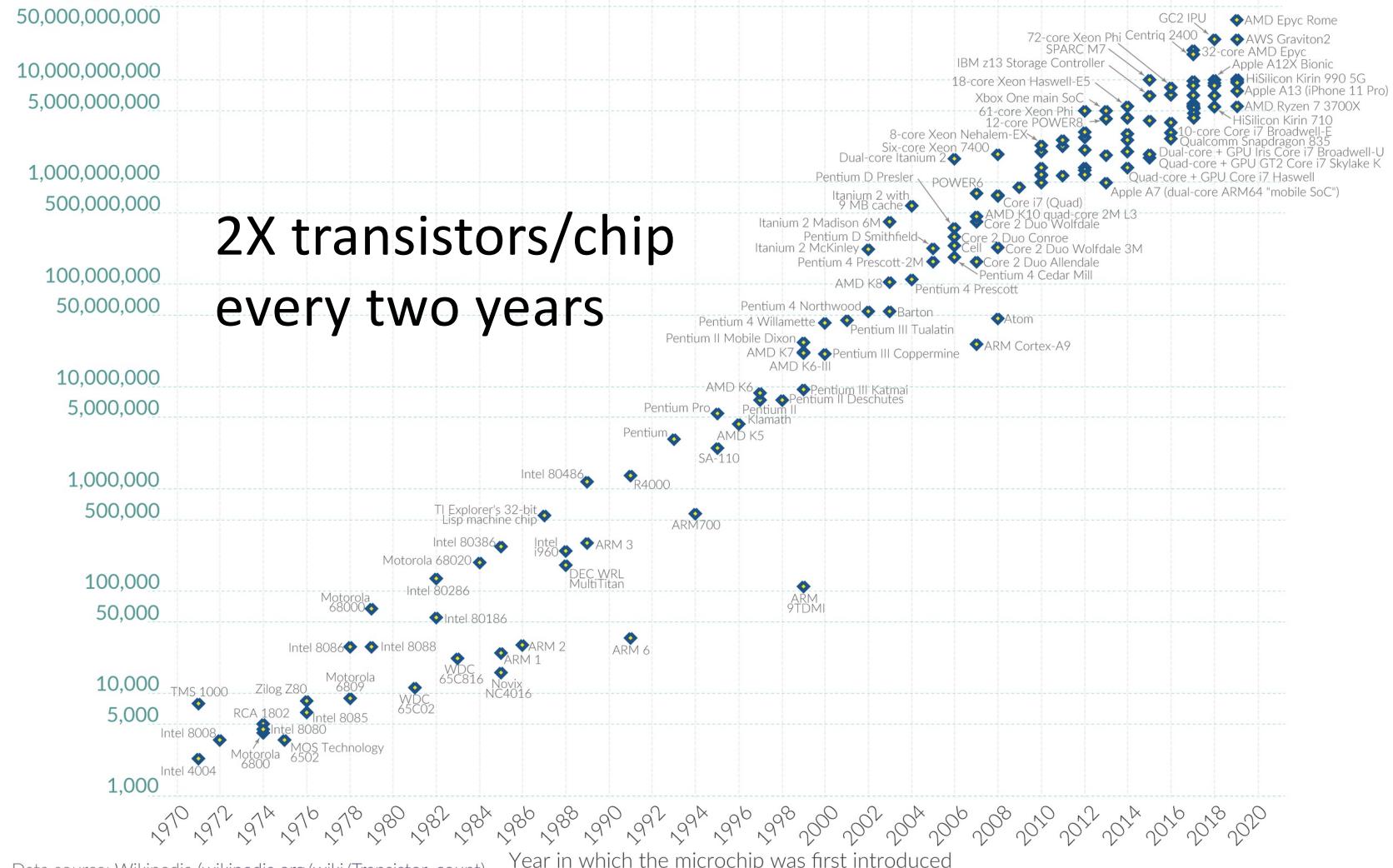


Apple M1, 2020
400 mm²
16 billion transistors

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Transistor count



Data source: Wikipedia ([wikipedia.org/wiki/Transistor_count](https://en.wikipedia.org/w/index.php?title=Transistor_count&oldid=910000000))

OurWorldInData.org – Research and data to make progress against the world's largest problems.

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

TRUE and FALSE



0 1

F T

False Off
True On

True and False are called logical values

- Logical variable is one that can be 1 or 0 (True or False)
- Boolean logic defines the operations on logic variables

Our Plan

Presenting information to digital circuits

- Representing numbers as a string of 1's and 0's
- Number systems: to set a foundation for efficient manipulation (add and subtract)

010101010100110
100110011010100
101001101011010
111011110101001
100010110010010
001001000010001
.....
.....

Operations on binary variables (1's and 0's)

- Logic gates to perform operations on binary variables

Breaking the digital abstraction (**self study**)

- 1's and 0's as continuous physical quantities (voltage)

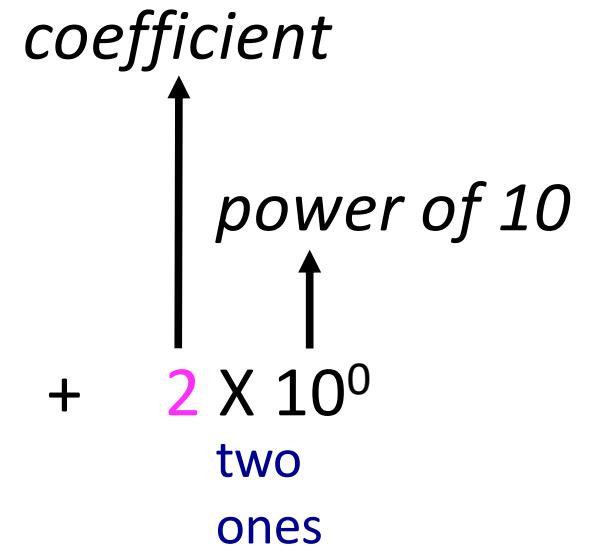
Decimal Number System

- Base 10 means 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- Multiple digits form longer decimal numbers
- Each column of a decimal number has 10 times the weight of the previous column

1
10's column
100's column
1000's column
1's column

$$9742 = 9 \times 10^3 + 7 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

nine thousands seven hundreds four tens two ones



Range of Decimal Numbers

An N-digit decimal number represents one of 10^N possibilities

- 0, 1, 2, 3, ..., $10^N - 1$
- 3 digits: 1000 possibilities in the range 0 – 999

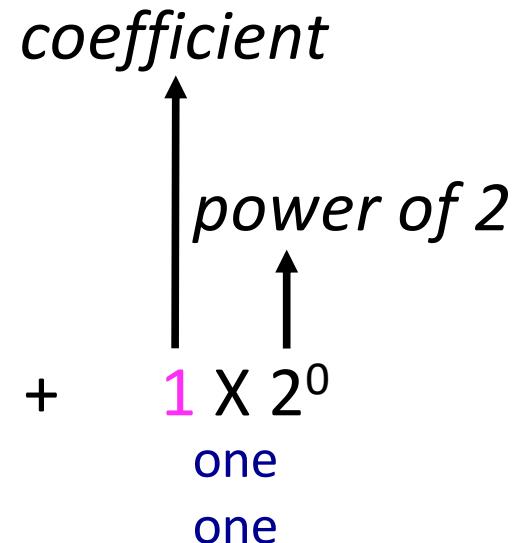
Binary Numbers

- Base 2 means 2 digits (0, 1)
- Multiple bits form longer binary numbers
- Each column of a binary number has **2** times the weight of the previous column

8's
4's
2's
1's
column
column
column
column

$$1001 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

one one zero one
eight four two one



Range of Binary Numbers

An N-bit binary number represents one of 2^N possibilities

- 0, 1, 2, 3, ..., $2^N - 1$
- 3 bits: 8 ($= 2 \times 2 \times 2$) possibilities in the range 0 – 7
- 4 bits: ?
- 5 bits: ?
- 10 bits: ?

Powers of 2

Columns #	Power of 2	Weight
0	2^0	1
1	2^1	2
2	2^2	4
3	2^3	8
4	2^4	16
5	2^5	32
6	2^6	64
7	2^7	128
8	2^8	256
9	2^9	512

Columns #	Power of 2	Weight	Kilo
10	2^{10}	1024	
11	2^{11}	2048	
12	2^{12}	4096	
13	2^{13}	8192	
14	2^{14}	16384	
15	2^{15}	32768	
16	2^{16}	65536	

Powers of 2

Power of 2	Decimal Value	Abbreviation
2^{10}	1024	Kilo (K)
2^{20}	1048576	Mega (M)
2^{30}	1073741824	Giga (G)

What is 2^{24} in decimal?

- $2^{20} \times 2^4 = 1 \text{ M} \times 16 = 16 \text{ M}$

What is 2^{17} in decimal?

- $2^{10} \times 2^7 = 1 \text{ K} \times 128 = 128 \text{ K}$

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

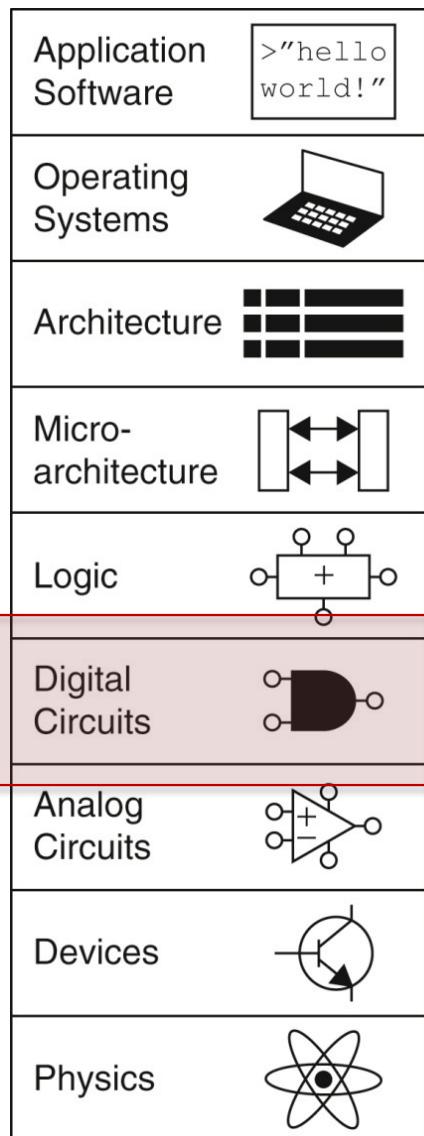
Plan: Week 2

Week 1: Digital abstraction and binary digits

Week 2: Number systems for binary variables, Logic gates

This Week: Boolean logic & Logic gates (contd)

This Week: Combinational logic (more than just gates)



Programs

Device Drivers

Instructions
Registers

Datapaths
Controllers

Adders
Memories

AND Gates
NOT Gates

Amplifiers
Filters

Transistors
Diodes

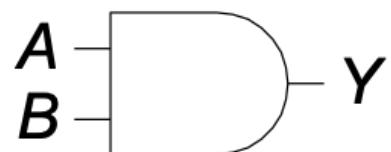
Electrons

We are here

The AND Function

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

Truth Table



AND Logic Gate

$$Y = AB$$

$$Y = A \cdot B \quad (\text{product})$$

$$Y = A \cap B \quad (\text{intersection})$$

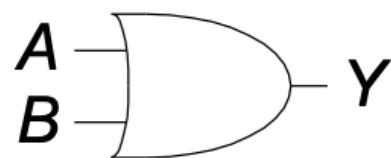
Boolean Equation

AND Function: *The output Y is 1 if and only if both A and B are 1*

The OR Function

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

Truth Table



OR Logic Gate

$$Y = A + B \text{ (sum)}$$
$$Y = A \cup B \text{ (union)}$$

Boolean Equation

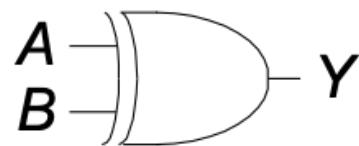
AND Function: *The output Y is 1 if either A or B are 1*

The XOR Function

eXclusive-OR

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

Truth Table



XOR Logic Gate

$$Y = A \oplus B$$

Boolean Equation

AND Function: *The output Y is 1 if A or B, but not both, are 1*

Terminology

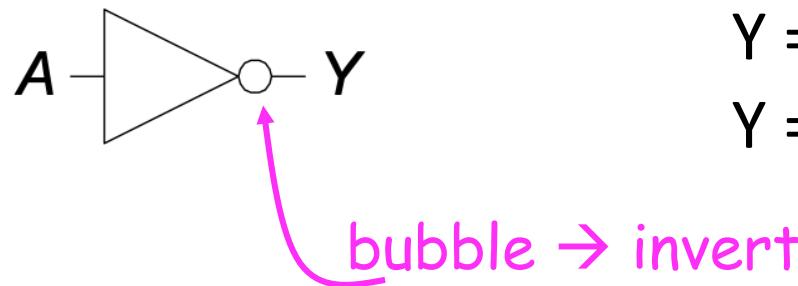
The term *exclusive* is used because the output is **1** if only one of the inputs is **1**

The OR function, on the other hand, produces an output **1**, if only one of the two sources is a **1**, or both sources are **1** (think of it as *inclusive* OR)

The NOT Unary Function

A	Y
0	1
0	1
1	0
1	0

The NOT gate has only one input (unary)



$$\begin{aligned} Y &= A' && \text{Read as} \\ Y &= \bar{A} && Y = \text{NOT } A \end{aligned}$$

Truth Table

NOT Logic Gate

Boolean Equation

NOT Function: *The output Y is the inverse of the input A*
The NOT gate is also known as an inverter

Inverting a Gate's Operation

Any gate can be followed by a bubble to invert its operation

NOT AND →

NAND



NOT OR →

NOR



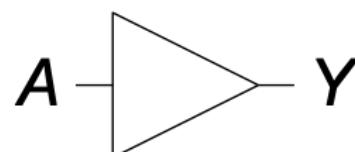
NOT XOR →

XNOR

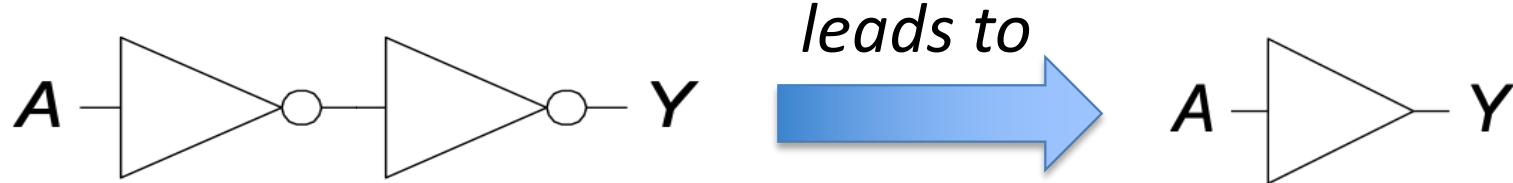


NOT NOT →

BUF



In Boolean logic, two wrongs make a right!



We say that two bubbles cancel each other's effect

The NAND Function

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

Truth Table



NAND Logic Gate

$$Y = (AB)'$$

Boolean Equation

NAND Function: *The output Y is 1 unless both inputs are 1*

The NOR Function

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

Truth Table



NOR Logic Gate

$$Y = (A + B)'$$

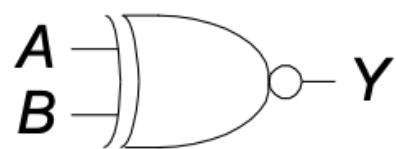
Boolean Equation

NOR Function: *The output Y is 1 if neither A nor B is 1*

The XNOR Function

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

Truth Table



XNOR Logic Gate

$$Y = (A \oplus B)'$$

Boolean Equation

XNOR Function: *The output Y is 1 if both A and B are 1 or both are 0*

XOR and XNOR are special

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

XOR

XOR: Output is 1 when inputs are not equal (odd number of 1's)

Parity Gate

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

XNOR

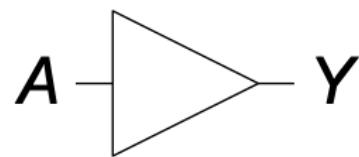
XNOR: Output is 1 when inputs are equal (even number of 1's)

Equality Gate

Buffer (BUF)

A	Y
0	0
0	0
1	1
1	1

Truth Table



BUF Logic Gate

$$Y = A$$

Boolean Equation

Buffer: *The output Y is equal to the input A*

Buffer (BUF)

- At the logic level, BUF is no more useful than a wire
- At a lower level of abstraction (analog level)
 - BUF can deliver a large amount of current to a motor
 - It can send output to many gates (think of an amplifier)

Critical to consider multiple layer of abstraction in the compute stack to understand the significance of various elements

Multiple-Input Gates

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

Gates with multiple inputs are possible

Looking at the truth table, can you guess the 3-input gate?



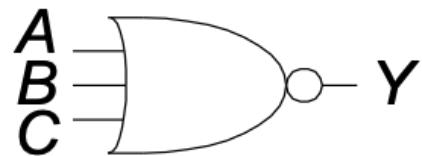
$$Y = ABC$$

Multiple-Input Gates

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

Gates with multiple inputs are possible

Looking at the truth table, can you guess the 3-input gate?



$$Y = (A + B + C)'$$

Bitwise Operations

All logical operators can be applied to two bit-patterns (i.e., a group of bits) of m bits each, where m is any # bits (8, 16, ...)

- Apply the operation individually to each pair of bits
- If A and B are 8-bit input sources (or source operands), then their AND or product, C, is also 8 bits

$$C = AB \text{ (bit-wise AND)}$$

A	0	0	0	0	1	1	0	1
B	1	1	1	1	1	1	1	1
C	0	0	0	0	1	1	0	1

$$C = A + B \text{ (bit-wise OR)}$$

A	0	0	0	0	1	1	0	1
B	0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	0	1

Bit Masks

Suppose we are interested in extracting the least significant four bits from **A**, while ignoring the right-most four bits

- If we AND **A** with **B**, and choose **B** as **00001111**, then we get the desired bit pattern in **C**
- **Bit mask:** A binary pattern (**B**) that separates the bits of **A** into two halves, the half we care about, and the half we wish to ignore

$$\text{C} = \text{AB} \text{ (bit-wise AND)}$$

A	0	1	1	0	1	1	0	1
B	0	0	0	0	1	1	1	1
C	0	0	0	0	1	1	0	1

Exercises

Suppose we have a bit pattern, $A = \textcolor{green}{11000010}$, and the rightmost two bits are of particular significance. Find a bitmask and a logical operation to mask out the values in the rightmost positions in a new bit pattern C. (**All other bits in C are set to 0.**)

Suppose we have a bit pattern, $A = \textcolor{red}{10110010}$, and the leftmost two bits are of particular significance. Find a bitmask and a logical operation to mask out the values in the leftmost positions in a new bit pattern C. (**All other bits in C are set to 1.**)

Exercise

Suppose we want to know if two bit-patterns A and B are identical. How can we find out if two bit-patterns are identical?

Verify that, $B \text{ AND } 1 = B$, where B is a binary variable. Also, verify that, $B \text{ OR } 0 = B$.

Verify that, $B \text{ AND } 0 = 0$, where B is a binary variable. Also, verify that, $B \text{ OR } 1 = 1$.

Exercise

Verify that, $B \text{ AND } B = B$, where B is a binary variable. Also, verify that, $B \text{ OR } B = B$.

Verify that, $B \text{ AND } B' = 0$, where B is a binary variable. Also, verify that, $B \text{ OR } B' = 1$.

Key Ideas

Any physical quantity can represent TRUE (1) and FALSE (0). Computers use voltage levels for representing one and zero as electronic components such as transistors can distinguish between these two voltage levels. Our ability to shrink transistors has enabled faster computers in a small chip area (400 mm² approx.).

Voltage is a continuous physical signal. We can split voltage into as many levels as we want. We use only two levels to represent and manipulate binary variables to simplify circuits.

Using binary variables and Boolean logic to build computers leads to more efficient circuits and computers.

We can do arithmetic in any other base (e.g., 2) without learning Boolean and digital logic. There is nothing special about adding binary numbers compared to adding decimal numbers.

Key Ideas

We need Boolean logic to understand the interaction between binary variables, and understanding the interaction requires us to learn about logic functions. Logic functions can eventually lead us to build more complex digital circuits that solve real-world problems, e.g., adding two large numbers.

We (humans and, more specifically, John von Neumann) found a system of representation for binary numbers called two's complement. This representation simplifies building arithmetic circuits as a single circuit for adding two numbers can handle addition and subtraction. The circuit itself does not know about two's complement. We build circuits and computers today, assuming two's complement signed integers.

Classification of Digital Circuits

Combinational Circuit: Output depends on the current values of the inputs only

- Memoryless (a *distinct* and *critical* feature)
- All logic gates are combinational

Sequential Circuit: Output depends on the current and previous values of the inputs

- The sequence of inputs dictate the output
- Sequential circuits have state or memory
- Example: Elevator controller (**State:** TRANSIT, GROUND, TOP)



Combinational Behavior

Example: Suppose a combinational circuit, consisting of an AND gate, with two inputs, A and B

<i>time →</i>	<i>t0</i>	<i>t1</i>	<i>t2</i>	<i>t2</i>	<i>t4</i>	<i>t5</i>	<i>t6</i>
A	0	1	1	0	1	0	1
B	0	1	0	0	1	0	1
Output	0	1	0	0	1	0	1

At time t_6 , the *sequence* of changes to A and B between $t_0 - t_5$ is irrelevant. The output is strictly determined by the values of A and B at t_6

Combinational Circuits

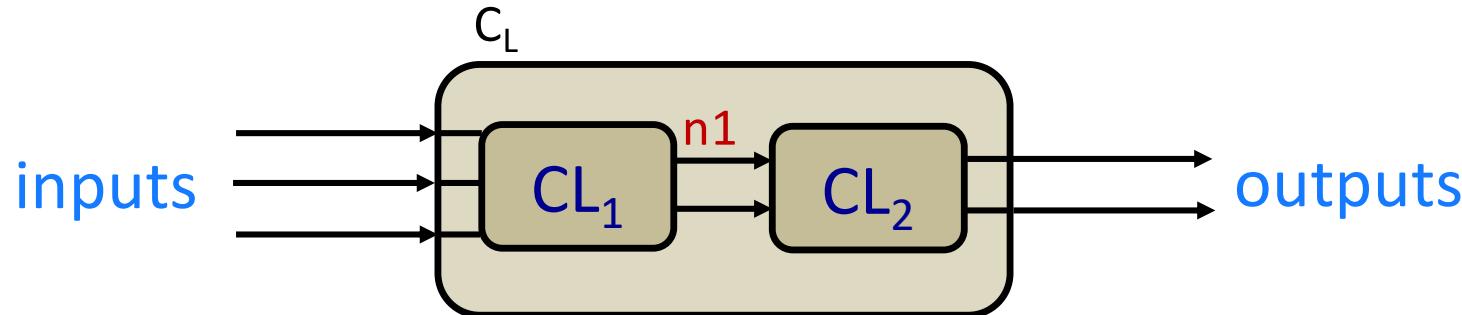


Functional specification: What is the circuit supposed to do?
What is the **output** for a given combination of **input** values?

Timing specification: How long does the circuit takes to produce the output?

- Worst-case: ten nanoseconds
- Best-case: one nanoseconds

Combinational Circuits



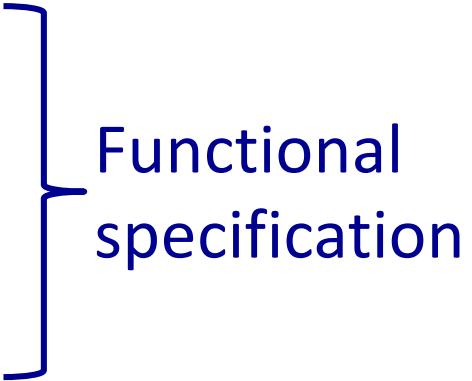
Hierarchy: The top-level circuit, C_L , is made up for of two sub-circuits (also combinational), CL_1 and CL_2

Nodes: $n1$ is an internal wire or node

Abstraction: The *input and output* interface, and the functional and timing specification is enough for someone to use C_L . They do not need to know the inner composition of C_L

Implementing Combinational Logic

Steps in implementing combinational Logic

1. Initial specification (e.g., in English)
 2. Construct the truth table
 3. Derive the Boolean equation
 4. Simplify the Boolean equation (use Boolean algebra)
 5. Implement the equation using logic gates
- 
- Functional specification

Specification

[Happiness detector] The students are back on campus. They are not *happy* if there is a *homework* deadline, or *Badger & Co.* is closed. Design a circuit that will output **1** only if students are happy.

[Multiplexer] Design a circuit with three inputs: D_0 , D_1 , *select*; and one *output*. The output is D_0 if select is **0**, and D_1 if select is **1**.

[Half Adder] Design a circuit that adds two binary variables: A and B . The circuit has two outputs: *sum* and *carry-out* (C_{out}).

[Full Adder] Design a circuit that adds three binary variables: A , B , and a *carry-in* (C_{in}). The circuit has two outputs: *sum* and *carry-out* (C_{out}).

Constructing Truth Tables

Identify inputs and outputs (interface)

- The inputs and outputs maybe implicitly specified
- *Or*, determining them may require some thought

Write all the possible combinations of input values

- For each input combination, determine the output
- *All **inputs** to the left, **outputs** to the right*

Truth Table: Happiness Detector

Specification: The students are back on campus. They are **not happy** if there is a *homework* deadline, or *Badger & Co.* is closed. Design a circuit that will output **1** only if students are *happy*.

Interface

Homework deadline? (D)

- 0: there is *not* a deadline
- 1: there is a deadline

Badger is closed? (B)

- 0: open
- 1: closed

Happy (H): 1 → ☺, 0 → ☹

Truth Table

D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

Deriving a Boolean Equation

Some Terminology first

- For any binary variable X , its *compliment* is X'
- True form (X) and complementary form (X') are called *literals*
- AND of one or more literals is called a *product* or *implicant*
 - $X, Y, XY, X'Y'Z, XYZ, XY'Z'$ are all implicants for a function of three variables
- **Minterm:** A product involving all the inputs to the function
 - XYZ is a minterm for a function of three variables X, Y , and Z
 - XY is not a minterm because it is missing one literal (Z)

Deriving a Boolean Equation

Order of operations

- NOT has the highest precedence
- Next is AND
- OR is last
- Example: $Y = A + BC'$
 - First, we find C'
 - Then, we find BC' (product/AND)
 - Finally, we perform $A + (\text{the result of } BC')$

Sum-of-Products Form

To write the Boolean equation for a truth table, *sum each of the minterms for which the output is 1*

A	B	Y1	minterm	name
0	0	0	$A'B'$	m_0
0	1	1	$A'B$	m_1
1	0	0	AB'	m_2
1	1	0	AB	m_3

Boolean Eq

$$Y1 = A'B$$

$Y1$ is 1 only when $A = 0$ and $B = 1$

Conversely, when $A' = 1$ and $B = 1$

Sum-of-Products Form

To write the Boolean equation for a truth table, *sum each of the minterms for which the output is 1*

A	B	Y1	minterm	name
0	0	0	$A'B'$	m_0
0	1	1	$A'B$	m_1
1	0	0	AB'	m_2
1	1	1	AB	m_3

Boolean Eq

$$Y1 = A'B + AB$$

$Y1$ is 1 **either** when $A = 0$ and $B = 1$

OR, when $A = 1$ and $B = 1$

$$Y1 = \Sigma(1,3)$$

Equation: Happiness Detector

Specification: The students are back on campus. They are **not happy** if there is a *homework* deadline, or *Badger & Co.* is closed. Design a circuit that will output **1** only if students are *happy*.

Truth Table

D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Eq

$$H = D'B'$$

$$H = (D)' \text{ AND } (B)'$$

From Equation to Gates

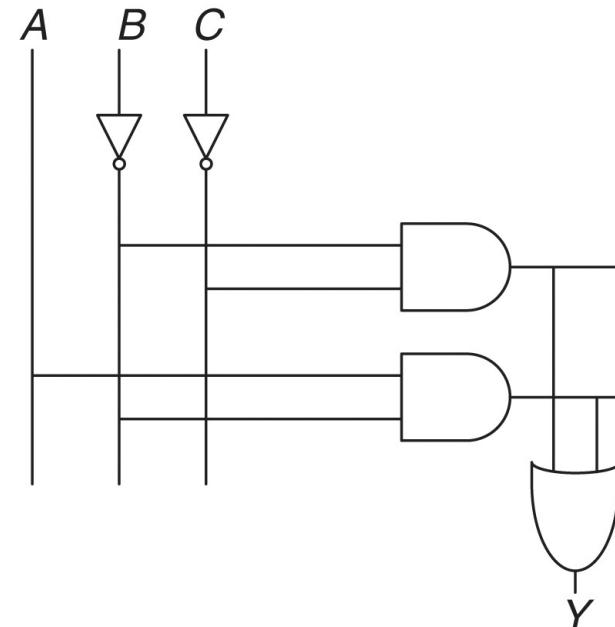
Schematic: A diagram of a digital circuits with elements (gates) and the wires that connect them together

Example Boolean Eq

$$Y = AB' + B'C'$$

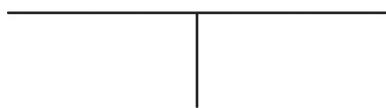
Schematic

1. Inputs are on the left (or top) side
2. Outputs are on the right
3. Gates flow from left to right
4. Use straight wires
5. Wires connect at a T junction
6. A dot where wires cross indicates a connection

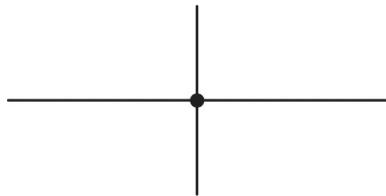


Rules for Connecting Wires

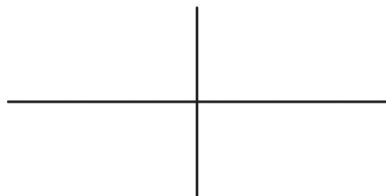
wires connect
at a T junction



wires connect
at a dot



wires crossing
without a dot do
not connect



Schematic: Happiness Detector

Specification: The students are back on campus. They are **not happy** if there is a *homework* deadline, or *Badger & Co.* is closed. Design a circuit that will output **1** only if students are *happy*.

Truth Table

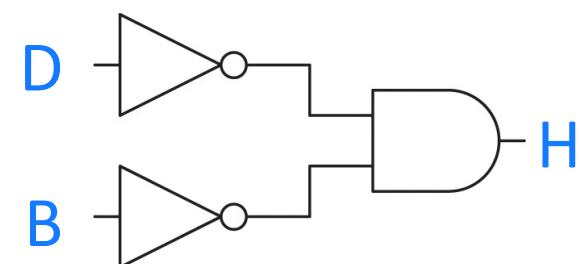
D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Eq

$$H = D'B'$$

$$H = (D)' \text{ AND } (B)'$$

Logic Gate Implementation



Schematic: Happiness Detector

Specification: The students are back on campus. They are **not happy** if there is a *homework* deadline, or *Badger & Co.* is closed. Design a circuit that will output **1** only if students are *happy*.

Truth Table

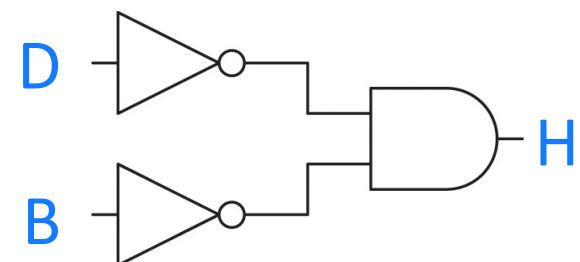
D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Eq

$$H = D'B'$$

$$H = (D)' \text{ AND } (B)'$$

Logic Gate Implementation



*Which (monolithic) gate
is this?*

Schematic: Happiness Detector

Specification: The students are back on campus. They are **not happy** if there is a *homework* deadline, or *Badger & Co.* is closed. Design a circuit that will output **1** only if students are *happy*.

Truth Table

D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

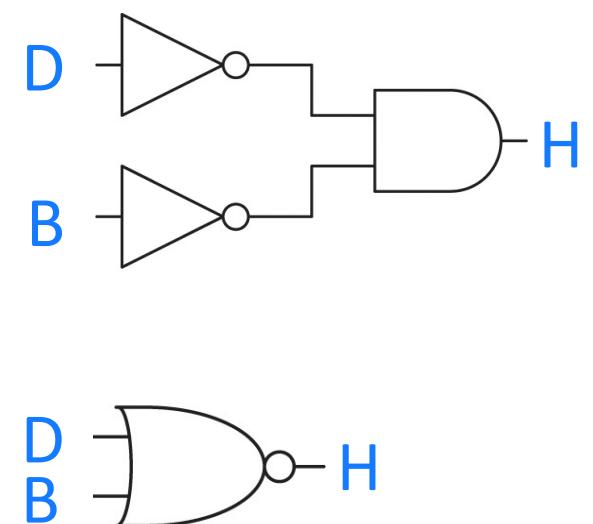
Boolean Eq

$$H = D'B'$$

$$H = (D)' \text{ AND } (B)'$$

*Which (monolithic) gate
is this? Answer: NOR gate*

Logic Gate Implementation



Multiplexer: T. Table + Eq

Specification: Design a circuit with three inputs: D_0 , D_1 , select (S); and one output (Y). The output is D_0 if select is 0, and D_1 if select is 1.

$$Y = S'D_1'D_0 + S'D_1D_0 + SD_1D_0' + SD_1D_0$$

$$Y = S'D_0 \underbrace{(D_1' + D_1)}_{=1} + SD_1 \underbrace{(D_0' + D_0)}_{=1}$$

$$Y = S'D_0 (1) + SD_1 (1)$$

$$Y = S'D_0 + SD_1$$

Truth Table

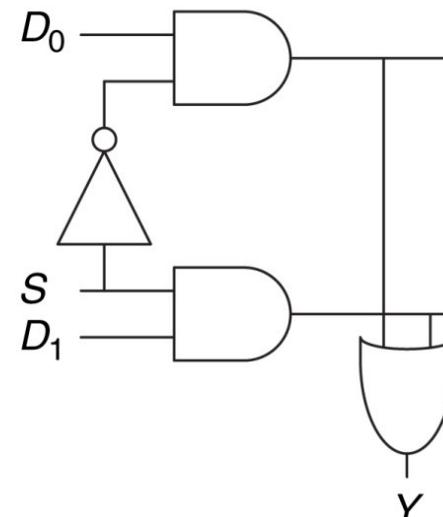
S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Multiplexer: Gate-Level Schematic

Specification: Design a circuit with three inputs: D_0 , D_1 , select (S); and one output (Y). The output is D_0 if select is 0, and D_1 if select is 1.

$$Y = S'D_0 + SD_1$$

Gate-Level Schematic



S	D_1	D_0	Y
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

Half Adder

Specification: Design a circuit that adds two binary variables: A and B. The circuit has two outputs: sum and carry-out (C_{out}).

Truth Table

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

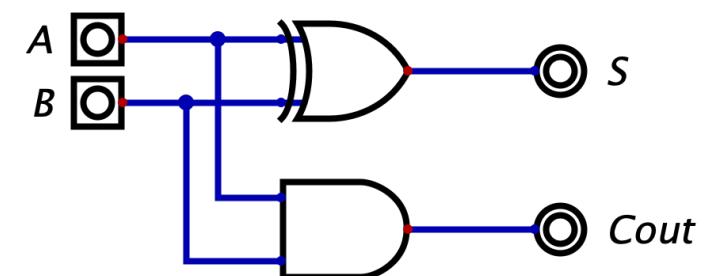
Boolean Eq

$$S = A'B + AB'$$

$$S = A \oplus B$$

$$C_{out} = AB$$

Schematic



Full Adder: T. Table + Eq

Specification: Design a circuit that adds two binary variables: A and B . The circuit has two outputs: **sum** and **carry-out** (C_{out}).

$$S = C_{in}'A'B + C_{in}'AB' + C_{in}A'B' + C_{in}AB$$

$$C_{out} = C_{in}'AB + C_{in}A'B + C_{in}AB' + C_{in}AB$$

Simplification via Boolean algebra

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = C_{in}(A \oplus B) + AB$$

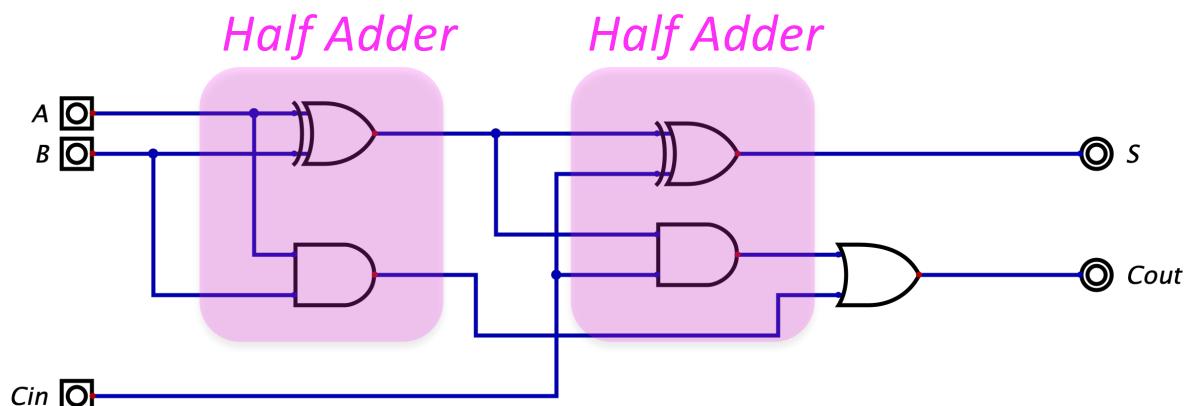
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder: Schematic

Specification: Design a circuit that adds two binary variables: A and B . The circuit has two outputs: sum and carry-out (C_{out}).

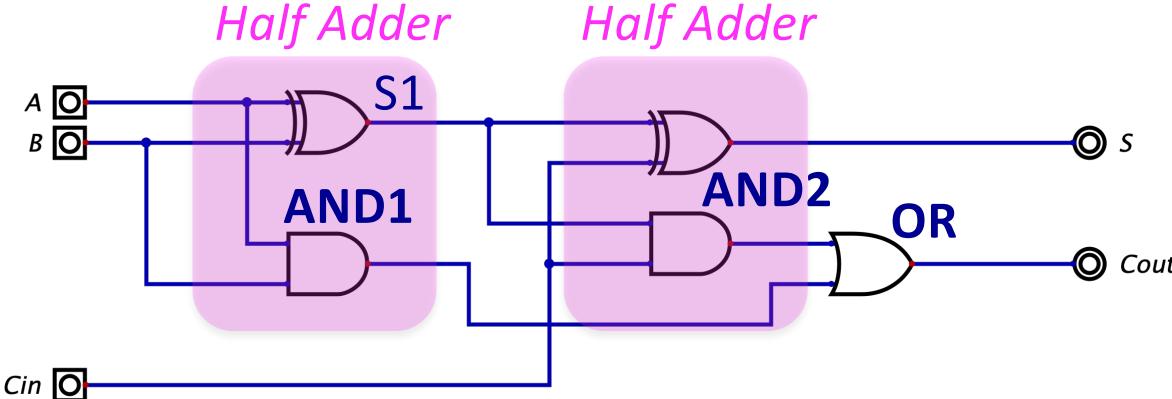
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = C_{in}(A \oplus B) + AB$$



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

Full Adder = Two Half Adders



What is **AND1** doing?

- Computes the carry out from $A + B$ (call it S_1)

What is **AND2** doing?

- Computes the carry out from $S_1 + C_{in}$

What is the **OR** gate doing?

- C_{out} is 1 if either the output of AND1 is 1 or the output of AND2 is 1
- What does the truth table reveal about C_{out} ?

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

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

Plan: Week 2

Week 1: Digital abstraction and binary digits

Week 2: Number systems for binary variables, Logic gates

This Week: Boolean logic & Logic gates (contd)

This Week: Combinational logic (more than just gates)

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Programs

Device Drivers

Instructions
Registers

Datapaths
Controllers

Adders
Memories

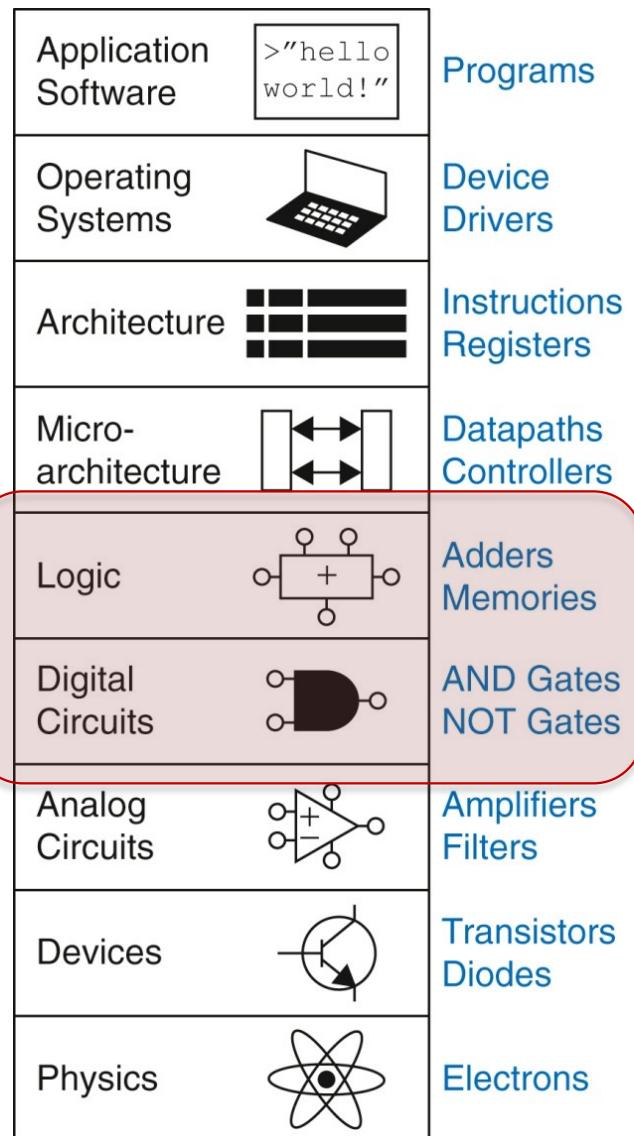
AND Gates
NOT Gates

Amplifiers
Filters

Transistors
Diodes

Electrons

We were here



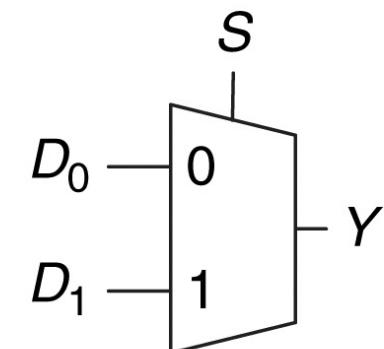
Broadening our horizon
“one layer at a time”

2:1 Multiplexer (Mux)

We have seen a 2:1 multiplexer (mux)

- Two data inputs (D_0 and D_1)
- Another input called the **select** signal
- Choose D_0 or D_1 based on the value of the select signal

We will use the high-level schematic for 2:1 mux
and ignore the gate-level implementation details

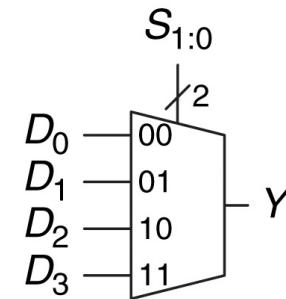


High-level Schematic

Wider (4:1) Multiplexer

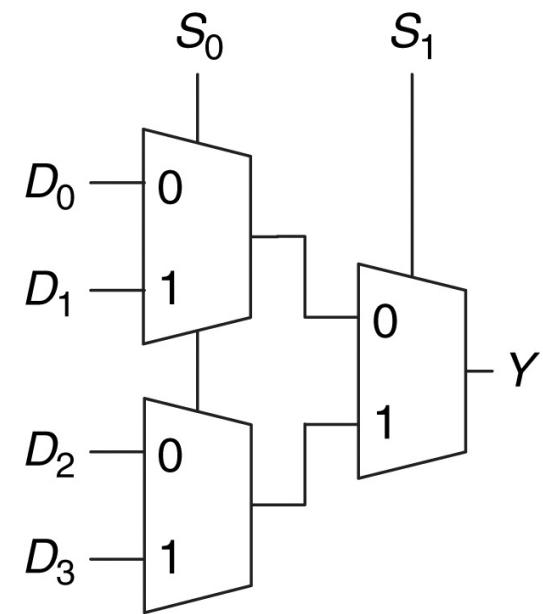
We want to build a 4:1 mux

- How many **select** signals?
 - Call them S_0 and S_1
 - A / and 2 implies a bus (of width 2) and not a 1-bit wire or input
- One option is to construct the truth table and derive the Boolean equations. How many rows will there be in the table? (**tedious!**)
- We will use **intuition** to build a 4:1 mux from two 2:1 multiplexers



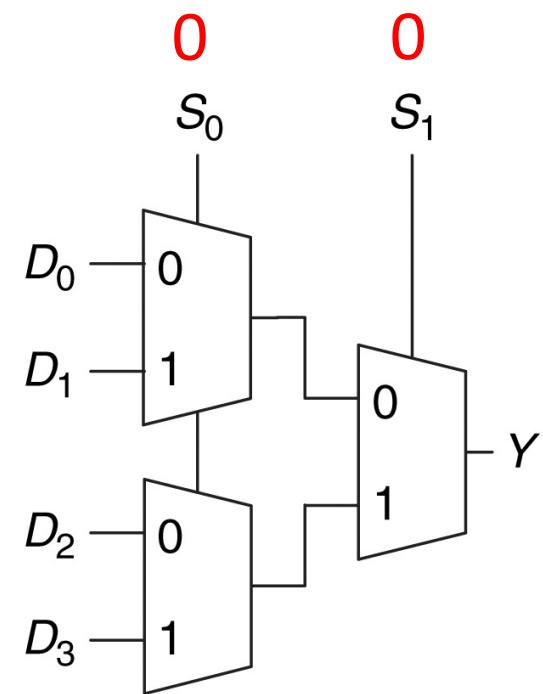
Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



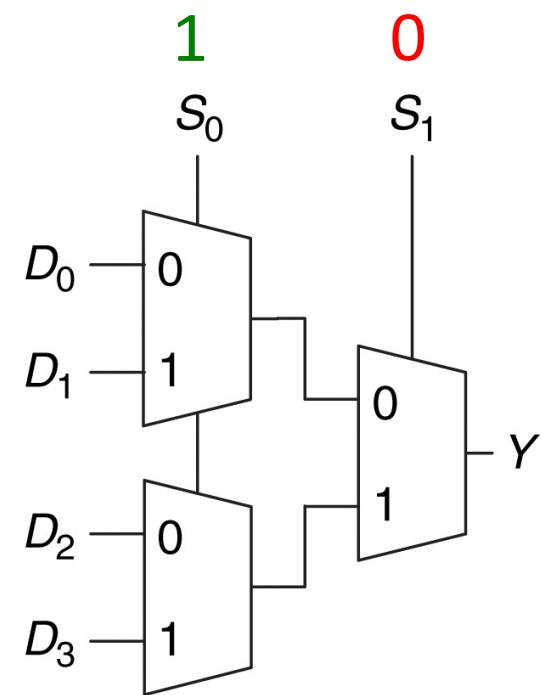
Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



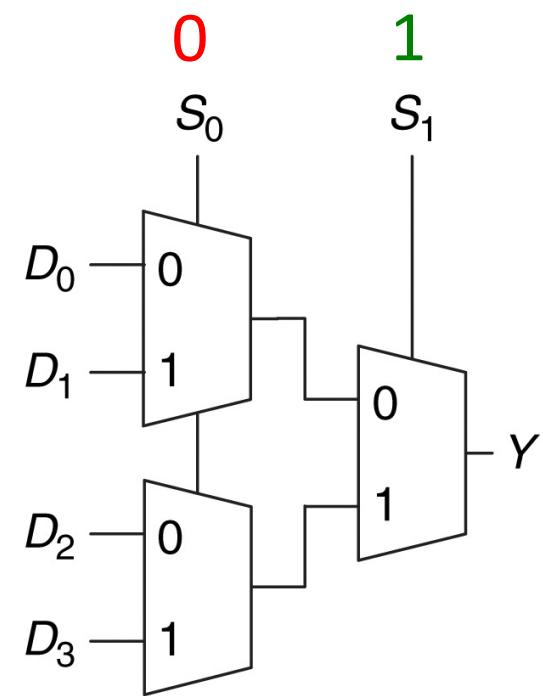
Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



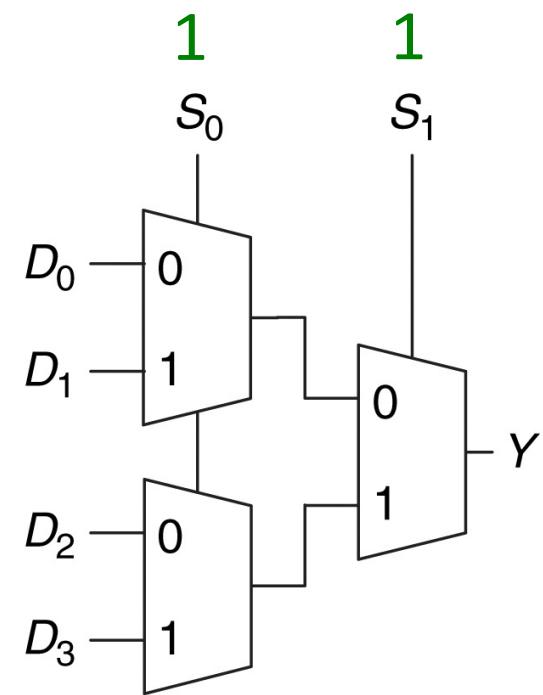
Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



Wider (4:1) Multiplexer

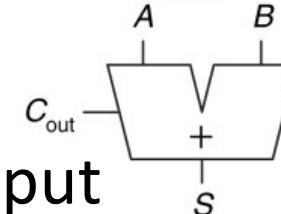
S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



Ripple Carry Adder

We have seen the half adder

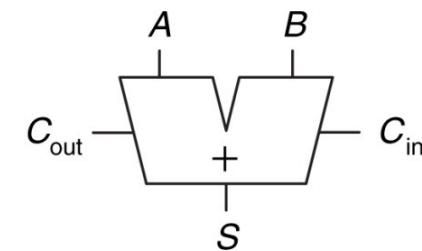
- Limitation of half adder: No carry input
- **Problem:** Adding multiple bits requires the need to add carry out from the previous column to the next column



$$\begin{array}{r} & \textcolor{blue}{1} \\ & 1001 \\ + & 0101 \\ \hline 1110 \end{array}$$

Full adder *solves* the problem

- Accepts three inputs including a carry in
- Signals flow from right to left to reflect the carry propagation in arithmetic circuits

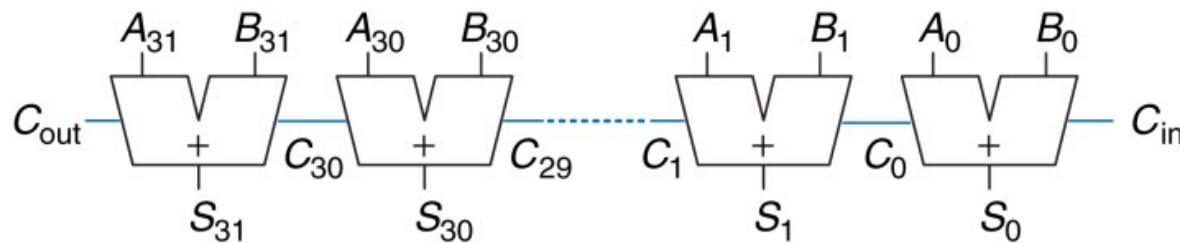


What if we want to add two N-bit numbers?

Ripple Carry Adder

What if we want to add two N-bit numbers?

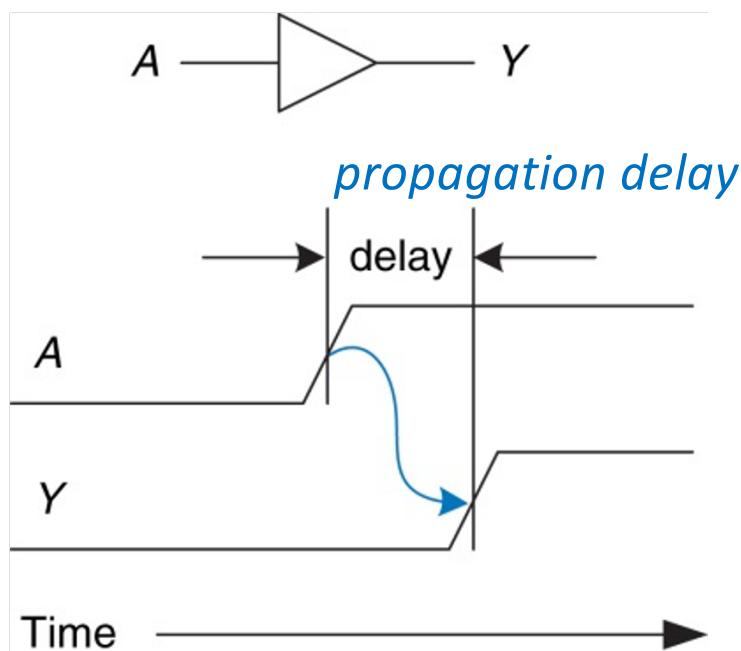
- *Connect a chain of full adders from right to left*



Ripple carry adder has a critical drawback!

Timing

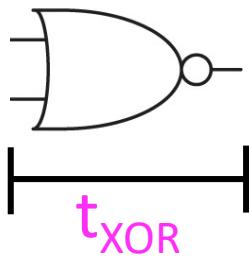
- Every combinational circuit has a *delay (seconds)*
 - *The time it takes for the output to reach a final stable value when the input changes (nanoseconds or picoseconds)*



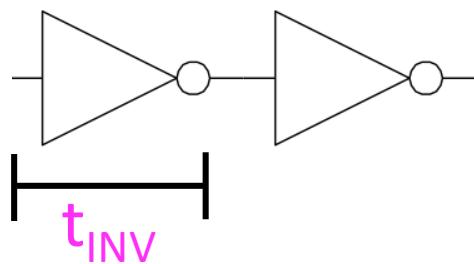
Examples

- **Example:** When the two inputs of the AND gate change from $(0,0)$ to $(1,1)$, how long does it take to reliably measure the output of the AND gate change from 0 to 1 ?
- **Another example:** When A , B , and C_{in} are supplied to a full adder, how long does it take to observe the final (and stable) S and C_{out} ?

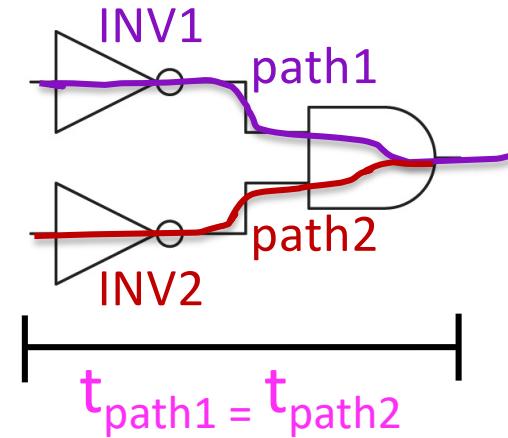
Our Notion of Timing/Delay



Each gate has
a delay



Chain of gates:
Sum the delay of
each gate in the
chain $2 \times t_{INV}$



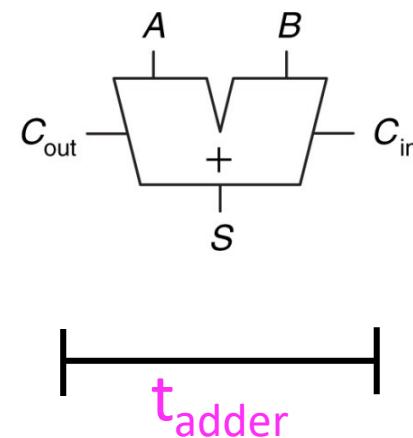
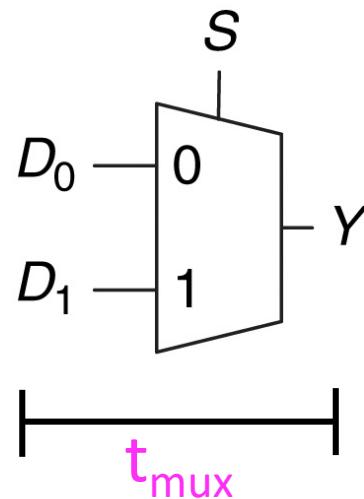
Multiple paths from
input to output
 $t_{path1} = t_{INV1} + t_{AND}$
 $t_{path2} = t_{INV2} + t_{AND}$

Critical and Shortest Path

- Many combinational circuits have multiple paths from input to the output
 - The slowest path (*longest delay*) is called the **critical path**
 - Critical path limits the speed at which the circuit operates
 - In contrast, the shortest path is the fastest
- For simplification, we will ignore the delay of nodes (wires)
 - Although the delay is non-trivial, it is studied best at lower levels of abstraction (e.g., analog circuits)

Our Notion of Timing/Delay

We will use a similar notion of delay for combinational circuits such as multiplexers and adders

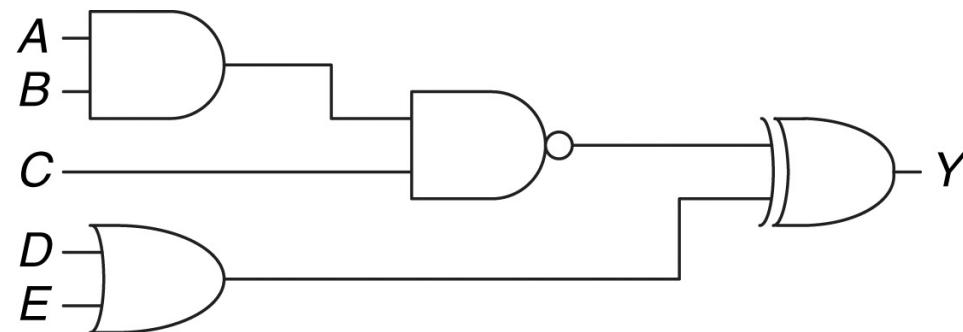


t_{mux}

t_{adder}

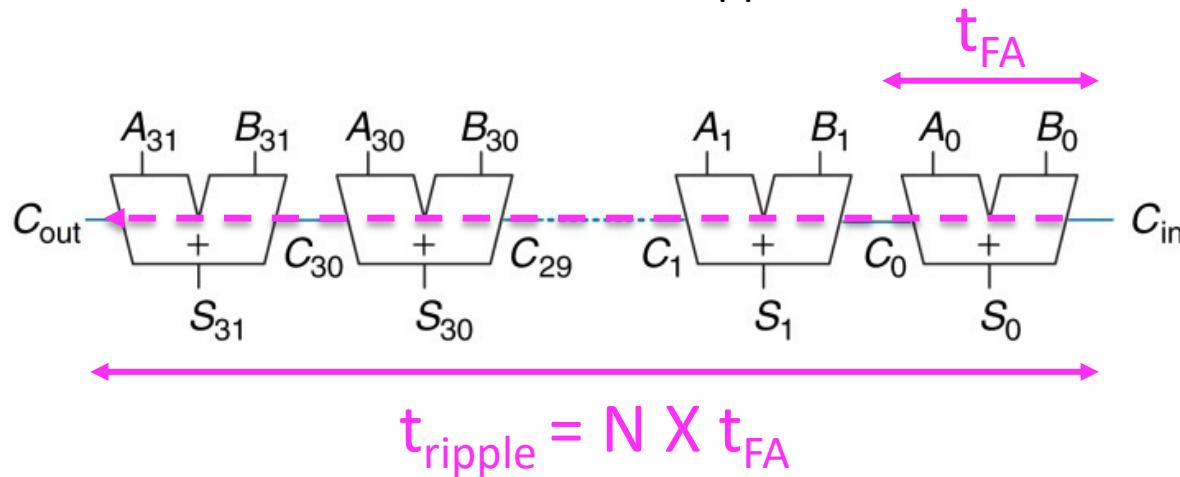
Exercise

Each gate has a propagation delay of 50 picoseconds. Find the shortest path. Find the critical path.



Drawback: Ripple Carry Adder

- If we abstract the delay of full adder as t_{FA} , then what is the delay of the ripple carry adder, t_{ripple} ?



- The critical path runs through the chain of full adders*
- Every full adder is on the critical path*
- The critical path consists of N full adders (slow when N is large)*

Carry-Lookahead Adder

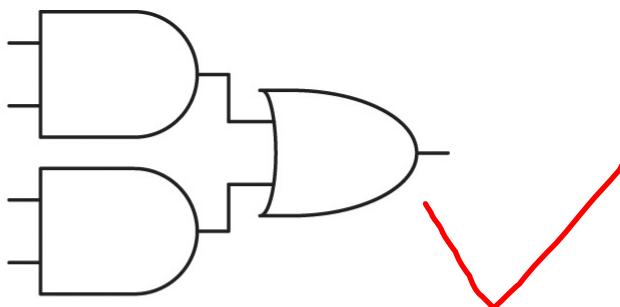
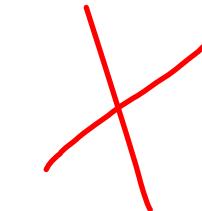
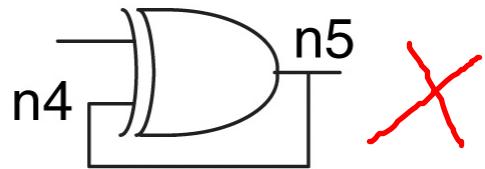
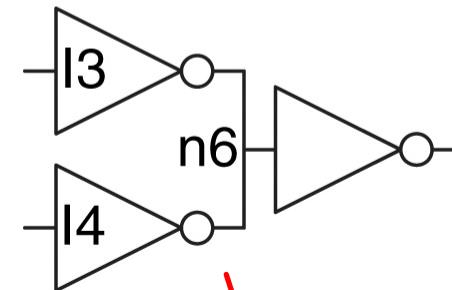
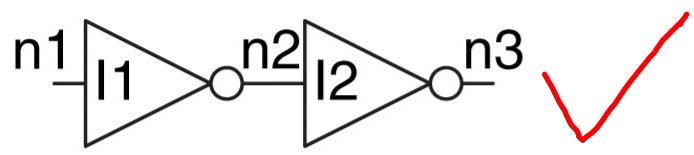
- Another one in the class of *carry propagate adders* that accelerates the computation of carry signals
- Pre-compute (lookahead) carry signals
- Use additional logic (area) to speed up addition
 - Details in Section 5.2.1 (**Optional self-study**)

In digital systems, there is very often a tradeoff in performance (speed) and hardware cost (area/power)

Combinational Composition Rules

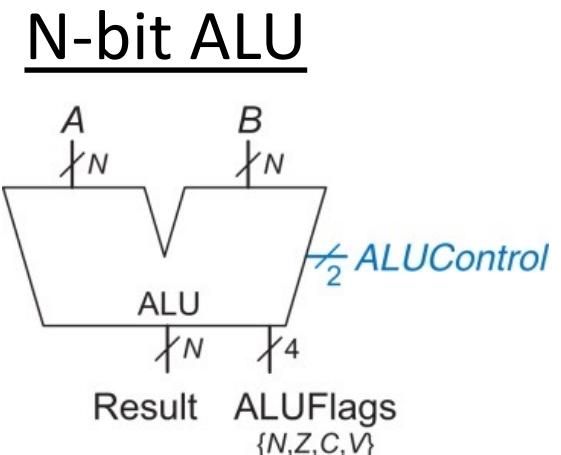
1. Every circuit element is itself combinational
2. Each node is either an input to the circuit or connects to exactly one output terminal of a circuit element
3. The circuit contains no cyclic paths; every path through the circuit visits each circuit node at most once

Which circuits are combinational?



Arithmetic and Logic Unit (ALU)

- The circuits we have looked so far can do one useful thing
 - XNOR gate *performs* equality testing
 - Adder *performs* addition
 - Multiplexer *performs* selection
- ALU is our first *general purpose* circuit
 - Can do a variety of mathematical and logical operations
 - Add, subtract, AND, OR
 - Yet we abstract its inner details as a monolithic unit



ALU Interface/Instructions

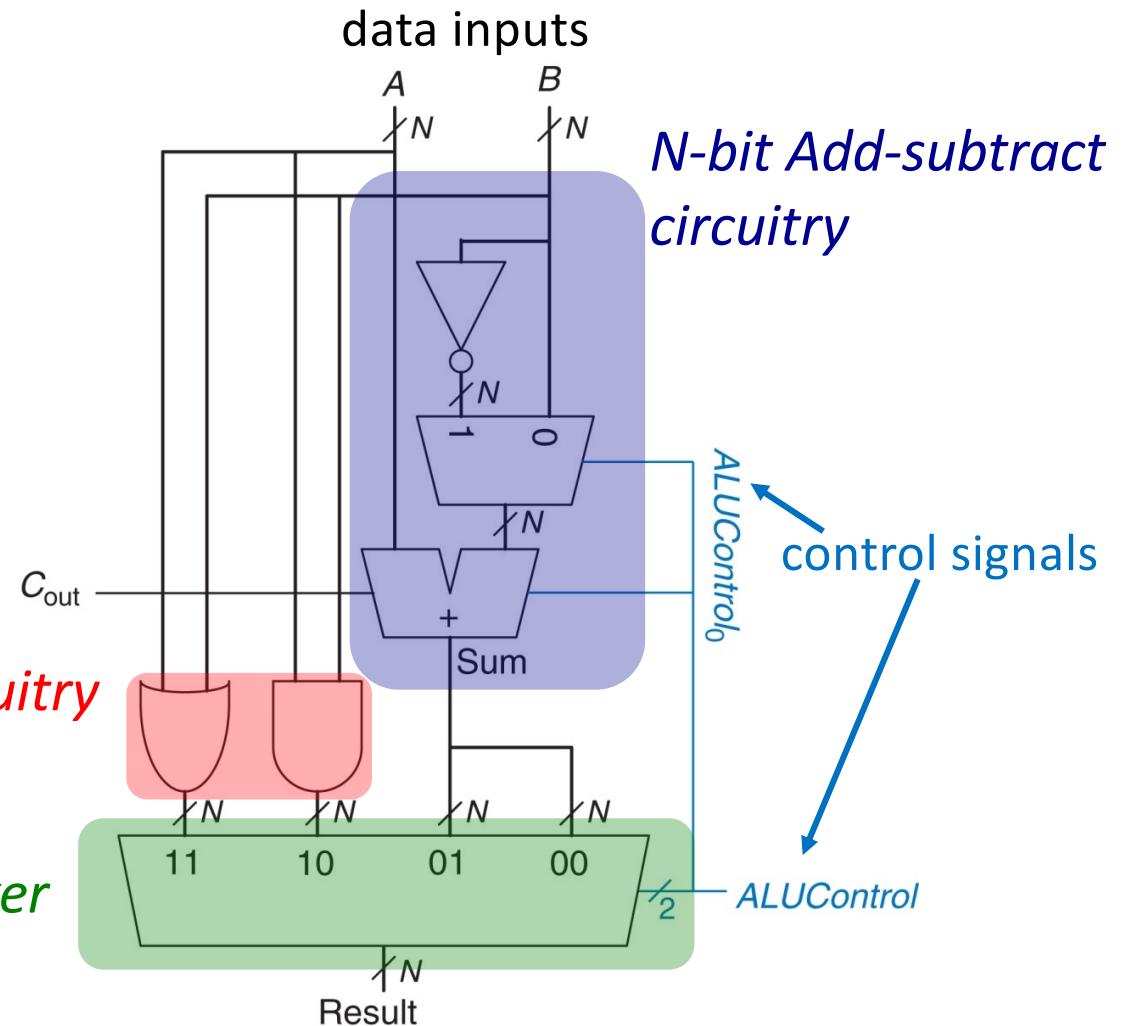
- N-bit *data inputs* and *outputs*
- 2-bit *control* input (ALUControl)
 - Pick one of four functions
 - A 2-bit signal specifies the function
 - Think of setting ALUControl to **00**, **01**, **10**, and **11** as giving “*instructions*” to the ALU
- The assignment of binary codes to ALU functions is not arbitrary
 - It is clever (**01** for Subtract in particular) as we will reveal

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

ALU Implementation

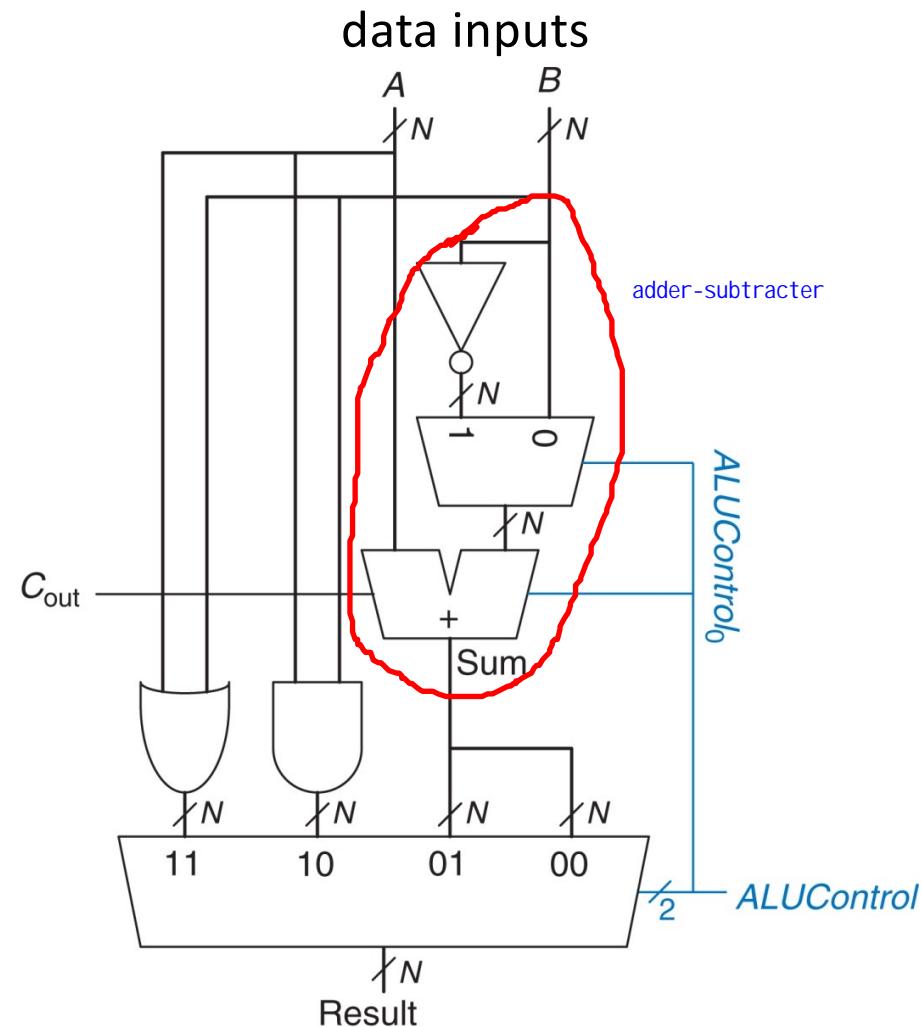
ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

*N-bit Logic circuitry
AND, OR*
4:1 multiplexer



ALU Implementation

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR



Add-Subtract Circuitry

- $A + B$
 - Normal addition
- $A - B$
 - $A + (-B)$
 - In 2's complement, $-B = B' + 1$
 - An inverter performs B'
 - We send ALUControl_0 as the carry input of the adder
 - ALUControl_0 is 1 when the ALU function is Subtract

Big Ideas

- **(Parallelism) Hardware is inherently parallel**
 - All logic gates in the ALU work in parallel when the circuit is presented with valid input
- **(Redundancy) Generality leads to redundancy**
 - ALU is a general-purpose circuit that can perform a variety of operations. Some work/effort is wasted
 - The output of OR/AND is wasted when ALUControl is **01**
- **(Control) Control circuitry comes with a cost**
 - ALU consumes more area than the individual functional units it combines (4:1 multiplexer is for controlling output)

ALUFLAGS

- We need information about the ALU output
 - Is the result negative (**N**)?
 - Is the result zero (**Z**)?
 - Is there a carry out (**C**)?
 - Is there an overflow (**V**)?
- Many scientific algorithms rely on flags for the “next steps”
 - If overflow, discard result, and redo
 - Carry out is the carry in for another operation
 - If the result is negative: do {...}; else do {...}

*Flags are only relevant for arithmetic operations (ALUControl₁ = **0**)*

ALUFLAGS

- Negative
 - Check the MSB of result
- Zero
 - NOR all bits of the result (same as invert then AND)
- Carry
 - AND ALUControl₁ with C_{out} from the adder
- Overflow
 - **Option # 1:** Use A and B to compute overflow
 - **Option # 2:** Use A and the output of 2:1 multiplexer to compute overflow

Option # 1 for Overflow

The following scenarios generate overflow, i.e., the overflow flag needs to be asserted (1)

	ALControl ₀	A ₃₁	B ₃₁	S ₃₁
Scenario # 1	0 (Add)	0	0	1
Scenario # 2	0 (Add)	1	1	0
Scenario # 3	1 (Subtract)	0	1	1
Scenario # 4	1 (Subtract)	1	0	0

Case # 1 in plain English: When doing A + B , if A and B are +ve, and the sum is -ve

Case # 2: A + B, if A and B are -ve, and the sum is +ve

Case # 3: A – B, if A is +ve and and B is -ve, and the sum is -ve

Case # 4: A – B, if A is -ve and and B is +ve, and the sum is +ve

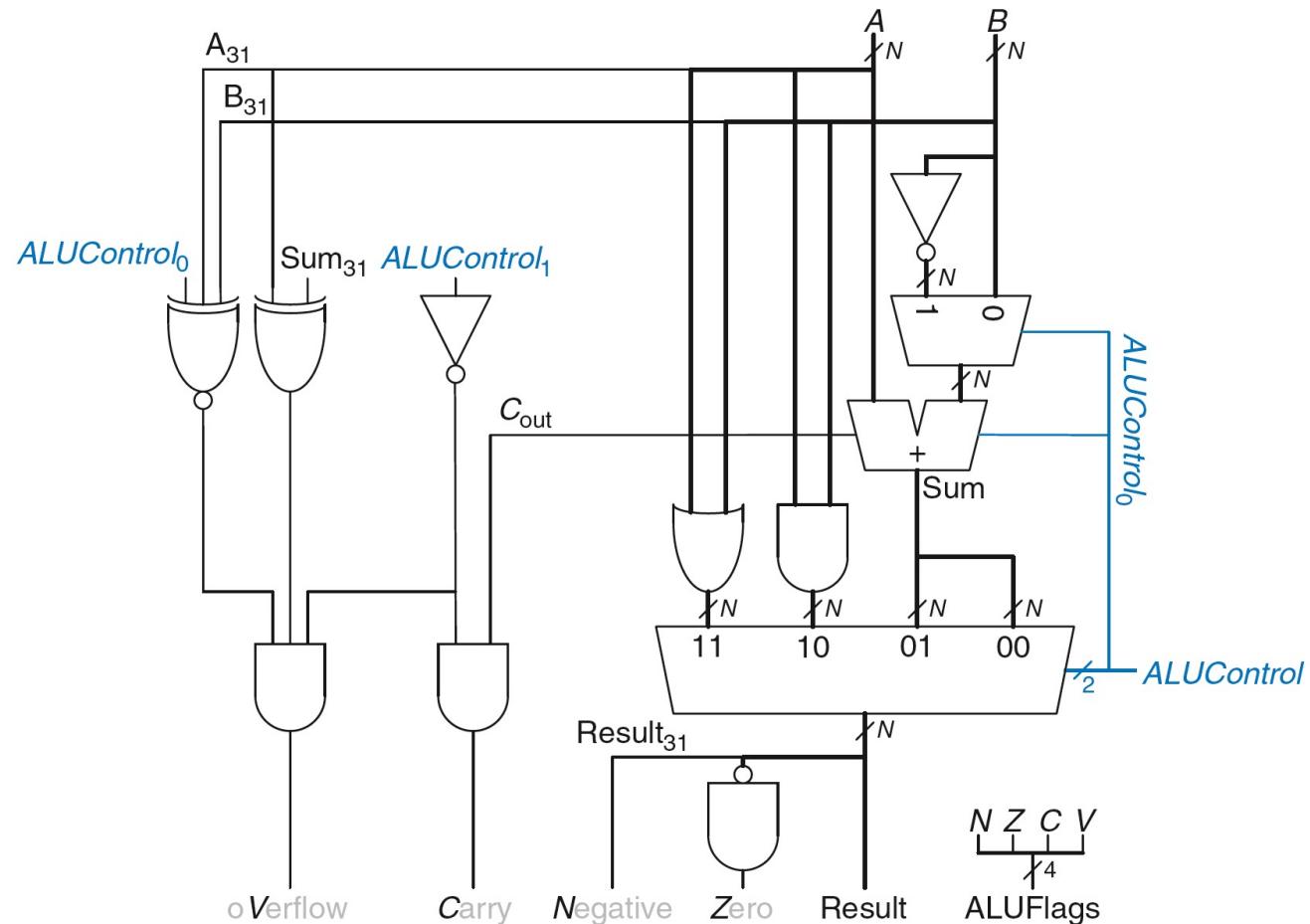
Option # 1 for Overflow

The following scenarios generate overflow, i.e., the overflow flag needs to be asserted (1)

	ALControl ₀	A ₃₁	B ₃₁	S ₃₁
Scenario # 1	0 (Add)	0	0	1
Scenario # 2	0 (Add)	1	1	0
Scenario # 3	1 (Subtract)	0	1	1
Scenario # 4	1 (Subtract)	1	0	0

- Overflow is 1 whenever there is an even number of 1's among ALUControl₀, A₃₁, and B₃₁
 - XNOR ALUControl₀, A₃₁, and B₃₁
- Overflow is 1 whenever A₃₁ and S₃₁ are different
 - XOR A₃₁ and S₃₁

Option # 1 for Overflow



Option # 2

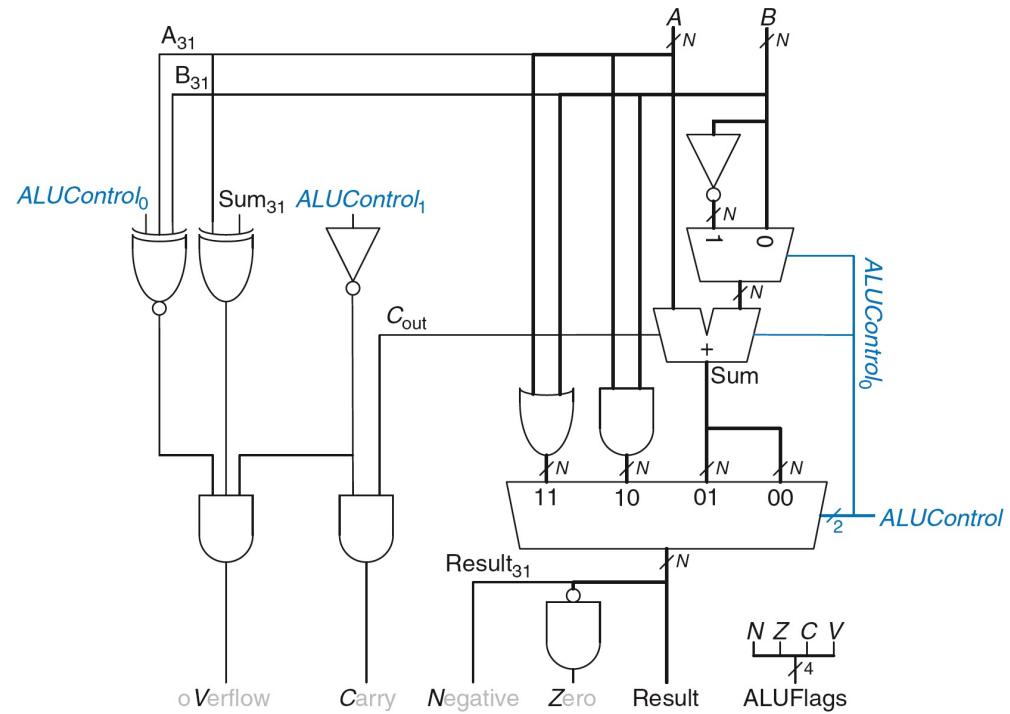
- Use A and the output of 2:1 mux
 - B if the instruction is an Add and $-B$ if the instruction is a subtract
- Easy to reason conceptually
 - If $A - B$ is the same as $A + (-B)$ then everything is an add
 - There is no need to consider subtract separately when reasoning about overflow generation
- The circuitry is also much simpler
 - Homework assignment: Figure out the circuitry for overflow generation with option # 2

ALU Timing Analysis

Homework

picoseconds (10^{-12} seconds) = ps

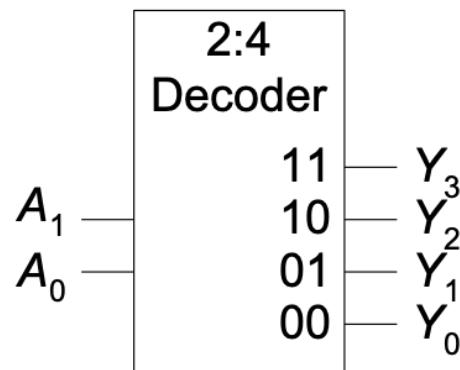
Element	Delay
Inverter	$t_{INV} = 1 \text{ ps}$
2:1 Mux	$t_{mux2} = 5 \text{ ps}$
4:1 Mux	$t_{mux4} = 8 \text{ ps}$
Adder	$t_{adder} = 14 \text{ ps}$
AND	$t_{AND} = 2 \text{ ps}$
OR	$t_{OR} = 2 \text{ ps}$



- Find t_{Result} in ps for the four ALU instructions/functions. Ignore overflow generation
 - Which function takes the longest time (and is the critical path)? Ignore wire delay
- Express t_{Result} in the form of an equation for Add and Subtract. What is the difference?

Decoders

- N **inputs** and 2^N **outputs**
- For each **input** combination, only one of the **outputs** is 1
 - The **outputs** are affectionately called *one-hot*



Decoders

- N **inputs** and 2^N **outputs**
- For each **input** combination, only one of the **outputs** is 1
 - The **outputs** are affectionately called *one-hot*

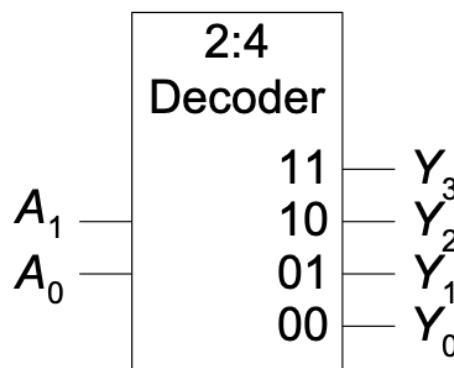
2:4 Decoder Truth Table

2:4 Decoder		A_1	A_0	Y_3	Y_2	Y_1	Y_0
A_1	11	0	0	0	0	0	1
	10	0	1	0	0	1	0
	01	1	0	0	1	0	0
	00	1	1	1	0	0	0

Decoders

- N **inputs** and 2^N **outputs**
- For each **input** combination, only one of the **outputs** is 1
 - The **outputs** are affectionately called *one-hot*

2:4 Decoder Truth Table and Boolean Equations

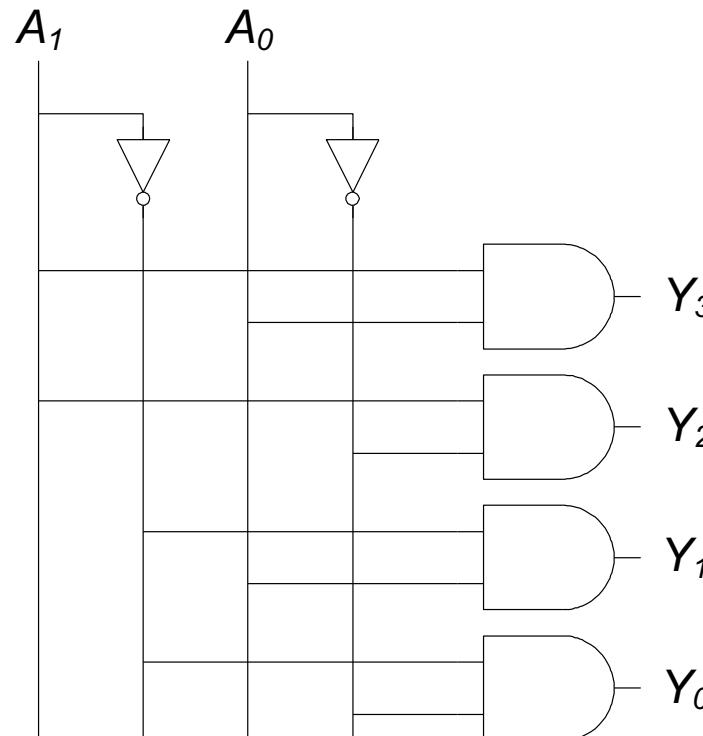
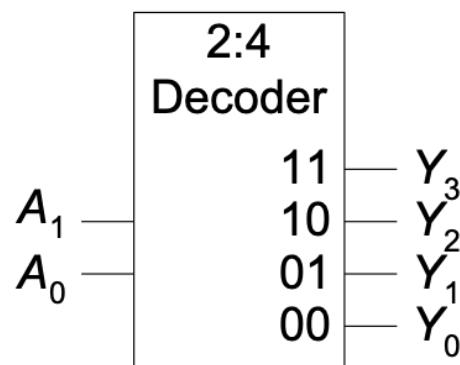


A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$$Y_0 = A_1'A_0'$$
$$Y_1 = A_1'A_0$$
$$Y_2 = A_1A_0'$$
$$Y_3 = A_1A_0$$

Decoders

- N **inputs** and 2^N **outputs**
- For each **input** combination, only one of the **outputs** is 1



$$Y_0 = A_1' A_0'$$

$$Y_1 = A_1' A_0$$

$$Y_2 = A_1 A_0'$$

$$Y_3 = A_1 A_0$$

Decoder Implementation

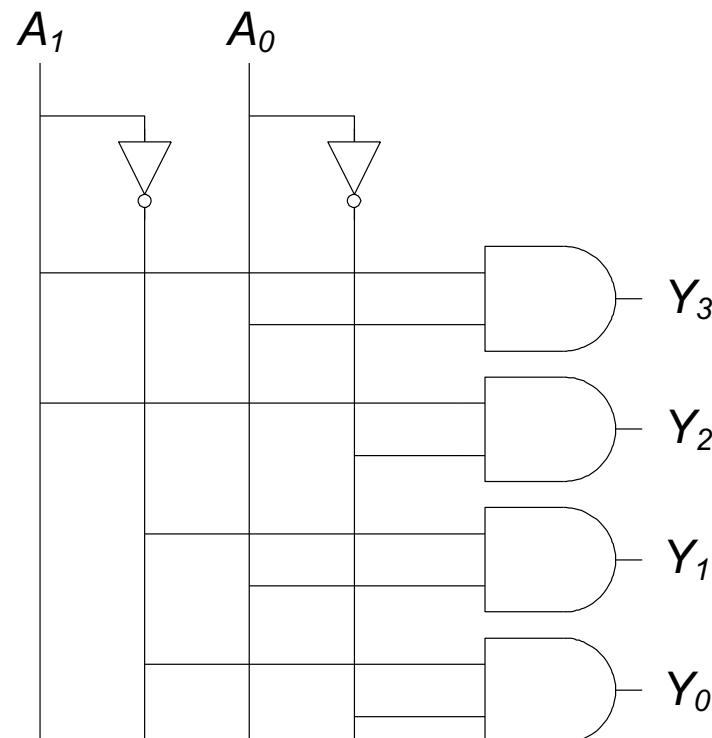
For each **input** combination, only one of the **outputs** is **1**

$$Y_0 = A_1' A_0'$$

$$Y_1 = A_1' A_0$$

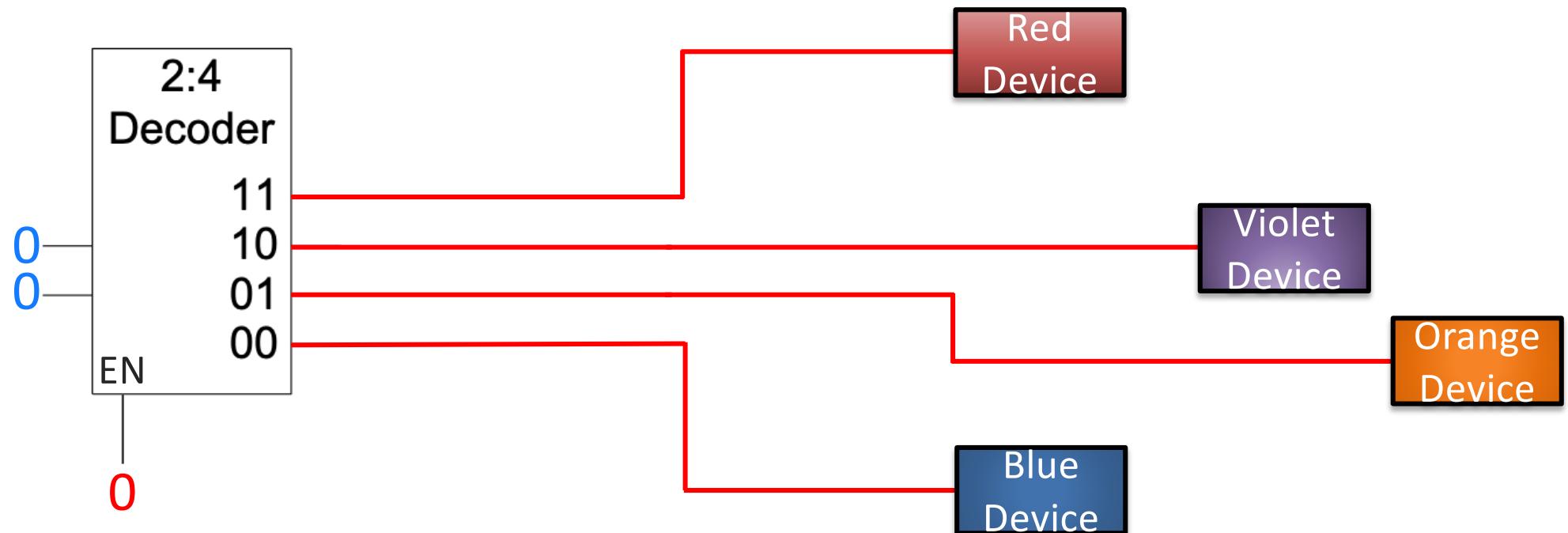
$$Y_2 = A_1 A_0'$$

$$Y_3 = A_1 A_0$$



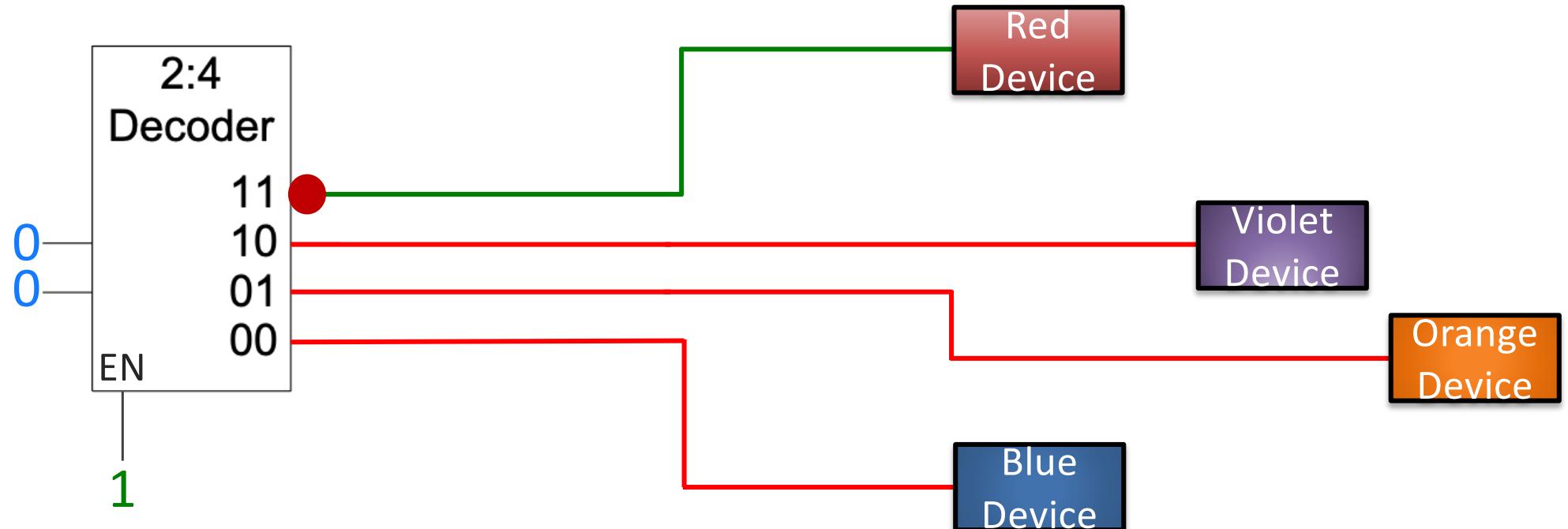
Uses of Decoders

For each **input combination**, only one of the **outputs** is **1**



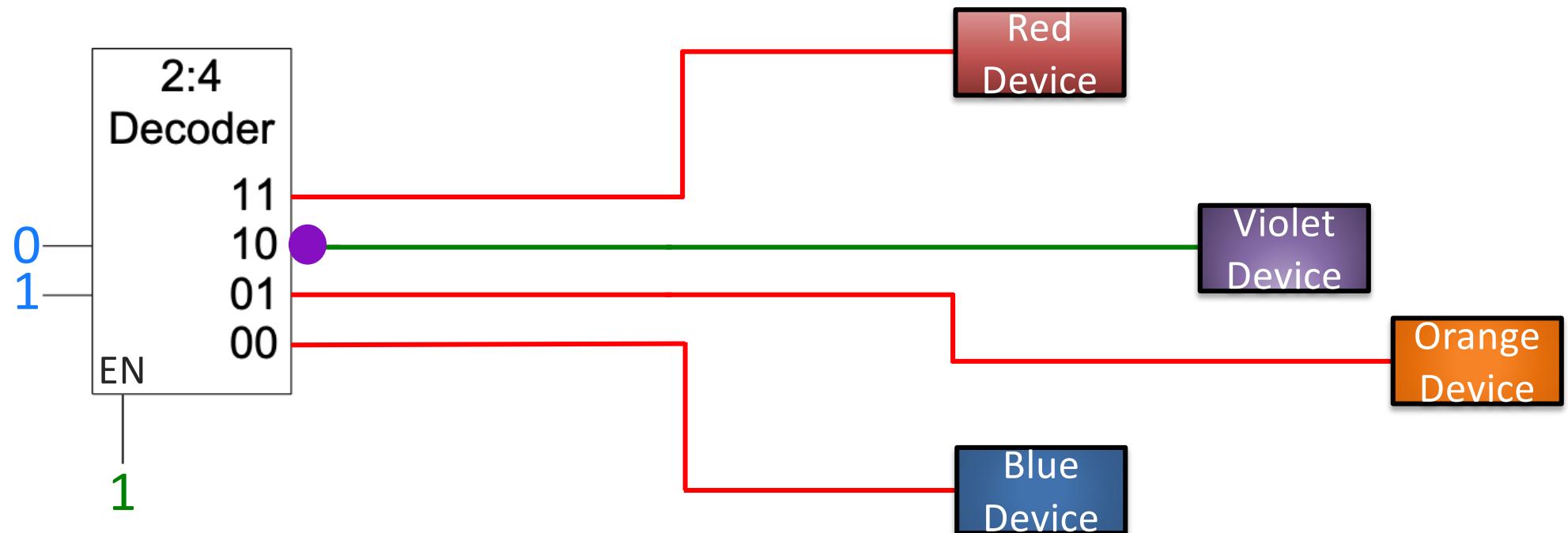
Uses of Decoders

For each **input combination**, only one of the **outputs** is **1**



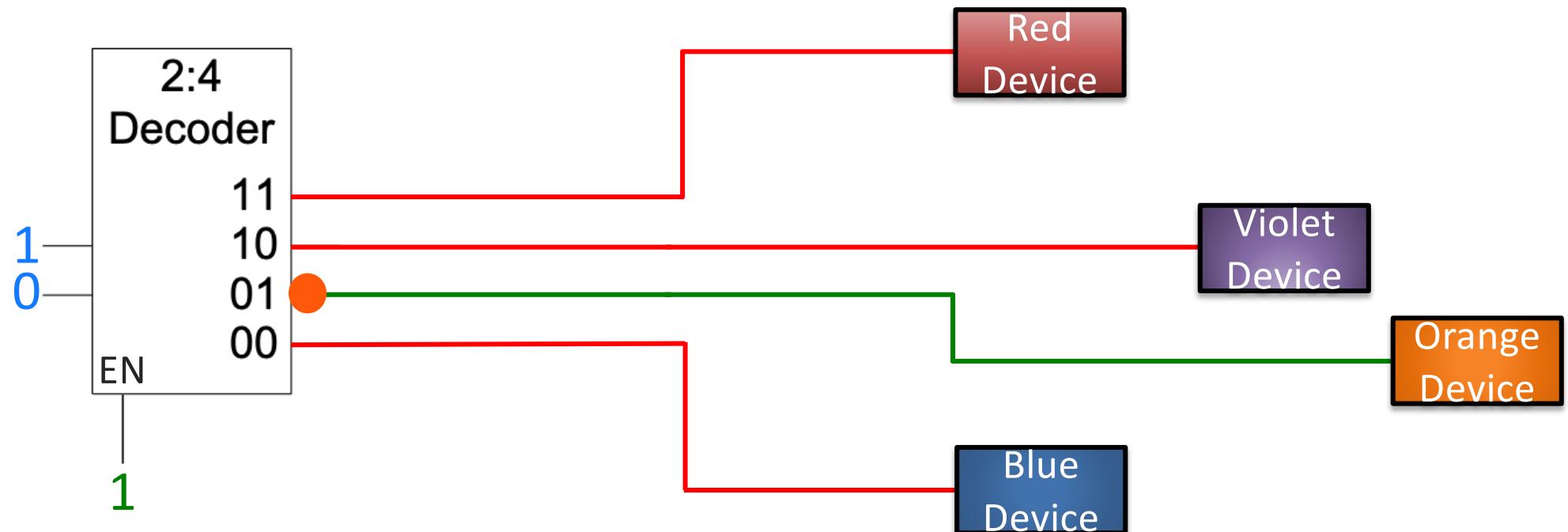
Uses of Decoders

For each **input combination**, only one of the **outputs** is **1**



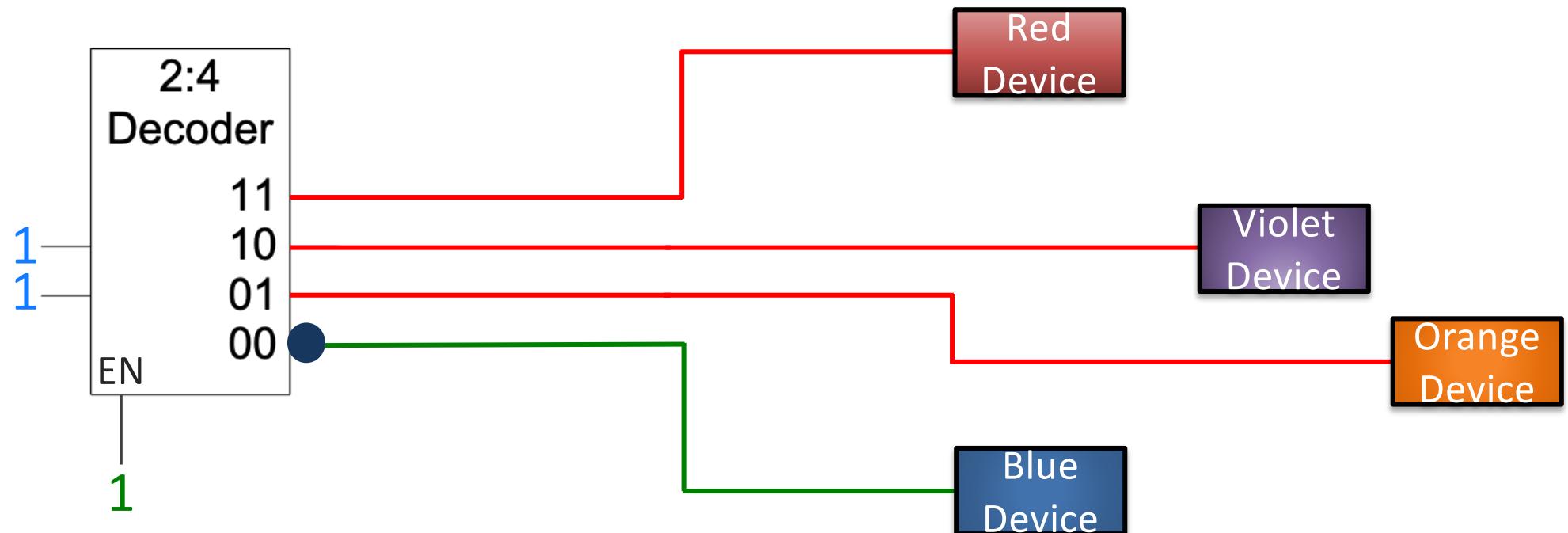
Uses of Decoders

For each **input combination**, only one of the **outputs** is **1**



Uses of Decoders

For each **input combination**, only one of the **outputs** is **1**



Multiplexer Logic

- We have seen how to build multiplexers and other combinational circuits with logic gates
- Question: Can we use a multiplexer to implement a logic gate?
 - If yes, build a 2-input AND gate from a 2:1 multiplexer
 - Note: You can connect any multiplexer input to 0 (*zero/ground*) or 1 (*high*)
 - Interestingly, the answer is yes!

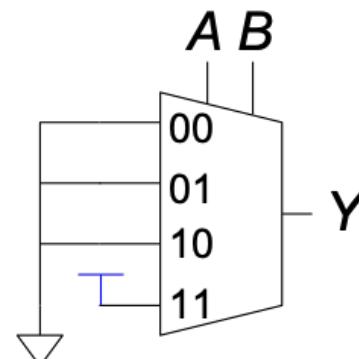
A 2^N -input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs

Multiplexer Logic

- Any truth table can be seen as a lookup table
 - If we lookup **00** in a 2-input AND truth table, we see **0**
 - If we lookup **10** in a 2-input OR truth table, we see **1**
- Multiplexers can be used as lookup tables
 - Connect the data inputs to **0** or **1**
 - Use inputs (A and B) as select lines/signals

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

$Y = AB$



ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

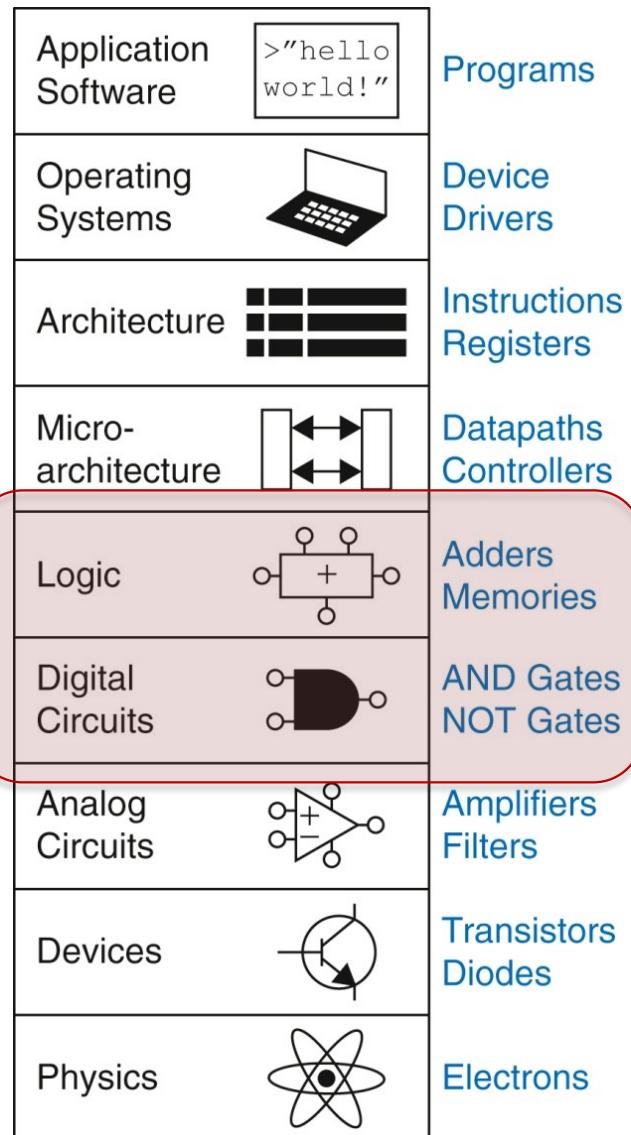
Plan: Week 3

Week 2: Logic gates & Combinational logic

Week 2: Multiplexers, ALU, and decoders

This Week: Boolean algebra (equation minimization)

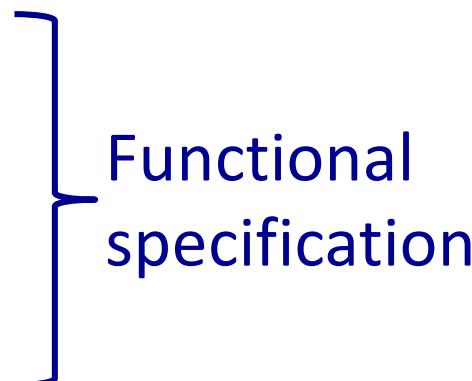
This Week: Sequential circuits (state & memory)



Broadening our horizon
“one layer at a time”

Implementing Combinational Logic

Steps in implementing combinational Logic

1. Initial specification (e.g., in English)
 2. Construct the truth table
 3. Derive the Boolean equation
 4. *Simplify the Boolean equation (use Boolean algebra)*
 5. Implement the equation using logic gates
- 
- Functional specification

Boolean Algebra

- The sum-of-products canonical form does not lead to the simplest logic gate implementation
 - In many cases, we can reduce the # AND gates
 - We can reduce the # literals in the equation
- We use Boolean algebra to simplify Boolean equations
 - Think of simplification in ordinary algebra except we are dealing with **0** and **1**

Boolean Algebra

- Boolean algebra consists of
 - Axioms (*correct by definition*)
 - Theorems of one variable
 - Theorems of several variables
- Any theorem can be proved via the axioms
 - An axiom is the ground truth (cannot be proven wrong)
- The *Principle of Duality*
 - If the symbols **0** and **1** and the operators AND and OR are interchanged, the statement will still be correct

Boolean Axioms

Number	Axiom	Dual	Name
A1	$B = 0 \text{ if } B \neq 1$	$B = 1 \text{ if } B \neq 0$	Binary Field
A2	$\bar{0} = 1$	$\bar{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	$1 + 0 = 0 + 1 = 1$	AND/OR

Dual: Replace: \bullet with $+$
0 with 1

Boolean Theorems of One Variable

Number	Theorem	Dual	Name
T1	$B \bullet 1 = B$	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	$B + B = B$	Idempotency
T4		$\bar{\bar{B}} = B$	Involution
T5	$B \bullet \bar{B} = 0$	$B + \bar{B} = 1$	Complements

Dual: Replace: \bullet with $+$
0 with 1

Theorems: Several Variable

#	Theorem	Dual	Name
T6	$B \bullet C = C \bullet B$	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	$(B + C) + D = B + (C + D)$	Associativity
T8	$B \bullet (C + D) = (B \bullet C) + (B \bullet D)$	$B + (C \bullet D) = (B + C) (B + D)$	Distributivity
T9	$B \bullet (B + C) = B$	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \bar{C}) = B$	$(B + C) \bullet (B + \bar{C}) = B$	Combining
T11	$(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) = (B \bullet C) + (\bar{B} \bullet D)$	$(B + C) \bullet (\bar{B} + D) \bullet (C + D) = (B + C) \bullet (\bar{B} + D)$	Consensus

Warning: T8' (dual of T8) differs from traditional algebra: OR (+) distributes over AND (\bullet)

Proving Theorems

- **Method 1:** Perfect induction
 - Check all possible input combinations (proof by exhaustion)
 - *Two expressions are equal if they produce the same value for every possible input combination*
- **Method 2:** Use other theorems/axioms to simplify equations
 - As in ordinary algebra, make one side of the equation look like the other

Example: Perfect Induction

Number	Theorem	Name
T6	$B \bullet C = C \bullet B$	Commutativity

B	C	BC	CB
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Example: Perfect Induction

Number	Theorem	Name
T9	$B \bullet (B+C) = B$	Covering

B	C	$(B+C)$	$B(B+C)$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Method 2: T9 (Covering)

Number	Theorem	Name
T9	$B \bullet (B + C) = B$	Covering

Method 2: Prove true using other axioms and theorems.

$$\begin{aligned} B \bullet (B + C) &= B \bullet B + B \bullet C && \text{T8: Distributivity} \\ &= B + B \bullet C && \text{T3: Idempotency} \\ &= B \bullet (1 + C) && \text{T8: Distributivity} \\ &= B \bullet (1) && \text{T2: Null element} \\ &= B && \text{T1: Identity} \end{aligned}$$

Method 2: T10 (Combining)

Number	Theorem	Name
T10	$(B \bullet C) + (B \bullet \bar{C}) = B$	Combining

Prove true using other axioms and theorems:

$$\begin{aligned} B \bullet C + B \bullet \bar{C} &= B \bullet (C + \bar{C}) && \text{T8: Distributivity} \\ &= B \bullet (1) && \text{T5': Complements} \\ &= B && \text{T1: Identity} \end{aligned}$$

Simplifying Boolean Equations

- A basic principle for simplifying sum-of-product equations
 - $PA + PA' = P$
 - P is any implicant
 - $Y = A'B + AB = B(A'+A) = B(1) = B$
- An equation is *minimized* if
 - *it uses the fewest number of implicants*
 - *if there are multiple equations with the same number of implicants, then the one with the fewest literals*

Simplification Example – 1

$$Y = AB + AB'$$

$$Y = A \quad \text{T10: Combining}$$

or

$$= A(B + B') \quad \text{T8: Distributivity}$$

$$= A(1) \quad \text{T5': Complements}$$

$$= A \quad \text{T1: Identity}$$

Simplification Example – 2

$$Y = A(AB + ABC)$$

$$= A(AB(1 + C)) \quad \text{T8: Distributivity}$$

$$= A(AB(1)) \quad \text{T2': Null Element}$$

$$= A(AB) \quad \text{T1: Identity}$$

$$= (AA)B \quad \text{T7: Associativity}$$

$$= AB \quad \text{T3: Idempotency}$$

Simplification Example – 3A

$$Y = AB'C + ABC + A'BC$$

$$= AC(B + B') + A'BC \quad T8: \text{Distributivity}$$

$$= AC(1) + A'BC \quad T5: \text{Complements}$$

$$= AC + A'BC \quad T1: \text{Identity}$$

- The two implicants AC and BC share the minterm ABC
- *Are we stuck with simplifying only one of the minterm pairs?*

Simplification Example – 3B

$$Y = AB'C + ABC + A'BC$$

$$= AB'C + ABC + ABC + A'BC \quad T3': \text{Idempotency}$$

$$= (AB'C+ABC) + (ABC+A'BC) \quad T7': \text{Associativity}$$

$$= AC + BC \quad T10: \text{Combining}$$

- *The two implicants AC and BC are called prime implicants*
- *They cannot be combined with any other implicants in the equation to get a new implicant with fewer literals*

Simplification Example – 4

$$Y = A'B'C' + AB'C' + AB'C$$

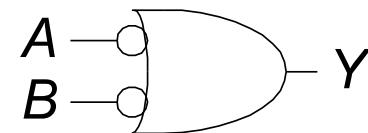
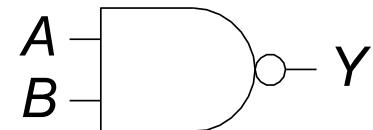
De Morgan's Theorem

#	Theorem	Dual	Name
T12	$\overline{B_0 \cdot B_1 \cdot B_2 \dots} = \overline{B_0 + B_1 + B_2 \dots}$	$\overline{B_0 + B_1 + B_2 \dots} = \overline{B_0 \cdot B_1 \cdot B_2 \dots}$	DeMorgan's Theorem

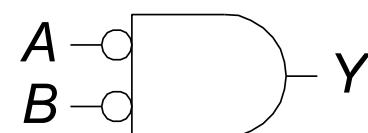
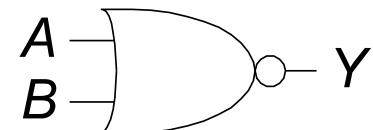
- *The complement of the product is the sum of the complements*
- ***Dual: The complement of the sum is the product of the complements***

De Morgan's Theorem

- $Y = \overline{AB} = \overline{A} + \overline{B}$



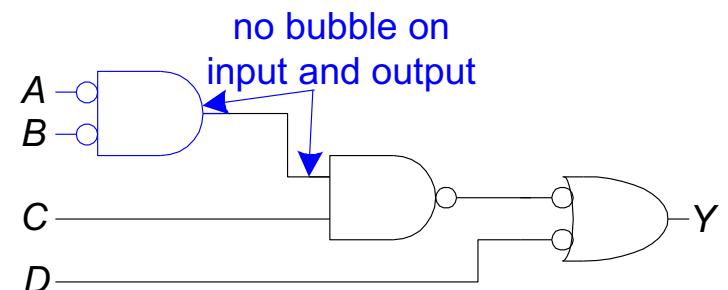
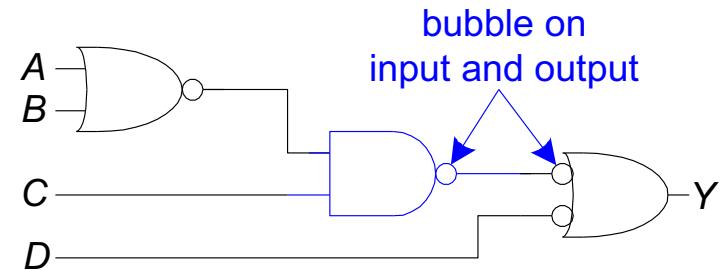
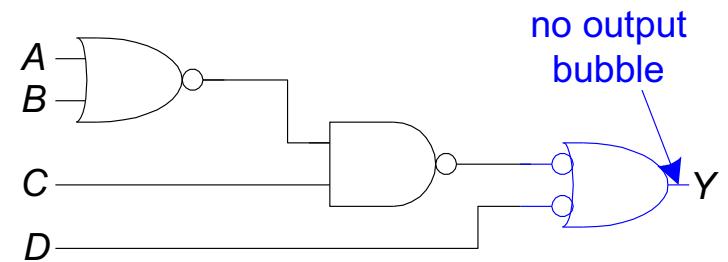
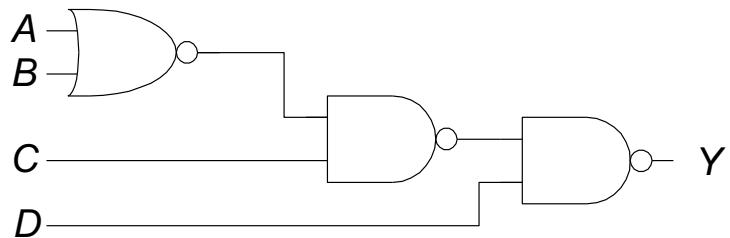
- $Y = \overline{A + B} = \overline{A} \cdot \overline{B}$



Bubble Pushing Rules

- Pushing bubbles backward/forward changes the body of the gate from AND/OR to OR/AND
- Pushing a bubble from output back to inputs put bubbles on all gate inputs
- Pushing bubbles on all gate inputs forward towards the output puts a bubble on the output

Bubble Pushing Example



$$Y = \overline{A}\overline{B}C + \overline{D}$$

Priority Circuit

Consider a theater reservation system. The system has four inputs, A_3, \dots, A_0 , and four outputs, Y_3, \dots, Y_0 . These signals can also be written as $A_{3:0}$ and $Y_{3:0}$. Each user asserts their input when they request the theater for the next day. The system asserts at most one output, granting the theater to the highest priority user. The dean, who is paying for the system, demands highest priority (3). The department chair, teaching assistant, and dorm social chair have decreasing priority. *Write a truth table and Boolean equations for the system. Sketch a circuit that performs this function.*

Note: The system is called a four-input priority circuit. We can write equations and simplify them using Boolean algebra. Fortunately, we can find the simplified equations via inspection

Priority Circuit

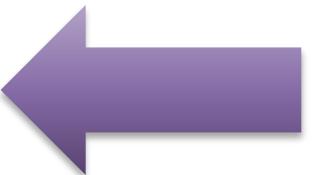
A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	x	0	0	1	0
0	1	x	x	0	1	0	0
1	x	x	x	1	0	0	0

$$Y_3 = A_3$$

$$Y_2 = A_3'A_2$$

$$Y_1 = A_3'A_2'A_1$$

$$Y_0 = A_3'A_2'A_1'A_0$$

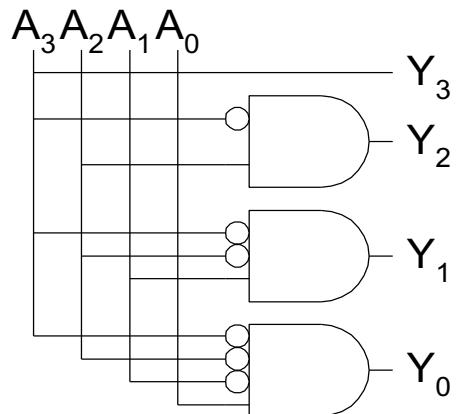


A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0

Priority Circuit

X = don't care (value does not impact output)

A_3	A_2	A_1	A_0		Y_3	Y_2	Y_1	Y_0
0	0	0	0		0	0	0	0
0	0	0	1		0	0	0	1
0	0	1	X		0	0	1	0
0	1	X	X		0	1	0	0
1	X	X	X		1	0	0	0



A_3	A_2	A_1	A_0	γ_3	γ_2	γ_1	γ_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

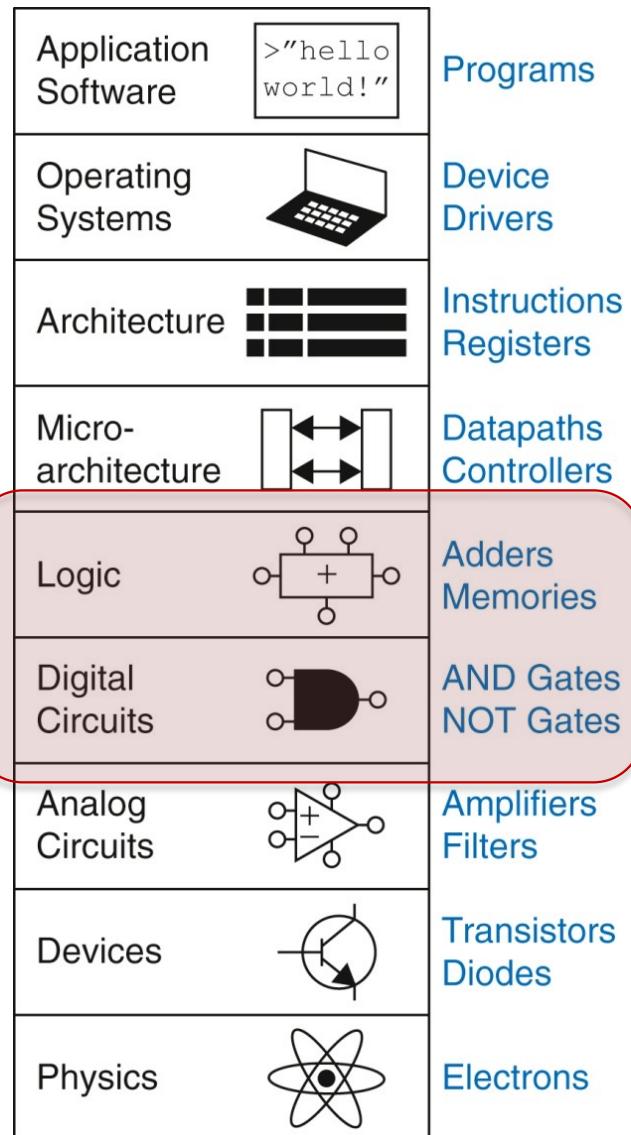
Plan: Week 3

Week 2: Logic gates & Combinational logic

Week 2: Multiplexers, ALU, and decoders

This Week: Boolean algebra (equation minimization)

This Week: Sequential circuits (state & memory)



Broadening our horizon
“one layer at a time”

Sequential Circuits

- Sequential logic has memory
- The output of sequential logic depends on both the current input values and the *state* of the system
- What is *state*?
 - A set of bits called *state variables* constitute *state*
 - *They inform us everything about the past we need to know to predict the future*

What is State?

- Consider a **very** simple traffic light controller
 - **RED, YELLOW, GREEN**
 - How many bits do I need to remember the state of the traffic light?
 - We need a minimum of two state variables (or bits) to store the state of the traffic light
- *To remember/store the state of the system (traffic light), we need an element with memory*

Storing One Bit

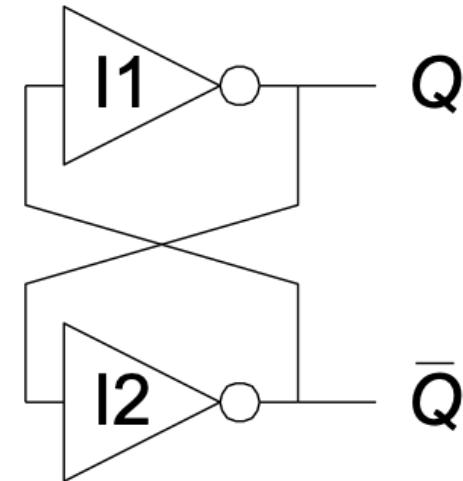
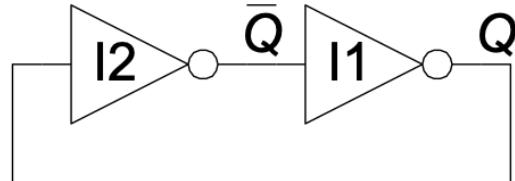
One bit represents two possible states. To store one bit, we need

- An element with two stable states
- The *ability to change the state*

Let us focus on the “*ability to change state*” later and first find the bistable element

Memory

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



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

Analysis of Bistable Element

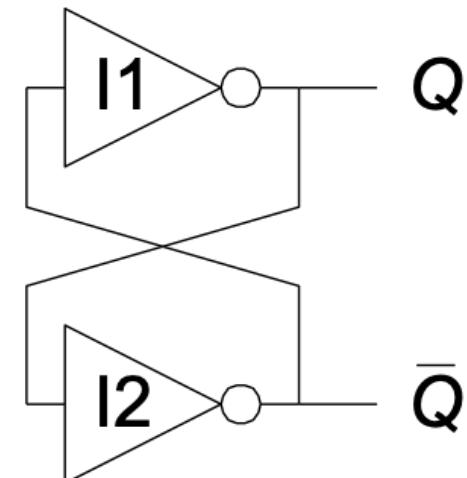
- How should we analyze circuits with cyclic paths?
 - When the circuit is switched on, Q is either **0** or **1** (**scenarios**)
 - Show that output is stable (**consequence – A**)
 - Show that Q and Q' are complements of each other (**consequence – B**)

Scenario # 1: Q = **0** (**FALSE**)

- I2 receives **0**, Q' = **1**: **B** is satisfied
- Q' = **1**, Q = **0**, consistent with our original assumption and hence stable: **A** is satisfied

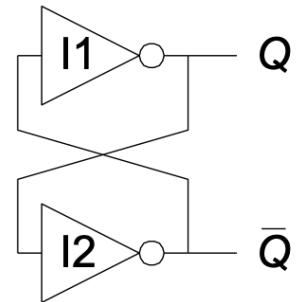
Scenario # 2: Q = **1** (**TRUE**)

- I2 receives **1**, Q' = **0**: **B** is satisfied
- Q' = **0**, Q = **1**, consistent with our original assumption and hence stable: **A** is satisfied



Bistable Element: Observations

- Bistable element has two stable states
 - Can store one bit of information
- The value of Q reveals about past, necessary to explain the future behavior
 - If $Q = 0$, it will remain 0 forever
 - If $Q = 1$, it will remain 1 forever
- Q' is an acceptable choice for storing the state variable
- When power is first applied, the state of the bistable element is unpredictable
 - Bistable element is *not practical* because the user lacks inputs to control state
 - We need something else!



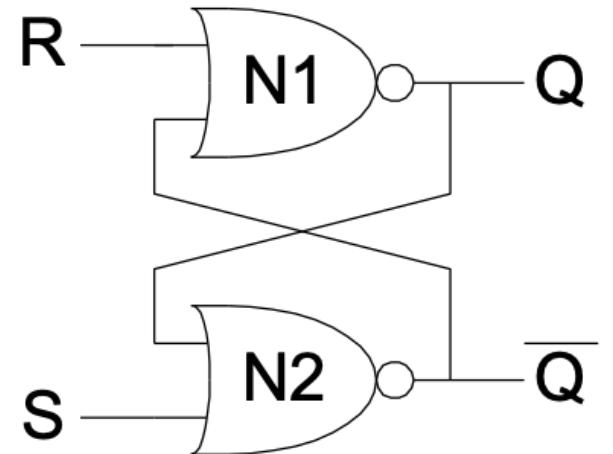
SR Latch

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

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

NOR gate revision

- When both inputs are 0, the output is 1
- If any of the inputs is 1, the output is 0

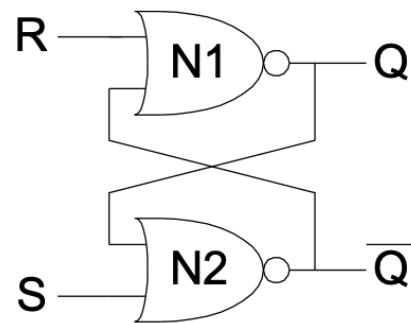


SR Latch: Analysis

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

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

Whiteboard: SR Latch



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

S	R	Q
0	0	
0	1	
1	0	
1	1	

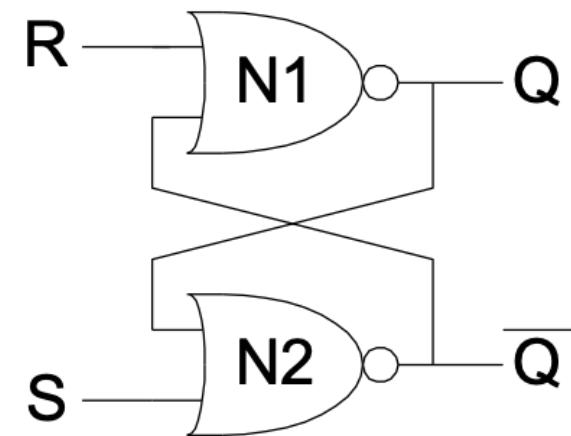
SR Latch: Analysis

Scenario # 1 (Reset): $R = 1, S = 0$

- N1 sees at least one TRUE (1) input
 - $Q = \text{FALSE } (0)$
- N2 sees both Q and S FALSE
 - $Q' = \text{TRUE } (1)$

Scenario # 2 (Set): $R = 0, S = 1$

- N1 sees 0 and Q'
 - What is Q' ?
- N2 sees at least one TRUE input
 - $Q' = \text{FALSE } (0)$
- Revisit N1 ($R = 0$ and $Q' = 0$)
 - $Q = \text{TRUE } (1)$



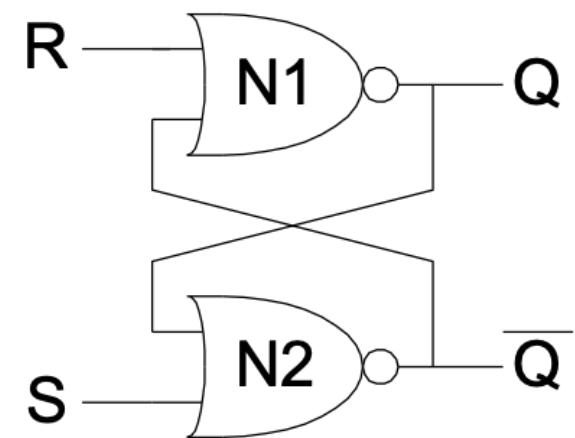
SR Latch: Analysis

Scenario # 3 (Sim-1): $R = 1, S = 1$

- N1 sees at least one TRUE (1) input
 - $Q = \text{FALSE } (0)$
- N2 sees at least one TRUE (1) input
 - $Q' = \text{FALSE } (0)$

Scenario # 4 (Sim-0): $R = 0, S = 0$

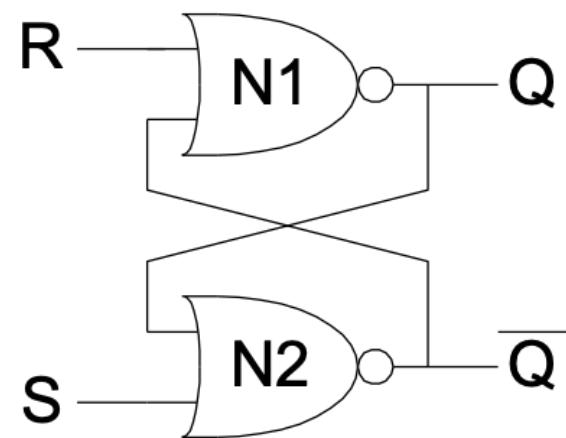
- N1 sees 0 and Q'
 - What is $Q'?$
- N2 sees 0 and Q
 - What is Q?
- **We are stuck!**
 - Wait: remember the cross-coupled inverter? Q can be 0 or 1 😊



SR Latch: Analysis

Scenario # 4-A (Sim-0): $R = 0, S = 0, Q = 0$

- N2 sees $S = 0$ and $Q = 0$
 - $Q' = 1$
- N1 sees one TRUE (1) input
 - $Q = 0$ (hindsight: Q is indeed 0 as assumed)



Scenario # 4-B (Sim-0): $R = 0, S = 0, Q = 1$

- N2 sees $Q = 1$
 - $Q' = \text{FALSE}$
- N1 receives two FALSE inputs
 - $Q = 1$ (hindsight: Q is indeed 1 as assumed)

SR Latch: Analysis

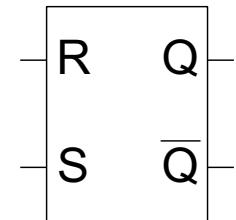
SR latch has inputs (unlike the cross-coupled inverters)
Four scenarios in the truth table (Sim = Simultaneous)

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

SR Latch: Observations

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

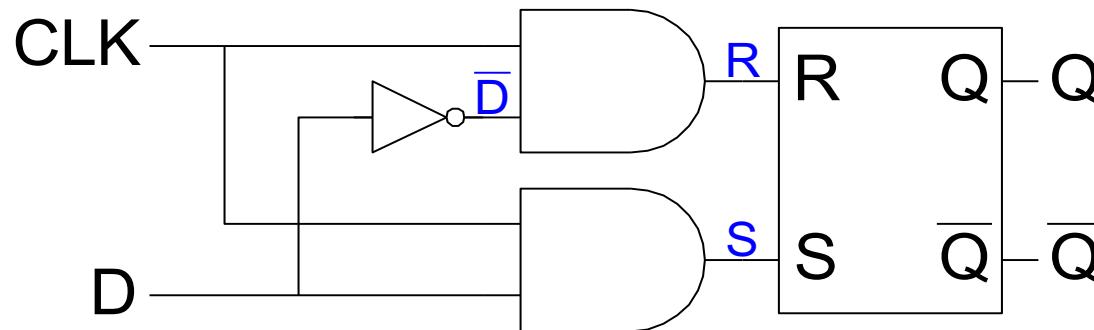
SR Latch
Symbol



D Latch

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

D Latch



Scenario # 1: $\text{CLK} = 0, \text{S} = 0, \text{R} = 0, \text{D} = X$

- The value of D is irrelevant
- $Q = Q_{\text{prev}}$ (remember the old value)

Latch is opaque (blocks new data from flowing to Q)

Scenario # 2: $\text{CLK} = 1, \text{D} = 0, \text{S} = 0, \text{R} = 1$

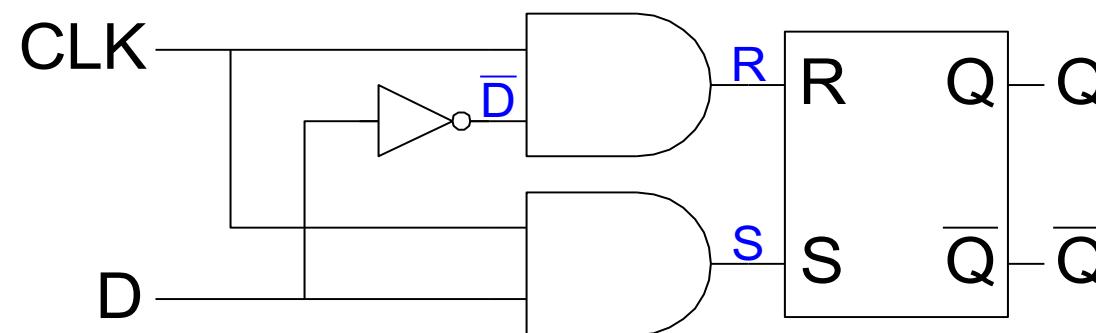
- The latch is reset

Latch is transparent (acts like a buffer)

Scenario # 3: $\text{CLK} = 1, \text{D} = 1, \text{S} = 1, \text{R} = 0$

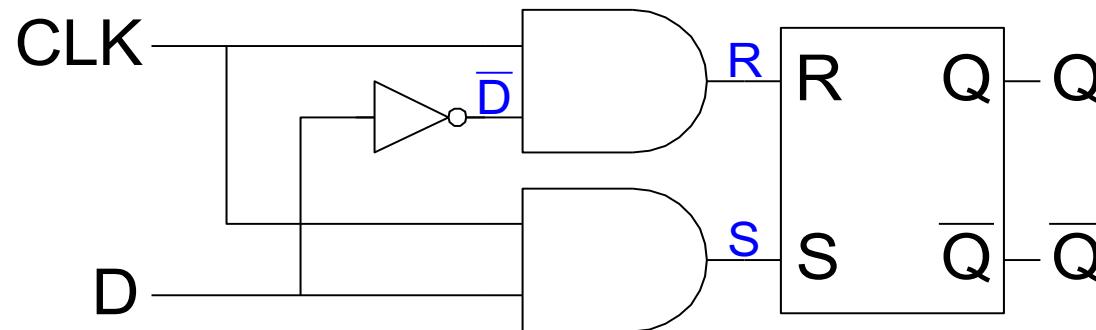
- The latch is set

Whiteboard: D Latch



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

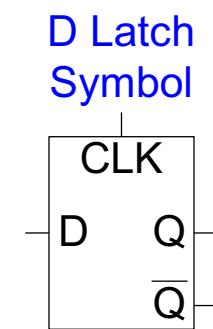
D Latch: Truth Table



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

D Latch: Observations

- D latch is a *level-triggered* or a *level-sensitive* circuit
 - Reacts to the level (**0** or **1**) of the CLK input
- D latch avoids the awkward case of both S and R asserted
- D latch changes its state continuously when CLK = **1**



Designing correct & efficient sequential circuits becomes easier when the state changes only at a specific instant in time instead of changing continuously. We need something else!

D Flip-Flop

- What is the problem with D latch?
 - The output changes continuously when $\text{CLK} = 1$
- We need a circuit element that *samples* the data input when CLK changes from **0** to **1** (or **1** to **0**)
 - This process is called sampling at the edge of a clock input
- The D flip-flop is called *edge-triggered*
- At all other times, the D flip-flop simply remembers its state



ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoib Akram

shoib.akram@anu.edu.au



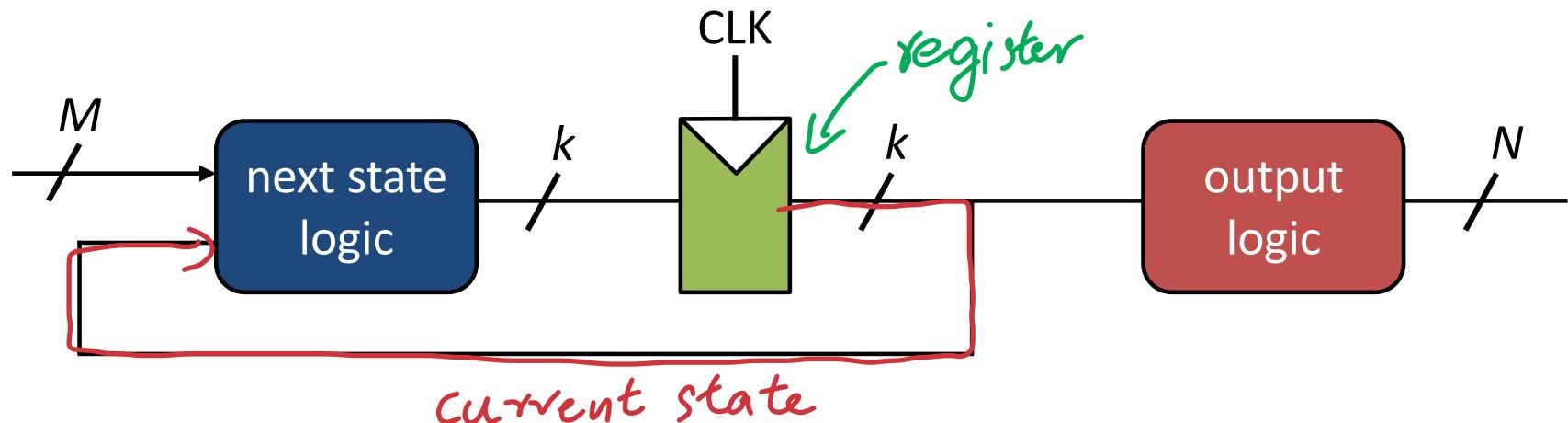
Australian
National
University

Two Sync. Sequential Circuits

- Two widely used synchronous sequential circuits
 - Finite state machine (FSM)
 - Pipelines
- FSMs can be used to solve many real-world problems
 - Traffic light controller, elevator controller, ...
- Pipelining helps reduce the clock period of synchronous sequential circuits (via parallelism)
- These circuits gives us greater insight into how synchronous sequential circuits behave

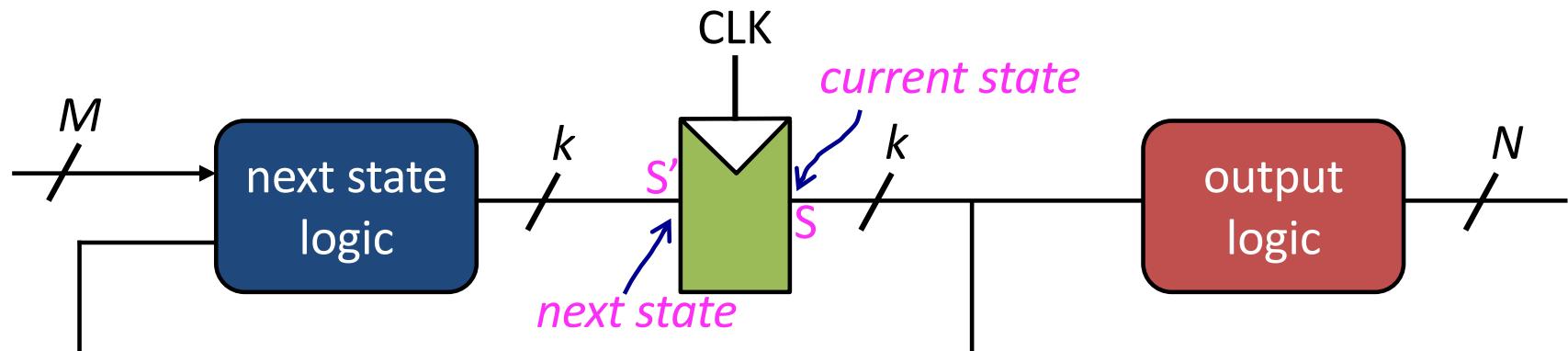
Finite State Machine (FSM)

- A *k-bit register* to store one of 2^k (finite) states
- The *next state logic* computes the next state
 - Next state depends on the current state and inputs
 - FSM advances to the next state on each clock edge
 - Remark: We use S for current state and S' for the next state
- The *output logic* computes the output based on the current state (Moore machine) and current state and inputs (Mealy machine)



Finite State Machine (FSM)

- In one cycle (say N), next state logic produces the next state (S') based on the current state (S) and the M inputs
- In the next cycle ($N+1$), the state register updates the current state (S) of the system to be equal to S'
- *Warning: S' (or S prime) has two possible meanings. We use the prime notation for NOT or Invert earlier, and now we will use it for next state (convention). The context will make it clear what we mean*

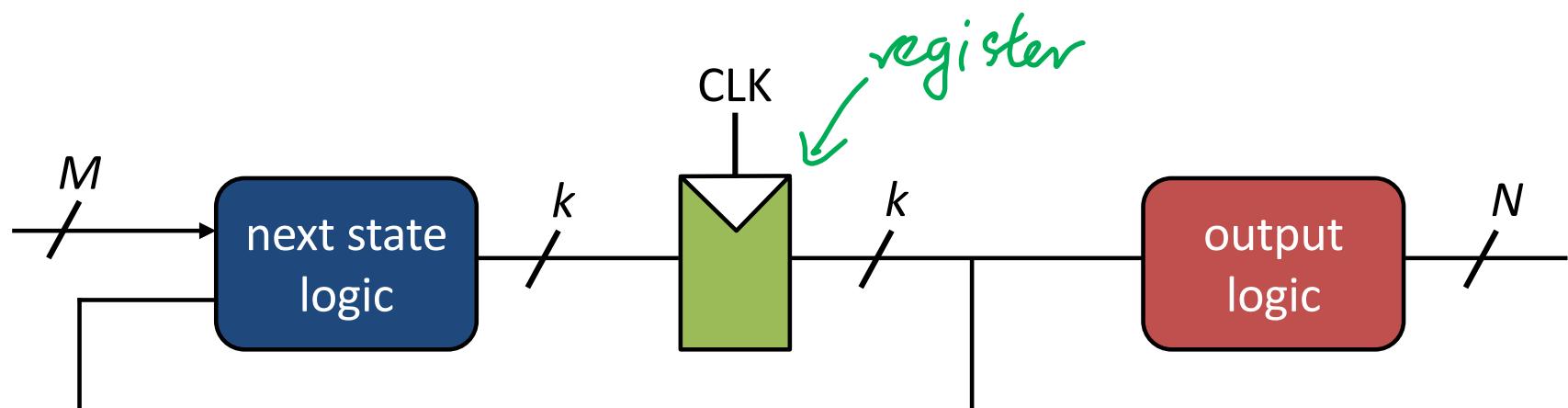


Slide added after the lecture

Implementing FSMs

Goal: Take the initial specification in English and build the FSM circuit using logic gates

- Use the FSM circuit template shown below
- Derive next state logic and output logic



Implementing FSMs

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

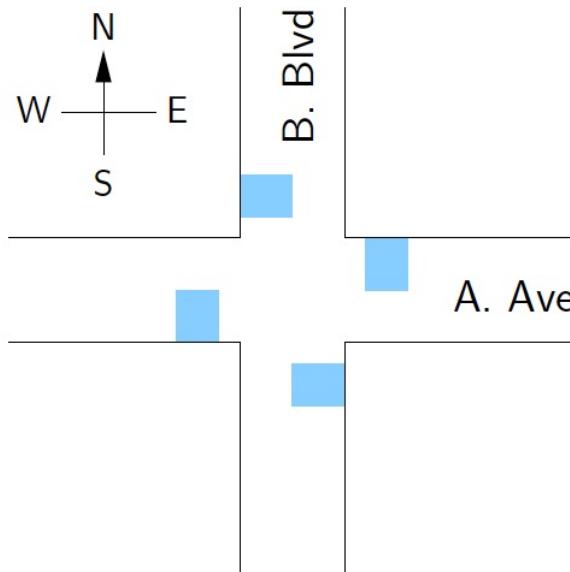
- Binary encoding for outputs
- Output table & Boolean equations

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

- Next state logic & output logic

Traffic Light Controller

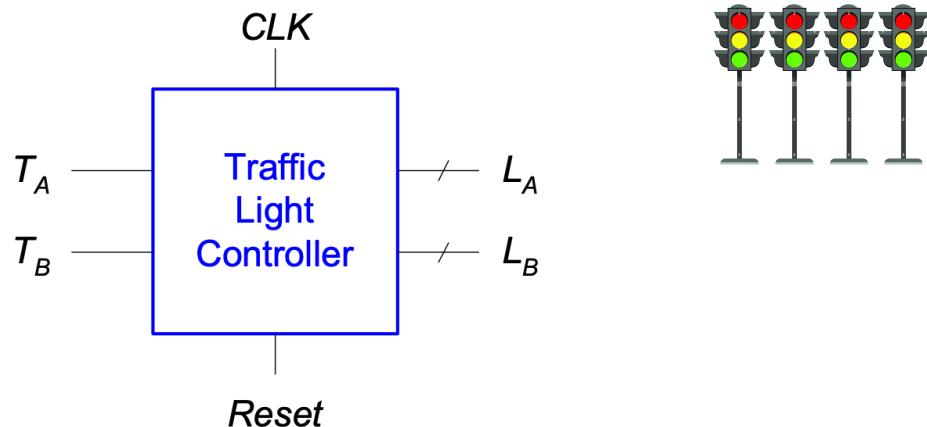
- Let's build a traffic light controller for a busy intersection
- Traffic sensors are built into the road
- Each sensor indicates if a street is empty or there are vehicles nearby



$T_A = (\text{eastbound traffic on } A) \text{ OR } (\text{westbound traffic on } A)$ *A是否有车*
 $T_B = (\text{northbound traffic on } B) \text{ OR } (\text{southbound traffic on } B)$ *B是否有车.*

Traffic Light Controller Problem

- 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

Traffic Light Controller Problem

- Clock with period of 5 seconds (frequency = 0.2 Hz)
 - On each rising edge, the lights may change based on traffic sensors
- A Reset input to put the controller in a known state

Implementing FSMs

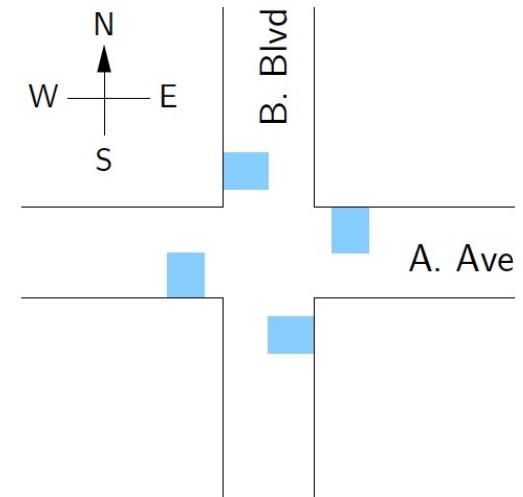
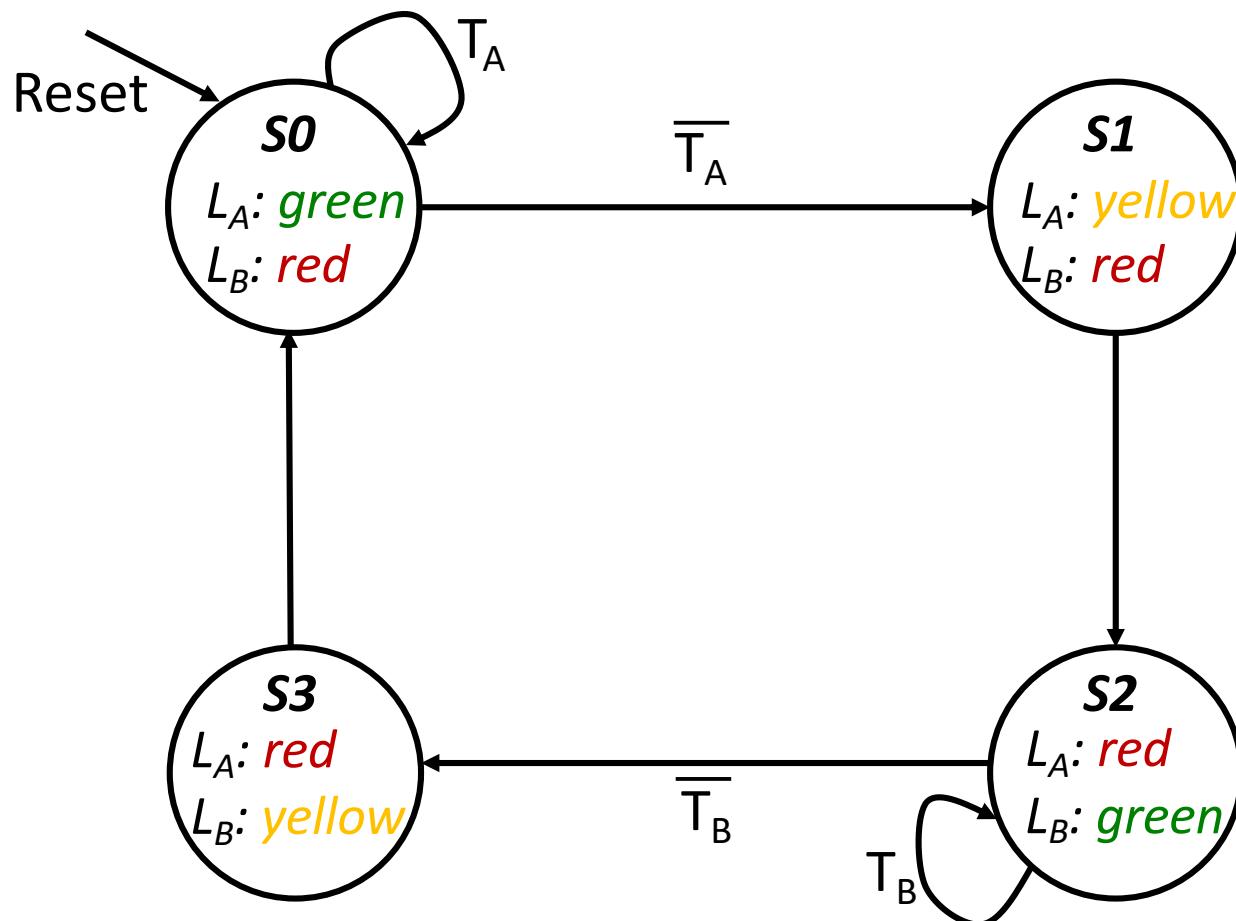
Step # 1: State transition diagram

- Formalize the specification and remove ambiguity

State Transition Diagram

- A state transition diagram graphically depicts
 - States with circles
 - Transitions (rising edge) with arcs
 - How does inputs affect the transitions?
 - How are outputs related to the current state?

State Transition Diagram



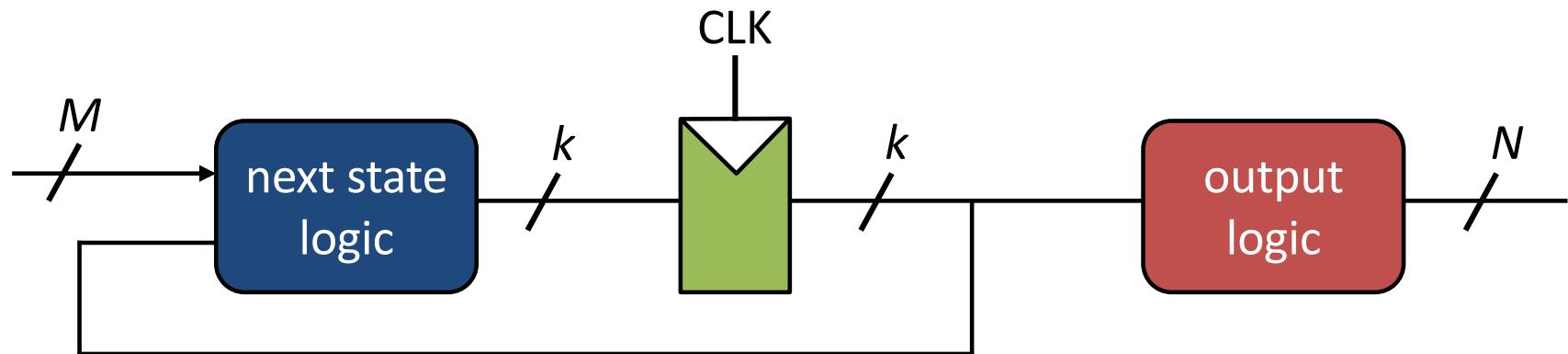
Implementing FSMs

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

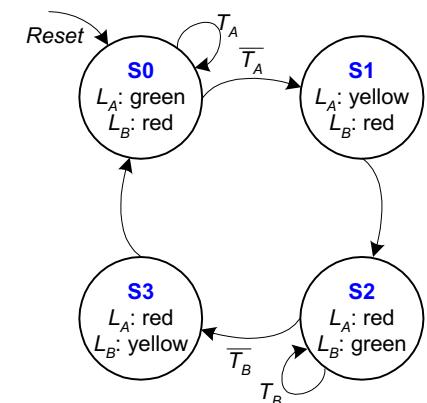


State Transition Table

- To move a step closer from specification to circuit design, a table listing the next states for all possible combinations of current state and input signals is helpful
 - Use X for don't cares to help keep the size of the table manageable

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



Encoding States

- There is a problem. Circuits do not understand S_0, S_1, \dots
- We need to store **0**'s and **1**'s in our state register
- Therefore, we need an encoding scheme for our states

State	Encoding
S_0	00
S_1	01
S_2	10
S_3	11

Note: $S_0 - S_3$ for states and S_0 and S_1 for bits in the state register

*S : current
 S' : next*

New State Transition Table

- We can substitute our state encodings into the state transition table
- The new version of the state transition table completely specifies the next state as a combinational logic function of the current state and input variables

Sum of products.

State Transition Table with Binary Encoding

State	Encoding	Current State		Inputs		Next State	
S ₀	00	S ₁	S ₀	T _A	T _B	S' ₁	S' ₀
S ₁	01	0	0	0	X	0	1
S ₂	10	0	0	1	X	0	0
S ₃	11	0	1	X	X	1	0
		1	0	X	0	1	1
		1	0	X	1	1	0
		1	1	X	X	0	0

Whiteboard: S'_1 Derivation

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 Transition Table with Binary Encoding

State	Encoding	Current State		Inputs		Next State	
S ₀	00	S ₁	S ₀	T _A	T _B	S' ₁	S' ₀
S ₁	01	0	0	0	X	0	1
S ₂	10	0	0	1	X	0	0
S ₃	11	0	1	X	X	1	0
$S'_1 = S_1 \oplus S_0$		1	0	X	0	1	1
$S'_0 = \overline{S_1} \overline{S_0} T_A + S_1 \overline{S_0} T_B$		1	0	X	1	1	0
		1	1	X	X	0	0

Implementing FSMs

Step # 1: State transition diagram ✓

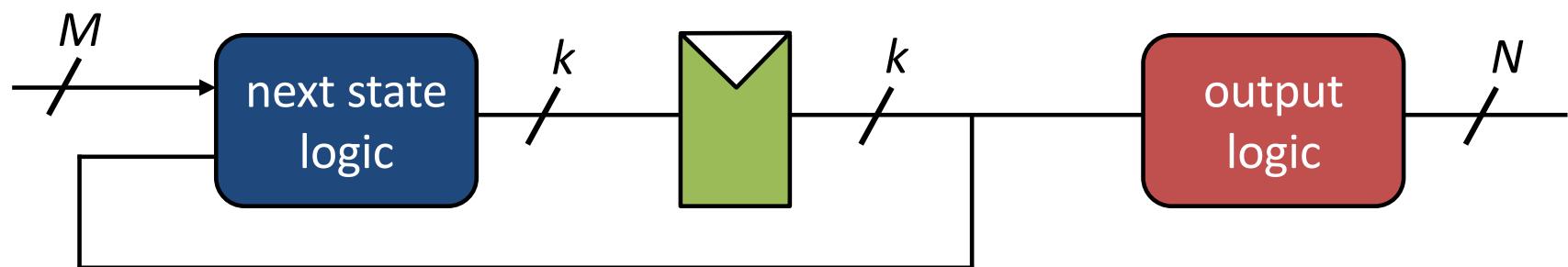
- Formalize the specification and remove ambiguity

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



Output Encoding & Table

In our Moore FSM, the output only depends on the current state

Output	Encoding	Current State		Outputs			
		S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
GREEN	00						
YELLOW	01	0	0	0	0	1	0
RED	10	0	1	0	1	1	0
Mealy : current state inputs.		1	0	1	0	0	0
		1	1	1	0	0	1

Output Encoding & Table

In our Moore FSM, the output only depends on the current state

Output	Encoding	Current State		Outputs			
		S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
GREEN	00	0	0	0	0	1	0
YELLOW	01	0	1	0	1	1	0
RED	10	1	0	1	0	0	0
$L_{A1} = S_1$		1	1	1	0	0	1
$L_{A0} = \overline{S_1}S_0$							
$L_{B1} = \overline{S_1}$							
$L_{B0} = S_1S_0$							

Implementing FSMs

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

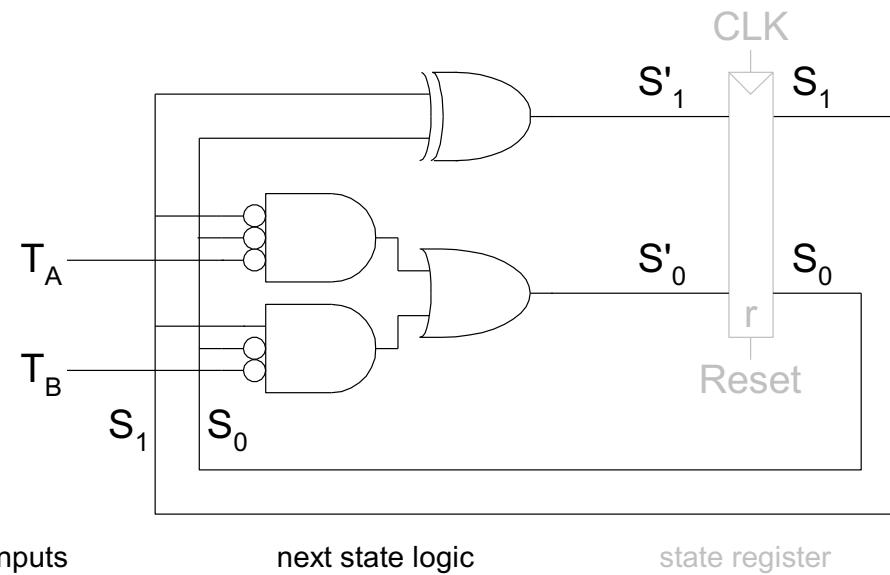
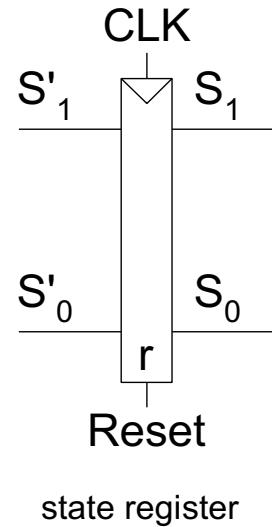
Step # 3: Derive the output logic ✓

- Binary encoding for outputs
- Output table & Boolean equations

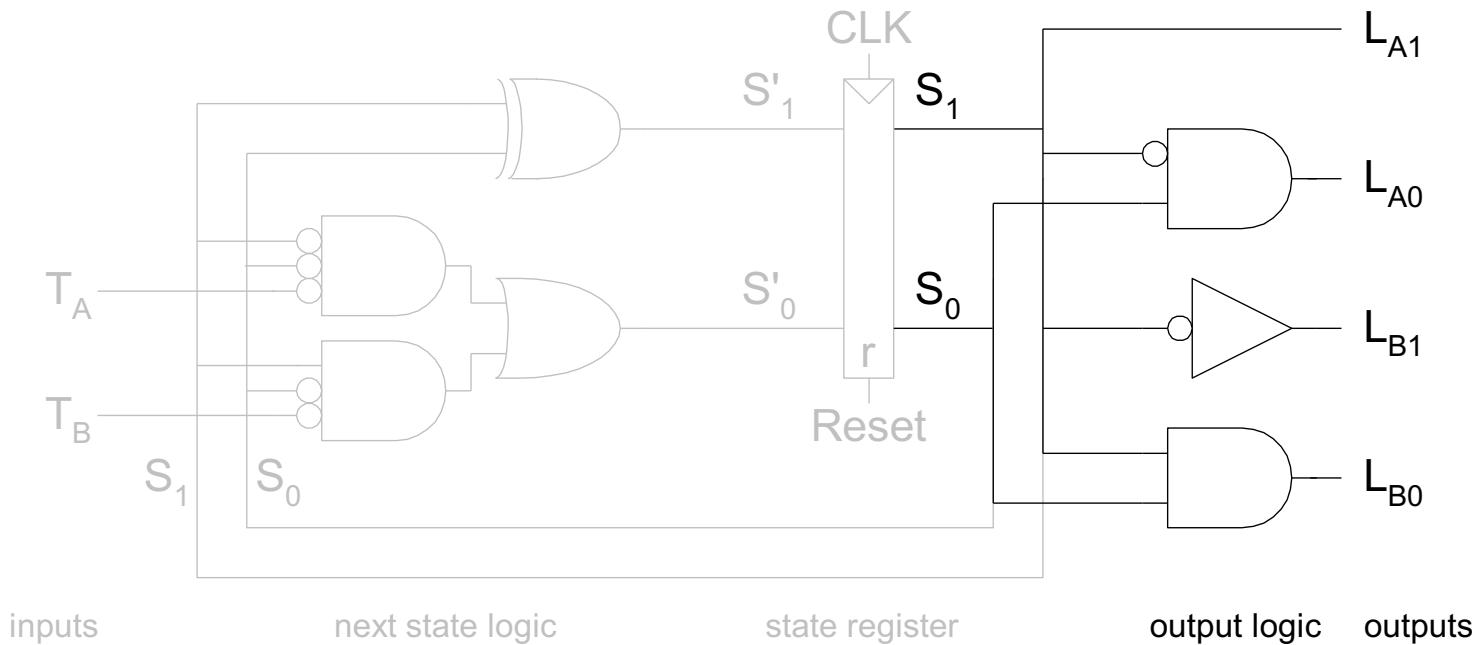
Step # 4: Turn the Boolean equations into logic gate implementation

- Next state logic & output logic

State Machine Circuit



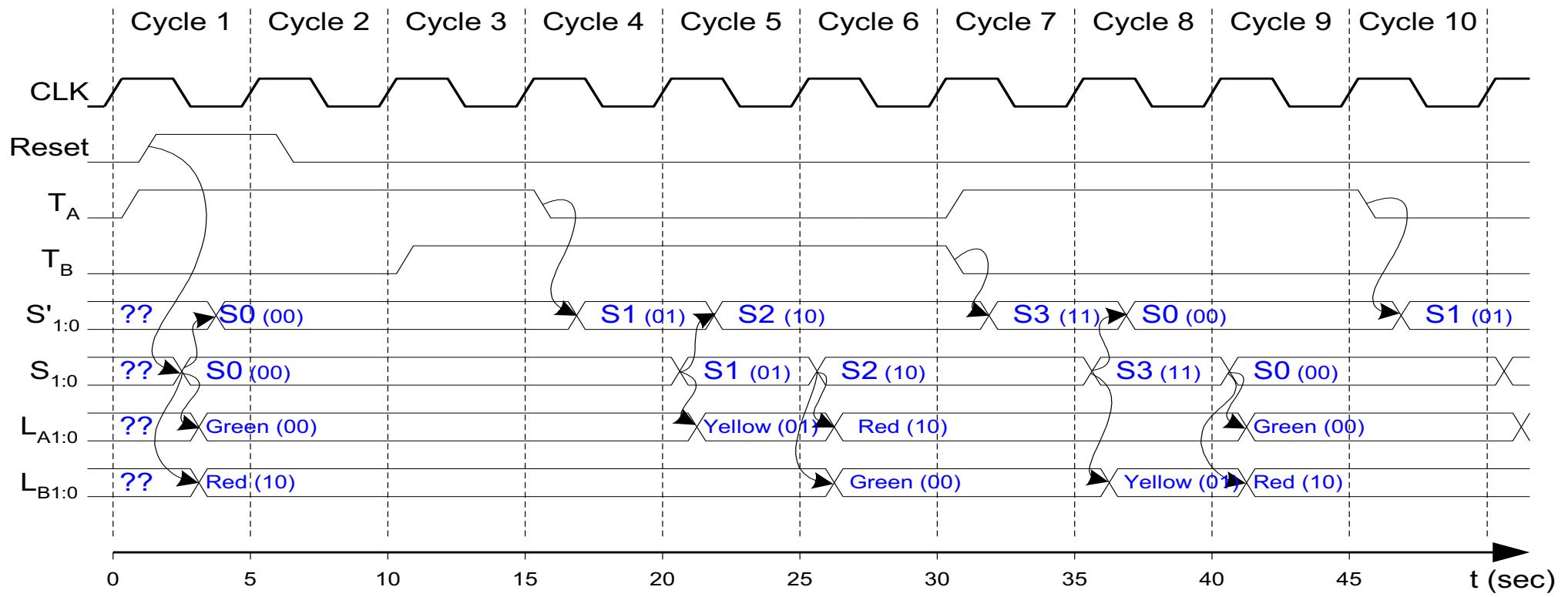
State Machine Circuit



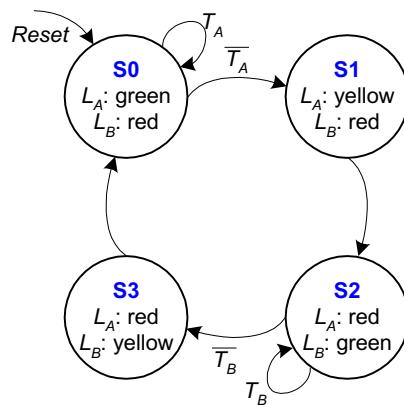
Note

Moore FSM: The outputs depend on the current state alone (e.g., traffic light controller)

Mealy FSM: The outputs depend on the current state and the inputs (simple extension to Moore FSM)



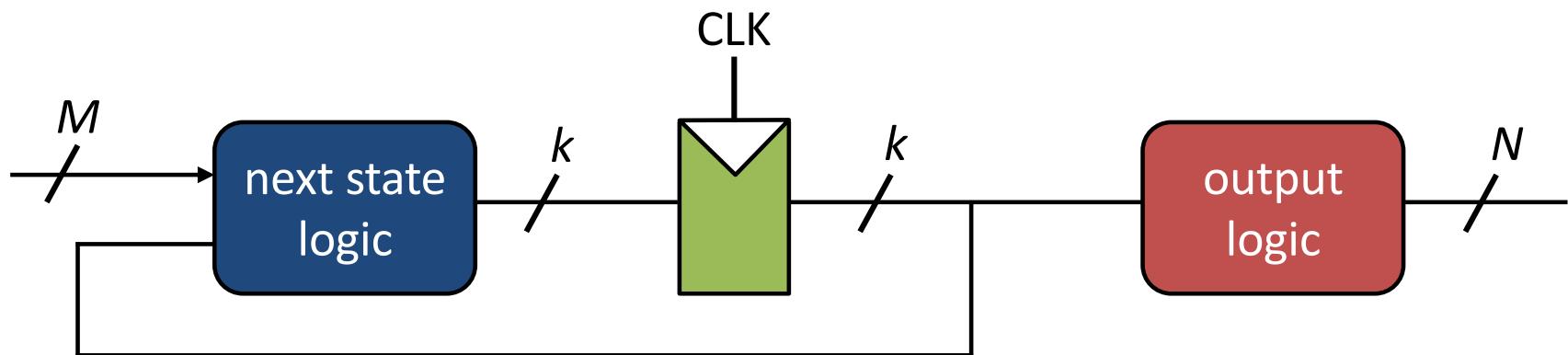
Timing Diagram



Quiz – 1

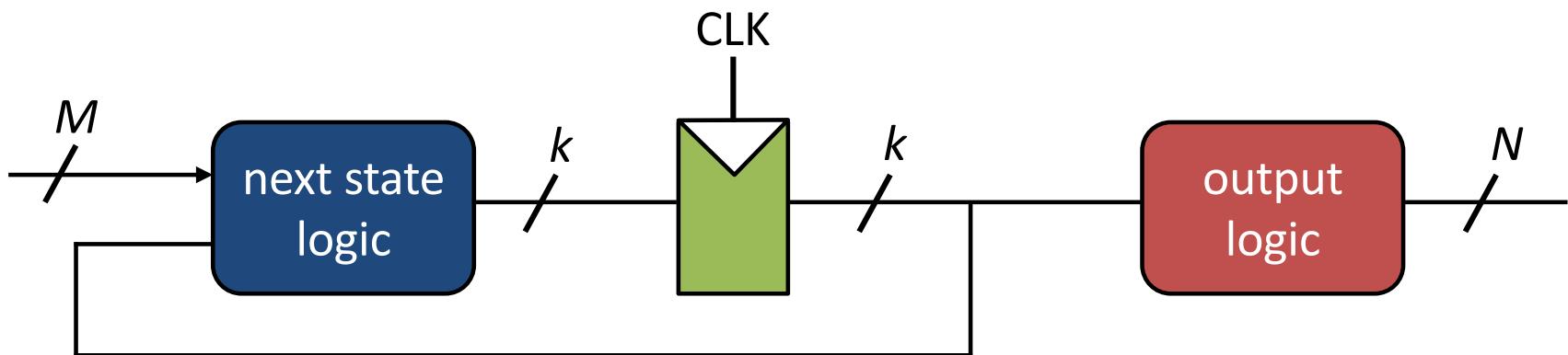
$$t_{pd} \quad T_C = 5.$$

- Suppose the next state logic has a delay of 5.5 seconds.
Will the traffic light controller work correctly?
- *What is the big lesson here regarding the minimum clock period (max. frequency) in synchronous sequential circuits?*



Quiz – 2

- What happens if we use a D latches to store the next state instead of D flipflops?
 - *Hint: The timing diagram can give insight*



ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoib Akram

shoib.akram@anu.edu.au



Australian
National
University

Plan: Week 4

Last week: Basics of sequential circuits

This Week: Finite state machines

This Week: Timing, parallelism & pipelining

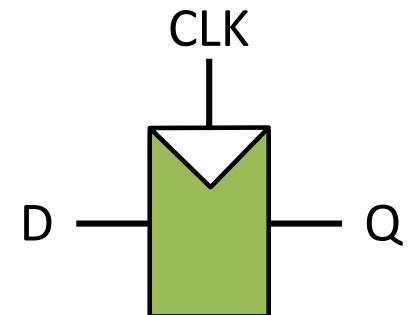
This Week: Gentle introduction to architecture

Two Sync. Sequential Circuits

- Two widely used synchronous sequential circuits
 - Finite state machine (FSM) ✓
 - Pipelines
- To understand pipelining, we need to first understand the timing specification of synchronous sequential circuits

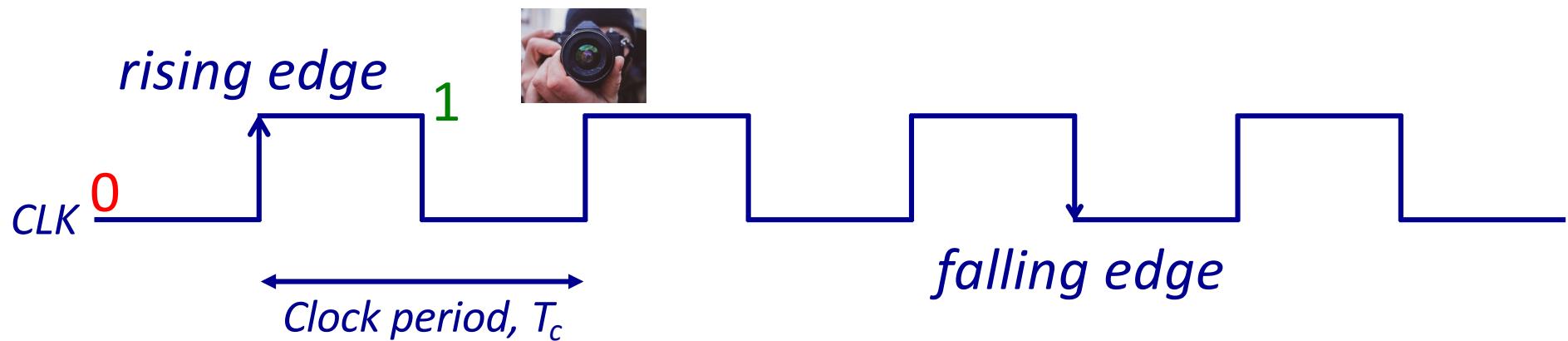
Timing in Sequential Circuits

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



Recall the Clock Waveform

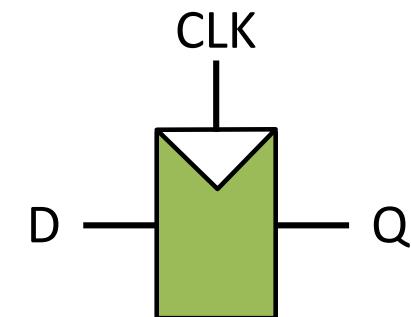
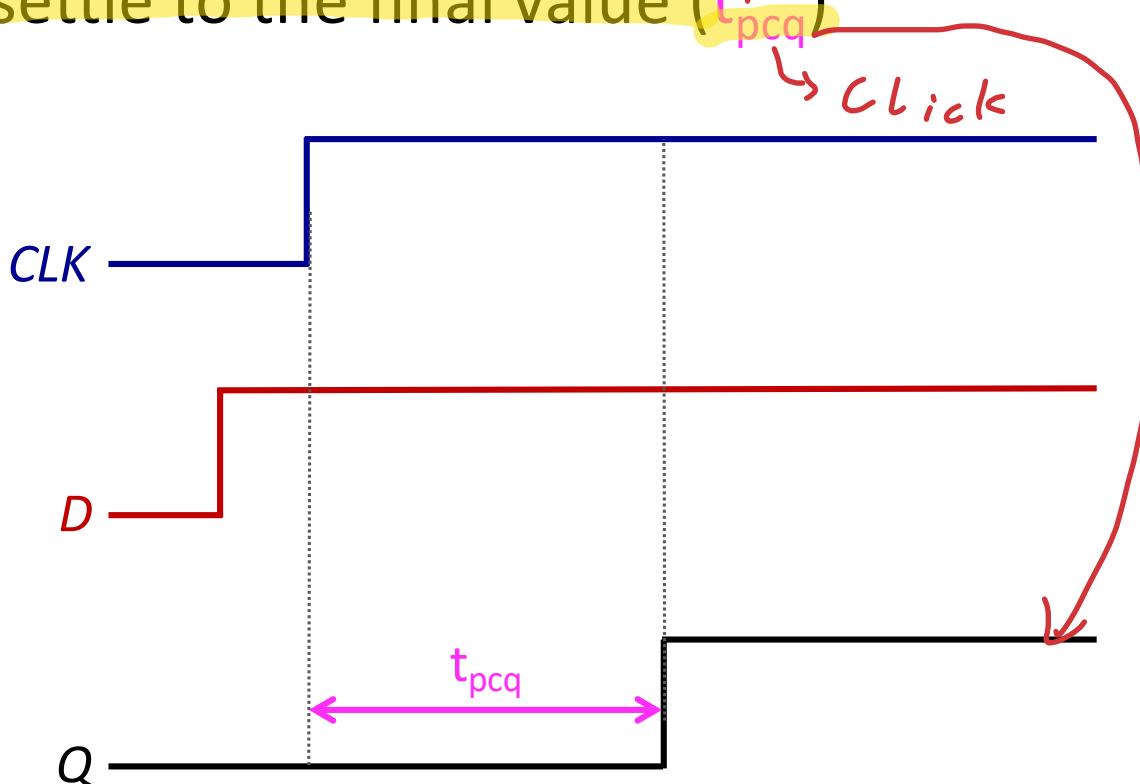
- Output does not change instantly when the rising edge arrives
- Input need to stay stable for some time period for the flipflop to take a reliable photograph



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

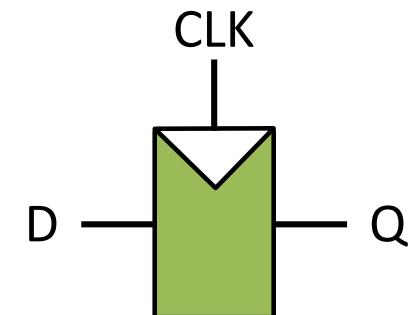
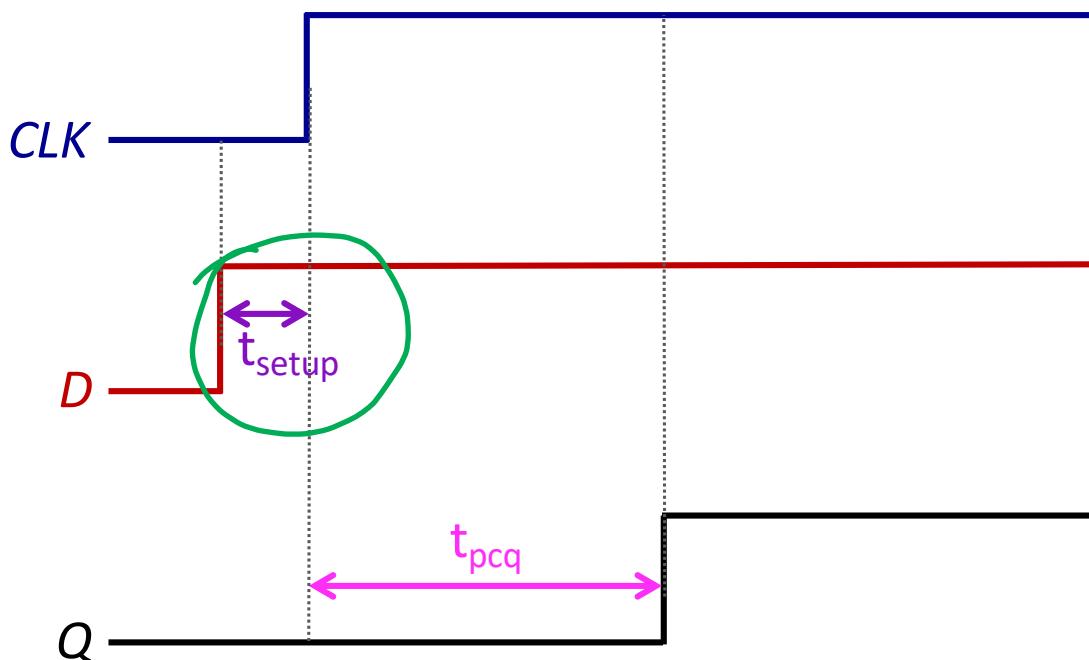
Clock-to-Q Propagation Delay

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



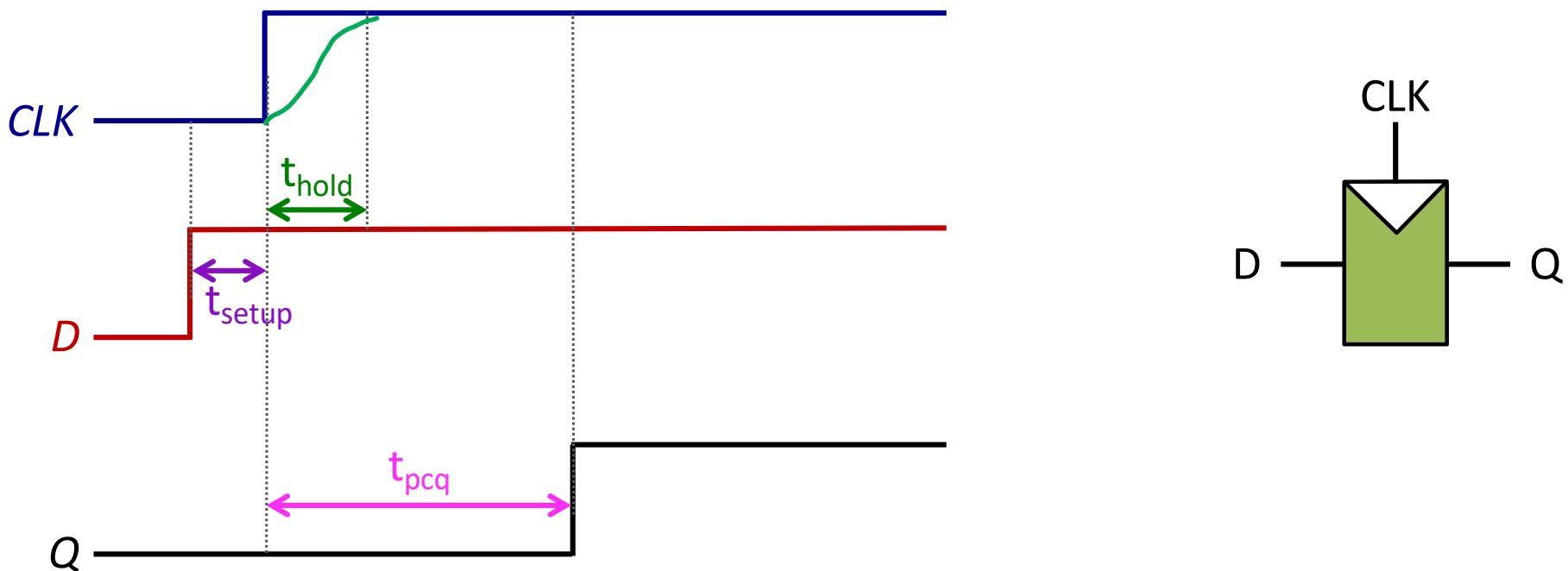
Setup Time

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



Hold Time

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



Aperture Time

The sum of the setup and hold times is called the aperture time of the circuit

- Total time for which the input must remain stable



Remark: Hold Time

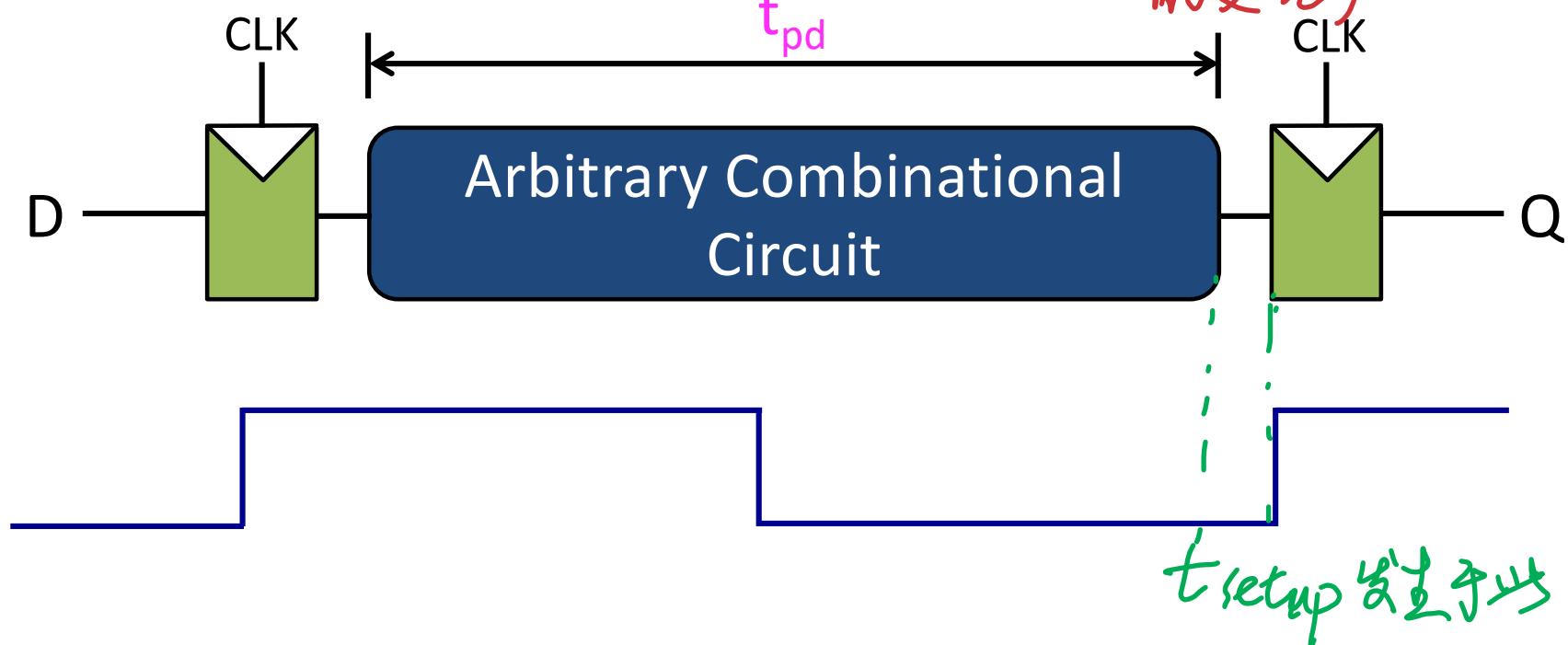
It is a reasonable assumption that modern flipflops have a hold time close to zero (*we can ignore hold time in subsequent discussions*)

Quiz

What is the clock period for the circuit below for it to work correctly?

1. t_{pd}
2. $t_{pd} + t_{pcq}$
3. $t_{pd} + t_{pcq} + \underline{t_{setup}}$

为 t_{pd} 时段之后的 t_{setup} . (计斧后结果
的变化)



Sequencing Overhead

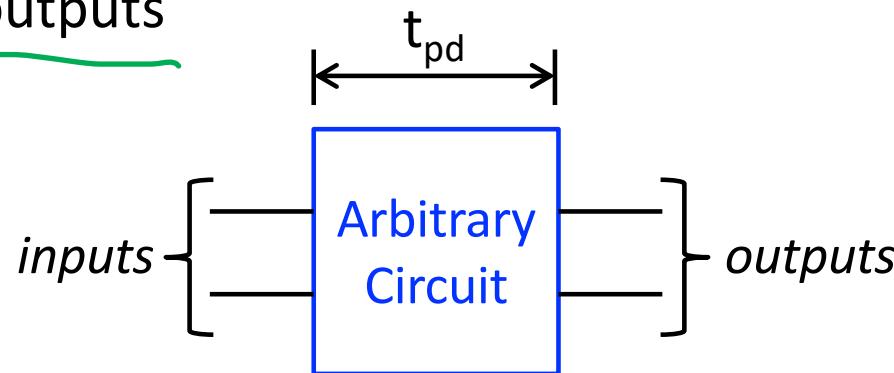
$t_{pcq} + t_{setup}$ is called the sequencing overhead of the flipflop

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

- Ideally the entire clock period should be spent on doing useful work in the combinational circuit
- The sequencing overhead of the flipflop cuts into this time

Speed of a Circuit

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



We need 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**): **20 min**
- Throughput (**trays/hour**): **3**



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 the key technique digital systems use to increase throughput

- Process several inputs at the same time
- Ideas?

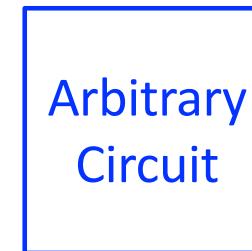
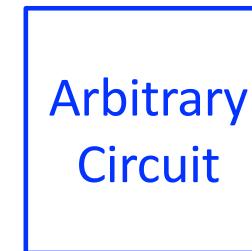
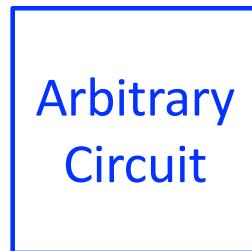
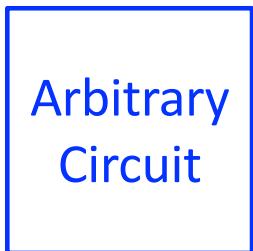
Defining a Task

Task: The process of producing a group of outputs from a group of inputs can be considered a task

- A circuit may need to perform several tasks
- A task can be as simple as adding two numbers
- More complex task is computing a Fourier transform

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

Note on Latency

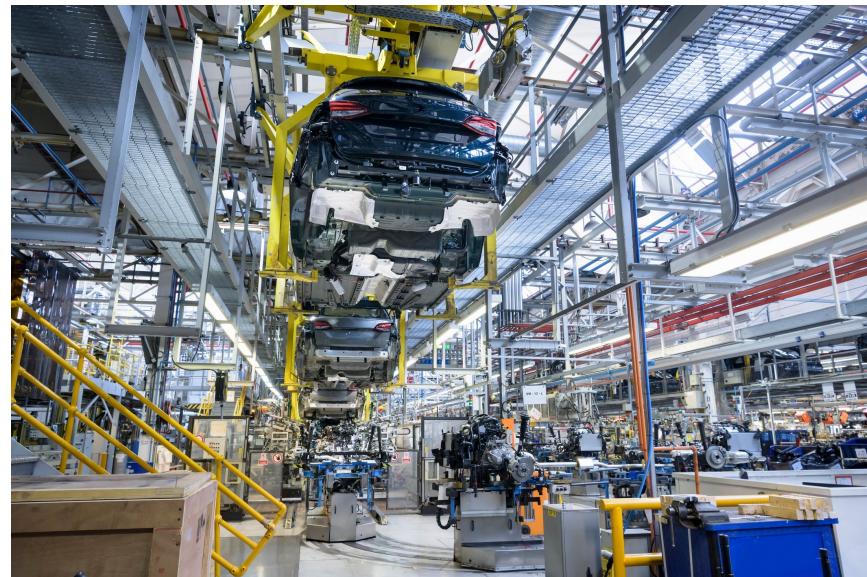
Spatial Parallelism does not improve (reduce) the latency of the circuit. We can finish more tasks per unit of time.
But each task still takes L seconds

Temporal Parallelism

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

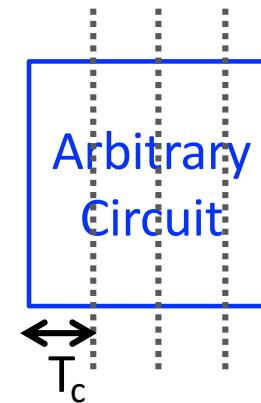
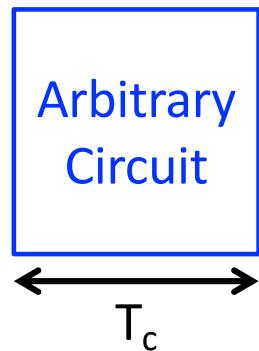
Analogy: Automotive pipeline

Work on multiple cars in parallel. Each car goes through all stages. Each stage requires different work. All stages should take roughly the same time for this to work



Pipelining

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



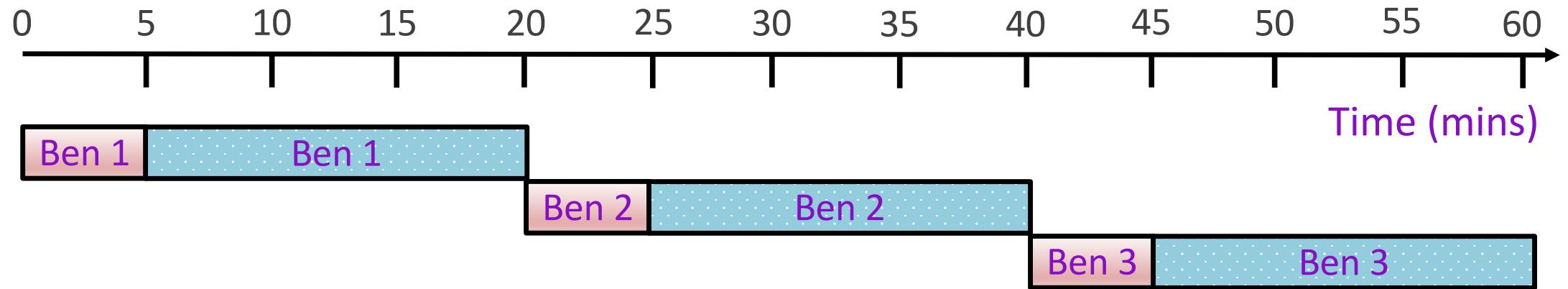
- *The challenge of pipelining is to find stages of equal length*
- *Let's go back to baking cookies*

Cookie Parallelism

Ben and Jon are making cookies. Let's study the latency and throughput of rolling/baking many cookie trays with

- No parallelism
- Spatial parallelism
- Pipelining
- Spatial parallelism + pipelining

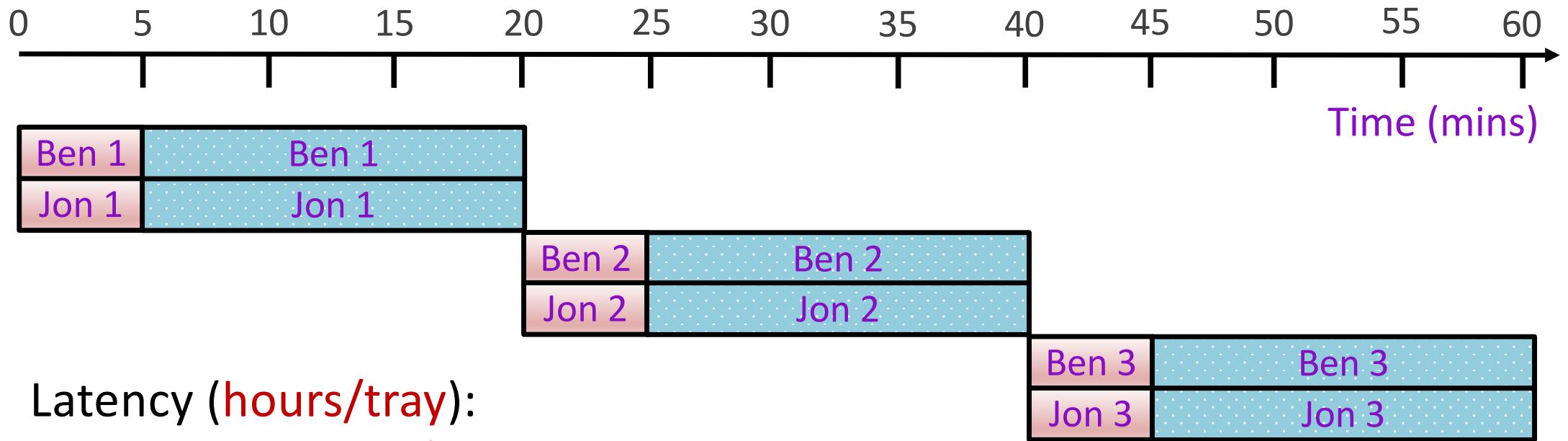
No Parallelism (Ben Only)



Latency (hours/tray):

Throughput (trays/hour):

Spatial Parallelism (Ben & Jon)

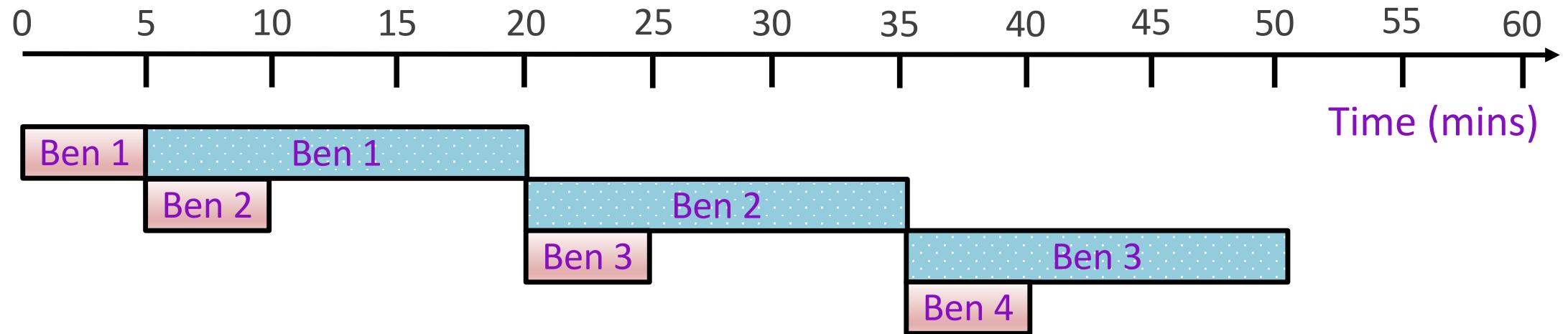


Latency (hours/tray):

Throughput (trays/hour):

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

Pipelining (Ben Only)

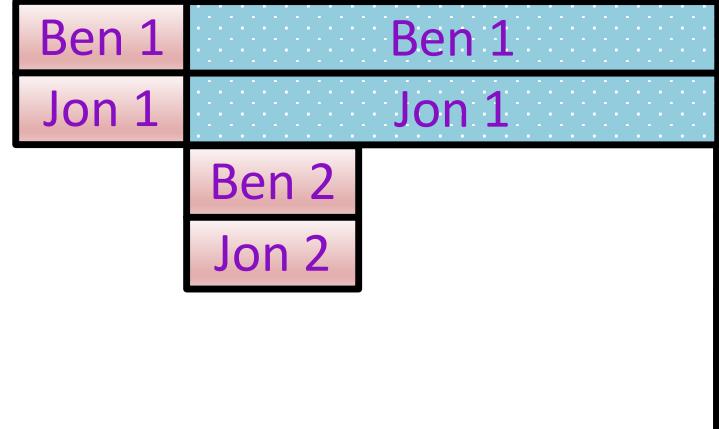
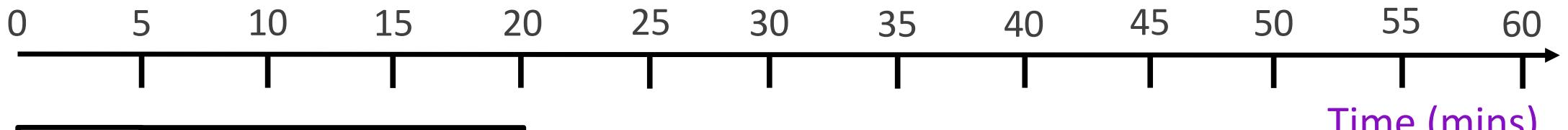


Latency (hours/tray): 20 min

Throughput (trays/hour): 4

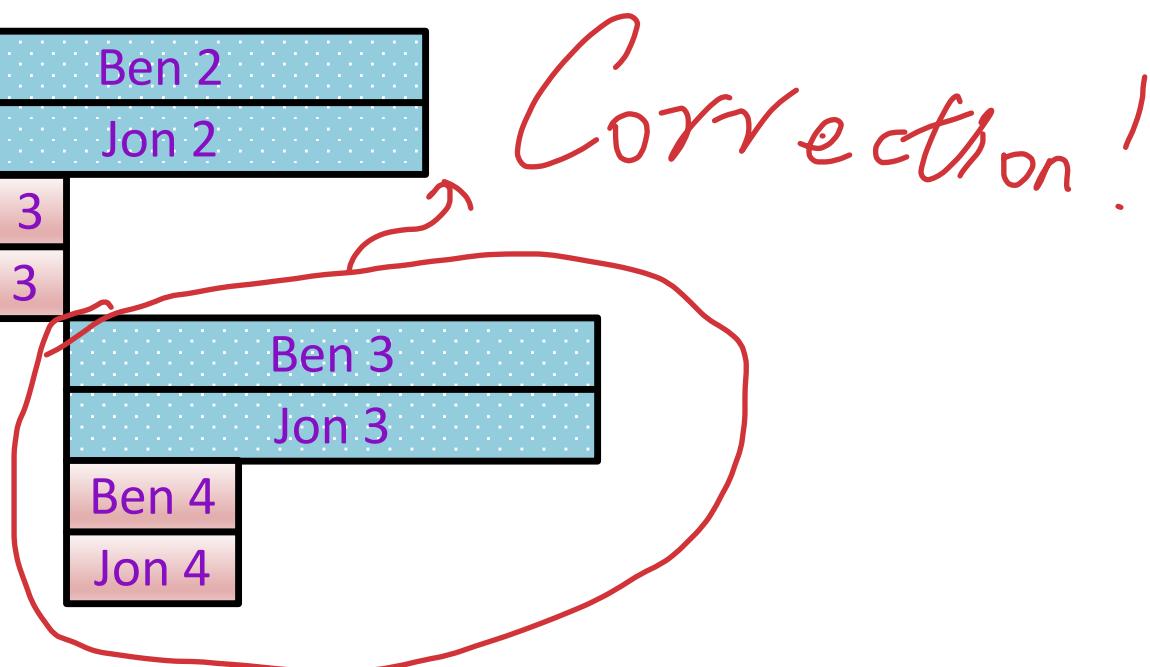
Note: Ben decides not to waste a separate tray and oven

Spatial + Temporal Parallelism



Latency (hours/tray):

Throughput (trays/hour):



Correction!

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

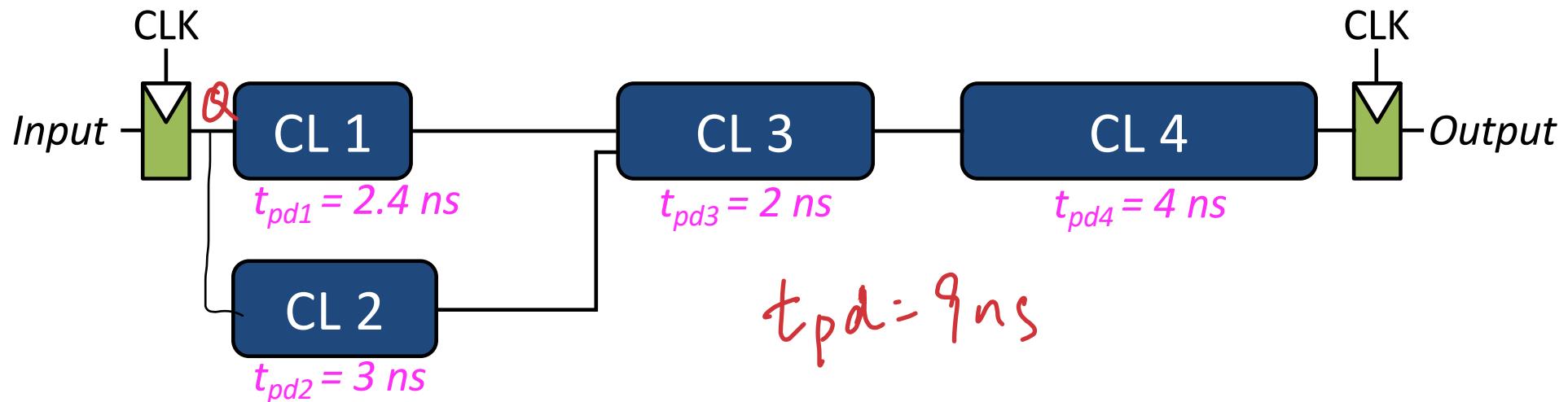
Pipelining Circuits

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

Exercise: Now with a circuit

Clock-to-Q propagation delay: 0.3 ns

Setup time : 0.2 ns

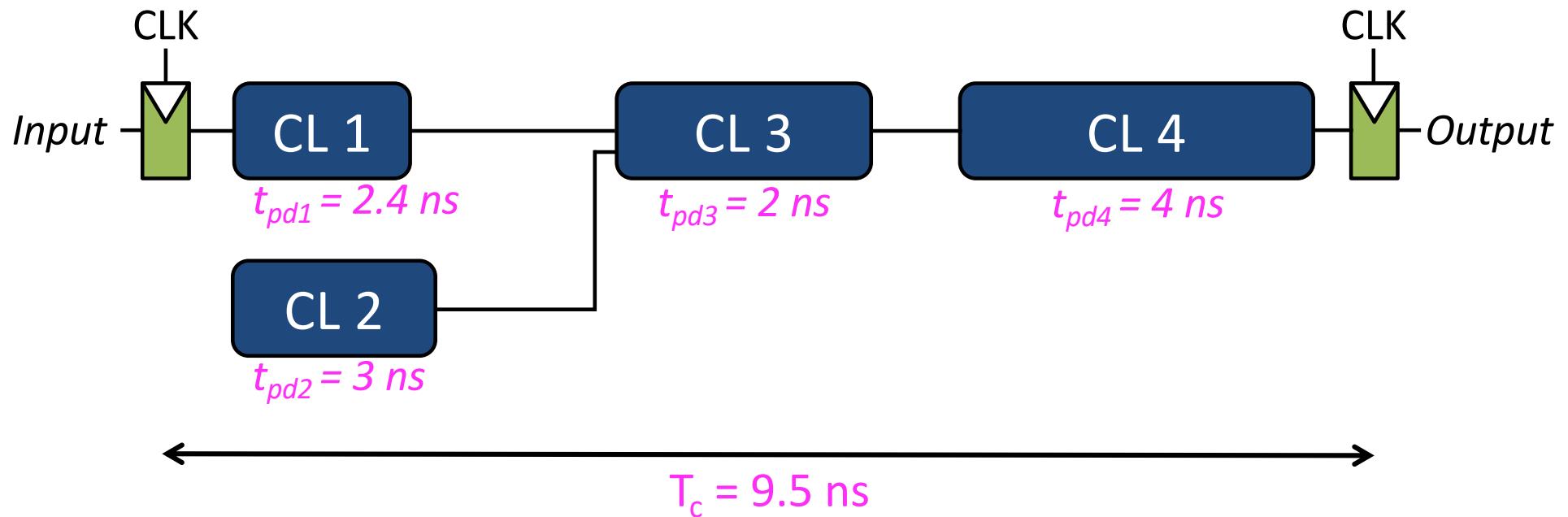


$$T_C = t_{pd} + \text{setup} + t_{pcq} = 9.5 \text{ ns}$$

Exercise: Now with a circuit

Clock-to-Q propagation delay: 0.3 ns

Setup time : 0.2 ns

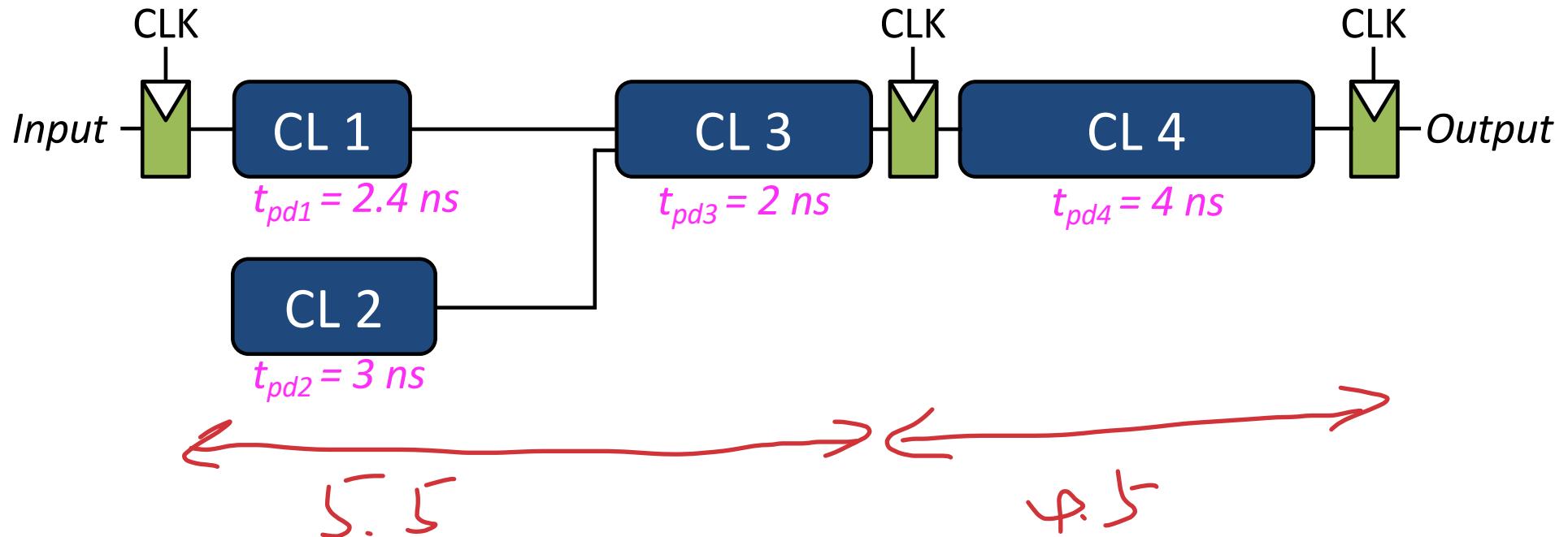


Latency = 9.5 ns

Frequency = $1/9.5 \text{ ns} = 105 \text{ MHz}$

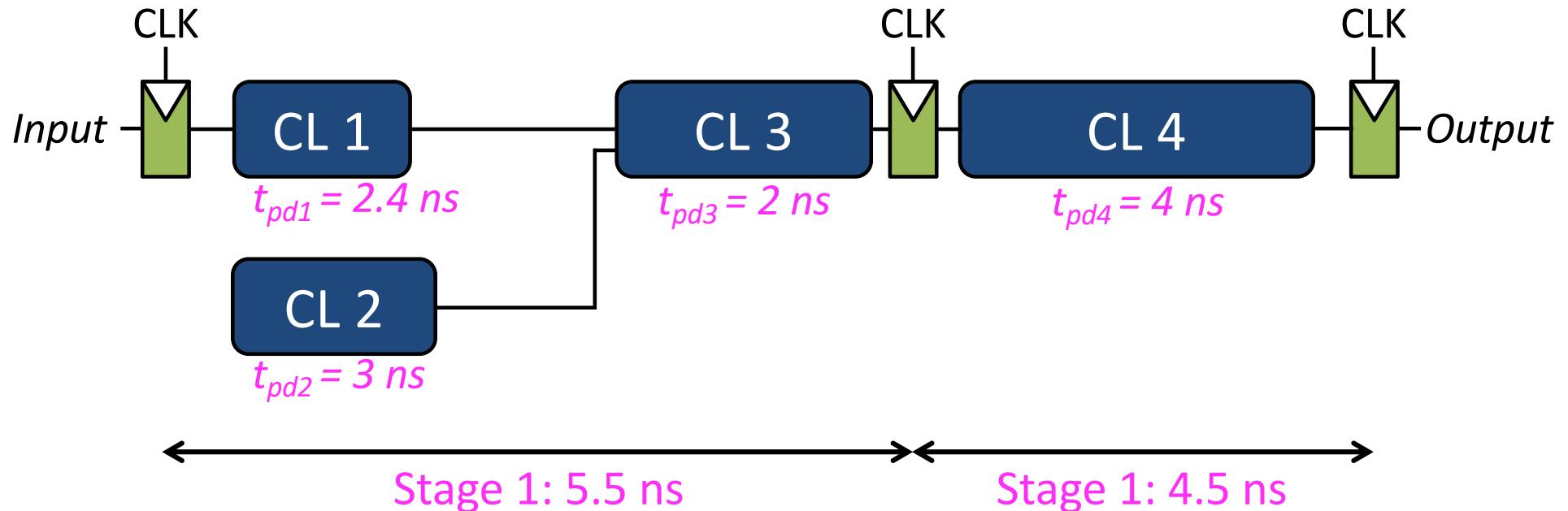
Exercise: 2-stage pipeline

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

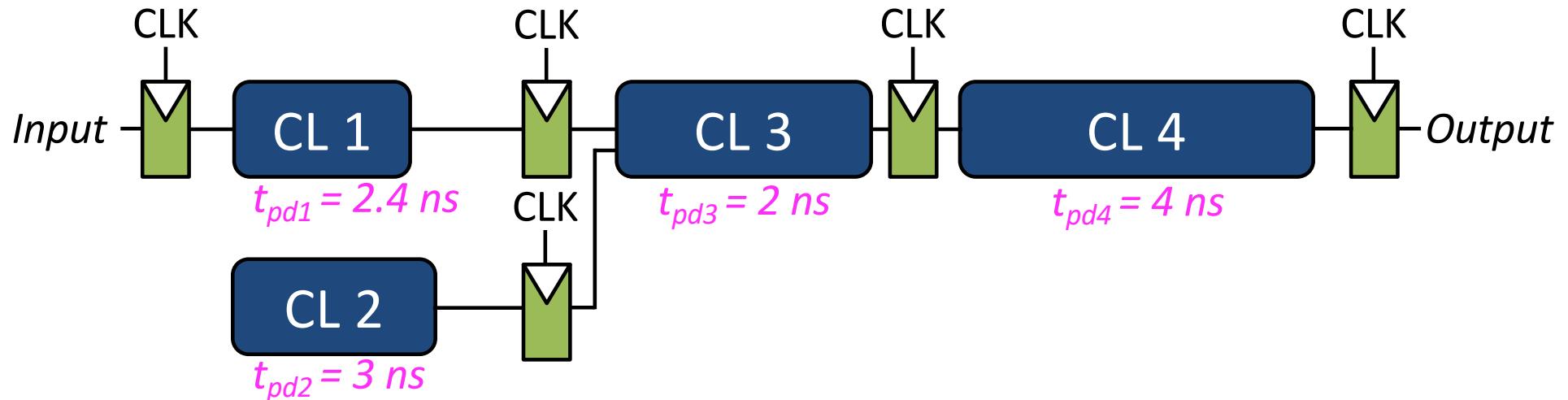


$$\text{Latency} = 2 \times 5.5 \text{ ns} = 11 \text{ ns}$$

$$\text{Frequency} = 1/5.5 \text{ ns} = 182 \text{ MHz}$$

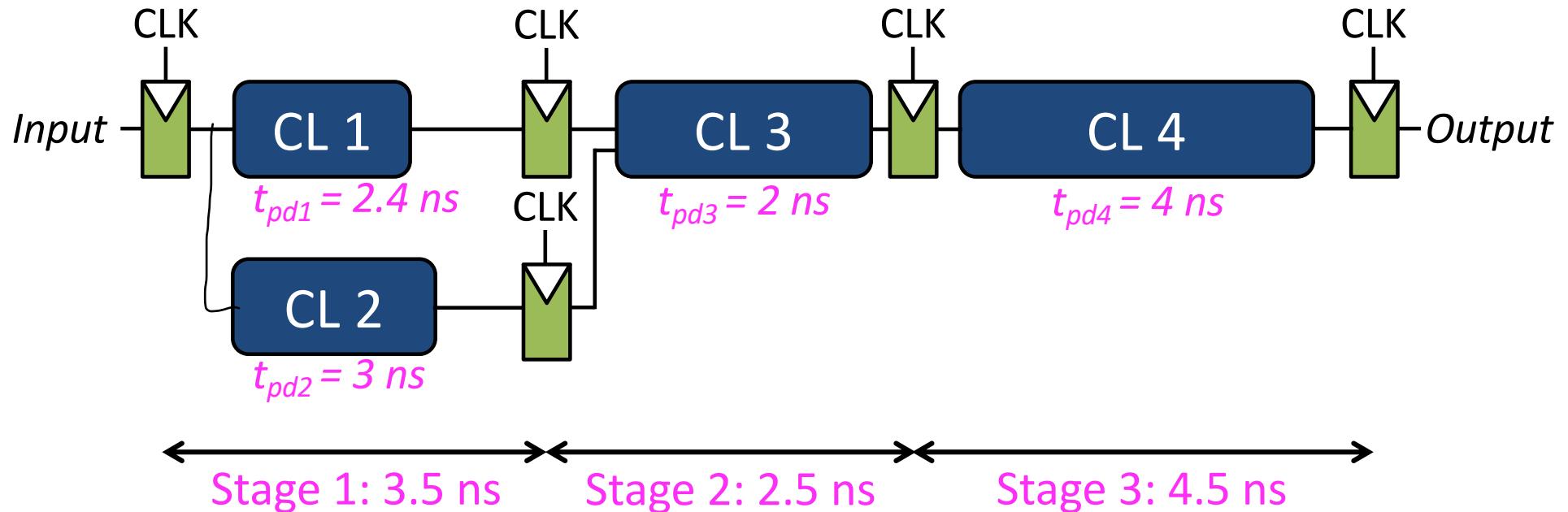
Exercise: 3-stage pipeline

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

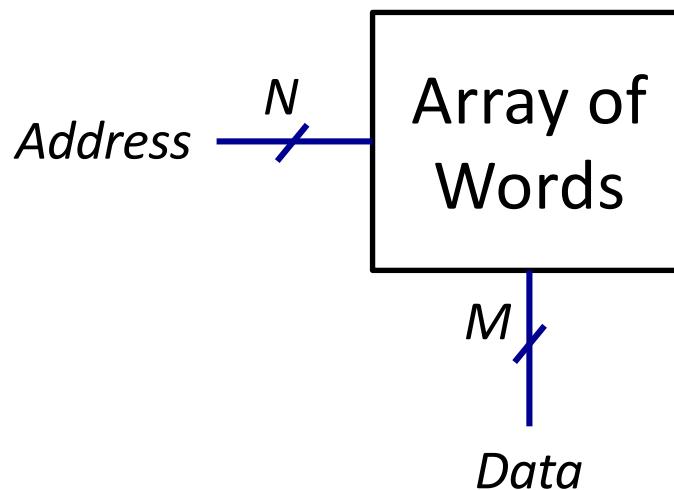


$$\text{Latency} = 3 \times 4.5\text{ ns} = 13.5\text{ ns}$$

$$\text{Frequency} = 1/4.5\text{ ns} = 222\text{ MHz}$$

Memory

- A two-dimensional array of memory cells
 - N-bit address, so 2^N rows
 - Each row is M-bit wide and contains one word of data
 - The array contains 2^N M-bit words



Address and Data

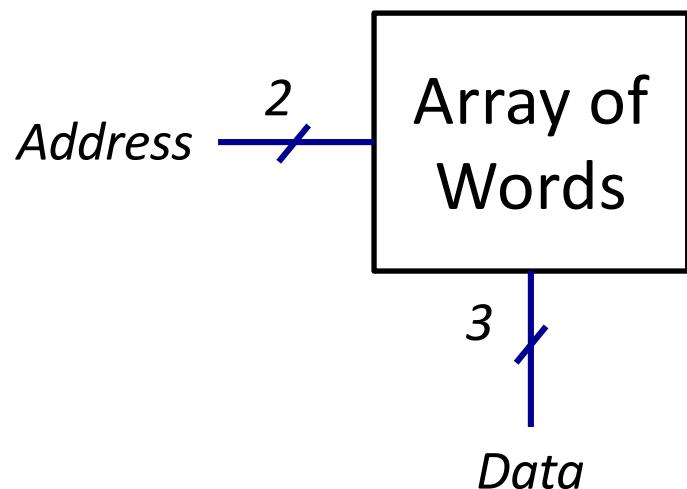
- Data is stored inside memory like the register file
- Address is presented to memory by an external circuit



- Each house is a memory cell (contains data)
- If we know the address, we can reach the house, but address has no physical existence

Example

- 2-bit address and 3-bit words



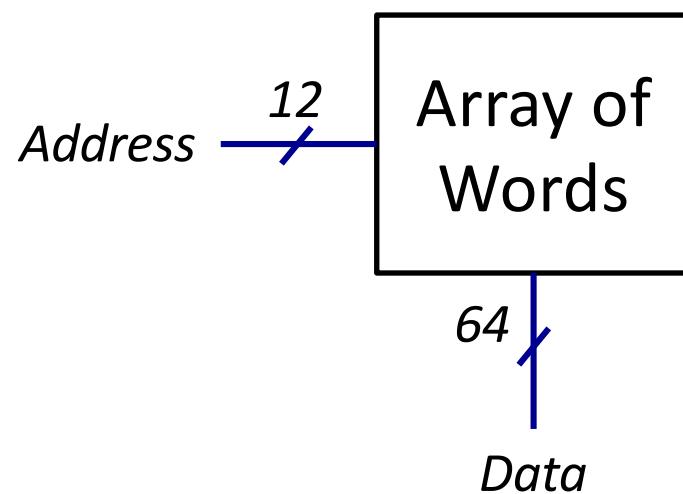
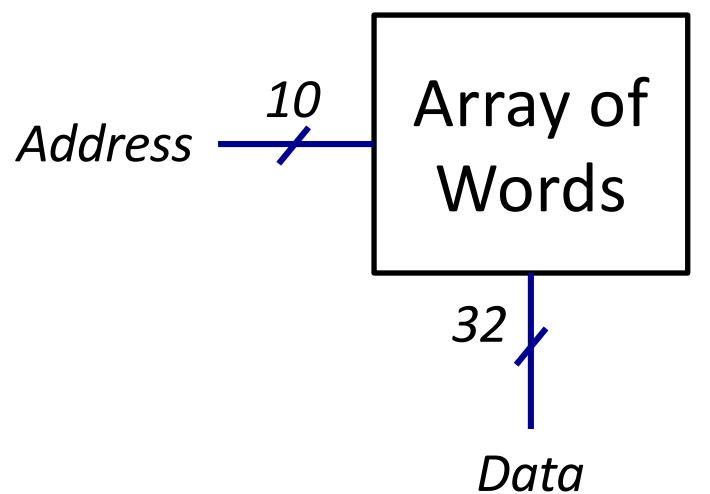
Address	Data		
11	0	1	1
10	1	1	0
01	1	1	1
00	1	0	0

A vertical double-headed arrow is positioned to the right of the table, indicating the width of the data bus. A horizontal double-headed arrow is located below the table, spanning the width of the data columns.

Example

- Size of memory (left) = $2^{10} = 1K$. $1K \times 32 \rightarrow 32K$ bits = $4KB$
- Size of memory (right) =

$$K \text{ bytes} = \frac{K \text{ bits}}{8}$$

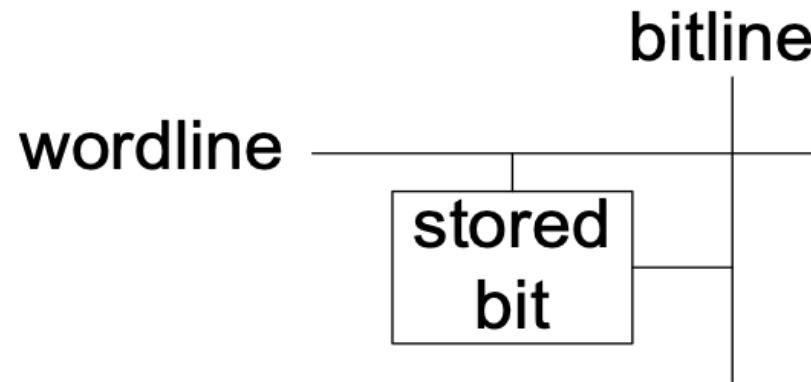


Read/Write Access

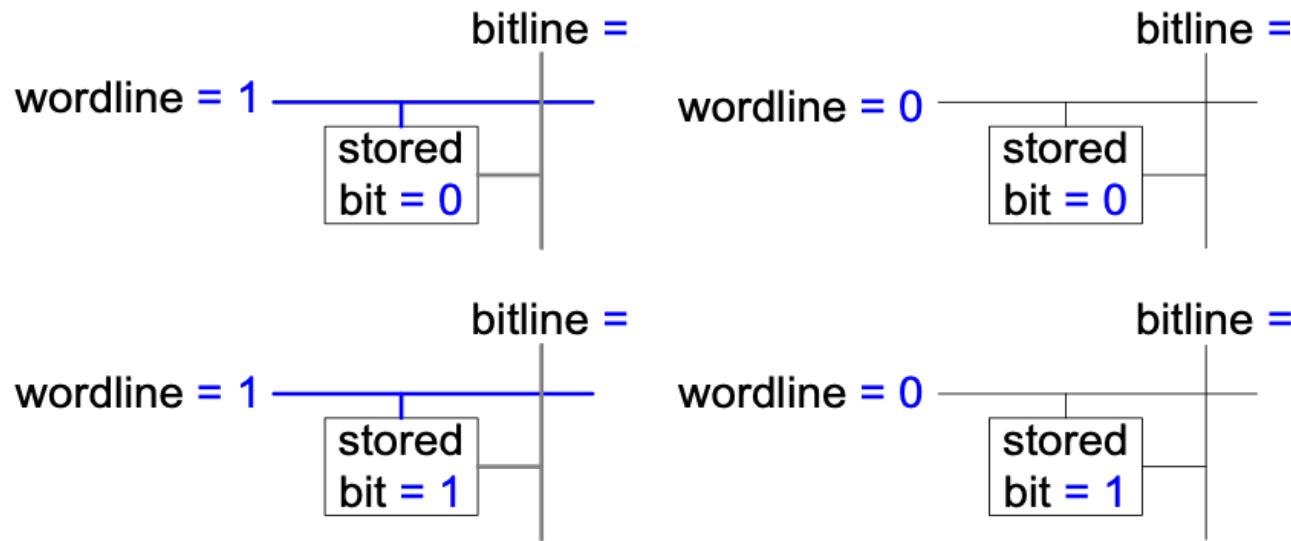
- Looking up a word stored at an address in memory is called a *read access* or simply *read*
- Updating a word stored at an address in memory is called a *write access* or simply *write*
- Typically, a read is called a *memory load* or simply *load* when the word is read from memory into register file
- Similarly, write is called a *memory store* or simply *store*

Memory Cell

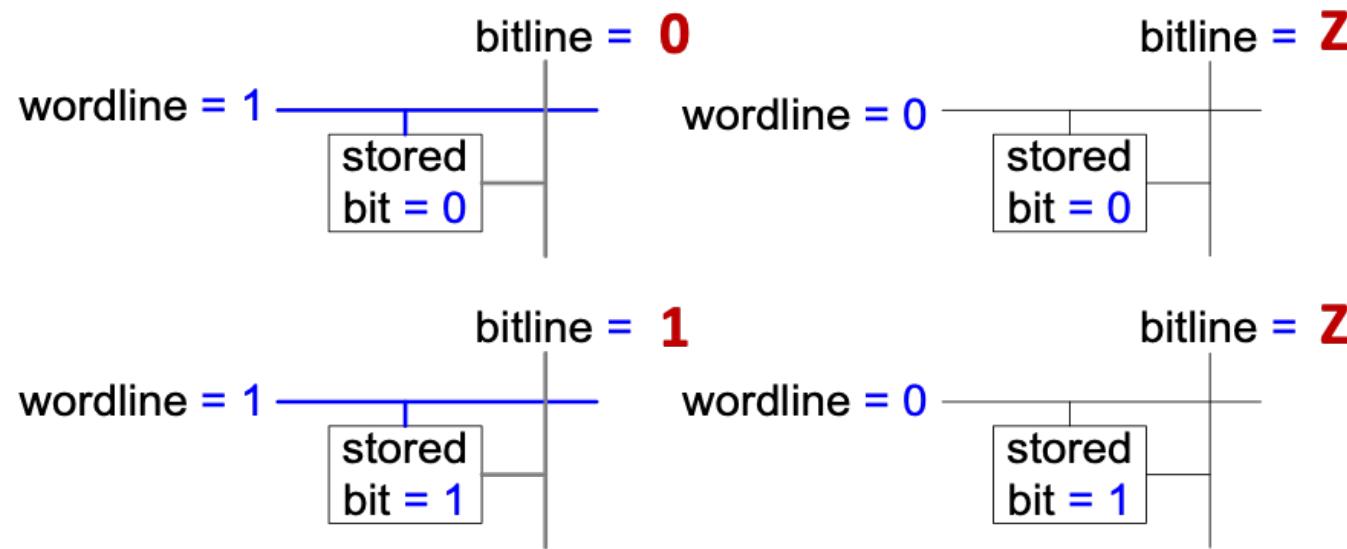
- We call it a bit cell (more technical term)
- A bit cell is connected to a bitline and a wordline
- Each bit cell contains one bit of data
- When the wordline is **HIGH**, the stored bit transfers to or from the bitline



Example: Bit Cell Operation



Example: Bit Cell Operation



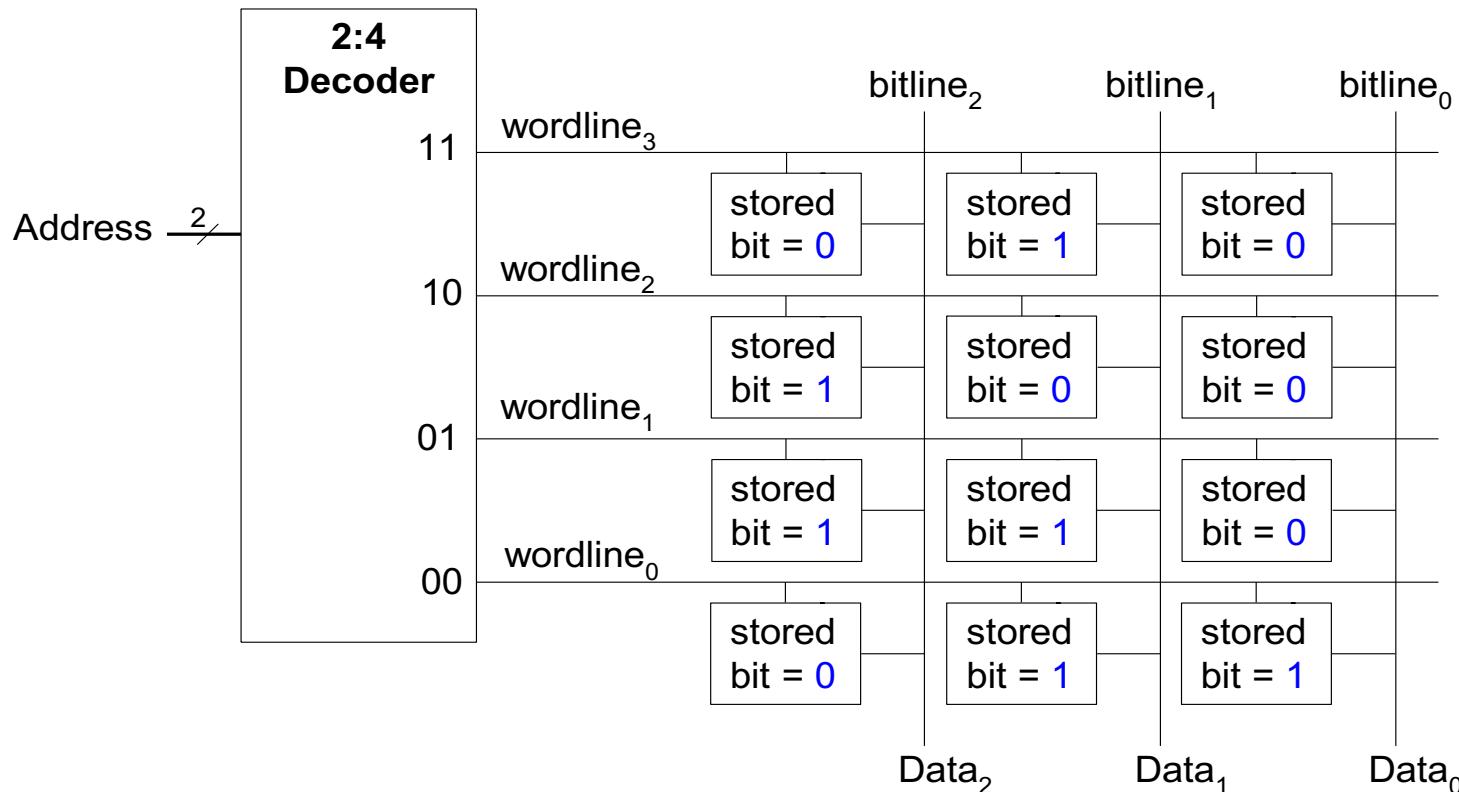
- Z is neither 0 nor 1 in digital electronics
- The wire is cut-off from the circuit (in this case the bit cell)

Reading and Writing Bit Cell

- Read
 - A special circuit is used to bring the bitline in Z state
 - The wordline is then set to HIGH
 - The stored value in the bit cell then drives the bitline
- Write
 - The bitline is driven (set) to 0 or 1
 - The wordline is turned on, connecting the bitline to the stored bit
 - The contents of the bit cell change to 0 or 1

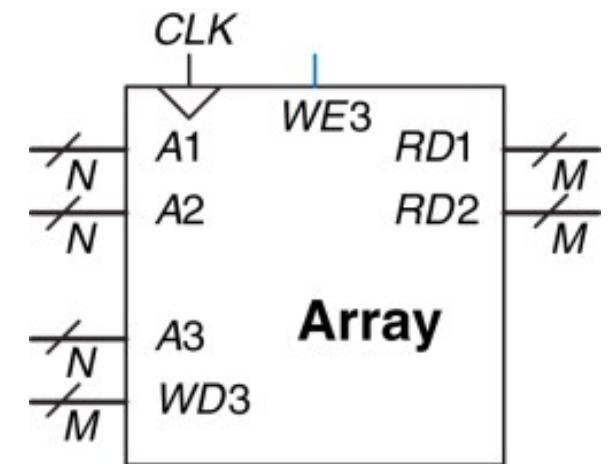
Memory Array Organization

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



Memory Ports

- Each memory port gives read and/or write access to one memory address
- Multiported memories can access several address 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 HIGH



Memory Specification

- Size
 - Width
 - Depth
- Ports
 - How many?
 - Type

Random Access Memory

- Random access memory or RAM is a type of memory for which accessing any data word results in the same delay as any other data word
- The main memory in a typical computer (e.g., your laptops) is RAM
- RAM is volatile
 - If the power is removed from the computer, the data in RAM is no longer there

RAM vs. Storage



- Hard disk (left) and tape (right)
 - Sequential access is faster than random access
 - Mechanical movement is required to access data
 - Non-volatile or persistent storage

Memory Classification

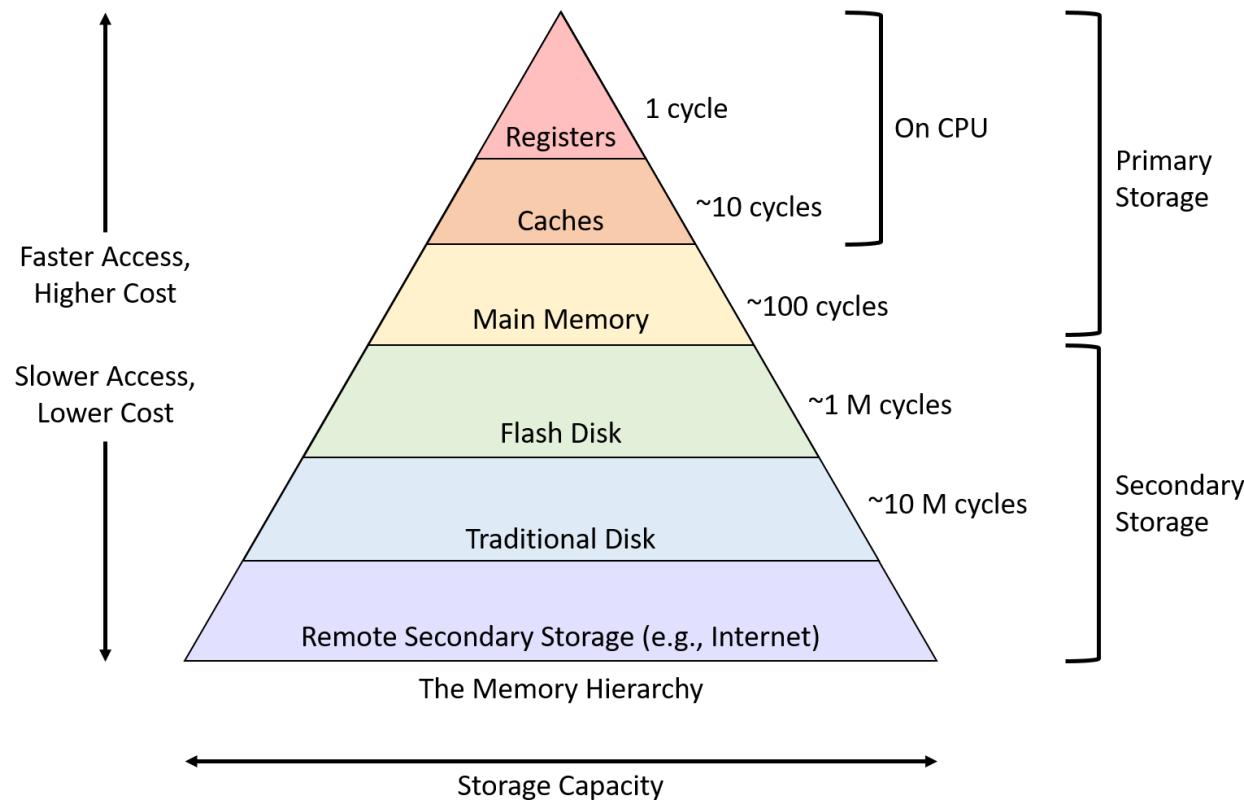
- RAM is classified according to the formation of the bit cells
 - **Static RAM** stores data bits using a pair of cross-coupled inverters
 - **Dynamic RAM** stores data bits using the presence or absence of charge on a capacitor (will return to this after the teaching break)

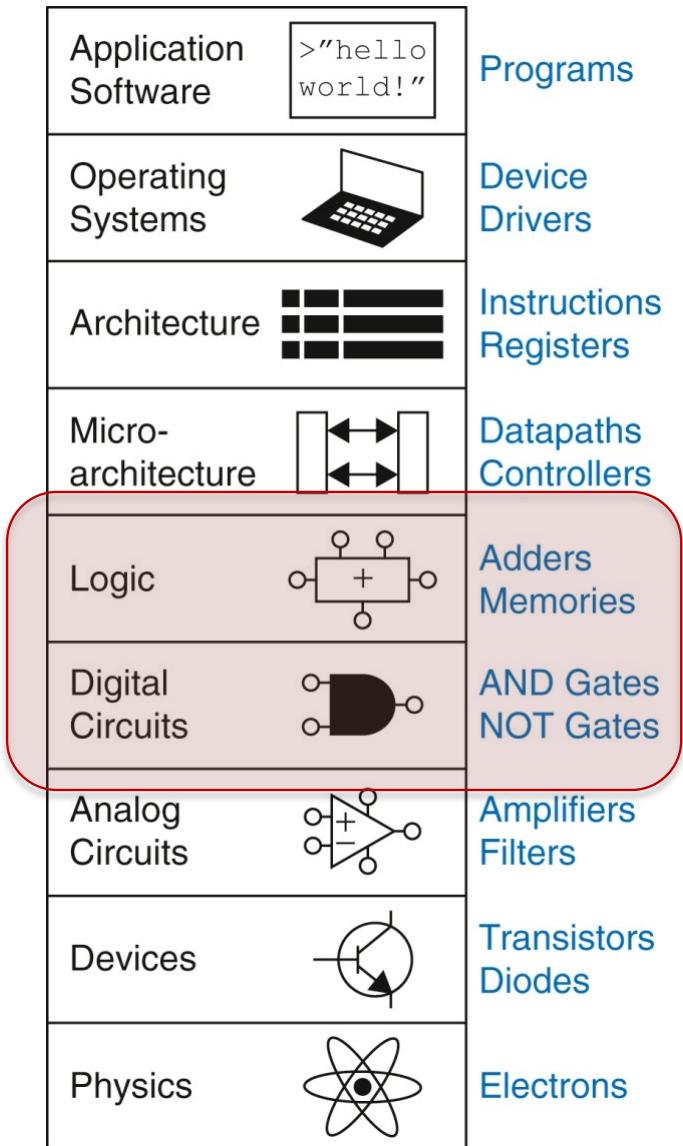
SRAM vs DRAM

- SRAM is fast and expensive
- Typically, the memory close to the CPU uses the SRAM technology
 - Register file
 - Cache (after the teaching break)
- The memory far from the processor uses the DRAM technology
 - Your computer's main memory is DRAM

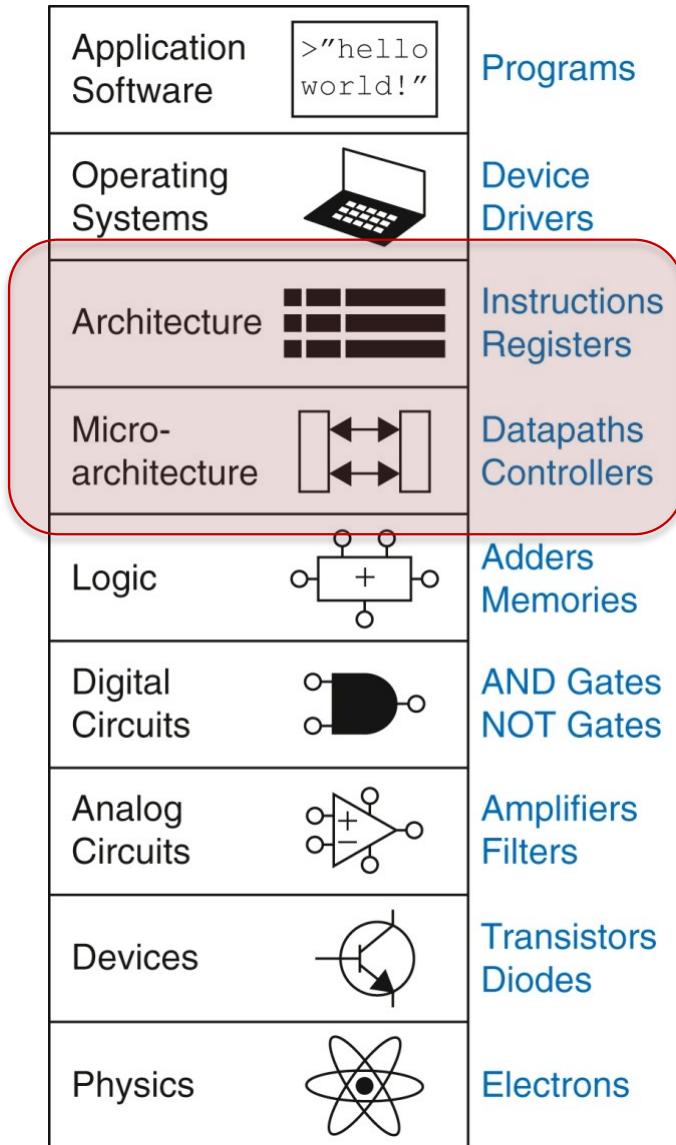
Memory Hierarchy

- We will return to this stuff after the teaching break





We are here



Moving up a few abstraction layers?

Week 5 : Architecture
 Week 6: Microarchitecture

The Architecture Layer

- Our goal in the first half of the course is to understand and build a processor
- We can specify the combinational circuits and the traffic light controller in English
- How should we write the specification of a processor?
 - We need a systematic approach to manage the complexity of a processor
- The formal specification of a processor (and computer) takes place at the “architecture” abstraction layer

Architecture/ISA

- The architecture or Instruction Set Architecture (**ISA**) is the programmer's view of the computer
- ISA specifies the set of instructions a computer can perform (think of it like the language of a computer)
 - Instruction = word
 - ISA = vocabulary
- Each instruction specifies
 - Operation (What exactly to do?)
 - Operands (Where to find the data to operate upon?)
- Example: **Add** two 16-bit binary numbers in registers **R1** and **R2** (recall the register file from Lab 3 – 4)

Operands

- Operands can be in register and memory
 - Note: The register file alone cannot house all the data programs need
 - Register file is fast and expensive (and hence small)
- If operands can be in memory, then we must have instructions to fetch the operands from memory
 - Load
 - Store
- Is there a third possibility for operand location?
 - From the instruction itself (keep this in mind)

Assembly Language

- Instructions written in a symbolic format so humans can read/understand them easily is called assembly language
 - ADD, SUB, LDR, STR, MUL, ROR, MOV, BIC
 - We will study all of the above ARM instructions
- A sequence of instructions is called assembly code

*Remark: Don't worry
about what this means!
Just want to show how
assembly looks*

SUB	R0,	R1,	R2
ADD	R8,	R4,	R5
ADD	R9,	R6,	R7
SUB	R3,	R8,	R9

Machine Language

- Instructions are encoded as 1's and 0's in a format called the machine language
 - ADD can be represented by binary code 0000
 - SUB can have a possible encoding of 0001
 - Register R1: 00 (example)
 - Register R2: 01

0	0	0	0	1	0	0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0	1	1	1	0	0	0

Hypothetical machine code (above) is a sequence of machine language instructions stored in memory

Instruction Format

- An instruction consists of several fields
- Each field has a different meaning
- An instruction format specifies the meaning of each field
- An ISA can have many instruction formats

- **Remark:** The ISA we use in labs ([QuAC](#)) and lectures ([ARM v4](#)) are fixed-width ISA. Each instruction format in the ISA uses a fixed number of bits (16 or 32)
- **Remark:** A popular ISA ([x86](#)) has variable-sized instructions (keep it in mind when you build the CPU and think of the difficulty of implementing such as ISA)

Microarchitecture

- An ISA is a specification
- Microarchitecture is the implementation of an ISA
 - The specific arrangement of registers, memories, ALUs, and other building blocks to form a processor is called microarchitecture
- The same ISA can have many different implementations
 - Tradeoffs in **performance**, **power**, **price**
 - A company **X** builds two processors for a high-end laptop and a cheap cell phone, respectively, that can both run programs targeting the same ISA named **Y**

More Examples

- Intel and AMD build processors targeting the x86 ISA
- QuAC ISA is for teaching purposes
 - Each group/student will build a CPU differently
 - One group may use two adders rather than one; different styles for register file
 - Both are building a processor for the same ISA
 - Their microarchitectures are different

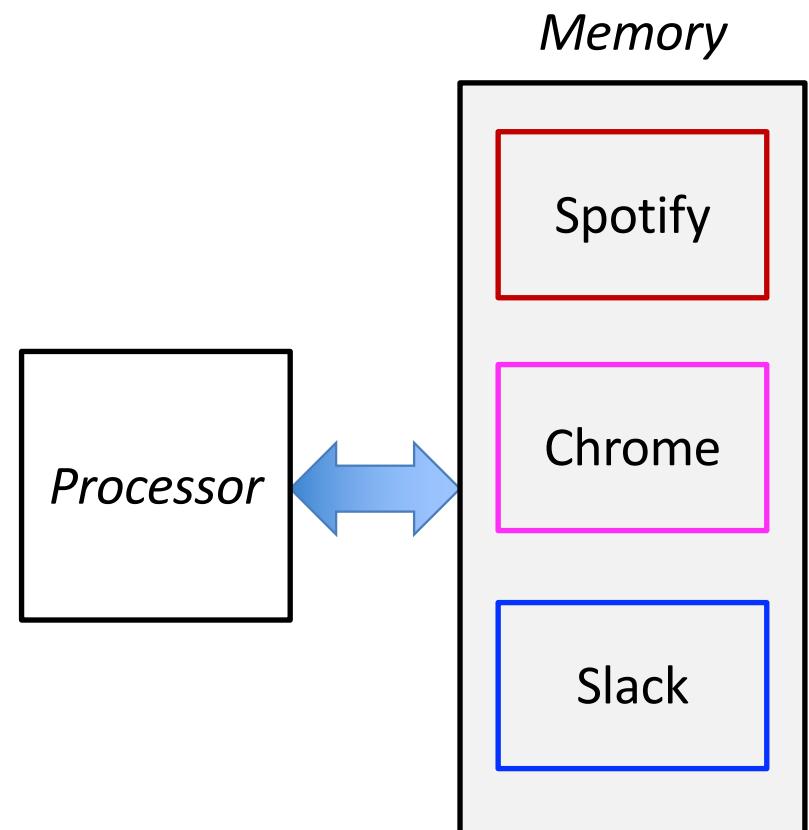


Remarks

- All programs running on a computer use the same instruction set
- All software applications, such as Spotify and Word, are eventually *compiled* into a series of simple instructions
- **Compiler:** A program that transforms a program written in a high-level language (C, C++, Python) to assembly instructions
- **Assembler:** A program that transforms assembly code to machine code

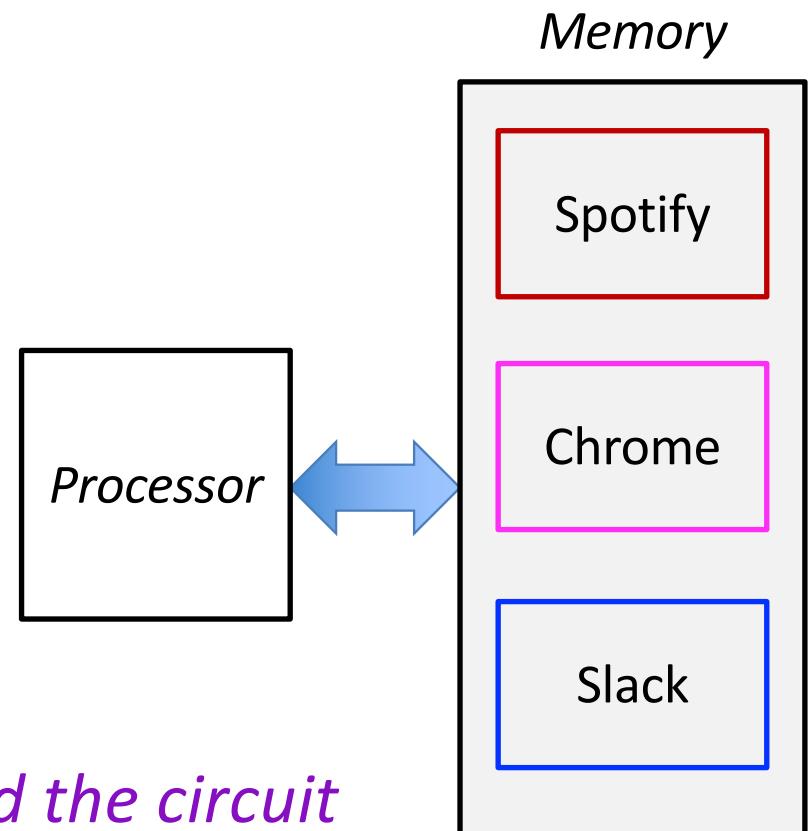
Stored Program Concept

- Two key principles
 - Instructions are represented as binary numbers
 - Programs (*in the form of machine code*) are stored in memory (*like data*)



Stored Program Concept

- How do processors execute machine code?
 - Fetch an instruction from memory
 - Decode the meaning of the instruction
 - Execute the instruction
 - Repeat until done



Your job in the labs/assignment is to build the circuit for fetching, decoding, and executing instructions

Popular ISAs

- Intel x86
 - High-performance desktop/laptop/server
 - Power-hungry (Apple's recent shift to Apple silicon)
- ARM
 - Popular in the mobile/embedded domain
 - Lecture/textbook focus
 - M1 (Apple) uses ARM architecture
- RISC-V
 - A new open-source ISA gaining momentum
- QuAC
 - Teaching purposes (invented at ANU)

Quiz

- Ben has an excellent idea to make computers more efficient. To try out his idea he first picks a popular existing ISA.
 - *What is the advantage of Ben's approach to pick an existing ISA instead of developing a new one?*
 - *What is the drawback of picking an existing ISA?*

Quiz

- Ben has an excellent idea to make computers more efficient. To try out his idea he first picks a popular existing ISA.
 - *What is the advantage of Ben's approach to pick an existing ISA instead of developing a new one?*
 - *Binary compatibility enables existing programs to make use of Ben's excellent idea without any effort*
 - *Historically, this aspect has led to ISA hegemony, where one popular ISA is dominant*
 - *What is the drawback of picking an existing ISA?*
 - *Think!*

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoib Akram

shoib.akram@anu.edu.au



Australian
National
University

Plan: Week 5

Last week: Finite State Machines, Timing, Pipelining

This Week: Finish looking at memories

This Week: Instruction Set Architecture (ISA)

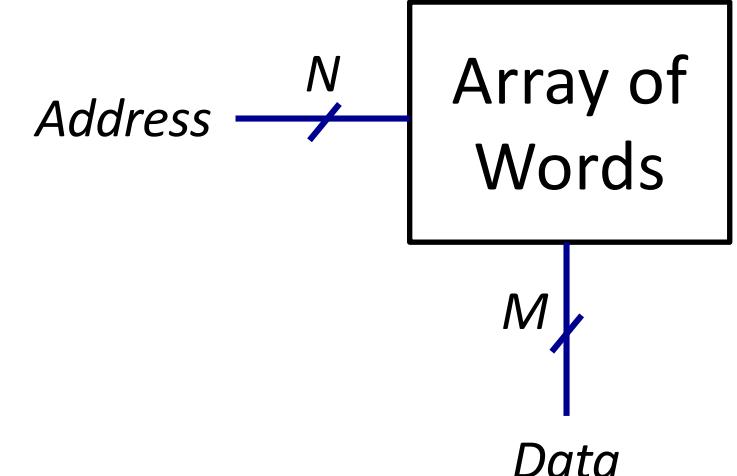
Memory

- A two-dimensional array of memory cells
 - Each cell contains one bit
 - A group of cells make up a row (word)
 - There are many rows, each with a unique address
- Memory dimensions
 - Each row is M-bits wide (Data)
 - The address is N bits: 2^N rows
 - The array contains 2^N M-bit words

Address

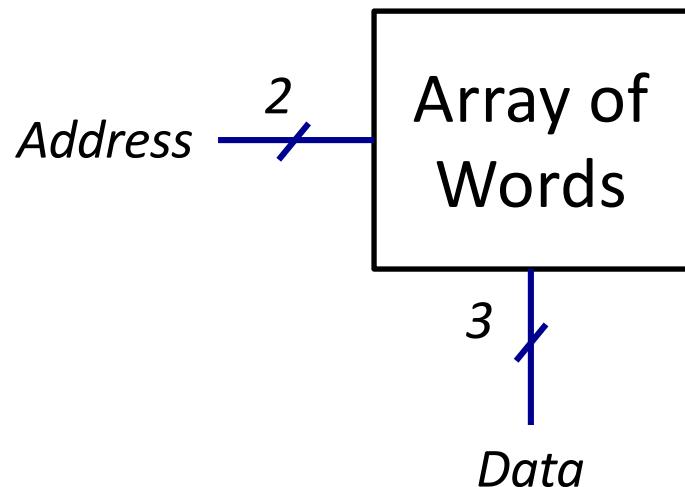
0
1
2
3

0	1	1	0
1	0	1	0
1	0	0	1
0	1	0	1



Example

- 2-bit address and 3-bit words



Address	Data		
11	0	1	1
10	1	1	0
01	1	1	1
00	1	0	0

The table has 4 rows and 4 columns. The first column is labeled 'Address' with values 11, 10, 01, 00. The other three columns are labeled 'Data' with values 011, 110, 111, 100 respectively. A vertical double-headed arrow on the right is labeled 'height'. A horizontal double-headed arrow at the bottom is labeled 'width'.

Address and Data

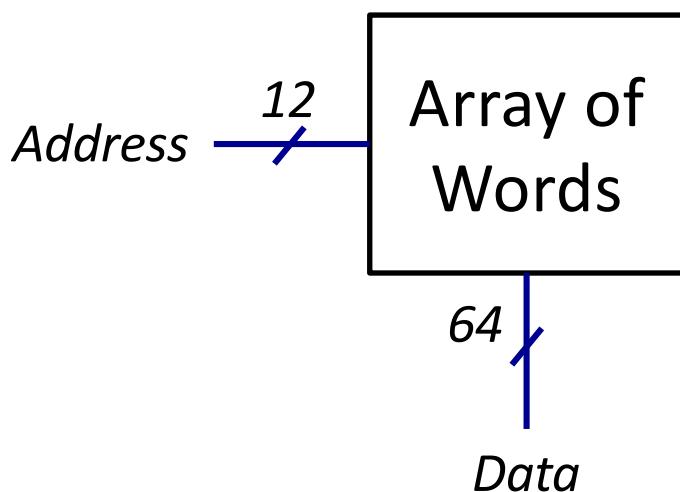
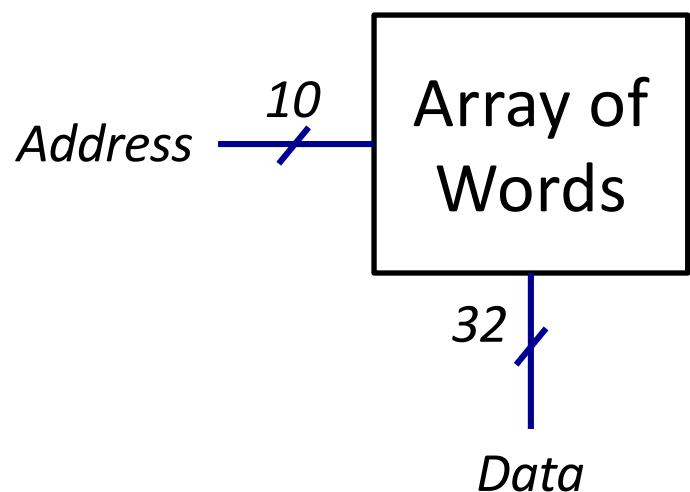
- Data is stored inside memory like the register file
- Address is presented to memory by an external circuit



- Each house is a memory cell (contains data/bit)
- Can reach house if address is known (address is not stored)
- Individual cells (bits) are not addressable
- Typically, a group of 8 bits (byte) has a unique address

Example

- Size of memory (left) =
- Size of memory (right) =

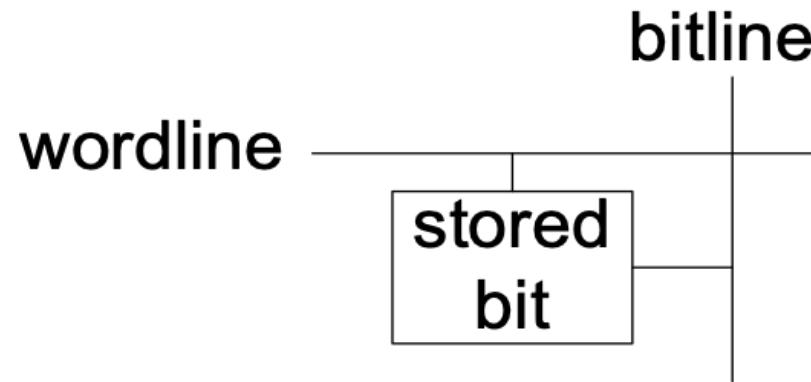


Read/Write Access

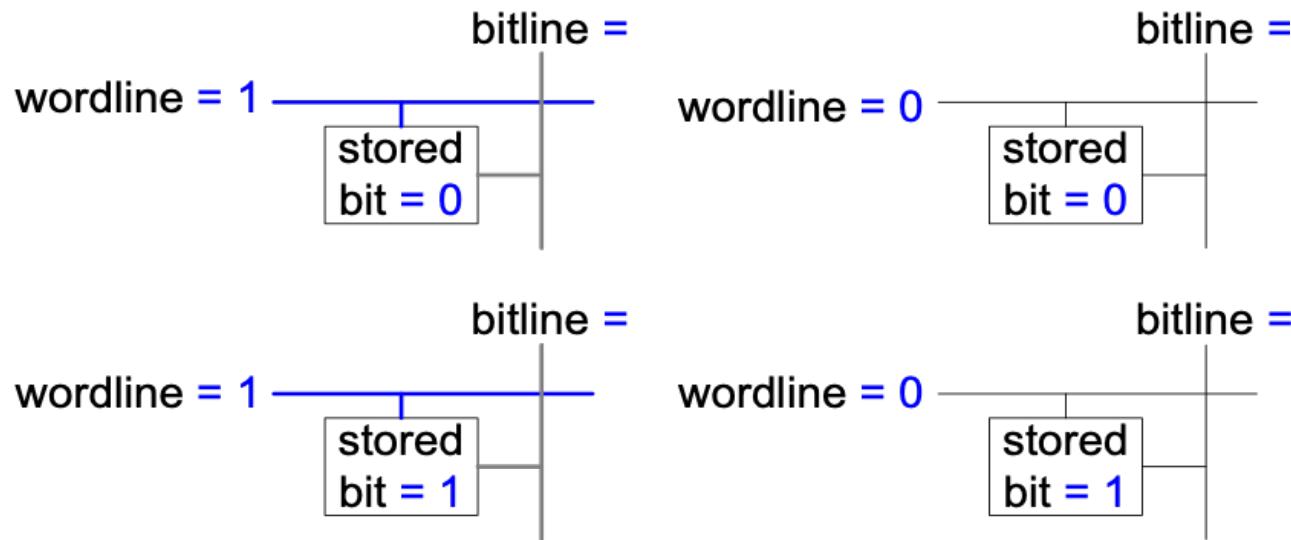
- Looking up a word stored at an address in memory is called a *read access* or simply *read*
- Updating a word stored at an address in memory is called a *write access* or simply *write*
- Typically, a read is called a *memory load* or simply *load* when the word is read from memory into register file
- Similarly, write is called a *memory store* or simply *store*

Memory Cell

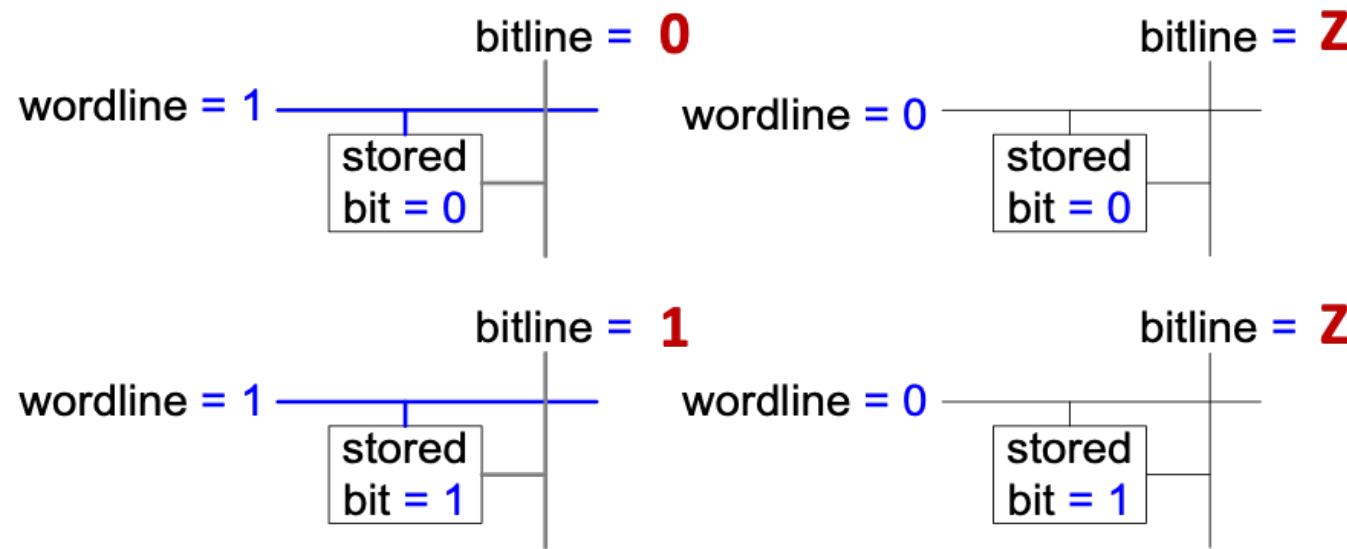
- We call it a bit cell (more technical term)
- A bit cell is connected to a bitline and a wordline
- Each bit cell contains one bit of data
- When the wordline is **HIGH**, the stored bit transfers to or from the bitline



Example: Bit Cell Operation



Example: Bit Cell Operation



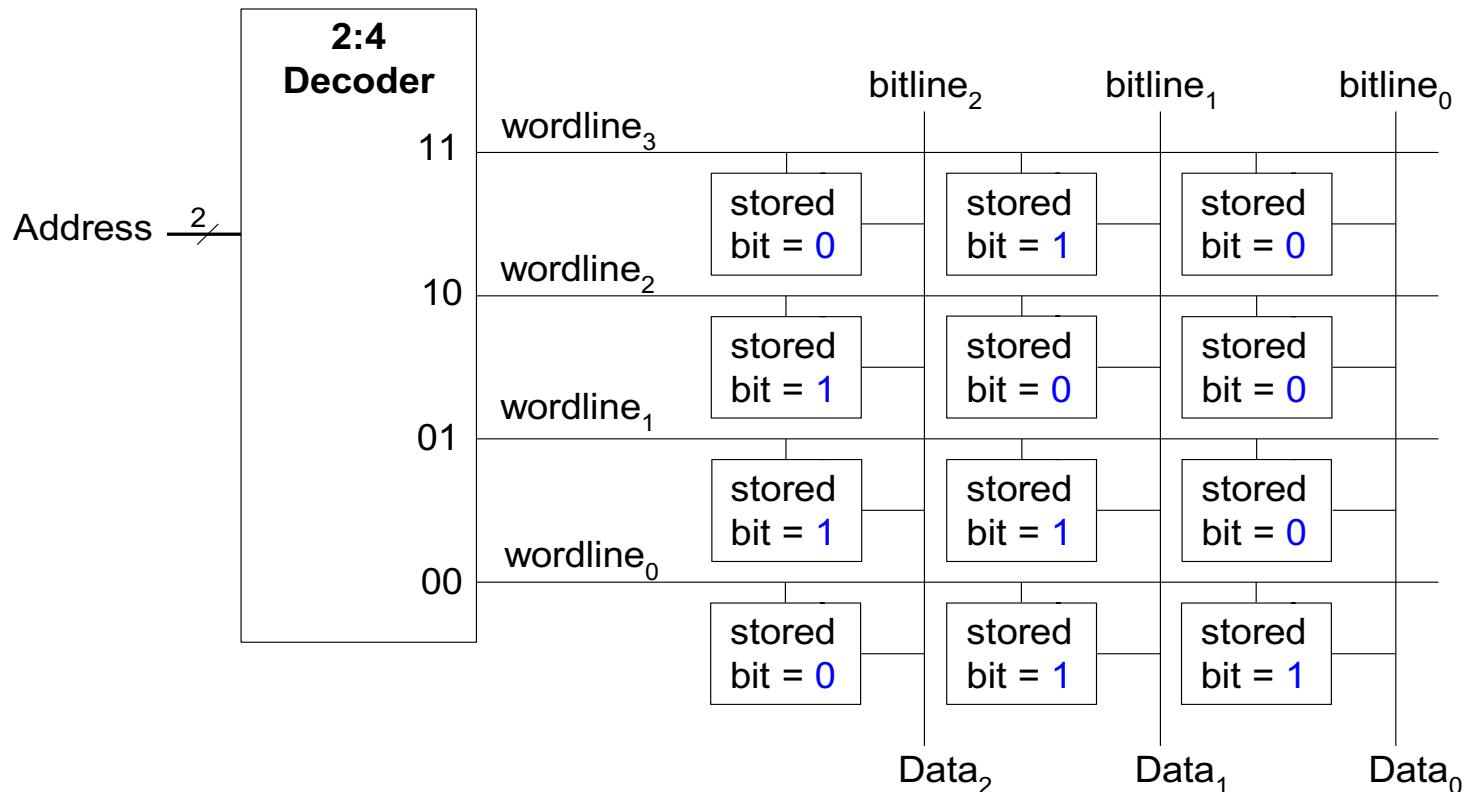
- Z is neither 0 nor 1 in digital electronics
- The wire is cut-off from the circuit (in this case the bit cell)

Reading and Writing Bit Cell

- Read
 - A special circuit is used to bring the bitline in Z state
 - The wordline is then set to HIGH
 - The stored value in the bit cell then drives the bitline
- Write
 - The bitline is driven (set) to 0 or 1
 - The wordline is turned on, connecting the bitline to the stored bit
 - The contents of the bit cell change to 0 or 1

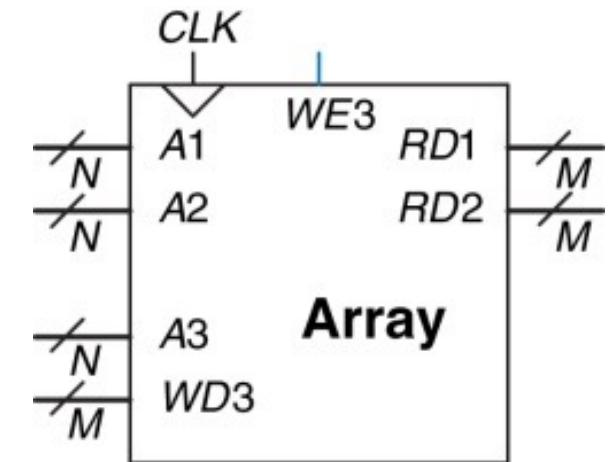
Memory Array Organization

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



Memory Ports

- Each memory port gives read and/or write access to one memory address
- Multiported memories can access several address 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 HIGH



Memory Specification

- Size
 - Width
 - Depth
- Ports
 - How many?
 - Type

Random Access Memory

- Random access memory or RAM is a type of memory for which accessing any data word results in the same delay as any other data word
- The main memory in a typical computer (e.g., your laptops) is RAM
- RAM is volatile
 - If the power is removed from the computer, the data in RAM is no longer there

RAM vs. Storage



- Hard disk (left) and tape (right)
 - Sequential access is faster than random access
 - Mechanical movement is required to access data
 - Non-volatile or persistent storage

Memory Classification

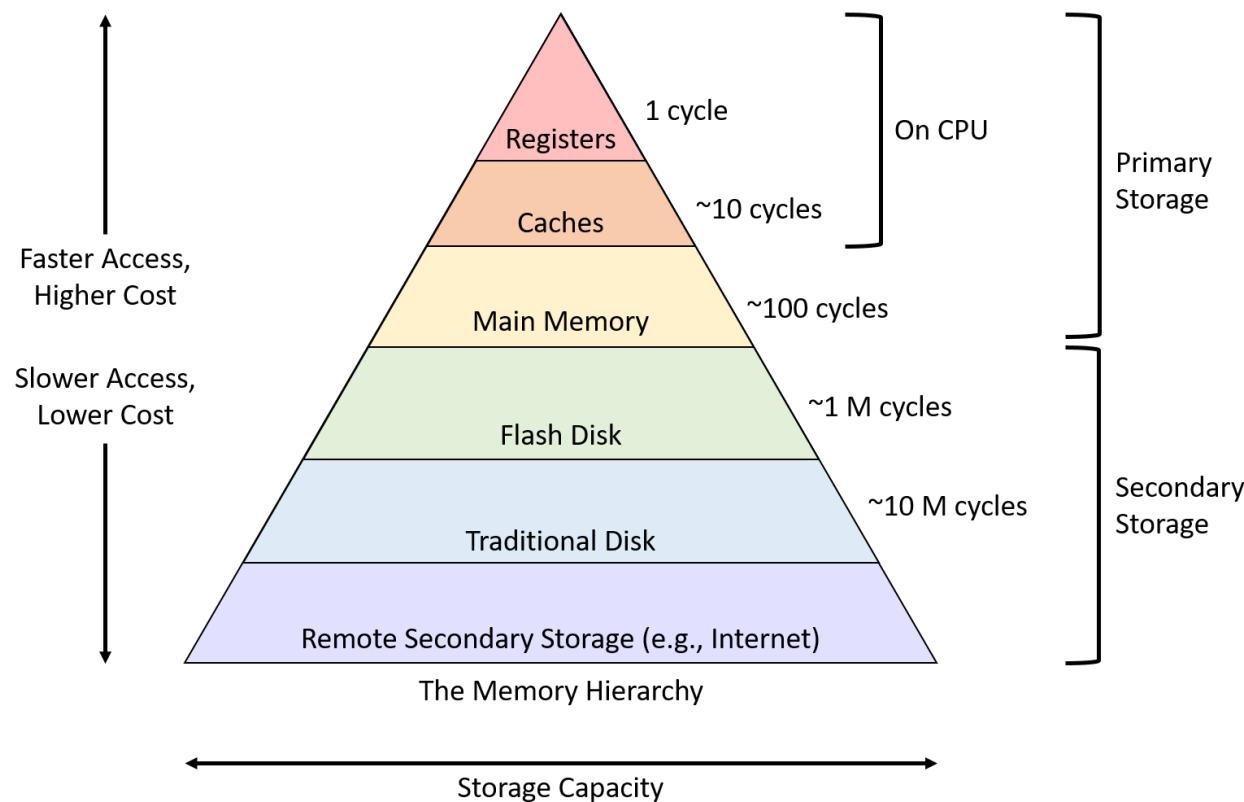
- RAM is classified according to the formation of the bit cells
 - **Static RAM** stores data bits using a pair of cross-coupled inverters
 - **Dynamic RAM** stores data bits using the presence or absence of charge on a capacitor

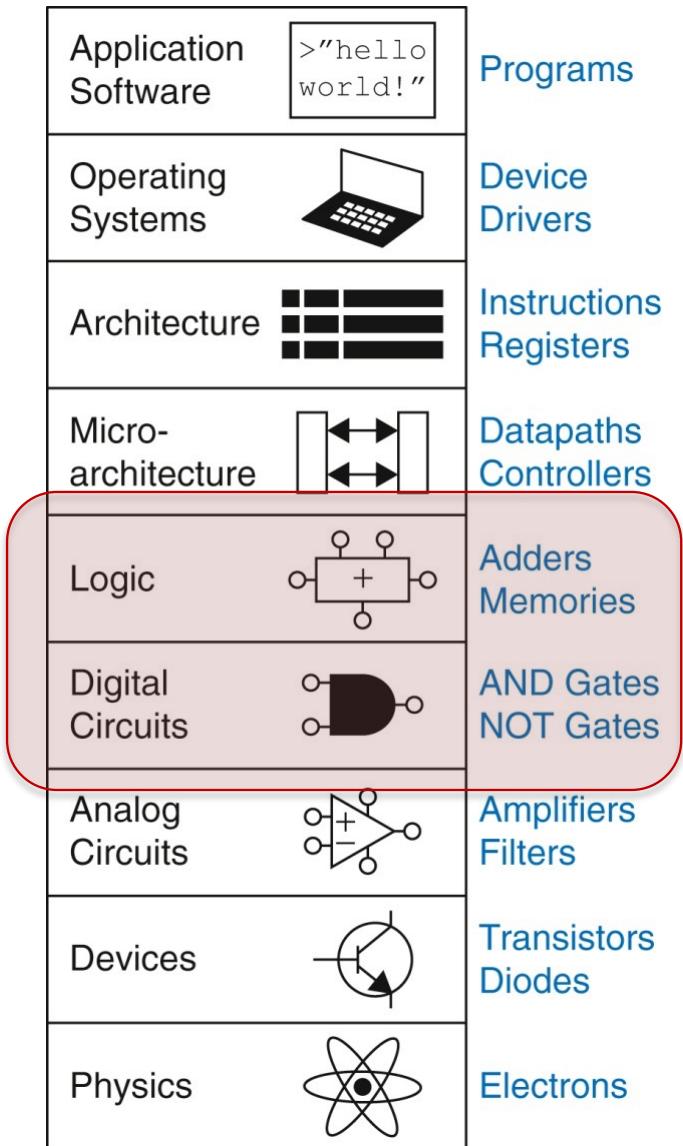
SRAM vs DRAM

- SRAM is fast and expensive
- Typically, the memory close to the CPU uses the SRAM technology
 - Register file
 - Cache (*after the teaching break*)
- The memory far from the processor uses the DRAM technology
 - Your computer's main memory is DRAM

Memory Hierarchy

- We will return to this stuff after the teaching break





We are here

Programs

Device Drivers

Instructions Registers

Datapaths Controllers

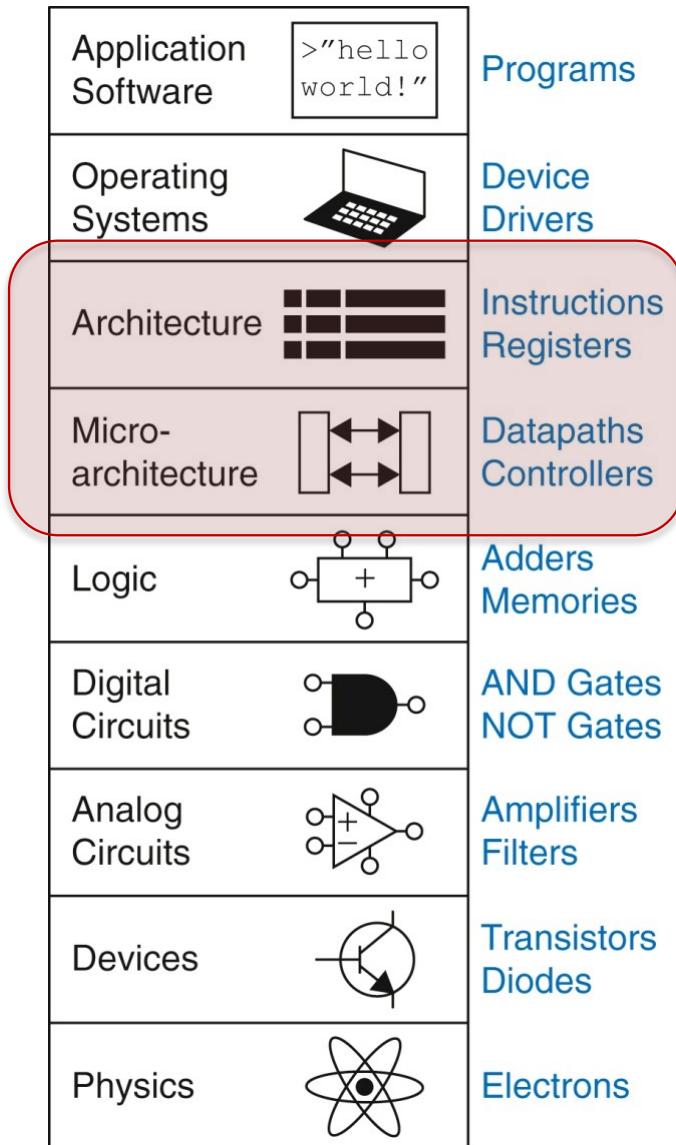
Adders Memories

AND Gates NOT Gates

Amplifiers Filters

Transistors Diodes

Electrons



Moving up a few abstraction layers?

First: Architecture
Then: Microarchitecture

The Architecture Layer

- Our goal in the first half of the course is to understand and build a processor
- We can specify the combinational circuits and the traffic light controller in English
- How should we write the specification of a processor?
 - We need a systematic approach to manage the *complexity* of a processor
- The formal specification of a processor (and computer) takes place at the “**architecture**” abstraction layer

Architecture/ISA

- The architecture or Instruction Set Architecture (**ISA**) is the programmer's view of the computer
- ISA specifies the set of instructions a computer can perform (think of it like the language of a computer)
 - Instruction = word
 - ISA = vocabulary
- Each instruction specifies
 - Operation (What exactly to do?)
 - Operands (Where to find the data to operate upon?)
- Example: **Add** two 16-bit binary numbers in registers **R1** and **R2** (recall the register file from Lab 3 – 4)

Operands

- Operands can be in register and memory
 - Programs need more than a few registers
 - Register file is small and expensive
- If operands can be in memory, then we must have instructions:
 - To *Load* from memory into registers
 - To *Store* from registers to memory
- Is there a third possibility for operand location?
 - Encoded in the instruction as 1's and 0's

Assembly Language

- Instructions written in a symbolic format so humans can read/understand them easily is called assembly language
 - ADD, SUB, LDR, STR, MUL, ROR, MOV, BIC
- A sequence of instructions is called assembly code

*Remark: Don't worry
about what this means!
Just want to show how
assembly looks*

SUB	R0,	R1,	R2
ADD	R8,	R4,	R5
ADD	R9,	R6,	R7
SUB	R3,	R8,	R9

Machine Language

- Instructions are encoded as 1's and 0's in a format called the machine language
 - ADD can be represented by binary code 0000
 - SUB can have a possible encoding of 0001
 - Register R1: 00 (example)
 - Register R2: 01

0	0	0	0	1	0	0	0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	0	1	1	1	0	0	0

Hypothetical machine code (above) is a sequence of machine language instructions stored in memory

Instruction Format

- An instruction consists of several fields
- Each field has a different meaning
- An instruction format specifies the meaning of each field
- An ISA can have many instruction formats

Microarchitecture

- An ISA is a specification
- Microarchitecture is the implementation of an ISA
 - The specific arrangement of registers, memories, ALUs, and other building blocks to form a processor is called microarchitecture
- The same ISA can have many different implementations
 - Tradeoffs in **performance**, **power**, **price**
 - A company **X** builds two processors for a high-end laptop and a cheap cell phone, respectively, that can both run programs targeting the same ISA named **Y**

More Examples

- Intel and AMD build processors targeting the x86 ISA
- QuAC ISA is for teaching purposes
 - Each group/student will build a CPU differently
 - One group may use two adders rather than one; different styles for register file
 - Both are building a processor for the same ISA
 - Their microarchitectures are different

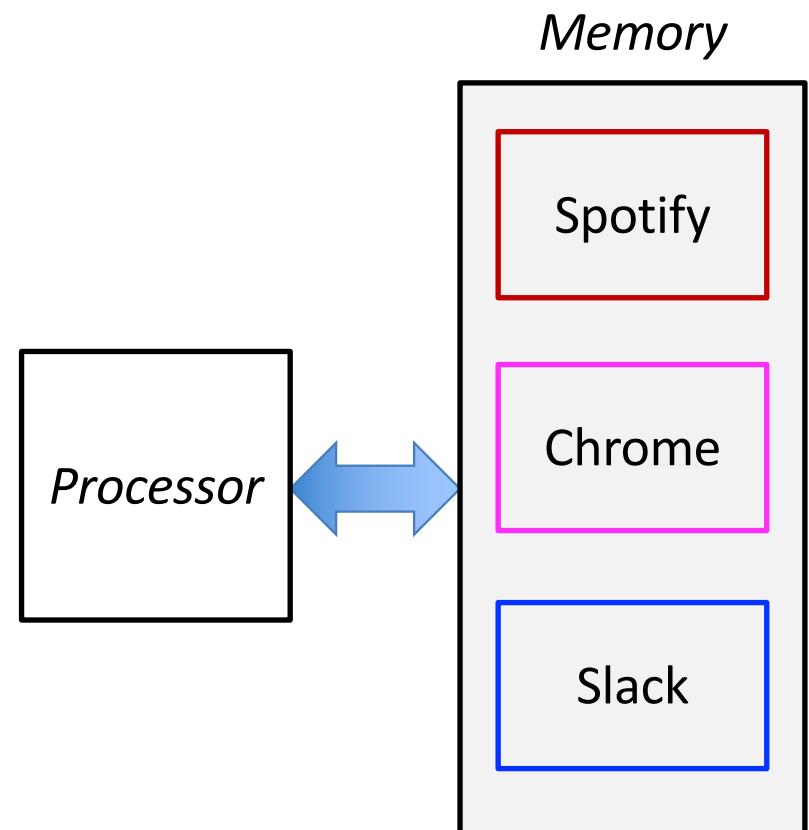


Some Remarks

- All programs running on a computer use the same instruction set
- All software applications, such as Spotify and Word, are eventually *compiled* into a series of simple instructions
- **Compiler:** A program that transforms a program written in a high-level language (C, C++, Python) to assembly instructions
- **Assembler:** A program that transforms assembly code to machine code

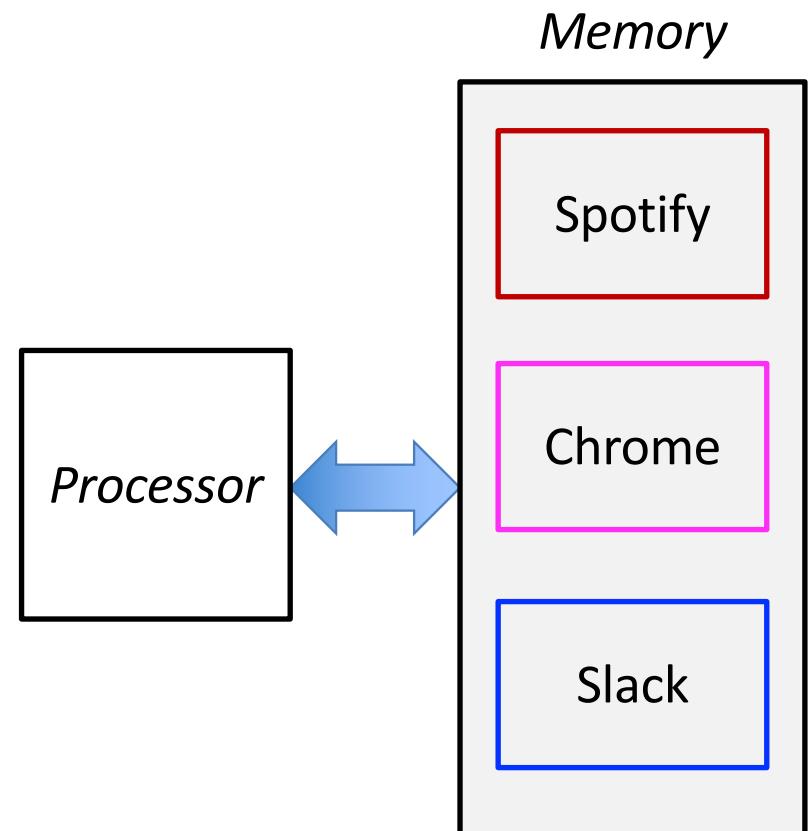
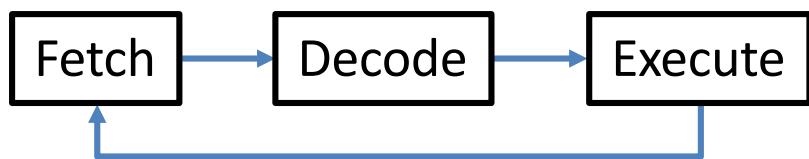
Stored Program Concept

- Two key principles
 - Instructions are represented as binary numbers
 - Programs (*in the form of machine code*) are stored in memory (*like data*)



Stored Program Concept

- How do processors execute machine code?
 - Fetch an **instruction** from memory
 - Decode the meaning of the **instruction**
 - Execute the **instruction**
 - Repeat until out of instructions



Popular ISAs

- Intel x86
 - High-performance desktop/laptop/server
 - Power-hungry (Apple's recent shift to Apple silicon)
- ARM
 - Popular in the mobile/embedded domain
 - Lecture/textbook focus
 - M1 (Apple) uses ARM architecture
- RISC-V
 - A new open-source ISA gaining momentum
- QuAC
 - Teaching purposes (invented at ANU)

Instruction Widths

- Intel x86
 - Instructions are *variable-sized* (*From one up to many bytes*)
 - Difficult to implement
- ARM, RISC-V, and QuAC
 - These ISAs have fixed-width instructions
 - ARM has 16-bit, 32-bit, and 64-bit variants
 - QuAC only has 16-bit instructions

Quiz

- Ben has an excellent idea to make computers more efficient. To try out his idea he first picks a popular existing ISA.
 - *What is the advantage of Ben's approach to pick an existing ISA instead of developing a new one?*
 - *What is the drawback of picking an existing ISA?*

Quiz

- Ben has an excellent idea to make computers more efficient. To try out his idea he first picks a popular existing ISA.
 - *What is the advantage of Ben's approach to pick an existing ISA instead of developing a new one?*
 - *Binary compatibility enables existing programs to make use of Ben's excellent idea without any effort*
 - *Historically, this aspect has led to ISA hegemony, where one popular ISA is dominant*
 - *What is the drawback of picking an existing ISA?*
 - *Think!*

ISA Choice

- We will study the **ARMv4** 32-bit ISA
 - Billions of devices (cell phones, laptops) use ARM ISA
- ISA choices impact microarchitecture complexity
 - ARM is known for low-power implementations
- The lab/assignment ISA (**QuAC**) is a 16-bit ISA
 - Implementing full ARM ISA is a multi-year effort
 - For teaching purposes, we designed QuAC



Remark: Even in lectures, we will first study a subset of ARM, and see its implementation, before moving on to more advanced features

Instruction Classes

What are the fundamental things a computer must do?

1. *Data processing instructions (arithmetic/logical instructions)*
2. *Data movement instructions (memory accesses)*
3. *Decision making instructions (if/else)*
4. *Function calls (modularity/structured programming)*

Data Processing Instructions

- Suppose I have a statement in C language
 - $a = b + c - d$

ARM Assembly Code
two instructions

ADD	t,	b,	c
SUB	a,	t,	d

- ADD and SUB are instruction mnemonics
- Instructions operate on operands
 - a, b, c are operands
- Computers operate on binary data and not variable names
- We need to specify the physical location of operands
 - Registers, memory, constants in instructions

Registers as Operands

- Instructions need fast access to operands, but memory is slow
 - Keep a small set of registers close to the CPU in a register file
 - ARM architecture uses 16 registers
 - 32-bit architecture means 32-bit registers
- Let's assume the following mapping of variables to registers in the C statement, $a = b + c - d$
 - $R0 = a, R1 = b, R2 = c, R3 = d, R4 = t$

ADD	t,	b,	c
SUB	a,	t,	d



ADD	R4,	R1,	R2
SUB	R0,	R4,	R3

Source/Destination Operand

- Instructions operate on one or two source operands and store the result after execution in a destination operand
- R1 and R2 are the *source operands* for the ADD instruction
- R4 is the *destination operand* for the ADD instruction

ADD	R4,	R1,	R2
SUB	R0,	R4,	R3

Exercise

- Translate the following C code into ARM assembly. Assume variables a – c are held in registers R0 – R2 and f – j are held in registers R3 – R7
 - $a = b - c$
 - $f = (g + h) - (i + j)$

```
    SUB    R0,    R1,    R2  
    ADD    R8,    R4,    R5  
    ADD    R9,    R6,    R7  
    SUB    R3,    R8,    R9
```

The Register Set

- ARM defines 16 *architectural* registers
- The register set is part of the ISA specification
- R0 – R12 are used for storing variables
- R13 – R15 have special uses

Constants/Immediates

- ARM instructions can use constant or immediate operands
- The value is available immediately from the instruction
 - No register or memory access
 - Immediates can be 8 – 12 bits (reason?)

In the following example, assume R7 = a , R8 = b

C code:

$a = a + 4$

$b = a - 12$

ARM Assembly Code

ADD	R7,	R7,	#4
-----	-----	-----	----

SUB	R8,	R7,	#0xC
-----	-----	-----	------

MOV Instruction

- MOV is a useful instruction for initializing register values
- MOV can also take a register source operand
 - MOV R1, R7 copies the contents of register R7 into R1

In the following example, assume R4 = i, R5 = x

C code:

```
i = 0;  
x = 4080;
```

ARM Assembly Code

```
MOV R4, #0  
MOV R5, #0xFF0
```

More Data Processing Insts.

- AND
- ORR (OR)
- EOR (XOR)
- BIC (Bit Clear)
- MVN (MoVe and Not)

The Bit Clear Instruction

- Bit Clear (BIC)
 - Masking bits (forcing unwanted bits to **0**)
- BIC R6, R1, R2
 - R2 is the mask (the bits we want to **clear** or **zero** in R1 are set to **1** in R2)
 - The instruction stores the result of R1 **AND** (**NOT** R2) in R6

Example: Data Processing

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code

AND R3, R1, R2
ORR R4, R1, R2
EOR R5, R1, R2
BIC R6, R1, R2
MVN R7, R2

Result

R3	0100 0110	1010 0001	0000 0000	0000 0000
R4	1111 1111	1111 1111	1111 0001	1011 0111
R5	1011 1001	0101 1110	1111 0001	1011 0111
R6	0000 0000	0000 0000	1111 0001	1011 0111
R7	0000 0000	0000 0000	1111 1111	1111 1111

Data Processing Instructions

- Problem
 - Programs need more than 16 registers
 - Memory is large but slow
- Solution
 - Use both register file and memory
 - Use registers whenever possible
- Two **instructions** to facilitate data movement
 - The **LDR** instruction: Bring data words from memory into the register file
 - The **STR** instruction: Store data words from the register file to memory

Memory Organization

ISA specifies the smallest unit of memory the CPU can access

All ISAs (including ARM) use byte-addressable memory

- Memory is organized into bytes
- Each byte has a unique address starting from 0 to $2^N - 1$

QuAC ISA (teaching purposes) uses word-addressable memory

- Memory is organized as 16-bit words
- Each word has a unique address starting from 0 to $2^{N-1} - 1$

Memory View (32 bits = 4 bytes)

Byte-addressable memory (each box is a byte; each row is a word)

Byte addresses (**left**) and 8-bit byte data (**right, 1 byte = 2 Hex digits**)

Byte Address	Word Address	Data	Word Number
•	•	•	•
13 12 11 10	00000010	CD 19 A6 5B	Word 4
F E D C	0000000C	40 F3 07 88	Word 3
B A 9 8	00000008	01 EE 28 42	Word 2
7 6 5 4	00000004	F2 F1 AC 07	Word 1
3 2 1 0	00000000	AB CD EF 78	Word 0

MSB LSB

4 Bytes

Reading from Memory

- Format of LoaD Register instruction
`LDR R0, [R1, #12]`
- Address calculation
 - Add base address (R1) to the offset (12)
 - Address = $(R1 + 12)$
 - Use any register for base address
 - R1 is a source (register) operand
- Result
 - R0 holds the data at memory address $(R1 + 12)$ after the instruction executes
 - R0 is a destination (register) operand

LDR Example

Read a 32-bit word of data at memory (byte) address 8 into R3. Use R2 as the base register. Show the contents of R3.

- Let's initialize R2 to 0, and add 8 as the offset

	Word Address	Data	Word Number
MOV R2, #0	:	:	:
LDR R3, [R2, #8]	00000010	CD 19 A6 5B	Word 4
	0000000C	40 F3 07 88	Word 3
R3 0x01EE2842	00000008	01 EE 28 42	Word 2
	00000004	F2 F1 AC 07	Word 1
	00000000	AB CD EF 78	Word 0

Writing to Memory

- Format of **STore Register** instruction
STR R0, [R1, #12]
- Address calculation
 - Add base address (R1) to the offset (12)
 - Address = $(R1 + 12)$
 - R0 and R1 are both source (register) operands
- Result
 - Memory address $(R1 + 12)$ contains the value in R0 after the instruction executes
 - The second operand is the destination, or the destination operand is memory address

STR Example

Store the value held in R7 into memory word 21.

- Let's initialize R5 to 0, and add 84 (21 \times 4) as the offset

```
MOV R5, #0  
STR R7, [R5, #0x54]
```

*The offset can be written in decimal or hexadecimal
84 (decimal) is 0x54 (Hex)*

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoib Akram

shoib.akram@anu.edu.au



Australian
National
University

Plan: Week 5

Last week: Finite State Machines, Timing, Pipelining

This Week: Finish looking at memories

This Week: Instruction Set Architecture (ISA)

Conditional Execution

- We may or may not want to execute a specific instruction
 - Example: *Only execute an instruction if a specific condition is satisfied*
- ARM allows conditional execution of instructions
 - **Step # 1:** Instruction sets the condition flags (**negative**, **zero**, **carry**, **overflow**)
 - **Step # 2:** Subsequent instructions execute based on the state of the condition flags
- Condition (status) flags are set by ALU
 - They are contained in a register called the Current Program Status Register (CPSR)

Setting the Condition Flags

- Method 1: Use the compare instruction

```
CMP R5, R6
```

- The instruction subtracts the second source operand from the first operand ($R6 - R5$)
- The instruction does not save any result
- Set flags as follows
 - Is 0, $Z = 1$
 - Is negative, $N = 1$
 - Causes a carry out, $C = 1$
 - Causes a signed overflow, $V = 1$

Setting the Condition Flags

- **Method 2:** Append the instruction mnemonic with S

```
ADDS R1, R2, R3
```

- The instruction adds source operands R2 and R3
- It sets the flags (S)
- It saves the result in R1

Condition Mnemonics

- Instruction may be conditionally executed based on condition flags
- Condition for execution is encoded as a ***condition mnemonic*** appended to the instruction mnemonic

CMP	R1,	R2		
SUB	NE	R3,	R5,	R8
ADDE	EQ	R1,	R2,	R3

- **NE** and **EQ** are condition mnemonics
- SUB executes only if R1 is not equal to R2 (i.e., Z = 0)

Condition Mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	\bar{Z}
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	\bar{C}
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	\bar{N}
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	\bar{V}
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z \text{ OR } \bar{C}$
1010	GE	Signed greater than or equal	$N \oplus V$
1011	LT	Signed less than	$\bar{N} \oplus V$
1100	GT	Signed greater than	$\bar{Z}(N \oplus V)$
1101	LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Instructions that affect condition flags

Type	Instructions	Condition Flags
Add	ADDS, ADCS	N, Z, C, V
Subtract	SUBS, SBCS, RSBS, RSCS	N, Z, C, V
Compare	CMP, CMN	N, Z, C, V
Shifts	ASRS, LSLS, LSRS, RORS, RRXS	N, Z, C
Logical	ANDS, ORRS, EORS, BICS	N, Z, C
Test	TEQ, TST	N, Z, C
Move	MOVS, MVNS	N, Z, C
Multiply	MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS	N, Z

Example – 1

- R5 = 17 and R9 = 23
- Will the SUBEQ and ORRMI instructions execute?
 - NZCV = ????

CMP	R5,	R9	
SUBEQ	R1,	R2,	R3
ORRMI	R4,	R0,	R9

Example – 2

- R2 = 0x80000000 and R3 = 0x00000001
 - NZCV = ????
 - Which instructions will execute?

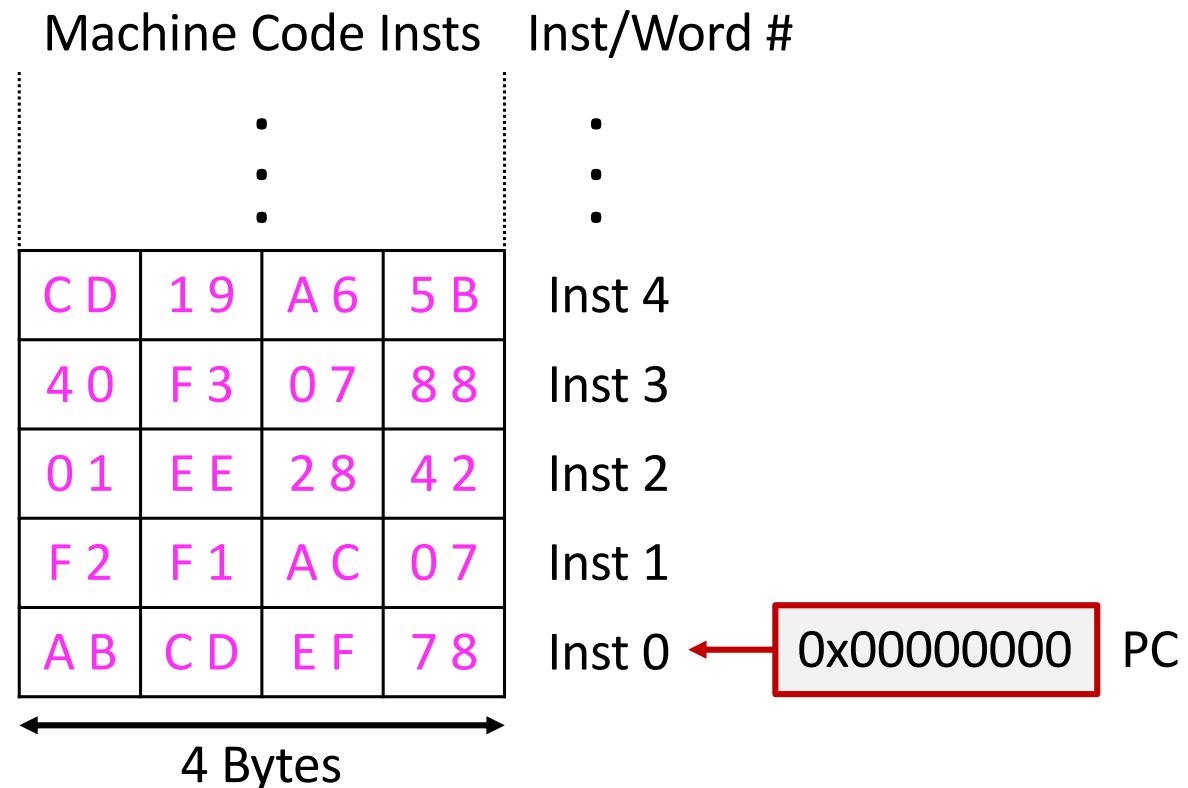
CMP	R2,	R3	
ADDEQ	R4,	R5,	#78
ANDHS	R7,	R8,	R9
ORRMI	R10,	R11,	R12
EORLT	R12,	R7,	R10

Branch Instructions

- Decision making is an important advantage of computers
 - Perform a different task based on the input
 - if and if/else statements
 - for and while loops
 - switch/case statements
- ARM provides branch instructions to skip/repeat code
- Branch instructions are more efficient/powerful than conditional execution
 - E.g., conditional execution cannot implement loops

Program Counter (PC)

- PC is a special register that points to the current instruction
- To execute the next instruction, the CPU increments the PC by 4
- In ARM, the register R15 is the PC
- Without any branch instruction, the CPU increments the PC and executes the next instruction



Branch Instructions

- Branch instructions change the program flow or the flow of code execution
 - Normal execution (without branch): Increment PC by 4 bytes ($PC = PC + 4$) to fetch the next instruction
 - Branch instruction: Change PC to the address of the target instruction, and fetch the target instruction
- Target instruction
 - Target instruction may be few or many bytes away
 - Remember: Target is an address in main memory

Type of Branches

- Branch (**B**)
 - Branches to another instruction
- Branch and Link (**BL**)
 - A special branch instruction to provide support for functions in high-level languages
- The branch instructions can be conditional or unconditional
- Examples of conditional branch instructions
 - **BEQ**
 - **BNE**

Unconditional Branch

- Branch is *unconditional* and thus *always taken*
- After encountering the branch, the CPU executes SUB instead of ORR

	ADD	R1,	R2	#17
	B	TARGET		
	ORR	R1,	R1,	R3
	AND	R3,	R1,	#0xFF
TARGET				
	SUB	R1,	R1,	#78

- The label TARGET is translated by the assembler into a memory address

Conditional Branch

Branch instructions can execute conditionally based on the condition mnemonics

- When the code reaches BEQ, the Z condition flag is **0** because **R0** is not equal to **R1**
- Branch is *not taken*
- The next instruction executed is the ORR instruction

MOV	R0,	#4	
ADD	R1,	R0,	R0
CMP	R0,	R1	
BEQ	THERE		
ORR	R1,	R1,	R1
THERE			
ADD	R1,	R1,	#78

if Statement

C code:

```
if (apples == oranges)
    f = i + 1;
f = f - i;
```

ARM Assembly Code

CMP	R0,	R1	
BNE	L1		
ADD	R2,	R3	#1
L1			
SUB	R2,	R2,	R3

- The assembly code below checks for the opposite condition in C code
- Skips the if block if the condition is not satisfied
- If the branch is *not taken*, if block is executed

apples == oranges?

if not equal, skip if block

if block: $f = i + 1$

$f = f - i$

R0 = apples, R1 = oranges, R2 = f, R3 = i

if Statement

- The assembly code tests the opposite condition of the one in C code
- Skips the if block if the condition is **not** satisfied
- If the branch is *not taken*, if block is executed

- Question: Is there a different way of writing the assembly code for the if statement?
 - Using the **BEQ** instruction instead of **BNE** (Try it yourself)
 - Using conditional execution (next)

if Statement with Conditional Execution

C code:

```
if (apples == oranges)
    f = i + 1;
f = f - i;
```

ARM Assembly Code

CMP	R0,	R1	
ADDEQ	R2,	R3	#1
SUB	R2,	R2,	R3

apples == oranges?

if block: $f = i + 1$ on equality ($Z = 1$)

$f = f - i$

R0 = apples, R1 = oranges, R2 = f, R3 = i

if Statement with Conditional Execution

- One fewer instruction
 - *The solution is shorter and faster*
 - *Branch instructions sometimes introduce extra delay*
- As the if block becomes longer, the branch becomes valuable
 - Conditional execution requires needless fetching of instructions
 - Tedious to write code

if/else Statement

- Execute one of two blocks of code depending on the condition
- When the condition in the if block is met, the if block is executed
- Otherwise, the else block is executed

if/else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
```

ARM Assembly Code

CMP	R0,	R1	
BNE	L1		
ADD	R2,	R3	#1
B	L2		
L1			
SUB	R2,	R2,	R3
L2			

if not equal, skip if block

skip else block

R0 = apples, R1 = oranges, R2 = f, R3 = i

if/else Statement

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
```

Homework Exercise: Find an alternative way to write the if/else statement.

if/else Statement with Conditional Execution

C code:

```
if (apples == oranges)
    f = i + 1;
else
    f = f - i;
```

ARM Assembly Code

CMP	R0,	R1	
ADDEQ	R2,	R3	#1
SUBNE	R2,	R2,	R3

Machine Language

- Instructions are stored in memory as 32-bit words
- ARM defines three instruction formats
 - Data processing (DP)
 - Memory
 - Branch
- Instruction *fields* encode the instruction operation and its operands
- Binary encoding of three instruction formats will allow us to build a CPU for the ARM ISA

Instruction Format – 1: DP

31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src2

- Operands
 - Rn: first source operand register (0000, 0001, ..., 1111)
 - Src2: second source register or immediate
 - Rd: destination register
- Control fields
 - cond: specifies conditional execution (**1110** for unconditional)
 - op: the operation code or opcode (**00**)
 - funct: the function/operation to perform

Instruction Format – 1: DP



- op = 00 for DP instructions
- cmd specifies the specific DP instruction (0100 for ADD and 0010 for SUB)
- I-bit
 - I = 0: Src2 is a register
 - I = 1: Src2 is an immediate
- S-bit: 1 if sets condition flags

Two DP Formats (Src2 Variations)

Immediate (assume 11:8 are 0 for now)

	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
cond	00	1		cmd	S	Rn	Rd	0 0 0 0	imm8

Register (assume 11:4 are zero for now)

	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
cond	00	0		cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

Plan: Week 6

Last week: Instruction Set Architecture (specification)

This Week: Microarchitecture (implementation)

Machine Language

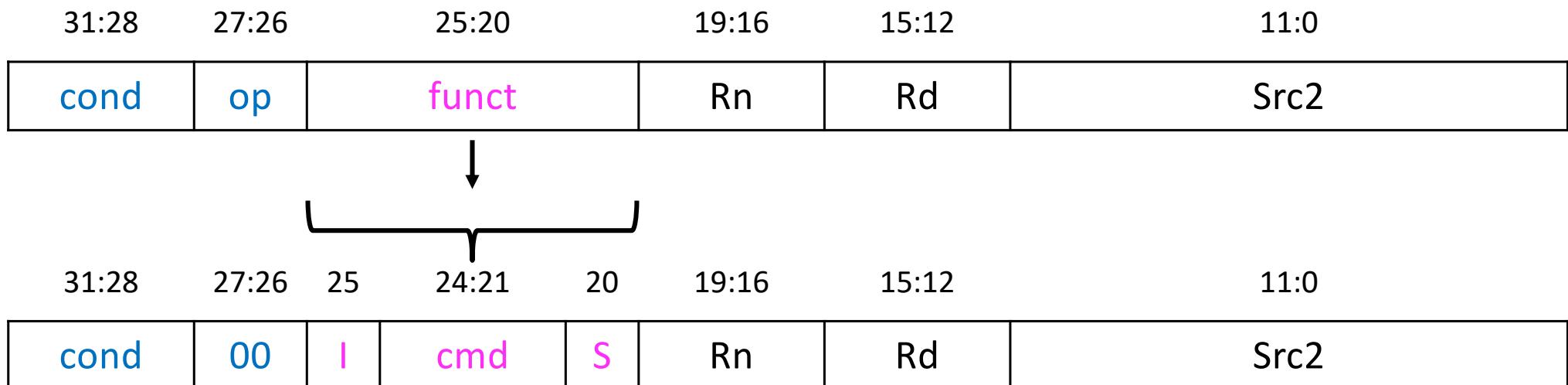
- Instructions are stored in memory as fixed-size words
 - For ARMv4, each instruction is 32 bits wide
- ARM defines three instruction formats
 - Data processing (DP)
 - Memory
 - Branch
- Instruction *fields* encode the instruction operation and its operands
- Understanding binary encoding of instructions is necessary for building the CPU

Instruction Format – 1: DP

31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src2

- Operands
 - Rn: first source operand register (0000, 0001, ..., 1111)
 - Src2: second source register or immediate
 - Rd: destination register
- Control fields
 - cond: specifies conditional execution ([1110](#) for unconditional)
 - op: the operation code or opcode ([00](#))
 - funct: the function/operation to perform

Instruction Format – 1: DP



- op = **00** for DP instructions
- cmd specifies the specific DP instruction (**0100** for ADD and **0010** for SUB)
- I-bit
 - I = **0**: Src2 is a register
 - I = **1**: Src2 is an immediate
- S-bit: **1** if the instruction sets the condition flags

DP with Src2 as Immediate

- Bit 25 (I) informs the CPU how to interpret Src2
 - I = 1, CPU interprets Src2[7:0] as an unsigned 8-bit constant
- Format (Src2 = immediate)

ADD R0, R1, #16

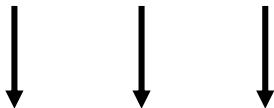
↓ ↓ ↓
ADD Rd, Rn, #imm8

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
cond	00	1	cmd	S	Rn	Rd	0 0 0 0	imm8

DP with Src2 as Register

- Bit 25 informs the CPU how to interpret Src2
 - $I = 0$, CPU interprets Src2[3:0] as a register
- Format (Src2 = Register)

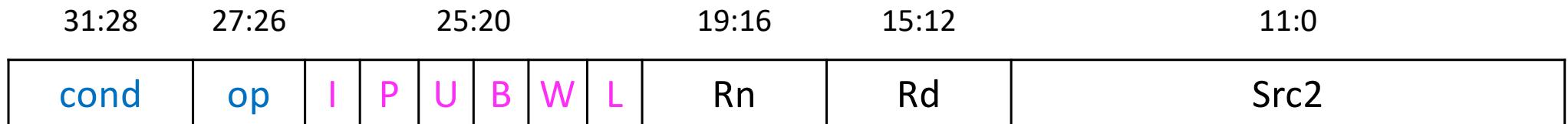
ADD R0, R1, R3



ADD Rd, Rn, Rm

31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0	Rm

Instruction Format – 2: Memory



- op = 01
- Rn = base register (base address)
- Rd = destination (load), source (store)
- Src2 = offset (register, shifted register, immediate)
- funct = 6 control bits
 - I (Bit 25): Encoding of Src2
 - L (Bit 20): Load or Store
 - Remaining bits (ignore)

LDR with Src2 as Immediate

- I (Bit 25) = 1: Src2 = imm12 where imm2 is a 12-bit unsigned offset added to the value in the base register (Rn)
- Format of LoaD Register instruction

LDR R0, [R1, #12]

↓ ↓ ↓
LDR Rd, [Rn, #imm12]

- L (Bit 20) = 1: CPU performs an LDR

31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 1	Rn	Rd	imm12

STR with Src2 as Immediate

- I (Bit 25) = **1**: Src2 = imm12 where imm2 is a 12-bit unsigned offset added to the value in the base register (Rn)
- Format of **STore Register** instruction

STR R0, [R1, #12]



STR Rd, [Rn, #imm12]

- L (Bit 20) = **0**: CPU performs an STR

	31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1	1 1 0 0 1	Rn	Rd	imm12

Instruction Format – 3: Branch

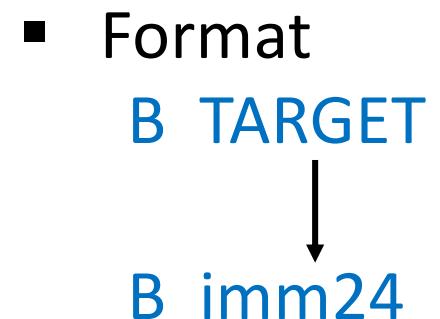
31:28	27:26	25:24	23:0
cond	op	1L	imm24

- op = 10
- imm24 = 24-bit **signed** immediate
- The two bits 25:24 form the **funct** field
 - Bit 25 is always 1
 - L bit: L = 0 for B (Branch)
 - L bit: L = 1 for BL (Branch and Link, ignore for now)

Branch with L = 0

31:28	27:26	25:24	23:0	imm24
cond	10	10		

- op = 10
- imm24 = 24-bit **signed** immediate
- The two bits 25:24 form the **funct** field
 - Bit 25 is always 1
 - L bit: L = 0 for B (Branch)

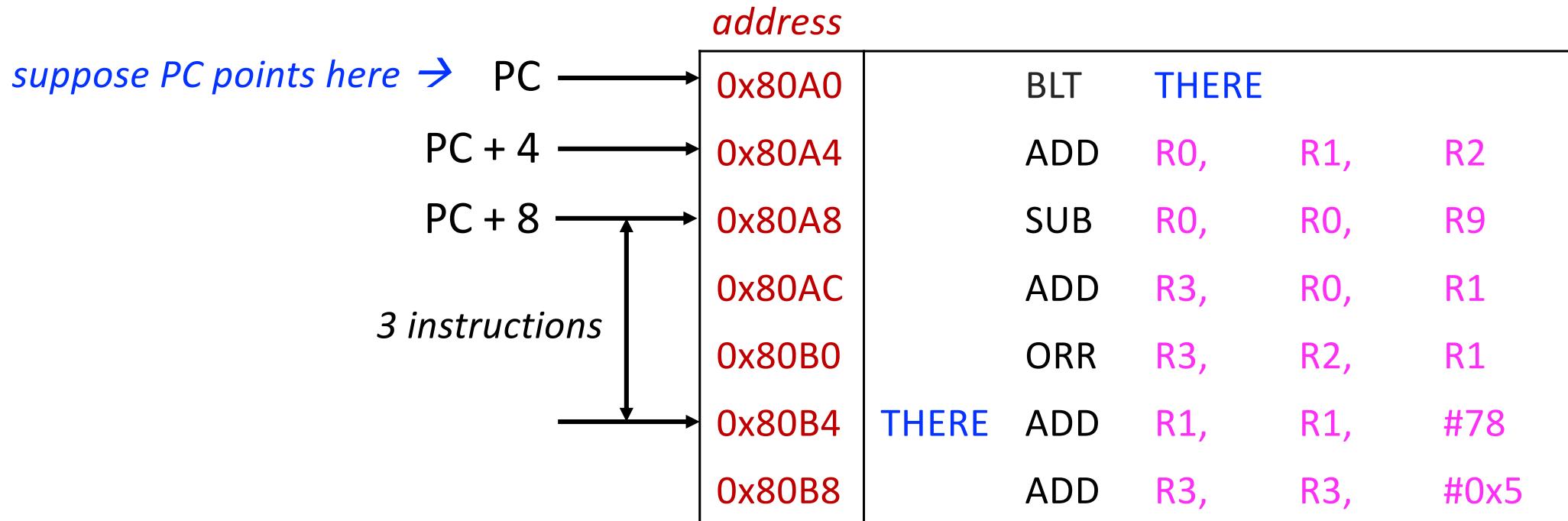


Branch Target Address (BTA)

31:28	27:26	25:24	23:0	imm24
cond	10	10		

- The 24-bit two's complement **imm24** field specifies the instruction address relative to **PC + 8**
 - Why **PC + 8?** (historical reasons)
- **BTA:** The address of the next instruction to execute if the branch is taken
- The imm8 field is the number of instructions between the BTA and PC + 8 (two instructions past the branch)

BTA Calculation Example



31:28 27:26 25:24

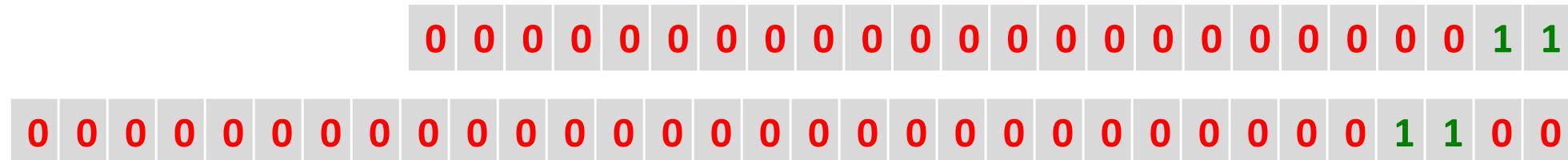
23:0

cond	10	10	imm24 = 3 (00000000000000000000000011)
------	----	----	--

BTA Calculation

The processor calculates the BTA in three steps

1. Shift left imm24 by 2 (to convert words to bytes)
2. Sign-extend (copy Instr_{25} into $\text{Instr}_{31:26}$)
3. Add PC + 8



Summary of Formats

	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
DP-I	cond	00	1	cmd	S	Rn	Rd	0 0 0 0	imm8
	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
DP-R	cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0 0 0	Rm
	31:28	27:26	25:20		19:16	15:12		11:0	
Mem	cond	01	1 1 1 0 0 L		Rn	Rd		imm12	
	31:28	27:26	25:24			23:0			
BR	cond	10	10				imm24		

Exercise – 1

- Write the 32-bit ARM machine language representation for the following instruction?
 - SUB R8, R9, R10
 - For SUB, cmd is 0010; for unconditional exec, cond = 1110

	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
DP-R	cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

Exercise – 1

- Write the 32-bit ARM machine language representation for the following instruction?
 - SUB R8, R9, R10
 - For SUB, cmd is 0010; for unconditional exec, cond = 1110

	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
DP-R	cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
SUB	1110	00	0	0010	0	1001	1000	0 0 0 0 0 0 0 0	1010

HEX: E0 49 80 0A

Exercise – 2

- Write the 32-bit ARM machine language representation for the following instruction?
 - ADD R0, R1, #42
 - For ADD, cmd is 0100; for unconditional exec, cond = 1110

	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
DP-I	cond	00	1	cmd	S	Rn	Rd	0 0 0 0	imm8

Exercise – 2

- Write the 32-bit ARM machine language representation for the following instruction?
 - ADD R0, R1, #42
 - For ADD, cmd is 0100; for unconditional exec, cond = 1110

	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
DP-I	cond	00	1	cmd	S	Rn	Rd	0 0 0 0	imm8
ADD	1110	00	1	0100	0	0001	0000	0 0 0 0	00101010

HEX: E2 81 00 2A

Exercise – 3

- Write the 32-bit ARM machine language representation for the following instructions?
 - LDR R7, [R0, #16]

RISC vs. CISC Architectures

- Reduced Instruction Set Computer (**RISC**)
 - Small # of simple instructions
 - Simple hardware (e.g., easy to decode)
 - ARM, RISC-V, QuAC, MIPS
- Complex Instruction Set Computer (**CISC**)
 - Many more instructions, and some instructions are very complex
 - x86 DP/ALU instructions can have memory operands
 - One CISC instruction = Many RISC instructions
 - Complex instructions are costly to implement

RISC Principles

Simplicity favors regularity

- Consistent number of operands in DP instructions

Make the common case fast

- Include a few frequently used instructions

Smaller is faster

- Use a small and fast register file
- Large # registers = Large decoding circuitry

Good design demands good compromises

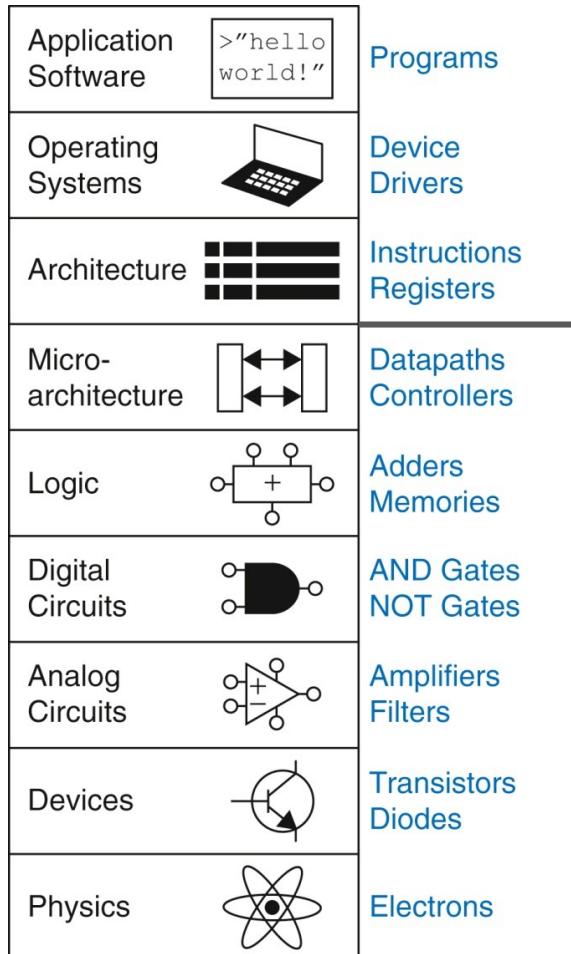
- Simplicity encourages a single instruction format, but that is too restrictive
- Define multiple formats, but have some regularity among instructions to simplify decoding



Patterson & Hennessy

2018 Turing Award

The Big Picture



- *The compiler translates high-level programs to assembly*
- *The assembler translates assembly code to machine code*

Software

*ISA is the Sw/Hw boundary
(Contract/agreement)*

Hardware



- *Machine code resides in main memory*
- *CPU fetches/executes instructions*
- *Microarchitecture is the specific arrangement of adders, memories, registers, etc., to implement an ISA*

Plan for Weeks 6 – 7

Week 5

- We simplified the ISA by setting some control bits to **0** or **1**

Week 6

- Build a CPU for the simplified ARM 32-bit instruction set

Week 7

- Study the remaining ISA features

Microarchitecture

- Two interacting parts
 - Datapath
 - Control unit
- Datapath *operate* on words of data
 - *Register file, ALU, memories*
- Control unit *informs* the datapath how to execute an instruction
 - *Reads the instruction and generate multiplexer selects, register enable, and memory write signals*

LDR Instruction

- We will add the logic for one instruction at a time beginning with the LDR instruction
- Format of LoaD Register instruction

LDR R0, [R1, #12]

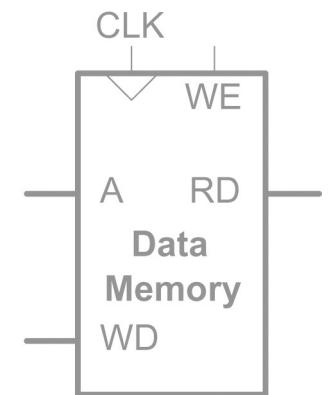
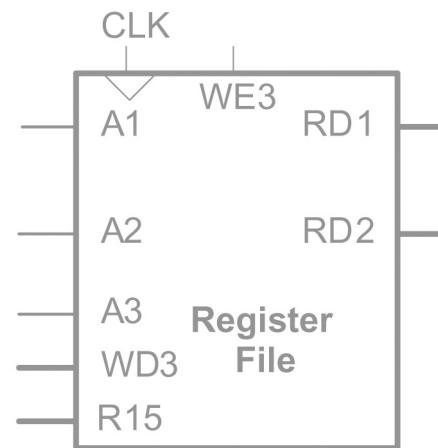
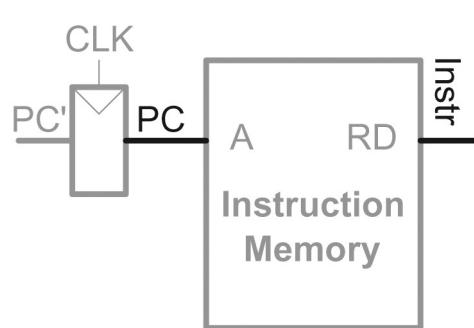


LDR Rd, [Rn, #imm12]

31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The LDR Datapath

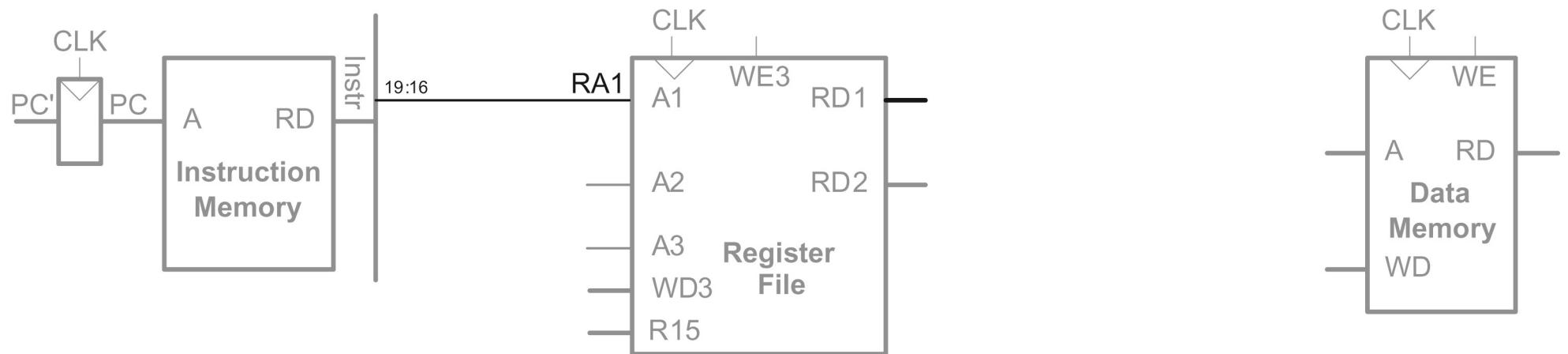
Step 1: Read (Fetch) instruction from memory



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The LDR Datapath

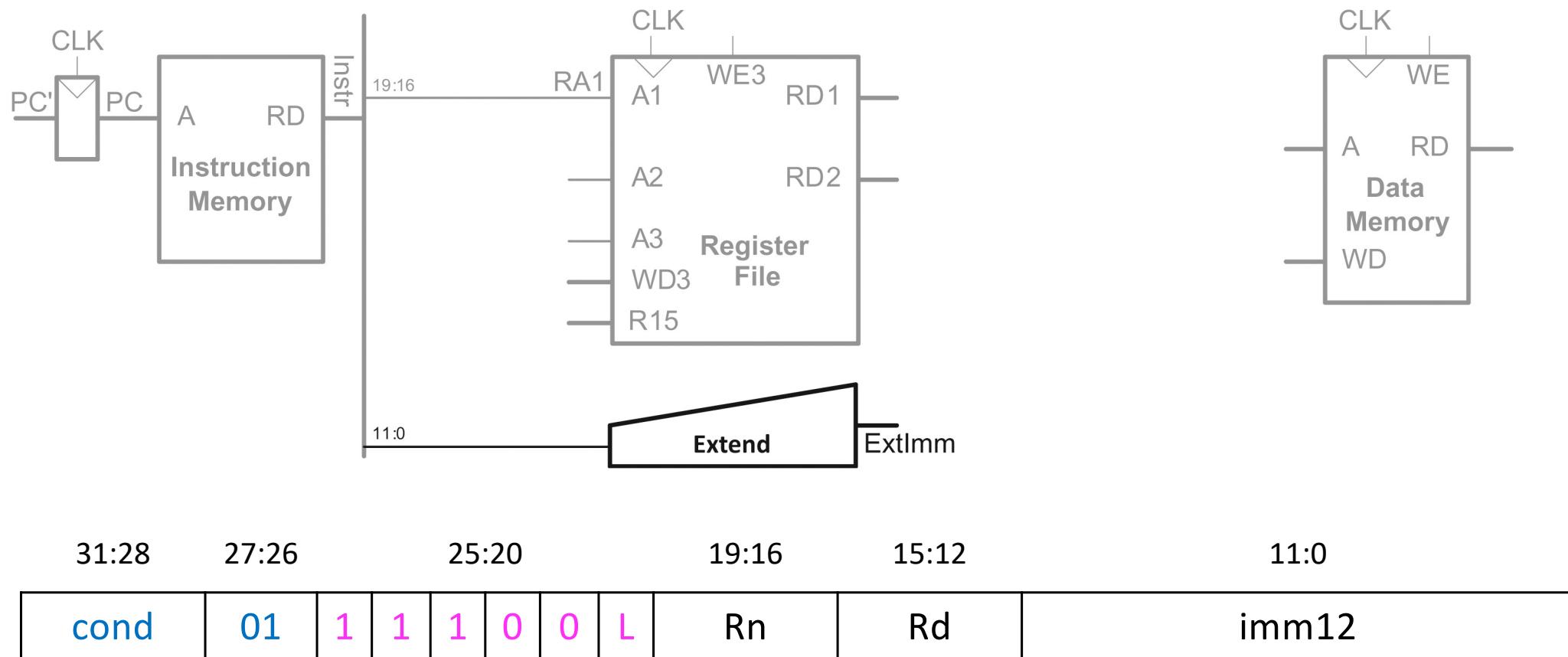
Step 2: Read source operand (base register) from register file



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

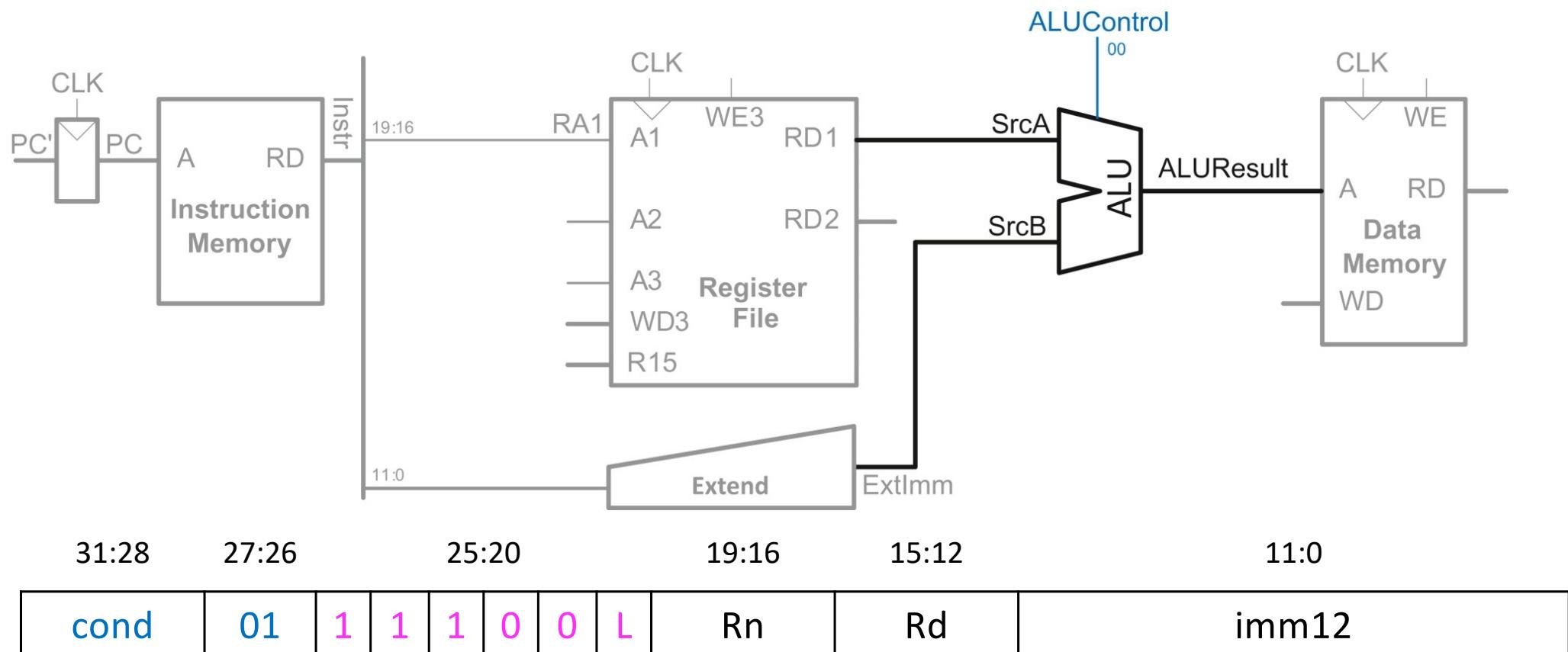
The LDR Datapath

Step 3: Zero-extend the immediate field stored in $\text{Instr}_{11:0}$



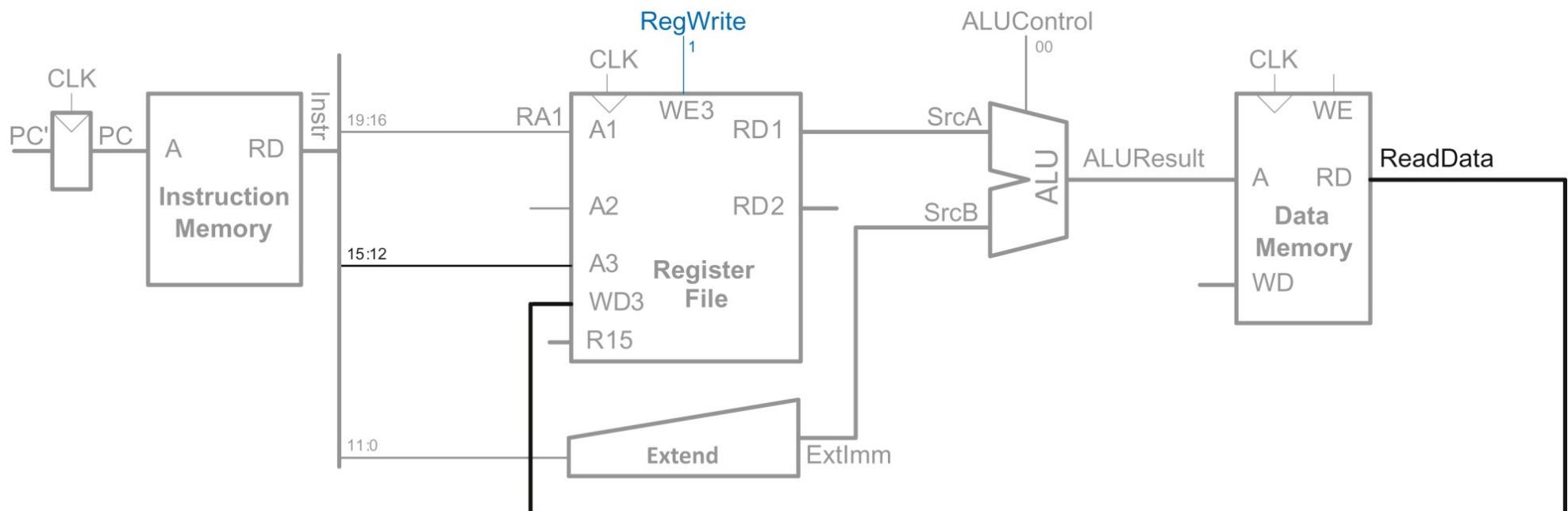
The LDR Datapath

Step 4: Compute memory address (ALUControl = 00)



The LDR Datapath

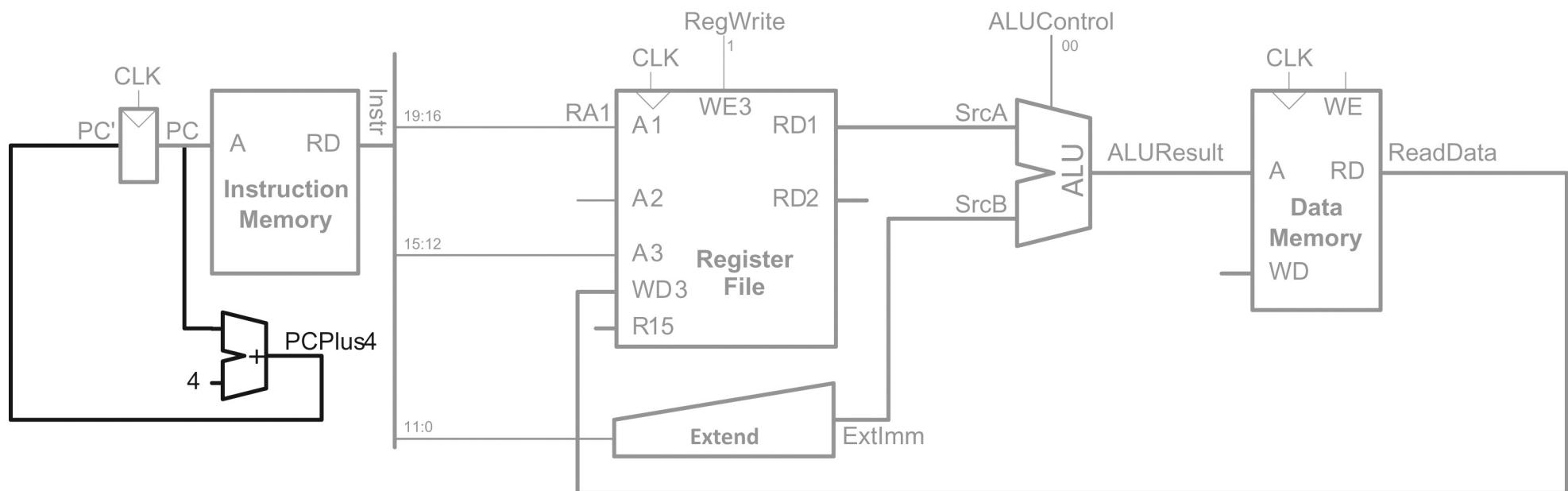
Step 5: Write back data from read by data memory to Rd in Reg File



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The LDR Datapath

Step 6: Compute address of next instruction ($PC' = PC + 4$)



PC will become PC' the following cycle (recall photography example)

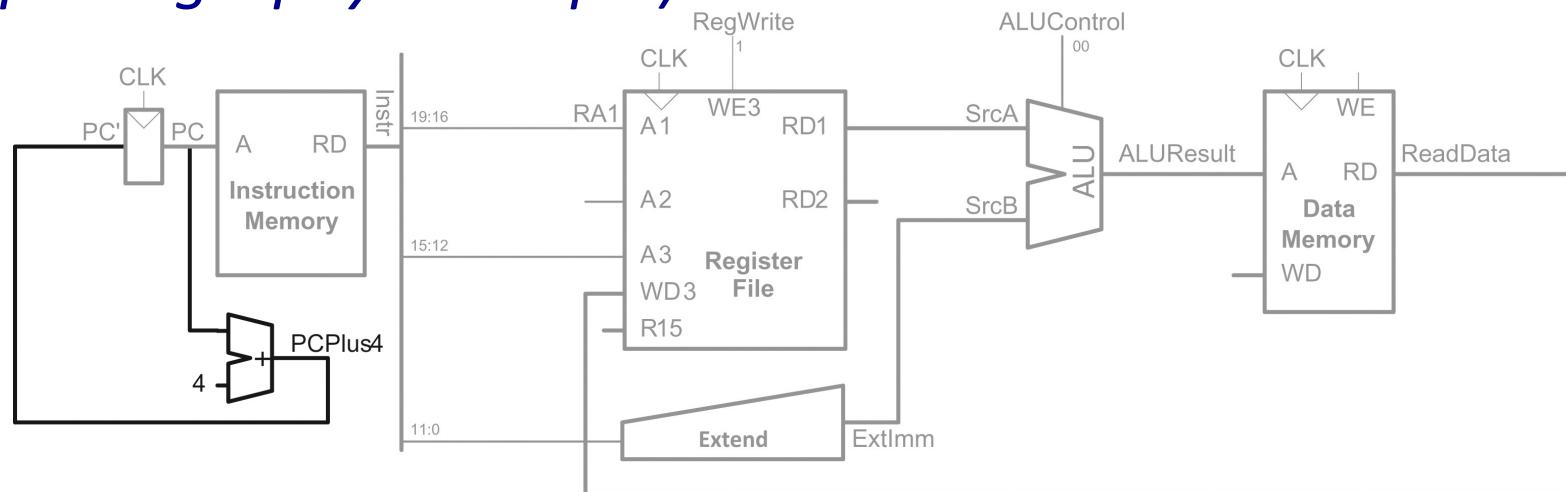
31:28 27:26 25:20 19:16 15:12 11:0

cond	01	1	1	1	0	0	L	Rn	Rd	imm12
------	----	---	---	---	---	---	---	----	----	-------

The LDR Datapath

Step 6: Compute address of next instruction ($PC' = PC + 4$)

*Important: PC will become PC' the following cycle
(recall photography example)*



31:28

27:26

25:20

19:16

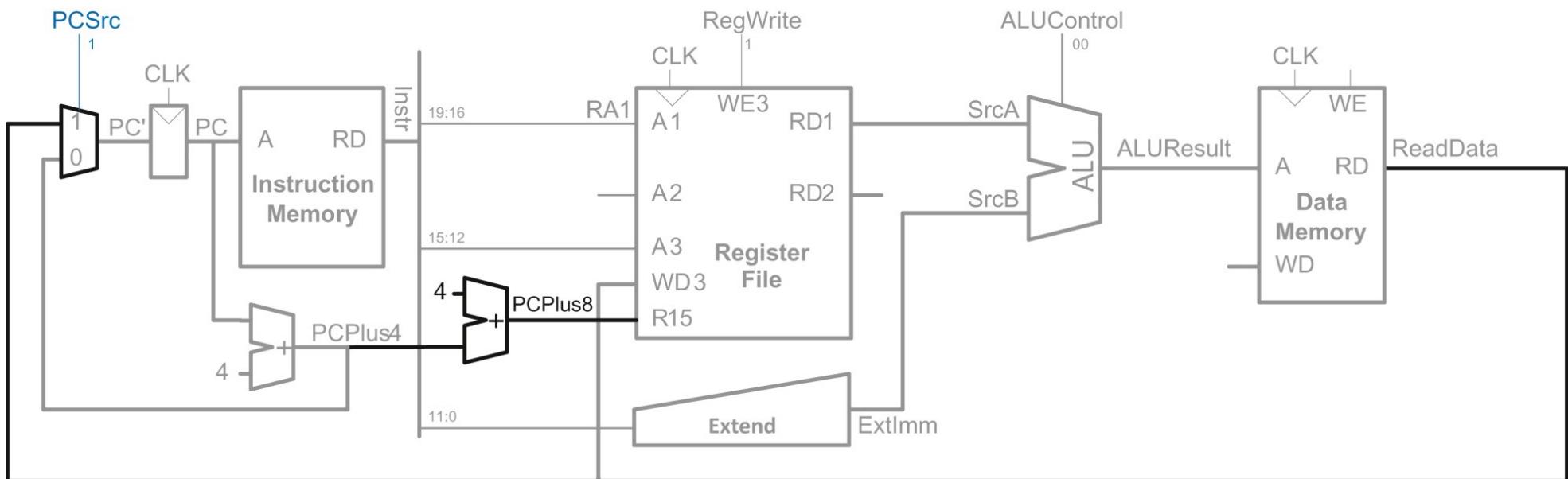
15:12

11:0

cond	01	1	1	1	0	0	L	Rn	Rd	imm12
------	----	---	---	---	---	---	---	----	----	-------

The LDR Datapath

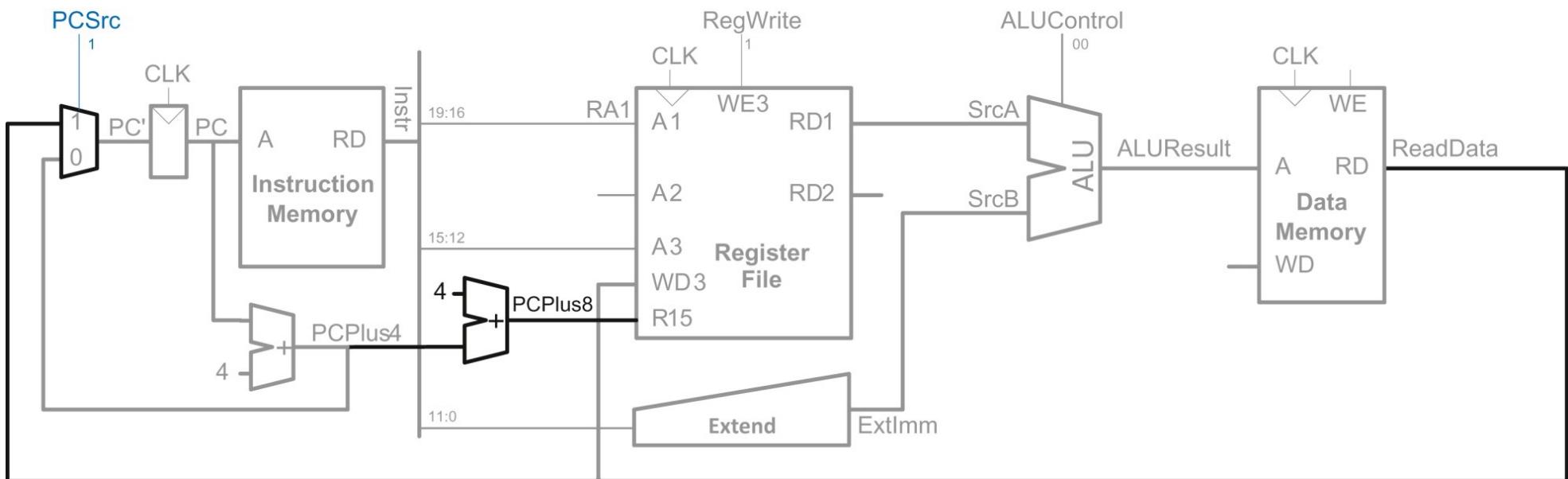
Step 7/a: Reading register R15 returns PC + 8



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The LDR Datapath

Step 7/b: Writing register R15 (PC may be an instruction's result)



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

STR Instruction

- STR instruction uses the same instruction format
- LDR and STR behave differently at the machine level
- Rd is a source operand (specifies the register to store to mem)
- Format of **STore Register** instruction

STR R0, [R1, #12]

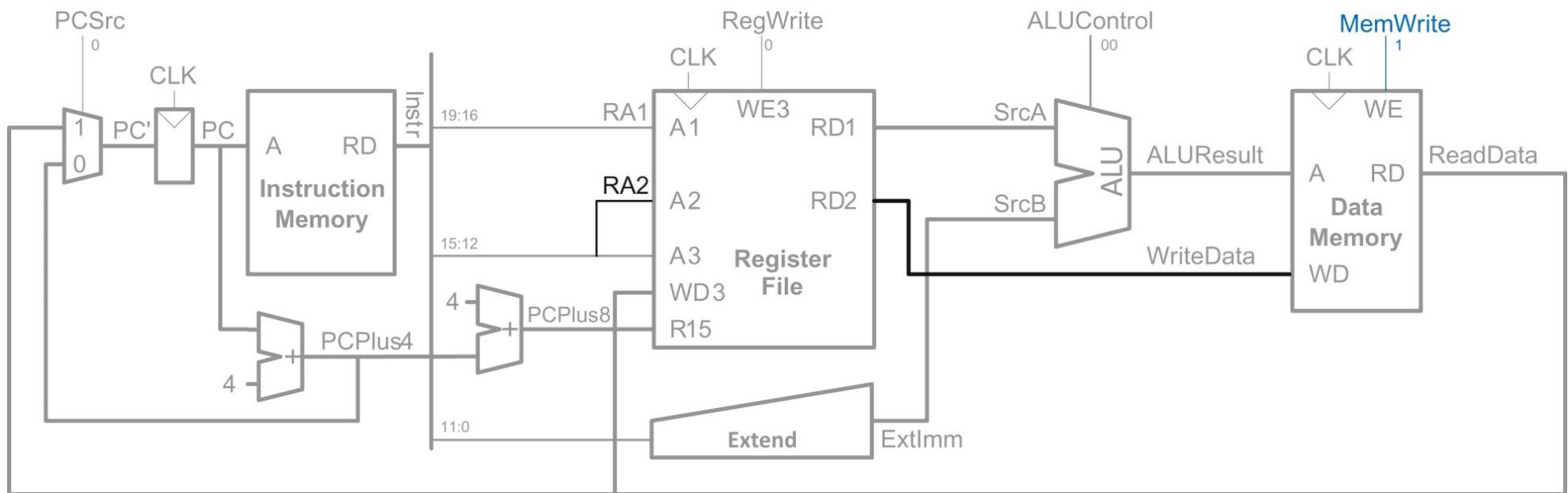


STR Rd, [Rn, #imm12]

31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

The STR Datapath

Step 8: Read a second register (Rd) and write its value to memory



31:28	27:26	25:20	19:16	15:12	11:0
cond	01	1 1 1 0 0 L	Rn	Rd	imm12

DP Instructions: Immediate

- Like the LDR instruction, but two important differences
 - imm8 instead of imm12
 - The destination register stores the result of the ALU operation instead of memory access
- Format

ADD R0, R1, #16
↓ ↓ ↓
ADD Rd, Rn, #imm8

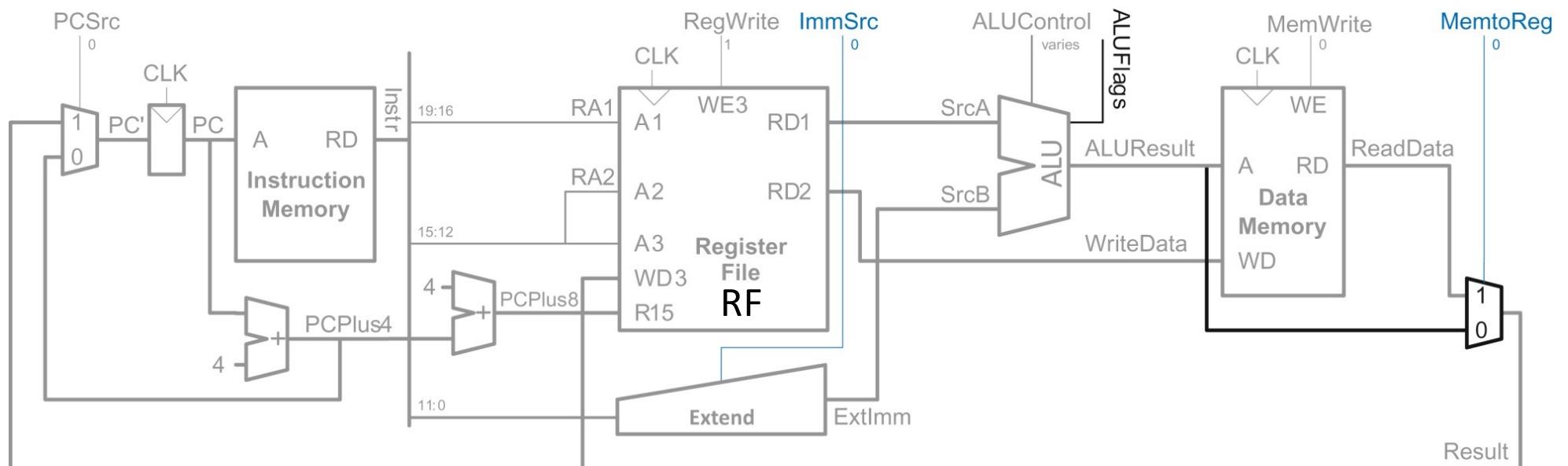
31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
cond	00	1	cmd	S	Rn	Rd	0 0 0 0	imm8

Recall the ALU Functions

ALUControl	Function
00	ADD
01	SUB
10	AND
11	ORR

DP-Immediate Datapath

Step 9: Change extend block, and add signal to write ALU result to RF



31:28 27:26 25 24:21 20 19:16 15:12 11:8 7:0

cond	00	1	cmd	S	Rn	Rd	0	0	0	0	imm8
------	----	---	-----	---	----	----	---	---	---	---	------

DP Instructions: Register

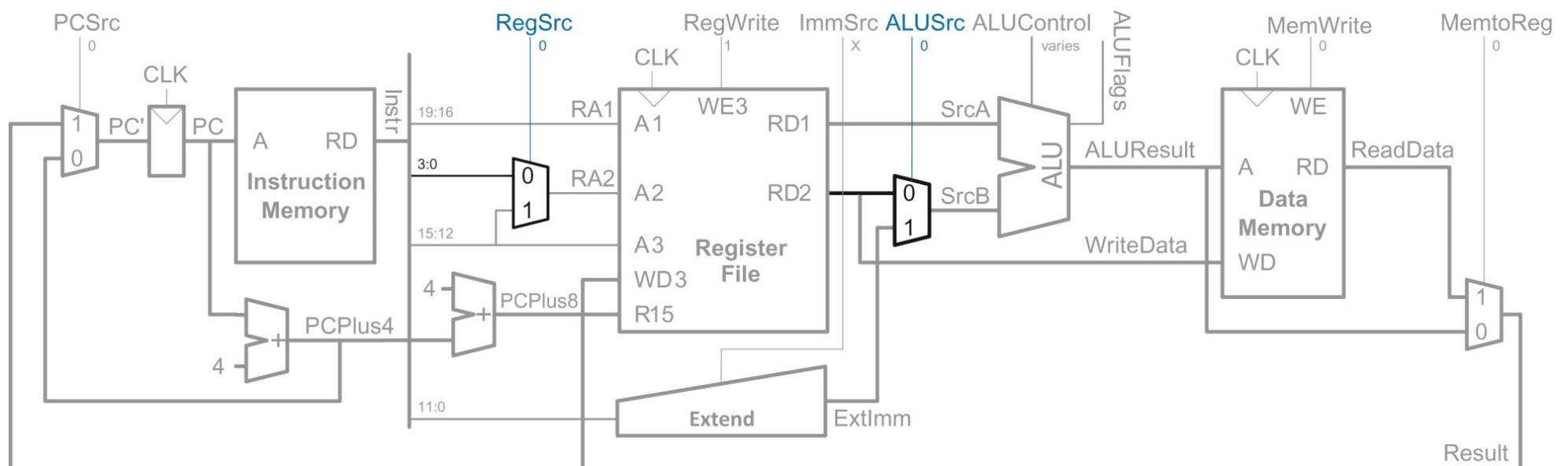
- The second source operand is Rm instead of an immediate
- Place Rm on the A2 port of the register file for DP instructions with register as the second operand
- Format

ADD R0, R1, R3
↓ ↓ ↓
ADD Rd, Rn, Rm

31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

DP-Register Datapath

Step 10: Read 2nd register (Rm) from Reg File and send RD2 to ALU



31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

Branch Instruction: Unconditional

- The second source operand is Rm instead of an immediate
- Place Rm on the A2 port of the register file for DP instructions with register as the second operand
- Format

B TARGET

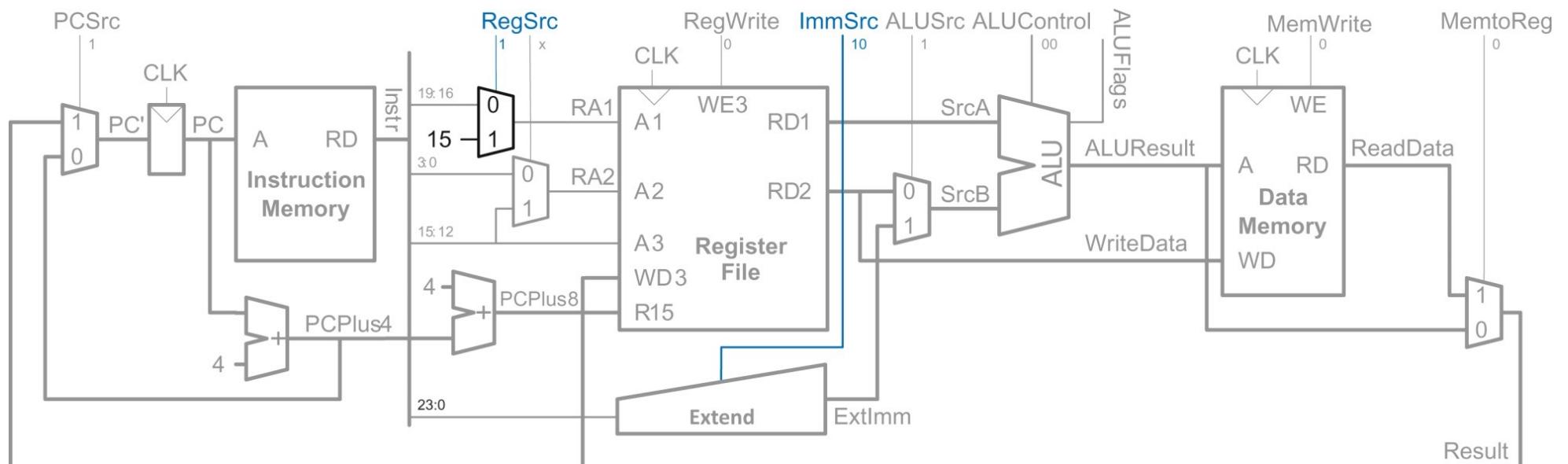


B imm24

31:28	27:26	25:24	23:0
1110	10	10	imm24

Branch Datapath

Step 11: Change extend block, and add a bit to RegSrc for branch



31:28 27:26 25:24

23:0

1110	10	10	imm24
------	----	----	-------

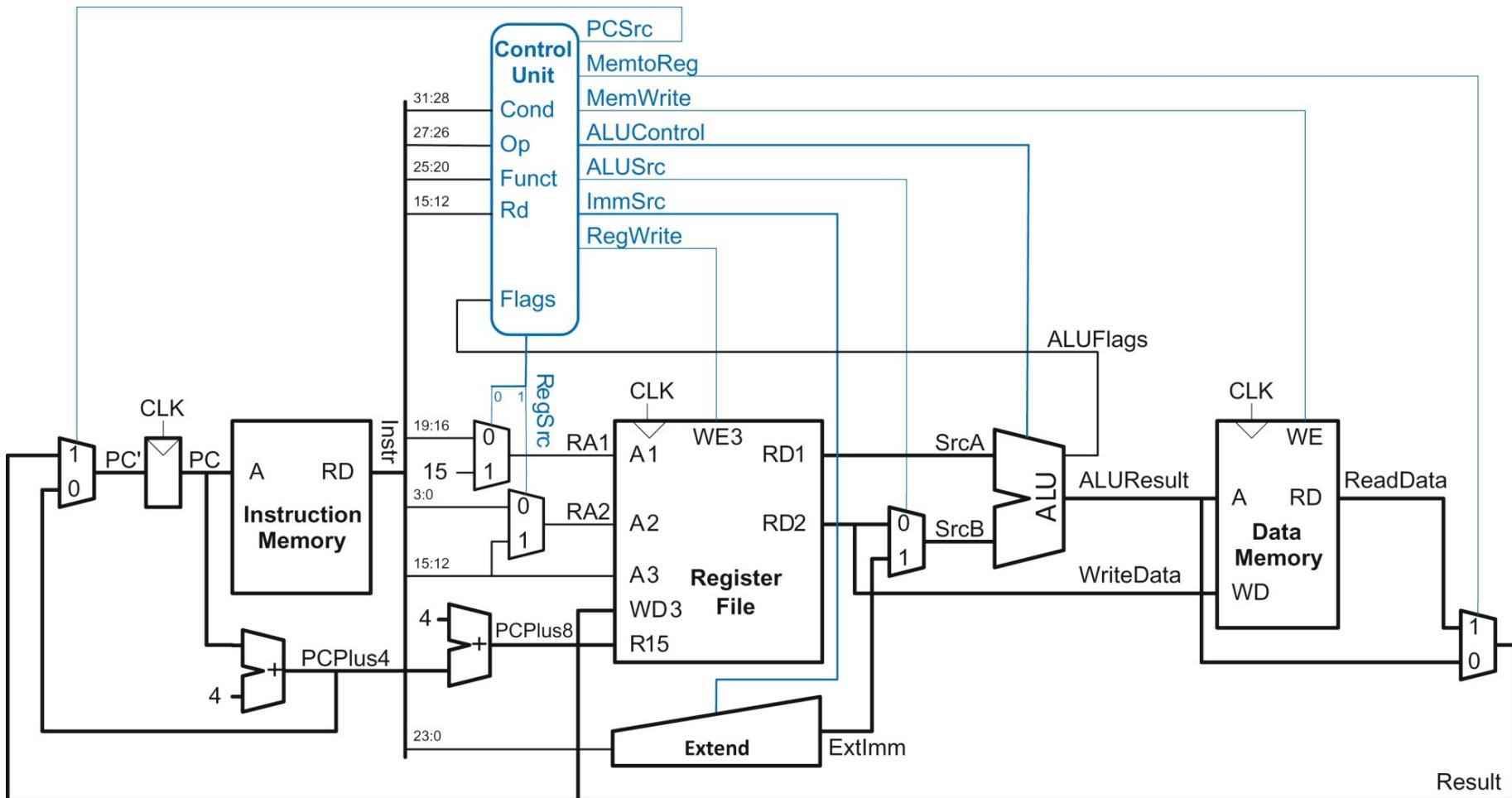
Operation of the Extend Block

Each of the three instruction formats interpret the immediate field differently

- $\text{ImmSrc}_{1:0}$ is the 2-bit control signal input to the extend block

ImmSrc_{1:0}	ExtImm	Description
00	{24'b0, Instr _{7:0} }	Zero-extended <i>imm8</i>
01	{20'b0, Instr _{11:0} }	Zero-extended <i>imm12</i>
10	{6{Instr ₂₃ }, Instr _{23:0} }	Sign-extended <i>imm24</i>

Datapath w/t Control



Control Unit

- Generate control signals based on instruction fields
 - $\text{Instr}_{31:20}$ (cond)
 - $\text{Instr}_{27:26}$ (op)
 - $\text{Instr}_{25:20}$ (funct)
 - Flags
 - Destination register (*why does the controller needs this?*)
- Controller for “*our microarchitecture*” is purely combinational
- Things to ponder
 - Ensure correct PC update (branch or Rd == R15)
 - Update state (Reg file, Data memory) based on conditional execution

Decoder Truth Table

- Only selected signals are shown in the truth table

Op	Funct ₅	Funct ₀	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
11	X	X	B	1	0	0	1	10	0	X1	0

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



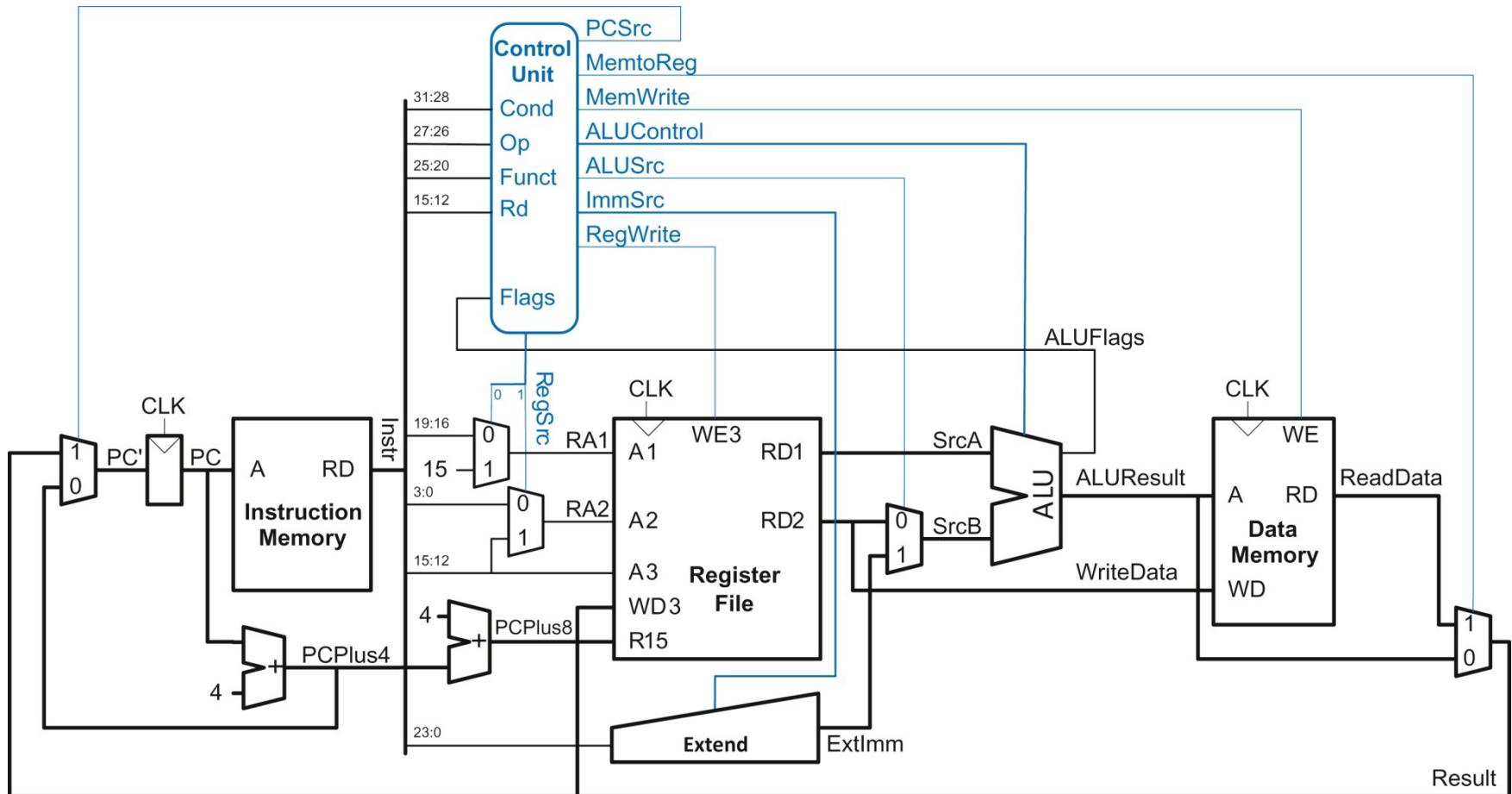
Australian
National
University

Plan: Week 6

Last week: Instruction Set Architecture (specification)

This Week: Microarchitecture (implementation)

Processor Operation: ORR

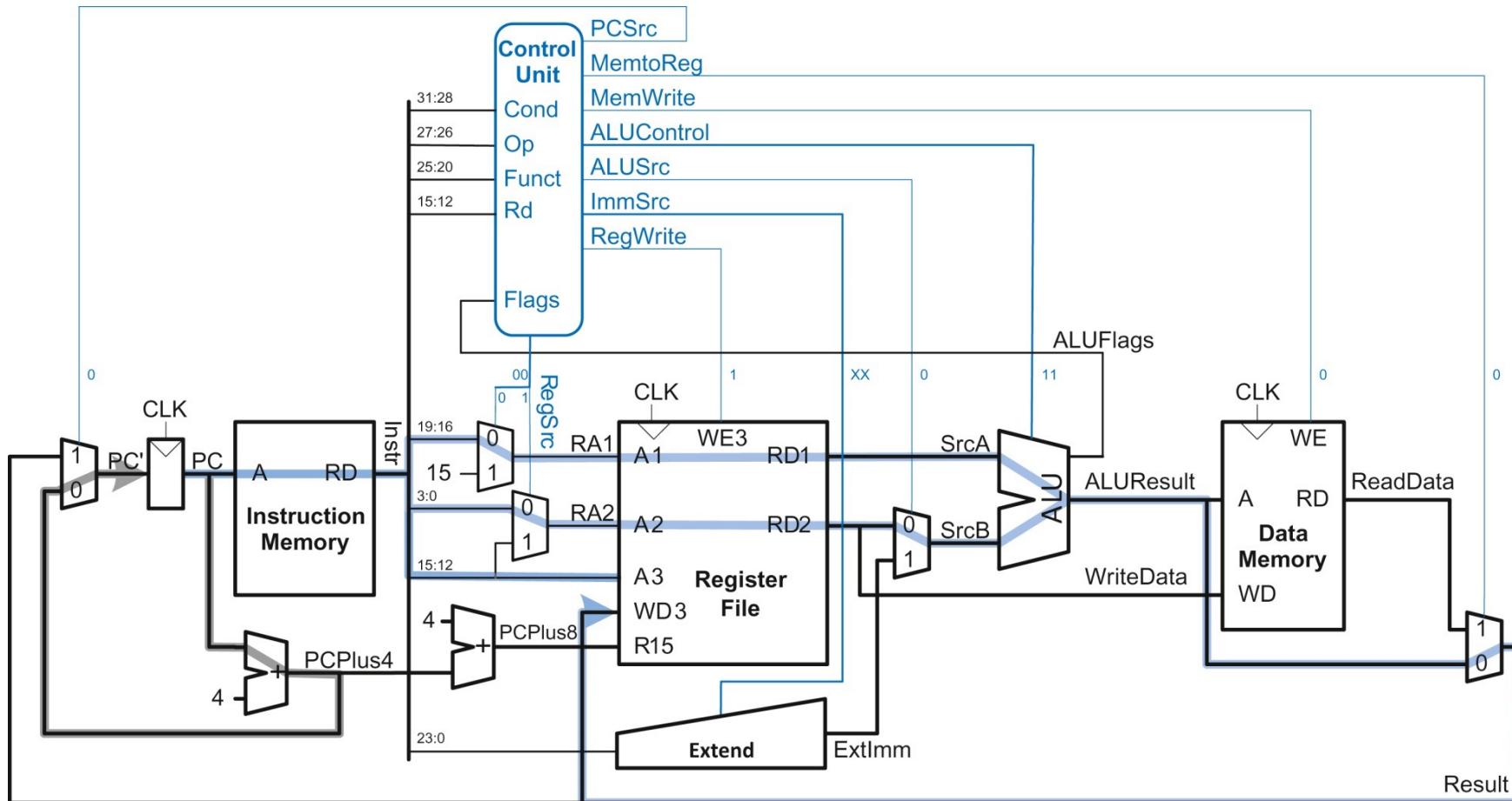


Processor Operation: ORR

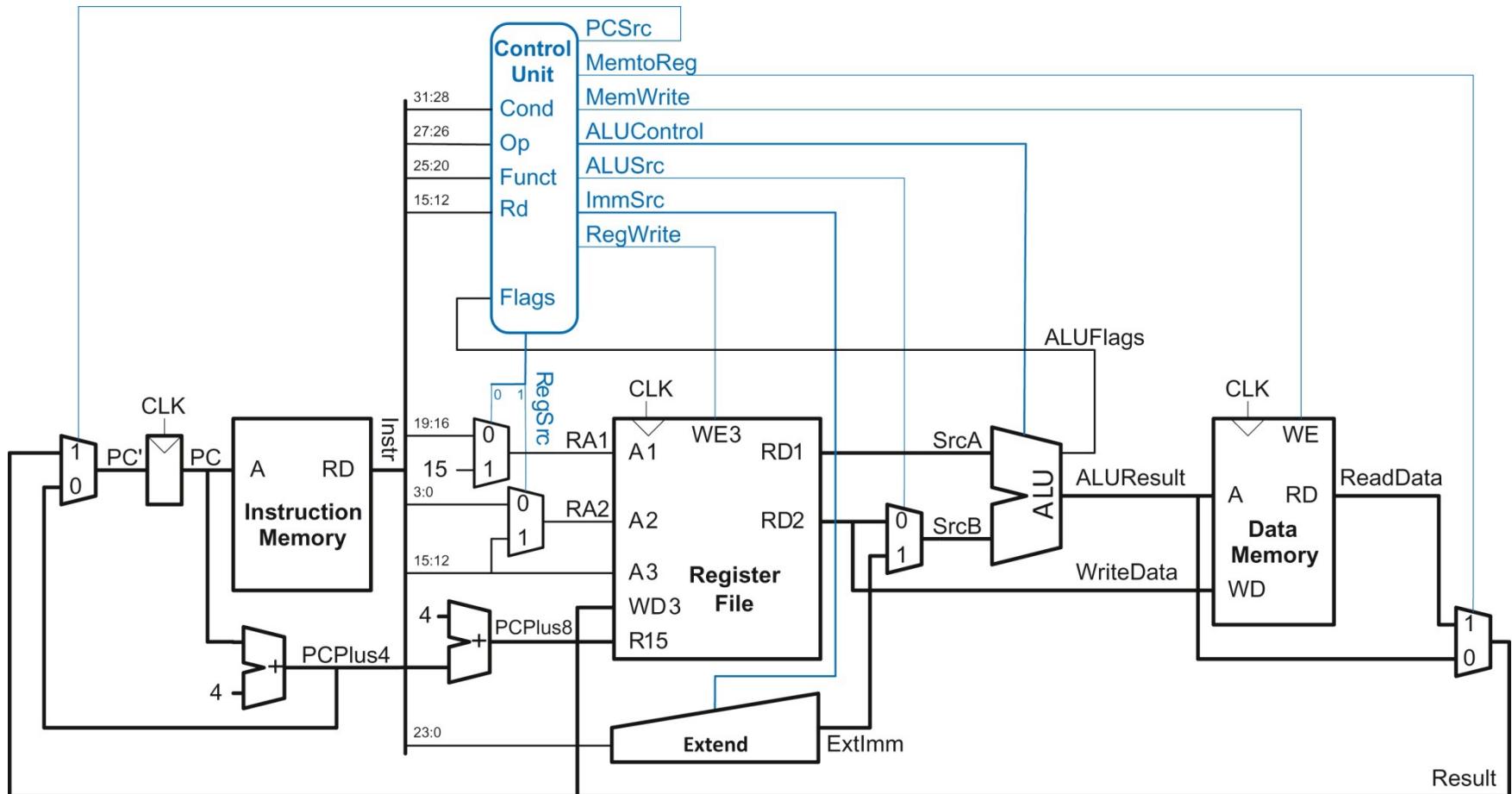
PCSrc	0
MemtoReg	0
MemWrite	0
ALUControl	11
ALUSrc	0
ImmSrc _{0:1}	XX
RegWrite	1
RegSrc _{0:1}	00

ALUControl	Function
00	ADD
01	SUB
10	AND
11	ORR

Processor Operation: ORR



Processor Operation: LDR



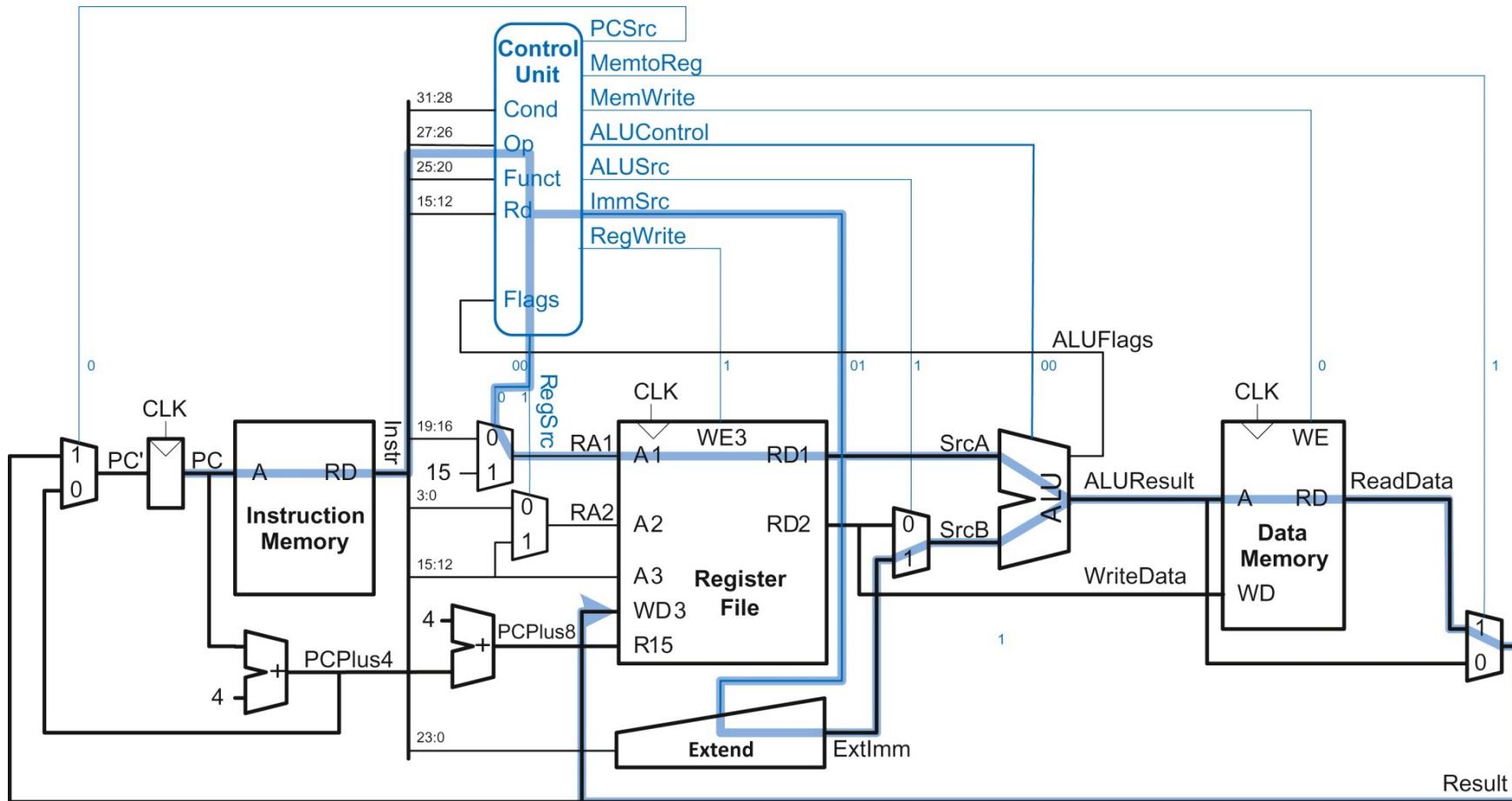
Processor Operation: LDR

PCSrc	0
MemtoReg	1
MemWrite	0
ALUControl	00
ALUSrc	1
ImmSrc _{0:1}	01
RegWrite	1
RegSrc _{0:1}	00

ALUControl	Function
00	ADD
01	SUB
10	AND
11	ORR

ImmSrc _{1:0}	ExtImm	Description
00	{24'b0, Instr _{7:0} }	Zero-extended <i>imm8</i>
01	{20'b0, Instr _{11:0} }	Zero-extended <i>imm12</i>
10	{6{Instr ₂₃ }, Instr _{23:0} }	Sign-extended <i>imm24</i>

Processor Operation: LDR



Whiteboard

Exercise: Generating PCSrc Signal

- PCSrc is **1** when
 - Destination register (Rd) is R15
 - RegW is **1** (ADD/SUB or LDR)
 - Instruction is a branch
- $\text{PCSrc} = ((\text{Rd} == 15) \& \text{RegW}) | \text{Branch}$
 - Assuming the control unit generates a signal called Branch when opcode is 10 (B or BL)
- **Important:** Be careful to take conditional execution into account in the lab task and assignment!

Critical Path Analysis

- Each instruction in our CPU takes one clock cycle
- To determine the clock cycle time requires us to find the critical path
- Different instructions use different resources
 - LDR uses instruction and data memory
 - ADD does not use data memory
 - STR does not write anything back to the register file
- Which instruction is the slowest?
 - Let's go back to the figures and find out!

Elements of Critical Path

Parameter	Description
t_{pcq_PC}	PC clock-to-Q delay
t_{mem}	Memory read
t_{dec}	Decoder propagation delay
t_{mux}	Multiplexer delay
t_{RFread}	Register file read
t_{ext}	Extension block delay
t_{ALU}	ALU delay
$t_{RFsetup}$	Set up RF for write (next cycle)

Critical Path: LDR

$$T_c = t_{\text{pcq_PC}} + t_{\text{mem}} + t_{\text{dec}} + \max[t_{\text{mux}} + t_{\text{RFread}}, t_{\text{ext}} + t_{\text{mux}}] + t_{\text{ALU}} \\ + t_{\text{mem}} + t_{\text{mux}} + t_{\text{RFsetup}}$$

- Memories & register files slower than combinational logic
 - Therefore, $t_{\text{mux}} + t_{\text{RFread}} \gg t_{\text{ext}} + t_{\text{mux}}$

Final Equation

$$T_c = t_{\text{pcq_PC}} + 2t_{\text{mem}} + t_{\text{dec}} + t_{\text{RFread}} + t_{\text{ALU}} + 2t_{\text{mux}} + t_{\text{RFsetup}}$$

Critical Path: DP-R

$$T_c = t_{\text{pcq_PC}} + t_{\text{mem}} + t_{\text{dec}} + t_{\text{mux}} + t_{\text{RFread}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} \\ + t_{\text{RFsetup}}$$

Final Equation

$$T_c = t_{\text{pcq_PC}} + t_{\text{mem}} + t_{\text{dec}} + t_{\text{RFread}} + t_{\text{ALU}} + 3t_{\text{mux}} + t_{\text{RFsetup}}$$

Critical Path Analysis

- Different instructions have different critical paths
 - LDR is the slowest instruction
 - DP-R and B have shorter critical paths because they do not need to access data memory
- Single-cycle processor is a synchronous sequential circuit
 - Clock period is constant and long enough to accommodate the slowest instruction
- The numerical values of different variables in the critical path equation depend on the specific technology

Single-Cycle Microarchitecture

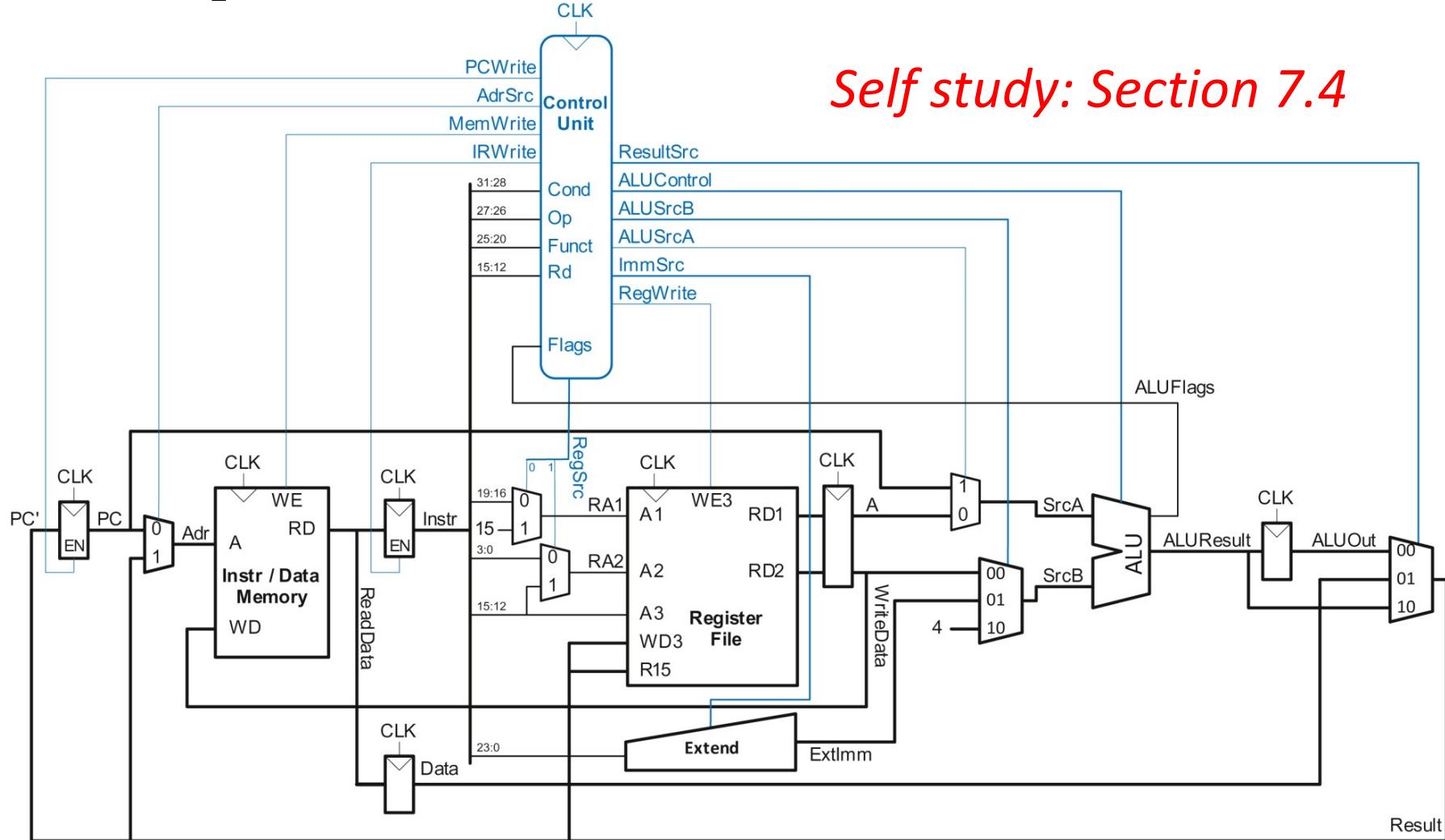
- Single-cycle microarchitecture
 - Execute the entire instruction in a single cycle
- Drawbacks
 - Separate memories for instructions and data
 - Cycle time is determined by the slowest instruction
 - Requires three adders (expensive)

Multicycle Microarchitecture

- Break the instruction into shorter steps
 - Access memory or use ALU in one step
 - Read instruction in one step and data in another step
 - Different instructions take different # steps or cycles
- Advantages
 - Some instructions finish faster than others
 - Needs one adder and one memory
- Drawbacks
 - Decoder complexity (decoder is an FSM)

Multicycle uarch

Self study: Section 7.4



Architectural State

- Modern systems run many applications at a time
- The operating system *interrupts* a program and runs another one, before restarting the interrupted program
- **Architectural state:** State needed to safely restart the program if it is interrupted by the operating system
- What state do we need to save and restore in ARM architecture?
 - 16 registers
 - Status register

Microarchitectural State

- Any other state in the CPU is called *microarchitectural* state or *non-architectural* state
 - This state is not visible to assembly programs
 - Programmers are unaware of microarchitectural state
- **Note:** Memory is part of the architectural state but typically we do not consider memory part of CPU
 - Memory and storage is part of the computer system

Pipelined Microarchitecture

- Multicycle microarchitecture
 - At any time, only one instruction is in execution
- Pipelined microarchitecture
- Pipelining divides the single-cycle CPU into stages
 - Each instruction goes through many stages
 - In each clock cycle, multiple instructions are active
 - One instruction can be in Fetch stage, another one can be in decode stage,
- Pipelining results in a shorter clock cycle time

Pipeline Stages

- What steps/stages do an LDR instruction goes through?
 - Fetch
 - Decode (Register Read)
 - Execute (ALU)
 - Memory Access
 - Writeback

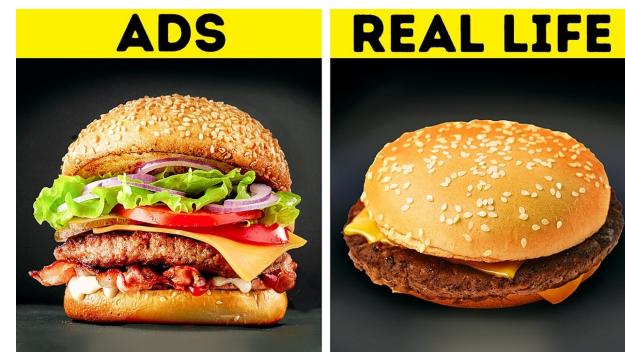
Pipeline Stages

- What steps/stages do an ADD instruction goes through?
 - Fetch
 - Decode (Register Read)
 - Execute (ALU)
 - ~~Memory Access~~
 - Writeback
- More details: We will return to pipelining details after a few lectures!

Performance Analysis

- We want the fastest (*best performing*) computer for a task
 - How should we measure and report performance?
- Which metric is *fair* for comparing two computers?
 - # Instructions
 - Clock frequency
 - Cycles per Instruction (**CPI**)
 - # Cores

*Important to understand the **true**,
gimmick-free measure of computer
performance*



Instructions

- RISC computer
 - Many more simple instructions
 - Simple hardware means smaller clock cycle
- CISC computer
 - Small number of instructions
 - Complex hardware means larger clock cycle

Bottomline: *Number of instructions alone is not a good metric for quantifying the performance of application A*

Clock Frequency

- Consider the two scenarios
 - Computer A has a faster clock than computer B, but A executes many more instructions than B
 - Computer A has a faster clock than computer B, but takes multiple cycles to finish/execute a single instruction
- Is A faster than B?

Bottomline: *Clock frequency alone is not a good metric for quantifying the performance of application A*

CPI

- Cycles per instruction
 - Ratio of # cycles to # instructions
- A program has 10 instructions. Each instruction takes one cycle
 - Instructions = 10, Cycles = 10, CPI = 1
- A program has 10 instructions. Two out of 10 instructions take two cycles
 - Instructions = 10, Cycles = 14, CPI = 1.4
- The inverse of CPI is called IPC
 - Instructions per cycle
 - For the above examples, IPC is 1 and 0.7

CPI

- How can each instruction take multiple cycles?
 - Multi-cycle CPU
 - Memory accesses take more than one cycle
- On most computers, LDR takes variable # cycles because
 - Data may be present in faster (SRAM) memory called CPU cache
 - Or it may be present in main memory which is much slower

CPI

- Two computers A and B have the same CPI for a specific program. Is there performance equivalent?
 - We need to know the cycle time
 - We need to know the # instructions

Bottomline: *CPI alone is not a good metric for quantifying the performance of application A*

Execution Time

- The time it takes for a program to execute from start to finish is the only true measure of performance

$$\text{Execution time} = (\#\text{instructions}) \left(\frac{\text{cycles}}{\text{instruction}} \right) \left(\frac{\text{seconds}}{\text{cycle}} \right)$$

- seconds per cycle = cycle time
- Execution time is measured in seconds
- Golden metric for quantifying computer performance!

Execution Time

$$\text{Execution time} = (\text{\#instructions}) \left(\frac{\text{cycles}}{\text{instruction}} \right) \left(\frac{\text{seconds}}{\text{cycle}} \right)$$

instructions

- Depends on the ISA, skill of programmer, compiler, algorithm

cycles per instruction

- Depends on the microarchitecture esp. memory system

seconds per cycle

- critical path, circuit technology, type of adders, gate-level details

Exercise: Perf Analysis

- Find the time it takes to execute a program with 100 billion instructions on a single-cycle CPU in 16 nm CMOS manufacturing process. See the table for delays of logic elements.

Parameter	Delay (ps)
t_{pcq_PC}	40
t_{mem}	200
t_{dec}	70
t_{mux}	25
t_{RFread}	100
t_{ALU}	120
$t_{RFsetup}$	60

$$T_c = t_{pcq_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$$

Measurement Methodology

- The **good practice**: Take a program of interest and measure its execution time
- The **better practice**: Take a collection of programs like the programs of interest, and measure their performance
 - You do not have the program yet
 - Some one else is measuring the performance independently
- This collection of programs is called a ***benchmark suite***
 - Dhrystone and CoreMark (embedded systems)
 - SPEC (Standard Performance Evaluation Corporation)
 - SPEC is standard suite for high-performance processors

For Loop in C

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1) {  
    sum = sum + i;  
}
```

- The variable “i” is called the loop index
- $i = 0$: index initialization
- $i < 10$: loop termination condition
- $i = i + 1$: loop advancement

For Loop: C to Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1) {  
    sum = sum + i;  
}
```

*check termination condition
to break out of the loop if
condition is met*

*keep iterating by
branching back*

ARM Assembly code

; $R0 = i, R1 = sum$

FOR	MOV	R1,	#0
	MOV	R0,	#0
	CMP	R0,	#10
	BGE	DONE	
	ADD	R1,	R1, R0
	ADD	R0,	R0, #1
	B	FOR	
DONE			

For Loop: Perf Analysis

ARM Assembly code

; $R0 = i, R1 = sum$

```
        MOV    R1,    #0
        MOV    R0,    #0
FOR
        CMP    R0,    #10
        BGE    DONE
        ADD    R1,    R1,    R0
        ADD    R0,    R0,    #1
        B      FOR
DONE
```

- The clock cycle time T_c is 840 ps
- Find the time it takes to execute the for loop
 - # instructions = ?
 - CPI = 1
 - Execution time = ?

Alternative For Loop

	MOV	R1,	#0
	MOV	R0,	#0
COND			
	CMP	R0,	#10
	BLT	LOOP	
	B	DONE	
LOOP			
	ADD	R1,	R1,
	ADD	R0,	R0,
	B	COND	
DONE			

- We can implement the for loop in a different way
- Find the time it takes to execute the for loop again
 - # instructions = ?
 - Execution time = ?

Bottom line: Execution time depends on how we write code and microarchitecture details

While Loop in C

C code:

```
int pow = 1;  
int x = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

- For loop: iterate N times
- While loop: Iterate until a condition is not met

While Loop in C

C code:

```
int pow = 1;  
int x = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

ARM Assembly code

; $R0 = \text{pow}$, $R1 = x$

```
MOV R0, #1  
MOV R1, #0  
WHILE  
CMP R0, #128  
BEQ DONE  
LSL R0, R0, #1  
ADD R1, R1, #1  
B WHILE
```

DONE

Exercise

- The clock cycle time T_c is 840 ps
- Find the time it takes to execute the While loop
 - # instructions = ?
 - CPI = 1
 - Execution time = ?

Shift Instructions

- Shift the value in a register left or right, drop bits off the end
 - Logical shift left (**LSL**)
 - Logical shift right (**LSR**)
 - Arithmetic shift right (**ASR**)
 - Rotate right (**ROR**)
- Logical shift: shifts the number to the left or right and fills the empty slots with zero
- Arithmetic shift: on right shifts fill the most significant bits with zero
- Rotate: rotates number in a circle such that empty spots are filled with bits shifted off the other end

Example: Shift Operations

- Immediate shift amount (5-bit immediate)
- Shift amount: 0-31

Source register				
R5	1111 1111	0001 1100	0001 0000	1110 0111
Assembly Code				Result
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011 1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111 1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010 0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111 1111 1000

Shifts: Machine Representation

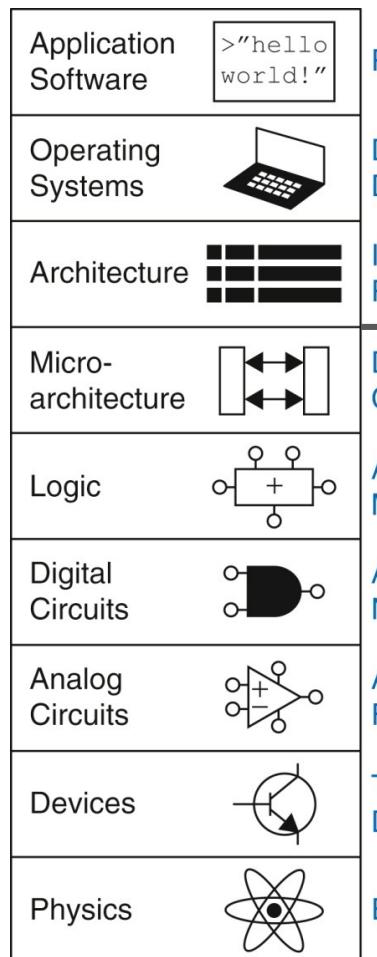
	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
DP-R	cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

Shift Instructions

	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
	cond	00	0	cmd	S	Rn	Rd	shamt5	sh	0	Rm

- cmd = 1101
- sh = 00 (LSL), 01 (LSR), 10 (ASR), 11 (ROR)
- Rn = 0
- shamt5 = 5-bit shift amount

Big Picture



Programs

Device
Drivers

Instructions
Registers

Datapaths
Controllers

Adders
Memories

AND Gates
NOT Gates

Amplifiers
Filters

Transistors
Diodes

Electrons

C Program

Assembly

Machine
Code

compiler

assembler

Software

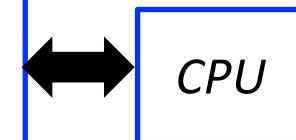
Hardware

*ISA is the boundary
(Contract)*



01010010
10101010
10101001
10000011

Memory



*Instructions stored as 0's and 1's
Fetch, decode, execute
instructions*

Happy Teaching Break!

Remember the Assignment (30%)

Keep the transformation hierarchy
fresh in your memory!

See you in two weeks ☺

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

Plan: Lectures

- C and assembly
 - *Hardware/software interaction*
- Memory and storage devices
 - *How do the devices work?*
 - *How are they exposed to C programs?*
- Hardware optimizations
 - *Caches and virtual memory (memory-side)*
 - *Pipelining (CPU-side)*

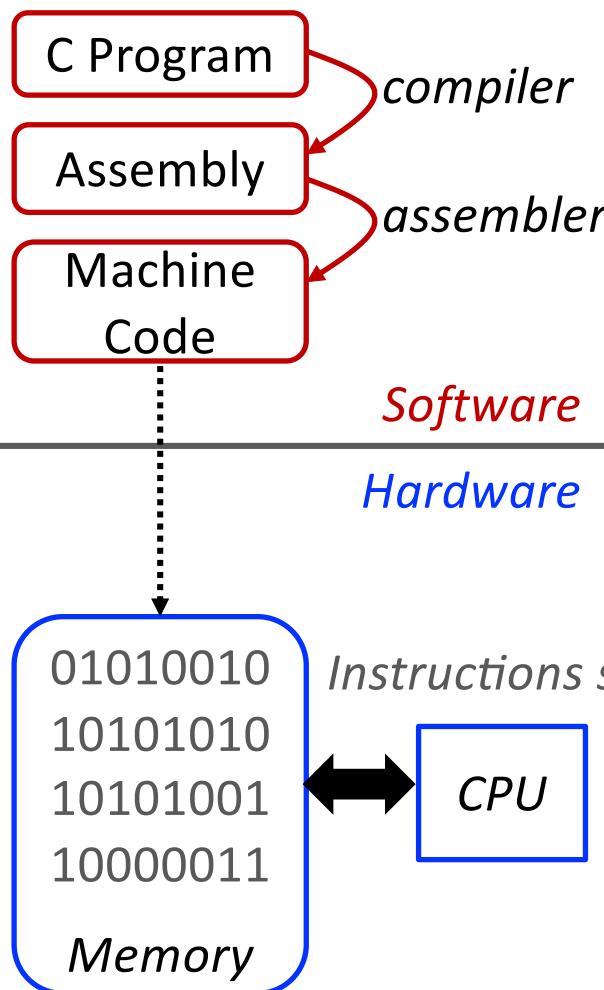
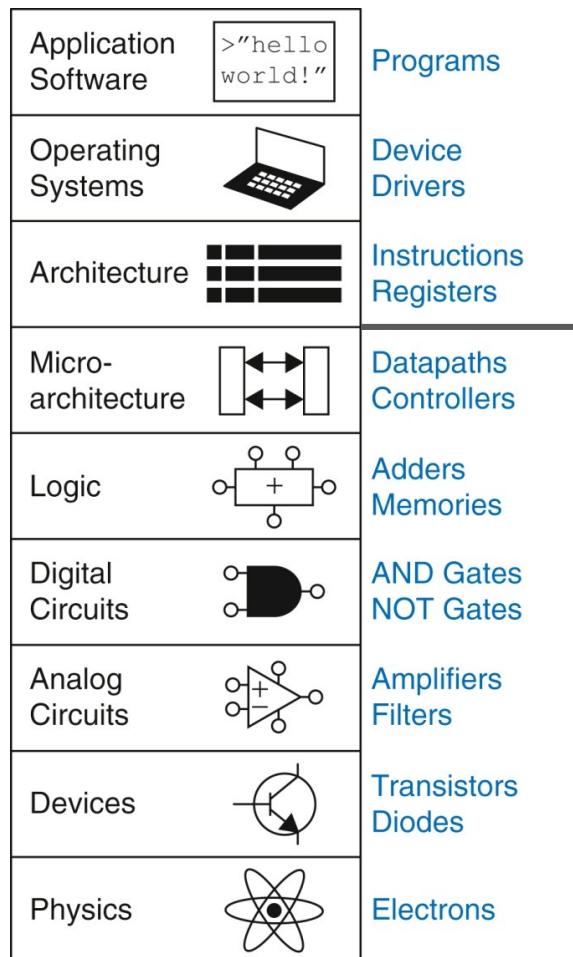
Plan: Labs

- Introduction to C – Part 1 ✓
- Introduction to C – Part 2
 - Control flow, bitwise operations, more pointers, strings
- Data Structures (beyond arrays)
 - Structs, unions, linked lists, read/write file I/O
- Dynamic Memory Allocation (rich topic)
- Assignment 2
 - Problem specification (QuAC CPU model, memory allocator)
 - Your task: Solve the problem in C (knowing assembly helps)
 - If you do Labs 1 – 3 diligently, you will (mostly) nail it!

Hardware/Software Interaction

- *Predominantly Assembly, some C (2 lectures)*
- *Exclusively C (2 – 3 lectures)*

Big Picture



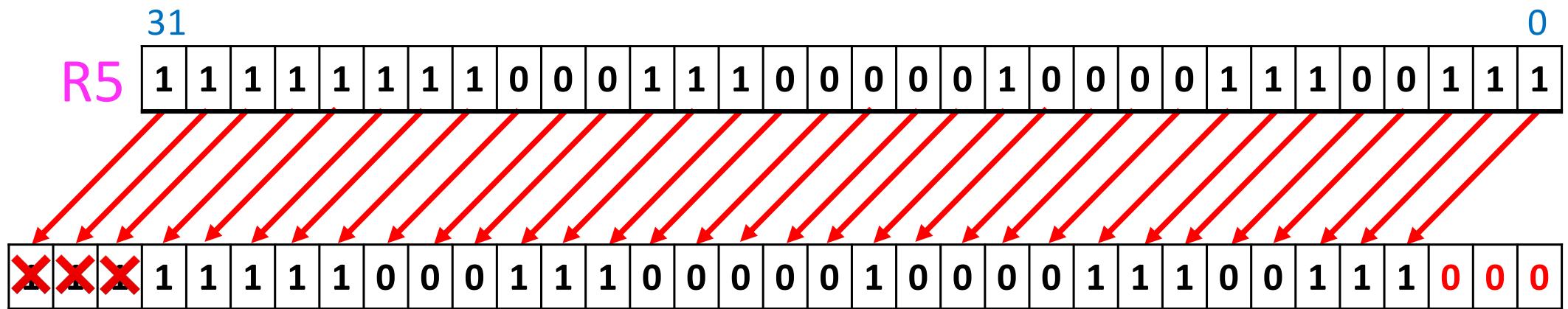
Shift Instructions

- Shift the value in a register left or right, drop bits off the end
 - Logical shift left ([LSL](#))
 - Logical shift right ([LSR](#))
 - Arithmetic shift right ([ASR](#))
 - Rotate right ([ROR](#))
- Logical shift: shifts the number to the left or right and fills the empty slots with zero
- Arithmetic shift: on right shifts fill the most significant bits with the sign bit
- Rotate: rotates number in a circle such that empty spots are filled with bits shifted off the other end

Logical Shift Left (LSL)

ARM Instruction

LSL R0, R5, #3



- Shift all bits left 3 positions, insert three 0's from the left
- Drop the 3 bits from the right

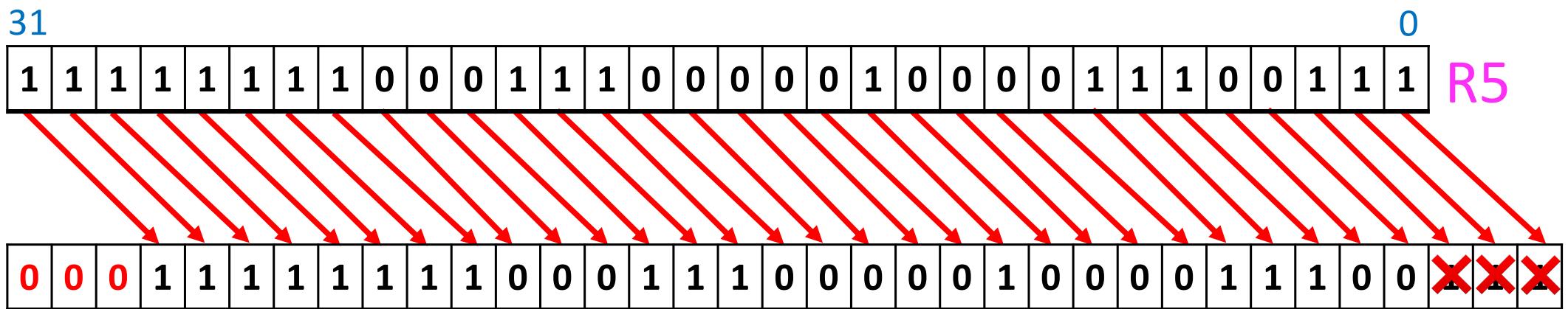
Result

R0 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 1 0 0 0 0 1 1 1 0 0 1 1 1 0 0 0

Logical Shift Right (LSR)

ARM Instruction

LSR R0, R5, #3



- Shift all bits right 3 positions, insert three 0's from the right
- Drop the 3 bits from the left

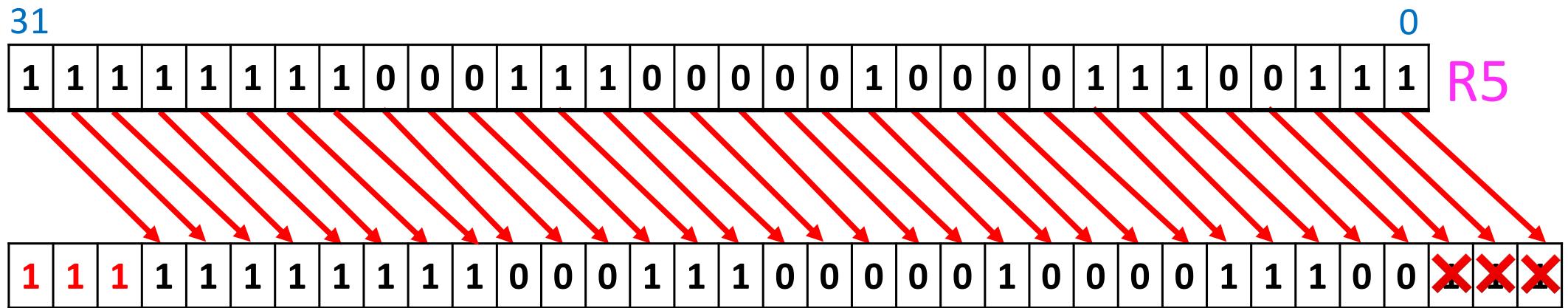
Result

00011111000111000000100000111000 R0

Arithmetic Shift Right (LSR)

ARM Instruction

ASR R0, R5, #3



- Shift all bits right 3 positions, insert three 0's from the right
- Drop the 3 bits from the left

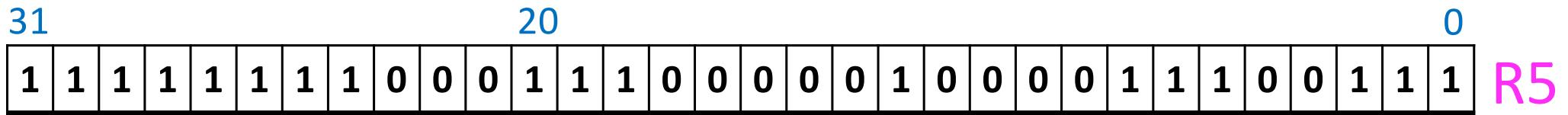
Result

11111111111100011100000010000001110000 R0

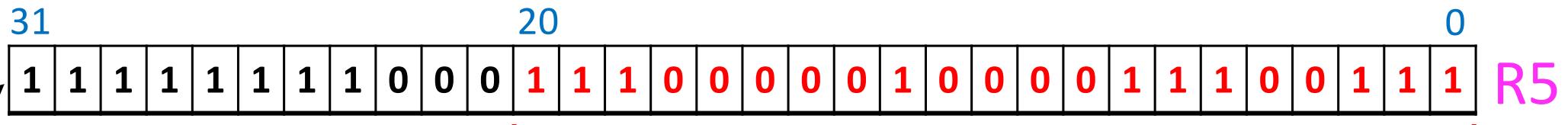
Rotate Right (ROR)

ARM Instruction

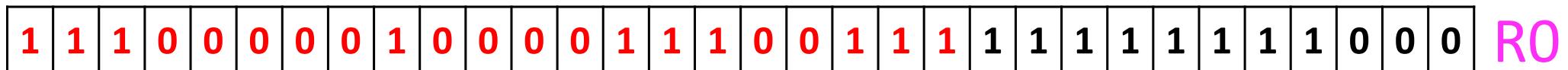
ROR R0, R5, #21



- Do a circular shift
- Right shift by 21 and put back bits that fall off at left end



Result



Shifts: Machine Representation

	31:28	27:26	25	24:21	20	19:16	15:12	11:4	3:0
DP-R	cond	00	0	cmd	S	Rn	Rd	0 0 0 0 0 0 0 0	Rm

Shift Instructions

	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
	cond	00	0	cmd	S	Rn	Rd	shamt5	sh	0	Rm

- cmd = 1101
- sh = 00 (LSL), 01 (LSR), 10 (ASR), 11 (ROR)
- Rn = 0
- shamt5 = 5-bit shift amount

Shifts: Machine Representation

- Format (Src2 = Register)

LSL R0, R5, #3

↓ ↓ ↓
LSL Rd, Rm, shamt5

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
cond	00	0	cmd	S	Rn	Rd	shamt5	sh	0	Rm

Shift Instructions

- **Immediate** shift amount (5-bit immediate)
- Shift amount: 0-31
- Rn is not used
- sh encodes the type of shift
- ARM also has instructions with shift amount held in a register

LSL R4, R8, R6

ROR R5, R8, R6

31:28 27:26 25 24:21 20 19:16 15:12 11:8 7 6:5 4 3:0

cond	00	0	cmd	S	Rn	Rd	Rs	0	sh	1	Rm
------	----	---	-----	---	----	----	----	---	----	---	----

Shift Instructions

- Having dedicated instructions for shift operations is useful for systems programming
- Bit masks are a common requirement in low-level hardware resource management
- Code that manages network protocols or file formats
- Anything related to compression/decompression or packing/unpacking of information
- Rotation is used in cyclic codes (cryptography, compression)

Control Flow

- In real programs, the *order* in which statements execute is not always sequential (one after the other)
- Decisions and iterating the same task repeatedly is common
 - `if/else` ✓
 - `switch`
 - `for` and `while`
- How can we write these statements in assembly?
 - Performance analysis
 - Evaluate alternatives

For Loop in C: Sum

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1) {  
    sum = sum + i;  
}
```

C code:

```
int i;  
int sum = 0;  
  
for (i = 9; i >= 0; i = i - 1) {  
    sum = sum + i;  
}
```

Decrement version

- The variable “i” is called the loop index/counter
- $i = 0$: index initialization
- $i < 10$: loop termination condition
- $i = i + 1$: loop advancement

Sum: Assembly

C code:

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1) {  
    sum = sum + i;  
}
```

*check termination condition
to break out of the loop if
condition is met*

*keep iterating by
branching back*

ARM Assembly code

; $R0 = i, R1 = sum$

FOR	MOV	R1,	#0
	MOV	R0,	#0
	CMP	R0,	#10
	BGE	DONE	
	ADD	R1,	R1, R0
	ADD	R0,	R0, #1
	B	FOR	
DONE			

Sum: Perf Analysis

ARM Assembly code

; $R0 = i, R1 = sum$

```
        MOV    R1,    #0
        MOV    R0,    #0
FOR
        CMP    R0,    #10
        BGE    DONE
        ADD    R1,    R1,    R0
        ADD    R0,    R0,    #1
        B      FOR
DONE
```

How long does it take to execute the loop (*frequency = 1 GHz, CPI = 1*)

- # instructions = ?
- Clock cycle time, T_c = ?
- Execution time = ?

Sum: Alternative Approach

	MOV	R1,	#0
	MOV	R0,	#0
COND			
	CMP	R0,	#10
	BLT	FOR	
	B	DONE	
FOR			
	ADD	R1,	R1, R0
	ADD	R0,	R0, #1
	B	COND	
DONE			

- More faithfully follow the for loop semantics in C
- Use BLT instead of BGE

How long does it take to execute the loop (*frequency = 1 GHz, CPI = 1*)

- Instruction count = ?
- Clock cycle time, T_c = ?
- Execution time = ?

Sum: Decrement Version

C code:

```
int i;  
int sum = 0;  
  
for (i = 9; i != 0; i = i - 1) {  
    sum = sum + i;  
}
```

ARM Assembly code

; $R0 = i, R1 = sum$

FOR	MOV	R1,	#0	
	MOV	R0,	#9	
	ADD	R1,	R1,	R0
	SUBS	R0,	R0,	#1
DONE	BNE	FOR		

sum = sum + 1

i = i - 1 and set flags

Saves 2 instructions per iteration:

- Decrement loop variable & compare: SUBS R0, R0, #1
- Only 1 branch – instead of 2

Exercise

- Find the time it takes to execute the loop now if the clock cycle time is 1 ns

Lessons

Execution time depends on how we write code and microarchitecture details

Always make the common case fast!

Note: *Eliminating a branch is always desirable in pipelined processors because the CPU needs to wait for the branch to finish execution in order to fetch the next instruction*

While Loop in C

- For loop iterate N times
 - Used when N is known in advance
- While loop
 - Iterate until the *controlling condition* is false
- Determine x such that $2^x = 128$

C code:

```
int pow = 1;  
int x = 0;
```

```
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

Two Interesting While Loops

C code:

```
while (1) {  
    // iterates forever  
}
```

```
while (0) {  
    // iterates 0 times  
}
```

While Loop: C and Assembly

C code:

```
int pow = 1;  
int x = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

Determine x such that
 $2^x = 128$

ARM Assembly code

; $R0 = pow, R1 = x$

```
MOV   R0, #1  
MOV   R1, #0  
WHILE  
CMP   R0, #128  
BEQ   DONE  
LSL   R0, R0, #1  
ADD   R1, R1, #1  
B     WHILE  
DONE
```

Exercise

- Find the time it takes to execute the While loop if the clock cycle time is 1 ns.
- Write `sum` as a while loop and find the time it takes to execute the resulting loop if all CPU parameters are the same as before.

switch/case Statement

C code:

```
switch(button) {  
    case 1: atm = 20; break;  
    case 2: atm = 50; break;  
    case 3: atm = 100; break;  
    default; atm = 0;  
}
```



C code for if...else ladder:

```
if (button == 1)  
    atm = 20;  
else if (button == 2)  
    atm = 50;  
else if (button == 3)  
    atm = 100;  
else  
    atm = 0;
```

- Execute one of several blocks of code depending on the condition and *break* out of the entire switch block
- If no conditions are met, the *default* block is executed

switch/case Statement

C code:

```
switch(button) {  
    case 1: atm = 20; break;  
    case 2: atm = 50; break;  
    case 3: atm = 100; break;  
    default: atm = 0;  
}
```

ARM Assembly code

; $R0 = \text{button}$, $R1 = \text{atm}$

CMP R0, #1	MOVEQ R1, #20	BEQ DONE
CMP R0, #2	MOVEQ R1, #50	BEQ DONE
CMP R0, #3	MOVEQ R1, #50	BEQ DONE
MOV R1, #0		

DONE

Arrays

- Arrays contain a collection of similarly typed elements
- Elements are stored contiguously in memory

int is 4 bytes on most architectures

C code:

```
int marks[5] = {19, 10, 8, 17, 9};  
  
int a = marks[0];  
  
int b = 2;  
  
marks[3] = b;
```

Address	Data	Index	Element
...
00000010	9	4	marks[4]
0000000C	17	3	marks[3]
00000008	8	2	marks[2]
00000004	10	1	marks[1]
00000000	19	0	marks[0]

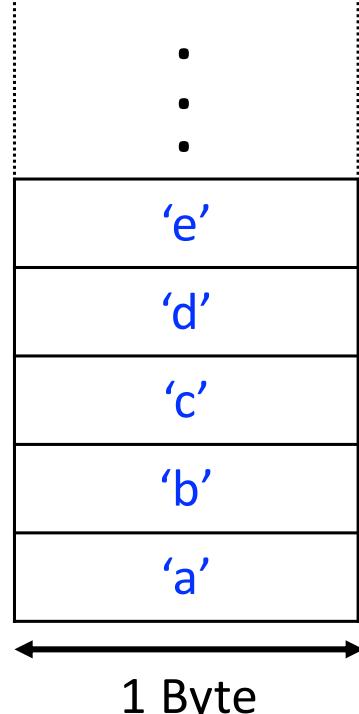
4 Bytes

Array of Characters

- Array of **characters** (**char** is a data type in C)
- **char** is used for representing characters

char is always 1 byte	Address	Data	Index	Element
C code:
char alphas[5] = {'a', 'b', 'c', 'd', 'e'};	00000004	'e'	4	alphas[4]
	00000003	'd'	3	alphas[3]
	00000002	'c'	2	alphas[2]
	00000001	'b'	1	alphas[1]
	00000000	'a'	0	alphas[0]

1 Byte



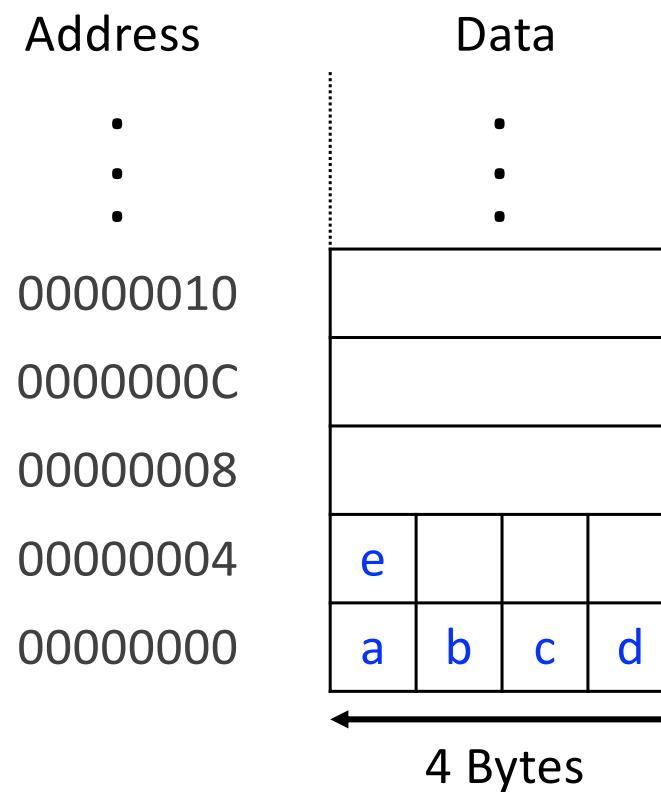
Array of Characters

- Array of **characters** (char is a data type in C)
- char is used for representing characters

char is always 1 byte

C code:

```
char alphas[5] = {'a', 'b', 'c', 'd', 'e'};
```



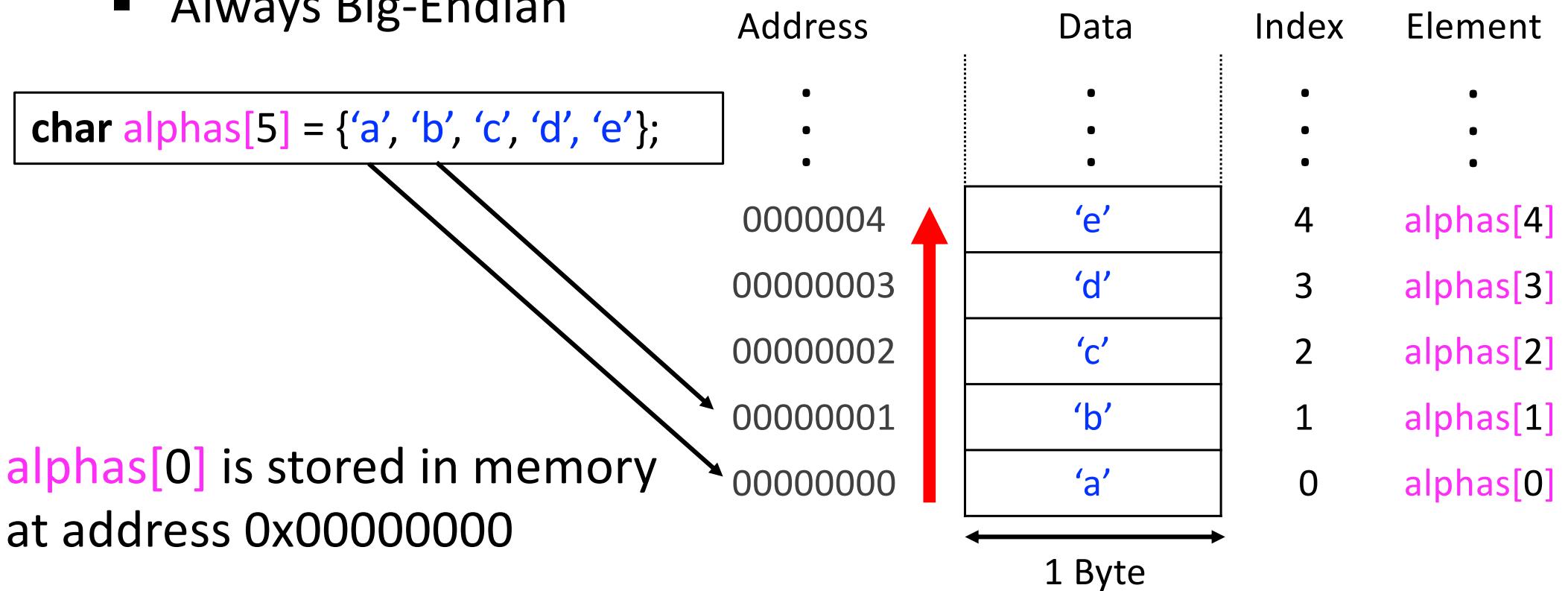
Endianness

For large objects (greater than 1 byte), byte order in memory matters: *Which byte of a 4-byte int is stored at the lowest address?*

- **Little Endian:** Little end (**LSB**) stored first (at lowest address)
 - Intel x86
- **Big Endian:** Big end (**MSB**) stored first,
 - SPARC, Motorola processors
- ARM is bi-endian (supports both)

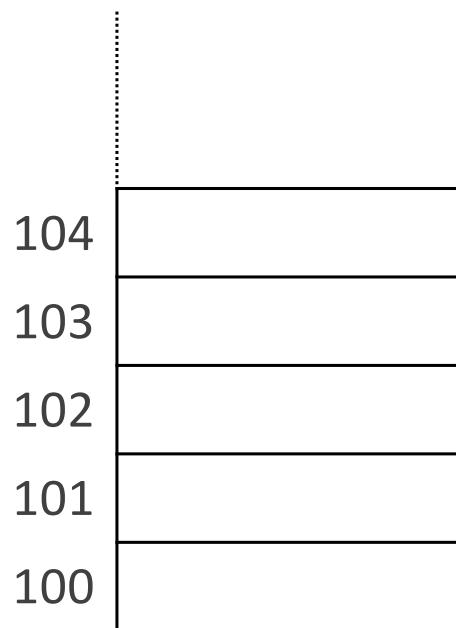
Characters and Endianness

- Characters are 1-byte each
- There is no ambiguity in which byte to store first
 - Always Big-Endian



Integers and Endianess

- A 32-bit integer is stored at memory address 100
- Stored in locations: 100, 101, 102, 103
- Which part of the 32-bit value is stored first?

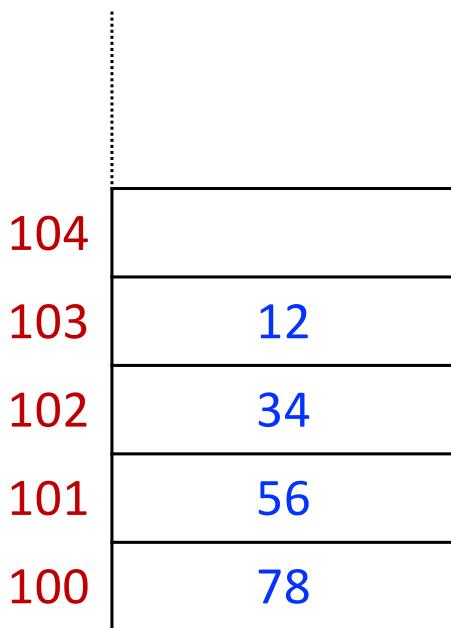


00010010001101000101011001111000

Hex 12 34 56 78

Integers and Endianess

- A 32-bit integer is stored at memory address 100
- Stored in locations: 100, 101, 102, 103
- Which part of the 32-bit value is stored first?



00010010001101000101011001111000

Hex 12 34 56 78

Little Endian: 78 at location 100

Integers and Endianess

- A 32-bit integer is stored at memory address 100
- Stored in locations: 100, 101, 102, 103
- Which part of the 32-bit value is stored first?

Address	Data
104	
103	78
102	56
101	34
100	12

00010010001101000101011001111000

Hex 12 34 56 78

Big Endian: 12 at location 100

Endianness: Pros

- Little Endian
 - Easy to create small values from large values
 - Previous example: Read byte at address 100
 - On Big Endian, add 4 to 100, then read byte to find out
- Big Endian
 - Easy to test sign and range of a value

QuAC architecture in labs evades the entire issue of endianness with word-addressable memory

Array Sum

- Example to illustrate how instructions are picked for ISAs
- Add a constant 10 to each element of the scores array

C code:

```
int i;  
int scores[200];  
// initialization code  
...  
for (i = 0; i<200; i++)  
    scores[i] = scores[i] + 10;
```

Assembly code:

```
; R0 = array base address, R1 = i  
MOV  R0, #0x14000000  
MOV  R1, 0  
LOOP  
    CMP  R1, #200  
    BGE L3  
    LSL  R2, R1, #2  
    LDR  R3, [R0, R2]  
    ADD  R3, R3, #10  
    STR  R3, [R0, R2]  
    ADD  R1, R1, #1  
    B    LOOP  
L3
```

- R0 = base address
- i = 0
- i < 200?
- if not, exit loop
- R2 = i*4
- R3 = scores[i]
- R3 = scores[i] + 10
- scores[i] += 10
- i = i + 1
- repeat loop

LDR with register as offset

- New LDR variant (LDR with register as offset)

LDR R3, [R0, R2]
↓ ↓ ↘
dest base offset

- Very common to load from memory with base + offset addressing, so there is an instruction for that
- R2 is called the index register

Condensing Array Sum – 1

- LSL/LDR combo often used in tandem in array traversals
 - There is support for that in the ISA
- Eliminates LSL instruction

*ARM has an instruction
that scales index reg. R1*

LDR R3, [R0, R1, LSL #2]

Left shift is a
multiply by 2

Memory address = R0 + (R1 * 4)

Assembly code:

```
; R0 = array base address, R1 = i
MOV R0, #0x14000000
MOV R1, 0
LOOP
    CMP R1, #200
    BGE L3
    LSL R2, R1, #2
    LDR R3, [R0, R2]
    ADD R3, R3, #10
    STR R3, [R0, R2]
    ADD R1, R1, #1
    B LOOP
L3
```

- R0 = base address
- i = 0
- i < 200?
- if not, exit loop
- R2 = i*4
- R3 = scores[i]
- R3 = scores[i] + 10
- scores[i] += 10
- i = i + 1
- repeat loop

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

Recap: Array Sum

Add 10 to each element of the 200-element scores array

C code:

```
int i;
int scores[200];
// initialization code not
// shown
...
for (i = 0; i<200; i++)
    scores[i] = scores[i] + 10;
```

Array Sum

Add 10 to each element of the 200-element scores array

```
C code:  
int i;  
int scores[200];  
// initialization code not  
//shown  
...  
for (i = 0; i<200; i++)  
    scores[i] = scores[i] + 10;
```

	address	data	
	0x14000010	90	scores[4]
	0x1400000C	76	...
	0x14000008	80	scores[2]
	0x14000004	40	scores[1]
base →	0x14000000	100	scores[0]

4 bytes

Showing the scores array in memory

Array Sum

Add 10 to each element of the 200-element scores array

C code:

```
int i;
int scores[200];
// initialization code not
// shown
...
for (i = 0; i<200; i++)
    scores[i] = scores[i] + 10;
```

address

address	data	
0x14000010	90	scores[4]
0x1400000C	76	...
0x14000008	80	scores[2]
0x14000004	40	scores[1]
base → 0x14000000	100	scores[0]

4 bytes

Assembly code:

```
; R0 = array base address
; R1 = i
    MOV R0, #0x14000000
    MOV R1, 0
LOOP
    CMP R1, #200
    BGE L3
    LSL R2, R1, #2
    LDR R3, [R0, R2]
    ADD R3, R3, #10
    STR R3, [R0, R2]
    ADD R1, R1, #1
    B LOOP
L3
```

- R0 = base addr
- i = 0
- i < 200?
- no? exit loop
- i = i + 1
- R3 = scores[i]
- R3 = R3 + 10
- scores[i] += 10
- i = i + 1
- repeat

Showing the scores array in memory

Array Sum

Add 10 to each element of the 200-element scores array

C code:

```
int i;
int scores[200];
// initialization code not
// shown
...
for (i = 0; i<200; i++)
    scores[i] = scores[i] + 10;
```

address

address	data
0x14000010	90
0x1400000C	76
0x14000008	80
0x14000004	40
base → 0x14000000	100

4 bytes

Assembly code:

```
; R0 = array base address
; R1 = i
    MOV R0, #0x14000000
    MOV R1, 0
LOOP
    CMP R1, #200
    BGE L3
    LSL R2, R1, #2
    LDR R3, [R0, R2]
    ADD R3, R3, #10
    STR R3, [R0, R2]
    ADD R1, R1, #1
    B LOOP
L3
```

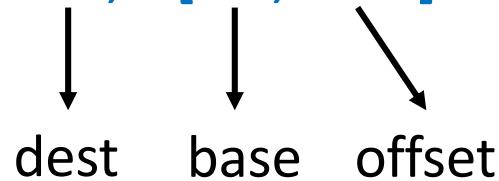
- R0 = base addr
- i = 0
- i < 200?
- no? exit loop
- i = i + 1
- R3 = scores[i]
- R3 = R3 + 10
- scores[i] += 10
- i = i + 1
- repeat

Showing the scores array in memory

Another LDR Variant

- We have seen two ways of specifying the offset so far

LDR R3, [R0, #16]



LDR R3, [R0, R2]



- LSL + LDR combo often used in tandem in array traversals
- ISA supports eliminating the extra LSL instruction

LDR R3, [R0, R1, LSL #2]



- Memory address

- Left shift R1 by 2 (scale R1)
- Add to R0
- Address = R0 + (R1 * 4)

Condensing Array Sum – 1

Add 10 to each element of the 200-element scores array

C code:

```
int i;
int scores[200];
// initialization code not
// shown
...
for (i = 0; i<200; i++)
    scores[i] = scores[i] + 10;
```

address

0x14000010	90
0x1400000C	76
0x14000008	80
0x14000004	40
base → 0x14000000	100

4 bytes

Assembly code:

```
; R0 = array base address
; R1 = i
    MOV R0, #0x14000000
    MOV R1, 0
LOOP
    CMP R1, #200
    BGE L3
    LDR R3, [R0, R1, LSL, #2]
    ADD R3, R3, #10
    STR R3, [R0, R2]
    ADD R1, R1, #1
    B LOOP
L3
```

Showing the scores array in memory

ARM Indexing Modes

- Offset Addressing LDR R0, [R1, R2]
 - *Address is the sum of base register + offset (#20, #-20, -R2)*
 - *Base register is unchanged*
- Pre-indexed Addressing LDR R0, [R1, R2]!
 - *Address is the sum of base register + offset*
 - *Base register is updated with the address*
- Post-index Addressing LDR R0, [R1], R2
 - *Address is the base register*
 - *Base register is updated with the new address only after the memory access*

ARM Indexing Modes

- Offset Addressing LDR R0, [R1, R2]
 - $Address = R1 + R2$
 - $R1 = \text{Unchanged}$
- Pre-indexed Addressing LDR R0, [R1, R2]!
 - $Address = R1 + R2$
 - $R1 = R1 + R2$
- Post-index Addressing LDR R0, [R1], R2
 - $Address = R1$
 - $R1 = R1 + R2$
- *In all cases, offset can be an immediate*

Condensing Array Sum – 2

Add 10 to each element of the 200-element scores array

C code:

```
int i;
int scores[200];
// initialization code not
// shown
...
for (i = 0; i<200; i++)
    scores[i] = scores[i] + 10;
```

address

0x14000010

90

scores[4]

0x1400000C

76

...

0x14000008

80

scores[2]

0x14000004

40

scores[1]

base → 0x14000000

100

scores[0]

4 bytes

Assembly code:

```
; R0 = array base address
; R1 = i
    MOV R0, #0x14000000
    MOV R1, R0, #800
LOOP
    CMP R0, R1
    BGE L3
    LDR R2, [R0]
    ADD R2, R2, #10
    STR R2, [R0], #4
    B LOOP
L3
```

- R0 = base addr
- R1 = base + 800
- end of array?
- yes? exit loop
- R2 = scores[i]
- scores[i] + 10
- store scores[i]
- and R0 = R0 + 4
- repeat loop

Showing the scores array in memory

Condensing Array Sum – 2

Add 10 to each element of the 200-element scores array

Assembly code:

```
; R0 = array base address
; R1 = i
    MOV  R0,  #0x14000000
    MOV  R1,  R0,  #800
LOOP
    CMP  R0,  R1
    BGE L3
    LDR  R2,  [R0]
    ADD  R2,  R2,  #10
    STR  R2,  [R0], #4
    B    LOOP
L3
```

- This version of Array Sum first computes the address of the last byte of the array (#0x14000800)
- Each iteration of LOOP checks if R0 is greater than or equal to #0x14000800
- If so, we are done, so step out of LOOP
- STR R2, [R0], #4
 - Stores R2 at [R0], and after that, adds 4 to R0

Explaining

1. **CMP R0, R1**
 2. **BGE LOOP**
-

- When CPU encounters: **CMP R0, R1**
 - It subtracts **R1** from **R0**
 - It sets the **flags** in the CSPR register
 - No register is updated with the result (no side-effects)
- When CPU encounters: **BGE LOOP**
 - CPU checks the **flags** to establish if **R0 is greater than or equal to R1**
- Some conditions are easy to establish, and others harder
 - **EQ** is easily established by looking at the **zero flag**
 - If the **zero flag** is **1**, it means **(R0 – R1)** is **0**, meaning **R0** and **R1** are equal

Conditional Execution

- Week 5, part 2 lecture
 - Two ways of setting the flags in the CPSR register
 - Conditional execution in general

Bytes and Characters

- Characters on the English keyboard can be encoded in a single byte
- **char** in C is an 8-bit integer under the hood
 - C operators close to the hardware (basic types)
 - No string, list, or other composite types
- ASCII standard is for mapping characters to integer codes
 - Other standards such as Unicode are ASCII supersets
- Need instructions to manipulate bytes!

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Loading/Storing Bytes

- LDRB
 - *Load byte in register, and zero extend to fill the 32 bits*
- LDRSB
 - *Load byte in register, and sign-extend to fill the 32 bits*
- STRB
 - *Store the LSB of the 32-bit integer into the specified byte in memory*
 - *More significant bits of the register are ignored*

Loading/Storing Bytes

- What is in `R1`, `R2`, and `memory` after each of the instruction has executed?
- Assume $R4 = 0$

<i>Byte Address</i>	<i>Data</i>	<i>Registers</i>	
4	...	xx xx xx xx	LDRB R1, [R4, #2]
3	F7	xx xx xx xx	LDRSB R2, [R4, #2]
2	8C	xx xx xx xx	
1	42	11 10 A1 9B	STRB R3, [R4, #3]
0	03		

Loading/Storing Bytes

- What is in `R1`, `R2`, and `memory` after each of the instruction has executed?
- Assume $R4 = 0$

<i>Byte Address</i>	<i>Data</i>	<i>Registers</i>	
4	...	00 00 00 8C	LDRB R1, [R4, #2]
3	9B	FF FF FF 8C	LDRSB R2, [R4, #2]
2	8C		
1	42		
0	03	xx xx xx 9B	STRB R3, [R4, #3]

Strings in C

- A series of characters is a string
 - `char welcome[6] = {'H', 'E', 'L', 'L', 'O', '\0'};`
 - `char welcome[] = "HELLO";`
- Compiler figures out the length
- $5 + 1$ for `'\0'`
- Manually track length (unlike Python)
- Compiler inserts a null terminator `'\0'` automatically
- Need a way to know the end of the string
- C strings are null-terminated

Ex: Manipulating Char Array

C code:

```
char array[10] = "ENGN2219!";
int i;

for (i = 0; i < 10; i = i + 1)
    array[i] = array[i] - 32;
```

Ex: Manipulating Char Array

- Transform the 10-character ASCII string, namely array, from lower case to upper case

C code:

```
char array[10] = "finalexam";
int i;

for (i = 0; i < 10; i = i + 1)
    array[i] = array[i] - 32;
```

Assembly code:

```
; R0 = base addr, R1 = i
    MOV   R0, #0
LOOP
    CMP   R1, #10
    BGE  DONE
    LDRB R2, [R0, R1]
    SUB   R2, R2, #32
    STRB R2, [R0, R1]
    ADD   R1, R1, #1
    B     LOOP
DONE
```

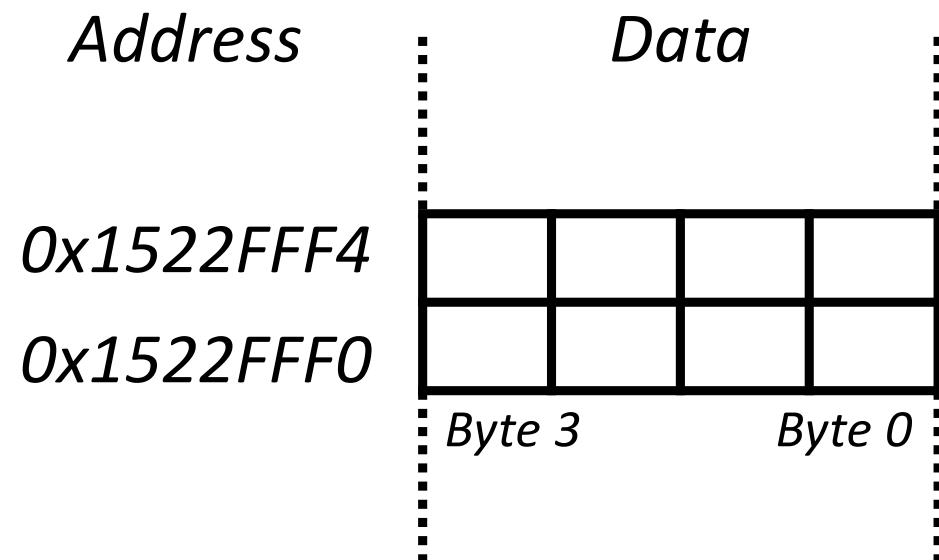
- i = 0
- i < 10?
- if i >= 10, exit
- R2 = array[i]
- subtract 32
- store array[i]
- i = i + 1
- repeat loop

Exercise: Strings in Memory

- Show how “HELLO!” is stored in memory below at address 0x1522FFF0.

ASCII Encoding

H	0x48
E	0x65
L	0x6C
O	0x6F
!	0x21
Null	0x00

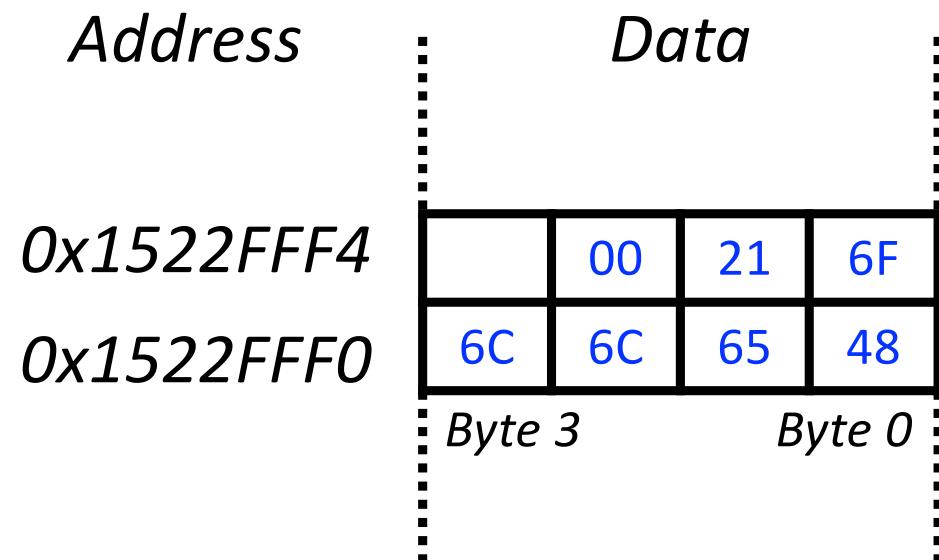


Exercise: Strings in Memory

- Show how “HELLO!” is stored in memory below at address 0x1522FFF0.

ASCII Encoding

H	0x48
E	0x65
L	0x6C
O	0x6F
!	0x21
Null	0x00



Practice

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
MOV R0, #0x60000000      ; R0 = 0x60000000

LDR R1, [R0]              ; R1 = array[0]
LSL R1, R1, #3            ; R1 = R1 << 3 = R1*8
STR R1, [R0]              ; array[0] = R1

LDR R1, [R0, #4]          ; R1 = array[1]
LSL R1, R1, #3            ; R1 = R1 << 3 = R1*8
STR R1, [R0, #4]          ; array[1] = R1
```

Exercise

C Code

```
int array[200];
int i;
for (i=199; i >= 0; i = i - 1)
    array[i] = array[i] * 8;
```

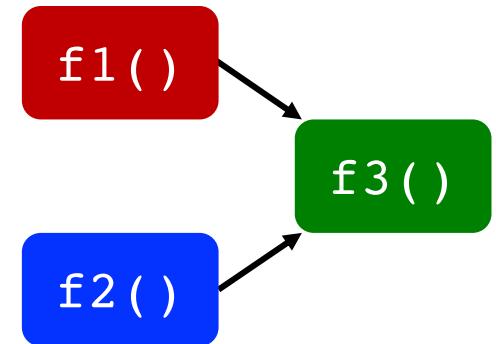
ARM Assembly Code

```
; R0 = array base address, R1 = i
MOV R0, 0x60000000
MOV R1, #199

FOR
    LDR R2, [R0, R1, LSL #2]      ; R2 = array(i)
    LSL R2, R2, #3                ; R2 = R2<<3 = R3*8
    STR R2, [R0, R1, LSL #2]      ; array(i) = R2
    SUBS R0, R0, #1               ; i = i - 1
                                ; and set flags
    BPL FOR                      ; if (i>=0) repeat
loop
```

Functions

- High-level languages offer functions to enable
 - Abstraction & Modularity
 - Code reuse
 - Readability
 - Testability & validation
 - Maintainability
- Functions are also called *procedures* or *subroutines*
- Functions are ubiquitous, encouraging **ISA** support
 - Special jump instructions
 - Special scratch space to store temporary variables
 - Ways to reduce interference b/w functions



Functions: Our Goal

- Architectural support for functions
 - Branch and Link instruction (**BL**)
 - Stack Pointer (**SP**)
 - Link Register (**LR**)
- Microarchitecture-level impacts of programming styles
(**Iteration** vs. **Recursion**)
- Provides a deeper understanding of **hardware/software** interaction and tradeoffs

Functions in C

C Code

```
void main()
{
    int y;
    y = sum(42, 7);
    ...
} ... 42 and 7 are function arguments provided by caller, i.e., main()
int sum(int a, int b)
{
    return (a + b);
}
```

- **main()** is caller (calling someone else)
 - Returns nothing (**void**)
 - No input arguments

- **sum()** is *callee* (being called by someone)
 - Two input arguments of type **int**: **a** and **b**
 - Return type: **integer**
 - Returns the sum of **a** and **b**

Leaf and Non-Leaf Functions

- `sum()` is a leaf function
 - It does not call another function
- `main()` is a non-leaf function
 - It calls another function
- Non-leaf functions are more complicated especially at the assembly level
- `sum()` can be called from many different functions
 - Code reuse

Functions as Detectives

- Secret mission
- Acquire necessary resources
- Perform the mission
- Leave no trace
- Return safely



Functions as Detectives

- Caller stores *arguments* in specific registers
- Caller transfers *flow control* to the callee (`call`)
- Callee acquires/allocates memory for doing work
- Callee executes the function body
- Callee stores the result in a specific register
- Callee *returns* control to the caller (`return`)

ARM Function Calls

- Instruction for calling the function
 - **BL** (**B**ranch and **L**ink)
 - CPU branches to the label specified by BL
 - CPU stores the *return address* in the link register (LR)
- Return address is the address of the next instruction after the function call
- Returning from function
 - Move the link register into PC
 - MOV PC, LR
- Passing arguments (convention)
 - R0, R1, R2, R3
- Returning value (convention)
 - R0

Function Calls

C Code

```
int main() {  
    simple();  
    a = b + c;  
}  
  
void simple() {  
    return;  
}
```

ARM Assembly Code

0x00000200	MAIN	BL SIMPLE
0x00000204		ADD R4, R5, R6
...		
0x00401020	SIMPLE	MOV PC, LR

- **BL** branches to SIMPLE
 $LR = PC + 4 = 0x00000204$
- **MOV PC, LR** makes $PC = LR$
(the next instruction executed is at 0x00000200)

- **MAIN** and **SIMPLE** are labels (memory addresses) in assembly
- **BL** transfers flow to **SIMPLE** and stores the *return address* in **LR**
- The function returns after **MOV**, and the next instruction (**ADD**) is executed

Example: Difference of Sums

```
C code:  
int main() {  
    int y;  
    ...  
    y = diffofsums(2, 3, 4, 5);  
    ...  
}  
  
int diffofsums(int f, int g, int h, int i) {  
    int result;  
    result = (f + g) - (h + i);  
    return result;  
}
```

ARM Assembly Code

```
; R4 = y
```

MAIN

```
...
MOV R0, #2          ; argument 0 = 2
MOV R1, #3          ; argument 1 = 3
MOV R2, #4          ; argument 2 = 4
MOV R3, #5          ; argument 3 = 5
BL  DIFFOFSUMS    ; call function
MOV R4, R0          ; y = returned value
...
```

```
; R4 = result
```

DIFFOFSUMS

```
ADD R8, R0, R1      ; R8 = f + g
ADD R9, R2, R3      ; R9 = h + i
SUB R4, R8, R9      ; result = (f + g) - (h + i)
MOV R0, R4          ; put return value in R0
MOV PC, LR          ; return to caller
```

Questions

- How can we pass more than 4 function arguments?
- How can we ensure that registers in use by the caller are not corrupted?
 - DIFFOFSUMS overwrites R4, R8, R9
 - MAIN may need these registers after return
- The Stack
 - A special area in memory used across function calls
 - Preserving registers, passing arguments, scratch space

The Stack

- Abstract view
 - Last In First Out (LIFO) Queue
- Stored in memory at some arbitrary address in memory
- Caller and callee can *push* things onto the stack and *pop* things off the stack
- Stack expands and contracts over time as function call and return



ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

Word Count (`wordcount.c`)

Problem statement: Count the number of words input by the user via keyboard

A *word* contains a sequence of characters (a-z, A-Z) without a blank space or a new line

- ✓ ENGN2219
- ✓ 2219ENGN
- ✓ ENGN
- ✗ 2219

Input/Ouput

Input: *Read from keyboard character-wise until end of file is encountered. Blank space and new-line starts a new word.*

Output: *# of words*

Flow of our solution

1. Read characters until the **end of file** is encountered
EOF is a symbolic constant in C (equivalent to pressing Ctrl-d)
2. For each character, check if it is in the range: **a to z or A to Z**
3. Keep count of the number of words seen so far

Reading characters

```
char c;  
while ((c = getchar()) != EOF) {
```

Checking for characters in range

In C, the *char* type or the character variable holds an ASCII value between 0 and 127

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0 000	NUL	(null)	32	20 040	 	Space		64	40 100	@	Ø	96	60 140	`	`		
1	1 001	SOH	(start of heading)	33	21 041	!	!	!	65	41 101	A	A	97	61 141	a	a		
2	2 002	STX	(start of text)	34	22 042	"	"	"	66	42 102	B	B	98	62 142	b	b		
3	3 003	ETX	(end of text)	35	23 043	#	#	#	67	43 103	C	C	99	63 143	c	c		
4	4 004	EOT	(end of transmission)	36	24 044	$	\$	\$	68	44 104	D	D	100	64 144	d	d		
5	5 005	ENQ	(enquiry)	37	25 045	%	%	%	69	45 105	E	E	101	65 145	e	e		
6	6 006	ACK	(acknowledge)	38	26 046	&	&	&	70	46 106	F	F	102	66 146	f	f		
7	7 007	BEL	(bell)	39	27 047	'	'	'	71	47 107	G	G	103	67 147	g	g		
8	8 010	BS	(backspace)	40	28 050	(((72	48 110	H	H	104	68 150	h	h		
9	9 011	TAB	(horizontal tab)	41	29 051)))	73	49 111	I	I	105	69 151	i	i		
10	A 012	LF	(NL line feed, new line)	42	2A 052	*	*	*	74	4A 112	J	J	106	6A 152	j	j		
11	B 013	VT	(vertical tab)	43	2B 053	+	+	+	75	4B 113	K	K	107	6B 153	k	k		
12	C 014	FF	(NP form feed, new page)	44	2C 054	,	,	,	76	4C 114	L	L	108	6C 154	l	l		
13	D 015	CR	(carriage return)	45	2D 055	-	-	-	77	4D 115	M	M	109	6D 155	m	m		
14	E 016	SO	(shift out)	46	2E 056	.	.	.	78	4E 116	N	N	110	6E 156	n	n		
15	F 017	SI	(shift in)	47	2F 057	/	/	/	79	4F 117	O	O	111	6F 157	o	o		
16	10 020	DLE	(data link escape)	48	30 060	0	Ø	Ø	80	50 120	P	P	112	70 160	p	p		
17	11 021	DC1	(device control 1)	49	31 061	1	1	1	81	51 121	Q	Q	113	71 161	q	q		
18	12 022	DC2	(device control 2)	50	32 062	2	2	2	82	52 122	R	R	114	72 162	r	r		
19	13 023	DC3	(device control 3)	51	33 063	3	3	3	83	53 123	S	S	115	73 163	s	s		
20	14 024	DC4	(device control 4)	52	34 064	4	4	4	84	54 124	T	T	116	74 164	t	t		
21	15 025	NAK	(negative acknowledge)	53	35 065	5	5	5	85	55 125	U	U	117	75 165	u	u		
22	16 026	SYN	(synchronous idle)	54	36 066	6	6	6	86	56 126	V	V	118	76 166	v	v		
23	17 027	ETB	(end of trans. block)	55	37 067	7	7	7	87	57 127	W	W	119	77 167	w	w		
24	18 030	CAN	(cancel)	56	38 070	8	8	8	88	58 130	X	X	120	78 170	x	x		
25	19 031	EM	(end of medium)	57	39 071	9	9	9	89	59 131	Y	Y	121	79 171	y	y		
26	1A 032	SUB	(substitute)	58	3A 072	:	:	:	90	5A 132	Z	Z	122	7A 172	z	z		
27	1B 033	ESC	(escape)	59	3B 073	;	:	:	91	5B 133	[[123	7B 173	{	{		
28	1C 034	FS	(file separator)	60	3C 074	<	<	<	92	5C 134	\	\	124	7C 174	|			
29	1D 035	GS	(group separator)	61	3D 075	=	=	=	93	5D 135]]	125	7D 175	}	}		
30	1E 036	RS	(record separator)	62	3E 076	>	>	>	94	5E 136	^	^	126	7E 176	~	~		
31	1F 037	US	(unit separator)	63	3F 077	?	?	?	95	5F 137	_	_	127	7F 177		DEL		

Source: www.LookupTables.com

A note on characters in C

```
if (((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z'))) {
```

Counting words

First, we need a variable to store the *state* we are in

Inside a word (**IN**)

Outside a word (**OUT**)

If the initial state is **OUT**

State transitions to **IN** if the user types a valid character

State transitions to **OUT** if the user inputs blank space or \n

OUT → IN

We have a new word (words++)

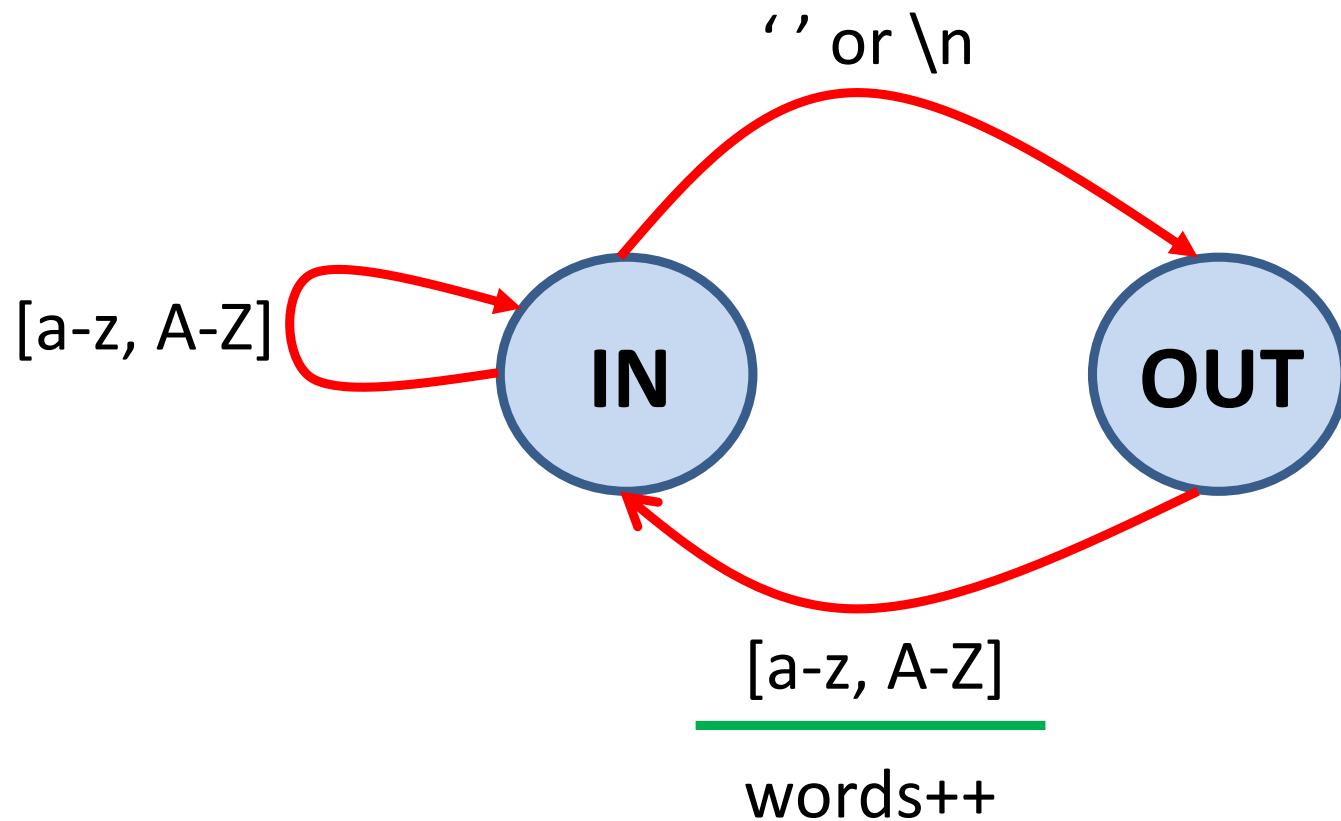
Question?

What happens if we are in state **IN** and the user inputs a valid character?

Printing the #words

```
printf("%d\n", words);
```

State machine diagram



wordcount.c

```
1 #include <stdio.h>

2 #define IN 1 // inside a word
3 #define OUT 0 // outside a word

4 int main(void) {

5     char c;
6     int inorout = OUT;
7     int words = 0;

8     while ((c = getchar()) != EOF) {
9         if (((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z'))) {
10             if (inorout == OUT) {
11                 inorout = IN;
12                 words++;
13             }
14         } else if ((c == ' ') || (c == '\n')) {
15             inorout = OUT;
16         } else {
17             // ignore
18         }
19     }

20     printf("%d\n", words);

21     return 0;
22 }
```

Address Space

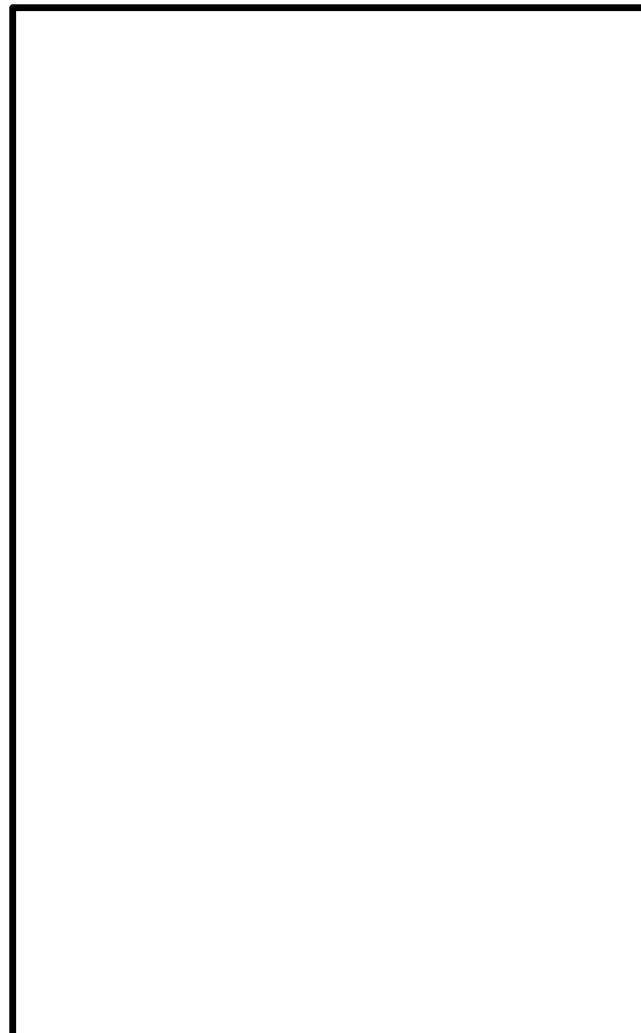
- **Address range**
 - A **32-bit** (ARM) CPU generates addresses in the range **0** to **0xFFFFFFFFC (4294967292)**
 - With a **4×10^9** address range, the CPU can access **4 billion** individual bytes
- **Address space**
 - The address space of a **32-bit** CPU is **2^{32}** bytes which equals **4 Gigabytes (GB)**

0xFFFFFFF0

Address Space

- Each word is 32 bits or 4 bytes. Address of first & last word is shown
- The address space is empty as shown here
 - Let's populate with stack and code and data

0x00000000



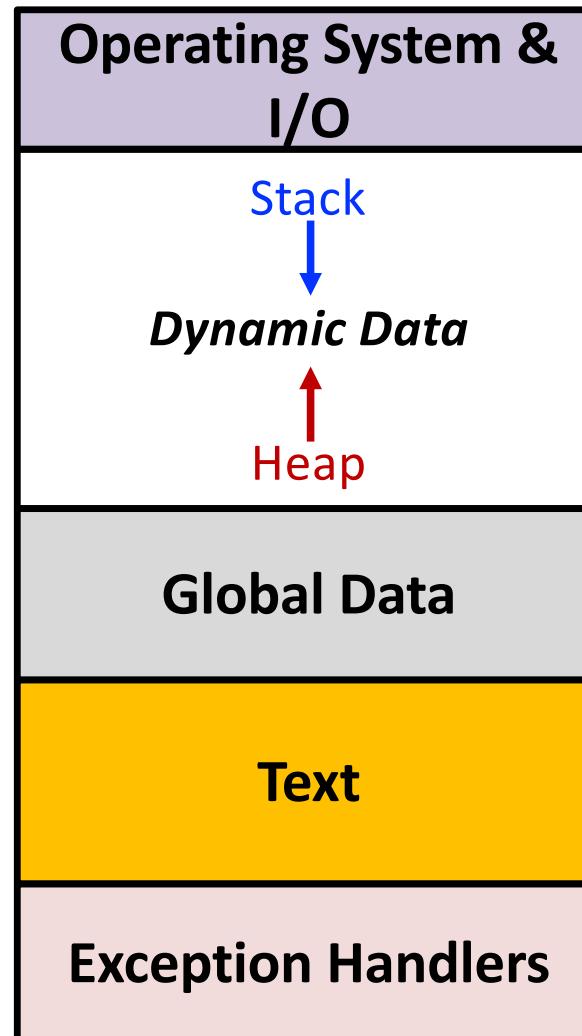
Questions

- Where is the code, data, and the stack in the address space?
- **Memory map**
 - Defines where code, data, and stack memory are in the program address space
 - Differs from architecture to architecture
 - The subsequent discussion pertains to ARM

ARM 32-bit Memory Map

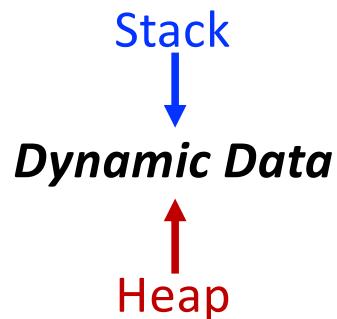
- Five parts or segments
 - text
 - global data
 - dynamic data
 - OS & I/O
 - Exception handlers

0xFFFFFFF0



0x00000000

Operating System & I/O



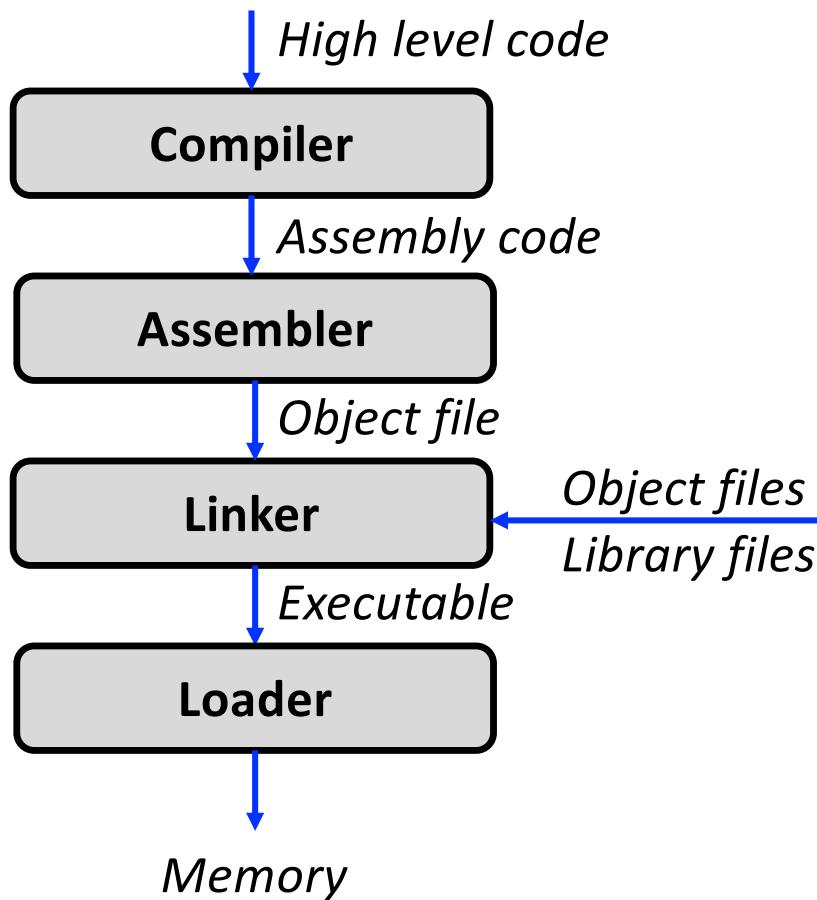
Global Data

Text

Exception Handlers

- *Data in this segment is dynamically allocated and deallocated during program execution*
- *Heap data is allocated by the program at runtime*
 - `malloc()` and `new`
- *Heap grows upward, stack grows downward*
- *Global variables visible to all functions (contrasted with local variables that are only visible to a function)*
- *Machine language program*
- *Also called read-only (RO) segment*
- *Literals (constants) such as "Hello"*

Translating/Starting Programs

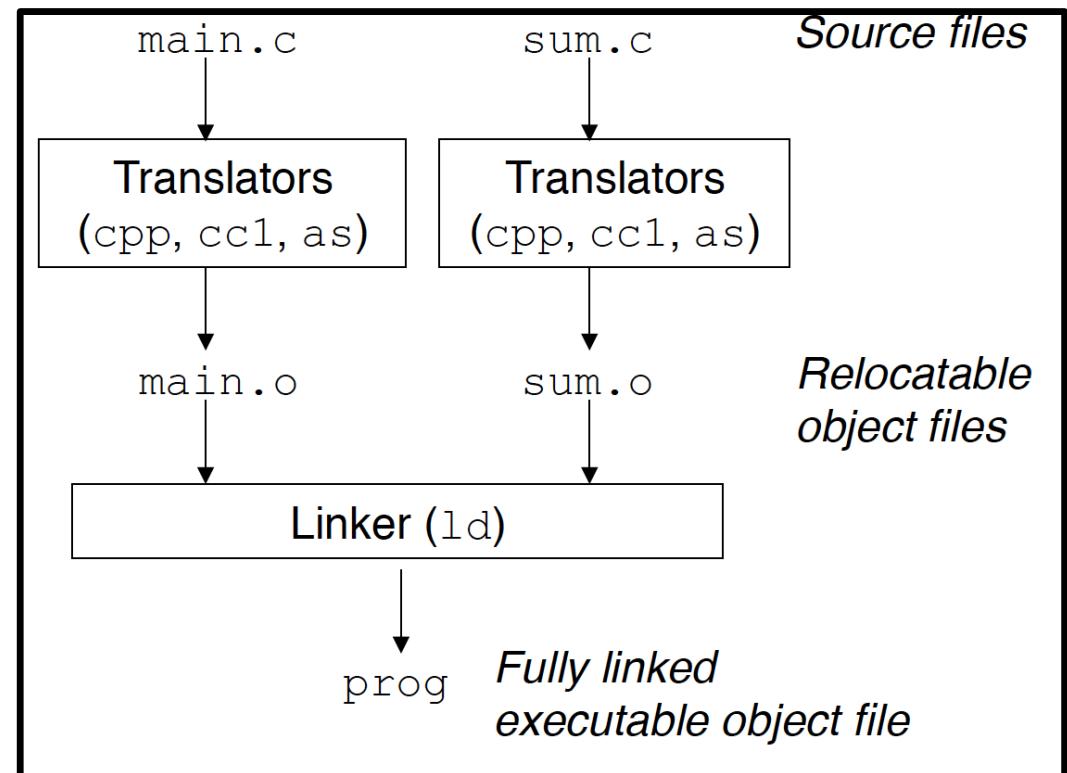


Translating/Starting Programs

- GNU compilation system & Linux specific
 - `gcc -o prog main.c sum.c`
- Invokes the GCC driver
 - GCC performs # steps

Translating/Starting Programs

- GNU compilation system & Linux specific
 - `gcc -o prog main.c sum.c`
- Invokes the GCC driver
 - GCC performs # steps



Example

sum.h:

```
int sum(int a, int b);
```

sum.c:

```
int sum(int a, int b) {  
    return a + b;  
}
```

main.c:

```
#include <stdio.h>  
#include "sum.h"  
int main() {  
    int c = sum(a,b);  
    printf("c = %i \n", c);  
    return 0;  
}
```

cpp, cc1, and as

- **cpp** is the C preprocessor: handles header file inclusion and macro expansion and generates an intermediate (.i) file
 - `#include <stdio.h>` copies the contents of `stdio.h`
 - `#define LEN 100` replaces `LEN` with `100` everywhere in the code
- **cc1** is the C compiler that generates an assembly (.s) file from the intermediate format (internal command)
- **as** is the assembler, which translates the assembly file into a binary relocatable (.o) object file

Relocatable Object File

- Contains binary code and data that can be combined with other relocatable object files at compile time to create an executable object file
- Symbols are not resolved
 - Variables with the same name declared in multiple sources
 - **Symbol resolution:** removes ambiguity by creating a single *linked executable file*
- Functions and variables are not bound to any specific address
- Addresses are still symbols (i.e., starting from 0 in each object file), and not properly assigned to an address in the memory map

Useful gcc commands

- Can break down the steps to generate the final executable file
 - `gcc -c main.c` (generates `main.o`)
 - `gcc -c sum.c` (generates `sum.o`)
 - `gcc -o prog main.o sum.o`
- Can generate the assembly (`.s`) file to view the assembly
 - `gcc -S main.c` (generates `main.s`)

Program vs. Process

- Program
 - A compiled executable binary lies dormant on a storage medium such as disk
- A process is a running program
 - The loader loads the executable file (image) in memory and fills up the memory map with code and data
 - Process has an execution environment (stack and heap)

Scope

- C identifiers (variables, functions, macros) have scope that delimits the regions where they can be accessed
- Four type of scope
 - File
 - Block
 - Function prototype
 - Function
- Scope is determined by where a variable is declared in the program

Example: Scope

```
int j; // file scope of j begins

void f(int i) { // block scope of i begins
    int j = 1; // block scope of j begins; hides file-scope j
    i++;        // i refers to function parameter
    for (int i = 0; i < 2; i++) { // block scope of loop-local i begins
        int j = 2; // block scope of inner j begins, hides outer j
        printf("%d\n", j); // inner j is in scope, prints ?
    }
    printf("%d\n", j); // outer j is in scope, prints ?
} // the block scope of i and j ends

void g(int j); // j has function prototype scope; hides file-scope j
```

Storage Duration

- Variables (objects) have a storage duration that determines their lifetime
 - There is a physical memory location reserved for the variable
- **Example:** Variables declared inside a code block or function definition are alive during the execution of the block or function
- **Example:** The loop index (declared inside the `for` statement) dies when loop terminates
- Four durations available in C
 - `automatic`
 - `static`
 - `allocated`
 - `thread` (related to concurrency, won't cover)

Storage Duration: automatic

- Consider the function definition below
 - Variable `life_and_death` has automatic storage duration
 - Implicit, no need to specify explicitly
 - It is alive only during the execution of function in which it is declared

```
void function(int i) {  
    int life_and_death = 1;  
    printf("%i\n", life_and_death);  
}
```

```
int main() {  
    int life_and_death = 2;  
    printf("%i\n", life_and_death);  
    return 0;  
}
```

Storage Duration: **automatic**

- More generally, variables declared in a code block demarcated by { . . . } have automatic storage duration

```
int main() {  
    for (int i = 0; i < 100; i++) → i has automatic storage duration  
        printf("%i\n", i);  
    return 0;  
}
```

- The compiler (or assembly programmer) can reclaim the register in which **i** is stored after the **for** loop terminates

Storage Duration: static

- Objects declared in file scope have **static** storage duration
- Array `big_array` has **static** storage duration
 - Lifetime → Entire execution of program
- Static storage duration is implicit for variables declared in file scope

```
int big_array[1L<<24]; //static
int huge_array[1L<<31]; // static

int main() {
    printf("%d\n", life_and_death);
    return 0;
}
```

Storage Duration: static

- We can explicitly use `static` storage duration for variables inside functions
- These variables persist after the function exits

```
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
    // ctr is only initialized once
    static unsigned int ctr = 0;
    ctr++;
    printf("ctr = %d\n", ctr);
}

int main() {
    for (int i=0; i<5; i++)
        increment();
    return 0;
}
```

Storage Duration: static

- We can explicitly use `static` storage duration for variables inside functions
- These variables persist after the function exits
- Output
 - `ctr = 1`
 - `ctr = 2`
 - `ctr = 3`
 - `ctr = 4`
 - `ctr = 5`

```
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
    // ctr is only initialized once
    static unsigned int ctr = 0;
    ctr++;
    printf("ctr = %d\n", ctr);
}

int main() {
    for (int i=0; i<5; i++)
        increment();
    return 0;
}
```

Storage Duration: static

- We can declare `ctr` outside the function `increment()`
- Good software engineering practice to limit the scope of a variable whenever possible
- Static variables cannot be initialized to the symbol/name of another variable
- Initialized only to constants such as `0`, `1`, “Hello”

```
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
    // ctr is only initialized once
    static unsigned int ctr = 0;
    ctr++;
    printf("ctr = %d\n", ctr);
}

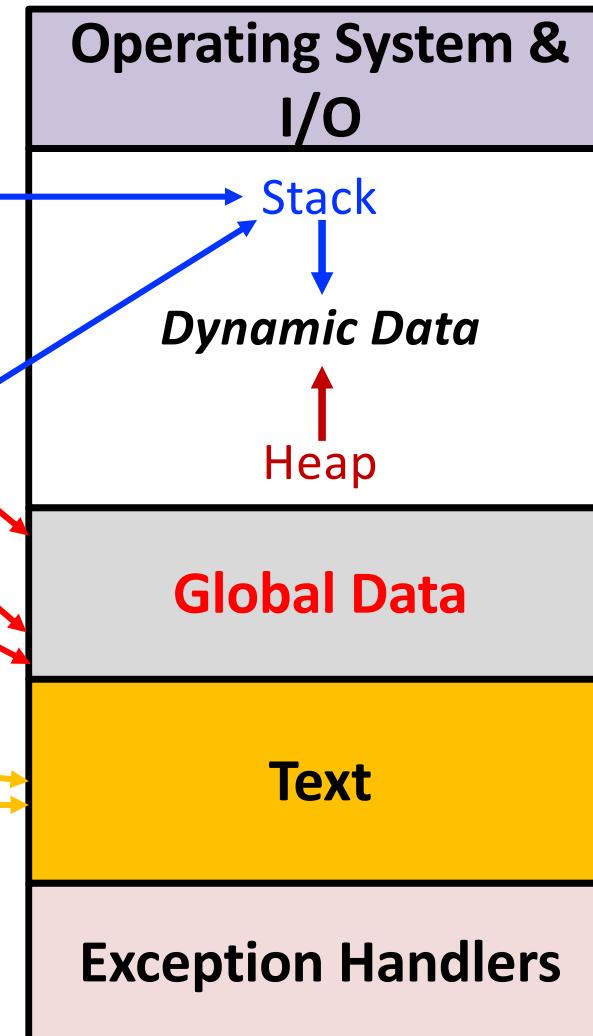
int main() {
    for (int i=0; i<5; i++)
        increment();
    return 0;
}
```

Where is everything mapped?

```
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
    // ctr is only initialized once
    static unsigned int ctr = 0;
    ctr++;
    printf("ctr = %d\n", ctr);
}

int main() {
    for (int i=0; i<5; i++)
        increment();
    int *ptr;
    return 0;
}
```



Statically Allocated Memory

```
int big_array[1L<<24];
int huge_array[1L<<31];
```

- The above arrays are statically allocated
 - Size must be known at compile time (*limitation of static*)
- We need to specify the size of the statically-allocated arrays

```
int N;
int big_array[N];X

int main() {
    scanf("&i", &N);✓
    int array[N];
}
```

This array is allocated on the stack

Limitations of Stack Allocation

- Memory is allocated and deallocated (freed) in a specific order
 - Last In First Out (LIFO)
- Memory is retained on the stack even if it is not needed
 - Memory is a precious resource
- No way for programmer to inform the compiler/OS to free unused memory
 - Deallocated only when function returns
 - And in a specific order

Example: Mem Waste

```
int caller_useless_func() {  
    int sum = 0;  
    int array[5] = {1, 2, 3, 4, 5};  
    for (int i = 0; i < 5; i++)  
        sum += array[i];  
    int x = useless_func(sum);  
    return x * sum;  
}  
  
int useless_func(int var) {  
    int multiplied = var * 5;  
    return multiplied;  
}
```

→ **array** is no longer
needed after this
statement, but it is still
retained on the stack

Another Problem: Stack Allocation

- Sharing data across functions

```
int caller_useless_func() {  
    int sum = 0;  
    int array[5] = {1, 2, 3, 4, 5};  
    for (int i = 0; i < 5; i++)  
        sum += array[i];  
    int *y = useless_func(sum);  
    printf("Now printing .... \n");  
    printf("%d\n", *y * sum);  
}  
  
int *useless_func(int var) {  
    int multiplied = var * 5;   
    return &multiplied;  
}
```

 **printf's** stack frame
likely corrupts the
useless_func's
stack frame

 **multiplied** is dead
after the function returns

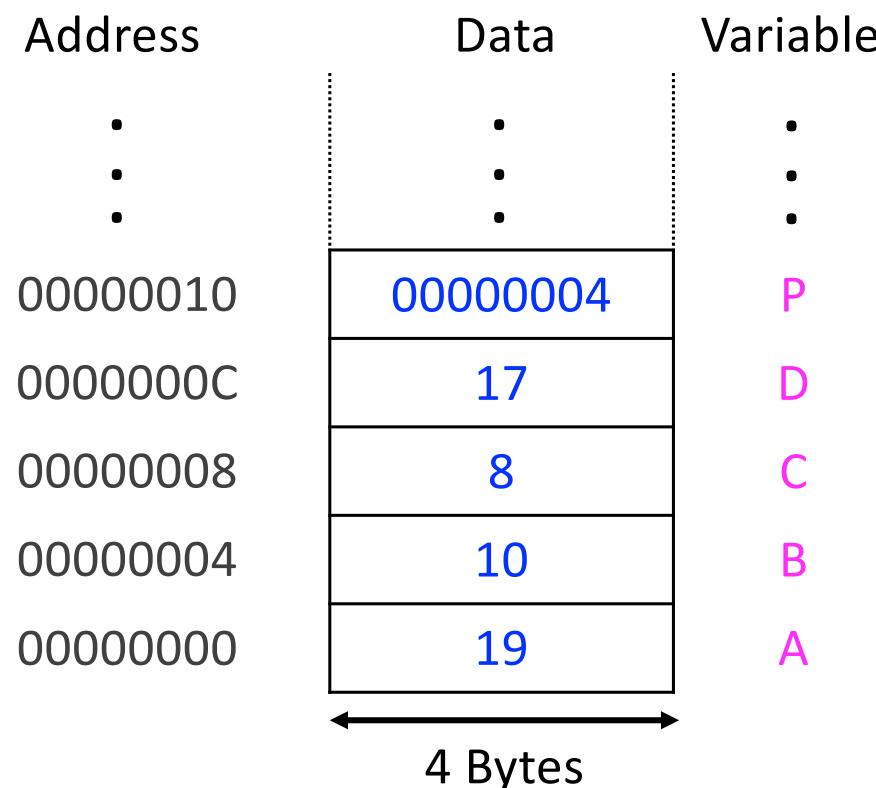
Problem: Stack Allocation

- We can solve the problem by allocating **multiplied** as a global variable
- What if we want to share an array b/w functions?
 - Statically allocate the array
 - Ok, but what if we only find out the exact length of the array at run-time? What if we want to resize the array?
 - E.g., number of records in a database not known in advance
 - Student numbers grow dynamically during the semester
- We use the heap for such “dynamically allocated” memory

Pointers: Revision

- A pointer is a variable that contains the address of another variable

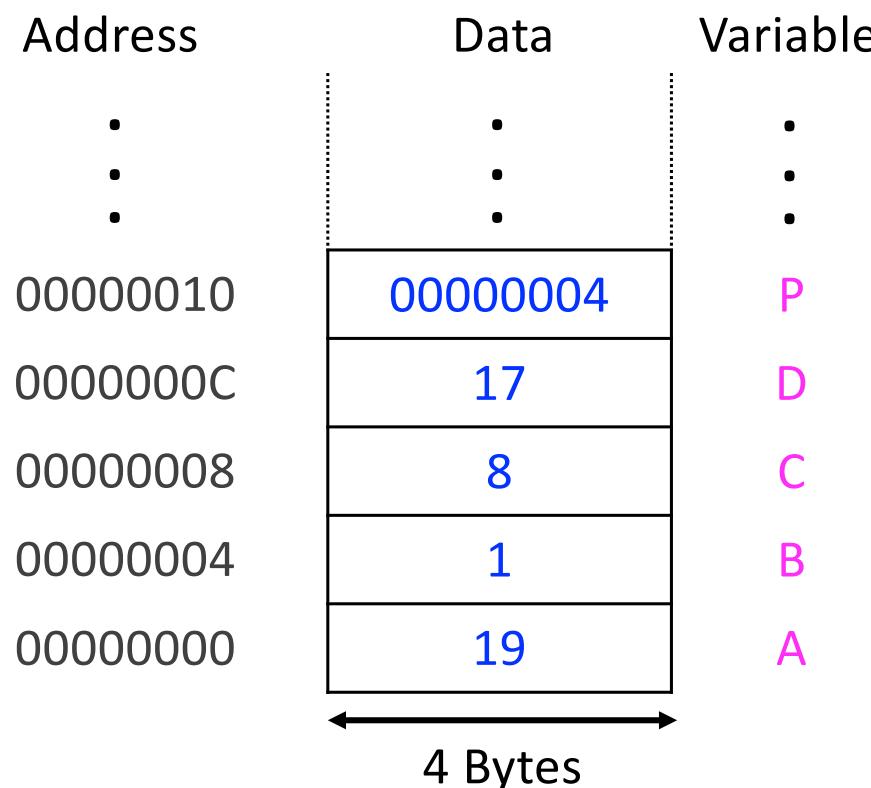
```
int A = 19;  
int B = 10;  
int C = 8;  
int D = 17;  
....  
int *P = &B;  
//unary operator & gives  
the address of a  
variable
```



Pointers: Revision

- Can use the pointer to access the value stored in a memory location

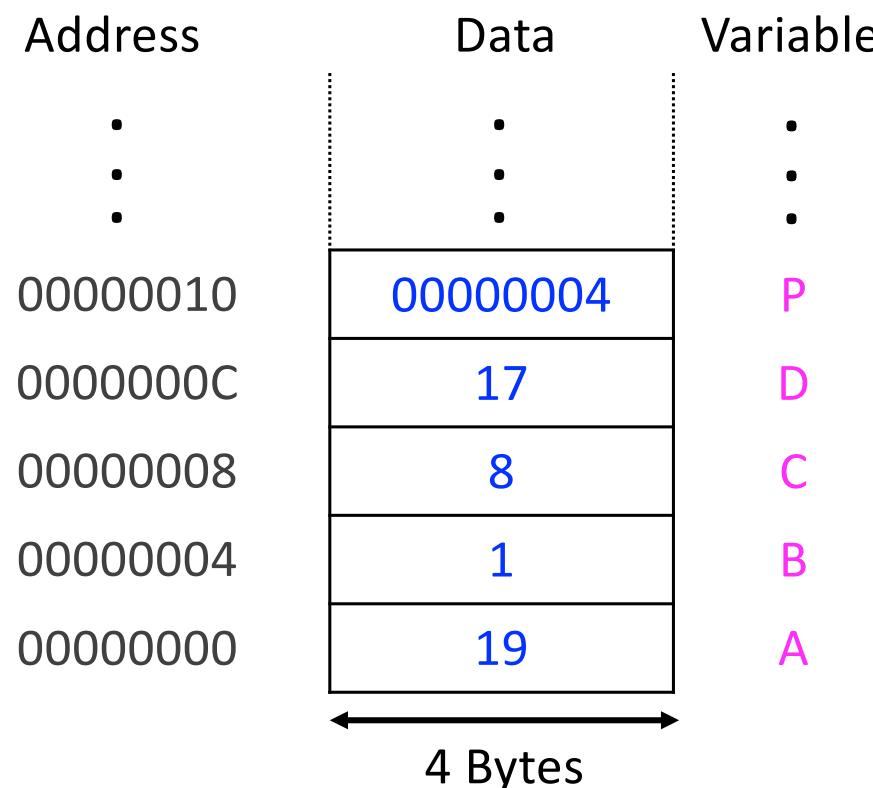
```
int A = 19;  
int B = 10;  
int C = 8;  
int D = 17;  
....  
int *P = &B;  
  
*P = 1;  
  
//dereferencing or  
// indirection operator  
//that accesses the value  
// stored at address in P
```



Pointers: Example

- A pointer is 4-bytes on a 32-bit system and 8-bytes on a 64-bits system & it can be stored on the stack or data segment like ordinary variables

```
int A = 19;  
int B = 1;  
int C = 8;  
int D = 17;  
....  
int *P = &B;  
char *Q = &B;  
// Both P and Q contain  
00000004  
printf("%i\n", *P); ??  
printf("%i\n", *Q); ??
```



Pointers: Bottomline

- A pointer points to a memory location
- Its content is a memory address
- It wears “datatype glasses”
 - Wherever it points, it sees through these glasses
- The variable stored at some memory address can be interpreted (via the dereferencing operator *) as character or integer or float, depending on the type of the pointer

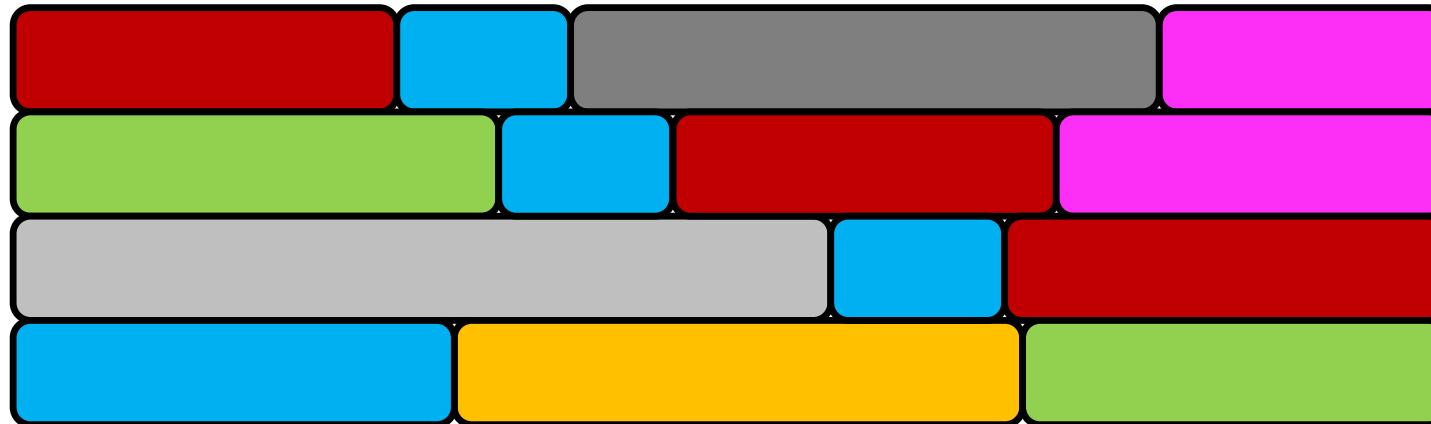


Heap

- Heap is for dynamically allocated memory
 - Contrast with statically allocated memory
 - Large subdividable block of memory
 - Programmers explicitly allocate and deallocate memory on the heap
- Lifetime of heap variables/arrays extend from allocation until deallocation
- Memory managers are libraries that manage heap for the programmer (we will refine this view)

Heap Organization

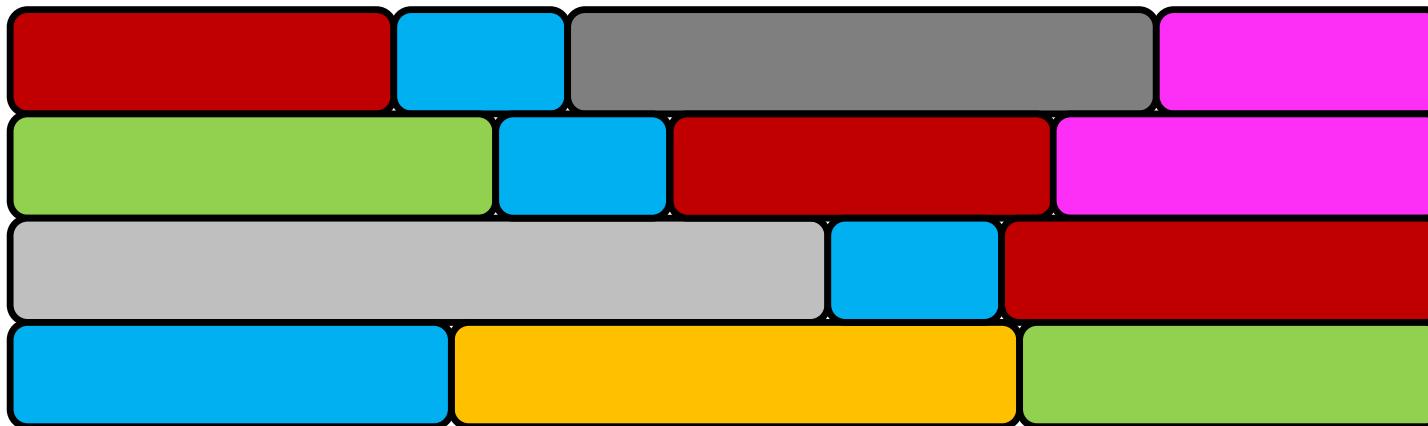
- Programs dynamically allocate objects (arrays, structures) of different types and sizes on the heap over time



- **Heap is now full!**

Heap Organization

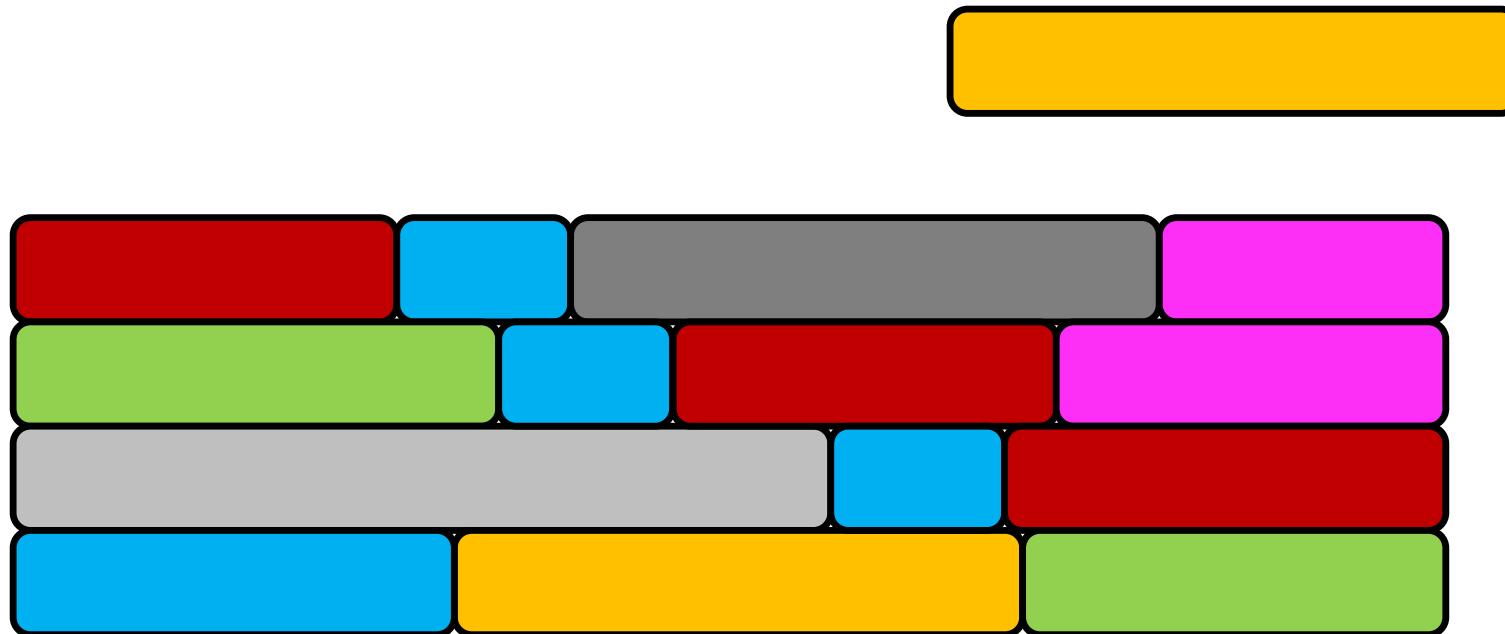
- C programmers need to track lifetime of heap variables
- Suppose we do not need anymore



- **Heap is now full!**

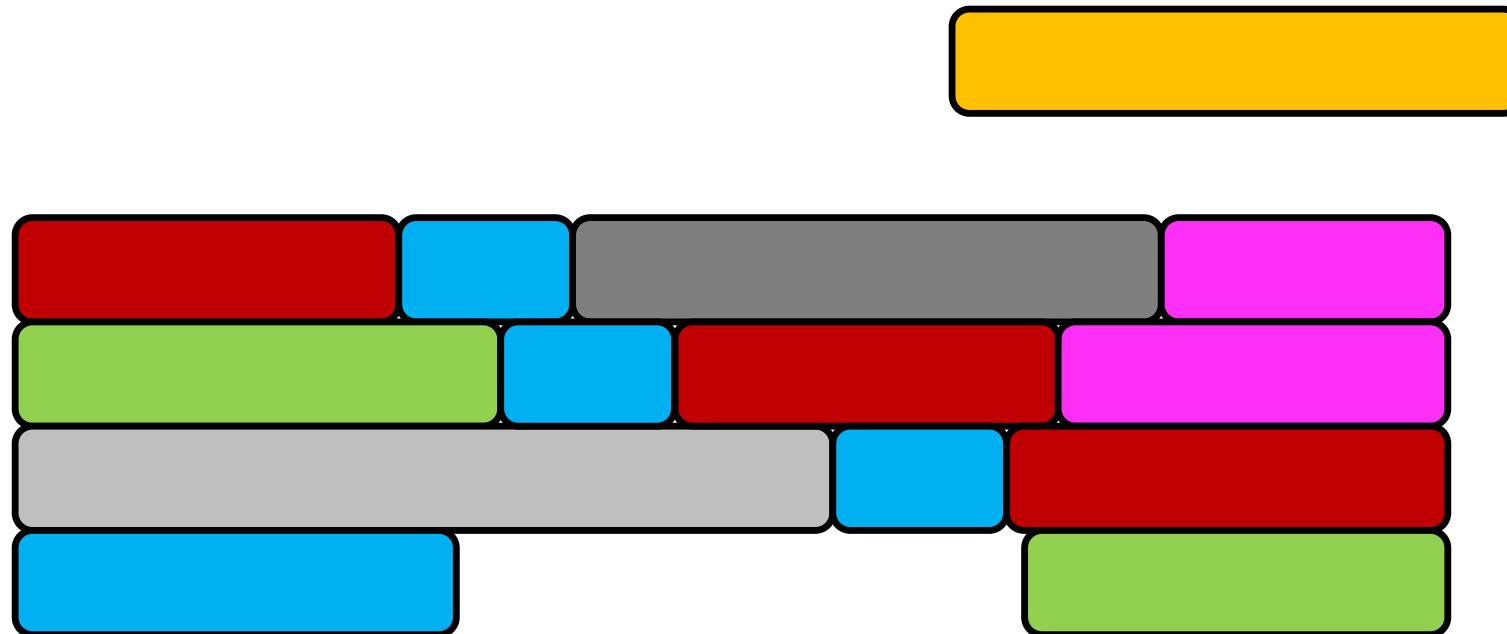
Heap Organization

- Let's free some space on the heap



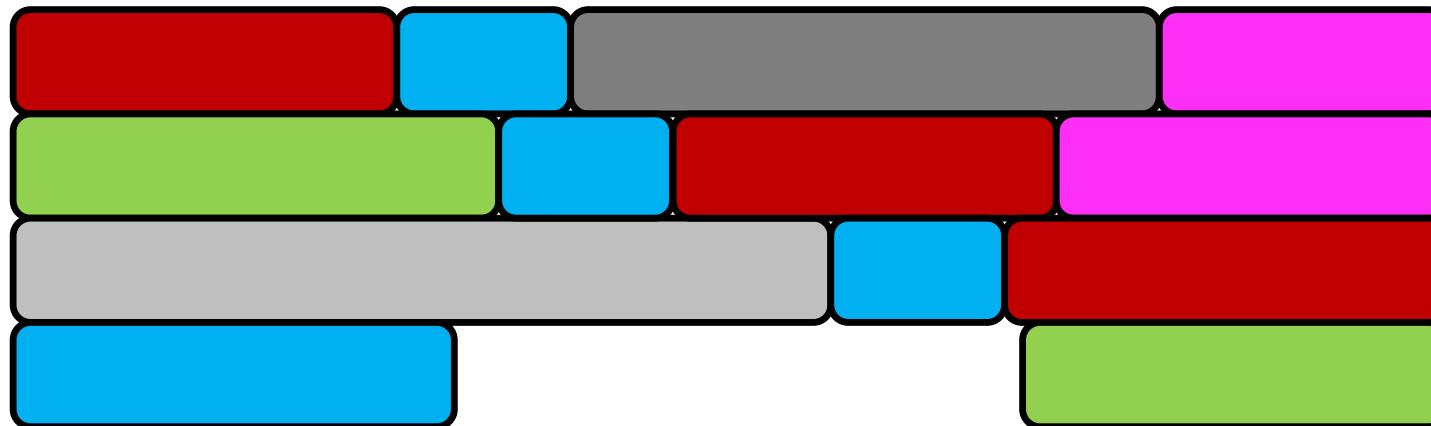
Heap Organization

- Let's free some space on the heap



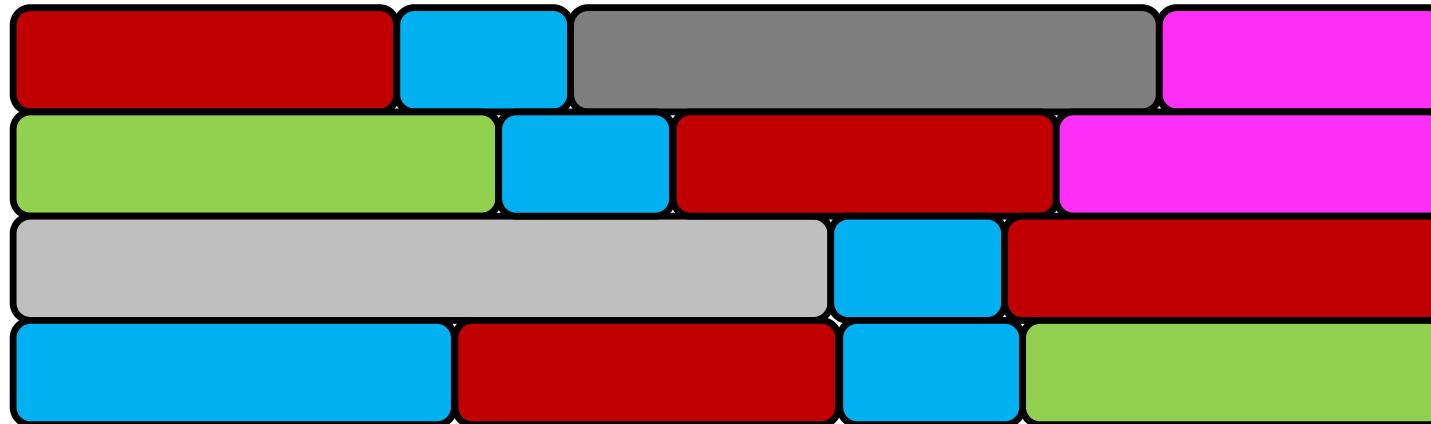
Heap Organization

- Now we can allocate new objects



Heap Organization

- Now we can allocate new objects



Heap Management

- We can deallocate (free) objects in any order (unlike the stack)
- Programmers need to deallocate objects that are no longer needed
 - Otherwise, heap will remain full even if we can have some free space
- Not returning unused heap back to the memory manager is called a *memory leak*

malloc and free

- C library provides **malloc** (short for memory allocate) and **free** to allocate and deallocate heap memory, respectively

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *array = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
        array[i] = i * i;
    int sum = array[0] + array[9];
    free(array);
    printf("%i\n", sum);
    return;
}
```

malloc and free

- **malloc**
 - Declaration in C library: `void *malloc(size_t size)`
 - Takes input as size (# bytes)
 - Returns a void pointer that can be casted to any pointer type
- **free**
 - Declaration in C library: `void free(void *ptr)`
 - Memory manager knows how many bytes to free, all it needs is the starting address

Quiz: Bug?

- Is there a memory-related bug in the following program?

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    int *array2 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++) {
        array1[i] = i * i;
        array2[i] = i + i;
    }
    int sum = array1[0] + array2[9];
    free(array1);
    free(array2);
    printf("%i\n", sum + array1[8]);
    return;
}
```

Quiz: Bug?

- Is there a memory-related bug in the following program?

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    int *array2 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++) {
        array1[i] = i * i;
        array2[i] = i + i;
    }
    int sum = array1[0] + array2[9];
    free(array1);
    free(array2);
    printf("%i\n", sum + array1[8]);
    return;
}
```

use after free

Quiz: Bug?

- Is there a memory-related bug in the following program?

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
        array1[i] = i * i;
    array1 = malloc(5 * sizeof(int));
    for (int i = 0; i < 10; i++)
        array1[i] = i * i;
    free(array1);
    printf("Done ....\n");
    return;
}
```

Quiz: Bug?

- Is there a memory-related bug in the following program?

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
        array1[i] = i * i;
    array1 = malloc(5 * sizeof(int));  
    for (int i = 0; i < 10; i++)  
        array1[i] = i * i;
    free(array1);
    printf("Done ....\n");
    return;
}
```

memory leak
Original 10-
element array
still on heap
(address is
gone, not saved)

out of bounds

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au

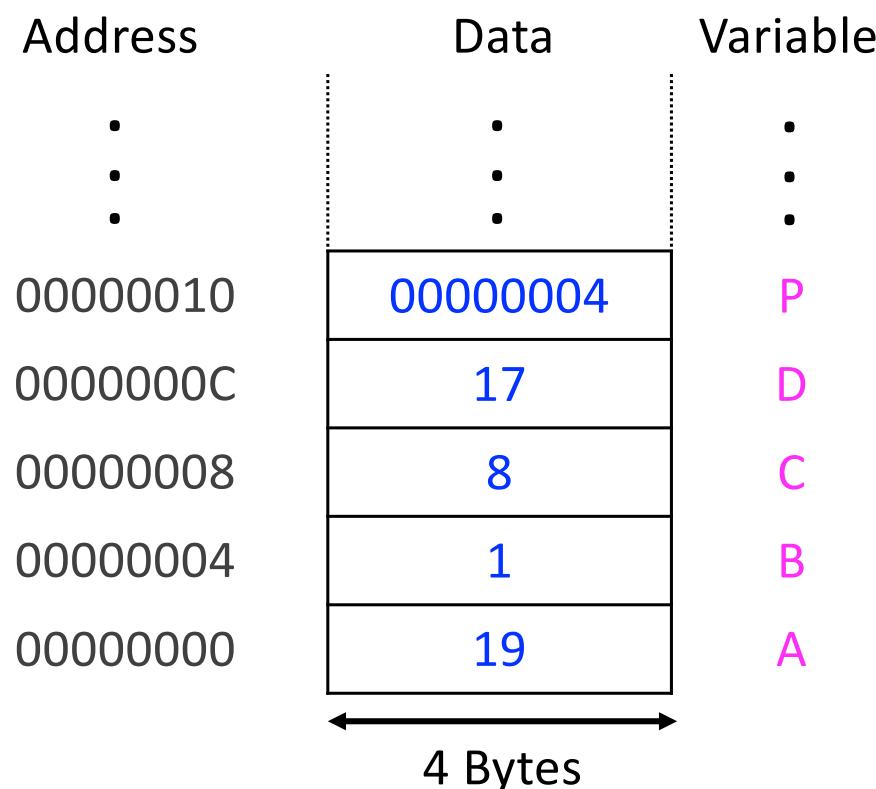


Australian
National
University

Pointers: Example

- Guess the output of the `printf()` statements

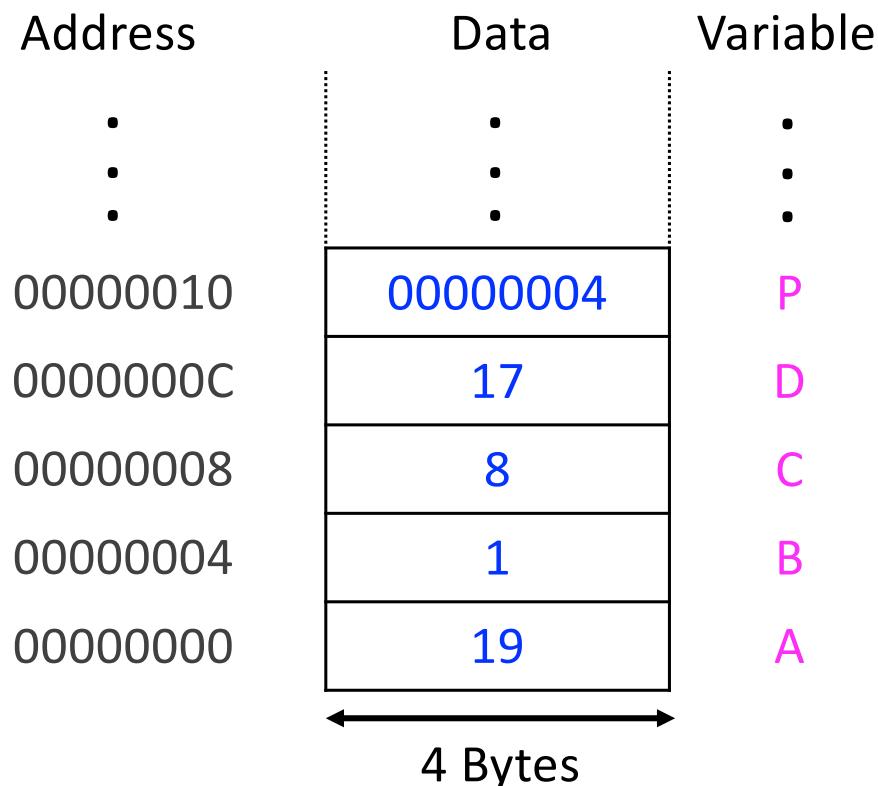
```
int A = 19;  
int B = 1;  
int C = 8;  
int D = 17;  
....  
int *P = &B;  
char *Q = &B;  
// Both P and Q contain  
    00000004  
printf("%i\n", *P); ??  
printf("%i\n", *Q); ??
```



Answer

- `printf("%i\n", *P);` Output is always 1
- `printf("%i\n", *Q);` Big Endian: 0, Little Endian: 1

```
int A = 19;
int B = 1;
int C = 8;
int D = 17;
...
int *P = &B;
char *Q = &B;
// Both P and Q contain
    00000004
printf("%i\n", *P); ???
printf("%i\n", *Q); ???
```



malloc and free

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
        array1[i] = i * i;
    printf("%i\n", sizeof(array1));
    free(array1);
    printf("Done ....\n");
    return;
}
```

include for
malloc()

use sizeof(int)
operator rather
than guessing
how big is an
int on a machine

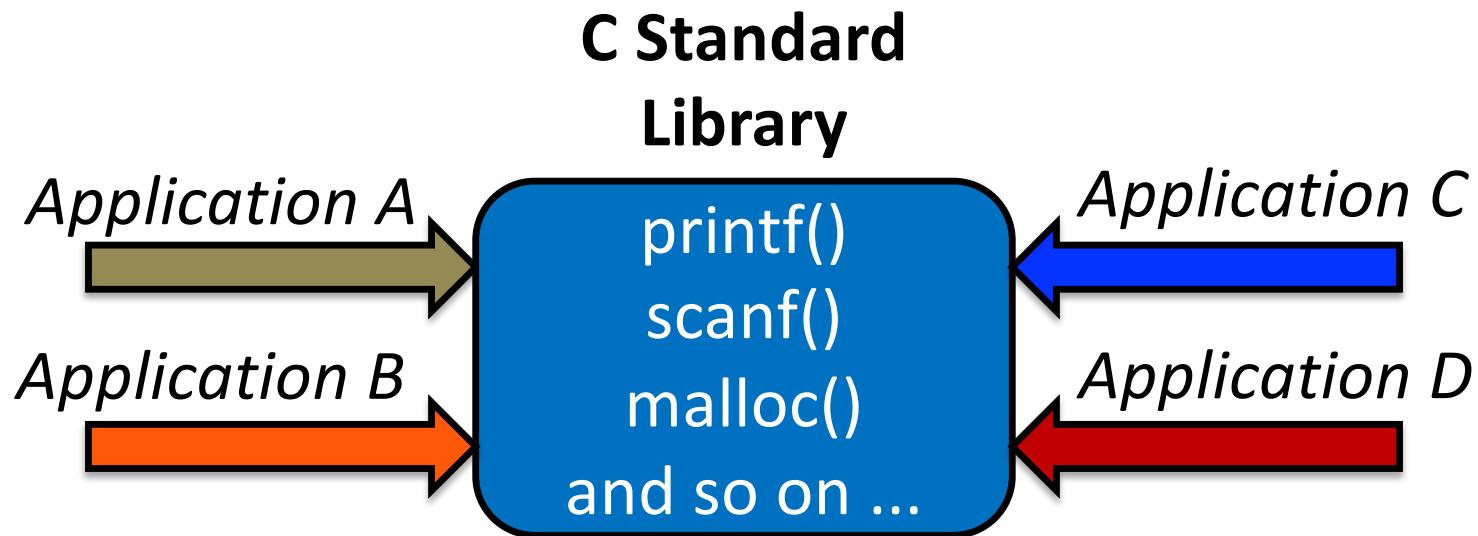
16 or 32 or 64
depending on
the machine
architecture

API

- *Application Programming Interface (API)*
 - Defines the interfaces by which one software program communicates with another at the source code level
 - **Abstraction:** API provides a standard set of interfaces that many different users (writing programs) can invoke
- *API defines the interface only*
 - The user of the API can ignore the implementation
 - Many implementations of an API can exist

API: Example

- The C standard library (`libc`) defines a family of basic and essential functions: memory mgmt. and string manipulation



- `libc` hides a lot of operating system details by interacting with the OS on behalf of the programmer

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Programs

Device Drivers

Instructions
Registers

Datapaths
Controllers

Adders
Memories

AND Gates
NOT Gates

Amplifiers
Filters

Transistors
Diodes

Electrons

ABI

- *Application Binary Interface*
 - Defines the binary interface b/w two or more pieces of software on a particular architecture
- *ABI ensure binary compatibility*
 - Calling convention, byte ordering, register use, linking, binary object format
 - Enforced by the toolchain (not a programmer's worry)
- ABI is a function of both the architecture (x86, ARM) and operating system (Linux, Windows)

Common Memory Errors

- *Forgetting to allocate memory*

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    char *src = "Hello!";
    char *dst;
    strcpy(dst, src);
}
```

- *Causes a segmentation fault*
 - *System is angry at you, because you did something wrong with memory*

Description

The C library function `char *strcpy(char *dest, const char *src)` copies the string pointed to, by `src` to `dest`.

Declaration

Following is the declaration for `strcpy()` function.

```
char *strcpy(char *dest, const char *src)
```

Parameters

- `dest` – This is the pointer to the destination array where the content is to be copied.
- `src` – This is the string to be copied.

Common Memory Errors

- *Forgetting to allocate memory*

```
//Fixed program

#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    char *src = "Hello!";
    char *dst = (char*) malloc(strlen(src) + 1);
    strcpy(dst, src);
}
```

Common Memory Errors

- *Not allocating sufficient memory
(a.k.a. buffer overflow)*

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    char *src = "Hello!";
    char *dst = (char*) malloc(strlen(src));
    strcpy(dst, src);
}
```

Common Memory Errors

- *Not allocating sufficient memory (a.k.a. buffer overflow)*
 - There maybe an unused variable at the end of allocated memory (no harm)
 - Memory manager might have allocated a little extra space
 - Fault and crash and security vulnerability is likely

- *Just because a program runs, does not mean it is correct*

Common Memory Errors

- *Forgetting to initialize allocated memory*
 - Zero initialization is important for avoiding hard to debug problems

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *n = (int*) malloc(sizeof(int));
    ...
    cond = *n;
    ...
    if (cond == 0) {
        ...
    }
}
```

Common Memory Errors

- *Forgetting to free memory (memory leak)*
 - Huge problem in long-running programs
- When the process exits, the operating system automatically reclaims all allocated memory by the process
 - Still a bad habit to not call `free()` in short-running programs

Example

- Memory leak example (from last lecture)

```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
        array1[i] = i * i;
    array1 = malloc(5 * sizeof(int));  
    for (int i = 0; i < 10; i++)  
        array1[i] = i * i;
    free(array1);
    printf("Done ....\n");
    return;
}
```

memory leak
Original 10-
element array
still on heap
(address is
gone, not saved)

out of bounds

Common Memory Errors

- *Freeing memory before the program is done with it*
 - Known as a dangling pointer
 - The subsequent use can crash the program

```
// example 1
void useless_func() {
    int n = 100;
    int *ptr = (int*) malloc(sizeof(int));
    if (n == 100) {
        int v = 10;
        ptr = &v;
    }
    // ptr is now a dangling pointer
}
```

Common Memory Errors

- *Freeing memory before the program is done with it*
 - Known as a dangling pointer
 - The subsequent use can crash the program

```
// example 2
void useless_func() {
    int *ptr = (int*) malloc(sizeof(int));
    ...
    ...
    free(ptr);
    ...
    ...
    *ptr = 2; // ptr is now a dangling pointer
}
```

Common Memory Errors

- *Freeing memory before the program is done with it*
 - Known as a dangling pointer
 - The subsequent use can crash the program

```
//example 2 fix, you will now get a runtime error
void useless_func() {
    int *ptr = (int*) malloc(sizeof(int));
    ...
    ...
    free(ptr);
    ptr = NULL; // good practice
    ...
    ...
    *ptr = 2; // ptr is no longer a dangling pointer
}
```

Common Memory Errors

- *Freeing memory before the program is done with it*
 - Known as a dangling pointer
 - The subsequent use can crash the program

```
// example 3

int *useless_func() {
    int x = 100;
    return &x;
}

// when the function returns, its stack is
// deallocated, and we must not use the address
// returned by useless_func()
```

Common Memory Errors

- *Freeing memory multiple times*
 - double free (typical name)

```
void useless_func() {  
    int *x = (int *) malloc(100 * sizeof(int));  
    free(x);  
    ...  
    free(x); // confuses the memory manager  
}
```

Common Memory Errors

- *Incorrectly calling free()*
 - Avoid these so-called **invalid frees**

```
void useless_func() {  
    int *x = (int *) malloc(100 * sizeof(int));  
    ...  
    x = x + 4;  
    ...  
    free(x);  
}
```

Multi-Dimensional Arrays

- *Statically allocated arrays and stack-allocated 2-dimensional arrays are simple*

```
#define R 5
#define C 4
int matrix[R][C];

for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++) {
        matrix[i][j] = i + j;
    }
}
```

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]
[3][0]	[3][1]	[3][2]	[3][3]
[4][0]	[4][1]	[4][2]	[4][3]

Multi-Dimensional Arrays

- *Dynamically allocated 2-dimensional arrays*

```
#define R 5
#define C 4

int **matrix;

void useless_func() {
    matrix = (int **) malloc(R * sizeof(int *));
    for (int i = 0; i < R; i++) {
        matrix[i] = malloc(C * sizeof(int));
    }
}
```

Multi-Dimensional Arrays

- *You can have jagged arrays, where each row has a different # columns*

```
#define R 3

int **matrix;

void useless_func() {
    matrix = (int **) malloc(R * sizeof(int *));
    matrix[0] = malloc(2 * sizeof(int));
    matrix[1] = malloc(5 * sizeof(int));
    matrix[2] = malloc(3 * sizeof(int));
}
```

Structs

- Lab 10 handout introduces structs in C
- How can we create a dynamically allocated array of structs?

```
#define TOTAL 100

struct student {
    char name[16];
    int id;
};

typedef struct student student_t;

student_t *students;
void useless_func() {
    students = (student_t *) malloc(TOTAL * sizeof(student_t));
    students[0].name = "Shane";
    students[0].id = 10;
}
```

can access
members with .
operator

Structs and -> Operator

- If we have a pointer to a struct, we can use the arrow operator (->) to access the members of a struct

```
struct student {  
    char name[16];  
    int id;  
};  
  
typedef struct student student_t;  
  
student_t *student;  
void useless_func() {  
    student = (student_t *) malloc(sizeof(student_t));  
    student->name = "Shane";  
    student->id = 10;  
}
```

Enumerations in C

- Enumeration (or enum) is a user defined data type in C
 - Used to assign names to integral constants (code readability)

```
// An example program to demonstrate working
// of enum in C
#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat,
Sun};

int main()
{
    enum week day;
    day = Wed;
    printf("%i",day);
    return 0;
}
```

Principle of Locality

- Programs tend to reference (access) data items (words) that:
 - Reside near other recently accessed items
 - Were recently accessed themselves
- Temporal Locality
 - A memory location that is referenced once is likely to be referenced again multiple times soon
- Spatial Locality
 - If a memory location is referenced once, then a nearby location is likely to be referenced soon
- Generally, programs with good locality are likely to run faster than programs with poor locality

Locality is Everywhere

- Computer designers speed up main memory accesses by keeping most recently accessed data items in small fast memories called cache memories
 - Main memory is slow but high capacity (DRAM technology uses capacitors)
 - Cache is fast but low capacity (SRAM uses cross-coupled inverters)
- Web browsers exploit temporal locality by caching recently referenced documents on disks
- One of the enduring ideas in computer systems

Example

- Does the program below exhibit good locality?

```
int sumarray(int a[n]) {  
    int i;  
    int sum = 0;  
    for (i = 0; i < n; i++)  
        sum = sum + a[i];  
    return sum;  
}
```

Address	0	4	8	12	16	20	24	28
Contents	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
Access Order	1	2	3	4	5	6	7	8

Example

- Does the program below exhibit good locality?

```
int sumvec(int a[n]) {  
    int i;  
    int sum = 0;  
    for (i = 0; i < n; i++)  
        sum = sum + a[i];  
    return sum;  
}
```

Variable	Spatial	Temporal
sum	Poor	Good
a	Good	Poor

Address	0	4	8	12	16	20	24	28
Contents	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
Access Order	1	2	3	4	5	6	7	8

Explanation

- **array a**
 - Each element is accessed only once
 - Neighboring element is accessed soon in the next loop iteration
- **sum**
 - **sum** is a scalar, so no spatial locality
 - **sum** has good temporal locality, as it is accessed in each iteration

Stride

- Visiting each element of an array sequentially is an example of a *stride-1* reference pattern
 - Stride-1 reference pattern is called sequential access pattern
- Visiting every k-th element of a contiguous array is a *stride-k* reference pattern
- As stride increases, the spatial locality decreases
- Sequential accesses are generally highly desirable

Row Major Order

- C arrays are laid out in memory row-wise
 - The entire row is stored contiguously (elements are next to each other, 0, 4, 80)
- a_{mn} : m is row, and n is column

Row 1



Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access Order	1	2	3	4	5	6

Example

- Sum the elements of the array in row-major order

```
int sumarrayrows(int a[M][N]) {  
    int i, j, sum = 0;  
  
    for (i = 0; i < M; i++)  
        for (j = 0; j < N; j++)  
            sum = sum + a[i][j];  
    return sum;  
}
```

good spatial
locality

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access Order	1	2	3	4	5	6

Example

- What is the impact of interchanging i and j loops?

```
int sumarrayrows(int a[M][N]) {  
    int i, j, sum = 0;  
  
    for (j = 0; j < N; j++)  
        for (i = 0; i < M; i++)  
            sum = sum + a[i][j];  
    return sum;  
}
```

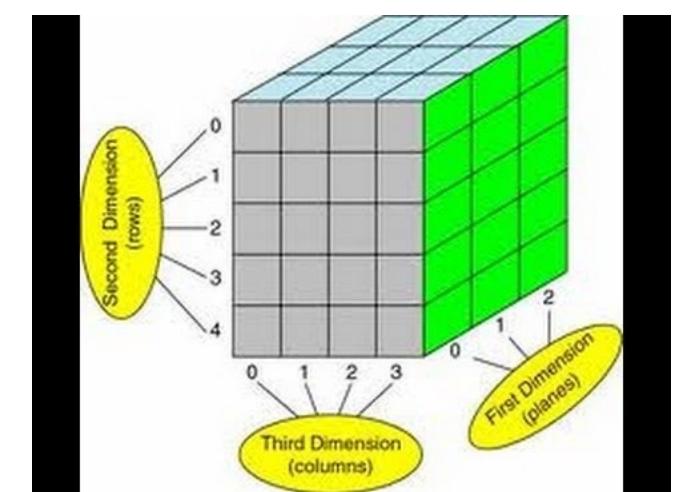
poor spatial
locality

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access Order	1	3	5	2	4	6

Practice Problem

Permute the loops in the following function so that it scans the 3-dimensional array `a` with a stride-1 reference pattern.

```
int sumarray3d(int a[N][N][N]) {  
    int i, j, k, sum = 0;  
  
    for (i = 0; i < N; i++)  
        for (j = 0; j < N; j++)  
            for (k = 0; k < N; k++)  
                sum = sum + a[k][i][j];  
  
    return sum;  
}
```



Locality of Instructions

- Program instructions are stored in memory and must be fetched by CPU
 - Locality is also relevant to instruction accesses
- Instructions in the loop body have high locality
 - Good spatial locality because instructions next to each other are executed in sequential order
 - Good temporal locality because the loop body is executed multiple times

Assignment 1 Grade Summary



Something was submitted by the deadline

Issues I Observed 😞

- Work in a group, but do not submit the group survey
- Dump the submission in the lecturer's inbox
 - Which gets ~100 emails on a slow-paced day
- Do not fork the assignment
- Submit only the top-level circuit and nothing else
- Do not submit report
- Do not respect the deadline, don't ask for extension
- Do not submit Statement of Originality

Good News

- Many reports were a pleasure to read
- Highest: 99.5
- Constantly surprised with the ambitious extensions
 - Multi-Cycle
 - Pipelining
 - New instructions for condensing the code
- On average, neatness and clarity was clearly seen in submissions that were pushed on time

ENGN2219/COMP6719

Computer Systems & Organization

Convenor: Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

Principle of Locality

- Temporal Locality (locality in time)
 - If an item is accessed (referenced), it is likely to be referenced again soon
- Spatial Locality (locality in space)
 - If an item is referenced, items whose addresses are close by are likely to be referenced soon
- Well-written programs exhibit good locality
 - Programmers need to aim for high locality in programs
 - Sequential access patterns lead to better performance
- ***Today: How can we build hardware to exploit locality?***

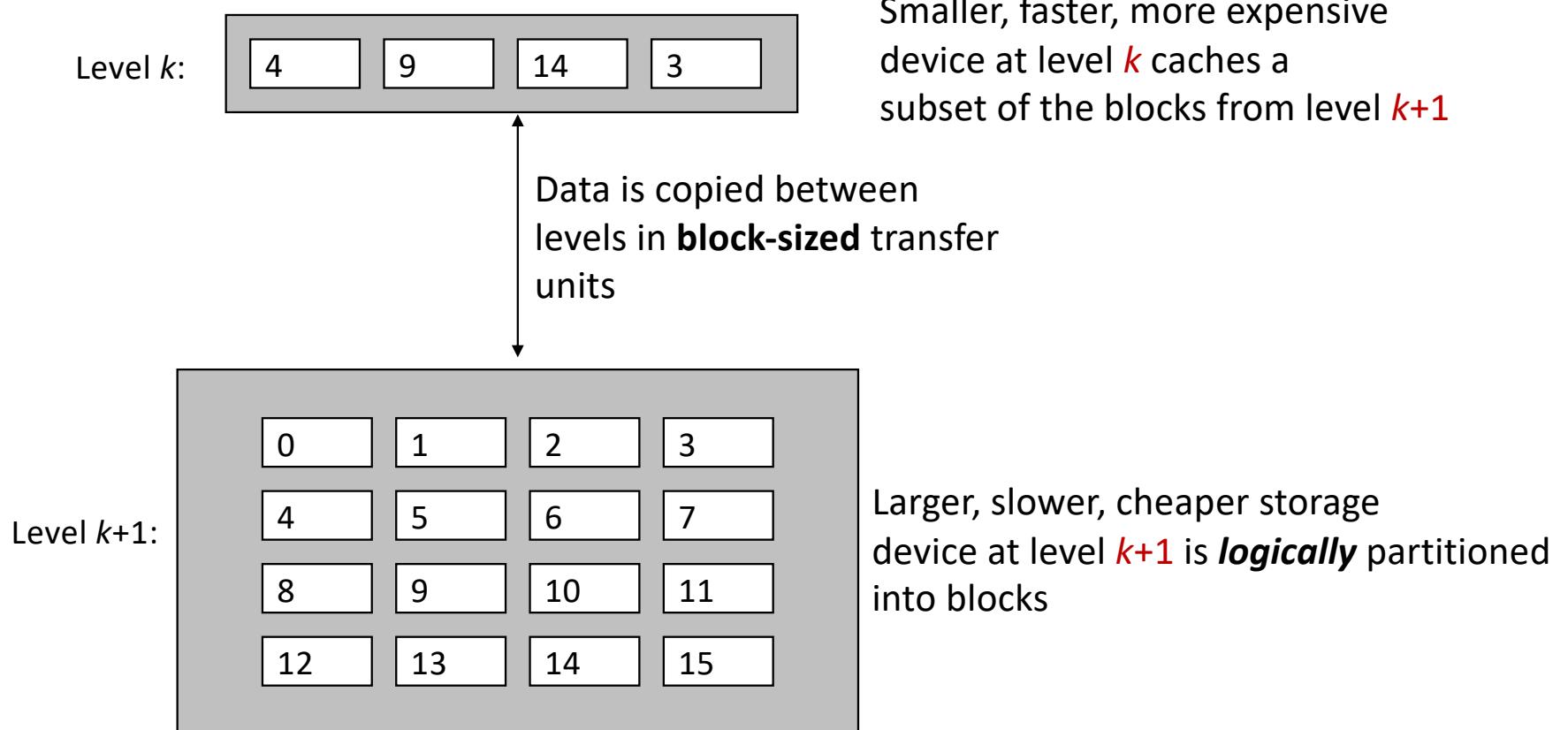
Real-Life Analogy

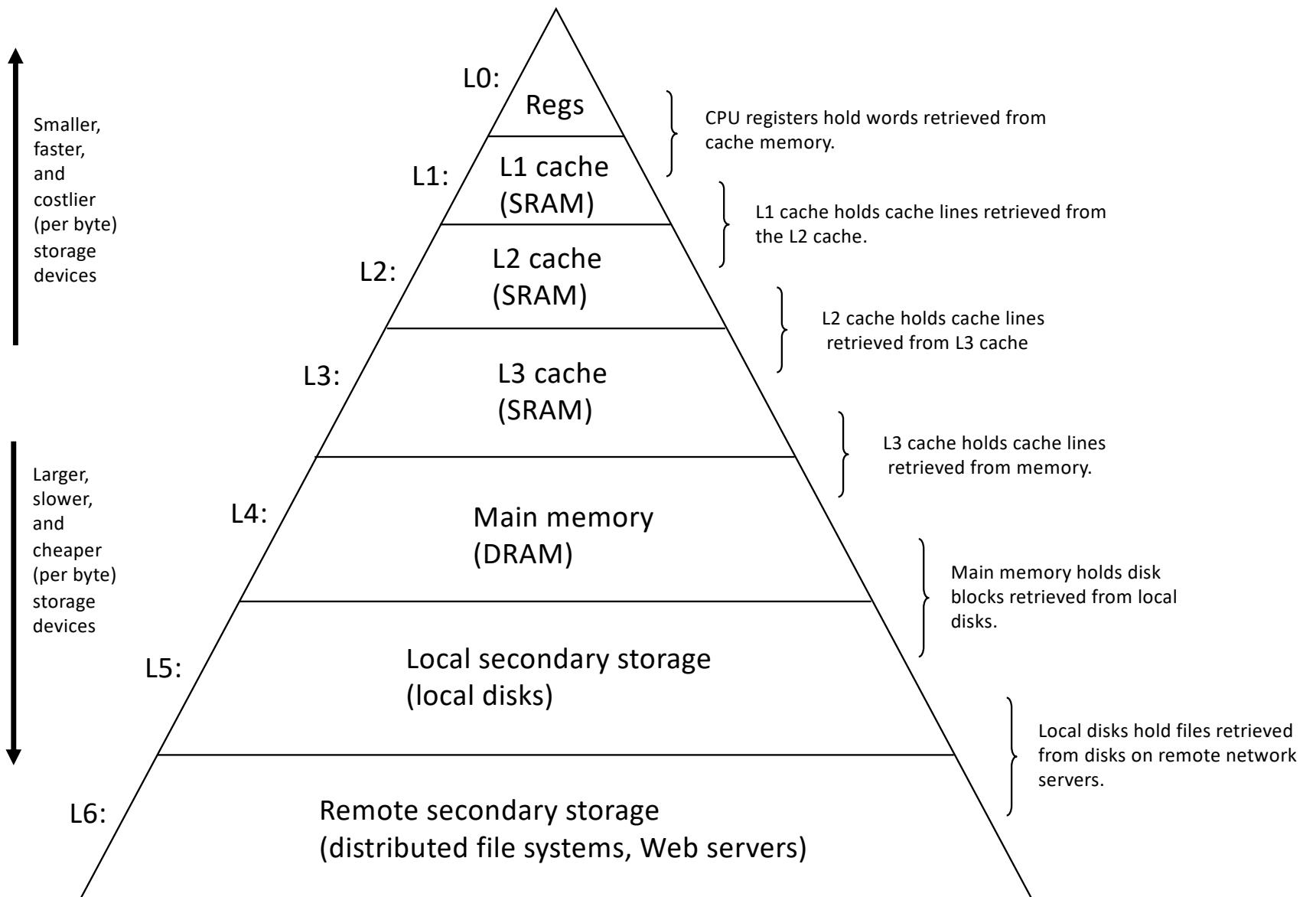


- In a library doing research on a subject
- *Temporal Locality*: If you brought a book to your desk to look up something, you are likely to look it up again
 - Keep the book on the desk instead of putting it back on the shelf
- *Spatial Locality*: Books on the same topic are put next to each other in a library
 - Bring several books on the subject to your desk to amortize the cost of walking to the shelf
- **Big Idea**: Desk serves as a small cache (*fast access*) for the large shelf (*slow access*)

Basic Idea of Caching

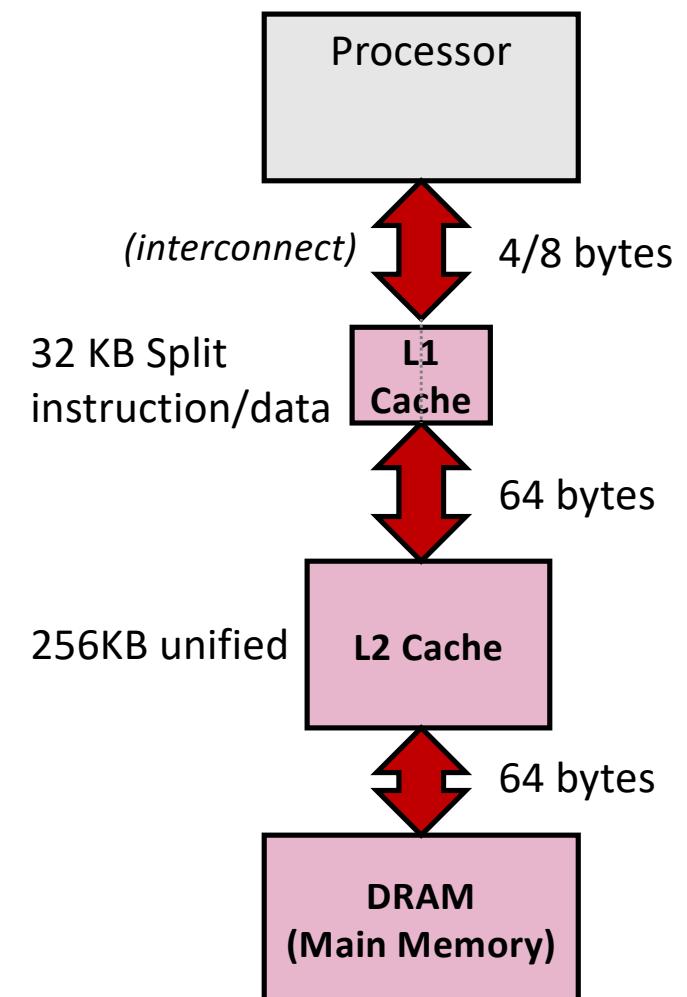
- **Cache:** A fast storage device that acts as a *staging area* for items stored in a larger, slower device





Transfer Granularity

- Data is always copied between level k and level $k+1$ in block-sized units
- Block size is fixed between any pair of adjacent levels in the hierarchy
 - Other pairs of levels can have different granularity
- Typical granularity
 - 32– 64 bits between RF and L1-Cache
 - 64-bytes between L1 and L2
 - 64 bytes between L2 and L3 (not shown)
 - 64 bytes between L3 (not shown) and DRAM main memory



Hit Rate (Ratio)

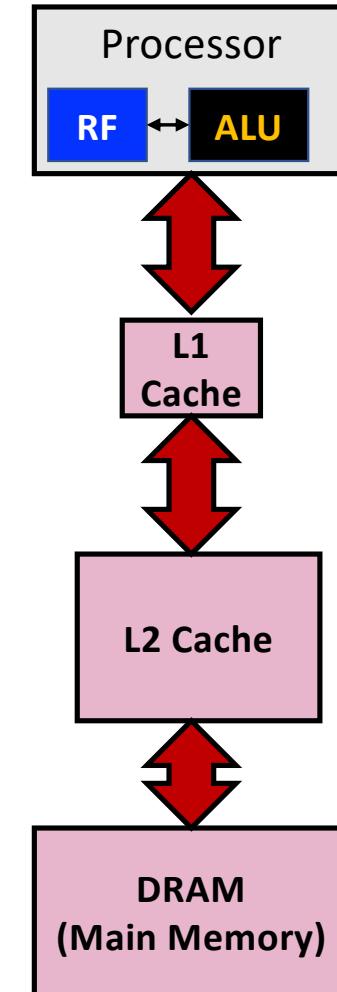
- Cache Hit
 - Program requests data word (d) at address A stored in level $k+1$
 - CPU first checks to see if the data word (d) is cached at level k
 - *If CPU finds d at level k , we call it a **cache hit***
 - *The CPU need not request d from level $k+1$*
- Hit Rate
 - Total # hits at level k **divided by** the total # accesses at level k

Miss Rate (Ratio)

- Cache Miss
 - Program requests data word (d) at address A stored in level $k+1$
 - CPU first checks to see if the data word (d) is cached at level k
 - *If CPU cannot find d at level k , we call it a **cache miss***
 - *The cache at level k needs to request d from level $k+1$*
- **Miss Rate:** Total # misses at level k divided by the total # accesses at level k
 - $1 - \text{hit-rate}$

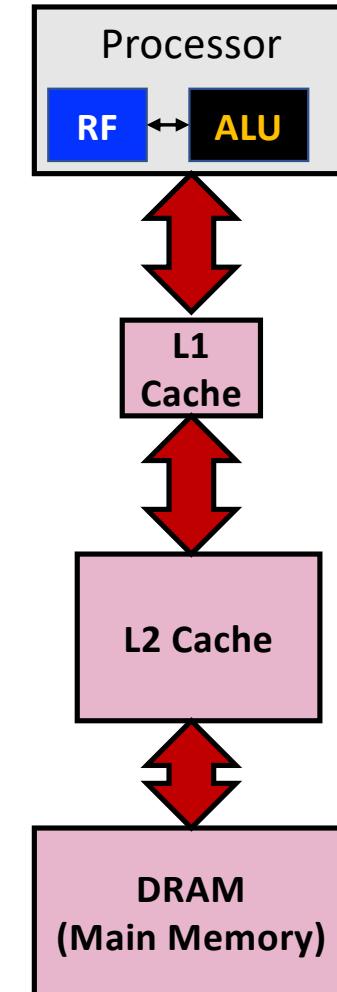
Hit Time

- The time required to:
 - Determine if a data word is cached at a particular level
 - Transfer the data word to the upper level
- Example: L1 Hit Time
 - Time it takes to determine if a data word is cached at Level 1
 - Time it takes to transfer the data word from L1-cache to the register file
 - Data movement from cache to RF includes both interconnect (wire) delay, read from SRAM cache, and write to the register



Miss Penalty

- In case of a cache miss at a particular level, the time required to
 - Retrieve the data word from the next level
 - Insert the data into the level that experienced the miss
- L1 Miss Penalty (assuming L2 Hit)
 - Time to find if data is in L2 (hit)
 - Transferring data from L2 to L1
- L1 Miss Penalty (assuming L2 Miss)
 - Time to find if data is in L2 (miss)
 - Transferring data from memory to L2
 - Transferring data from L2 to L1



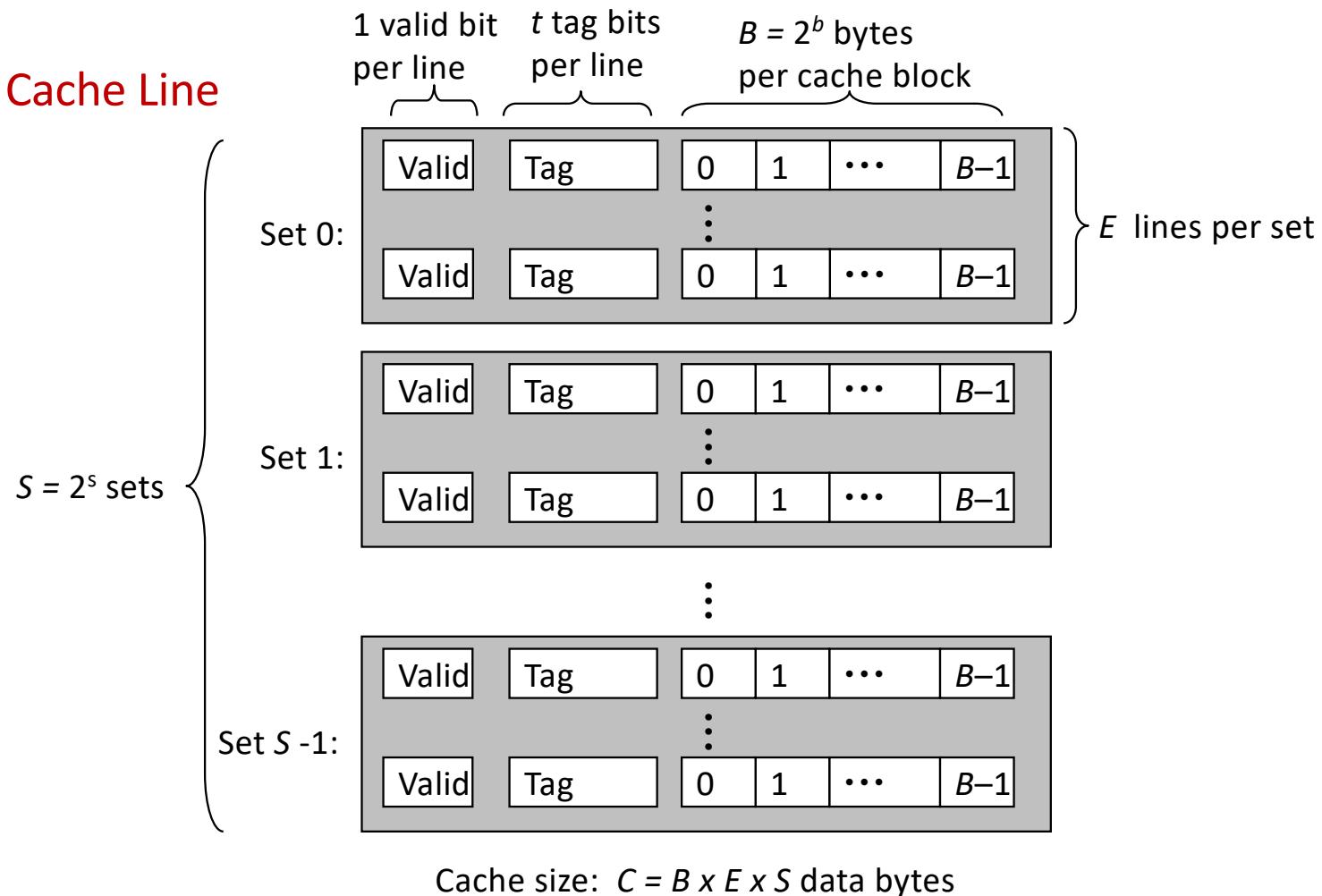
Generic Cache Organization

- Computer system with **m-bit** memory address
 - 2^m unique memory addresses
- A cache for such a system can only retain a small number of addresses
- When the CPU wants to read a word from address **A**, it first sends the request to cache memory
- Questions we will answer
 - How does the cache know whether it contains a copy of the word at address **A**?
 - How quickly can the cache determine whether it contains the copy of the word at address **A**?
 - How can we design the cache to maximize hit rate?

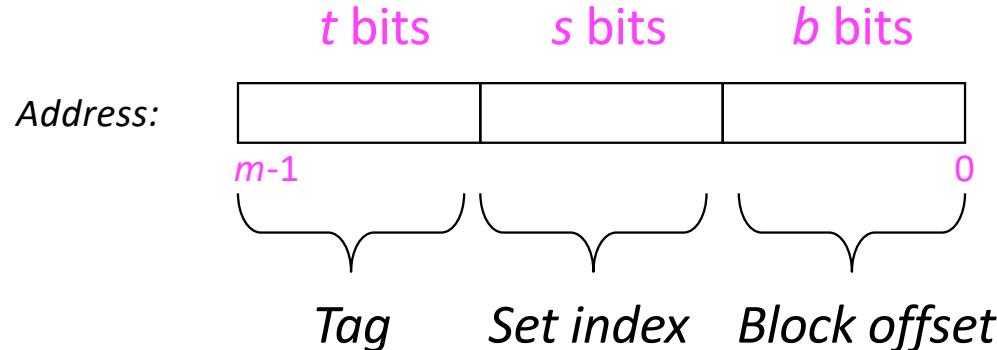
Generic Cache Organization

Note

Cache Block = Cache Line



Partitioning Address Bits



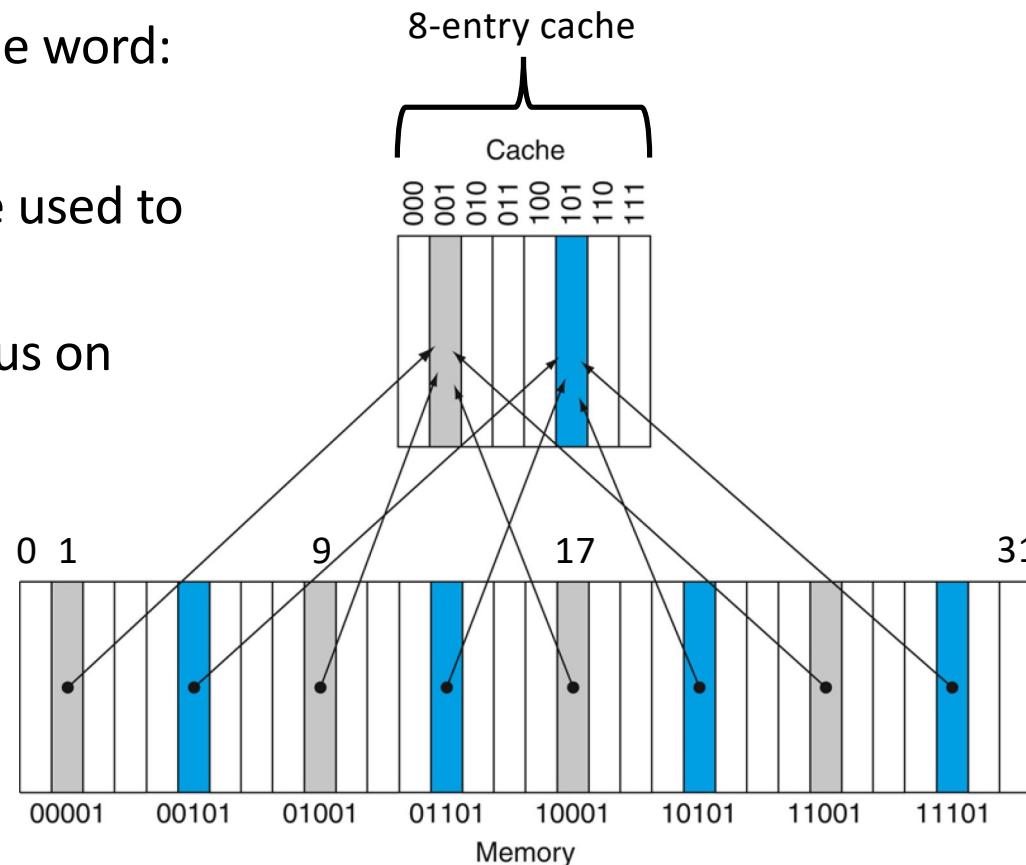
- Partition the m address bits in address \mathbf{A} into three fields
 - The s set index bits form an index into the array of S sets ($0, 1, 2, \dots$)
 - The t tag bits in \mathbf{A} determine which line in set contains the word
 - The b block offset bits give the offset of the word in the B -byte data block
- A line in the set contains the word if and only if
 - The *valid bit* is set
 - The *tag bits* in the line match the tag bits in the address \mathbf{A}

Direct-Mapped Cache

- A cache with exactly one line per set ($E = 1$)
- Each line in the address space can reside at only one location in the direct-mapped cache
- Any cache needs to perform three actions
 - Set selection
 - Line matching
 - Word extraction
- Set selection in direct-mapped cache
 - Block address **modulo** # blocks in cache
- Line matching
 - Compare the tag bits in the line to the tag bits in the address
- Word extraction: use the block offset bits to extract the word

Direct-Mapped Cache

- An address X maps to direct-mapped cache word:
 - $X \text{ modulo } 8$
- Low order 3 bits are used to index the cache
- To simplify, let's focus on block accesses



- Memory is partitioned into several fixed-sized blocks

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110		$10\textcolor{magenta}{110} \bmod 8 = \textcolor{magenta}{110}$
26	11010		$11\textcolor{magenta}{010} \bmod 8 = \textcolor{magenta}{010}$
22	10110		$10\textcolor{magenta}{110} \bmod 8 = \textcolor{magenta}{110}$
26	11010		$11\textcolor{magenta}{010} \bmod 8 = \textcolor{magenta}{010}$
16	10000		$10\textcolor{magenta}{000} \bmod 8 = \textcolor{magenta}{000}$
3	00011		$00\textcolor{magenta}{011} \bmod 8 = \textcolor{magenta}{011}$
16	10000		$10\textcolor{magenta}{000} \bmod 8 = \textcolor{magenta}{000}$
18	10010		$10\textcolor{magenta}{010} \bmod 8 = \textcolor{magenta}{010}$
16	10000		$10\textcolor{magenta}{000} \bmod 8 = \textcolor{magenta}{000}$

Initial State of the Cache

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010		$11010 \text{ mod } 8 = 010$
22	10110		$10110 \text{ mod } 8 = 110$
26	11010		$11010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$
3	00011		$00011 \text{ mod } 8 = 011$
16	10000		$10000 \text{ mod } 8 = 000$
18	10010		$10010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$

Miss 10110

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010	miss	$11010 \text{ mod } 8 = 010$
22	10110		$10110 \text{ mod } 8 = 110$
26	11010		$11010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$
3	00011		$00011 \text{ mod } 8 = 011$
16	10000		$10000 \text{ mod } 8 = 000$
18	10010		$10010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$

Miss 11010

Index	V	Tag	Data
000	0		
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010	miss	$11010 \text{ mod } 8 = 010$
22	10110	hit	$10110 \text{ mod } 8 = 110$
26	11010		$11010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$
3	00011		$00011 \text{ mod } 8 = 011$
16	10000		$10000 \text{ mod } 8 = 000$
18	10010		$10010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$

Hit 10110

Index	V	Tag	Data
000	0		
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010	miss	$11010 \text{ mod } 8 = 010$
22	10110	hit	$10110 \text{ mod } 8 = 110$
26	11010	hit	$11010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$
3	00011		$00011 \text{ mod } 8 = 011$
16	10000		$10000 \text{ mod } 8 = 000$
18	10010		$10010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$

Hit 11010

Index	V	Tag	Data
000	0		
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010	miss	$11010 \text{ mod } 8 = 010$
22	10110	hit	$10110 \text{ mod } 8 = 110$
26	11010	hit	$11010 \text{ mod } 8 = 010$
16	10000	miss	$10000 \text{ mod } 8 = 000$
3	00011		$00011 \text{ mod } 8 = 011$
16	10000		$10000 \text{ mod } 8 = 000$
18	10010		$10010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$

Miss 10000

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010	miss	$11010 \text{ mod } 8 = 010$
22	10110	hit	$10110 \text{ mod } 8 = 110$
26	11010	hit	$11010 \text{ mod } 8 = 010$
16	10000	miss	$10000 \text{ mod } 8 = 000$
3	00011	miss	$00011 \text{ mod } 8 = 011$
16	10000		$10000 \text{ mod } 8 = 000$
18	10010		$10010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$

Miss 00011

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010	miss	$11010 \text{ mod } 8 = 010$
22	10110	hit	$10110 \text{ mod } 8 = 110$
26	11010	hit	$11010 \text{ mod } 8 = 010$
16	10000	miss	$10000 \text{ mod } 8 = 000$
3	00011	miss	$00011 \text{ mod } 8 = 011$
16	10000	hit	$10000 \text{ mod } 8 = 000$
18	10010		$10010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$

Hit 10000

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010	miss	$11010 \text{ mod } 8 = 010$
22	10110	hit	$10110 \text{ mod } 8 = 110$
26	11010	hit	$11010 \text{ mod } 8 = 010$
16	10000	miss	$10000 \text{ mod } 8 = 000$
3	00011	miss	$00011 \text{ mod } 8 = 011$
16	10000	hit	$10000 \text{ mod } 8 = 000$
18	10010	miss	$10010 \text{ mod } 8 = 010$
16	10000		$10000 \text{ mod } 8 = 000$

Miss 10010

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	10	Mem[10010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

- Due to a conflict, we must remove a valid block from the cache

Accessing a Cache

Address (Decimal)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	$10110 \text{ mod } 8 = 110$
26	11010	miss	$11010 \text{ mod } 8 = 010$
22	10110	hit	$10110 \text{ mod } 8 = 110$
26	11010	hit	$11010 \text{ mod } 8 = 010$
16	10000	miss	$10000 \text{ mod } 8 = 000$
3	00011	miss	$00011 \text{ mod } 8 = 011$
16	10000	hit	$10000 \text{ mod } 8 = 000$
18	10010	miss	$10010 \text{ mod } 8 = 010$
16	10000	hit	$10000 \text{ mod } 8 = 000$

Hit 10000

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	10	Mem[10010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Line Replacement

- Once a miss, the cache needs to retrieve the block from the next level
- If there are valid blocks in the cache, then one of them must be evicted (in case of conflict) to make room for the new block
 - Which of the many possible blocks to evict is decided by the cache replacement policy
- For a direct-mapped cache, the replacement policy is trivial
 - Replace the current block (line) with the newly inserted block (line)

Example: Direct Mapped Cache

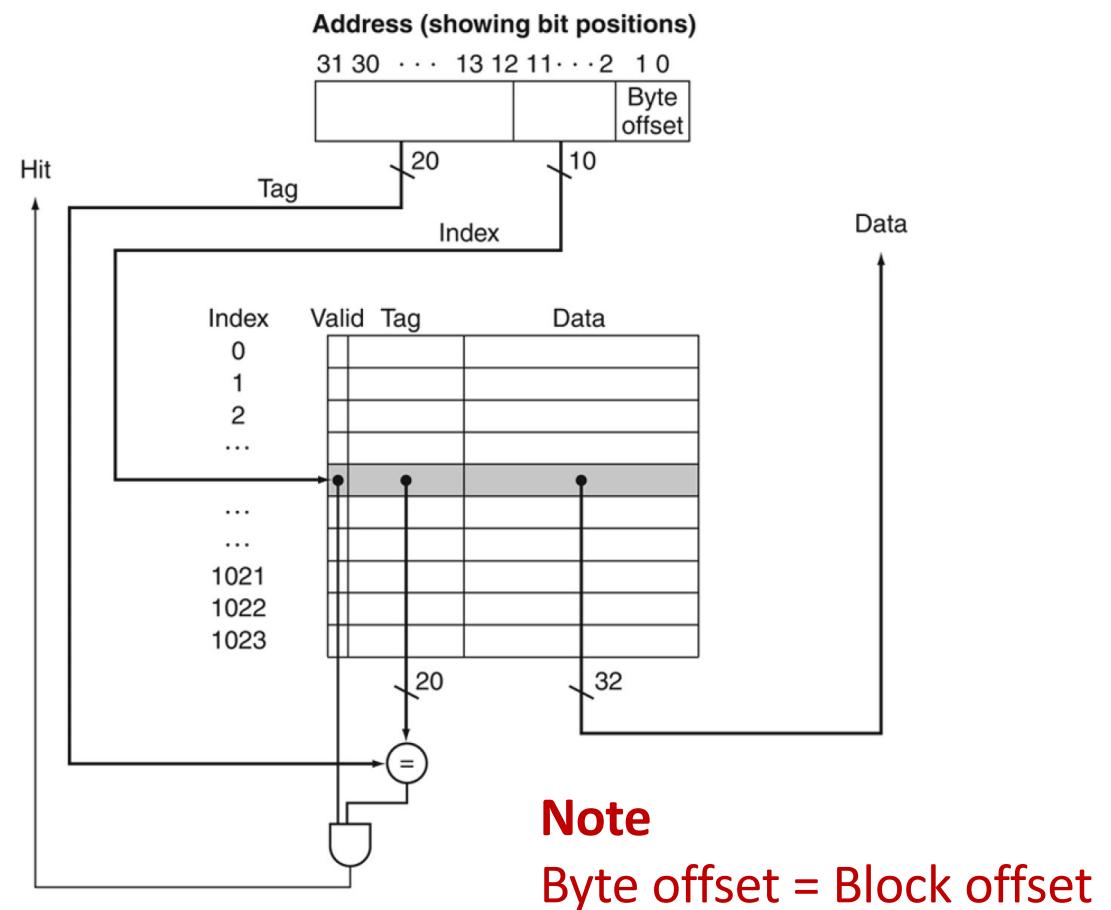
Block size = 1 word = 4 Bytes

Size of cache in bits

- 1024 words
- 4 KB (tag/valid not counted)
- Index = 10 bits
- Tag = $32 - 10 - 2 = 20$ bits

Operation

- valid bit is 1
- tag bits match
- cache hit!



Exercise

How many total bits are required for a direct-mapped cache (including valid and tag bits) with 16 KB of data and 4-word blocks, assuming a 32-bit address? The word size is 4 bytes.

Block size = 16 Bytes → 4 bits for block offset

Blocks = 16 KB/16 = 1024 → 10 bits for set index

bits for tag = $32 - 10 - 4 = 18$ bits for tag

bits for meta-data = 1 bit for valid

$$\text{total bits} = 2^{10} \times (4 \times 32 + 19) = 147 \text{ Kbits}$$

Exercise

Consider a cache with 64 blocks and a block size of 16 bytes. To which block number does byte address 1200 maps?

Address of block is given by byte address divided by bytes per block

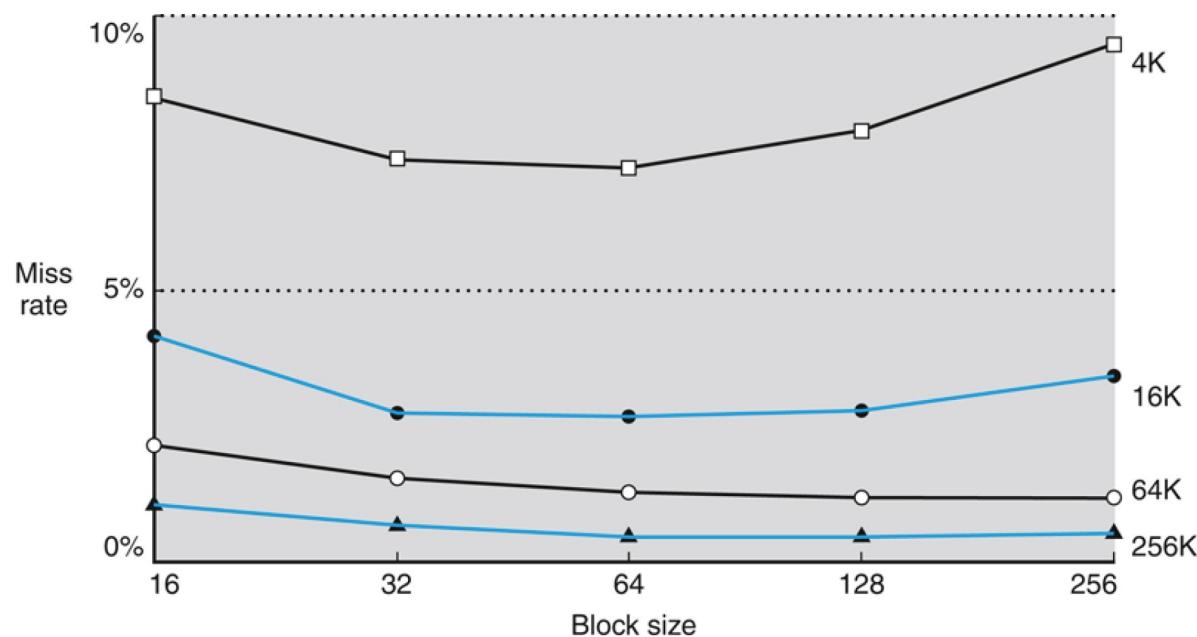
- $= 1200/16 = 75$ (interpretation: every 16 bytes, a new block starts)

The (cache) block is given by: Block address **modulo** (# blocks in cache)

- $= 75 \text{ modulo } 64 = 11$

Miss Rate vs. Block Size

- When the block size becomes a significant fraction of the cache size, then two aspects become critical
 - There is a small # blocks in the cache
 - There is a greater competition for the small # blocks



Flexible Block Placement

- **Direct-mapped:** *Each block can go to only location only*
 - Leads to a lot of conflict misses
 - **E-way set associative:** *Each block can go to one of E locations inside a set (also called n -way set-associative)*
 - **Fully associative:** *Each block can go to any location inside the cache*



Finding a Block with Associativity

- Set selection in E-way set-associate cache
 - Block address **modulo** # sets in cache
- Line matching
 - Compare the tag bits of each block in the set to the tag bits in the address
- Word extraction
 - Use the block offset bits to extract the word

Every Cache is a Set-Associative Cache!

For a cache with 8 entries

8 sets, 1 way

One-way set associative
(direct mapped)

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

4 sets, 2 way

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

2 sets, 4 way

Eight-way set associative (fully associative)

Tag	Data														

1 set, 8 way

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped**
 - 2-way set associative
 - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 2$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0						
8						
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative
 - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 2$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8						
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped**
 - 2-way set associative
 - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 2$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped**
 - 2-way set associative
 - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 2$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped**
 - 2-way set associative
 - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 2$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]			Mem[6]
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped**
 - 2-way set associative
 - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 2$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

Common Replacement Policies

- Set associative caches need a more elaborate replacement **policy**
- Random
 - *Pick a random block to make room for the newly fetched block*
- Least Recently Used (LRU)
 - *Pick a line that was last accessed the furthest in the past*
 - References: B, C, C, C, B, B, A → Replace C with A
 - C is the least recently used, i.e., oldest access time
 - B has better temporal locality
- Least Frequently Used (LFU)
 - *Pick a line that has been referenced the fewest times over some past time window*
- Most Recently Used (MRU)
 - *Pick a line that has been referenced the latest in the past*

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative**
 - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

2-way set associative, LRU replacement

Address mappings

0 → 0
6 → 0
8 → 0

Block address	Cache index	Hit/miss	Cache content after access	
			Set 0	Set 1
0	0	miss	Mem[0]	
8				
0				
6				
8				

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative**
 - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

2-way set associative, LRU replacement

Address mappings

0 → 0
6 → 0
8 → 0

Block address	Cache index	Hit/miss	Cache content after access	
			Set 0	Set 1
0	0	miss	Mem[0]	
8	0	miss	Mem[0]	Mem[8]
0				
6				
8				

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative**
 - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

2-way set associative, LRU replacement

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 0$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access	
			Set 0	Set 1
0	0	miss	Mem[0]	
8	0	miss	Mem[0]	Mem[8]
0	0	hit	Mem[0]	Mem[8]
6	0			
8	0			

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative**
 - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

2-way set associative, LRU replacement

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 0$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access	
			Set 0	Set 1
0	0	miss	Mem[0]	
8	0	miss	Mem[0]	Mem[8]
0	0	hit	Mem[0]	Mem[8]
6	0	miss	Mem[0]	Mem[6]
8	0			

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative**
 - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

2-way set associative, LRU replacement

Address mappings
 $0 \rightarrow 0$
 $6 \rightarrow 0$
 $8 \rightarrow 0$

Block address	Cache index	Hit/miss	Cache content after access	
			Set 0	Set 1
0	0	miss	Mem[0]	
8	0	miss	Mem[0]	Mem[8]
0	0	hit	Mem[0]	Mem[8]
6	0	miss	Mem[0]	Mem[6]
8	0	miss	Mem[8]	Mem[6]

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative
 - Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8						
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative
 - Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative
 - Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative
 - Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
 - Direct-mapped
 - 2-way set associative
 - Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

Performance Comparison

- In terms of speed (hit time)
 - Direct mapped cache is the fastest
 - Only one place to look for the requested block
 - Fully-associative cache is the slowest
 - Many places to search for the requested block
 - n-way set-associative cache is a compromise
- In terms of miss rate
 - Direct mapped cache is the worst
 - Many conflicts due to lack of flexible placement
 - Fully-associative cache is the best
 - Flexibility is very high
 - n-way set-associative cache is a compromise

Common Addressing Scheme



- 2-way set-associative
 - 64-byte block/line size
 - 2048 blocks (1024 sets)
 - 32-bit address
 - How many bits are needed for the tag, index, and block offset?



Practice

A cache has 4096 blocks and a 16-byte block size. Assuming 32-bit addresses, find the total number of sets and the total number of tag bits for caches that are (1) direct-mapped (2) 2-way set associative (3) 4-way set-associative (4) fully associative.

bits for index + tag = $32 - 4 = 28$ (4-word block means 16 addressable bytes in each block)

sets in direct-mapped cache = $4096 = 12\text{-bit index and } 16\text{-bit tags}$

tag bits = $16 * 4096 = 66 \text{ K tag bits}$

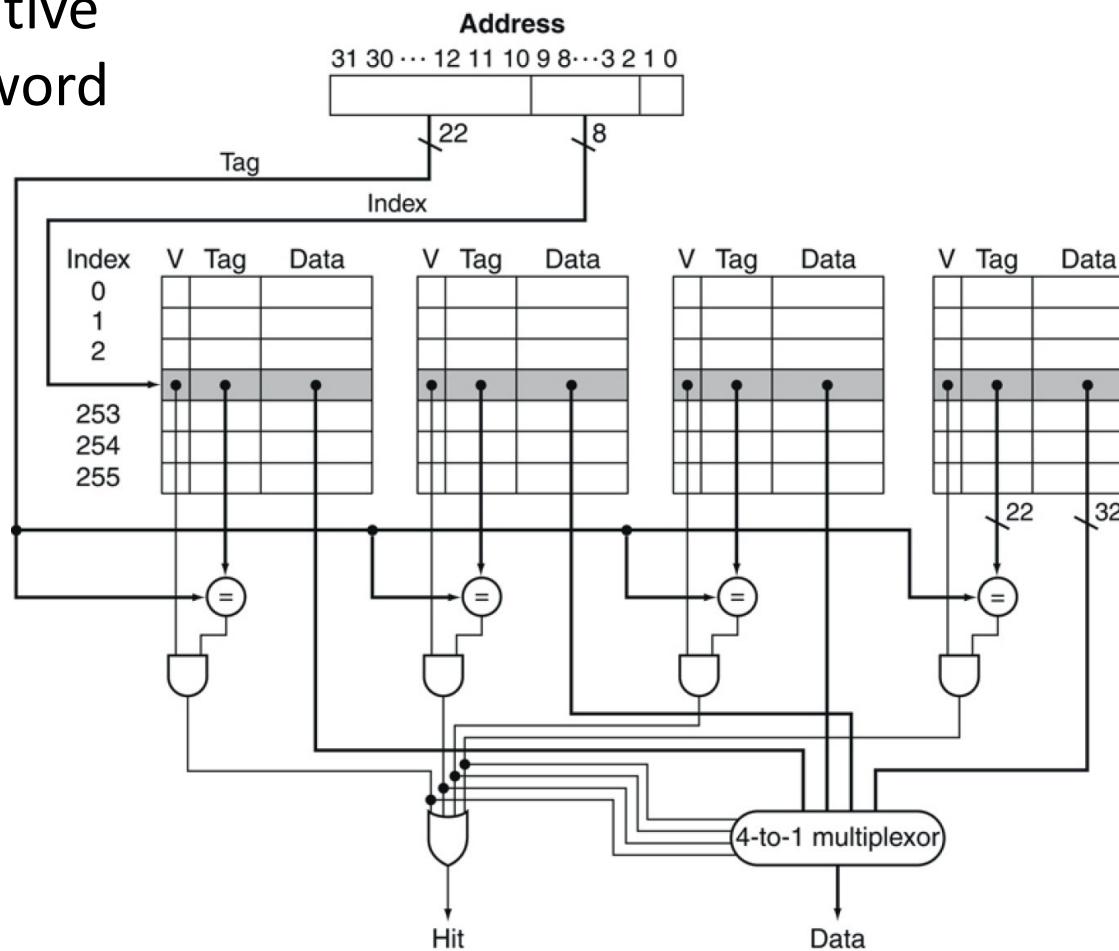
Each degree of associativity decreases the # sets by 2 and decreases the # bits to index the cache by 1 and increases the tag bits by 1. A fully-associative cache has one set.

- 2-way cache: 2048 sets, tag bits = $(28 - 11) * 2 * 2048 = 70 \text{ Kbits}$
- 4-way cache: 1024 sets, tag bits = $(28 - 10) * 4 * 1024 = 74 \text{ Kbits}$
- Fully associative: $28 * 4096 * 1 = 115 \text{ Kbits}$

Architecture: Set-Associative Cache

4-way set-associative

Block size = one word
= 4 Bytes



Area Comparison

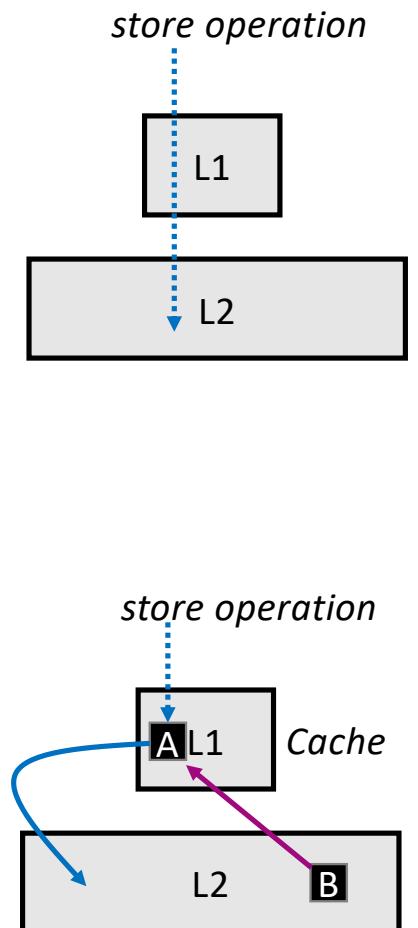
- Today's SRAM caches are on the same die as the processor
 - We call these on-chip or on-die caches
 - Reducing the area complexity is important so we can have a cache hierarchy with high capacity
- In terms of die area (logic complexity)
 - Direct mapped cache takes up the least die area as there is only one comparator and no multiplexer
 - Increasing associativity increases the # comparators and the width of the multiplexer

Issues with Writes

- Reads (loads) are relatively straightforward
- Writes (stores) open up a number of interesting problems
- Write hit
 - On a write to a cache line (write hit), what should we do about updating the copy of the cache line in the next lower level?
- Write miss
 - To allocate or not to allocate a block on a write miss

Write-Through vs. Write-Back

- Write-through
 - Immediately write the updated cache block to the next lower level
 - Simple to implement
 - Write traffic on the bus with every write
- Write-back
 - On a write hit, update the block in the cache only
 - Defer the write to the next lower level as much as possible
 - Write the modified block (A) to the next level only when it is evicted from the cache by the replacement algorithm
 - Need an additional meta-data bit: *dirty bit*



Allocating Blocks on Writes

- Write-Allocate
 - Load the corresponding block from the next lower level and then update the block
 - Exploits spatial locality
 - Every miss results in a block transfer from the next lower level
- No-Write-Allocate
 - Bypass the cache and write the word directly to the next lower level
- Popular choices
 - Write-through + no-write-allocate
 - Write-back + write-allocate

Writeback Buffer

- On a cache miss, a write-back cache needs to perform two main tasks involving data movement
 - Write the replacement candidate to the next lower level of the memory hierarchy
 - Bring the cache block from the next lower level and insert it in the level that initiated the miss
- Which one should the cache do first?
- There is only one answer as the cache cannot overwrite the modified block (*write the replacement candidate first*)
 - **Optimization:** Prioritize returning data word to the processor
 - Move the evicted block to a special ***writeback buffer*** and first bring the data from the next level into the cache

The 3C Model

- Compulsory (Cold) Misses
 - These misses are caused by the first access to a block that has never been in the cache
- Capacity Misses
 - These are cache misses caused when the cache cannot contain all the blocks needed during program execution
 - One way to think: *if we end up replacing blocks with a fully-associative cache*
- Conflict (Collision) Misses
 - Misses that occur when multiple blocks compete for the same set
 - One way to think: *eliminated by a fully-associative cache of the same size*

Software Interaction with Caches

```
struct point_vector {  
    int point_x;  
    int point_y;  
};  
struct point_vector p[1024]  
...  
...  
while (i < 1024) {  
    p[i].point_x *= 10;  
}
```

Question: Can we transform the array of structs (AoS) representation into a format that results in better locality of reference?

```
struct point_vector {  
    int point_x[1024];  
    int point_y[1024];  
};  
struct point_vector p;  
...  
...  
while (i < 1024) {  
    p.point_x[i] *= 10;  
}
```

Struct of Arrays (SoA)



SoA vs AoS

Array of Structs

point_x	point_y	point_x	point_y	point_x	point_y	point_x	point_y	...
---------	---------	---------	---------	---------	---------	---------	---------	-----

Struct of Arrays

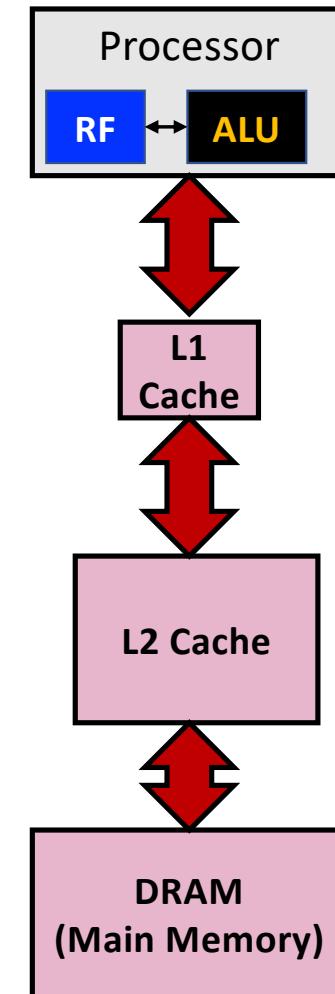
point_x	...							
---------	---------	---------	---------	---------	---------	---------	---------	-----

Cache line

If we are only interested in manipulating all points of a single type (x in our case) then the SoA layout can bring from memory and locate more points in a single cache line

Exercise (Offline)

- Find the average memory access time given
 - L1 Hit Time = 1 Cycle
 - L1 Hit Rate = 80%
 - L2 Hit Time = 10 cycles
 - L2 Hit Rate = 50%
 - L2 Miss Penalty = 200 cycles



Cache Size vs. Associativity

- If we use a 2-way set associative cache w/t eight blocks (instead of four blocks), and compare its miss rate to a fully-associative cache with four blocks, we find that
 - There are no replacements in the 2-way set associative cache with eight blocks
 - Its miss rate is the same as the fully associative cache
- If we repeat the exercise with 16 blocks for all three caches, we find that
 - All three caches will have the same miss rate
- **Bottomline**
 - Cache size and associativity both determine cache performance