Software

Python

How does Python **function** ?

Hmmm

4 Hmmm problems
+ 1 loop problem
due **Tues. 3/5**

Assembly Language

Machine Language

Fall break?!...

CS 5 this week

RAM

registers

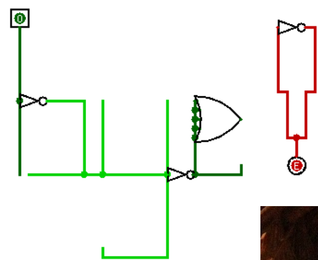
1-bit memory: flip-flops

arithmetic

bitwise functions

logic gates

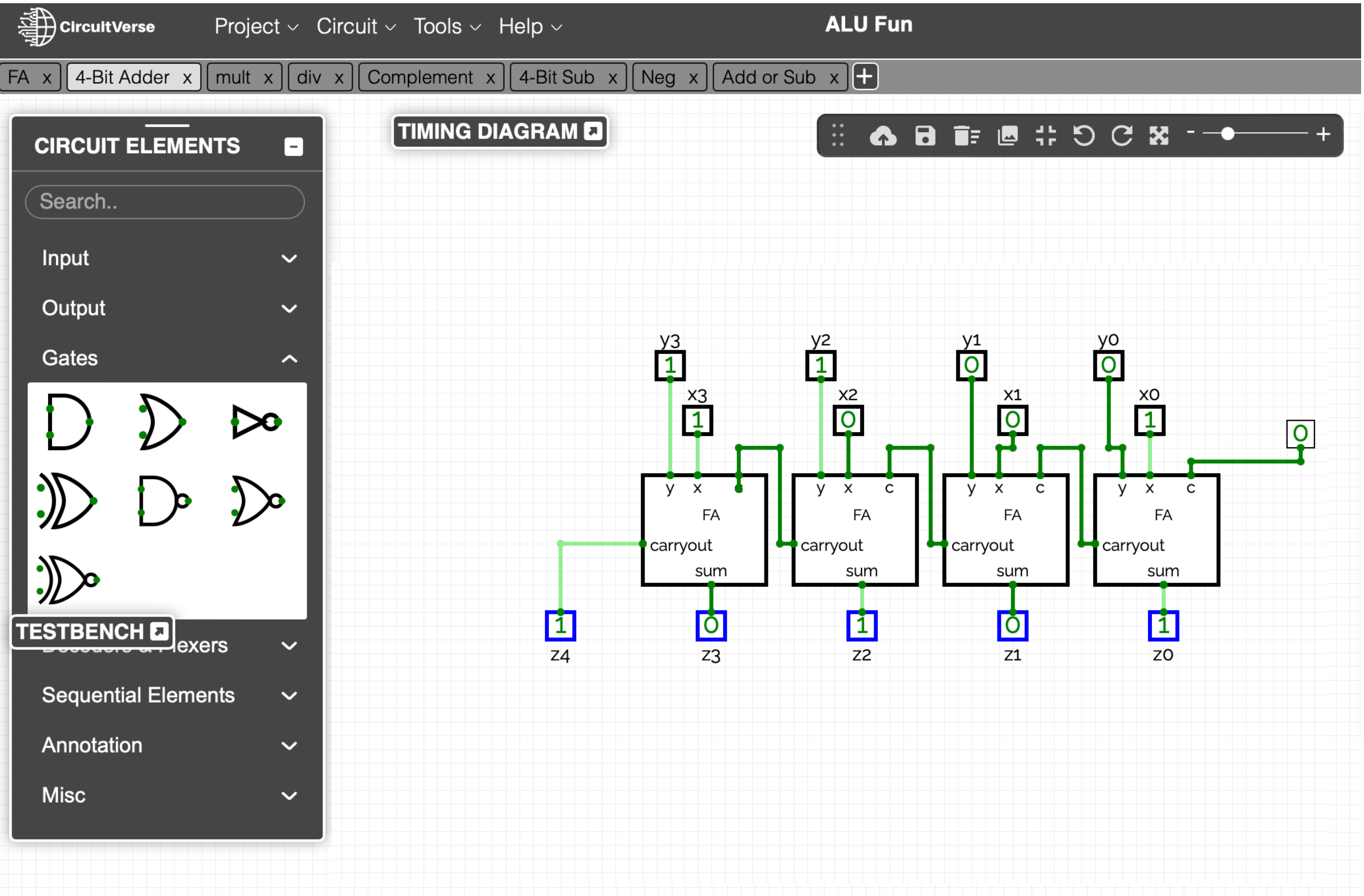
transistors / switches



I have a looming sense...

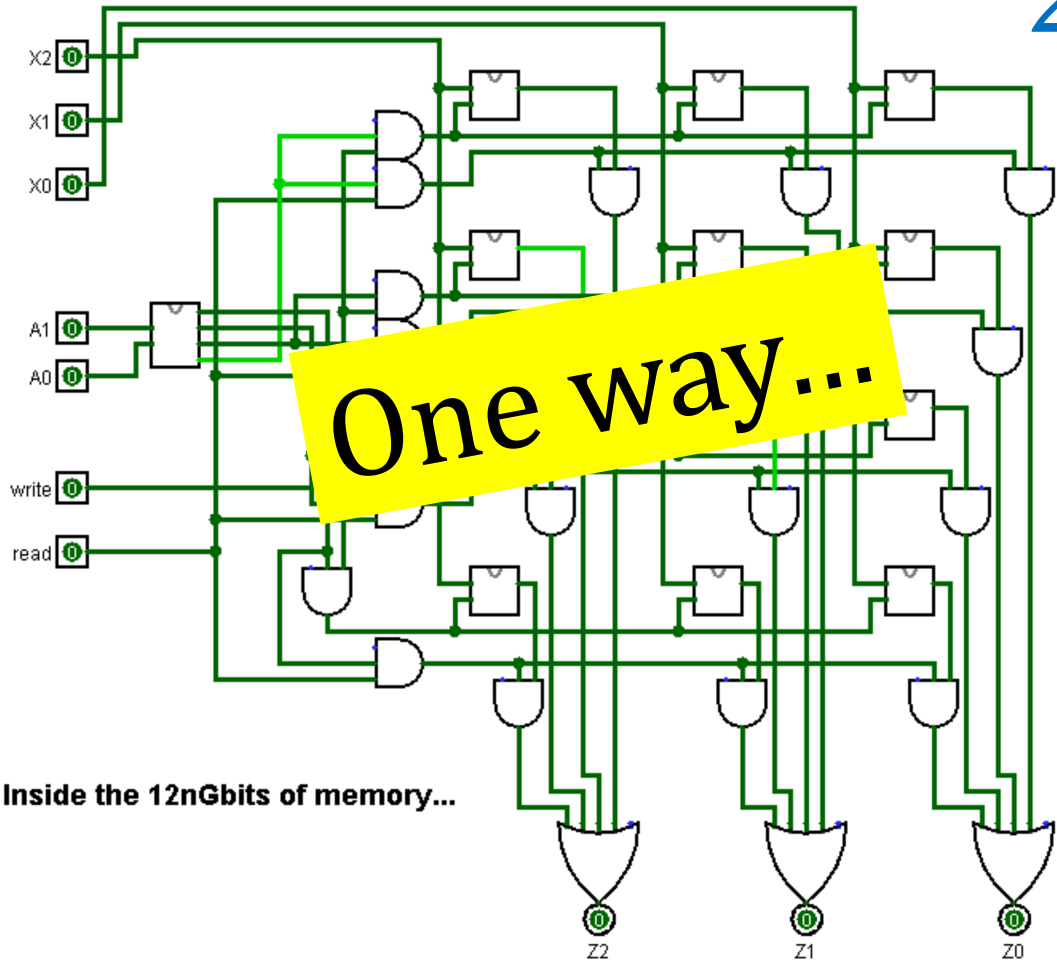
Hardware

Fun with circuits?



Making memories…

~1952-2024



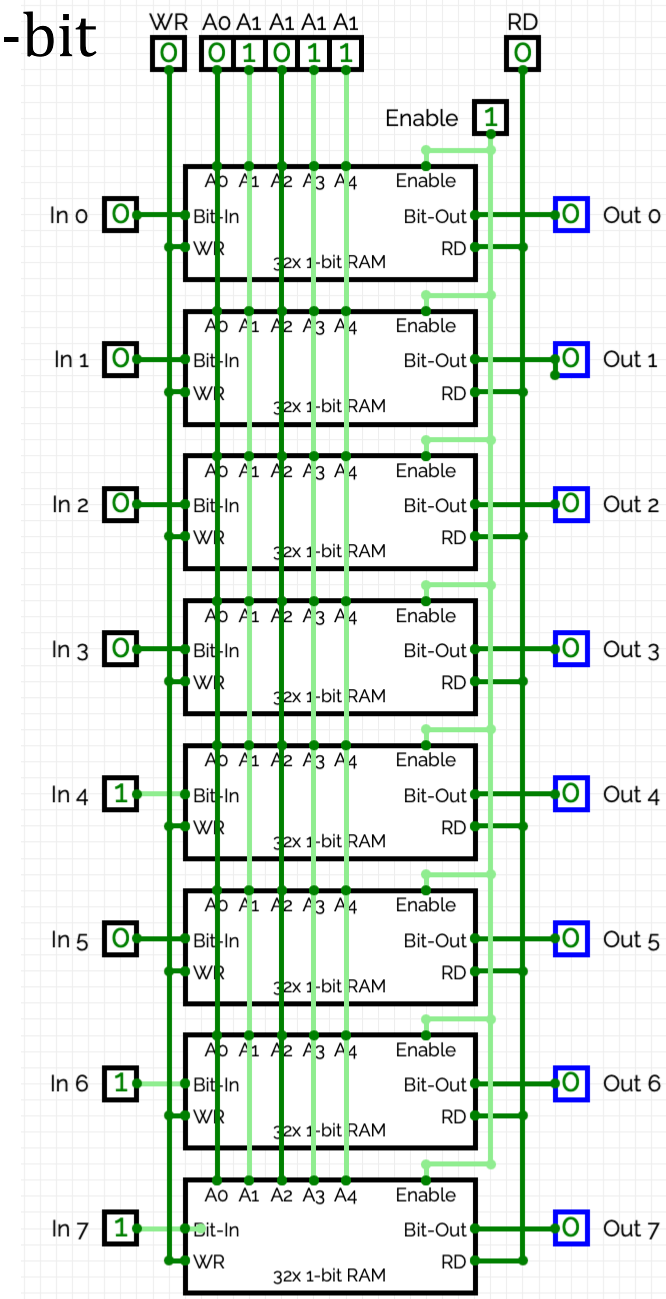
Inside the 12nGbits of memory…

Circuits ~ Memory!

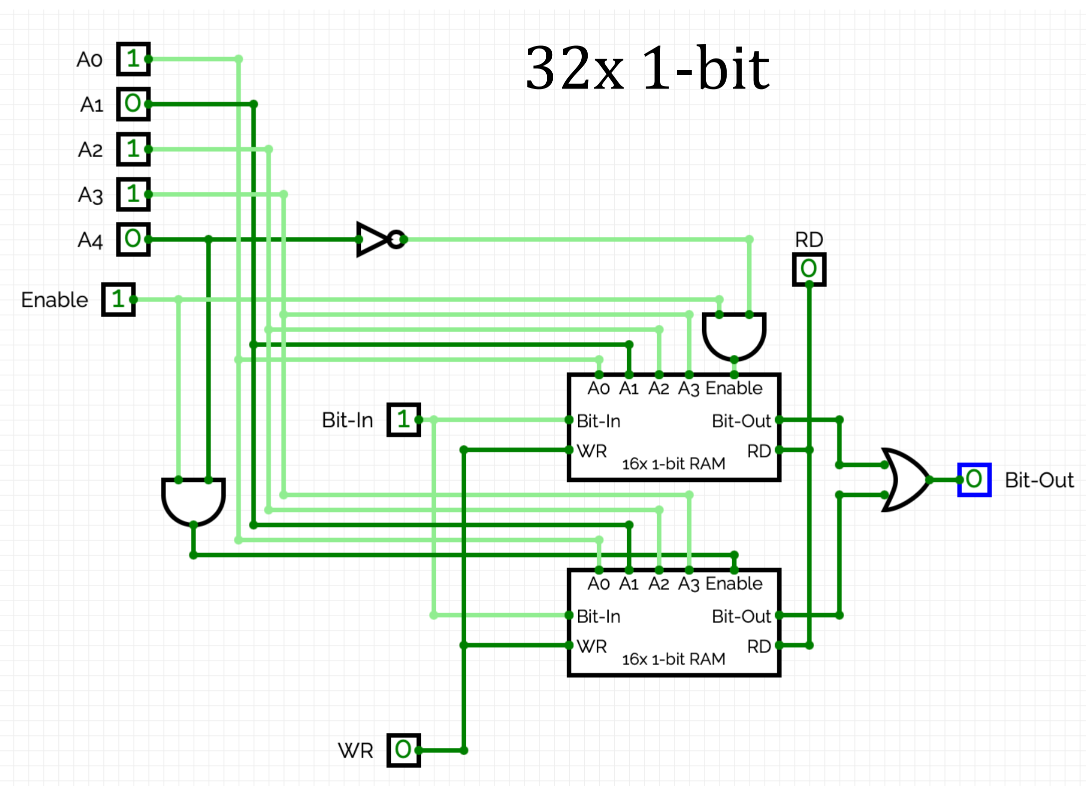
32 bytes of memory

the power of composition

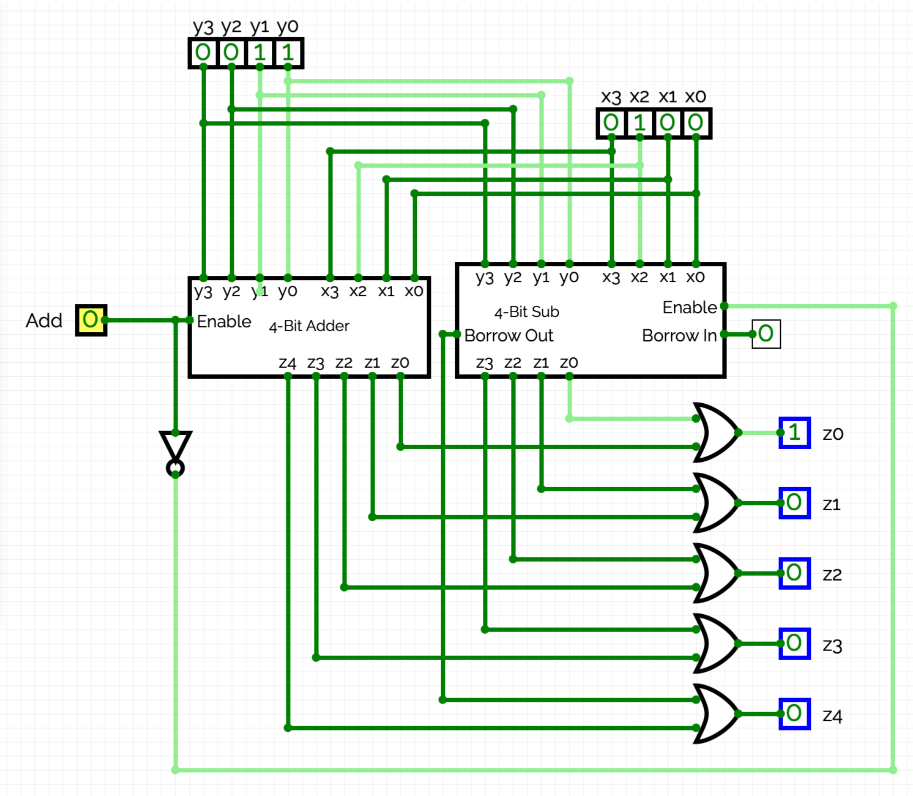
32x 8-bit

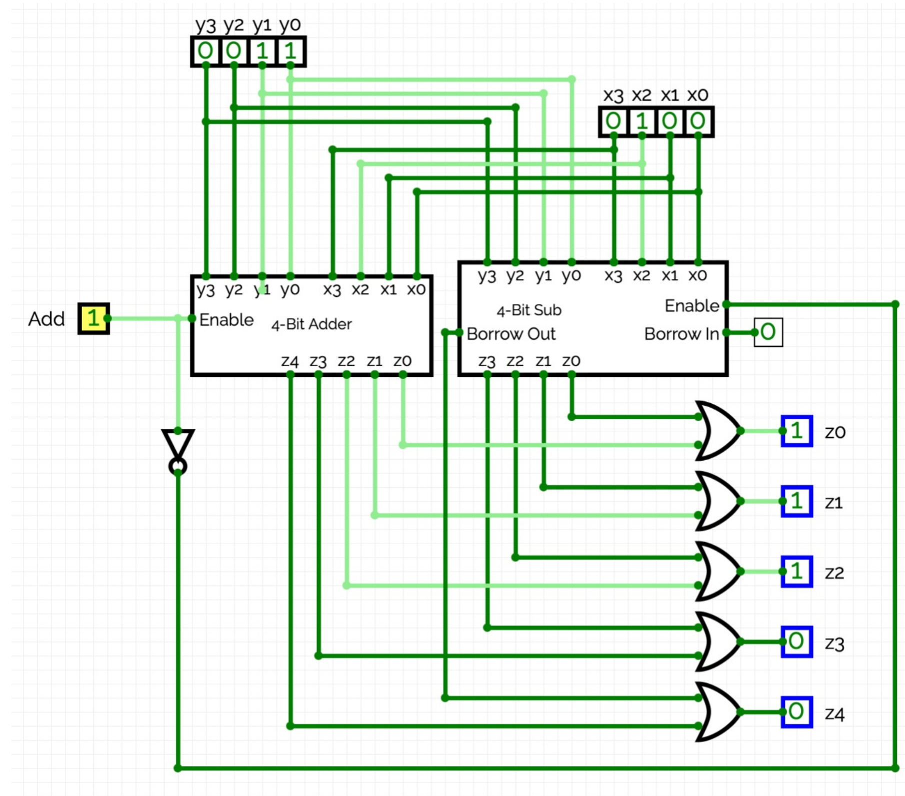


32x 1-bit

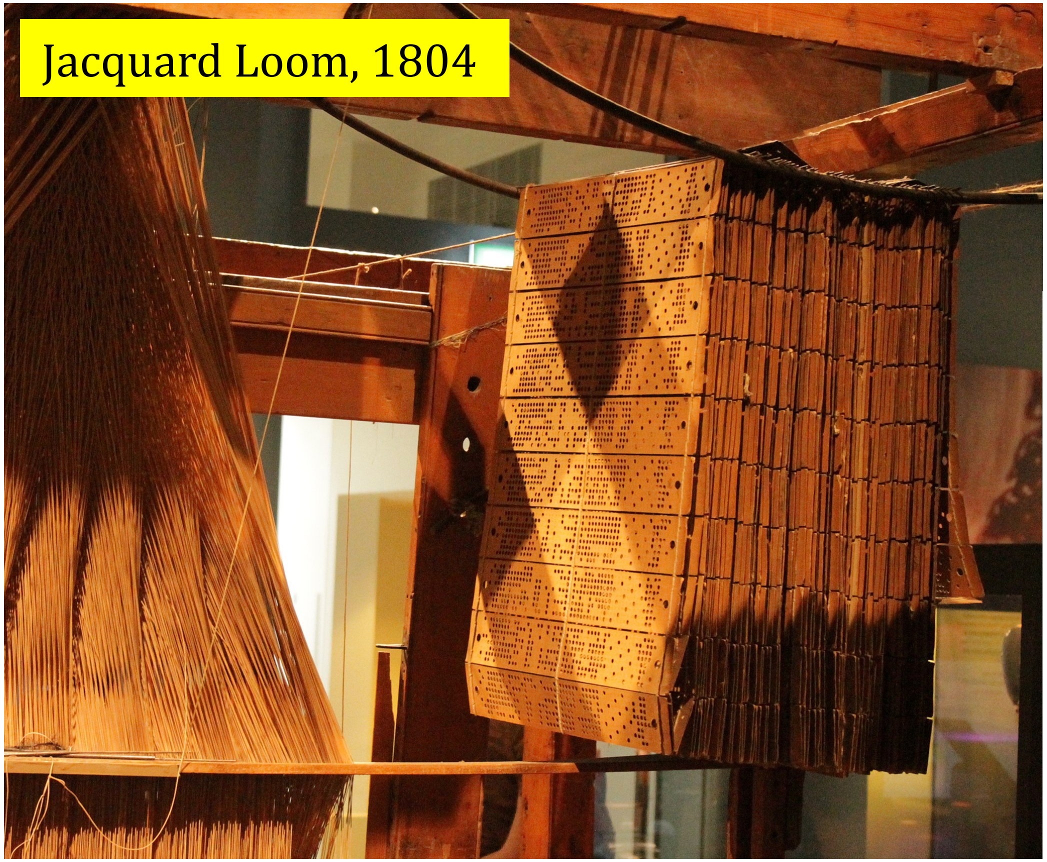


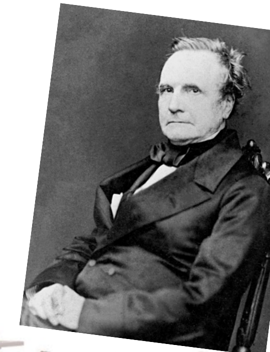
Fun with control?

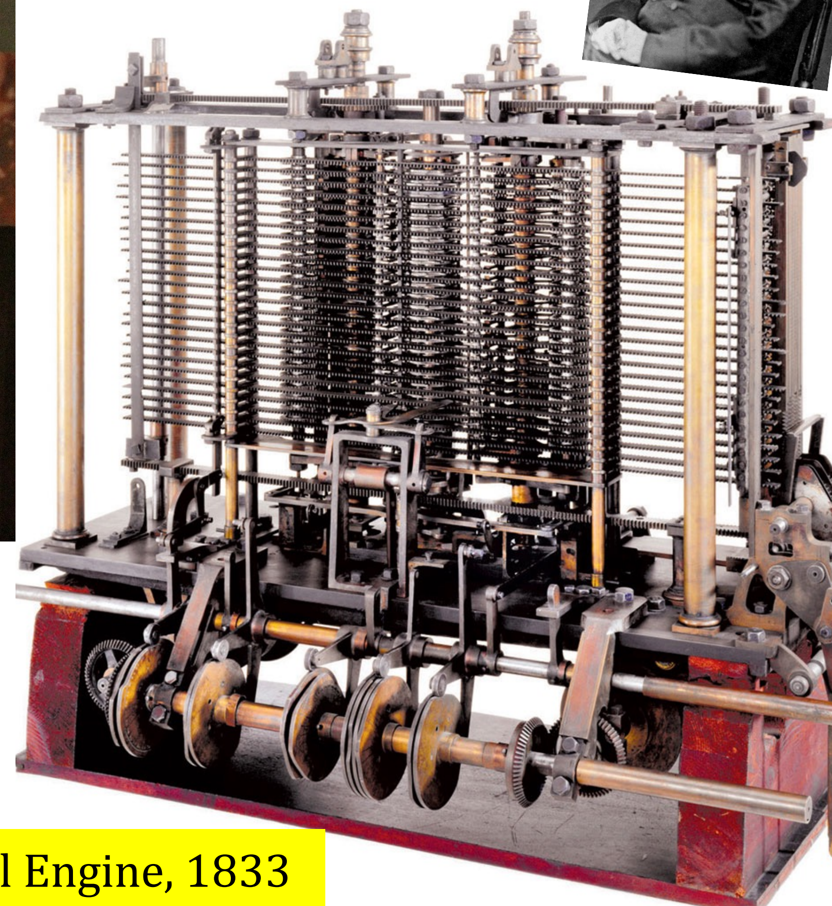




Early Binary Control…


Jacquard Loom, 1804

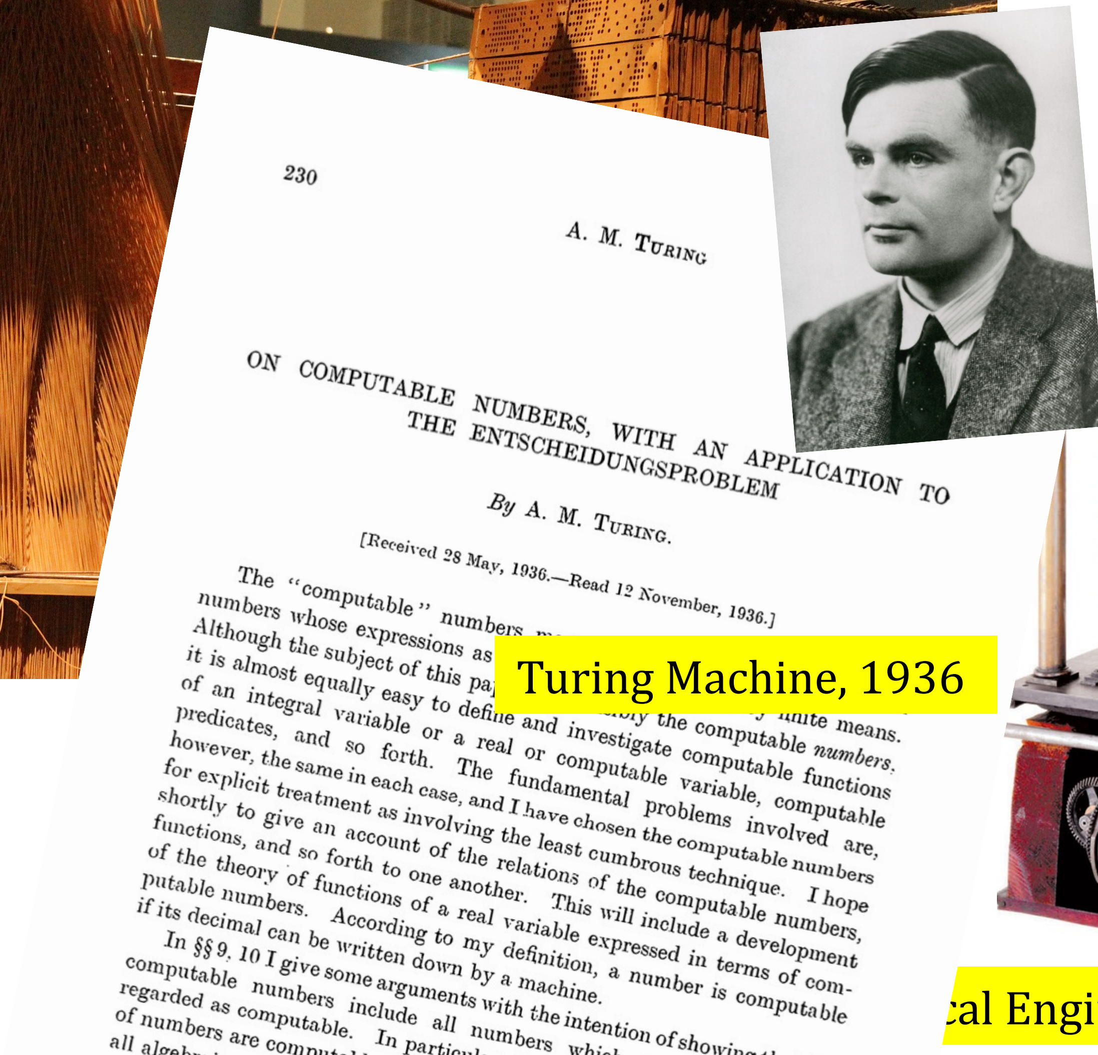



Babbage's Analytical Engine, 1833

Big idea: Control = Data

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

Jacquard Loom, 1804



230

A. M. Turing

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

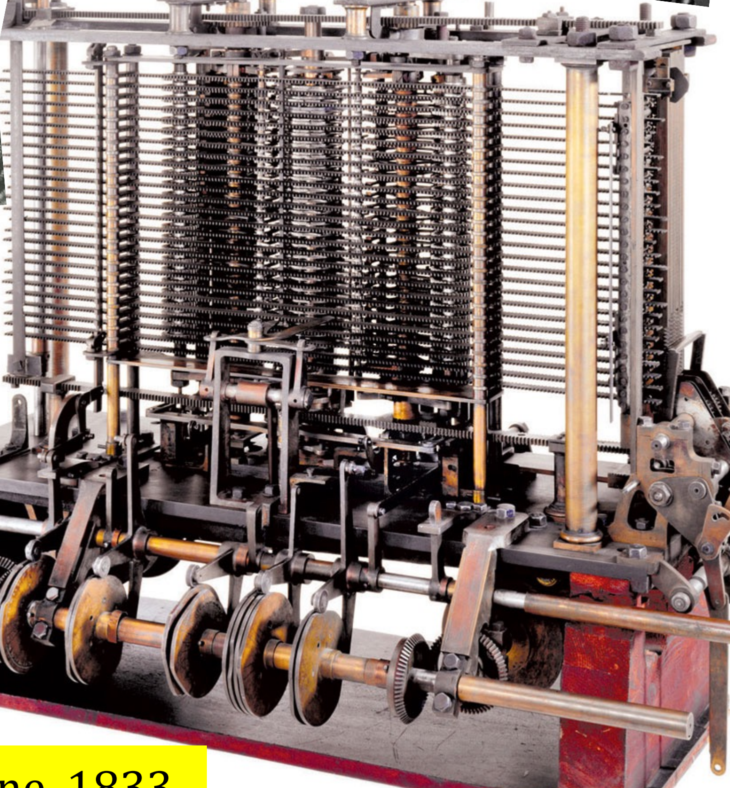
By A. M. Turing.

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

The "computable" numbers m... numbers whose expressions as ... finite means. Although the subject of this pa... ...bly the computable numbers, 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 th... computable numbers include all numbers which ... regarded as computable. In particul... of numbers are computabl... all algeb...

Turing Machine, 1936

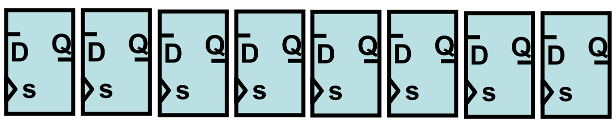


...cal Engine, 1833

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)

memory from *logic gates*

"Leaky Bucket" capacitors

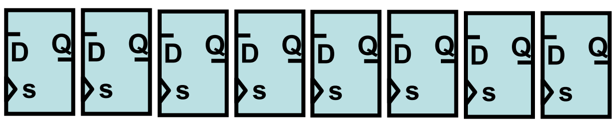
remagnetizing surfaces

Some memory is more equal than others...

Registers

on the Central Processing Unit

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

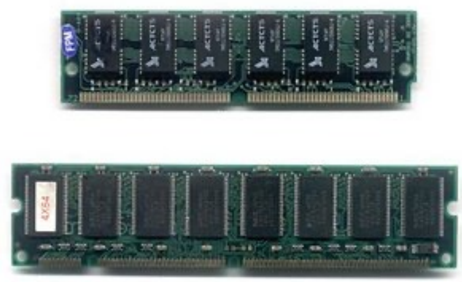


100 Registers of 64 bits each

~ 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)

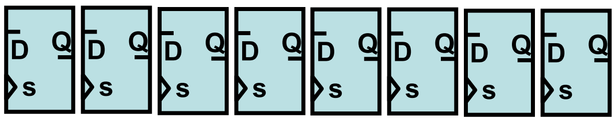
At least at my store!

	Registers	Main Memory	Disk Drive
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

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

Main Memory
(replaceable RAM)

10 GB memory

~ 100 billion bits

***running* programs are stored here...**

Disk Drive
magnetic storage

4 TB drive

~ 42 trillion bits (or more)

"Off" data is saved way out here...

P

Ti

sec

10^{-2} sec

If a c... cycle
== 1 minute

1 min

1.5 hours

19 YEARS

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

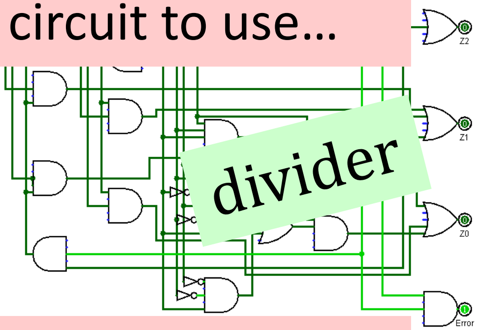
processor **CPU**

stores all instructions and almost all data

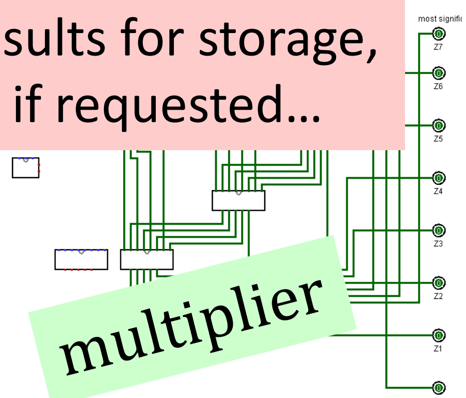
RAM live memory

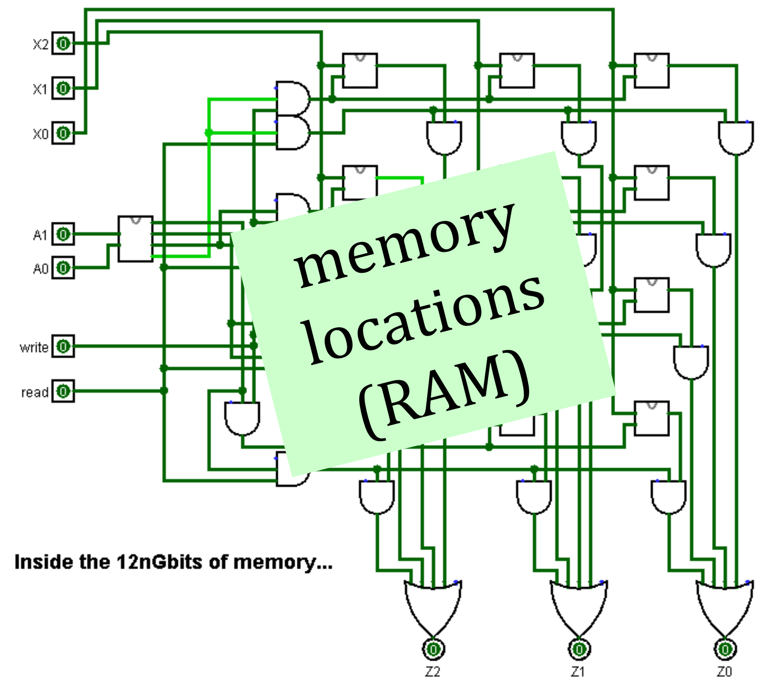
the instruction's bits select which circuit to use...

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***

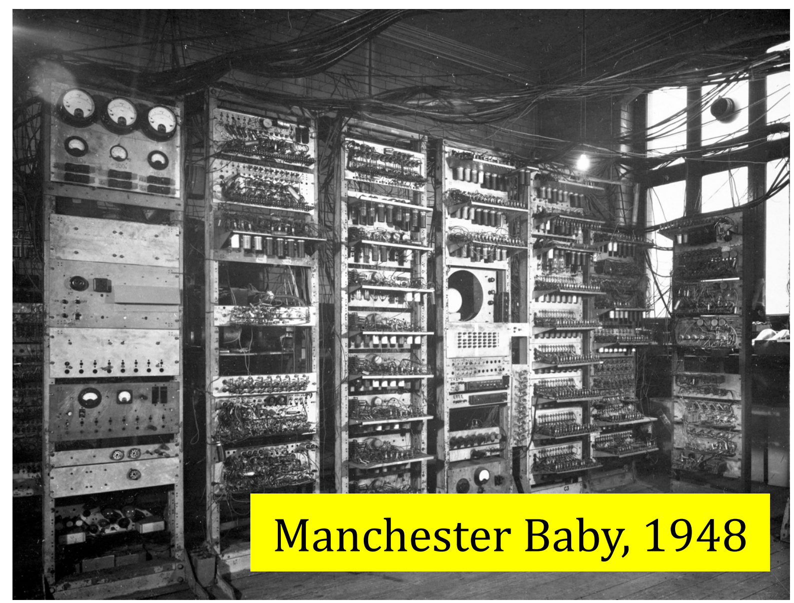
execute

fetch

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

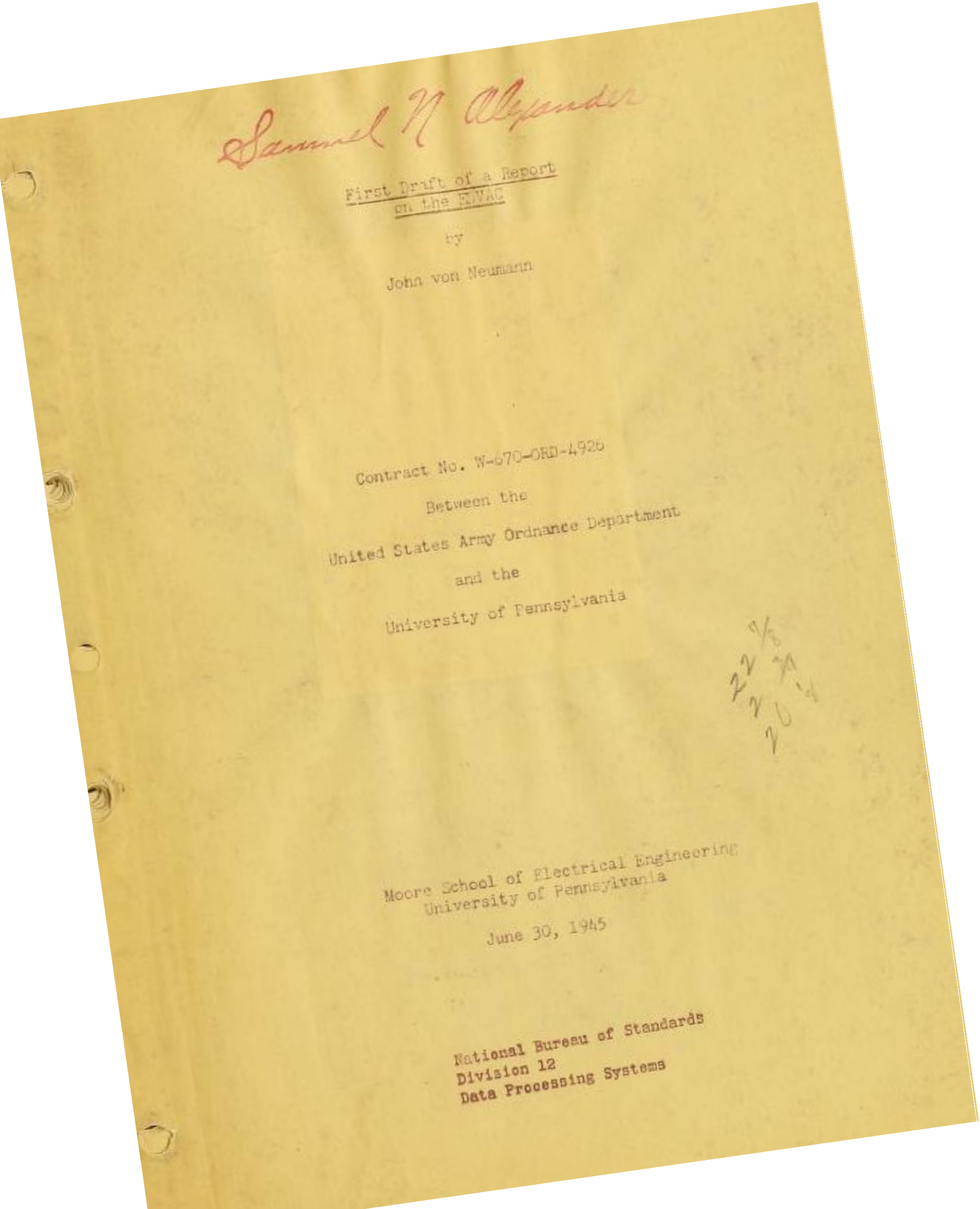
RAM

random access memory locations

limited, fast **registers**
+ arithmetic

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

John von Neumann



- 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

the `read` instruction

r1 General-purpose register, r1

r2 General-purpose register, r2

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 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
setn rX, #	`0001 xxxx #### ####`	Load an 8-bit integer # (-128 to +127) into register rX
loadr rX, rY	`0100 xxxx yyyy 0000`	Load register rX from memory word addressed by rY: rX = memory[rY]
storer rX, rY	`0100 xxxx yyyy 0001`	Store contents of register rX into memory word addressed by rY: memory[rY] = rX
popr rX rY	`0100 xxxx yyyy 0010`	Load contents of register rX from stack pointed to by register rY: rY –= 1; rX = memory[rY]
pushr rX rY	`0100 xxxx yyyy 0011`	Store contents of register rX onto stack pointed to by register rY: memory[rY] = rX; rY += 1
loadn rX, #	`0010 xxxx #### ####`	Load register rX with memory word at address #
storen rX, #	`0011 xxxx #### ####`	Store contents of register rX into memory word at address #
addn rX, #	`0101 xxxx #### ####`	Add the 8-bit integer # (-128 to 127) to register rX
copy rX, rY	`0110 xxxx yyyy 0000`	Set rX = rY
neg rX, rY	`0111 xxxx 0000 yyyy`	Set rX = -rY
add rX, rY, rZ	`0110 xxxx yyyy zzzz`	Set rX = rY + rZ
sub rX, rY, rZ	`0111 xxxx yyyy zzzz`	Set rX = rY - rZ
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
jumpr rX	`0000 xxxx 0000 0011`	Set program counter to address in rX
jumpn n	`1011 0000 #### ####`	Set program counter to address #
jeqzn rX, #	`1100 xxxx #### ####`	If rX = 0 then set program counter to address #
jnezn rX, #	`1101 xxxx #### ####`	If rX ≠ 0 then set program counter to address #
jgtzn rX, #	`1110 xxxx #### ####`	If rX > 0 then set program counter to address #
jltzn rX, #	`1111 xxxx #### ####`	If rX < 0 then set program counter to address #
calln rX, #	`1011 xxxx #### ####`	Set rX 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:

the **read** instruction

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[illegible] Prints "Enter number: " to prompt user for input
write r*X*	0000 xxx[illegible] 0010	Print the contents of register r*X* on standard output
setn r*X*, #	0001 xxxx #[illegible] ###	Load an 8-bit integer # (-128 to +127) into register r*X*
loadr r*X*, r*Y*	0100 xxxx yyy[illegible]	[illegible]ory word addressed by r*Y*: r*X* = memory[r*Y*]
storer r*X*, r*Y*	0100 xxxx yyyy [illegible]	[illegible]*X* into memory word addressed by r*Y*: memory[r*Y*] = r*X*
popr r*X* r*Y*	0100 xxxx yyyy [illegible]	[illegible]*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 [illegible]	[illegible]*X* onto stack pointed to by register r*Y*: memory[r*Y*] = r*X*; r*Y* += 1
loadn r*X*, #	0010 xxxx #### [illegible]	[illegible]ory word at address #
storen r*X*, #	0011 xxxx #### [illegible]	[illegible]*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[illegible] r*X* = r*Y* * r*Z*
div r*X*, r*Y*, r*Z*	1001 xxxx yyyy zzz[illegible]	[illegible] r*Z*
mod r*X*, r*Y*, r*Z*	1010 xxxx yyyy zzz[illegible]	[illegible]
jumpr rX	0000 xxxx 0000 0011	[illegible]
jumpn n	1011 0000 #### ####	[illegible]
jeqzn r*X*, #	1100 xxxx #### ####	[illegible] to address #
jnezn r*X*, #	1101 xxxx #### ####	[illegible] 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 #

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

the `read` instruction

r2 General-purpose register, r2

0	0000 0001 0000 0001	`read r1`
1	1000 0010 0001 0001	`mul r2 r1 r1`
2	0110 0010 0010 0001	`add r2 r2 r1`
3	0000 0	`write r2`
4	0000 0	`halt`
5		
6		

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

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...	
mod rX rY rZ	Set rX = rY % rZ (returns the ...ainder) ... 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

The Hmmm I...

There are 26 diff... accept a signed numerical ... between 0 and 255. All othe... hile numerical argum... instructions are: ... +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, #...

“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
```

Demo

of assembly-language programming in **Hmmm**...

in hw6, CS stands for Chin-Scratching?!

Example #1:

Screen: 6 (input)

CPU
central processing unit ***registers***

Von Neumann bottleneck

RAM
random access memory locations

r1
General-purpose register r1

r2
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

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

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		

2019

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

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%

2021...

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

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		Visual Basic	5.86%	+1.85%
7	7		JavaScript	2.12%	-0.33%
8	8		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	v	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	v	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%

October 2022

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

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	[illegible]	[illegible]
18	20	↑	Rust	[illegible]	[illegible]
19	30	⇑	COBOL	1.01%	[illegible]
20	16	⇓	Ruby	[illegible]	[illegible]

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	v	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	vv	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%

I FEEL LIKE

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		

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

Software is **written** in many languages

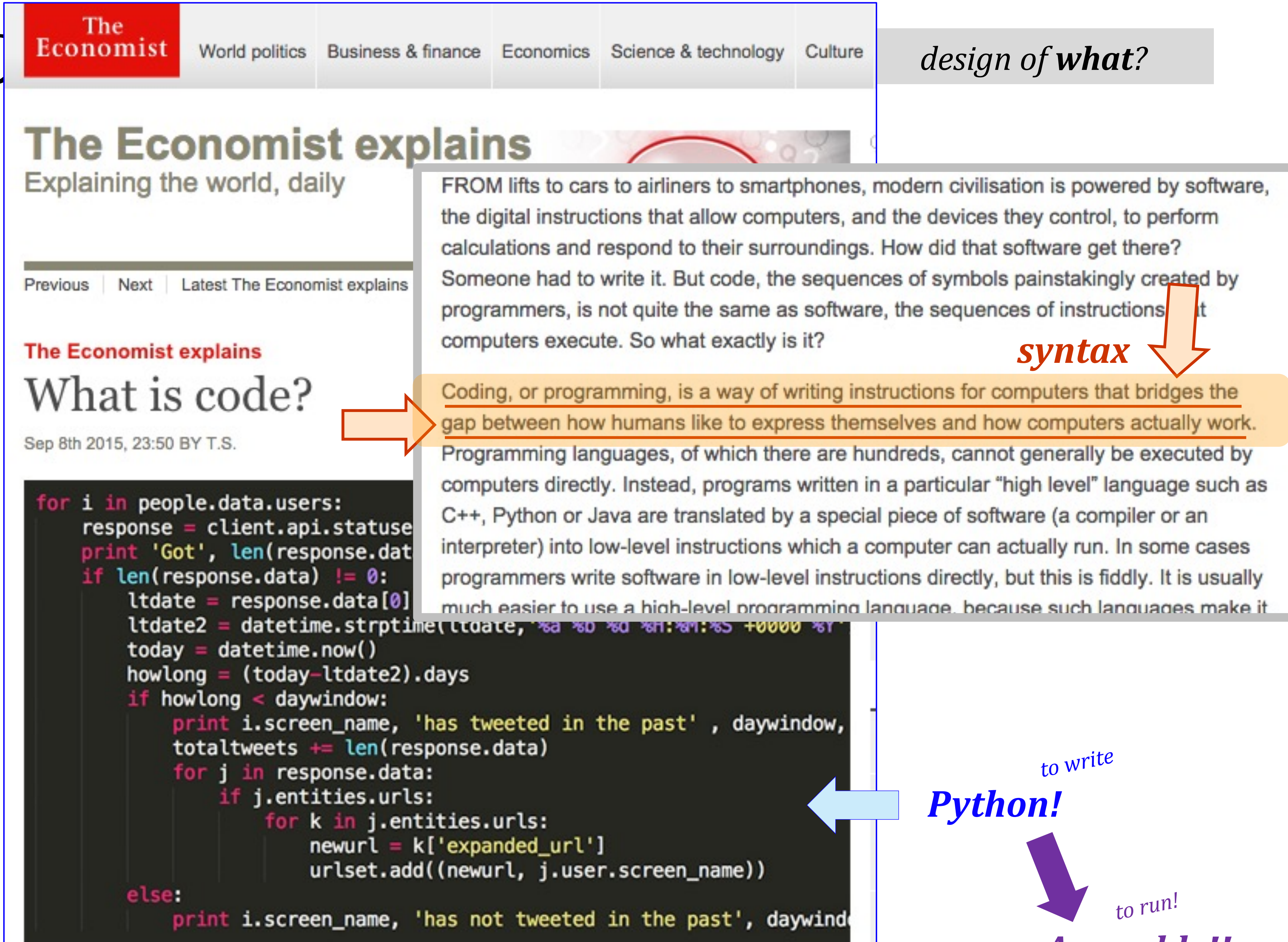
February 2024

I FEEL LIKE

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

22

design of ***what?***



The Economist | World politics | Business & finance | Economics | Science & technology | Culture

The Economist explains

Explaining the world, daily

Previous | Next | Latest The Economist explains

The Economist explains

What is code?

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

```
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
```

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

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

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

Assembly 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

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

`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!

Names(s): ____________________

"Quiz"

Try it!

screen

100 (input)

(output)

42

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 2

General-purpose register r4

RAM

random access memory

		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`	

100

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 50

General-purpose register r3

r4 2

General-purpose register r4

RAM

random access memory

		Python
0	read r1 ← 100	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...!?

dd 56

Extra! Change the instruction on line 4 to creat the overall output of 56 or 349 or 0 or 6 ... ?

mod 0

div 6

mul 349

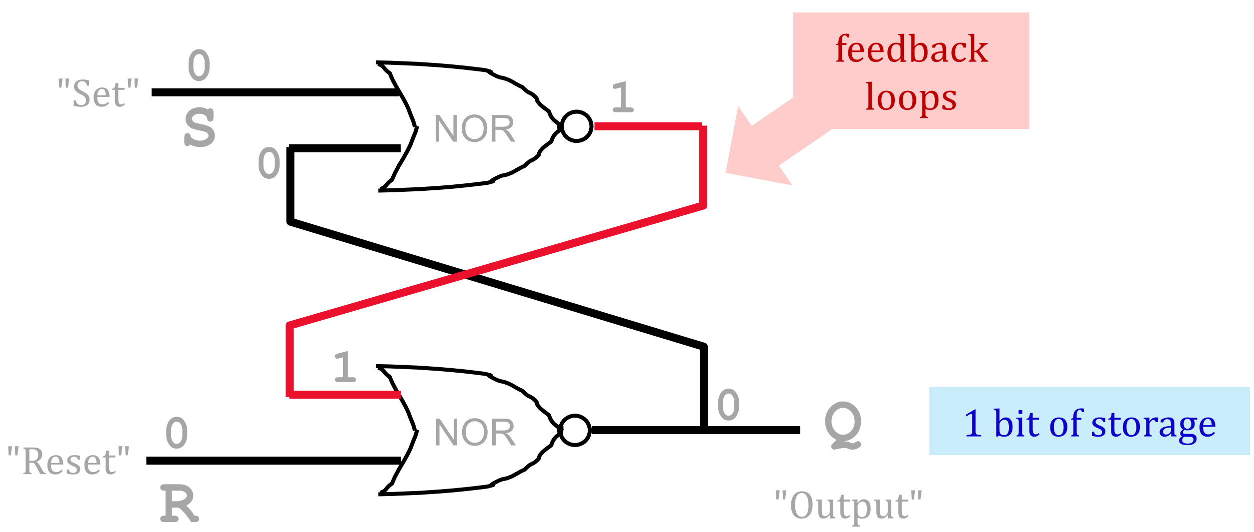
Is this all we need?

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.



Loops and **ifs**

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

It's too linear!

loop

jumpn!

"straight-line code"

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

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	jumpn 3	jumpn 2	jumpn 0
Screen 42 no crash	Screen 42 no crash	Screen 42 crash!	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 **eq**ual to **z**ero…	THEN jump to line number 42
`jgtzn`	if **g**reater **t**han **z**ero …	EN jump to line number 42
`jltzn`	if **l**ess **t**han **z**ero…	THEN jump to line number 42
`jnezn`	if **n**ot **e**qual to **z**ero …	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

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

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

Jumps!

off-processor access... [Thursday]

jgtzn

What Python f'n is this?

CPU
central processing unit

r1

General-purpose register r1

r2

General-purpose register r2

Screen

-6 (input)

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!

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

	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 <u>only line 3</u> 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 <u>test</u> 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...)