CS 42 Today

Racket

Hmmm...

PC: Program Counter
IR: Instruction Register
RAM
registers
1-bit memory: flip-flops
arithmetic
bitwise functions
logic gates
transistors / switches

Hmmm... What is this all about?

Racket

Harvey mudd miniature machine

How computers dream

```assembly
_main:
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $36, %esp
call __i686.get_pc_thunk.bx
"L000000000001$pb":
  movl $6, -20(%ebp)
  movl $7, -16(%ebp)
  movl -20(%ebp), %eax
  imull -16(%ebp), %eax
  movl %eax, -12(%ebp)
  movl -12(%ebp), %eax
  movl %eax, 4(%esp)
  leal LC0-"L000000000001$pb"(%ebx)
  movl %eax, (%esp)
call L_printf$stub
  addl $36, %esp
  popl %ebx
  popl %ebp
  ret
```

our path...
Von Neumann Architecture

Will the *real* von Neumann please stand up?

instructions executed here

**CPU**
central processing unit

programs stored here

**RAM**
random access memory

A few, fast registers + arithmetic

“Slow” memory no computation

Von Neumann bottleneck
Von Neumann Architecture

<table>
<thead>
<tr>
<th>Instructions executed here</th>
<th>Programs stored here</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td><strong>RAM</strong></td>
</tr>
<tr>
<td>Central processing unit</td>
<td>Random access memory</td>
</tr>
</tbody>
</table>

**Computers require machine language (1s and 0s)**

<table>
<thead>
<tr>
<th>Location</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000100000001</td>
</tr>
<tr>
<td>1</td>
<td>1000001000010001</td>
</tr>
<tr>
<td>2</td>
<td>011001000010001</td>
</tr>
<tr>
<td>3</td>
<td>000001000000010</td>
</tr>
<tr>
<td>4</td>
<td>000000000000000</td>
</tr>
<tr>
<td>5</td>
<td>000000000000000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>255</td>
<td>...</td>
</tr>
</tbody>
</table>

255 total memory locations
Von Neumann Architecture

instructions executed here

CPU
central processing unit

Von Neumann bottleneck

RAM
random access memory

programs stored here

Assembly language is human-readable machine language

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

255 total memory locations
Von Neumann Architecture

CPU
central processing unit

RAM
random access memory

Von Neumann bottleneck

instructions executed here

programs stored here

Each instruction is transferred (in order) to the CPU to be executed

<p>| | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>mul r2 r1 r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>add r2 r2 r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>write r2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>halt</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>
<tr>
<td>255</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

255 total memory locations
The BIG IDEA

• Instructions are simple functions for the computer to perform
  move some data (between registers and/or memory)
  Add some numbers
  Go back to a previous instruction
  Etc.
• We can represent functions in logic
  Truth tables, algebra, logic gates
• We ENCODE instructions as sequences of 0s and 1s…
• …Then we can build HARDWARE to execute these instructions

The instructions are just INPUTS to LOGIC FUNCTIONS!
(The output is the RESULT of executing the instruction)
More abstract

More concrete

Assembly Language

Loops in AL

Functions and recursion in AL

Machine Language

Computer Architecture

Digital circuit design

A translation of the 1s and 0s into something human-readable

how to turn functions into hardware

collections of circuits that can perform all the needed functions to manipulate data

sequences of 1s and 0s that drive the hardware to perform the necessary functions

how to use AL to do stuff that's useful to humans
The Hmmmm Instruction Set

There are 23 different instructions in Hmmmm, each of which accepts between 0 and 3 arguments. Two of the instructions, loadn and addn, accept a signed numerical argument between -128 and 127. The load, store, call, and jump instructions accept an unsigned numerical argument between 0 and 255. All other instruction arguments are registers. In the code below, register arguments will be represented by ‘rX’, ‘rY’, and ‘rZ’, while numerical arguments will be represented by ‘#’. In real code, any of the 16 registers could take the place of ‘rX’ ‘rY’ or ‘rZ’. The available instructions are:

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0000 0000 0000 0000</td>
<td>Halt program</td>
</tr>
<tr>
<td>read rX</td>
<td>0000 0000 0000 0001</td>
<td>Stop for user input, which will then be stored in register rX (input is an integer from -32768 to +32767). Prints “Enter number: ” to prompt user for input</td>
</tr>
<tr>
<td>write rX</td>
<td>0000 0000 0000 0010</td>
<td>Print the contents of register rX on standard output</td>
</tr>
<tr>
<td>jump rX</td>
<td>0000 0000 0000 0011</td>
<td>Set program counter to address in rX</td>
</tr>
<tr>
<td>load rX, #</td>
<td>0001 0000 0000 0000</td>
<td>Load an 8-bit integer (#-128 to +127) into register rX</td>
</tr>
<tr>
<td>store rX, #</td>
<td>0010 0000 0000 0000</td>
<td>Load register rX with memory word at address #</td>
</tr>
<tr>
<td>loadi rX, rY</td>
<td>0100 0000 0000 0000</td>
<td>Load register rX from memory word addressed by rY: rX = memory[rY]</td>
</tr>
<tr>
<td>storei rX, rY</td>
<td>0100 0000 0000 0001</td>
<td>Store contents of register rX into memory word addressed by rY: memory[rY] = rX</td>
</tr>
<tr>
<td>addn rX, #</td>
<td>0101 0000 0000 0000</td>
<td>Add the 8-bit integer (#-128 to +127) to register rX</td>
</tr>
<tr>
<td>add rX, rY, rZ</td>
<td>0110 0000 0000 0000</td>
<td>Set rX = rY + rZ</td>
</tr>
<tr>
<td>mov rX, rY</td>
<td>0110 0000 0000 0000</td>
<td>Set rX = rY</td>
</tr>
<tr>
<td>nop</td>
<td>0110 0000 0000 0000</td>
<td>Do nothing</td>
</tr>
<tr>
<td>sub rX, rY, rZ</td>
<td>0111 0000 0000 0000</td>
<td>Set rX = rY - rZ</td>
</tr>
<tr>
<td>neg rX, rY</td>
<td>0111 0000 0000 0000</td>
<td>Set rX = -rY</td>
</tr>
<tr>
<td>mul rX, rY, rZ</td>
<td>1000 0000 0000 0000</td>
<td>Set rX = rY * rZ</td>
</tr>
<tr>
<td>div rX, rY, rZ</td>
<td>1001 0000 0000 0000</td>
<td>Set rX = rY / rZ</td>
</tr>
<tr>
<td>mod rX, rY, rZ</td>
<td>1010 0000 0000 0000</td>
<td>Set rX = rY % rZ</td>
</tr>
<tr>
<td>jump n</td>
<td>1011 0000 0000 0000</td>
<td>Set program counter to address #</td>
</tr>
<tr>
<td>call rX, #</td>
<td>1011 0000 0000 0000</td>
<td>Set rX to (next) program counter, then set program counter to address #</td>
</tr>
<tr>
<td>jeqz rX, #</td>
<td>1100 0000 0000 0000</td>
<td>If rX = 0 then set program counter to address #</td>
</tr>
<tr>
<td>jnez rX, #</td>
<td>1101 0000 0000 0000</td>
<td>If rX ≠ 0 then set program counter to address #</td>
</tr>
<tr>
<td>jgtz rX, #</td>
<td>1110 0000 0000 0000</td>
<td>If rX &gt; 0 then set program counter to address #</td>
</tr>
<tr>
<td>jltz rX, #</td>
<td>1111 0000 0000 0000</td>
<td>If rX &lt; 0 then set program counter to address #</td>
</tr>
</tbody>
</table>
Hmmm
The Harvey Mudd Miniature Machine

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 0
r1
r2 ...

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

r15

mul r2 r1 r1
add r2 r2 r1
write r2
halt

read r1

255 memory locations of 16 bits

0 1 2 3 4 5 6 ...

255
NOTE:
The Hmmm "machine" is does not exist in hardware.
It exists in simulation only (in Python).
We'll see it in action next time.
Hmmmm

The Harvey Mudd Miniature Machine

**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**

- **r0**: Register 0 is "hard-wired" to store 0
- **r1**
- **r2**
- **r15**: 16 registers, each 16 bits
  - They can hold values from -32768 to 32767

**RAM**

- 255 memory locations of 16 bits
  - 0:
    - `read r1`
  - 1:
    - `mul r2 r1 r1`
  - 2:
    - `add r2 r2 r1`
  - 3:
    - `write r2`
  - 4:
    - `halt`
  - 255:
    - ...
Hmmm: the *fetch - execute* cycle

**CPU**
- Central processing unit
- Initially 0

**Program Counter**
- Holds address of the next instruction
- Initially 0

**Instruction Register**
- Holds the current instruction
- r0: register 0 is “hard-wired” to store 0
- r1: General-purpose register r1
- r2: General-purpose register r2

**RAM**
- Random access memory
  - 0: read r1
  - 1: mul r2 r1 r1
  - 2: add r2 r2 r1
  - 3: write r2
  - 4: halt
Assembly Language

**add r2 r2 r2**  
reg2 = reg2 + reg2  
crazy, perhaps, but surprisingly useful

**sub r2 r1 r4**  
reg2 = reg1 - reg4

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

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

**loadn 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 Racket using Hmmm Assembly?

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

_fetch-execute cycle_
It's not enough!

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

It's all too linear!

"straight-line code"

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

jump!
Hmmm, Let's **jump**!

**CPU**
- **central processing unit**

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

**Instruction Register**
- Holds the current instruction

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

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

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

**RAM**
- **random access memory**

- **0**: `loadn r1 42`
- **1**: `write r1`
- **2**: `addn r1 1`
- **3**: `jump 1`
- **4**: `halt`

Replace 1 with 2? with r1?
jumps

Unconditional jump

```
jump 42
```

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

Conditional jumps

```
jeqz r1 #
jgtz r1 #
jl tz r1 #
jnez r1 #
```

IF

IF r1 == 0 THEN jump to line number #

IF r1 > 0 THEN jump to the location in #

IF r1 < 0 THEN jump to the location in #

IF r1 != 0 THEN jump to the location in #

Indirect jump

```
jumpi r1
```

Jump to the line # stored in reg1!
What is this code computing about its input?
Follow this assembly-language program from top to bottom. Use \( r1 = 42 \) and \( r2 = 5 \).

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?

(2) How could you change this program so that, if the original two inputs were equal, it asked the user for new inputs?

**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 \( r2 = 5 \).

(1) What does this program compute in general?

(2) How could you change this program so that, if the original two inputs were \textit{equal}, it asked the user for new inputs?
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.
Factorial

Hmmm CPU

- **r0**: 0
- **r1**: Input value: x
- **r13**: Final result - in progress

Hmmm RAM

- 0: read r1
- 1: loadn r13 1
- 2: jeqz r1 6
- 3: mul r13 r13 r1
- 4: addn r1 -1
- 5: jump 2
- 6: write r13
- 7: halt

Input value: x
Final result - in progress
What is 42?

Base 10

42

4 tens + 2 ones

123

1 hundred + 2 tens + 3 ones
Base 2

101010

Base 10

42

4 tens + 2 ones

Write 123 in binary...

123

1 hundred + 2 tens + 3 ones

each column represents another power of the base
Counting in base $b$

Count from zero to six in each of the following bases:

Base 2:

Base 3:

What’s the “algorithm” for counting in a general base $b$?
Is There Such a Thing as Base 1?

Unary!

$1^3 \ 1^2 \ 1^1 \ 1^0$

Now we’re using powers of 1 (Weird!)

Are we going to use 0 as our only digit?
Comparing Representations in Different Bases

Consider the number $10^9$ in base 1, 2, 3, 10, and 20:

Base 1: \[11111111111111111111111111111111111111111111\ldots\]

At 10 “1’s” per inch, this will be **1578 miles long**!

Base 2: \[11101110011010110010100000000\]

Base 3: \[2120200200021010001\]

Base 10: \[1000000000\]

Base 20: \[FCA0000\]

What’s the ratio between the lengths of a number in bases $x$ and $y$?
Binary math

Decimal math

tables of basic facts

Addition

Multiplication
Addition

Base 10 Addition

\[
\begin{array}{c c c}
10^2 & 10^1 & 10^0 \\
\hline
4 & 3 & \\
+ & 8 & 9 \\
\end{array}
\]
Addition

Base 10 Addition

\[
\begin{array}{c|c|c|c}
10^2 & 10^1 & 10^0 \\
4 & 3 & \text{Carry} \\
+ & 8 & 9 \\
\hline
& 1 & 2
\end{array}
\]

That’s a “10”
## Base 10 Addition

<table>
<thead>
<tr>
<th>10^2</th>
<th>10^1</th>
<th>10^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Move the “1” to the ten’s place.
## Addition

### Base 10 Addition

<table>
<thead>
<tr>
<th>10^2</th>
<th>10^1</th>
<th>10^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>2</td>
</tr>
</tbody>
</table>

Done!
### Base 10 Addition

<table>
<thead>
<tr>
<th></th>
<th>10^2</th>
<th>10^1</th>
<th>10^0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**Try it in base 2**

### Base 2 Addition

<table>
<thead>
<tr>
<th></th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiplication

Base 10 Multiplication

\[
\begin{array}{c|c|c}
10^2 & 10^1 & 10^0 \\
\hline
3 & 4 & 1 \\
\times & 1 & 0 & 2 \\
\hline
\end{array}
\]
Base 10 Multiplication

\[
\begin{array}{c}
10^2 & 10^1 & 10^0 \\
\hline
3 & 4 & 1 \\
\times & 1 & 0 & 2 \\
\hline
6 & 8 & 2 \\
0 & 0 & 0 \\
+ & 3 & 4 & 1 \\
\hline
3 & 4 & 1
\end{array}
\]
Base 10 Multiplication

\[ \begin{array}{ccc}
10^2 & 10^1 & 10^0 \\
3 & 4 & 1 \\
\times & 1 & 0 & 2 \\
\hline
6 & 8 & 2 \\
0 & 0 & 0 \\
+ & 3 & 4 & 1 \\
\hline
3 & 4 & 7 & 8 & 2 \\
\end{array} \]
"Quiz" ✨ There are 10 kinds of "people" in the universe: those who use binary, and those who do not!

Convert these two binary numbers to decimal:

<table>
<thead>
<tr>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>110011</td>
<td>10001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Convert these two decimal numbers to binary:

<table>
<thead>
<tr>
<th>110011</th>
<th>10001000</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>101</td>
</tr>
</tbody>
</table>

Add these two binary numbers:

\[
\begin{array}{c}
101101 \\
+ \quad 1110 \\
\hline
111010
\end{array}
\]

Multiply these binary numbers:

\[
\begin{array}{c}
101101 \\
* \quad 1110 \\
\hline
1110
\end{array}
\]

\[
\begin{array}{c}
529 \\
+ \quad 742 \\
\hline
1271
\end{array}
\]

*Hint:* Remember this algorithm? It's the same...