COMP2300-COMP6300-ENGN2219
Computer Organization &
Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian
National
University

Shoaib Akram

School of Computing, (Jan 2020 –)

Ph.D., 2019

Teaching: COMP2300, COMP2310, Computer Microarchitecture (3710)

Interests: Hardware/software interaction, storage systems

- Interesting time to learn and research about computer systems
- Technology limits vs. societal demand for more compute power; cost and energy efficiency; and sustainability
- Big Data, AI/ML, social media platforms, search engines, communications, mobile apps, drones, among others.

My current research focus is on efficiently processing big datasets

Logistics

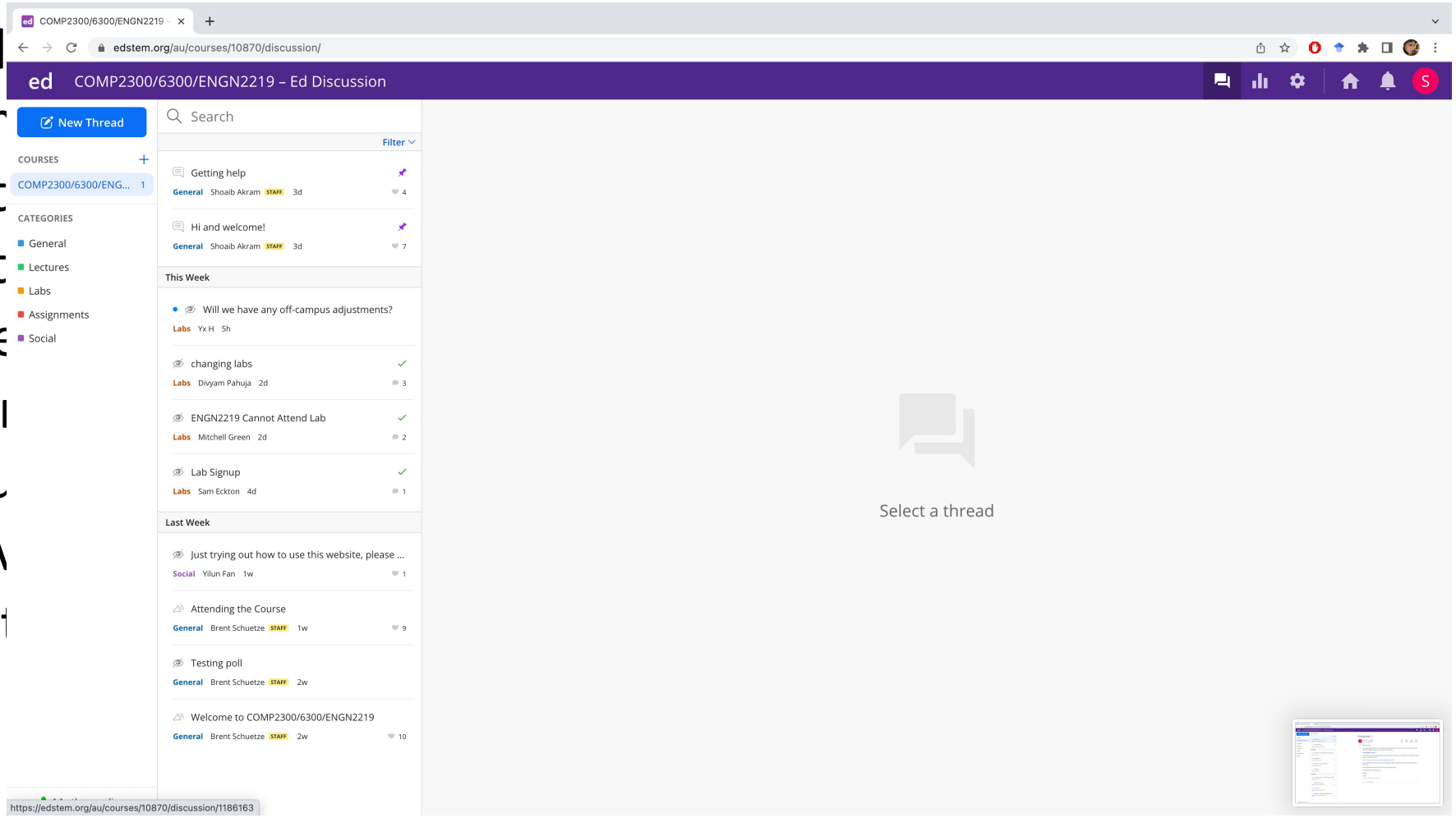
- Course webpage: https://comp.anu.edu.au/courses/comp2300/
- **Lectures (**on the website**)**
  - Lecture slides
  - Lecture videos (I will do my own recording)
- **Policies**
  - General conduct, assignment submissions, support, management, grading, late submissions
- **Resources**
  - Frequently asked questions
  - Writing design documents
  - Stuff needed to finish the assignments

Communication

- We will use **Ed** for
  - Announcements
  - Addressing your concerns
  - Answering your questions
- Students are added automatically
  - If not send an email to **comp2300@anu.edu.au**

Ed

- We will
  - Anr
    lect
    lect
- Ask que
- Solutio
- Ask you
- Ask priv
- Studen



Communication

- The course email is an alternative form of communication
  - **comp2300@anu.edu.au**

Tutorials/Labs

- Labs are a critical component of this course (one every week)
- Handouts should be posted on the website
- First six labs
  - Assignment 1 (30%)
- Last six labs
  - Assignment 2 (30%)

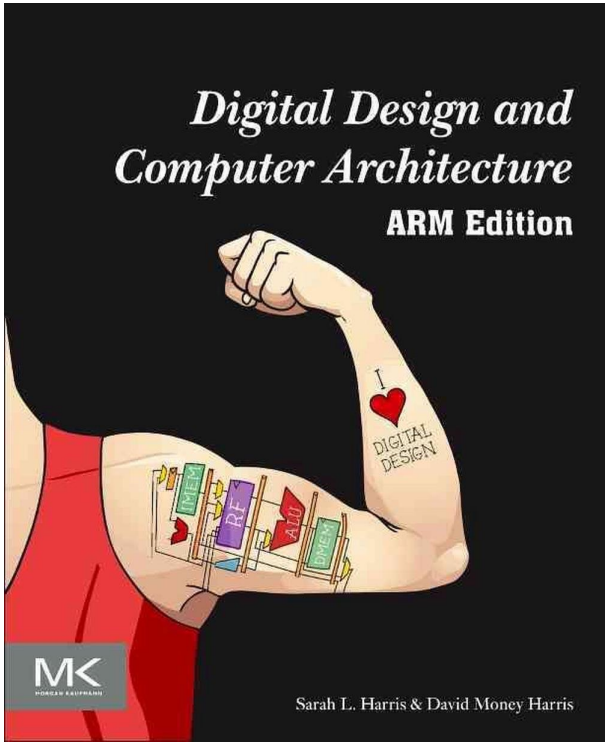
Assessments

- Two assignments (60%)
  - CPU design assignment (30%)
  - ARM Programming assignment (30%)
- Final Exam (40%)

Assignment Submission

- 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
- Extensions
  - https://comp.anu.edu.au/courses/comp2300/policies/#extensions

Textbook



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

COMP2300

- Asks a question
  - Let’s give it a shot!

COMP2300

- How does a computer **actually** work?

COMP2300

- How does a computer work **under the hood**?

COMP2300

- How does a computer perform a useful task?

COMP2300

- How does a computer perform a wide variety of useful tasks?

COMP2300

- How do we make a computer perform a wide variety of useful tasks?

COMP2300

- How do we make electrons perform a wide variety of useful tasks?

COMP2300

- How do we make electrons execute a wide variety of useful programs?

- Computer Organization & Program Execution

How do we make electrons do the work?

Problem Statement: “Save the planet”

How do we make electrons do the work?

Problem Statement: “Save the planet”

The Algorithm

How do we make electrons do the work?

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

How do we make electrons do the work?

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

Instruction Set Architecture (ISA)

How do we make electrons do the work?

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

Instruction Set Architecture (ISA)

Microarchitecture

How do we make electrons do the work?

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

Instruction Set Architecture (ISA)

Microarchitecture

Circuits

How do we make electrons do the work?

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

Instruction Set Architecture (ISA)

Microarchitecture

Circuits

Devices

How do we make electrons do the work?

- Using a sequence of systematic transformations
  - Developed over six decades
- Each step must be studied and improved for the whole stack to work efficiently

Transformation Hierarchy

- We call the steps of the process: Levels of transformation OR Transformation hierarchy
- At each level of the stack, we have choices
  - **Language:** Java, Python, Ruby, Scala, C++, C#
  - **ISA:** ARM, x86, SPARC, PowerPC, RISC-V
  - **Microarchitecture:** Intel, AMD, IBM
- If we ignore any of the steps, then we cannot
  - Make the best use of computer systems
  - Build the **best** system for a set of programs

Problem

Algorithm

Program

Architecture

micro-arch

circuits

devices

Transformation Hierarchy

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

Instruction Set Architecture (ISA)

Microarchitecture

Circuits

Devices

Program Execution

Transformation Hierarchy

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

Instruction Set Architecture (ISA)

Microarchitecture

Circuits

Devices

Program Execution

Computer Organization

Transformation Hierarchy

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

Instruction Set Architecture (ISA)

Microarchitecture

Circuits

Devices

Program Execution

Computer Organization

Computer Architecture

Transformation Hierarchy & Us

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

2300 Instruction Set Architecture (ISA)

2300 Microarchitecture

2300 Circuits

Devices

Program Execution

Computer Organization

Computer Architecture

Transformation Hierarchy & Us

Problem Statement: “Save the planet”

The Algorithm

Program in a High-Level Language

2310 Compiler and Third-Party Libraires/Binaries

2310 Operating System

2300 Instruction Set Architecture (ISA)

2300 Microarchitecture

2300 Circuits

Devices

Program Execution

Computer Architecture

Computer Organization

Hardware and Software

Problem Statement: “Save the planet”

Software	The Algorithm
	Program in a High-Level Language
	Compiler and Third-Party Libraires/Binaries
	Operating System

ISA = Hw/Sw

boundary/interface Instruction Set Architecture (ISA)

	Microarchitecture
	Circuits
Hardware	Devices 

ISA and Microarchitecture

- **ISA:** Specification of a set of definite instructions that the computer can carry out
  - ADD, MUL, DIV, MOV
- **Microarchitecture:** Implementation of the ISA using circuits

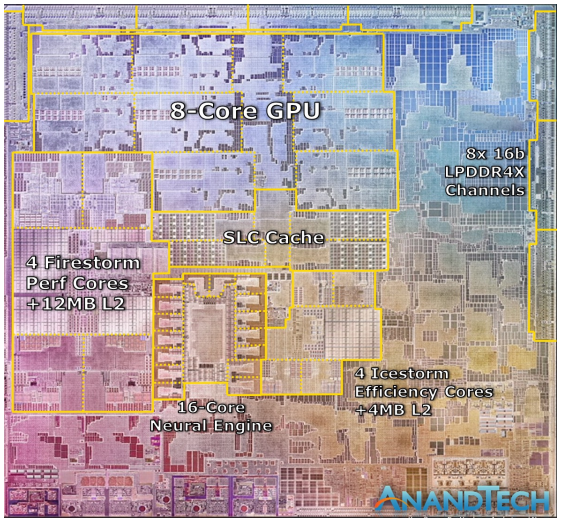
Two Recurring Themes

- The notion of abstraction
- Hardware versus software

The Notion of Abstraction

- **Abstraction:** Know components from a high level of detail

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



Apple M1 Chip
Billions of transistors
All working in parallel

The Notion of Abstraction

- **Abstraction**: View the world from a higher level
- Focus on the important aspects
  - Input? Output? X = ADD or MULTIPLY

- Raise the level of abstraction for productivity and efficiency
- But what if the world below does not work as expected?
  - To deal with it, we need to go below the abstraction layer
- **Deconstruction**: To un-abstract when needed
  - Important skill

The Notion of Abstraction

- We will use this theme a lot!
  - Each layer in the transformation hierarchy is an abstraction layer!

Problem

Algorithm

Program

Architecture

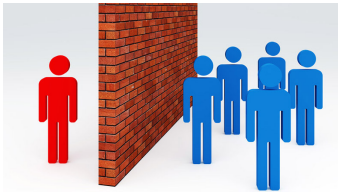
micro-arch

circuits

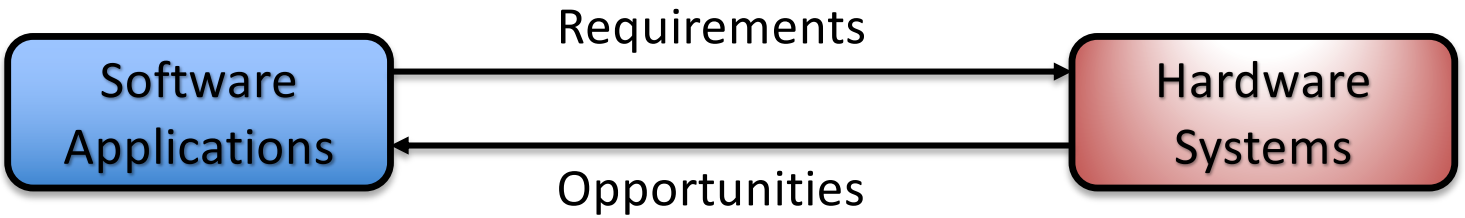
devices

Hardware versus Software

- Hardware versus software
  - Hardware: Physical computer
  - Software: Programs, operating systems, compilers
- **One view:** Ok to be an expert at one of these



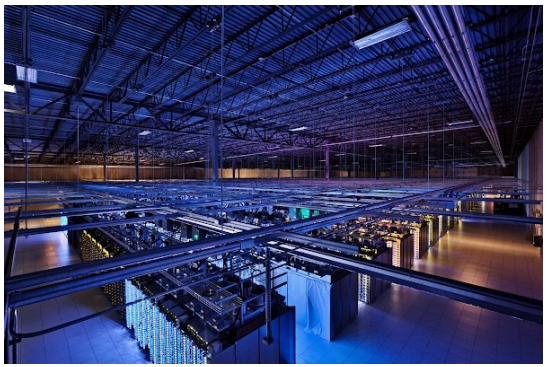
- Hw and Sw: Two parts of the computer system
  - **COMP2300 view:** Knowing the capabilities/limitations of each leads to better overall systems



Two Key Ideas

- Key components of a computer are the same
- All computers can compute the same things

Idea 1: Key components are same



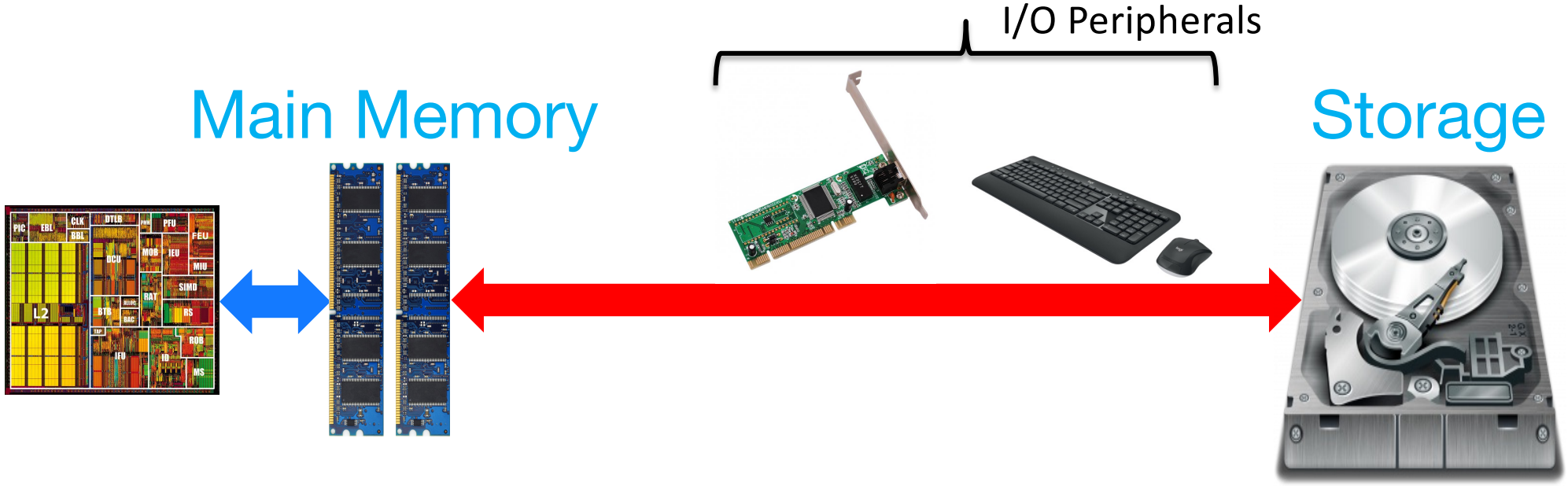
Council Bluffs, Iowa data center, Google (115, 000 sq. feet)

Self-flying nano drone
94 milli-watts

Research server for my students with special memory & storage devices

A Canonical Computer System

- **Most computer systems can be viewed as below**
  - Key resources: CPU, memory, I/O devices, storage
  - CPU (processor/microprocessor) does the actual computation
  - Processor can access memory much faster than storage



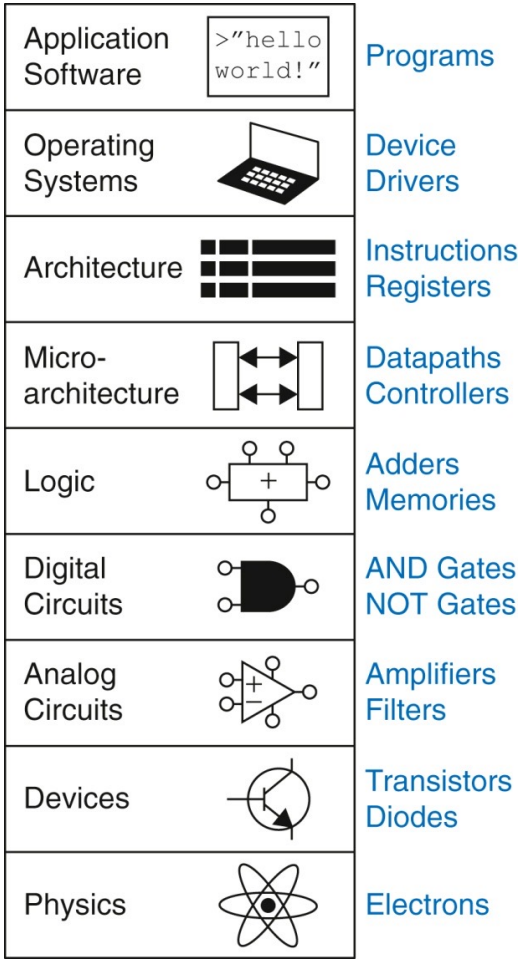
Idea 1: Key components are same

- **Key Idea 1: 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 (disks or SSDs) for storing long-term or persistent information
  - **I/O devices** to communicate with the external environment
    - Sensors
    - Peripherals

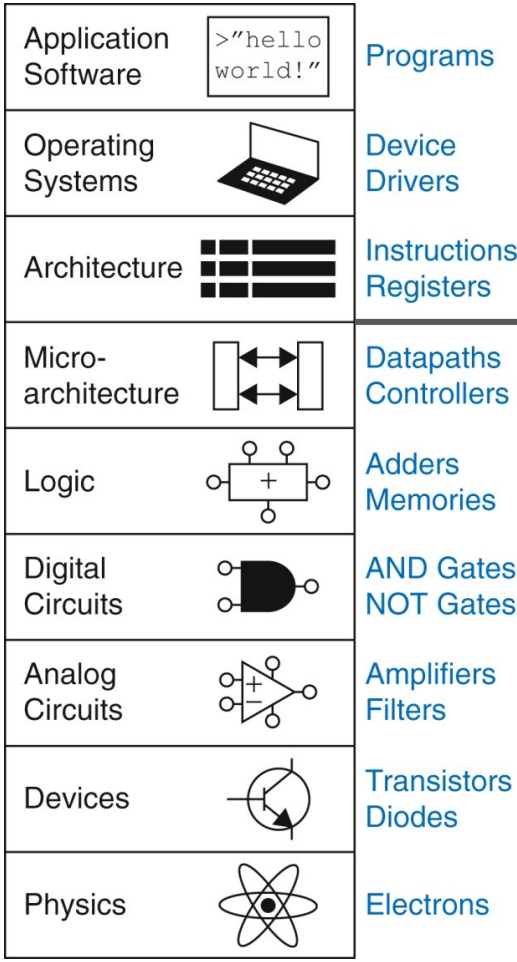
Idea 2: They all can solve the same problems

- **Key Idea 2: All computers regardless of size, cost, and speed can compute the same things if they are given enough time and memory**
  - Anything a fast computer can do, a slow computer can do
  - Let's explore this idea further ....

Program Ex.


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

Program Ex.

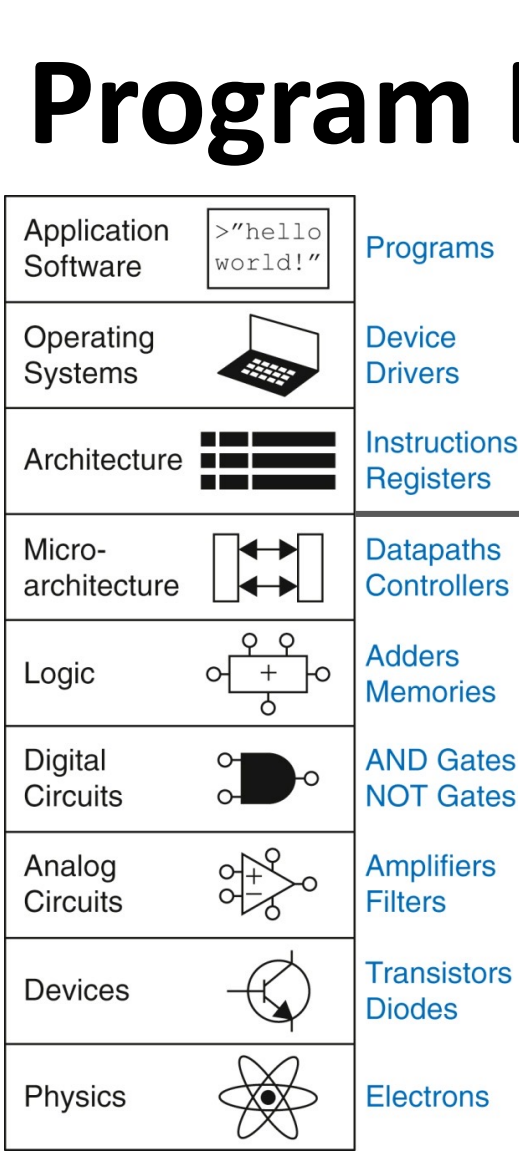


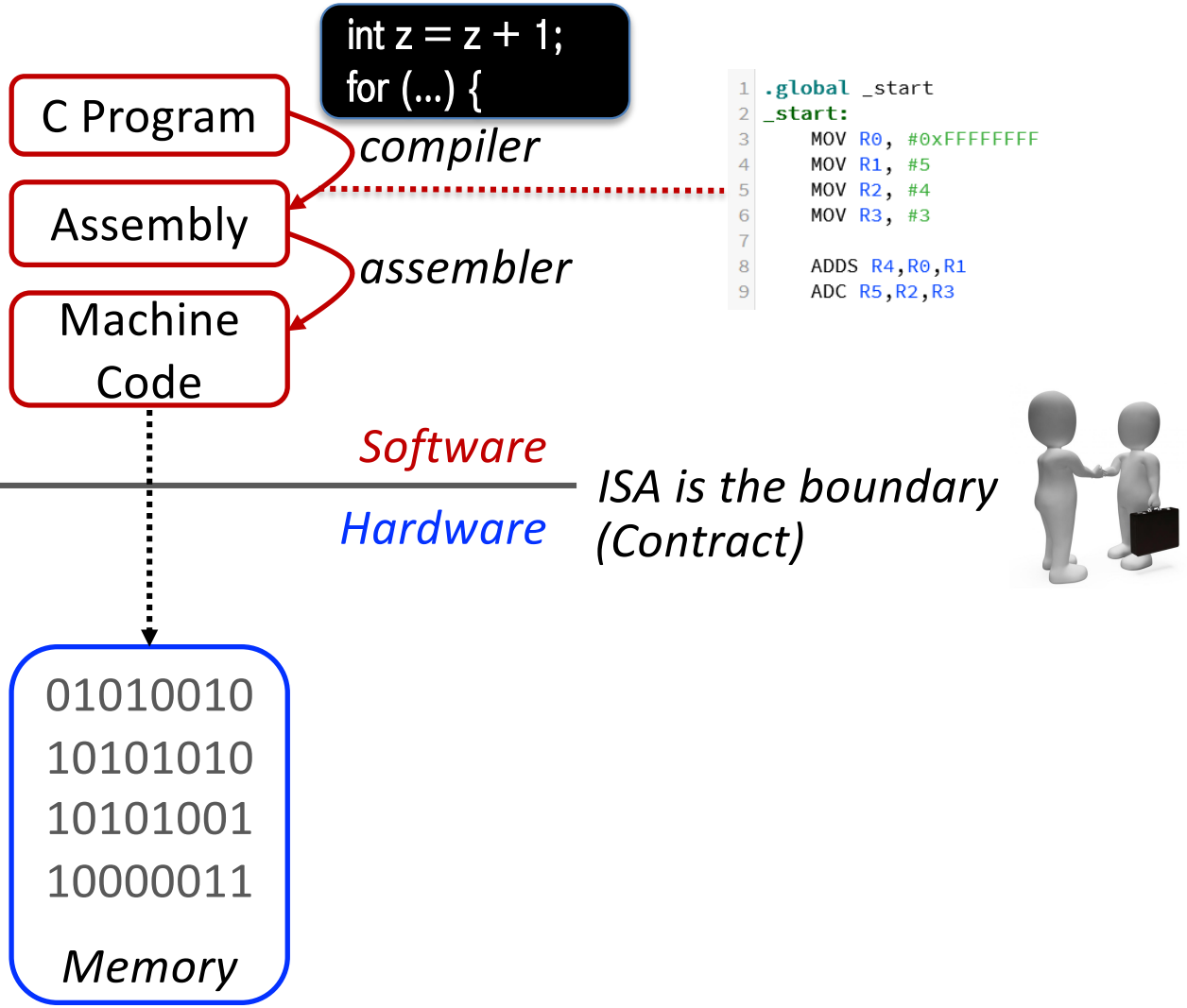
Software

Hardware

ISA is the boundary (Contract)

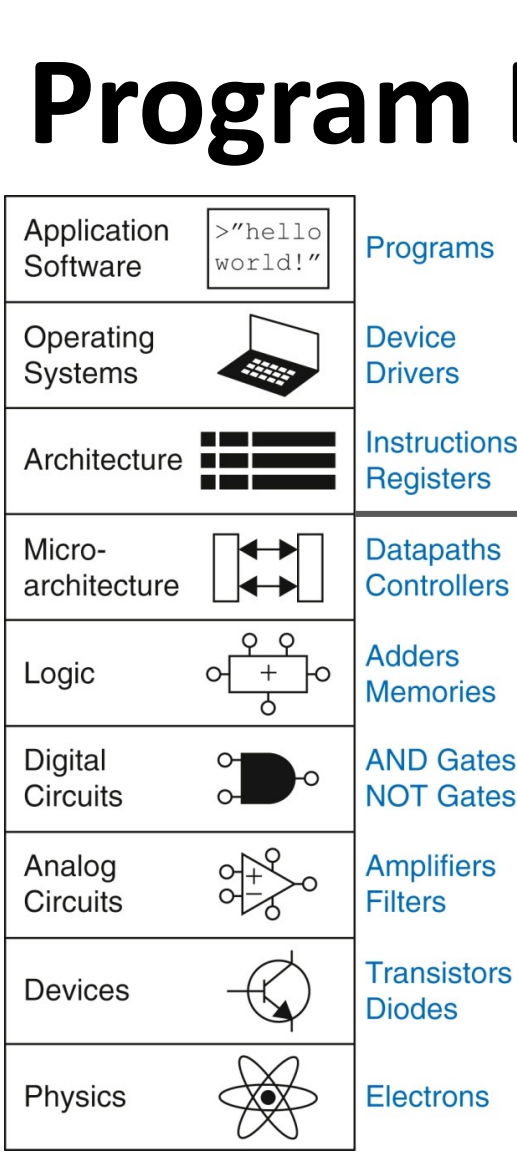
Program Ex.

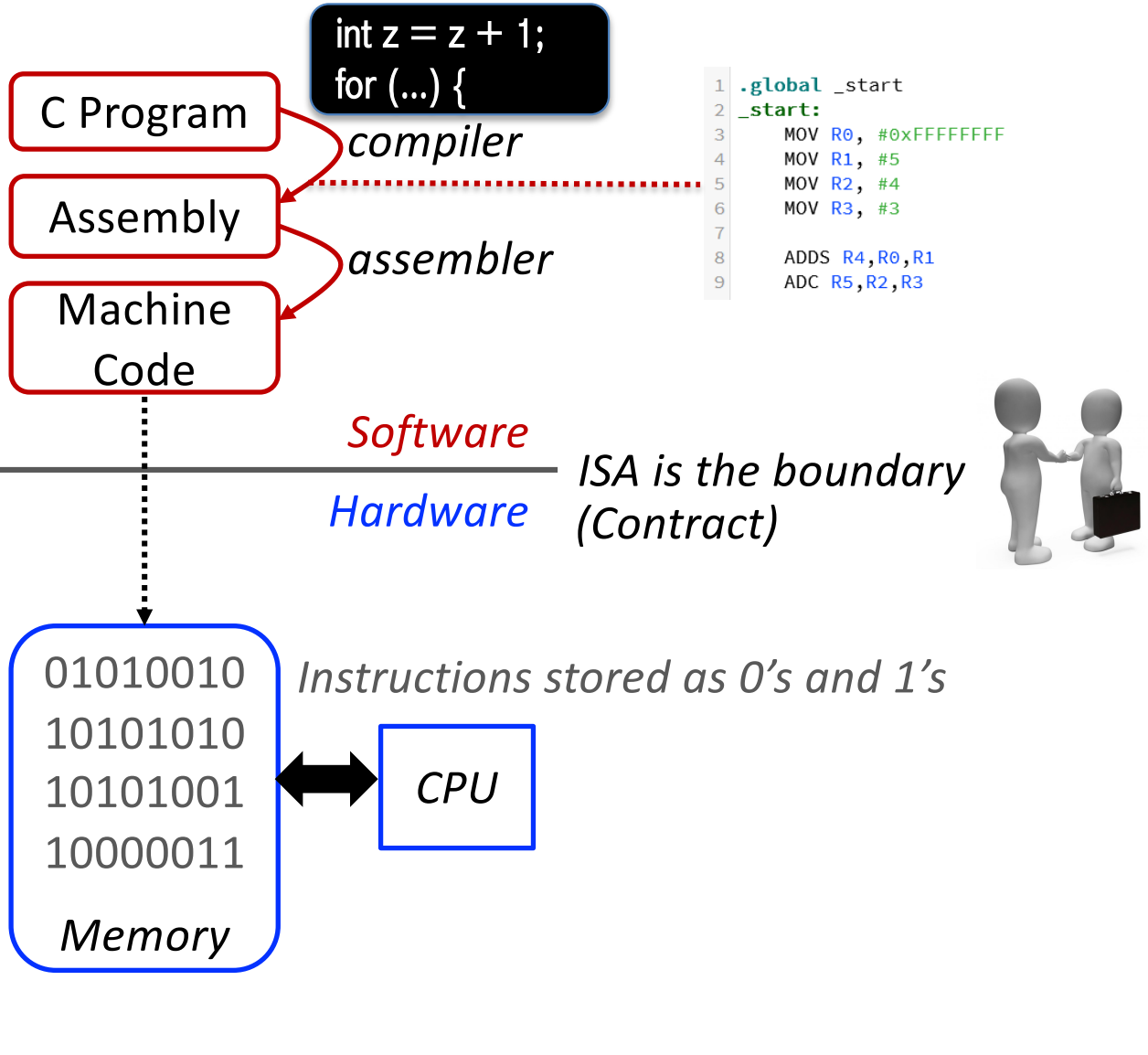




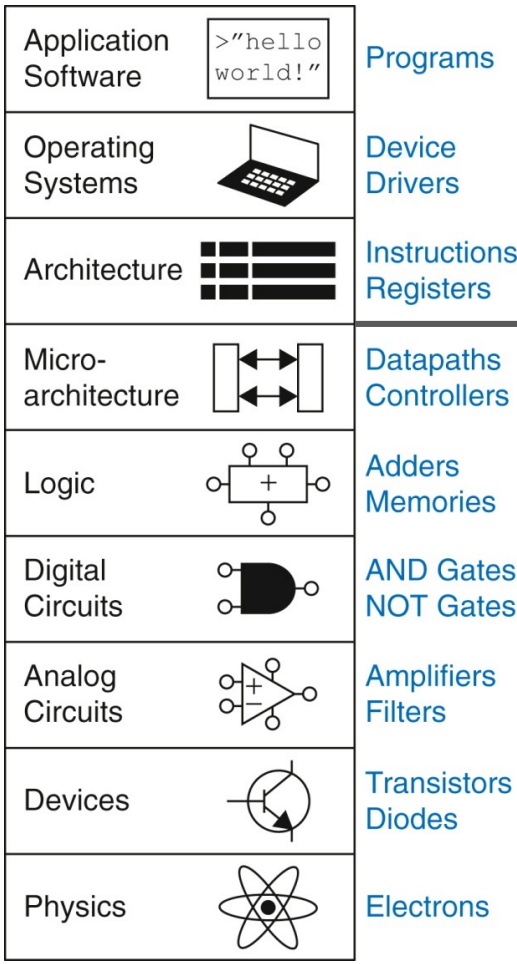

```
1 .global _start
2 _start:
3     MOV R0, #0xFFFFFFFF
4     MOV R1, #5
5     MOV R2, #4
6     MOV R3, #3
7
8     ADDS R4,R0,R1
9     ADC R5,R2,R3
```

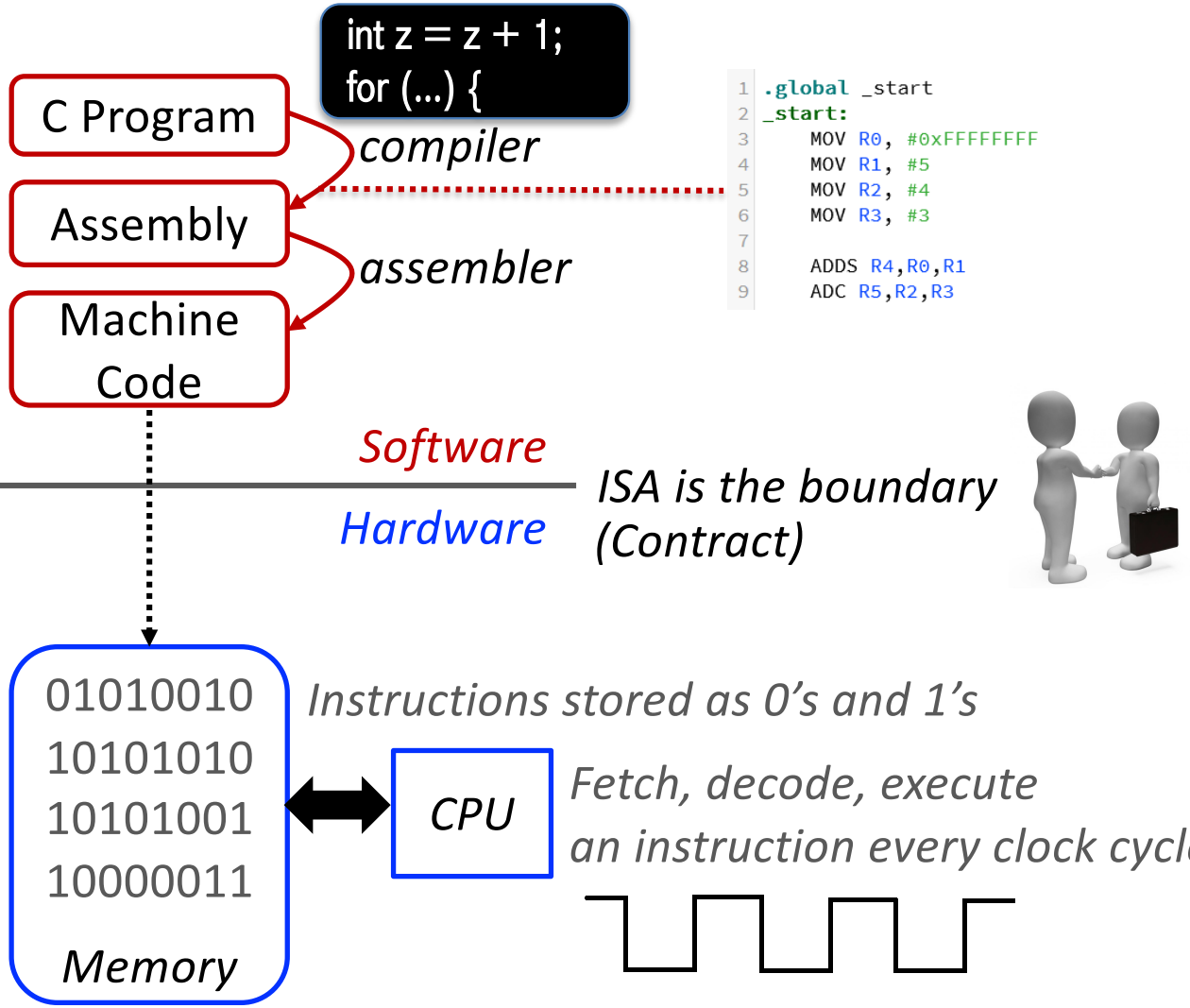
Program Ex.


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


int z = z + 1;
for (...) {
C Program
compiler
Assembly
assembler
Machine Code
.global _start
_start:
MOV R0, #0xFFFFFFFF
MOV R1, #5
MOV R2, #4
MOV R3, #3
ADDS R4,R0,R1
ADC R5,R2,R3
Software
Hardware
ISA is the boundary (Contract)
01010010
10101010
10101001
10000011
Memory
Instructions stored as 0's and 1's
CPU

Program Ex.


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


int z = z + 1;
for (...) {
C Program
compiler
Assembly
assembler
Machine Code
.global _start
_start:
MOV R0, #0xFFFFFFFF
MOV R1, #5
MOV R2, #4
MOV R3, #3
ADDS R4,R0,R1
ADC R5,R2,R3
Software
Hardware
ISA is the boundary (Contract)
01010010
10101010
10101001
10000011
Memory
Instructions stored as 0's and 1's
CPU
Fetch, decode, execute
an instruction every clock cycle

Program Ex.

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

```
int z = z + 1;
for (...) {
```

C Program

compiler

Assembly

assembler

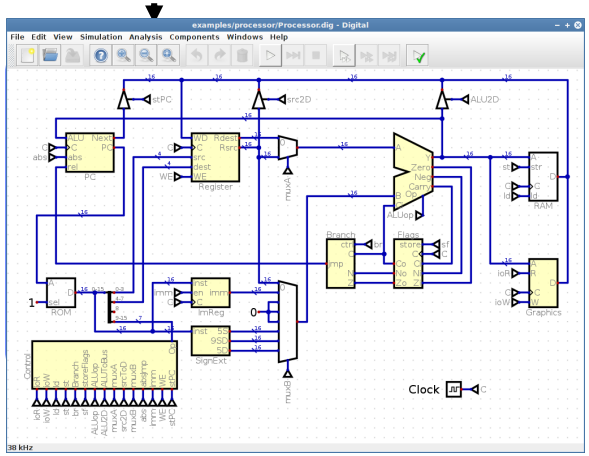
Machine Code

```
1  .global _start
2  _start:
3      MOV R0, #0xFFFFFFFF
4      MOV R1, #5
5      MOV R2, #4
6      MOV R3, #3
7
8      ADDS R4,R0,R1
9      ADC R5,R2,R3
```

Software

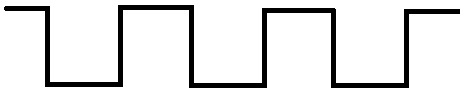
Hardware

ISA is the boundary (Contract)



stored as 0's and 1's

Fetch, decode, execute an instruction every clock cycle



Assignment 1: Build CPU

Program Ex.

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

```
int z = z + 1;
for (...) {
```

C Program

compiler

Assembly

assembler

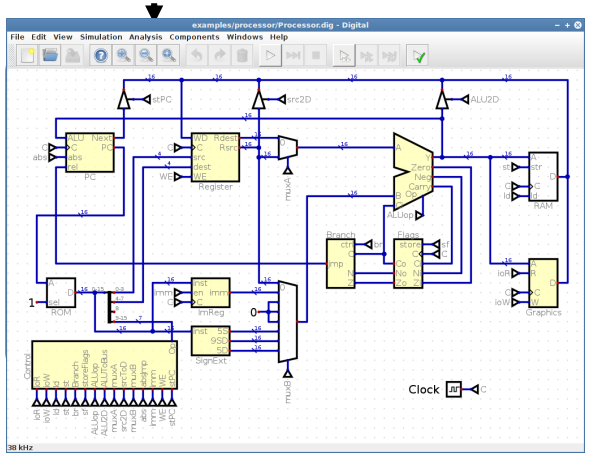
Machine Code

Assignment 2: Program CPU

Software

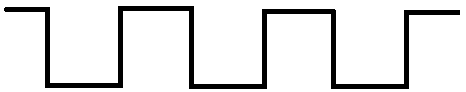
Hardware

ISA is the boundary (Contract)



stored as 0's and 1's

Fetch, decode, execute an instruction every clock cycle



Assignment 1: Build CPU

Universal Computational Device

- Anything that can be computed, can be computed by a computer provided it has enough time and enough memory
- We **instruct** the computer how to do X, and it obliges by **interpreting** our **instructions**
- Instructions are stored in memory like regular data
- Computer is **programmable** because we can rewrite instructions to make it do something else
  - Add, subtract, multiply, sort, search, ...

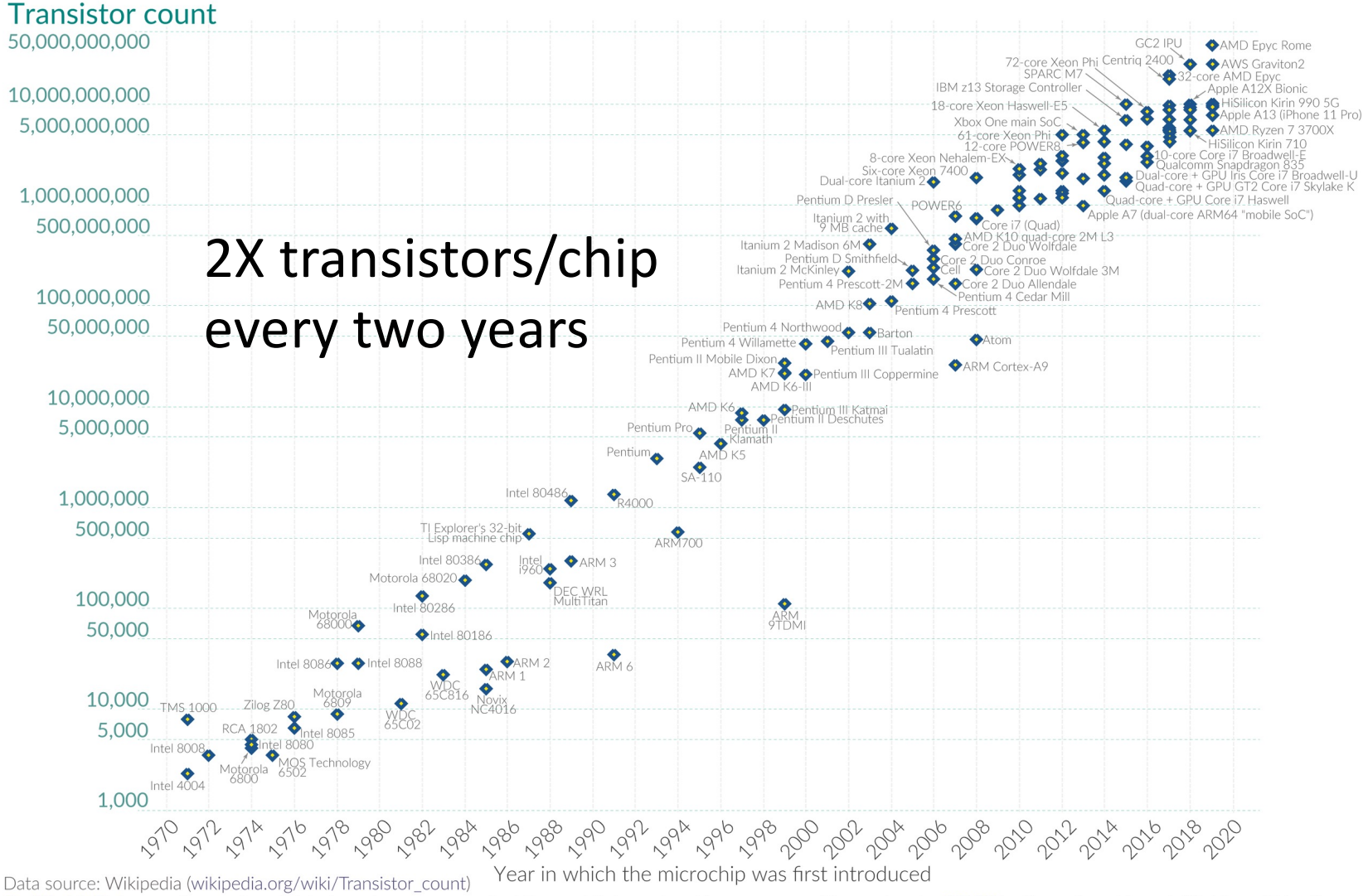
Technology Trends

- Moore’s Law
- Uniprocessor performance
- Memory wall
- Memory hierarchy

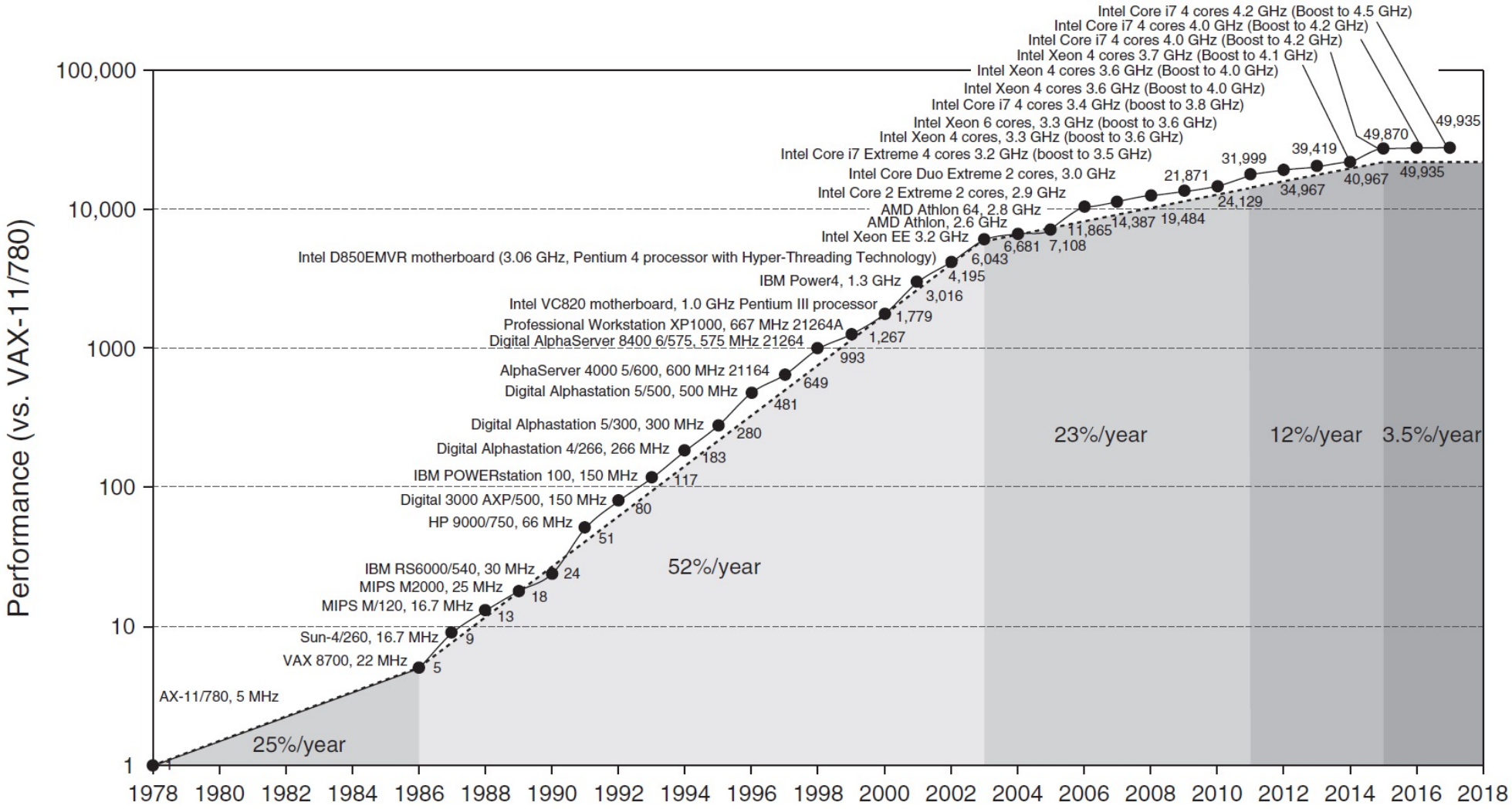
Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

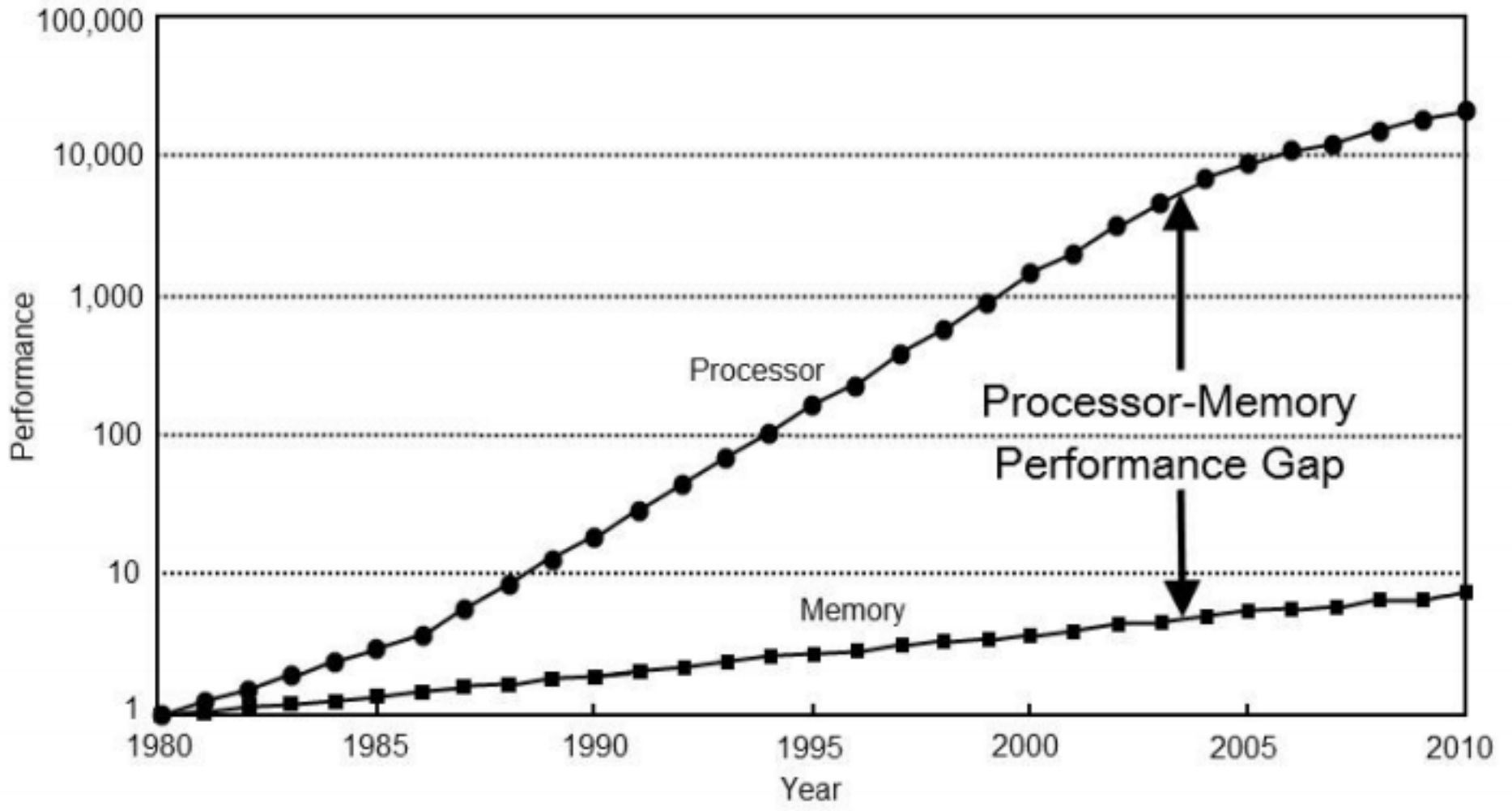
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

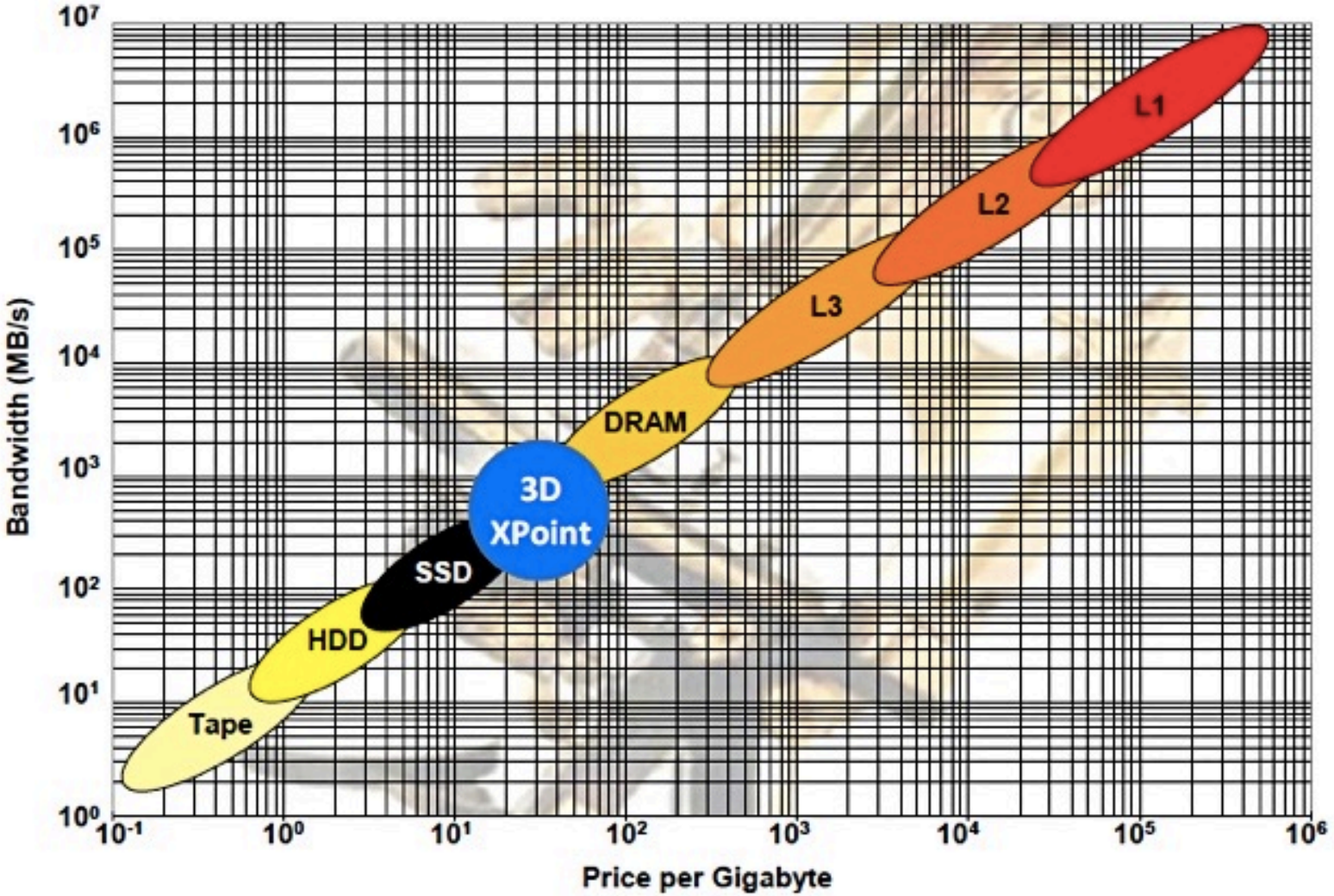


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)


Performance (vs. VAX-11/780)
100,000
10,000
1000
100
10
1
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016
2018
AX-11/780, 5 MHz
VAX 8700, 22 MHz
Sun-4/260, 16.7 MHz
MIPS M/120, 16.7 MHz
MIPS M2000, 25 MHz
IBM RS6000/540, 30 MHz
HP 9000/750, 66 MHz
Digital 3000 AXP/500, 150 MHz
IBM POWERstation 100, 150 MHz
Digital Alphastation 4/266, 266 MHz
Digital Alphastation 5/300, 300 MHz
Digital Alphastation 5/500, 500 MHz
AlphaServer 4000 5/600, 600 MHz 21164
Digital AlphaServer 8400 6/575, 575 MHz 21264
Professional Workstation XP1000, 667 MHz 21264A
Intel VC820 motherboard, 1.0 GHz Pentium III processor
IBM Power4, 1.3 GHz
Intel D850EMVR motherboard (3.06 GHz, Pentium 4 processor with Hyper-Threading Technology)
Intel Xeon EE 3.2 GHz
AMD Athlon, 2.6 GHz
AMD Athlon 64, 2.8 GHz
Intel Core 2 Extreme 2 cores, 2.9 GHz
Intel Core Duo Extreme 2 cores, 3.0 GHz
Intel Core i7 Extreme 4 cores 3.2 GHz (boost to 3.5 GHz)
Intel Xeon 4 cores, 3.3 GHz (boost to 3.6 GHz)
Intel Xeon 6 cores, 3.3 GHz (boost to 3.6 GHz)
Intel Core i7 4 cores 3.4 GHz (boost to 3.8 GHz)
Intel Xeon 4 cores 3.6 GHz (Boost to 4.0 GHz)
Intel Xeon 4 cores 3.6 GHz (Boost to 4.0 GHz)
Intel Xeon 4 cores 3.7 GHz (Boost to 4.1 GHz)
Intel Core i7 4 cores 4.0 GHz (Boost to 4.2 GHz)
Intel Core i7 4 cores 4.0 GHz (Boost to 4.2 GHz)
Intel Core i7 4 cores 4.2 GHz (Boost to 4.5 GHz)
5
9
13
18
24
51
80
117
183
280
481
649
993
1,267
1,779
3,016
4,195
6,043
6,681
7,108
11,865
14,387
19,484
21,871
24,129
31,999
34,967
39,419
40,967
49,870
49,935
49,935
25%/year
52%/year
23%/year
12%/year
3.5%/year

CPU/Memory performance

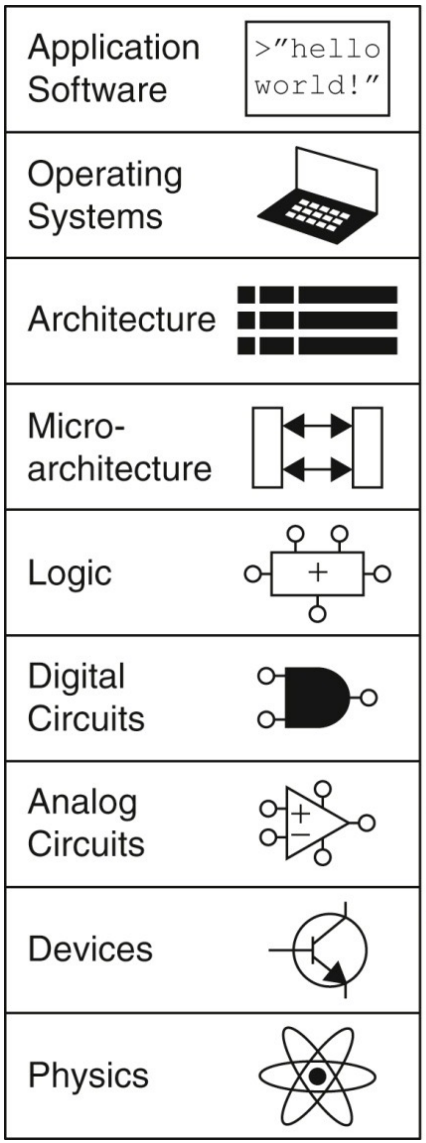




Source: Objective Analysis, 2019

Course Plan

- Navigate the transformation hierarchy
  - Bottom-up



Week 4

Week 5, 6

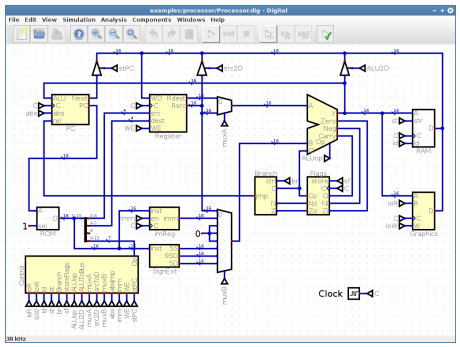
Week 2, 3

Week 1

Assignment 2

Week 7, 8, 9

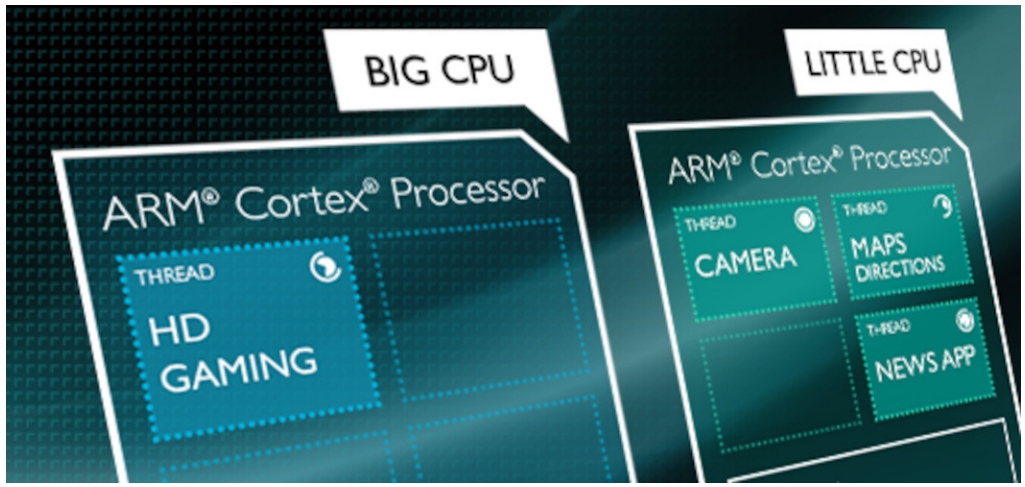
Week 10, 11, 12
(I/O and Advanced microarchitecture optimizations)



Assignment 1: Logic simulator

Hw/Sw Interaction an Important Skill

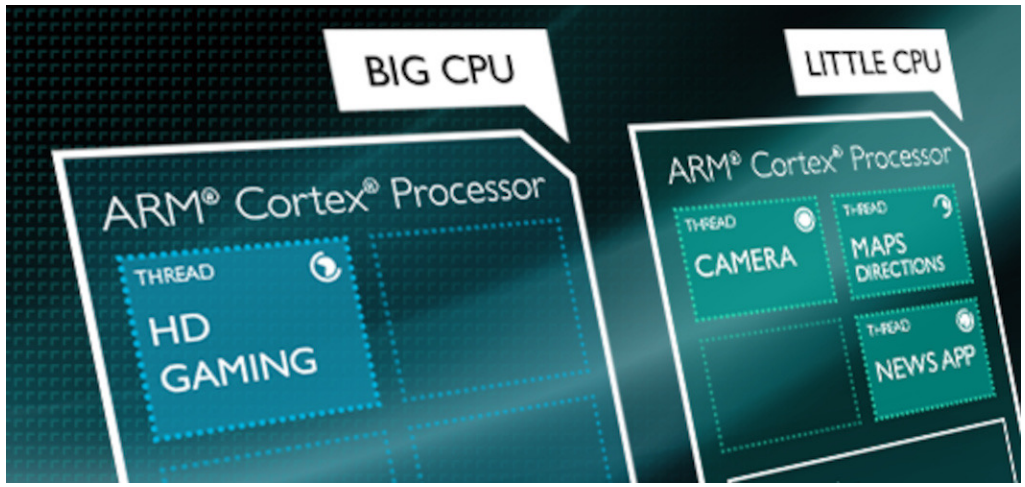
- Hardware is increasingly heterogeneous
  - Software needs to exploit better what is being offered at the lower levels



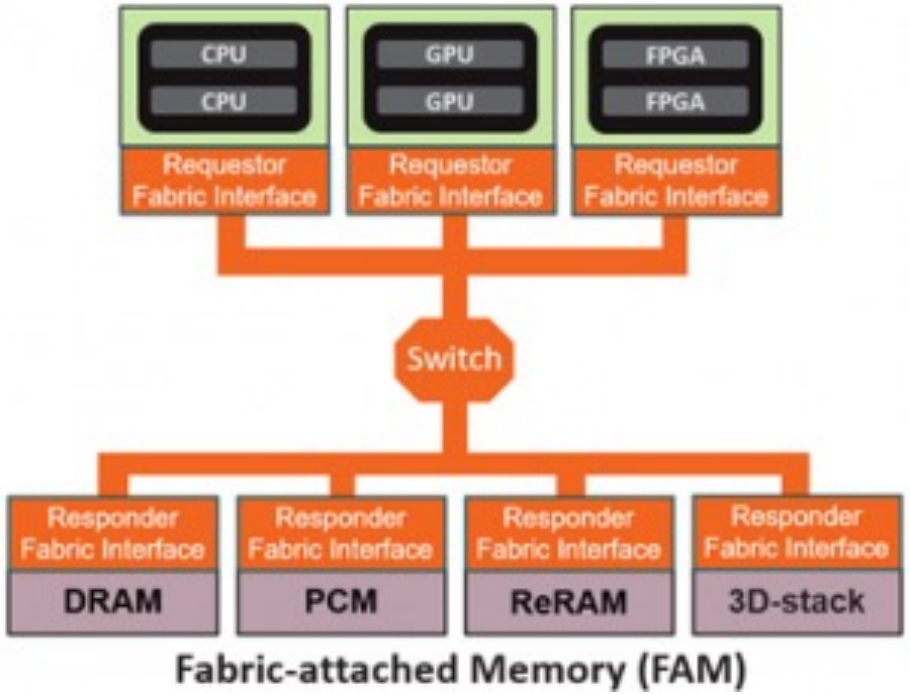
Heterogeneous CPU

Hw/Sw Interaction an Important Skill

- Hardware is increasingly heterogeneous
  - Software needs to exploit better what is being offered at the lower levels



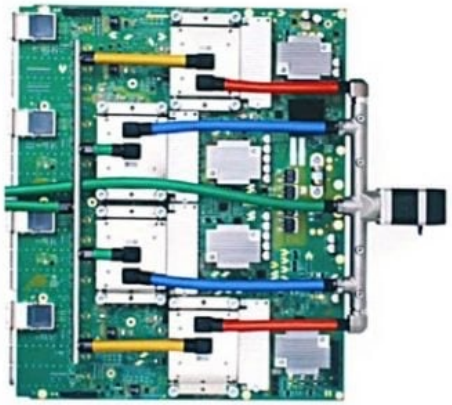
Heterogeneous CPU



Heterogeneous Memory

Hw/Sw Interaction an Important Skill

- Hardware is increasingly heterogeneous
  - New ML/AI accelerators
  - Programming models intertwined with hardware details
  - Many times: No luxury of a commodity language



TPU V4

Hw/Sw Interaction an Important Skill

- Sustainability is a new concern
  - ICT contribution to global GHG emissions around 4%, and increasing
  - Need to optimize the entire compute stack for energy efficiency

Hw/Sw Interaction an Important Skill

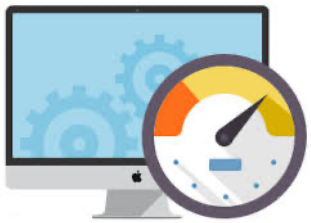
- Security trumps performance in many environments
  - Recent cyber attacks target vulnerabilities at the bottom
  - Early mitigations handled in software
  - Today's hardware built for performance
- Need to build fundamentally secure Hw/Sw systems

Hw/Sw Interaction an Important Skill

- Writing compilers, operating systems, virtual machines

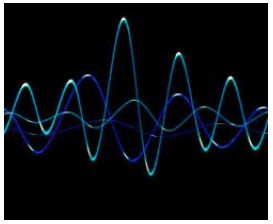
- Programming embedded computers
  - Nano drones, IoT devices, wearable computing

- Debugging performance issues better than the average programmer

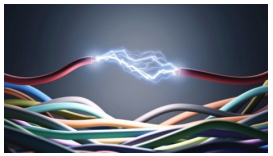


Representing Information

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



Frequency of oscillation



Voltage on a wire

Temperature

Answer: Infinite

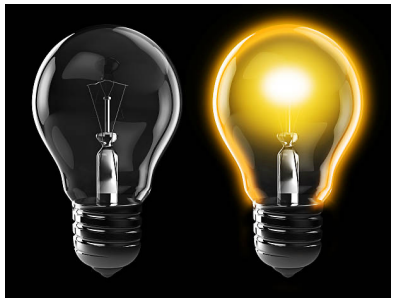
All these are continuous signals

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 NAY DIGIT

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

$$D = Log_2 N \ bits$$

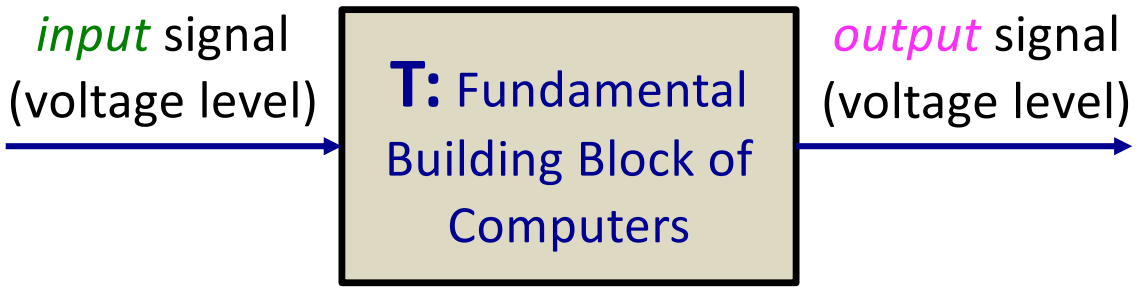
The color set has four states: N = 4, # bits = 2
The alphabet set has 26 states: N = 26, # bits = 5

Conversely,

If D is 2, N = 4

If D is 5, N = 32

Why do computers use binary?

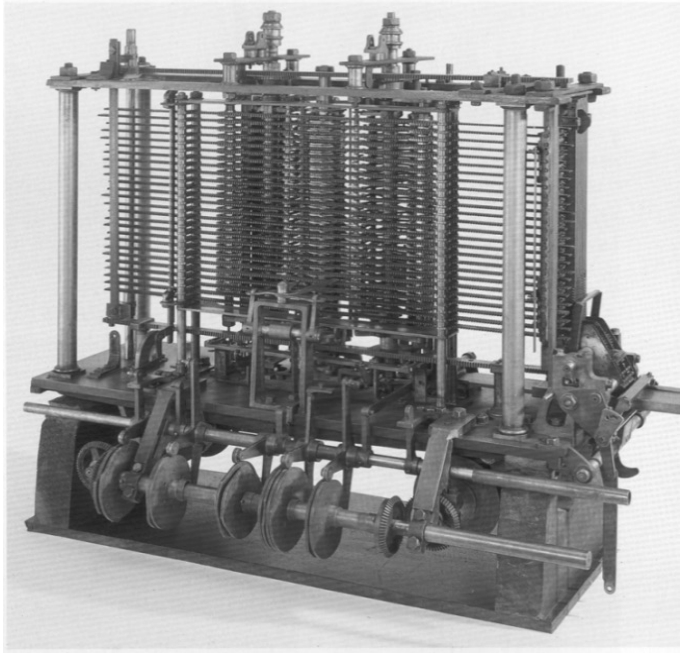


We can divide a continuous voltage range into ten levels to represent 0 – 9, but that would make **T** very complex

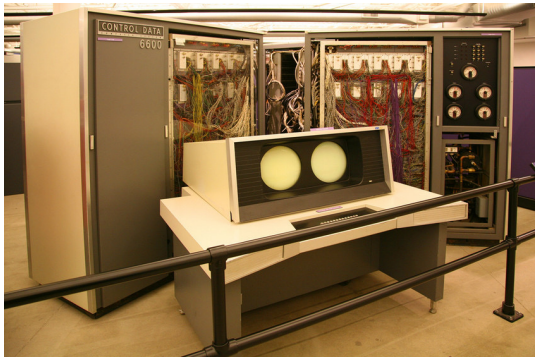
The fundamental building block of all computers is a transistor. A transistor can distinguish two voltage levels easily. We call these voltage levels 1 and 0

Voltages and Transistors, Why?

Mechanical parts: Not easy to scale to do large computations

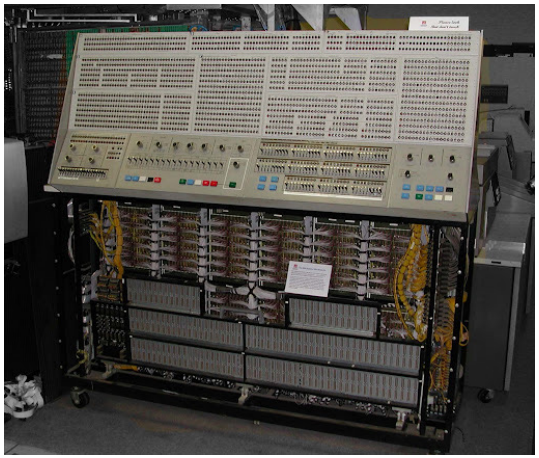


The Analytical Engine
Charles Babbage
1834 – 1871



CDC 6600, 1964

Apple M1, 2020
400 mm^2
16 billion transistors



IBM 360, 1964

Cost-effective shrinking

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 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)*

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

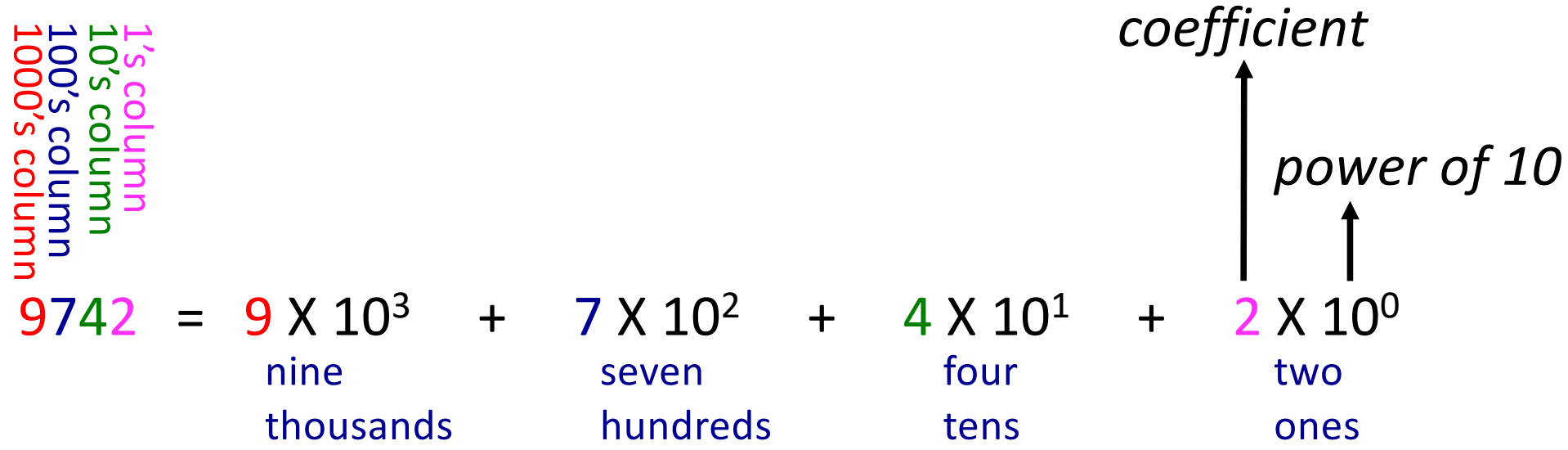
→ *Logic gates to perform operations on binary variables*

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



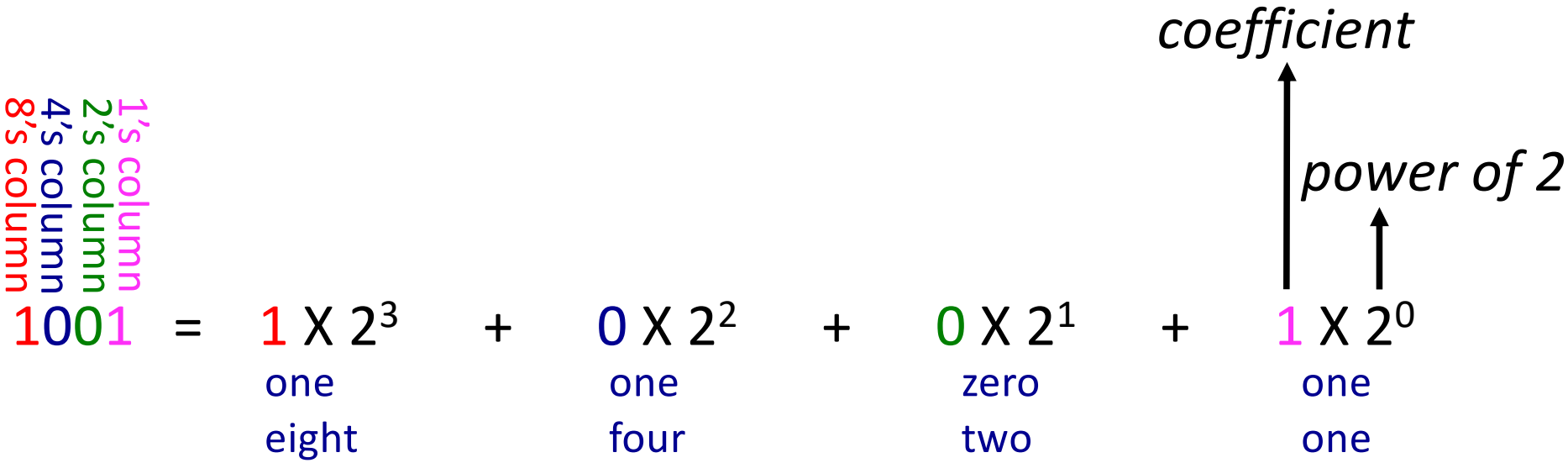
Range of Decimal Numbers

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



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 X 2 X 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
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

Kilo

Powers of 2

Power of 2	Decimal Value	Abbreviation	
2^{10}	1024	Kilo (K)	~ 1000
2^{20}	1048576	Mega (M)	~ 1000, 000
2^{30}	1073741824	Giga (G)	~ 1000, 000, 000

What is 2^{24} in decimal?

- 2^{20} X 2^{4} = 1 M X 16 = 16 M

What is 2^{17} in decimal?

- 2^{10} X 2^{7} = 1 K X 128 = 128 K

Terminology

Byte
- 8 bits

Nibble
- 4 bits

Word
- Machine-dependent
- 8 – 16 bits (gadgets)
- 32 – 64 bits (high-end)

Most Significant Bit

The bit in the highest position

Least Significant Bit

The bit in the lowest position

Terminology

Most Significant Byte

The byte in the highest position

Least Significant Byte

The byte in the lowest position

Decimal to Binary Conversion

Method # 1: Find the largest power of 2, subtract, and repeat

Example: Convert 53_{10} to binary

53	32 X 1
53 – 32 = 21	16 X 1
21 – 16 = 5	4 X 1
5 – 4 = 1	1 X 1

2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1

Decimal to Binary Conversion

Method # 2: Repeatedly divide by 2, remainder goes in the most significant position

Example: Convert 53_{10} to binary

53/2	=	26	R: 1
26/2	=	13	R: 0
13/2	=	6	R: 1
6/2	=	3	R: 0
3/2	=	1	R: 1
1/2	=	0	R: 1

2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1

Hexadecimal Numbers

Motivation: Tedious and error-prone to write long binary numbers

Hexadecimal or base 16: A group of four bits represent 2^4 or 16 possibilities

16 digits: 0 – 9, A, B, C, D, E, F

Column weights: 1, 16, 16^2 (or 256), 16^3 (or 4096)

Hexadecimal Numbers

Hex Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Binary to Hexadecimal

Binary	1	1	1	1	0	1	1	1
Hexa	F	7						

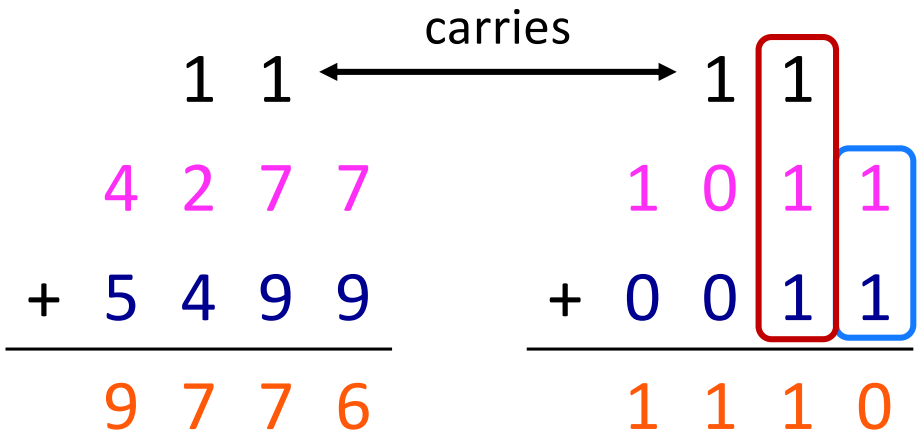
Binary	1	1	1	1	1	1	1	0
Hexa	F	E						

Hexadecimal to Binary

Hexa	D	7	4	1

Binary	1	1	0	1	0	1	1	1	0	1	0	0	0	0	0	1

Binary Addition



1 + 1 = 2 (10 in binary), but a binary variable can either be 0 or 1

- We record the 1's digit (0), and carry the 2's digit (1) over to the next column

1 + 1 + 1 = 3 (11 in binary), but a binary variable can either be 0 or 1

- We record the 1's digit (1), and carry the 2's digit (1) over to the next column

Overflow

- Suppose we have two 4-bit numbers
  - If A = 1111 and B = 1111
  - What is A + B?
  - What is the largest value 4 bits can represent?
- Overflow
  - The result is too big to fit inside the available bits
  - We check the carry bit out of the most significant column to detect overflow

		1	1	1	
	1	1	1	1	15
+	1	1	1	1	15
1	1	1	1	0	30

Signed Binary Numbers

We need both positive and negative numbers to solve real-world problems

How do we make a string of 1's and 0's represent both positive and negative numbers?

If we write all possible combinations of 0's and 1's in a disciplined fashion, maybe we can find a way

Most significant bit		least significant bit
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Signed Binary Numbers

Use the *most significant bit* to represent the sign: 0 means positive and 1 means negative

N bit sign/magnitude system: 1 bit for sign and N-1 bits for magnitude (absolute)

			Decimal
0	0	0	+0
0	0	1	+1
0	1	0	+2
0	1	1	+3
1	0	0	-0
1	0	1	-1
1	1	0	-2
1	1	1	-3

Signed Binary Numbers

What are the drawbacks of sign/magnitude representation?

Ordinary binary addition does not work for sign/magnitude numbers

- What is the sum of +3 and -3 and *does the result make sense*?

Awkward to have two different representations of the same number (0)

			Decimal
0	0	0	+0
0	0	1	+1
0	1	0	+2
0	1	1	+3
1	0	0	-0
1	0	1	-1
1	1	0	-2
1	1	1	-3

One's Complement

Tried in some early computers, Control Data Corporation (CDC) 6600

Negative number: Take the representation of a positive integer and flip all the bits

Known commonly as 1's complement

- Same problems as the sign/magnitude representation

	Decimal
0 0 0	+0
0 0 1	+1
0 1 0	+2
0 1 1	+3
1 0 0	-3
1 0 1	-2
1 1 0	-1
1 1 1	-0

Two’s Complement

A third system of representation for signed integers for which

- Ordinary addition works
- Single representation for *zero*

Problem: If A + B = C, and A is known, then find B, such that C = 0

			Binary				Decimal
	A	=	0	1	0	1	?
+	B	=	?	?	?	?	?
	C	=	0	0	0	0	0

Problem: If A + B = C, and A is known, then find B, such that C = 0

			Binary				Decimal
	A	=	0	1	0	1	+5
+	B	=	?	?	?	?	?
	C	=	0	0	0	0	0

Problem: If A + B = C, and A is known, then find B, such that C = 0

			Binary				Decimal
	A	=	0	1	0	1	+5
+	B	=	?	?	?	?	-5
	C	=	0	0	0	0	0

Problem: If A + B = C, and A is known, then find B, such that C = 0

			Binary				Decimal
	A	=	0	1	0	1	+5
+	B	=	1	0	1	1	-5
	C	=	0	0	0	0	0

Problem: If A + B = C, and A is known, then find B, such that C = 0

What is the relationship between A and B?

			Binary				Decimal
	A	=	0	1	0	1	+5
+	B	=	1	0	1	1	-5
	C	=	0	0	0	0	0

Some Observations

Observation # 1: If A + B = C, and A is +5, and C is 0, then B must be -5. (We have found a new representation for negative numbers.)

Observation # 2: To transform A to B (i.e., +5 to -5), we need to take 1's complement of A and then add +1. We say that B is **2's complement** of A

Observation # 3: Like sign/magnitude numbers, positive numbers have the MSB set to 0, and negative numbers have the MSB set to 1

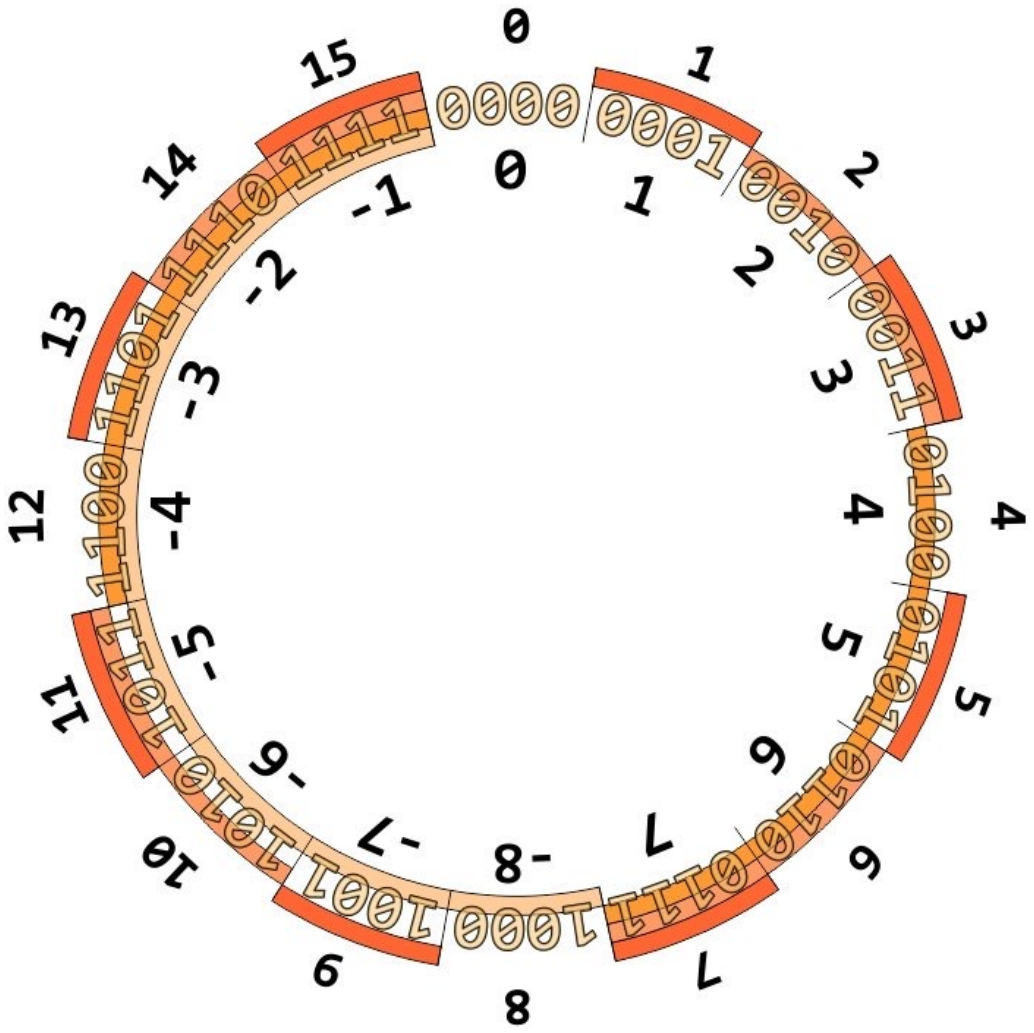
Some Observations

Observation # 4: Ordinary addition works

- What is the sum of +3 and -3 in two's complement system, and *does the result make sense*?
- Since ordinary addition works, a circuit to add numbers can handle both addition and subtraction
  - Recall that, X – A is equivalent to X + (–A)

$$
\begin{array}{rcccc}
 & 1 & & & \\
 & 0 & 1 & 1 & +3 \\
+ & 1 & 0 & 1 & -3 \\
\hline
 & 0 & 0 & 0 &
\end{array}
$$

2’s Complement Circle



More Observations

Observation # 5: There is only one representation for *zero*

Observation # 6: There is one more negative number than positive number

- With 3 bits, this number is 100
- With 4 bits, this number is 1000
- This negative number has no positive counterpart
- It is called the *weird number*
- The 2's complement of the weird number is the weird number (verify!)

2’s Complement to Decimal

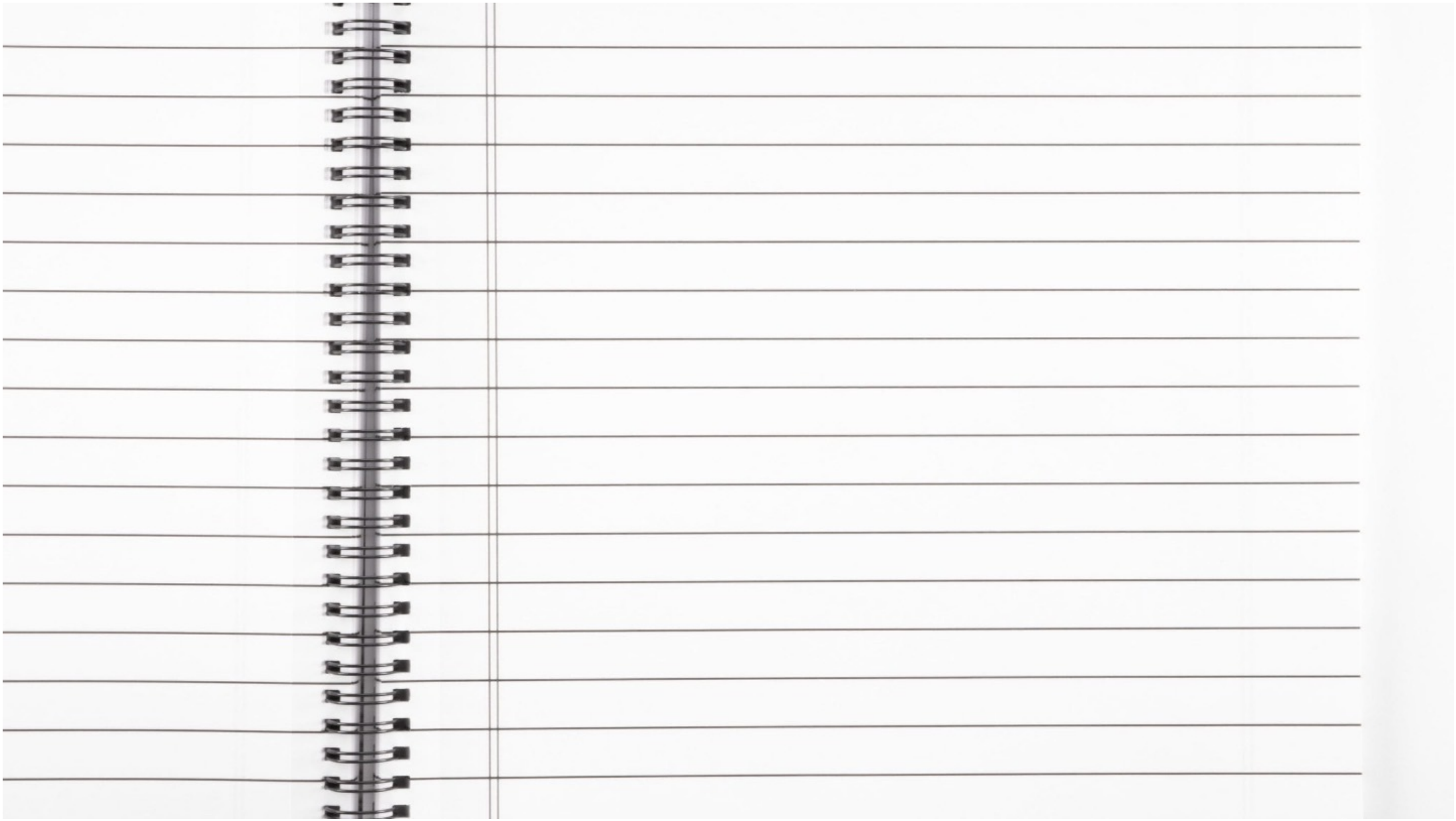
If MSB is 0

- *It is a positive number. The magnitude is represented by the remaining N-1 bits*

If MSB is 1

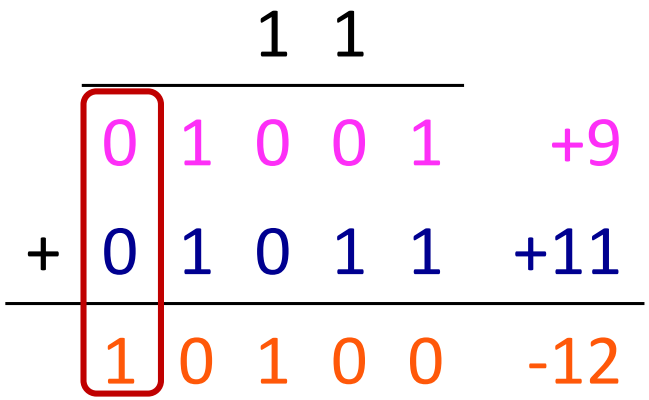
- *It is a negative number. Take the two’s complement of the (binary) number. The magnitude (of the negative number) is represented by the N – 1 bits*

Practice and test your understanding using the two’s complement circle



Overflow in 2's Complement

- Suppose we have two 5-bit numbers
  - A = 01001 and B = 01011
  - What is A + B?
  - What is the largest value 5 bits can represent in 2's complement?
- Overflow
  - The result is too big to fit inside the available bits
  - Sum of two positive integers cannot produce a negative integer!



Overflow in 2's Complement

- Suppose we have two 5-bit numbers
  - A = 10100 and B = 11010
  - What is A + B?
  - What is the *smallest* value 5 bits can represent in 2's complement?
- Overflow
  - The result is too big to fit inside the available bits
  - Sum of two negative integers cannot produce a positive integer!

	1	0	1	0	0	-12
+	1	1	0	1	0	-6
	0	1	1	1	0	14

Overflow in 2's Complement

Observation # 1: If two number being added have the same sign bit and the result has the opposite sign bit (easy!)

Observation # 2: Unlike unsigned numbers, a carry out of the most significant bit does not indicate overflow

Observation # 3: The sum of a negative number and a positive number never produces an overflow (**prove yourself!**)

Range of Number Systems

Number System	Minimum	Maximum
Unsigned	0	$2^N - 1$
Sign/Magnitude	$-2^{N-1} + 1$	$2^{N-1} - 1$
Two's Complement	-2^{N-1}	$2^{N-1} - 1$

N = 3

Unsigned: 0 to 7

Sign/Magnitude: -3 to 3

2's Complement: -4 to 3

N = 4

Unsigned: 0 to ?

Sign/Magnitude: -? to ?

2's Complement: -? to ?

Comparing Number Systems

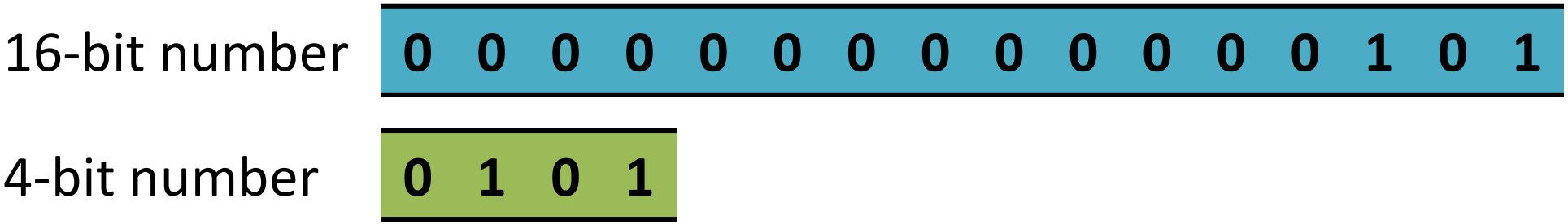
Binary Representation	Decimal Value Represented			
	Unsigned	Signed Magnitude	1's Complement	2's Complement
000	0	0	0	0
001	1	1	1	1
010	2	2	2	2
011	3	3	3	3
100	4	-0	-3	-4
101	5	-1	-2	-3
110	6	-2	-1	-2
111	7	-3	-0	-1

Quiz: See any errors?

	Unsigned	Signed	1's Comp.	2's Comp.
0 0 0 0	0	0	0	0
0 0 0 1	1	1	1	1
0 0 1 0	2	2	2	2
0 0 1 1	3	3	3	3
0 1 0 0	4	4	4	4
0 1 0 1	5	5	5	5
0 1 1 0	6	6	6	6
0 1 1 1	7	7	7	7
1 0 0 0	8	-0	-7	-1
1 0 0 1	9	-1	-6	-2
1 0 1 0	10	-2	-5	-3
1 0 1 1	11	-3	-4	-4
1 1 0 0	12	-4	-3	-5
1 1 0 1	13	-5	-2	-6
1 1 1 0	14	-6	-1	-7
1 1 1 1	15	-7	-0	-8

Sign Extension

Question: What the difference between the 16-bit and 8-bit numbers below?

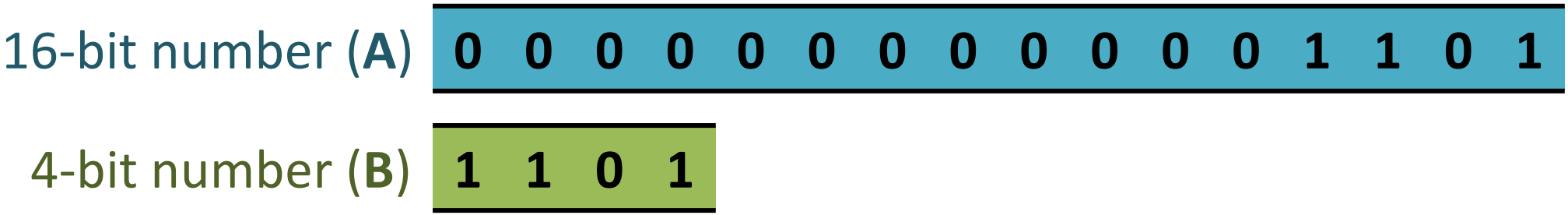


Answer: None. They both represent the positive number 5
Leading zeros do not impact the magnitude of a binary number

There are times when it is useful to allocate a small number of bits to represent a value

Sign Extension

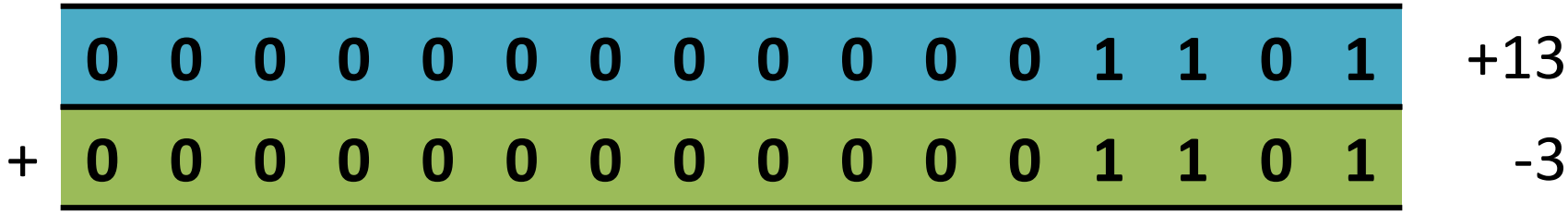
What value to the two numbers below represent?



What is the sum of **A** and **B**?

- Scenario # 1: Assume the absence of bits in **B** to be 0
- Scenario # 2: Assume the absence of bits in **B** to be 1

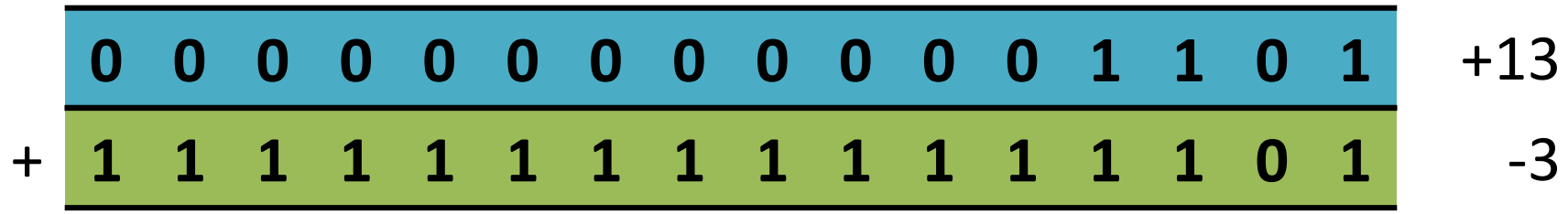
Scenario # 1



X

The assumption that appending 0's will lead to correct addition was wrong

Scenario # 2



✓

The assumption that appending 1's will lead to correct addition was right

Sign Extension

Leading 0's do not change the magnitude of the positive number

Leading 1's do not change the magnitude of the negative number

When a 2's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. We refer to the operation is Sign-EXTension or SEXT

Boolean Logic

A system of logic for binary variables

- Helps us reason about the interactions between two binary variables
  - X is 1 and Y is 0. What is X ∩ Y?
- Formalizes *key* (simple) logical operations on binary variables
- Logical operations are the steppingstone for composing sophisticated operations (including arithmetic)

Logic Functions vs Gates

Logic gates are digital circuits that take one or more inputs and produce a binary output

- Logic gate is the physical realization of a logical function
  - The logic AND gate (*built with transistors*) implements the logical AND function
- The inputs are to the left, and the output is to the right
- The relationship between inputs and the output is described by a *truth table* or a *Boolean equation*

Truth Table

A convenient way to describe the behavior of logical functions

- Suppose **A** and **B** are input operands and **Y** is the output
  - **A** can be 0 or 1
  - **B** can be 0 or 1
  - A total of four combinations (rows)
  - Three columns (2 inputs and an output)

For the contrived example on the right, the Boolean equation for **Y** is, **Y** = 0

- The values of **A** and **B** does not matter

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

Truth Table with More Inputs

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

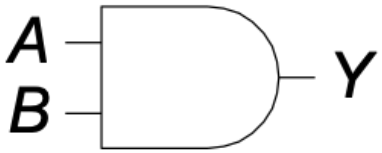
Boolean Equation for output Y: Y = 1

Note: *Soon we will see more interesting logic functions than Y = 0 and Y = 1*

The AND Function

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

Truth Table



AND Logic Gate

$Y = AB$

$Y = A.B$ (product)

$Y = A \cap B$ (intersection)

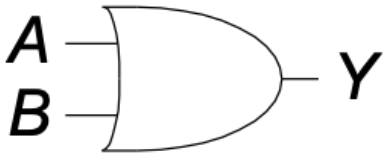
Boolean Equation

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 (sum)

Y = A ∪ B (union)

Boolean Equation

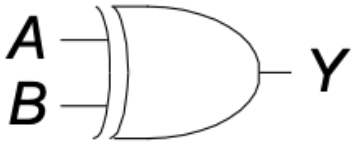
AND Function: *The output Y is 1 if either A or B are 1*

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

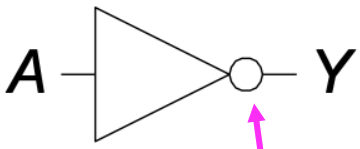
The OR function, on the other hand, produces an output 1, if only one of the two sources is a 1, or both sources are one (think of it as *inclusive* OR)

The NOT Unary Function

A	Y
0	1
0	1
1	0
1	0

Truth Table

The NOT gate has only one input (unary)



bubble → invert

Y = A'

Y = $\bar{A}$

NOT Logic Gate

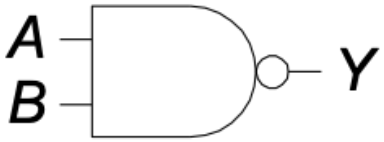
Boolean Equation

NOT Function: *The output Y is the inverse of the input A*

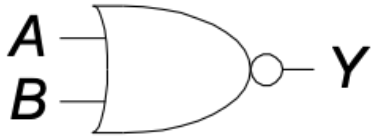
Inverting a Gate's Operation

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

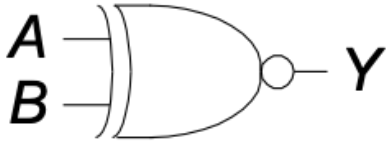
NOT AND → NAND



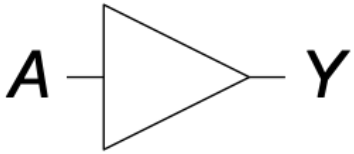
NOT OR → NOR



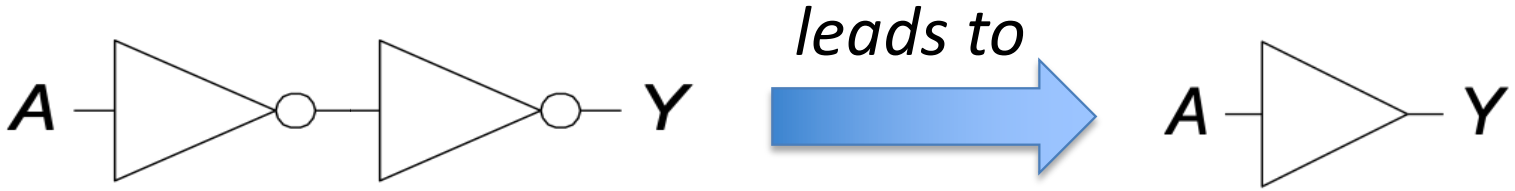
NOT XOR → XNOR



NOT NOT → BUF



In Boolean logic, two wrongs make a right!

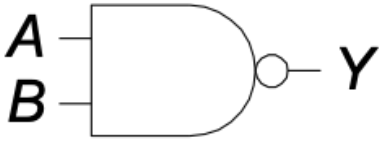


We say that two bubbles cancel each other's effect

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)’

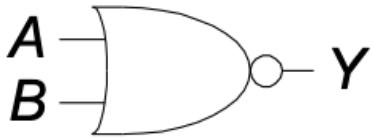
Boolean Equation

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

$Y = (A + B)'$

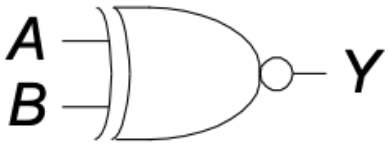
Boolean Equation

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*

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

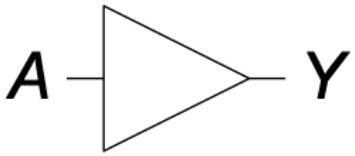
XNOR: Output is 1 when inputs are equal (even number of 1's)

Equality Gate

Buffer (BUF)

A	Y
0	0
0	0
1	1
1	1

Truth Table



BUF Logic Gate

$$Y = A$$

Boolean Equation

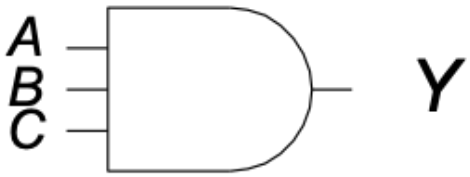
Buffer: *The output Y is equal to the input A*

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?



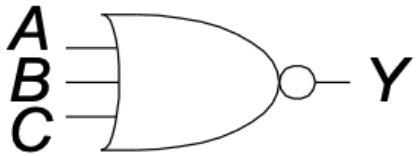
Y = ABC

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 (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 (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

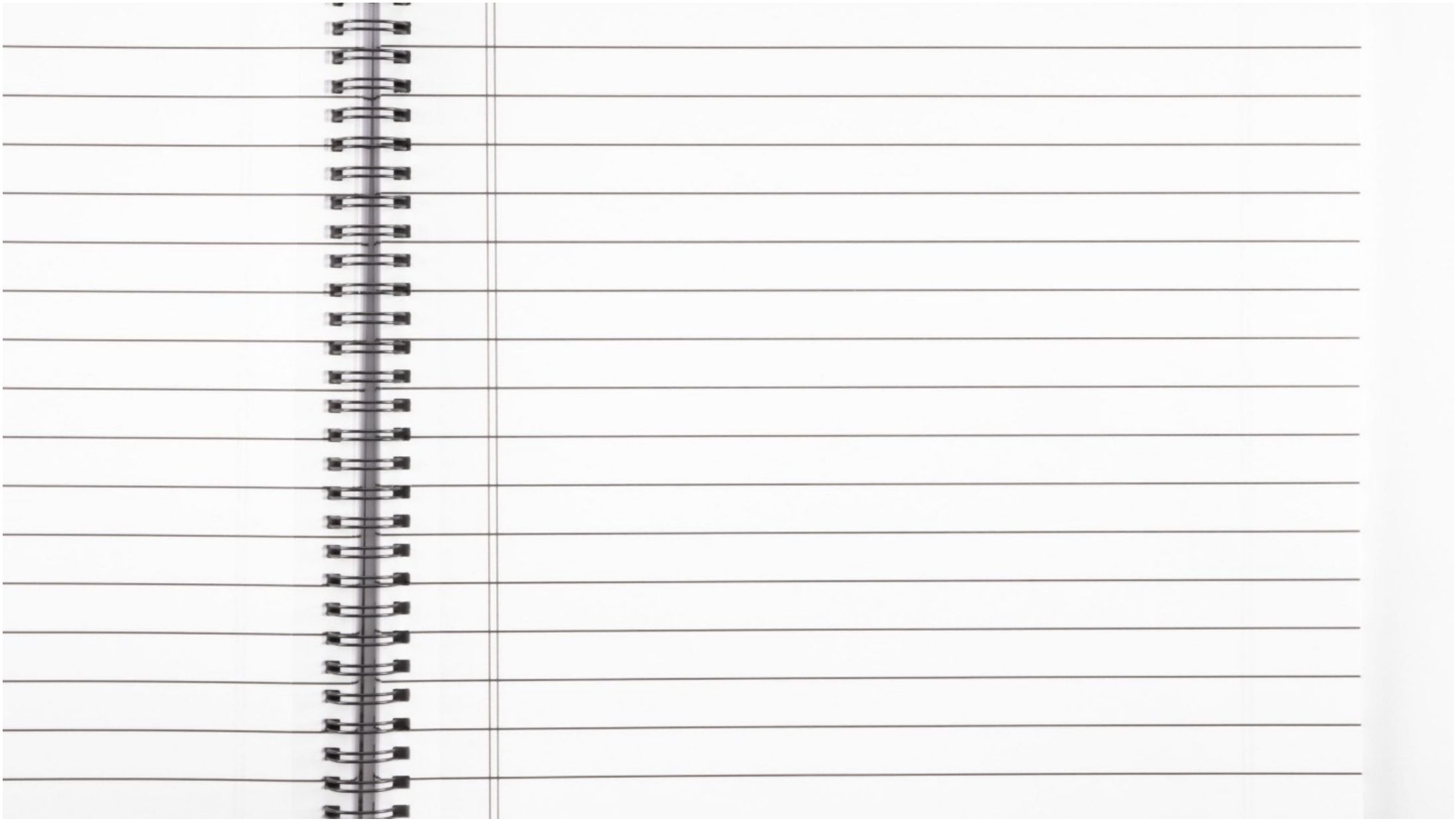
C = AB (bit-wise AND)

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 = 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 = 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, 1 AND X = X, where X is a binary variable. Also, verify that, 0 OR X = X.