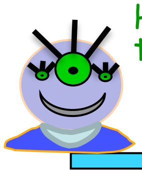


CS 42 Today

Harvey
mudd
miniature
machine

Racket

Hmmm…

PC: Program Counter

IR: Instruction Register

RAM

registers

1-bit memory: flip-flops

arithmetic

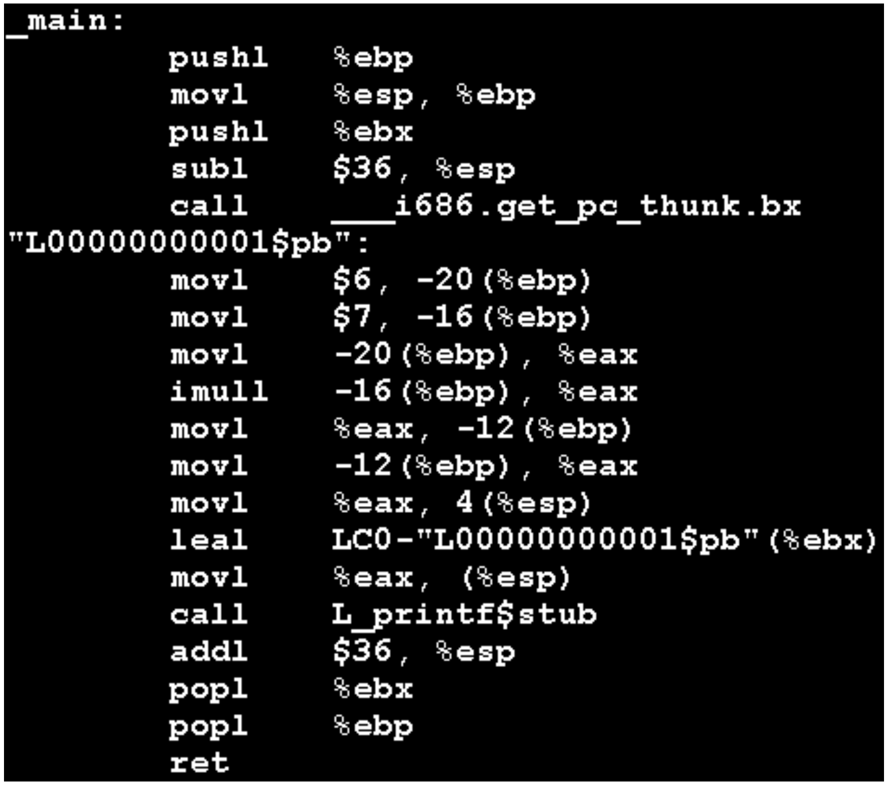
bitwise functions

logic gates

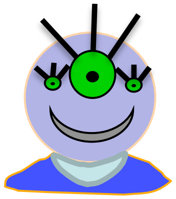
our path…

transistors / switches

How computers dream



```
_main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $36, %esp
        call    ___i686.get_pc_thunk.bx
"L00000000001$pb":
        movl    $6, -20(%ebp)
        movl    $7, -16(%ebp)
        movl    -20(%ebp), %eax
        imull   -16(%ebp), %eax
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        movl    %eax, 4(%esp)
        leal    LC0-"L00000000001$pb"(%ebx)
        movl    %eax, (%esp)
        call    L_printf$stub
        addl    $36, %esp
        popl    %ebx
        popl    %ebp
        ret
```



Von Neumann Architecture

Will the ***real*** von Neumann please stand up?

instructions executed here

programs stored here

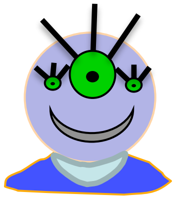
CPU
central processing unit

Von Neumann
bottleneck

RAM
random access memory

A few, fast registers + arithmetic

“Slow” memory no computation



Von Neumann Architecture

instructions executed here

programs stored here

CPU

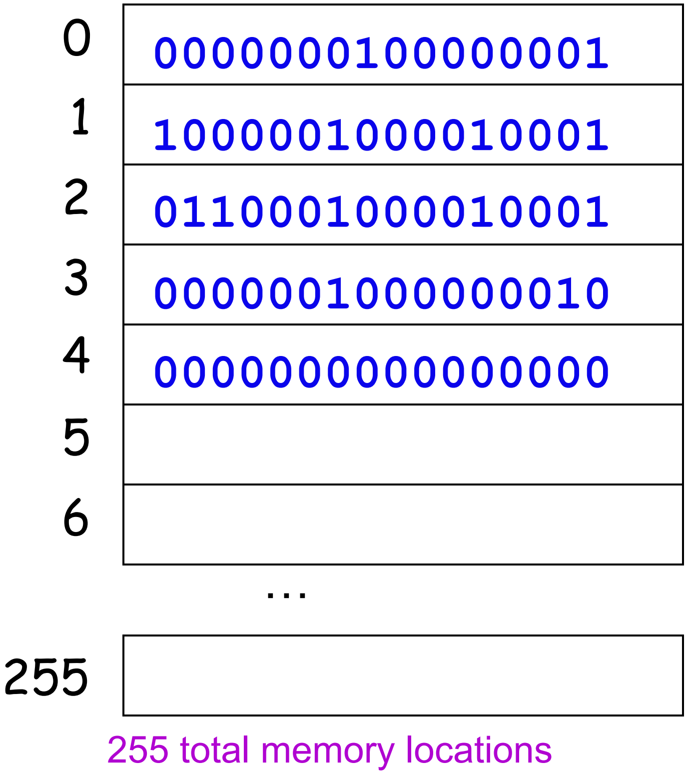
central processing unit

Von Neumann
bottleneck

RAM

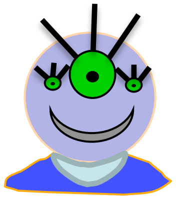
random access memory

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



0	0000000100000001
1	1000001000010001
2	0110001000010001
3	0000001000000010
4	0000000000000000
5	
6	
…	
255	

255 total memory locations



Von Neumann Architecture

instructions executed here

CPU

central processing unit

↔ Von Neumann bottleneck

programs stored here

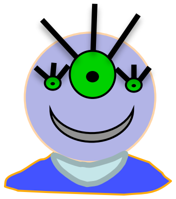
RAM

random access memory

Assembly language is human-readable machine language

0	`read r1`
1	`mul r2 r1 r1`
2	`add r2 r2 r1`
3	`write r2`
4	`halt`
5	
6	
…	
255	

255 total memory locations



Von Neumann Architecture

instructions executed here

programs stored here

CPU
central processing unit

Von Neumann
bottleneck

RAM
random access memory

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

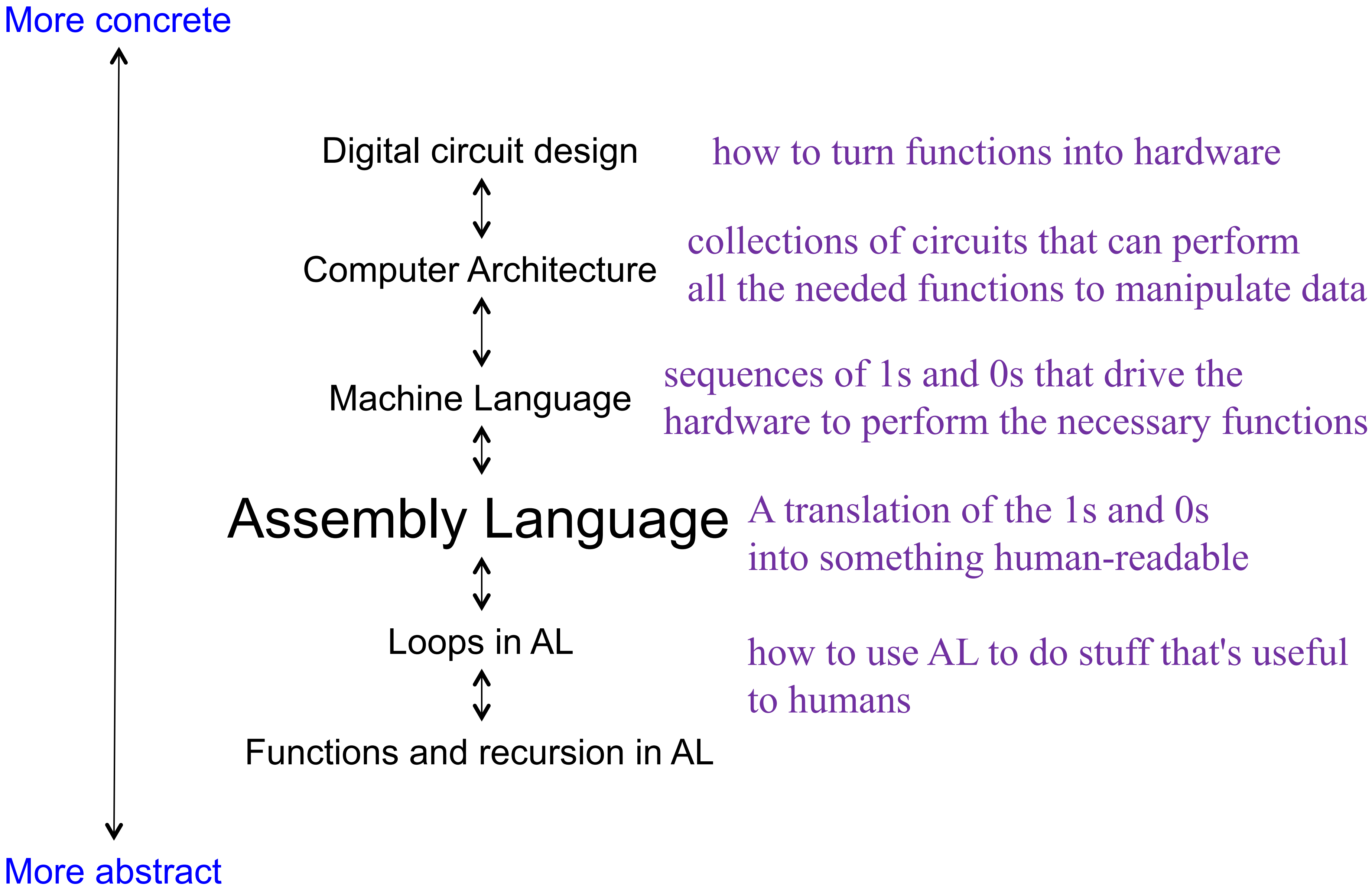
0	`read r1`
1	`mul r2 r1 r1`
2	`add r2 r2 r1`
3	`write r2`
4	`halt`
5	
6	
	…
255	

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)

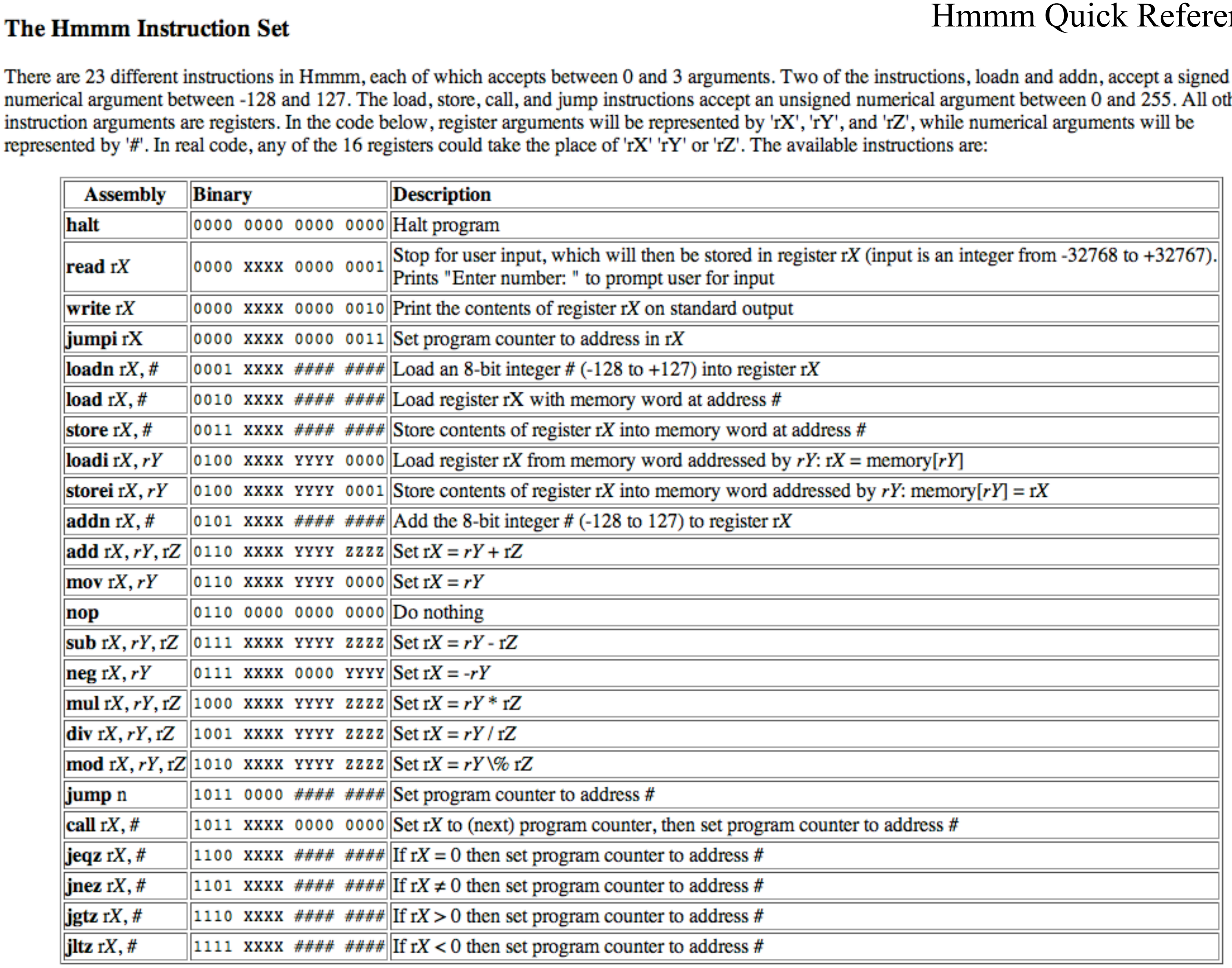


More concrete

Digital circuit design	how to turn functions into hardware
Computer Architecture	collections of circuits that can perform all the needed functions to manipulate data
Machine Language	sequences of 1s and 0s that drive the hardware to perform the necessary functions
Assembly Language	A translation of the 1s and 0s into something human-readable
Loops in AL	how to use AL to do stuff that's useful to humans
Functions and recursion in AL	

More abstract

Hmmm Quick Reference



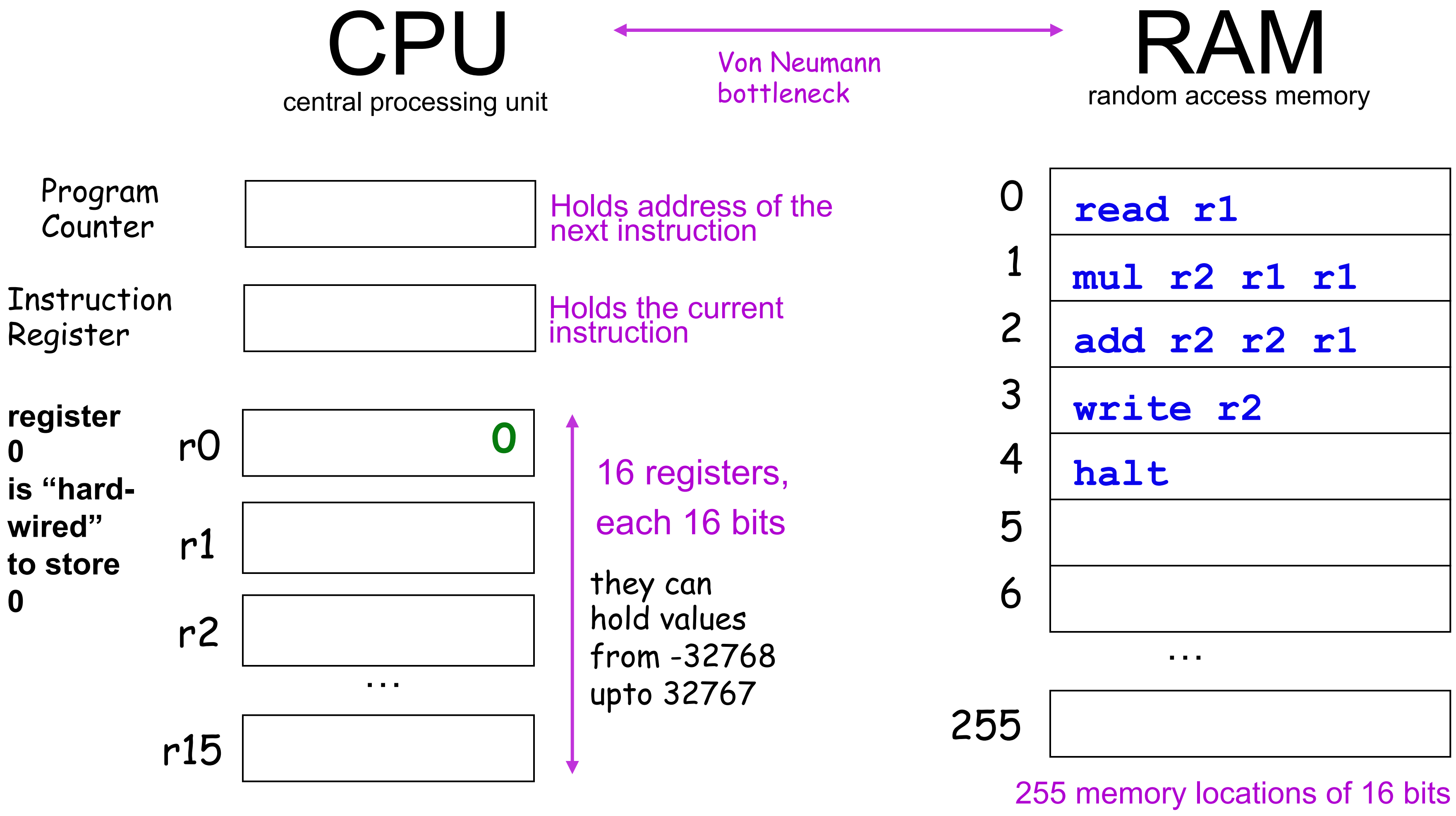
The Hmmm Instruction Set

There are 23 different instructions in Hmmm, 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:

Assembly	Binary	Description
halt	`0000 0000 0000 0000`	Halt program
read rX	`0000 XXXX 0000 0001`	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
write rX	`0000 XXXX 0000 0010`	Print the contents of register rX on standard output
jumpi rX	`0000 XXXX 0000 0011`	Set program counter to address in rX
loadn rX, #	`0001 XXXX #### ####`	Load an 8-bit integer # (-128 to +127) into register rX
load rX, #	`0010 XXXX #### ####`	Load register rX with memory word at address #
store rX, #	`0011 XXXX #### ####`	Store contents of register rX into memory word at address #
loadi rX, rY	`0100 XXXX YYYY 0000`	Load register rX from memory word addressed by rY: rX = memory[rY]
storei rX, rY	`0100 XXXX YYYY 0001`	Store contents of register rX into memory word addressed by rY: memory[rY] = rX
addn rX, #	`0101 XXXX #### ####`	Add the 8-bit integer # (-128 to 127) to register rX
add rX, rY, rZ	`0110 XXXX YYYY ZZZZ`	Set rX = rY + rZ
mov rX, rY	`0110 XXXX YYYY 0000`	Set rX = rY
nop	`0110 0000 0000 0000`	Do nothing
sub rX, rY, rZ	`0111 XXXX YYYY ZZZZ`	Set rX = rY - rZ
neg rX, rY	`0111 XXXX 0000 YYYY`	Set rX = -rY
mul rX, rY, rZ	`1000 XXXX YYYY ZZZZ`	Set rX = rY * rZ
div rX, rY, rZ	`1001 XXXX YYYY ZZZZ`	Set rX = rY / rZ
mod rX, rY, rZ	`1010 XXXX YYYY ZZZZ`	Set rX = rY \% rZ
jump n	`1011 0000 #### ####`	Set program counter to address #
call rX, #	`1011 XXXX 0000 0000`	Set rX to (next) program counter, then set program counter to address #
jeqz rX, #	`1100 XXXX #### ####`	If rX = 0 then set program counter to address #
jnez rX, #	`1101 XXXX #### ####`	If rX ≠ 0 then set program counter to address #
jgtz rX, #	`1110 XXXX #### ####`	If rX > 0 then set program counter to address #
jltz rX, #	`1111 XXXX #### ####`	If rX < 0 then set program counter to address #

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	
…	
r15	

16 registers, each 16 bits

they can hold values from -32768 upto 32767

0	`read r1`
1	`mul r2 r1 r1`
2	`add r2 r2 r1`
3	`write r2`
4	`halt`
5	
6	
…	
255	

255 memory locations of 16 bits

Hmmm

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.

Prograr
Countei

Instructi
Register

register 0 is "hard-wired" to store 0

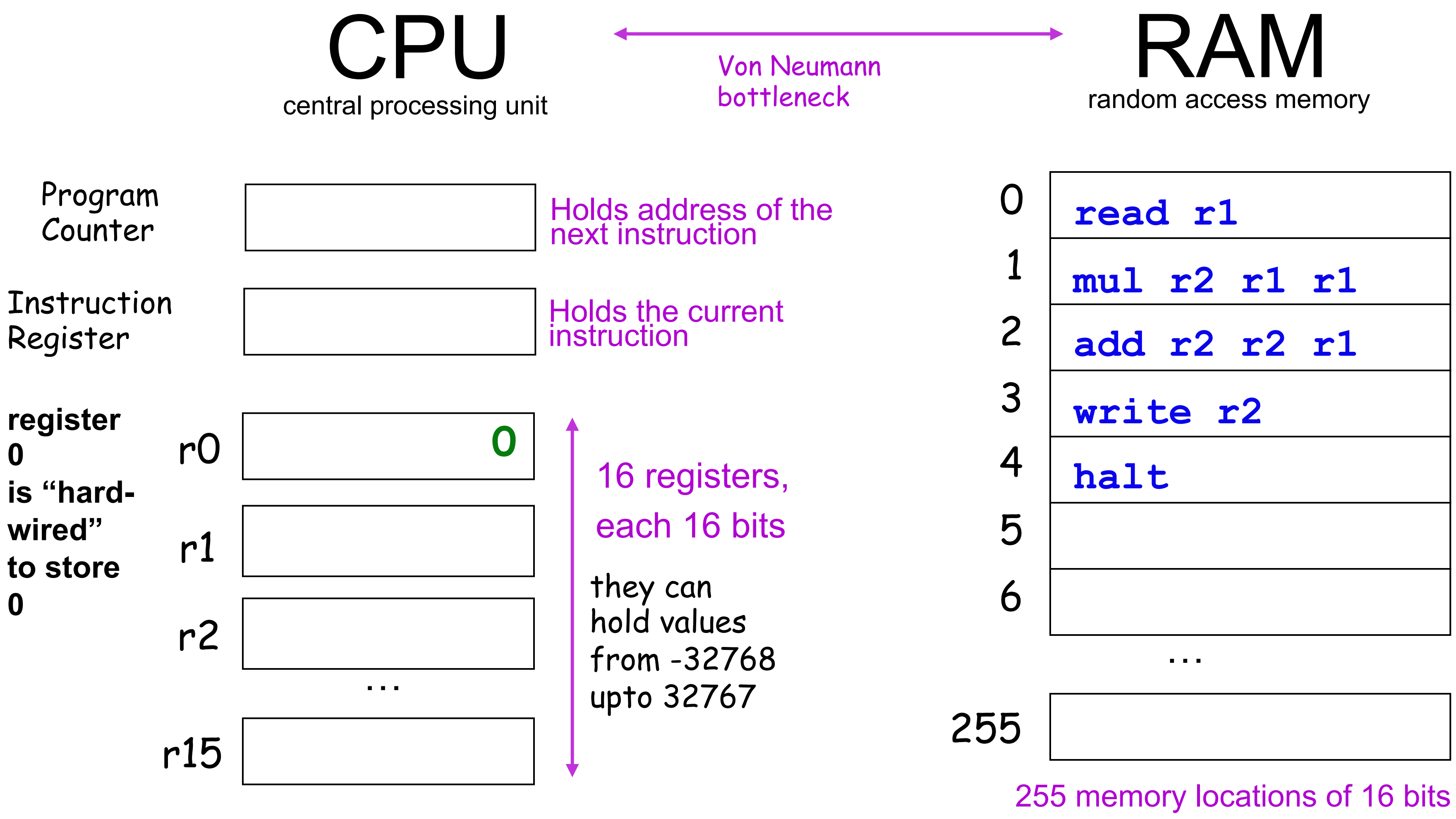
r15

255

255 memory locations of 16 bits

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	
…	
r15	

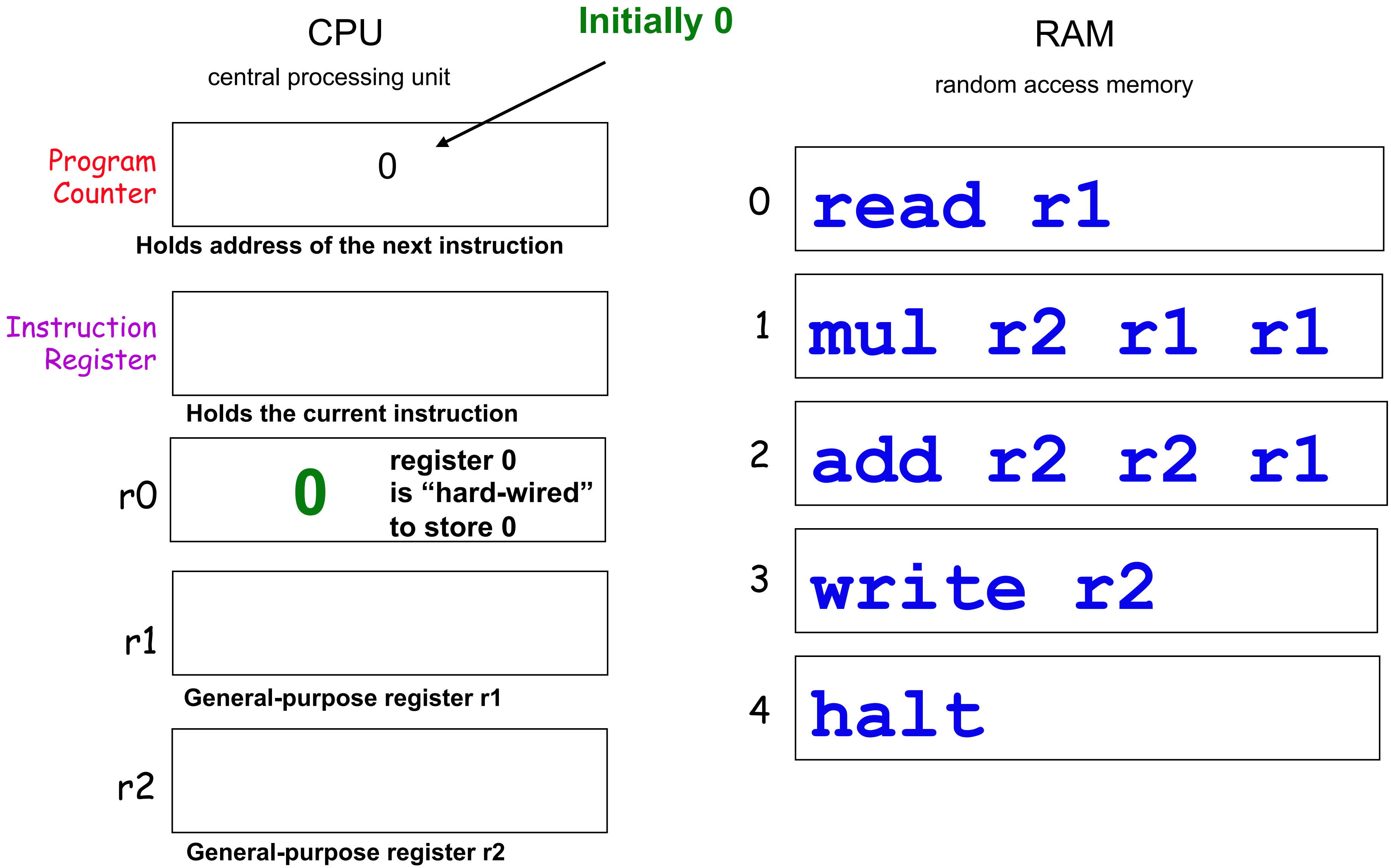
16 registers, each 16 bits

they can hold values from -32768 upto 32767

0	`read r1`
1	`mul r2 r1 r1`
2	`add r2 r2 r1`
3	`write r2`
4	`halt`
5	
6	
…	
255	

255 memory locations of 16 bits

Hmmm: the *fetch - execute* cycle



CPU

central processing unit

Initially 0

Program Counter: 0

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

Assembly Language

register-level programming

`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
write r0
```

read from keyboard
and write to screen

Each of these instructions (and many more) get implemented for a particular processor and particular machine… .

Is this enough?

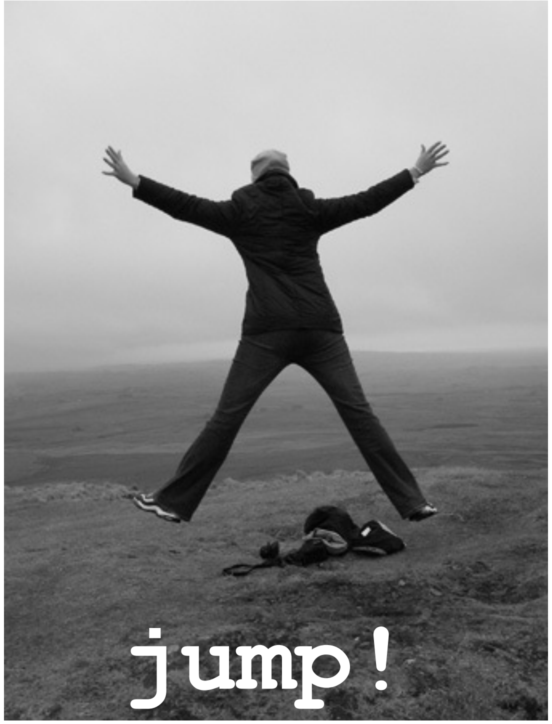
Could we implement Racket using Hmmm Assembly?

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

fetch-execute cycle

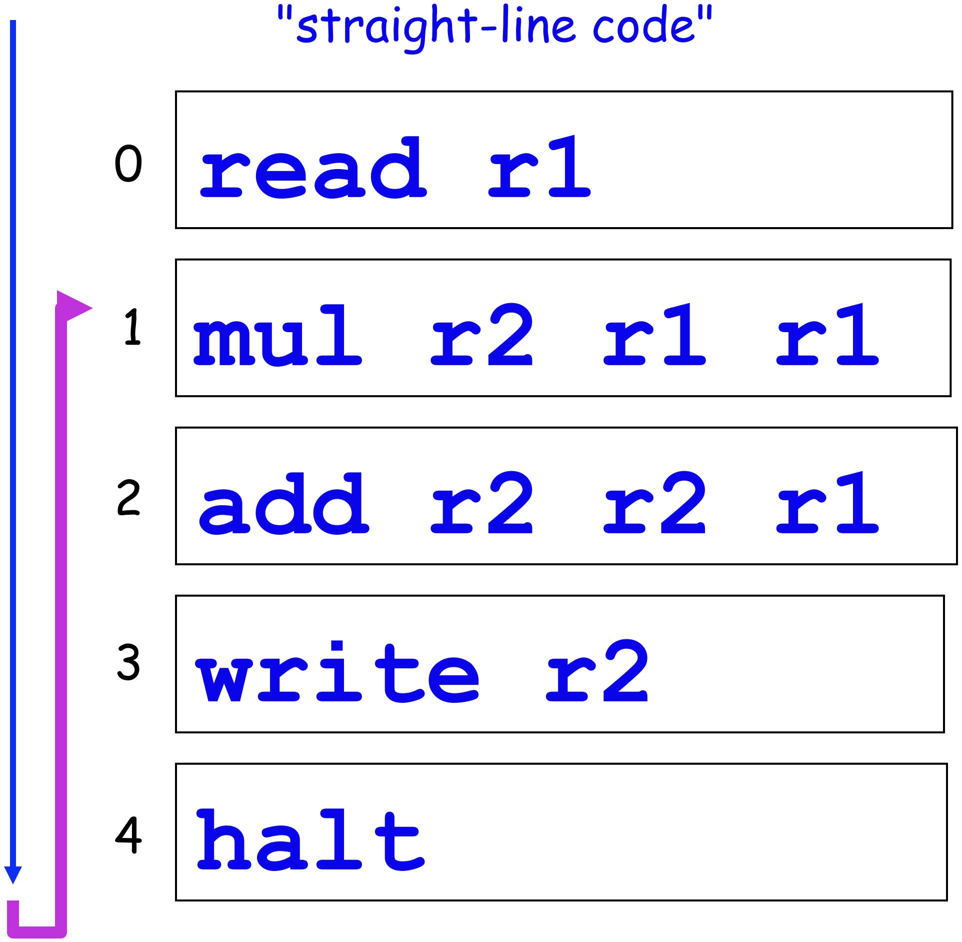
It's not enough!

Could we implement 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
```

Hmmm, Let's `jump` !

CPU

central processing unit

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	`loadn r1 42`
1	`write r1`
2	`addn r1 1`
3	`jump 1`
4	`halt`

Replace 1 with 2? with r1?

`jump`s

Unconditional **jump** GOTO

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

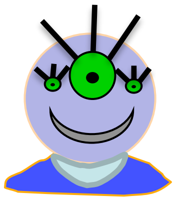
Conditional **jumps** IF

`jeqz r1 #`	IF `r1 == 0` THEN jump to line number `#`
`jgtz r1 #`	IF `r1 > 0` THEN jump to the location in `#`
`jltz r1 #`	IF `r1 < 0` THEN jump to the location in `#`
`jnez r1 #`	IF `r1 != 0` THEN jump to the location in `#`

Indirect **jump**

`jumpi r1`	Jump to the line `#` stored in `reg1`!

jgtz



CPU

central processing unit

Program Counter

Holds address of the next instruction

Instruction Register

Holds the current instruction

r1

General-purpose register r1

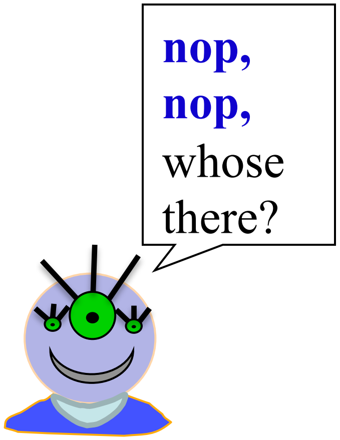
r2

General-purpose register r2

RAM

random access memory

0	`read r1`
1	`jgtz r1 7`
2	`loadn r2 -1`
3	`mul r1 r1 r2`
4	`nop`
5	`nop`
6	`nop`
7	`write r1`
8	`halt`



What is this code computing about its input?

Worksheet

1 Follow this assembly-language program from top to bottom. Use `r1 = 42` and `r2 = 5`.

Registers - CPU

r0	0
r1	
r2	
r3	
r4	

Memory - RAM

0	`read r1`
1	`read r2`
2	`sub r3 r2 r1`
3	`nop`
4	`jltz r3 7`
5	`write r1`
6	`jump 8`
7	`write r2`
8	`halt`
9	

(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?

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!

1 Follow this assembly-language program from top to bottom. Use `r1 = 42` and `r2 = 5`.

Registers - CPU

r0 0

r1

r2

r3

Memory - RAM

0	`read r1`
1	`read r2`
2	`sub r3 r2 r1`
3	`nop`
4	`jltz r3 7`
5	`write r1`
6	`jump 8`
7	`write r2`
8	`halt`
9	

(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?

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.

factorial

plan

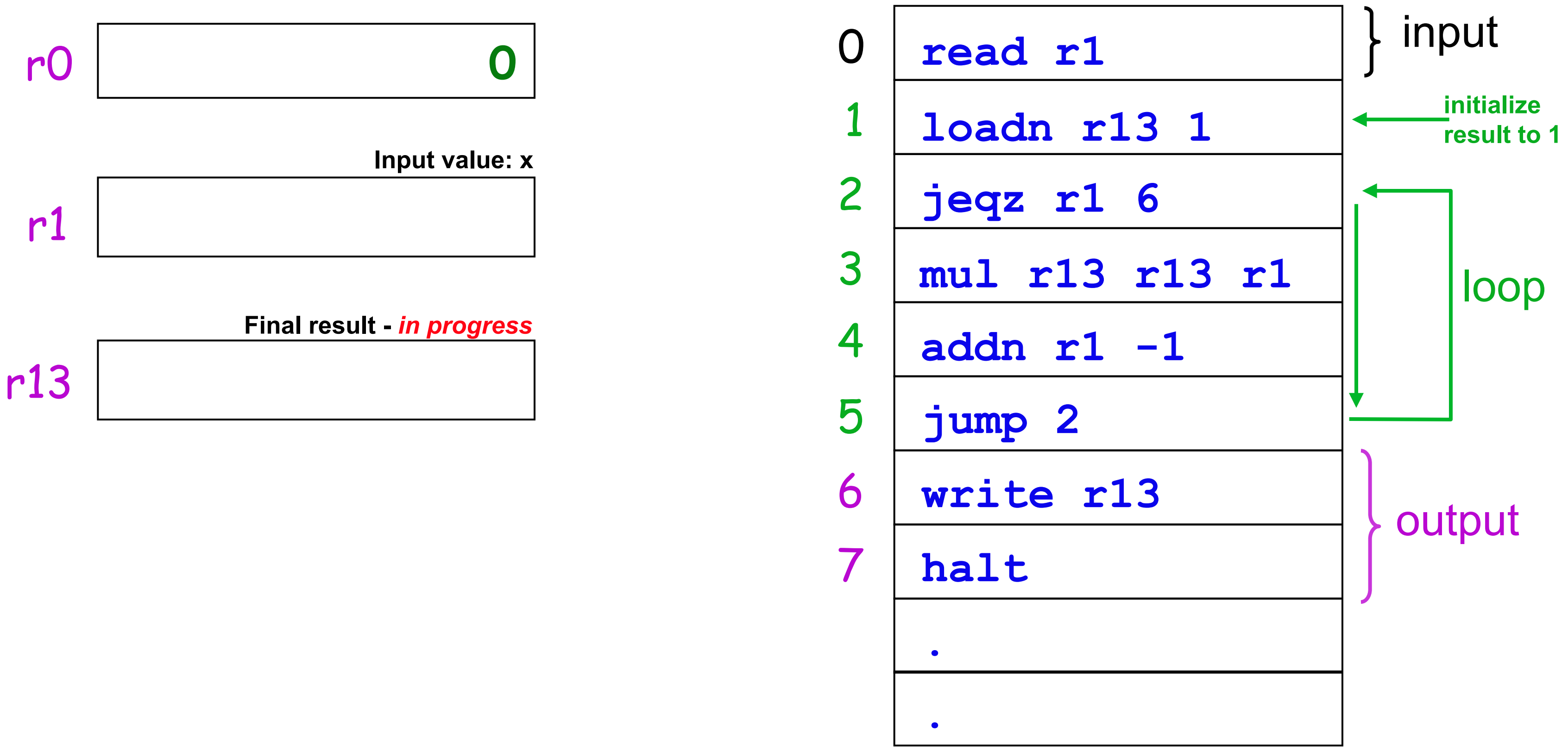
Program Counter

Instruction Register

Factorial

Hmmm CPU

Hmmm RAM



r0: 0

r1 — Input value: x

r13 — Final result - *in progress*

```
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
   .
   .
```

What is 42 ?

Base 10

TENs column
ONEs column

42

4 tens + 2 ones

HUNDREDs column
TENs column
ONEs column

123

1 hundred + 2 tens + 3 ones

Base 2

THIRTYTWOs col.
SIXTEENs col.
EIGHTs column
FOURs column
TWOs column
ONEs column

101010

128s column
SIXTYFOURs col
THIRTYTWOs col.
SIXTEENs col.
EIGHTs column
FOURs column
TWOs column
ONEs column

_ _ _ _ _ _ _ _

Write 123 in binary…

each column
represents
another power
of the base

Base 10

TENs column
ONEs column

42

4 tens + 2 ones

HUNDREDs column
TENs column
ONEs column

123

1 hundred + 2 tens + 3 ones

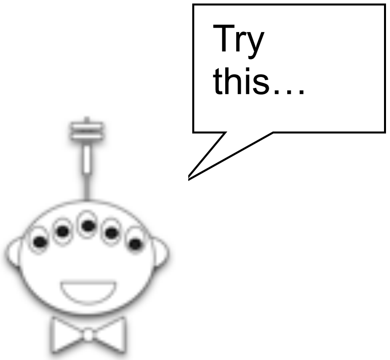
Counting in base *b*

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!

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

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: `11111111111111111111111111111111111111111111111111…`

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

Base 2: `111011100110101100101000000000`

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

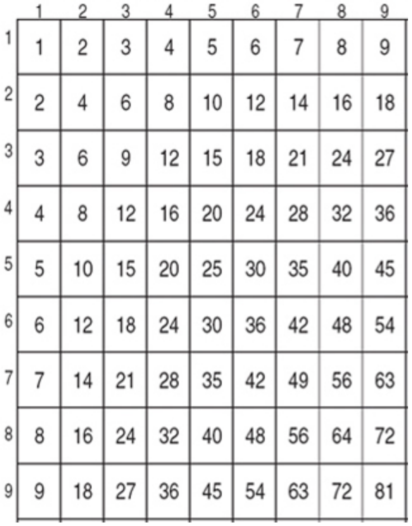
Addition

+

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Multiplication

*****



	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Addition

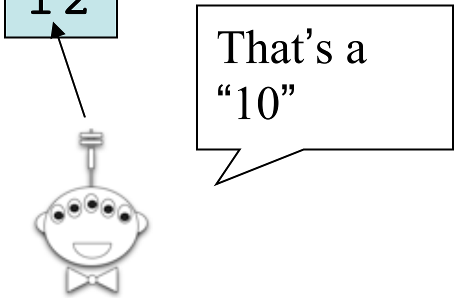
Base 10 Addition

$$\begin{array}{ccc}
10^2 & 10^1 & 10^0 \\
\hline
 & 4 & 3 \\
+ & 8 & 9 \\
\hline
\end{array}$$

Addition

Base 10 Addition

10^2	10^1	10^0
	4	3
+	8	9
		12



Addition

Base 10 Addition

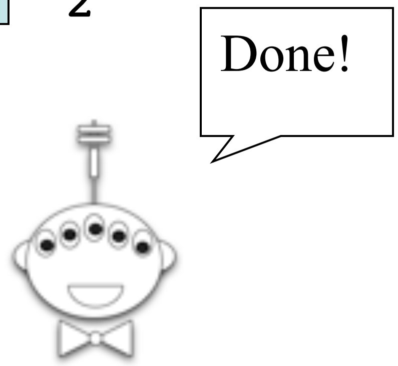
10^2	10^1	10^0
	1	
	4	3
+	8	9
		2

Move the “1” to the ten’s place

Addition

Base 10 Addition

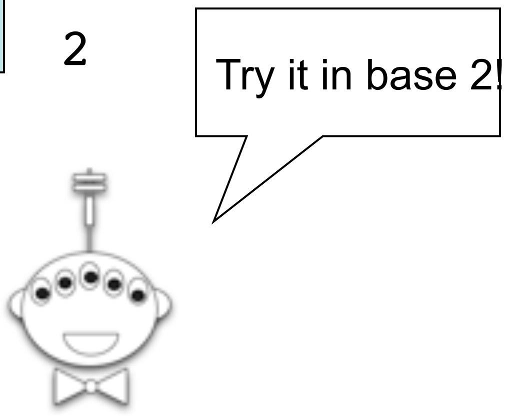
	10^2	10^1	10^0
		1	
		4	3
	+	8	9
		13	2



Addition

Base 10 Addition

	10^2	10^1	10^0
		1	
		4	3
+		8	9
		13	2



Base 2 Addition

	2^2	2^1	2^0
	1	0	1
+	0	1	1

Multiplication

Base 10 Multiplication

$$\begin{array}{cccc} & 10^2 & 10^1 & 10^0 \\ \hline & 3 & 4 & 1 \\ \times & 1 & 0 & 2 \\ \hline \end{array}$$

Multiplication

Base 10 Multiplication

$$\begin{array}{cccccc}
 & & & 10^2 & 10^1 & 10^0 \\ \hline
 & & & 3 & 4 & 1 \\
 & & \times & 1 & 0 & 2 \\ \hline
 & & & 6 & 8 & 2 \\
 & & 0 & 0 & 0 & \\
+ & 3 & 4 & 1 & & \\ \hline
\end{array}$$

Multiplication

Base 10 Multiplication

$$\begin{array}{cccccc}
 & & & 10^2 & 10^1 & 10^0 \\
\cline{3-6}
 & & & 3 & 4 & 1 \\
 & & \times & 1 & 0 & 2 \\
\cline{3-6}
 & & & 6 & 8 & 2 \\
 & & 0 & 0 & 0 & \\
+ & 3 & 4 & 1 & & \\
\cline{1-6}
 & 3 & 4 & 7 & 8 & 2
\end{array}$$

Multiplication

Base 10 Multiplication

$$\begin{array}{rrrrrr}
 & & & 10^2 & 10^1 & 10^0 \\
\hline
 & & & 3 & 4 & 1 \\
 & & \times & 1 & 0 & 2 \\
\hline
 & & & 6 & 8 & 2 \\
 & & 0 & 0 & 0 & \\
+ & 3 & 4 & 1 & & \\
\hline
 & 3 & 4 & 7 & 8 & 2
\end{array}$$

Base 2 Multiplication

$$\begin{array}{rrrr}
 & 2^2 & 2^1 & 2^0 \\
\hline
 & 1 & 1 & 1 \\
\times & 1 & 0 & 1 \\
\hline
\end{array}$$

"Quiz"

There are 10 kinds of "people" in the universe: those who use binary, and those who do not!

Name:

Convert these two binary numbers to decimal:

32 16 8 4 2 1

110011

10001000

Convert these two decimal numbers to binary:

28

101

WITHOUT converting to decimal !

Add these two binary numbers:

```
  101101
+   1110
```

Multiply these binary numbers:

```
  101101
*   1110
```

```
   1
  529
+ 742
 1271
```

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