Hardware

transistors / switches

logic gates

bitwise functions

arithmetic

1-bit memory: flip-flops

registers

RAM

4 Hmm problems
+ 1 loop problem
due Mon. 3/11

Hmmm

Python

How does Python function?

Machine Language

Assembly Language

Software

CS 5 this week

RAM
Hardware

transistors / switches
logic gates
bitwise functions
arithmetic
1-bit memory: flip-flops
registers
RAM

CS 5 this week

Software

How does Python function?

4 Hmmm problems + 1 loop problem due Mon. 3/11

Grace Hopper

John Von Neumann

Bell labs + beyond

Turing, et al.

Python

Hmmm

How does Python function?
Hardware

CS 5 this week

Logic gates, transistors, switches

Bitwise functions, arithmetic

1-bit memory: flip-flops, registers, RAM

Hmmm 4 problems + 1 loop problem due Mon. 10/23

Python

Fall break is also a CS hw break…

Software

Turing, et al.

Grace Hopper

John Von Neumann

Bell labs + beyond
Farewell, Logisim!
The flip-flop

D data

"strobe"

"we are ready to handle the data"

the flip-flop's diagram

1 bit of memory!

Q is 1 bit of storage

The flip-flop's diagram and its functions are explained in detail. The diagram shows how data (D) and the strobe signal are handled to store 1 bit of memory (Q). The set and reset controls are also shown, indicating the system's ability to keep a value.
The flip-flop

inputs

D data

"strobe"

AND

But there's a LOT more than 1 bit of memory...!

1 bit of memory!
Random Access Memory

Extra this week:  \textit{Design 12nGbits of RAM}

\textbf{Simplified Prototype for Accessing Memory}

12 bits of RAM

Inputs
- 2 data address bits
- 3 data input bits
- write enable line
- read enable line

Which row?

3 rows of 4 bits

Outputs
- 3 data output bits

\begin{itemize}
  \item 3 bits stored at location 00
  \item 3 bits stored at location 01
  \item 3 bits stored at location 10
  \item 3 bits stored at location 11
\end{itemize}
Ex Cr

0. Make data input bits 101
1. Give 01 to the decoder (the 1 goes on)
2. Make the "Write Enable" high
3. How do the * AND gates make sure that the value does go into memory location #1?
4. How do the * AND gates make sure that the value does NOT go into memory location #0?

STORE the value 5 into mem. loc. #1

0 \rightarrow 2; 4 \rightarrow 3

two other memory lines and their flip-flops are not drawn
0. Suppose 101 is in Location #1
1. Give 01 to the decoder (the 1 goes on)
2. Make the "Read Enable" high
3. Which gates will ensure bits from memory location #1 are read out?
4. Which gates will ensure bits from memory location #0 are not read out?
5. Draw where the "Read Enable" wire should go!

Two other memory lines and their flip-flops are not drawn.
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
  - ~10,000 bits

**Main Memory**
- (replaceable RAM)
- 10 GB memory
  - ~100 billion bits

**Disk Drive**
- magnetic storage
- 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)
Some memory is more equal than others...

**Registers**
on the Central Processing Unit

<table>
<thead>
<tr>
<th>8 flip-flops are an 8-bit <strong>register</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Register Image" /></td>
</tr>
</tbody>
</table>

100 Registers of 64 bits each
~ 10,000 bits

**Main Memory**
(replaceable RAM)

<table>
<thead>
<tr>
<th>10 GB memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2.png" alt="Main Memory Image" /></td>
</tr>
</tbody>
</table>

~ 100 billion bits

**Disk Drive**
magnetic storage

<table>
<thead>
<tr>
<th>4 TB drive</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Disk Drive Image" /></td>
</tr>
</tbody>
</table>

~ 42 trillion bits (or more)

<table>
<thead>
<tr>
<th><strong>Price</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>~$100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Time</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 clock cycle</td>
</tr>
<tr>
<td>10^{-9} sec</td>
</tr>
</tbody>
</table>

| 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
- ~ 10,000 bits

**Main Memory**
(replaceable RAM)

- 10 GB memory
- ~ 100 billion bits

**Disk Drive**
magnetic storage

- 4 TB drive
- ~ 42 trillion bits (or more)

---

programs 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

- 1 min
- 1.5 hours
- 19 YEARS
How do we execute sequences of operations?

Processor

CPU

stores all instructions and almost all data

RAM

live memory

sends next instruction to the CPU...

memory locations (RAM)

sends next instruction to the CPU...

inside the 12nGbits of memory...

sends next instruction to the CPU...

multiplier

divider

the instruction's bits select which circuit to use...

runs 1 instruction and sends back results for storage, if requested...
70 years ago...

Von Neumann architecture
From Wikipedia, the free encyclopedia

limited, fast registers + arithmetic

larger, slower memory + no computation
70 years later...

Von Neumann architecture
From Wikipedia, the free encyclopedia

Jon V.N.

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

central processing unit
CPU
random access memory locations
RAM

processing
fetch
execute

stored program

70 years later…
Programs are stored in memory in *machine language*.
Von Neumann Architecture

Programs are shown in assembly language.

Central processing unit registers

CPU

Von Neumann bottleneck

RAM

random access memory locations

r1

General-purpose register, r1

r2

General-purpose register, r2

Processing

"mnemonics" instead of bits

0 0000 0000 0000 0000
1 1000 0010 0001 0001
2 0110 0010 0010 0001
3 0000 0010 0000 0010
4 0000 0000 0000 0000
5
6

0000 0001 0000 0001
1000 0010 0001 0001
0110 0010 0010 0001
0000 0010 0000 0010
0000 0000 0000 0000

read r1
mul r2 r1 r1
add r2 r2 r1
write r2
halt
Von Neumann Architecture

Programs are shown in assembly language

CPU

central processing unit registers

RAM

random access memory locations

Von Neumann bottleneck

r1

General-purpose register, r1

r2

General-purpose register, r2

The \textbf{mul} instruction

\begin{align*}
0 & 0000 0001 0000 0001 \\
1 & 1000 0010 0001 0001 \\
2 & 0110 0010 0010 0001 \\
3 & 0000 0010 0000 0010 \\
4 & 0000 0000 0000 0000 \\
5 & \text{mul r2 r1 r1} \\
6 & \text{add r2 r2 r1} \\
7 & \text{write r2} \\
8 & \text{halt}
\end{align*}

"mnemonics" instead of bits
**Von Neumann Architecture**

Diagram showing the relationship between the CPU (central processing unit) and RAM (random access memory). The diagram includes general-purpose registers `r1` and `r2`.

**Assembly language is human-readable machine language**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read r1</td>
</tr>
<tr>
<td>1</td>
<td>mul r2 r1 r1</td>
</tr>
<tr>
<td>2</td>
<td>add r2 r2 r1</td>
</tr>
<tr>
<td>3</td>
<td>write r2</td>
</tr>
<tr>
<td>4</td>
<td>halt</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

"mnemonics" instead of bits

Human readable? I doubt it!

---

**Processing**

- CPU (central processing unit) with registers
- RAM (random access memory locations)

**Program**

- Assembly language is human-readable

**Von Neumann bottleneck**

- CPU reads from RAM
- CPU writes to RAM
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.

Yeah... but it depends how you want to spend your life. See, my philosophy is—

Unsafe vehicles, hills, and philosophy go hand in hand.
Demo

of "in vivo" assembly language
and machine language
int main()
{
    int answer = 84;
    answer = answer + 42;

    printf("The result is \%d.\n\n", answer);
}

126 will be printed here...
Demo

of "in vivo" assembly-language

int main()
{
    int answer = 84;
    answer = answer + 42;

    printf("\nThe result is %d.\n\n", answer);
}

ADD instruction

126 will be printed here...

84 == 01010100
42 == 00101010
ADD == 01000101

2a
in hex

45
in hex
Demo of "in vivo" assembly-language

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

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

```
int main()
{
    int answer = 84;
    answer = answer + 42;

    printf("\nThe result is \%d.\n\n", answer);
}
```

```
84 == 01010100
42 == 00101010
ADD == 01000101

2a in hex
45 in hex

original

SUB == 01101101
6d in hex

changed

Now, we'll see the "right" answer...
Example #1:

- **CPU**
  - Central processing unit (registers)
  - General-purpose register r1
  - 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
```

**Von Neumann bottleneck**

**Screen**

`6` (input)
Hmmm: Harvey mudd miniature machine

CPU

central processing unit *registers*

Von Neumann bottleneck

RAM

random access memory locations

**CPU**

*General-purpose register r1*

**RAM**

*General-purpose register r2*

16 registers

256 memory locations

read r1

mul r2 r1

halt

vs. 2018?
Hmmm vs 2018

CPU

Von Neumann bottleneck

RAM

central processing unit *registers*

random access memory locations

General-purpose register r1

General-purpose register r2

16 registers

256 memory locations

2018 Intel: *50-100* registers per core

2018: *~10,000,000,000* mem loc's
Demo

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

hw7 is chin-scratchingly challenging!
### System instructions

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

### Arithmetic

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

### Jumps!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>jumpn N</td>
<td>Set program counter to address N</td>
<td></td>
</tr>
<tr>
<td>jmpn rX</td>
<td>Set program counter to address in rX</td>
<td></td>
</tr>
<tr>
<td>jeqzn rX N</td>
<td>If rX == 0, then jump to line N</td>
<td>jump</td>
</tr>
<tr>
<td>jeqz rX</td>
<td>If rX == 0, then jump to line N</td>
<td></td>
</tr>
<tr>
<td>jnezn rX N</td>
<td>If rX != 0, then jump to line N</td>
<td>jnez</td>
</tr>
<tr>
<td>jnez rX</td>
<td>If rX != 0, then jump to line N</td>
<td></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>jgtz rX</td>
<td>If rX &gt; 0, then jump to line N</td>
<td></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>jltz rX</td>
<td>If rX &lt; 0, then jump to line N</td>
<td></td>
</tr>
<tr>
<td>calln 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>
</tr>
</thead>
<tbody>
<tr>
<td>pushr rX rY</td>
<td>Store contents of register rX onto stack pointed to by reg. rY</td>
</tr>
<tr>
<td>popr rX rY</td>
<td>Load contents of register rX from stack pointed to by reg. rY</td>
</tr>
<tr>
<td>loadn rX N</td>
<td>Load register rX with the contents of memory address N</td>
</tr>
<tr>
<td>storen rX N</td>
<td>Store contents of register rX into memory address N</td>
</tr>
<tr>
<td>loadr rX rY</td>
<td>Load register rX with data from the address location held in reg. rY</td>
</tr>
<tr>
<td>storer rX rY</td>
<td>Store contents of register rX into memory address held in reg. rY</td>
</tr>
</tbody>
</table>
Assembly Language

**read** r1  
reads 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!**
**Screen**

100 (input)

42 (output)

---

**CPU**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>100</td>
</tr>
<tr>
<td>r2</td>
<td>7</td>
</tr>
<tr>
<td>r3</td>
<td>504342</td>
</tr>
<tr>
<td>r4</td>
<td>2</td>
</tr>
</tbody>
</table>

**RAM**

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read r1</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>setn r2 7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mod r4 r1%r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>div r3 r1/r4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sub r3 r3-r2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>addn r3 -1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>write r3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>halt</td>
<td></td>
</tr>
</tbody>
</table>

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

**Hmmm...!?**

General-purpose register r1

General-purpose register r2

General-purpose register r3

General-purpose register r4

**Note:**
- `r1 = 100`
- `r1 = 100`
Try this on the back page first!

CPU
- r1: General-purpose register r1
  - Value: 100
- r2: General-purpose register r2
  - Value: 7
- r3: General-purpose register r3
- r4: General-purpose register r4

RAM
- read r1
  - Value: 100
- setn r2 7
  - Value: 7
- mod r4 r1 r2
  - Value: r1 % r2
- div r3 r1 r4
  - Value: r1 // r4
- sub r3 r3 r2
  - Value: r3 - r2
- addn r3 -1
  - Value: r3 + -1
- write r3
- halt

Extra! Change only line 4's instruction to create an output of 0 or 6 or 349 instead?
Try this on the back page first!

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

**RAM**
- Random access memory

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

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

Hmmm...!?
Could you write a Hmmm program that computes

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

or

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

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

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

$\frac{1}{\sqrt{x}}$

when you'd want to!
Could you write a Hmmm program to compute $x + 3x - 4$ or $1/x$?

Motivation

The inverse square root of a floating point number is used in calculating a normalized vector. Since a 3D graphics program uses these normalized vectors to determine lighting and reflection, millions of these calculations must be done per second. Before the creation of specialized hardware to handle transform and lighting, software computations could be slow. Specifically, when the code was developed in the early 1990s, most floating point processing power lagged behind the speed of integer processing.

$1/\sqrt{x}$ when you'd want to!
### 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>IRET</td>
<td>Return from interrupt</td>
</tr>
</tbody>
</table>

A few of the many basic processor instructions (Intel)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPSADDBW</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>

Two *more recent* Intel instructions (SSE4 subset)
Who writes all of the assembly language that gets executed?
Who writes all of the assembly language that gets executed?
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?

What's missing here?

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read r1</td>
</tr>
<tr>
<td>1</td>
<td>mul r2 r1 r1</td>
</tr>
<tr>
<td>2</td>
<td>add r2 r2 r1</td>
</tr>
<tr>
<td>3</td>
<td>write r2</td>
</tr>
<tr>
<td>4</td>
<td>halt</td>
</tr>
</tbody>
</table>

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 if statements

We couldn't implement Python using Hmmm so far...

'It's too linear!'
CPU
central processing unit

r1
General-purpose register r1

r2
General-purpose register r2

Screen

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?
Jumps in Hmmm

**Conditional** jumps

- jeqzn r1 42  
  IF r1 == 0  THEN jump to line number 42

- jgtzn r1 42  
  IF r1 > 0  THEN jump to line number 42

- jltzn r1 42  
  IF r1 < 0  THEN jump to line number 42

- jnezn r1 42  
  IF r1 != 0  THEN jump to line number 42

**Unconditional** jump

- jumpn 42  
  Jump to program line # 42

This is making me jumpy!
Jumps in Hmmm

**Conditional jumps**

- **jeqzn**: if equal to zero... THEN jump to line number 42
- **jgtzn**: if greater than zero ... EN jump to line number 42
- **jltzn**: if less than zero... THEN jump to line number 42
- **jnezr**: if not equal to zero ... HEN jump to line number 42

**Unconditional jump**

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

This is making me jumpy!
### System instructions

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

### Arithmetic

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

### Jumps!

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

### Interacting with memory (RAM)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushr rX rY</td>
<td>Store contents of register rX onto stack pointed to by reg. rY</td>
</tr>
<tr>
<td>popr rX rY</td>
<td>Load contents of register rX from stack pointed to by reg. rY</td>
</tr>
<tr>
<td>loadr rX N</td>
<td>Load register rX with the contents of memory address N</td>
</tr>
<tr>
<td>storer rX N</td>
<td>Store contents of register rX into memory address N</td>
</tr>
<tr>
<td>loadi rY</td>
<td>Load register rX with data from the address location held in reg. rY</td>
</tr>
<tr>
<td>storei rY</td>
<td>Store contents of register rX into memory address held in reg. rY</td>
</tr>
</tbody>
</table>
With an input of -6, what does this code write out?
Try it!

Follow this Hmmm 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:

(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*...)

---

### Memory - RAM

<table>
<thead>
<tr>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( 4 )</th>
<th>( 5 )</th>
<th>( 6 )</th>
<th>( 7 )</th>
<th>( 8 )</th>
<th>( 9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>read</strong> ( r1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>write</strong> ( r2 )</td>
<td></td>
</tr>
<tr>
<td>( r1 )</td>
<td></td>
<td>( 5 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( r2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( r3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Registers - CPU

**Run 1**
- \( r1 = 42 \)
- \( r2 = 5 \)

**Run 2**
- \( r1 = 5 \)
- \( r2 = 42 \)

**Output 1**
- \( 42 \)

**Output 2**
- \( 5 \)
factorial: the *plan* ...

let **r1** be the input and the "counter"

let **r2** be the output

fac(5) is $1 \times 5 \times 4 \times 3 \times 2 \times 1$

let **r2** *become* the output

starting value!
Try it!

I think this language has injured my craniuhmm!

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

Write an assembly-language program that reads a positive integer into \( r_1 \). The program should compute the \textit{factorial} of the input in \( r_2 \). 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 \textbf{common function} does this compute?

\textit{Hint: try the inputs in both orders...}

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

\textit{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!

\textbf{Extra!} How few lines can you use here? (Fill the rest with \texttt{nop}s...)
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>Memory - RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

(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?
a factorial solution

Registers - CPU

r1
input

r2
result – so far

r3
not needed, but OK to use!

Memory - RAM

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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...

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

... in Hmmm assembly language

See you there!
My examples of Richter-like displays are shown below using 9 colours chosen at random within each square of a 9 x 9 grid. There are often apparent clusters and patterns in the colours. Can you spot the fake piece of random art?

Which one is NOT random...?