Welcome to
“Computer Organization and Design Logic”

CS 64: Computer Organization and Design Logic
Lecture #1
Winter 2018

Ziad Matni
Dept. of Computer Science, UCSB

A Word About Registration for CS64

FOR THOSE OF YOU NOT YET REGISTERED:

- This class is currently **FULL**

- No one else is getting into this class, unless others drop it… ☹
  - **If you want to add this class, see me after lecture**
  - I will resolve all reg.s by NO LATER THAN ***tomorrow***

Your Instructor

Your instructor: **Ziad Matni** *(zee-ahd mat-knee)*

Email: ***zmatni@cs.ucsb.edu***

(please put CS64 at the start of the subject header – I teach 2 other classes!!!)

My office hours: Mondays **12:00 PM – 1:00 PM**, at **SMSS 4409**

(or by appointment)

Your TAs

All labs will take place in **PHELPS 3525**
All TA office hours will take place in “Open Lab” Time in **PHELPS 3525**

LAB TAs

Bay-Yuan Hsu

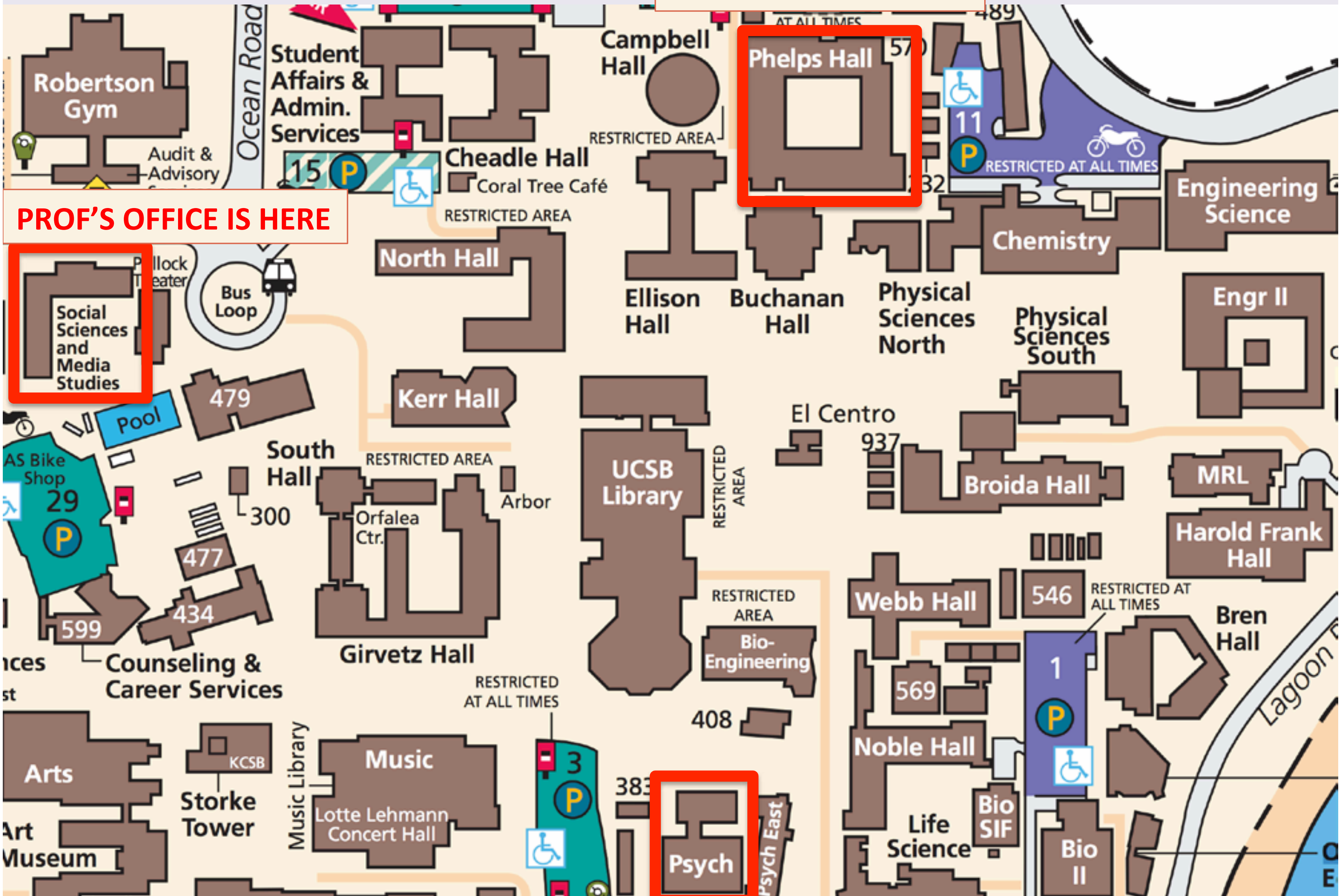
Fatih Bakir

GRADER

Zimu Yang

YOUR LABS ARE HERE

PROF'S OFFICE IS HERE



YOUR LECTURES ARE HERE

You!

With a show of hands, tell me… how many of you…

A. Are Freshmen? Sophomores? Juniors? Seniors?
B. Are CS majors? Other?
C. Know: C, C++, Java, Python, JavaScript, PERL, Bash programming?
D. Have NOT used a Linux or UNIX system before?
E. Have *seen* actual “assembly code” before?
F. Programmed in assembly before?
G. Written/seen code for *firmware*?
H. Understand basic binary logic (i.e. OR, AND, NOT)?
I. Designed a digital circuit before?

This Class

- This is an **introductory** course in **low-level programming** and **computer hardware**.
  - Two separate but very intertwined areas

- What happens between your C/C++/Java/Python command:

  int a = 3, b =4, c = a+b;

  and the actual ***"digital mechanisms***" in the CPU
  that process this "simple" command?

- This class will move *fast* – so please prepare accordingly.

Lecture Etiquette!

- I need you INVOLVED and ACTIVE!

- **Phones OFF!** and laptops/tablets are for **NOTES** only
  - No tweeting, texting, FB-ing, surfing, gaming, Snapchatting, spitting, etc.!
  - I will ask you to leave class if you do not follow this policy. Especially if you are disrupting others.

- To succeed in this class, you need to take thorough notes
  - I'll provide my slides, but not class notes
  - Studies show that ***written*** notes are superior to typing them into a laptop!

Class Website

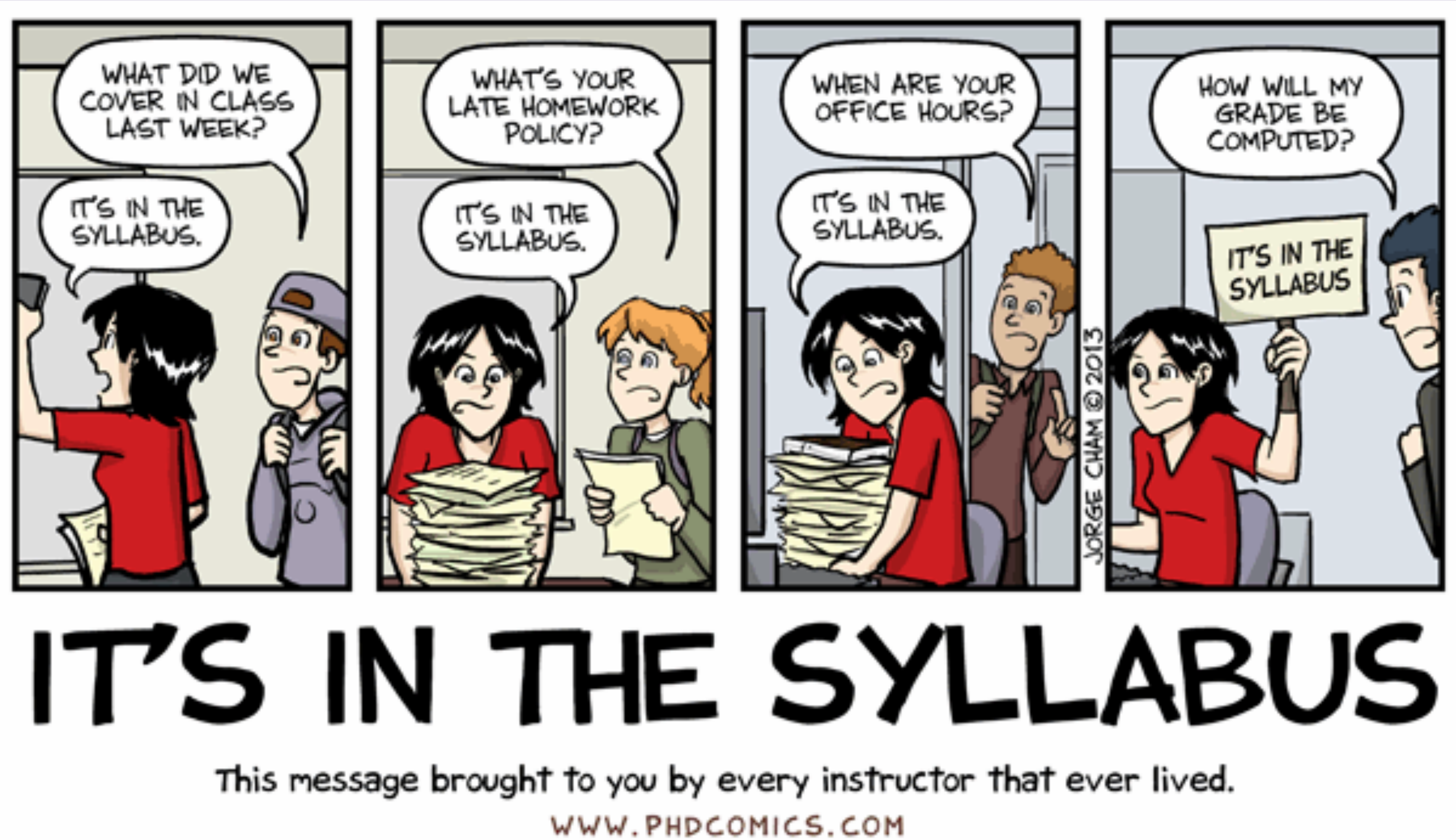
Website:

https://ucsb-cs64-s18.github.io

On there, I will keep:

- Latest syllabus
- Class assignments
- Lecture slides (after I've given them)
- Interesting handouts and articles

Just in Case…


WHAT DID WE COVER IN CLASS LAST WEEK?
IT'S IN THE SYLLABUS.
WHAT'S YOUR LATE HOMEWORK POLICY?
IT'S IN THE SYLLABUS.
WHEN ARE YOUR OFFICE HOURS?
IT'S IN THE SYLLABUS.
JORGE CHAM © 2013
HOW WILL MY GRADE BE COMPUTED?
IT'S IN THE SYLLABUS
IT'S IN THE SYLLABUS
This message brought to you by every instructor that ever lived.
WWW.PHDCOMICS.COM

So… let's take a look at that syllabus…

Electronic version found at:

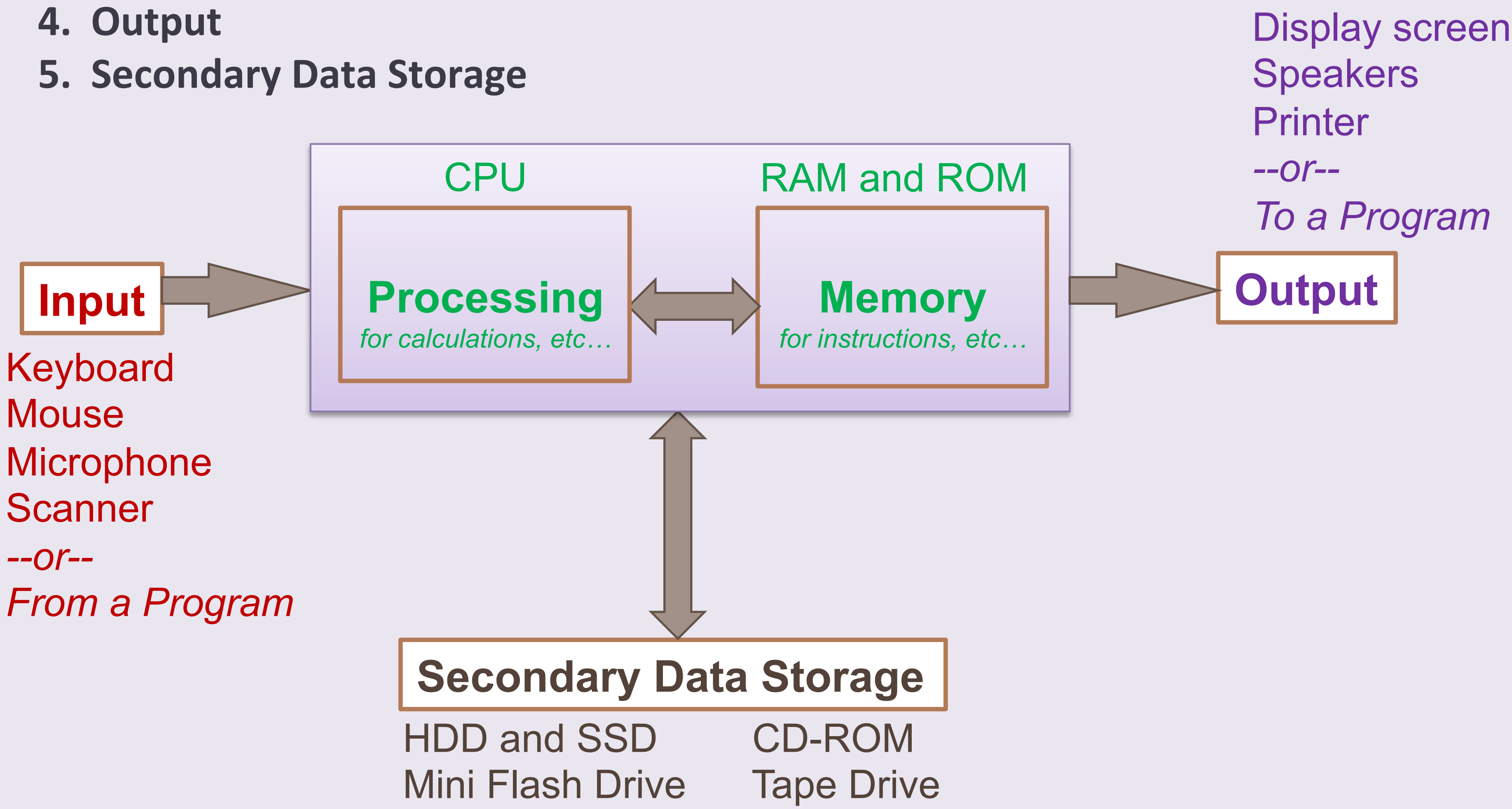
http://cs.ucsb.edu/~zmatni/syllabi/CS64S18_syllabus.pdf

A Simplified View of Modern Computer Architecture

a.k.a von Neumann Architecture

The 5 Main Components of a Computer:

1. **Processor**
2. **Memory**
3. **Input**
4. **Output**
5. **Secondary Data Storage**



Computer Memory

- Usually organized in two parts:
  - Address: ***Where*** can I find my data?
  - Data (payload): ***What*** is my data?

- The smallest representation of the data
  - A binary *bit* (“0”s and “1”s)
  - A common collection of bits is a *byte*
    - 8 bits = 1 byte
  - What is a *nibble*?
    - 4 bits = 1 nibble – not used as often…
  - **What is the minimum number of bits needed to convey an alphanumeric character? And WHY?**

What is the Most Basic Form of Computer Language?

- Binary *a.k.a* Base-2

- Expressing data AND instructions in either "1" or "0"
  - So,

"01010101 01000011 01010011 01000010 00100001 00100001"

could mean an *instruction* to "calculate 2 + 3"

Or it could mean an *integer number* (856,783,663,333)

Or it could mean a *string of 6 characters* ("UCSB!!")

Or other things…!

So… Like…
What Processes Stuff In A Computer?

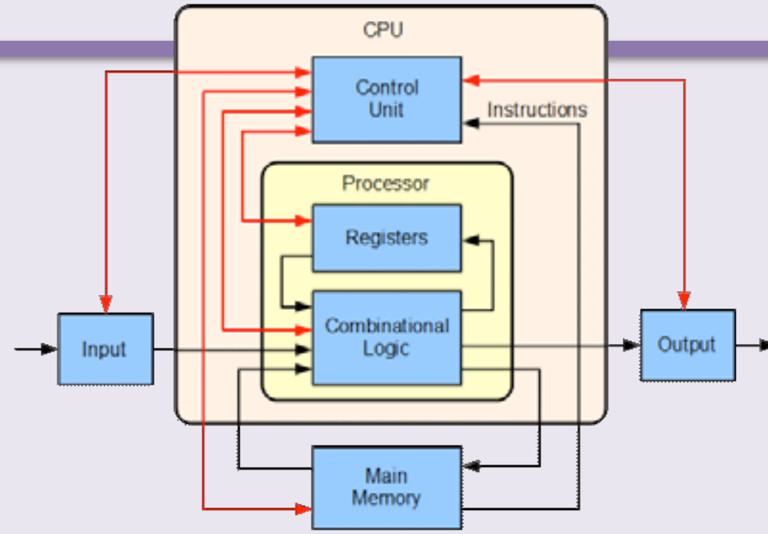
- The Central Processing Unit (CPU)
  - Executes program instructions

- Typical capabilities of CPU include:
  - Add
  - Subtract
  - Multiply
  - Divide
  - Move data from location to location

You can do just about anything with a computer with just these simple instructions!

Parts of the CPU

- The CPU is made up of 2 main parts:
  - The Arithmetic Logic Unit (ALU)
  - The Control Unit (CU)



- The ALU does the calculations in binary using "registers" (small RAM) and logic circuits

- The CU handles breaking down instructions into control codes for the ALU and memory

Image from wikimedia.org

The CPU's Fetch-Execute Cycle

- **Fetch** the next instruction
- **Decode** the instruction
- **Get data** if needed
- **Execute** the instruction
- ***Why is it a cycle???***

This is what happens inside a computer interacting with a program at the "lowest" level

Computer Languages and the F-E Cycle

- Instructions get executed in the CPU in machine language (i.e. all in “1”s and “0”s)
- Even *small* instructions, like “add 2 to 3 then multiply by 4”, need *multiple* cycles of the CPU to get executed fully
- But **THAT’S OK!** Because, typically, CPUs can run *many millions* of instructions per second

Computer Languages and the F-E Cycle

- But **THAT'S OK!** Because, typically,
  CPUs can run *many millions* of instructions per second

- In *low-level languages*, you need to spell those cycles out

- In *high-level languages*, you don't
  - 1 HLL statement, like "*x = c*(a + b)*" is enough to get the job done
  - This would translate into multiple statements in LLLs

"high level" vs. "low level" Programming

- High Level computer languages, like C++ or Java,
  are A LOT simpler to use!
- Uses syntax that "resembles" human language
- Easy to read and understand:

  x = c(a + b)* *vs.* *101000111010111*

- But, still… the CPU *NEEDS* machine language to do what it's supposed to do!

- So *SOMETHING* has to "translate" high level code
  into machine language…

Compilers

- *SOMETHING* has to "translate" high level code into machine language…
- Compilers are programs that do this
- Compilers are "language-specific"

Machine vs. Assembly Language

- **Machine language** is the actual 1s and 0s

Example:

```
10111101110111000001010101000
```

- **Assembly language** is one step above
  - Instructions are given **mnemonic codes** but still displayed one step at a time
  - Advantage? Better human readability

Example:

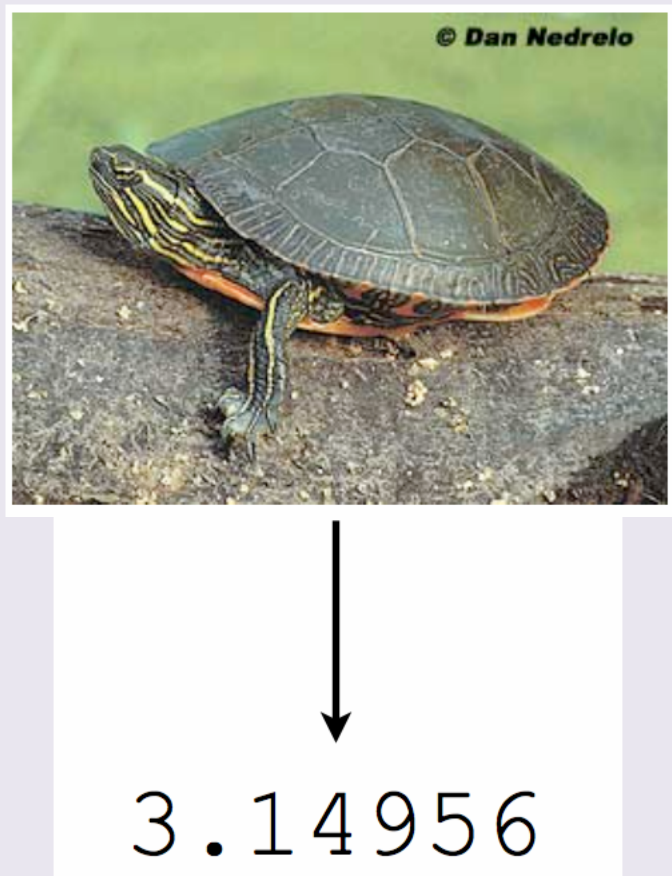
```
lw   $t0, 4($gp)    # fetch N
mult $t0, $t0, $t0  # multiply N by itself
                    # and store the result in N
```

```
int main(int argc, char** argv) {
  ...



}
```

3.14956

```
int main(int argc, char** argv) {
 ...
```

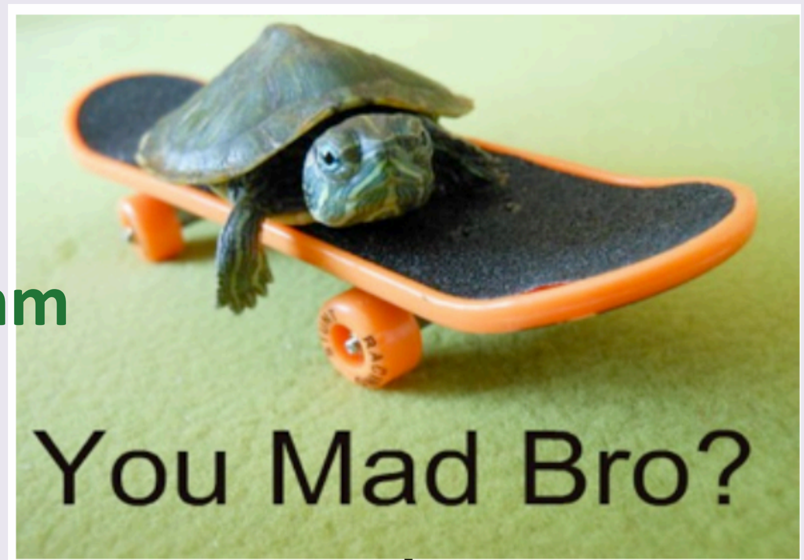
In reality… →




```
}
```

```c
int main(int argc, char** argv) {
 ...
}
```

With a more efficient algorithm →



3.14956

Why Can Programs be Slow?

- After all, isn't just as "simple" as
  1. getting an instruction,
  2. finding the value in memory,
  3. and doing stuff to it???

- Yes… except for the "simple" part…

- ***Ordering*** the instructions matters
  Where in memory the value is matters
  How instructions get "broken down" matters
  What order these get "pipelined" matters

The Point…

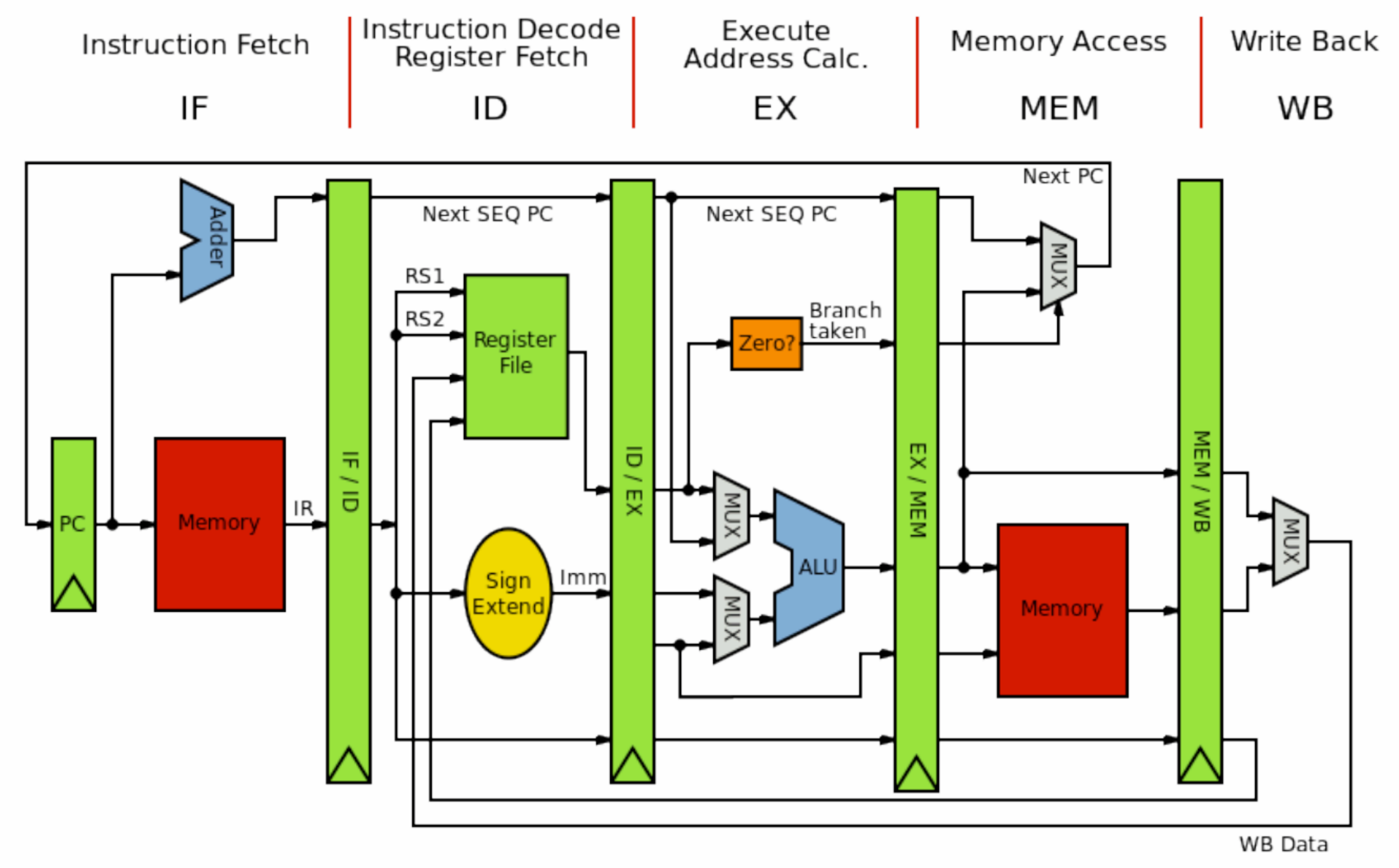
- If you really want performance, you need to know how the "magic" works

- If you want to write a naive compiler (CS 160), you need to know some low-level details of how the CPU does stuff
- If you want to write a *fast* compiler, you need to know tons of low-level details

So Why Digital Design?

- Because that’s where the “magic” happens
- Logical decisions are made with 1s and 0s
- Physically (*engineering-ly?*), this comes from electrical currents switching one way or the other
- These currents modify semiconducting material
  that obeys the laws of electromagnetism
  that is… physics…

So Why Digital Design?



Digital Design in this Course

- We will not go into “deep” dives with digital design in this course
  - For that, check out CS 154 (Computer Architecture) and also courses in ECE

- We will, however, delve deep enough to understand the ***fundamental*** workings of digital circuits and how they are used for ***computing purposes***.

COMPUTERS ARE DIGITAL MACHINES

THEY ARE DESIGNED TO COUNT IN…

2

Counting Numbers in Different Bases

- We "normally" count in 10s
  - Base 10: **decimal** numbers
  - We use 10 numerical symbols in Base 10: "0" thru "9"
- Computers count in 2s
  - Base 2: **binary** numbers
  - We use 2 numerical symbols in Base 2: "0" and "1"
- Represented with **1 bit** ($2^1 = 2$)

Counting Numbers in Different Bases

Other convenient bases in computer architecture:

- Base 8: **octal** numbers
  - Number symbols are 0 thru 7
  - Represented with **3 bits** ($2^3 = 8$)

- Base 16: **hexadecimal** numbers
  - Number symbols are 0 thru F:

    A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
  - Represented with **4 bits** ($2^4 = 16$)

- **Why are 4 bit representations convenient???**

What's in a Number?

642

What *is* that???

Well, what NUMERICAL BASE are you expressing it in?

Decimal Numbers

Counting **642** as 600 + 40 + 2
is counting in TENS (aka BASE 10) --- what we're used to

There are 6 HUNDREDS	6 x 100
There are 4 TENS	4 x 10
There are 2 ONES	2 x 1

6	4	2
100	10	1

642 = 600 + 40 + 2

Positional Notation in Decimal

Continuing with our example…

642 in base 10 *positional notation* is:

$$
\begin{aligned}
& 6 \times 10^2 = 6 \times 100 = 600 \\
+ & 4 \times 10^1 = 4 \times 10 = 40 \\
+ & 2 \times 10^0 = 2 \times 1 = 2 \quad = 642 \text{ in base 10}
\end{aligned}
$$

6	4	2
100	10	1

642 (base 10) = 600 + 40 + 2

Numerical Bases and Their Symbols

- How many “symbols” or “digits” do we use in Decimal (Base 10)?
- Base 2 (Binary)?
- Base 16 (Hexadecimal)?
- Base N?

Positional Notation

This is how you convert any base number into decimal!

What if "642" is expressed in the base of 13?

$$
\begin{aligned}
& 6 \times 13^2 &= \;& 6 \times 169 &= 1014 \\
+\; & 4 \times 13^1 &= \;& 4 \times 13 &= 52 \\
+\; & 2 \times 13^0 &= \;& 2 \times 1 &= 2
\end{aligned}
$$

6	4	2
13^2	13^1	13^0

$$642_{\text{(base 13)}} = 1014 + 52 + 2$$

$$= 1068_{\text{(base 10)}}$$

Positional Notation in Binary

11101 in base 2 *positional notation* is:

$$
\begin{aligned}
& 1 \times 2^4 = 1 \times 16 = 16 \\
+\ & 1 \times 2^3 = 1 \times 8 \quad = 8 \\
+\ & 1 \times 2^2 = 1 \times 4 \quad = 4 \\
+\ & 0 \times 2^1 = 1 \times 2 \quad = 0 \\
+\ & 1 \times 2^0 = 1 \times 1 \quad = 1
\end{aligned}
$$

So, **11101** in base 2 is 16 + 8 + 4+ 0 + 1 = **29** in base 10

YOUR TO-DOs

- Read Handout #1
- Assignment #1
  - Meet up in the lab on Monday
  - Do the lab assignment: setting up CSIL + exercises
  - You have to submit it using ***turnin***
  - Due on **Friday, 4/13, by 11:59 PM**

</LECTURE>