**Hardware**

CS 5 this week

- Logic gates
- Transistors / switches
- Bitwise functions
- Arithmetic
- 1-bit memory: flip-flops
- RAM
- Registers
- Registers

**Software**

- Python
  - How does Python function?
- Jotto Corner
  - AM guess
  - my guess
  - HS guess
  - my guess

- Machine Language
  - Assembly Language

**Farewell, Logisim!**

Now, where were we...?

Memory!

- Inside the 128 bits of memory...

- Speaking of data...

- Turing, et al.
- John Von Neumann
- Bell labs + beyond
Some memory is more equal than others...

<table>
<thead>
<tr>
<th>Registers</th>
<th>Main Memory</th>
<th>Disk Drive</th>
</tr>
</thead>
<tbody>
<tr>
<td>on the Central Processing Unit</td>
<td>(replaceable RAM)</td>
<td>magnetic storage</td>
</tr>
</tbody>
</table>

- 8 flip-flops are an 8-bit register
- 100 Registers of 64 bits each
  - ~ 10,000 bits
- ~ 100 billion bits

"640K ought to be enough for anybody" - Bill Gates (contested)

~ 100 billion bits
~ 42 trillion bits (or more)

memory from logic gates
"Leaky Bucket" capacitors
remagnetizing surfaces

Von Neumann Architecture

processing
CPU
central processing unit registers

Von Neumann bottleneck

program
RAM
random access memory locations

Why Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?

Von Neumann Architecture

70 years ago...

limited, fast registers + arithmetic
larger, slower memory + no computation

80x86

Why
Assembly?
Demo
of "in vivo" assembly language
and machine language

```c
int main() {
    int answer = 84;
    answer = answer + 42;
    printf("\nThe result is %d.\n\n", answer);
}
```

a very small C program

126 will be printed here...

Demo
of "in vivo" assembly-language

```c
int main() {
    int answer = 84;
    answer = answer + 42;
    printf("\nThe result is %d.\n\n", answer);
}
```

We'll change the machine-language instructions, not the C source code!

original

changed

Now, we'll see the "right" answer...

```c
int main() {
    int answer = 84;
    answer = answer + 42;
    printf("\nThe result is %d.\n\n", answer);
}
```

Example #1:

CPU
central processing unit registers

<table>
<thead>
<tr>
<th>register</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>r1</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>write</td>
</tr>
<tr>
<td></td>
<td>r2</td>
</tr>
</tbody>
</table>

RAM
random access memory locations

<table>
<thead>
<tr>
<th>memory location</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
</tr>
<tr>
<td>1</td>
<td>mul</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
</tr>
<tr>
<td>3</td>
<td>write</td>
</tr>
<tr>
<td>4</td>
<td>halt</td>
</tr>
</tbody>
</table>

Hmmm: Harvey mudd miniature machine

CPU
central processing unit registers

<table>
<thead>
<tr>
<th>register</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>16</td>
</tr>
</tbody>
</table>

RAM
random access memory locations

<table>
<thead>
<tr>
<th>memory location</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
</tr>
<tr>
<td>1</td>
<td>mul</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
</tr>
<tr>
<td>3</td>
<td>write</td>
</tr>
<tr>
<td>4</td>
<td>halt</td>
</tr>
</tbody>
</table>

Von Neumann bottleneck

Screen
6 (input)

256 memory locations

16 registers

2018 vs.

Hmmm: Harvey mudd miniature machine
**Hmmm** vs **2018**

**CPU**
- central processing unit
- registers

**RAM**
- random access memory locations

**General-purpose register r1**

**General-purpose register r2**

**256 memory locations**

**16 registers**

**Demo**

of assembly-language programming in **Hmmm**...

**Assembly Language**

read r1

writes from keyboard into reg r1

write r2

outputs reg r2 onto the screen

**setn r1 42**

`reg1 = 42`

you can replace 42 with anything from -128 to 127

**addn r1 -1**

`reg1 = reg1 - 1` 

a shortcut

This is why assignment is written R to L in Python!

**add r3 r1 r2**

reg3 = reg1 + reg2

**sub r3 r1 r2**

reg3 = reg1 - reg2

**mul r2 r1 r1**

reg2 = reg1 * reg1

**div r1 r1 r2**

reg1 = reg1 / reg2

ints only!
Could you write a Hmmm program that computes

\[ x^2 + 3x - 4 \]

or

\[ 1/\sqrt{x} \]

when would you want to?

Could you write a Hmmm program to compute

\[ x + 3x - 4 \]

or

\[ 1/x \]

when you’d want to!

Fast inverse square root

From Wikipedia, the free encyclopedia

Fast inverse square root (sometimes referred to as Fast InvSqrt() or by the hexadecimal constant 0x5f3759df) is a method of calculating \( x^{-1/2} \), the reciprocal (or multiplicative inverse) of a number.
**Real Assembly Languages**

Hmmm is a subset common to **all** real assembly languages.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLT</td>
<td>Enter halt state</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed divide</td>
</tr>
<tr>
<td>IMUL</td>
<td>Signed multiply</td>
</tr>
<tr>
<td>IN</td>
<td>Input from port</td>
</tr>
<tr>
<td>INC</td>
<td>Increment by 1</td>
</tr>
<tr>
<td>INT</td>
<td>Call to interrupt</td>
</tr>
<tr>
<td>INTO</td>
<td>Call to interrupt if overflow</td>
</tr>
<tr>
<td>RET</td>
<td>Return from interrupt</td>
</tr>
</tbody>
</table>

A few of the many basic processor instructions (Intel)

**Who** writes all of the assembly language that gets executed?

Two **more recent** Intel instructions (SSE4 subset)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPSADBW</td>
<td>Compute eight offset sums of absolute differences (i.e., $</td>
</tr>
<tr>
<td>PHMINPOSUW</td>
<td>Sets the bottom unsigned 16-bit word of the destination to the smallest unsigned 16-bit word in the source, and the next-from-bottom to the index of that word in the source.</td>
</tr>
</tbody>
</table>

Could you write a **Python program** that writes a **Hmmm program** that computes

$$x^2 + 3x - 4$$

or

$$1/\sqrt{x}$$

Yes – you already have! **much better!**

Is this all we need?

```
0 read r1
1 mul r2 r1 r1
2 add r2 r2 r1
3 write r2
4 halt
```

What’s missing here?

Why **couldn’t** we implement Python using only Hmmm assembly language up to this point?
For systems, innovation is adding an edge to *create a cycle*, not just an additional node.

Loops and *ifs*

We couldn't implement Python using Hmmm so far... *it's too linear!*

"straight-line code"

```
0: setn r1 42
1: write r1
2: addn r1 1
3: jumpn 1
4: halt
```

Jumps in Hmmm

*Conditional* jumps

- `j eqzn r1 42` IF r1 == 0 THEN jump to line number 42
- `j gtzn r1 42` IF r1 > 0 THEN jump to line number 42
- `j ltzn r1 42` IF r1 < 0 THEN jump to line number 42
- `j nezn r1 42` IF r1 != 0 THEN jump to line number 42

*Unconditional* jump

```
jumpn 42  Jump to program line # 42
```

**CPU**

central processing unit

```
r1
General-purpose register r1
```

```
r2
General-purpose register r2
```

**RAM**

random access memory

```
0: setn r1 42
1: write r1
2: addn r1 1
3: jumpn 1
4: halt
```

What would happen *IF*...

- we replace line 3's 1 with a 0?
- we replace line 3's 1 with a 2?
- we replace line 3's 1 with a 3?
- we replace line 3's 1 with a 4?
With an input of -6, what does this code write out?

factorial: the plan ...

```
fac(5) is 1*5*4*3*2*1
```

Try it!

Write an assembly-language program that reads a positive integer into r1. The program should compute the factorial of the input in r2. Once it’s computed, it should write out that factorial. Two lines are provided:

```assembly
read r1
setn r2 1
```

(1) What common function does this compute? 
**Hint:** try the inputs in both orders...

(2) Extra! How could you change only line 3 so that, if inputs $r_1$ and $r_2$ are equal, the program will ask for new inputs?

Extra! How few lines can you use here? (Fill the rest with nops...)

Try it!

Follow this Hmmm program. First run: use $r_1 = 42$ and $r_2 = 5$. Next run: use $r_1 = 5$ and $r_2 = 42$.

```
run 1
read r1
setn r2 1
```

```
run 2
read r1
setn r2 1
```

(1) What common function does this compute? 
**Hint:** try the inputs in both orders...

(2) Extra! How could you change only line 3 so that, if inputs $r_1$ and $r_2$ are equal, the program will ask for new inputs?

Extra! How few lines can you use here? (Fill the rest with nops...)

Try it!

Follow this Hmmm program. First run: use $r_1 = 42$ and $r_2 = 5$. Next run: use $r_1 = 5$ and $r_2 = 42$.

```
read r1
setn r2 1
```

(1) What common function does this compute? 
**Hint:** try the inputs in both orders...

(2) Extra! How could you change only line 3 so that, if inputs $r_1$ and $r_2$ are equal, the program will ask for new inputs?

Extra! How few lines can you use here? (Fill the rest with nops...)

Follow this assembly-language program from top to bottom. First use $r1 = 42$ and $r2 = 5$, then swap them on the next run:

<table>
<thead>
<tr>
<th>Run #1</th>
<th>Memory - RAM</th>
<th>Run #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>42</td>
<td>5</td>
</tr>
<tr>
<td>r2</td>
<td>5</td>
<td>42</td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 | read r1
1 | read r2
2 | sub r3 r1 r2
3 | nop
4 | jgtzn r3 7
5 | write r1
6 | jumpn 8
7 | write r2
8 | halt

1. What function does this program compute in general?
2. **Extra!** How could you change only line 3 so that, if the original two inputs were equal, the program asked for new inputs?

This week in lab:

**Randohmmmm Numbers...**

where you'll write your own random number generator...

...in Hmmmm assembly language

See you there!
Name(s) __________________________

**CPU**  
central processing unit

| r1 | General-purpose register r1 |
| 0  | 100 |

| r2 | General-purpose register r2 |
| 1  | setn r2 7 |
| 2  | mod r4 r1 r2 |
| 3  | div r3 r1 r4 |
| 4  | sub r3 r3 r2 |
| 5  | addn r3 -1 |
| 6  | write r3 |

**RAM**  
random access memory

| 0  | read r1 |
| 1  | | r1 = 100 |
| 2  | | |
| 3  | | |
| 4  | | |
| 5  | | |
| 6  | | |

Extra! Change only line 4’s instruction to create an output of 0 or 6 or 349 instead?

Hmmm...!?