Why Assembly Language?

It’s only the foolish who never climb Mt. Fuji -- or who climb it again.
Why Assembly Language?

It’s only the foolish who never climb Mt. Fuji -- or who climb it again.
Why Assembly Language?

It’s only the foolish who never climb Mt. Fuji -- or who climb it again.
How should software be assembled?

Python!

How does Python function?

Hmmm

No Lab or HW next week!
Python!
How does Python function?

Hmmm

Jon Von Neumann, advisor

CS 5
Today

Don Gillies, assembler

Grace Hopper, admiral + author of the first high-level (human-level) language, COBOL

Instructions vs. Functions
Python! How does Python function? Hmmm

Jon Von Neumann, advisor

IAS machine

Today

Grace Hopper, admiral + author of the first high-level (human-level) language, COBOL

Don Gillies, assembler

Instructions vs. Functions

winning side - eventually
Grace Murray Hopper ’28 taught math and physics at Vassar for 12 years before joining the Navy reserves in 1943. During the second world war she programmed 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 popularized 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 “human-friendly” computer language, COBOL.

GMH dedications

“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.”
<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>Argon started</td>
</tr>
<tr>
<td></td>
<td>stopped - argon</td>
</tr>
<tr>
<td>1000</td>
<td>-033 MP - MC 2.130 4767415 (3) 4.615 925059 (2)</td>
</tr>
<tr>
<td></td>
<td>-033 PRO 2</td>
</tr>
<tr>
<td></td>
<td>conv. 2.130 4767415</td>
</tr>
<tr>
<td></td>
<td>Relays 6-2 in 033 failed special speed test</td>
</tr>
<tr>
<td></td>
<td>in Relay</td>
</tr>
<tr>
<td></td>
<td>Relays changed</td>
</tr>
<tr>
<td>1100</td>
<td>Started Cosine Tape (Sine check)</td>
</tr>
<tr>
<td>1525</td>
<td>Started Multi-Adder Test</td>
</tr>
<tr>
<td>1545</td>
<td>Relay #70 Panel F</td>
</tr>
<tr>
<td></td>
<td>(Moth) in relay.</td>
</tr>
</tbody>
</table>

First actual case of bug being found.

I'm glad it's not called demothing.
The first bug?

“The OED Supplement records sense (4b) of the noun bug (“a defect or fault in a machine, plan, or the like”) as early as 1889. In that year the Pall Mall Gazette reported (11 Mar: 1) that ‘Mr. Edison ... had been up the two previous nights discovering a ‘bug’ in his phonograph—an expression for solving a difficulty, and implying that some imaginary insect has secreted itself inside and is causing all the trouble.’....

This meaning was common enough by 1934 to be recognized in Webster's New International Dictionary: ‘bug, n.... 3. A defect in apparatus or its operation... Slang, U.S.’” (citation)

I'm glad it's not called deething.
functions vs. instructions

Functions: Python
Instructions: Hmmm
```python
def main():
    x = input()
    y = fun(x)
    print(y)

def fun(x):
    y = x*(x-1)
    return y

Fun: Python

Instructions: Hmmm
```
functions vs. instructions

Goal: To automatically assemble Python functions into Hmmm instructions...

Approach: Use lots of conventions...!

Get into a rut -- and stay there! - V. Rokhlin
let's agree to put the return value in r13

```python
def main():
    r1 = input()
    r13 = fun(r1)
    print(r13)

def fun(r1):
    r13 = r1*(r1-1)
    return r13
```

let's agree to put the function input into r1

return there!

conventions!
calln + jump r

def main():
    r1 = input()
    r13 = fun(r1)
    print r13

def fun( r1 ):
    r13 = r1*(r1-1)
    return r13

jump r14

calln r14 4

store next here, jump there

calln puts the next line # into r14

calln r14 4

jumps to line 4, and puts the next line # into r14

jump r14

jumps to the line # in r14

Who's calln?
calln (call)  
jump (return)  

Instructions: Hmmm
def main():
x = input()
y = fun(x)
print(y)
def fun(x):
y = x*(x-1)
return y

functions vs. instructions

Fun: Python

Instructions: Hmmm

1-1 correspondence is very rare!
Python

```python
x = input()
y = fac(x)
print(y)

def fac(x):
    if x==0:
        return 1
    else:
        RES = fac(x-1)
        return x*RES
```

Hmmm

```
00 read    r1
01 setn    r15 42
02 nop
03 calln   r14 7
04 nop
05 write   r13
06 halt
07 jnezn   r1  11
08 setn    r13 1
09 nop
10 jumpn   r14
11 pushr   r1  r15
12 pushr   r14 r15
13 addn    r1 -1
14 nop
15 calln   r14 7
16 nop
17 popr    r14 r15
18 popr    r1  r15
19 mul     r13 r1 r13
20 jumpn   r14
```

What's the *machine* doing here?

Usually, high-level languages produce LOTS of assembly-level instructions

**Base Case**
- `if x==0:
  return 1`

**Recursive Call**
- `RES = fac(x-1)`
- `return x*RES`

**Factorial Function**
- `def fac(x):
  ...`
Our example...

\[ \text{fac}(3) \Rightarrow 6 \]
```python
def fac(x):
    """ factorial w/ printing """

    if x == 0:
        print("x:", x, " Res: 1")
        return 1
    else:
        print("Next:\nfac(",x-1,")")
        smaller = fac(x-1)
        result = x * smaller
        print("x:",x," Res:",result)
        return result
```

This looks familiar!
Quiz

def fac(x):
    """ factorial w/ printing """
    print("x is", x)

    if x == 0:
        print("x:", x, " Res: 1")
        return 1
    else:
        print("Next: fac(" , x-1 , ")")
        smaller = fac( x-1 )
        result = x * smaller
        print("x:",x," Res:", result)
    return result
def fac(x):
    """ factorial w/ printing """
    print("x is", x)
    if x == 0:
        print("x: ", x, " Res: 1")
        return 1
    else:
        print("Next: fac("",x-1,"")")
        smaller = fac( x-1 )
        result = x * smaller
        print("x:"",x," Res:"",result)
        return result

What will this call to fac(3) print?

[ and by which line? ]

what's printed...
circle the line #

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fac(3)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>print(&quot;x is&quot;, x)</td>
<td>x is 3</td>
</tr>
<tr>
<td>3</td>
<td>if x == 0:</td>
<td>x is 3</td>
</tr>
<tr>
<td>4</td>
<td>print(&quot;x: &quot;, x, &quot; Res: 1&quot;)</td>
<td>Next: fac(2)</td>
</tr>
<tr>
<td>5</td>
<td>return 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>print(&quot;Next: fac(&quot;&quot;,x-1,&quot;&quot;)&quot;)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>smaller = fac( x-1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>fac(3)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>if x == 0:</td>
<td>x is 3</td>
</tr>
<tr>
<td>4</td>
<td>print(&quot;x: &quot;, x, &quot; Res: 1&quot;)</td>
<td>Next: fac(2)</td>
</tr>
<tr>
<td>5</td>
<td>return 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>print(&quot;Next: fac(&quot;&quot;,x-1,&quot;&quot;)&quot;)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>smaller = fac( x-1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>if x == 0:</td>
<td>x is 3</td>
</tr>
<tr>
<td>4</td>
<td>print(&quot;x: &quot;, x, &quot; Res: 1&quot;)</td>
<td>Next: fac(2)</td>
</tr>
<tr>
<td>5</td>
<td>return 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>print(&quot;Next: fac(&quot;&quot;,x-1,&quot;&quot;)&quot;)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>smaller = fac( x-1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>print(&quot;x: &quot;, x, &quot; Res: 1&quot;)</td>
<td>Next: fac(2)</td>
</tr>
<tr>
<td>5</td>
<td>return 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>print(&quot;Next: fac(&quot;&quot;,x-1,&quot;&quot;)&quot;)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>smaller = fac( x-1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>return 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>print(&quot;Next: fac(&quot;&quot;,x-1,&quot;&quot;)&quot;)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>smaller = fac( x-1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>print(&quot;Next: fac(&quot;&quot;,x-1,&quot;&quot;)&quot;)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>smaller = fac( x-1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>print(&quot;Next: fac(&quot;&quot;,x-1,&quot;&quot;)&quot;)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>smaller = fac( x-1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>smaller = fac( x-1 )</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>result = x * smaller</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>print(&quot;x:&quot;&quot;,x,&quot; Res:&quot;&quot;,result)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>return result</td>
<td></td>
</tr>
</tbody>
</table>
def fac(x):
    """ factorial w/ printing """
    print("x is", x)
    if x == 0:
        print("x: ", x, " Res: 1")
        return 1
    else:
        print("Next: fac(" , x-1, ")")
        smaller = fac( x-1 )
        result = x * smaller
        print("x: ", x," Res:",result)
        return result

What will this call to fac(3) print?
[ and by which line? ]
circle the line #
what's printed...

<table>
<thead>
<tr>
<th>x is 3</th>
<th>2 · 4 · 7 · 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next: fac(2)</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>x is 2</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>Next: fac(1)</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>x is 1</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>Next: fac(0)</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>x is 0</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>x:0 Res:1</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>x:1 Res:1</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>x:2 Res:2</td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>x:3 Res:6</td>
<td>2 · 4 · 7 · 10</td>
</tr>
</tbody>
</table>
def fac(x):
    """ factorial w/ printing """
    print("x is", x)
    if x == 0:
        print("x: ", x, " Res: 1")
        return 1
    else:
        print("Next: fac(" , x-1 , ")")
        smaller = fac(x-1)
        result = x * smaller
        print("x: ", x, " Res: ", result)
        return result

Quiz

What will this call to fac(3) print?

Hand these in and up!

<table>
<thead>
<tr>
<th>x is</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2·4</td>
</tr>
<tr>
<td>3</td>
<td>2·4·7</td>
</tr>
<tr>
<td>4</td>
<td>2·4·7·10</td>
</tr>
<tr>
<td>5</td>
<td>2·4·7·10</td>
</tr>
<tr>
<td>6</td>
<td>2·4·7·10</td>
</tr>
<tr>
<td>7</td>
<td>2·4·7·10</td>
</tr>
<tr>
<td>8</td>
<td>2·4·7·10</td>
</tr>
<tr>
<td>9</td>
<td>2·4·7·10</td>
</tr>
<tr>
<td>10</td>
<td>2·4·7·10</td>
</tr>
</tbody>
</table>

What will this call to fac(3) print?

[and by which line?]
circle the line #

hand these in and up!
Smooth functioning...

What does `fac(3)` do?

<table>
<thead>
<tr>
<th>x is 3</th>
<th>2\cdot4\cdot7\cdot10</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:3 Res:6</td>
<td>2\cdot4\cdot7\cdot10</td>
</tr>
</tbody>
</table>

What's printed... circle the line #

What does `fac(3)` do?

`fac(3)`

1. `def fac(x):`
2. `x is 3`  
3. `x:3 Res:6`
4. `2\cdot4\cdot7\cdot10`
Rough execution...!

What does \texttt{fac(3)} do?

\textbf{Goal:} To automatically assemble Python functions into Hmmm instructions!

\textbf{Challenge:} Functions clobber each other's stuff!
Rough execution...!

```
def fac(x):
    """factorial w/ printing""
    print("x is", x)
    if x == 0:
        print("x:", x, "Res: 1")
        return 1
    else:
        print("Next: fac(\(x-1\))")
        smaller = fac(x-1)
        result = x * smaller
        print("x:", x, "Res:", result)
        return result
```

```
0 fac(3)
```

<table>
<thead>
<tr>
<th>x</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

What does `fac(3)` do?

```
x is 3  2 · 4 · 7 · 10
Next: fac(2)  2 · 4 · 7 · 10
x is 2  2 · 4 · 7 · 10
Next: fac(1)  2 · 4 · 7 · 10
x is 1  2 · 4 · 7 · 10
Next: fac(0)  2 · 4 · 7 · 10
x is 0  2 · 4 · 7 · 10
x:0 Res:1  2 · 4 · 7 · 10
x:1 Res:1  2 · 4 · 7 · 10
x:2 Res:2  2 · 4 · 7 · 10
x:3 Res:6  2 · 4 · 7 · 10
```
but... functions *clobber* stuff!

We need the *earlier* value of x (r1) ... *after* the function call!

```python
def main():
    x = input()
    y = fun(x)
    print(y + x)

def fun(x):
    y = x * (x - 1)
    return y
```

```
0:   read r1:  
1:   calln r14 4
2:     add r13 r13 r1
3:     write r13
4:     copy r13 r1
5:     addn r1 -1
6:     mul r13 r13 r1
7:     jump r14
```

We need the *earlier* value of x (r1) ... *after* the function call!
functions != instructions

if we wanted

\[ y = \text{fun}(x) \]

\[ \text{print } y + x \]

We'd need the \textit{earlier} value of \( x \) (r1) ... \textit{after} the function call!
There is only ONE set of registers...

... but we want LOTS of function calls.

Functions will **overwrite** registers

We want it to **SEEM** like each function has its own registers!
There is only ONE set of registers...

... but we want LOTS of function calls.

We want it to SEEM like each function has its own registers!

<table>
<thead>
<tr>
<th>64-bit register</th>
<th>Lower 32 bits</th>
<th>Lower 16 bits</th>
<th>Lower 8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>eax</td>
<td>ax</td>
<td>al</td>
</tr>
<tr>
<td>rbx</td>
<td>ebx</td>
<td>bx</td>
<td>bl</td>
</tr>
<tr>
<td>rcx</td>
<td>ecx</td>
<td>cx</td>
<td>cl</td>
</tr>
<tr>
<td>rdx</td>
<td>edx</td>
<td>dx</td>
<td>dl</td>
</tr>
<tr>
<td>rsi</td>
<td>esi</td>
<td>si</td>
<td>sil</td>
</tr>
<tr>
<td>rdi</td>
<td>edi</td>
<td>di</td>
<td>dil</td>
</tr>
<tr>
<td>rbp</td>
<td>ebp</td>
<td>bp</td>
<td>bpl</td>
</tr>
<tr>
<td>rsp</td>
<td>esp</td>
<td>sp</td>
<td>spl</td>
</tr>
<tr>
<td>r8</td>
<td>r8d</td>
<td>r8w</td>
<td>r8b</td>
</tr>
<tr>
<td>r9</td>
<td>r9d</td>
<td>r9w</td>
<td>r9b</td>
</tr>
<tr>
<td>r10</td>
<td>r10d</td>
<td>r10w</td>
<td>r10b</td>
</tr>
<tr>
<td>r11</td>
<td>r11d</td>
<td>r11w</td>
<td>r11b</td>
</tr>
<tr>
<td>r12</td>
<td>r12d</td>
<td>r12w</td>
<td>r12b</td>
</tr>
<tr>
<td>r13</td>
<td>r13d</td>
<td>r13w</td>
<td>r13b</td>
</tr>
<tr>
<td>r14</td>
<td>r14d</td>
<td>r14w</td>
<td>r14b</td>
</tr>
<tr>
<td>r15</td>
<td>r15d</td>
<td>r15w</td>
<td>r15b</td>
</tr>
</tbody>
</table>

Even 2018's chips have very few registers...!
```python
def main():
    r1 = input()
    r13 = fun(r1)
    print(r13)

def fun(r1):
    r13 = r1*(r1-1)
    return r13
```

**Problem**
Functions clobber stuff!

**Solution**
CALL our function and let it clobber stuff!

There's no stopping the destruction...

*let's just keep it all in one place!*

College administrations like this approach, too...!
```python
def main():
    r1 = input()
    r13 = fun(r1)
    print(r13)

def fun(r1):
    r13 = r1*(r1-1)
    return r13
```

**Problem**
Functions clobber stuff!

**Solution**
CALL our function and let it clobber stuff!

There's no stopping the destruction... *let's just keep it all in one place!*
```
import.

main:
    r1 = input()
    r13 = fun(r1)
    print r13

fun:
    r13 = r1*(r1-1)
    return r13
```

Solution:

1. **PUSH** important data to the stack!
   - r1 gets stored...
   - r14 gets stored...

2. **call** our f'n and let it destroy stuff!

3. **jump** back to return

4. **POP** important data from the stack!
   - r14 is back as before!
   - r1 is back as before!
   - r13 now holds the result...

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

**System instructions**

**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>mov</td>
<td></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></td>
</tr>
<tr>
<td>mul rX rY rZ</td>
<td>Set rX = rY * rZ (returns the remainder of integer</td>
</tr>
<tr>
<td>div rX rY</td>
<td></td>
</tr>
<tr>
<td>mod rX rY rZ</td>
<td>Set rX = rY % rZ (returns the remainder of integer</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>jeq x rX N</td>
<td>If rX == 0, then jump to line N</td>
</tr>
<tr>
<td>jne rX N</td>
<td>If rX != 0, then jump to line N</td>
</tr>
<tr>
<td>jgt x rX N</td>
<td>If rX &gt; 0, then jump to line N</td>
</tr>
<tr>
<td>jlt x rX N</td>
<td>If rX &lt; 0, then jump to line N</td>
</tr>
<tr>
<td>call n 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>push rX rY</td>
<td>Store contents of register rX into memory address by reg. rY</td>
</tr>
<tr>
<td>pop rX rY</td>
<td>Load contents of register rX from memory address by reg. rY</td>
</tr>
<tr>
<td>loadn rX N</td>
<td>Load contents of register rX from memory address</td>
</tr>
<tr>
<td>store rX N</td>
<td>Store contents of register rX into memory address</td>
</tr>
</tbody>
</table>

**Hmmm**

the complete reference

**pushr r1 r15**

**popr r1 r15**
**pushr stores TO memory**

- **CPU**
  - \( r1 \): Input value: \( x \)
  - \( r14 \): Return address: 47
  - \( r15 \): Stack pointer

- **RAM**
  - Instruction set:
    - `setn r1 3`
    - `setn r14 47`
    - `setn r15 42`
    - `pushr r1 r15`
    - `pushr r14 r15`
    - `addn r1 -1`
    - `setn r14 9001`
    - `popr r14 r15`
    - `popr r1 r15`

- **Stack**
  - `3`

**pushr r1 r15** pushes \( r1 \) into \( r15 \)'s MEMORY LOCATION (not into \( r15 \) itself)

and adds 1 to \( r15 \)
popr loads FROM memory

CPU

Input value: x

Return address: 47

Stack pointer

RAM

<table>
<thead>
<tr>
<th></th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>setn r1 3</td>
<td>setn r14 47</td>
<td>setn r15 42</td>
<td>pushr r1 r15</td>
<td>pushr r14 r15</td>
<td>addn r1 -1</td>
<td>setn r14 9001</td>
<td>popr r14 r15</td>
<td>popr r1 r15</td>
</tr>
<tr>
<td>0</td>
<td>setn r1 3</td>
<td>setn r14 47</td>
<td>setn r15 42</td>
<td>pushr r1 r15</td>
<td>pushr r14 r15</td>
<td>addn r1 -1</td>
<td>setn r14 9001</td>
<td>popr r14 r15</td>
<td>popr r1 r15</td>
</tr>
<tr>
<td>1</td>
<td>47</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9001</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>pushr r1 r15</td>
<td>pushr r14 r15</td>
<td>addn r1 -1</td>
<td>setn r14 9001</td>
<td>popr r14 r15</td>
<td>popr r1 r15</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

first subtracts 1 from r15 then

pops INTO r1 the data in r15's MEMORY LOCATION (not into r15 itself)
popr loads FROM memory

With calln, jumper, pushr, and popr, we can now implement any function – Python or another language – in assembly...

Let's see it in action...
```python
x = input()
y = fac(x)
print(y)

def fac(x):
    if x == 0:
        return 1
    else:
        RES = fac(x-1)
        return x*RES
```

**Python**

**Input, Call, and Output**

```plaintext
x = input()  →  y = fac(x)  →  print(y)
```

**Hmmm**

```plaintext
00 read r1
01 setn r15 42
02 nop
03 calln r14 7
04 nop
05 write r13
06 halt
```

**What's the machine doing here?**

1. **Base Case**
   - `if x==0:`
     - `return 1`

2. **Recursive Call**
   - `RES = fac(x-1)`
   - `return x*RES`

3. ** Recursive Step**
   - `pushr r1 r15`
   - `calln r14 7`
   - `popr r14 r15`
   - `pushr r14 r15`
   - `calln r14 7`
   - `popr r14 r15`
   - `pushr r14 r15`
   - `calln r14 7`
   - `popr r14 r15`
   - `mul r13 r1 r13`
   - `jumpr r14`

4. **Factorial Function**
   - `def fac(x):`
   - `if x==0:`
   - `return 1`
   - `else:`
   - `RES = fac(x-1)`
   - `return x*RES`

5. **Base Case**
   - `07 jnezn r1 11`
   - `08 setn r13 1`
   - `09 nop`
   - `10 jumpr r14`

6. **Recursive Call**
   - `11 pushr r1 r15`
   - `12 pushr r14 r15`
   - `13 addn r1 -1`
   - `14 nop`
   - `15 calln r14 7`
   - `16 nop`
   - `17 popr r14 r15`
   - `18 popr r1 r15`
   - `19 mul r13 r1 r13`
   - `20 jumpr r14`

---

**Note:** The machine code shown is based on the Python code and represents what the machine would do to execute the factorial function. The specific instructions shown are for illustrative purposes and may not correspond exactly to the machine architecture described.
For an input of 0, trace what happens here...

```
00 read r1
01 setn r15 42
02 nop
03 calln r14 7
04 nop
05 write r13
06 halt
07 jnezn r1 11
08 setn r13 1
09 nop
10 jump r14
11 pushr r1 r15
12 pushr r14 r15
13 addn r1 -1
14 nop
15 calln r14 7
16 nop
17 popr r14 r15
18 popr r1 r15
19 mul r13 r1 r13
20 jump r14
```

Main Memory
"the stack"

Yesterday I'd never heard of Recursive Assembly, but today I r1.
For an input of 0, trace what happens here ...

00 read  r1
01 setn  r15  42
02 nop
03 calln r14  7
04 nop
05 write r13
06 halt

07 jnezn r1  11
08 setn  r13  1
09 nop
10 jumper r14

11 pushr r1  r15
12 pushr r14 r15
13 addn  r1  -1
14 nop
15 calln r14  7
16 nop
17 popr  r14  r15
18 popr  r1  r15
19 mul   r13 r1  r13
20 jumper r14

You try it

CPU + Registers
Main Memory
"the stack"

Input: x
return value (the "result")
return address (line #)
the stack pointer

0, 1, 2, 3,  7, 8, 9, 10,  4, 5, 6
It's easy to get lost! Follow these line numbers, at the left, to stay on track...
For an input of 3, trace what happens here...

00 read r1
01 setn r15 42
02 nop
03 calln r14 7
04 nop
05 write r13
06 halt

07 jnezn r1 11
08 setn r13 1
09 nop
10 jump r14
11 pushr r1 r15
12 pushr r14 r15
13 addn r1 -1
14 nop
15 calln r14 7
16 nop
17 popr r14 r15
18 popr r1 r15
19 mul r13 r1 r13
20 jump r14

CPU + Registers

Main Memory
"the stack"

Input: x

r1

r13

r14

r15

Running fac(3) in Hmmm

It's easy to get lost! Follow these line numbers, below, to stay on track...
For an input of 3, trace what happens here...

00 read  r1
01 setn  r15 42
02 nop
03 calln r14 7
04 nop
05 write r13
06 halt

setup

base case

factorial

recursive step

CPU + Registers

Main Memory

"the stack"

07 jnezn r1  11
08 setn  r13 1
09 nop
10 jumpr r14
11 pushr r1 r15
12 pushr r14 r15
13 addn  r1 -1
14 nop
15 calln r14 7
16 nop
17 popr  r14 r15
18 popr  r1  r15
19 mul   r13 r1 r13
20 jumpr r14

Input: x

r1

r13

r14

r15

let's ignore this: "it does the right thing!"

return value (the "result")

return address (line #)

the stack pointer

Finished? Extra! Change the code so that it instead raises r2 to the r1 power — you need to change very few lines!

hw6pr4

It's easy to get lost! Follow these line numbers, below, to stay on track... 0, 1, 2, 3, 7, 11, 12, 13, 14, 15, 7, 11, 12, 13, 14, 15, 7, 8, 9, 10, 16, 17, 18, 19, 20, 16, 17, 18, 19, 20, 16, 17, 18, 19, 20, 4, 5, 6
For an input of 3, trace what happens in registers and memory (the stack) as this program runs.

```assembly
00 read r1
01 setn r15 42
02 call r14 5
03 jumpn 21
04 nop
05 jnezn r1 8
06 setn r13 1
07 jump r14
08 storer r1 r15
09 addn r15 1
10 storer r14 r15
11 addn r15 1
12 addn r1 -1
13 call r14 5
14 addn r15 1
15 load r14 r15
16 addn r15 -1
17 load r1 r15
18 mul r13 r13 r1
19 jump r14
20 nop
21 write r13
22 halt
```

**CPU + Registers**

- r1
- r13
- r14
- r15

**Main Memory**

Input: x

- return value (the "result")
- return address (line #)

**Yikes!**

Finished? **Extra!**Change the code so that it instead raises r2 to the r1 power — you need to change **very few lines** !

hw6pr4

It's easy to get lost! Follow these line numbers below to stay on track...
For an input of $3$, trace what happens here ...

00 read    r1
01 setn    r15  42
02 nop
03 calln   r14  7
04 nop
05 write   r13
06 halt

CPU + Registers

Main Memory
"the stack"

07 jnezn    r1    11
08 setn    r13  1
09 nop
10 jump r   r14

factorial

r1

Input: x

r13

return value (the "result")

r14

return address (line #)

r15

the stack pointer

11 pushr   r1    r15
12 pushr   r14   r15
13 addn    r1   -1
14 nop
15 calln   r14   7
16 nop
17 popr    r14   r15
18 popr    r1    r15
19 mul     r13   r1   r13
20 jump r   r14

Finished? Extra! Change the code so that it instead raises r2 to the r1 power – you need to change very few lines!

hw6pr4
How much will you need to change to implement a recursive power function?

Programmatically!
Start with this recursive factorial function...

```
fac( N )
```

Note that fac's input is similar to the POWER (not the base).

How much will you need to change to implement a recursive power function?

```
power( B, N )
```

Hint for hw7pr4...
Strategy: **standardize** registers

Simplify by having a **standard place** for **standard data**.

**r1**
- *input(s)* will be in r1, r2, ...

**r13**
- *the return value* (result) will be in r13

**r14**
- *the return address* will be in r14

**r15**
- *the "stack pointer"* will be in r15

This is the NEXT location in memory for storing data

Organizationally
x = input()
y = fac(x)
print y

def fac(x):
    if x==0:
        return 1
    else:
        RES = fac(x-1)
        return x*RES
Hmmmwork #7

Countdown Lab!

hw7pr1.hmmm  RandoHmmm #s

hw7pr2.hmmm  Hmmm power!

hw7pr3.hmmm  Fibonacci
1, 1, 2, 3, 5, 8, ...

hw7pr4.hmmm  Recursive power!
you'll start from the functional factorial example we used today

hw7pr5.py  Python loops
we'll look at these next...