Lecture 3: Hardware II

AP Computer Science Principles

.

Logical Gates

Encoding Data

Computer logic attempts to solve two problems: **encoding data** and **processing data**.

In the previous lecture, we talked about encoding data such as text and images using ASCII and various UTF encodings.

We'll talk about how a computer process data, e.g., display text, show image or play a video.

Processing Data

Encoding data is mapping the data such as text to a sequence of bits.

Processing data is mapping an input sequence of bits to some output sequence of bits. These mappings can be as basic as math operations or as complicated as applying a filter to an image.

All processing operations can be broken down into simple operations called **boolean functions.**

Boolean Function

A **boolean function** is a function that takes an m-sequence binary input and maps it into an n-sequence binary output.

For example, let f be a function that takes 6 bits sequence as inputs and output 0 if the number of 1's is even and 1 if it is odd.

f(001100) = 0

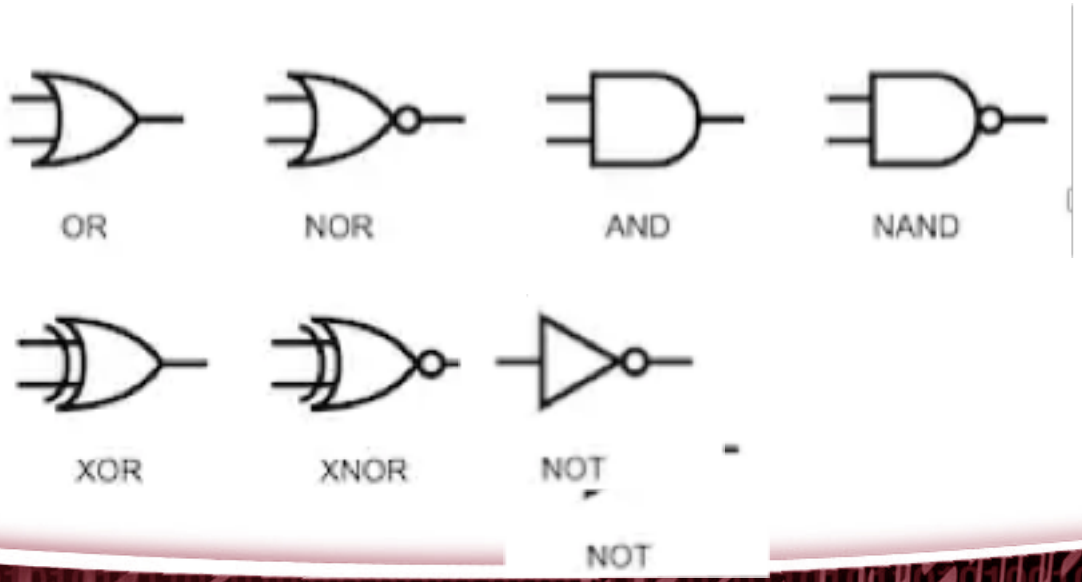
f(001110) = 1

etc..

Boolean Function

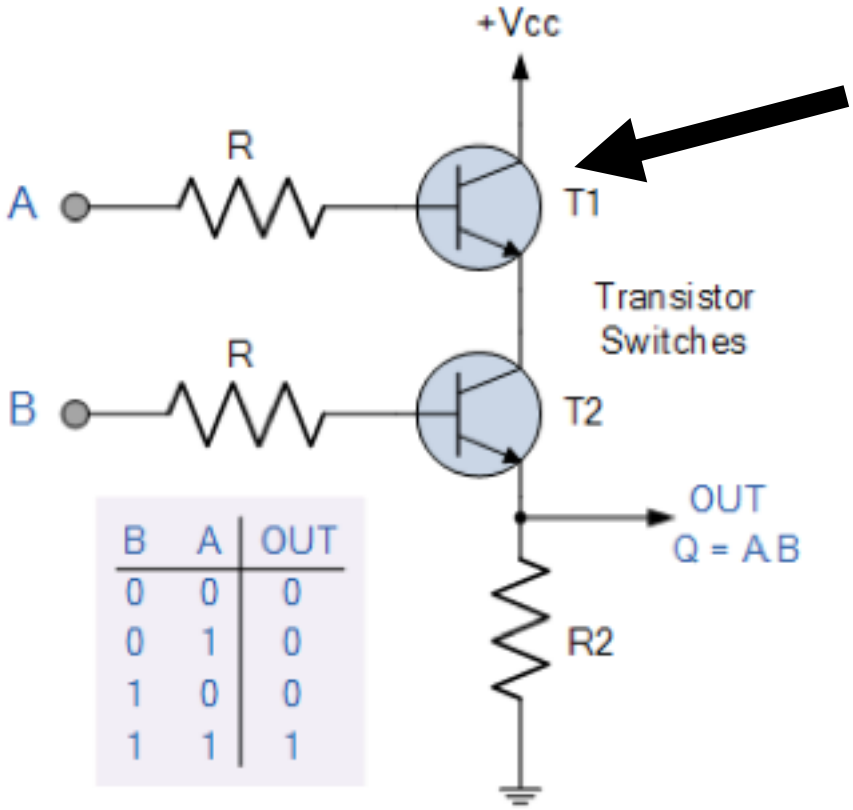
Any boolean function can be decomposed further into one of seven basic boolean functions: OR, XOR(Exclusive-OR), NOR, XNOR, AND, NAND, and NOT.

The NOT function takes one bit input and output one bit. The other functions all take two bits input and output one bit.



Logical Gate: AND

AND: The output is "true”(1) when both inputs are "true." Otherwise, the output is "false”(0).



B	A	OUT
0	0	0
0	1	0
1	0	0
1	1	1

transistor: a building block for all integrated chips.

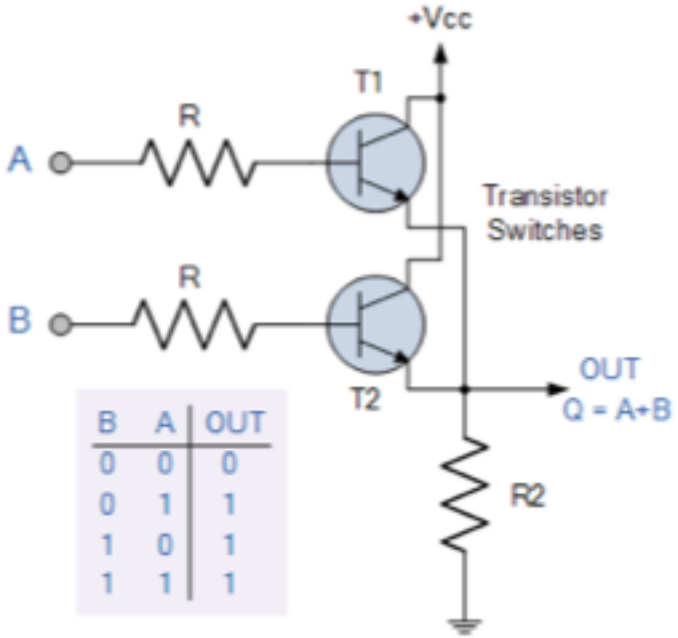
Logical Gate: OR

OR: The output is "true”(1) if one or both inputs are "true." Otherwise, the output is "false”(0). This is also known as an **inclusive OR**.

Example:
The following statement is true:

Boston is the capital of Massachusetts

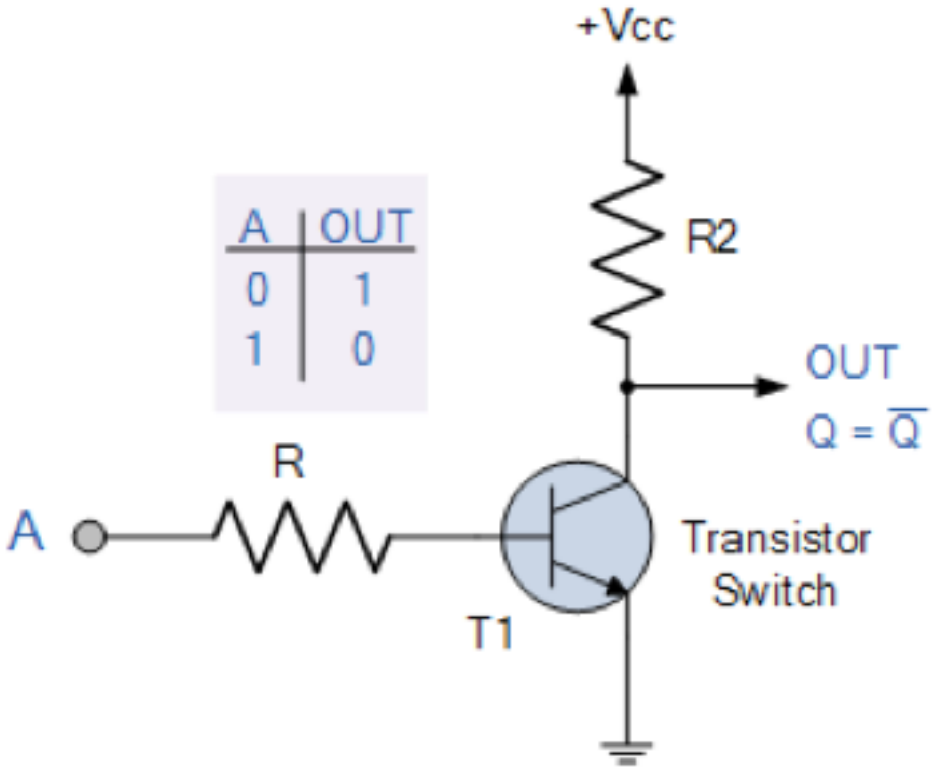
or 2 + 2 = 5.



B	A	OUT
0	0	0
0	1	1
1	0	1
1	1	1

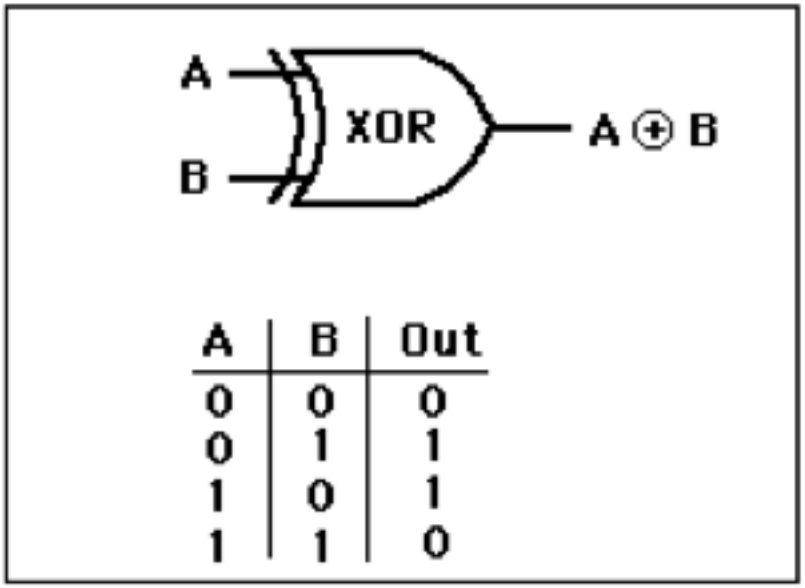
Logical Gate: NOT

NOT: The NOT function only takes one input. It inverts the input.



Logical Gate: XOR

XOR: The output is "true”(1) if exactly one of the inputs is “true”. Otherwise, the output is "false”(0). This is also known as an **exclusive OR**.



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Logical Gates: NAND

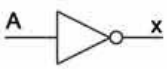
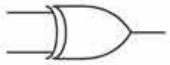
The remaining gates are simply composing one of the previous gate with a NOT. For example, NAND gate operates as an AND gate followed by a NOT gate.

It is a fact in boolean logic that you don't need all seven. For example, the set of gates {AND, OR, NOT} can generate all boolean functions.

Even better, the NAND gate by itself can generate all boolean functions! For this reason, NAND is called a **universal gate**. The only other universal gate is the NOR gate.

Computer Logic

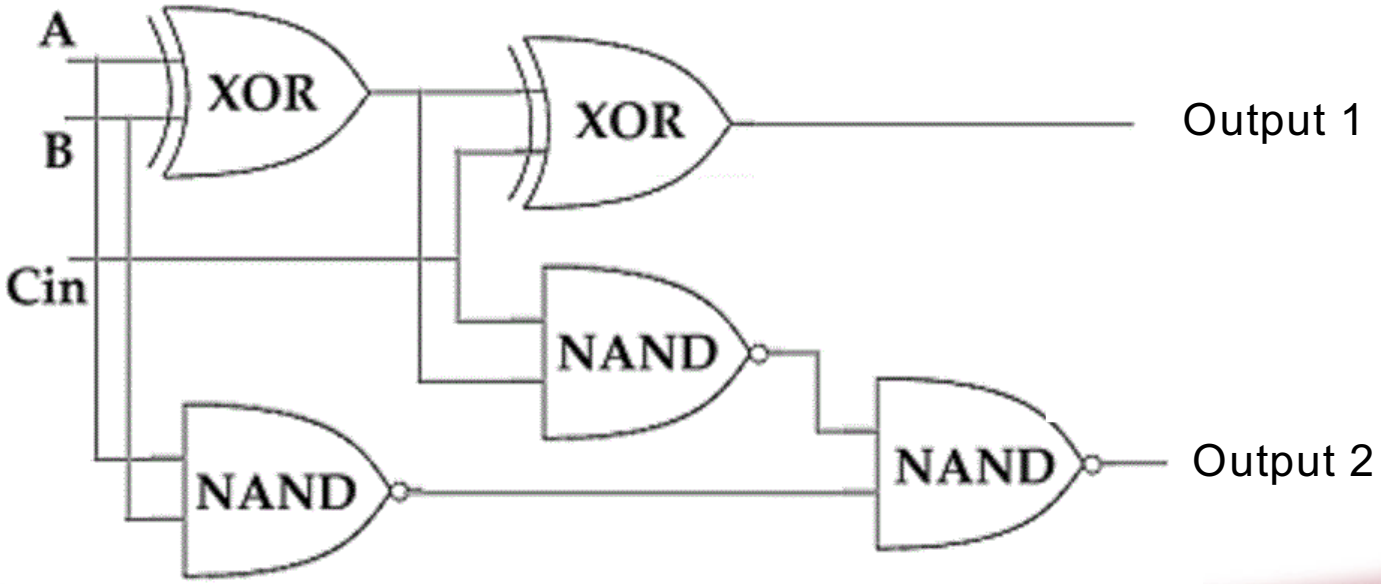
Logic Gates

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR
Alg. Expr.	$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
Symbol							
Truth Table	A | X 0 | 1 1 | 0	B A | X 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	B A | X 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0	B A | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1	B A | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0	B A | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	B A | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

Computer Logic

The basic gates can be combined to form operations such add, subtract, multiply etc... And those in turn can be combined to form more complicated operations.

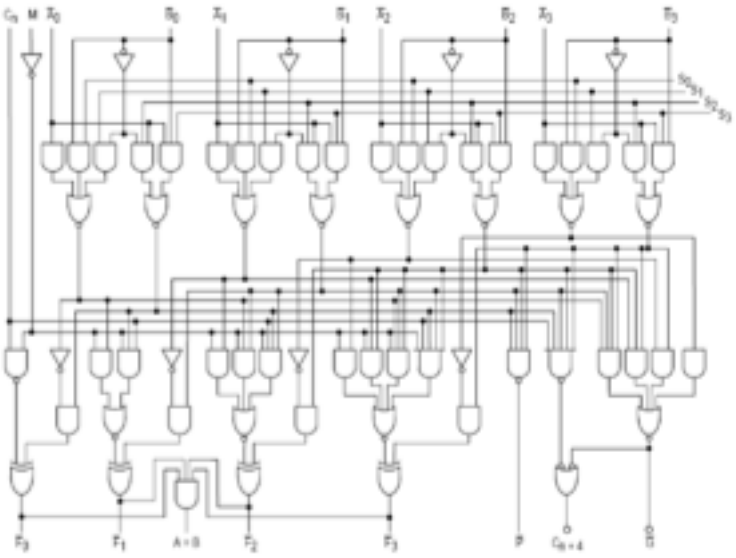
Can you see the purpose of this circuit? See worksheet.

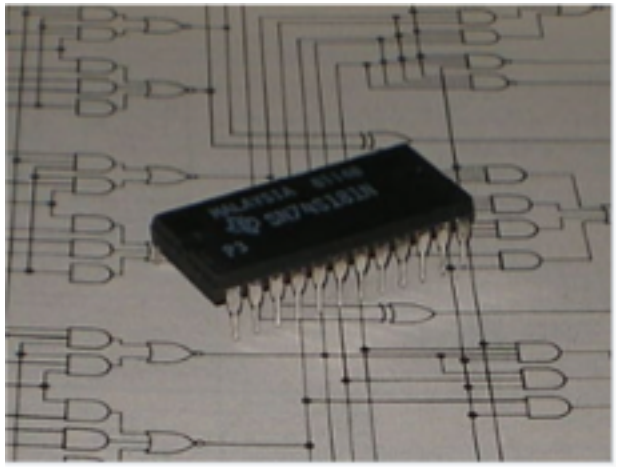


ALU

Amazingly, the entire logical circuit of a computer can be built up using only the basic gates(in fact, using only NAND gates or only the NOR gates).

Here's a 4-bit ALU(Arithmetic Logic Unit), one of the building blocks of a CPU built from basic gates.





CPU

CPU

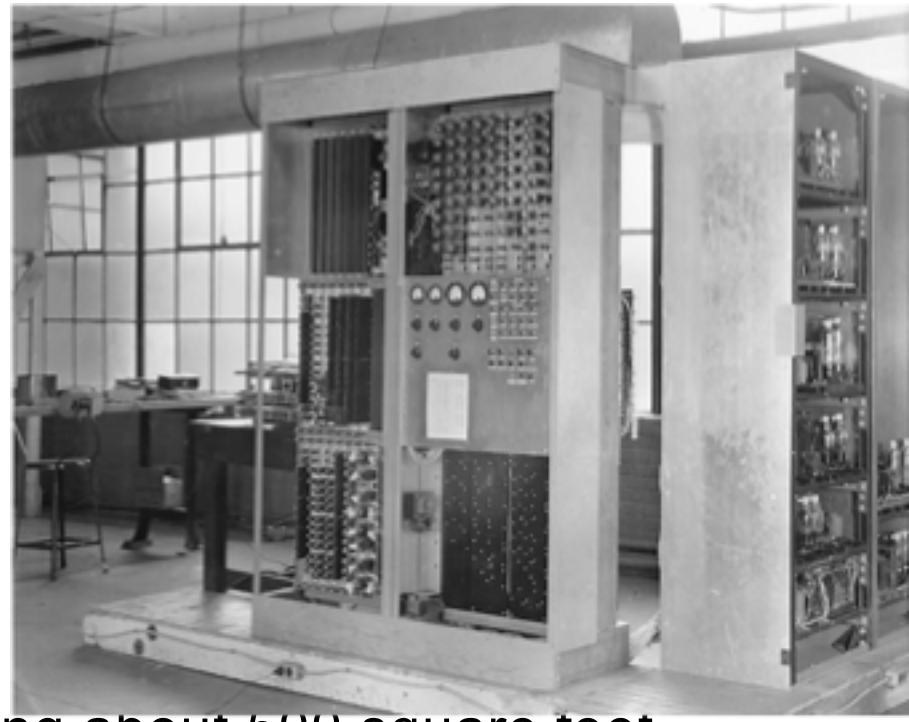
AMD Phenom II
Socket AM3

Intel Core i7
LGA 1366

Intel Core i5
LGA 1156

EDVAC

One of the first computers capable of executing stored instructions was EDVAC (Electronic Discrete Variable Automatic Computer), developed in the mid-1940s for the United States Army by the University of Pennsylvania.



Weighing almost 9 tons and covering about 500 square feet, EDVAC could add two numbers together in about 850 microseconds.

Processes

The CPU's main job is to handle the execution of **processes**, which is a single instance of a program being executed. An operating system will typically allow you to view the current processes that are executing on your machine.

So, processes have a set of instructions that tell your CPU how the program actually works.

Task Manager

On Windows, you can press the infamous Ctrl-Alt-Delete to open up the task manager, while on Mac OS X, you can open a program called the Activity Manager.

If your computer starts to slow down, applications like the Task Manager on Windows and Activity Manager on Mac can be useful for identifying which currently-running process is the culprit (as a result of greedy memory or CPU usage).

These applications also allow you to kill processes that are hanging or no longer responding, which should shut down a stalled program.

CPU Instructions

However, a CPU doesn't understand English sentences like "open Google Chrome”. In reality, the instructions that can be understood by a CPU are much more primitive.

What we would consider a simple task (like reading an email) actually requires a huge number of CPU instructions to complete because each individual instruction is so simple and low-level.

CPU Instructions

So, the CPU will typically be given a sequence of these small instructions that when run in some predetermined order create some larger functionality like reading an email.

For example, reading an email requires your computer to download the email, display its contents, remember that the message has been read, etc.

CPU Instructions

Then, the task of displaying an email can be broken down even further, as it requires figuring out what font to use, what color the text should be drawn in, and actually displaying something on your screen.

Even then, these instructions are still at a much, much higher level than the instructions understood by a CPU; in many cases, a CPU instruction is a task that can't be broken down much further.

CPU Instructions

Instead, more realistic CPU instructions could include "add two numbers together," "flip the fourth bit in this 4-byte number," or "determine if these two numbers are equal."

These are the instructions that can be implemented at the low level; the level that deals with gates and boolean functions discussed earlier.

Doesn't get much simpler than that! But, Tetris is in fact really just a very long sequence of instructions like these.

CPU Instruction Set

The list of all the different types of instructions that can be understood by a CPU is called its **instruction set**. Not all CPUs are alike, which means a CPU manufactured by AMD may understand a different set of instructions than a CPU made by Intel does.

The instructions in a CPU's instruction set can be broken up into three main groups: **data, arithmetic, and control flow**.(See the video The Making of a Chip, on the last slide).

Data

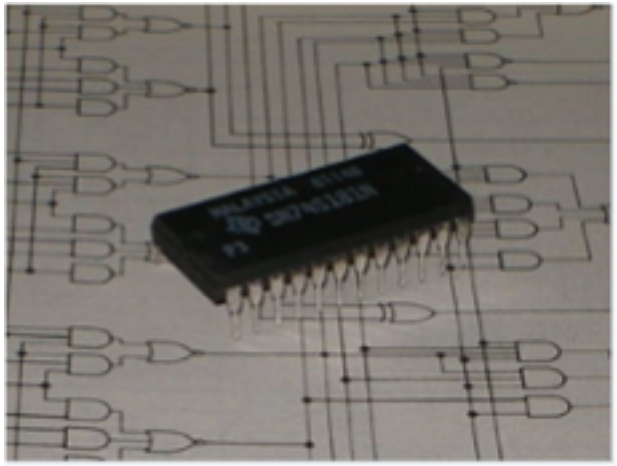
Data handling instructions involve retrieving or storing data in the computer's memory.

For example, saving a document in Microsoft Word requires a sequence of instructions that involves creating or updating a Word document that you can access later.

Arithmetic

Arithmetic instructions are just what they sound like: it's very common for the CPU to perform mathematical operations. The CPU can add, subtract, multiply, and divide, as well as compare numbers and perform operations on the individual bits of a number.

The **ALU(Arithmetic Logic Unit)** of the CPU executes all of these basic operations.



Control Flow

Control flow instructions help the CPU decide what to do next. A sequence of instructions may contain a built-in fork in the road; based on your computer's current conditions, the CPU might decide to skip over a few instructions or repeat some instructions.

Much more on this topic a bit later, but for now, just remember that some instructions can affect the order in which other instructions are executed!

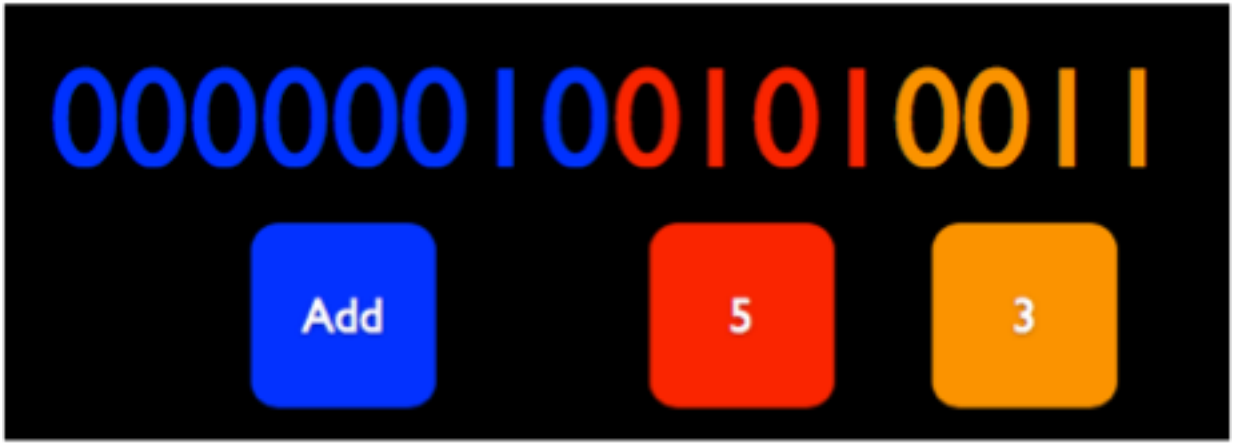
Operation Code

Each of these instructions is represented with a unique number called an **opcode**, which is short for "operation code". Any additional data used by an opcode are called **operands**.

For example, if an arithmetic instruction tells the CPU to add two numbers, the CPU is going to need to know what numbers to add. Since both opcodes and operands are represented with numbers, all of the instructions passed to a CPU can be expressed in binary.

In that sense, then, everything your computer does really does boil down to the zeroes and ones we saw earlier.

Decode An Operation Code



Add 5+3

Pipeline

Let's take a look at how the CPU actually goes about executing these instructions. One of the main goals of a CPU is to be as fast as possible. To do so, the CPU runs each instruction through a **pipeline**, which is just like a factory assembly line.

The task of building a car can be completed in a sequence of phases, where the assembly line will typically have a number of separate machines dedicated to performing a single task like securing a wheel or attaching the engine.

Fetch and Decode

Fetch. The first thing the CPU needs to do is... figure out what to do! Recall that our CPU will be executing some sequence of tasks. So, the fetch phase of the pipeline is responsible for determining the instruction that actually needs to be executed next.

Decode. Remember, our CPU's instructions are actually given as opcodes and operands that are represented using zeros and ones. The decode phase of the pipeline is responsible for figuring out what those zeroes and ones actually mean. After this phase is complete, the CPU knows exactly what it needs to do to execute the instruction.

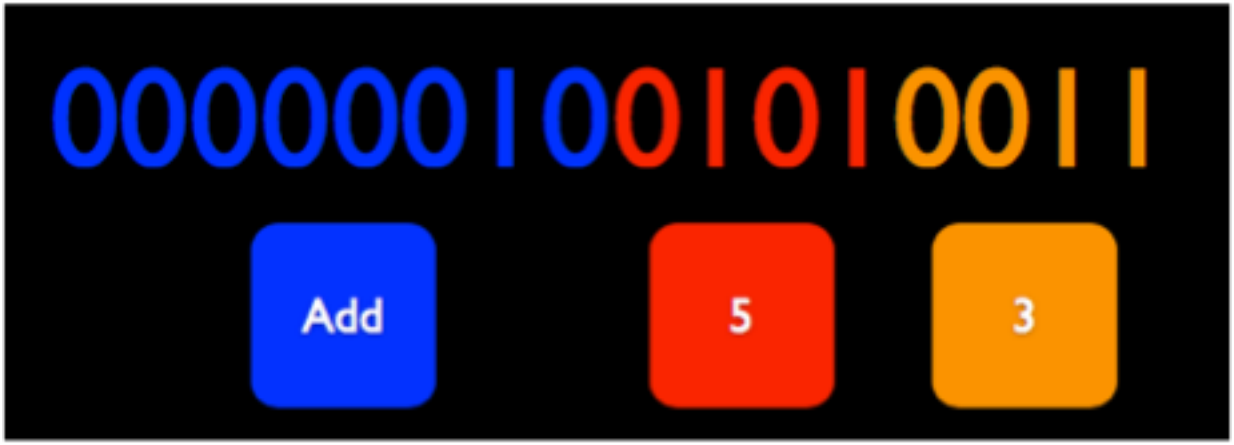
Execute and Writeback

Execute. The execute phase does just that: runs the requested computation. Here's where your CPU will perform an arithmetic operation, grab something from memory, etc.

Writeback. Our fourth and final stage is where we store the result of the computation. Instead, the writeback phase will write the result to memory somewhere, often in a special location on the CPU itself, so the result of the instruction can be used later.

Fetch

Decode



Add 5+3

Execute

5+3=8

Writeback

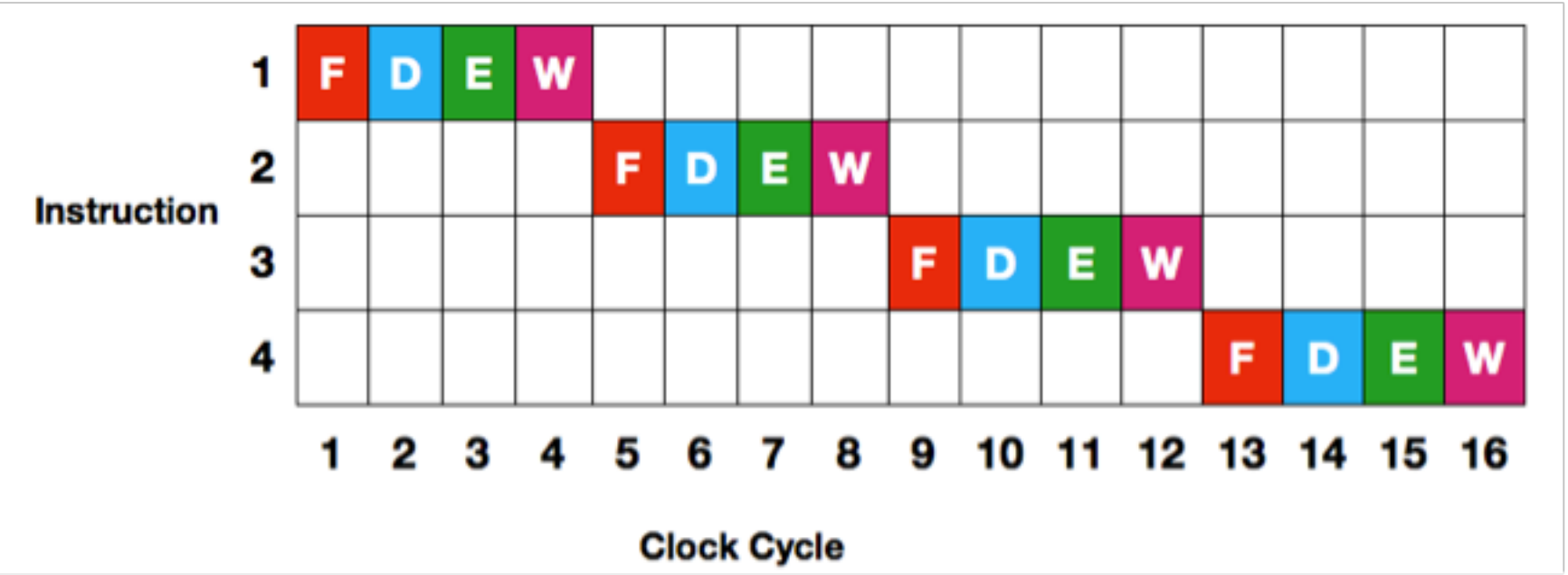
Save the value 8 in memory!

Cycle

Each instruction that we want the CPU to run will have to go through all four of these phases. Because the CPU is constantly running instructions, putting new things through the CPU pipeline is a cyclic process.

Visually, running four instructions through our pipeline looks something like the below, where each block in the diagram represents a single CPU **cycle**.

Pipeline

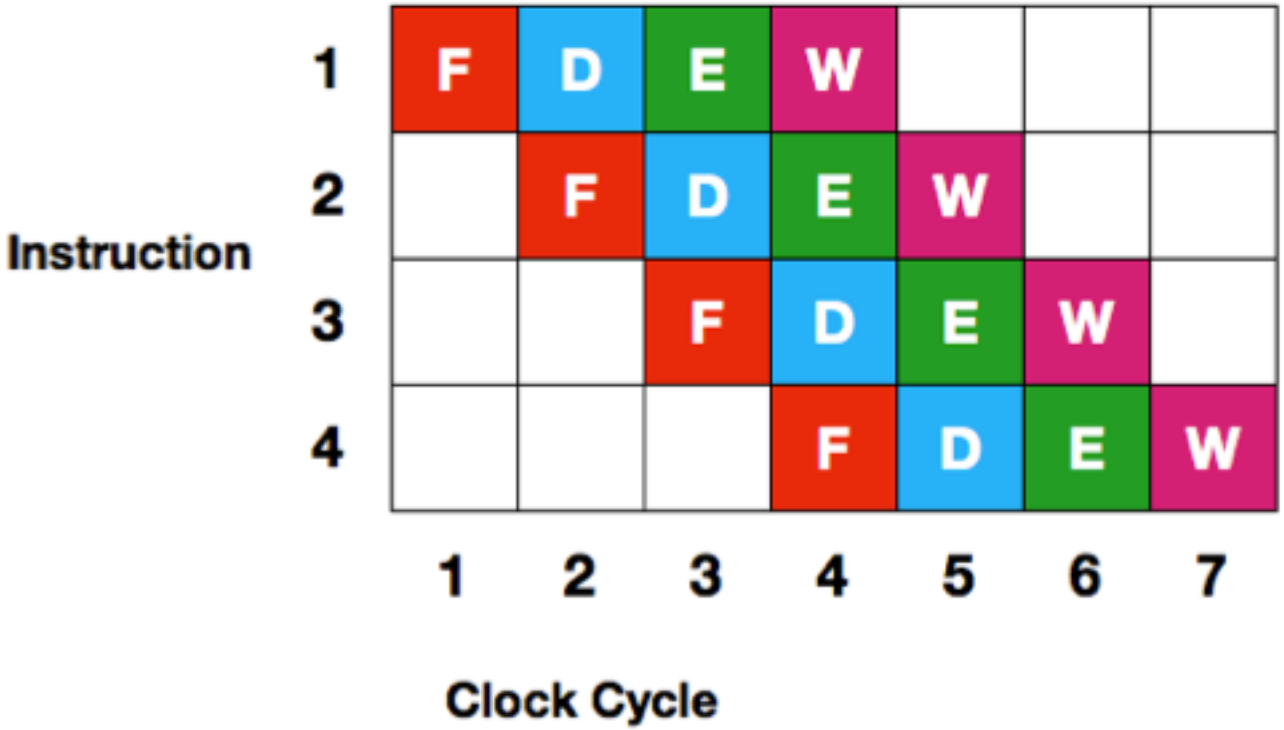


Parallelism

Because different parts of the CPU handle different parts of the pipeline, we can be more efficient through multi-tasking, or **parallelism**.

Rather than waiting for the rest of the pipeline to finish an instruction, the CPU can start fetching the *next* instruction before the current one completes! That way, the CPU is wasting a lot less time. Our new, more efficient pipeline now looks something like this:

Parallelism



Seven cycles instead of 16 cycles! So, we were able to make our CPU significantly faster by taking advantage of the fact that we can execute different stages of the pipeline in parallel, or at the same time.

Cores

While shopping for a new computer, you may have seen advertisements for **dual-core** or **quad-core** CPUs on new machines. In general, **multi-core** processors combine multiple CPUs into one. So, at any given time, a multi-core will be executing multiple pipelines at once.

Cores

A dual-core CPU has two cores, which is just like having two CPUs on your machine. Similarly, a quad-core CPU has four cores. However, adding more cores to your CPU won't necessarily make all software perform better.

For example, Intel i3 has two cores, i5 and i7 has four cores.

In fact, many computer programs were written before multi-core processors were commonly found in consumer computers. If these programs were written with only a single core in mind, then the existence of other cores won't really help if they're just sitting there doing nothing.

Clock Speed

There isn't one silver bullet when it comes to comparing the performance of two CPUs. A CPU's **clock speed**, nowadays measured in gigahertz (GHz), describes the rate at which the CPU executes instructions.

A gigahertz is a billion computations per second. Specifically, the clock speed measures how many cycles the CPU can complete in one second.

Moore’s Law

A higher clock speed means that the CPU can execute more instructions in a single second, so a CPU with a higher clock speed will perform faster than an identical CPU with a lower clock speed.

The clock speed of CPUs has been rising over the past several decades largely due to a trend known as **Moore's Law**, which states that processing power will double every 18 months. Specifically, it states that the number of **transistors** that can be fit into a chip of a fixed size can be doubled every 18 months.

Clock Speed

However, other factors also influence the overall performance of a CPU, so simply comparing the clock speeds of two CPUs won't necessarily tell you which is faster.

As we've seen, parallelism is an important contributor to the performance of modern processors. So, processors with two or four cores may have better performance than their single-core equivalents. The instruction set of a CPU can also affect its performance.

Megahertz Myth

For example, if one processor can complete a task using one instruction that takes another processor five instructions, then naturally, it may be able to complete certain tasks faster even if it has a slower clock speed.

Finally, the size of a CPU's pipeline can affect its performance, as explained in the video below(See last slide). In this 2001 MacWorld keynote, Apple coined the term "Megahertz Myth" to describe the issues surrounding the comparison of two CPUs using only clock speed.

Memory

Bytes

Name	Abbreviation	Size in bytes	Storage example
Byte	B	8 bits	Characters of text
Kilobyte	KB	10^3 bytes	Word document, small image
Megabyte	MB	10^6 bytes	MP3 song, large image
Gigabyte	GB	10^9 bytes	Movie, ~350 photos, ~250 songs
Terabyte	TB	10^{12} bytes	~350,000 photos, ~250,000 songs

Registers

Turns out that a few different types of memory are actually found directly on the CPU. The smallest and fastest memory on the CPU is found in **registers**. Registers hold extremely small amounts of data: on the order of several bytes.

Earlier, when we said that a CPU could add two numbers together, we took for granted where those numbers would be coming from and where the answer would go.

Registers

This is where registers come in! While the CPU is in the process of adding two numbers together, both numbers are stored in registers, and once the CPU has computed the answer, the result is also stored in a register.

Registers are also used to keep track of things like the instruction that is currently being decoded or executed as well as what instruction should be put into the pipeline next.

Registers

Since registers are so small, there's not much more that could even fit in a CPU register. But, in order for the CPU to quickly perform its addition, accessing register data must be extremely fast, so we have a bit of a trade-off between the size of the memory and the speed of the memory.

In fact, we'll see this trade-off become a trend throughout this section! The number of registers on a CPU can vary, with some CPUs having just 16 registers and others having as many as 256 registers.

RAM

Okay, but a few registers that can't hold much more than a 32-bit number don't seem sufficient for playing a several megabyte song or watching a several gigabyte movie.

In order to efficiently handle tasks like these, we'll need some short-term memory with a larger capacity. That's where **RAM**, or Random-Access Memory, comes in. RAM is a block of memory that can be used by currently-running process to store data.

RAM

For example, a web browser might need to store what website you're currently browsing, and your email client might need to store all of the emails in your inbox. So, the more RAM you have, the more space you have available for processes to store information.

Because all of the currently-running processes on your computer will probably need to utilize RAM for short-term storage, having more RAM in your computer will allow you to run more programs at the same time efficiently.

RAM

A laptop on today's market will likely have between four and eight gigabytes of RAM, so we're talking *much* more space than something like a CPU register. In fact, when you see any language that refers to the amount of "memory" in a computer, there's a very good chance it's referring to RAM, since RAM is the main source of short-term memory in your computer.

RAM

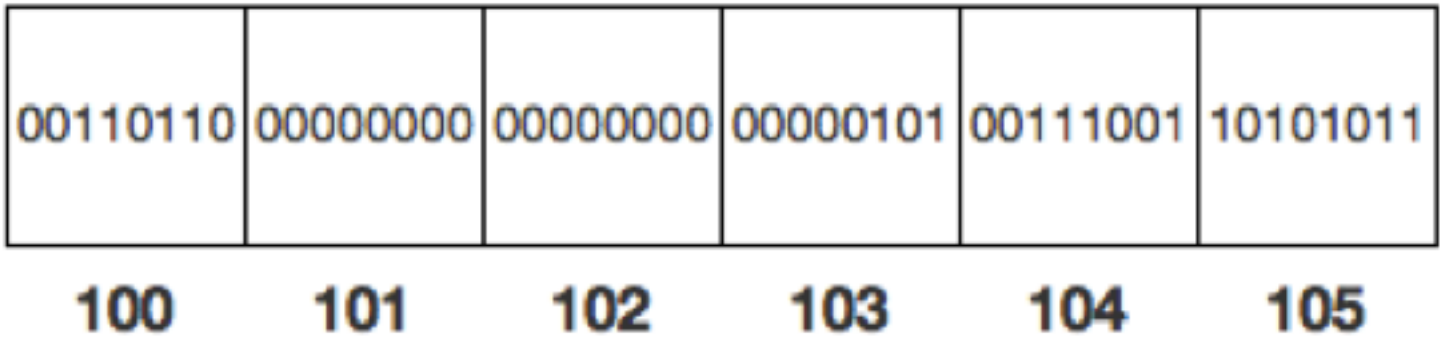
Your laptop probably contains at least a couple of sticks of RAM. The CPU has the ability to **read** values from RAM (i.e., access already-stored data) and **write** values to RAM (i.e., store new data).

We can think about RAM as a really long street with lots of houses. Each house has an **address**, which is simply a unique whole number starting from zero, that is used to identify it. Each one of these houses can store exactly one byte (remember, 8 bits!) of data.

RAM

So, the total number of houses available to your computer depends on the total size of the available RAM. With 2 GB of RAM, for example, your computer will have about two billion houses, and with 4 GB of RAM, your computer will have about four billion houses.

When the CPU needs to read or write information from RAM, it will do so using a memory address. For example, let's say the following represents a portion of RAM.



Let's say that a 32-bit (aka 4-byte) number has been broken up into chunks and stored at the addresses 101-104. Putting these four blocks of memory together, we can see that the number stored at the address 101 in RAM is 00000000000000000000010100111001, or 1337.

RAM

We call this way of representing numbers in RAM, in which the bytes are read from left to right, **big-endian**. Here, the most significant byte is stored at a lower address than the least significant byte.

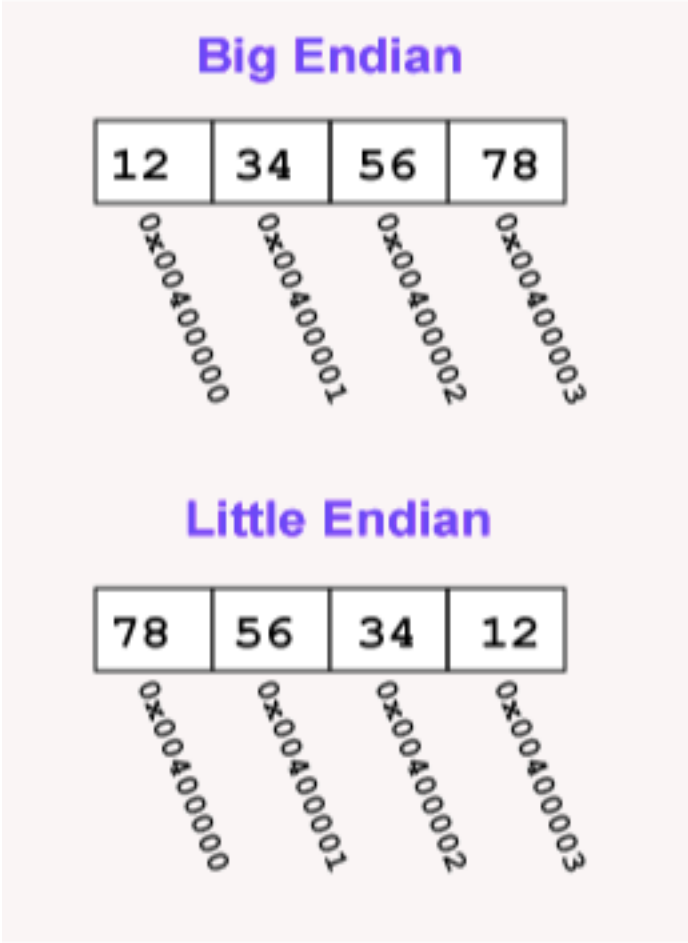
It's actually not uncommon for computers to store bytes in the opposite order, with more significant bytes stored at *higher* addresses in memory. This is called **little-endian**.

RAM

Suppose you want to store 32-bit Hexadecimal number 0x12345678. (the 0x is to indicate that it is in Hexadecimal.)

Big Endian: “big end” or the most significant byte is stored at the lower end address.

Small Endian: “small end” or the least Significant byte is stored at the lower end address



RAM

The "Random-Access" part of RAM refers to the fact that accessing any address in RAM takes the same amount of time. It's no faster, for example, to access the information stored at the address 0 than it is to access the information stored at the address 1048576. As we'll see shortly, this isn't the case with all types of memory.

Cache

Cache

However, accessing data stored in RAM is significantly slower than accessing data stored in a CPU register. In fact, the CPU can typically complete a cycle much faster than it can read a value from RAM, so the CPU could waste cycles while waiting for a value to be read from RAM.

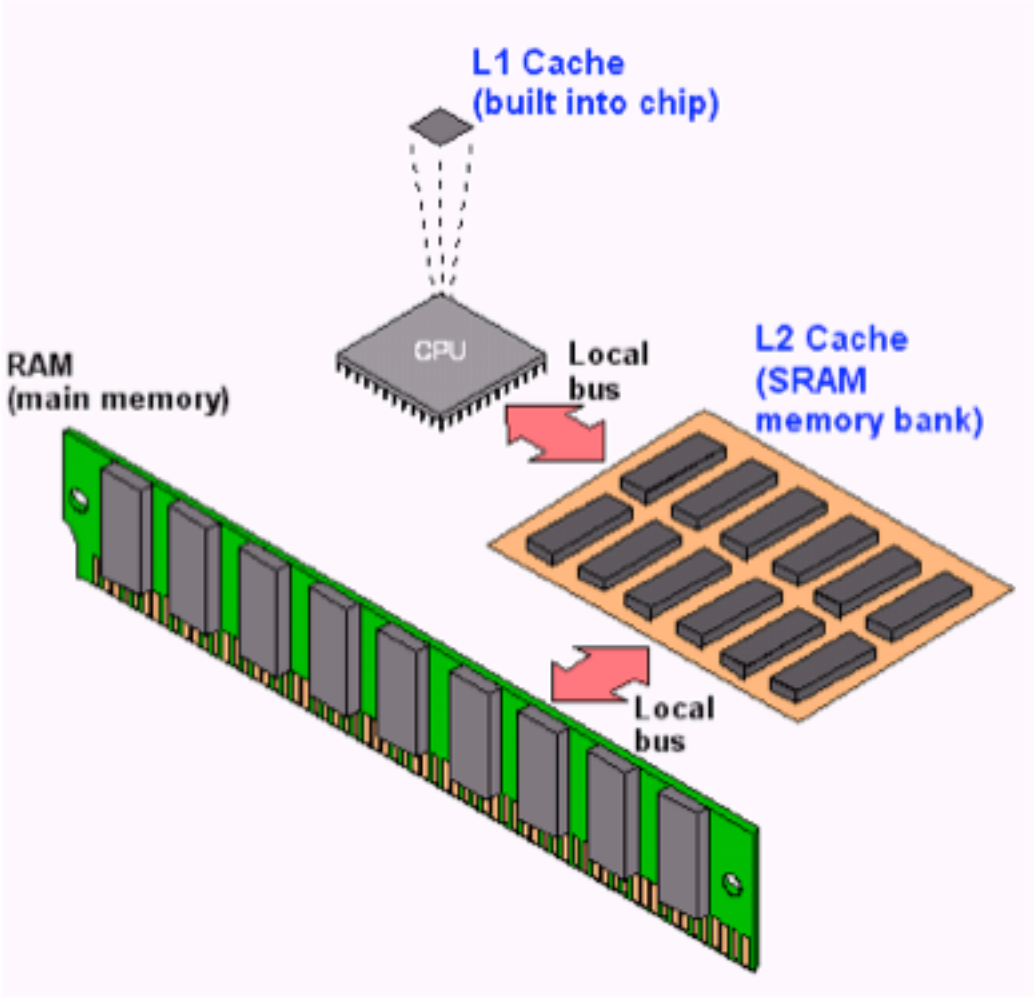
In order to increase efficiency and waste less time, the CPU also utilizes a layer of storage called the **processor cache**. Not only is the cache physically located closer to the CPU, but data stored on the cache can be accessed more quickly than data stored in RAM.

Cache

In many cases, the CPU will need to use some value from RAM more than once, just like you might listen to a song on repeat.

Rather than going all the way to RAM multiple times to grab a frequently-accessed value, the CPU might instead place it on the cache after getting the value once from RAM, where it can be fetched much more quickly in the future.

Now, the next time we need that value, we can take a much shorter trip to the cache to get it, rather than going all the way to RAM.


L1 Cache
(built into chip)
CPU
Local
bus
L2 Cache
(SRAM
memory bank)
RAM
(main memory)
Local
bus

Cache

Processor caches are able to hold significantly less data than RAM. As shown in the above diagram, computers typically have several **levels** of caches, each having different sizes and speeds.

The **L1 cache** is the smallest, fastest, and closest to the CPU, and the L1 cache typically comprises several (e.g., 32 or 64) kilobytes in size. Next up is the L2 cache, which is typically a bit farther away from the CPU and slower to access, but usually holds up to several megabytes (e.g., 4 or 8) of data.

Cache

Finally, the L3 cache is even slower, but can hold even more data. Not all CPUs have all three levels of caches—some CPUs don't utilize an L1 cache, while others don't have an L3 cache. All that being said, accessing data in the CPU cache is still *much* faster than accessing data from RAM.

HDD

You computer's **hard drive** is its primary form of long-term storage. Unlike RAM, which holds only a few gigabytes, you'll find that modern hard drives often hold 500, 750, or 1000+ GB.

There are two options for a hard drive: **HDD(Hard Disk Drive)** or **SSD(Solid State Drive)**. HDD is mechanical and can tend to fail but is cheaper. SSD is NAND-based flash memory, which is a type of **non-volatile memory**, memory that is persistent even without power.

The hard drive is even further away from the CPU and accessing data from the hard drive is much slower than accessing it from the RAM.

HDD

A hard disk from 1956.

Yes, the entire box. Its capacity?

5 mb.

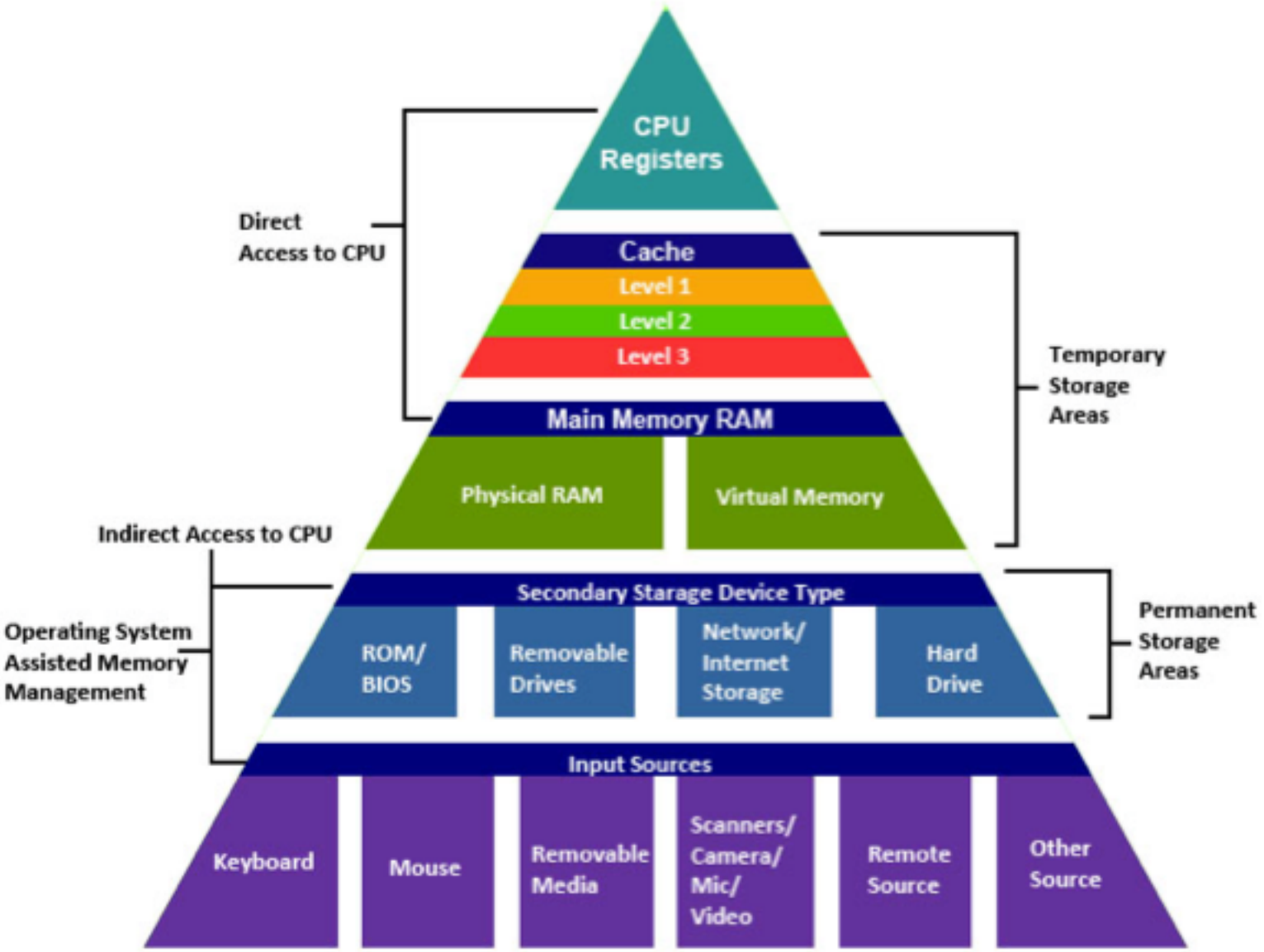
(Can store Despacito, the Justin Bieber version of course.)

Filesystem

How exactly your files are accessed and stored depends on how it is **formatted**, which determines which **filesystem** it uses.

Many Windows PCs have hard drives that utilize the proprierary **NTFS** filesystem, while USB thumb drives may be formatted using **FAT32**, an older filesystem. **exFat** is the newer version of FAT32.

While operating systems may support multiple filesystems, you may sometimes encounter a device that your computer can't read from or write to. For example, some versions of OS X can read files from an NTFS drive, but can't write any new data. exFat is a good option if you want read and write capability for both Windows and Mac.


CPU Registers
Direct Access to CPU
Cache
Level 1
Level 2
Level 3
Temporary Storage Areas
Main Memory RAM
Physical RAM
Virtual Memory
Indirect Access to CPU
Secondary Starage Device Type
Operating System Assisted Memory Management
ROM/ BIOS
Removable Drives
Network/ Internet Storage
Hard Drive
Permanent Storage Areas
Input Sources
Keyboard
Mouse
Removable Media
Scanners/ Camera/ Mic/ Video
Remote Source
Other Source

How long to retrieve 1MB?

Storage type	Access time	Relative access time
L1 cache	0.5 ns	Blink of an eye
L2 cache	7 ns	4 seconds
1MB from RAM	0.25 ms	5 days
1MB from SSD	1 ms	23 days
HDD seek	10 ms	231 days
1MB from HDD	20 ms	1.25 years

Homework

1) Read and reread these lecture notes.
2) Do the problem set.
3) Megahertz Myth: https://www.youtube.com/watch?v=tPBtXUUeFK0
4) Inside a HDD: https://www.youtube.com/watch?v=kdmLvl1n82U
5) SSD vs HDD: https://www.youtube.com/watch?v=i84eEjP-RL4
6) The Making of a Chip: https://www.youtube.com/watch?v=UvluuAIiA50

References

This lecture is a summary of a lecture from an OpenCourseWare course below.

Computer Science E-1 at Harvard Extension School
Understanding Computers and the Internet
by Tommy MacWilliam.