CS 42 Today: Beyond Binary

Midterm Logistics

- First Midterm: Next week.
- Review session on Thursday.

- A6 due Monday night

  Extra credit accepted w/o euro through Friday

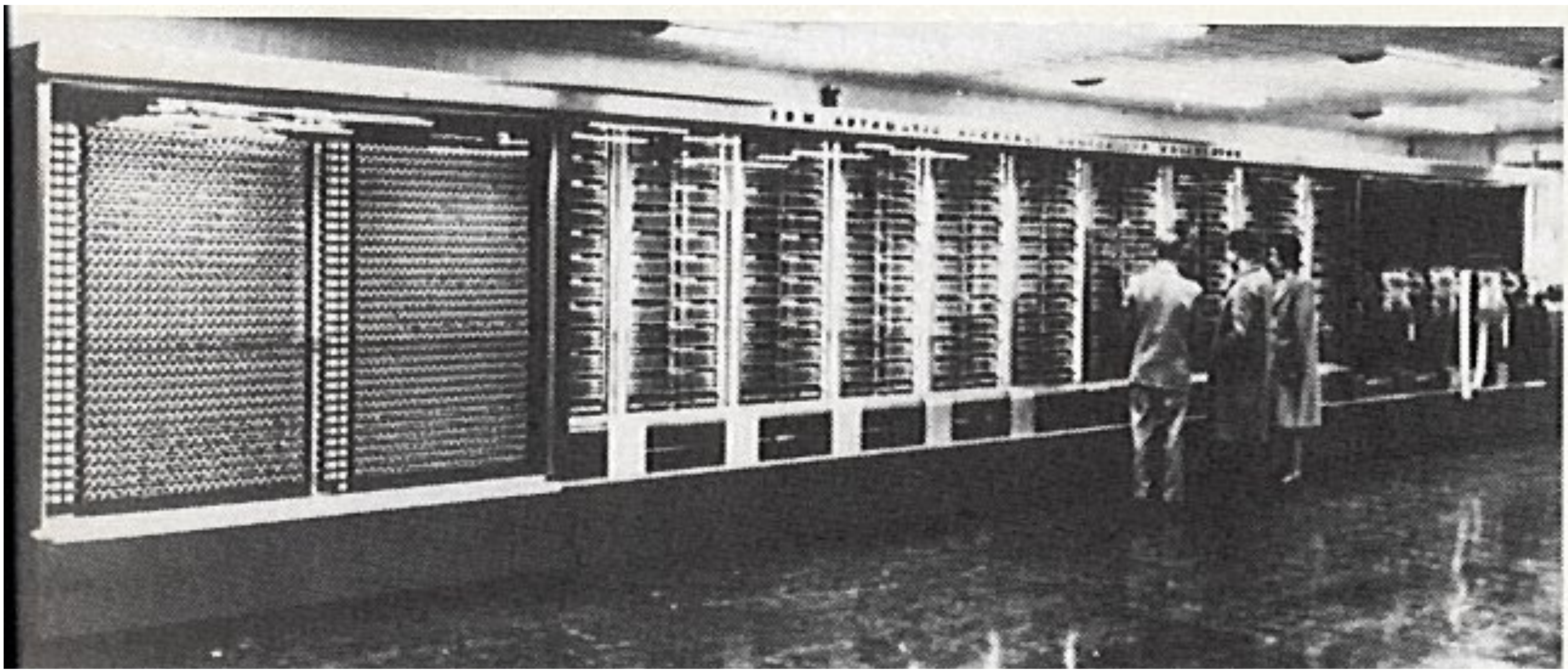
  E-mail me if you turn it in "late"

- We have to make a choice:
  - 75-minute in-class exam Tuesday
  - Or, longer (2-hour) take-home exam, taken sometime between Friday 5pm and Wednesday 9am.

The Mark 1

Howard Aiken, Grace Hopper at Harvard

relay-based computer



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

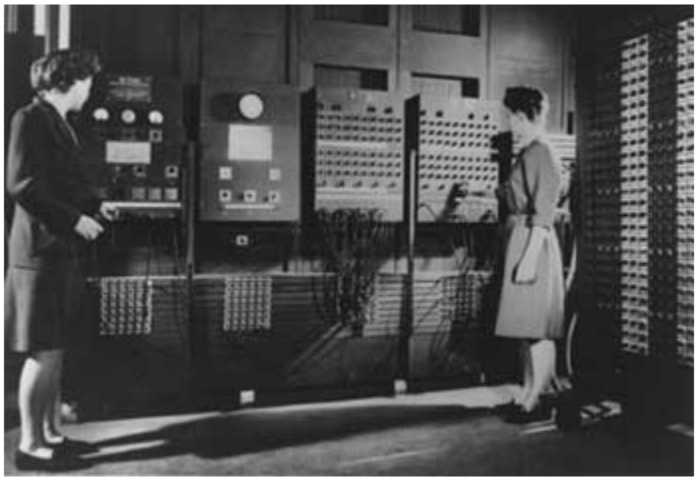
Pentium 0.00001 MHz

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

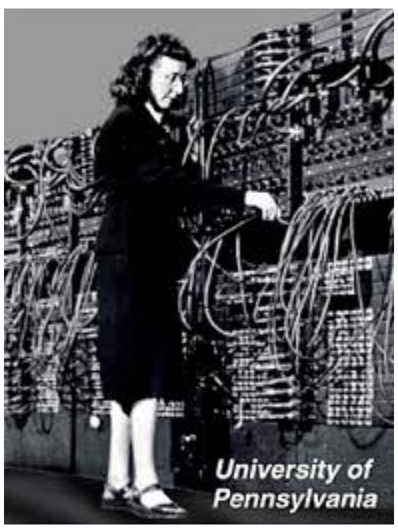
5 tons, 530 miles of wiring
765,299 distinct parts!

The Eniac, 1946

http://bit-player.org/wp-content/eniac.jpg



http://www.plyojump.com/classes/images/computer_history/eniac_with_programmer2.jpg



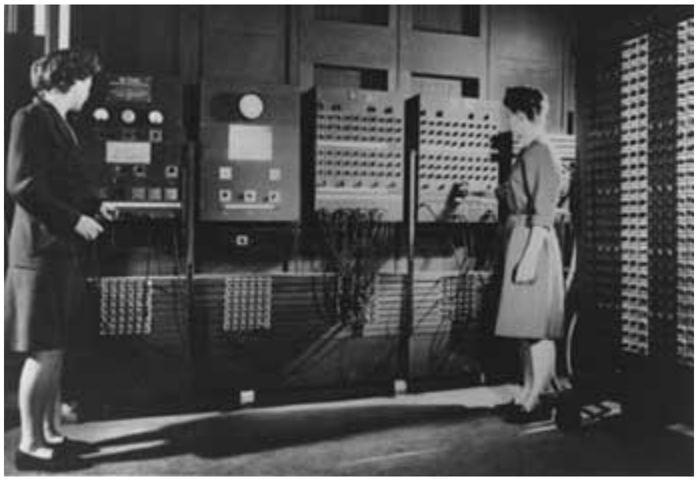
Eniac's developers

J. Prespert Eckert

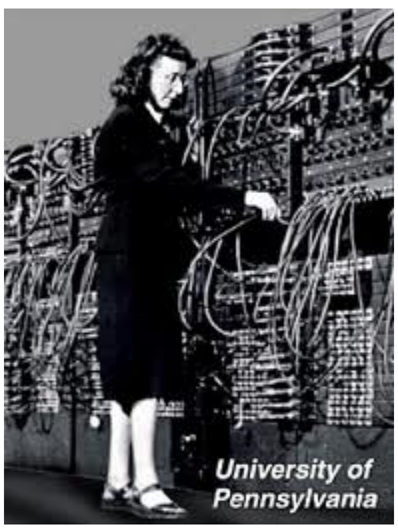
John W. Mauchly

The Eniac, 1946

http://bit-player.org/wp-content/eniac.jpg

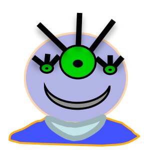


http://www.plyojump.com/classes/images/computer_history/eniac_with_programmer2.jpg

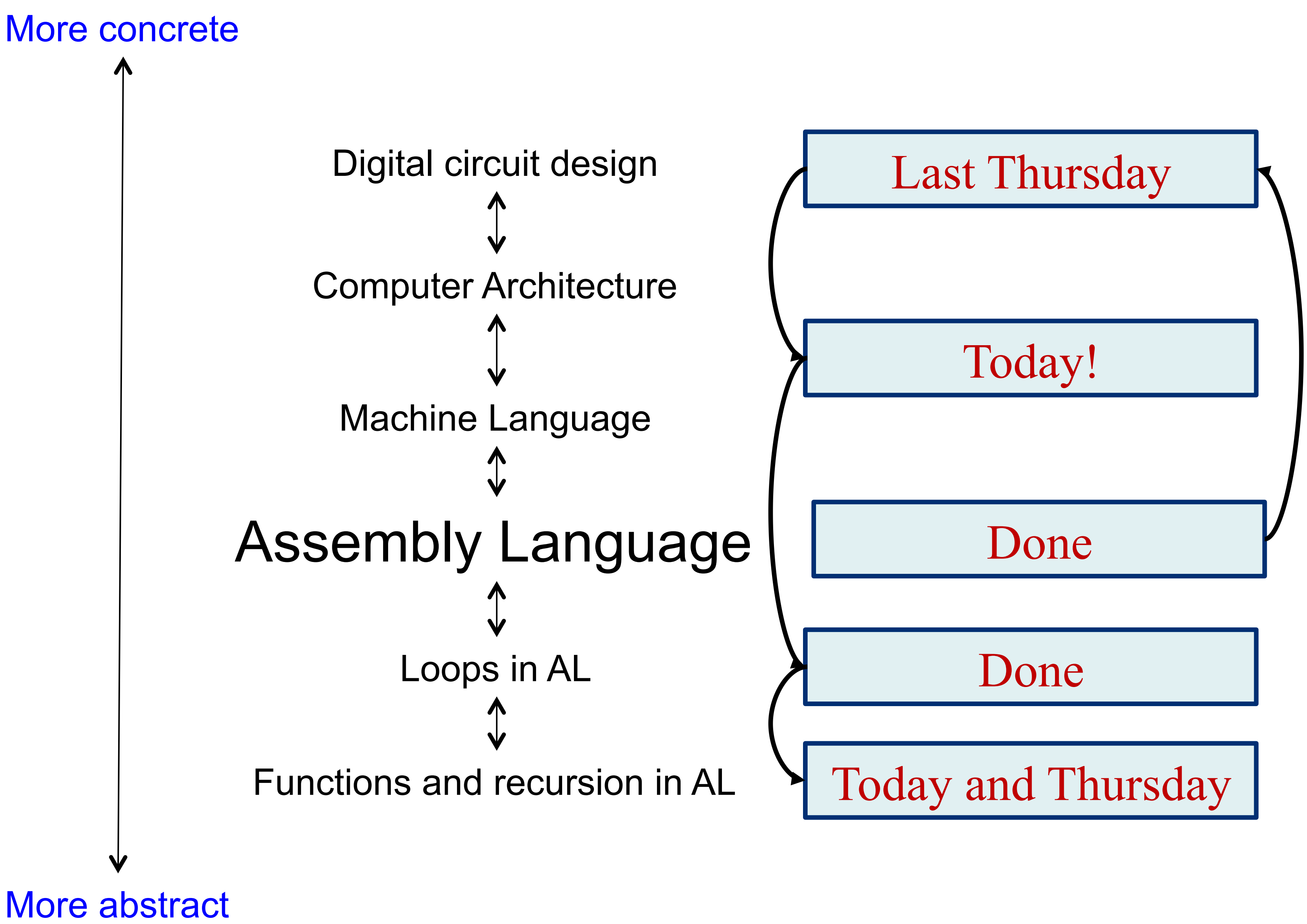


Eniac's Programmers!

Rear left to right: Kathy Kleiman (author), Jean Bartik, Marlyn Meltzer, Kay Mauchly Antonelli *Front:* Betty Holberton. *Not pictured:* Ruth Lichterman Teitelbaum and Frances Bilas Spence

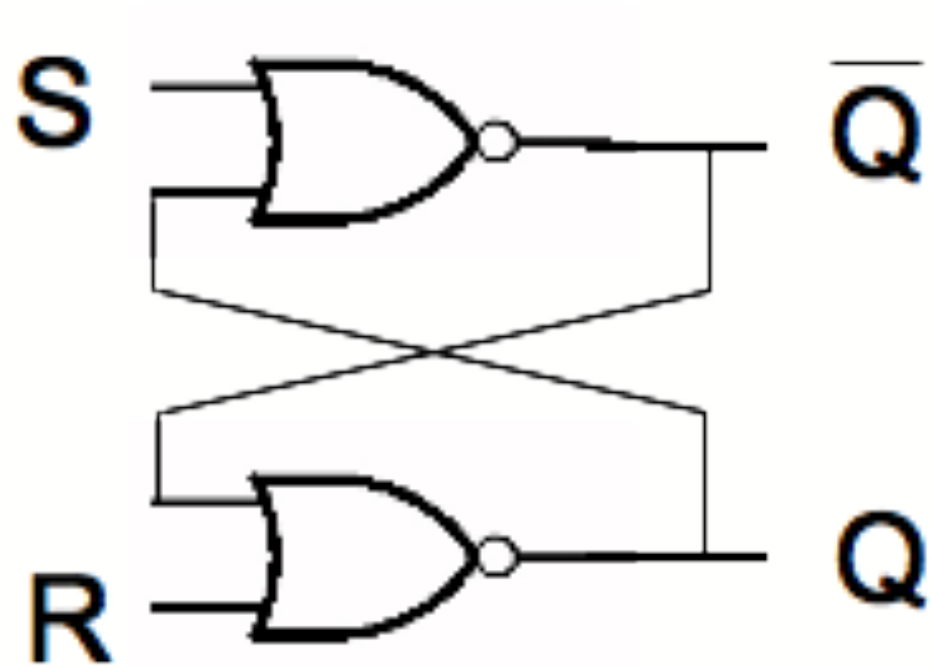


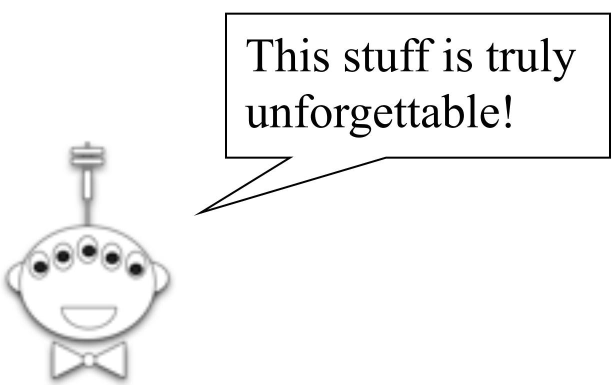
And I thought programming in Hmmm was painful!

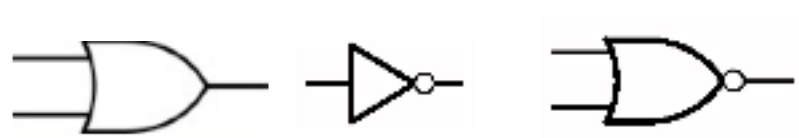


More concrete
Digital circuit design
Computer Architecture
Machine Language
Assembly Language
Loops in AL
Functions and recursion in AL
More abstract
Last Thursday
Today!
Done
Done
Today and Thursday

A 1-bit Memory



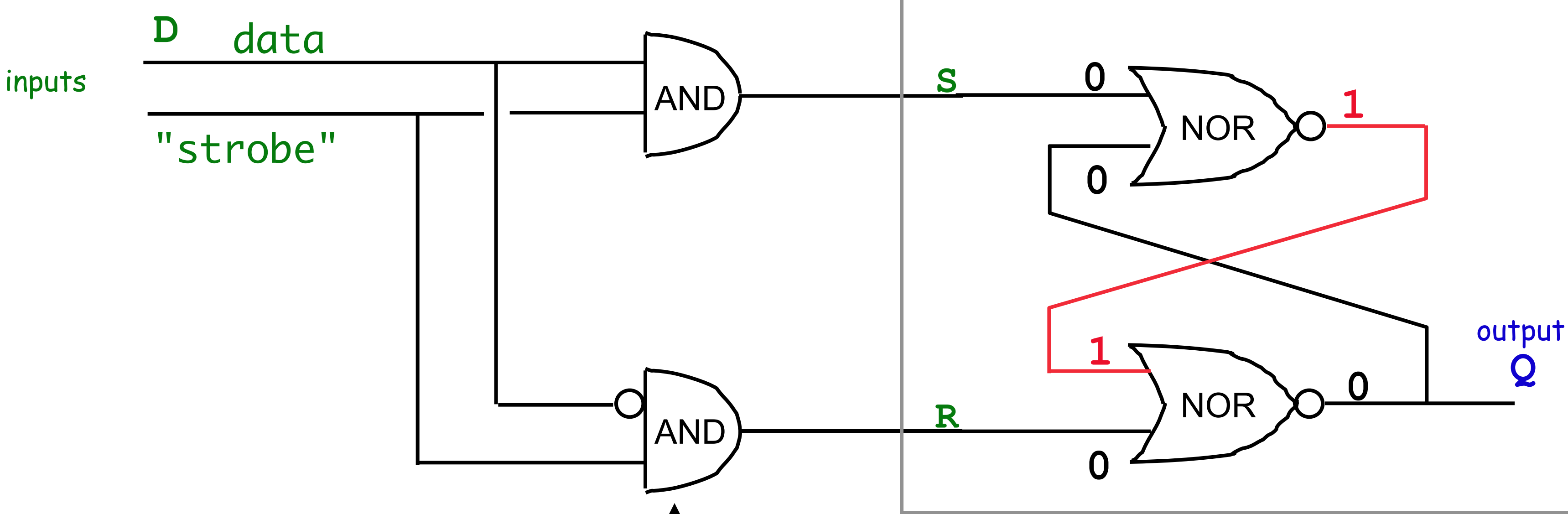




OR + NOT = NOR

The D latch

The SR latch



These ANDs ensure

if D == 1, S is 1

if D == 0, R is 1

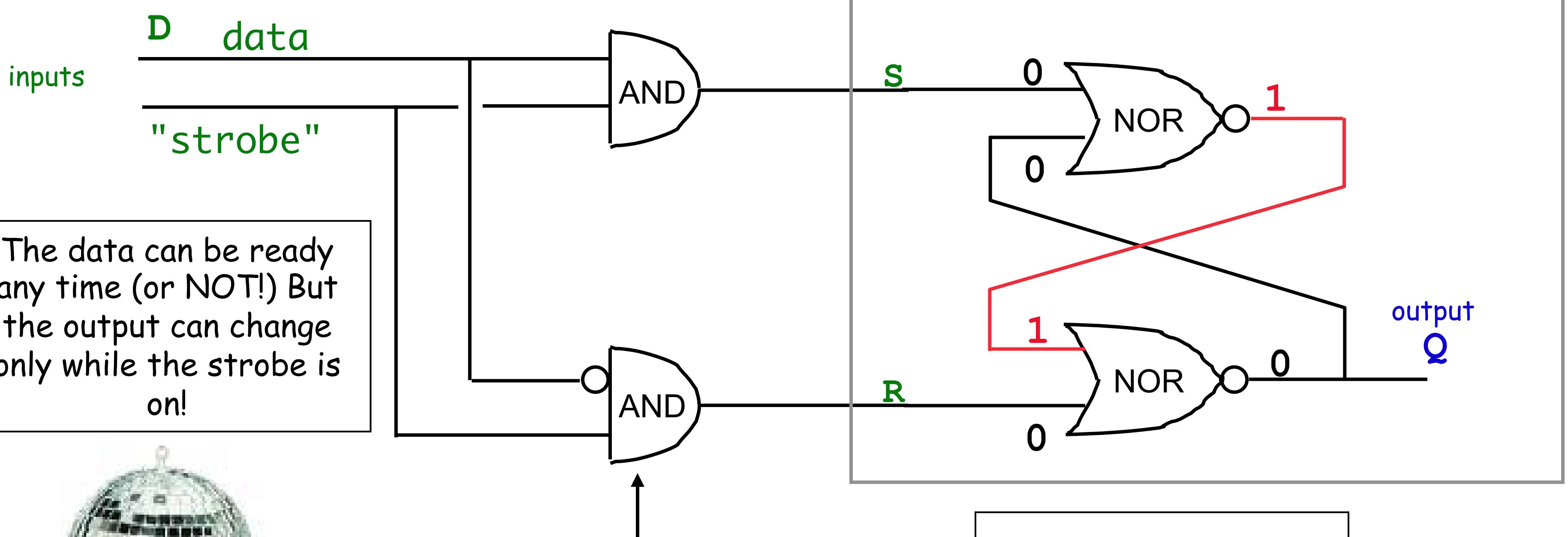
S sets Q to 1.

R resets Q to 0.

1 bit of memory!

The D latch

The SR latch



The data can be ready any time (or NOT!) But the output can change only while the strobe is on!

These ANDs ensure

if D == 1, S is 1

if D == 0, R is 1

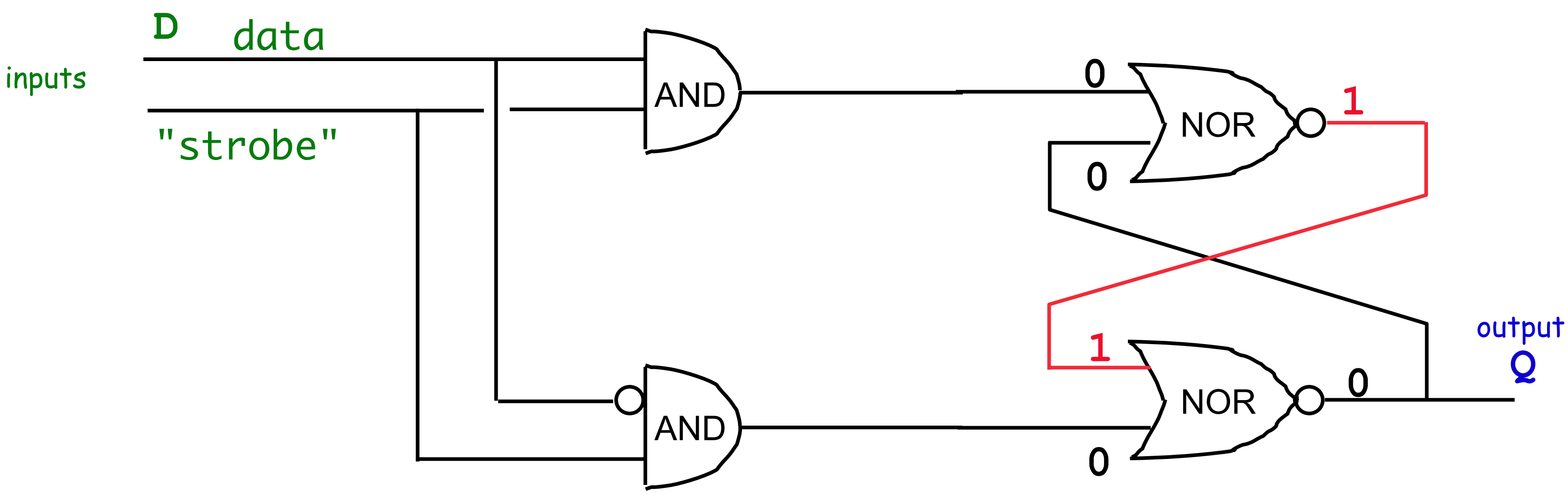
S sets Q to 1.

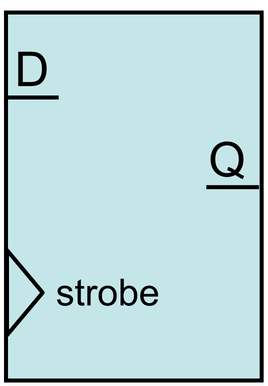
R resets Q to 0.

1 bit of memory!

The D latch

The SR latch



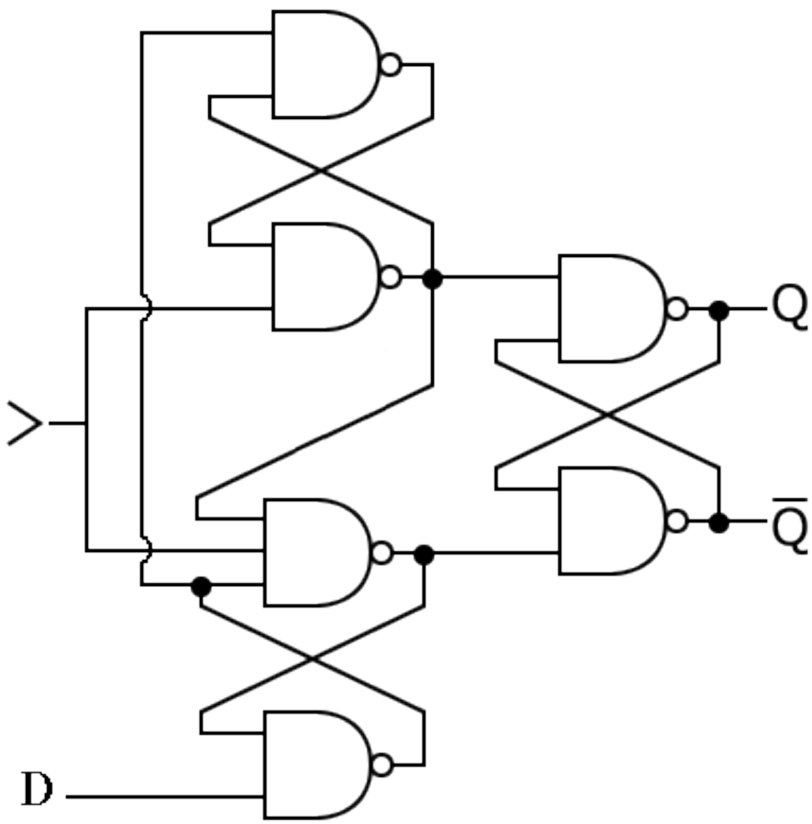


its circuit element

1 bit of memory!

Slightly Fancier Memory Elements

“Positive-edge-triggered D flip-flop (http://en.wikipedia.org/wiki/Flip-flop_(electronics))”



"flip-flop"

Random Access Memory "on-chip"

"640K ought to be enough for anybody"

- not Bill Gates!

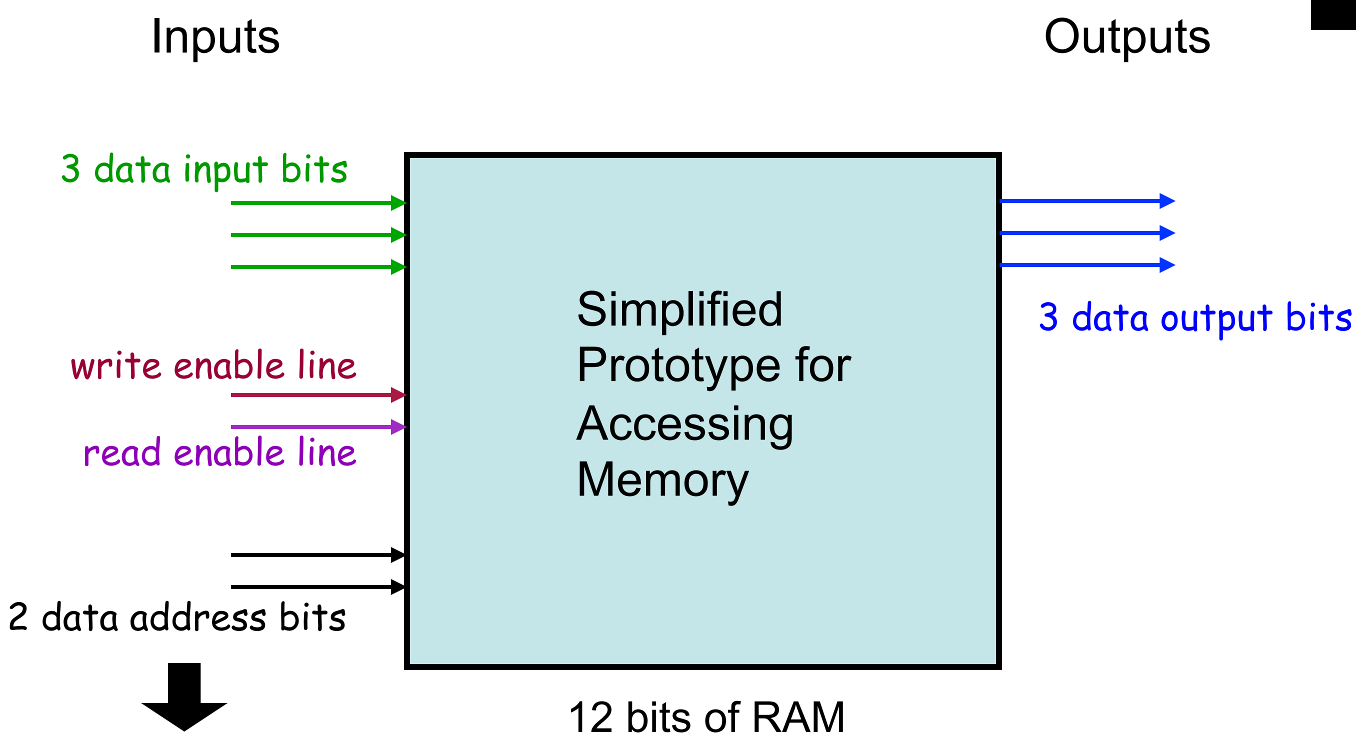
RAM!

A 512KB RAM
(About 4.2 million bits)

A 1GB RAM
(About 8.9 billion bits)

Random Access Memory "on-chip"

We can use data latches to create a 12 nG bit RAM !

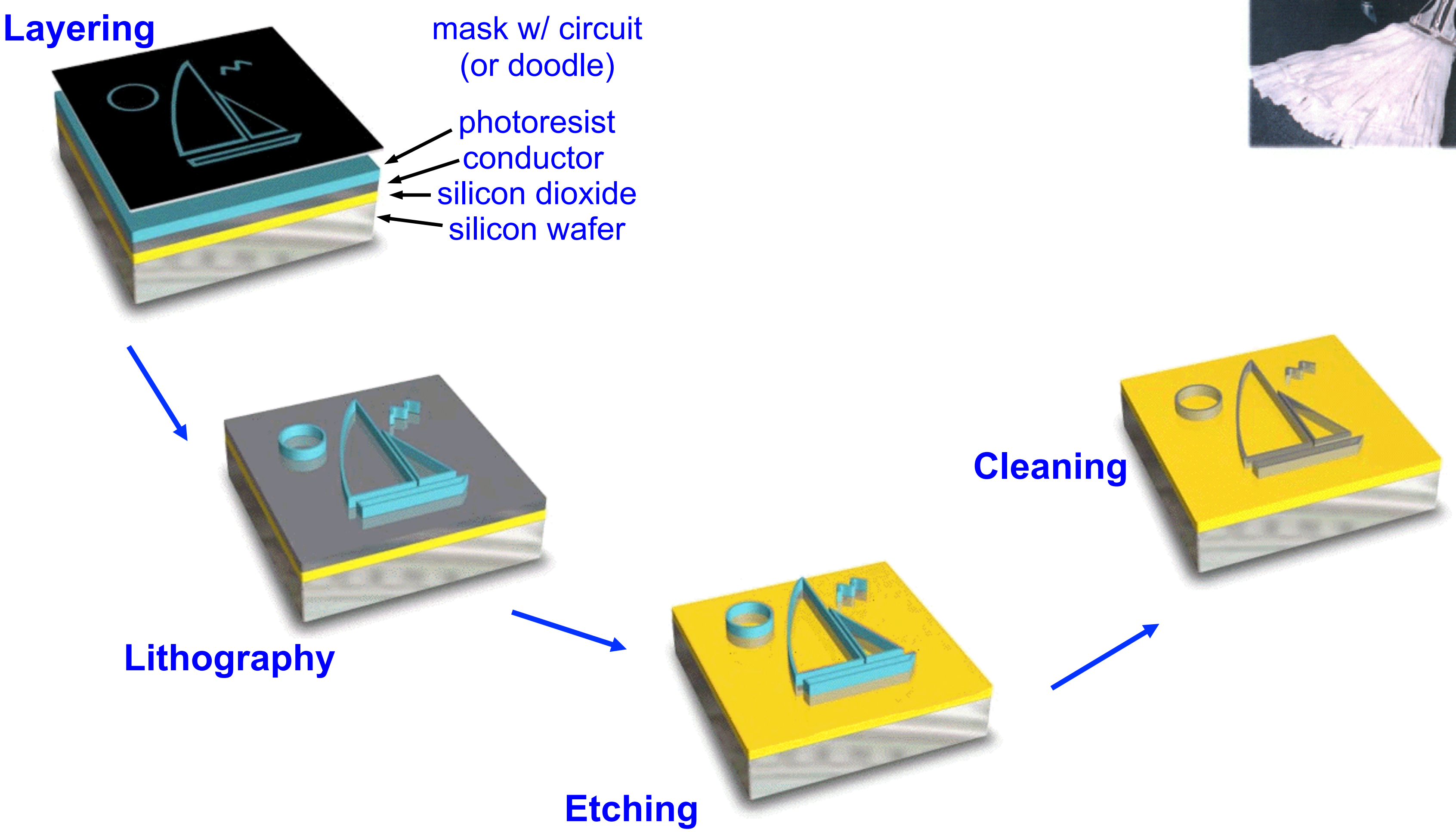


3 bits stored at location 00
3 bits stored at location 01
3 bits stored at location 10
3 bits stored at location 11

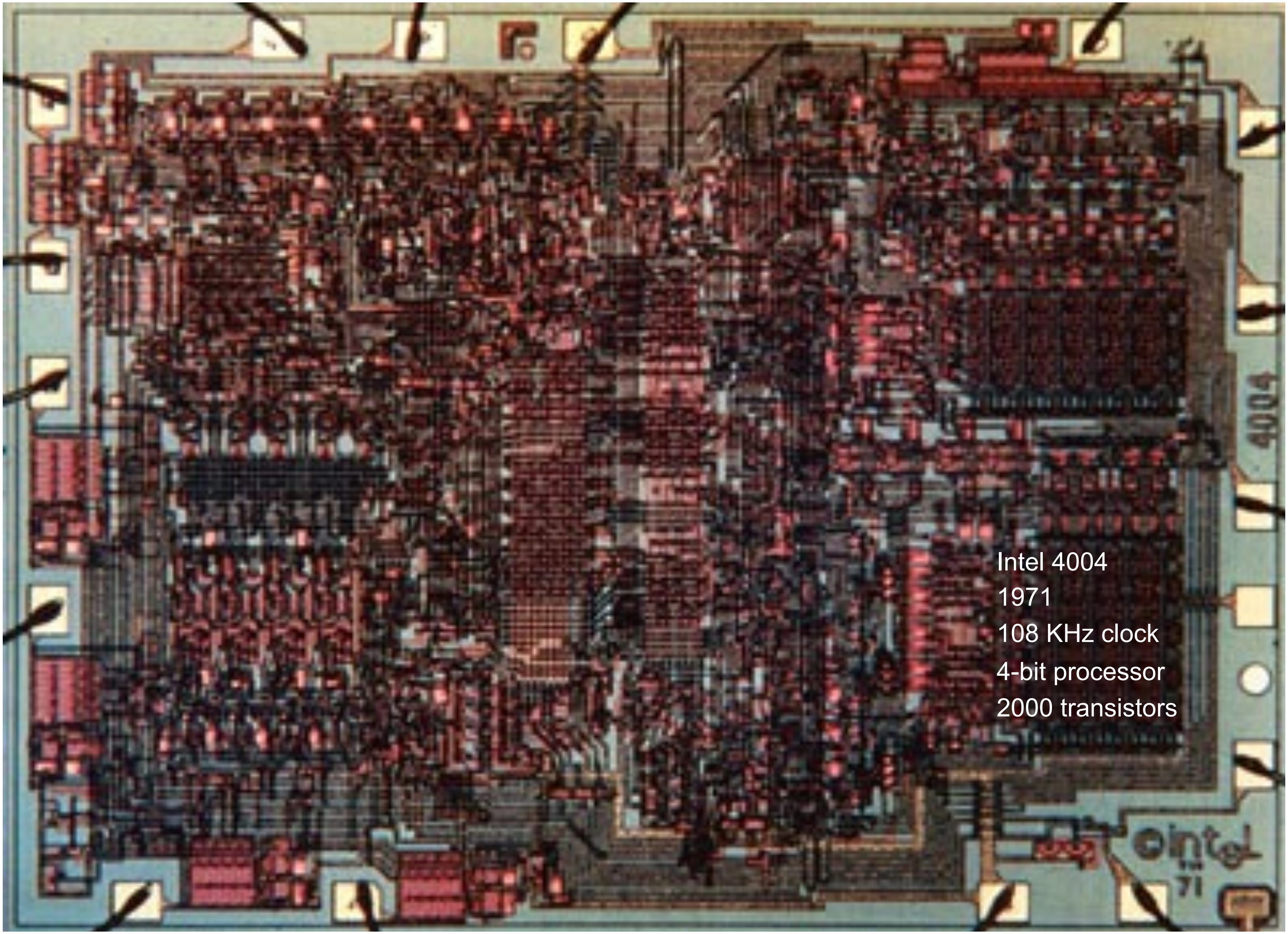
It's totally FAB

http://www.techfak.uni-kiel.de/matwis/amat/elmat_en/makeindex.html

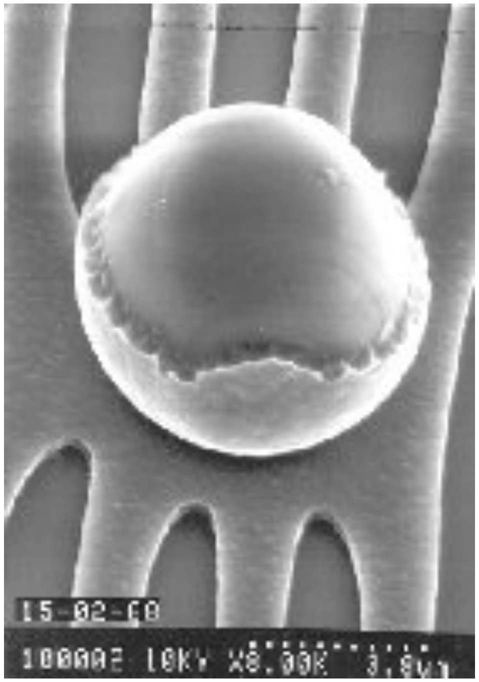
How are all of these gates actually realized?



x 200 or so


4004
intel
71
Intel 4004
1971
108 KHz clock
4-bit processor
2000 transistors

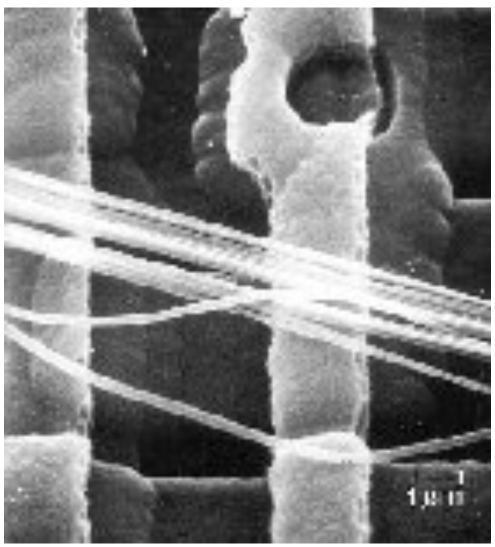
2000
Intel Pentium P4
1.5 GHz clock
32-bit processor
42 million transistors



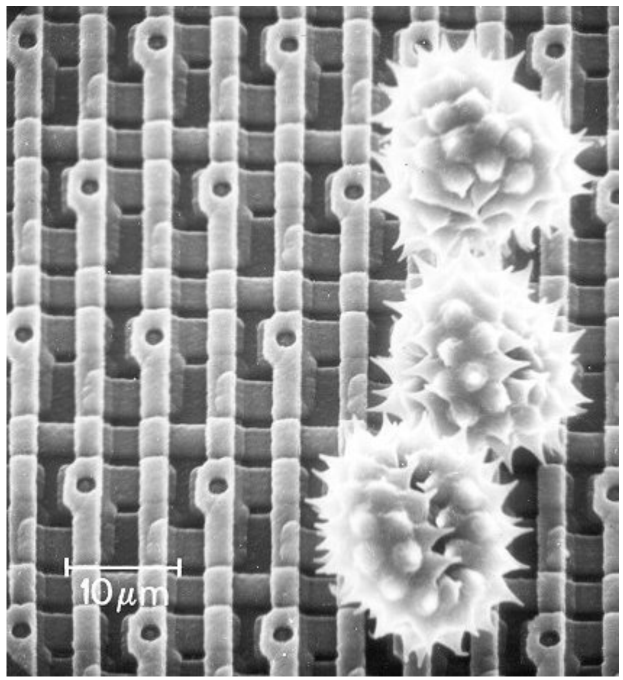
Ouch!

The enemy ...

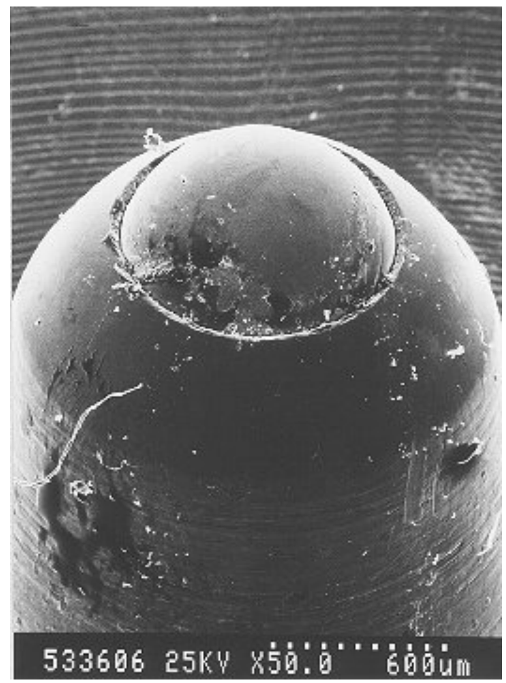
Keeping particles out of the fabrication process is paramount!



Nature still impresses...

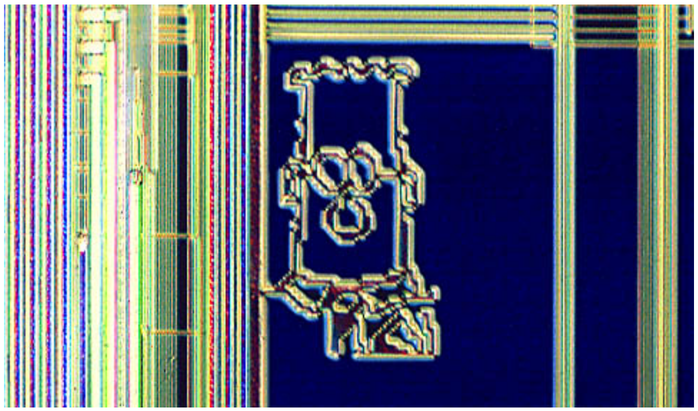


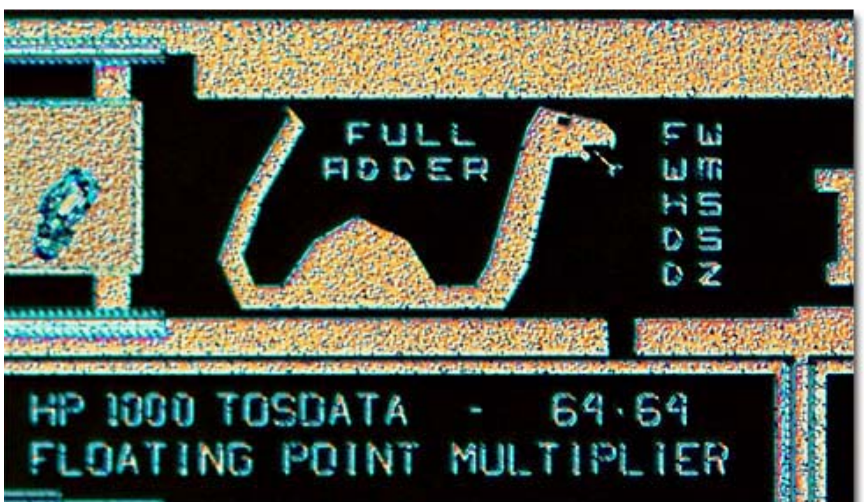
Pollen on a memory chip



A ruthless invader...

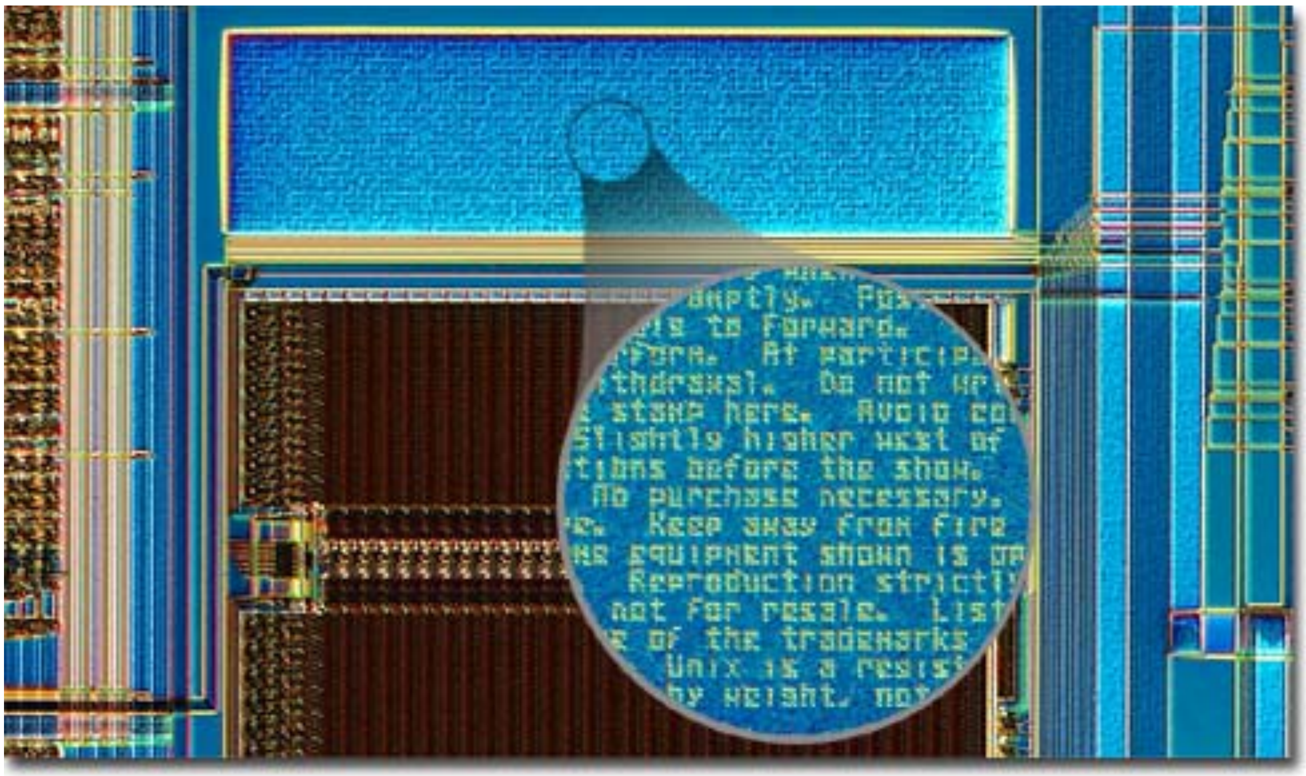
What to do with all that extra silicon?





The "silicon zoo": micro.magnet.fsu.edu/creatures/index.html

What to do with all that extra silicon?

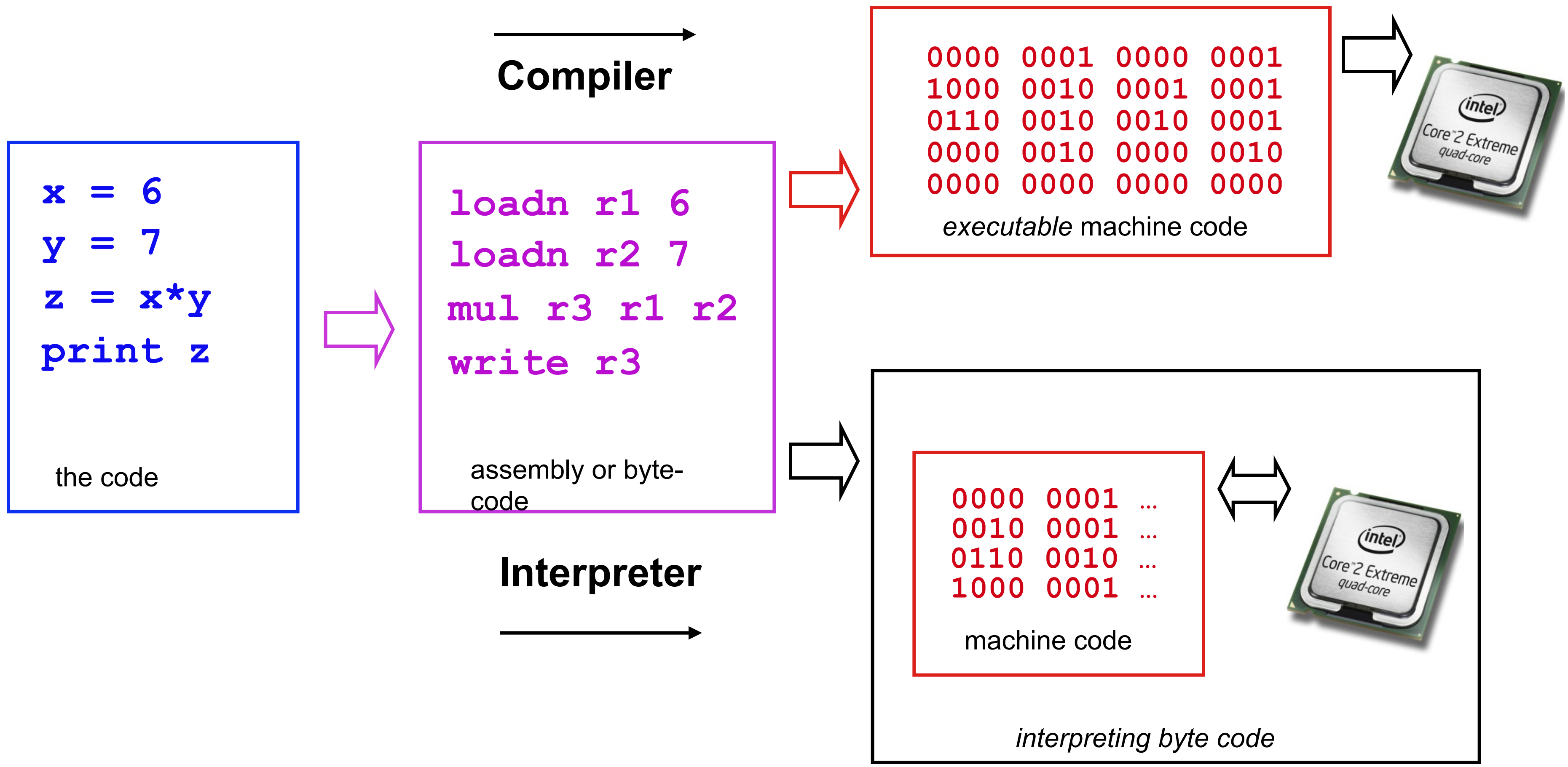


The "silicon zoo": micro.magnet.fsu.edu/creatures/index.html

Assembly Language	Machine Language
loadn r0 42	0011000000101010
add r2 r1 r0	0111001000010000
sub r5 r4 r3	1000010101000011
mul r11 r9 r8	1001101110011000
div r6 r7 r12	1010011001111100
halt	0000000000000000
register-level programming	*bit-level programming*

the compiler

a program that translates from human-usable language into assembly language and machine langauge



What's in an instruction

One possible (simplified, not Hmmm!) instruction set architecture

$\underbrace{00}_{\text{which instruction?}}\underbrace{011}_{\text{Source register 1 AND destination register}}\underbrace{100}_{\text{Source register 2}}$

which instruction?

00	add
01	subtract
10	multiply
11	divide

Source register 1
AND destination register

Source register 2

"Add the value in register 3 to the value in register 4 and put the result back in register 3"

Instructions are just 1s and 0s… it is up to the computer architect to give the 1s and 0s meaning.

What's in an instruction

One possible instruction set architecture

00011100

which instruction?

00 add
01 subtract
10 multiply
11 divide

Source register 1
AND destination register

Source register 2

"Add the value in register 3 to the value in register 4 and put the result back in register 3"

The computer architect builds circuitry to make this computation happen… but what does that circuitry look like…?

A 4-instruction 2-bit ALU
(Arithmetic-Logic Unit)

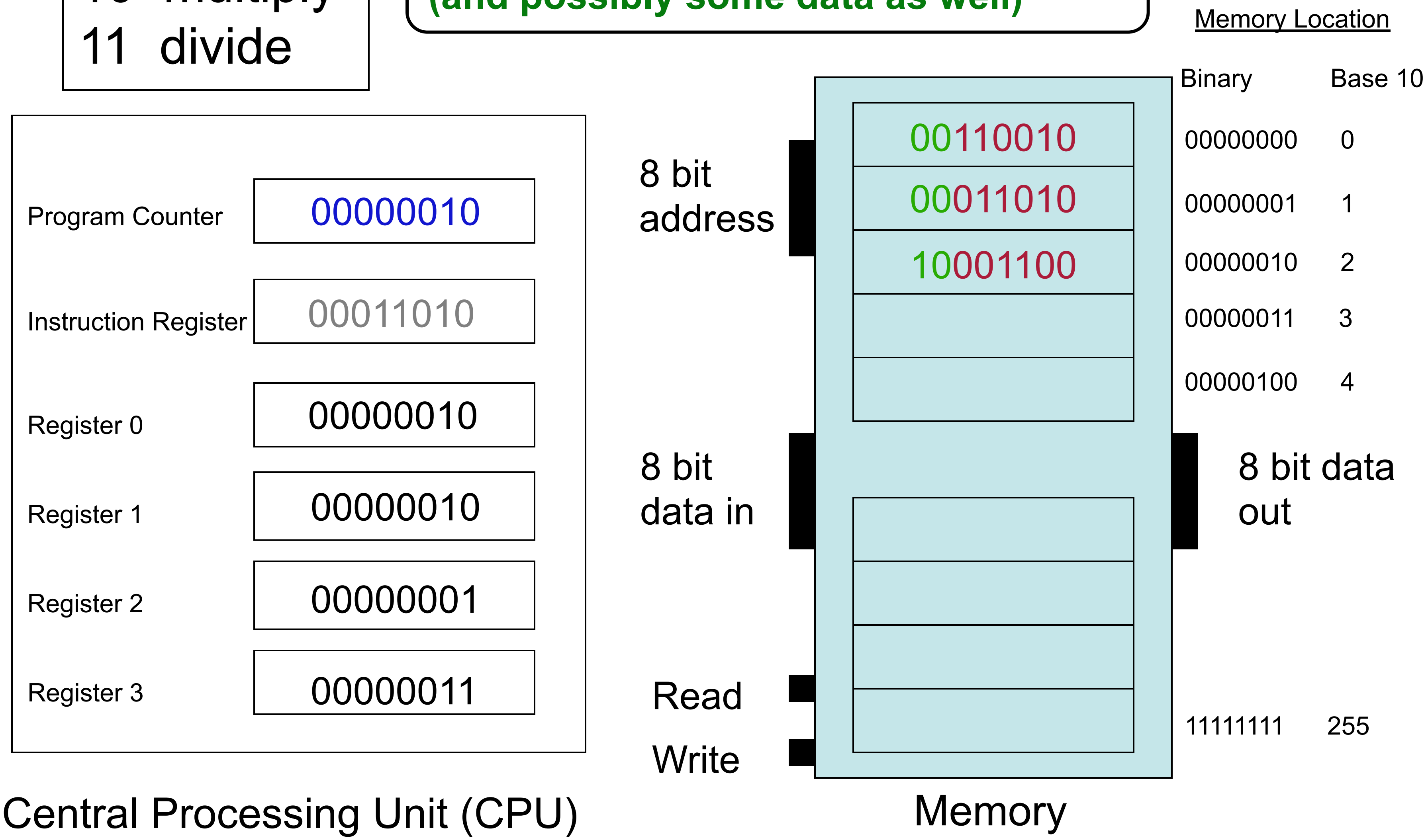
op1	op2	x1	x2	y1	y2	out1	out2
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	1
…							
0	0	1	1	1	1	1	0
…							
0	1	1	1	1	1	0	0
…							
1	0	1	1	1	0		
1	0	1	1	1	1	0	1

00	add
01	subtract
10	multiply
11	divide

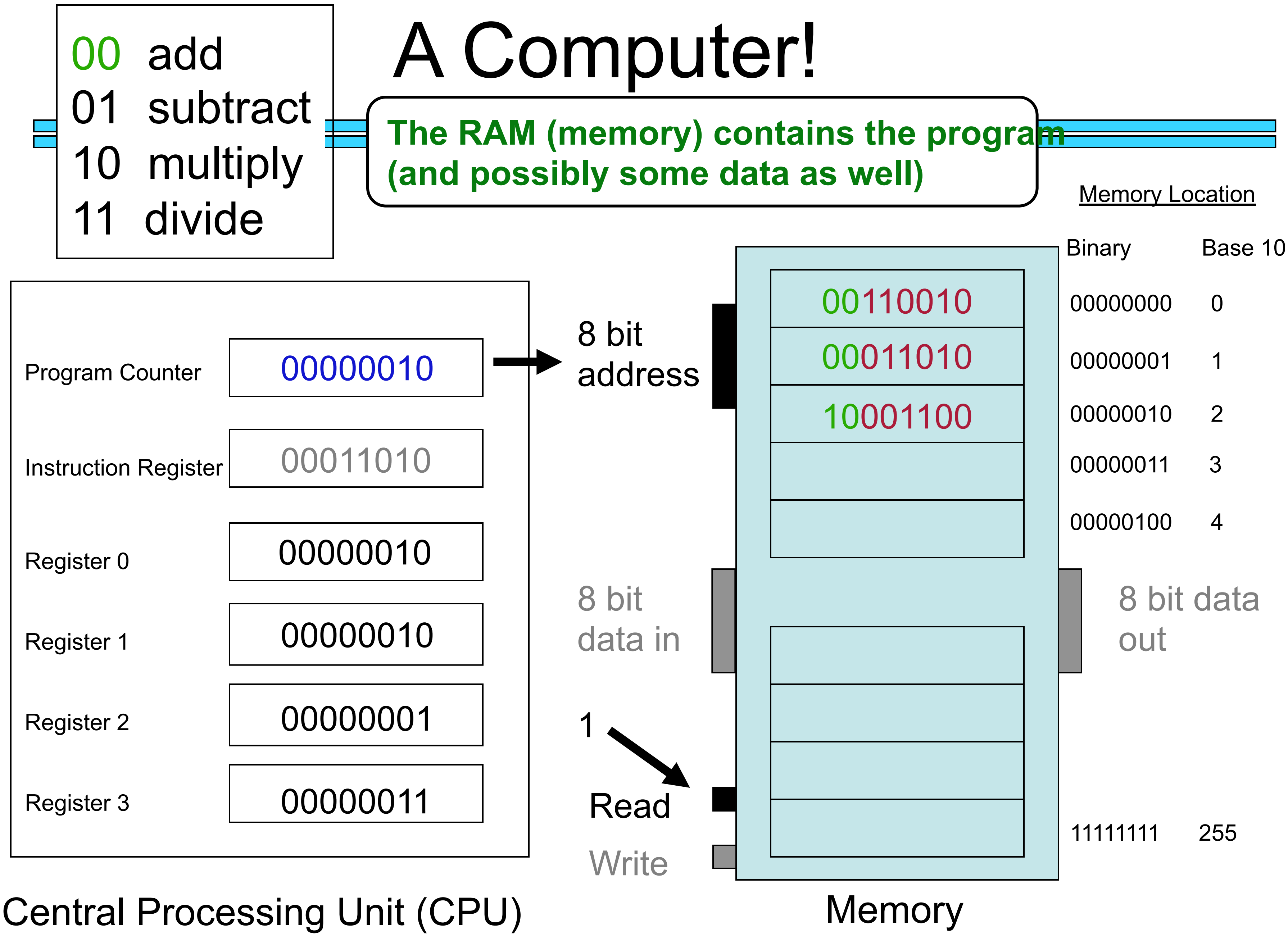
A Closer Look

00 add
01 subtract
10 multiply
11 divide

The RAM (memory) contains the program (and possibly some data as well)



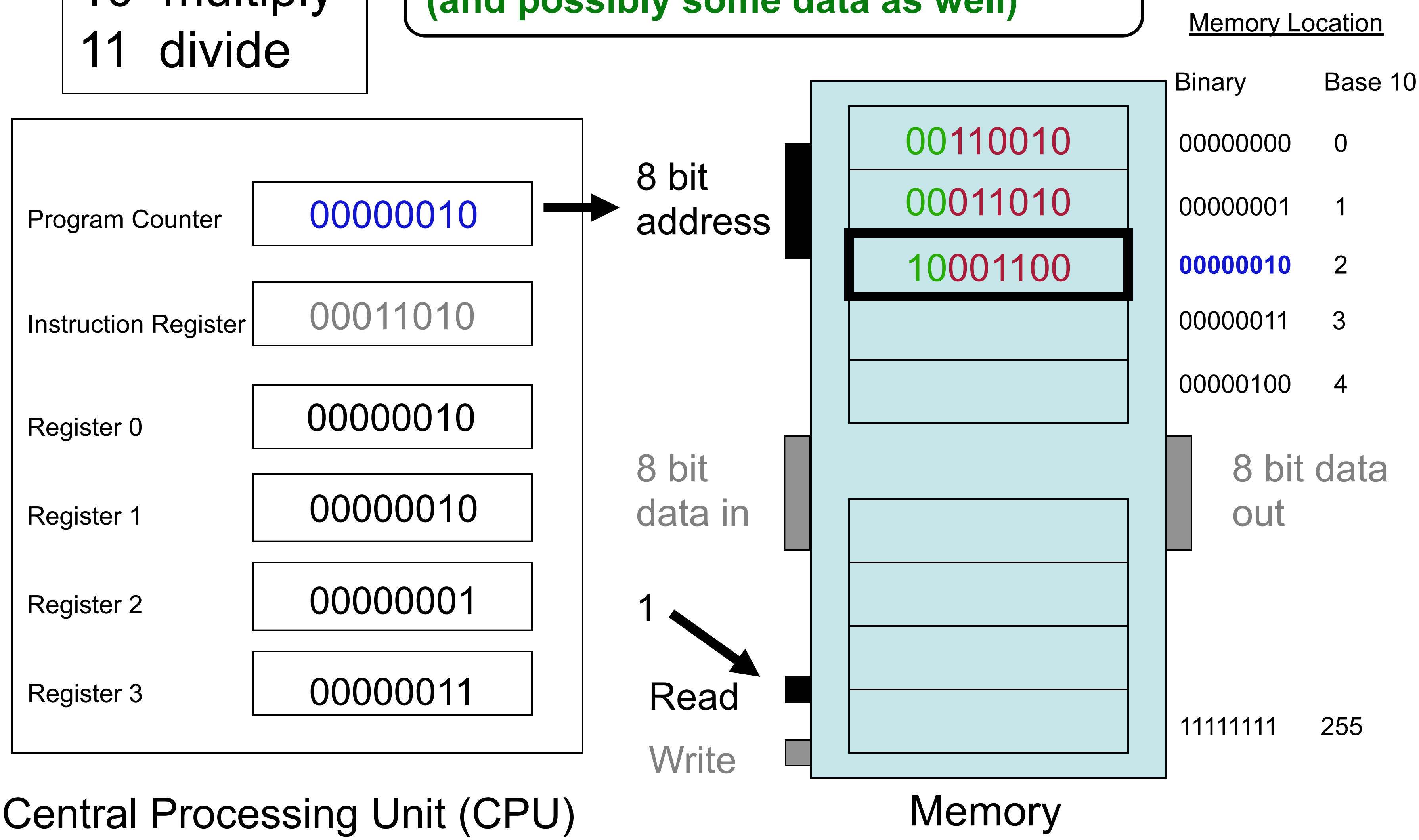
A Computer!


00 add
01 subtract
10 multiply
11 divide
The RAM (memory) contains the program
(and possibly some data as well)
Program Counter
00000010
Instruction Register
00011010
Register 0
00000010
Register 1
00000010
Register 2
00000001
Register 3
00000011
Central Processing Unit (CPU)
8 bit address
8 bit data in
1
Read
Write
00110010
00011010
10001100
Memory
8 bit data out
Memory Location
Binary
Base 10
00000000 0
00000001 1
00000010 2
00000011 3
00000100 4
11111111 255

00 add
01 subtract
10 multiply
11 divide

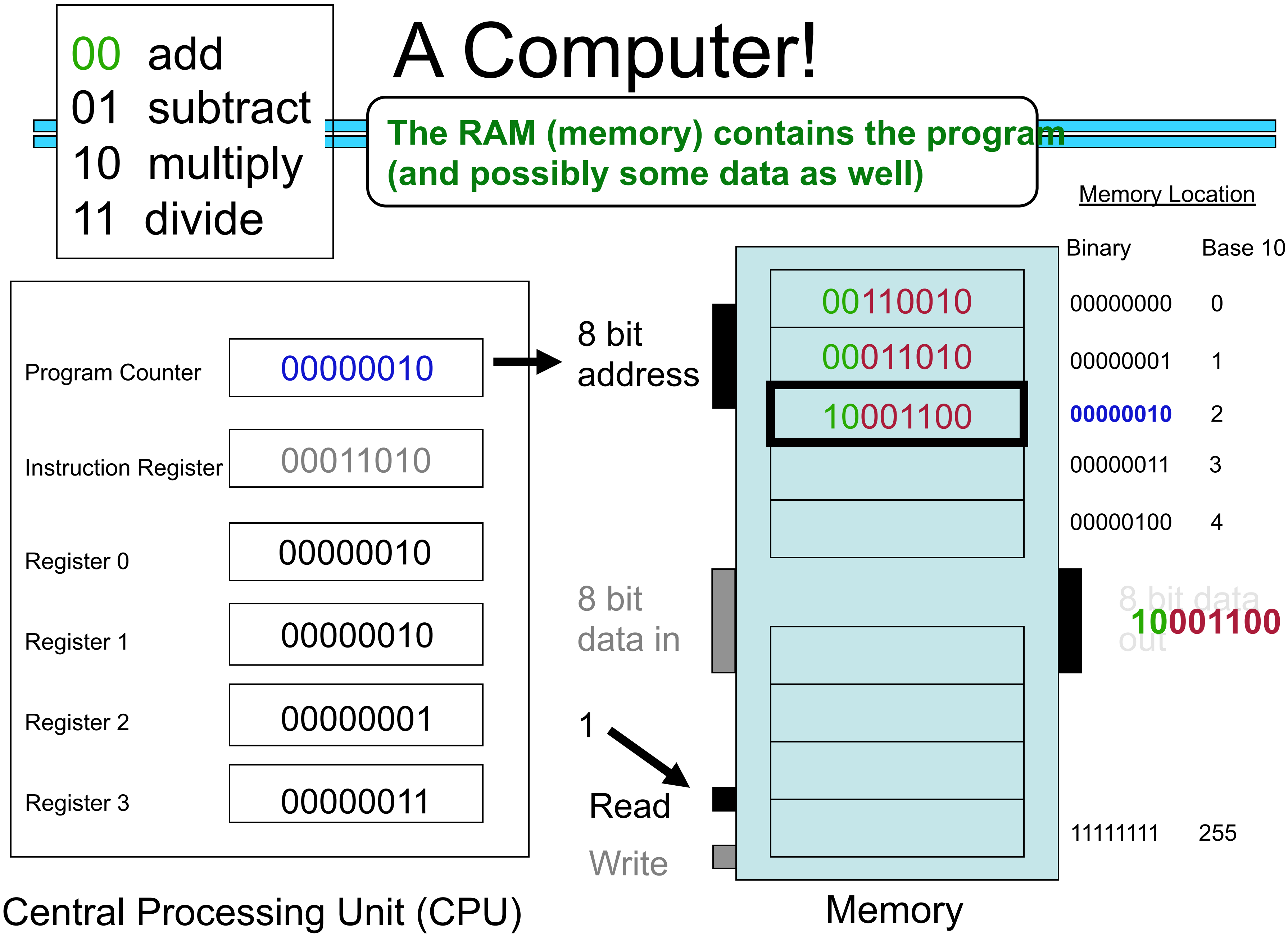
A Computer!

The RAM (memory) contains the program (and possibly some data as well)

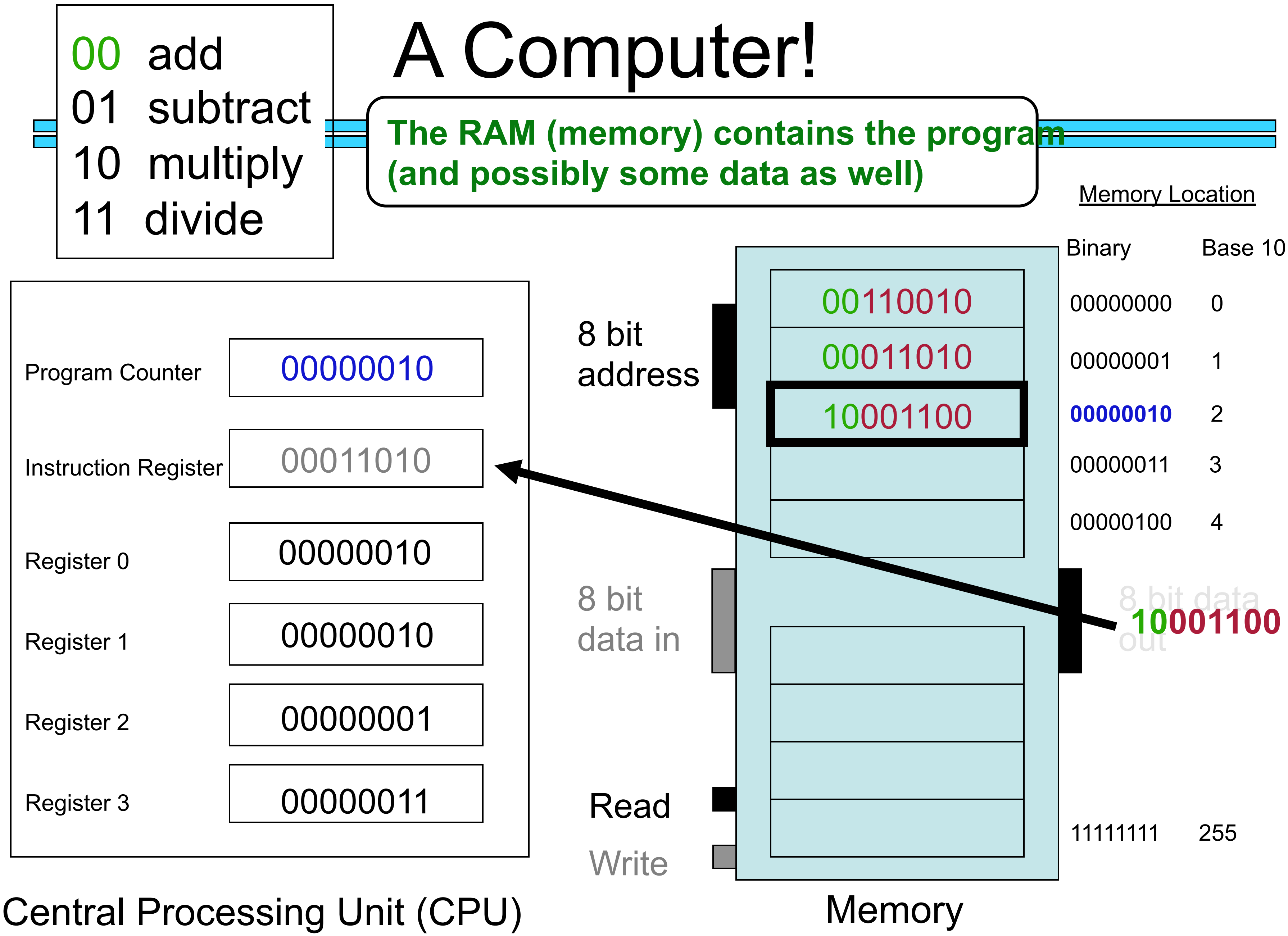


A Computer!

The RAM (memory) contains the program (and possibly some data as well)

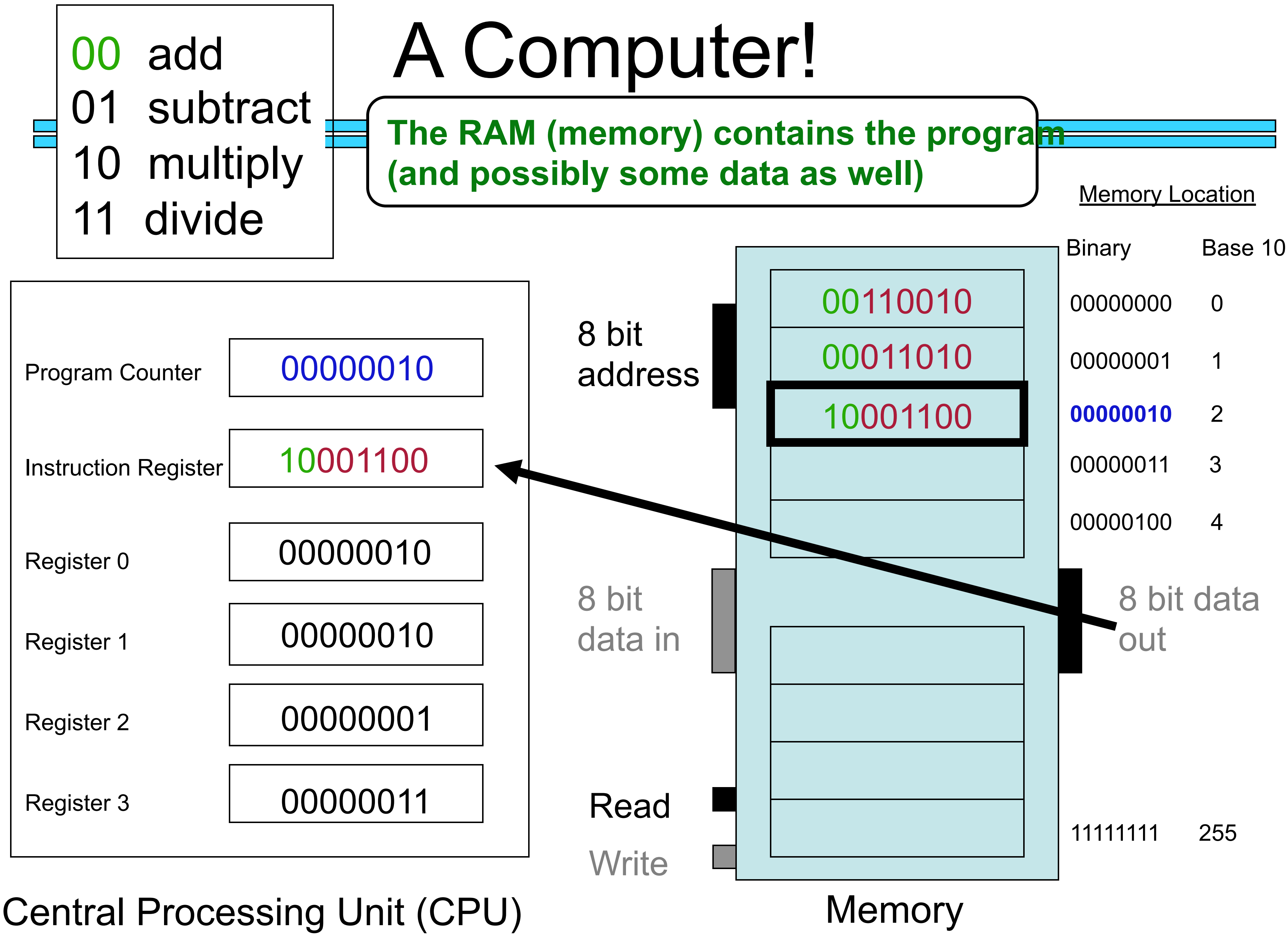


A Computer!


00 add
01 subtract
10 multiply
11 divide
The RAM (memory) contains the program
(and possibly some data as well)
Program Counter
00000010
Instruction Register
00011010
Register 0
00000010
Register 1
00000010
Register 2
00000001
Register 3
00000011
Central Processing Unit (CPU)
8 bit address
8 bit data in
Read
Write
00110010
00011010
10001100
Memory
8 bit data out
10001100
Memory Location
Binary
Base 10
00000000 0
00000001 1
00000010 2
00000011 3
00000100 4
11111111 255

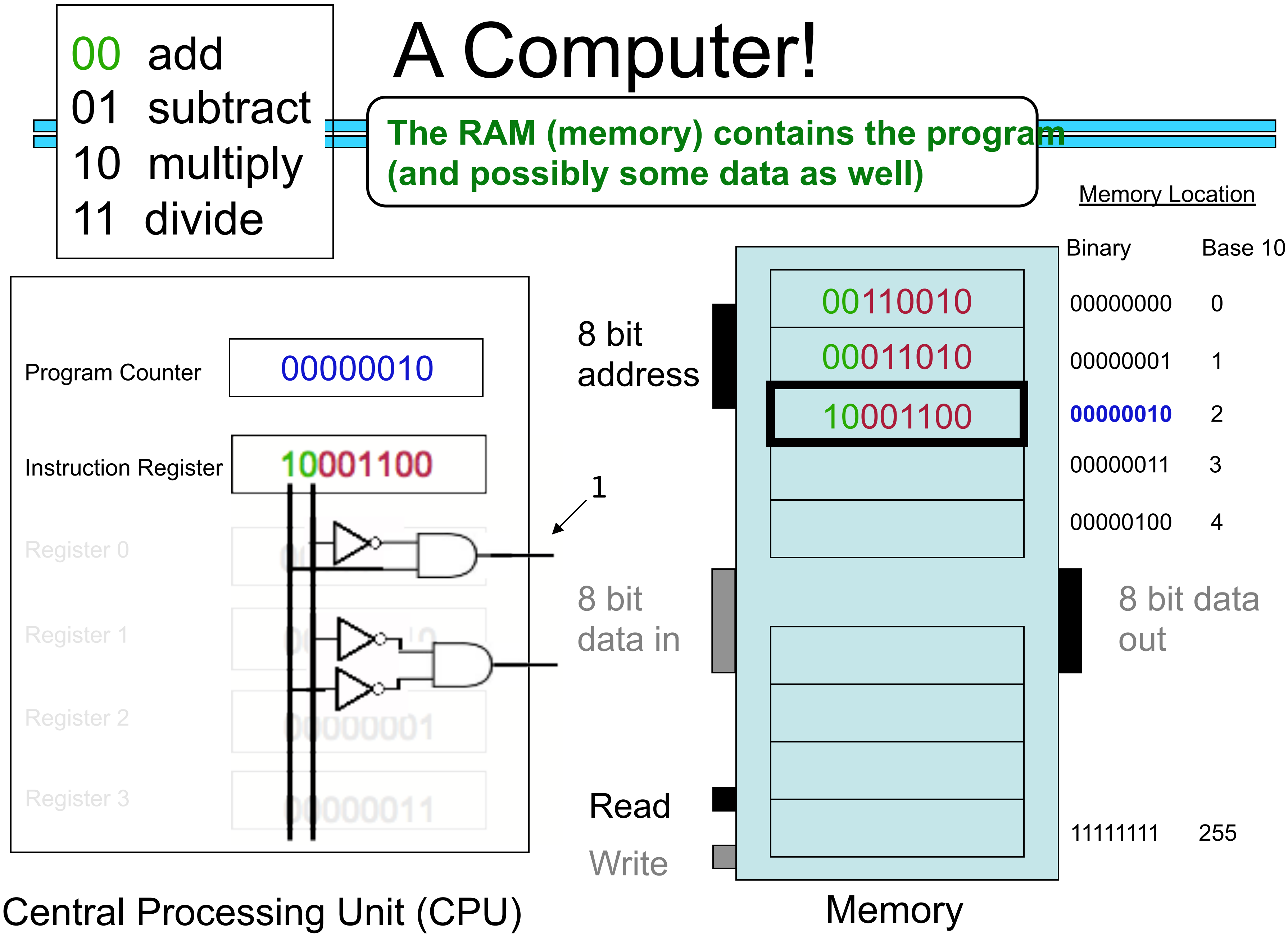
A Computer!

The RAM (memory) contains the program (and possibly some data as well)



A Computer!

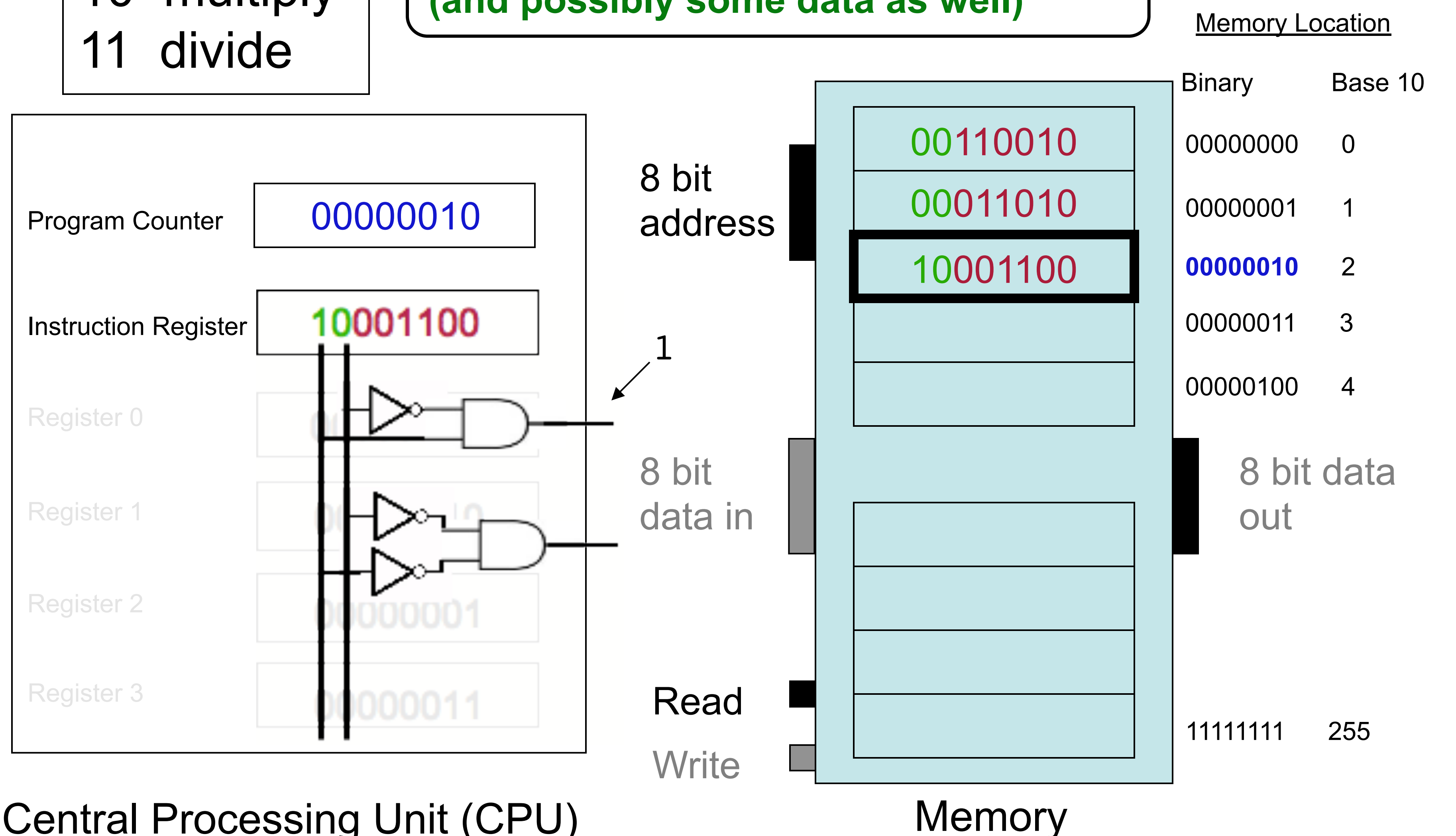
The RAM (memory) contains the program (and possibly some data as well)



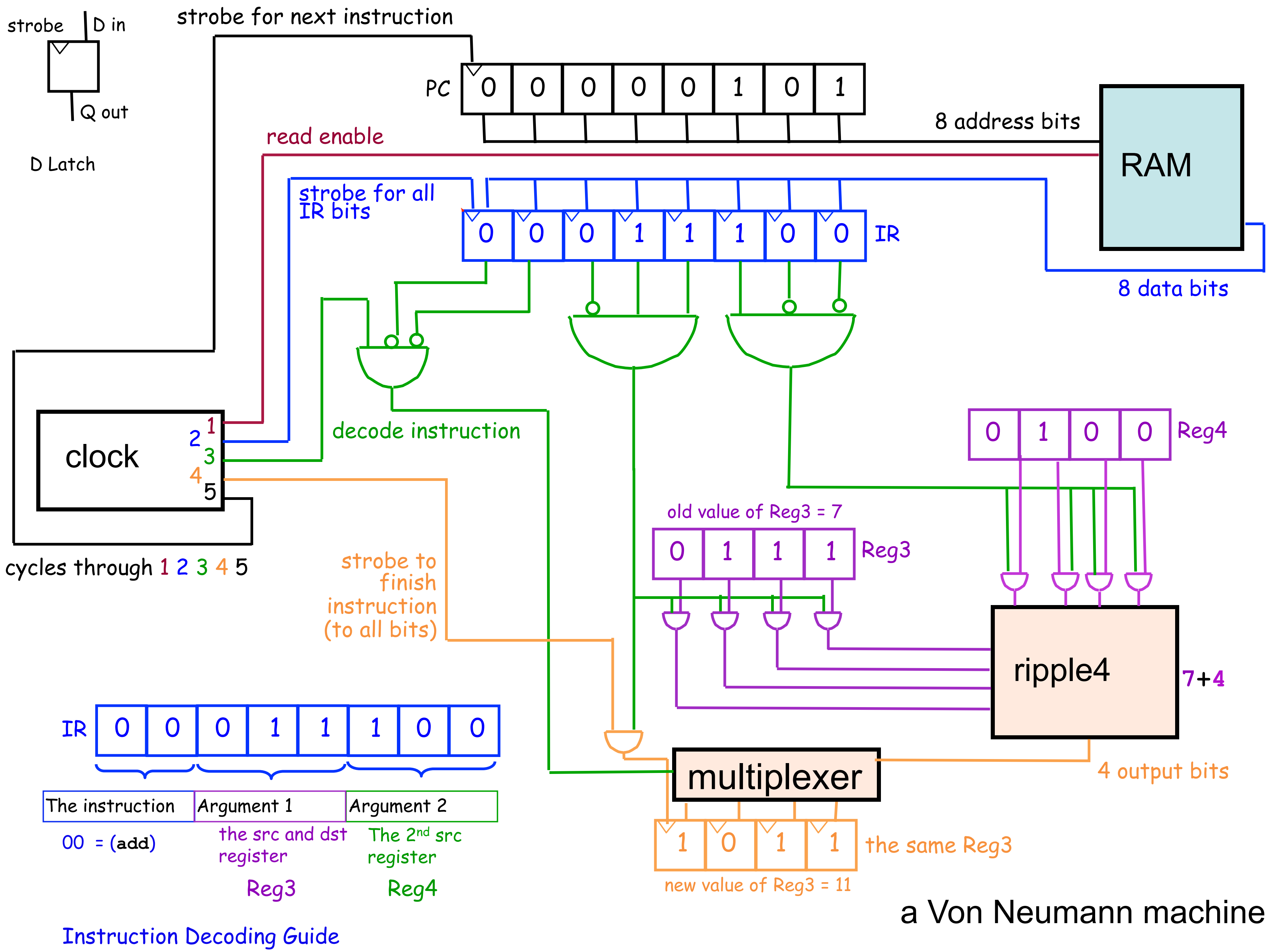
A Computer!

00 add
01 subtract
10 multiply
11 divide

The RAM (memory) contains the program (and possibly some data as well)



The fetch-execute cycle



a Von Neumann machine

Hmmm

The Harvey Mudd Miniature Machine

CPU

central processing unit

Von Neumann bottleneck

RAM

random access memory

Program Counter — Holds address of the next instruction

Instruction Register — Holds the current instruction

register 0 is "hard-wired" to store 0

r0: 0

r1

r2

...

r15

16 registers, each 16 bits

they can hold values from -32768 upto 32767

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

255 memory locations of 16 bits

Hmmm

NOTE:
The Hmmm "machine" is does not exist in hardware.
It exists in simulation only (in Python).
After next Monday,
it may also exist in Logisim…

Progra
Counte

Instructi
Register

register 0 is "hard-wired" to store 0

r15

255

255 memory locations of 16 bits

Function Calls in HMMM?

```
(define (main)
   (display (dist2 2 3)))

(define (dist2 x y)
    (+ (sq x) (sq y)))

(define (sq z)
   (* z z))
```

Option 1: Inlining

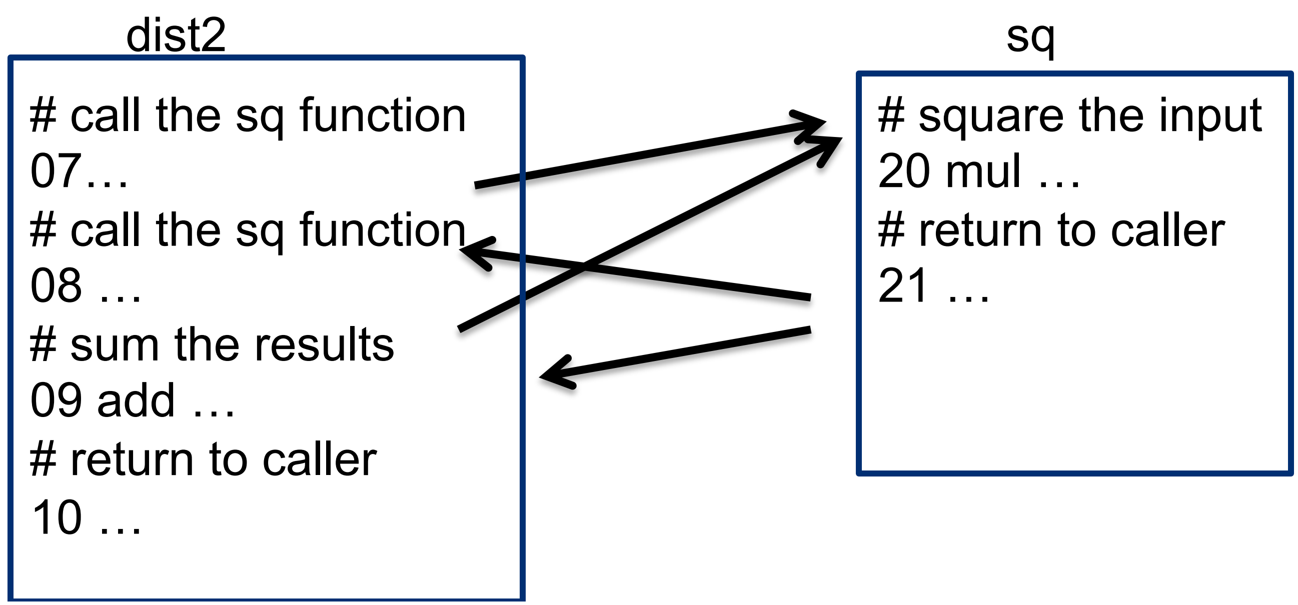
```
(define (main)
   (display (dist2 2 3)))

(define (dist2 x y)
    (+ (sq x) (sq y)))

(define (sq z)
   (* z z))
```

```
(define (main)
   (display
     (+ (* 2 2) (* 3 3))))
```

```
(define (dist2 x y)
   (display
     (+ (* x x) (* y y))))
```

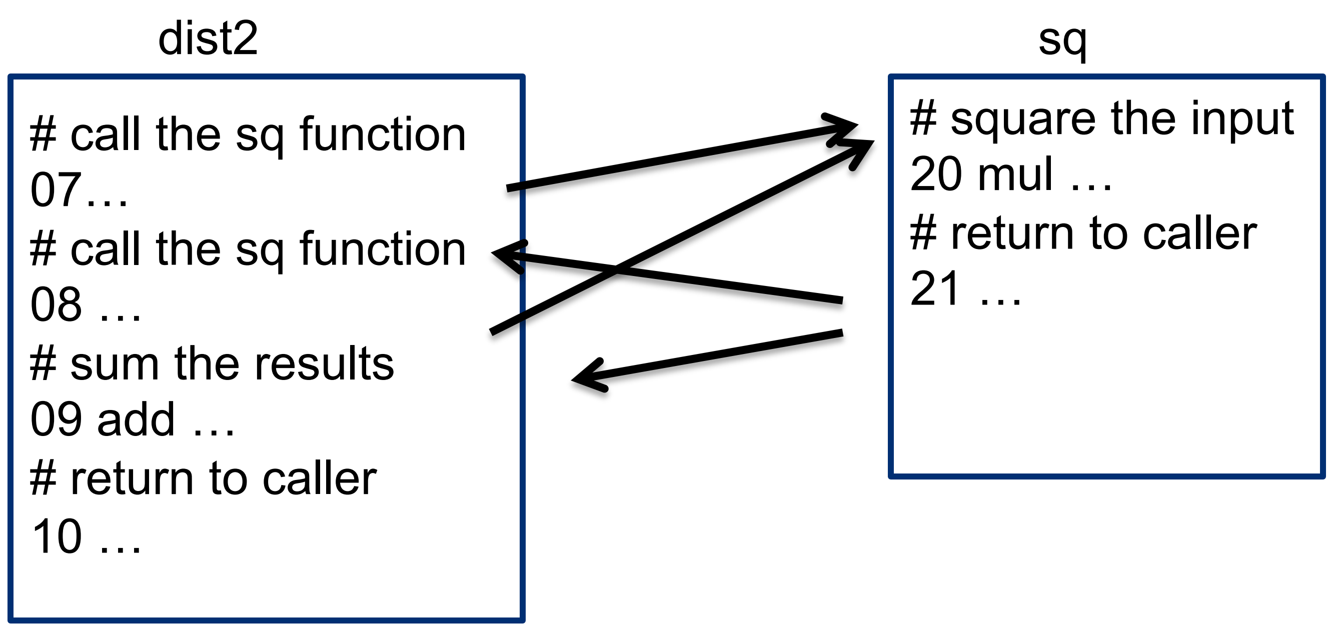
Option 2: Call And Return

Each function is a single block of machine code
When dist2 calls sq:
    code for dist2 jumps to the code for sq
When sq is done:
    jump back into the middle of dist2
    (immediately after the jump to sq)



Calling Conventions

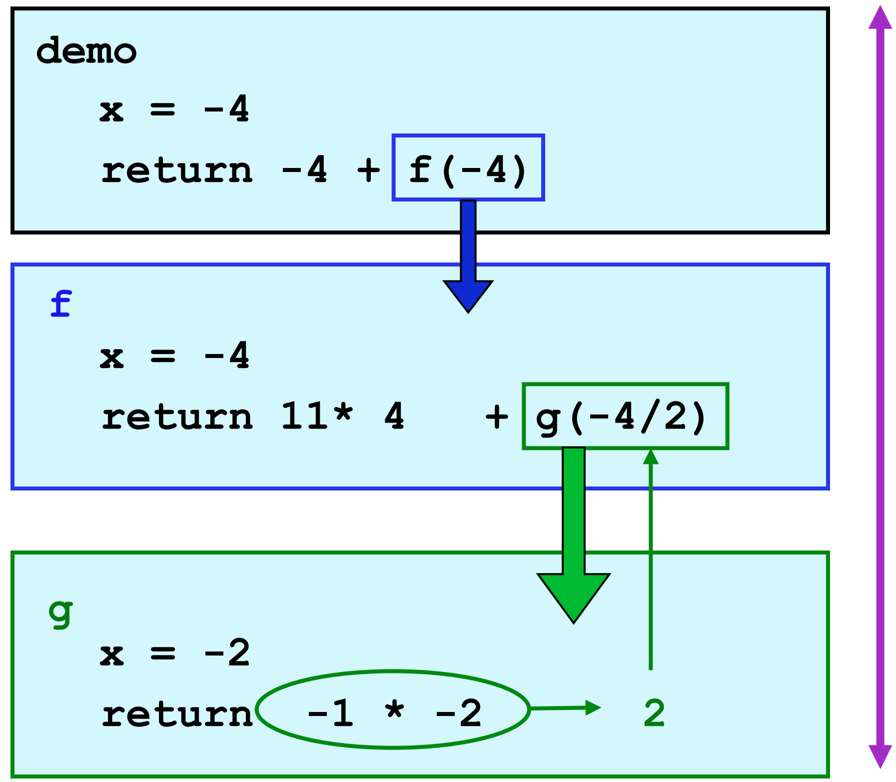
How does sq know what “the input” is?
How does dist2 know where to jump **to**?
How does sq know where to jump **back to**?
How do we keep sq from overwriting the registers that dist2 cares about?



Remember function *stack*ing?

```
(define (demo x)
    (+ x (f x)))


(define (f x)
     (+ (* 11 (g x))
        (g (/ x 2))))


(define (g x)
     (* -1 x))
```

What is `demo(-4)` ?



"The stack"

Remembers separate functions' values for variables…

Remembers where to send results back to…

HMMM Conventions

Register agreement + the STACK

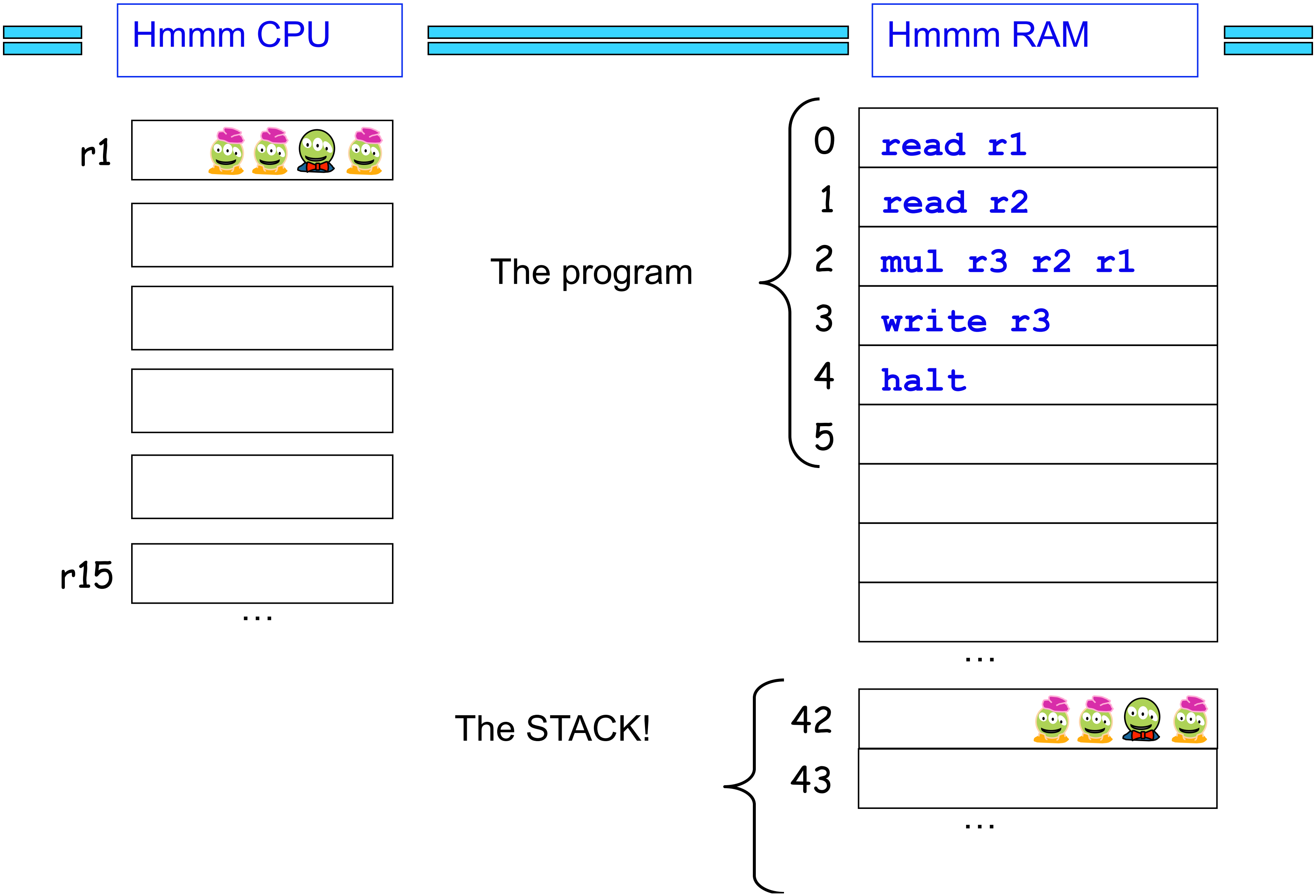
OUR "agreements":

function inputs	`r1,r2,…`
return **value**	`r13`
return **address**	`r14`
stack pointer	`r15`

When a function calls another function it will…

1. Save all of its important data in the stack
2. Put the inputs in r1, r2, r3…
3. Call the function…
4. (automatically) storing its return address in r14
5. And expect the results in r13
6. Retrieve its data from the stack

Where is the stack?



Key Idea: What's in RAM

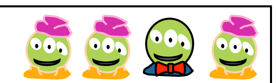
Hmmm RAM

The program

0	`read r1`
1	`read r2`
2	`mul r3 r2 r1`
3	`write r3`
4	`halt`
5	

...

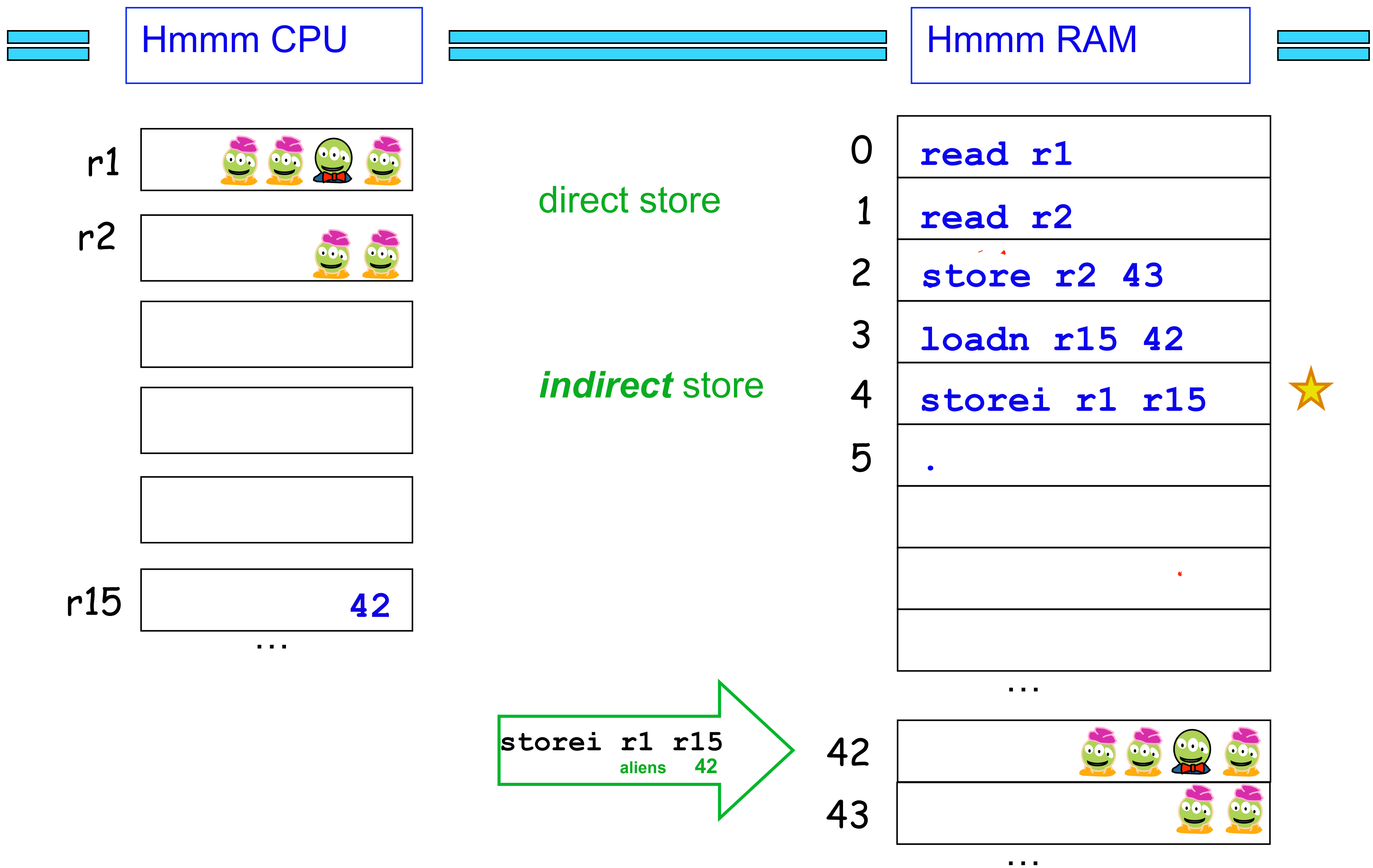
The STACK!

42	
43	

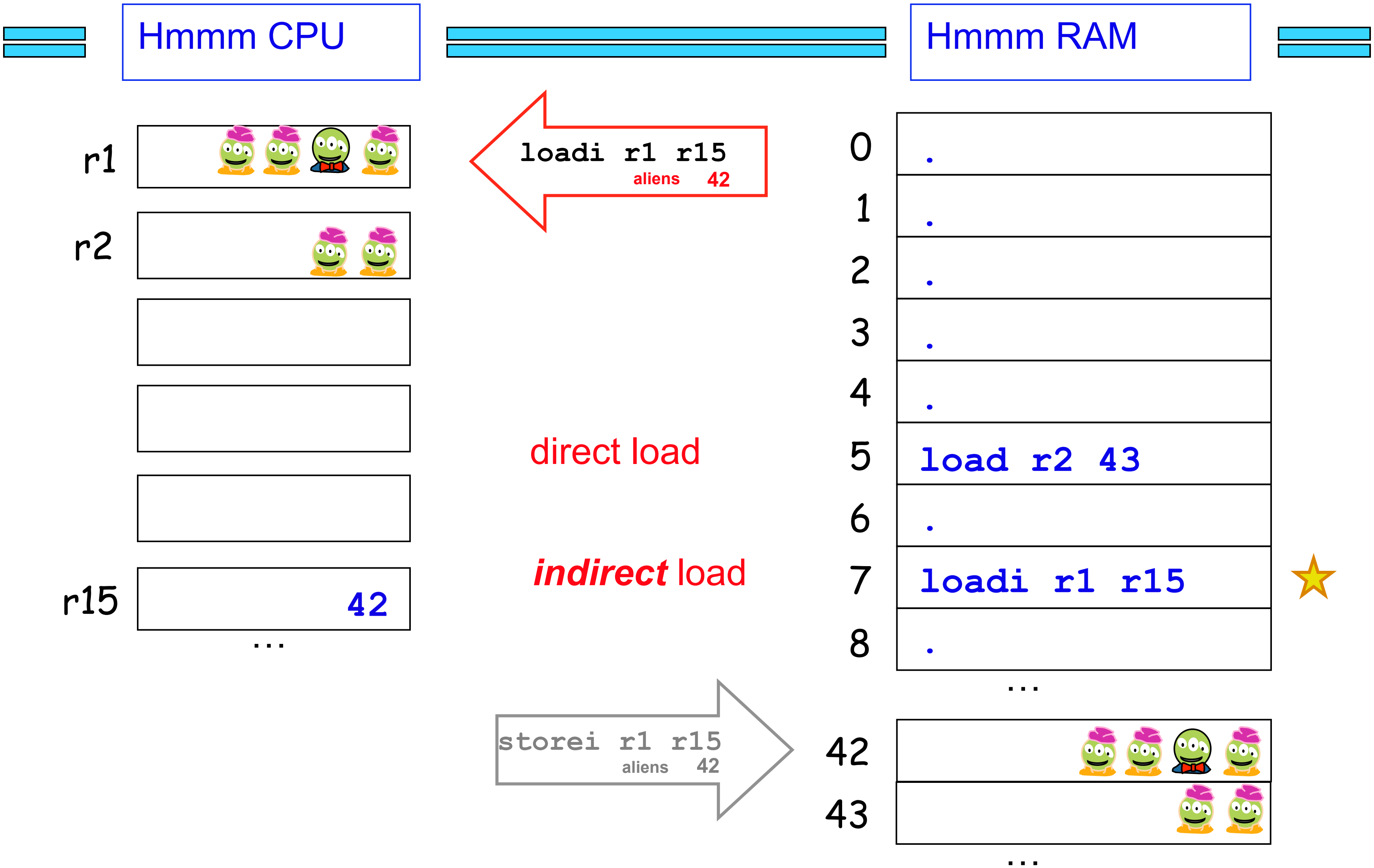
...

The program and
the stack SHARE
the RAM
program = low RAM
stack = high RAM

`store` goes TO memory



`load` comes FROM memory



call

A function call in Racket:

```
(define (main)
   (let* ((r1 (read-line))
          (result (demo r1)))
      (display result)))
```

```
(define (demo x)
    (+ x (f x)))
```

Hmmm's **call** operation:

call r14 4

puts NEXT line # into **r14**, then jumps to line 4

How functions REALLY work…

```
(define (main)
   (let* ((r1 (read-line))
          (result (demo r1)))
      (display result)))
```

```
00 read r1          # Get the user's input
01 loadn r15 SA     # Store the top of the stack
02 call r14 DEMO    # Call demo, starts at line ??
03 write r13        # Demo has returned,
                    # so just write and halt
04 halt
```

TBD

I want my input in **r1**, and I'll put my output in **r13**

```
(define (demo x)
    (+ x (f x)))
```

I want my input in **r1**, and I'll put my output in **r13**

```
(define (f x)
    (+ (* 11 (g x))
       (g (/ x 2))))
```

I want my input in **r1**, and I'll put my output in **r13**

```
(define (g x)
    (* -1 x))
```

How functions REALLY work…

```
(define (main)
   (let* ((r1 (read-line))
          (result (demo r1)))
      (display result)))
```

I want my input in **r1**, and I'll put my output in **r13**

```
(define (demo x)
    (+ x (f x)))
```

I want my input in **r1**, and I'll put my output in **r13**

```
(define (f x)
    (+ (* 11 (g x))
       (g (/ x 2))))
```

I want my input in **r1**, and I'll put my output in **r13**

```
(define (g x)
    (* -1 x))
```

```
00 read r1           # Get the user's input
01 loadn r15 SA      # Store the top of the stack
02 call r14 DEMO     # Call demo, starts at line ??
03 write r13         # Demo has returned,
                     # so just write and halt
04 halt
                     # ** demo begins here **
                     # First calculate f(x)
                     # Prepare: store data
05 storei r1 r15     # Store x on the stack
06 addn r15 1        # Increment stack pointer
07 storei r14 r15    # Store our return address
08 addn r15 1        # increment the stack pointer
                     # Prepare input: r1 already
                     # contains x
09 call r14 F        # Call f(x)
10 addn r15 -1       # F has returned. decr sp
11 loadi r14 r15     # Get our return address
12 addn r15 -1       # decrement the stack pointer
13 loadi r1 r15      # Get x back
14 add r13 r13 r1    # r13 = f(x) + x
15 jumpi r14         # Demo is done so return
```

CPU (registers)

r0: 0

r1 — Input value: x

r13 — Final result - return value - *in progress*

r14 — location / line to return TO

r15 — Stack pointer

Main memory (RAM)

```
00 read r1           # Get the user's input
01 loadn r15 SA      # Store the top of the stack
02 call r14 DEMO     # Call demo, starts at line ??
03 write r13         # Demo has returned,
                     # so just write and halt
04 halt
                     # ** demo begins here **
                     # First calculate f(x)
                     # Prepare: store data
05 storei r1 r15     # Store x on the stack
06 addn r15 1        # Increment stack pointer
07 storei r14 r15    # Store our return address
08 addn r15 1        # increment the stack pointer
                     # Prepare input: r1 already
                     # contains x
09 call r14 F        # Call f(x)
10 addn r15 -1       # F has returned. decr sp
11 loadi r14 r15     # Get our return address
12 addn r15 -1       # decrement the stack pointer
13 loadi r1 r15      # Get x back
14 add r13 r13 r1    # r13 = f(x) + x
15 jumpi r14         # Demo is done so return
… [MORE CODE HERE]

50
51
52
53
```

Let's say SA is 50…
(We can't really know until we know
how much code we have)

CPU (registers)

r0: 0

Input value: x

r1

Final result - return value - *in progress*

r13

location / line to return TO

r14

Stack pointer

r15

Main memory (RAM)

```
00 read r1          # Get the user's input
01 loadn r15 50     # Store the top of the stack
02 call r14 5       # Call demo, starts at line 9
03 write r13        # Demo has returned,
                    # so just write and halt
04 halt
                    # ** demo begins here **
                    # First calculate f(x)
                    # Prepare: store data
05 storei r1 r15    # Store x on the stack
06 addn r15 1       # Increment stack pointer
07 storei r14 r15   # Store our return address
08 addn r15 1       # increment the stack pointer
                    # Prepare input: r1 already
                    # contains x
09 call r14 F       # Call f(x)
10 addn r15 -1      # F has returned. decr sp
11 loadi r14 r15    # Get our return address
12 addn r15 -1      # decrement the stack pointer
13 loadi r1 r15     # Get x back
14 add r13 r13 r1   # r13 = f(x) + x
15 jumpi r14        # Demo is done so return
… [MORE CODE HERE]

50
51
52
53
```

CPU (registers)

Main memory (RAM)

Store data

Prepare data to pass

Call function

Restore data
(in REVERSE order)

```
00 read r1           # Get the user's input
01 loadn r15 50      # Store the top of the stack
02 call r14 5        # Call demo, starts at line 9
03 write r13         # Demo has returned,
                     # so just write and halt
04 halt
                     # ** demo begins here **
                     # First calculate f(x)
                     # Prepare: store data
05 storei r1 r15     # Store x on the stack
06 addn r15 1        # Increment stack pointer
07 storei r14 r15    # Store our return address
08 addn r15 1        # increment the stack pointer
                     # Prepare input: r1 already
                     # contains x
09 call r14 F        # Call f(x)
10 addn r15 -1       # F has returned. decr sp
11 loadi r14 r15     # Get our return address
12 addn r15 -1       # decrement the stack pointer
13 loadi r1 r15      # Get x back
14 add r13 r13 r1    # r13 = f(x) + x
15 jumpi r14         # Demo is done so return
… [MORE CODE HERE]

50
51
52
53
```

```
(define (main)
   (let* ((r1 (read-line))
          (result (demo r1)))
     (display result)))
```

```
00 read r1          # Get the user's input
01 loadn r15 50     # Store the top of the stack
02 call r14 5       # Call demo
03 write r13        # Demo has returned,
                    # so just write and halt
04 halt
```

```
(define (demo x)
    (+ x (f x)))
```

```
                    # ** demo begins here **
                    # First calculate f(x)
                    # Prepare: store data
05 storei r1 r15    # Store x on the stack
06 addn r15 1       # Increment stack pointer
07 storei r14 r15   # Store our return address
08 addn r15 1       # increment the stack pointer
                    # Prepare input: r1 already
                    # contains x
09 call r14 16      # Call f(x)
10 addn r15 -1      # F has returned. decr sp
11 loadi r14 r15    # Get our return address
12 addn r15 -1      # decrement the stack pointer
13 loadi r1 r15     # Get x back
14 add r13 r13 r1   # r13 = f(x) + x
15 jumpi r14        # Demo is done so return
```

```
(define (f x)
    (+ (* 11 (g x))
       (g (/ x 2))))
```

```
(define (g x)
    (* -1 x))
```

```
(define (main)
   (let* ((r1 (read-line))
          (result (demo r1)))
      (display result)))


(define (demo x)
    (+ x (f x)))
```

```
(define (f x)
    (+ (* 11 (g x))
       (g (/ x 2))))
```

```
                  # ****f begins here
16 storei r1 r15  # store our x
17 addn r15 1     # increment the stack pointer
18 storei r14 r15 # store our return address
19 addn r15 1     # increment the stack pointer
                  # r1 already contains x
20 call r14 44    # Call g(x)
                  # g(x) has returned
21 addn r15 -1    # decrement the stack pointer
22 loadi r14 r15  # get our return address back
23 addn r15 -1    # decrement the stack pointer
24 loadi r1 r15   # get x back
25 loadn r5 11    # load 11 into r5
26 mul r5 r13 r5  # r5 = 11*g(x)
27 storei r1 r15  # store x, prep for g again
28 addn r15 1     # increment stack pointer
29 storei r5 r15  # store our intermediate calculation
30 addn r15 1     # increment stack pointer
31 storei r14 r15 # store our return address
32 addn r15 1     # increment stack pointer
33 loadn r6 2     # prepare input
34 div r1 r1 r6   # x = x / 2
35 call r14 44    # call g(x/2)
36 addn r15 -1    # g is back, decr sp
37 loadi r14 r15  # get the return address
38 addn r15 -1    # decrement sp
39 loadi r5 r15   # get our intermediate value
40 addn r15 -1    # decrement the stack
41 loadi r1 r15   # get x back
42 add r13 r13 r5 # return g(x/2) + 11*g(x) (r13 + r5)
43 jumpi r14      # F is done!
```

```
(define (g x)
    (* -1 x))
```

```
                  # *** g starts here ***
44 loadn r5 -1    # prepare to multiply
45 mul r13 r1 r5  # return -1 * x
46 jumpi r14      # done, so return
```

Implementing functions

non-destructively!

(0) Use **`r15`** as the ***stack pointer***.

```
loadn r15 42
```

or some other large-enough value

(1) Before the function call,

Store all valuable data to the stack

store the return address **`r14`** and the inputs: **`r1`**, (**`r2`**), (**`r3`**)

```
storei r1 r15
addn r15 1
```

(2) Get **`r1`**, (**`r2`**), (**`r3`**), … ready as function "inputs."

There may or may not be some lines of code necessary to do this.

(3) Make the function call.

The result, if any, will be in `r13`.

```
call r14 #
```

line # of the function

(4) After the function call,

Load valuable data back from the stack

```
addn r15 -1
loadi r1 r15
```

for each item stored

Recursion?

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

"The Stack"

fac(5)

5 * fac(4)

4 * fac(3)

3 * fac(2)

2 * fac(1)

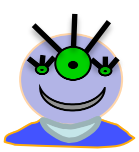
1

Remembers all of the individual calls to `fac`

Factorial via Recursion…

Python

```
x = input()
y = fac(x)
print y

def fac(x):
  """ recursive
      factorial! """
  if x == 0:
    return 1
  else:
    REC = fac(x-1)
    return x*REC
```



This is same as `return x*f(x-1)` but done in 2 steps…

Hmmm

```
00 read r1
01 loadn r15 42
02 call r14 5
03 jump 21
04 nop
05 jnez r1 8
06 loadn r13 1
07 jumpi r14
08 storei r1 r15
09 addn r15 1
10 storei r14 r15
11 addn r15 1
12 addn r1 -1
13 call r14 5
14 addn r15 -1
15 loadi r14 r15
16 addn r15 -1
17 loadi r1 r15
18 mul r13 r13 r1
19 jumpi r14
20 nop
21 write r13
22 halt
```