Hmmm… (The Harvey Mudd Miniature Machine)
Stuff

- HW 5 (OK to turn in by next Tuesday)
- HW 6 (Due next Tuesday as usual; 75 points)
- One lab problem next week but that’s it!
- No homework over Spring Break
- We’re in Chapter 4 in the book!
- Exams returned on Thursday but sample solutions distributed at start of lab today!
Von Neumann Architecture

Programs are stored in memory in *machine language* (bits)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000100111101000</td>
<td>111110100100001</td>
<td>0001011011111001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1010100111110010</td>
<td>0000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0000100111101000</td>
<td>111110100100001</td>
<td>0001011011111001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1010100111110010</td>
<td>0000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Von Neumann Architecture

Assembly language is human-readable machine language substituting words for bits

Main Memory (aka RAM)

CPU

the processing central processing unit

Von Neumann bottleneck

0
read r1

1
mul r2 r1 r1

2
add r2 r2 r1

3
write r2

4
halt

5

6

...
the fetch - execute cycle

CPU
central processing unit *registers*

RAM
random access memory locations

Von Neumann bottleneck

Program Counter
Holds address of the next instruction

Instruction Register
Holds the current instruction

General-purpose register r1

General-purpose register r2

0
read r1

1
mul r2 r1 r1 r1

2
add r2 r2 r2 r1

3
write r2

4
halt
16 Registers, Lots of Memory!

macOS Catalina
Version 10.15.3

MacBook Pro (13-inch, 2019, Four Thunderbolt 3 ports)
Processor 2.4 GHz Quad-Core Intel Core i5
Memory 8 GB 2133 MHz LPDDR3
Graphics Intel Iris Plus Graphics 655 1536 MB
Serial Number C02ZJ1YZLVDC

System Report... Software Update...
Hmmm

The Harvey Mudd Miniature Machine

CPU
central processing unit

RAM
random access memory

Von Neumann bottleneck

Program Counter
Holds address of the next instruction

Instruction Register
Holds the current instruction

r0 0
Holds value 0

r1

r2

r15

16 registers, each 16 bits
they can hold values from -32768 to 32767

register 0 is "hard-wired" to store 0

0 read r1
1 mul r2 r1 r1
2 add r2 r2 r1
3 write r2
4
5 halt
6

255

255 memory locations of 16 bits
Hmmm: the *fetch - execute* cycle

**CPU**

- Central processing unit

**Program Counter**

- Holds address of the next instruction
  - Initially 0

**Instruction Register**

- Holds the current instruction
  - Initially 0
  - register 0 is “hard-wired” to store 0

**RAM**

- Random access memory

0. **read r1**

1. **mul r2 r1 r1**

2. **add r2 r2 r1**

3. **write r2**

4. **halt**

**DEMO!**
Assembly Language

add r2 r2 r2
sub r2 r1 r4
mul r7 r6 r2
div r1 r1 r1
setn r1 42
addn r1 -1
read r0
write r0

reg2 = reg2 + reg2
crazy, perhaps, but used ALL the time
reg2 = reg1 - reg4
which is why it is written this way in python!
reg7 = reg6 * reg2
reg1 = reg1 / reg1
INNERGER division - no remainders
reg1 = 42
you can replace 42 with anything from -128 to 127
reg1 = reg1 - 1
a shortcut

Each of these instructions (and many more) get implemented for a particular processor and particular machine….
n is for number...

\[
\begin{array}{c|c}
\text{r1} & 5 \\
\text{r2} & 42 \\
\end{array}
\]

\[
\begin{align*}
&\text{add r1 r1 r2} \\
&\text{addn r1 42}
\end{align*}
\]
Hmmm, Let's get jumpn!

**CPU**
central processing unit

- **Program Counter**
  - Holds address of the next instruction

- **Instruction Register**
  - Holds the current instruction
  - register 0 is “hard-wired” to store 0

- **r0**
  - 0

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

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

**RAM**
random access memory

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

**Unconditional jump**  \texttt{GOTO}

\texttt{jumpn 42}  

replaces the PC (program counter) with \texttt{42}. 
"jumpnto line number \texttt{42}"

**Conditional jumps**  \texttt{IF}

\texttt{jeqzn r1 93}  

IF \texttt{r1} == \texttt{0} THEN jump to line number \texttt{93}

\texttt{jgtzn r1 93}  

IF \texttt{r1} > \texttt{0} THEN jump to line number \texttt{93}

\texttt{jltzn r1 93}  

IF \texttt{r1} < \texttt{0} THEN jump to line number \texttt{93}

\texttt{jnezn r1 93}  

IF \texttt{r1} \neq \texttt{0} THEN jump to line number \texttt{93}
What is this code computing about its input?
1. Follow this assembly-language program from top to bottom. Use $r1 = 42$ and $r2 = 5$. Then, try $r1 = 5$ and $r2 = 42$.

<table>
<thead>
<tr>
<th>Registers - CPU</th>
<th>Memory - RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

- read r1
- read r2
- sub r3 r2 r1
- nop
- jltzn r3 7
- write r1
- jumpn 8
- write r2
- halt

(1) What does this program compute in general?

2. Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

<table>
<thead>
<tr>
<th>Registers - CPU</th>
<th>Memory - RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
</tr>
</tbody>
</table>

- read r1
- read r2
- sub r3 r2 r1
- nop
- jltzn r3 7
- write r1
- jumpn 8
- write r2
- halt

*Hint:* Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
1. Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

2. Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
1. Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

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

(1) What does this program compute in general?

2. Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

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

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
1 Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \). Then, try \( r1 = 5 \) and \( r2 = 42 \).

2 Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Follow this assembly-language program from top to bottom. Use $r1 = 42$ and $r2 = 5$. Then, try $r1 = 5$ and $r2 = 42$.

Write an assembly-language program that reads one integer as keyboard input. Then, the program should compute the factorial of that input and write it out. You may assume without checking that the input will be a positive integer.

(1) What does this program compute in general?

**Hint:** Take in an input. Next, set up a “result” register starting with 1 in it. Then modify the “result” until it’s right!
Factorial...

00 read r1   # read input into r1
01 setn r2 1  # r2 = 1
02 jeqzn r1 6  # if r1 == 0, jump to line 6
03 mul r2 r2 r1  # r2 = r2 * r1
04 addn r1 -1  # r1 = r1 - 1
05 jumpn 2  # jump to line 2
06 write r2
07 halt

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

### System instructions

### Setting register data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>setn</strong> rX N</td>
<td>Set register rX equal to the integer N (-128 to +127)</td>
<td></td>
</tr>
<tr>
<td><strong>addn</strong> rX N</td>
<td>Add integer N (-128 to 127) to register rX</td>
<td></td>
</tr>
<tr>
<td><strong>copy</strong> 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><strong>add</strong> rX rY rZ</td>
<td>Set rX = rY + rZ</td>
<td></td>
</tr>
<tr>
<td><strong>sub</strong> rX rY rZ</td>
<td>Set rX = rY - rZ</td>
<td></td>
</tr>
<tr>
<td><strong>neg</strong> rX rY</td>
<td>Set rX = -rY</td>
<td></td>
</tr>
<tr>
<td><strong>mul</strong> rX rY rZ</td>
<td>Set rX = rY * rZ</td>
<td></td>
</tr>
<tr>
<td><strong>div</strong> rX rY rZ</td>
<td>Set rX = rY / rZ (integer division; no remainder)</td>
<td></td>
</tr>
<tr>
<td><strong>mod</strong> 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><strong>jumpn</strong> N</td>
<td>Set program counter to address N</td>
<td></td>
</tr>
<tr>
<td><strong>jump</strong> rX</td>
<td>Set program counter to address in rX</td>
<td></td>
</tr>
<tr>
<td><strong>jeqzn</strong> rX N</td>
<td>If rX == 0, then jump to line N</td>
<td>jeqz</td>
</tr>
<tr>
<td><strong>jnezn</strong> rX N</td>
<td>If rX != 0, then jump to line N</td>
<td>jnez</td>
</tr>
<tr>
<td><strong>jgtzn</strong> rX N</td>
<td>If rX &gt; 0, then jump to line N</td>
<td>jgtz</td>
</tr>
<tr>
<td><strong>jltzn</strong> rX N</td>
<td>If rX &lt; 0, then jump to line N</td>
<td>jltz</td>
</tr>
<tr>
<td><strong>calln</strong> 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><strong>pushr</strong> rX rY</td>
<td>Store contents of register rX onto stack pointed to by reg. rY</td>
<td></td>
</tr>
<tr>
<td><strong>popr</strong> rX rY</td>
<td>Load contents of register rX from stack pointed to by reg. rY</td>
<td></td>
</tr>
<tr>
<td><strong>loadn</strong> rX N</td>
<td>Load register rX with the contents of memory address N</td>
<td></td>
</tr>
<tr>
<td><strong>storen</strong> rX N</td>
<td>Store contents of register rX into memory address N</td>
<td></td>
</tr>
<tr>
<td><strong>loadr</strong> 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><strong>storer</strong> rX rY</td>
<td>Store contents of register rX into memory address held in reg. rY</td>
<td>storei, store</td>
</tr>
</tbody>
</table>
A function call in python:

```python
def main():
    r1 = input()
    result = factorial(r1)
    print result

def factorial( r1 ):
    # do work
    return result
```

Hmmm's call operation:

```
<table>
<thead>
<tr>
<th></th>
<th>read r1</th>
<th>calln r14 4</th>
<th>write r13</th>
<th>halt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

puts NEXT line # into r14, then jumps to line 4
def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)

def factorial( r1 ):
    # do work
    return result

def main():
    r1 = input()
    result = factorial(r1)
    print(result)
A function call in python:

Hmmm's `calln` operation:

- puts NEXT line # into r14, then jumps to line 4

United Nations Resolution 424242

- Functions always receive their input in register r1 (and r2, r3, and so forth if there more inputs)
- Functions return their answers in register r13
- Functions `jumpr r14` to return to where they were called

0   read r1
1   calln r14 4
2   write r13
3   halt
4   do stuff...
5   put in r13
6   jumpr r14
What does this do?

00 read r1
01 jeqzn r1 5  # if r1==0 then jump to line 5
02 calln r14 6
03 write r13
04 jumpn 0     # jump to line 0
05 halt
06 setn r13 1  # r13 = 1
07 setn r2 2   # r2 = 2
08 jeqzn r1 12
09 addn r1 -1
10 mul r13 r13 r2
11 jumpn 8
12 jump r14    # jump to the address in r14
The Collatz Function!

def collatz(n):
    counter = 0
    while n != 1:
        counter += 1  # increment counter
        if n % 2 == 0:  n = n // 2
        else: n = 3*n + 1
    return counter

>>> collatz(8)
>>> collatz(5)
>>> collatz(27)
Collatz... Hmmm...

```python
def main():
    read r1
    r13 = collatz(r1)
    write r13
    halt

def collatz(r1):
    r13 = 0 # this is the result
    TOP:
        if r1 == 1: jump to END
    else:
        r13 = r13 + 1
        if r1 is even:
            r1 = r1 // 2
            jump back to TOP
        else:
            r1 = 3 * r1 + 1
            jump back to TOP
    END: return to function that called us!
```

One tricky thing is that there's no way to test if \( r1 == 1 \).

```
00 read r1
01 calln r14 4
02 write r13
03 halt
04 setn r13 0    # r13 = 0
05 setn r2 -1    # r2 = -1
06 setn r3 2     # r3 = 2
07 setn r4 3     # r4 = 3
08                # TOP
```

```assembly
mod r3 r5 r11
r3 = r5 % r11
```