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

How does Python function?

CS 5 this week
Hardware

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

Software

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

Python

How does Python function?

Hmmm

RAM
registers

John Von Neumann

Bell labs + beyond

Grace Hopper

Turing, et al.

CS 5 this week

Turing,
et al.

John Von Neumann

Grace Hopper

Bell labs + beyond
Hardware

CS 5 this week

1-bit memory: flip-flops

Logic gates/transistors/switches

Bitwise functions, arithmetic

John Von Neumann

Turing, et al.

Grace Hopper

Bell labs + beyond

Python fall break is also a CS hw break...

Hmmm

4 Hmmm problems + 1 loop problem due Mon. 10/23
Farewell, Logisim!
The flip-flop

inputs

"strobe"
"we are ready to handle the data"

1 bit of memory!

The flip-flop's diagram

Q is 1 bit of storage
The flip-flop

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

1 bit of memory!
Random Access Memory

Extra this week: *Design 12nGbits of RAM*

**Inputs**
- 3 data input bits
- 2 data address bits
  - Should I write?
  - Should I read?

**Simplified Prototype for Accessing Memory**
- 12 bits of RAM

**Outputs**
- 3 data output bits

3 bits stored at location 00
3 bits stored at location 01
3 bits stored at location 10
3 bits stored at location 11
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?

memory location

write enable line

read enable line

two other memory lines and their flip-flops are not drawn

STORE
the value 5 into mem. loc. #1

0

1

Binary Address Decoder

data address, in binary

3 data input bits

3 data output bits

0.1

2 - 2; 4 - 3

A0

A1

write enable line

read enable line

3 data output bits
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!

**Diagram:**

- 3 data input bits
- Memory location
- Binary Address Decoder
- 01 to the decoder
- 1 goes on
- Make the "Read Enable" high
- Which gates ensure bits from memory location #1 are read out?
- Which gates ensure bits from memory location #0 are not read out?
- Draw the "Read Enable" wire

- Data address, in binary
- Write enable line
- Read enable line
- Two other memory lines and their flip-flops are not drawn
- 3 data output bits
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...

<table>
<thead>
<tr>
<th><strong>Registers</strong></th>
<th><strong>Main Memory</strong> (replaceable RAM)</th>
<th><strong>Disk Drive</strong> magnetic storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>on the Central Processing Unit</td>
<td>~ 10,000 bits</td>
<td>~ 42 trillion bits (or more)</td>
</tr>
<tr>
<td>8 flip-flops are an 8-bit register</td>
<td>~ 100 billion bits</td>
<td>~ 100,000 bits</td>
</tr>
<tr>
<td>100 Registers of 64 bits each</td>
<td>100 cycles</td>
<td>10^7 cycles</td>
</tr>
<tr>
<td>~ $100</td>
<td>~ $100</td>
<td>~ $100</td>
</tr>
<tr>
<td>1 clock cycle</td>
<td>10^{-9} sec</td>
<td>10^{-7} sec</td>
</tr>
<tr>
<td>1 min</td>
<td>1.5 hours</td>
<td>19 YEARS</td>
</tr>
</tbody>
</table>

If a clock cycle == 1 minute
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...

- “Off” data is saved way out here...
- 10⁻² sec

If a clock cycle == 1 minute

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

- **Processor**
  - CPU
  - RAM
    - Stores all instructions and almost all data
    - Live memory
    - Sends next instruction to the CPU
    - Sends next instruction to the CPU

- **Circuitry**
  - Multiplier
  - Divider
  - Memory locations (RAM)
    - Inside the 12nGbits of memory

- **Process**
  - The instruction's bits select which circuit to use.
  - CPU runs 1 instruction and sends back results for storage, if requested.
  - CPU sends next instruction to the CPU.
70 years ago...

Von Neumann architecture
From Wikipedia, the free encyclopedia

processing
CPU
central processing unit registers

fetch
execute

stored program
RAM
random access memory locations

limited, fast registers + arithmetic

larger, slower memory + no computation

Jon V.N.
70 years later...

Von Neumann architecture
From Wikipedia, the free encyclopedia

limited, fast **registers** + arithmetic

larger, slower **memory** + *no* computation
Programs are stored in memory in *machine language*.
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 read instruction

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

"mnemonics" instead of bits
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>mul r2 r1 r1</td>
<td>1000 0010 0001 0001</td>
</tr>
<tr>
<td>r2</td>
<td>add r2 r2 r1</td>
<td>0110 0010 0001 0001</td>
</tr>
<tr>
<td></td>
<td>write r2</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td></td>
<td>halt</td>
<td>0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

"mnemonics" instead of bits
Von Neumann Architecture

**CPU**
- Central processing unit *registers*
- General-purpose register, r1
- General-purpose register, r2

**RAM**
- Random access memory locations
- Program
  - `read r1`
  - `mul r2 r1 r1`
  - `add r2 r2 r1`
  - `write r2`
  - `halt`

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

"mnemonics" instead of *bits"
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
Demo

of "in vivo" assembly language
and machine language

```
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

```
int main()
{
    int answer = 84;
    answer = answer + 42;
    printf("\nThe result is %d.\n\n", answer);
}
```
Demo
of "in vivo" assembly-language

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

original

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

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

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

changed

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

CPU
central processing unit \textit{registers}

RAM
random access memory locations

\textbf{r1}
General-purpose register r1

\textbf{r2}
General-purpose register r2

\textbf{read r1}

\textbf{mul r2 r1 r1}

\textbf{add r2 r2 r1}

\textbf{write r2}

\textbf{halt}

\textbf{6} (input)

\textbf{42}

\textbf{6}

Von Neumann bottleneck
Hmmm: Harvey mudd miniature machine

CPU
central processing unit registers

RAM
random access memory locations

Von Neumann bottleneck

r1
General-purpose register r1

r2

16 registers

0
read r1

1
mul r2 r1

2

3

4
halt

256 memory locations

vs. 2018?
Hmmm vs 2018

CPU
central processing unit *registers*

RAM
random access memory locations

Von Neumann bottleneck

r1
General-purpose register r1

r2

16 registers

0
read r1

1
mul r2 r1

2

3

4
halt

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>
<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>mov</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>jumpr rX</td>
<td>Set program counter to address in rX</td>
<td>jump</td>
</tr>
<tr>
<td>jeqzn rX N</td>
<td>If rX == 0, then jump to line N</td>
<td>jeqz</td>
</tr>
<tr>
<td>jnezr 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>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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pushr rX rY</td>
<td>Store contents of register rX onto stack pointed to by reg. rY</td>
<td></td>
</tr>
<tr>
<td>popr rX rY</td>
<td>Load contents of register rX from stack pointed to by reg. rY</td>
<td></td>
</tr>
<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>

---

At [www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html](http://www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html)

Today

Thurs.
Assembly Language

read r1  
write r2  

reads from keyboard into reg r1  
outputs reg r2 onto the screen  

reg3 = reg1 + reg2  
reg3 = reg1 - reg2  
reg2 = reg1 * reg1  
reg1 = reg1 / reg2  

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

sets r1 42  
add r1 -1  
reg1 = 42  
reg1 = reg1 - 1  
you can replace 42 with anything from -128 to 127  
a shortcut

add r3=r1+r2  
sub r3=r1-r2  
mul r2=r1*r1  
div r1=r1÷r2  

ought to be called register language
**Name(s) ________________________**

**Quiz**

**CPU**
central processing unit

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

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

- **r3**
  - 42 ≠ 50
  - General-purpose register r3

- **r4**
  - 2
  - General-purpose register r4

Hmmm...!

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

**RAM**
random access memory

- **read r1**
  - 100
  - r1 = 100

- **setn r2 7**

- **mod r4 = r1 % r2**

- **div r3 = r1 / r4**

- **sub r3 = r3 - r2**

- **addn r3 -1**

- **write r3**

- **halt**

**screen**

100 (input)

42 (output)
Try this on the back page first!

CPU
central processing unit

r1
General-purpose register r1

100

r2
General-purpose register r2

7

r3
General-purpose register r3

r4
General-purpose register r4

Hmmm...!

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

Python

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

RAM
random access memory

read r1
setn r2 7
mod r4 r1 r2
div r3 r1 r4
sub r3 r3 r2
addn r3 -1
write r3
halt

100 (input)

(output)

Try this on the back page first!
Try this on the back page first!

CPU

central processing unit

r1

General-purpose register r1

100

r2

General-purpose register r2

r3

General-purpose register r3

r4

General-purpose register r4

RAM

random access memory

read

0

write

r3

print r3

halt

mod r4 r1 r2

r4 = r1 % r2

div r3 r1 r4

r3 = r1 // r4

sub r3 r3 r2

r3 = r3 - r2

addn r3 -1

r3 = r3 + -1

r2 = 7

r1 = 100

r2 = 7

r4 = r1 % r2

r3 = r1 // r4

r3 = r3 - r2

r3 = r3 + -1

print r3

Hmmm...!?

Extra! Change only line 4’s instruction to create an output of 0 or 6 or 349 instead?
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 $1/x$ when you'd want to?
Could you write a Hmmm program

$1/\sqrt{x}$

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?

other programs!
Could you write a Python program that writes a Hmmm program that computes

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

or

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

?  

Yes – you already have! much better!
Is this all we need?

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!*

```
jumpn!
```

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

"straight-line code"
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

This is making me jumpy!

**Unconditional jump**

- `jumpn 42`  
  Jump to program line # 42
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
- `jnezrn` if `not equal to zero`... HEN jump to line number 42

Unconditional jump

- `jumpn 42` Jump to program line # 42

Mnemonics!

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>
<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 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>jeqzn rX N</td>
<td>If rX == 0, then jump to line N</td>
</tr>
<tr>
<td>jnez rX N</td>
<td>If rX != 0, then jump to line N</td>
</tr>
<tr>
<td>jgtzn rX N</td>
<td>If rX &gt; 0, then jump to line N</td>
</tr>
<tr>
<td>jltzn rX N</td>
<td>If rX &lt; 0, then jump to line N</td>
</tr>
<tr>
<td>calln 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>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>
With an input of -6, what does this code write out?
Try it! Follow this Hmm program.
First run: use \( r1 = 42 \) and \( r2 = 5 \).
Next run: use \( r1 = 5 \) and \( r2 = 42 \).

(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...)

---

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
jgtn r3 7
write r1
jumpn 8
write r2
halt
```

---

**Memory - RAM**

<table>
<thead>
<tr>
<th>Registers - CPU</th>
<th>Memory - RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read r1</td>
</tr>
<tr>
<td>1</td>
<td>read r2</td>
</tr>
<tr>
<td>2</td>
<td>sub r3 r1 r2</td>
</tr>
<tr>
<td>3</td>
<td>nop</td>
</tr>
<tr>
<td>4</td>
<td>jgtzn r3 7</td>
</tr>
<tr>
<td>5</td>
<td>write r1</td>
</tr>
<tr>
<td>6</td>
<td>jumpn 8</td>
</tr>
<tr>
<td>7</td>
<td>write r2</td>
</tr>
<tr>
<td>8</td>
<td>write r2</td>
</tr>
<tr>
<td>9</td>
<td>halt</td>
</tr>
</tbody>
</table>

**Register Values**

- \( r1 \) : 42, 5
- \( r2 \) : 5, 42
- \( r3 \) : not needed; OK to use

**Memory - RAM**

- Read \( r1 \)
- Set \( r2 \) to 1
- Write \( r2 \)
- Halt
factorial: the plan ...

let r1 be the input and the "counter"

let r2 become the output

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

starting value!
Try it! Follow this Hmm program.  
First run: use \( r_1 = 42 \) and \( r_2 = 5 \). 
Next run: use \( r_1 = 5 \) and \( r_2 = 42 \). 

1. Write an assembly-language program that reads a positive integer into \( r_1 \). The program should compute the 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 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?
   
   *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...)

---

2. Write an assembly-language program that reads a positive integer into \( r_1 \). The program should compute the factorial of the input in \( r_2 \). Once it's computed, it should write out that factorial. Two lines are provided:

```
read r1
setn r2 1
```

Memory - RAM

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>42</td>
</tr>
<tr>
<td>r2</td>
<td>5</td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>

Output 1

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>5</td>
</tr>
<tr>
<td>r2</td>
<td>42</td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>

Output 2

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>42</td>
</tr>
<tr>
<td>r2</td>
<td>5</td>
</tr>
<tr>
<td>r3</td>
<td></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>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>setn r2 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>jeqzn r1 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mul r2 r2 r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>addn r1 -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>jumpn 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>nop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>write r2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>halt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

space for future expansion!
This week in lab:

Randohmmm
Numbers...

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

... in Hmmm assembly language

See you there!
Can you spot the fake piece of random art?

Which one is NOT random...?