Jotto, *Cornered*

Mo/Kepa

AM guess	my guess	PM guess	my guess
alien: 2	diner: 1	**alien: 2**	diner: 0
whole: 1	savvy: 1	**bears: 2**	savvy: 1
sloth: 1	flock: 2	**flour: 1**	flock: 2
grump: 0	thumb: 1	**crazy: 2**	thumb: 1
dorky: 0	flesh: 0	**whine: 1**	blobs: 2
61	13	47	48

words remaining

alpha-betical

anagrams

abcio - cobia 1
abcno - bacon banco 2
abcor - carob cobra 2
acdot - octad 1
aciot - coati 1
ackrt - track 1
acmor - carom macro 2
acnot - canto cotan 2
acort - actor taroc 2
akort - tarok troak 2
bcory - corby 1
cdkuy - ducky 1
cnoty - cyton 1

aaitw - await 1
abbes - abbes babes 2
abess - bases sabes 2
abett - betta 1
acees - cease 1
acefs - cafes faces 2
aceft - facet 1
acess - cases 1
acesv - caves 1
acett - tacet tecta 2
acetx - exact 1
aeefs - fease 1
aeess - eases 1
aeesv - eaves 1
aefss - safes 1
aefsx - faxes 1
aefsz - fazes 1
aessv - saves vases 2
aessx - saxes 1

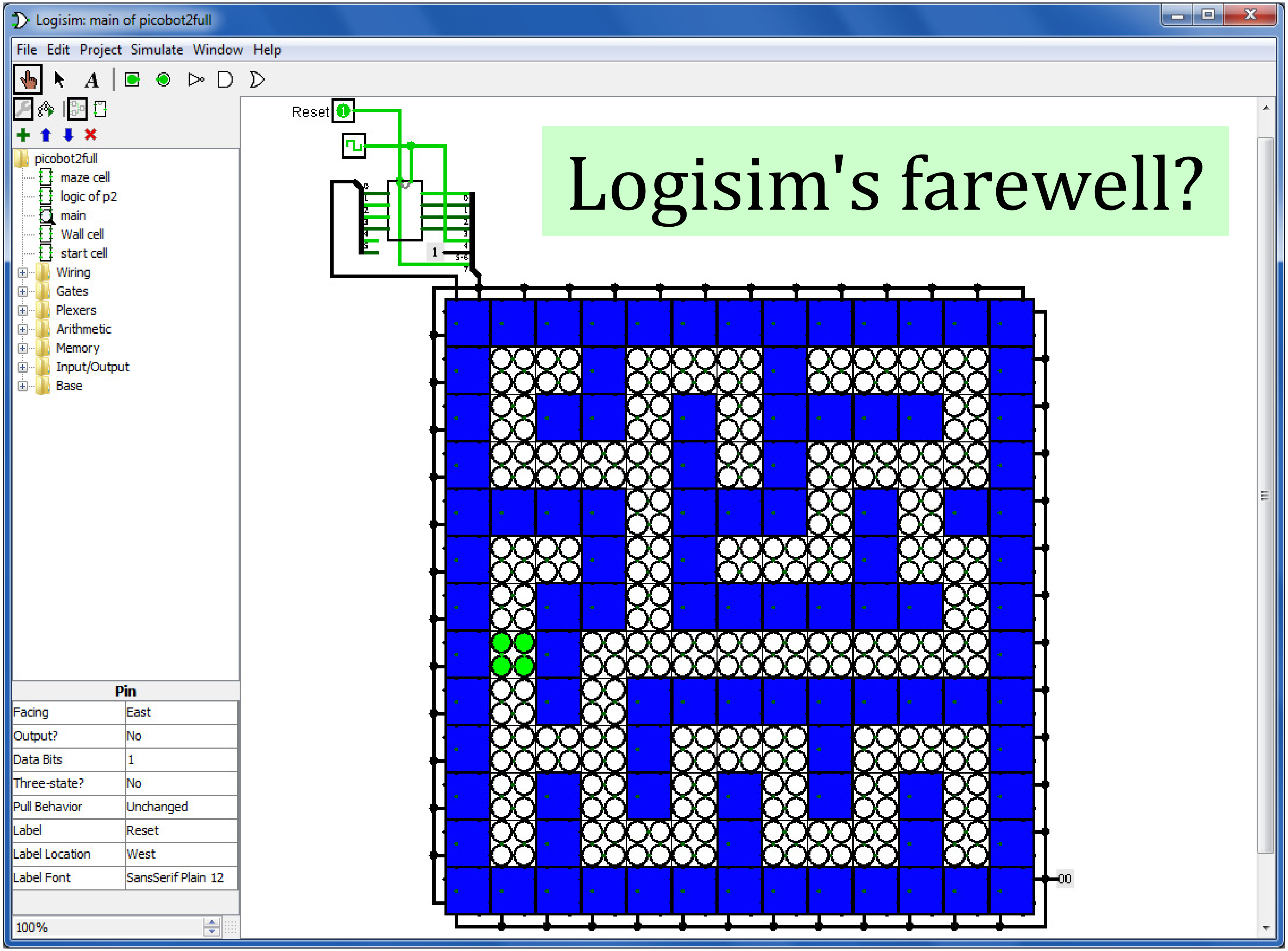
aesxz - zaxes 1
aetzz - tazze 1
afhiz - hafiz 1
afisw - waifs 1
afnsw - fawns 1
ahijj - hajji 1
anssw - snaws swans 2
beens - benes 1
befit - befit 1
beijs - jibes 1
beisv - vibes 1
beitz - zibet 1
cchin - cinch 1
ceens - cense scene 2
ceiss - sices 1
ceisv - vices 1
ceitv - civet evict 2
cffil - cliff 1
cfhin - finch 1
ciilv - civil 1
eeiss - seise 1
eeisv - sieve 1
eeisz - seize 1
eeitv - evite 1
eenss - sense 1
eensv - evens seven 2
eentt - tenet 1
eentv - event 1
effis - fiefs fifes 2
efisv - fives 1
efisx - fixes 1
eiisv - ivies 1
eijsv - jives 1
eisss - sises 1
eissv - vises 1
eissx - sixes 1
eissz - sizes 1

aabcl - cabal 1
aahlo - aloha 1
aglot - gloat 1
ahllo - hallo holla 2
allmo - molal 1
allot - allot atoll 2
almoo - moola 1
alott - total 1
bflyy - flyby 1
chops - chops 1
chosw - chows 1
cllsu - culls scull 2
clpsu - sculp 1
cmops - comps 1
coost - coots scoot 2
copsu - coups 1
cosst - costs scots 2
fflsu - luffs sluff 2
ffost - toffs 1
fglsu - gulfs 1
fhoos - hoofs 1
foost - foots 1
fopsu - poufs 1
fosst - softs 1
fostt - tofts 1
hkoos - hooks shook 2
hkops - kophs 1
hlloy - holly 1
jloty - jolty 1
kklsu - skulk 1
kllsu - skull 1
klssu - sulks 1
koost - kotos stook 2
lllmoy - molly 1
lotyz - zloty 1

aaimr - maria 1
aairt - atria riata tiara 3
aairv - varia 1
aanrt - antra ratan 2
aanrv - navar varna 2
abcno - bacon banco 2
ablwy - bylaw 1
abnuy - bunya 1
acdef - faced 1
aceft - facet 1
acemo - cameo comae 2
acetu - acute 1
achls - clash 1
aclsw - claws 1
acnos - canso 1
adefy - fayed 1
adefz - fazed 1
adeoz - adoze 1
adgnr - grand 1
adiir - radii 1
adipr - padri pardi rapid 3
adirt - triad 1
adirx - radix 1

adknr - drank 1
aeefz - feaze 1
aeeoz - zoeae 1
aeguz - gauze 1
aeotz - azote 1
agirt - tragi 1
agirv - virga 1
agnpr - prang 1
agnrr - gnarr 1
agnrt - grant 1
ahlsy - hylas shaly 2
aikmr - mikra 1
aikrt - krait traik 2
aimpr - prima 1
aiprt - atrip tapir 2
airtt - trait 1
airvx - varix 1
aknpr - prank 1
alswy - yawls 1
anosz - azons 1
ansuy - unsay yuans 2

Logisim's farewell?



Pin

Facing	East
Output?	No
Data Bits	1
Three-state?	No
Pull Behavior	Unchanged
Label	Reset
Label Location	West
Label Font	SansSerif Plain 12

CS 5 this week

Software

Python

How does Python **function** ?

Hmmm

☆

RAM

registers

1-bit memory: flip-flops

arithmetic

bitwise functions

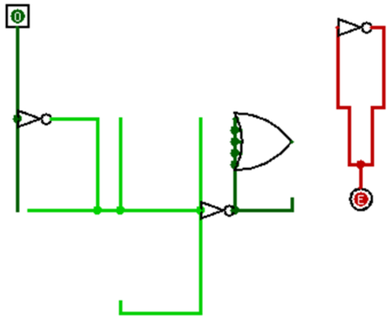
logic gates

transistors / switches

Hardware

4 Hmmm problems
+ 2 loop problems
due **Mon. 10/26**

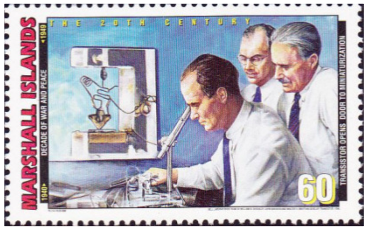
Fall break is also a
CS hw break...



Equilibrium
punctuaters

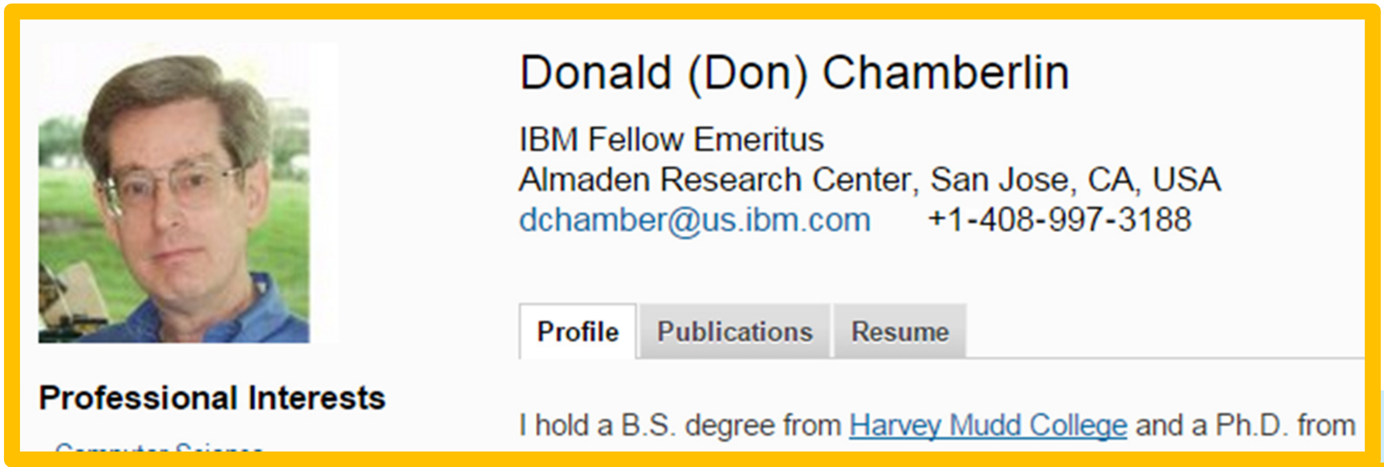
Grace
Hopper

Bell labs



Von Neumann

Turing, et al.



Donald (Don) Chamberlin

IBM Fellow Emeritus
Almaden Research Center, San Jose, CA, USA
dchamber@us.ibm.com +1-408-997-3188

Professional Interests

I hold a B.S. degree from Harvey Mudd College and a Ph.D. from

Don Chamberlain

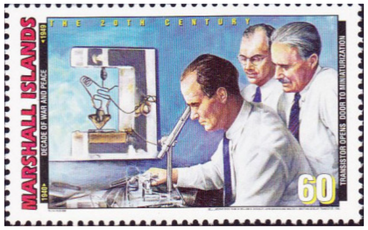
(Thursday ~ 25 min)

"50 years of data"

week

↑

Equilibrium
punctuaters



Von Neumann

Turing, et al.

re

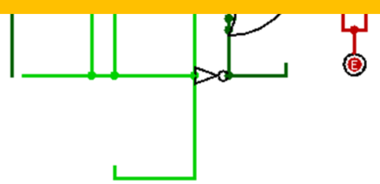
1-bit memory: fl

ari

bitwise functions

logic gates

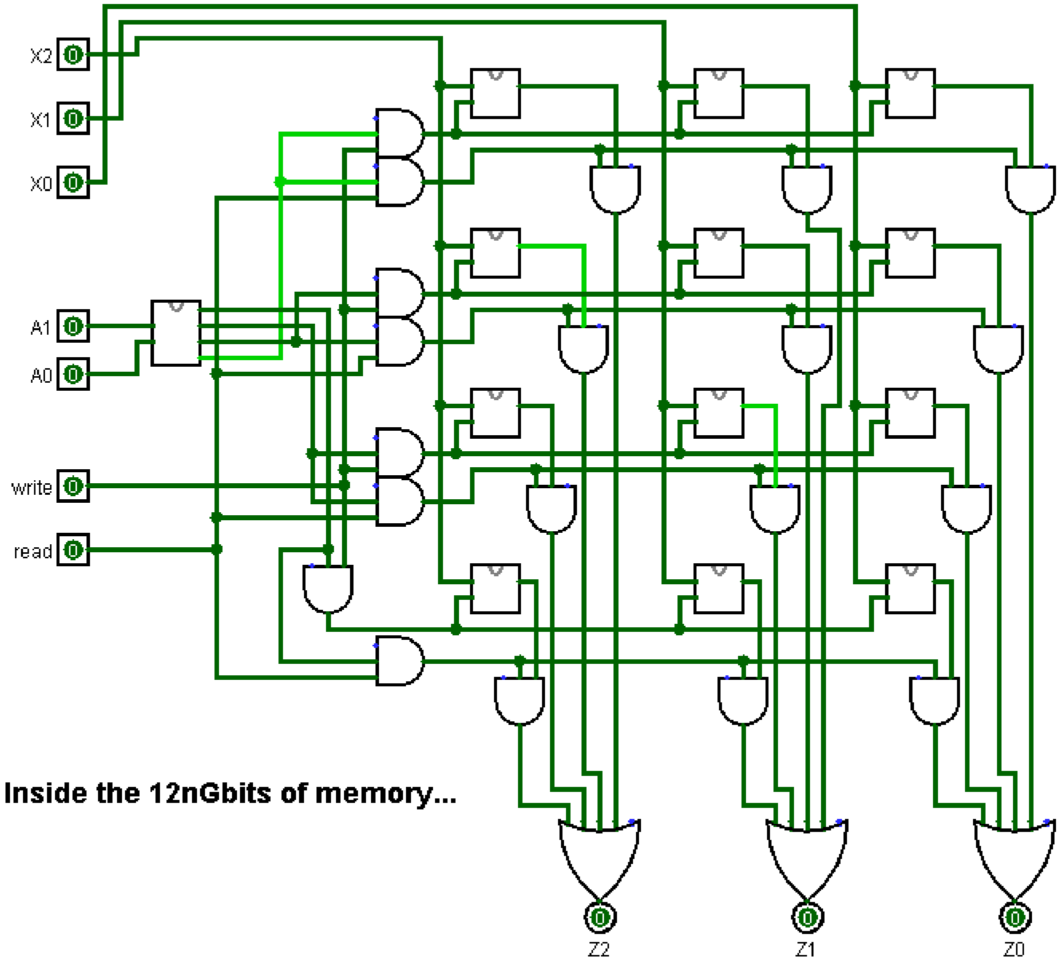
transistors / switches



Hardware

Graders' thoughts…

Now, where were we… ?



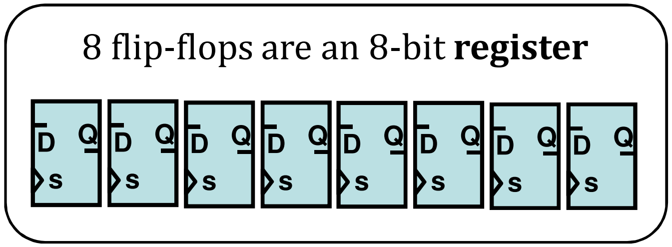
Inside the 12nGbits of memory...

Memory!

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

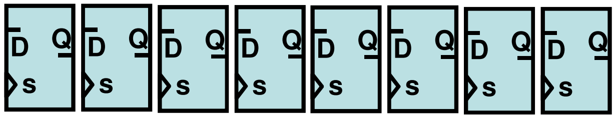
~ 42 trillion bits (or more)

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

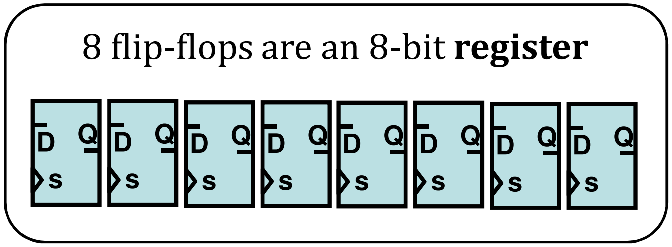
"640K ought to be enough for anybody"

- Bill Gates (*contested*)

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

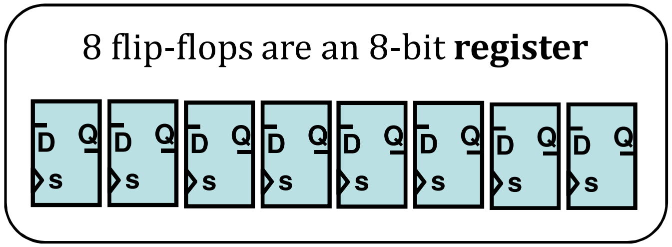
~ 42 trillion bits (or more)

Price	~$100	~$100	~$100
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**	

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)

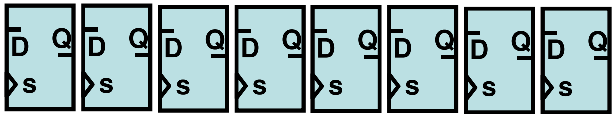
Price	~$100	~$100	~$100
Time	1 clock cycle 10^{-9} sec	100 cycles 10^{-7} sec	10^7 cycles 10^{-2} sec
If a clock cycle == 1 minute	**1 min**	**1.5 hours**	**19 YEARS**

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)

P

Ti

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

sec

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

sec

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

10^{-2} sec

If a clock cycle == 1 minute

1 min

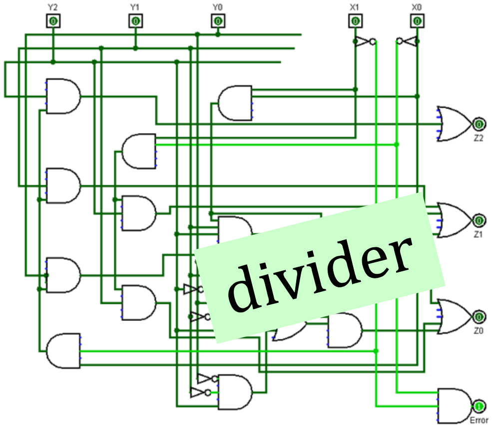
1.5 hours

19 YEARS

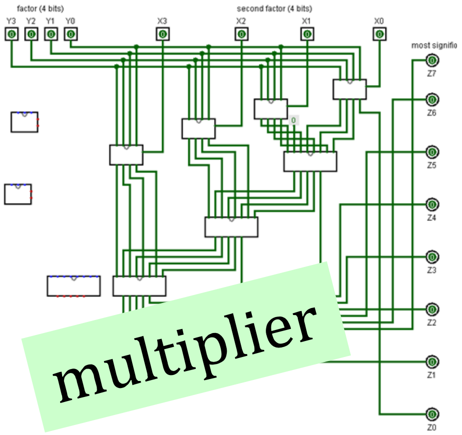
How do we execute ***sequences*** of operations?

processor CPU

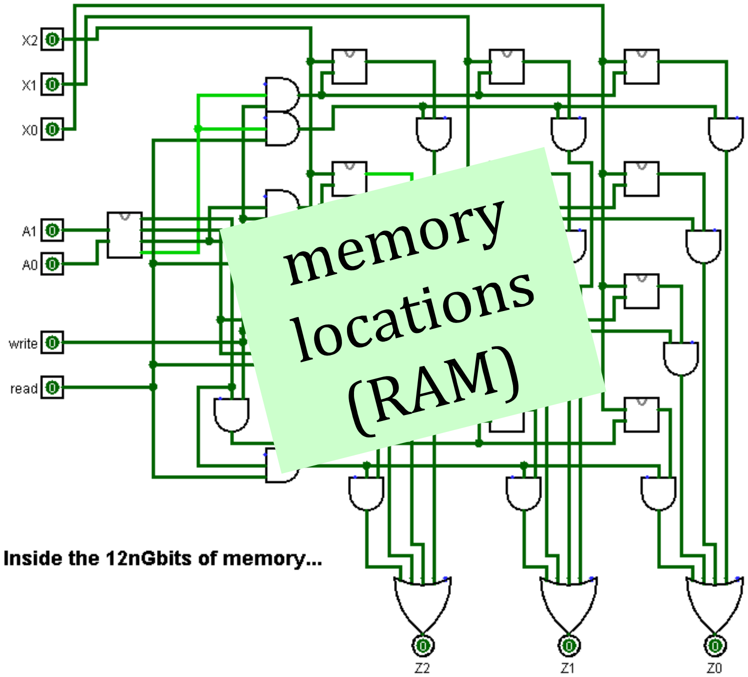
RAM live memory



divider



multiplier



Inside the 12nGbits of memory...

memory locations (RAM)

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

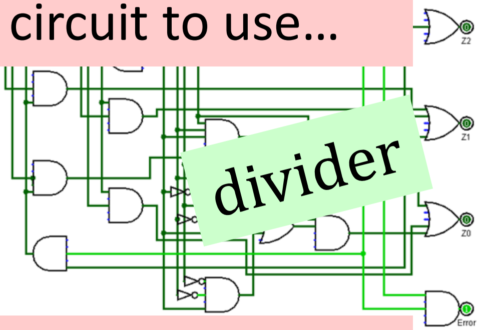
processor **CPU**

stores all instructions and almost all data

RAM live memory

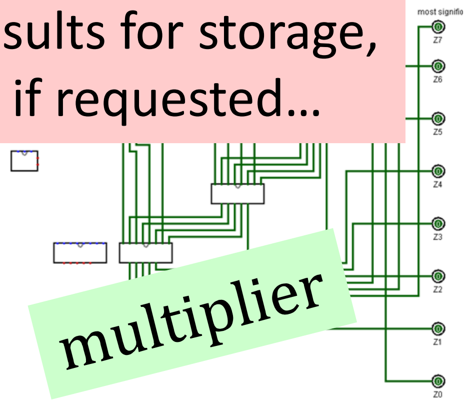
the instruction's bits select which circuit to use...

sends next instruction to the CPU ...

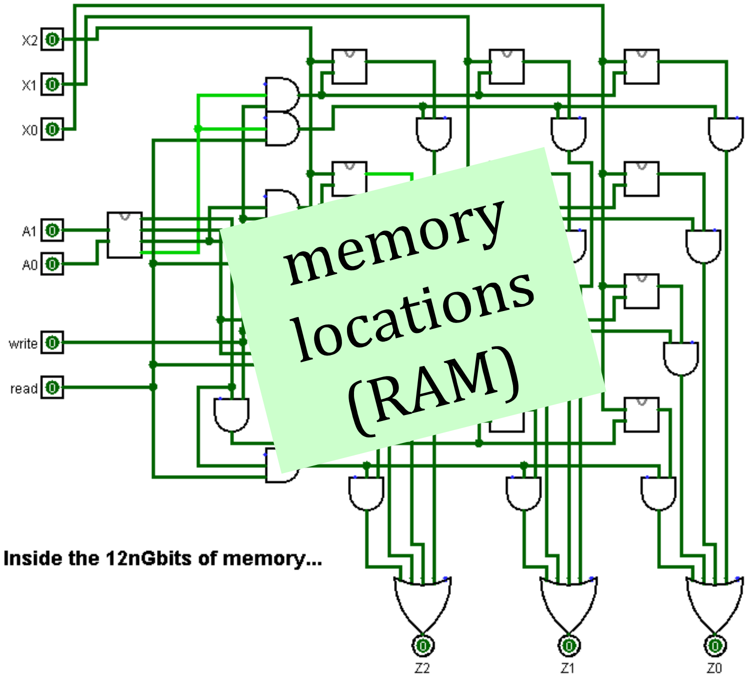


divider

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



multiplier

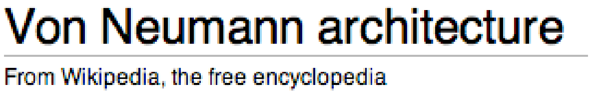


Inside the 12nGbits of memory...

memory locations (RAM)

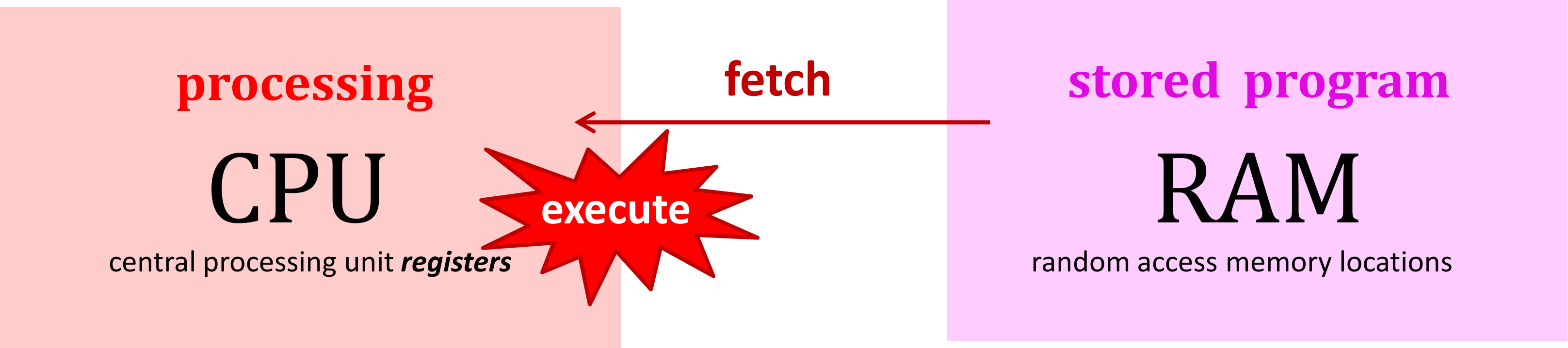
sends next instruction to the CPU ...

70 years ago…


Von Neumann architecture

From Wikipedia, the free encyclopedia

Jon V.N.


processing

CPU

central processing unit ***registers***

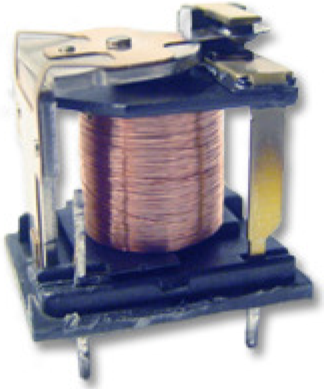
execute

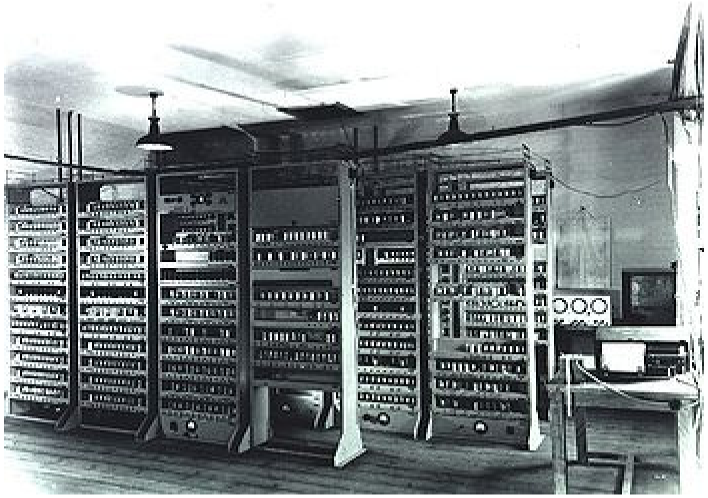
fetch

stored program

RAM

random access memory locations





limited, fast **registers**
+ arithmetic

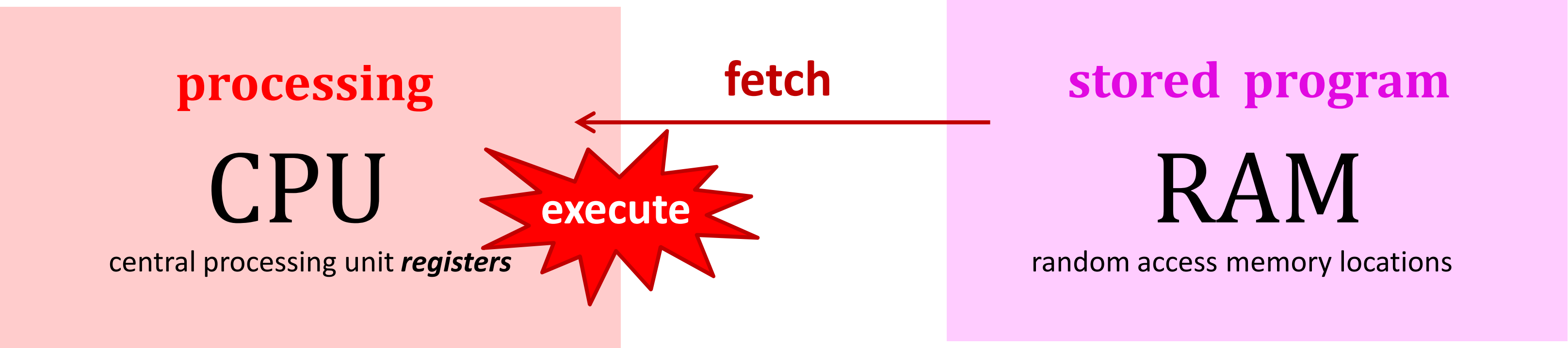
larger, slower **memory**
+ *no* computation

70 years later...

Von Neumann architecture

From Wikipedia, the free encyclopedia

Jon V.N.



processing

CPU

central processing unit ***registers***

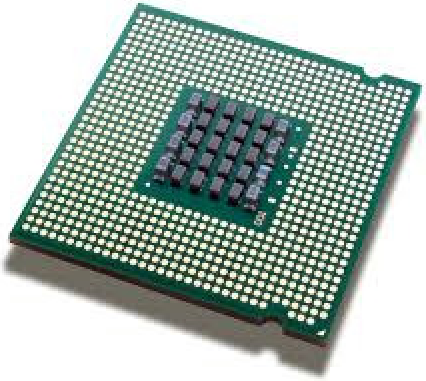
execute

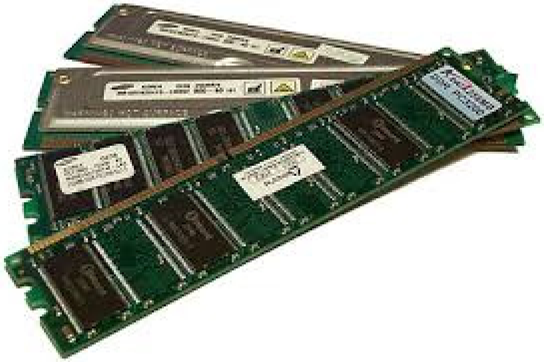
fetch

stored program

RAM

random access memory locations





limited, fast **registers**
+ arithmetic

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

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	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 stored in memory in *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

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!

Demo

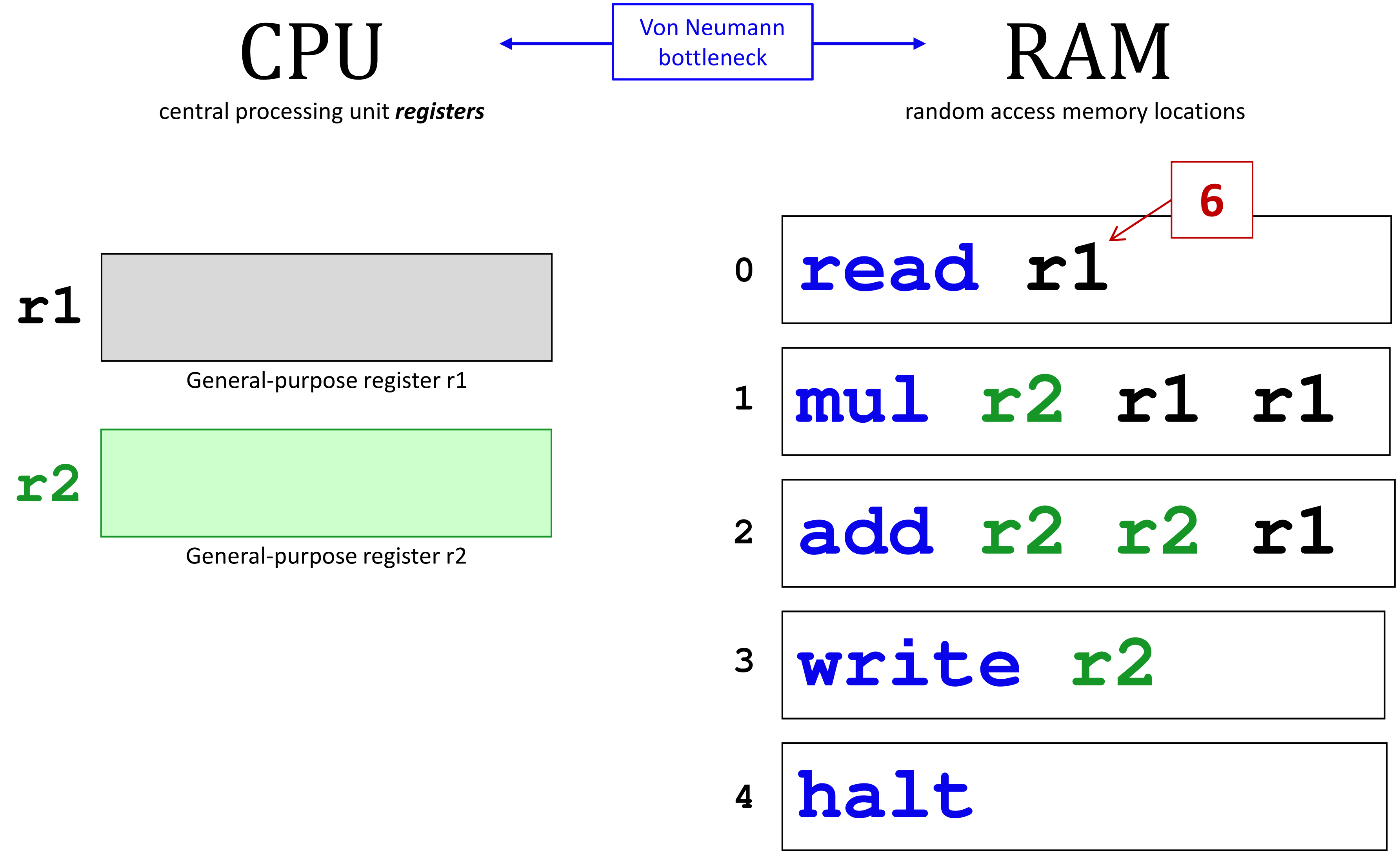
of "in vivo" assembly-language

hw6's big picture...

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

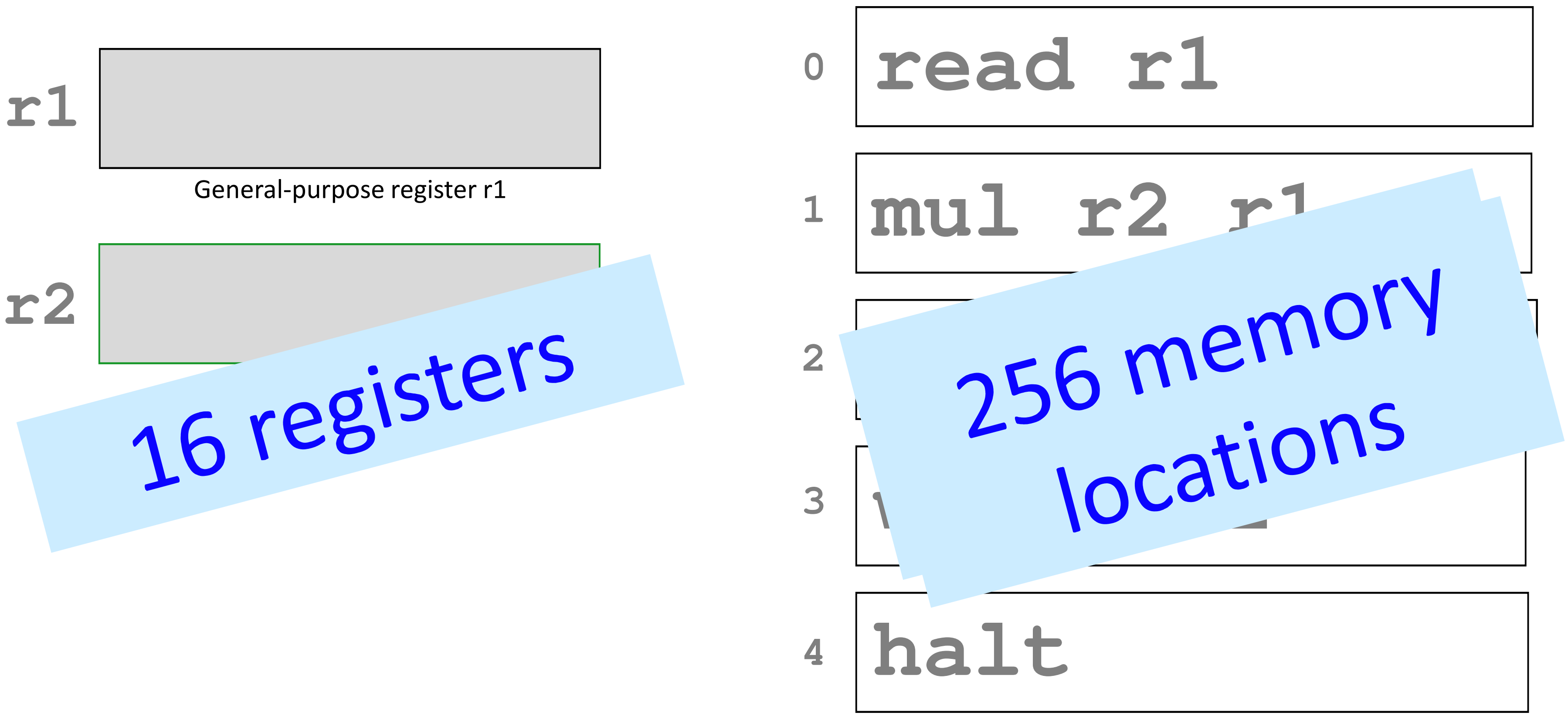
```
0  read r1
1  mul r2 r1 r1
2  add r2 r2 r1
3  write r2
4  halt
```

Hmmm: Harvey Mudd Miniature Machine

CPU ← Von Neumann bottleneck → RAM

central processing unit ***registers***

random access memory locations



r1

General-purpose register r1

r2

16 registers

0 `read r1`

1 `mul r2 r1`

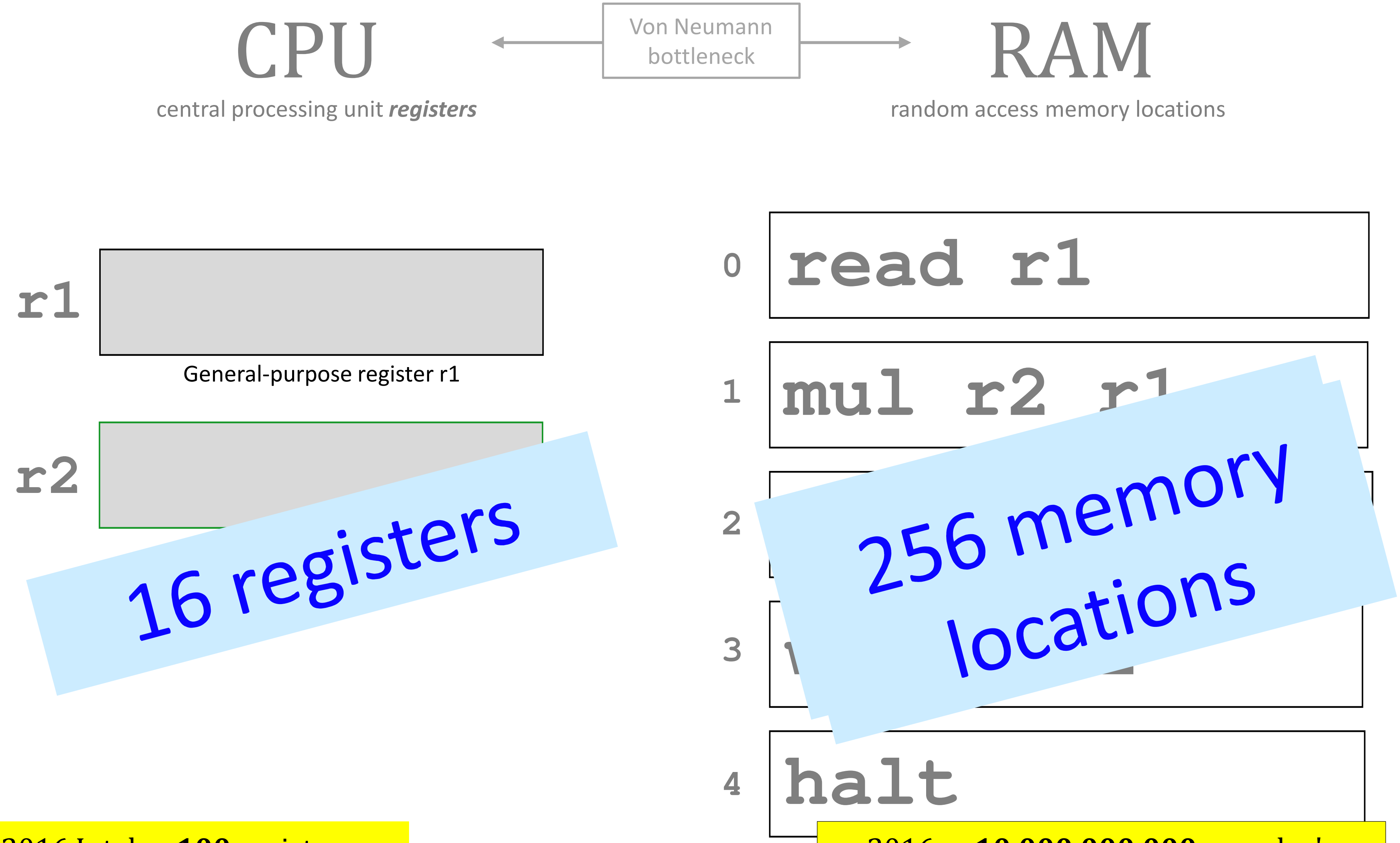
2

3

4 `halt`

256 memory locations

Hmmm vs 2016



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

2016 Intel: **~100** registers

2016: **~10,000,000,000** mem loc's

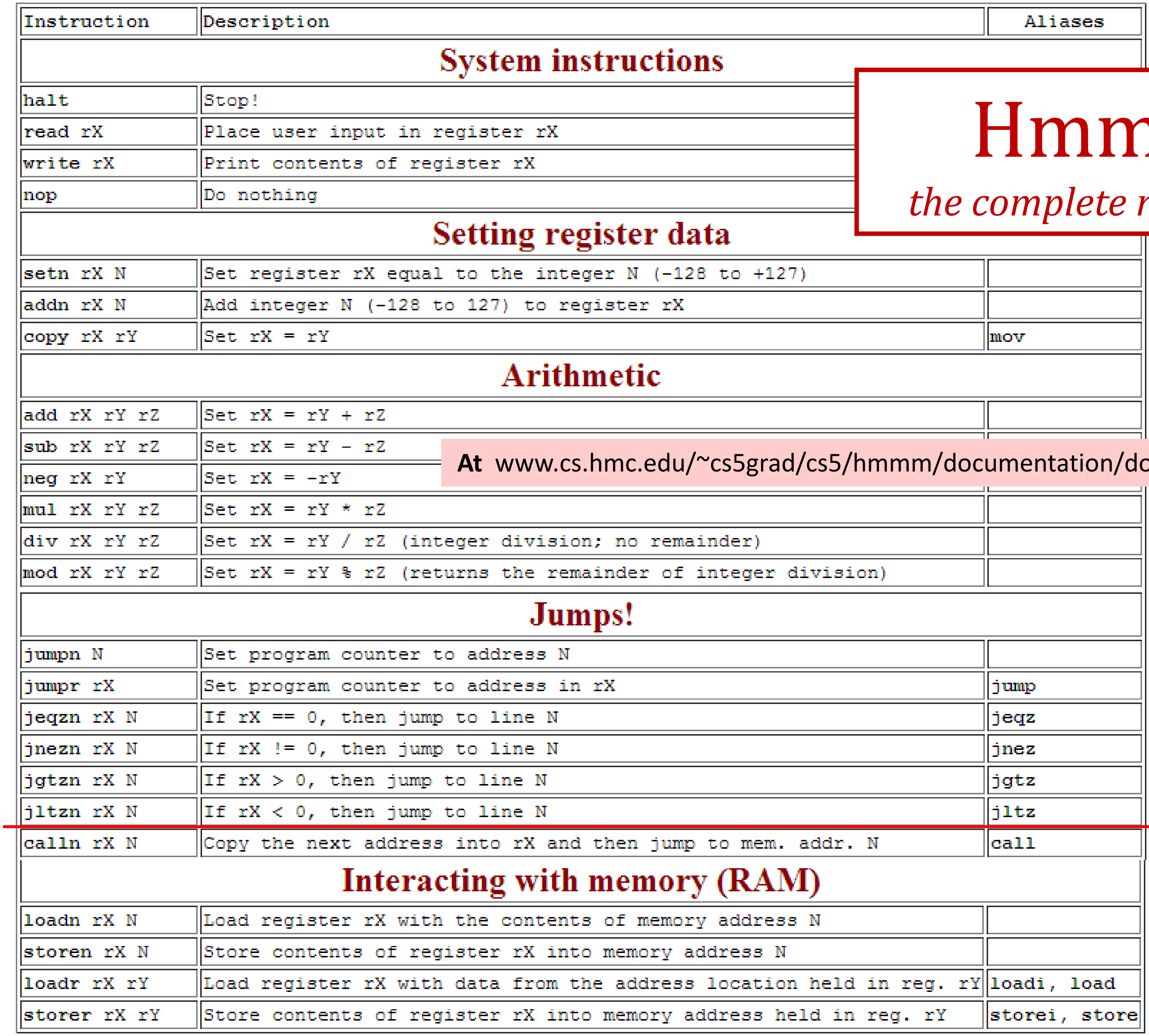
Demo

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

hw6's more detailed picture…

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)	
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

Thu.

Assembly Language

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

`read r1`	reads from keyboard into **`reg1`**	
`write r2`	outputs **`reg2`** 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 they're written R to L in Python!

Name(s) ____________________

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
2  mod r4 r1 r2
3  div r3 r1 r4
4  sub r3 r3 r2
5  addn r3 -1
6  write r3
7  halt
```

Extra! Here, the ***input*** r1 was 100. As a challenge, can you find any inputs (r1) that yield an ***output*** of 100? (They do exist!)

Try this on the back page first!

Quiz

screen

100 (input)

(output)

CPU

central processing unit

r1	100	General-purpose register r1
r2	7	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`	

Extra! Here, the ***input*** r1 was 100. Can you find any inputs (r1) that yield an ***output*** of 100? (They do exist!) [325, 544, + one more…]

Could you write a Hmmm program to compute

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

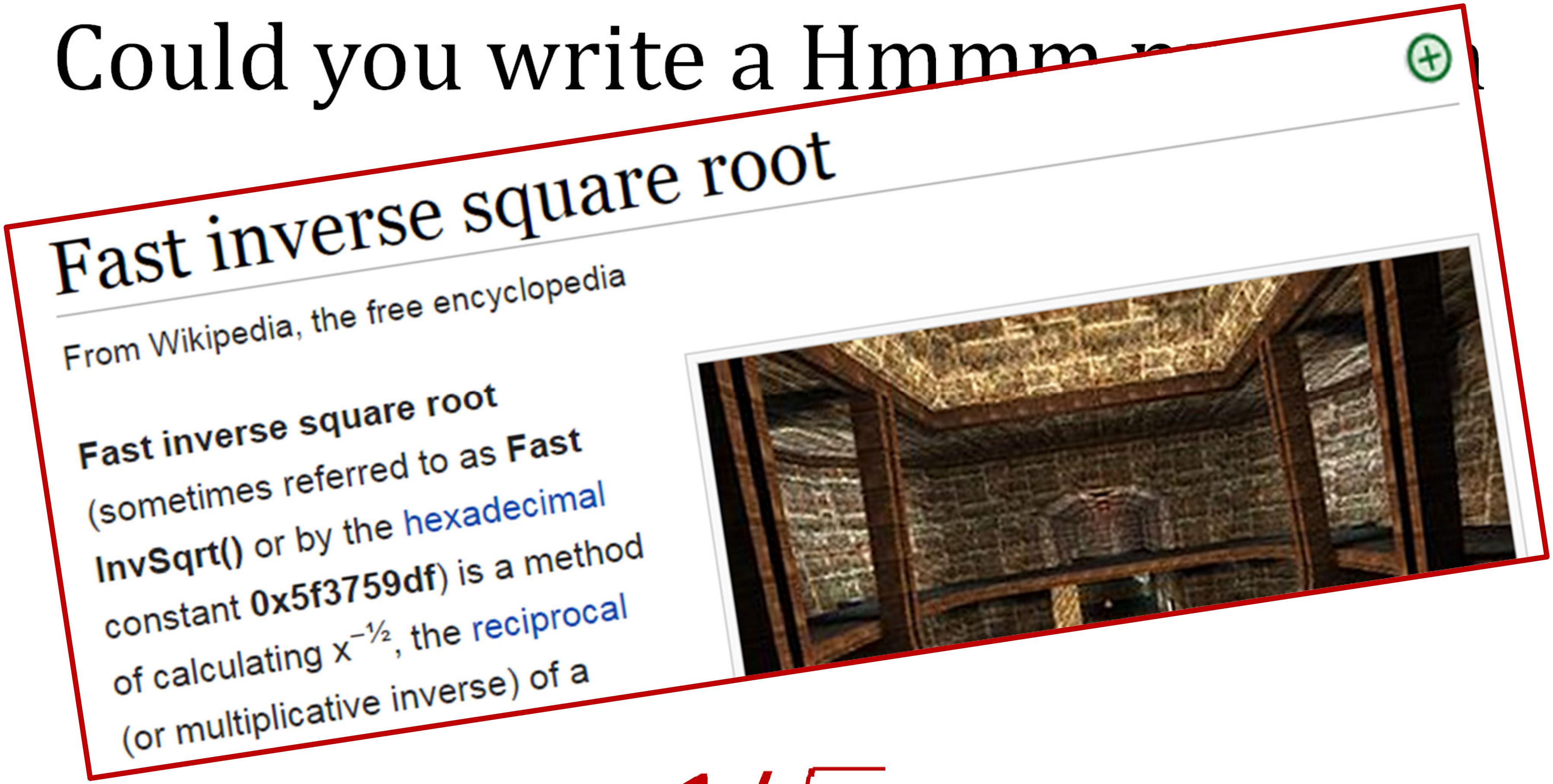
or

$$1/\sqrt{x}$$

?

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 ***Python*** program

to write a Hmmm program

to compute

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

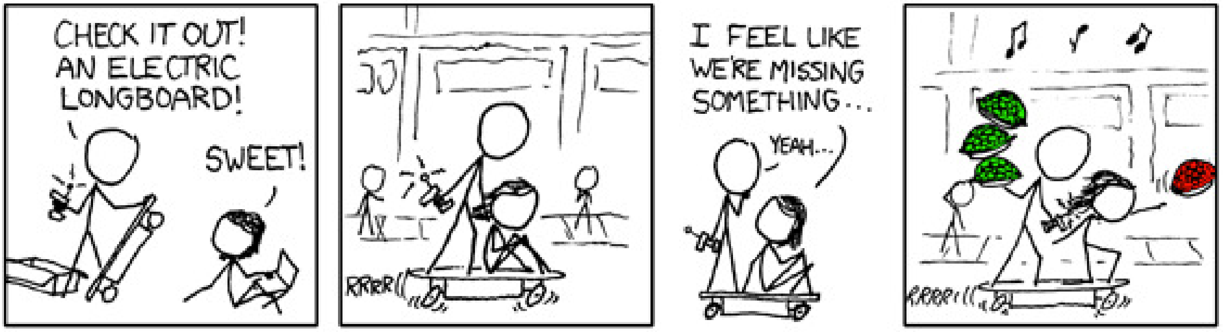
or

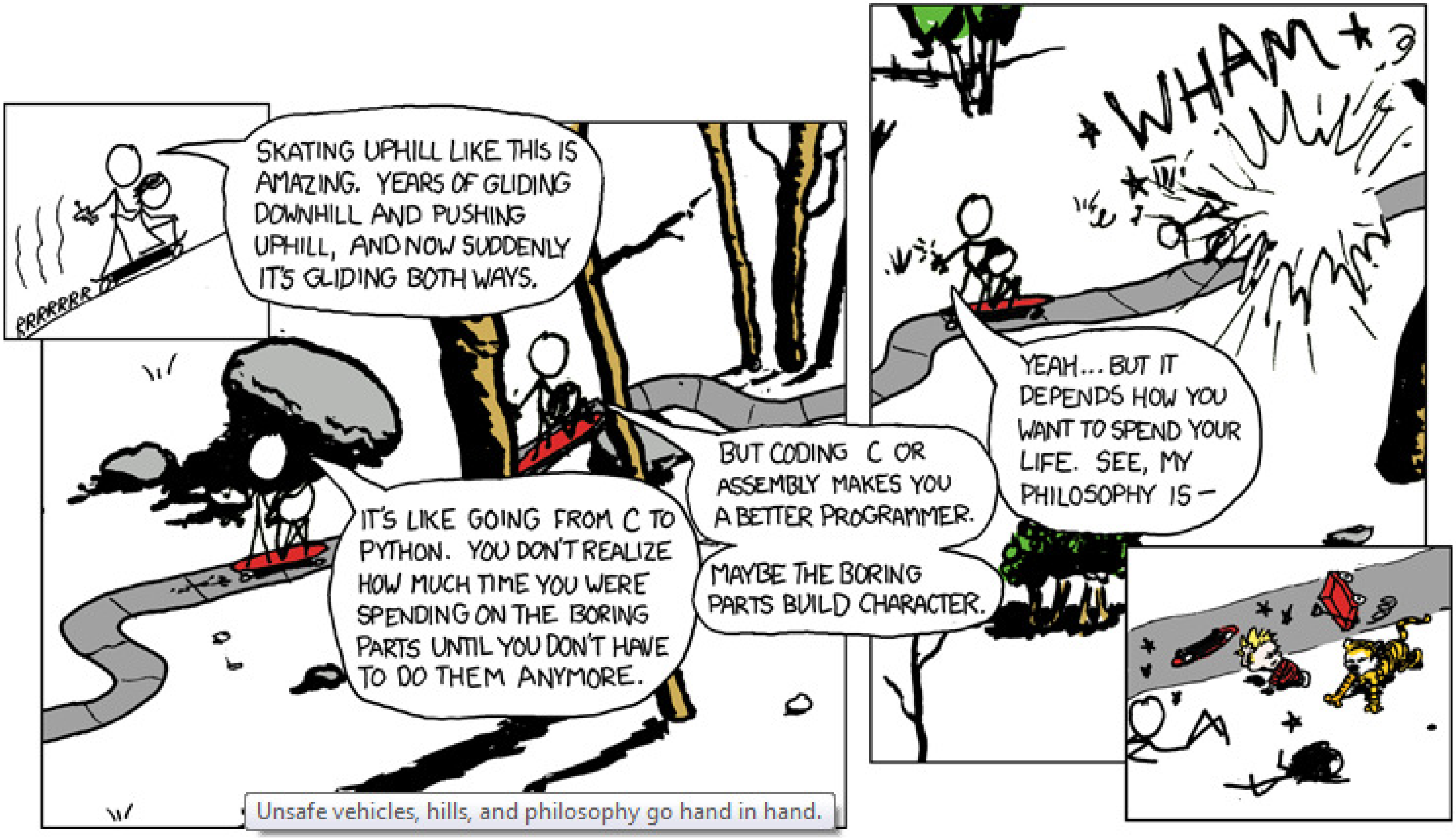
$$1/\sqrt{x}$$

?

much better!

Why Assembly?

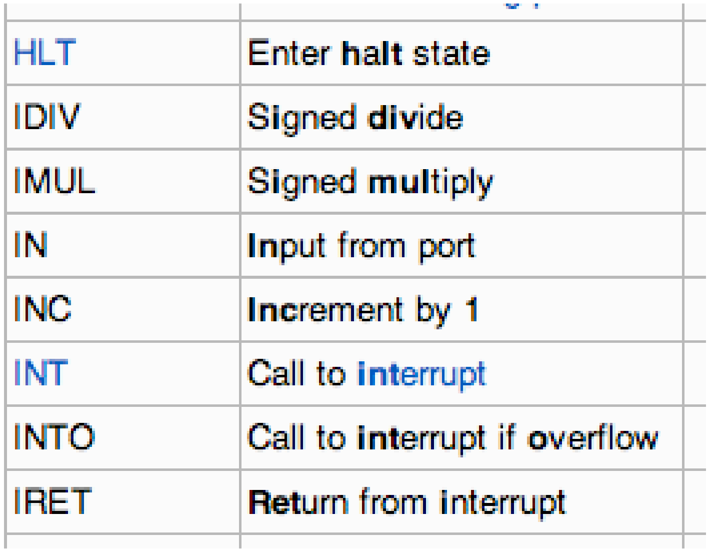




Unsafe vehicles, hills, and philosophy go hand in hand.

Real Assembly Languages

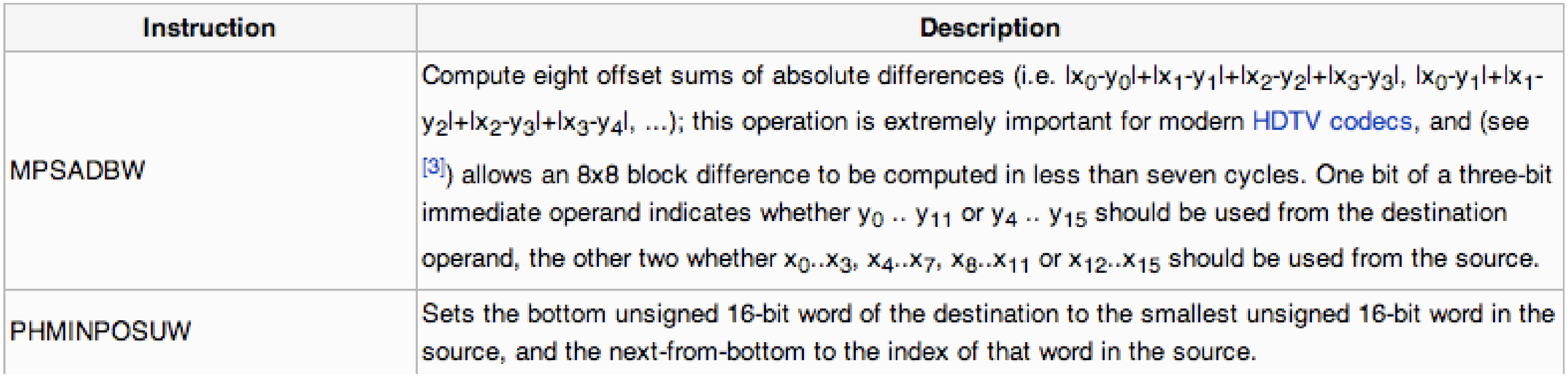
Hmmm is a subset common to ***all*** real assembly languages.



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. $\|x_0-y_0\|+\|x_1-y_1\|+\|x_2-y_2\|+\|x_3-y_3\|$, $\|x_0-y_1\|+\|x_1-y_2\|+\|x_2-y_3\|+\|x_3-y_4\|$, ...); 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.

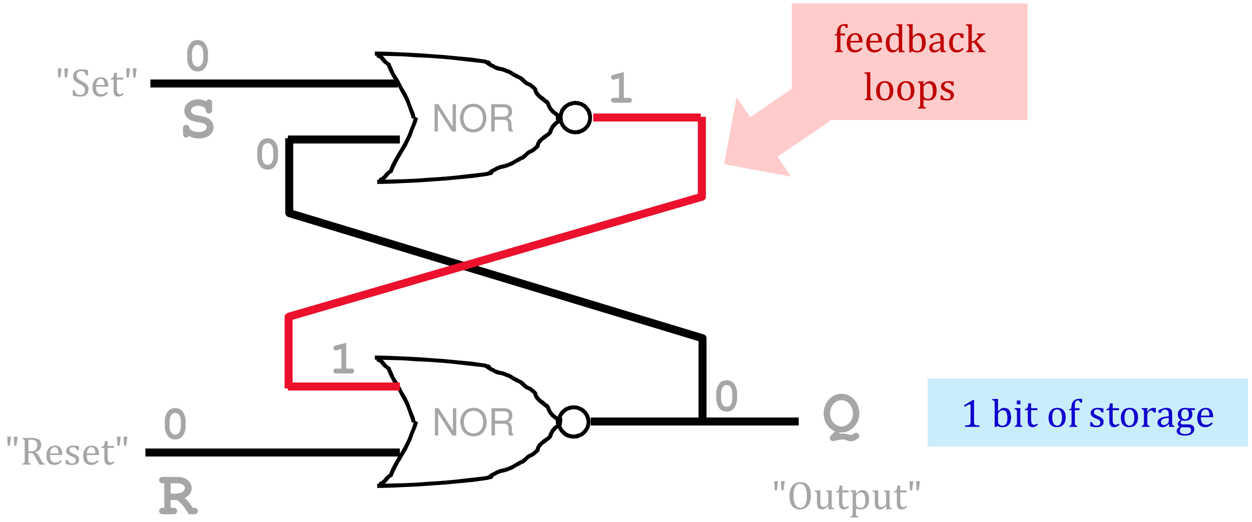
Is this enough?

What's missing?

```
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 our Hmmm assembly language up to this point?

For systems, a face-lift is to add an edge that *creates a cycle*, not just an additional node.



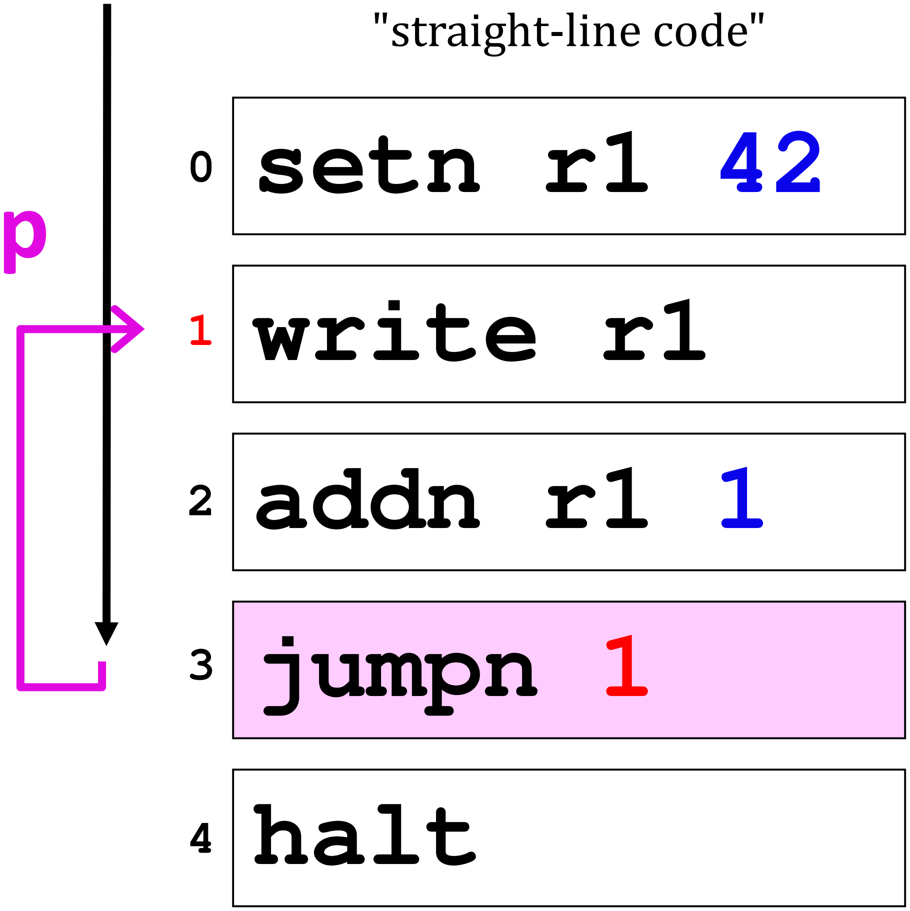
feedback loops

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
```

CPU

central processing unit

r1

General-purpose register r1

r2

not used in this program...

General-purpose register r2

Screen

i
v
jumpn!

RAM

random access memory

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

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

- we replace line 3's `1` with a `0`?
- we replace line 3's `1` with a `2`?
- we replace line 3's `1` with a `3`?
- we replace line 3's `1` with a `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**

Unconditional jump

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

Indirect jump

`jumpr r1`	Jump to the line# ***stored*** in `r1`

Jumps in Hmmm

Conditional jumps

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

`jgtzn` if greater than 0... THEN jump to line number **42**

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

`jnezn` if not equal to 0... THEN jump to line number **42**

Unconditional jump

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

This is making me jumpy!

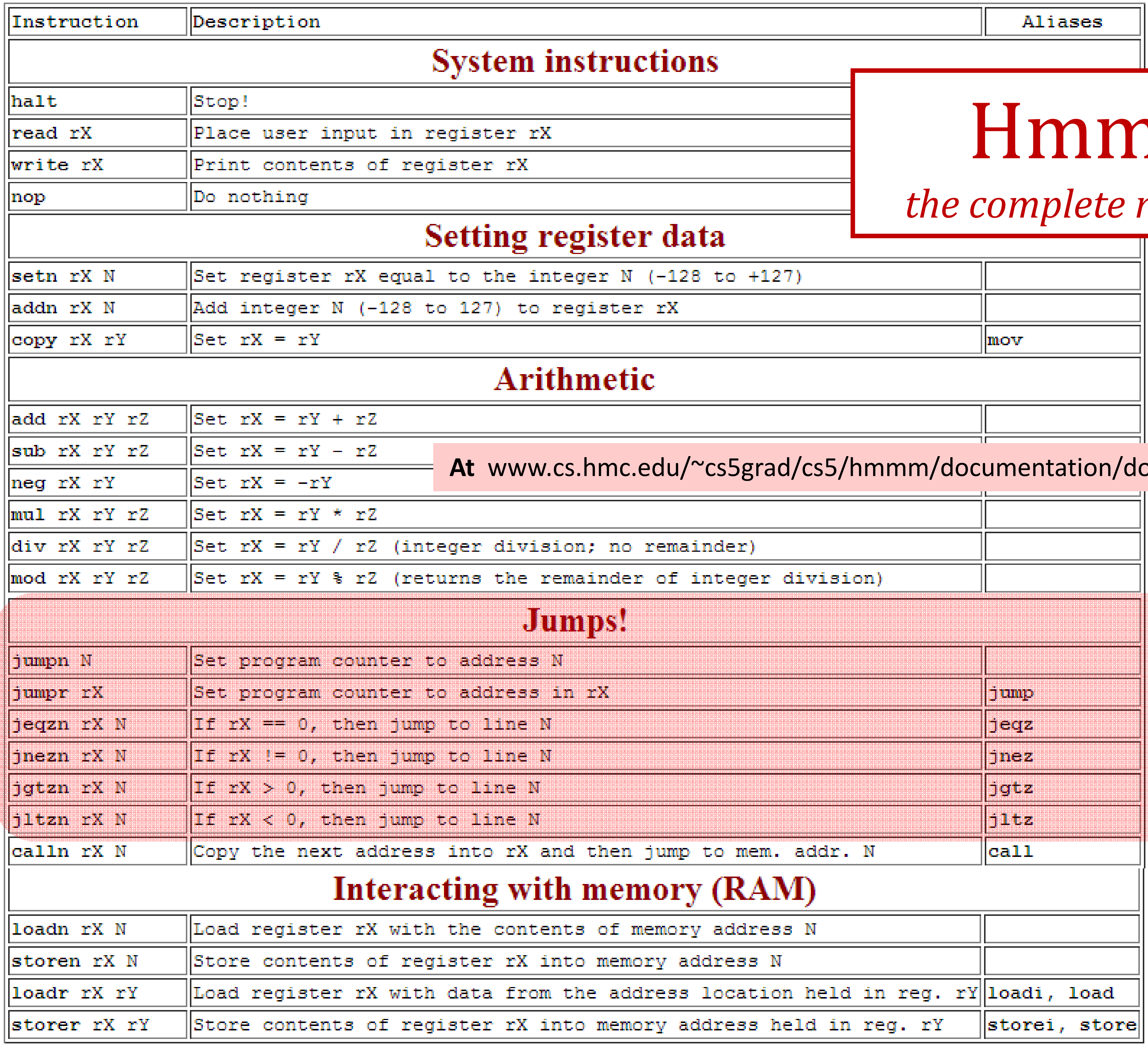
Indirect jump

`jumpr r1` Jump to the line# ***stored*** in `r1`

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)		
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!

`jgtzn`

Gesundheit!

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 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	

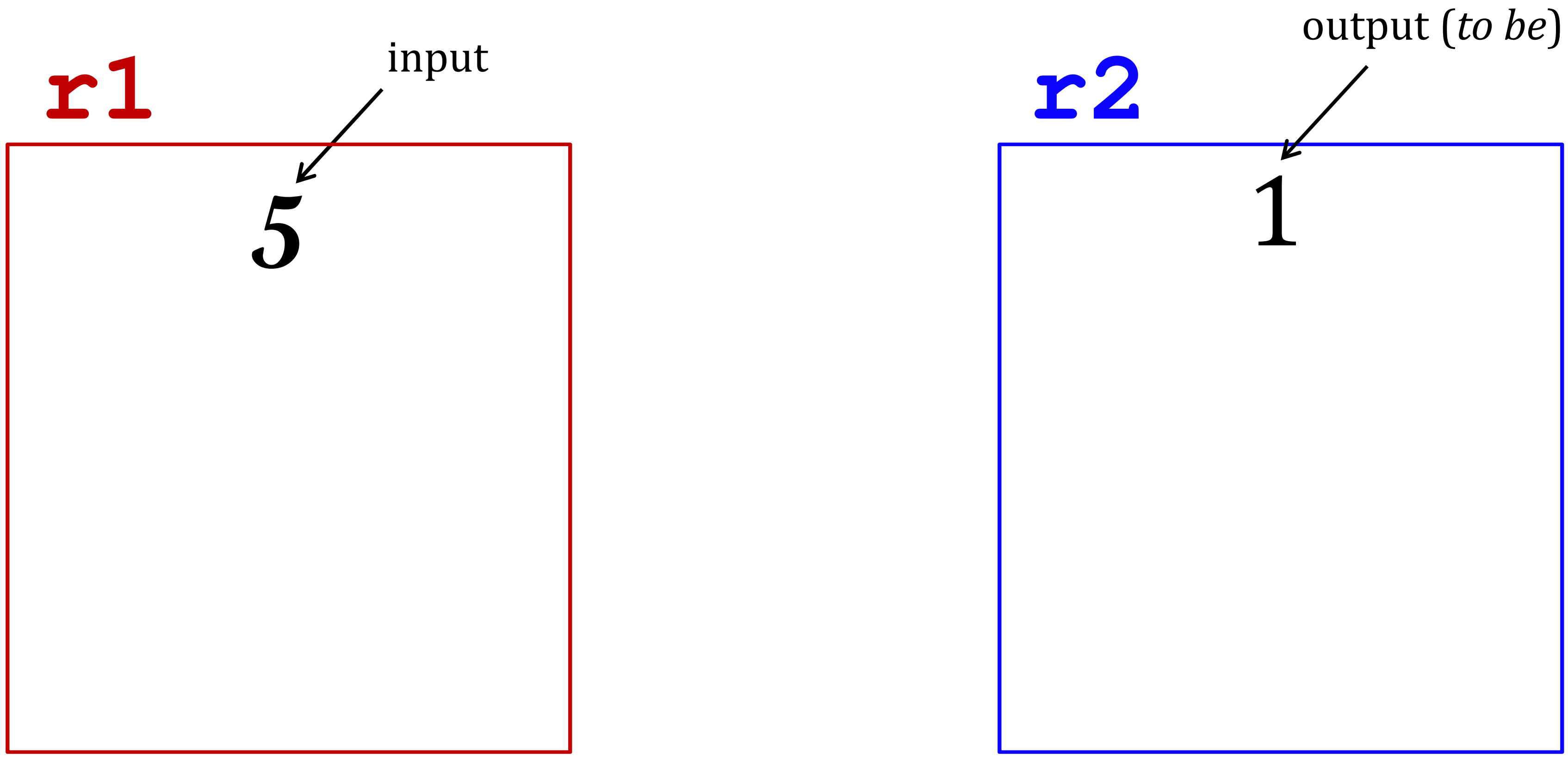
Memory - RAM

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* ...



let r1 be the input
and the "counter"

let r2 ***become***
the output

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	

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
9
```

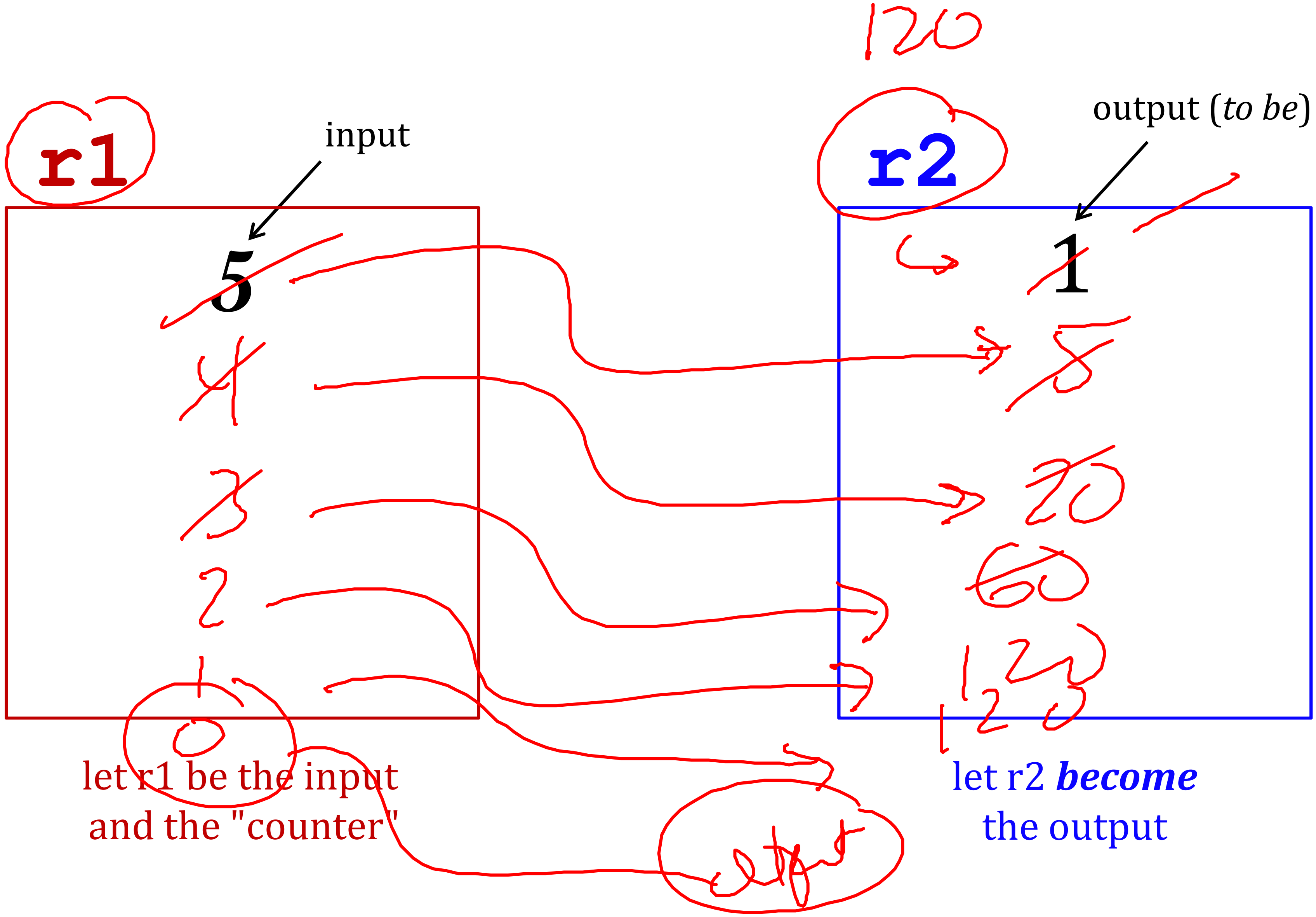
Run #2

r1	*5*
r2	*42*
r3	

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

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

factorial: the *plan* ...



let r1 be the input
and the "counter"

let r2 ***become***
the output

one factorial code

Registers - CPU

r1 input

r2 result – so far

r3 not needed, but OK to use!

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:

Randohmmm Numbers...

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

See you there!