Why Assembly Language?

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

Jon Von Neumann, advisor

Don Gillies, assembler

Instructions vs. Functions

Grace Hopper

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

The first bug?

I’m glad it’s not called demotbing.
**Goal:** To automatically assemble Python functions into Hmmm instructions...

**Approach:** Use lots of conventions...!

Get into a rut -- and stay there! - V. Rokhlin

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

def fun( x ):
    y = x*(x-1)
    return y
```
functions vs. instructions

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

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

Fun: Python

1-1 correspondence is very rare!

Instructions: Hmmm

---

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 jmpn 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 r13 r1
20 jmpn r14
```

What's the machine doing here?
Our example...

**fac(3) → 6**

Smooth functioning...

What does **fac(3)** do?

Rough execution...!

What does **fac(3)** do?

**Goal**: To automatically assemble Python functions into Hmmm instructions!

**Challenge**: Functions clobber each other's stuff!
but... functions clobber stuff!

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

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

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

functions != instructions

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

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

We'd need the earlier value of x (r1) … 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!

Even 2018's chips have very few registers...!
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!...

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

Hmmm the complete reference

pushr stores TO memory

CPU

Input value: x
Return address: 47
Stack pointer

RAM

setn r1 3
setn r14 47
setn r15 42
pushr r1 r15
pushr r14 r15
pushr r14 r15
popr r14 r15
popr r1 r15

pushr r1 r15
pushr r1 r15
pushr r1 r15
pushr r1 r15

setn r14 9001
popr r14 r15
popr r1 r15

and adds 1 to r15

Storing in memory (RAM)

interact with memory (RAM)

popr r1 r15
popr r1 r15
popr r1 r15
popr r1 r15

pushr r1 r15
pushr r1 r15
pushr r1 r15
pushr r1 r15

setn r1 3
setn r14 47
setn r15 42
pushr r1 r15
pushr r14 r15
pushr r14 r15
popr r14 r15
popr r1 r15

setn r14 9001
popr r14 r15
popr r1 r15

and adds 1 to r15
popr loads FROM memory

CPU

RAM

popr r1 r15

first subtracts 1 from r15 then
pops INTO r1 the data in r15's MEMORY LOCATION (not into r15 itself)

STACK

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

CPU + Registers

Main Memory

"the stack"

r1

Input: x

r13

return value (the "result")

r14

base case

r15

return address (line #)

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

running fac(0) in Hmmm

Python

Input, Call, and Output

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

Hmmmm

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

What's the machine doing here?

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

CPU + Registers

Main Memory

"the stack"

r1

Input: x

r13

return value (the "result")

r14

base case

r15

return address (line #)

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

Names(s) ________________________ Quiz

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

**power( B, N )**

Programmatically!

Start with this recursive factorial function...

**fac( N )**

Strategy: standardize registers

- **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

Organizationally

**Journey**: instructions to functions...

**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
```

**Assembly**

- Function call!
  - **push** everything to memory
  - make the function call
  - **destroy data as needed**
    - **jumper** to return
    - **pop** all back from memory

**Conceptually**

Hmmmwork #7

- **hw7pr1.hmmm** Count down
- **hw7pr2.hmmm** Rando
- **hw7pr3.hmmm** Fibonacci
- **hw7pr4.hmmm** Recursive power!
- **hw7pr5.py** Python loops
```
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

Names(s) ________________

What will this call to \texttt{fac(3)} print?

[ and by \texttt{which} line? ]

<table>
<thead>
<tr>
<th>Line</th>
<th><code>x is 3</code></th>
<th><code>2 · 4 · 7 · 10</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td>11</td>
<td><code>x:3</code></td>
<td><code>Res:6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 · 4 · 7 · 10</td>
</tr>
</tbody>
</table>

\textbf{x:3 Res:6} \textcolor{red}{2 · 4 · 7 · 10}