CS 42 Today: Beyond Binary
Midterm Logistics

• First Midterm: Next week.
• Review session on Thursday.

• A6 due Monday night
  Extra credit accepted w/o euro through Friday
  E-mail me if you turn it in “late”

• We have to make a choice:
  – 75-minute in-class exam Tuesday
  – Or, longer (2-hour) take-home exam, taken sometime between Friday 5pm and Wednesday 9am.
The Mark 1

Howard Aiken, Grace Hopper at Harvard

relay-based computer

Pentium 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!

The Eniac, 1946

Eniac's developers

J. Prespert Eckert

John W. Mauchly
The Eniac, 1946

Eniac's Programmers!

Rear left to right: Kathy Kleiman (author), Jean Bartik, Marlyn Meltzer, Kay Mauchly Antonelli Front: Betty Holberton. Not pictured: Ruth Lichterman Teitelbaum and Frances Bilas Spence

And I thought programming in Hmmm was painful!
More concrete

- Digital circuit design
- Computer Architecture
- Machine Language
- Assembly Language
- Loops in AL
- Functions and recursion in AL

More abstract

- Last Thursday
- Today!
- Done
- Today and Thursday
A 1-bit Memory

This stuff is truly unforgettable!

\[ \text{OR} + \text{NOT} = \text{NOR} \]
The D latch

inputs

D  data

"strobe"

These ANDs ensure
if D == 1, S is 1
if D == 0, R is 1

The SR latch

S sets Q to 1.
R resets Q to 0.

1 bit of memory!
The D latch

- **D** (data)
- "strobe"

The data can be ready any time (or NOT!) But the output can change only while the strobe is on!

These ANDs ensure:
- if $D = 1$, $S$ is 1
- if $D = 0$, $R$ is 1

1 bit of memory!

The SR latch

- **S** sets $Q$ to 1.
- **R** resets $Q$ to 0.
The D latch

- **D** input (data)
- "strobe" input

The SR latch

- **AND** gate
- **NOR** gate

1 bit of memory!

its circuit element

1 bit of memory!
Slightly Fancier Memory Elements

Random Access Memory

"640K ought to be enough for anybody"

- not Bill Gates!

A 512KB RAM
(About 4.2 million bits)

A 1GB RAM
(About 8.9 billion bits)
Random Access Memory

We can use data latches to create a 12 nG bit RAM!

**Inputs**
- 3 data input bits
- 2 data address bits
- write enable line
- read enable line

**Outputs**
- 3 data output bits
- 12 bits of RAM

3 bits stored at location 00
3 bits stored at location 01
3 bits stored at location 10
3 bits stored at location 11
It’s totally FAB

How are all of these gates actually realized?

Layering
- mask w/ circuit (or doodle)
- photoresist
- conductor
- silicon dioxide
- silicon wafer

Lithography

Etching

Cleaning

x 200 or so
Intel 4004
1971
108 KHz clock
4-bit processor
2000 transistors
2000
Intel Pentium P4
1.5 GHz clock
32-bit processor
42 million transistors
The enemy ...

Keeping particles out of the fabrication process is paramount!

Nature still impresses...

Ouch!

Pollen on a memory chip

A ruthless invader...
What to do with all that extra silicon?

The "silicon zoo": micro.magnet.fsu.edu/creatures/index.html
What to do with all that extra silicon?

The "silicon zoo": micro.magnet.fsu.edu/creatures/index.html
<table>
<thead>
<tr>
<th>Assembly Language</th>
<th>Machine Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loadn r0 42</code></td>
<td><code>0011000000101010</code></td>
</tr>
<tr>
<td><code>add r2 r1 r0</code></td>
<td><code>0111001000010000</code></td>
</tr>
<tr>
<td><code>sub r5 r4 r3</code></td>
<td><code>1000010101000011</code></td>
</tr>
<tr>
<td><code>mul r11 r9 r8</code></td>
<td><code>1001101110011000</code></td>
</tr>
<tr>
<td><code>div r6 r7 r12</code></td>
<td><code>1010011001111100</code></td>
</tr>
<tr>
<td><code>halt</code></td>
<td><code>0000000000000000</code></td>
</tr>
</tbody>
</table>

*register-level programming*  
*bit-level programming*
the compiler

a program that translates from human-usable language into assembly language and machine language

\[
x = 6 \\
y = 7 \\
z = x*y \\
print z
\]

the code
What's in an instruction

One possible (simplified, not Hmmm!) instruction set architecture

Instructions are just 1s and 0s… it is up to the computer architect to give the 1s and 0s meaning.

```
00011100
```

which instruction?

- 00 add
- 01 subtract
- 10 multiply
- 11 divide

Source register 1
AND destination register

"Add the value in register 3 to the value in register 4 and put the result back in register 3"
What's in an instruction

One possible instruction set architecture

which instruction?

00 add
01 subtract
10 multiply
11 divide

00011100

Source register 1

AND destination register

Source register 2

"Add the value in register 3 to the value in register 4 and put the result back in register 3"

The computer architect builds circuitry to make this computation happen… but what does that circuitry look like…?
A 4-instruction 2-bit ALU (Arithmetic-Logic Unit)

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>x1</th>
<th>x2</th>
<th>y1</th>
<th>y2</th>
<th>out1</th>
<th>out2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

00 add
01 subtract
10 multiply
11 divide
A Closer Look

The RAM (memory) contains the program (and possibly some data as well)

Central Processing Unit (CPU)

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>00000010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Register</td>
<td>00011010</td>
</tr>
<tr>
<td>Register 0</td>
<td>00000010</td>
</tr>
<tr>
<td>Register 1</td>
<td>00000010</td>
</tr>
<tr>
<td>Register 2</td>
<td>00000001</td>
</tr>
<tr>
<td>Register 3</td>
<td>00000011</td>
</tr>
</tbody>
</table>

Memory Location

<table>
<thead>
<tr>
<th>Binary</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

8 bit address

8 bit data in

Read

Write

Memory

The RAM (memory) contains the program (and possibly some data as well)

8 bit data out

00 add
01 subtract
10 multiply
11 divide

0110010
001100
10001100

The RAM (memory) contains the program (and possibly some data as well)
A Computer!

The RAM (memory) contains the program (and possibly some data as well)

Central Processing Unit (CPU)

Program Counter
Instruction Register
Register 0
Register 1
Register 2
Register 3

8 bit address
8 bit data in
Read
Write

Memory Location

Binary          Base 10
00000000       0
00000001       1
00000010       2
00000011       3
00000100       4
11111111       255

00  add
01  subtract
10  multiply
11  divide

00110010
00011010
10001100

A Computer!

8 bit
address

The RAM (memory) contains the program (and possibly some data as well)
A Computer!

The RAM (memory) contains the program (and possibly some data as well).

Central Processing Unit (CPU)

Program Counter: 00000010
Instruction Register: 00011010
Register 0: 00000010
Register 1: 00000010
Register 2: 00000001
Register 3: 00000011

Memory Location

<table>
<thead>
<tr>
<th>Binary</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

1. Read
2. Write

8 bit address

8 bit data in

8 bit data out
A Computer!

The RAM (memory) contains the program (and possibly some data as well)

Central Processing Unit (CPU)

Program Counter: 00000010
Instruction Register: 00011010
Register 0: 00000010
Register 1: 00000010
Register 2: 00000001
Register 3: 00000011

Memory Location

<table>
<thead>
<tr>
<th>Binary</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

8 bit address

00110010
00011010
10001100

8 bit data in
1
Read

8 bit data out
10001100

Write

11111111
255

add
subtract
multiply
divide

00
01
10
11
A Computer!

The RAM (memory) contains the program (and possibly some data as well)

Central Processing Unit (CPU)

Program Counter: 00000010
Instruction Register: 00011010
Register 0: 00000010
Register 1: 00000010
Register 2: 00000001
Register 3: 00000011

Memory Location

<table>
<thead>
<tr>
<th>Binary</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

8 bit address: 00110010
8 bit data in: 001100
8 bit data out: 10001100

Read
Write

10001100
A Computer!

The RAM (memory) contains the program (and possibly some data as well)

Central Processing Unit (CPU)

Program Counter: 00000010
Instruction Register: 10001100
Register 0: 00000010
Register 1: 00000010
Register 2: 00000001
Register 3: 00000011

8 bit address

Memory Location

Binary | Base 10
--- | ---
00000000 | 0
00000001 | 1
00000010 | 2
00000011 | 3
00000100 | 4
11111111 | 255

The RAM (memory) contains the program (and possibly some data as well)
A Computer!

The RAM (memory) contains the program (and possibly some data as well)

Central Processing Unit (CPU)

Program Counter: 00000010
Instruction Register: 10001100

Memory Location

<table>
<thead>
<tr>
<th>Binary</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
</tr>
</tbody>
</table>

- 111010: 001
- 001100: 01

The RAM (memory) contains the program and possibly some data as well.

Memory Location

<table>
<thead>
<tr>
<th>Binary</th>
<th>Base 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110010</td>
<td></td>
</tr>
<tr>
<td>00011100</td>
<td></td>
</tr>
<tr>
<td>10001100</td>
<td></td>
</tr>
</tbody>
</table>

11111111: 255

8 bit address

Read
Write

8 bit data in

8 bit data out
A Computer!

The RAM (memory) contains the program (and possibly some data as well)

Central Processing Unit (CPU)

Program Counter

Instruction Register

00000010

1001100

Register 0

Register 1

Register 2

Register 3

8 bit address

1

8 bit data in

Read

Write

Memory

The fetch-execute cycle

Memory Location

Binary          Base 10
00000000        0
00000001        1
00000010        2
00000011        3
00000100        4
11111111        255
The instruction 00 = (add)

Argument 1: the src and dst register
   Reg3

Argument 2: The 2nd src register
   Reg4

The old value of Reg3 = 7
The new value of Reg3 = 11

D Latch

A Von Neumann machine
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

register 0 is “hard-wired” to store 0

r0 0

r1

r2

... 16 registers, each 16 bits

they can hold values from -32768 up to 32767

r15

---

0 read r1
1 mul r2 r1 r1
2 add r2 r2 r1
3 write r2
4 halt
5
6
7...
255 memory locations of 16 bits
NOTE:
The Hmmm "machine" is does not exist in hardware.
It exists in simulation only (in Python).
After next Monday, it may also exist in Logisim…
(define (main)
  (display (dist2 2 3)))

(define (dist2 x y)
  (+ (sq x) (sq y)))

(define (sq z)
  (* z z))
Option 1: Inlining

(define (main)
  (display (dist2 2 3)))

(define (dist2 x y)
  (+ (sq x) (sq y)))

(define (sq z)
  (* z z))

(define (main)
  (display (+ (* 2 2) (* 3 3))))

(define (dist2 x y)
  (display (+ (* x x) (* y y))))
Option 2: Call And Return

Each function is a single block of machine code

When dist2 calls sq:
  code for dist2 jumps to the code for sq
When sq is done:
  jump back into the middle of dist2
  (immediately after the jump to sq)
Calling Conventions

How does sq know what “the input” is?
How does dist2 know where to jump to?
How does sq know where to jump back to?
How do we keep sq from overwriting the registers that dist2 cares about?

```
dist2
# call the sq function
07…
# call the sq function
08 …
# sum the results
09 add …
# return to caller
10 …
```

```
sq
# square the input
20 mul …
# return to caller
21 …
```
Remember function **stacking**?

\[
\text{(define (demo } x) \\
\quad (+ x (f x)))
\]

\[
\text{(define (f } x) \\
\quad (+ (* 11 (g x)) \\
\quad \quad (g (/ x 2))))
\]

\[
\text{(define (g } x) \\
\quad (* -1 x))
\]

What is demo(-4) ?

\[\text{demo} \quad x = -4 \\
\text{return } -4 + f(-4)\]

\[\text{f} \quad x = -4 \\
\text{return } 11 \times 4 + g(-4/2)\]

\[\text{g} \quad x = -2 \\
\text{return } -1 \times -2 \rightarrow 2\]

"The stack"

Remembers separate functions' values for variables...

Remembers where to send results back to...
HMMM Conventions

Register agreement + the STACK

OUR "agreements":

- function inputs: $r_1, r_2, \ldots$
- return value: $r_{13}$
- return address: $r_{14}$
- stack pointer: $r_{15}$

When a function calls another function it will…
1. Save all of its important data in the stack
2. Put the inputs in $r_1, r_2, r_3\ldots$
3. Call the function…
4. (automatically) storing its return address in $r_{14}$
5. And expect the results in $r_{13}$
6. Retrieve its data from the stack
Where is the stack?

Hmmm CPU

r1

The program

0
read r1
1
read r2
2
mul r3 r2 r1
3
write r3
4
halt
5

Hmmm RAM

r15

... The STACK!

42

43

...
Key Idea: What's in RAM

The program and the stack SHARE the RAM
program = low RAM
stack = high RAM

Hmm RAM

The program

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>1</td>
<td>read r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mul r3 r2 r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>write r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
store goes to memory

Hmmm CPU

r1

r2

r15

Hmmm RAM

0

1

2

3

4

5

read r1

read r2

store r2 43

loadn r15 42

storei r1 r15

storei r1 r15

42

43

...
load comes FROM memory

Hmmm CPU

\[
\begin{array}{ll}
\text{r1} & \begin{array}{cccc}
\star & \star & \star & \star \\
\end{array} \\
\text{r2} & \begin{array}{c}
\star \\
\end{array} \\
\text{r15} & \begin{array}{c}
\ldots \\
42 \\
\ldots \\
\end{array}
\end{array}
\]

Hmmm RAM

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
\ldots \\
42 \\
43 \\
\ldots
\end{array}
\]

- **direct load**
- **indirect load**

**loadi r1 r15**

aliens

42

**load r2 43**

**loadi r1 r15**

**storei r1 r15**

aliens

42

43

\ldots
A function call in Racket:

```scheme
(define (main)
  (let* ((r1 (read-line)))
    (result (demo r1)))
  (display result)))

(define (demo x)
  (+ x (f x)))
```

Hmmm's call operation:

```
call r14 4
```

puts NEXT line # into r14, then jumps to line 4
How functions REALLY work…

I want my input in \( r1 \), and I'll put my output in \( r13 \)

\[
\text{(define (main)} \ \\
\text{ (let* ((r1 (read-line)))} \ \\
\text{ (result (demo r1)))} \ \\
\text{(display result)\n)}
\]

TBD

I want my input in \( r1 \), and I'll put my output in \( r13 \)

\[
\text{(define (demo x)} \ \\
\text{ (+ x (f x)))}
\]

I want my input in \( r1 \), and I'll put my output in \( r13 \)

\[
\text{(define (f x)} \ \\
\text{ (+ (* 11 (g x)))} \ \\
\text{ (g (/ x 2)))})
\]

I want my input in \( r1 \), and I'll put my output in \( r13 \)

\[
\text{(define (g x)} \ \\
\text{ (* -1 x))}
\]
How functions REALLY work…

(define (main)
  (let* ((r1 (read-line)))
    (result (demo r1)))
  (display result)))

00 read r1          # Get the user's input
01 loadn r15 SA     # Store the top of the stack
02 call r14 DEMO    # Call demo, starts at line ??
03 write r13        # Demo has returned,
                    # so just write and halt
04 halt             # ** demo begins here **
                    # First calculate f(x)
                    # Prepare: store data
05 storei r1 r15    # Prepare input: r1 already
                    # contains x
06 addn r15 1       # increment the stack pointer
07 storei r14 r15   # increment the stack pointer
08 addn r15 1       # Prepare input: r1 already
                    # contains x
09 call r14 F       # Call f(x)
10 addn r15 -1      # F has returned. decr sp
11 loadi r14 r15    # Get our return address
12 addn r15 -1      # decrement the stack pointer
13 loadi r1 r15     # Get x back
14 add r13 r13 r1   # r13 = f(x) + x
15 jumpi r14        # Demo is done so return
Let's say SA is 50…
(We can't really know until we know how much code we have)
CPU (registers)

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r13</td>
<td></td>
</tr>
<tr>
<td>r14</td>
<td></td>
</tr>
<tr>
<td>r15</td>
<td></td>
</tr>
</tbody>
</table>

Main memory (RAM)

```
00 read r1          # Get the user's input
01 loadn r15 50     # Store the top of the stack
02 call r14 5       # Call demo, starts at line 9
03 write r13        # Demo has returned,
                      # so just write and halt
04 halt

** demo begins here **
# First calculate f(x)
# Prepare: store data
05 storei r1 r15    # Store x on the stack
06 addn r15 1       # Increment stack pointer
07 storei r14 r15   # Store our return address
08 addn r15 1       # increment the stack pointer
                      # Prepare input: r1 already
                      # contains x
09 call r14 F       # Call f(x)
10 addn r15 -1      # F has returned. decr sp
11 loadi r14 r15    # Get our return address
12 addn r15 -1      # decrement the stack pointer
13 loadi r1 r15     # Get x back
14 add r13 r13 r1   # r13 = f(x) + x
15 jumpi r14        # Demo is done so return

... [MORE CODE HERE]
```

Input value: x

Final result - return value - in progress

location / line to return TO

Stack pointer

Main memory (RAM)
00 read r1        # Get the user's input
01 loadn r15 50   # Store the top of the stack
02 call r14 5     # Call demo, starts at line 9
03 write r13      # Demo has returned,
                  # so just write and halt
04 halt           # ** demo begins here **
                  # First calculate f(x)
                  # Prepare: store data
05 storei r1 r15  # Store x on the stack
06 addn r15 1     # Increment stack pointer
07 storei r14 r15 # Store our return address
08 addn r15 1     # increment the stack pointer
                  # Prepare input: r1 already
                  # contains x
09 call r14 F     # Call f(x)
10 addn r15 -1    # F has returned. decr sp
11 loadi r14 r15  # Get our return address
12 addn r15 -1    # decrement the stack pointer
13 loadi r1 r15   # Get x back
14 add r13 r13 r1 # r13 = f(x) + x
15 jumpi r14      # Demo is done so return
                  ... [MORE CODE HERE]
(define (main)
  (let* ((r1 (read-line)))
    (result (demo r1)))
  (display result)))

(define (demo x)
  (+ x (f x)))

(define (f x)
  (+ (* 11 (g x))
      (g (/ x 2)))))

(define (g x)
  (* -1 x))

00 read r1    # Get the user's input
01 loadn r15 50 # Store the top of the stack
02 call r14 5  # Call demo
03 write r13   # Demo has returned,
                # so just write and halt
04 halt

# ** demo begins here **
# First calculate f(x)
# Prepare: store data
05 storei r1 r15  # Store x on the stack
06 addn r15 1  # Increment stack pointer
07 storei r14 r15 # Store our return address
08 addn r15 1  # increment the stack pointer
                # Prepare input: r1 already
                # contains x
09 call r14 16  # Call f(x)
10 addn r15 -1  # F has returned. decr sp
11 loadi r14 r15 # Get our return address
12 addn r15 -1  # decrement the stack pointer
13 loadi r1 r15 # Get x back
14 add r13 r13 r1 # r13 = f(x) + x
15 jumpi r14    # Demo is done so return
(define (main)
  (let* ((r1 (read-line)))
    (result (demo r1)))
  (display result)))

(define (demo x)
  (+ x (f x)))

(define (f x)
  (+ (* 11 (g x))
      (g (/ x 2))))

(define (g x)
  (* -1 x))
Implementing functions

non-destructively!

(0) Use \texttt{r15} as the \textit{stack pointer}.

\begin{itemize}
  \item \texttt{loadn r15 42}
  \item \texttt{storei r1 r15}
  \item \texttt{addn r15 1}
\end{itemize}

(1) Before the function call,

\textbf{Store all valuable data to the stack}

(2) Get \texttt{r1}, (\texttt{r2}), (\texttt{r3}), \ldots ready as function "inputs."

(3) Make the function call.

\textbf{The result, if any, will be in r13.}

(4) After the function call,

\textbf{Load valuable data back from the stack}

\begin{itemize}
  \item \texttt{call r14 #}
  \item \texttt{addn r15 -1}
  \item \texttt{loadi r1 r15}
\end{itemize}
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)  
5 * fac(4)  
4 * fac(3)  
3 * fac(2)  
2 * fac(1)  
1

"The Stack"

Remembers all of the individual calls to fac
Factorial via Recursion...

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

def fac(x):
    """ recursive factorial! ""
    if x == 0:
        return 1
    else:
        REC = fac(x-1)
        return x*REC
```

This is same as `return x*f(x-1)` but done in 2 steps...

```
00 read r1
01 loadn r15 42
02 call r14 5
03 jump 21
04 nop
05 jnez r1 8
06 loadn r13 1
07 jumpi r14
08 storei r1 r15
09 addn r15 1
10 storei r14 r15
11 addn r15 1
12 addn r1 -1
13 call r14 5
14 addn r15 -1
15 loadi r14 r15
16 addn r15 -1
17 loadi r1 r15
18 mul r13 r13 r1
19 jumpi r14
20 nop
21 write r13
22 halt
```