How Computers (really) Work!

Starring…

- The CS 5 Gold 3-eyed alien
- The CS 5 Black 5-eyed alien
- The CS 5 Green magic turtle!
The “von Neumann Architecture”

the “Central Processing Unit” (CPU)
small amount of fast memory

main memory
huge memory, but takes a “long time” to access
The “von Neumann Architecture”

the “Central Processing Unit” (CPU)
small amount of fast memory

main memory
huge memory, but takes a “long time” to access
The “von Neumann Architecture”

typically 16 or 32 “registers” to store stuff

typically billions of slots to store programs and large amounts of data

```python
def check_this_out()
    counter = 0
    for num in range(0, 42):
        counter += 10**9
    print(counter)
```

`num` and `counter` are variables used in the code.
The “von Neumann Architecture”

CPU

Program Counter (PC)

Instruction Register

num

counter

Typically 16 or 32 “registers” to store stuff

Typically billions of slots to store programs and large amounts of data

Fetch!

Please go to memory location 1 and fetch that instruction

def check_this_out():
    counter = 0
    for num in range(0, 42):
        counter += x
    ...

0

1

2

3

10^9

...
The “von Neumann Architecture”

CPU

- Program Counter (PC)
- Instruction Register
- num
- counter

typically 16 or 32 “registers” to store stuff

Here you are... It's counter = 0

```
def check_this_out():
    counter = 0
    for num in range(0, 42):
        counter += x
    ...  
```

typically billions of slots to store programs and large amounts of data
The “von Neumann Architecture”

**CPU**

- Program Counter (PC)
- Instruction Register
- num
- counter

typically 16 or 32 “registers” to store stuff

```python
def check_this_out():
    counter = 0
    for num in range(0, 42):
        counter += x

    ... # counter now has a value
```

typically billions of slots to store programs and large amounts of data

Now I'll execute...
`counter = 0`

```python
counter = 0
```
The “von Neumann Architecture”

CPU

- Program Counter (PC)
  - counter
  - num

- Instruction Register
  - counter = 0

- Typically 16 or 32 “registers” to store stuff

- Typically billions of slots to store programs and large amounts of data

```
def check_this_out():
    counter = 0
    for num in range(0, 42):
        counter += x

    ... 
```

Now I’ll execute...

```
counter = 0
num
counter = 0
... 
10^9
... 
```
The “von Neumann Architecture”

CPU

Program Counter (PC)

Instruction Register

num

counter

And now increment the program counter (PC) by 1

def check_this_out():
    counter = 0
    for num in range(0, 42):
        counter += x
    ...

typically 16 or 32 “registers” to store stuff

typically billions of slots to store programs and large amounts of data

0
1
2
3
10^9

...
The “von Neumann Architecture”

- Typically 16 or 32 “registers” to store stuff
- Typically billions of slots to store programs and large amounts of data

```
def check_this_out():
    counter = 0
    for num in range(0, 42):
        counter += x
    ...
```

And now increment the program counter (PC) by 1.
Binary representation of numbers...

<table>
<thead>
<tr>
<th>binary</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1010</td>
<td>10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0</td>
</tr>
</tbody>
</table>
Binary representation of characters...

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Name</th>
<th>Char</th>
<th>Ctrl-char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Null</td>
<td>NUL</td>
<td></td>
<td>32</td>
<td>20</td>
<td>$</td>
<td>64</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Start of heading</td>
<td>SOH</td>
<td>CTRL-A</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Start of text</td>
<td>STX</td>
<td>CTRL-B</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>End of text</td>
<td>ETX</td>
<td>CTRL-C</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>End of xmit</td>
<td>EOT</td>
<td>CTRL-D</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>Enquiry</td>
<td>ENQ</td>
<td>CTRL-E</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Acknowledge</td>
<td>ACK</td>
<td>CTRL-F</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>Bell</td>
<td>BEL</td>
<td>CTRL-G</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>Backspace</td>
<td>BS</td>
<td>CTRL-H</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>Horizontal tab</td>
<td>HT</td>
<td>CTRL-I</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>1A</td>
<td>Line feed</td>
<td>LF</td>
<td>CTRL-J</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>1B</td>
<td>Vertical tab</td>
<td>VT</td>
<td>CTRL-K</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>1C</td>
<td>Form feed</td>
<td>FF</td>
<td>CTRL-L</td>
<td>44</td>
<td>2C</td>
<td>;</td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>1D</td>
<td>Carriage feed</td>
<td>CR</td>
<td>CTRL-M</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>1E</td>
<td>Shift out</td>
<td>SO</td>
<td>CTRL-N</td>
<td>46</td>
<td>2E</td>
<td>/</td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>1F</td>
<td>Shift in</td>
<td>SI</td>
<td>CTRL-O</td>
<td>47</td>
<td>2F</td>
<td>\</td>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Data line escape</td>
<td>DLE</td>
<td>CTRL-P</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 1</td>
<td>DC1</td>
<td>CTRL-Q</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 2</td>
<td>DC2</td>
<td>CTRL-R</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 3</td>
<td>DC3</td>
<td>CTRL-S</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Device control 4</td>
<td>DC4</td>
<td>CTRL-T</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Neg acknowledge</td>
<td>NAK</td>
<td>CTRL-U</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Synchronous idle</td>
<td>SYN</td>
<td>CTRL-V</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>End of xmit block</td>
<td>ETB</td>
<td>CTRL-W</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Cancel</td>
<td>CAN</td>
<td>CTRL-X</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>End of medium</td>
<td>EM</td>
<td>CTRL-Y</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Substitute</td>
<td>SUB</td>
<td>CTRL-Z</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Escape</td>
<td>ESC</td>
<td>CTRL-[</td>
<td>59</td>
<td>3B</td>
<td>{</td>
<td>91</td>
<td>5B</td>
<td>[</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>File separator</td>
<td>FS</td>
<td>CTRL\</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Group separator</td>
<td>GS</td>
<td>CTRL-]</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Record separator</td>
<td>RS</td>
<td>CTRL^</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Unit separator</td>
<td>US</td>
<td>CTRL-`</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>`</td>
</tr>
</tbody>
</table>

in Python, try `chr(7), ord('a')`
Binary representation of instructions...

some shocking news...

... there are no turtles inside and I don’t actually speak Python!

What! No turtles inside!?
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>000100111101000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>111110100100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>000101101111101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1010100111000010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0000000000000000</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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Mark 1
relay-based computer

Grace Hopper + Howard Aiken, Harvard ~ 1944
ran at 0.00001 MHz

Addition: 0.6 seconds
Multiplication: 5.7 seconds
Division: 15.3 seconds

5 tons
530 miles of wiring
765,299 distinct parts!
Grace Hopper

Grace Murray Hopper ’28 taught math and physics at Vassar for 12 years before joining the Navy reserves in 1943. During the war she learned to program the Mark I, the world’s first large-scale computer, which was used to perform the calculations needed to position the Navy’s weaponry: guns, mines, rockets, and, eventually, the atomic bomb.

In 1945, she coined the term “debugging” after finding a moth stuck in the computer’s machinery. Over the course of her career, Hopper invented the compiler to automate common computer instructions, became the first to start writing computer programs in English, and helped to develop the first “user-friendly” computer language, COBOL.

“In the days they used oxen for heavy pulling, when one ox couldn’t budge a log, they didn’t try to grow a larger ox. We shouldn’t be trying for bigger and better computers, but for better systems of computers.”
The first bug?

<table>
<thead>
<tr>
<th>Time</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>Arcan started</td>
</tr>
</tbody>
</table>
| 1000  | Arcan stopped - Arcan
| 13:06 (03:3) MP - MC
| 03:3 PRO - 2.130764675
|       | Conv 2.130764675 |
|       | Relays 6-2 in 03:3 failed squaw test
|       | In relay
| 11:00 | Started Cosine Tape (Sine check) |
| 15:25 | Started Multi-Adder Test |
| 15:15 | Relay #70 Panel F
|       | (moth) in relay |

First actual case of bug being found.

I'm glad it's not called de-mothing.
Von Neumann Architecture

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

| 0      | 00001001111010000  |
| 1      | 1111110100100001    |
| 2      | 0001011011111001101010011100001000000000000000000 |
| 3      | 1010100111000010    |
| 4      | 000000000000000000  |
| 5      |                   |
| 6      |                   |
| ...    |                   |
Von Neumann Architecture

CPU  central processing unit

Main Memory (aka RAM)

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>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Don Gillies Sr.
the *fetch - execute* cycle

Central processing unit (CPU) registers

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

- **Instruction Register**
  - Holds the current instruction

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

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

Von Neumann bottleneck

Random access memory locations

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000100111101000</td>
</tr>
<tr>
<td>1</td>
<td>1111110100100001</td>
</tr>
<tr>
<td>2</td>
<td>0001011011111001</td>
</tr>
<tr>
<td>3</td>
<td>10101001111000010</td>
</tr>
<tr>
<td>4</td>
<td>00000000000000000</td>
</tr>
</tbody>
</table>
the *fetch - execute* cycle

**CPU**
- Central processing unit *registers*

**RAM**
- Random access memory locations

**Von Neumann bottleneck**

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Instruction Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holds address of the next instruction</td>
<td>Holds the current instruction</td>
</tr>
</tbody>
</table>

**General-purpose register r1**

**General-purpose register r2**

<table>
<thead>
<tr>
<th>Step</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>
CPU
central processing unit

Von Neumann bottleneck

RAM
random access memory

Program Counter
Holds address of the next instruction

Instruction Register
Holds the current instruction

register 0 is "hard-wired" to store 0

r0

r1

r2

... 16 registers, each 16 bits

r15

read r1
mul r2 r1 r1
add r2 r2 r1
write r2
halt

... 256 memory locations of 16 bits

0 1 2 3 4 5 6... 255

r0

0

r1

... values from -32768 to 32767 (inclusive)

r15
Hmmm: the *fetch - execute* cycle

**CPU**
- Initially 0
- Central processing unit
- Holds address of the next instruction

**Instruction Register**
- Holds the current instruction
- General-purpose register r1
- General-purpose register r2

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

What does this program do?
Assembly Language

`add r2 r2 r2`  
`reg2 = reg2 + reg2`  
crazy, perhaps, but used ALL the time

`sub r2 r1 r4`  
`reg2 = reg1 - reg4`  
which is why it is written this way in python!

`mul r7 r6 r2`  
`reg7 = reg6 * reg2`

`div r1 r1 r1`  
`reg1 = reg1 / reg1`  
INTEGER division - no remainders

`setn r1 42`  
`reg1 = 42`  
you can replace 42 with anything from -128 to 127

`addn r1 -1`  
`reg1 = reg1 - 1`  
a shortcut

`read r0`  
read from keyboard and write to screen

`write r0`  
each of these instructions (and many more) get implemented for a particular processor and particular machine… .
Is this enough?

Could we implement Python using Hmmm Assembly?

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

*fetch-execute cycle*
It’s not enough!

Could we implement Python using our Hmmm Assembly Language so far?

It's all too linear!

"straight-line code"

<table>
<thead>
<tr>
<th>Line</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>
Hmmm, Let's get jumpn!

**CPU**  
<table>
<thead>
<tr>
<th>Central processing unit</th>
</tr>
</thead>
</table>

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

**Instruction Register**  
Holds the current instruction

- **r0**  
  0 register 0 is “hard-wired” to store 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

```
jumpn  42
```
replaces the PC (program counter) with 42 "jump to line number 42"

*Conditional* jumps

```
jeqzn  r1   93
jgtzn  r1   93
jltzn  r1   93
jnezn  r1   93
```
IF r1 == 0 THEN jump to line number 93
IF r1 > 0 THEN jump to line number 93
IF r1 < 0 THEN jump to line number 93
IF r1 != 0 THEN jump to line number 93

*Indirect register* jump

```
jump r1
```
Jump to the line # stored in register r1!
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 \( r2 = 42 \).

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.

**Memory - RAM**

<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>

**Registers - CPU**

1. read r1
2. read r2
3. sub r3 r2 r1
4. jltzn r3 7
5. write r1
6. jumpn 8
7. write r2
8. halt

What does this program compute in general?

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

**Memory - RAM**

<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>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
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.

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!
Worksheet

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
```

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!
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.

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.

**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.

**Registers - CPU**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td>5</td>
</tr>
<tr>
<td>r2</td>
<td>42</td>
</tr>
<tr>
<td>r3</td>
<td>37</td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
</tbody>
</table>

**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>read r2</td>
</tr>
<tr>
<td>2</td>
<td>sub r3 r2 r1</td>
</tr>
<tr>
<td>3</td>
<td>nop</td>
</tr>
<tr>
<td>4</td>
<td>jltzn 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>halt</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

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.

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
```

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!
### Worksheet

1. Follow this assembly-language program from top to bottom. Use $r1 = 42$ and $r2 = 5$. Then, try $r1 = 5$ and $r2 = 42$.

   **Registers - CPU**
   
   - $r0$: 0
   - $r1$: 5
   - $r2$: 42
   - $r3$: 37
   - $r4$: 

   **Memory - RAM**
   
   0: `read r1`
   1: `read r2`
   2: `sub r3 r2 r1`
   3: `nop`
   4: `jltzn r3 7`
   5: `write r1`
   6: `jumpn 8`
   7: `write r2`
   8: `halt`
   9: 

   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.

   **Registers - CPU**
   
   - $r0$: 0
   - $r1$: 
   - $r2$: 
   - $r3$: 
   - $r4$: 
   - $r5$: 

   **Memory - RAM**
   
   0: 
   1: 
   2: 
   3: 
   4: 
   5: 
   6: 
   7: 
   8: 
   9: 
   10: 

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

```python
def fac(r1):
    r2 = 1
    while r1 > 0:
        r2 = r2 * r1
        r1 = r1 - 1
    print(r2)
    return
```

Hmmm

```
00 read r1              # read input number and put in r1
01 setn r2 1            # this will be our result
02 jeqzn r1 6           # if r1 == 0, we are done
03 mul r2 r2 r1         # r2 = r2 * r1
04 addn r1 -1           # r1 = r1 - 1
05 jumpn 2              # jump back to line 2
06 write r2             # write result
07 halt                 # halt
```
Hmmm Demo...

worksheet.py
A Short Aside…

- **CPU**
  - 16 Registers ("bytes")
  - Actual time: 1 cycle
  - If “cycle” = 1 sec: $10^{-9}$ sec
  - 1 sec

- **Main Memory (RAM)**
  - 10⁹ “bytes” of memory
  - Actual time: 200 cycles
  - If “cycle” = 1 sec: $10^{-7}$ sec
  - 3-5 minutes

- **Disk Drive**
  - 10¹² “bytes” of memory
  - Actual time: 10⁷ cycles
  - If “cycle” = 1 sec: $10^{-2}$ sec
  - 4.5 MONTHS!
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System instructions</strong></td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td><strong>Setting register data</strong></td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td><strong>Jumps!</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jumpn N</td>
<td>Set program counter to address N</td>
<td></td>
</tr>
<tr>
<td>jumpn 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>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>calln rX N</td>
<td>Copy addr. of next instr. into rX and then jump to mem. addr. N</td>
<td>call</td>
</tr>
<tr>
<td><strong>Interacting with memory (RAM)</strong></td>
<td></td>
<td></td>
</tr>
<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>
A function call in Python:

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

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

Hmmmm’s `call` operation:

```
0  read r1
1  calln r14 4
2  write r13
3  halt
4  do stuff and
5  answer in r13
6  jumper r14
```

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

Who ya gonna call?
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
This week in lab:

Randohmmmm Numbers...

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

... in Hmmm assembly language

See you there!