Software

Python

How does Python **function** ?

Hmmm ☆

4 Hmmm problems
+ 1 loop problem

Assembly Language
Machine Language

CS 5 this week

RAM

registers

1-bit memory: flip-flops

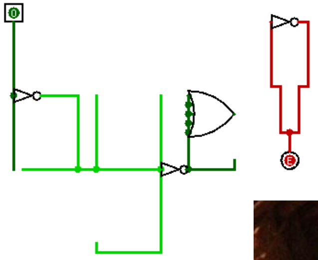
arithmetic

bitwise functions

logic gates

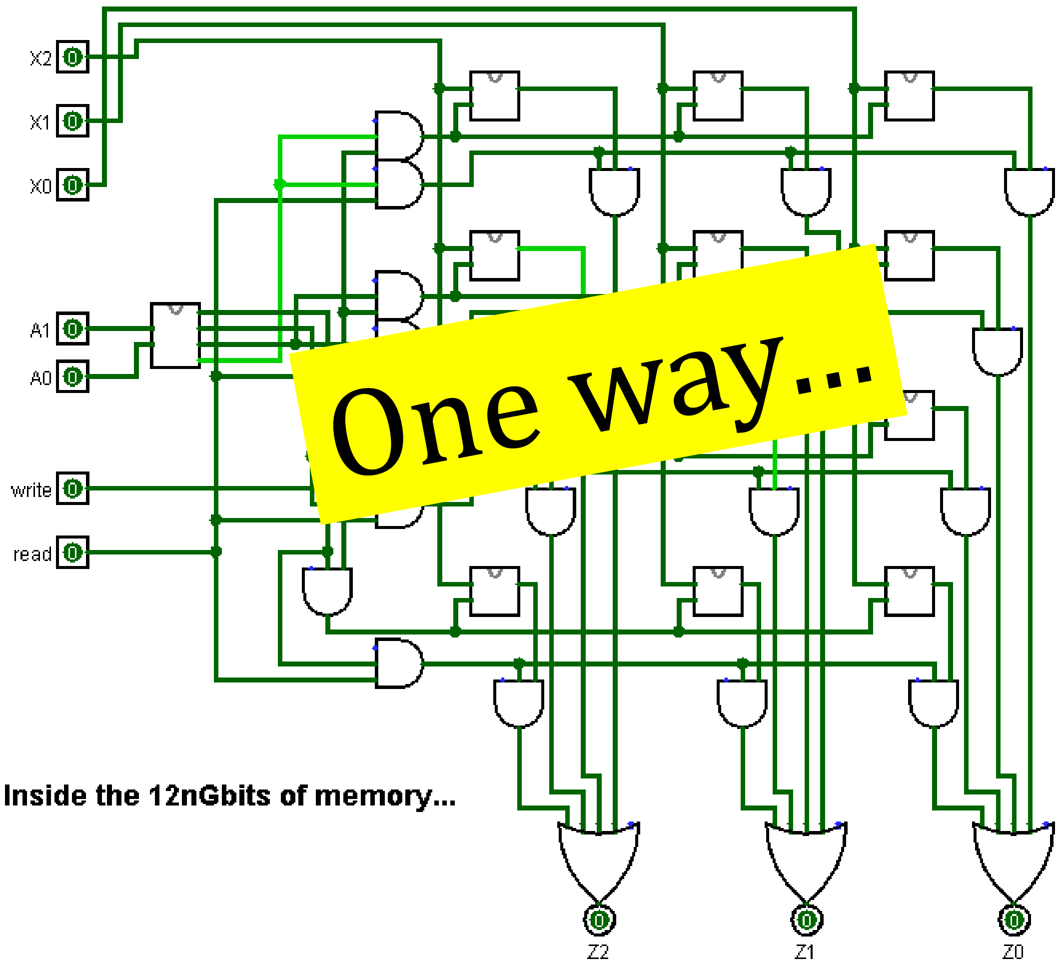
transistors / switches

Hardware



I have a looming sense…

Making memories...



One way...

Inside the 12nGbits of memory...

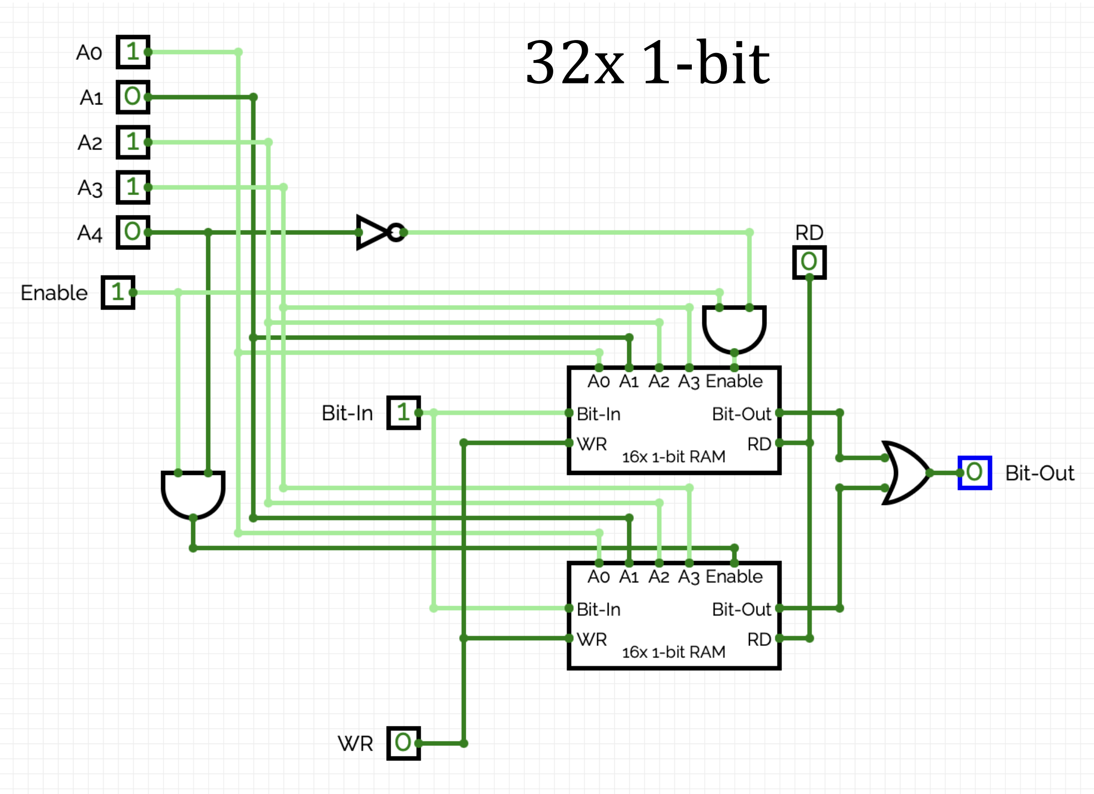
My head is spinning...

Circuits ~ Memory!

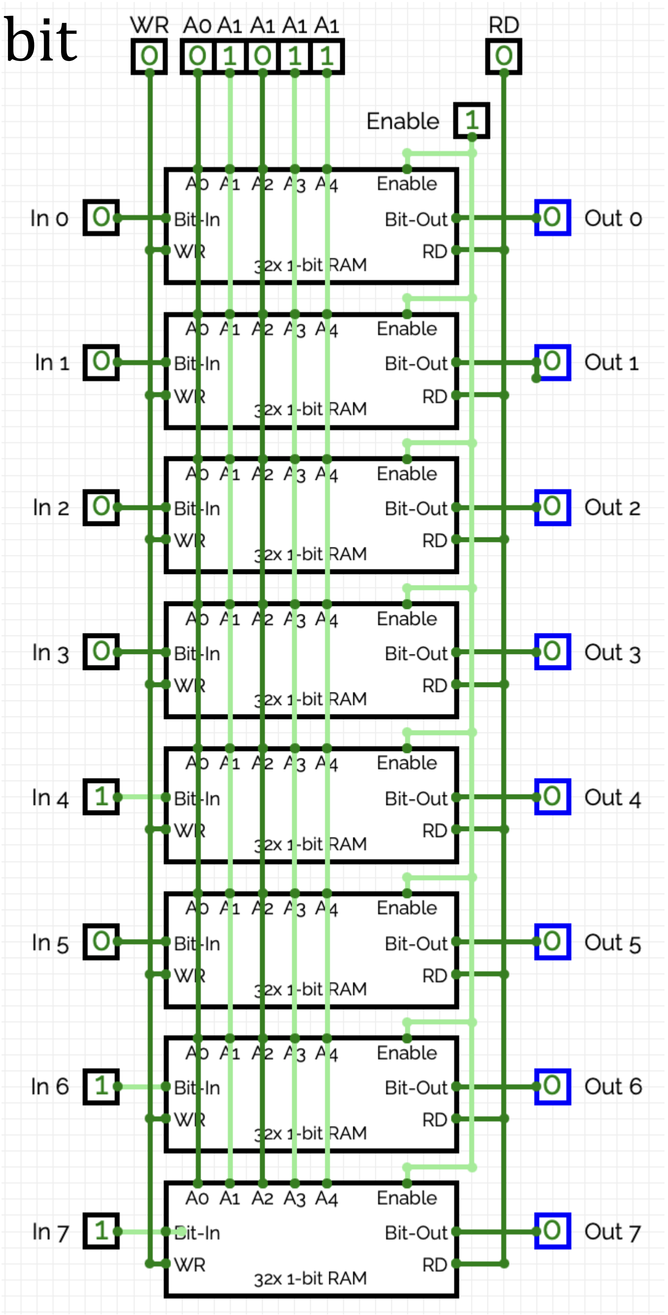
32 bytes of memory

the power of composition

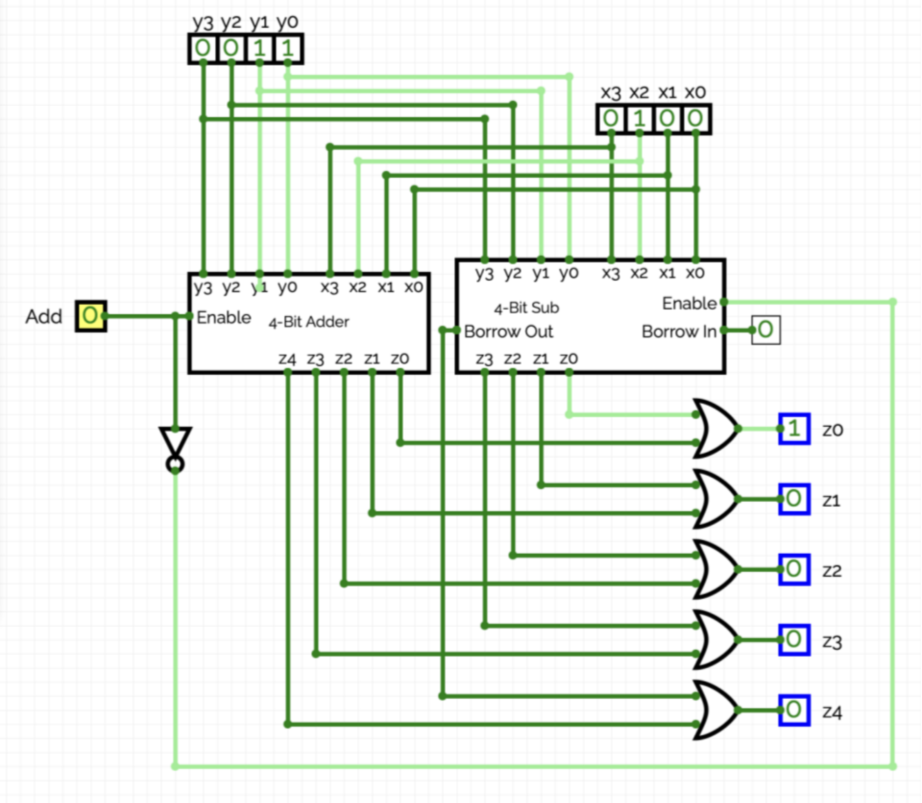
32x 1-bit

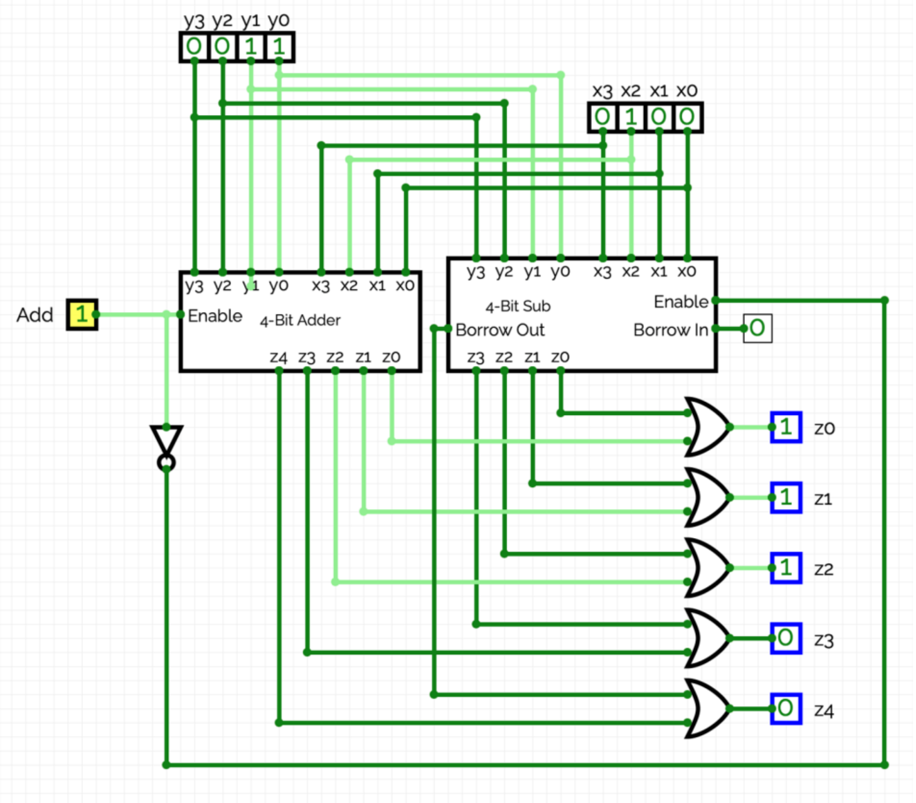


32x 8-bit

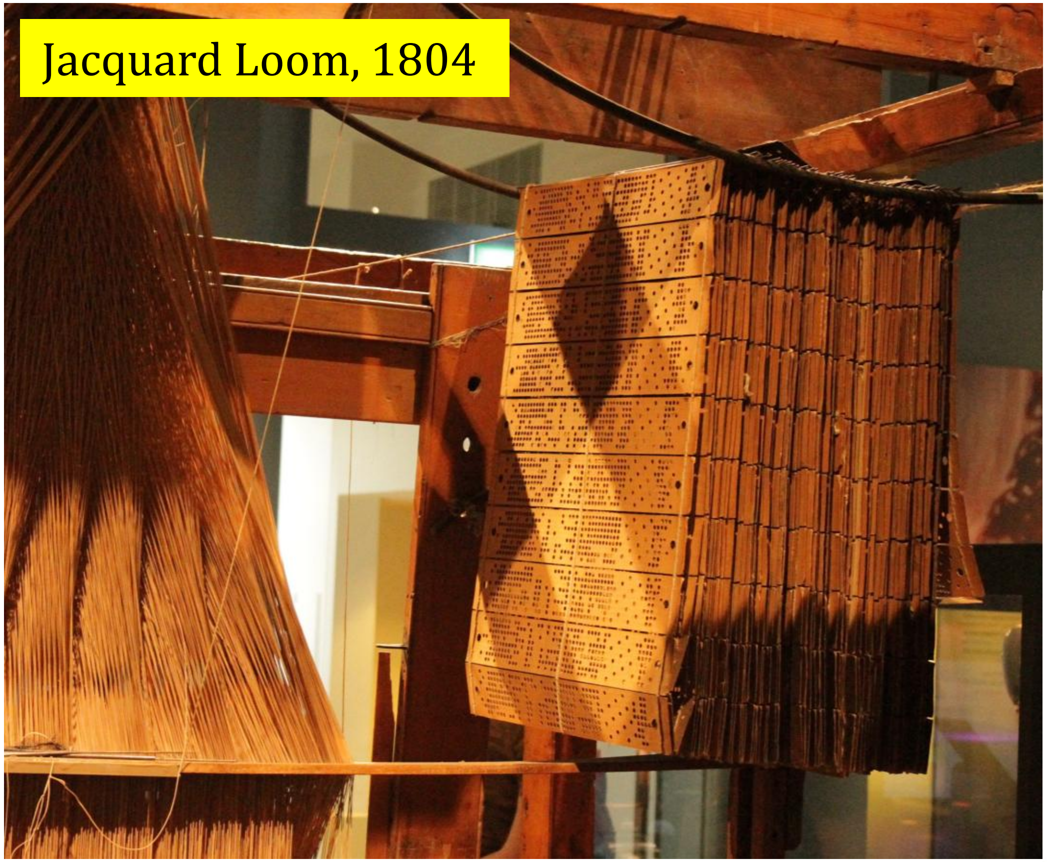


Fun with control?

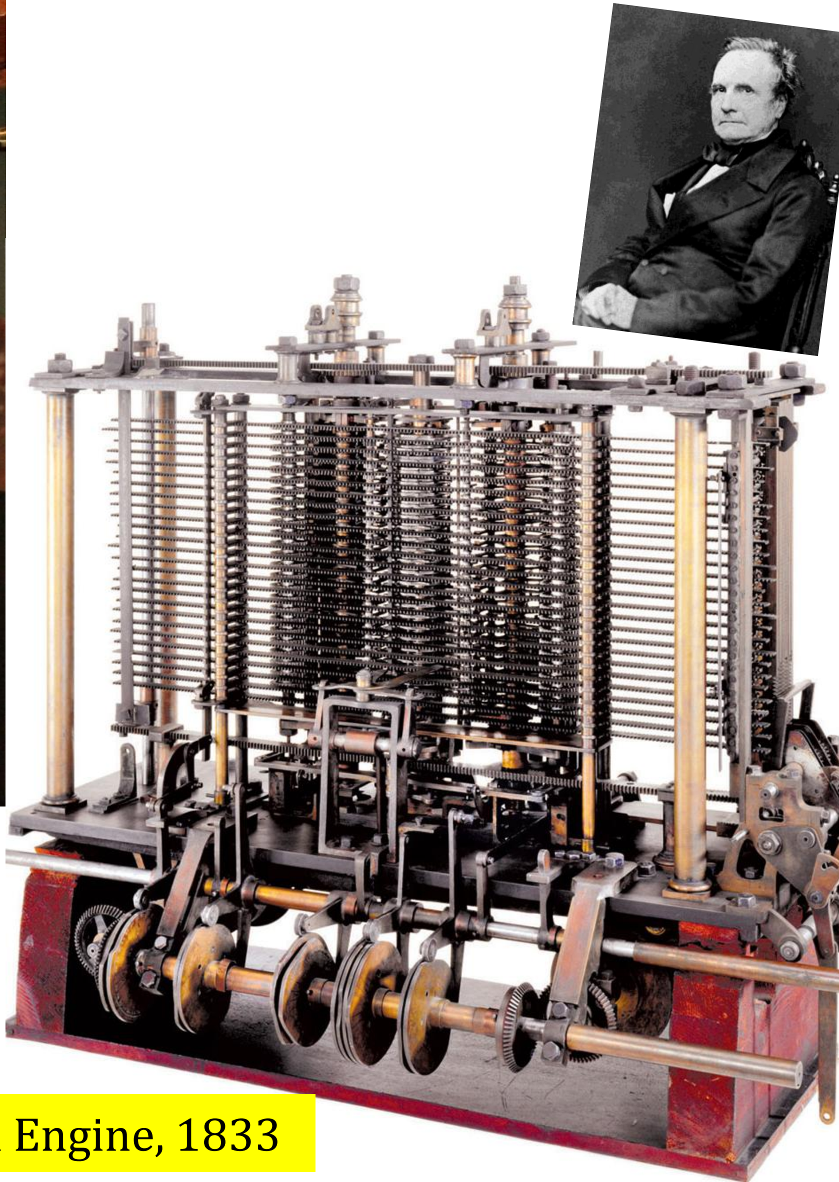




Early Binary Control…



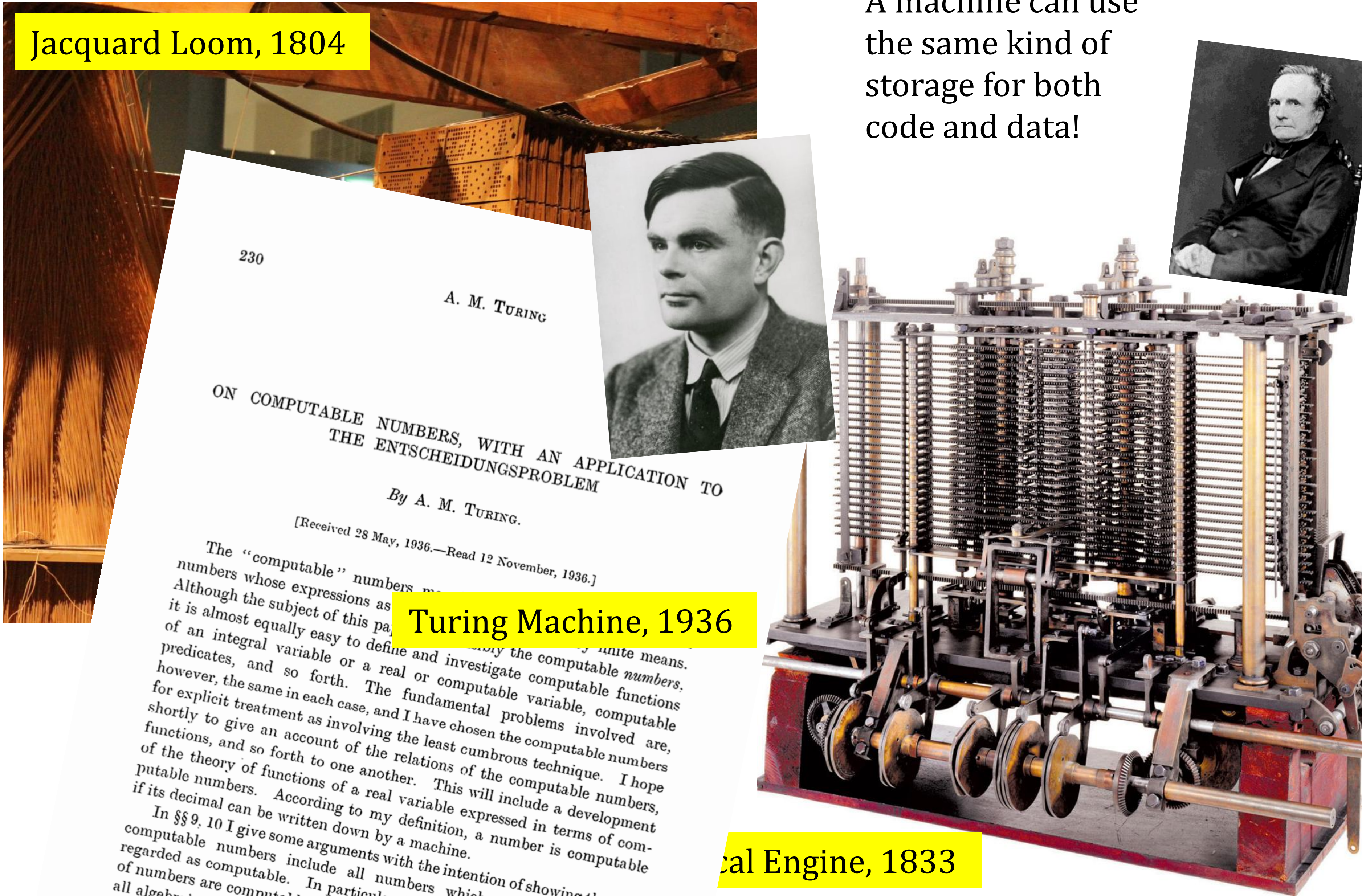
Jacquard Loom, 1804



Babbage's Analytical Engine, 1833

Big idea: Control = Data

A machine can use
the same kind of
storage for both
code and data!



Jacquard Loom, 1804

Turing Machine, 1936

cal Engine, 1833

230

A. M. Turing

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO
THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers
numbers whose expressions as
Although the subject of this pa
it is almost equally easy to define and investigate computable functions
of an integral variable or a real or computable variable, computable
predicates, and so forth. The fundamental problems involved are,
however, the same in each case, and I have chosen the computable numbers
for explicit treatment as involving the least cumbrous technique. I hope
shortly to give an account of the relations of the computable numbers,
functions, and so forth to one another. This will include a development
of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable
if its decimal can be written down by a machine.

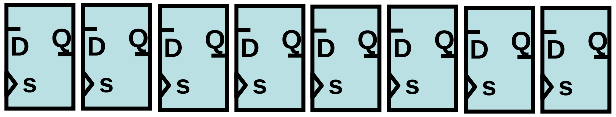
In §§ 9, 10 I give some arguments with the intention of showing
computable numbers include all numbers which
regarded as computable. In particul
of numbers are comput
all algebr

Some memory is more equal than others...

Registers

on the Central Processing Unit

8 flip-flops are an 8-bit **register**



100 Registers of 64 bits each

~ 10,000 bits

memory from *logic gates*

Main Memory

(replaceable RAM)

10 GB memory

~ 100 billion bits

"Leaky Bucket" capacitors

Disk Drive

magnetic storage

4 TB drive

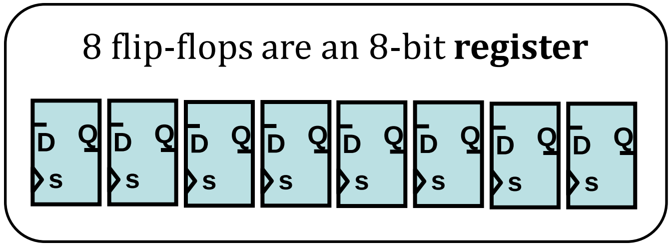
~ 42 trillion bits (or more)

remagnetizing surfaces

Some memory is more equal than others...

Registers

on the Central Processing Unit



8 flip-flops are an 8-bit **register**

100 Registers of 64 bits each

~ 10,000 bits

Main Memory

(replaceable RAM)

10 GB memory

~ 100 billion bits

Disk Drive

magnetic storage

4 TB drive

~ 42 trillion bits (or more)

At least at my store!

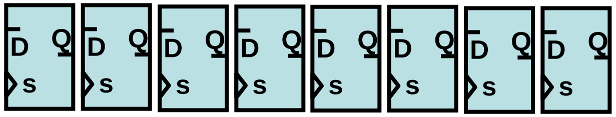
Price	~$50	~$50	~$50
Time	1 clock cycle 10^{-9} sec	100 cycles 10^{-7} sec	10^{7} cycles 10^{-2} sec
If a clock cycle == 1 minute	**1 min**	**1.5 hours**	**19 YEARS**

Some memory is more equal than others…

Registers

on the Central Processing Unit

8 flip-flops are an 8-bit **register**



100 Registers of 64 bits each

~ 10,000 bits

Main Memory

(replaceable RAM)

10 GB memory

~ 100 billion bits

Disk Drive

magnetic storage

4 TB drive

~ 42 trillion bits (or more)

programs are fetched and executed 1 instruction at a time here…

***running* programs are stored here…**

"Off" data is saved way out here…

P

Ti

sec

10^{-2} sec

If a c ycle
== 1 minute

1 min

1.5 hours

19 YEARS

How do we execute ***sequences*** of operations?

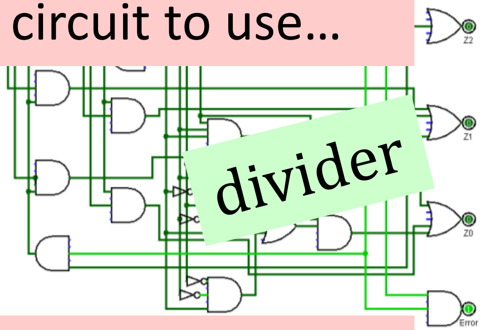
processor **CPU**

stores all instructions and almost all data

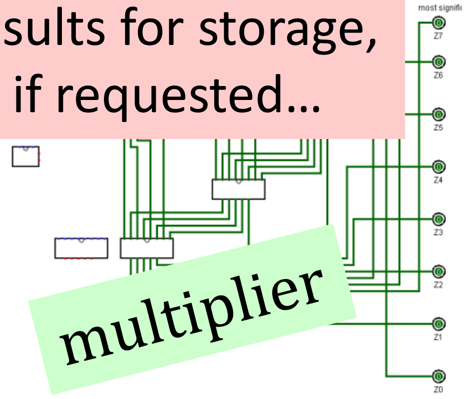
RAM live memory

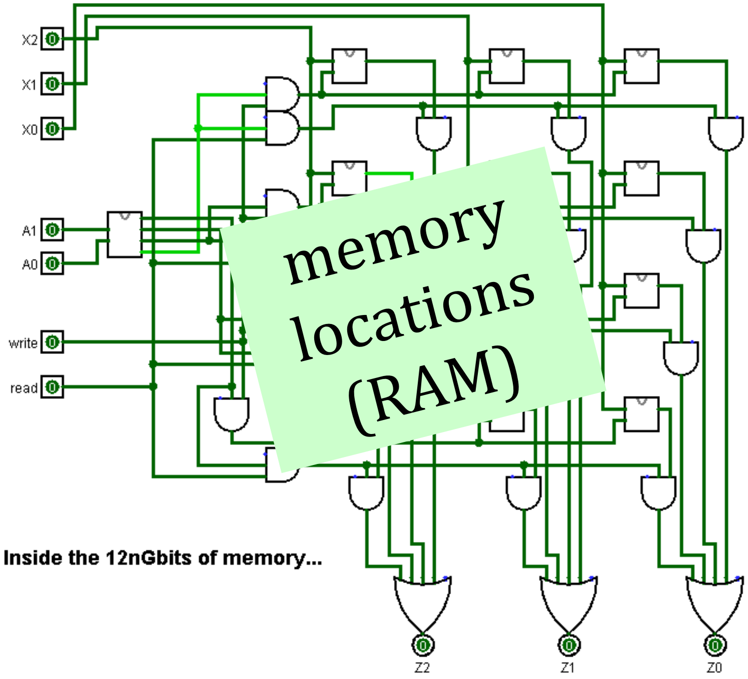
the instruction's bits select which circuit to use...

sends next instruction to the CPU ...



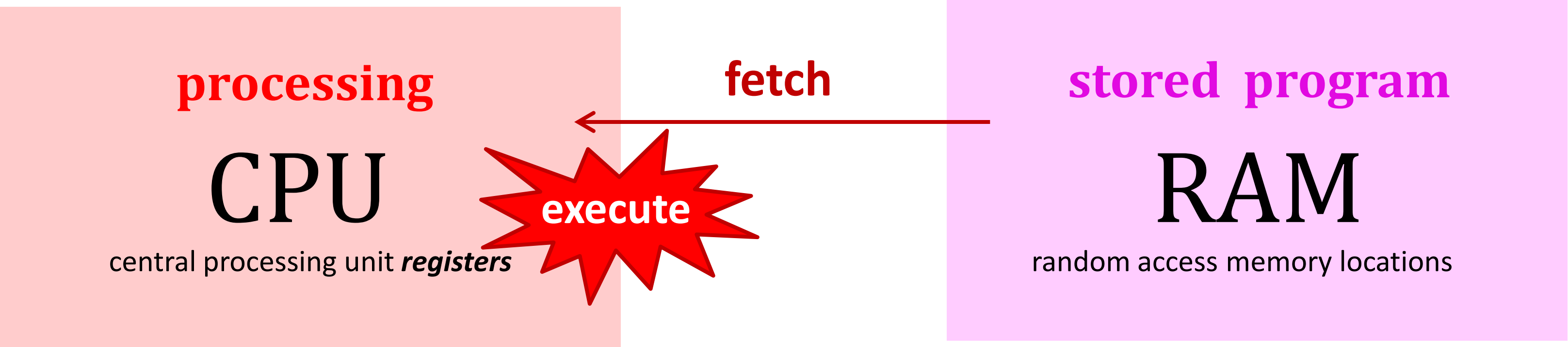
runs 1 instruction and sends back results for storage, if requested...





sends next instruction to the CPU ...

75 years ago…



processing

CPU

central processing unit ***registers***

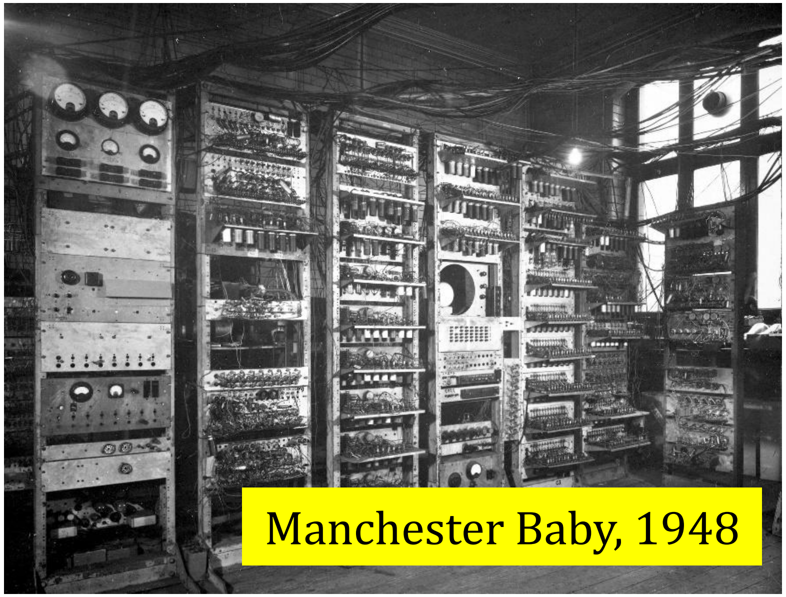
fetch

execute

stored program

RAM

random access memory locations

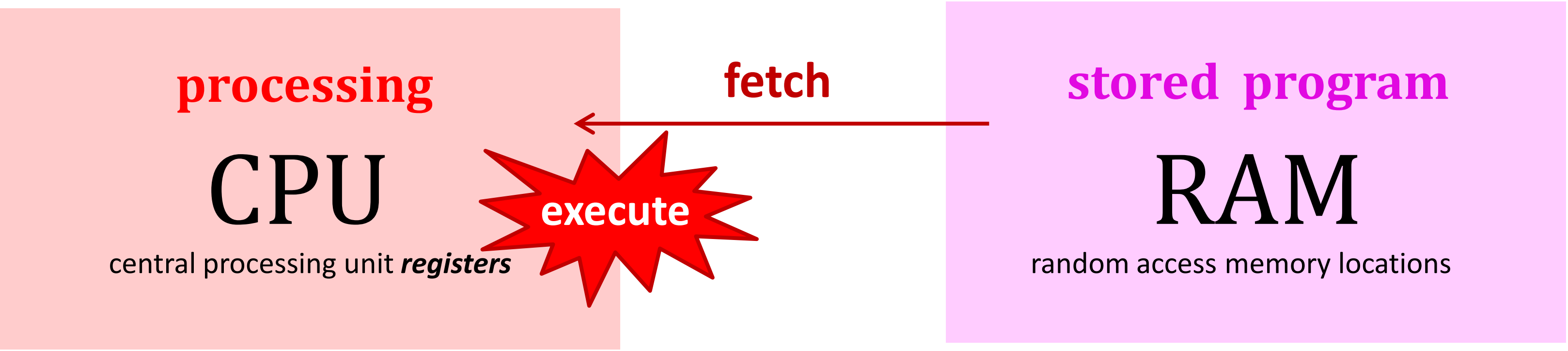


Manchester Baby, 1948

limited, fast **registers**
+ arithmetic

larger, slower **memory**
+ *no* computation

75 years later...



processing

CPU

central processing unit ***registers***

execute

fetch

stored program

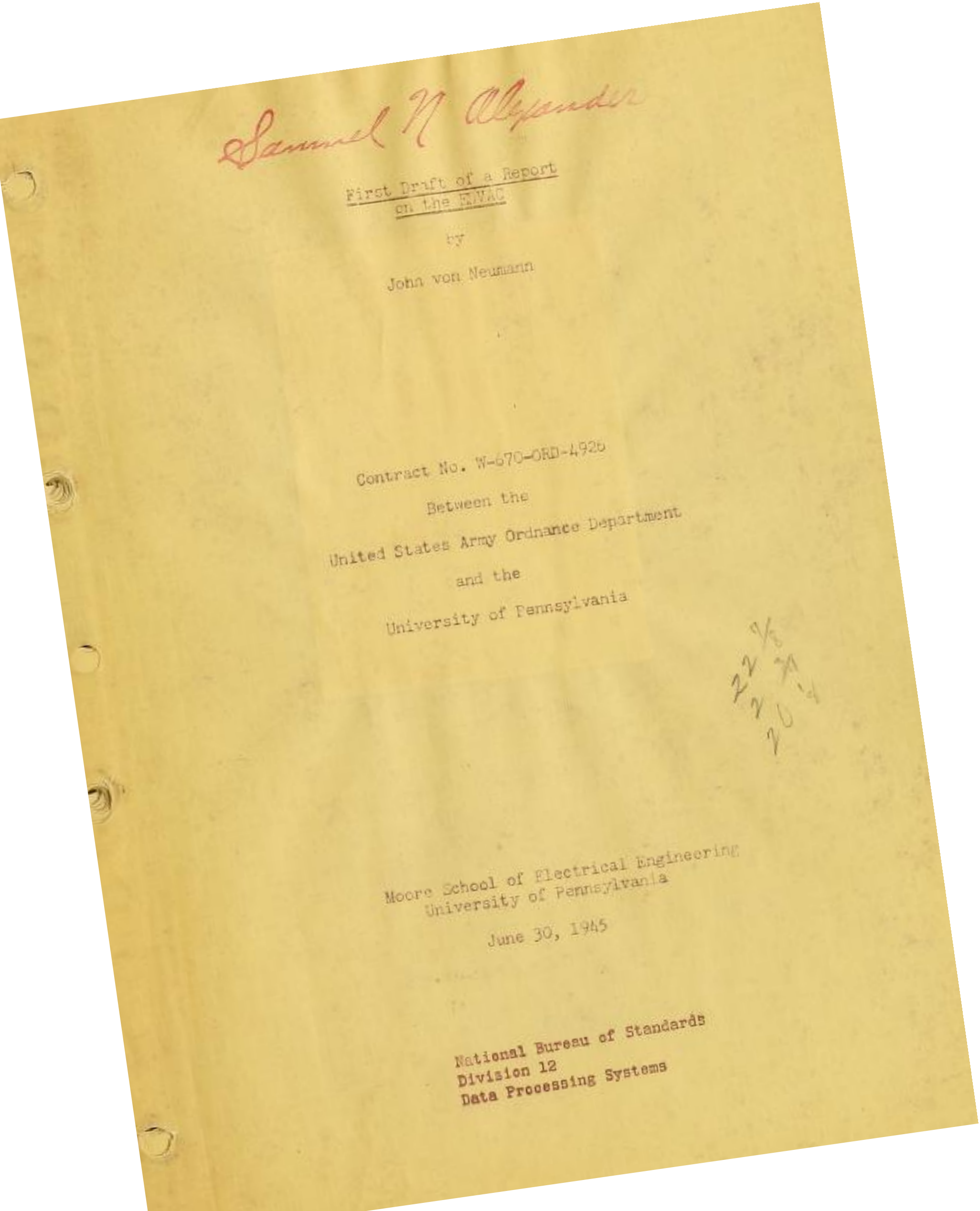
RAM

random access memory locations

limited, fast **registers**
+ arithmetic

larger, slower **memory**
+ *no* computation

John von Neumann



Samuel N Alexander

First Draft of a Report
on the EDVAC

by

John von Neumann

Contract No. W-670-ORD-4926

Between the

United States Army Ordnance Department

and the

University of Pennsylvania

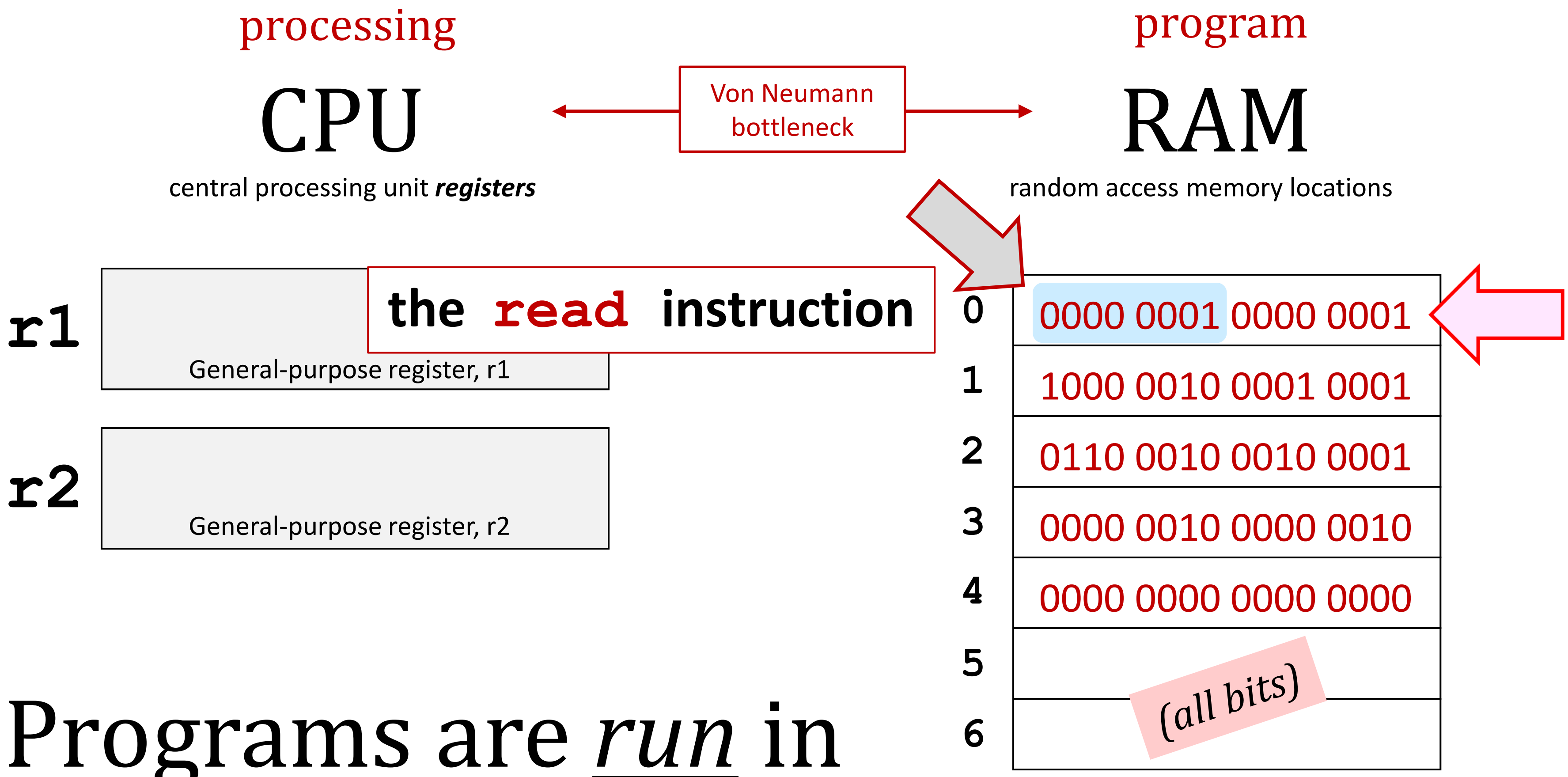
Moore School of Electrical Engineering
University of Pennsylvania

June 30, 1945

National Bureau of Standards
Division 12
Data Processing Systems

- Polymath
- On EDVAC team…
  - Wasn't first stored-program computer!
- Based on the work of J. Presper Eckert and John Mauchly and other EDIAC/EDVAC designers.
  - Prevented their patent.

“Von Neumann” Architecture



processing

CPU

central processing unit ***registers***

Von Neumann bottleneck

program

RAM

random access memory locations

`r1` General-purpose register, r1

`r2` General-purpose register, r2

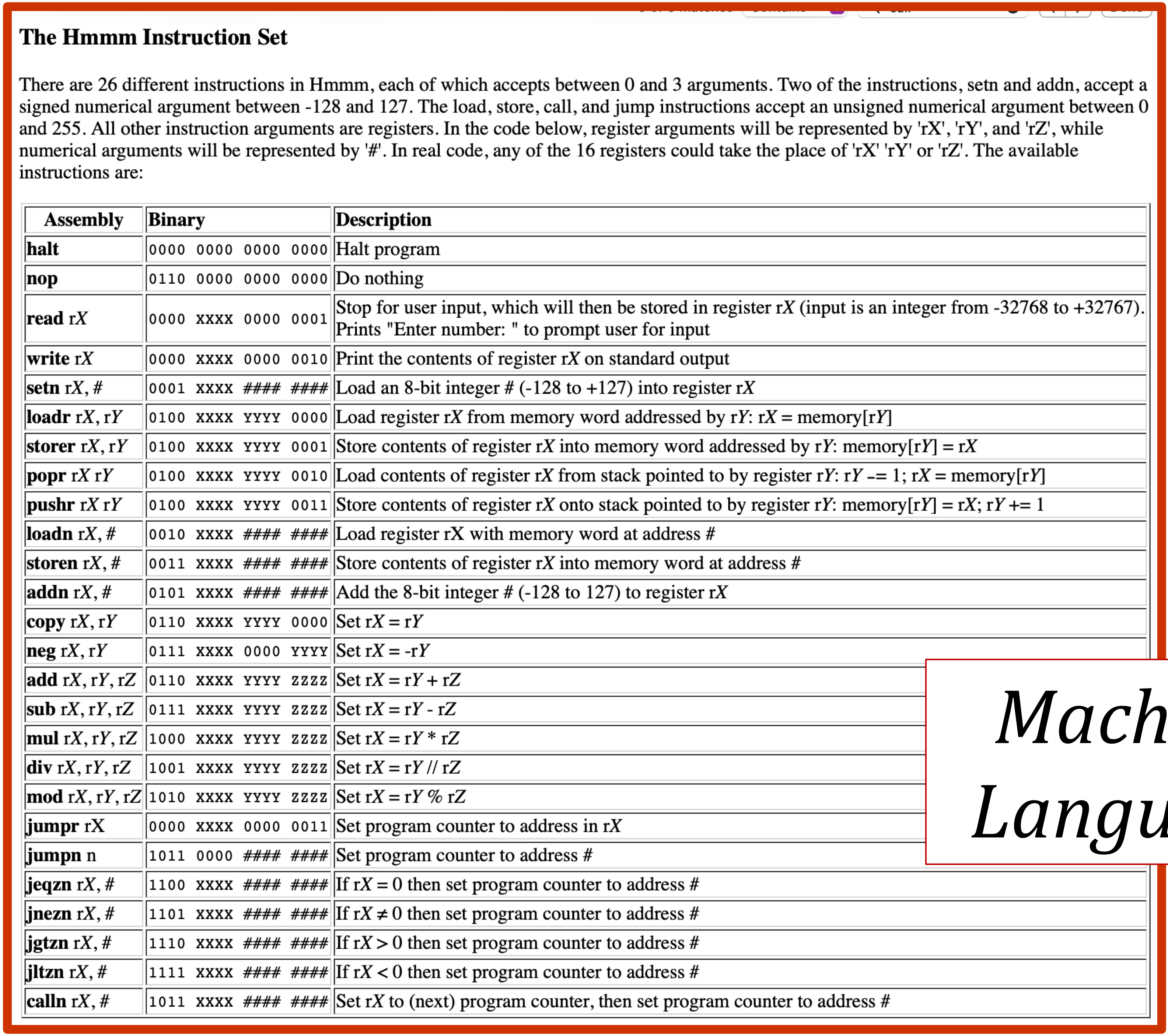
the `read` instruction

0	0000 0001 0000 0001
1	1000 0010 0001 0001
2	0110 0010 0010 0001
3	0000 0010 0000 0010
4	0000 0000 0000 0000
5	
6	

(all bits)

Programs are *run* in

machine language

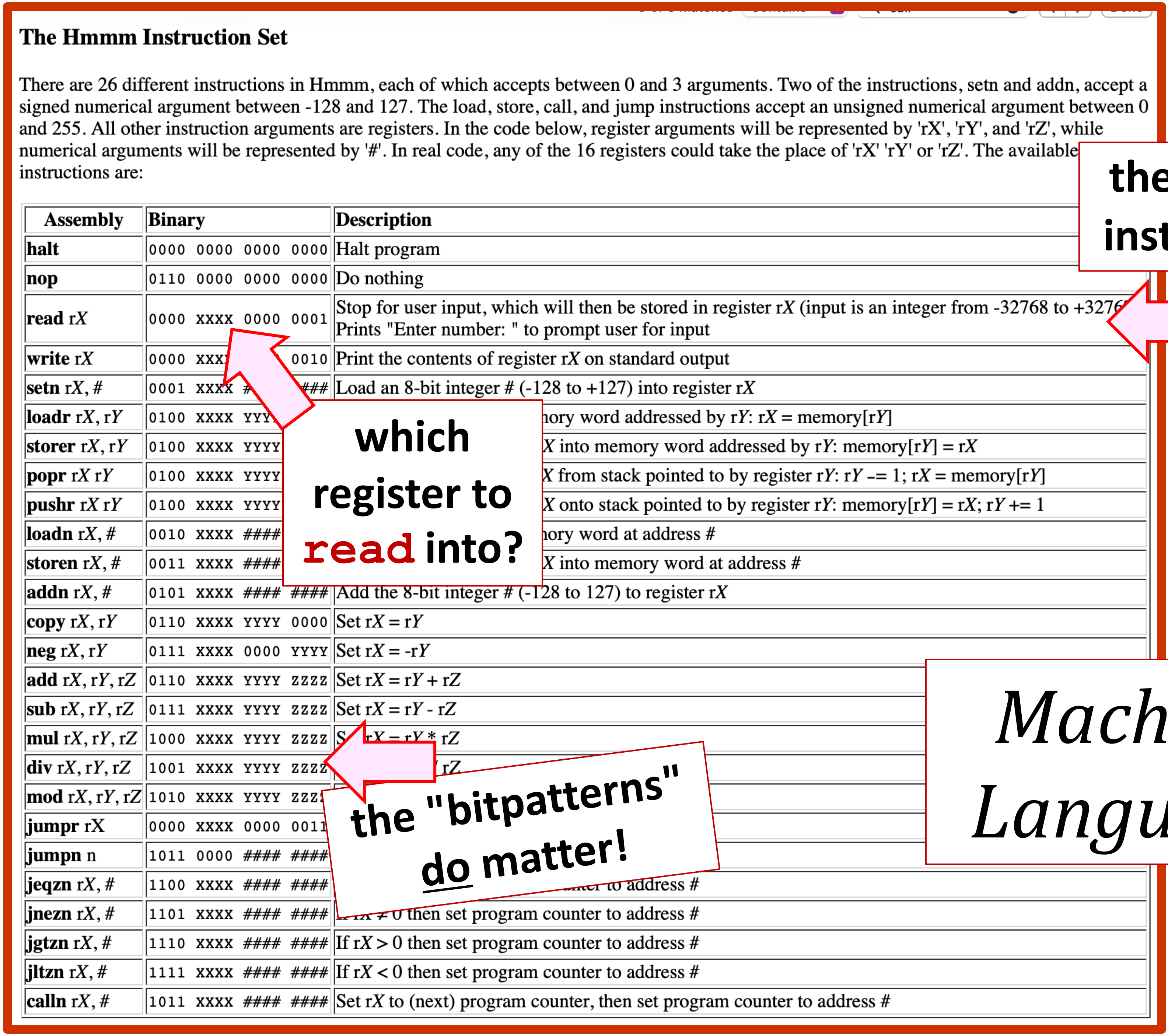


The Hmmm Instruction Set

There are 26 different instructions in Hmmm, each of which accepts between 0 and 3 arguments. Two of the instructions, setn 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
nop	`0110 0000 0000 0000`	Do nothing
read r*X*	`0000 XXXX 0000 0001`	Stop for user input, which will then be stored in register r*X* (input is an integer from -32768 to +32767). Prints "Enter number: " to prompt user for input
write r*X*	`0000 XXXX 0000 0010`	Print the contents of register r*X* on standard output
setn r*X*, #	`0001 XXXX #### ####`	Load an 8-bit integer # (-128 to +127) into register r*X*
loadr r*X*, r*Y*	`0100 XXXX YYYY 0000`	Load register r*X* from memory word addressed by r*Y*: r*X* = memory[r*Y*]
storer r*X*, r*Y*	`0100 XXXX YYYY 0001`	Store contents of register r*X* into memory word addressed by r*Y*: memory[r*Y*] = r*X*
popr r*X* r*Y*	`0100 XXXX YYYY 0010`	Load contents of register r*X* from stack pointed to by register r*Y*: r*Y* -= 1; r*X* = memory[r*Y*]
pushr r*X* r*Y*	`0100 XXXX YYYY 0011`	Store contents of register r*X* onto stack pointed to by register r*Y*: memory[r*Y*] = r*X*; r*Y* += 1
loadn r*X*, #	`0010 XXXX #### ####`	Load register rX with memory word at address #
storen r*X*, #	`0011 XXXX #### ####`	Store contents of register r*X* into memory word at address #
addn r*X*, #	`0101 XXXX #### ####`	Add the 8-bit integer # (-128 to 127) to register r*X*
copy r*X*, r*Y*	`0110 XXXX YYYY 0000`	Set r*X* = r*Y*
neg r*X*, r*Y*	`0111 XXXX 0000 YYYY`	Set r*X* = -r*Y*
add r*X*, r*Y*, rZ	`0110 XXXX YYYY ZZZZ`	Set r*X* = r*Y* + rZ
sub r*X*, r*Y*, rZ	`0111 XXXX YYYY ZZZZ`	Set r*X* = r*Y* - rZ
mul r*X*, r*Y*, rZ	`1000 XXXX YYYY ZZZZ`	Set r*X* = r*Y* * rZ
div r*X*, r*Y*, rZ	`1001 XXXX YYYY ZZZZ`	Set r*X* = r*Y* // rZ
mod r*X*, r*Y*, rZ	`1010 XXXX YYYY ZZZZ`	Set r*X* = r*Y* % rZ
jumpr rX	`0000 XXXX 0000 0011`	Set program counter to address in r*X*
jumpn n	`1011 0000 #### ####`	Set program counter to address #
jeqzn r*X*, #	`1100 XXXX #### ####`	If r*X* = 0 then set program counter to address #
jnezn r*X*, #	`1101 XXXX #### ####`	If r*X* ≠ 0 then set program counter to address #
jgtzn r*X*, #	`1110 XXXX #### ####`	If r*X* > 0 then set program counter to address #
jltzn r*X*, #	`1111 XXXX #### ####`	If r*X* < 0 then set program counter to address #
calln r*X*, #	`1011 XXXX #### ####`	Set r*X* to (next) program counter, then set program counter to address #

Machine Language



The Hmmm Instruction Set

There are 26 different instructions in Hmmm, each of which accepts between 0 and 3 arguments. Two of the instructions, setn 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
nop	`0110 0000 0000 0000`	Do nothing
read r*X*	`0000 xxxx 0000 0001`	Stop for user input, which will then be stored in register r*X* (input is an integer from -32768 to +3276 Prints "Enter number: " to prompt user for input
write r*X*	`0000 xxx 0010`	Print the contents of register r*X* on standard output
setn r*X*, #	`0001 xxxx # ###`	Load an 8-bit integer # (-128 to +127) into register r*X*
loadr r*X*, r*Y*	`0100 xxxx yyy`	ory word addressed by r*Y*: r*X* = memory[r*Y*]
storer r*X*, r*Y*	`0100 xxxx yyyy`	*X* into memory word addressed by r*Y*: memory[r*Y*] = r*X*
popr r*X* r*Y*	`0100 xxxx yyyy`	*X* from stack pointed to by register r*Y*: r*Y* -= 1; r*X* = memory[r*Y*]
pushr r*X* r*Y*	`0100 xxxx yyyy`	*X* onto stack pointed to by register r*Y*: memory[r*Y*] = r*X*; r*Y* += 1
loadn r*X*, #	`0010 xxxx ####`	ory word at address #
storen r*X*, #	`0011 xxxx ####`	*X* into memory word at address #
addn r*X*, #	`0101 xxxx #### ####`	Add the 8-bit integer # (-128 to 127) to register r*X*
copy r*X*, r*Y*	`0110 xxxx yyyy 0000`	Set r*X* = r*Y*
neg r*X*, r*Y*	`0111 xxxx 0000 yyyy`	Set r*X* = -r*Y*
add r*X*, r*Y*, r*Z*	`0110 xxxx yyyy zzzz`	Set r*X* = r*Y* + r*Z*
sub r*X*, r*Y*, r*Z*	`0111 xxxx yyyy zzzz`	Set r*X* = r*Y* - r*Z*
mul r*X*, r*Y*, r*Z*	`1000 xxxx yyyy zzzz`	S r*X* = r*Y* * r*Z*
div r*X*, r*Y*, r*Z*	`1001 xxxx yyyy zzz`	r*Z*
mod r*X*, r*Y*, r*Z*	`1010 xxxx yyyy zzz`	
jumpr rX	`0000 xxxx 0000 0011`	
jumpn n	`1011 0000 #### ####`	
jeqzn r*X*, #	`1100 xxxx #### ####`	to address #
jnezn r*X*, #	`1101 xxxx #### ####`	then set program counter to address #
jgtzn r*X*, #	`1110 xxxx #### ####`	If r*X* > 0 then set program counter to address #
jltzn r*X*, #	`1111 xxxx #### ####`	If r*X* < 0 then set program counter to address #
calln r*X*, #	`1011 xxxx #### ####`	Set r*X* to (next) program counter, then set program counter to address #

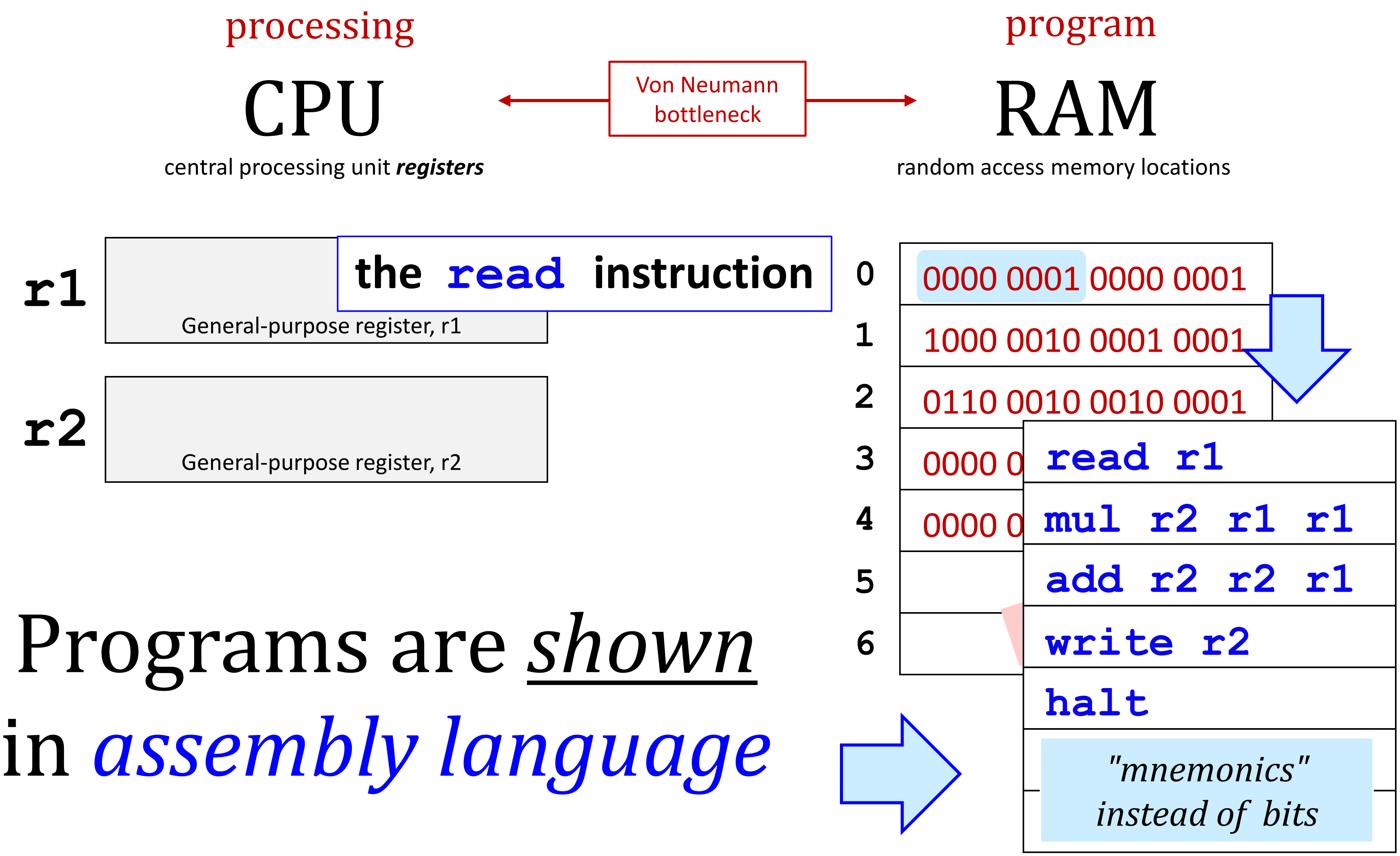
the `read` instruction

which register to `read` into?

the "bitpatterns" do matter!

Machine Language

“Von Neumann” Architecture



processing

CPU

central processing unit ***registers***

Von Neumann bottleneck

program

RAM

random access memory locations

r1 General-purpose register, r1

r2 General-purpose register, r2

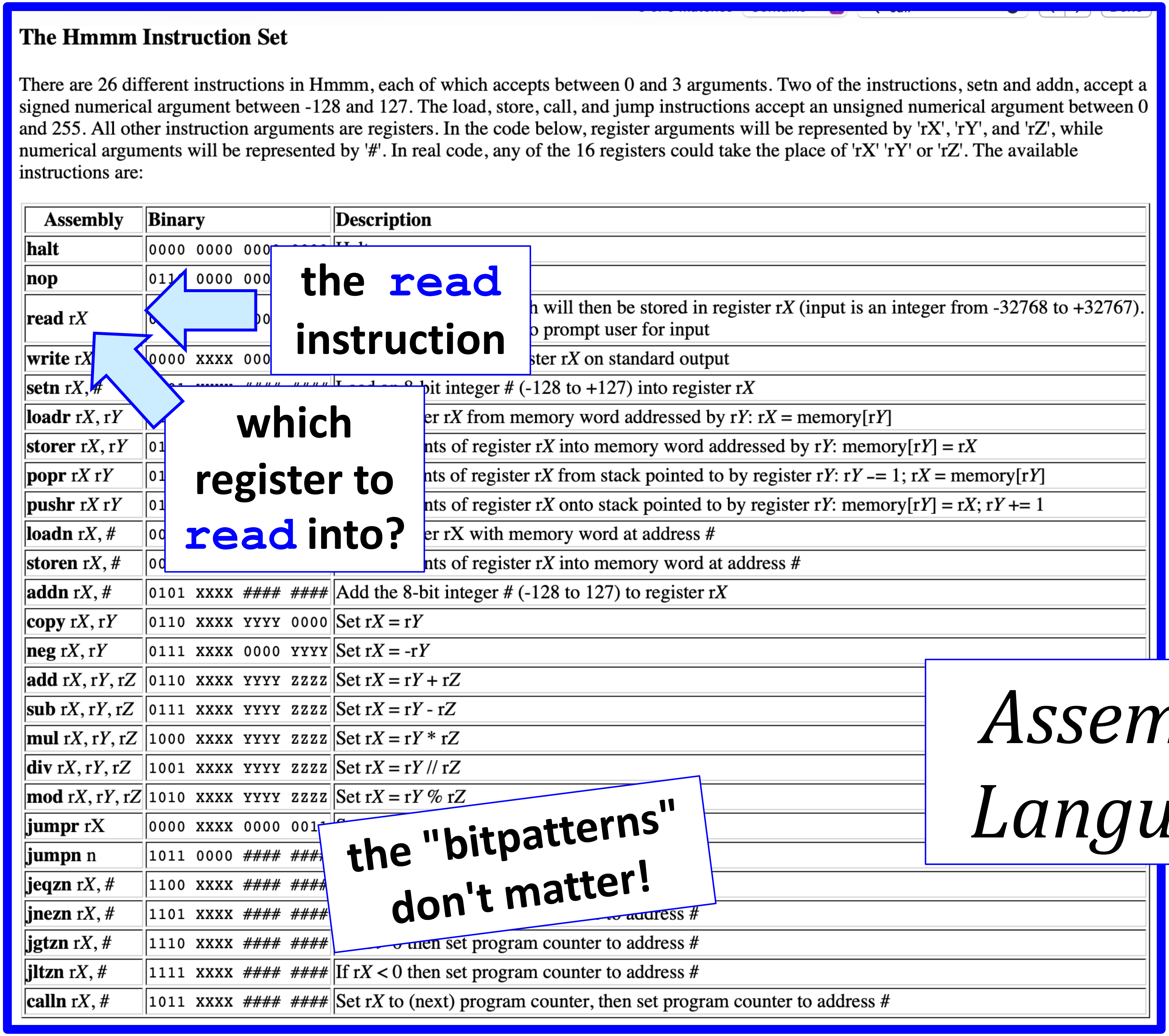
the **`read`** instruction

0 0000 0001 0000 0001
1 1000 0010 0001 0001
2 0110 0010 0010 0001
3 0000 0
4 0000 0
5
6

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

"mnemonics" instead of bits

Programs are <u>*shown*</u> in *assembly language*



The Hmmm Instruction Set

There are 26 different instructions in Hmmm, each of which accepts between 0 and 3 arguments. Two of the instructions, setn 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:

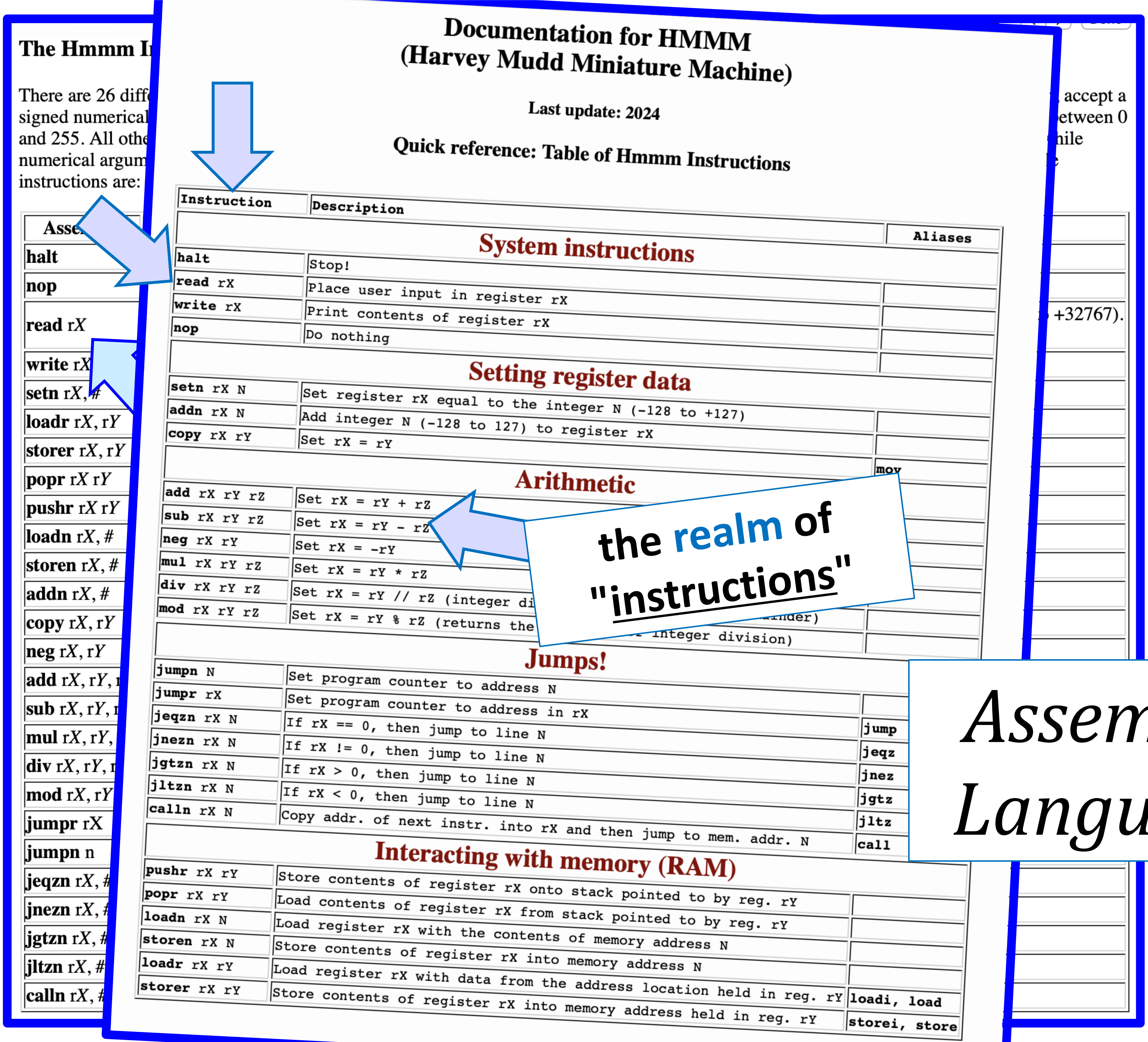
Assembly	Binary	Description
halt	`0000 0000 000`	
nop	`011 0000 000`	
read r*X*	`0 0`	h will then be stored in register r*X* (input is an integer from -32768 to +32767). o prompt user for input
write r*X*	`0000 xxxx 000`	ster r*X* on standard output
setn r*X*, #		bit integer # (-128 to +127) into register r*X*
loadr r*X*, r*Y*		er r*X* from memory word addressed by r*Y*: r*X* = memory[r*Y*]
storer r*X*, r*Y*	`01`	nts of register r*X* into memory word addressed by r*Y*: memory[r*Y*] = r*X*
popr r*X* r*Y*	`01`	nts of register r*X* from stack pointed to by register r*Y*: r*Y* -= 1; r*X* = memory[r*Y*]
pushr r*X* r*Y*	`01`	nts of register r*X* onto stack pointed to by register r*Y*: memory[r*Y*] = r*X*; r*Y* += 1
loadn r*X*, #	`00`	er rX with memory word at address #
storen r*X*, #	`00`	nts of register r*X* into memory word at address #
addn r*X*, #	`0101 xxxx #### ####`	Add the 8-bit integer # (-128 to 127) to register r*X*
copy r*X*, r*Y*	`0110 xxxx yyyy 0000`	Set r*X* = r*Y*
neg r*X*, r*Y*	`0111 xxxx 0000 yyyy`	Set r*X* = -r*Y*
add r*X*, r*Y*, r*Z*	`0110 xxxx yyyy zzzz`	Set r*X* = r*Y* + r*Z*
sub r*X*, r*Y*, r*Z*	`0111 xxxx yyyy zzzz`	Set r*X* = r*Y* - r*Z*
mul r*X*, r*Y*, r*Z*	`1000 xxxx yyyy zzzz`	Set r*X* = r*Y* * r*Z*
div r*X*, r*Y*, r*Z*	`1001 xxxx yyyy zzzz`	Set r*X* = r*Y* // r*Z*
mod r*X*, r*Y*, r*Z*	`1010 xxxx yyyy zzzz`	Set r*X* = r*Y* % r*Z*
jumpr rX	`0000 xxxx 0000 001`	
jumpn n	`1011 0000 #### ###`	
jeqzn r*X*, #	`1100 xxxx #### ####`	
jnezn r*X*, #	`1101 xxxx #### ####`	address #
jgtzn r*X*, #	`1110 xxxx #### ####`	then set program counter to address #
jltzn r*X*, #	`1111 xxxx #### ####`	If r*X* < 0 then set program counter to address #
calln r*X*, #	`1011 xxxx #### ####`	Set r*X* to (next) program counter, then set program counter to address #

the `read` instruction

which register to `read` into?

the "bitpatterns" don't matter!

Assembly Language



The Hmmm I

There are 26 diff
signed numerical
and 255. All othe
numerical argum
instructions are:

accept a
etween 0
hile

+32767).

Asse
halt
nop
read rX
write rX
setn rX, #
loadr rX, rY
storer rX, rY
popr rX rY
pushr rX rY
loadn rX, #
storen rX, #
addn rX, #
copy rX, rY
neg rX, rY
add rX, rY, r
sub rX, rY, r
mul rX, rY,
div rX, rY, r
mod rX, rY
jumpr rX
jumpn n
jeqzn rX, #
jnezn rX, #
jgtzn rX, #
jltzn rX, #
calln rX, #

Documentation for HMMM
(Harvey Mudd Miniature Machine)

Last update: 2024

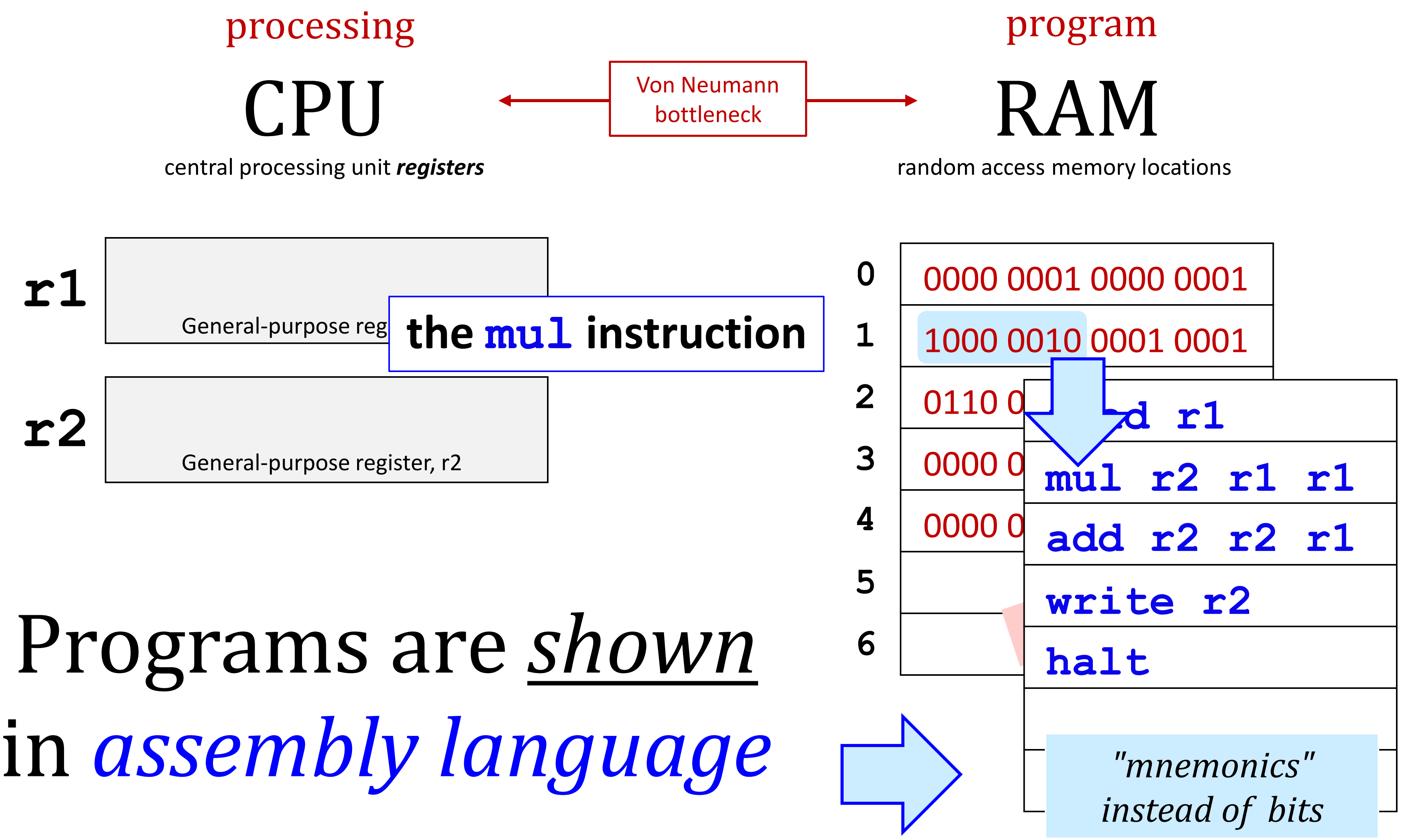
Quick reference: Table of Hmmm Instructions

Instruction	Description	Aliases
System instructions		
halt	Stop!	
read rX	Place user input in register rX	
write rX	Print contents of register rX	
nop	Do nothing	
Setting register data		
setn rX N	Set register rX equal to the integer N (-128 to +127)	
addn rX N	Add integer N (-128 to 127) to register rX	
copy rX rY	Set rX = rY	**mov**
Arithmetic		
add rX rY rZ	Set rX = rY + rZ	
sub rX rY rZ	Set rX = rY - rZ	
neg rX rY	Set rX = -rY	
mul rX rY rZ	Set rX = rY * rZ	
div rX rY rZ	Set rX = rY // rZ (integer di nder)	
mod rX rY rZ	Set rX = rY % rZ (returns the integer division)	
Jumps!		
jumpn N	Set program counter to address N	
jumpr rX	Set program counter to address in rX	**jump**
jeqzn rX N	If rX == 0, then jump to line N	**jeqz**
jnezn rX N	If rX != 0, then jump to line N	**jnez**
jgtzn rX N	If rX > 0, then jump to line N	**jgtz**
jltzn rX N	If rX < 0, then jump to line N	**jltz**
calln rX N	Copy addr. of next instr. into rX and then jump to mem. addr. N	**call**
Interacting with memory (RAM)		
pushr rX rY	Store contents of register rX onto stack pointed to by reg. rY	
popr rX rY	Load contents of register rX from stack pointed to by reg. rY	
loadn rX N	Load register rX with the contents of memory address N	
storen rX N	Store contents of register rX into memory address N	
loadr rX rY	Load register rX with data from the address location held in reg. rY	**loadi, load**
storer rX rY	Store contents of register rX into memory address held in reg. rY	**storei, store**

the **realm** of "instructions"

Assembly Language

“Von Neumann” Architecture



processing

CPU

central processing unit ***registers***

Von Neumann bottleneck

program

RAM

random access memory locations

r1 General-purpose reg

the `mul` instruction

r2 General-purpose register, r2

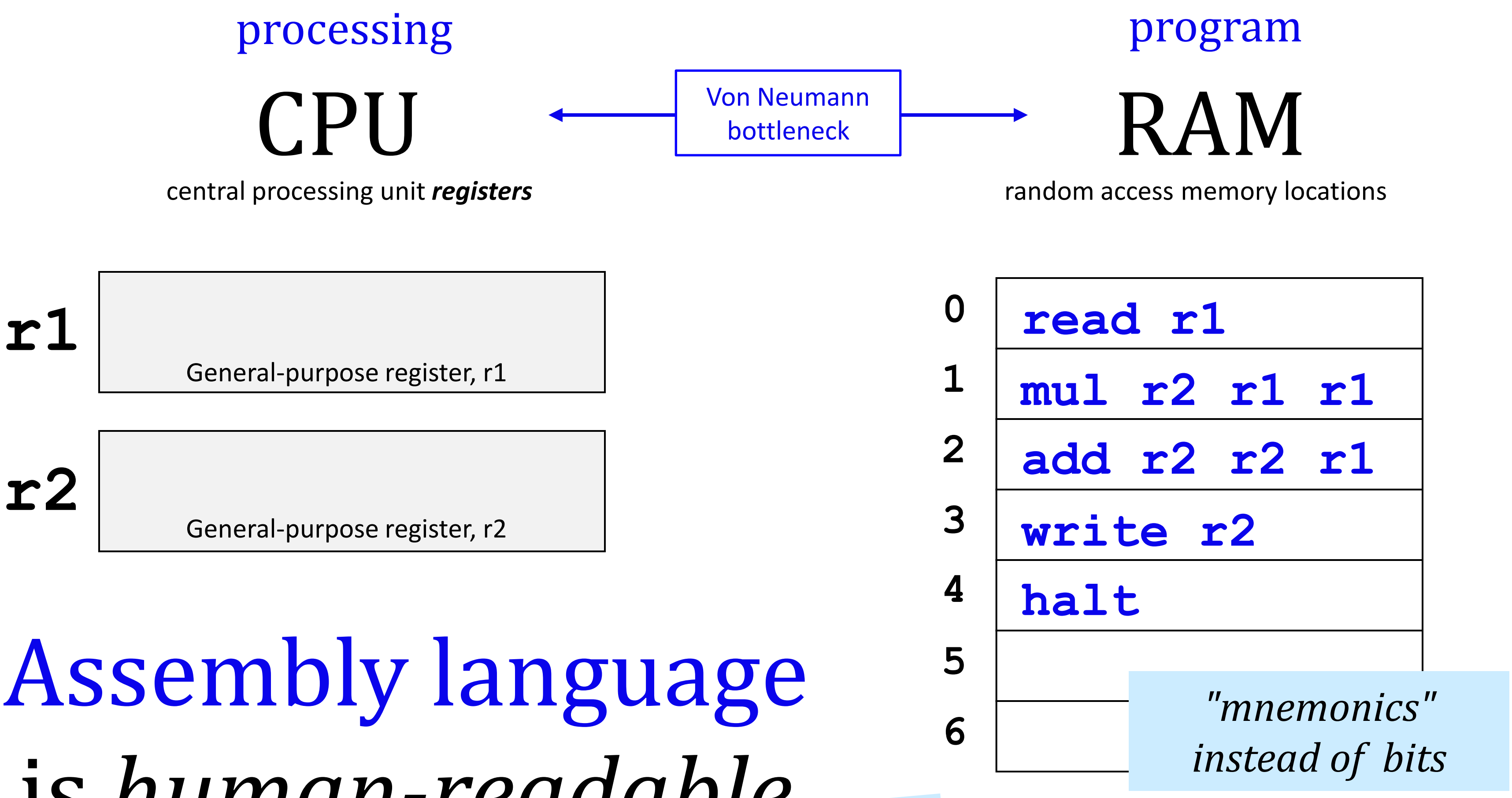
0	0000 0001 0000 0001
1	1000 0010 0001 0001
2	0110 0
3	0000 0
4	0000 0
5	
6	

```
 d r1
mul r2 r1 r1
add r2 r2 r1
write r2
halt
```

"mnemonics" instead of bits

Programs are *<u>shown</u>* in *assembly language*

“Von Neumann” Architecture



processing

CPU

central processing unit ***registers***

Von Neumann bottleneck

program

RAM

random access memory locations

r1 General-purpose register, r1

r2 General-purpose register, r2

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

"mnemonics" instead of bits

Assembly language is *human-readable* machine language

Human readable? I doubt it!

Example #1:

Screen

6 (input)

a five-line assembly-language program

6

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

Example #1:

Screen

6 (input)

42

CPU

central processing unit ***registers***

Von Neumann bottleneck

RAM

random access memory locations

r1 6

General-purpose register r1

r2 42 ~~36~~

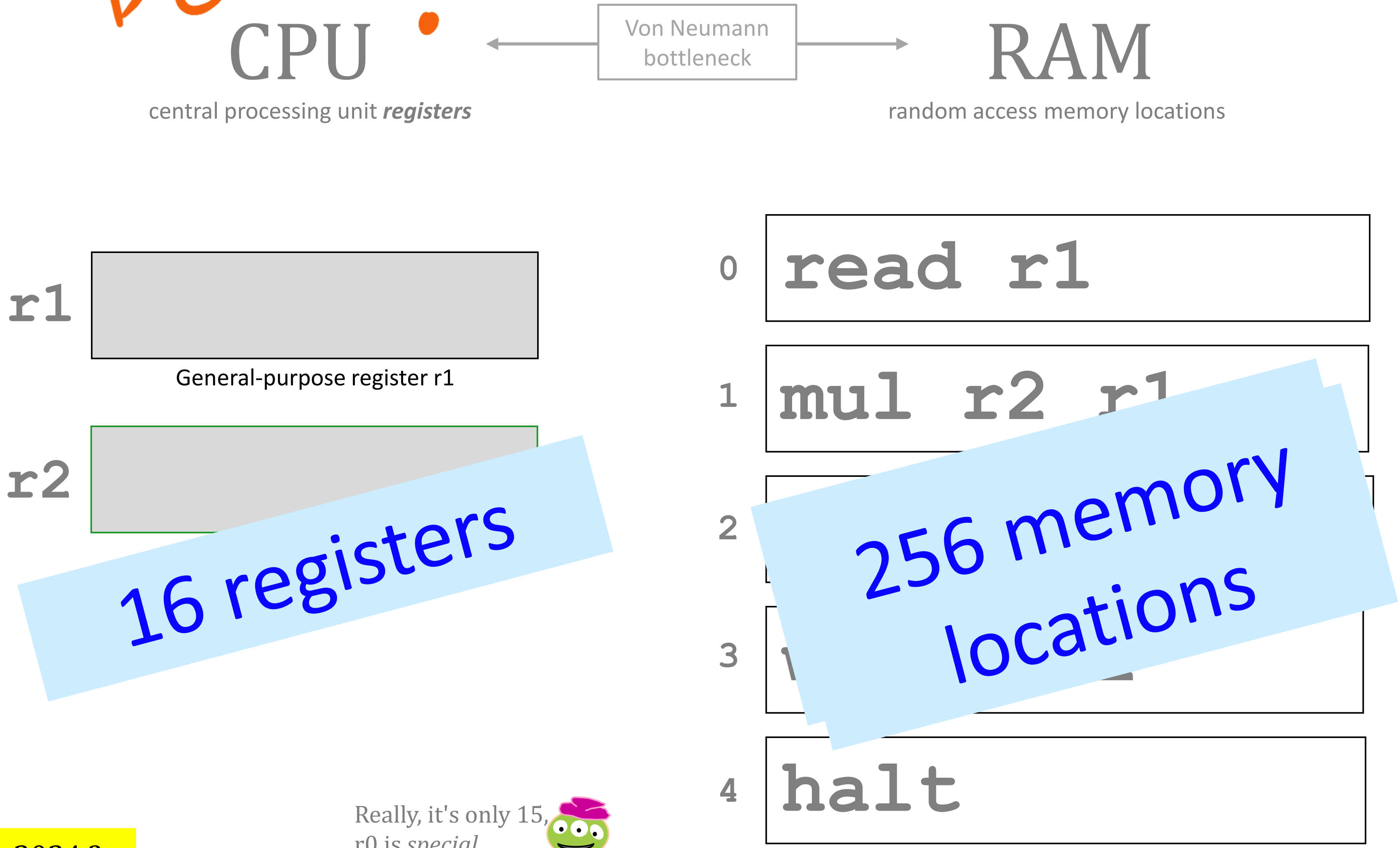
General-purpose register r2

6

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

Hmmm: Harvey mudd miniature machine

Demo!



CPU

central processing unit ***registers***

Von Neumann bottleneck

RAM

random access memory locations

Really, it's only 15,
r0 is *special*

vs. 2024 ?

Hmmm vs 2024

CPU

central processing unit ***registers***

Von Neumann bottleneck

RAM

random access memory locations

`r1`

General-purpose register r1

`r2`

16 registers

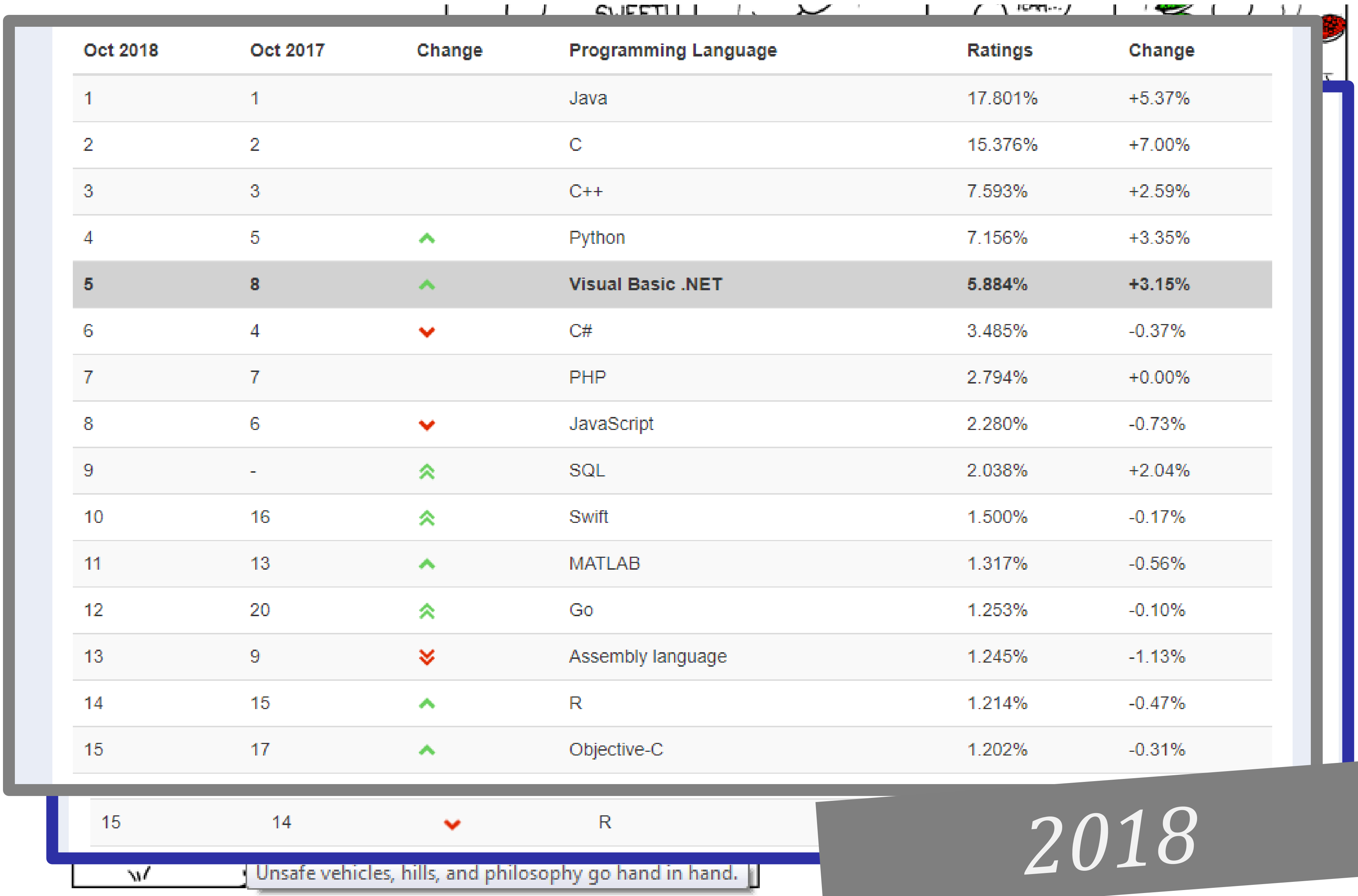
```
0  read r1
1  mul r2 r1
2
3
4  halt
```

256 memory locations

2022 Arm M1: **37-40** registers per core

2024: **~16,000,000,000** mem loc's

Why Assembly?



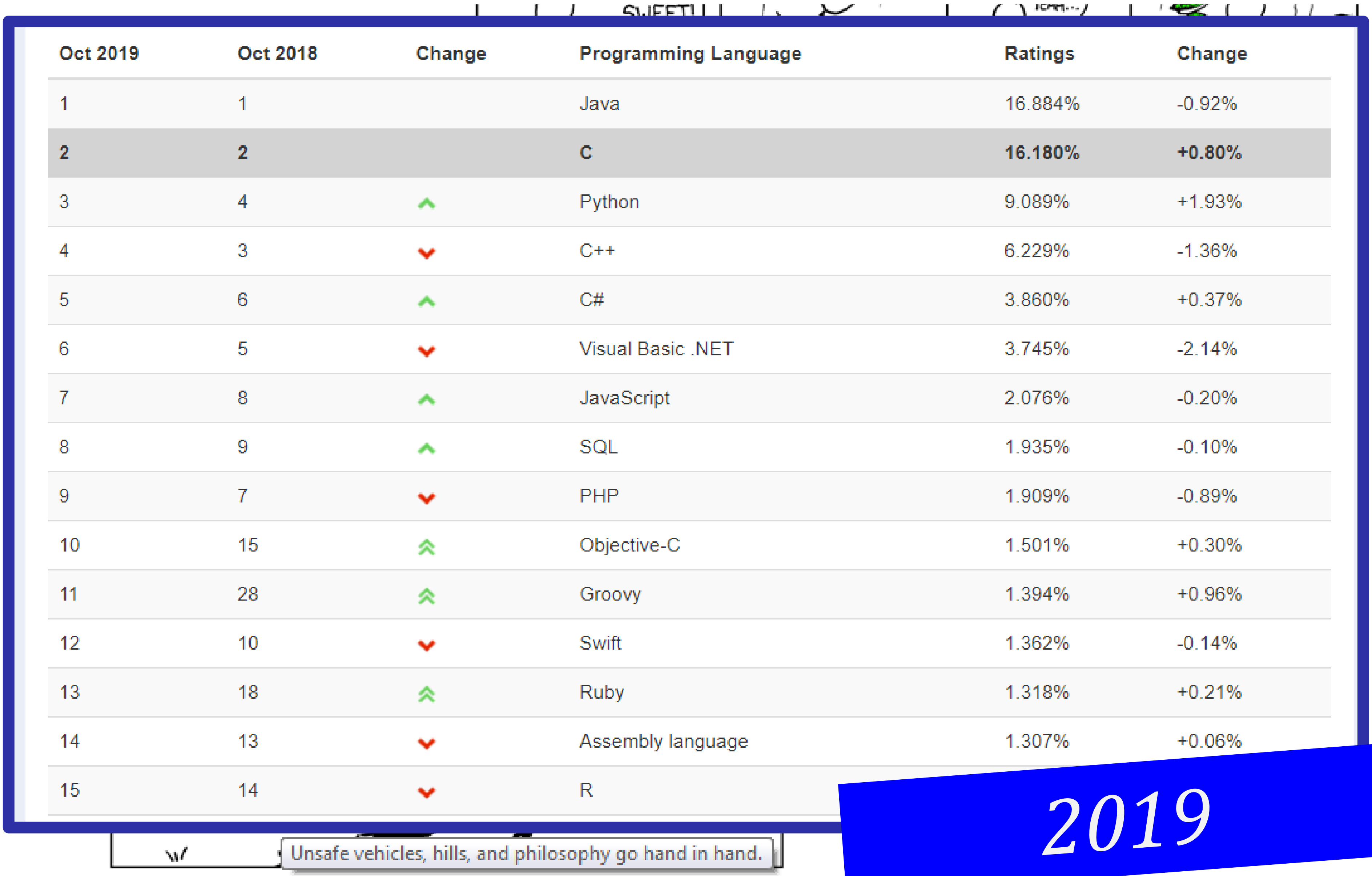
Oct 2018	Oct 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.801%	+5.37%
2	2		C	15.376%	+7.00%
3	3		C++	7.593%	+2.59%
4	5	▲	Python	7.156%	+3.35%
5	**8**	▲	**Visual Basic .NET**	**5.884%**	**+3.15%**
6	4	▼	C#	3.485%	-0.37%
7	7		PHP	2.794%	+0.00%
8	6	▼	JavaScript	2.280%	-0.73%
9	-	▲▲	SQL	2.038%	+2.04%
10	16	▲▲	Swift	1.500%	-0.17%
11	13	▲	MATLAB	1.317%	-0.56%
12	20	▲▲	Go	1.253%	-0.10%
13	9	▼▼	Assembly language	1.245%	-1.13%
14	15	▲	R	1.214%	-0.47%
15	17	▲	Objective-C	1.202%	-0.31%

15 14 ▼ R

Unsafe vehicles, hills, and philosophy go hand in hand.

2018

Why Assembly?

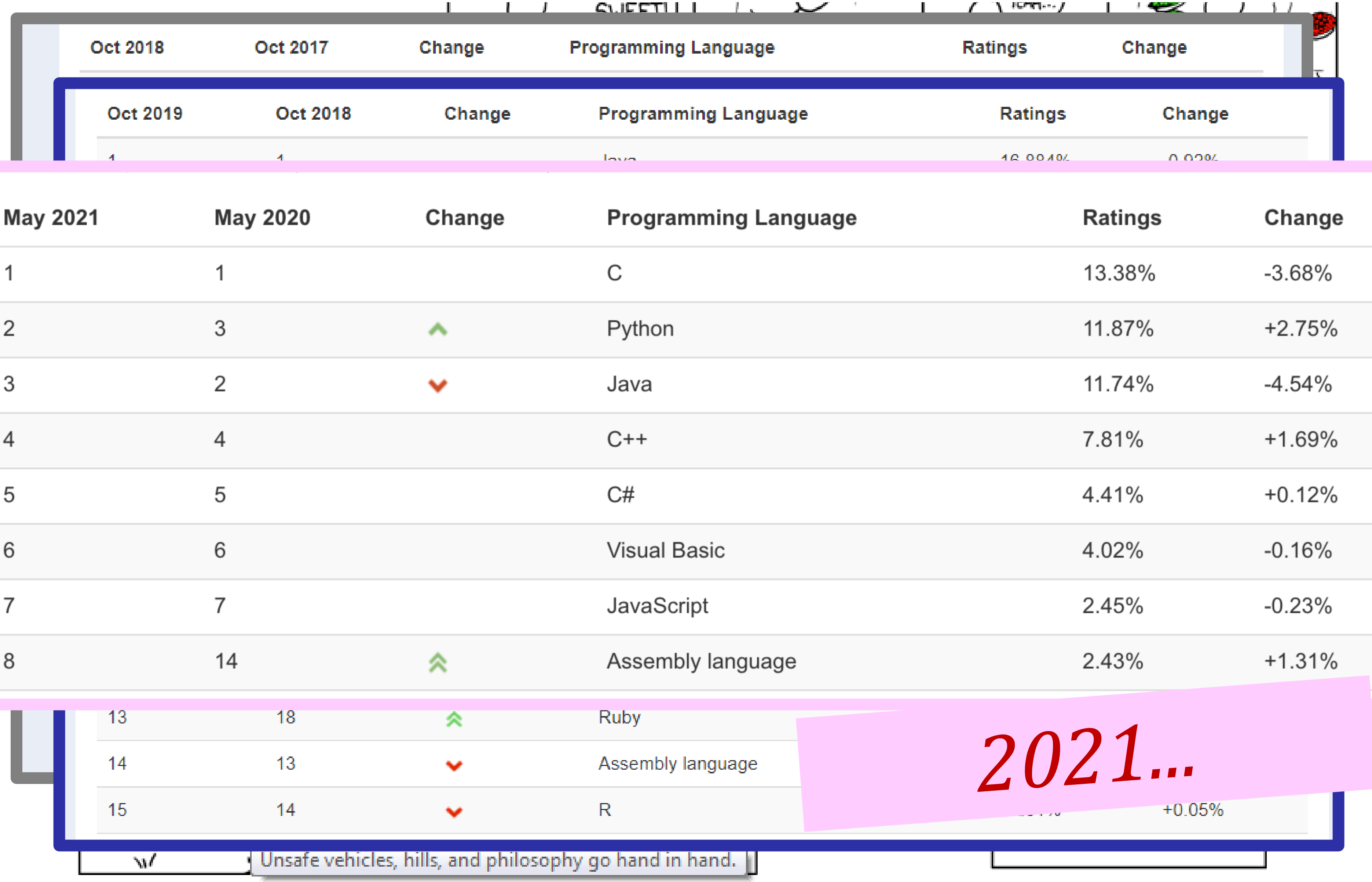


Oct 2019	Oct 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.884%	-0.92%
2	**2**		**C**	**16.180%**	**+0.80%**
3	4	▲	Python	9.089%	+1.93%
4	3	▼	C++	6.229%	-1.36%
5	6	▲	C#	3.860%	+0.37%
6	5	▼	Visual Basic .NET	3.745%	-2.14%
7	8	▲	JavaScript	2.076%	-0.20%
8	9	▲	SQL	1.935%	-0.10%
9	7	▼	PHP	1.909%	-0.89%
10	15	▲▲	Objective-C	1.501%	+0.30%
11	28	▲▲	Groovy	1.394%	+0.96%
12	10	▼	Swift	1.362%	-0.14%
13	18	▲▲	Ruby	1.318%	+0.21%
14	13	▼	Assembly language	1.307%	+0.06%
15	14	▼	R		

Unsafe vehicles, hills, and philosophy go hand in hand.

2019

Why Assembly?



Oct 2018	Oct 2017	Change	Programming Language	Ratings	Change

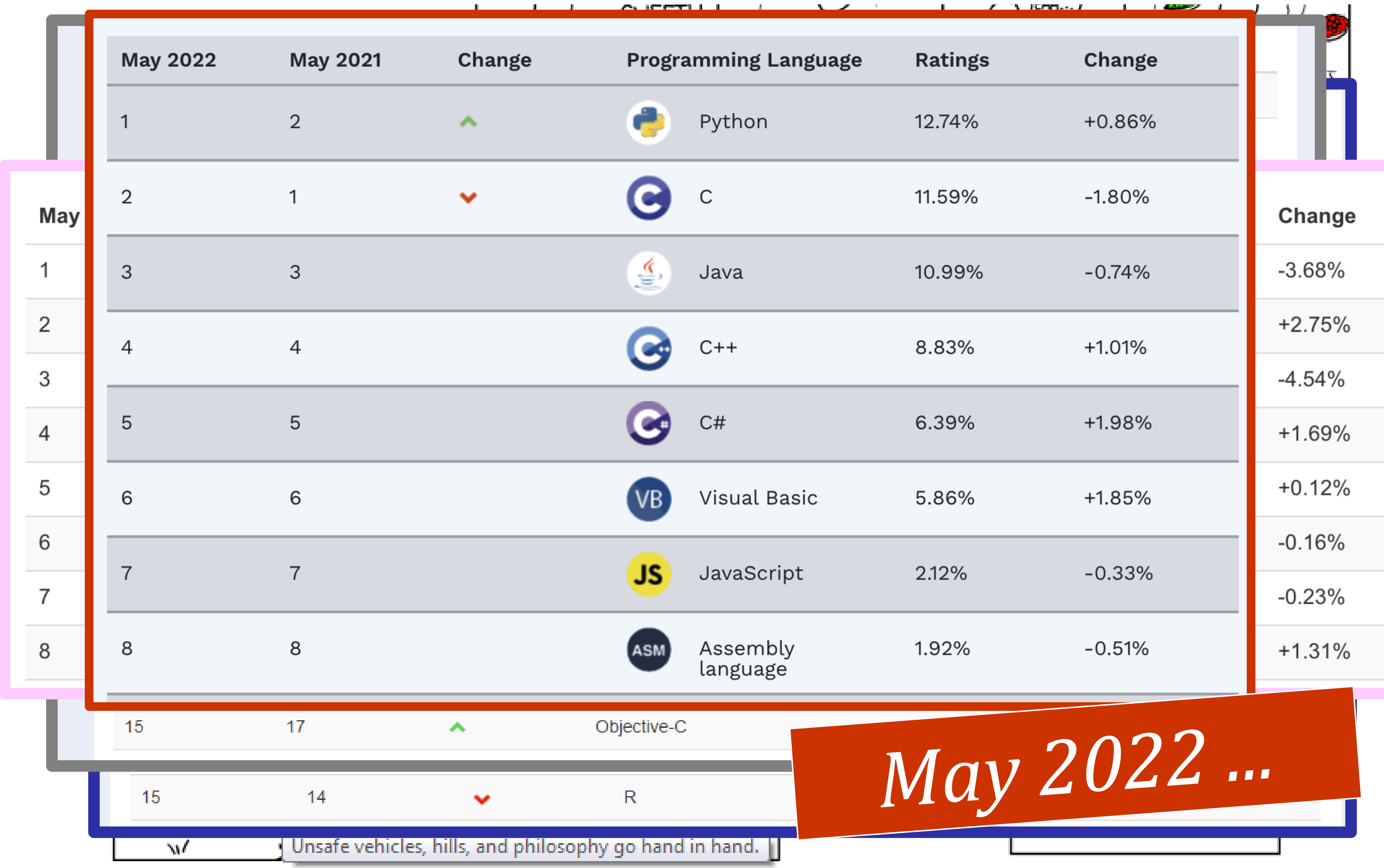
Oct 2019	Oct 2018	Change	Programming Language	Ratings	Change
13	18	⏫	Ruby		
14	13	▼	Assembly language		
15	14	▼	R		+0.05%

May 2021	May 2020	Change	Programming Language	Ratings	Change
1	1		C	13.38%	-3.68%
2	3	▲	Python	11.87%	+2.75%
3	2	▼	Java	11.74%	-4.54%
4	4		C++	7.81%	+1.69%
5	5		C#	4.41%	+0.12%
6	6		Visual Basic	4.02%	-0.16%
7	7		JavaScript	2.45%	-0.23%
8	14	⏫	Assembly language	2.43%	+1.31%

Unsafe vehicles, hills, and philosophy go hand in hand.

2021...

Why Assembly?



May 2022	May 2021	Change	Programming Language	Ratings	Change
1	2	▲	Python	12.74%	+0.86%
2	1	▼	C	11.59%	-1.80%
3	3		Java	10.99%	-0.74%
4	4		C++	8.83%	+1.01%
5	5		C#	6.39%	+1.98%
6	6		VB Visual Basic	5.86%	+1.85%
7	7		JS JavaScript	2.12%	-0.33%
8	8		ASM Assembly language	1.92%	-0.51%

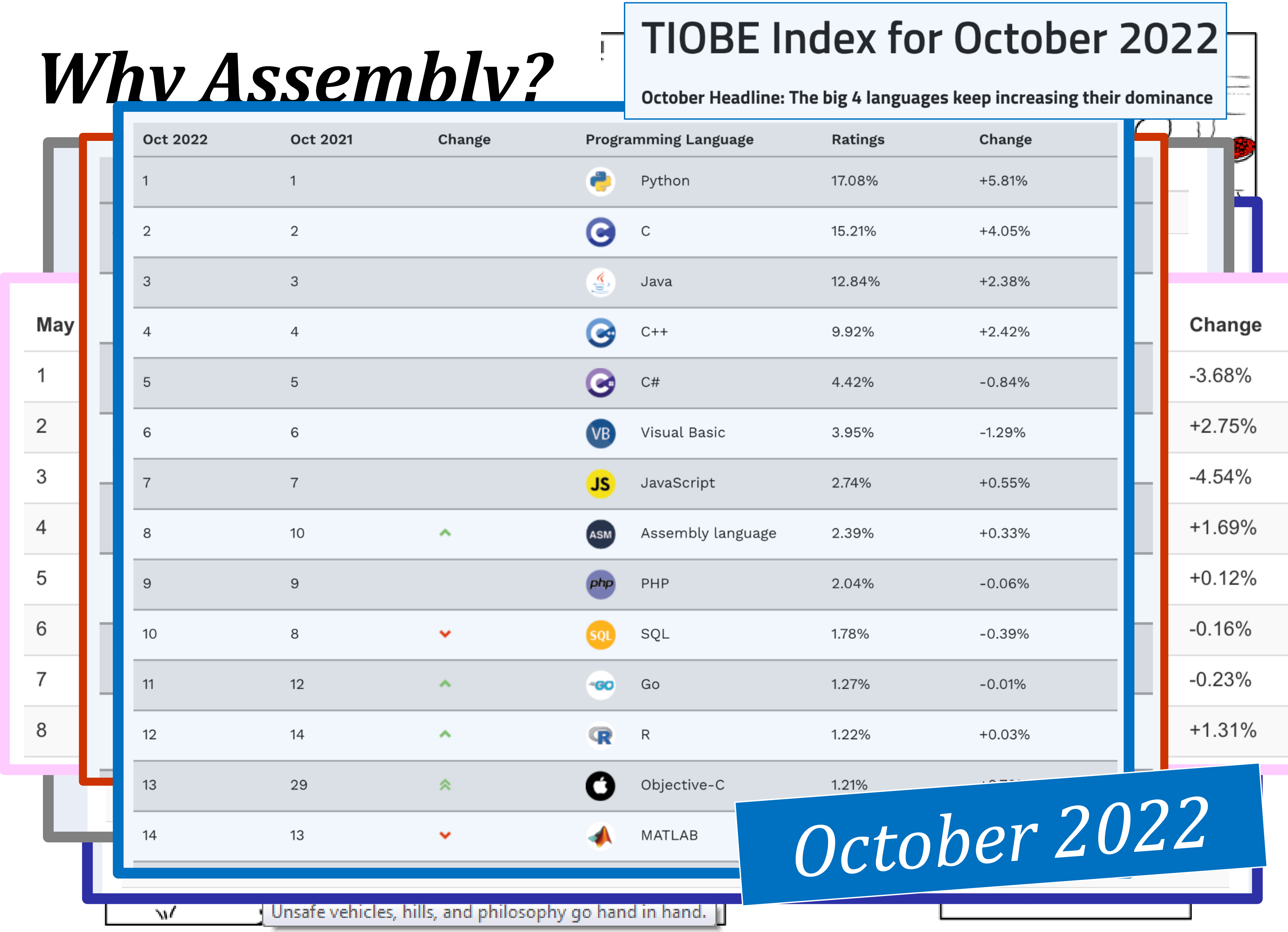
May	Change
1	-3.68%
2	+2.75%
3	-4.54%
4	+1.69%
5	+0.12%
6	-0.16%
7	-0.23%
8	+1.31%

15	17	▲	Objective-C
15	14	▼	R

Unsafe vehicles, hills, and philosophy go hand in hand.

May 2022 ...

Why Assembly?



TIOBE Index for October 2022

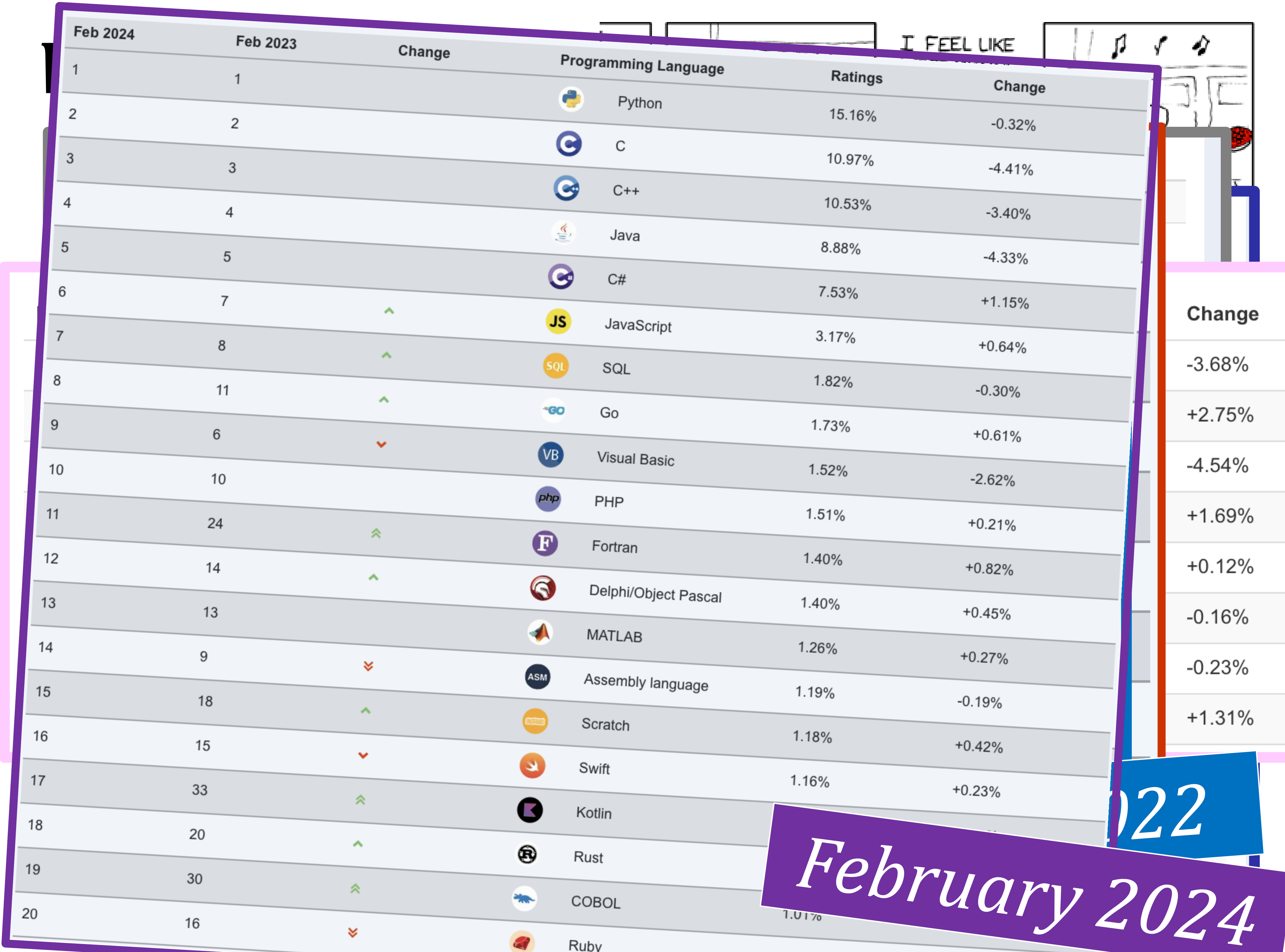
October Headline: The big 4 languages keep increasing their dominance

Oct 2022	Oct 2021	Change	Programming Language	Ratings	Change
1	1		Python	17.08%	+5.81%
2	2		C	15.21%	+4.05%
3	3		Java	12.84%	+2.38%
4	4		C++	9.92%	+2.42%
5	5		C#	4.42%	-0.84%
6	6		Visual Basic	3.95%	-1.29%
7	7		JavaScript	2.74%	+0.55%
8	10	˄	Assembly language	2.39%	+0.33%
9	9		PHP	2.04%	-0.06%
10	8	˅	SQL	1.78%	-0.39%
11	12	˄	Go	1.27%	-0.01%
12	14	˄	R	1.22%	+0.03%
13	29	⇈	Objective-C	1.21%	
14	13	˅	MATLAB		

May	Change
1	-3.68%
2	+2.75%
3	-4.54%
4	+1.69%
5	+0.12%
6	-0.16%
7	-0.23%
8	+1.31%

Unsafe vehicles, hills, and philosophy go hand in hand.

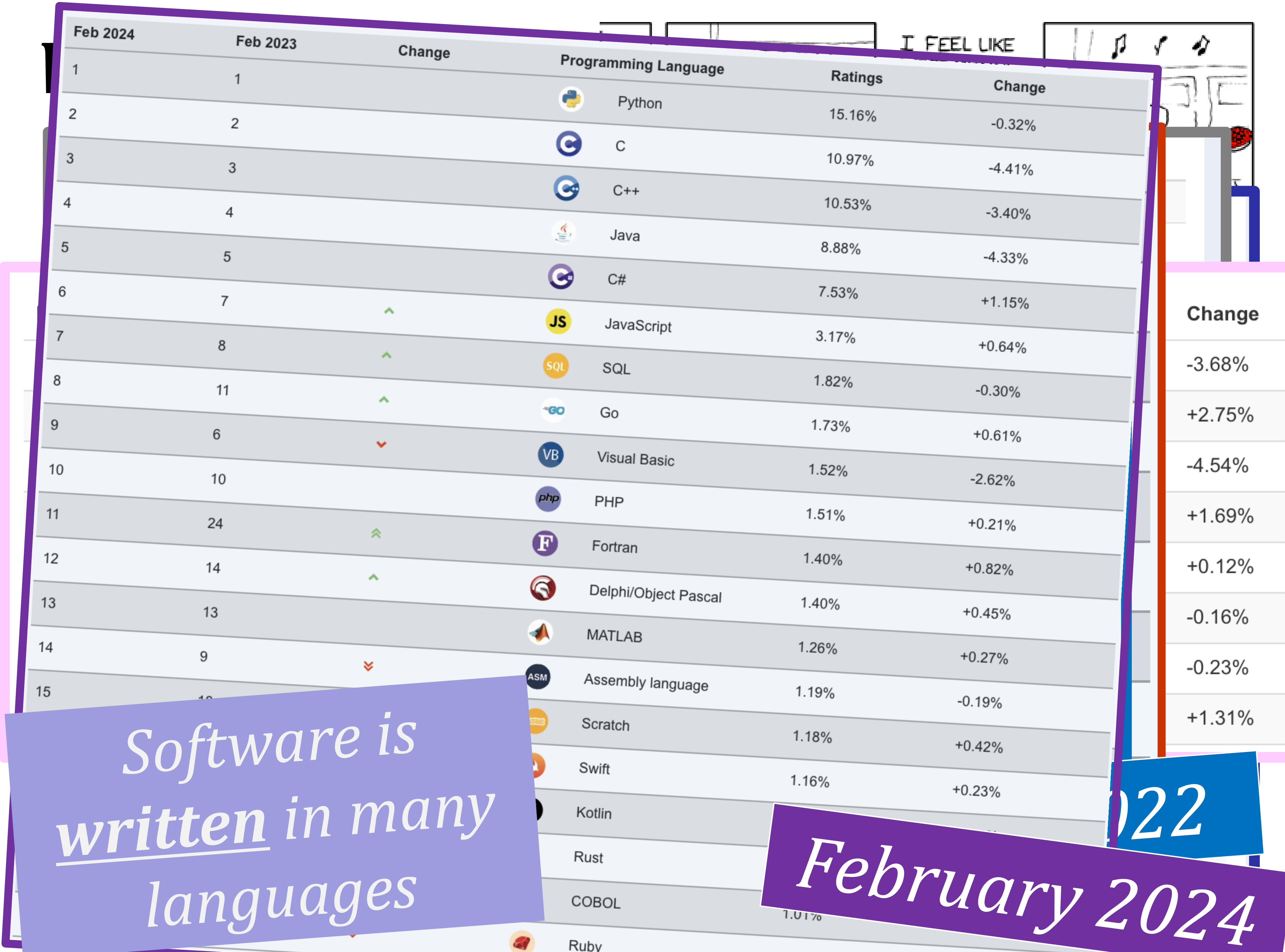
October 2022



February 2024

Feb 2024	Feb 2023	Change	Programming Language	Ratings	Change
1	1		Python	15.16%	-0.32%
2	2		C	10.97%	-4.41%
3	3		C++	10.53%	-3.40%
4	4		Java	8.88%	-4.33%
5	5		C#	7.53%	+1.15%
6	7	↑	JavaScript	3.17%	+0.64%
7	8	↑	SQL	1.82%	-0.30%
8	11	↑	Go	1.73%	+0.61%
9	6	↓	Visual Basic	1.52%	-2.62%
10	10		PHP	1.51%	+0.21%
11	24	⇈	Fortran	1.40%	+0.82%
12	14	↑	Delphi/Object Pascal	1.40%	+0.45%
13	13		MATLAB	1.26%	+0.27%
14	9	⇊	Assembly language	1.19%	-0.19%
15	18	↑	Scratch	1.18%	+0.42%
16	15	↓	Swift	1.16%	+0.23%
17	33	⇈	Kotlin		
18	20	↑	Rust		
19	30	⇈	COBOL	1.01%	
20	16	⇊	Ruby		

Change
-3.68%
+2.75%
-4.54%
+1.69%
+0.12%
-0.16%
-0.23%
+1.31%

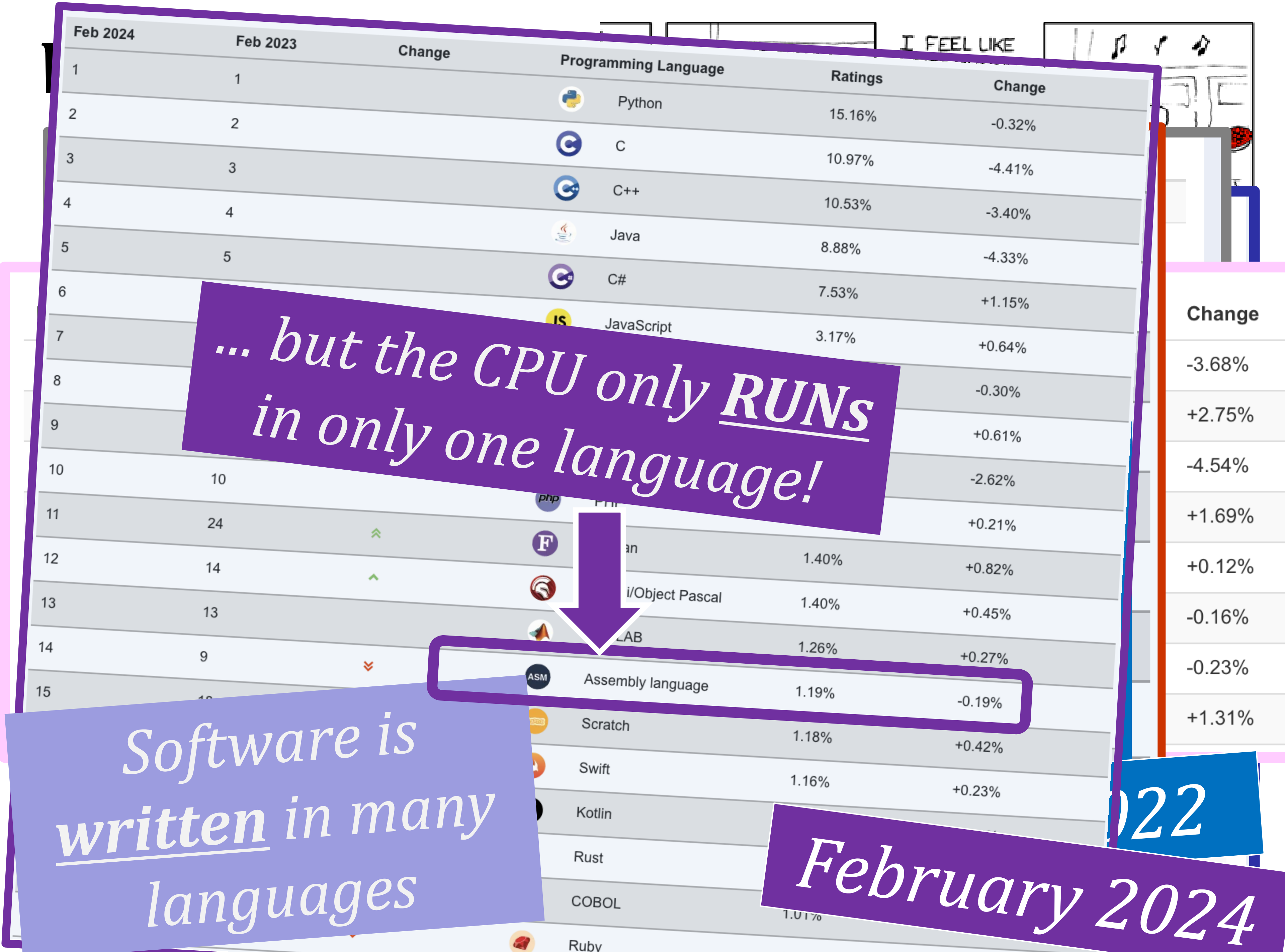


Feb 2024	Feb 2023	Change	Programming Language	Ratings	Change
1	1		Python	15.16%	-0.32%
2	2		C	10.97%	-4.41%
3	3		C++	10.53%	-3.40%
4	4		Java	8.88%	-4.33%
5	5		C#	7.53%	+1.15%
6	7	↑	JavaScript	3.17%	+0.64%
7	8	↑	SQL	1.82%	-0.30%
8	11	↑	Go	1.73%	+0.61%
9	6	↓	Visual Basic	1.52%	-2.62%
10	10		PHP	1.51%	+0.21%
11	24	⇈	Fortran	1.40%	+0.82%
12	14	↑	Delphi/Object Pascal	1.40%	+0.45%
13	13		MATLAB	1.26%	+0.27%
14	9	⇊	Assembly language	1.19%	-0.19%
15			Scratch	1.18%	+0.42%
			Swift	1.16%	+0.23%
			Kotlin		
			Rust		
			COBOL	1.01%	
			Ruby		

Change
-3.68%
+2.75%
-4.54%
+1.69%
+0.12%
-0.16%
-0.23%
+1.31%

Software is ***written*** in many languages

February 2024



Feb 2024	Feb 2023	Change	Programming Language	Ratings	Change
1	1		Python	15.16%	-0.32%
2	2		C	10.97%	-4.41%
3	3		C++	10.53%	-3.40%
4	4		Java	8.88%	-4.33%
5	5		C#	7.53%	+1.15%
6			JavaScript	3.17%	+0.64%
7					-0.30%
8					+0.61%
9					-2.62%
10	10		PH		+0.21%
11	24	⇈	an	1.40%	+0.82%
12	14	^	i/Object Pascal	1.40%	+0.45%
13	13		LAB	1.26%	+0.27%
14	9	⇊	Assembly language	1.19%	-0.19%
15			Scratch	1.18%	+0.42%
			Swift	1.16%	+0.23%
			Kotlin		
			Rust		
			COBOL	1.01%	
			Ruby		

Change
-3.68%
+2.75%
-4.54%
+1.69%
+0.12%
-0.16%
-0.23%
+1.31%

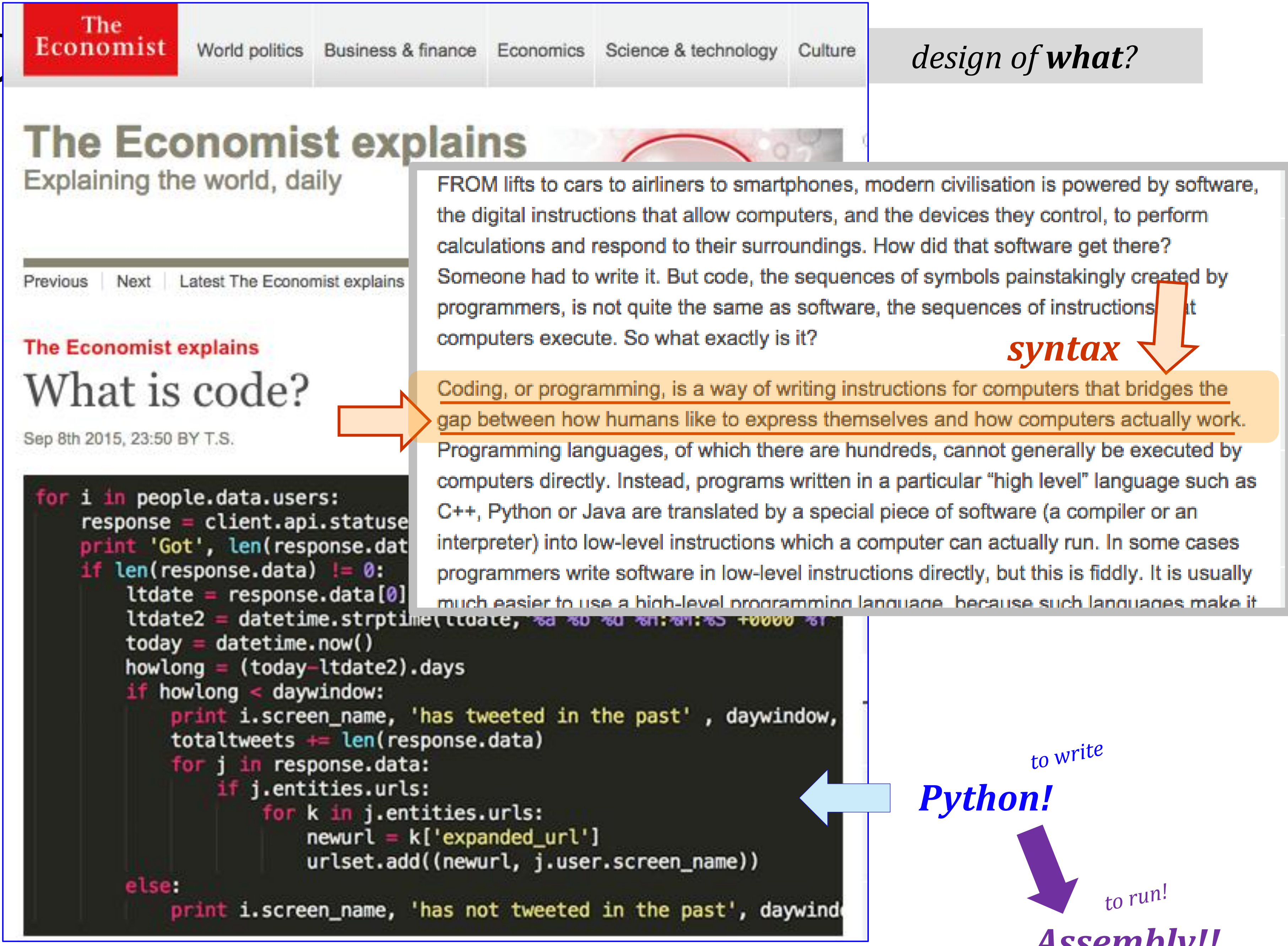
Software is ***written*** in many languages

… but the CPU only ***RUNs*** in only one language!

February 2024

D

design of ***what****?*



The Economist explains

Explaining the world, daily

The Economist explains

What is code?

Sep 8th 2015, 23:50 BY T.S.

FROM lifts to cars to airliners to smartphones, modern civilisation is powered by software, the digital instructions that allow computers, and the devices they control, to perform calculations and respond to their surroundings. How did that software get there? Someone had to write it. But code, the sequences of symbols painstakingly created by programmers, is not quite the same as software, the sequences of instructions that computers execute. So what exactly is it?

Coding, or programming, is a way of writing instructions for computers that bridges the gap between how humans like to express themselves and how computers actually work. Programming languages, of which there are hundreds, cannot generally be executed by computers directly. Instead, programs written in a particular "high level" language such as C++, Python or Java are translated by a special piece of software (a compiler or an interpreter) into low-level instructions which a computer can actually run. In some cases programmers write software in low-level instructions directly, but this is fiddly. It is usually much easier to use a high-level programming language, because such languages make it

```
for i in people.data.users:
    response = client.api.statuse
    print 'Got', len(response.dat
    if len(response.data) != 0:
        ltdate = response.data[0]
        ltdate2 = datetime.strptime(ltdate, '%a %b %d %H:%M:%S +0000 %Y
        today = datetime.now()
        howlong = (today-ltdate2).days
        if howlong < daywindow:
            print i.screen_name, 'has tweeted in the past' , daywindow,
            totaltweets += len(response.data)
            for j in response.data:
                if j.entities.urls:
                    for k in j.entities.urls:
                        newurl = k['expanded_url']
                        urlset.add((newurl, j.user.screen_name))
    else:
        print i.screen_name, 'has not tweeted in the past', daywind
```

syntax

to write

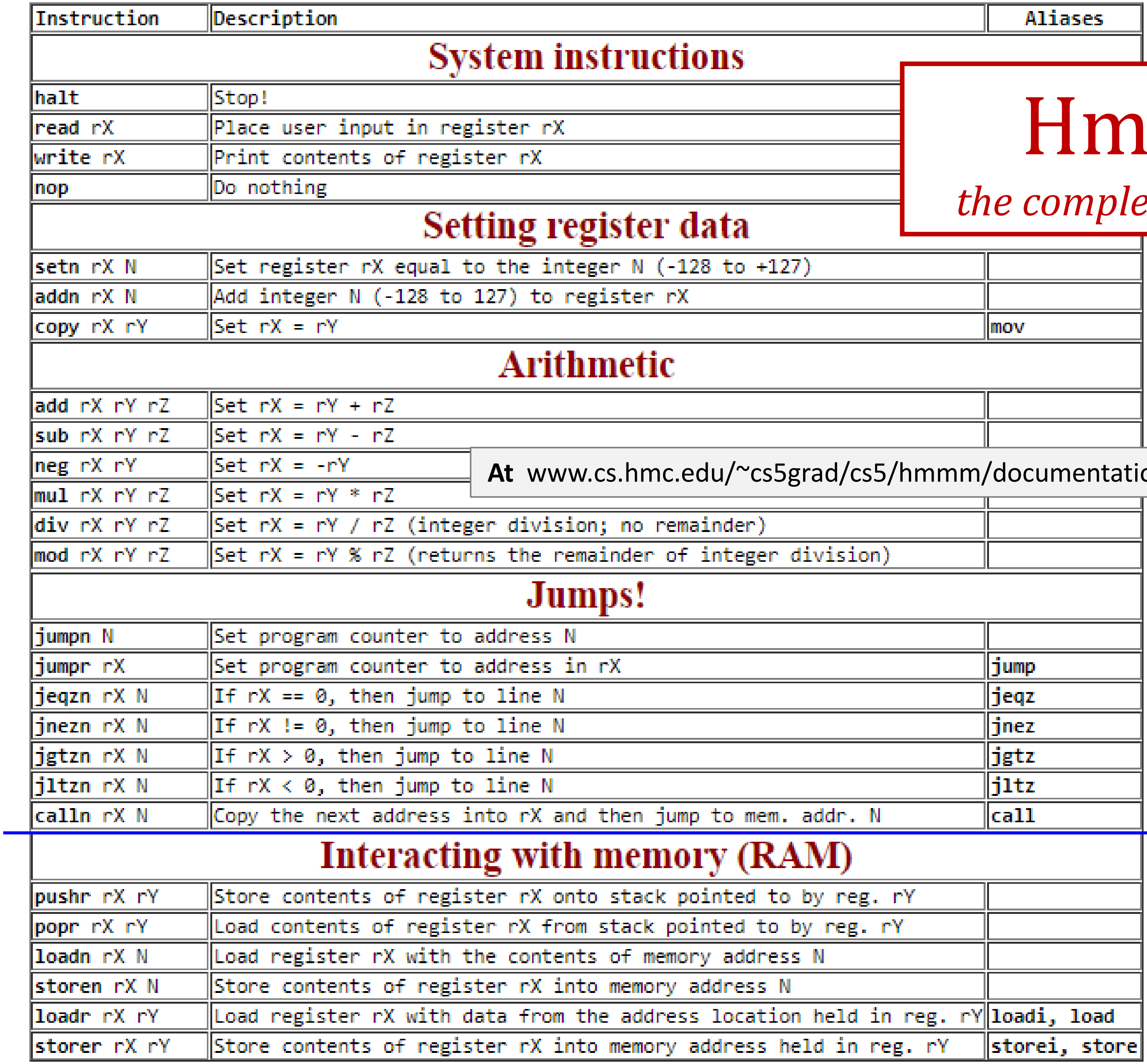
Python!

to run!

Assembly!!

Hmmm

the complete reference



Instruction	Description	Aliases
System instructions		
halt	Stop!	
read rX	Place user input in register rX	
write rX	Print contents of register rX	
nop	Do nothing	
Setting register data		
setn rX N	Set register rX equal to the integer N (-128 to +127)	
addn rX N	Add integer N (-128 to 127) to register rX	
copy rX rY	Set rX = rY	mov
Arithmetic		
add rX rY rZ	Set rX = rY + rZ	
sub rX rY rZ	Set rX = rY - rZ	
neg rX rY	Set rX = -rY	
mul rX rY rZ	Set rX = rY * rZ	
div rX rY rZ	Set rX = rY / rZ (integer division; no remainder)	
mod rX rY rZ	Set rX = rY % rZ (returns the remainder of integer division)	
Jumps!		
jumpn N	Set program counter to address N	
jumpr rX	Set program counter to address in rX	jump
jeqzn rX N	If rX == 0, then jump to line N	jeqz
jnezn rX N	If rX != 0, then jump to line N	jnez
jgtzn rX N	If rX > 0, then jump to line N	jgtz
jltzn rX N	If rX < 0, then jump to line N	jltz
calln rX N	Copy the next address into rX and then jump to mem. addr. N	call
Interacting with memory (RAM)		
pushr rX rY	Store contents of register rX onto stack pointed to by reg. rY	
popr rX rY	Load contents of register rX from stack pointed to by reg. rY	
loadn rX N	Load register rX with the contents of memory address N	
storen rX N	Store contents of register rX into memory address N	
loadr rX rY	Load register rX with data from the address location held in reg. rY	loadi, load
storer rX rY	Store contents of register rX into memory address held in reg. rY	storei, store

At www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html

Today

Thursday

Assembly Language

ought to be called ***register*** language

`read r1`	reads from keyboard into **`reg r1`**	
`write r2`	outputs **`reg r2`** onto the screen	
`setn r1 42`	`reg1 = 42`	you can replace 42 with anything from -128 to 127
`addn r1 -1`	`reg1 = reg1 - 1`	a shortcut
`add r3 r1 r2`	`reg3 = reg1 + reg2`	
`sub r3 r1 r2`	`reg3 = reg1 - reg2`	
`mul r2 r1 r1`	`reg2 = reg1 * reg1`	
`div r1 r1 r2`	`reg1 = reg1 // reg2`	ints only!

This is why assignment is written R to L in Python!

Names(s): ____________________

Try it!

"Quiz"

screen

100 (input)

(output)

CPU

central processing unit

r1

General-purpose register r1

r2

General-purpose register r2

r3

General-purpose register r3

r4

General-purpose register r4

RAM

random access memory

100

		Python
0	`read r1`	r1 = 100
1	`setn r2 7`	r2 = 7
2	`mod r4 r1 r2`	r4 = r1 % r2
3	`div r3 r1 r4`	r3 = r1 // r4
4	`sub r3 r3 r2`	r3 = r3 – r2
5	`addn r3 -1`	r3 = r3 + -1
6	`write r3`	print r3
7	`halt`	

Hmmm...!?

Extra! Change only the <u>instruction</u> on line **4** to create the overall output of **<u>56</u>** or **<u>349</u>** or **<u>0</u>** or **<u>6</u>** ... ?

exQuiz.hmmm

solutions...

Try this on the back page first!

Quiz

screen

100 (input)

42 (output)

CPU

central processing unit

r1 100

General-purpose register r1

r2 7

General-purpose register r2

r3 42 43 50

General-purpose register r3

r4 <u>2</u>

General-purpose register r4

RAM

random access memory

100

		Python
0	`read r1`	r1 = 100
1	`setn r2 7`	r2 = 7
2	`mod r4 r1 r2`	r4 = r1 % r2
3	`div r3 r1 r4`	r3 = r1 // r4
4	`sub r3 r3 r2`	r3 = r3 – r2
5	`addn r3 -1`	r3 = r3 + -1
6	`write r3`	print r3
7	`halt`	

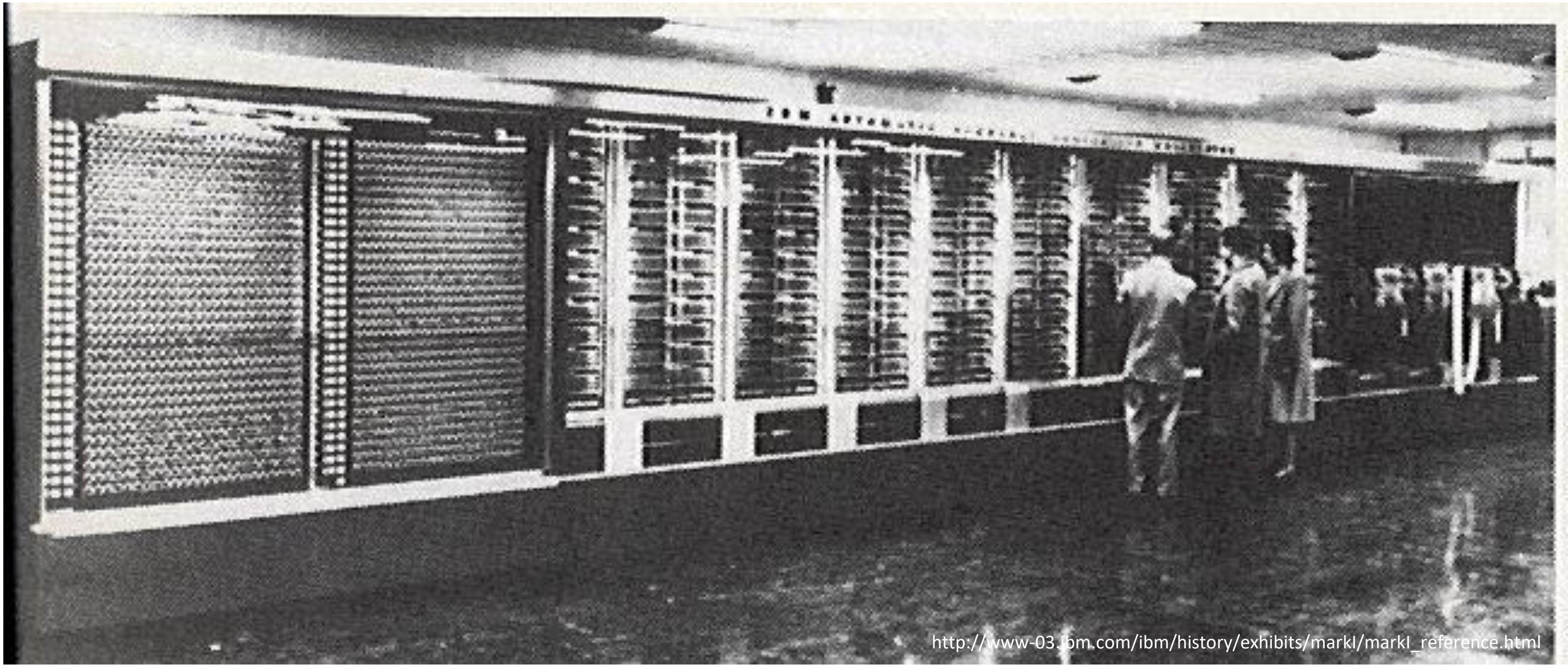
Hmmm...!?

Extra! Change the <u>instruction</u> on line **4** to create the overall output of <u>**56**</u> or <u>**349**</u> or <u>**0**</u> or <u>**6**</u> ... ?

add	mul	mod	div
56	349	0	6

The Mark 1

relay-based computer


http://www-03.ibm.com/ibm/history/exhibits/markI/markI_reference.html

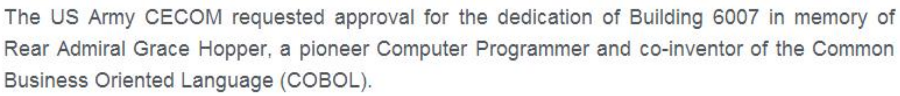
Grace Hopper + Howard Aiken, Harvard ~ 1944

ran at 0.00001 MHz

5 tons
530 miles of wiring
765,299 distinct parts!

Addition: **0.6 seconds**
Multiplication: **5.7 seconds**
Division: **15.3 seconds**

Grace Hopper



The US Army CECOM requested approval for the dedication of Building 6007 in memory of Rear Admiral Grace Hopper, a pioneer Computer Programmer and co-inventor of the Common Business Oriented Language (COBOL).

Building 6007 is named after Grace Hopper.

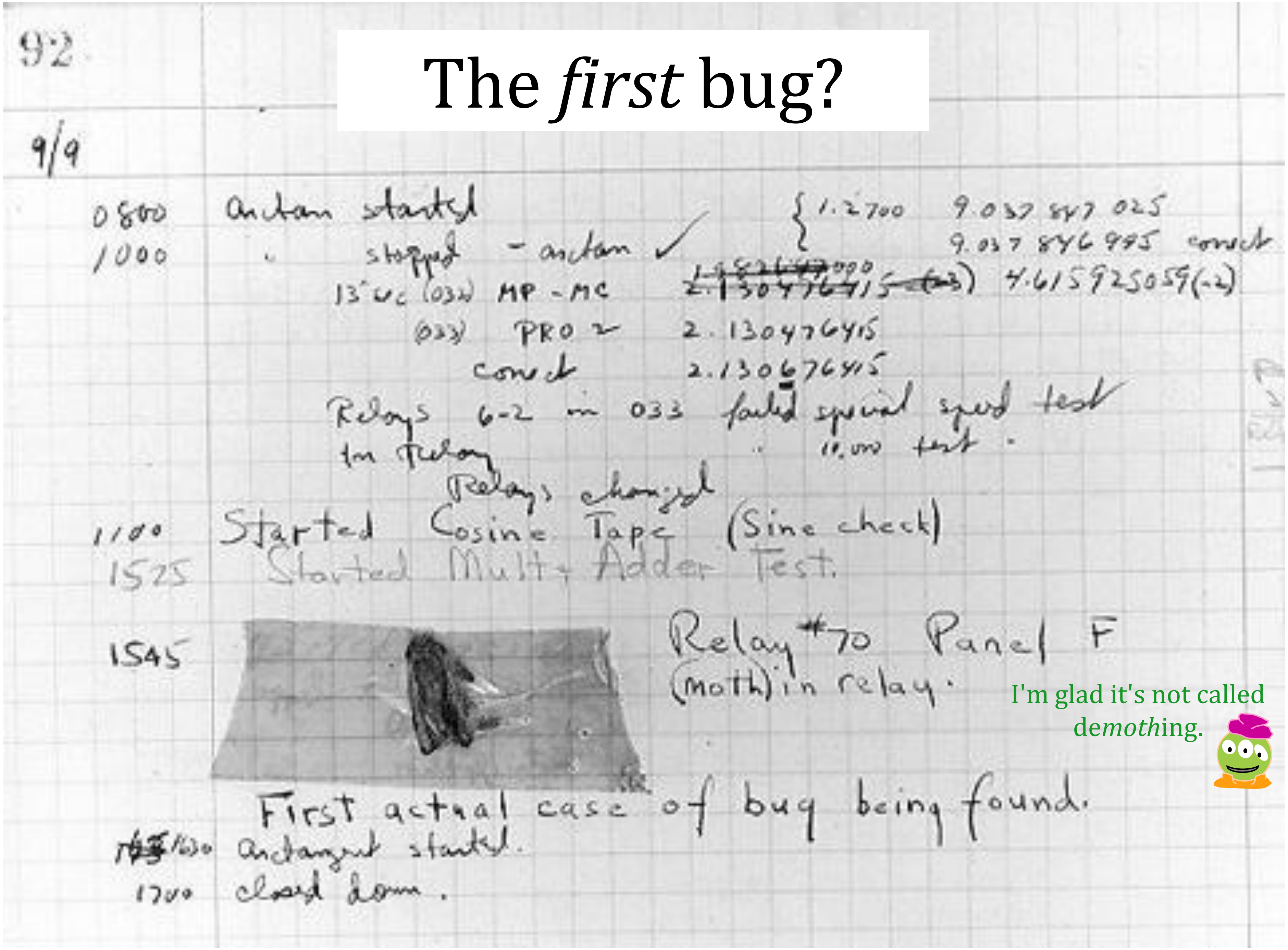
GMH dedications

Grace Murray Hopper ’28 taught math and physics at Vassar for 12 years before joining the Navy reserves in 1943. During the war she learned to **program 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 coined 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 “user-friendly” computer language, COBOL**

> **“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.”**

The *first* bug?



92

9/9

0800 Arctan started {1.2700 9.037 847 025

1000 " stopped - arctan ✓ 9.037 846 995 correct

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3)) 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In Relay " 10,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult+ Adder Test.

1545 Relay #70 Panel F

(moth) in relay.

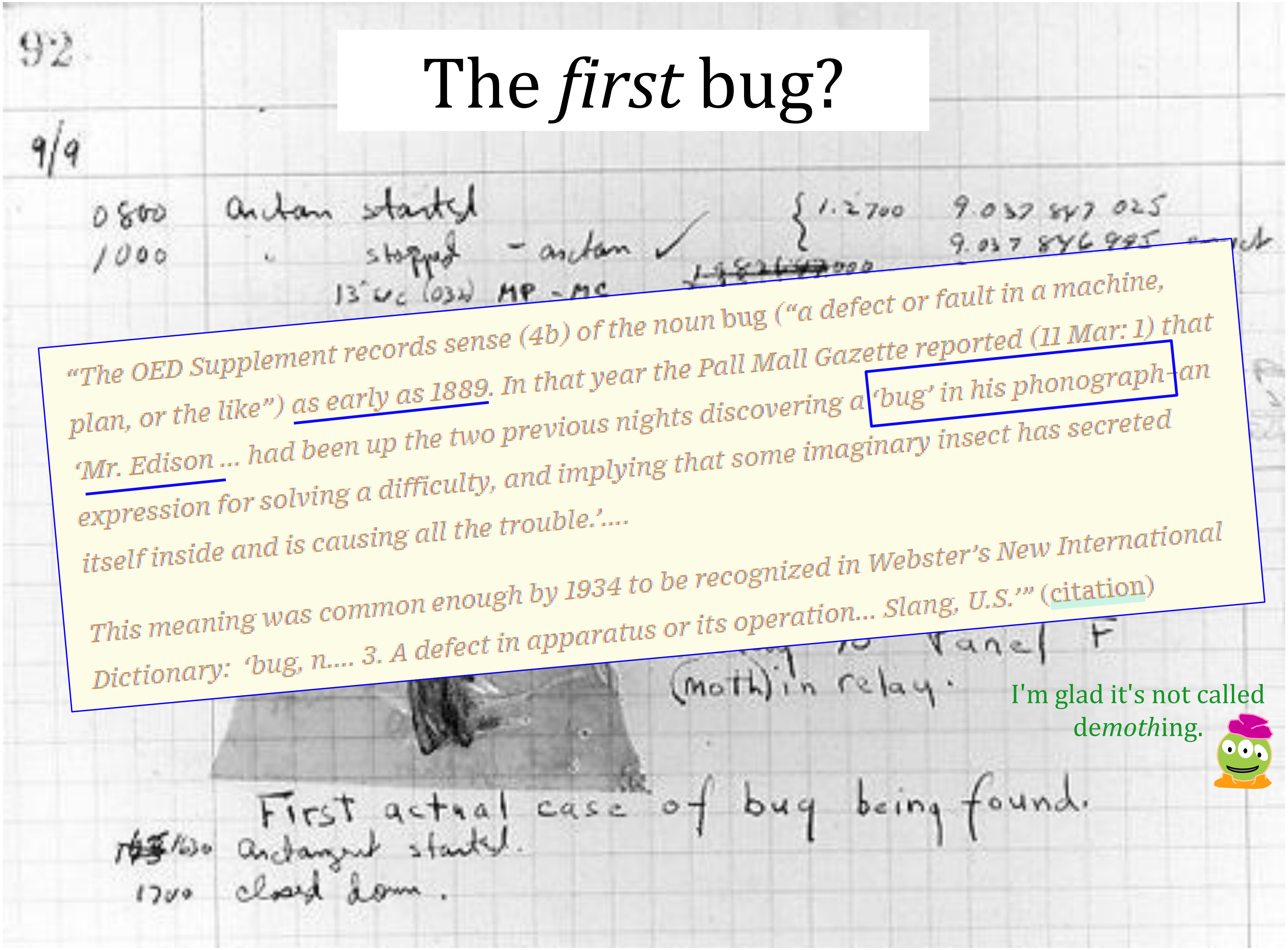
First actual case of bug being found.

~~1600~~ 1630 Arctangent started.

1700 closed down.

I'm glad it's not called de*moth*ing.

The *first* bug?



92

9/9

0800 Anctan started {1.2700 9.037 847 025

1000 " stopped - anctan ✓ 9.037 846 995 correct

13" uc (032) MP-MC ~~1.982647000~~

"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)

Panel F

(moth) in relay.

First actual case of bug being found.

1630 Antangent started.

1700 closed down.

I'm glad it's not called de*moth*ing.

Could you write a Hmmm program that computes

$$x^2 + 3x - 4$$

Hmmm... !

or

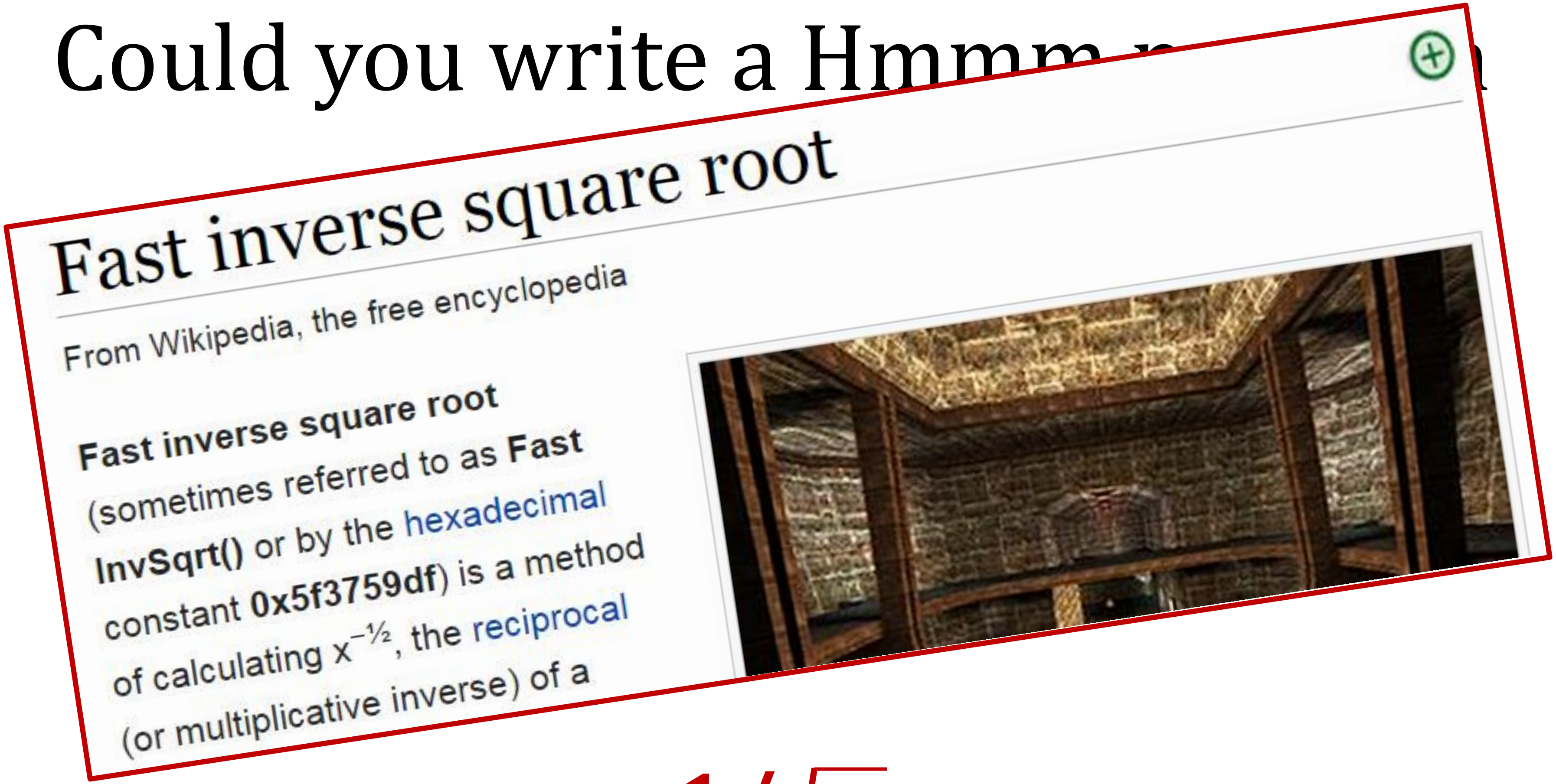
$$1/\sqrt{x}$$

Hmmm... ?

?

when would you ***want*** to?

Could you write a Hmmm



Fast inverse square root

From Wikipedia, the free encyclopedia

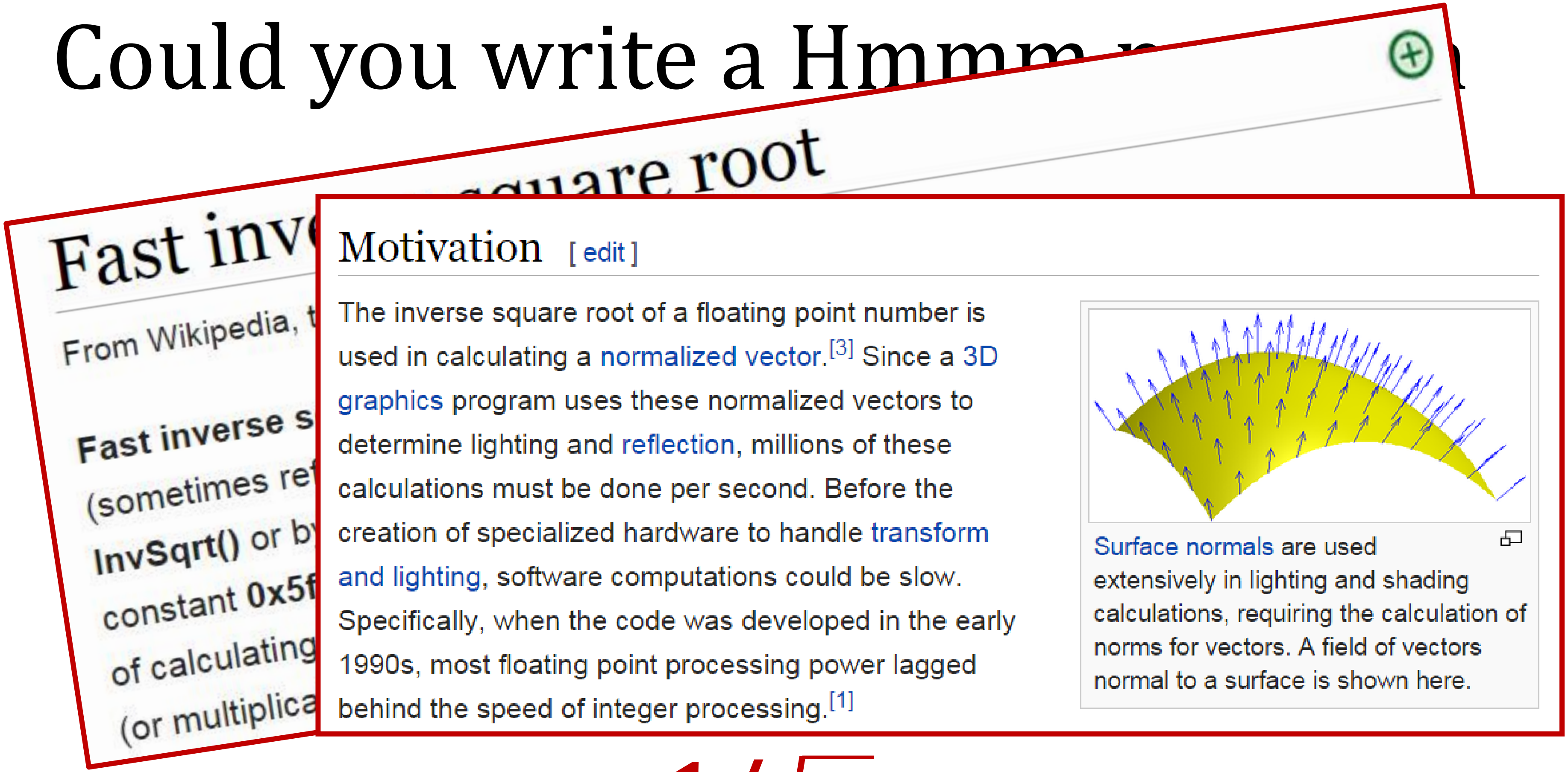
Fast inverse square root (sometimes referred to as **Fast InvSqrt()** or by the hexadecimal constant **0x5f3759df**) is a method of calculating $x^{-1/2}$, the reciprocal (or multiplicative inverse) of a

$$1/\sqrt{x}$$

?

when you'd ***want*** to!

Could you write a Hmmm



Fast inv... square root

From Wikipedia, t

Fast inverse s

(sometimes ref

InvSqrt() or by

constant **0x5f**

of calculating

(or multiplica

Motivation [edit]

The inverse square root of a floating point number is used in calculating a normalized vector.[3] Since a 3D graphics program uses these normalized vectors to determine lighting and reflection, millions of these calculations must be done per second. Before the creation of specialized hardware to handle transform and lighting, software computations could be slow. Specifically, when the code was developed in the early 1990s, most floating point processing power lagged behind the speed of integer processing.[1]

Surface normals are used extensively in lighting and shading calculations, requiring the calculation of norms for vectors. A field of vectors normal to a surface is shown here.

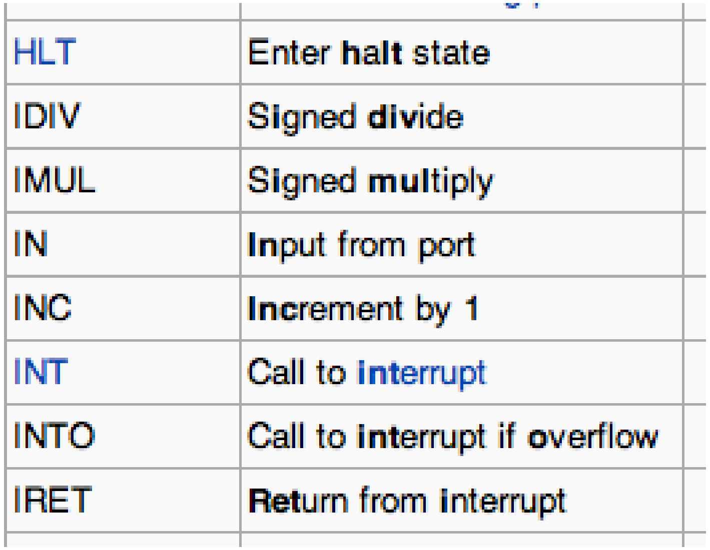
$$1/\sqrt{x}$$

?

when you'd ***want*** to!

Real Assembly Languages

Hmmm is a subset common to ***all*** real assembly languages.

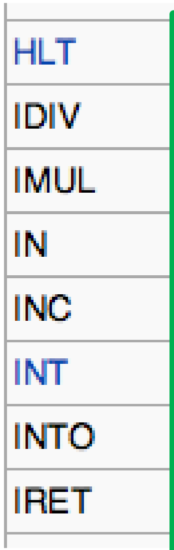


Instruction	Description
HLT	Enter halt state
IDIV	Signed divide
IMUL	Signed multiply
IN	Input from port
INC	Increment by 1
INT	Call to interrupt
INTO	Call to interrupt if overflow
IRET	Return from interrupt

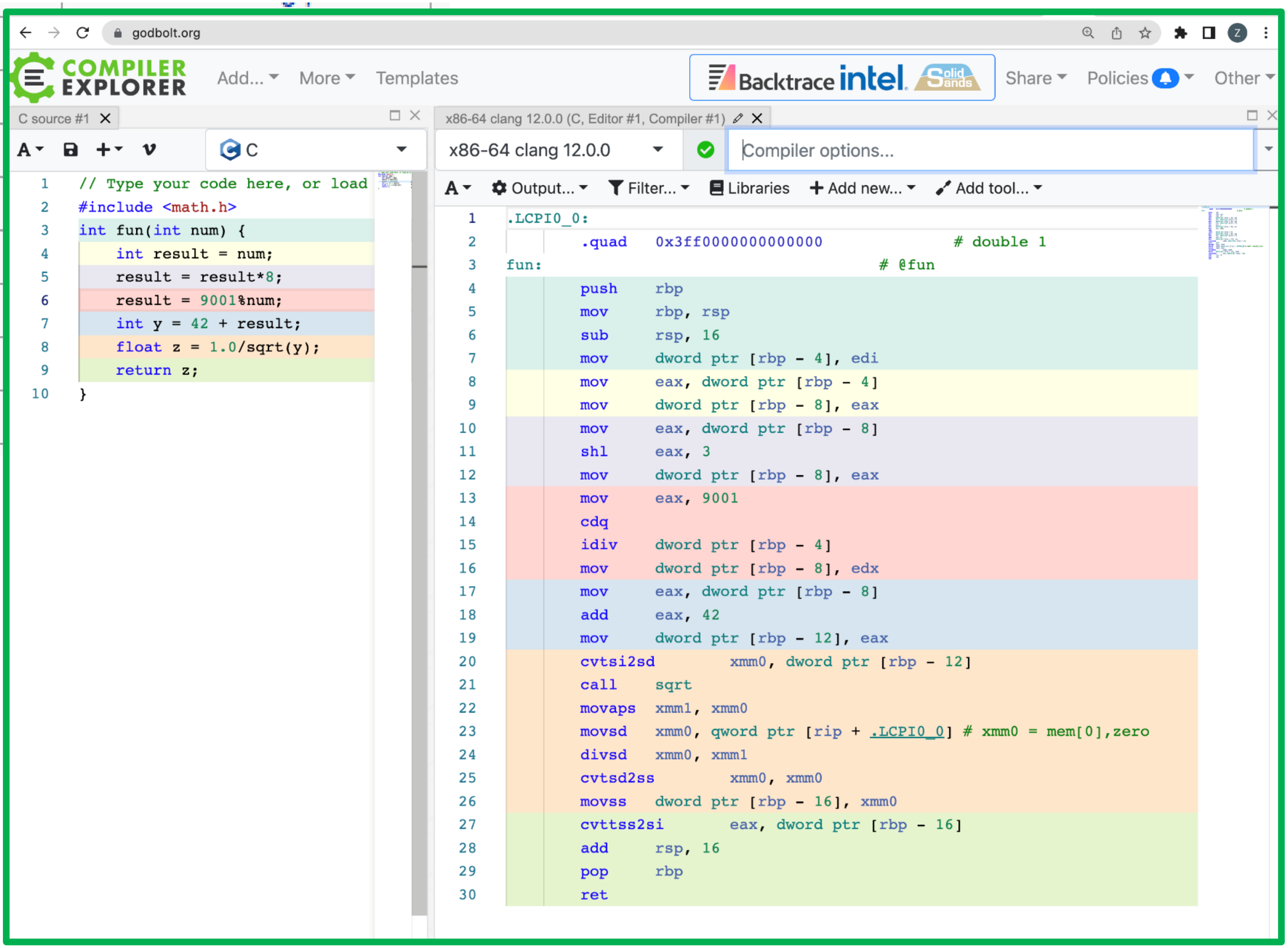
A few of the many basic processor instructions (Intel)

Real Assembly Languages

Hmmm is a subset common to ***all*** real assembly languages.



Instruction	Description
HLT	
IDIV	
IMUL	
IN	
INC	
INT	
INTO	
IRET	



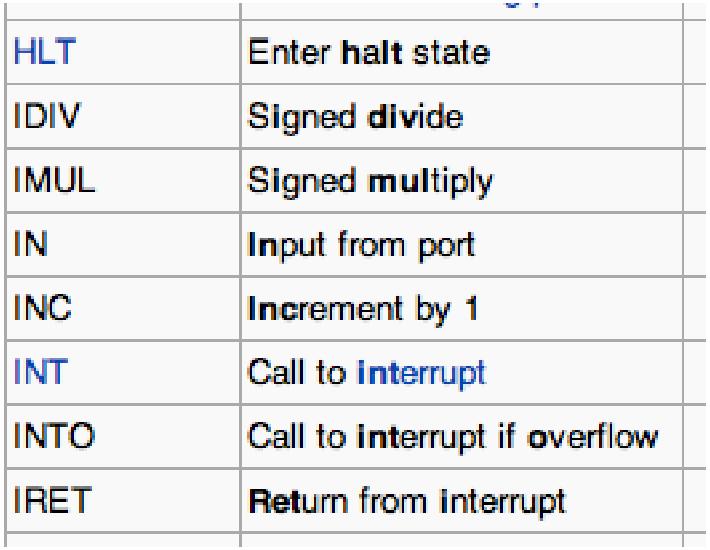
```c
// Type your code here, or load
#include <math.h>
int fun(int num) {
    int result = num;
    result = result*8;
    result = 9001%num;
    int y = 42 + result;
    float z = 1.0/sqrt(y);
    return z;
}
```

```
.LCPI0_0:
        .quad   0x3ff0000000000000              # double 1
fun:                                    # @fun
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], edi
        mov     eax, dword ptr [rbp - 4]
        mov     dword ptr [rbp - 8], eax
        mov     eax, dword ptr [rbp - 8]
        shl     eax, 3
        mov     dword ptr [rbp - 8], eax
        mov     eax, 9001
        cdq
        idiv    dword ptr [rbp - 4]
        mov     dword ptr [rbp - 8], edx
        mov     eax, dword ptr [rbp - 8]
        add     eax, 42
        mov     dword ptr [rbp - 12], eax
        cvtsi2sd        xmm0, dword ptr [rbp - 12]
        call    sqrt
        movaps  xmm1, xmm0
        movsd   xmm0, qword ptr [rip + .LCPI0_0] # xmm0 = mem[0],zero
        divsd   xmm0, xmm1
        cvtsd2ss        xmm0, xmm0
        movss   dword ptr [rbp - 16], xmm0
        cvttss2si       eax, dword ptr [rbp - 16]
        add     rsp, 16
        pop     rbp
        ret
```

```c
#include <math.h>
int fun(int num) {
int result = num;
result = result*8;
result = 9001%num;
int y = 42 + result;
float z = 1.0/sqrt(y);
return z;
}
```

Real Assembly Languages

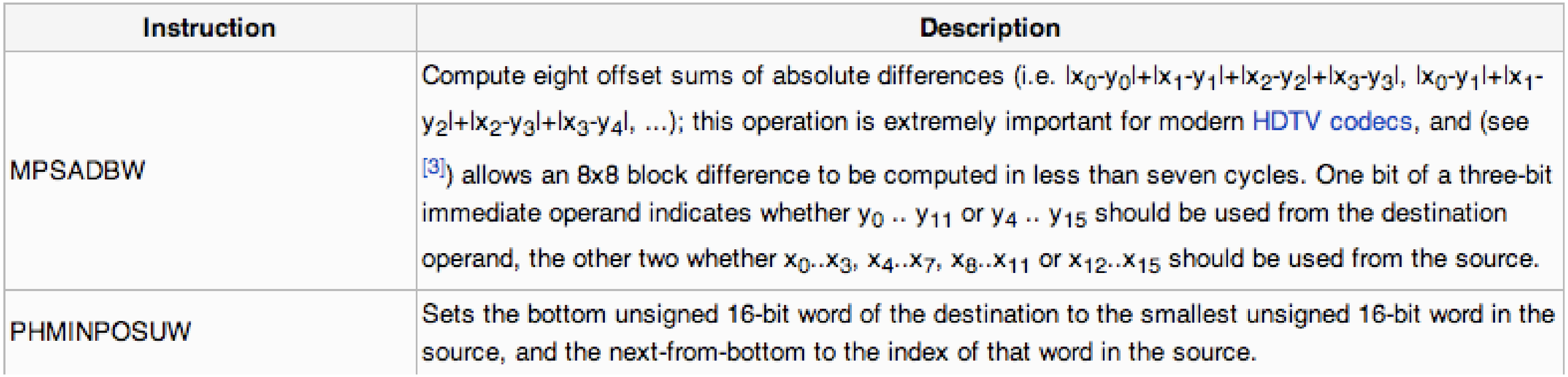
Hmmm is a subset common to ***all*** real assembly languages.



Instruction	Description
HLT	Enter halt state
IDIV	Signed divide
IMUL	Signed multiply
IN	Input from port
INC	Increment by 1
INT	Call to interrupt
INTO	Call to interrupt if overflow
IRET	Return from interrupt

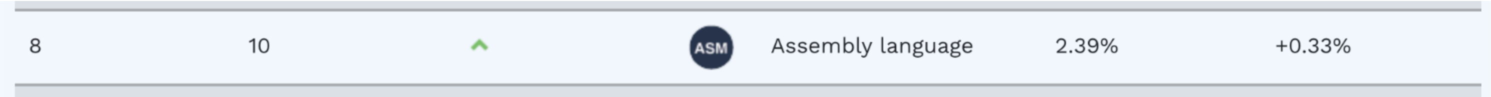
A few of the many basic processor instructions (Intel)

two ***more recent*** Intel instructions (SSE4 subset)



Instruction	Description
MPSADBW	Compute eight offset sums of absolute differences (i.e. $\lvert x_0-y_0\rvert+\lvert x_1-y_1\rvert+\lvert x_2-y_2\rvert+\lvert x_3-y_3\rvert$, $\lvert x_0-y_1\rvert+\lvert x_1-y_2\rvert+\lvert x_2-y_3\rvert+\lvert x_3-y_4\rvert$, ...); this operation is extremely important for modern HDTV codecs, and (see [3]) allows an 8x8 block difference to be computed in less than seven cycles. One bit of a three-bit immediate operand indicates whether $y_0 .. y_{11}$ or $y_4 .. y_{15}$ should be used from the destination operand, the other two whether $x_0..x_3$, $x_4..x_7$, $x_8..x_{11}$ or $x_{12}..x_{15}$ should be used from the source.
PHMINPOSUW	Sets the bottom unsigned 16-bit word of the destination to the smallest unsigned 16-bit word in the source, and the next-from-bottom to the index of that word in the source.

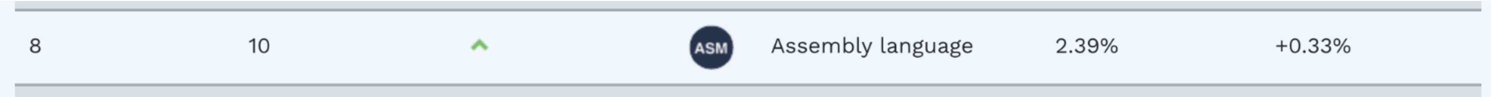
Who writes all the assembly language that gets executed?



8	10	▲	ASM	Assembly language	2.39%	+0.33%

clearly, it's not people!

Who writes all the assembly language that gets executed?

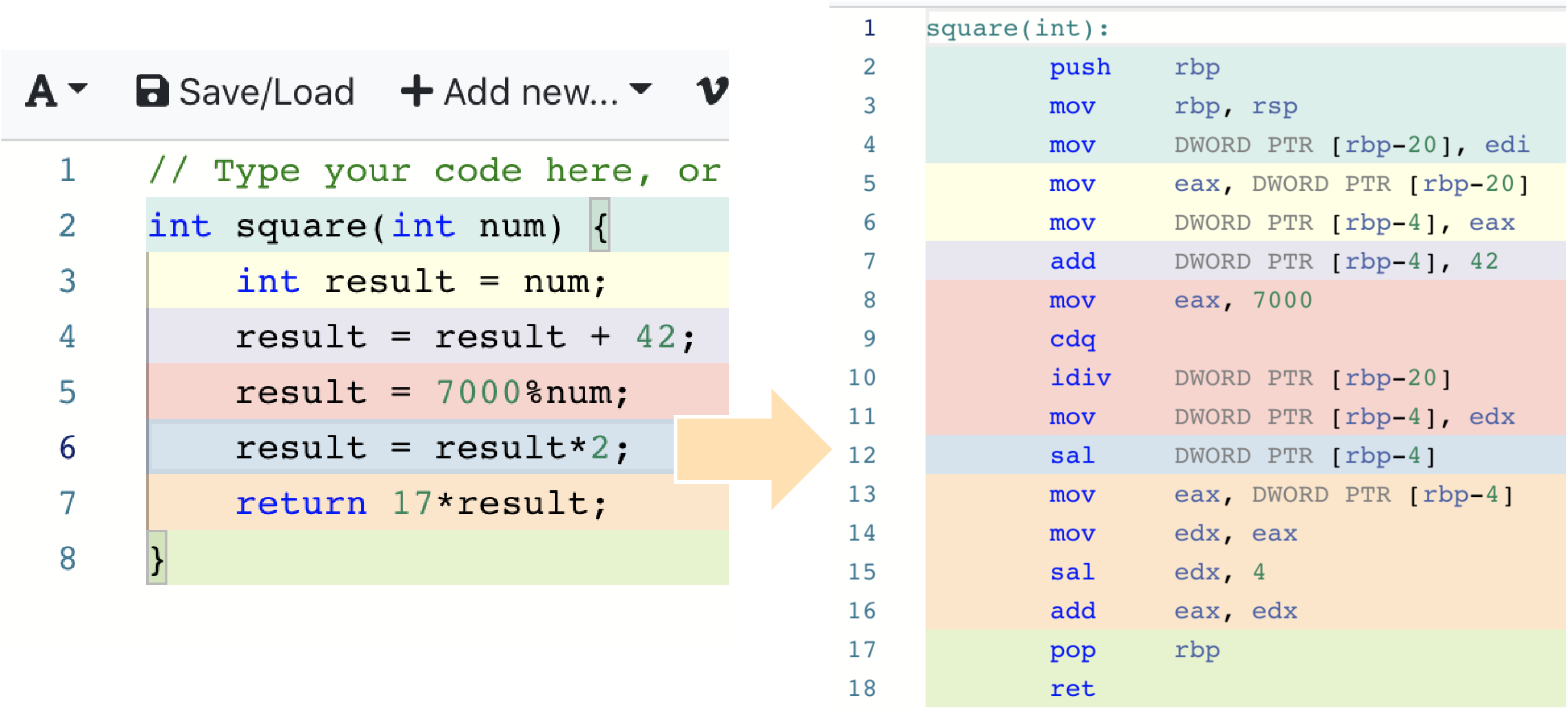


8	10	^	ASM	Assembly language	2.39%	+0.33%

clearly, it's not people!

other programs!

Who writes all of the assembly language that gets executed?



```
// Type your code here, or
int square(int num) {
    int result = num;
    result = result + 42;
    result = 7000%num;
    result = result*2;
    return 17*result;
}
```

```
square(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        mov     DWORD PTR [rbp-4], eax
        add     DWORD PTR [rbp-4], 42
        mov     eax, 7000
        cdq
        idiv    DWORD PTR [rbp-20]
        mov     DWORD PTR [rbp-4], edx
        sal     DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-4]
        mov     edx, eax
        sal     edx, 4
        add     eax, edx
        pop     rbp
        ret
```

other programs!

https://godbolt.org/

Could you write a ***Python*** program

that writes a Hmmm program

that computes

$$\mathbf{x^2 + 3x - 4}$$

or

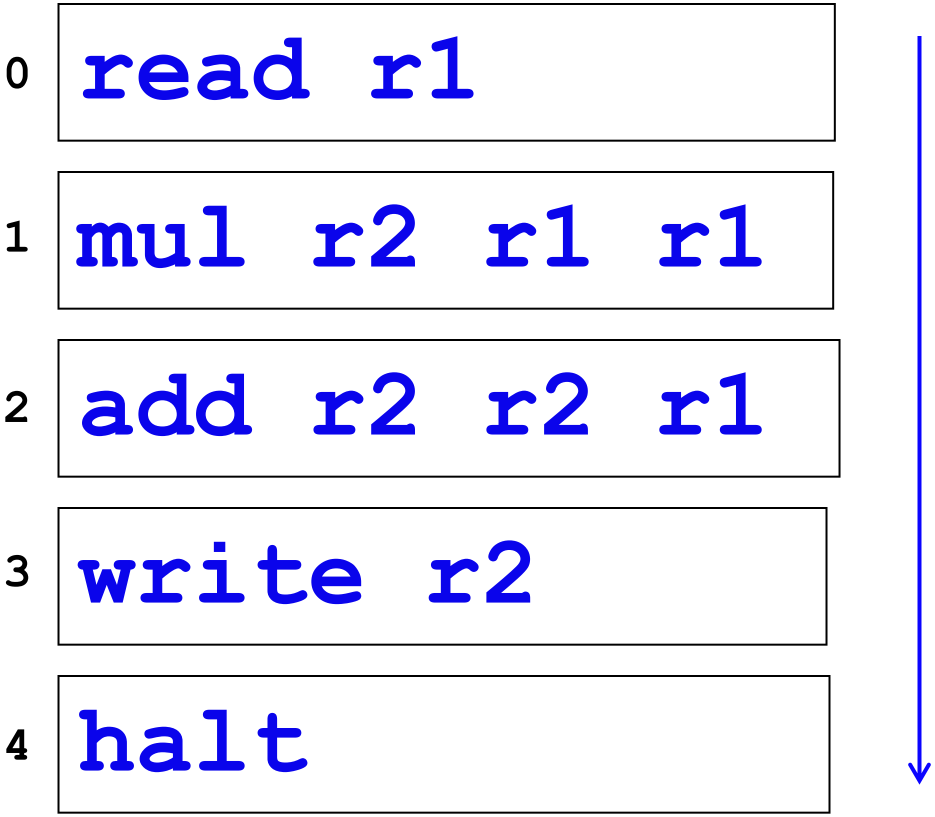
$$\mathbf{1/\sqrt{x}}$$

?

Yes – and you already have!

Is this all we need?

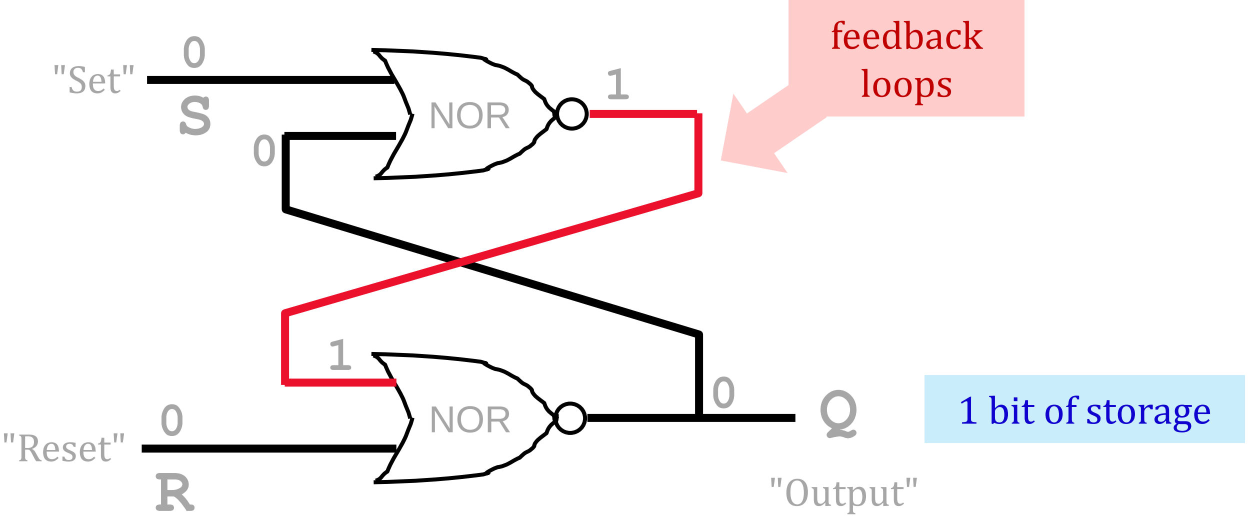
What's missing here?



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

Why ***couldn't*** we implement Python using only our Hmmm assembly language up to this point?

For systems, innovation is adding an edge to *create a cycle*, not just an additional node.



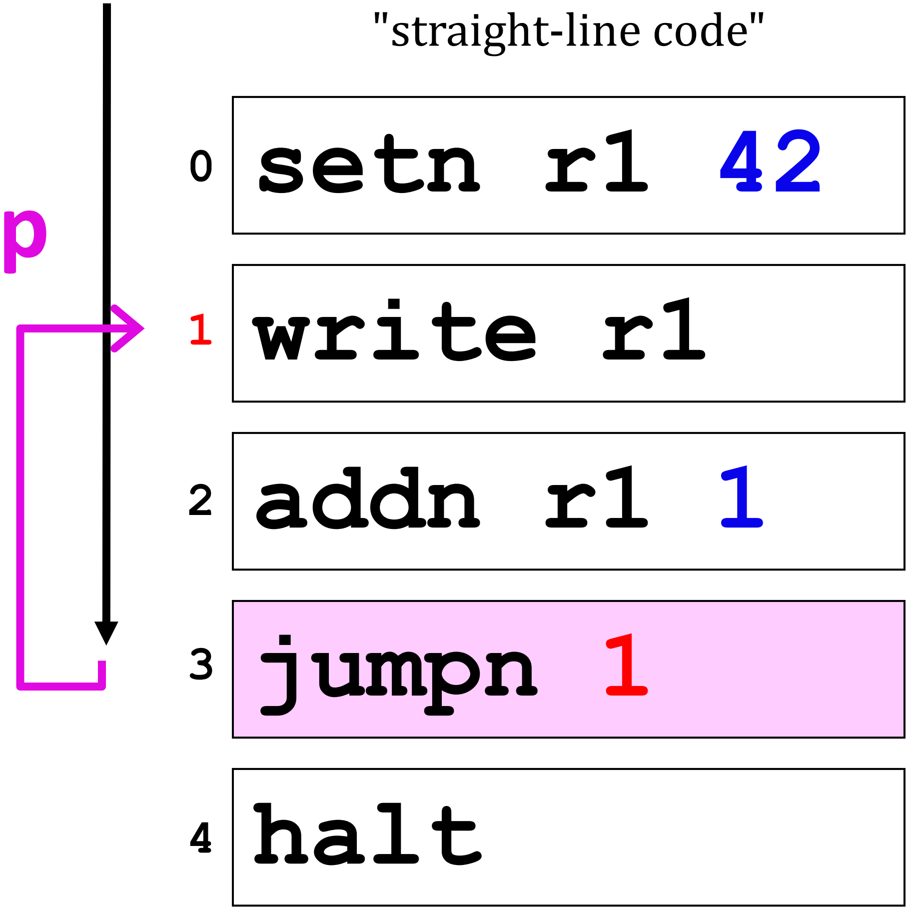
feedback loops

1 bit of storage

Loops and **ifs**

We ***couldn't*** implement Python using Hmmm so far...

It's too linear!



"straight-line code"

```
0  setn r1 42
1  write r1
2  addn r1 1
3  jumpn 1
4  halt
```

exJump.hmmm

CPU

central processing unit

r1

General-purpose register r1

r2

not used in this program...

General-purpose register r2

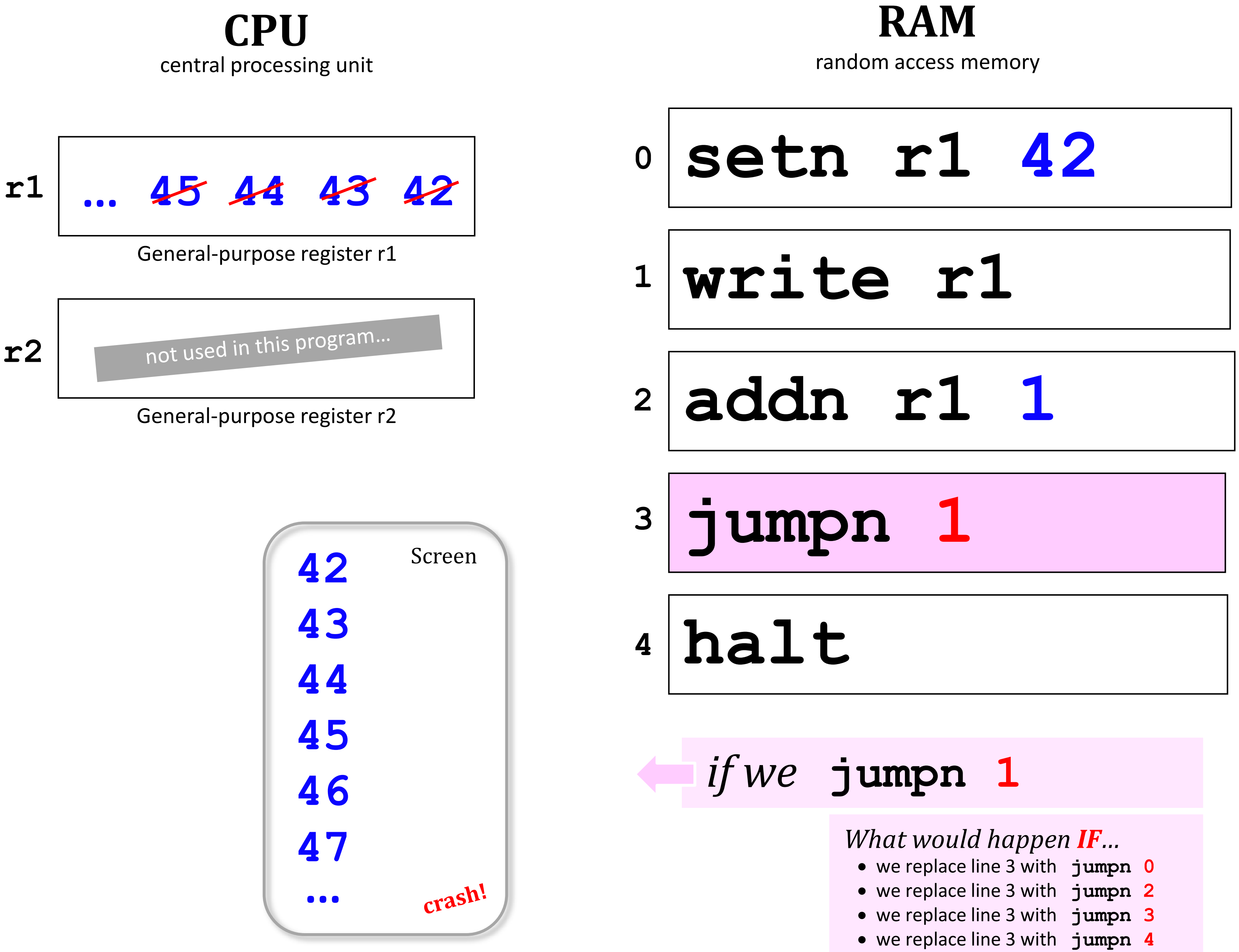
Screen

RAM

random access memory

```
0  setn r1 42
1  write r1
2  addn r1 1
3  jumpn 1
4  halt
```

jumpn!



CPU

central processing unit

r1 … ~~45~~ ~~44~~ ~~43~~ ~~42~~

General-purpose register r1

r2 not used in this program…

General-purpose register r2

Screen

```
42
43
44
45
46
47
…
```

crash!

RAM

random access memory

```
0 setn r1 42
1 write r1
2 addn r1 1
3 jumpn 1
4 halt
```

if we `jumpn 1`

What would happen ***IF***…

- we replace line 3 with `jumpn 0`
- we replace line 3 with `jumpn 2`
- we replace line 3 with `jumpn 3`
- we replace line 3 with `jumpn 4`

CPU

central processing unit

r1

General-purpose register r1

r2

not used in this program...

General-purpose register r2

RAM

random access memory

```
0  setn r1 42
1  write r1
2  addn r1 1
3  jumpn 1
4  halt
```

Screen

42
43
44
45
46
...

crash!

*What would happen **IF**...*

- we replace line 3 with `jumpn 0`
- we replace line 3 with `jumpn 2`
- we replace line 3 with `jumpn 3`
- we replace line 3 with `jumpn 4`

Screen

42

no crash

Screen

42

no crash

Screen

42

crash!

Screen

42
42
42
42
42
...

no crash

CPU
central processing unit

r1

General-purpose register r1

r2

jumpn answers

jumpn 4

Screen
42

no crash

jumpn 3

Screen
42

no crash

jumpn 2

Screen
42

crash!

jumpn 0

Screen
42
42
42
42
42
...

no crash

RAM
random access memory

```
0  setn r1 42
1  write r1
2  addn r1 1
3  jumpn 1
4  halt
```

Screen
42
43
44
45
46
...

crash!

*What would happen **IF**...*

- we replace line 3 with `jumpn 0`
- we replace line 3 with `jumpn 2`
- we replace line 3 with `jumpn 3`
- we replace line 3 with `jumpn 4`

Jumps in Hmmm

Conditional jumps

`jeqzn r1 42`	IF **r1 == 0** THEN jump to line number **42**
`jgtzn r1 42`	IF **r1 > 0** THEN jump to line number **42**
`jltzn r1 42`	IF **r1 < 0** THEN jump to line number **42**
`jnezn r1 42`	IF **r1 != 0** THEN jump to line number **42**

This is making me jumpy!

Unconditional jump

`jumpn 42`	Jump to program line # **42**

Jumps in Hmmm

Conditional jumps

`jeqzn` if equal to zero… THEN jump to line number **42**

`jgtzn` if greater than zero … EN jump to line number **42**

`jltzn` if less than zero… THEN jump to line number **42**

`jnezn` if not equal to zero … HEN jump to line number **42**

Mnemonics!

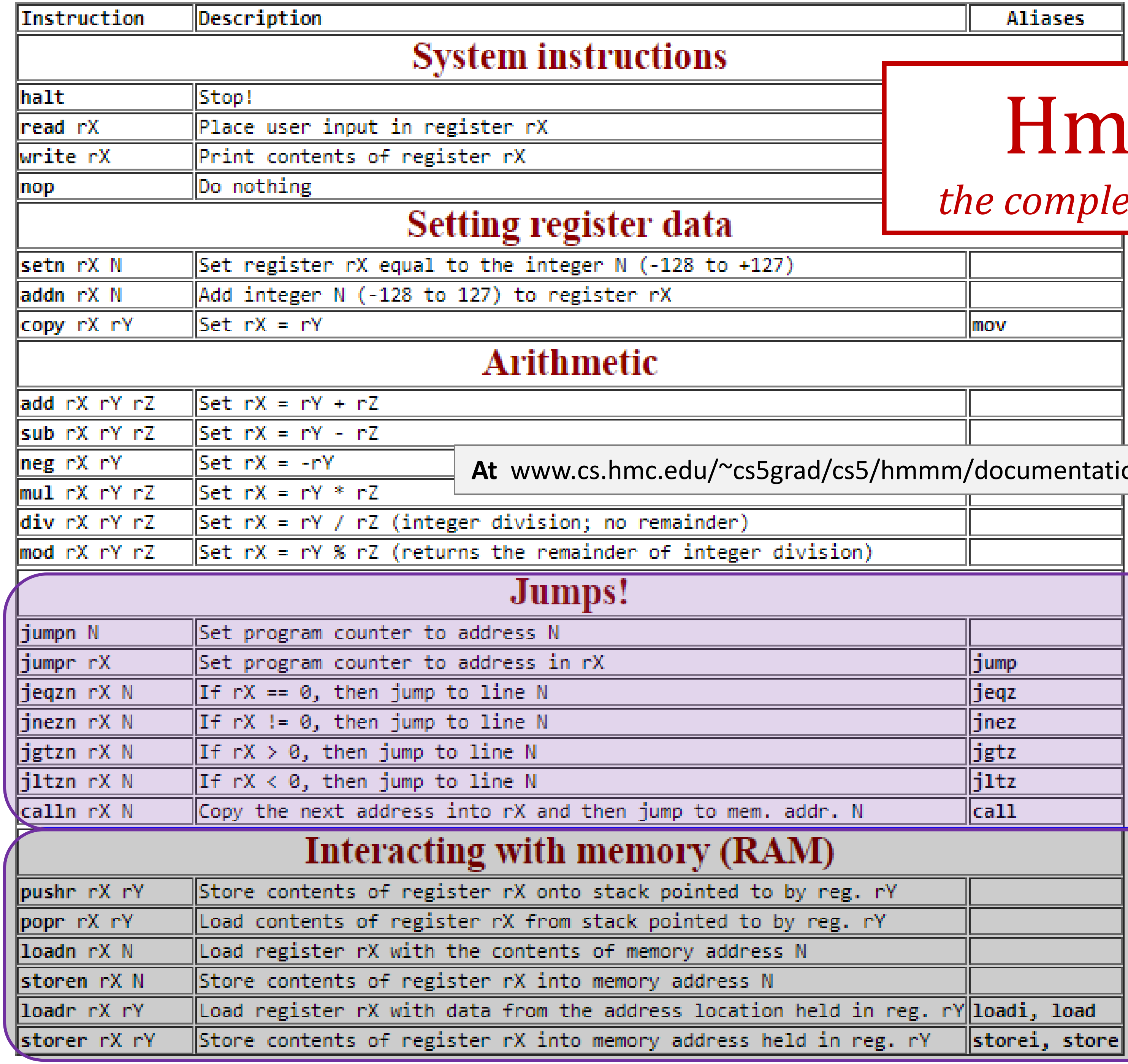
This is making me jumpy!

Unconditional jump

`jumpn 42` Jump to program line # **42**

Hmmm

the complete reference



Instruction	Description	Aliases
	System instructions	
halt	Stop!	
read rX	Place user input in register rX	
write rX	Print contents of register rX	
nop	Do nothing	
	Setting register data	
setn rX N	Set register rX equal to the integer N (-128 to +127)	
addn rX N	Add integer N (-128 to 127) to register rX	
copy rX rY	Set rX = rY	mov
	Arithmetic	
add rX rY rZ	Set rX = rY + rZ	
sub rX rY rZ	Set rX = rY - rZ	
neg rX rY	Set rX = -rY	
mul rX rY rZ	Set rX = rY * rZ	
div rX rY rZ	Set rX = rY / rZ (integer division; no remainder)	
mod rX rY rZ	Set rX = rY % rZ (returns the remainder of integer division)	
	Jumps!	
jumpn N	Set program counter to address N	
jumpr rX	Set program counter to address in rX	jump
jeqzn rX N	If rX == 0, then jump to line N	jeqz
jnezn rX N	If rX != 0, then jump to line N	jnez
jgtzn rX N	If rX > 0, then jump to line N	jgtz
jltzn rX N	If rX < 0, then jump to line N	jltz
calln rX N	Copy the next address into rX and then jump to mem. addr. N	call
	Interacting with memory (RAM)	
pushr rX rY	Store contents of register rX onto stack pointed to by reg. rY	
popr rX rY	Load contents of register rX from stack pointed to by reg. rY	
loadn rX N	Load register rX with the contents of memory address N	
storen rX N	Store contents of register rX into memory address N	
loadr rX rY	Load register rX with data from the address location held in reg. rY	loadi, load
storer rX rY	Store contents of register rX into memory address held in reg. rY	storei, store

At www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html

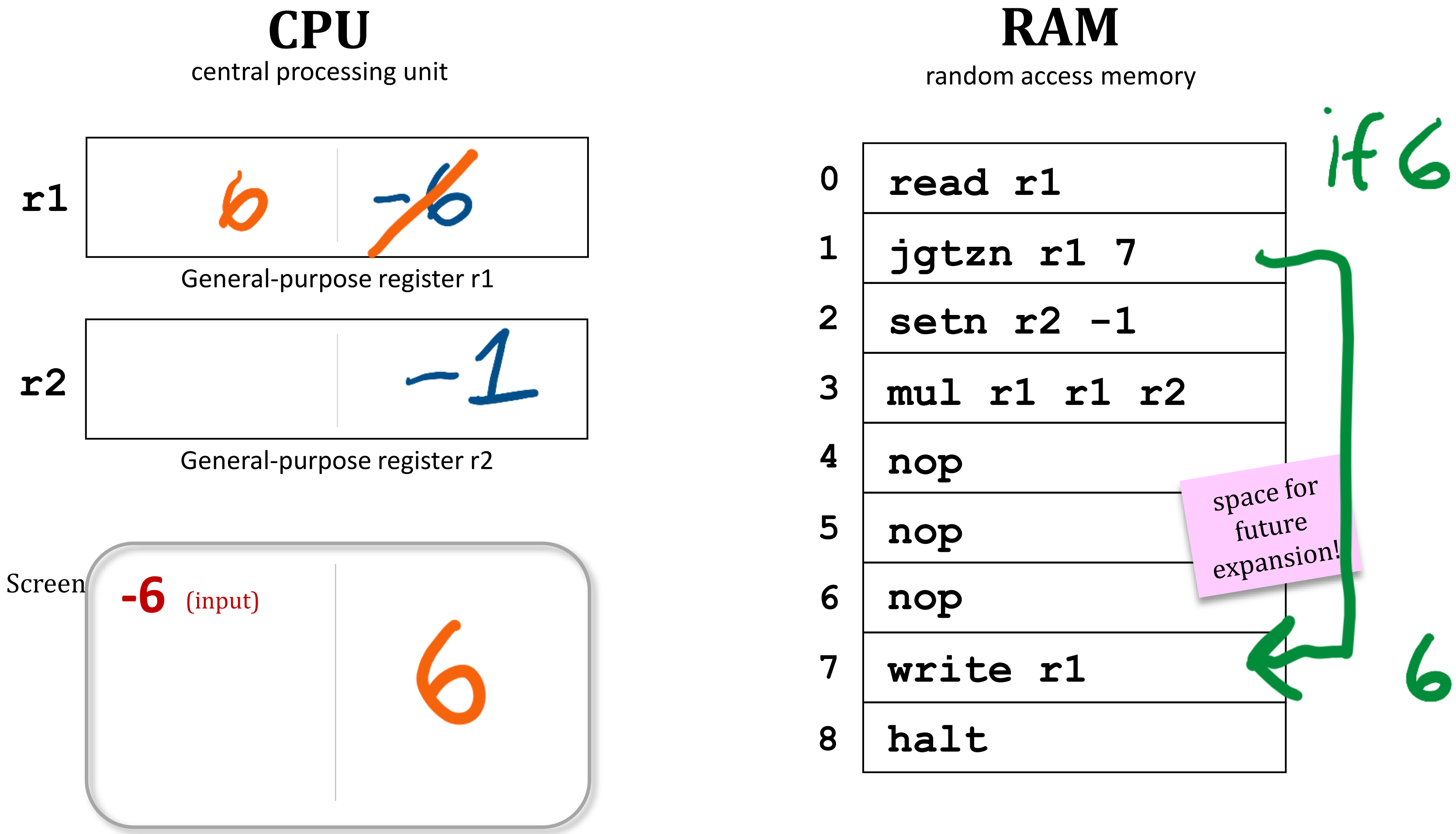
Jumps!

off-processor access...
[Thursday]

jgtzn

Gesundheit!

What Python f'n is this?



CPU

central processing unit

r1 6 ~~-6~~

General-purpose register r1

r2 -1

General-purpose register r2

Screen

-6 (input)

6

RAM

random access memory

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

space for future expansion!

if 6

6

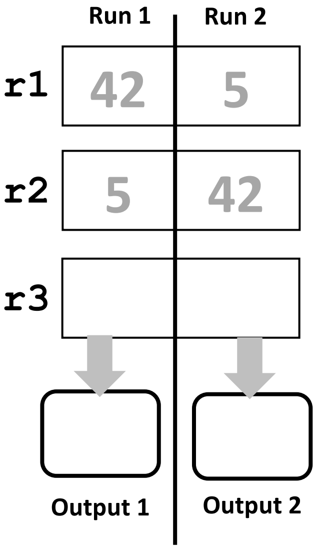
With an input of -6, what does this code write out?

Try it!

I think this language has
injured my *craniuhmmm*!

1 Follow this Hmmm program.
First run: use **r1 = 42** and **r2 = 5**.
Next run: use **r1 = 5** and **r2 = 42**.

Registers - CPU



Memory - RAM

0	`read r1`
1	`read r2`
2	`sub r3 r1 r2`
3	`nop`
4	`jgtzn r3 7`
5	`write r1`
6	`jumpn 8`
7	`write r2`
8	`halt`

(1) What **common function** does this compute?
Hint: try the inputs in both orders...

(2) ***Extra!*** How could you change only line 3 so that, if inputs **r1** and **r2** are ***equal***, the program will ask for new inputs?

2 Write an assembly-language program that reads a positive integer into **r1**. The program should compute the ***factorial*** of the input in **r2**. Once it's computed, it should write out that factorial. Two lines are provided:

Registers - CPU

r1 input 5
r2 result – so far 1
r3
not needed; OK to use

Memory - RAM

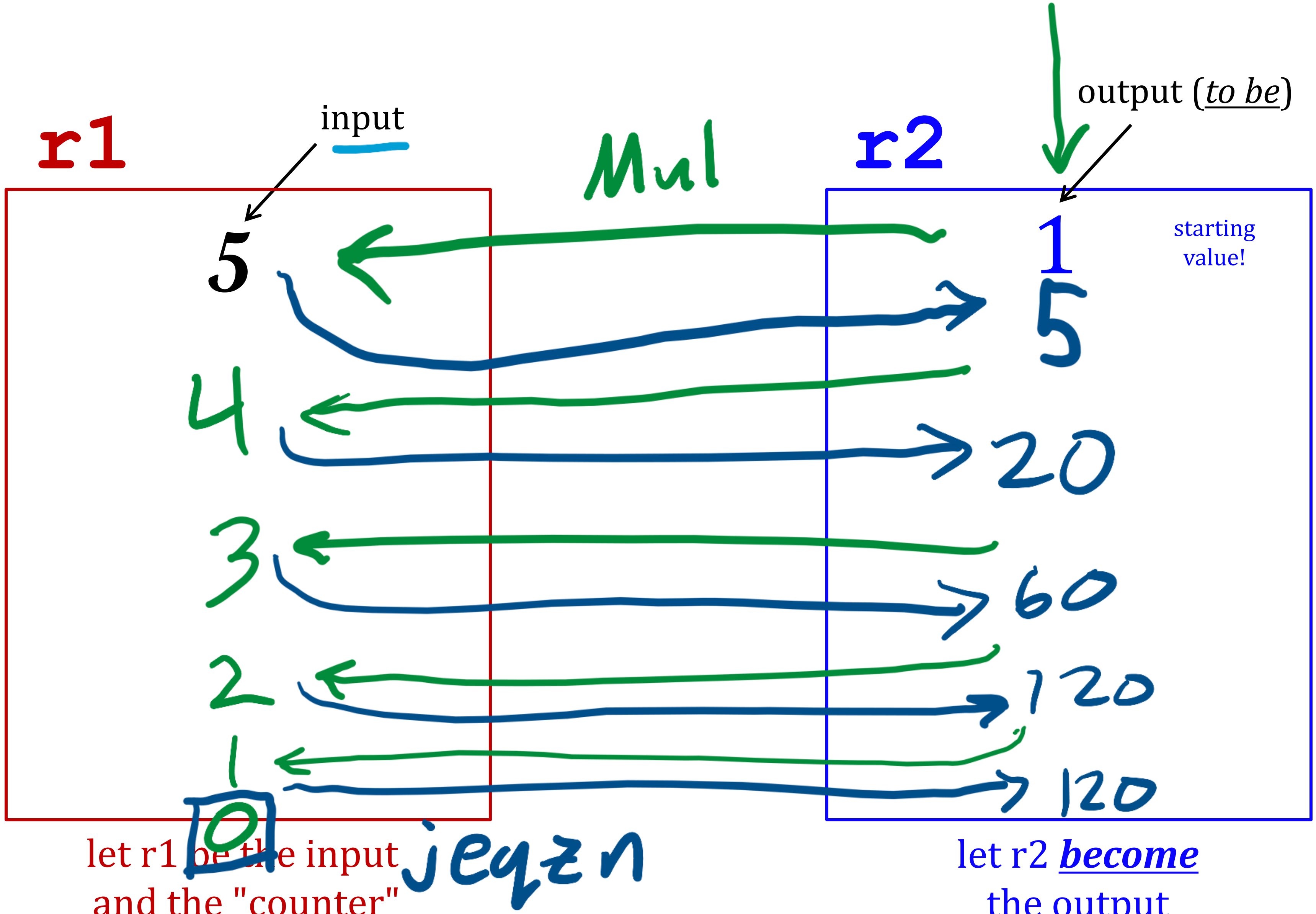
0	`read r1`
1	`setn r2 1`
2	
3	
4	
5	
6	
7	
8	`write r2`
9	`halt`

Hint: On line 2, could you write a test that checks if the factorial is finished; if it's not, compute one piece and then jump back!

Extra! How few lines can you use here? (Fill the rest with `nop`s...)

factorial: the *plan* ...

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



input

output (*to be*)

starting value!

let r1 be the input and the "counter"

let r2 ***become*** the output

Try it!

I think this language has
injured my *craniuhmmm*!

1 Follow this Hmmm program.
First run: use **r1 = 42** and **r2 = 5**.
Next run: use **r1 = 5** and **r2 = 42**.

Registers - CPU

	Run 1	Run 2
r1	42	5
r2	5	42
r3		

Output 1 Output 2

Memory - RAM

0	`read r1`
1	`read r2`
2	`sub r3 r1 r2`
3	`nop`
4	`jgtzn r3 7`
5	`write r1`
6	`jumpn 8`
7	`write r2`
8	`halt`

(1) What **common function** does this compute?
Hint: try the inputs in both orders...

(2) ***Extra!*** How could you change only line 3 so that, if inputs **r1** and **r2** are ***equal***, the program will ask for new inputs?

2 Write an assembly-language program that reads a positive integer into **r1**. The program should compute the ***factorial*** of the input in **r2**. Once it's computed, it should write out that factorial. Two lines are provided:

Registers - CPU

r1	input 5
r2	result – so far 1
r3	not needed; OK to use

Memory - RAM

0	`read r1`
1	`setn r2 1`
2	
3	
4	
5	
6	
7	
8	`write r2`
9	`halt`

Hint: On line 2, could you write a test that checks if the factorial is finished; if it's not, compute one piece and then jump back!

Extra! How few lines can you use here? (Fill the rest with **nop**s...)

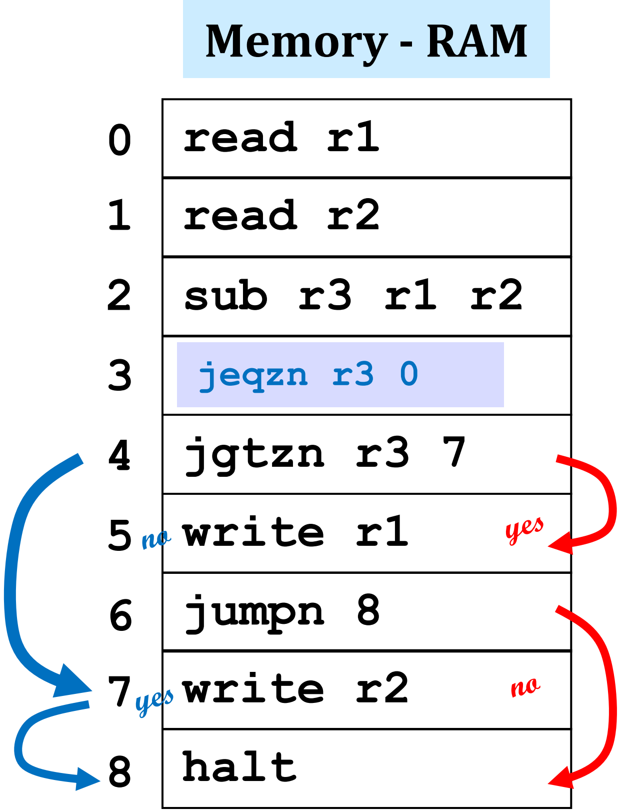
1 Follow this assembly-language program from top to bottom.
First use **r1 = 42** and **r2 = 5**, then swap them on the next run:

Run #1

r1 = *42*
r2 = *5*
r3 = *37*

5
output

Memory - RAM



```
0  read r1
1  read r2
2  sub r3 r1 r2
3  jeqzn r3 0
4  jgtzn r3 7
5  write r1
6  jumpn 8
7  write r2
8  halt
```

Run #2

r1 = *5*
r2 = *42*
r3 = *-37*

5
output

(1) What function does this program compute in general?

(2) ***Extra!*** How could you change <u>only line 3</u> so that, if the original two inputs were *equal*, the program asked for new inputs?

a factorial solution

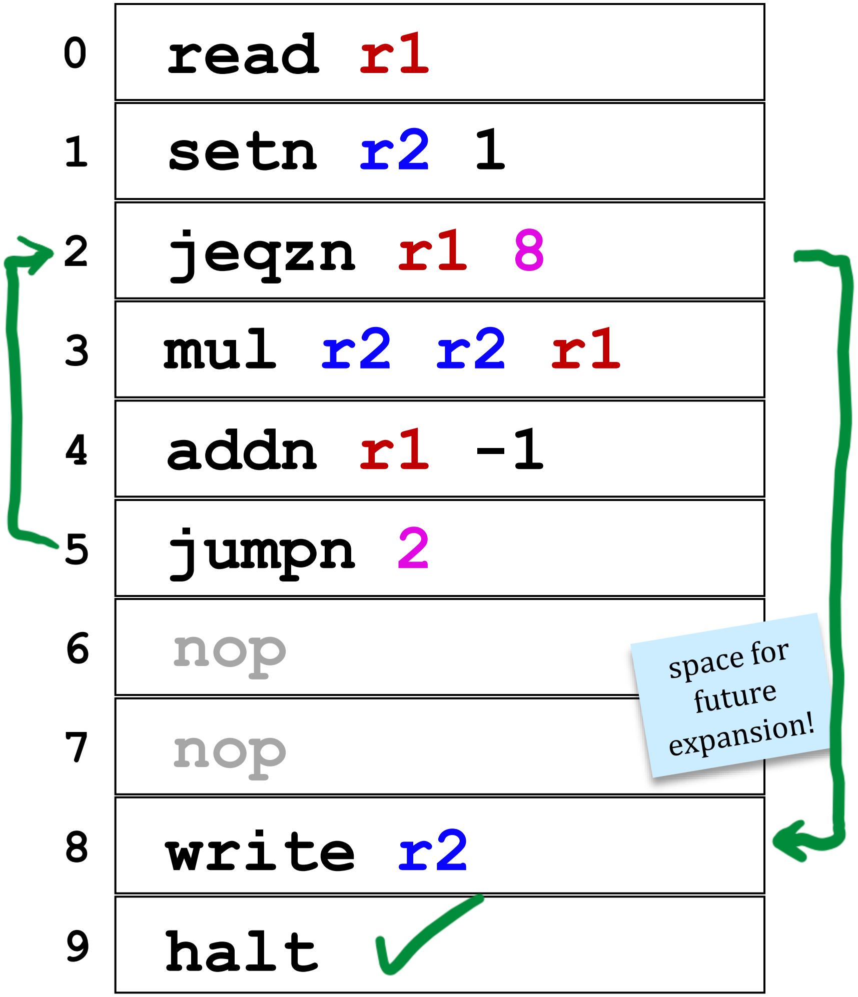
Registers - CPU

r1 input 0 ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~

r2 result - so far 120 ~~120~~ ~~60~~ ~~20~~ ~~5~~ ~~1~~

r3 not needed, but OK to use! 120

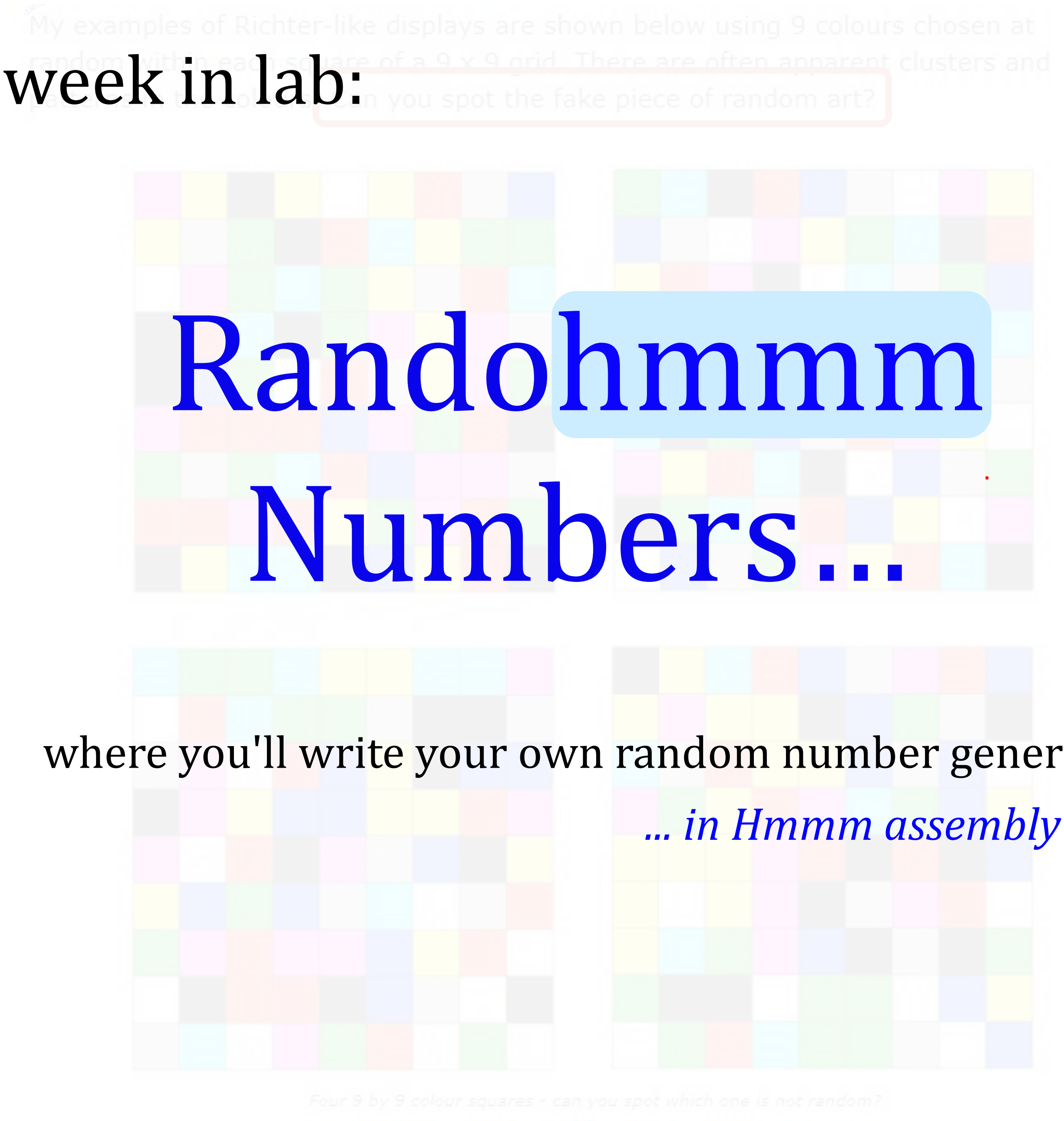
Memory - RAM



0	`read r1`
1	`setn r2 1`
2	`jeqzn r1 8`
3	`mul r2 r2 r1`
4	`addn r1 -1`
5	`jumpn 2`
6	`nop`
7	`nop`
8	`write r2`
9	`halt` ✓

space for future expansion!

This week in lab:



My examples of Richter-like displays are shown below using 9 colours chosen at random within each square of a 9 x 9 grid. There are often apparent clusters and patterns in the colours. Can you spot the fake piece of random art?

Four 9 by 9 colour squares - can you spot which one is not random?

Randohmmm Numbers...

where you'll write your own random number generator...

... in Hmmm assembly language

https://understandinguncertainty.org/node/1066

My examples of Richter-like displays are shown below using 9 colours chosen at random within each square of a 9 x 9 grid. There are often apparent clusters and patterns in the colours. Can you spot the fake piece of random art?

Four 9 by 9 colour squares - can you spot which one is not random?

Enjoy the Randomly-generating lab!

Which one is ***NOT*** random... ?

https://understandinguncertainty.org/node/1066

CS ~ Compositional expression

building blocks can be bits, circuits, data, functions, programs, ...

Farewell, Wires!

Welcome, Registers!

music- mation

https://www.youtube.com/watch?v=hyClpKAIFyo