

CSCI 476: Computer Security

Lecture 2: Computer Systems Review

Reese Pearsall
Fall 2023

Announcements

TA

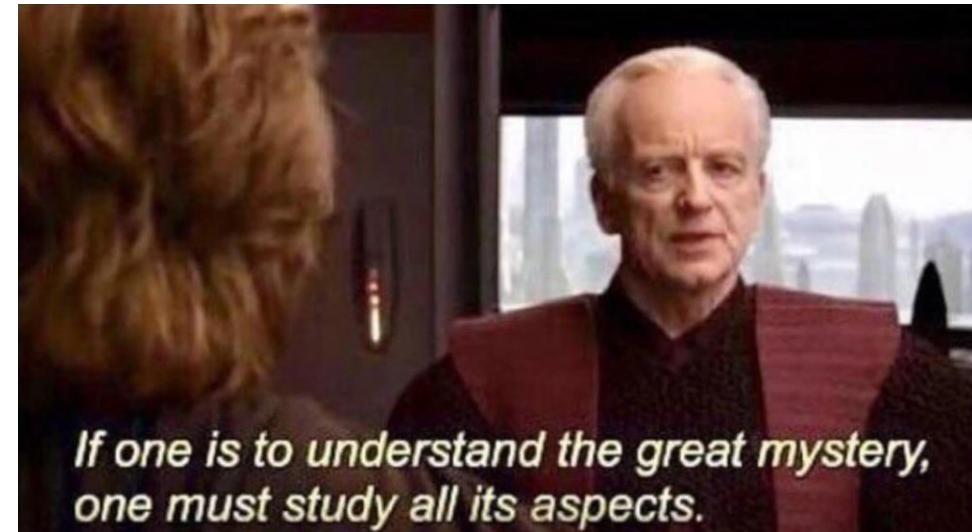
- **Gerard Fuhnwi**
- gerardfuhnwi@yahoo.com
- Office Hours: Tuesdays 10am – 12pm
- Location: Barnard 259

Lab 0 posted → Due **Sunday** September 10th

- I pushed it back a week
 - I uploaded some instructions to the course website for M1/M2 users
 - I want to make sure **everyone** can get something working

Computer Systems Review

To understand the technical aspects of security, we must have a good understanding of how computers work



*If one is to understand the great mystery,
one must study all its aspects.*

What is a computer?



What is a computer?

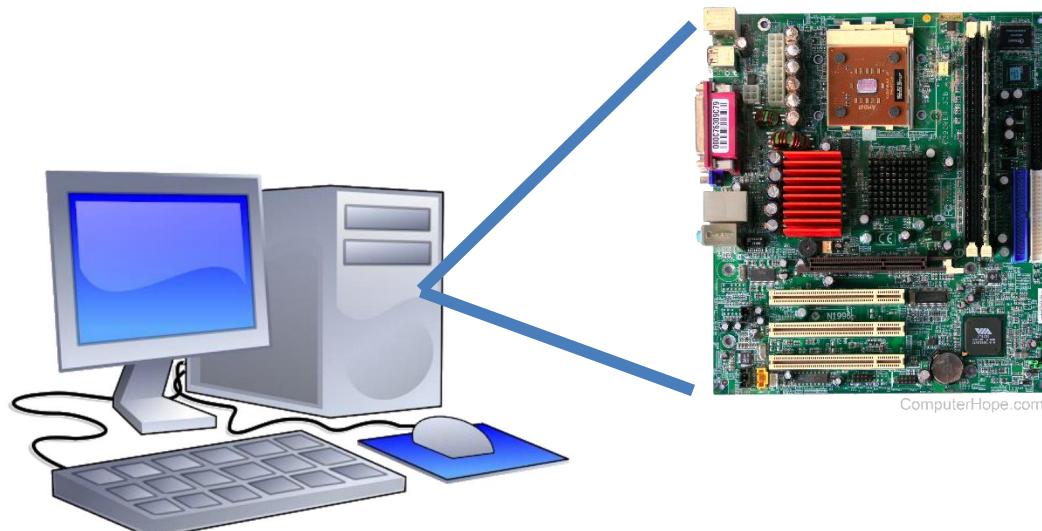


A magical box that does stuff



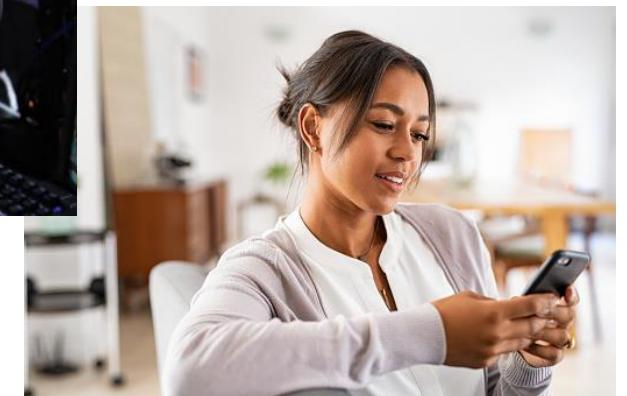
What is a computer?

A **semi**-magical box that does stuff **executes instructions**



What is so magical about a computer?

We use computers every day for many different things



What is so magical about a computer?

Big Idea

Computers only understand instructions in the form of 0s and 1s (binary)

Welcome to CSCI 476



01010111 01100101 01101100 01100011 01101111
01101101 01100101 00100000 01110100 01101111
00100000 01000011 01010011 01000011 01001001
00100000 00110100 00110111 00110110



How does this happen?

??????

??????

From a high level, we will divide a computer system into two parts

How does this happen?

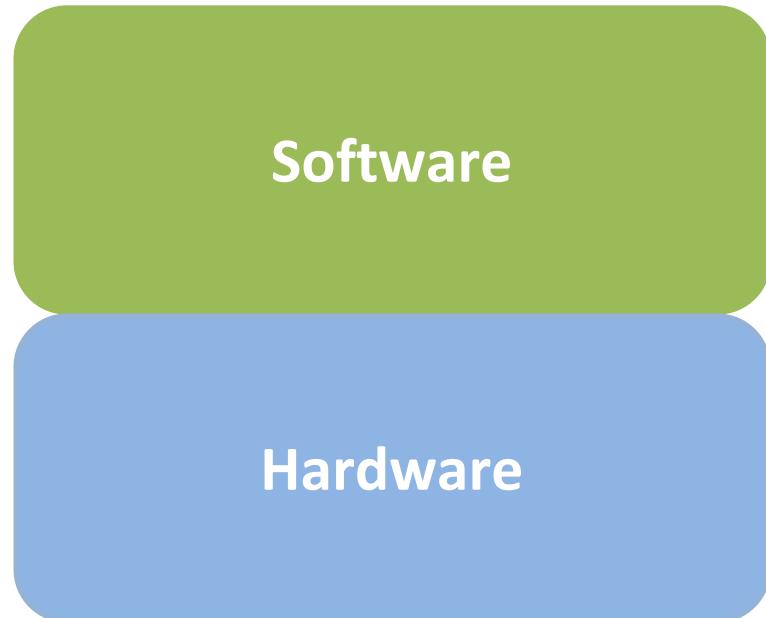
??????

Hardware

From a high level, we will divide a computer system into two parts

I. Hardware

How does this happen?



From a high level, we will divide a computer system into two parts

- I. **Hardware**
- II. **Software**

How does this happen?

Software

Hardware

Symbiotic relationship



From a high level, we will divide a computer system into two parts

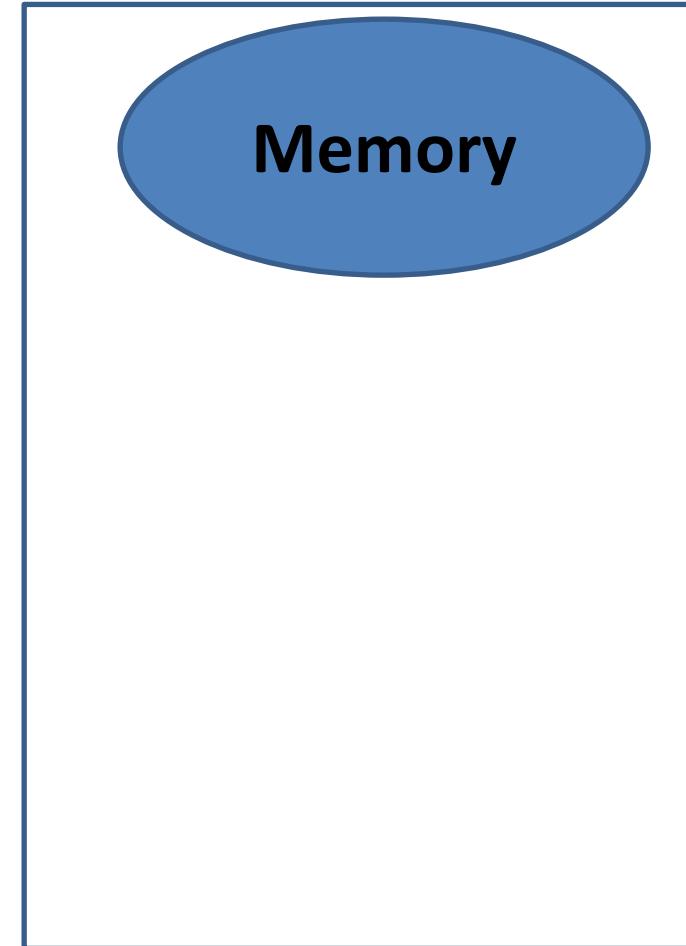
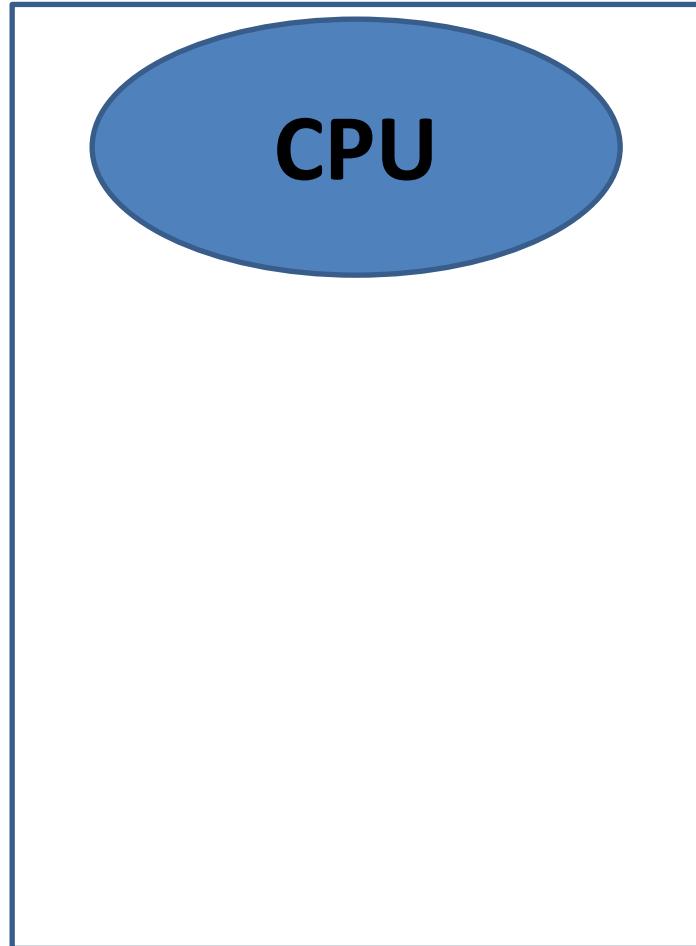
- I. **Hardware**
- II. **Software**

I. Hardware

The **physical** parts of a computer

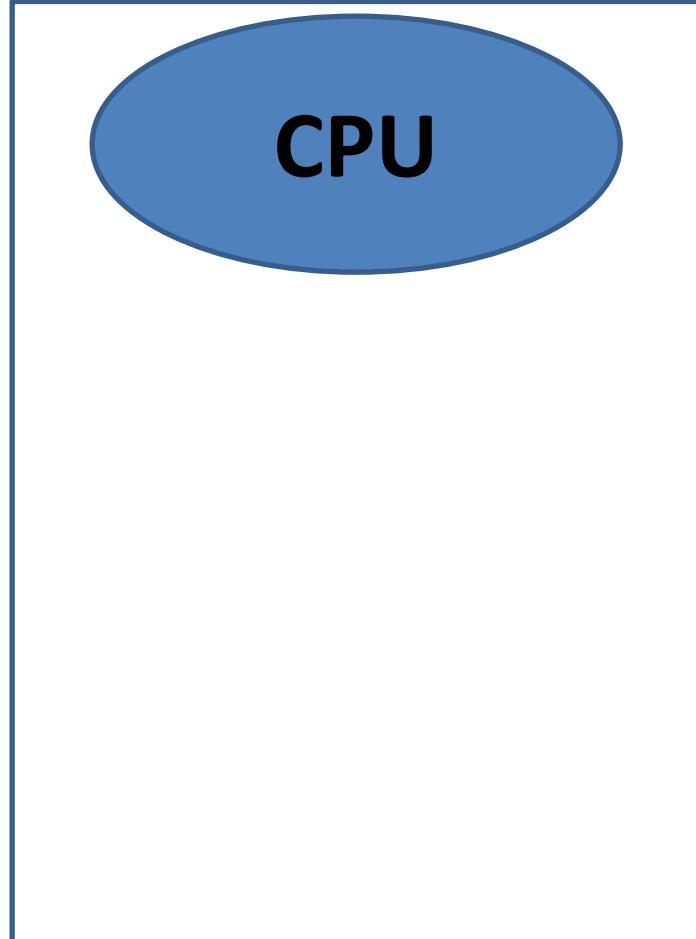


I. Hardware

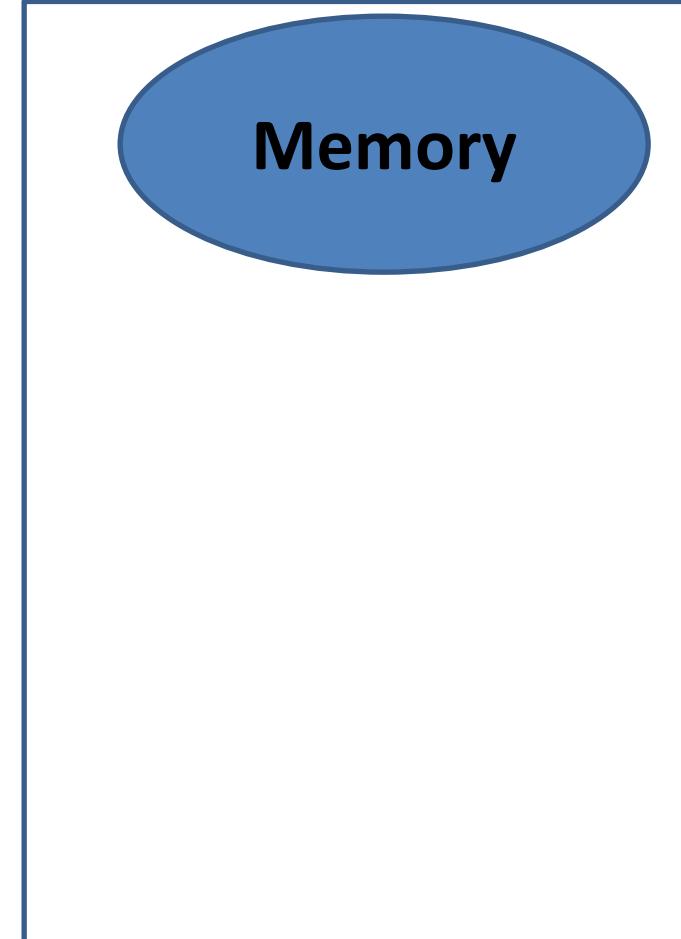


I. Hardware

Brain with no short-term memory

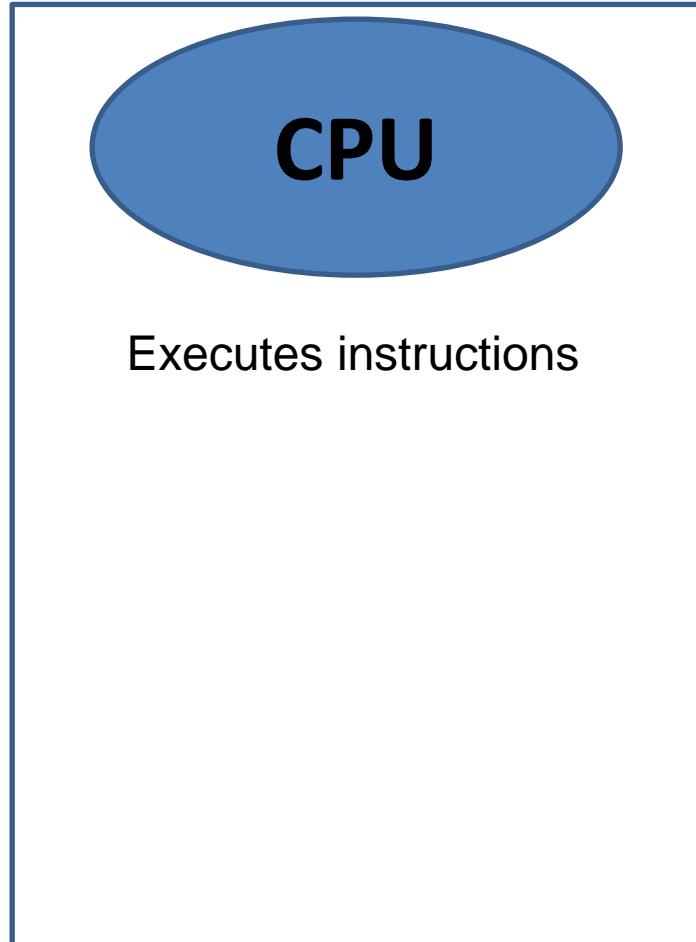


Scratch Pad

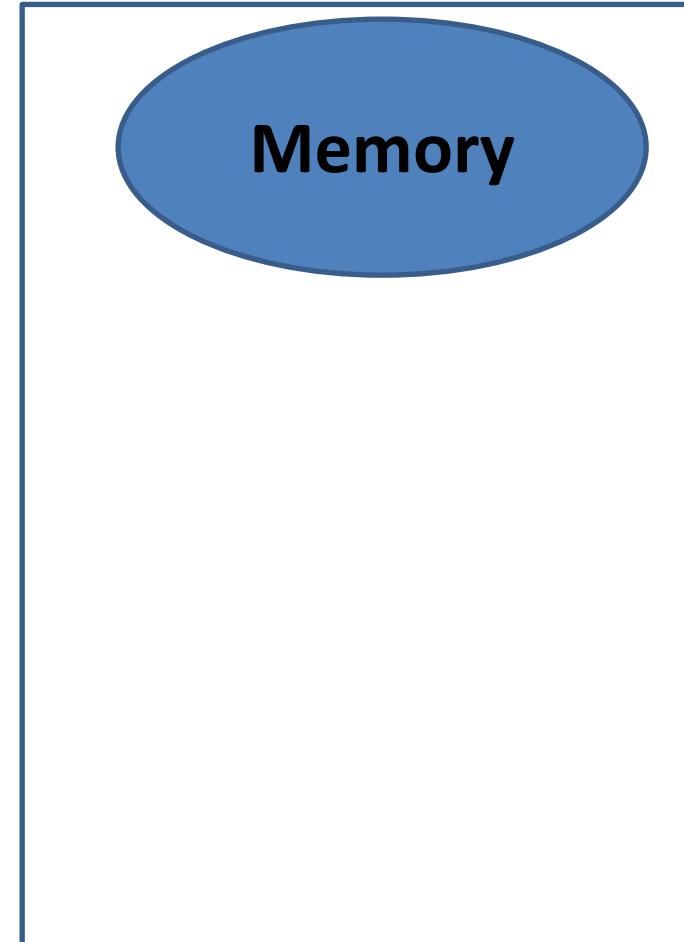


I. Hardware

Brain with no short-term memory



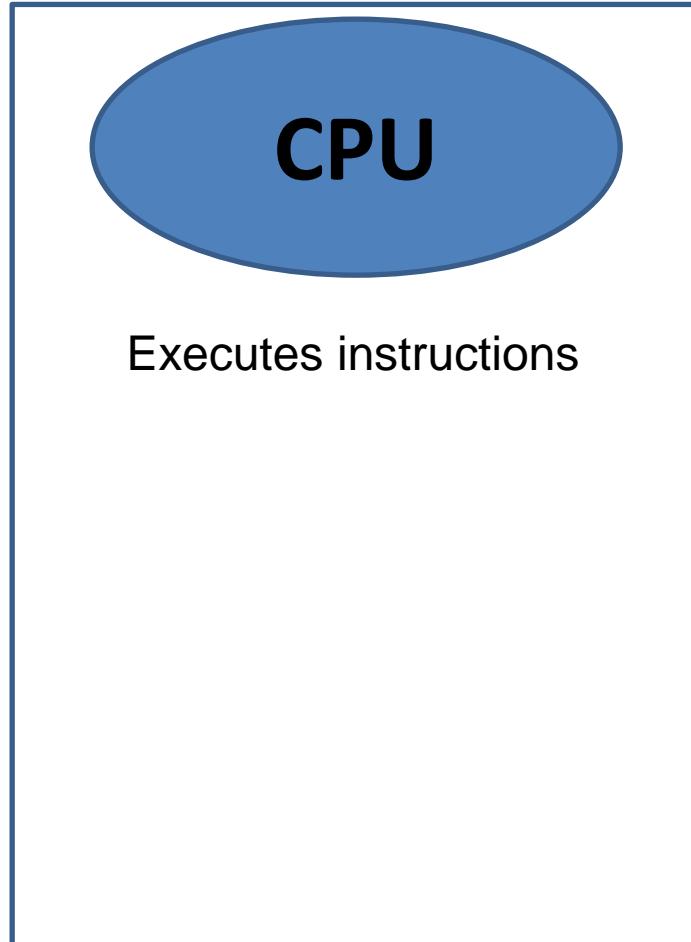
Scratch Pad



I. Hardware



Brain with no short-term memory



How does it “execute” instructions?

It is sent instructions from another part of the computer

I. Hardware



Brain with no short-term memory

CPU

Executes instructions

How does it “execute” instructions?

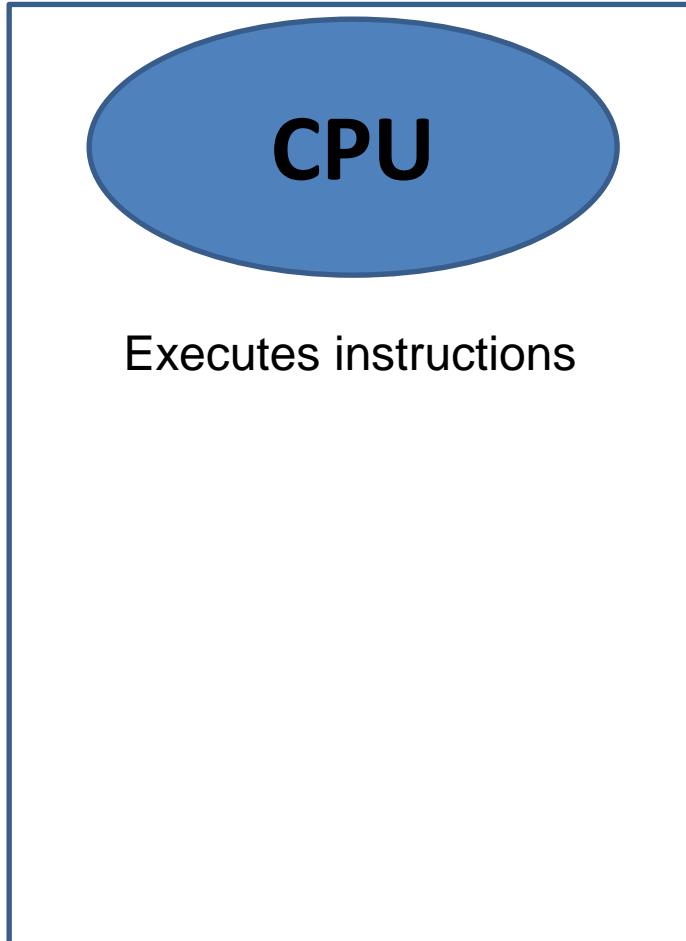
It is sent instructions from another part of the computer

01001100000000110100011100001010

I. Hardware



Brain with no short-term memory



How does it “execute” instructions?

It is sent instructions from *another part of the computer*

00000000101000010001100000100000 → 00 A1 18 20

00 A1 18 20

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Hex_(hexadecimal) is a common representation for binary

I. Hardware



Brain with no short-term memory

CPU

Executes instructions

How does it “execute” instructions?

It is sent instructions from another part of the computer

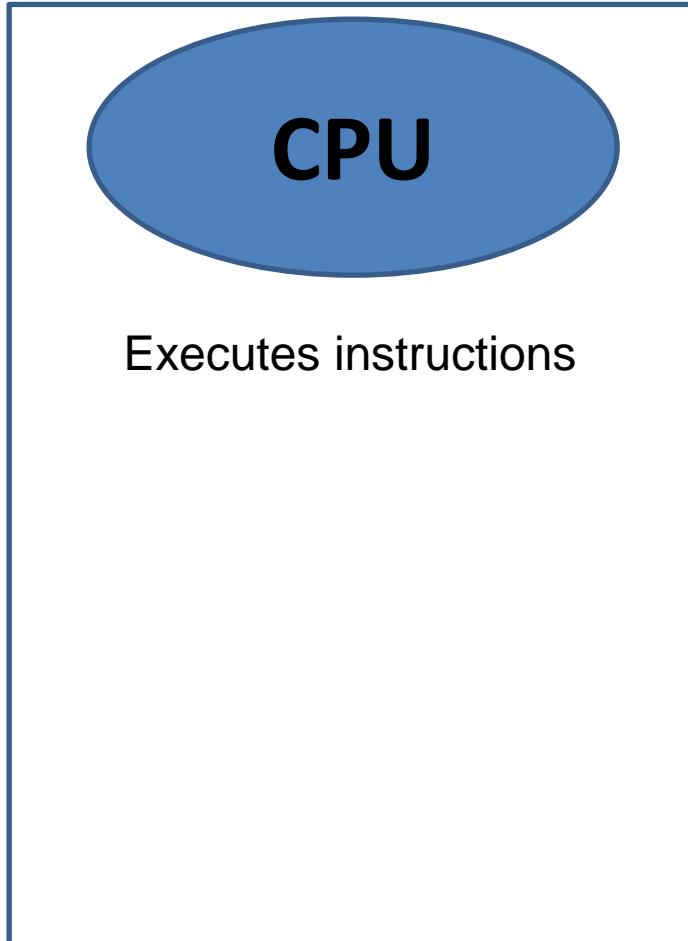
00000000101000010001100000100000

What does this instruction do?????

I. Hardware



Brain with no short-term memory



How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

Opcode



Common MIPS instructions

Notes: *op, funct, rd, rs, rt, imm, address, shamt* refer to fields in the instruction format
PC is assumed to point to the next instruction, **Mem** is the byte addressed main memory

Assembly Instruction	Instr Format	op op/funct	Meaning	Comments
add \$rd, \$rs, \$rt	R	0/32	\$rd = \$rs + \$rt	Add contents of two registers
sub \$rd, \$rs, \$rt	R	0/34	\$rd = \$rs - \$rt	Subtract contents of two registers
addi \$rt, \$rs, imm	I	8	\$rt = \$rs + imm	Add signed constant
addu \$rd, \$rs, \$rt	R	0/33	\$rd = \$rs + \$rt	Unsigned, no overflow
subu \$rd, \$rs, \$rt	R	0/35	\$rd = \$rs - \$rt	Unsigned, no overflow
addiu \$rt, \$rs, imm	I	9	\$rt = \$rs + imm	Unsigned, no overflow
mfc0 \$rt, \$rd	R	16	\$rt = \$rd	\$rd = coprocessor register (e.g. epc, cause, status)
mult \$rs, \$rt	R	0/24	Hi, Lo = \$rs * \$rt	64 bit signed product in Hi and Lo
multu \$rs, \$rt	R	0/25	Hi, Lo = \$rs * \$rt	64 bit unsigned product in Hi and Lo
div \$rs, \$rt	R	0/26	Lo = \$rs / \$rt, Hi = \$rs mod \$rt	
divu \$rs, \$rt	R	0/27	Lo = \$rs / \$rt, Hi = \$rs mod \$rt (unsigned)	
mfhi \$rd	R	0/16	\$rd = Hi	Get value of Hi
mflo \$rd	R	0/18	\$rd = Lo	Get value of Lo
and \$rd, \$rs, \$rt	R	0/36	\$rd = \$rs & \$rt	Logical AND
or \$rd, \$rs, \$rt	R	0/37	\$rd = \$rs \$rt	Logical OR
andi \$rt, \$rs, imm	I	12	\$rt = \$rs & imm	Logical AND, unsigned constant
ori \$rt, \$rs, imm	I	13	\$rt = \$rs imm	Logical OR, unsigned constant
sll \$rd, \$rs, shamt	R	0/0	\$rd = \$rs << shamt	Shift left logical (shift in zeros)
srl \$rd, \$rs, shamt	R	0/2	\$rd = \$rs >> shamt	Shift right logical (shift in zeros)

Common MIPS instructions

Notes: *op, funct, rd, rs, rt, imm, address, shamt* refer to fields in the instruction format

PC is assumed to point to the next instruction, **Mem** is the byte addressed main memory

Assembly Instruction	Instr Format	op op/funct	Meaning	Comments
add \$rd, \$rs, \$rt	R	0/32	\$rd = \$rs + \$rt	Add contents of two registers
sub \$rd, \$rs, \$rt	R	0/34	\$rd = \$rs - \$rt	Subtract contents of two registers
addi \$rt, \$rs, imm	I	8	\$rt = \$rs + imm	Add signed constant



MIPS Instruction formats

Format	Bits 31-26	Bits 25-21	Bits 20-16	Bits 15-11	Bits 10-6	Bits 5-0
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt		imm	
J	op			address		

Instruction	Format	Op	Rs	Hi	Lo	Comments
div \$rs, \$rt	R	0/26		Lo = \$rs / \$rt, Hi = \$rs mod \$rt		
divu \$rs, \$rt	R	0/27		Lo = \$rs / \$rt, Hi = \$rs mod \$rt (unsigned)		
mfhi \$rd	R	0/16		\$rd = Hi		Get value of Hi
mflo \$rd	R	0/18		\$rd = Lo		Get value of Lo
and \$rd, \$rs, \$rt	R	0/36		\$rd = \$rs & \$rt		Logical AND
or \$rd, \$rs, \$rt	R	0/37		\$rd = \$rs \$rt		Logical OR
andi \$rt, \$rs, imm	I	12		\$rt = \$rs & imm		Logical AND, unsigned constant
ori \$rt, \$rs, imm	I	13		\$rt = \$rs imm		Logical OR, unsigned constant
sll \$rd, \$rs, shamt	R	0/0		\$rd = \$rs << shamt		Shift left logical (shift in zeros)
srl \$rd, \$rs, shamt	R	0/2		\$rd = \$rs >> shamt		Shift right logical (shift in zeros)

I. Hardware



Brain with no short-term memory

CPU

Executes instructions



Must decipher
what instruction
to execute

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

Opcode

I. Hardware



Brain with no short-term memory

CPU

Executes instructions



Must decipher
what instruction
to execute

How does it “execute” instructions?

It is sent instructions from another part of the computer

0000000010100010001100000100000

Opcode \$rs

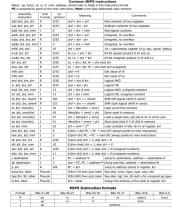
I. Hardware



Brain with no short-term memory

CPU

Executes instructions



Must decipher
what instruction
to execute

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

Opcode \$rs \$rt

I. Hardware



Brain with no short-term memory

CPU

Executes instructions



Must decipher
what instruction
to execute

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

Opcode \$rs \$rt \$rd

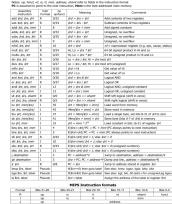
I. Hardware



Brain with no short-term memory

CPU

Executes instructions



Must decipher
what instruction
to execute

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

Opcode \$rs \$rt \$rd shamt

I. Hardware



Brain with no short-term memory

CPU

Executes instructions



Must decipher
what instruction
to execute

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

Opcode \$rs \$rt \$rd shamt funct

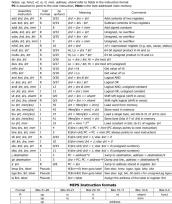
I. Hardware



Brain with no short-term memory

CPU

Executes instructions



Must decipher
what instruction
to execute

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

Opcode

\$rs

\$rt

\$rd

shamt

funct

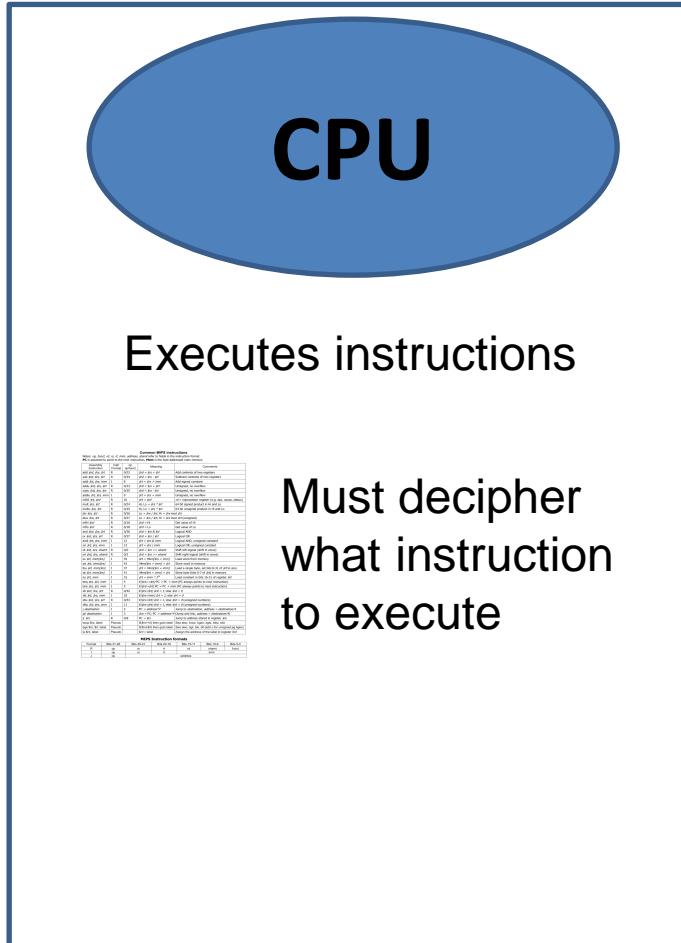
Damn.... I kinda **don't care**



I. Hardware



Brain with no short-term memory



00001
00011
00101
...
11111

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

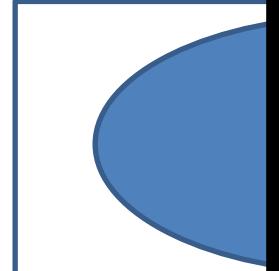
Opcode \$rs \$rt \$rd shamt funct



\$ denotes that it is a **register**

I. Hardware

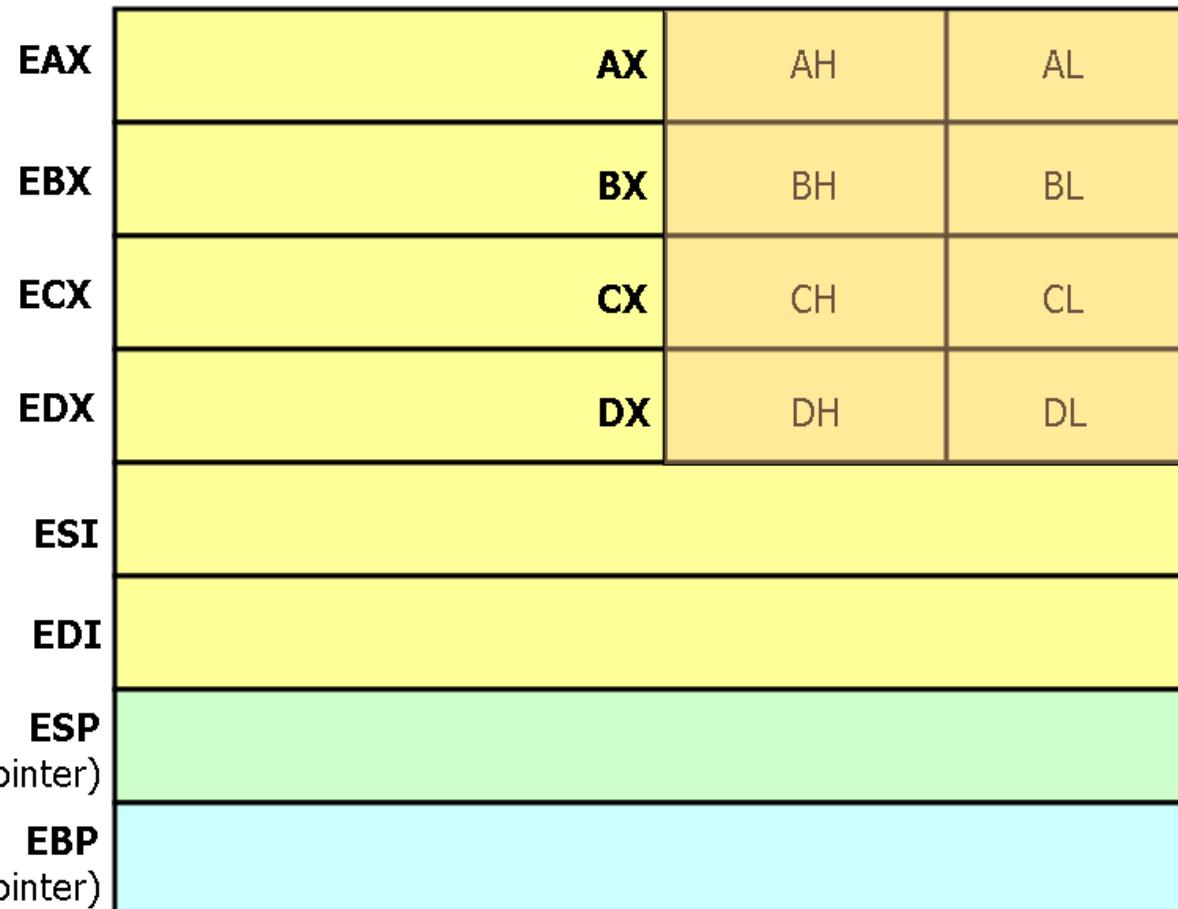
Brain with no



Execution



General-purpose Registers



Instructions?

part of the computer

00000100000

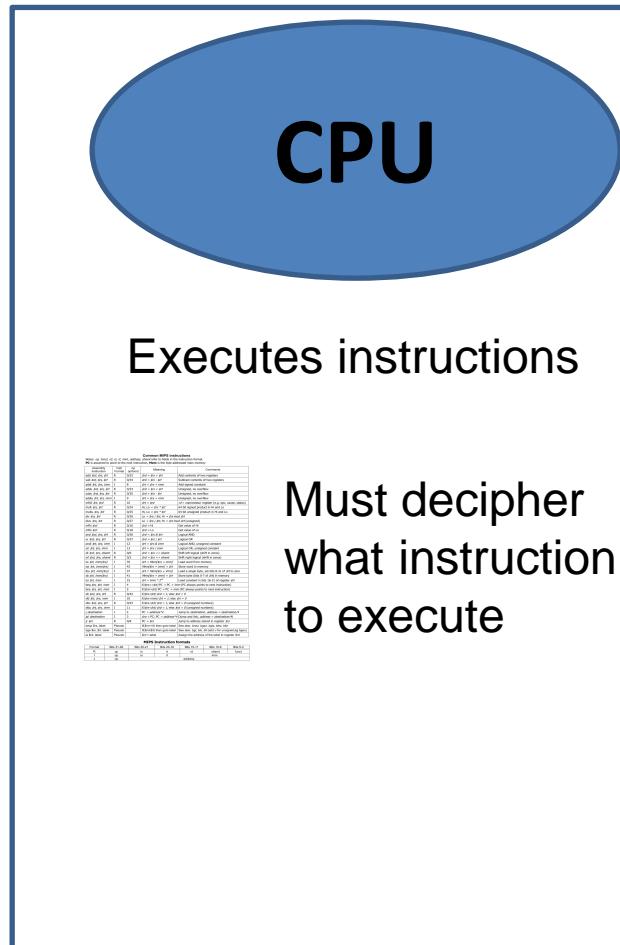
shamt funct

er

I. Hardware



Brain with no short-term memory



Registers

000001

00011

00101

11111

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

Opcodes

\$rs

\$r

\$re

sham

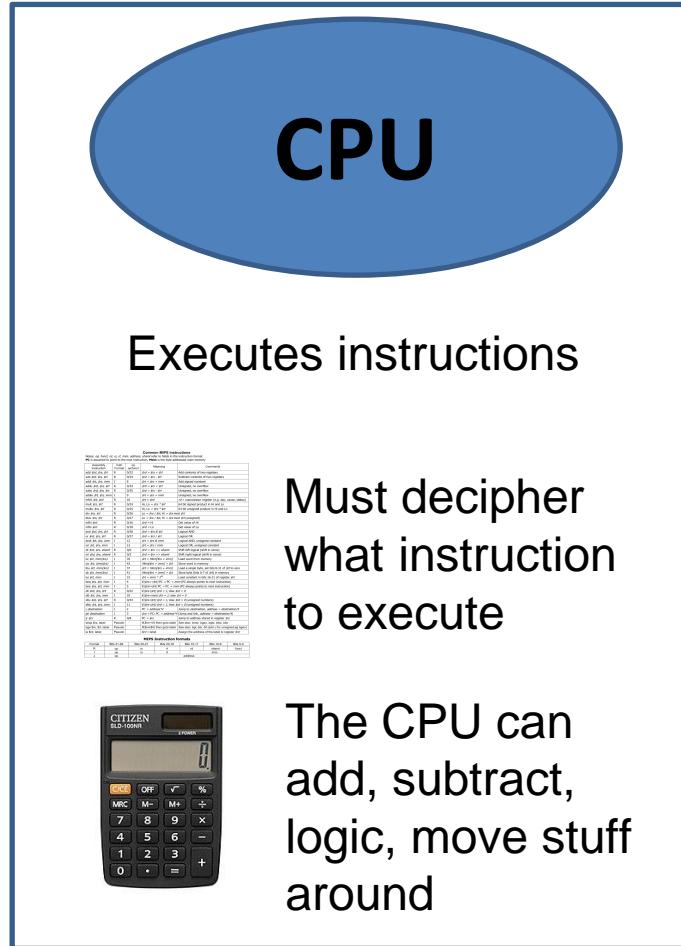
funct

ADD \$rs, \$rt, \$rc

I. Hardware



Brain with no short-term memory



Registers

5
??
3
...
11111

How does it “execute” instructions?

It is sent instructions from another part of the computer

00000000101000010001100000100000

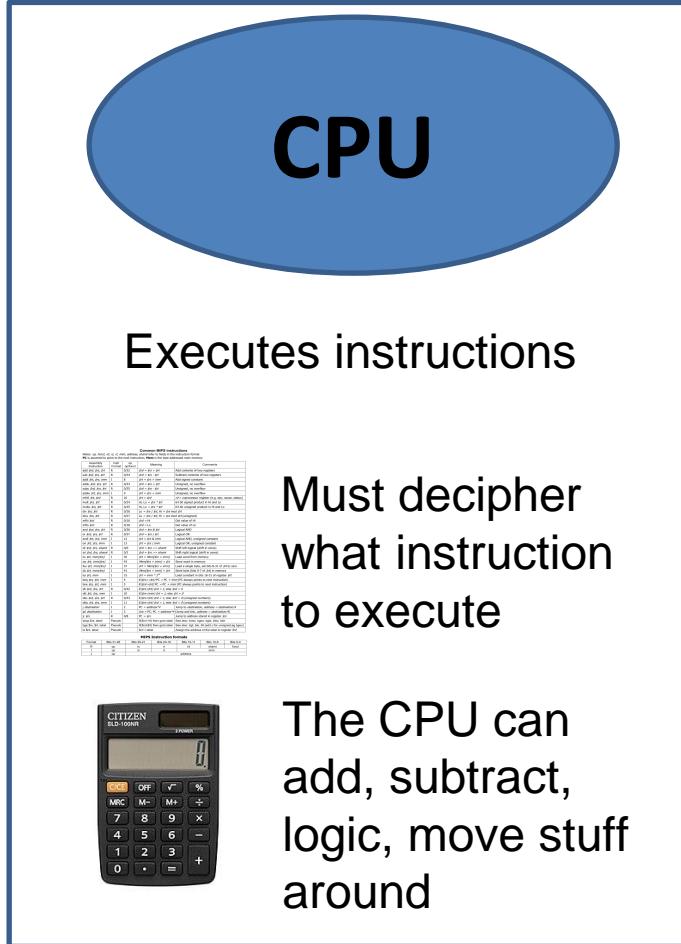
Opcode \$rs \$rt \$rd shamt funct

ADD \$rs, \$rt, \$rd

I. Hardware



Brain with no short-term memory



Registers

5
8
3
...
11111

How does it “execute” instructions?

It is sent instructions from another part of the computer

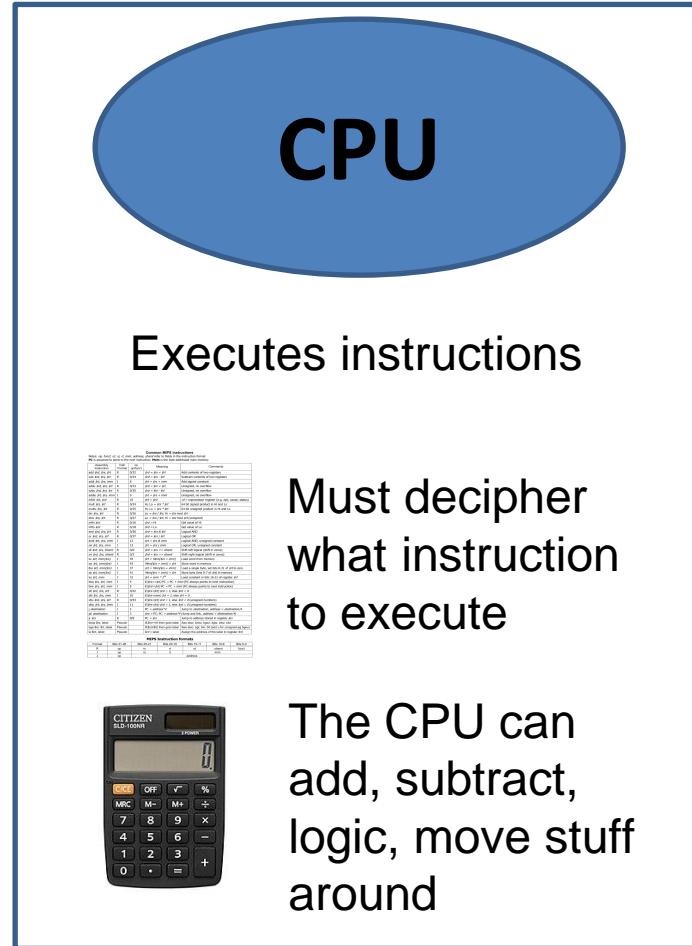
00000000101000010001100000100000

Opcode \$rs \$rt \$rd shamt funct

ADD \$rs, \$rt, \$rd

I. Hardware

Brain with no short-term memory



Regis

0000

000

1

5

8

3

...

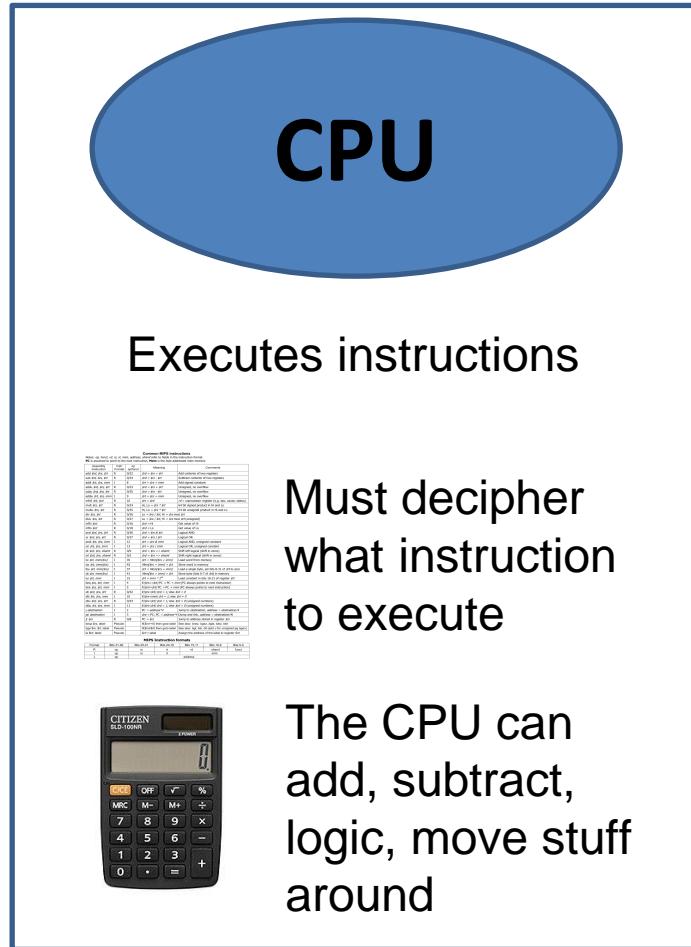
CPU uses

Electricity™



I. Hardware

Brain with no short-term memory

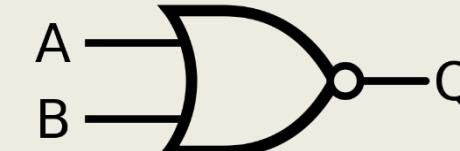
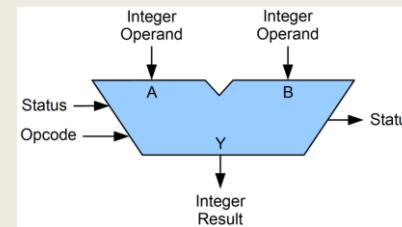


Registers

0000
0001
0010
0011
...1

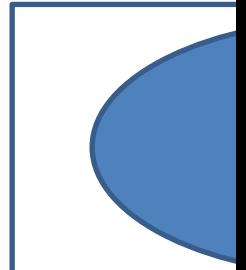
CPU uses
Electricity™

To decipher and execute instructions

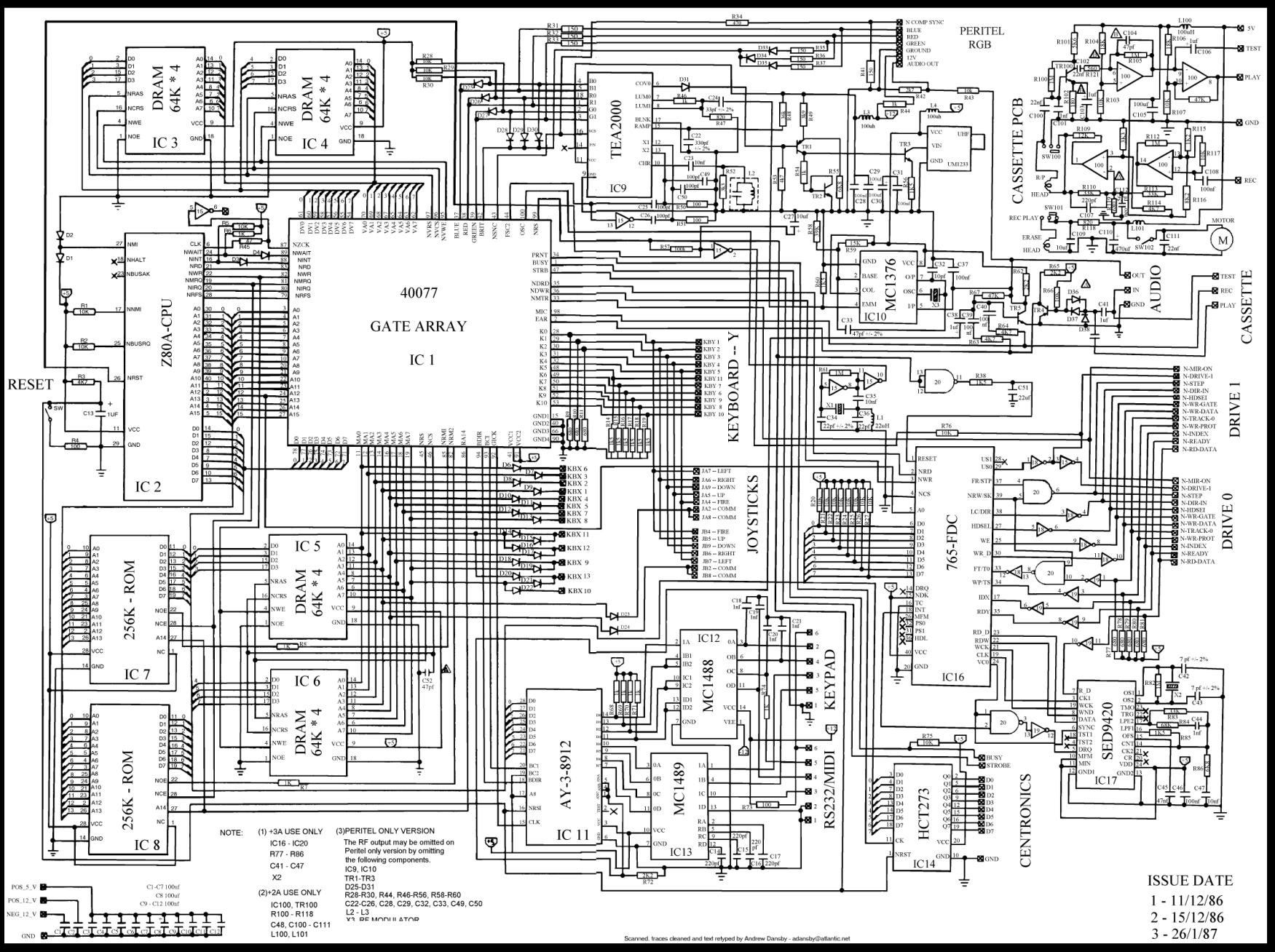


I. Har

Brain with



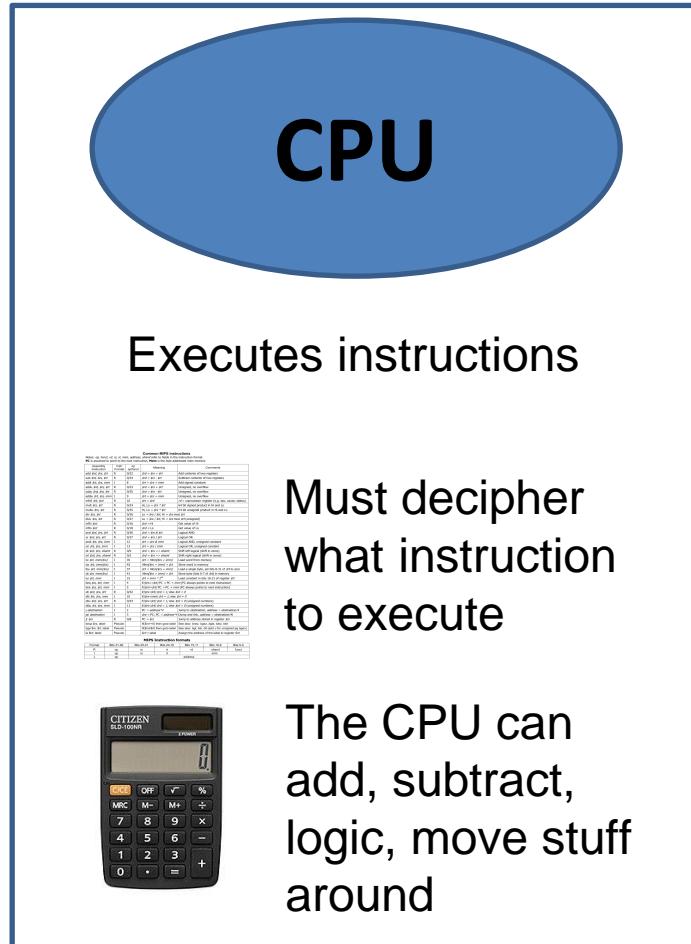
Exec



ISSUE DATE
1 - 11/12/86
2 - 15/12/86
3 - 26/1/87

I. Hardware

Brain with no short-term memory



Regis...

0000
0001

1

5

8

3

...

1

CPU uses

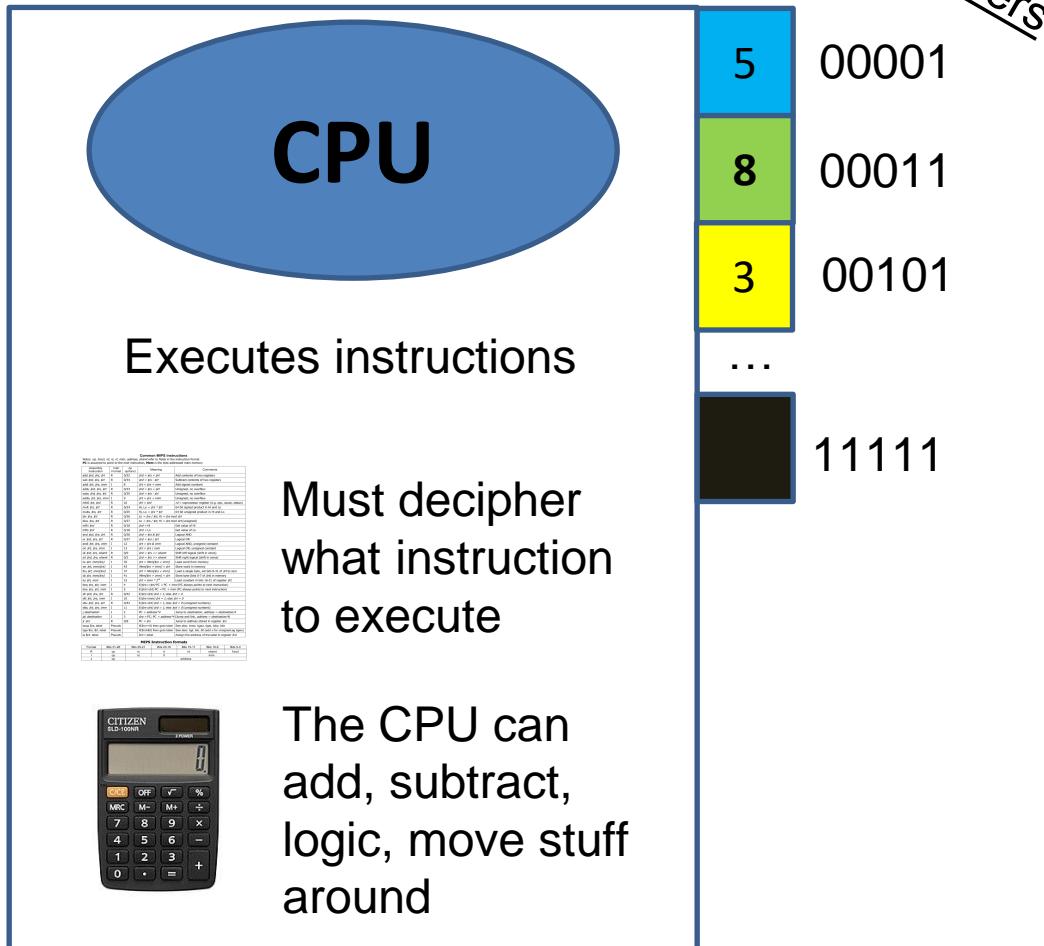
Electricity™

To decipher and execute instructions

*Everything is just electricity
on or electricity off*

I. Hardware

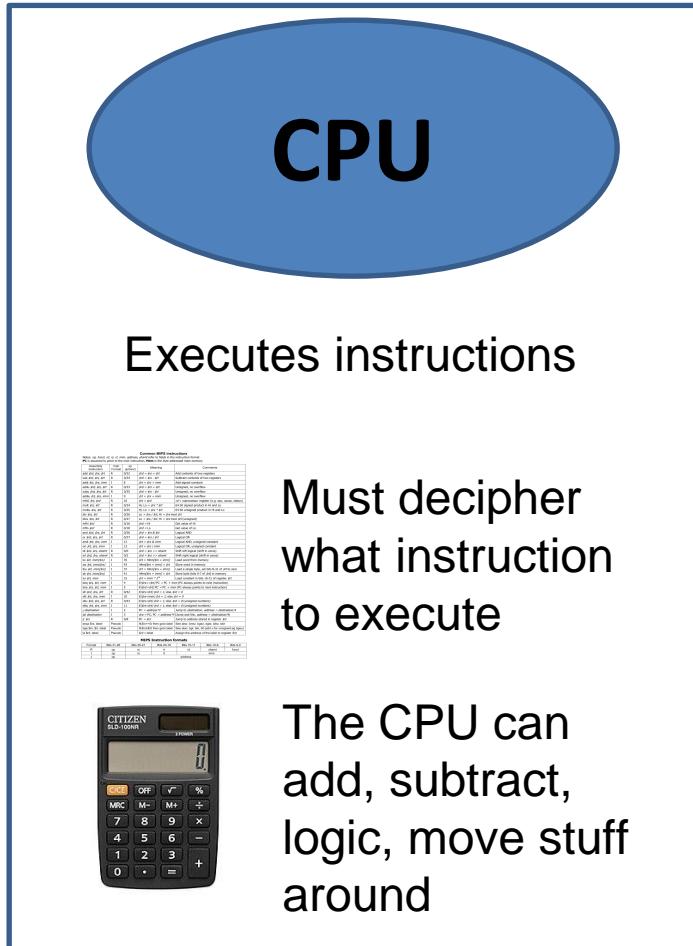
Brain with no short-term memory



32 bit vs 64 bit?

I. Hardware

Brain with no short-term memory

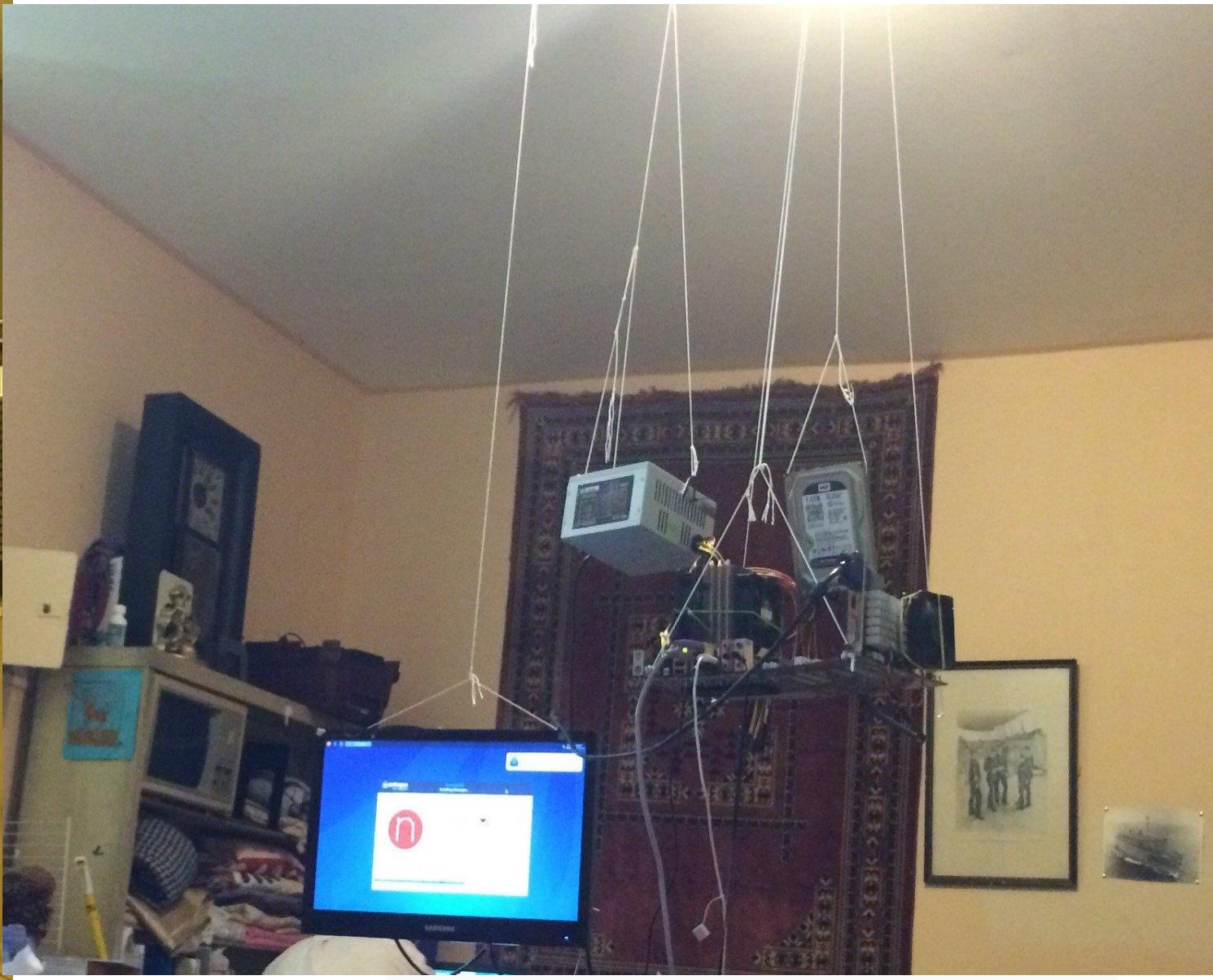
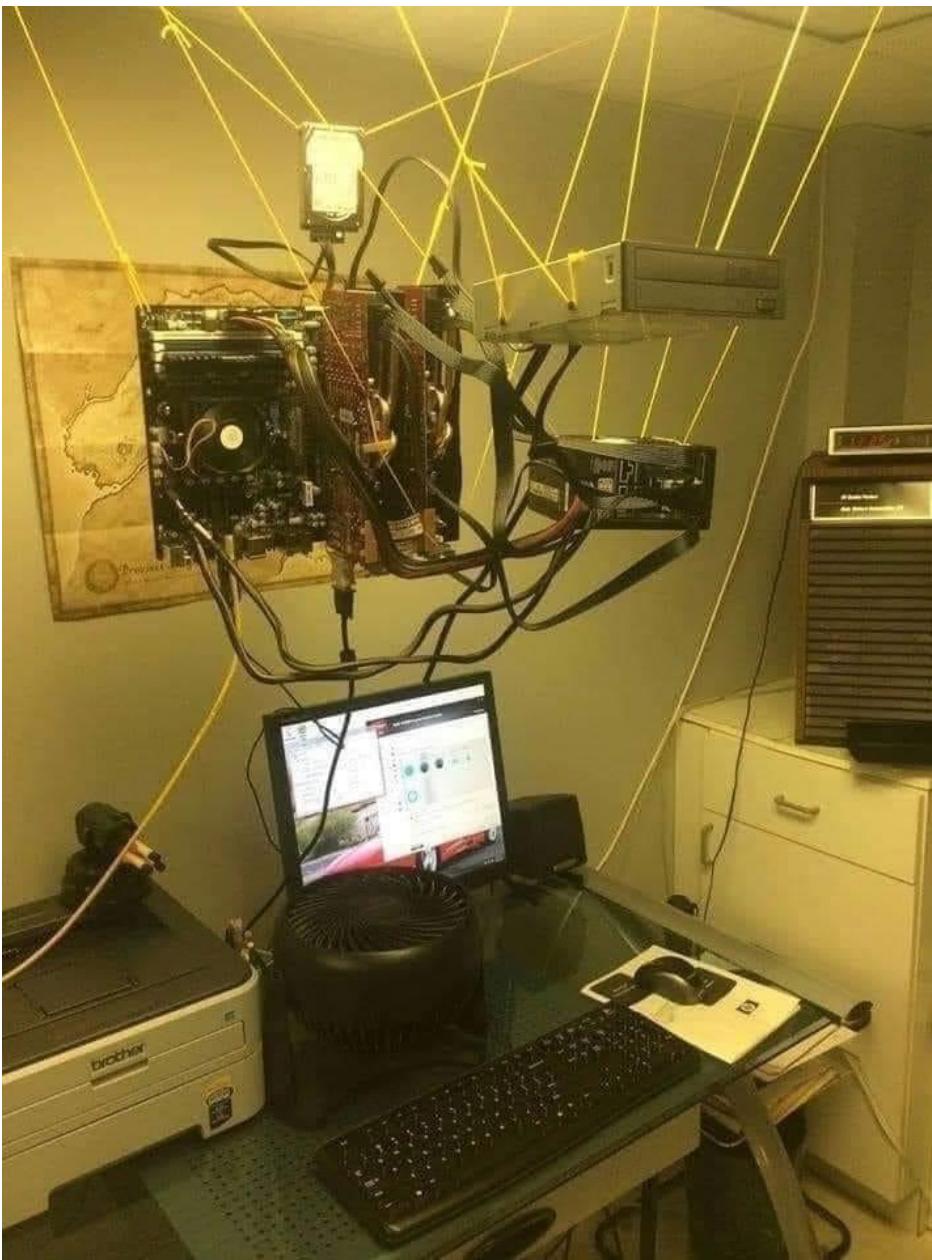


Registers

5 00001
8 00011
3 00101
...
11111

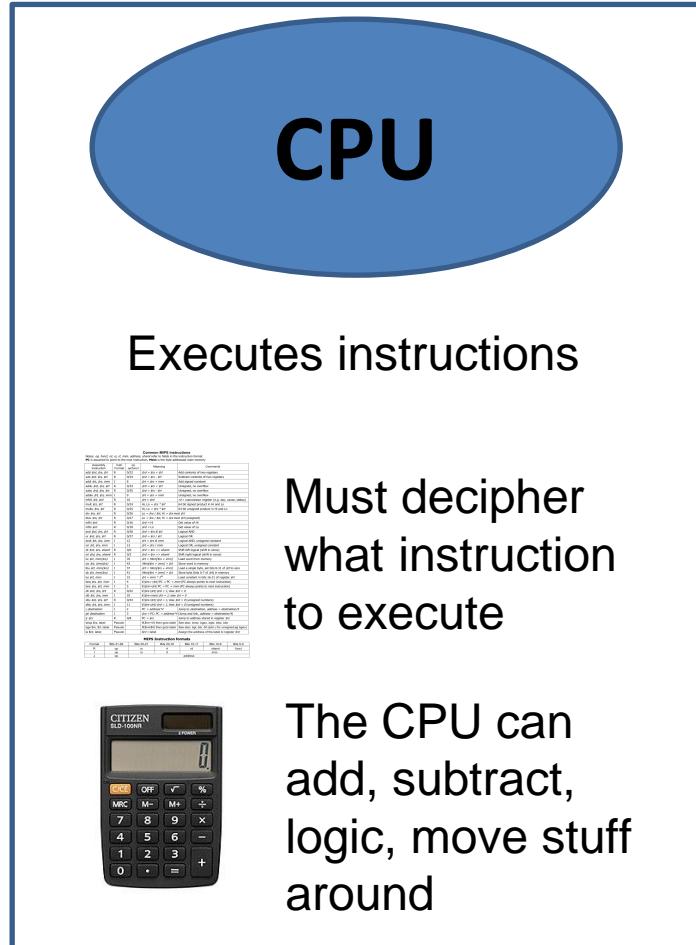
32 bit vs 64 bit?

Parameter	32-bit processors	64-bit processors
Addressable space	It has 4 GB addressable space	64-bit processors have 16 exabytes addressable space
Application support	64-bit applications and programs won't work	32-bit applications and programs will work
OS support	Need a 32-bit operating system.	It can run on 32 and the 64-bit operating system.
Support for multi-tasking	Not an ideal option for stress testing and multi-tasking.	Works best for performing multi-tasking and stress testing.
OS and CPU requirement	32-bit operating systems and applications require 32-bit CPUs	64-bit OS demands 64-bit CPU, and 64-bit applications require 64-bit OS and CPU.
System available	Support Windows 7, 8 Vista, XP, and, Linux.	Windows XP Professional, Windows Vista, Windows 7, Windows 8, Windows 10, Linux, and Mac OS X.



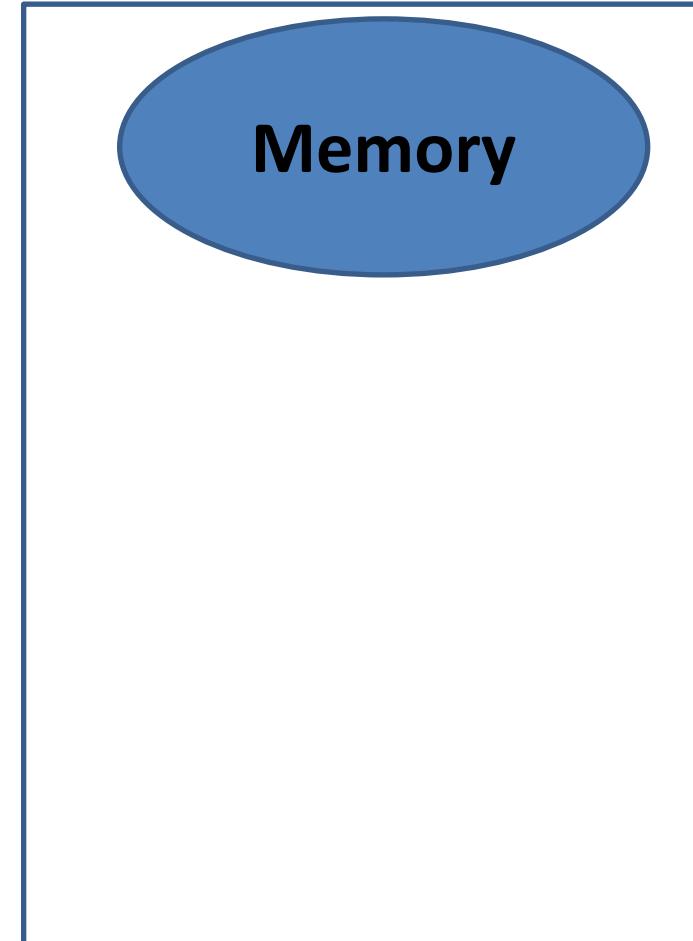
I. Hardware

Brain with no short-term memory

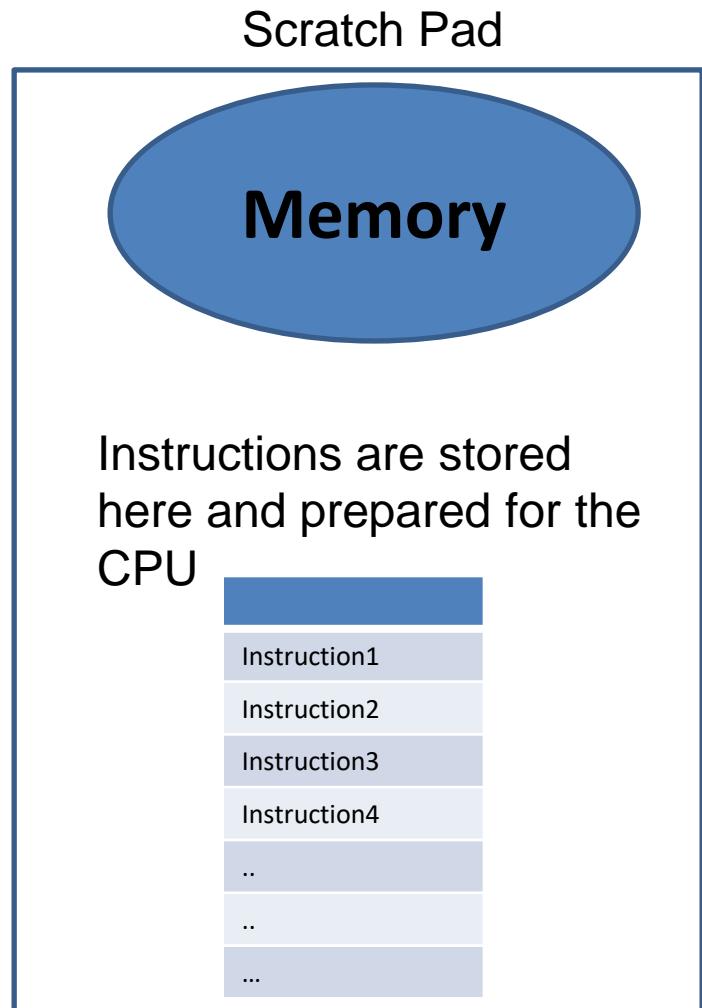


Registers

Scratch Pad

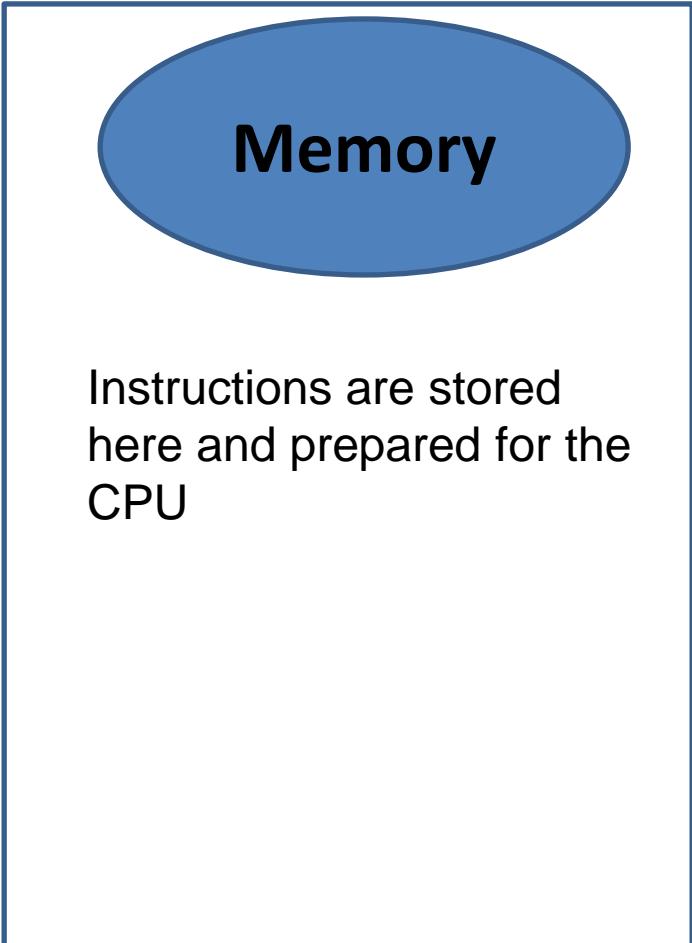


I. Hardware



I. Hardware

Scratch Pad



Memory

Instructions are stored here and prepared for the CPU

When computer programs are executed, their instructions will eventually get stored in memory

I. Hardware

Scratch Pad

The diagram shows a blue oval containing the word "Memory". This oval is positioned within a larger white rectangular frame. The word "Memory" is written in a bold, black, sans-serif font. The entire diagram is set against a white background.

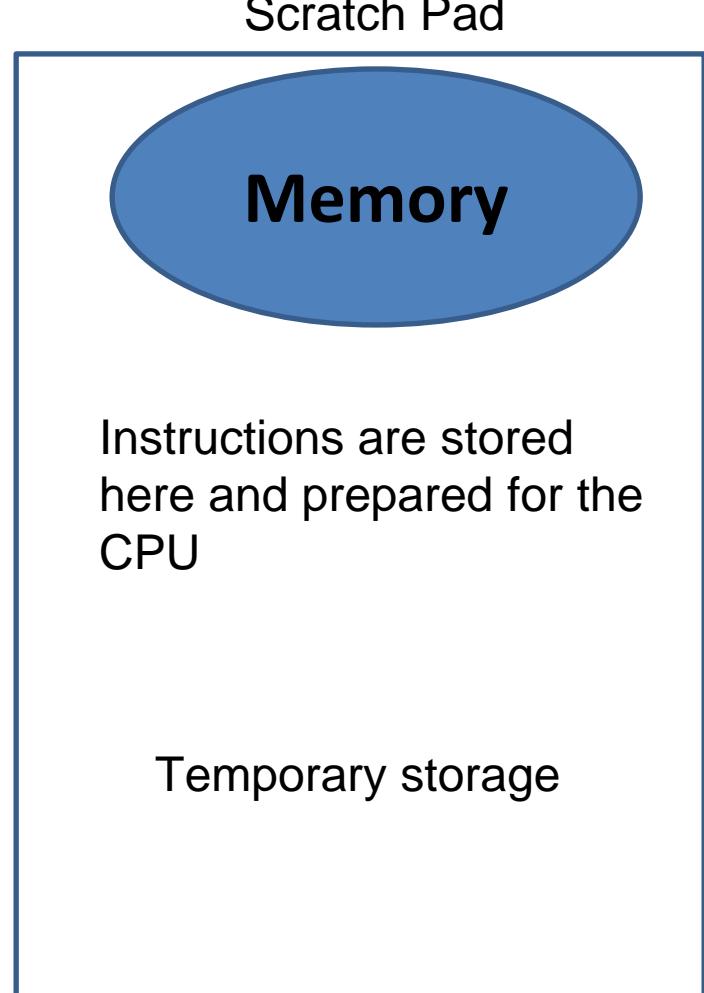
Memory

Instructions are stored here and prepared for the CPU

When computer programs are executed, their instructions will eventually get stored in memory

Permanently?!?!

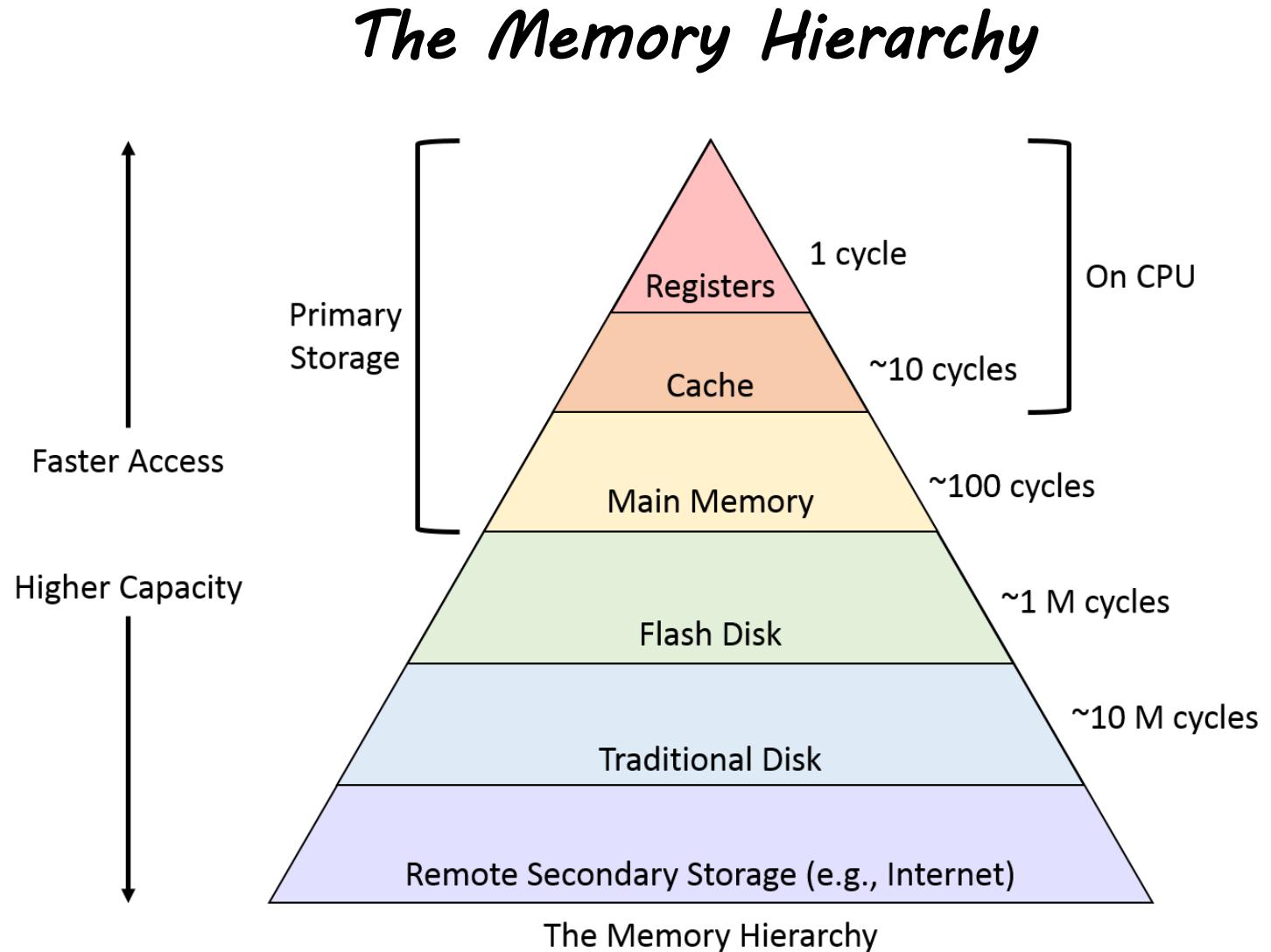
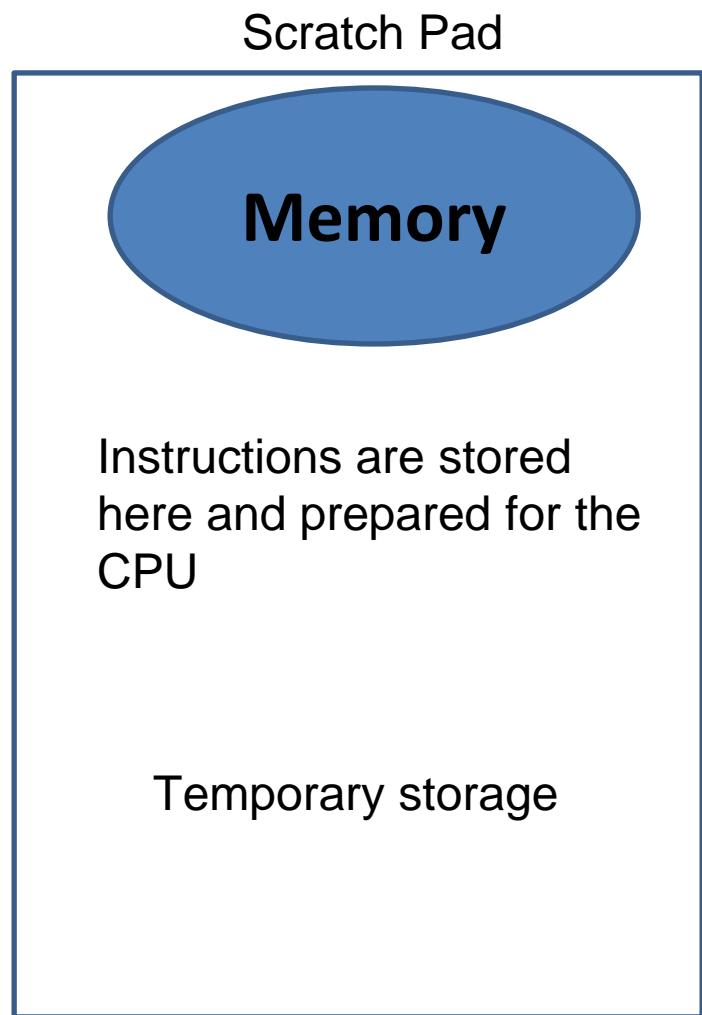
I. Hardware



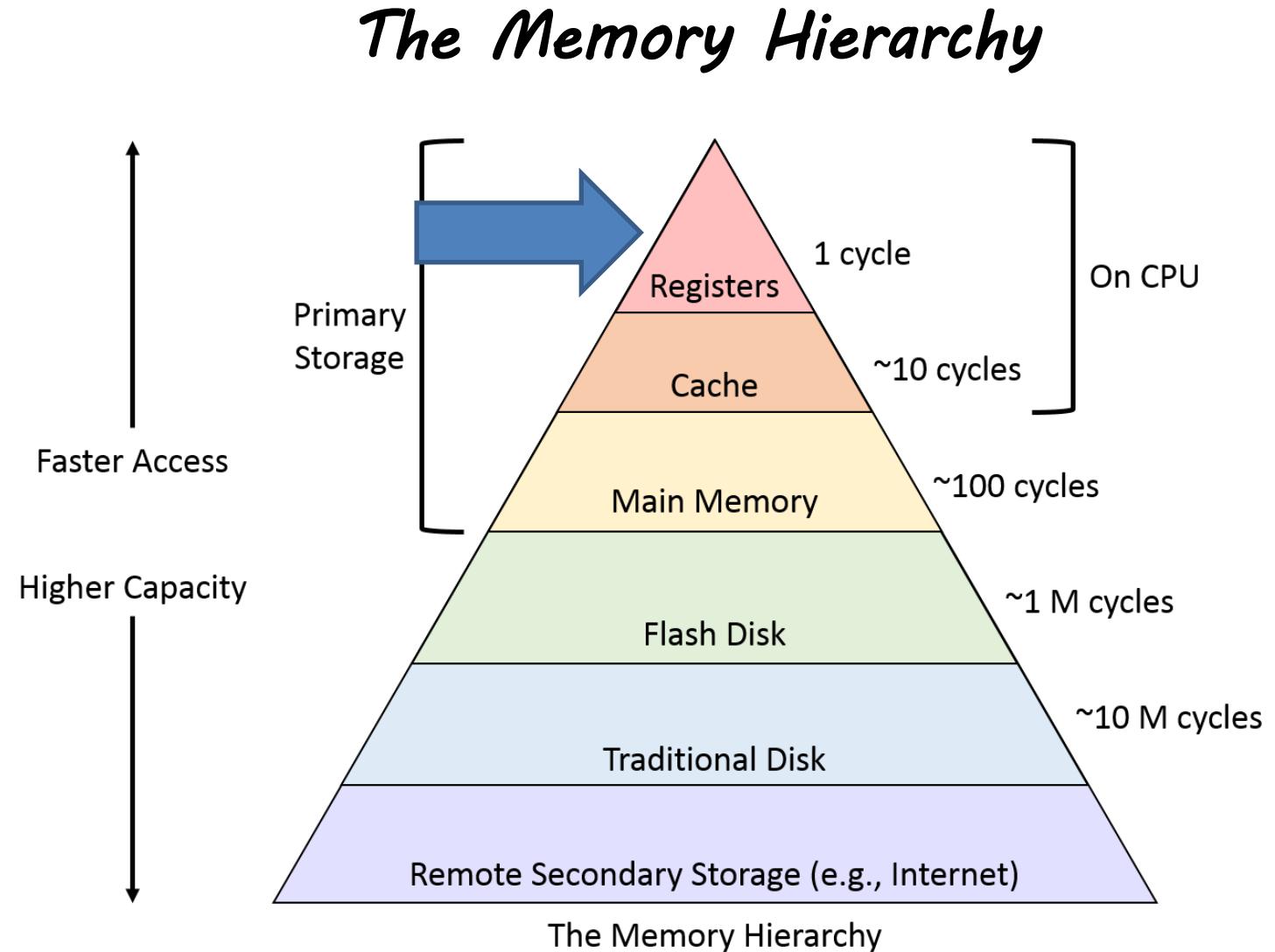
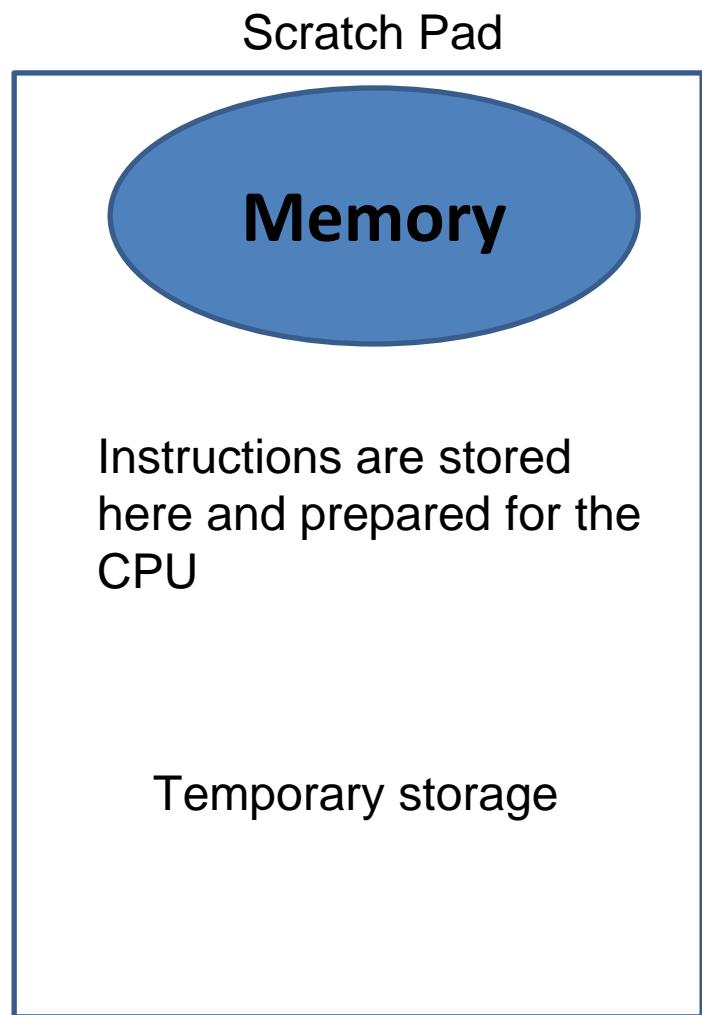
When computer programs are executed, their instructions will eventually get stored in memory

Main memory is **volatile**

I. Hardware



I. Hardware

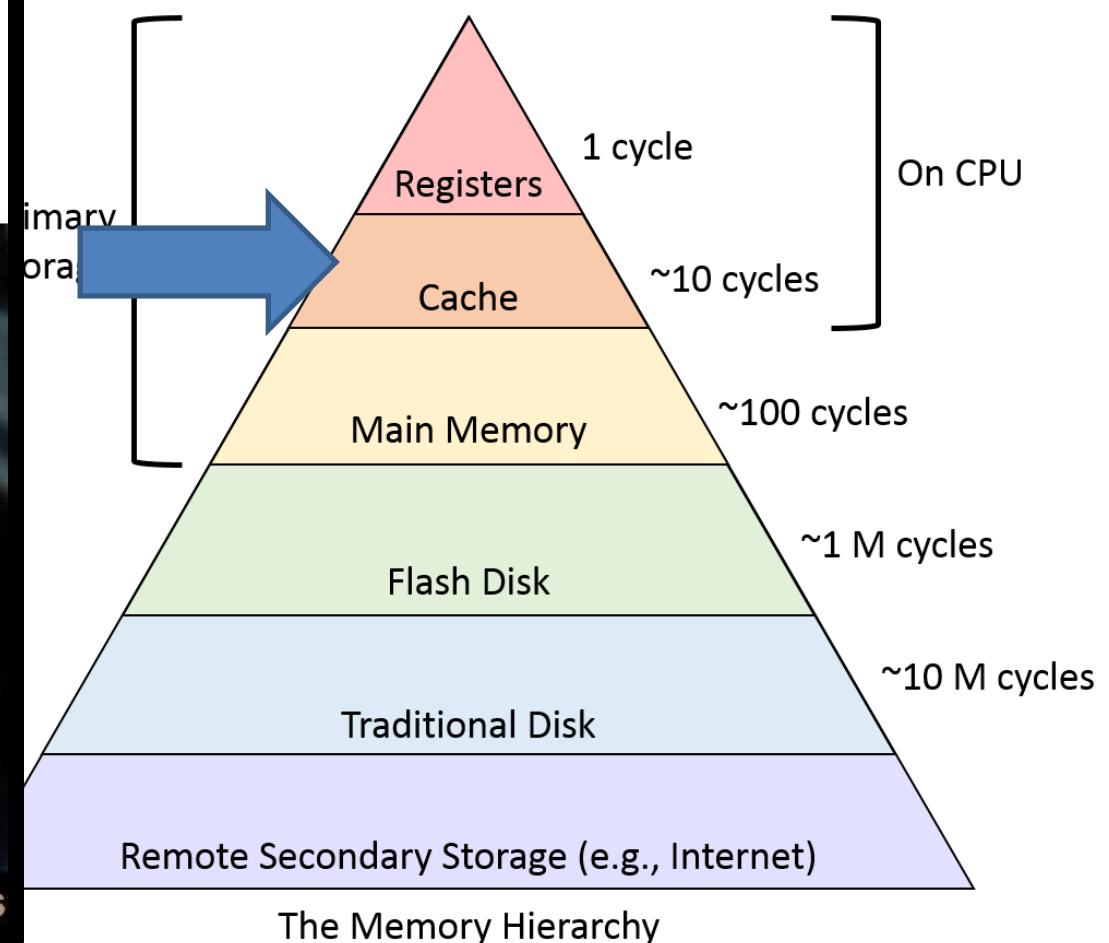


I. Hardware

My CPU when the L1 cache misses

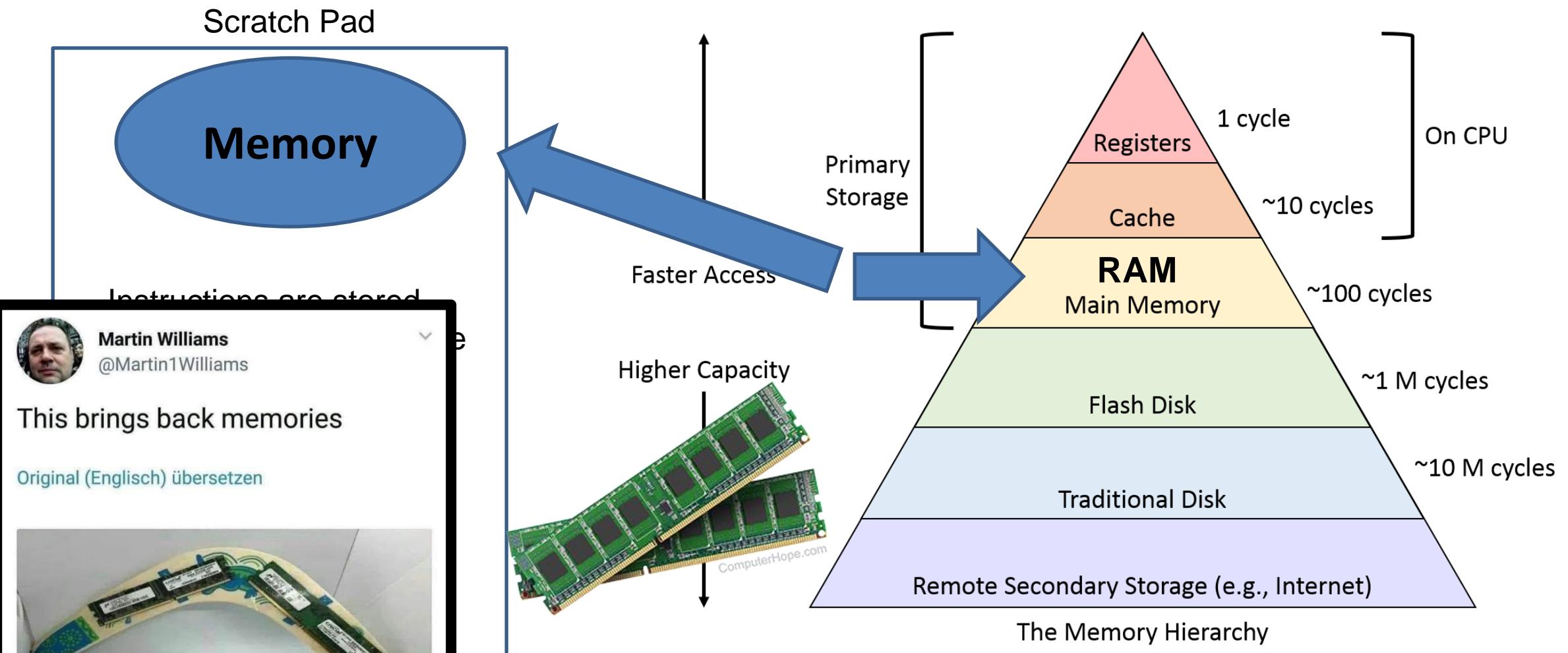


The Memory Hierarchy

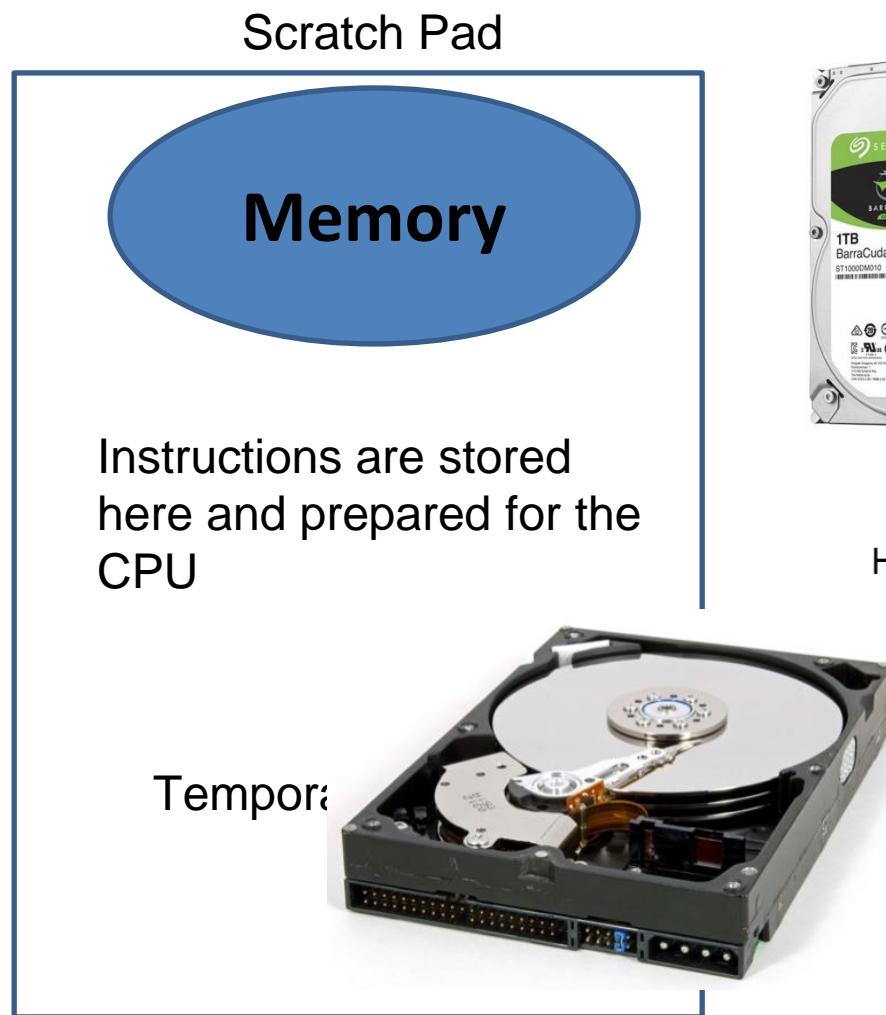


I. Hardware

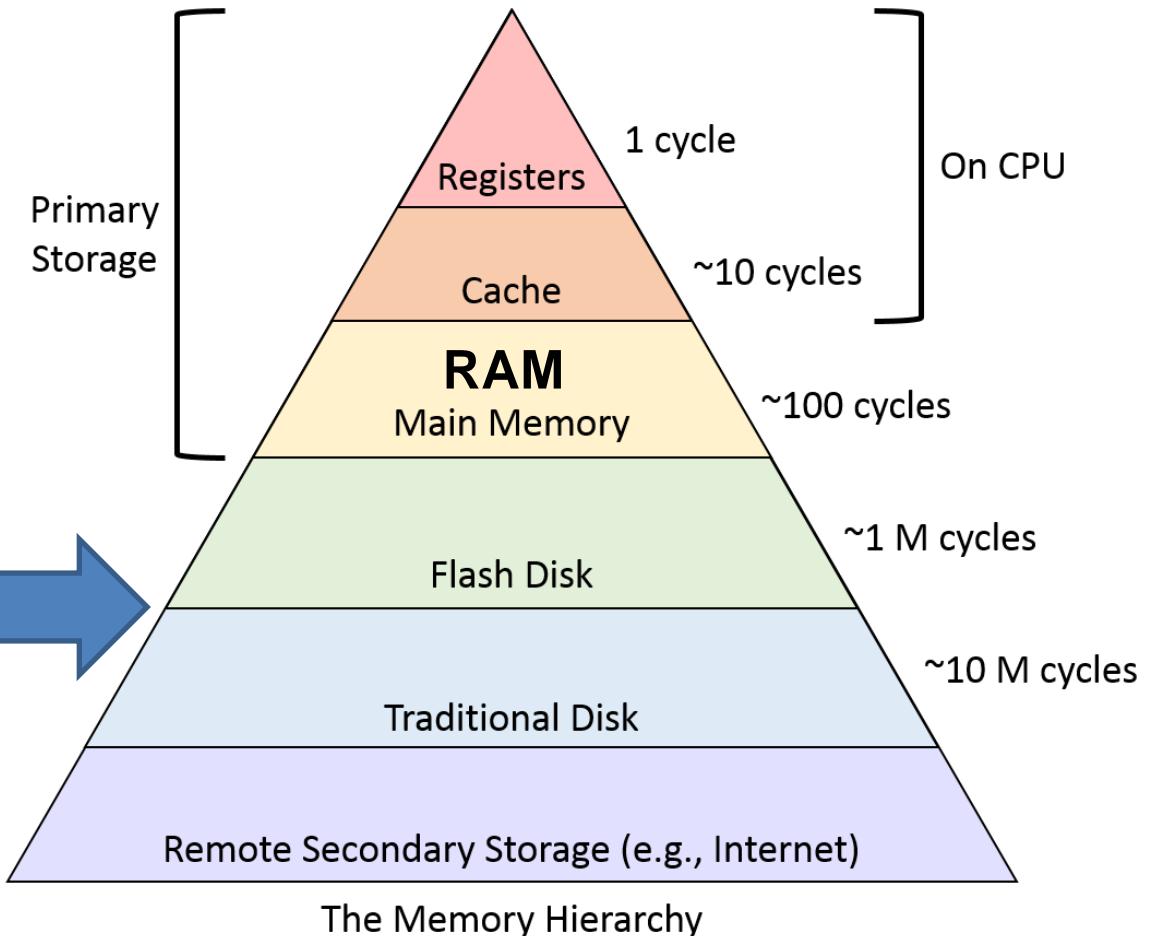
The Memory Hierarchy



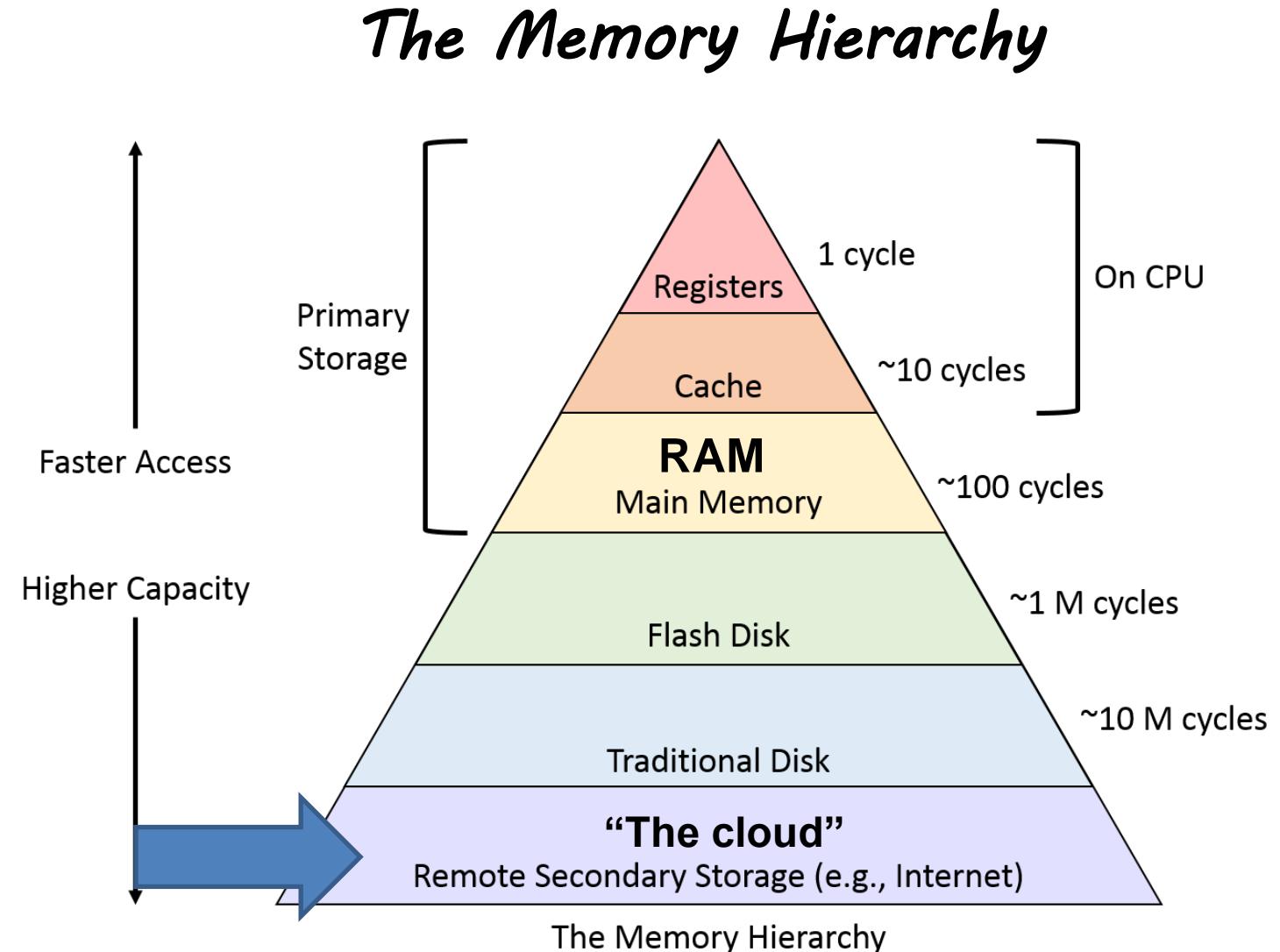
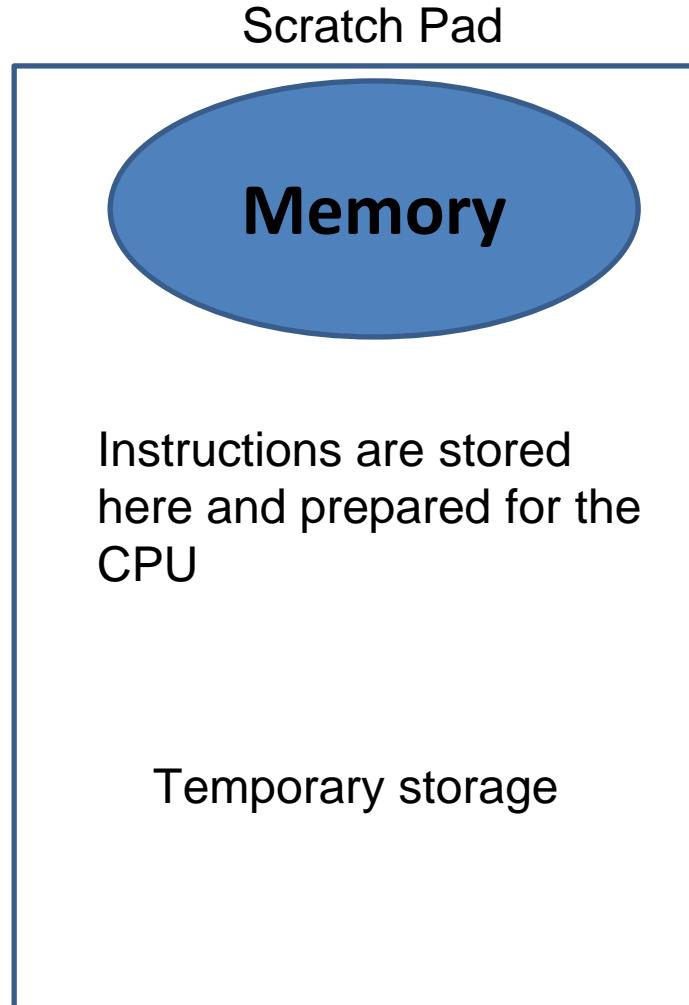
I. Hardware



The Memory Hierarchy



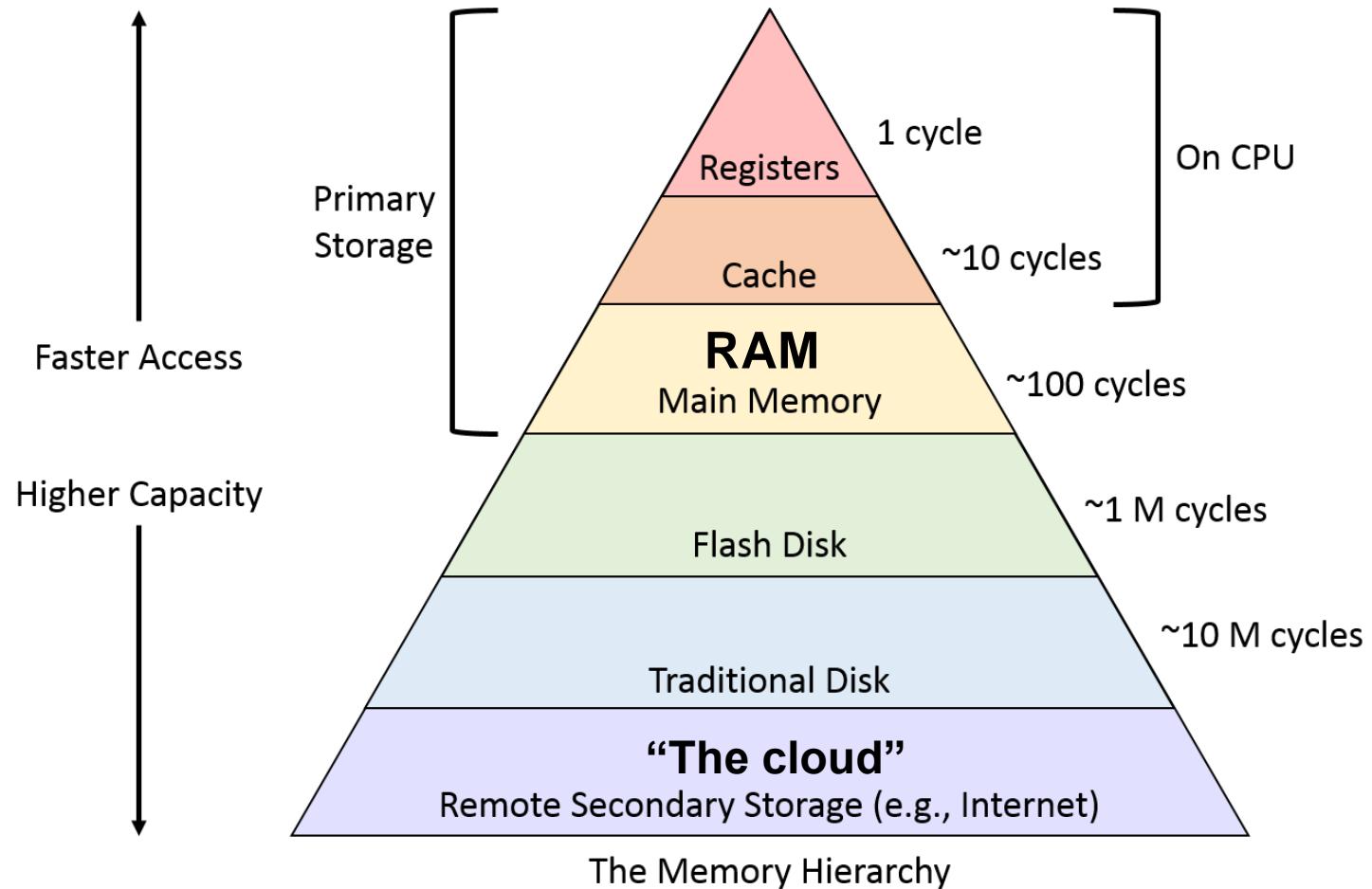
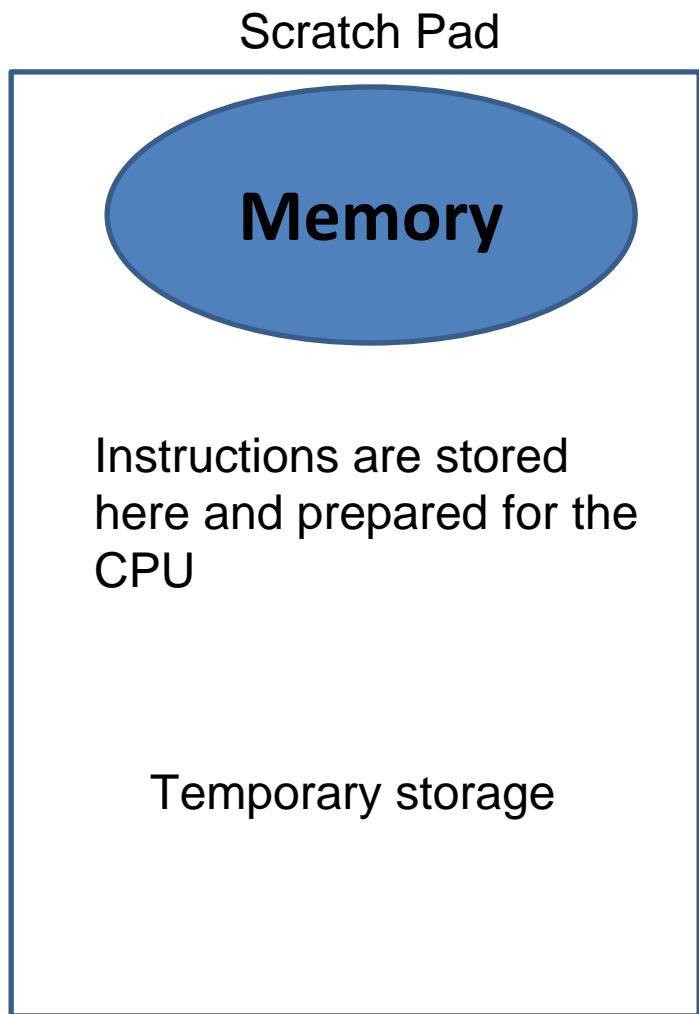
I. Hardware



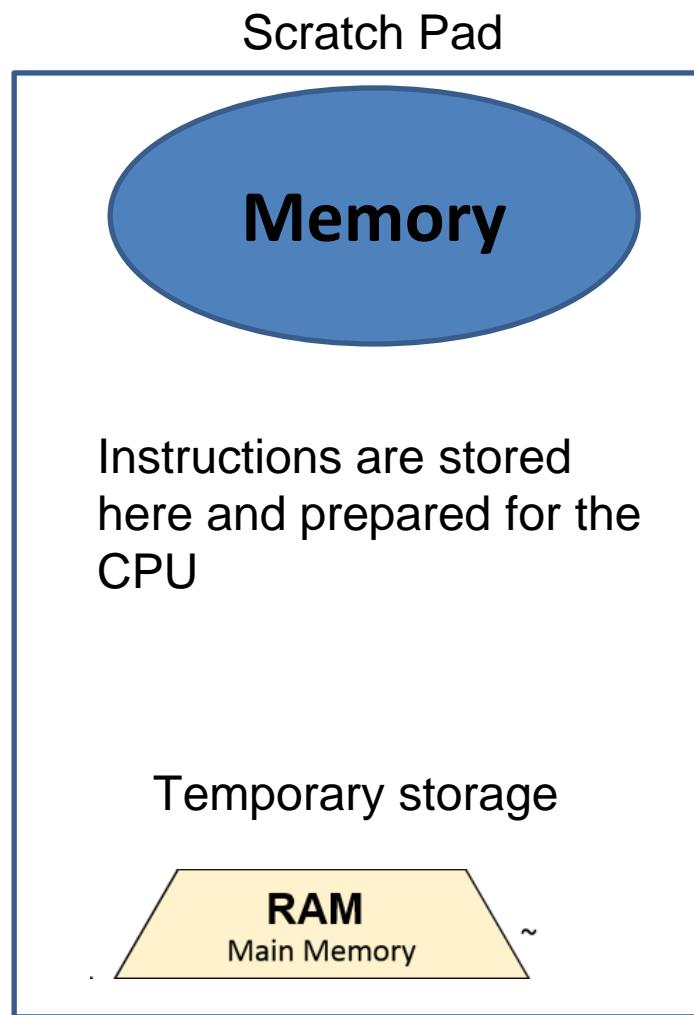
I. Hardware



The Memory Hierarchy

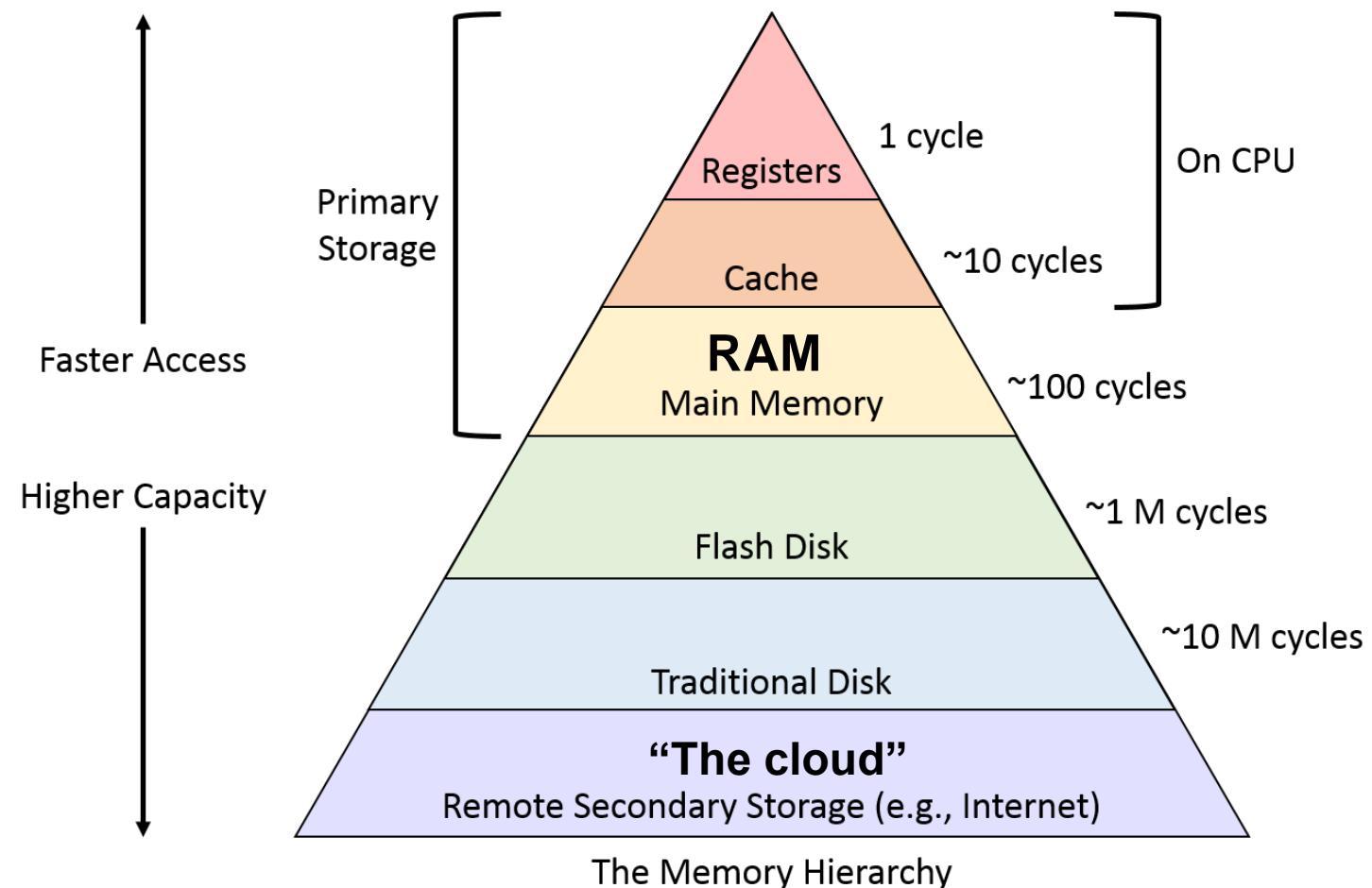


I. Hardware

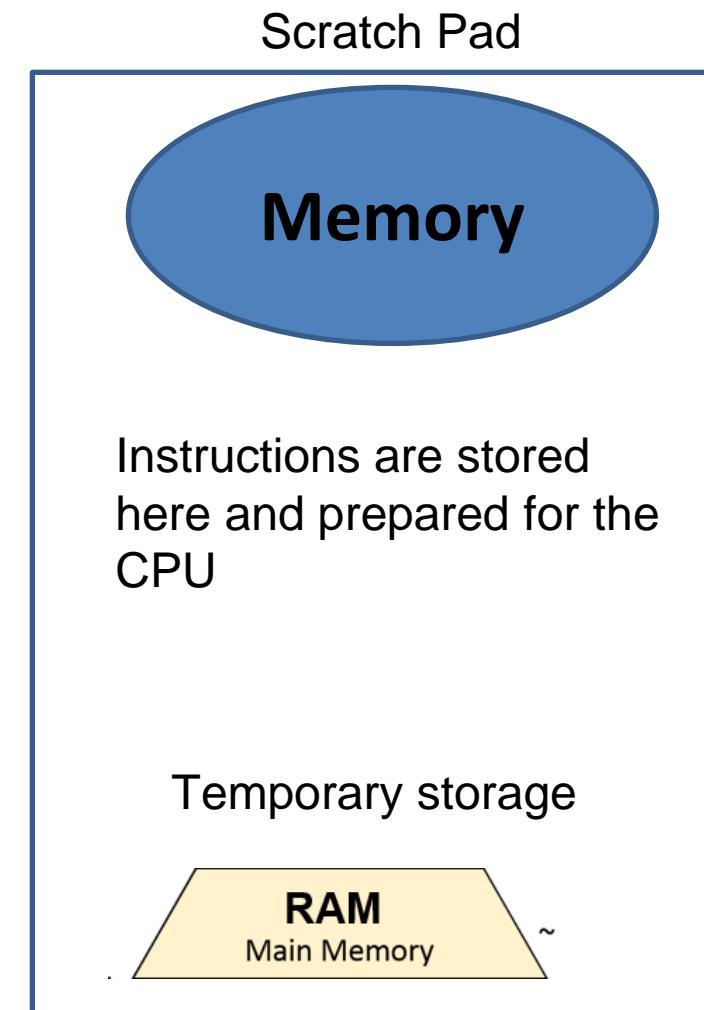
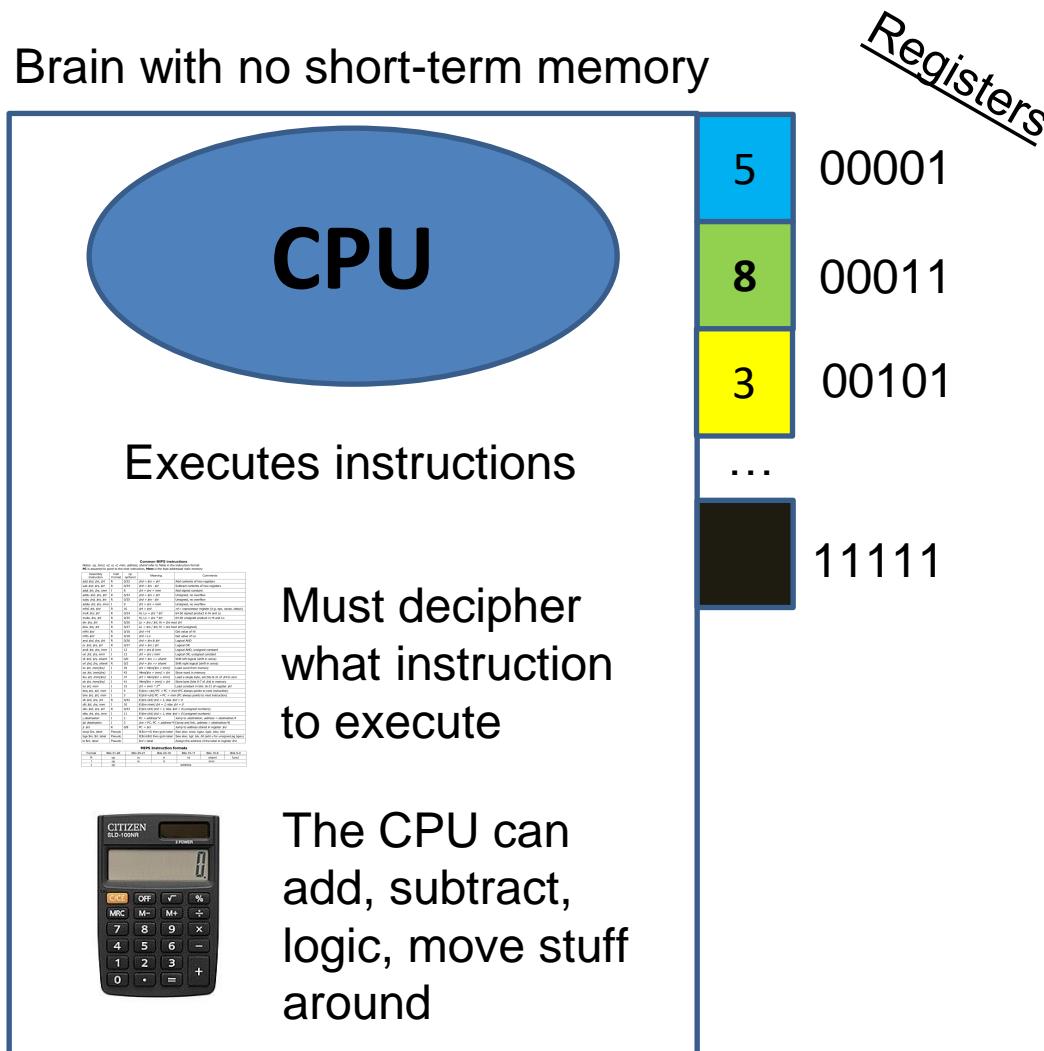


*Why not use
memory/registers
for everything??*

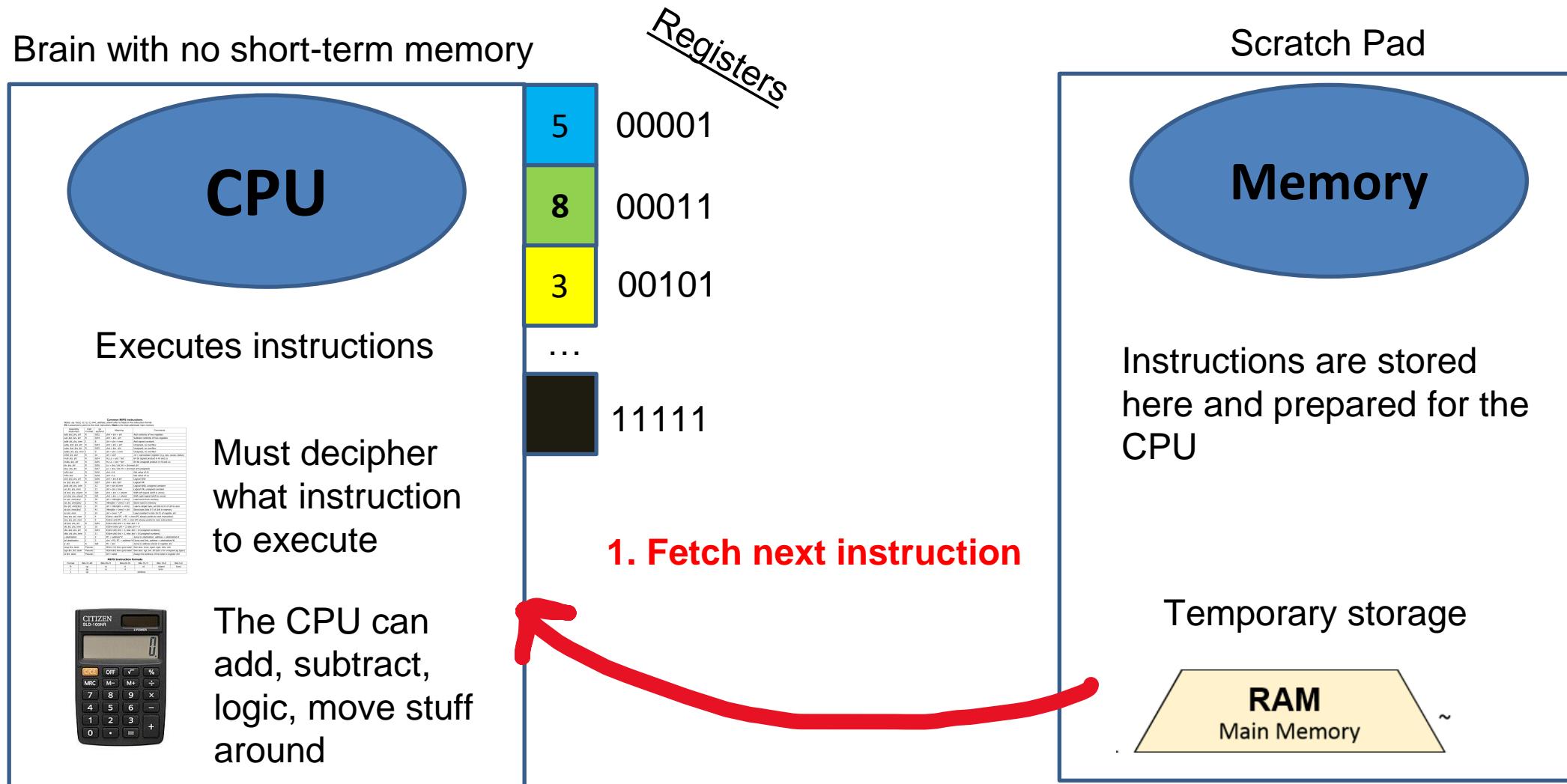
The Memory Hierarchy



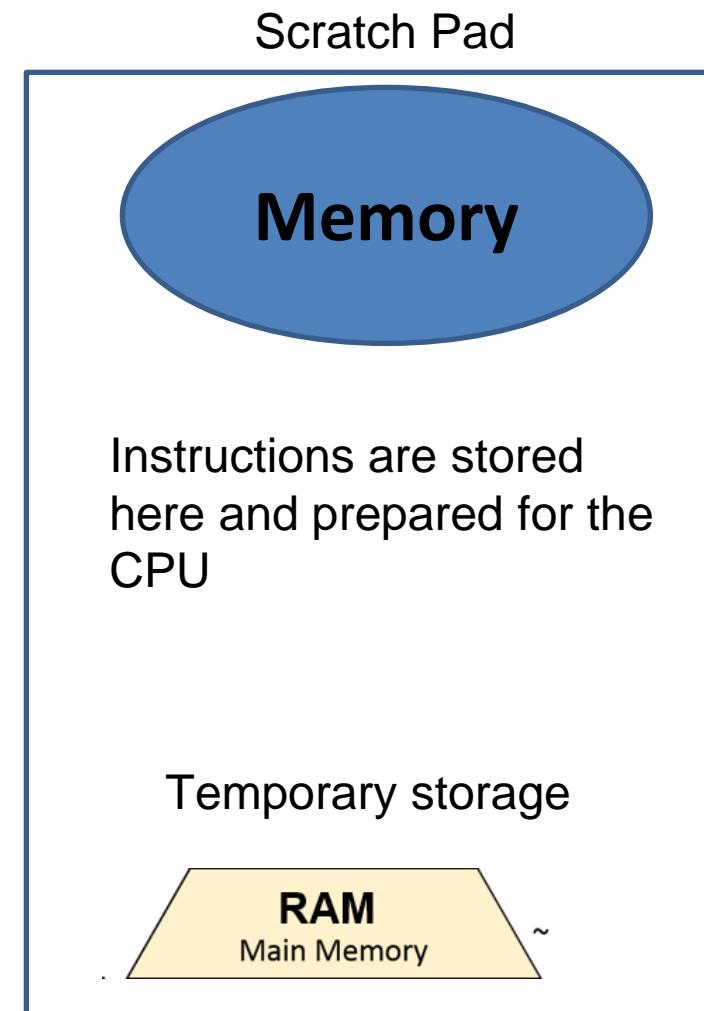
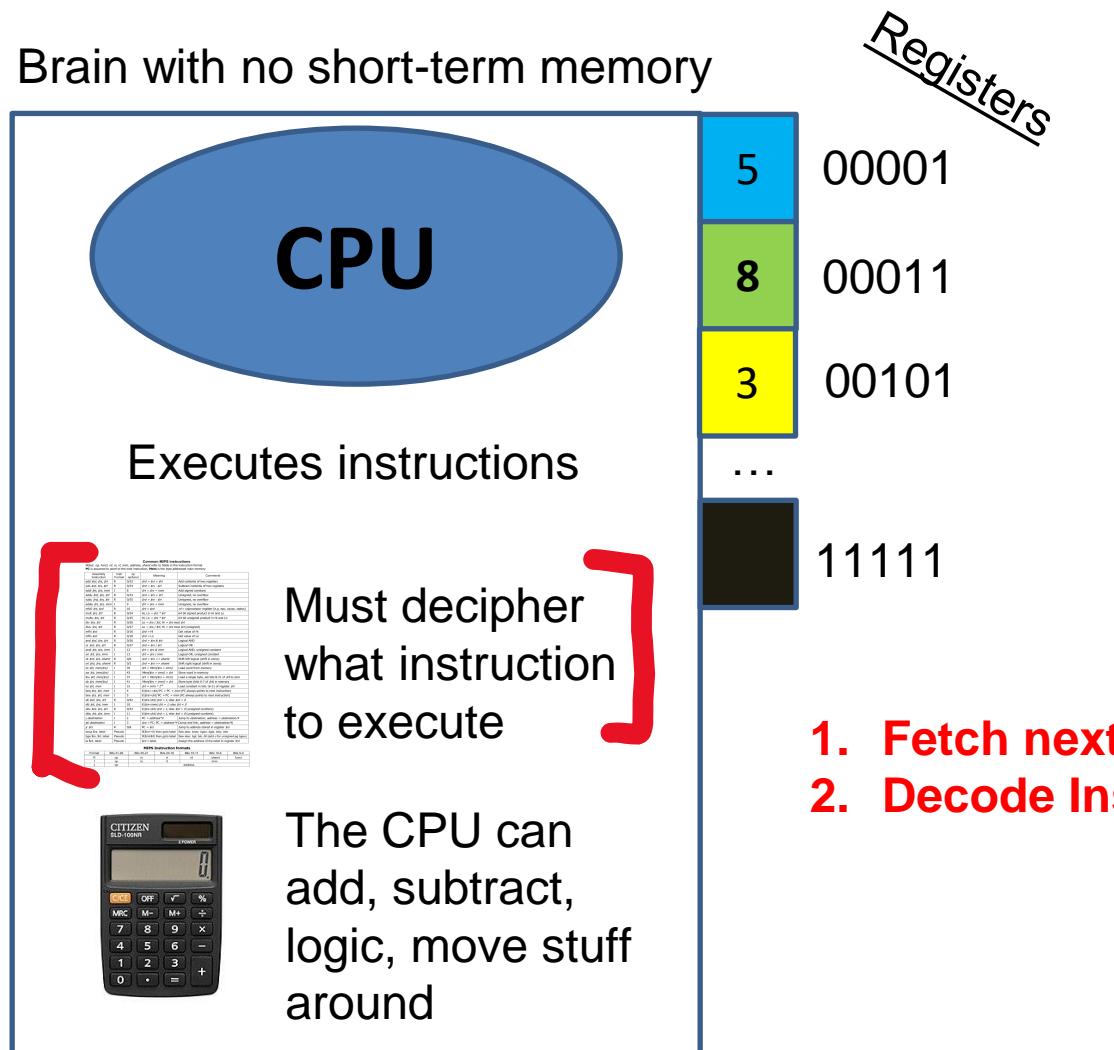
I. Hardware



I. Hardware

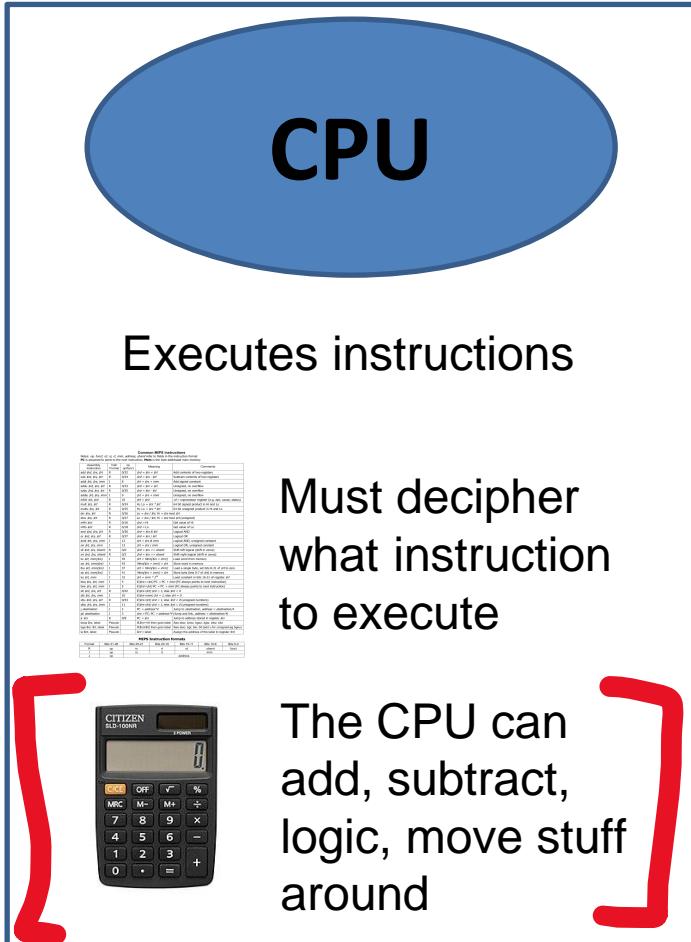


I. Hardware



I. Hardware

Brain with no short-term memory

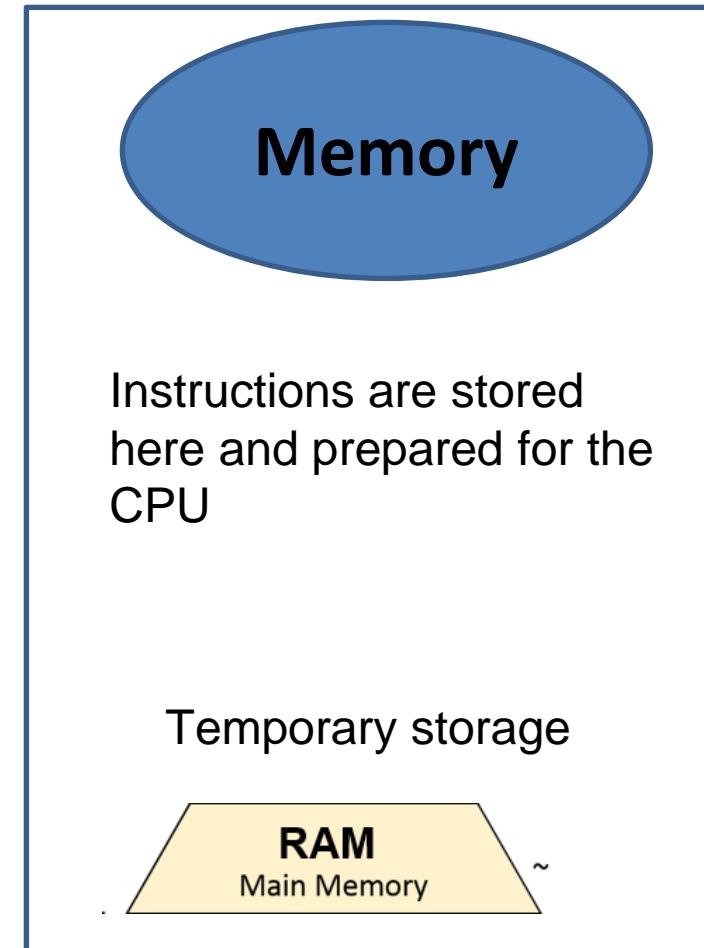


Registers

00001
00011
00101
...
11111

1. Fetch next instruction
2. Decode Instruction
3. Execute Instruction

Scratch Pad

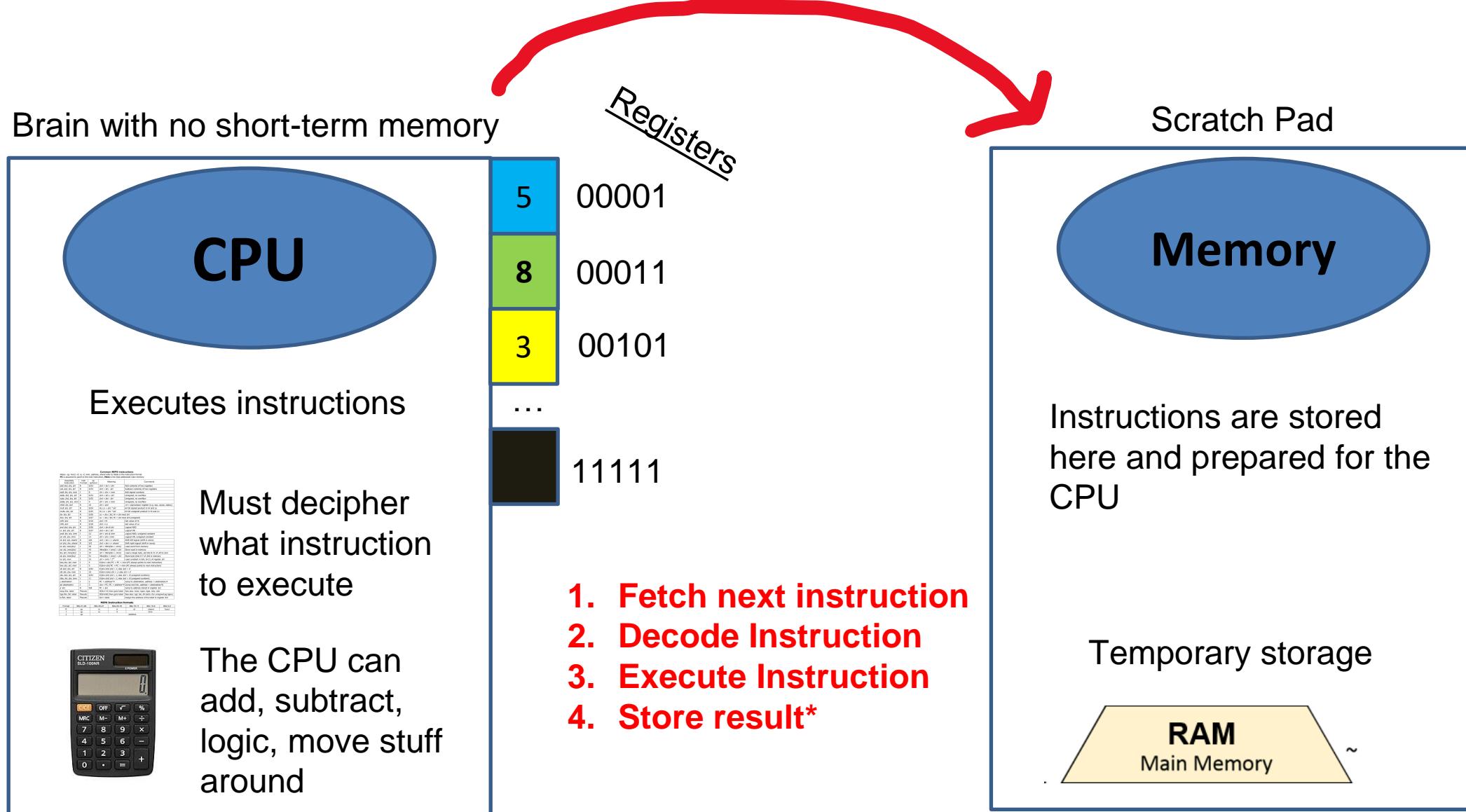


Instructions are stored here and prepared for the CPU

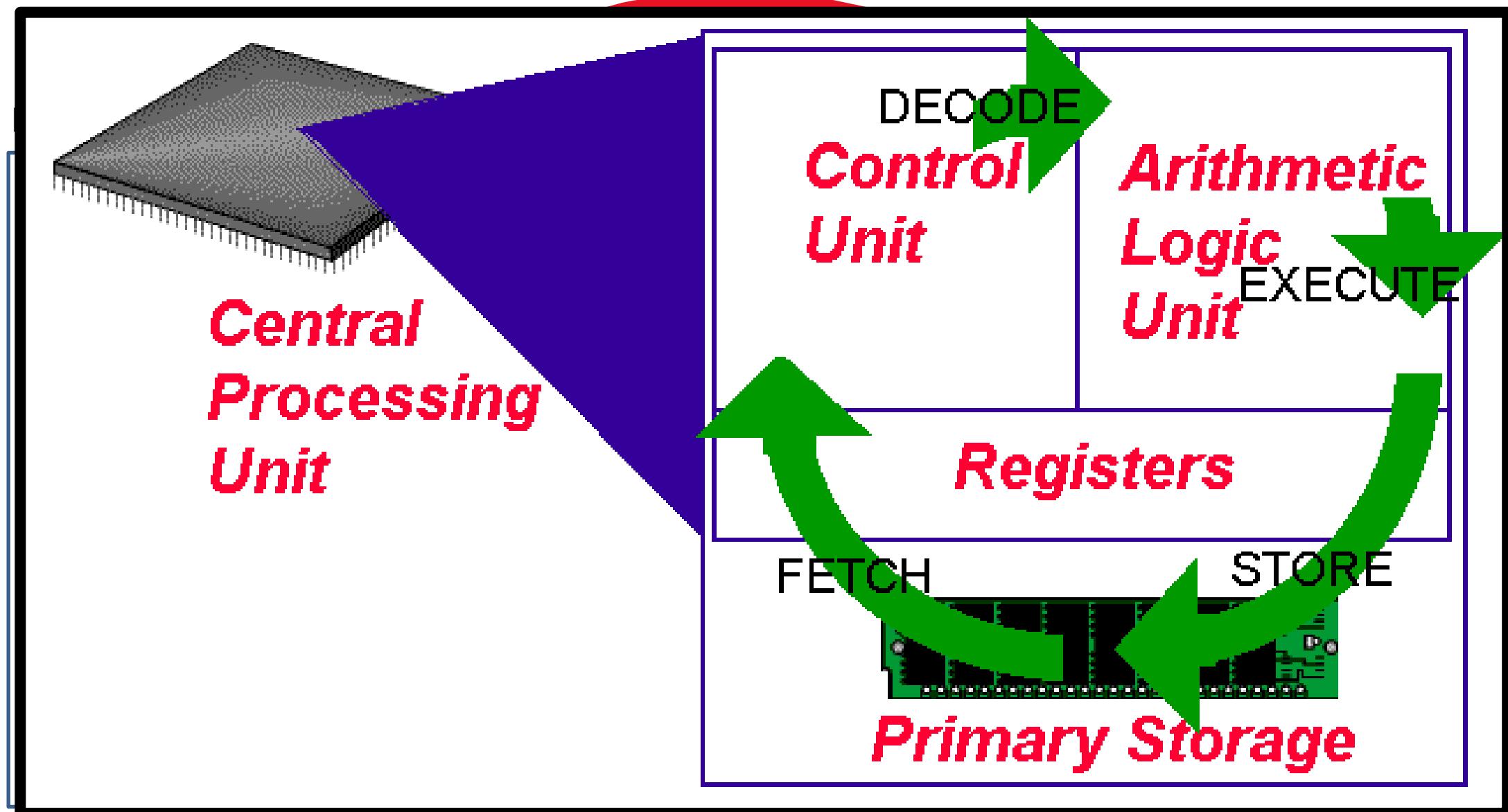
Temporary storage

RAM
Main Memory

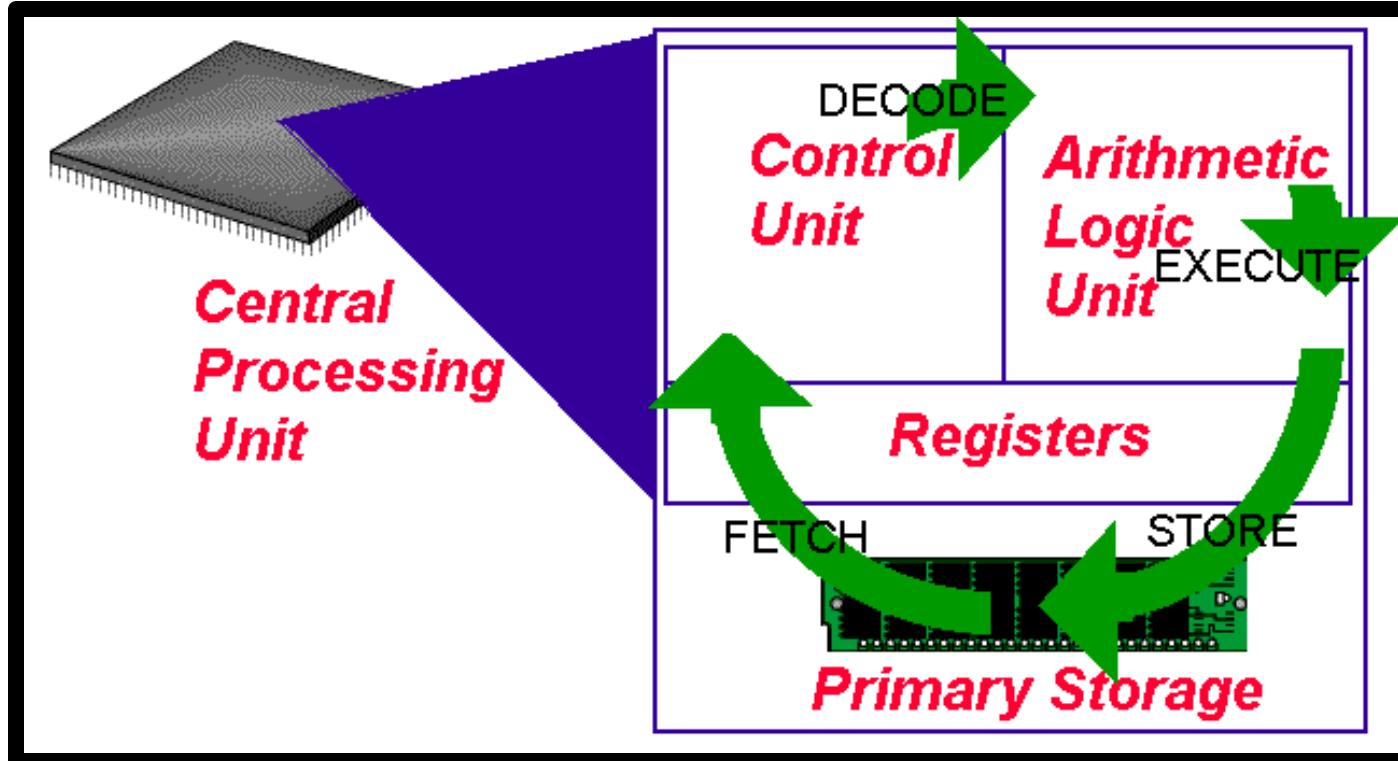
I. Hardware



I. Hardware

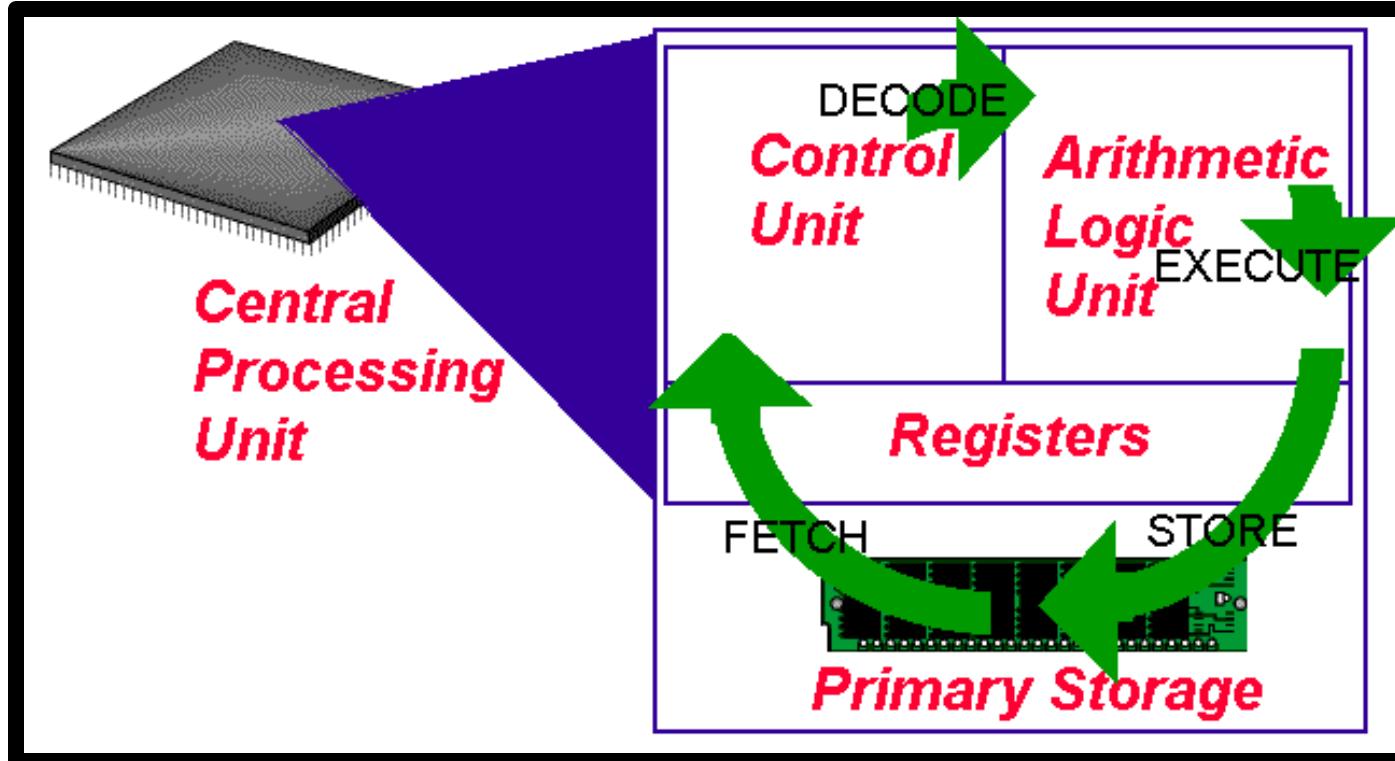


I. Hardware



This process happens really fast

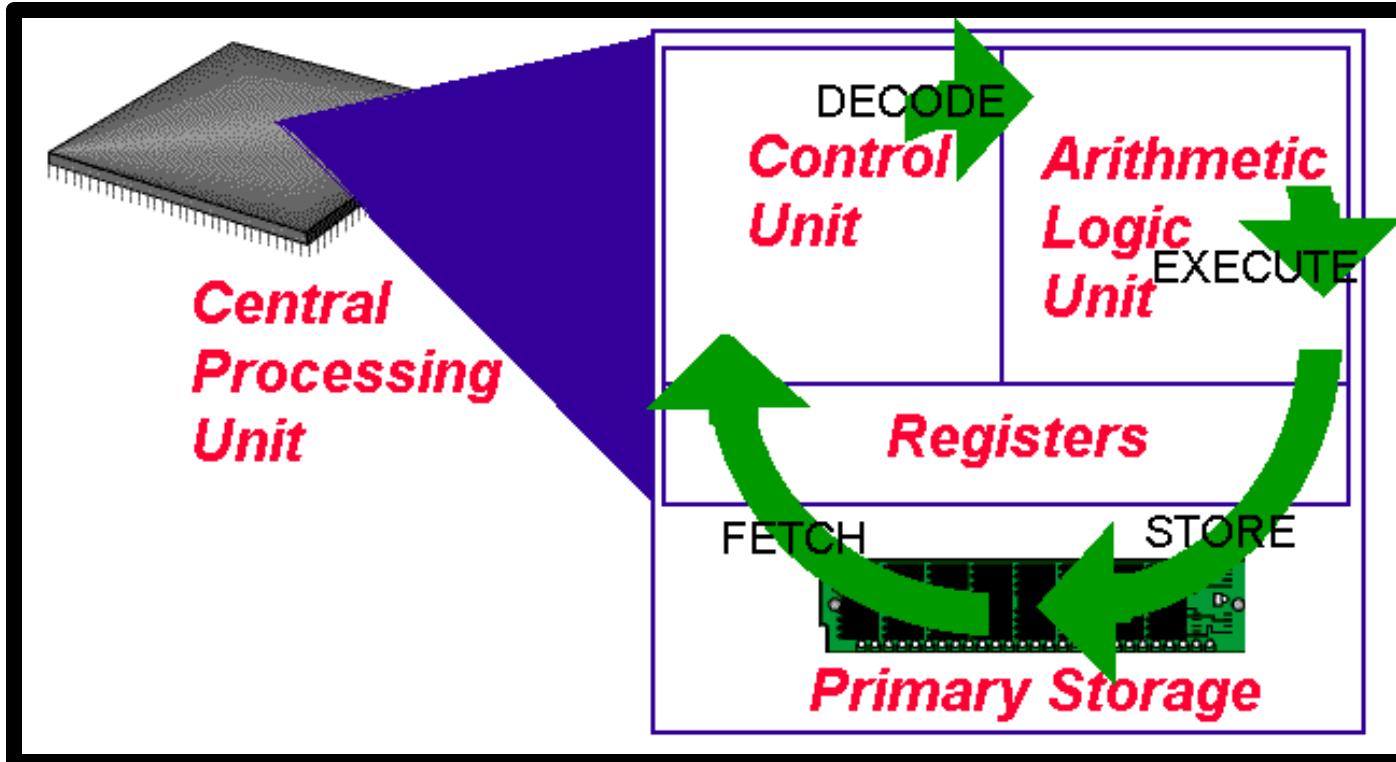
I. Hardware



This process happens really fast

... like *really* fast

I. Hardware

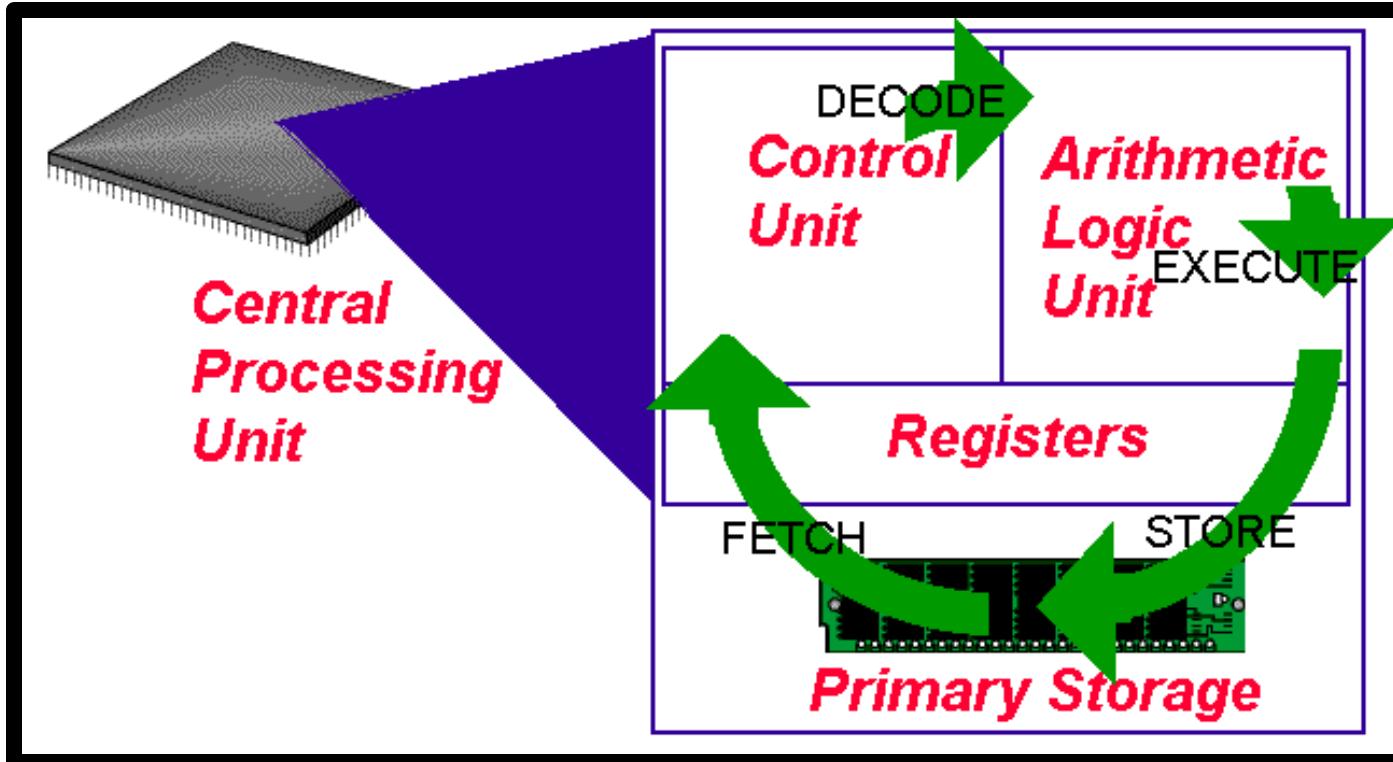


This process happens really fast

... like *really* fast

Computers can execute one or more instructions per clock cycle*

I. Hardware



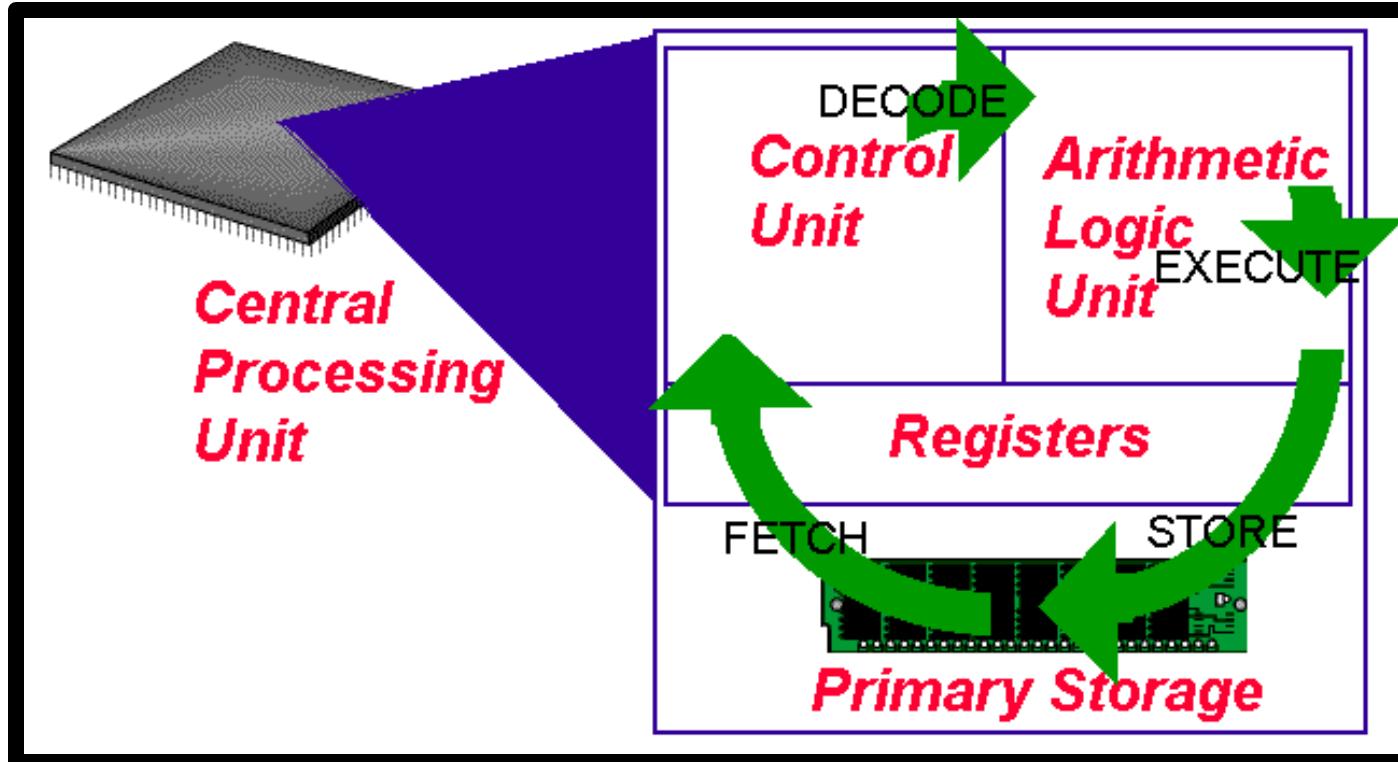
This process happens really fast

... like *really* fast

Computers can execute one or more instructions per clock cycle*

4GHz CPU speed = 4,000,000,000 clock cycles per second

I. Hardware



This process happens really fast

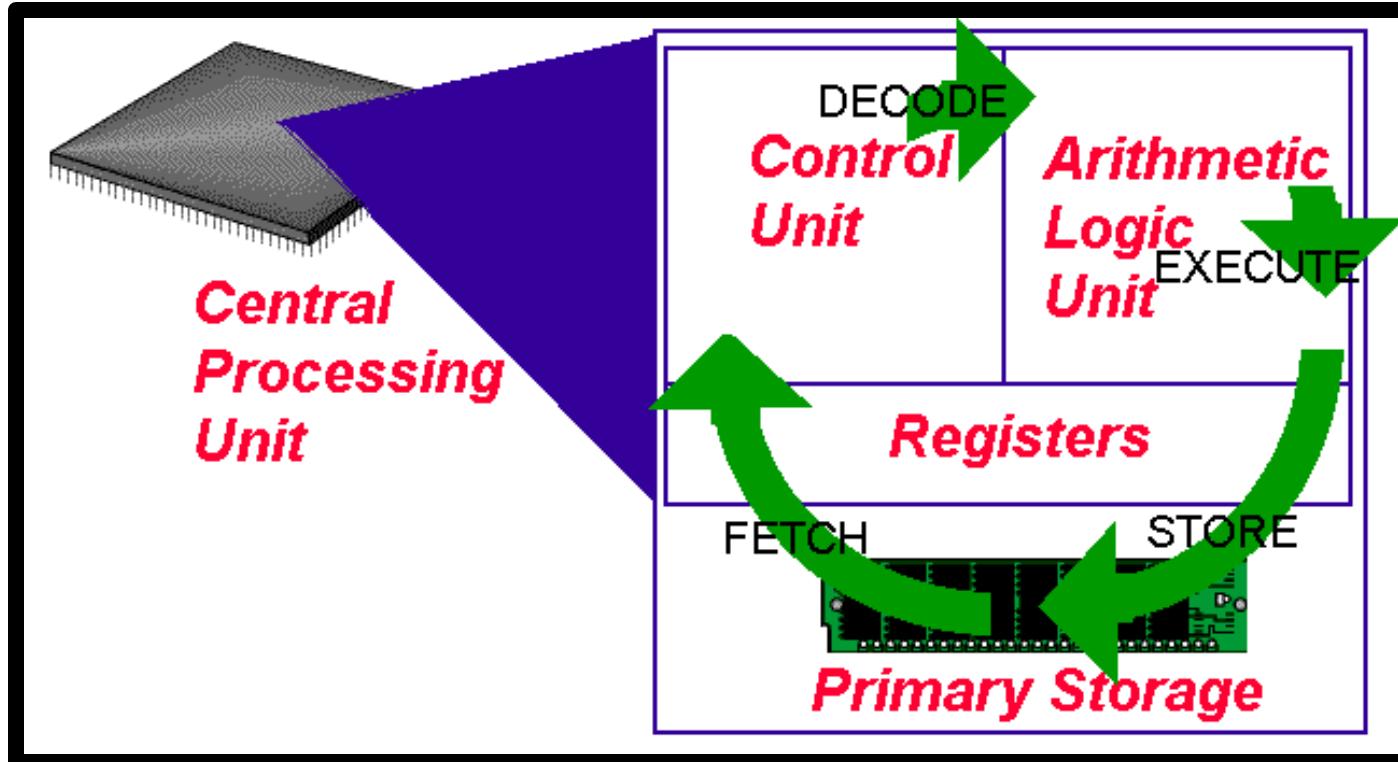
... like *really* fast

Computers can execute one or more instructions per clock cycle*



4GHz CPU speed = 4,000,000,000 clock cycles per second

I. Hardware



This process happens really fast

... like *really* fast

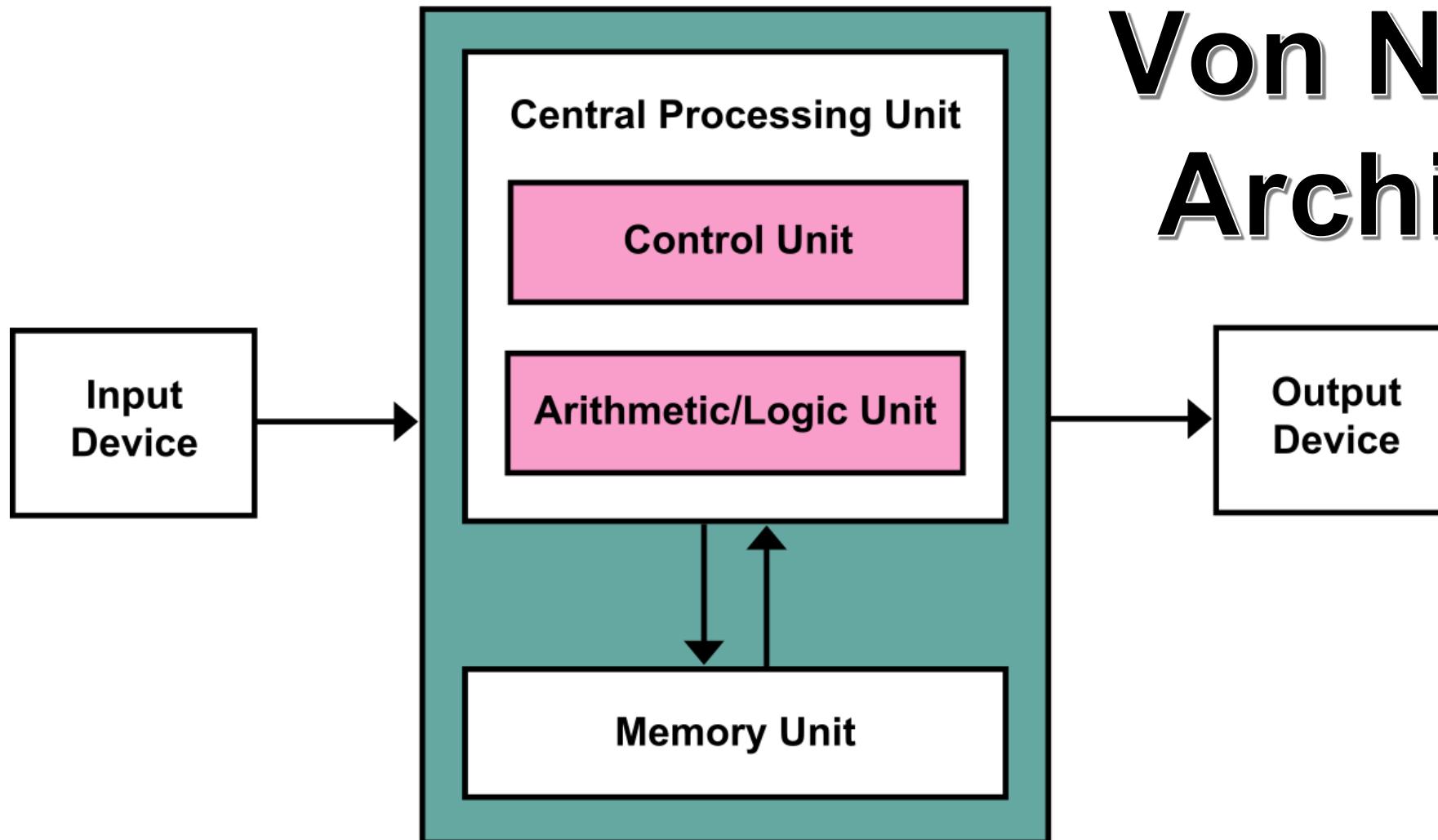
Computers can execute one or more instructions per clock cycle*



4GHz CPU speed = 4,000,000,000 clock cycles per second

Multi-core systems are even faster

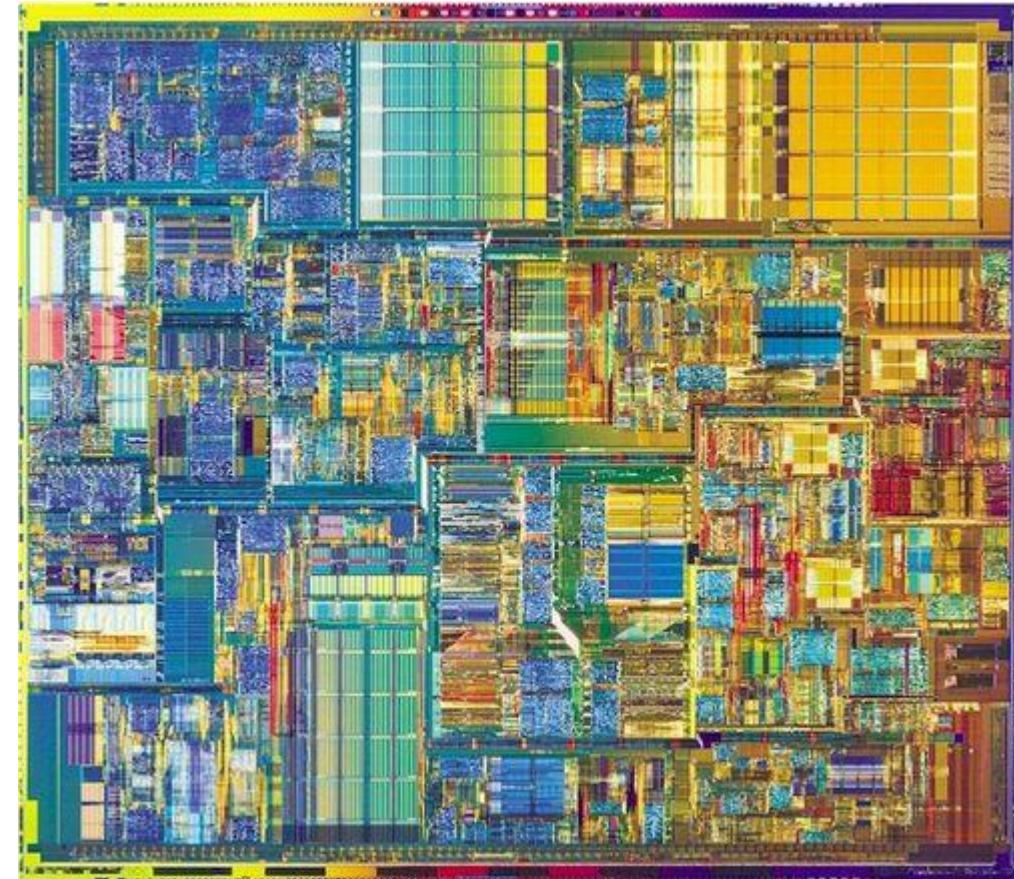
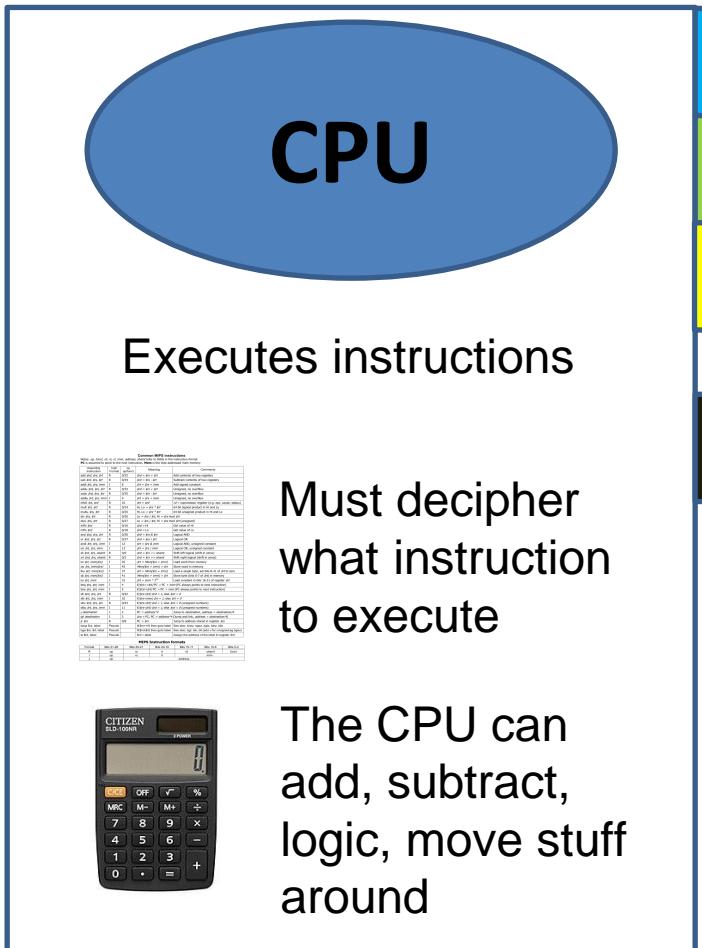
I. Hardware



Von Neumann Architecture

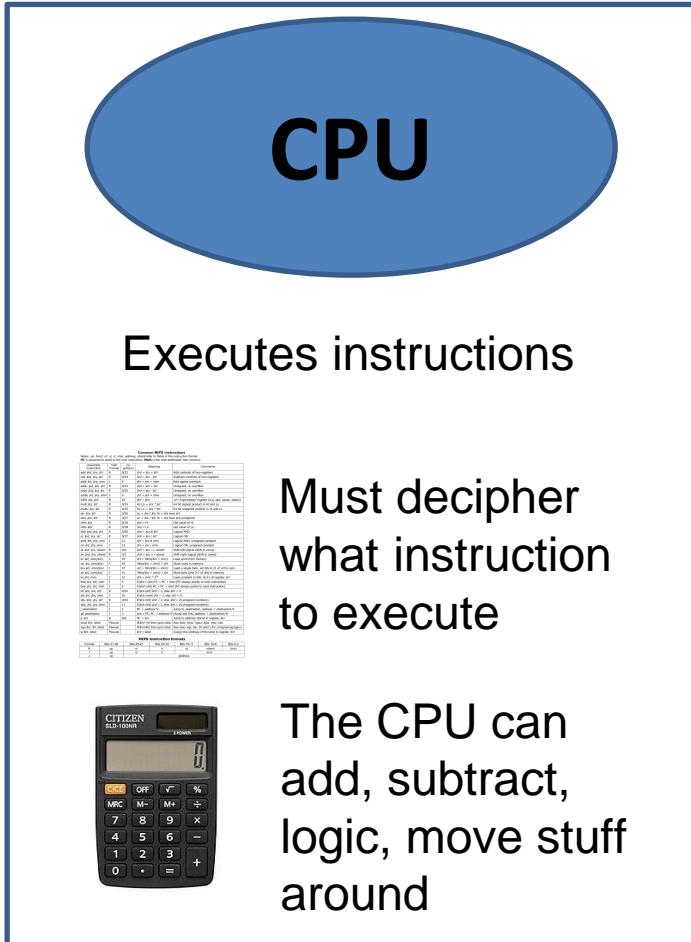
I. Hardware

Brain with no short-term memory



I. Hardware

Brain with no short-term memory



People have been able to create CPU components and fully functional, multi-core computers in games such as Minecraft

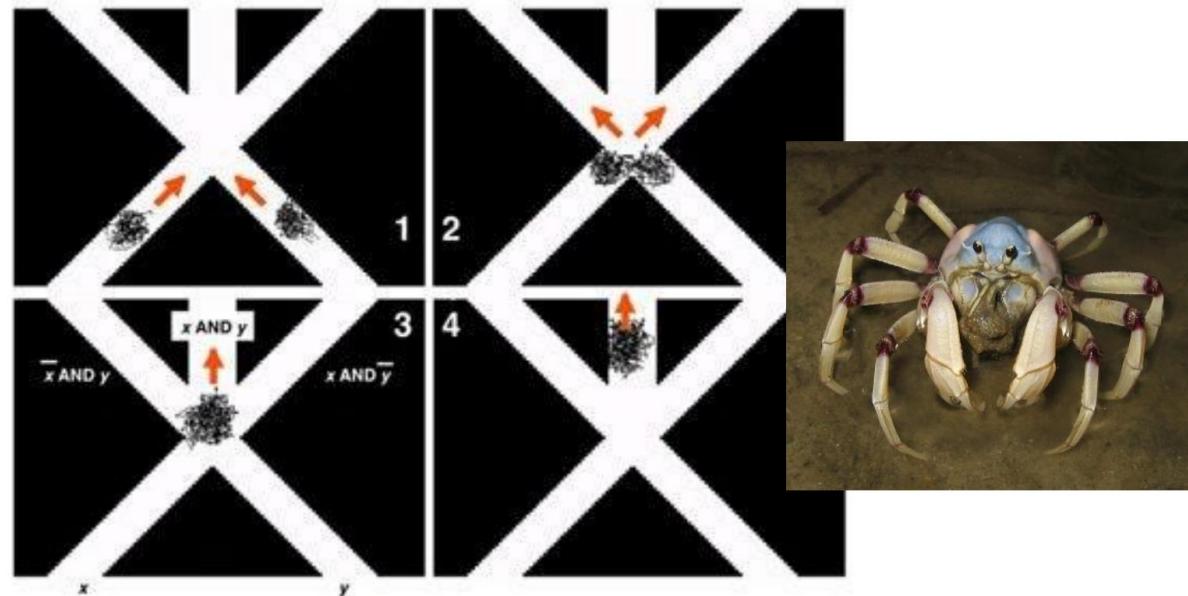
I. Hardware

WIRED STAFF

BUSINESS 04.14.2012 03:28 PM

Computer Built Using Swarms Of Soldier Crabs

Computer scientists at Kobe University in Japan have built a computer that draws inspiration from the swarming behavior of soldier crabs. The computer is based on theories from the early 1980s that studies how it could be possible to build a computer out of billiard balls. Proposed by Edward Fredkin and Tommaso Toffoli, the mechanical [...]



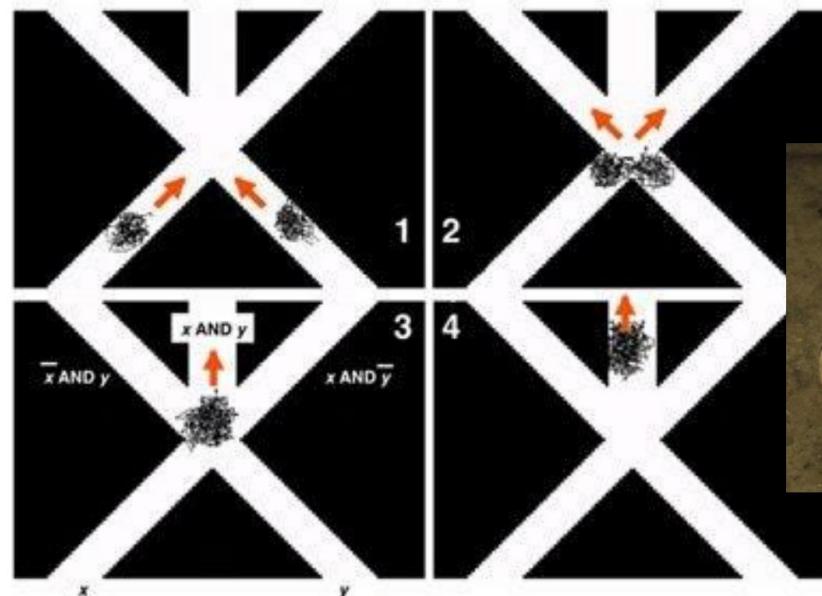
I. Hardware

WIRED STAFF

BUSINESS 04.14.2012 03:28 PM

Computer Built Using Swarms Of Soldier Crabs

Computer scientists at Kobe University in Japan have built a computer that draws inspiration from the swarming behavior of soldier crabs. The computer is based on theories from the early 1980s that studies how it could be possible to build a computer out of billiard balls. Proposed by Edward Fredkin and Tommaso Toffoli, the mechanical [...]



This is very real

Robust Soldier Crab Ball Gate

**Yukio-Pegio Gunji
Yuta Nishiyama**
Department of Earth and Planetary Sciences
Kobe University
Kobe 657-8501, Japan

Andrew Adamatzky
Unconventional Computing Centre
University of the West of England
Bristol, United Kingdom

Soldier crabs *Mictyris guinotae* exhibit pronounced swarming behavior. Swarms of the crabs are tolerant of perturbations. In computer models and laboratory experiments we demonstrate that swarms of soldier crabs can implement logical gates when placed in a geometrically constrained environment.

1. Introduction

All natural processes can be interpreted in terms of computations. To implement a logical gate in a chemical, physical, or biological spatially extended medium, Boolean variables must be assigned to disturbances, defects, or localizations traveling in the medium. These traveling patterns collide and the outcome of their collisions are converted

<https://wpmedia.wolfram.com/uploads/sites/13/2018/02/20-2-2.pdf>

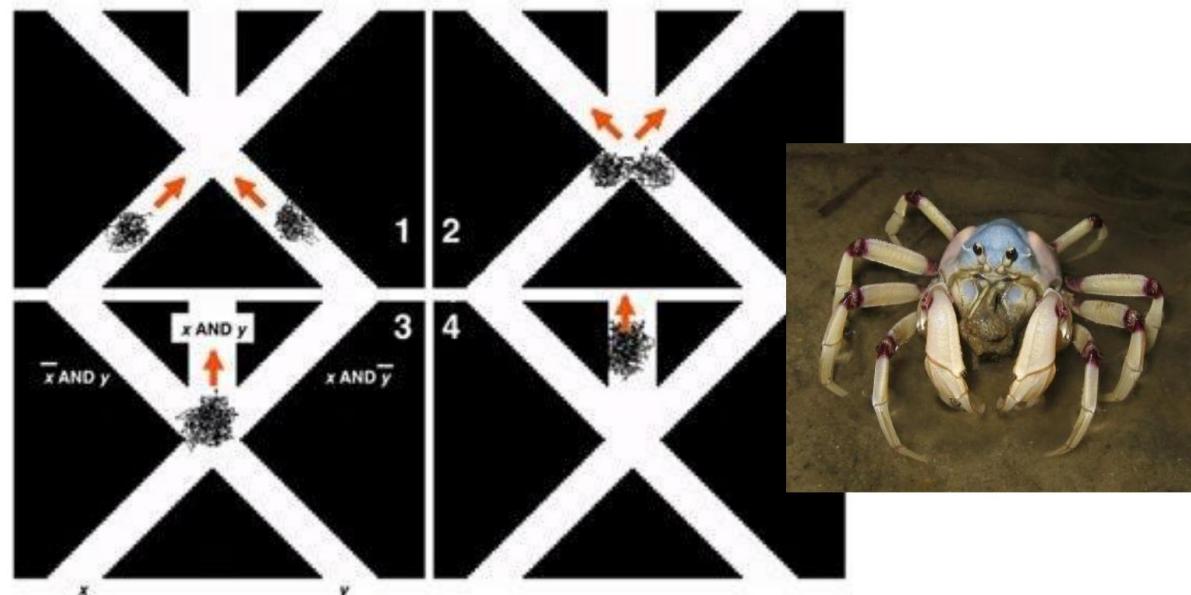
I. Hardware

WIRED STAFF

BUSINESS 04.14.2012 03:28 PM

Computer Built Using Swarms Of Soldier Crabs

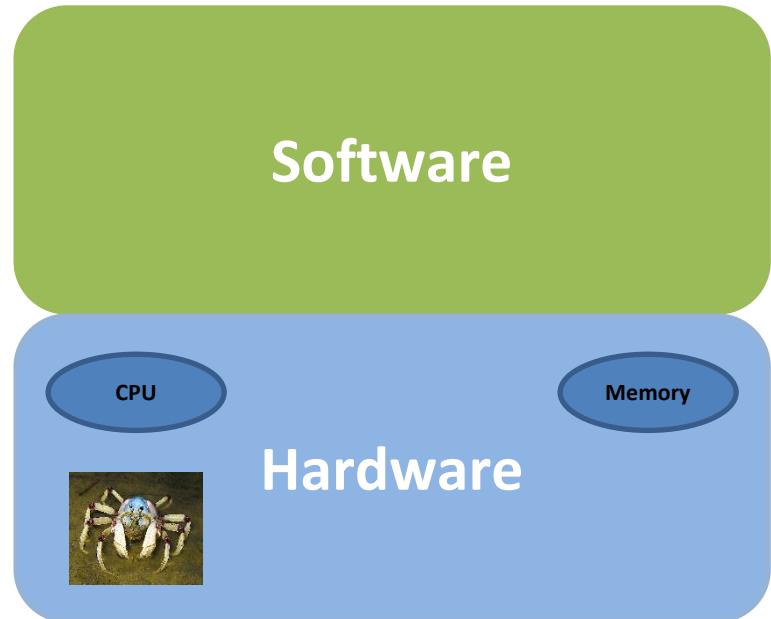
Computer scientists at Kobe University in Japan have built a computer that draws inspiration from the swarming behavior of soldier crabs. The computer is based on theories from the early 1980s that studies how it could be possible to build a computer out of billiard balls. Proposed by Edward Fredkin and Tommaso Toffoli, the mechanical [...]



(In theory) If you wanted to play Doom (1993) using a CPU made from soldier crabs, you would need 22 million crabs

(source: twitter)

How does this happen?



From a high level, we will divide a computer system into two parts

- I. **Hardware**
- II. **Software**

II. Software

A sequence of instructions, or **program**,
that tells the computer how to work



brightspace
by D2L

II. Software

A sequence of instructions, or **program**,
that tells the computer how to work

Humans write code in binary ?

```
01110011 00111011 01101111 01110101 01101100 01101110 01100111 01101001  
01100001 00111011 01110011 01100111 01101110 00111011 01110011 01101011  
01100100 01110110 01101110 01100001 00111011 01100100 01100111 01110100  
00111011 01001111 01001100 01001011 01010011 01000100 01000110 00111011  
01001100 01010011 01000100 01001000 01000111 01000100 01001100 00111011  
01000110 01010011 01001011 01000111 01100110 01100100 01101000 01100100  
01100110 01110011 01101000 01110011 01100110 01110111 01100101 01110111  
01100101 01110010 01110111 01100101 01110010 01100110 01110110 01100111  
01100111 01100100 01100010 01100100 01100110 01101000 01100010 01100100  
01100110 01100100 01110011 01100100 01100110 01110011 01100100 01100110  
01110011 01100100 01100110 01110011 01100100 01101000 01100100 01100110  
01101000 01110011 01100100 01100110 01110011 01100100 01100110 01110011  
01100100 01100110 01101000 01110100 01110010 01100100 01100110 01101000  
01100100 01100110 01100111 01101000 01100110 01100111 01100111 01110100  
01110010 01111001 01101000 01110100 01110010 01101000 01100100 01100110  
01100110 01101000 01100111 01100110 01101000 01110011 01110010 01110100  
01101000 01110100 01110010 01101000 01110011 01110100 01110010 01100100  
01100110 01101000 01100110 01100110 01101000 01100100 01100110 01101000  
01100100 01100110 01101000 01100101 01110111 01100110 01110011 01100100  
01100110 01110010 01100101 01110111 01100100 01100111 01100101 01100111  
01100100 01100110 01110011 01100111 01110011 01100110 01100100 01100111  
00111011 01101111 01110101 01101100 01101110 01100111 01101001 01100001  
00111011 01110011 01100111 01101110 00111011 01110011 01101011 01100100  
01110110 01101110 01100001 00111011 01100100 01100111 01110100 00111011  
01001111 01001100 01001011 01010011 01000100 01000110 00111011 01001100  
01010011 01000100 01001000 01000111 01000100 01001100 00111011 01000110  
01010011 01001011 01000111 01100110 01100100 01101000 01100100 01100110  
01110011 01101000 01110011 01100110 01110111 01100101 01110111 01100101  
01110010 01110111 01100101 01110010 01100110 01110110 01100111 01110011  
01100100 01100010 01100100 01100110 01101000 01100010 01100100 01100110  
01100100 01110011 01100100 01100110 01110011 01100111 01100100 01110011  
01100100 01100110 01110011 01100100 01101000 01100100 01100110 01101000  
01110011 01100100 01100110 01110011 01100100 01100110 01110011 01100100  
01100110 01101000 01110100 01110010 01100100 01100110 01101000 01100100  
01100110 01100111 01101000 01100110 01100111 01100111 01100111 01110100  
01111001 01101000 01110100 01110010 01101000 01110010 01110011 01100110  
01101000 01100111 01100110 01101000 01110011 01110010 01110100 01110100
```

II. Software

A sequence of instructions, or **program**,
that tells the computer how to work

Humans write code in binary ?

NO

```
01110011 00111011 01101111 01110101 01101100 01101110 01100111 01101001  
01100001 00111011 01110011 01100111 01101110 00111011 01110011 01101011  
01100100 01110110 01101110 01100001 00111011 01100100 01100111 01110100  
00111011 01001111 01001100 01001011 01010011 01000100 01000110 00111011  
01001100 01010011 01000100 01001000 01000111 01000100 01001100 00111011  
01000110 01010011 01001011 01000111 01100110 01100100 01101000 01100100  
01100110 01110011 01101000 01110011 01100110 01110111 01100101 01110111  
01100101 01110010 01110111 01100101 01110010 01100110 01110110 01100111  
01100111 01100100 01100010 01100100 01100110 01101000 01100010 01100100  
01100110 01100100 01110011 01100100 01100110 01110011 01100100 01100110  
01110011 01100100 01100110 01110011 01100100 01101000 01100100 01100110  
01101000 01110011 01100100 01100110 01110011 01100100 01100110 01110011  
01100100 01100110 01101000 01110100 01110010 01100100 01100110 01101000  
01100100 01100110 01100111 01101000 01100110 01100111 01100111 01110100  
01110010 01111001 01101000 01110100 01110010 01101000 01100100 01100110  
01100110 01101000 01100111 01100110 01101000 01110011 01110010 01110100  
01101000 01110100 01110010 01101000 01110011 01110100 01110010 01100100  
01100110 01101000 01100110 01100110 01101000 01100100 01100110 01101000  
01100110 01100110 01101000 01100101 01110111 01100110 01110011 01100100  
01100110 01110010 01100101 01110111 01100110 01100111 01100101 01100111  
01100100 01100110 01110011 01100111 01110011 01100110 01100100 01100111  
00111011 01101111 01110101 01101100 01101110 01100111 01101001 01100001  
00111011 01110011 01100111 01101110 00111011 01110011 01101011 01100100  
01110110 01101110 01100001 00111011 01100100 01100111 01110100 00111011  
01001111 01001100 01001011 01010011 01000100 01000110 00111011 01001100  
01010011 01000100 01001000 01000111 01000100 01001100 00111011 01000110  
01010011 01001011 01000111 01100110 01100100 01101000 01100100 01100110  
01110011 01101000 01110011 01100110 01110111 01100101 01110111 01100101  
01110010 01110111 01100101 01110010 01100110 01110110 01100111 01110011  
01100100 01100010 01100100 01100110 01101000 01100010 01100100 01100110  
01100100 01110011 01100100 01100110 01110011 01100111 01100100 01110011  
01100100 01100110 01110011 01100100 01101000 01100100 01100110 01101000  
01110011 01100100 01100110 01110011 01100100 01100110 01110011 01100100  
01100110 01101000 01110100 01110010 01100100 01100110 01101000 01100100  
01100110 01100111 01101000 01100110 01100111 01100111 01100100 01110100  
01111001 01101000 01110100 01110010 01101000 01110010 01110011 01100110  
01101000 01100111 01100110 01101000 01110011 01110010 01110100 01101000
```

II. Software

We write programs in a readable, higher-level language

```
#include <stdio.h>

int main() {
    printf("Hello WOrld! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

    printf("%d %d %d \n",x,y,z);
    return 0;
}

class Person():
    #method to initialize name and age attributes.
    def __init__(self,name, age):
        self.name = name
        self.age = age
    #method to demonstrate what a person eats
    def eat(self):
        print(self.name.title() + "eats Matooke and rice")
        print("She is"+ str(self.age) + " years old")
    def drink(self):
        print("Drinks water")
    #instantiating a class.
    my_sister = Person("Haniifa", 30)
    #Accessing the class method through the class object.
    my_sister.eat()
```



II. Software

We need a way to convert **source** code to **binary**

```
#include <stdio.h>

int main() {
    printf("Hello WOrld! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

    printf("%d %d %d \n",x,y,z);
    return 0;
}
```

```
class Person():
    #method to initialize name and age attributes.
    def __init__(self,name, age):
        self.name = name
        self.age = age
    #method to demonstrate what a person eats
    def eat(self):
        print(self.name.title() + " eats Matooke and rice")
        print("She is"+ str(self.age) + " years old")
    def drink(self):
        print("Drinks water")
    #instantiating a class.
    my_sister = Person("Haniifa", 30)
    #Accessing the class method through the class object.
    my_sister.eat()
```

We need a way to convert **source** code to **binary**

II. Software

```
#include <stdio.h>

int main() {
    printf("Hello World! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

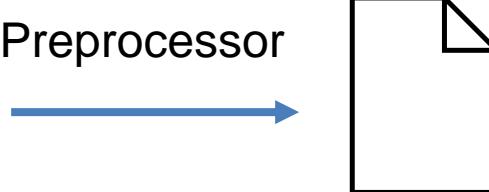
    printf("%d %d %d \n",x,y,z);
    return 0;
}
```

We need a way to convert **source** code to **binary**

II. Software

```
#include <stdio.h>
int main() {
    printf("Hello World! \n");
    int x = 0;
    int y = 3;
    int z = x + y;
    printf("%d %d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

We need a way to convert **source** code to **binary**

II. Software

```
#include <stdio.h>

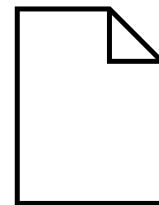
int main() {
    printf("Hello World! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

    printf("%d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
0: f3 0f 1e fa        endbr64
4: 55                 push  %rbp
5: 48 89 e5           mov   %rsp,%rbp
8: 48 83 ec 10       sub   $0x10,%rsp
c: 48 8d 3d 00 00 00  lea   0x0(%rip),%rdi      # 13 <main+0x13>
13: e8 00 00 00 00    callq 18 <main+0x18>
18: c7 45 f4 00 00 00  movl  $0x0,-0xc(%rbp)
1f: c7 45 f8 03 00 00  movl  $0x3,-0x8(%rbp)
26: 8b 55 f4           mov   -0xc(%rbp),%edx
29: 8b 45 f8           mov   -0x8(%rbp),%eax
2c: 01 d0               add   %edx,%eax
2e: 89 45 fc           mov   %eax,-0x4(%rbp)
31: 8b 4d fc           mov   -0x4(%rbp),%ecx
34: 8b 55 f8           mov   -0x8(%rbp),%edx
37: 8b 45 f4           mov   -0xc(%rbp),%eax
3a: 89 c6               mov   %eax,%esi
3c: 48 8d 3d 00 00 00  lea   0x0(%rip),%rdi      # 43 <main+0x43>
43: b8 00 00 00 00    mov   $0x0,%eax
48: e8 00 00 00 00    callq 4d <main+0x4d>
4d: b8 00 00 00 00    mov   $0x0,%eax
52: c9                 leaveq 
53: c3                 retq
```

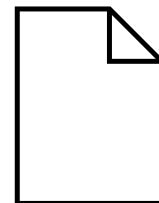
- Converted to assembly code
- .s file

We need a way to convert **source** code to **binary**

II. Software

```
#include <stdio.h>
int main() {
    printf("Hello World! \n");
    int x = 0;
    int y = 3;
    int z = x + y;
    printf("%d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
0: f3 0f 1e fa        endbr64
4: 55                 push  %rbp
5: 48 89 e5          mov   %rsp,%rbp
8: 48 83 ec 10       sub   $0x10,%rsp
c: 48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
13: e8 00 00 00 00 00 00  callq 18 <main+0x18>
18: c7 45 f4 00 00 00 00  movl  $0x0,-0xc(%rbp)
1f: c7 45 f8 03 00 00 00  movl  $0x3,-0x8(%rbp)
26: 8b 55 f4          mov   -0xc(%rbp),%edx
29: 8b 45 f8          mov   -0x8(%rbp),%eax
2c: 01 d0              add   %edx,%eax
2e: 89 45 fc          mov   %eax,-0x4(%rbp)
31: 8b 4d fc          mov   -0x4(%rbp),%ecx
34: 8b 55 f8          mov   -0x8(%rbp),%eax
37: 8b 45 f4          mov   -0xc(%rbp),%eax
3a: 89 c6              mov   %eax,%esi
3c: 48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
43: b8 00 00 00 00 00 00  mov   $0x0,%eax
48: e8 00 00 00 00 00 00  callq 4d <main+0x4d>
4d: b8 00 00 00 00 00 00  mov   $0x0,%eax
52: c9                 leaveq 
53: c3                 retq
```

- Converted to assembly code
- .s file

1	00000000 00000100 0000000000000000
2	01011110 00001100 11000010 0000000000000010
3	11101111 00010110 00000000000000101
4	11101111 10011110 0000000000001011
5	11111000 10101101 11011111 00000000000010010
6	01100010 11011111 00000000000010101
7	11101111 00000010 11110111 00000000000010111
8	11110100 10101101 11011111 00000000000011110
9	00000011 10100010 11011111 0000000000100001
10	11101111 00000010 11110111 0000000000100100
11	01111110 11110100 10101101
12	11111000 10101110 11000101 0000000000101011
13	00000110 10100010 11111011 0000000000110001
14	11101111 00000010 11111011 0000000000110100
15	01010000 11010100 0000000000111011
16	00000100 0000000000111101

Assembler

**THIS PROCESS IS NOT TRUE FOR EVERY LANGUAGE

II. Software

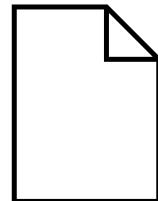
```
#include <stdio.h>
int main() {
    printf("Hello World! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

    printf("%d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
0: f3 0f 1e fa        endbr64
4: 55                 push  %rbp
5: 48 89 e5          mov   %rsp,%rbp
8: 48 83 ec 10       sub   $0x10,%rsp
c: 48 8d 3d 00 00 00 00
13: e8 00 00 00 00 00 00
18: c7 45 f4 00 00 00 00
1f: c7 45 f8 03 00 00 00
26: 8b 55 f4          movl  $0x3,-0x8(%rbp)
29: 8b 45 f8          movl  -0xc(%rbp),%edx
2c: 01 d0              add   %edx,%eax
2e: 89 45 fc          mov   %eax,-0x4(%rbp)
31: 8b 4d fc          movl  -0x4(%rbp),%ecx
34: 8b 55 f8          movl  -0x8(%rbp),%edx
37: 8b 45 f4          movl  -0xc(%rbp),%eax
3a: 89 c6              mov   %eax,%esi
3c: 48 8d 3d 00 00 00 00
43: b8 00 00 00 00 00 00
48: e8 00 00 00 00 00 00
4d: b8 00 00 00 00 00 00
52: c9                 mov   $0x0,%eax
53: c3                 leaveq
                         retq
```

- Converted to assembly code
- .s file

We still need to resolve function calls
i.e. printf, sleep, sqrt, etc

Linker

1	00000000 00000100 0000000000000000
2	01011110 00001100 11000010 0000000000000010
3	11101111 00010110 00000000000000101
4	11101111 10011110 0000000000001011
5	11111000 10101101 11011111 0000000000010010
6	01100010 11011111 0000000000010101
7	11101111 00000010 11110111 0000000000010111
8	11110100 10101101 11011111 0000000000011110
9	00000011 10100010 11011111 0000000000100001
10	11101111 00000010 11110111 0000000000100100
11	01111110 11110100 10101101
12	11111000 10101110 11000101 0000000000101011
13	00000110 10100010 11110111 0000000000110001
14	11101111 00000010 11110111 0000000000110100
15	01010000 11010100 0000000000111011
16	00000100 0000000000111101

Assembler

We need a way to convert **source** code to **binary**

II. Software

```
#include <stdio.h>

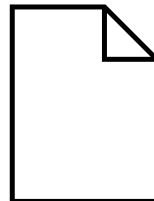
int main() {
    printf("Hello World! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

    printf("%d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
0: f3 0f 1e fa        endbr64
4: 55                 push  %rbp
5: 48 89 e5          mov   %rsp,%rbp
8: 48 83 ec 10       sub   $0x10,%rsp
c: 48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
13: e8 00 00 00 00 00 00  callq 18 <main+0x18>
18: c7 45 f4 00 00 00 00  movl  $0x0,-0xc(%rbp)
1f: c7 45 f8 03 00 00 00  movl  $0x3,-0x8(%rbp)
26: 8b 55 f4          mov   -0xc(%rbp),%edx
29: 8b 45 f8          mov   -0x8(%rbp),%eax
2c: 01 d0              add   %edx,%eax
2e: 89 45 fc          mov   %eax,-0x4(%rbp)
31: 8b 4d fc          mov   -0x4(%rbp),%ecx
34: 8b 55 f8          mov   -0xc(%rbp),%eax
37: 8b 45 f4          mov   -0x8(%rbp),%eax
3a: 89 c6              mov   %eax,%esi
3c: 48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
43: b8 00 00 00 00 00 00  mov   $0x0,%eax
48: e8 00 00 00 00 00 00  callq 4d <main+0x4d>
4d: b8 00 00 00 00 00 00  mov   $0x0,%eax
52: c9                 leaveq 
53: c3                 retq
```

Assembler

- Converted to assembly code
- .s file

Two methods:



Library 1

string.h

Library 2

stdio.h

Linker

```
1      00000000 00000100 0000000000000000
2 01011110 00001100 11000010 0000000000000010
3           11101111 00010110 00000000000000101
4           11101111 10011110 0000000000001011
5 11111000 10101101 11011111 00000000000010010
6           01100010 11011111 00000000000010101
7 11101111 00000010 11111011 00000000000010111
8 11110100 10101101 11011111 00000000000011110
9 00000011 10100010 11011111 0000000000100001
10 11101111 00000010 11111011 0000000000100100
11 01111110 11110100 10101101
12 11111000 10101110 11000101 0000000000101011
13 00000110 10100010 11111011 0000000000110001
14 11101111 00000010 11111011 0000000000110100
15           01010000 11010100 0000000000111011
16           00000100 0000000000111101
```

**THIS PROCESS IS NOT TRUE FOR EVERY LANGUAGE

II. Software

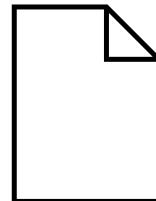
```
#include <stdio.h>
int main() {
    printf("Hello World! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

    printf("%d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
0: f3 0f 1e fa        endbr64
4: 55                 push  %rbp
5: 48 89 e5           mov   %rsp,%rbp
8: 48 83 ec 10       sub   $0x10,%rsp
c: 48 8d 3d 00 00 00 00
13: e8 00 00 00 00 00 00
18: c7 45 f4 00 00 00 00
1f: c7 45 f8 03 00 00 00
26: 8b 55 f4           movl  $0x3,-0x8(%rbp)
29: 8b 45 f8           movl  -0xc(%rbp),%edx
2c: 01 d0               add   %edx,%eax
2e: 89 45 fc           movl  %eax,-0x4(%rbp)
31: 8b 4d fc           movl  -0x4(%rbp),%ecx
34: 8b 55 f8           movl  -0x8(%rbp),%eax
37: 8b 45 f4           movl  -0xc(%rbp),%eax
3a: 89 c6               movl  %eax,%esi
3c: 48 8d 3d 00 00 00 00
43: b8 00 00 00 00 00 00
48: e8 00 00 00 00 00 00
4d: b8 00 00 00 00 00 00
52: c9                 movl  $0x0,%eax
53: c3                 leaveq
                         retq
```

Assembler

- Converted to assembly code
- .s file

Static Linking- required code and data copied into executable at compile time



Linker

1	00000000 00000100 0000000000000000
2	01011110 00001100 11000010 0000000000000010
3	11101111 00010110 00000000000000101
4	11101111 10011110 0000000000001011
5	11111000 10101101 11011111 00000000000010010
6	01100010 11011111 00000000000010101
7	11101111 00000010 11110111 00000000000010111
8	11110100 10101101 11011111 00000000000011110
9	00000011 10100010 11011111 0000000000100001
10	11101111 00000010 11110111 0000000000100100
11	01111110 11110100 10101101
12	11111000 10101110 11000101 0000000000101011
13	00000110 10100010 11110111 0000000000110001
14	11101111 00000010 11110111 0000000000110100
15	01010000 11010100 0000000000111011
16	00000100 0000000000111101

We need a way to convert **source** code to **binary**

II. Software

```
#include <stdio.h>

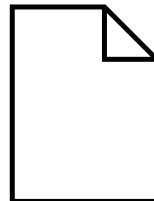
int main() {
    printf("Hello World! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

    printf("%d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
0: f3 0f 1e fa      endbr64
4: 55              push  %rbp
5: 48 89 e5        mov   %rsp,%rbp
8: 48 83 ec 10    sub   $0x10,%rsp
c: 48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
13: e8 00 00 00 00 00 00  callq 18 <main+0x18>
18: c7 45 f4 00 00 00 00  movl  $0x0,-0xc(%rbp)
1f: c7 45 f8 03 00 00 00  movl  $0x3,-0x8(%rbp)
26: 8b 55 f4        mov   -0xc(%rbp),%edx
29: 8b 45 f8        mov   -0x8(%rbp),%eax
2c: 01 d0          add   %edx,%eax
2e: 89 45 fc        mov   %eax,-0x4(%rbp)
31: 8b 4d fc        mov   -0x4(%rbp),%ecx
34: 8b 55 f8        mov   -0xc(%rbp),%eax
37: 8b 45 f4        mov   -0x8(%rbp),%eax
3a: 89 c6          mov   %eax,%esi
3c: 48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
43: b8 00 00 00 00 00 00  mov   $0x0,%eax
48: e8 00 00 00 00 00 00  callq 4d <main+0x4d>
4d: b8 00 00 00 00 00 00  mov   $0x0,%eax
52: c9              leaveq 
53: c3              retq 

# 13 <main+0x13>
# 43 <main+0x43>
```

Assembler

- Converted to assembly code
- .s file

Two methods:



Library 1

string.h

Library 2

stdio.h

Linker

```
1 00000000 00000100 0000000000000000
2 01011110 00001100 11000010 0000000000000010
3 11101111 00010110 00000000000000101
4 11101111 10011110 0000000000001011
5 11111000 10101101 11011111 0000000000010010
6 01100010 11011111 0000000000010101
7 11101111 00000010 11111011 0000000000010111
8 11110100 10101101 11011111 0000000000011110
9 00000011 10100010 11011111 0000000000100001
10 11101111 00000010 11111011 0000000000100100
11 01111110 11110100 10101101
12 11111000 10101110 110000101 0000000000101011
13 00000110 10100010 11111011 0000000000110001
14 11101111 00000010 11111011 0000000000110100
15 01010000 11010100 0000000000111011
16 00000100 0000000000111101
```

**THIS PROCESS IS NOT TRUE FOR EVERY LANGUAGE

We need a way to convert **source** code to **binary**

II. Software

```
#include <stdio.h>

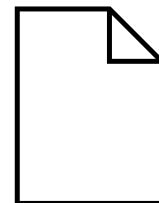
int main() {
    printf("Hello World! \n");

    int x = 0;
    int y = 3;

    int z = x + y;

    printf("%d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
0: f3 0f 1e fa        endbr64
4: 55                 push  %rbp
5: 48 89 e5          mov   %rsp,%rbp
8: 48 83 ec 10       sub   $0x10,%rsp
c: 48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
13: e8 00 00 00 00 00 00  callq 18 <main+0x18>
18: c7 45 f4 00 00 00 00  movl  $0x0,-0xc(%rbp)
1f: c7 45 f8 03 00 00 00  movl  $0x3,-0x8(%rbp)
26: 8b 55 f4          mov   -0xc(%rbp),%edx
29: 8b 45 f8          mov   -0x8(%rbp),%eax
2c: 01 d0              add   %edx,%eax
2e: 89 45 fc          mov   %eax,-0x4(%rbp)
31: 8b 4d fc          mov   -0x4(%rbp),%ecx
34: 8b 55 f8          mov   -0xc(%rbp),%eax
37: 8b 45 f4          mov   -0x8(%rbp),%eax
3a: 89 c6              mov   %eax,%esi
3c: 48 8d 3d 00 00 00 00  lea   0x0(%rip),%rdi
43: b8 00 00 00 00 00 00  mov   $0x0,%eax
48: e8 00 00 00 00 00 00  callq 4d <main+0x4d>
4d: b8 00 00 00 00 00 00  mov   $0x0,%eax
52: c9                 leaveq 
53: c3                 retq 

# 13 <main+0x13>
# 43 <main+0x43>
```

Assembler

- Converted to assembly code
- .s file

Two methods:

string.h

Library 1

stdio.h

Library 2

Program A

.exe

Linker

```
1      00000000 00000100 0000000000000000
2      01011110 00001100 11000010 0000000000000010
3      11101111 00010110 00000000000000101
4      11101111 10011110 0000000000001011
5      11111000 10101101 11011111 0000000000010010
6      01100010 11011111 0000000000010101
7      11101111 00000010 11111011 0000000000010111
8      11110100 10101101 11011111 0000000000011110
9      00000011 10100010 11011111 0000000000100001
10     11101111 00000010 11111011 0000000000100100
11     01111110 11110100 10101101
12     11111000 10101110 110000101 0000000000101011
13     00000110 10100010 11111011 0000000000110001
14     11101111 00000010 11111011 0000000000110100
15           01010000 11010100 0000000000111011
16           00000100 0000000000111101
```

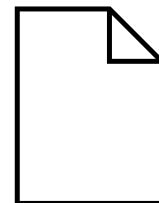
Dynamic Linking - required code and data is linked to executable at runtime

We need a way to convert **source** code to **binary**

II. Software

```
#include <stdio.h>
int main() {
    printf("Hello World! \n");
    int x = 0;
    int y = 3;
    int z = x + y;
    printf("%d %d \n",x,y,z);
    return 0;
}
```

Preprocessor



- Removal of comments
- Expand Macros

Compiler

```
0000000000000000 <main>:
0: f3 0f 1e fa      endbr64
4: 55              push  %rbp
5: 48 89 e5        mov   %rsp,%rbp
8: 48 83 ec 10    sub   $0x10,%rsp
c: 48 8d 3d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
13: e8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
18: c7 45 f4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1f: c7 45 f8 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00
26: 8b 55 f4        movl  $0x3,-0x8(%rbp)
29: 8b 45 f8        movl  -0xc(%rbp),%edx
2c: 01 d0          add   %edx,%eax
2e: 89 45 fc        movl  %eax,-0x4(%rbp)
31: 8b 4d fc        movl  -0x4(%rbp),%ecx
34: 8b 55 f8        movl  -0x8(%rbp),%eax
37: 8b 45 f4        movl  -0xc(%rbp),%eax
3a: 89 c6          movw  %eax,%esi
3c: 48 8d 3d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
43: b8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
48: e8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4d: b8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
52: c9              movq  $0x0,%eax
53: c3              leaveq
54: retq
```

Assembler

- Converted to assembly code
- .s file

To be continued

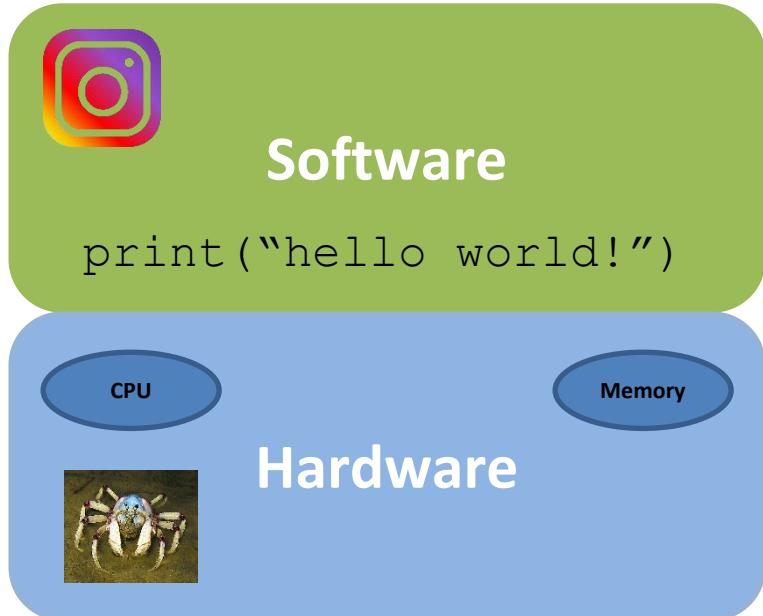


Linker

1	00000000 00000100 0000000000000000
2	01011110 00001100 11000010 0000000000000010
3	11101111 00010110 00000000000000101
4	11101111 10011110 0000000000001011
5	11111000 10101101 11011111 000000000010010
6	01100010 11011111 0000000000010101
7	11101111 00000010 11110111 000000000010111
8	11110100 10101101 11011111 000000000011110
9	00000011 10100010 11011111 000000000100001
10	11101111 00000010 11110111 000000000100100
11	01111110 11110100 10101101
12	11111000 10101110 11000101 000000000101011
13	00000110 10100010 11110111 000000000110001
14	11101111 00000010 11110111 000000000110100
15	01010000 11010100 000000000111011
16	00000100 000000000111101

**THIS PROCESS IS NOT TRUE FOR EVERY LANGUAGE

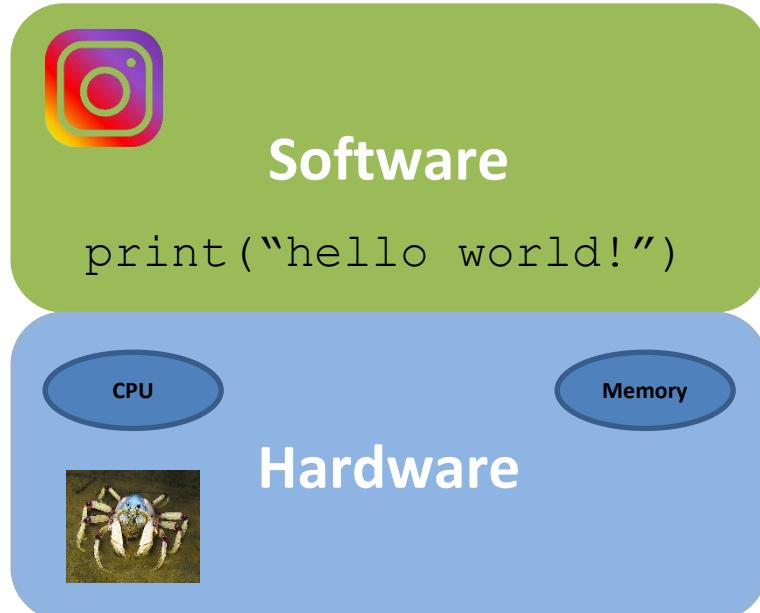
How does this happen?



From a high level, we will divide a computer system into two parts

- I. **Hardware**
- II. **Software**

How does this happen?

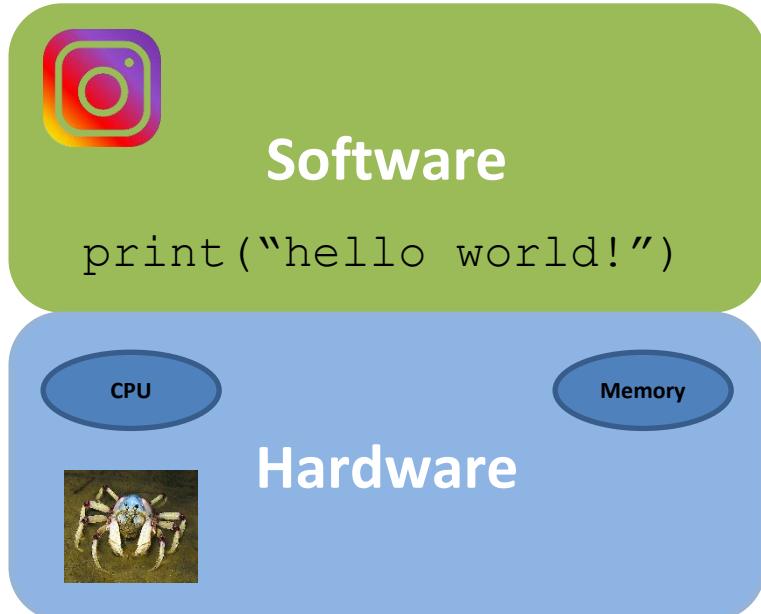


From a high level, we will divide a computer system into two parts

- I. **Hardware**
- II. **Software**

Software is nothing without hardware

How does this happen?

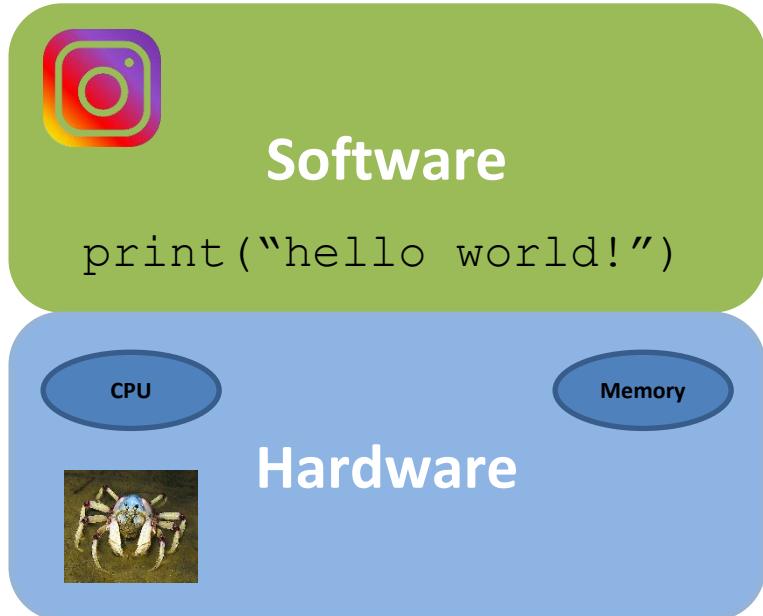


From a high level, we will divide a computer system into two parts

- I. **Hardware**
- II. **Software**

Hardware is *mostly* nothing without software

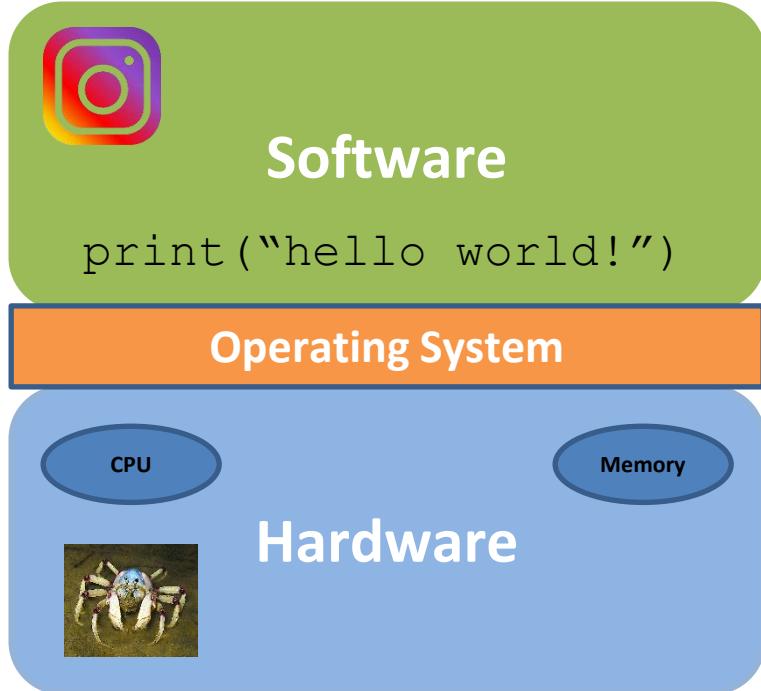
How does this happen?



From a high level, we will divide a computer system into two parts

- I. **Hardware**
- II. **Software**
- III. **???**

How does this happen?

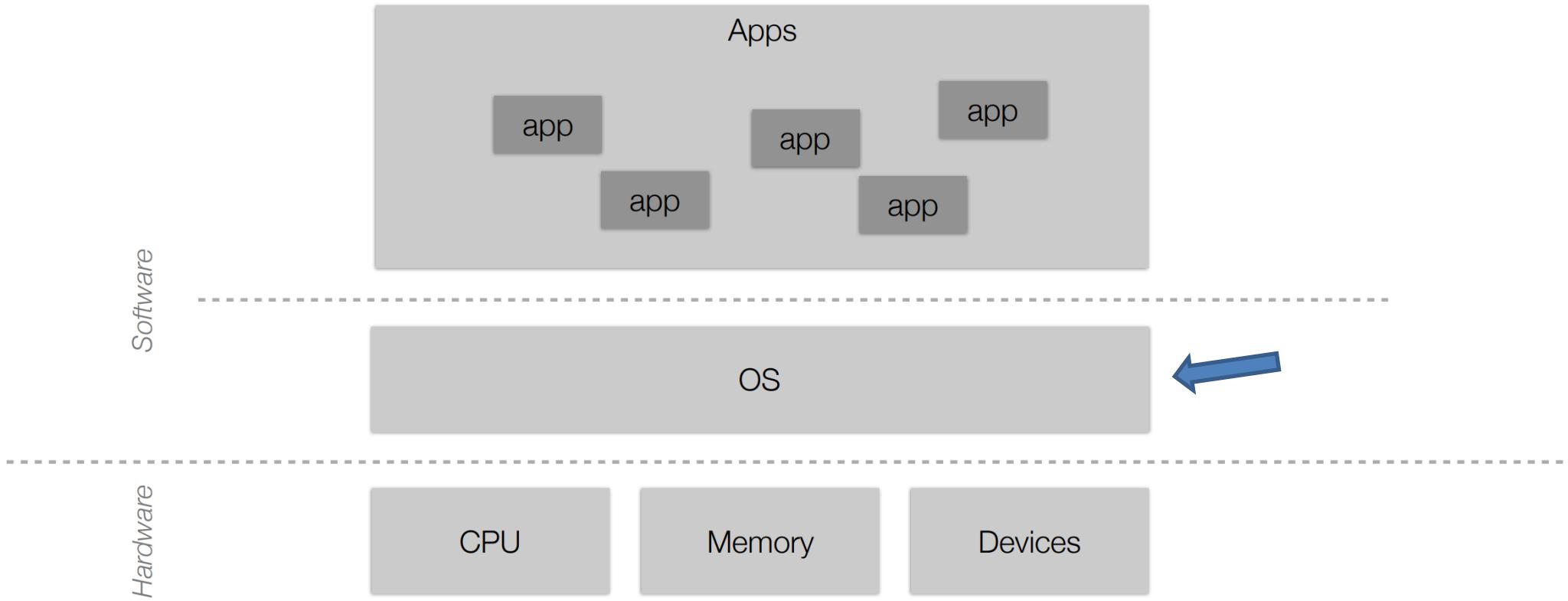


From a high level, we will divide a computer system into two parts

- I. **Hardware**
- II. **Software**
- III. **Operating System**

Typical Layers of a Computer

The **operating system** is a vital component of a computer



Typical Layers of a Computer

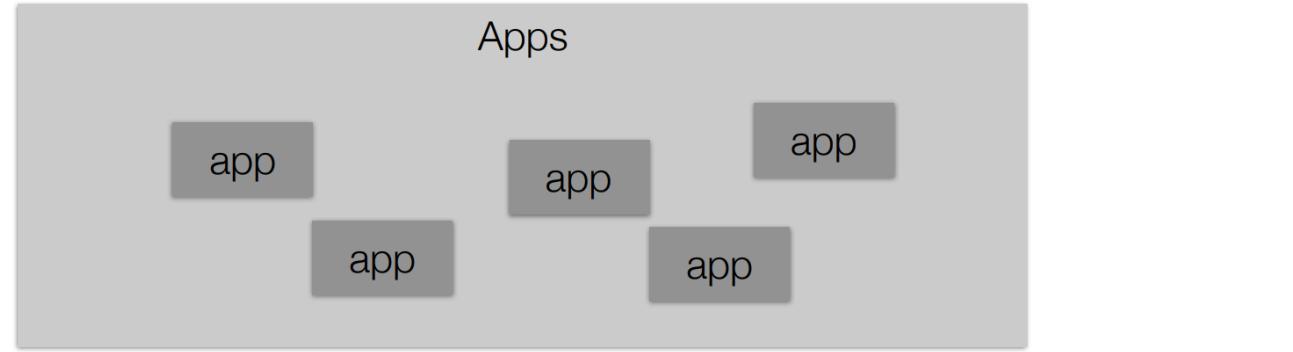
The **operating system** is a vital component of a computer

What do we trust?

What do we not trust?

Software

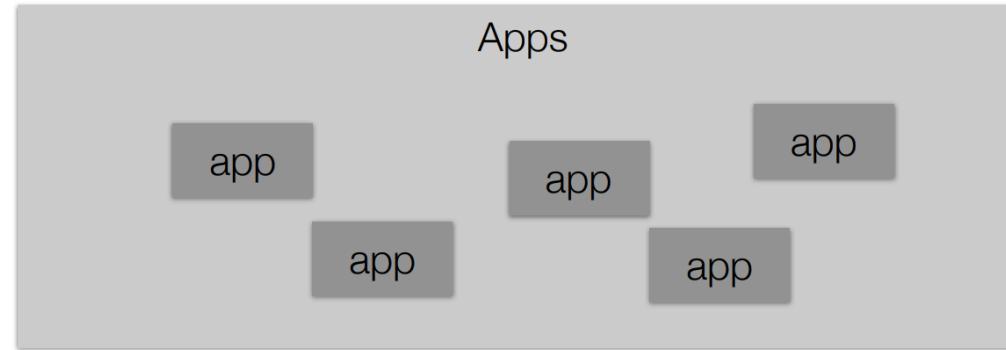
Hardware



Typical Layers of a Computer

The **operating system** is a vital component of a computer

Not privileged



Software

Privileged



Hardware

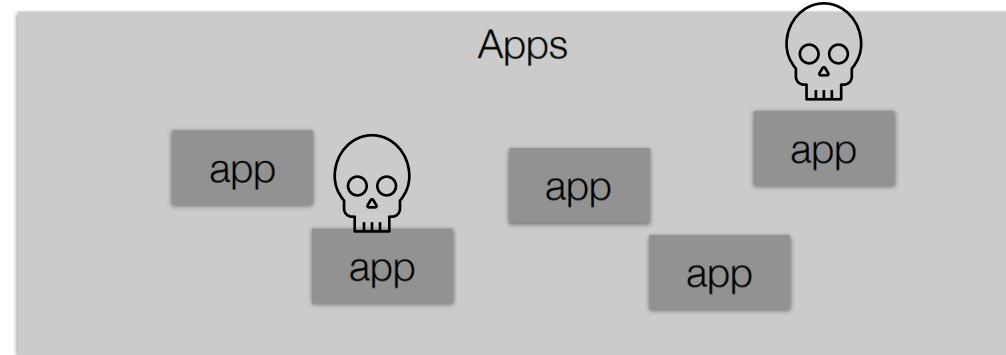


Hardware

Typical Layers of a Computer

The **operating system** is a vital component of a computer

Not privileged



Privileged



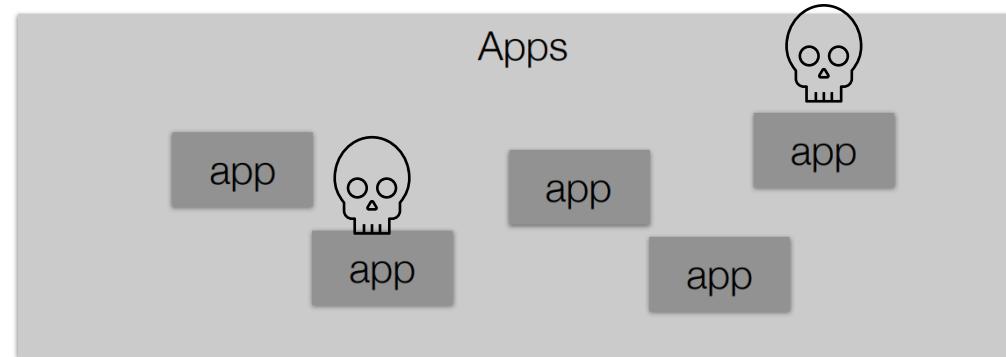
Hardware



Typical Layers of a Computer

The **operating system** is a vital component of a computer

Not privileged



Privileged



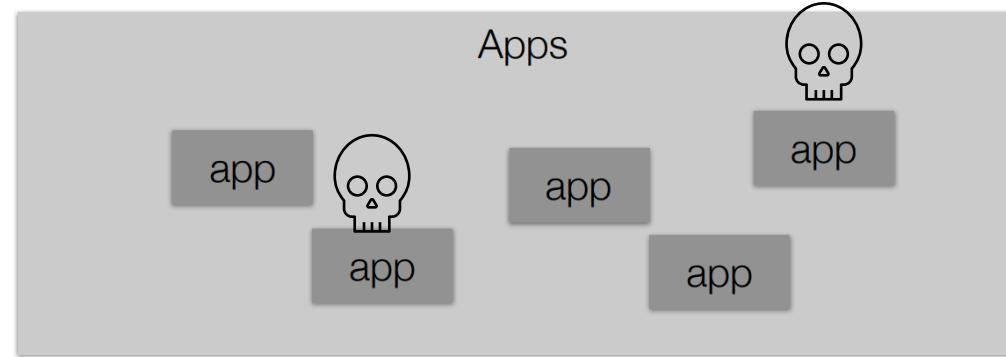
Hardware



Typical Layers of a Computer

The **operating system** is a vital component of a computer

Not privileged

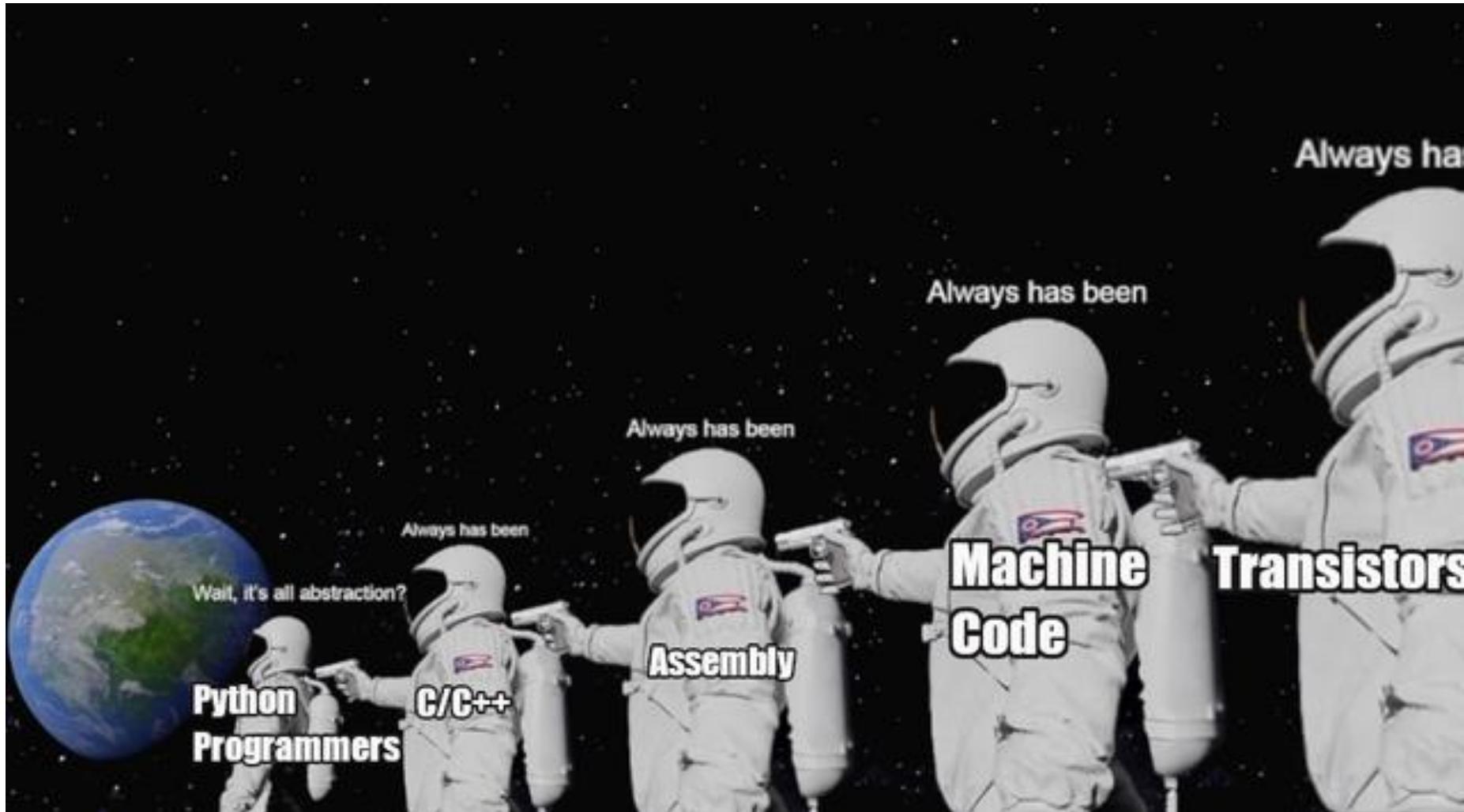


Privileged



Hardware





Meme credit: Carson Gross