Logisim's farewell?
Hmmm

4 Hmmm problems + 2 loop problems due Mon. 10/26

Fall break is also a CS hw break...

Equilibrium punctuaters

Grace Hopper

Bell labs

Von Neumann

Turing, et al.

Hardware

1-bit memory: flip-flops

logic gates

bitwise functions

arithmetic

registers

RAM

transistors / switches

Software

Python

How does Python function?

CS 5 this week

Python

Fall break is also a CS hw break...
This week

Hardware

CS 5 this week

logic gates

transistors / switches

bitwise functions

arithmetic

1-bit memory: flip-flops

registers

RAM

Hmmm

4 Hmmm problems + 2 loop problems due Mon. 10/26

Python

Fall break is also a CS hw break…

Turing, et al.

Equilibrium punctuaters

Von Neumann

Bell labs

Grace Hopper

How does Python function?

Software

Don Chamberlain (Thursday ~ 25 min)

"50 years of data"
Graders' thoughts...
Now, where were we...?

Inside the 12nGbits of memory...

Memory!
Some memory is more equal than others...

**Registers**

on the Central Processing Unit

8 flip-flops are an 8-bit register

100 Registers of 64 bits each

\(\sim 10,000\) bits

**Main Memory**

(replaceable RAM)

10 GB memory

\(\sim 100\) billion bits

**Disk Drive**

magnetic storage

4 TB drive

\(\sim 42\) trillion bits (or more)
Some memory is more equal than others...

Registers
on the Central Processing Unit

Main Memory
(replaceable RAM)

Disk Drive
magnetic storage

8 flip-flops are an 8-bit register

100 Registers of 64 bits each
~ 10,000 bits

10 GB memory
~ 100 billion bits

4 TB drive
~ 42 trillion bits (or more)

memory from logic gates

"Leaky Bucket" capacitors

remagnetizing surfaces

"640K ought to be enough for anybody"
- Bill Gates (contested)
### Registers
on the Central Processing Unit

- 8 flip-flops are an 8-bit register
- 100 Registers of 64 bits each
- \( \sim 10,000 \) bits

### Main Memory
(replaceable RAM)

- 10 GB memory
- \( \sim 100 \) billion bits

### Disk Drive
magnetic storage

- 4 TB drive
- \( \sim 42 \) trillion bits (or more)

### Price

- \( \sim $100 \)

### Time

- 1 clock cycle
  - \( 10^{-9} \) sec
- 100 cycles
  - \( 10^{-7} \) sec
- \( 10^7 \) cycles
  - \( 10^{-2} \) sec

**If a clock cycle == 1 minute**

- 1 min
- 1.5 hours
Some memory is more equal than others...

Registers
on the Central Processing Unit

Main Memory
(replaceable RAM)

Disk Drive
magnetic storage

8 flip-flops are an 8-bit register

100 Registers of 64 bits each
~ 10,000 bits

10 GB memory
~ 100 billion bits

4 TB drive
~ 42 trillion bits (or more)

Price
~ $100
~ $100
~ $100

Time
1 clock cycle
10^{-9} sec
100 cycles
10^{-7} sec
10^7 cycles
10^{-2} sec

If a clock cycle == 1 minute
1 min
1.5 hours
19 YEARS
Some memory is more equal than others...

**Registers**
on the Central Processing Unit

- 8 flip-flops are an 8-bit register
- 100 Registers of 64 bits each
  - \( \sim 10,000 \) bits

**Main Memory**
(replaceable RAM)

- 10 GB memory
  - \( \sim 100 \) billion bits

**Disk Drive**
magnetic storage

- 4 TB drive
  - \( \sim 42 \) trillion bits (or more)

+ are fetched and executed 1 instruction at a time here...

running programs are stored here...

"Off" data is saved way out here...

If a clock cycle \( == 1 \) minute

<table>
<thead>
<tr>
<th>Time</th>
<th>1 min</th>
<th>1.5 hours</th>
<th>19 YEARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>~ $100</td>
<td>~ $100</td>
<td>~ $100</td>
</tr>
<tr>
<td>Registers</td>
<td>100 Registers of 64 bits each</td>
<td>100 Registers of 64 bits each</td>
<td>100 Registers of 64 bits each</td>
</tr>
<tr>
<td>Main Memory</td>
<td>10 GB memory</td>
<td>10 GB memory</td>
<td>10 GB memory</td>
</tr>
<tr>
<td>Disk Drive</td>
<td>4 TB drive</td>
<td>4 TB drive</td>
<td>4 TB drive</td>
</tr>
</tbody>
</table>

10 GB memory

If a clock cycle \( == 1 \) minute
How do we execute *sequences* of operations?

processor → CPU → RAM → live memory

memory locations (RAM)

Inside the 12nGbits of memory...
How do we execute *sequences* of operations?

**CPU**
- Stores all instructions and almost all data.
- Sends next instruction to the CPU.
- Sends next instruction to the CPU.
- The instruction's bits select which circuit to use.

**RAM**
- Live memory.
- Sends next instruction to the CPU.

**Processor**
- Runs 1 instruction and sends back results for storage, if requested.

**Multiplier**
- Inside the 12nGbits of memory.

**Divider**
70 years ago...

Von Neumann architecture
From Wikipedia, the free encyclopedia

Jon V.N.

limited, fast **registers** + arithmetic

larger, slower **memory** + *no* computation

processing

**CPU**

central processing unit **registers**

execute

**fetch**

**RAM**

random access memory locations

stored program
limited, fast **registers** + arithmetic

larger, slower **memory** + *no* computation

**70 years later...**

Jon V.N.
Von Neumann Architecture

Programs are stored in memory in **machine language**

**CPU**
- central processing unit **registers**

**RAM**
- random access memory locations

---

**r1**
- General-purpose register, r1

**r2**
- General-purpose register, r2

---

Von Neumann bottleneck

---

```
0000 0001 0000 0001
1000 0010 0001 0001
0110 0010 0010 0001
0000 0010 0000 0010
0000 0000 0000 0000
(all bits)
```
Von Neumann Architecture

Processing

CPU

central processing unit registers

RAM

random access memory locations

Von Neumann bottleneck

program

0
read r1

1
mul r2 r1 r1

2
add r2 r2 r1

3
write r2

4
halt

5

6
"mnemonics" instead of bits

Assembly language is human-readable machine language

General-purpose register, r1

General-purpose register, r1

Human readable? I doubt it!
Demo

of "in vivo" assembly-language
Example #1:

**CPU**

central processing unit *registers*

r1
General-purpose register r1

r2
General-purpose register r2

**RAM**

random access memory locations

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

Screen

6 (input)
**Hmmm:** Harvey Mudd Miniature Machine

**CPU**
- central processing unit registers

**RAM**
- random access memory locations

16 registers

256 memory locations

0: read r1
1: mul r2 r1
2: write r2
3: write r2
4: halt
Hmmm vs 2016

CPU (central processing unit) registers

Von Neumann bottleneck

RAM (random access memory locations)

r1 General-purpose register r1

r2 General-purpose register r2

16 registers

2016 Intel: ~100 registers

2016: ~10,000,000,000 mem loc's

read r1

mul r2 r1 r1

write r2

halt
Demo

of assembly-language programming in Hmm...
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>Stop!</td>
<td></td>
</tr>
<tr>
<td>read rX</td>
<td>Place user input in register rX</td>
<td></td>
</tr>
<tr>
<td>write rX</td>
<td>Print contents of register rX</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>Do nothing</td>
<td></td>
</tr>
</tbody>
</table>

### Setting register data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>setn rX N</td>
<td>Set register rX equal to the integer N (-128 to +127)</td>
<td></td>
</tr>
<tr>
<td>addn rX N</td>
<td>Add integer N (-128 to +127) to register rX</td>
<td></td>
</tr>
<tr>
<td>copy rX rY</td>
<td>Set rX = rY</td>
<td></td>
</tr>
</tbody>
</table>

### Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add rX rY rZ</td>
<td>Set rX = rY + rZ</td>
<td></td>
</tr>
<tr>
<td>sub rX rY rZ</td>
<td>Set rX = rY - rZ</td>
<td></td>
</tr>
<tr>
<td>neg rX rY</td>
<td>Set rX = -rY</td>
<td></td>
</tr>
<tr>
<td>mul rX rY rZ</td>
<td>Set rX = rY * rZ</td>
<td></td>
</tr>
<tr>
<td>div rX rY rZ</td>
<td>Set rX = rY / rZ (integer division; no remainder)</td>
<td></td>
</tr>
<tr>
<td>mod rX rY rZ</td>
<td>Set rX = rY % rZ (returns the remainder of integer division)</td>
<td></td>
</tr>
</tbody>
</table>

### Jumps!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jumpn N</td>
<td>Set program counter to address N</td>
<td></td>
</tr>
<tr>
<td>jump rX</td>
<td>Set program counter to address in rX</td>
<td>jump</td>
</tr>
<tr>
<td>jeqn rX N</td>
<td>If rX == 0, then jump to line N</td>
<td>jegz</td>
</tr>
<tr>
<td>jnezn rX N</td>
<td>If rX != 0, then jump to line N</td>
<td>jnez</td>
</tr>
<tr>
<td>jgtzn rX N</td>
<td>If rX &gt; 0, then jump to line N</td>
<td>jgtz</td>
</tr>
<tr>
<td>jltzn rX N</td>
<td>If rX &lt; 0, then jump to line N</td>
<td>jltz</td>
</tr>
<tr>
<td>call rX N</td>
<td>Copy the next address into rX and then jump to mem. addr. N</td>
<td>call</td>
</tr>
</tbody>
</table>

### Interacting with memory (RAM)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loadn rX N</td>
<td>Load register rX with the contents of memory address N</td>
<td></td>
</tr>
<tr>
<td>storen rX N</td>
<td>Store contents of register rX into memory address N</td>
<td></td>
</tr>
<tr>
<td>loadr rX rY</td>
<td>Load register rX with data from the address location held in reg. rY</td>
<td>loadi, load</td>
</tr>
<tr>
<td>storer rX rY</td>
<td>Store contents of register rX into memory address held in reg. rY</td>
<td>storei, store</td>
</tr>
</tbody>
</table>
Assembly Language

**read r1**
reads from keyboard into reg1

**write r2**
outputs reg2 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

**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

This is why they're written R to L in Python!

ought to be called register language
Extra! Here, the input r1 was 100. As a challenge, can you find any inputs (r1) that yield an output of 100? (They do exist!)
Try this on the back page first!

**CPU**
central processing unit

- **r1**: 100
  - General-purpose register r1

- **r2**: 7
  - General-purpose register r2

- **r3**: 
  - General-purpose register r3

- **r4**: 
  - General-purpose register r4

**RAM**
random access memory

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read r1</td>
<td>read r1 from RAM into r1 (r1 = 100)</td>
</tr>
<tr>
<td>1</td>
<td>setn r2 7</td>
<td>set r2 to 7</td>
</tr>
<tr>
<td>2</td>
<td>mod r4 r1 r2</td>
<td>r4 = r1 % r2</td>
</tr>
<tr>
<td>3</td>
<td>div r3 r1 r4</td>
<td>r3 = r1 / r4</td>
</tr>
<tr>
<td>4</td>
<td>sub r3 r3 r2</td>
<td>r3 = r3 - r2</td>
</tr>
<tr>
<td>5</td>
<td>addn r3 -1</td>
<td>r3 = r3 + -1</td>
</tr>
<tr>
<td>6</td>
<td>write r3</td>
<td>write r3 to RAM</td>
</tr>
<tr>
<td>7</td>
<td>halt</td>
<td>program halt</td>
</tr>
</tbody>
</table>

**Python**

```python
r1 = 100
r2 = 7
r4 = r1 % r2
r3 = r1 / r4
r3 = r3 - r2
r3 = r3 + -1
print(r3)
```

**Extra!** Here, the **input** r1 was 100. Can you find any inputs (r1) that yield an **output** of 100? (They do exist!) [325, 544, + one more...]

**Screen**

- **100** (input)
- **(output)**

---

**Quiz**

- **CPU**
  - General-purpose register r1
  - General-purpose register r2
  - General-purpose register r3
  - General-purpose register r4

- **RAM**
  - 0: read r1
  - 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
  - 7: halt
Could you write a Hmmm program to compute

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

or

\[ 1/\sqrt{x} \]

when would you want to?
Could you write a Hmmm program to compute \( x + 3x - 4 \) or \( \frac{1}{\sqrt{x}} \)?

1/\( \sqrt{x} \)

when you'd want to!
Could you write a *Python* program to write a *Hmmm program* to compute

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

or

\[ \frac{1}{\sqrt{x}} \]

?  

*much better!*
Why Assembly?

Skating uphill like this is amazing. Years of gliding downhill and pushing uphill, and now suddenly it's gliding both ways.

It's like going from C to Python. You don't realize how much time you were spending on the boring parts until you don't have to do them anymore.

But coding C or assembly makes you a better programmer. Maybe the boring parts build character.

Unsure vehicles, hills, and philosophy go hand in hand.
**Real Assembly Languages**

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

A few of the many basic processor instructions (Intel):

<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>IRET</td>
<td>Return from interrupt</td>
</tr>
</tbody>
</table>

**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>
Is this enough?

What's missing?

Why *couldn't* we implement Python using our Hmmm assembly language up to this point?
For systems, a face-lift is to add an edge that *creates a cycle*, not just an additional node.
Loops and if's

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
```
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?
## Jumps in Hmmm

### Conditional jumps

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jeqzn r1 42</td>
<td>IF r1 == 0 THEN jump to line number 42</td>
</tr>
<tr>
<td>jgtzn r1 42</td>
<td>IF r1 &gt; 0 THEN jump to line number 42</td>
</tr>
<tr>
<td>jltzn r1 42</td>
<td>IF r1 &lt; 0 THEN jump to line number 42</td>
</tr>
<tr>
<td>jnezn r1 42</td>
<td>IF r1 != 0 THEN jump to line number 42</td>
</tr>
</tbody>
</table>

### Unconditional jump

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jumpn 42</td>
<td>Jump to program line # 42</td>
</tr>
</tbody>
</table>

### Indirect jump

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jumper r1</td>
<td>Jump to the line# stored in r1</td>
</tr>
</tbody>
</table>

This is making me jumpy!
Jumps in Hmmm

**Conditional jumps**

- `jeqzn r1`: if equal to zero... THEN jump to line number 42
- `jgtzn r1`: if greater than 0... THEN jump to line number 42
- `jltzn r1`: if less than zero... THEN jump to line number 42
- `jnezn r1`: if not equal to 0... THEN jump to line number 42

**Unconditional jump**

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

**Indirect jump**

- `jumprr r1`: Jump to the line# stored in r1

This is making me jumpy!
### System instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>Stop!</td>
<td></td>
</tr>
<tr>
<td>read rX</td>
<td>Place user input in register rX</td>
<td></td>
</tr>
<tr>
<td>write rX</td>
<td>Print contents of register rX</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>Do nothing</td>
<td></td>
</tr>
</tbody>
</table>

### Setting register data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>setn rX N</td>
<td>Set register rX equal to the integer N (-128 to +127)</td>
<td></td>
</tr>
<tr>
<td>addn rX N</td>
<td>Add integer N (-128 to 127) to register rX</td>
<td></td>
</tr>
<tr>
<td>copy rX rY</td>
<td>Set rX = rY</td>
<td></td>
</tr>
</tbody>
</table>

### Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add rX rY rZ</td>
<td>Set rX = rY + rZ</td>
<td></td>
</tr>
<tr>
<td>sub rX rY rZ</td>
<td>Set rX = rY - rZ</td>
<td></td>
</tr>
<tr>
<td>neg rX rY</td>
<td>Set rX = -rY</td>
<td></td>
</tr>
<tr>
<td>mul rX rY rZ</td>
<td>Set rX = rY * rZ</td>
<td></td>
</tr>
<tr>
<td>div rX rY rZ</td>
<td>Set rX = rY / rZ (integer division; no remainder)</td>
<td></td>
</tr>
<tr>
<td>mod rX rY rZ</td>
<td>Set rX = rY % rZ (returns the remainder of integer division)</td>
<td></td>
</tr>
</tbody>
</table>

### Jumps!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jumpn N</td>
<td>Set program counter to address N</td>
<td></td>
</tr>
<tr>
<td>jump rX</td>
<td>Set program counter to address in rX</td>
<td></td>
</tr>
<tr>
<td>jegz rX N</td>
<td>If rX == 0, then jump to line N</td>
<td></td>
</tr>
<tr>
<td>jnez rX N</td>
<td>If rX != 0, then jump to line N</td>
<td></td>
</tr>
<tr>
<td>jgtz rX N</td>
<td>If rX &gt; 0, then jump to line N</td>
<td></td>
</tr>
<tr>
<td>jltz rX N</td>
<td>If rX &lt; 0, then jump to line N</td>
<td></td>
</tr>
<tr>
<td>call rX N</td>
<td>Copy the next address into rX and then jump to mem. addr. N</td>
<td></td>
</tr>
</tbody>
</table>

### Interacting with memory (RAM)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loadn rX N</td>
<td>Load register rX with the contents of memory address N</td>
<td></td>
</tr>
<tr>
<td>storen rX N</td>
<td>Store contents of register rX into memory address N</td>
<td></td>
</tr>
<tr>
<td>loadr rX rY</td>
<td>Load register rX with data from the address location held in reg. rY</td>
<td></td>
</tr>
<tr>
<td>storerr rX rY</td>
<td>Store contents of register rX into memory address held in reg. rY</td>
<td></td>
</tr>
</tbody>
</table>

*At www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html*
With an input of -6, what does this code write out?
Try it!

I think this language has injured my craniummm!

Follow this Hmm program.
First run: use $r1 = 42$ and $r2 = 5$.
Next run: use $r1 = 5$ and $r2 = 42$.

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:

```
read r1
read r2
sub r3 r1 r2
nop
jgtzn r3 7
write r1
jumpn 8
write r2
halt
```

(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 $r1$ and $r2$ are **equal**, the program will ask for new inputs?

**Hint:** On line 2, could you write a test that checks if the factorial is finished; if it's not, compute one piece and then jump back!

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

let \( r_1 \) be the input and the "counter"

let \( r_2 \) become the output

\( r_1 \rightarrow 5 \) input

\( r_2 \rightarrow 1 \) output (to be)
Follow this assembly-language program from top to bottom. First use $r1 = 42$ and $r2 = 5$, then swap them on the next run:

(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?
factorial: the *plan* ...

let \( r_1 \) be the input and the "counter"

let \( r_2 \) become the output

output (to be)
**one factorial code**

### Registers - CPU

- **r1**: input
- **r2**: result – so far
- **r3**: not needed, but OK to use!

### Memory - RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read r1</td>
</tr>
<tr>
<td>1</td>
<td>setn r2 1</td>
</tr>
<tr>
<td>2</td>
<td>jeqzn r1 8</td>
</tr>
<tr>
<td>3</td>
<td>mul r2 r2 r1</td>
</tr>
<tr>
<td>4</td>
<td>addn r1 -1</td>
</tr>
<tr>
<td>5</td>
<td>jumpn 2</td>
</tr>
<tr>
<td>6</td>
<td>nop</td>
</tr>
<tr>
<td>7</td>
<td>nop</td>
</tr>
<tr>
<td>8</td>
<td>write r2</td>
</tr>
<tr>
<td>9</td>
<td>halt</td>
</tr>
</tbody>
</table>

- **space for future expansion!**
This week in lab:

Randohmmmm Numbers...

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

See you there!