lucid, systematic,
and penetrating
treatment of basic
and dynamic data
structures, sorting,
recursive algorithms,
language structures,
and compiling

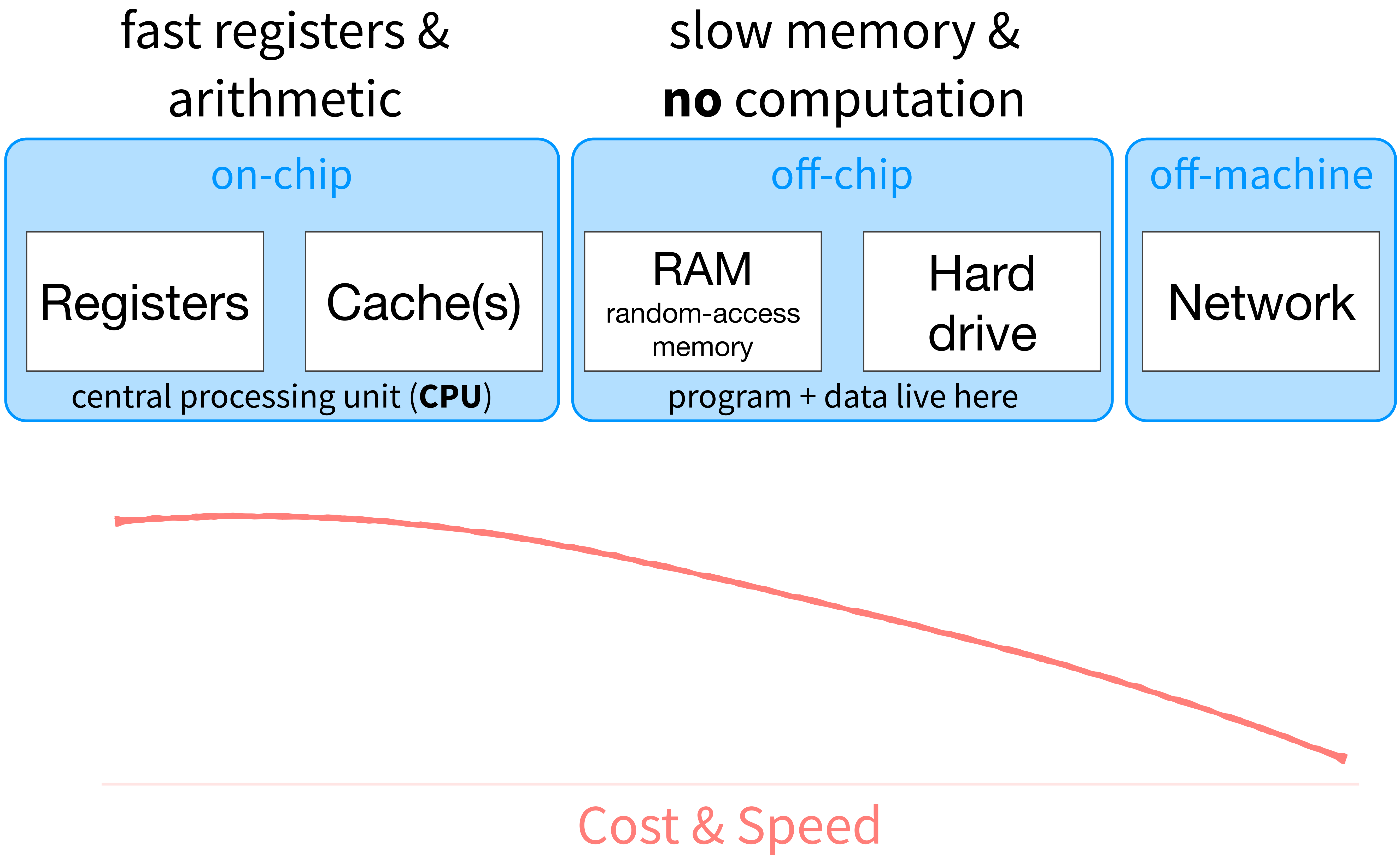
PRENTICE-HALL
SERIES IN
AUTOMATIC
COMPUTATION

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

Programs $\equiv$ Data

Data hierarchy



Our focus: registers and RAM

registers are like variables

Registers

central processing unit (**CPU**)

RAM
random-access memory

program + data live here

(1) What are **A**'s three inputs, and how is **A**'s output used?

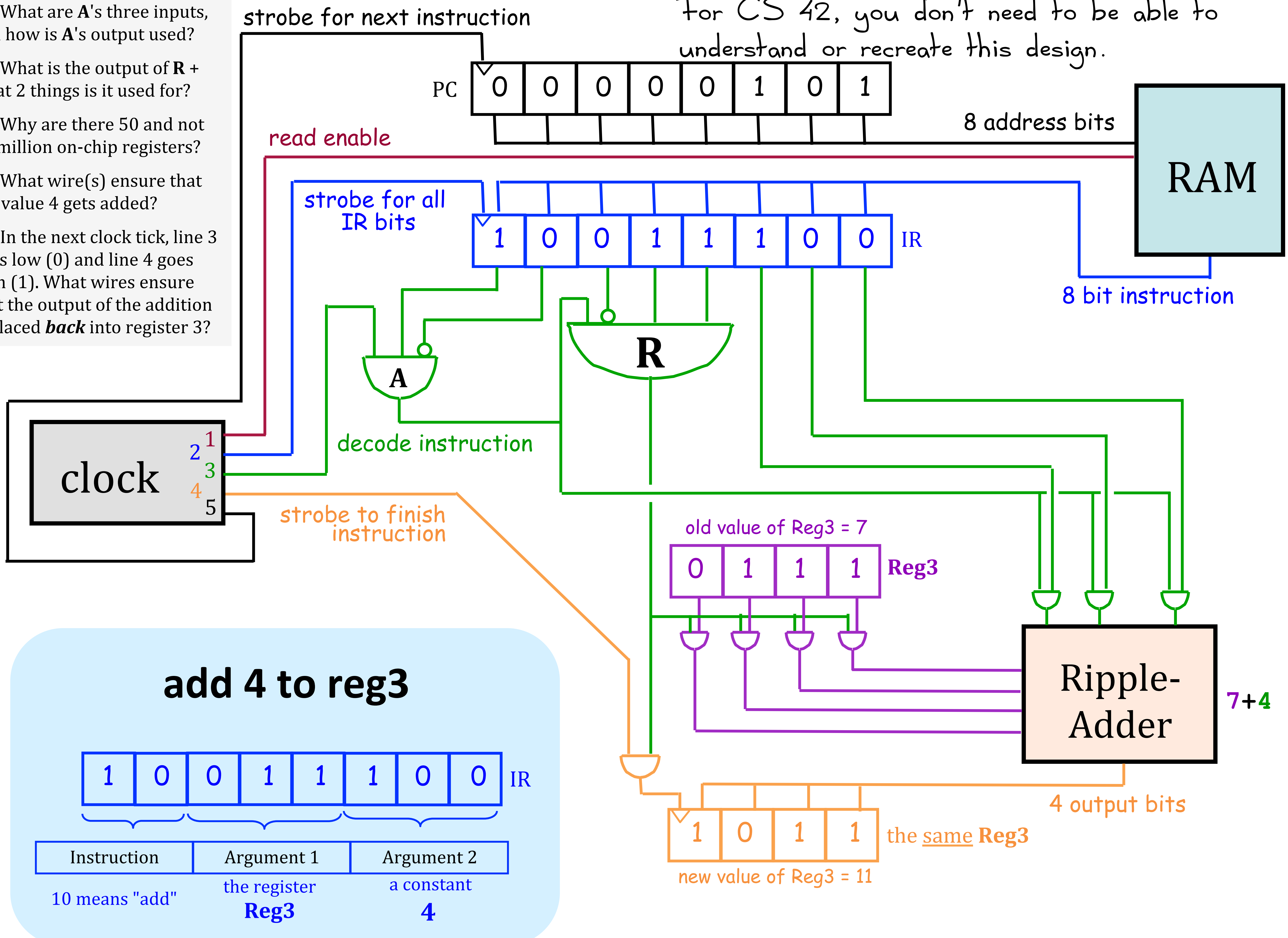
(2) What is the output of **R** + what 2 things is it used for?

(3) Why are there 50 and not 50 million on-chip registers?

(4) What wire(s) ensure that the value 4 gets added?

(5) In the next clock tick, line 3 goes low (0) and line 4 goes high (1). What wires ensure that the output of the addition is placed ***back*** into register 3?

This is just for your enjoyment!
For CS 42, you don't need to be able to understand or recreate this design.



What kinds of **problems** can **computers** solve?

What counts as a problem?

Decision problems on finite, bitstring inputs.

Can sequential logic solve all the problems that a DFA can? How about a Turing Machine?

What counts as a computer?

Harvey Mudd Miniature Machine (Hmmm)

registers are like variables

Registers

central processing unit (**CPU**)

RAM
random-access memory

program + data live here

16 registers

256 memory locations

For now, think of this as:
We can have programs with no more than 256 lines of code.

Hmmm operations: reading and writing

`read` `r1`	`r1 =` *`user input`*
`write` `r1`	`print r1's value to screen`

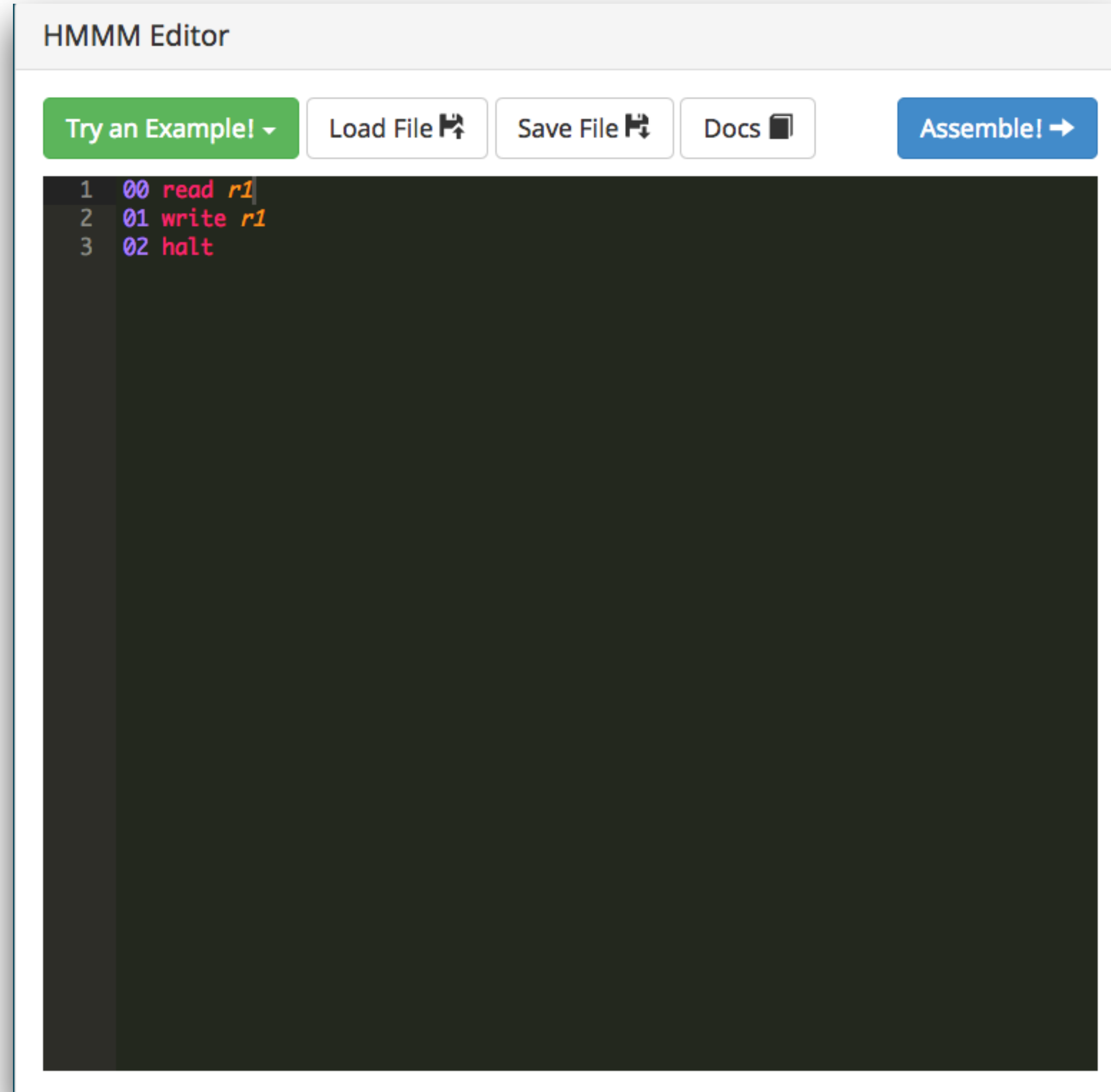
HMMM programs

Must have line numbers and must end with a `halt` instruction

```
00 read r1

01 write r1

02 halt
```



Hmmm operations: arithmetic

Translate these Hmmm operations into a language you understand.

```
setn r1 42        add r3 r2 r1
addn r1 42        sub r3 r2 r1
copy r2 r1        neg r3 r2
                  mul r3 r2 r1
```

Bonus questions (if you have time):

Use `addn` to infer the range of numbers that can be added to a register.

What happens if you forget `halt`?

Why do you think there is an `addn` *and* and `add` instruction?

Firstname Lastname T. 9 / 25

(Your response)

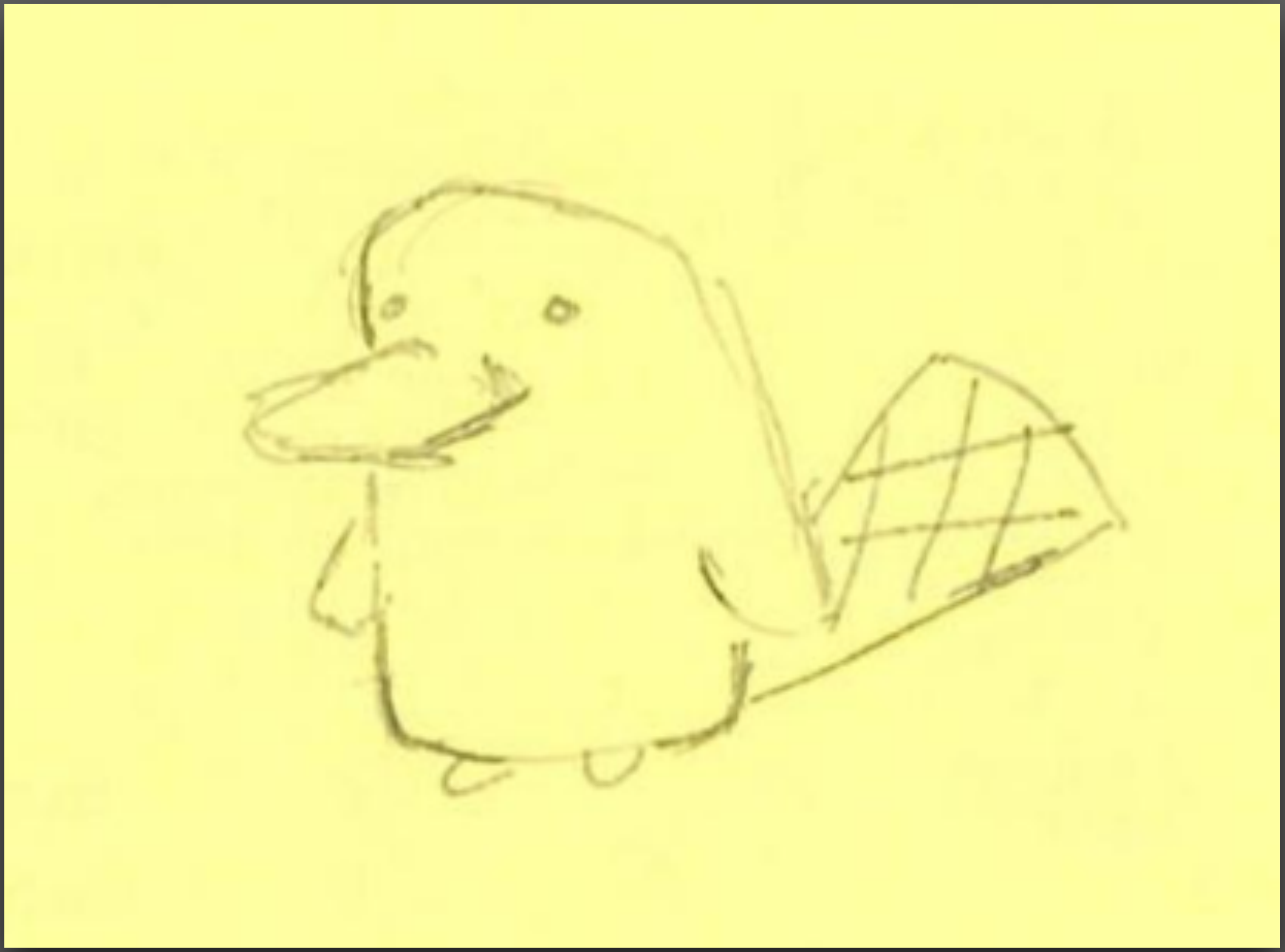
tinyurl.com/hmc-hmmm

Data operations are like assignments

Read from left to right

numbers in range -128 to 127

```
setn r1 42         r1 = 42
addn r1 42         r1 = r1 + 42
copy r1 r2         r1 = r2
add  r3 r1 r2      r3 = r1 + r2
sub  r3 r1 r2      r3 = r1 − r2
neg  r3 r1         r3 = −r1
mul  r3 r1 r2      r3 = r1 * r2
div  r3 r1 r2      r3 = r1 / r2
mod  r3 r1 r2      r3 = r1 % r2
```



Jumps control the program's behavior

Goto a particular line (possibly after comparing a register value to `0`)

```
jumpn 42          goto line 42
jeqzn r1 42       if r1 == 0, goto line 42
jnezn r1 42       if r1 != 0, goto line 42
jgtzn r1 42       if r1 >  0, goto line 42
jltzn r1 42       if r1 <  0, goto line 42
```

Longer Hmmm programs

What common function does this program compute?

```
00  read r1
01  read r2
02  sub r3 r1 r2
03  nop # "do nothing"
04  jgtzn r3 7
05  write r1
06  jumpn 8
07  write r2
08  halt
```

Write a Hmmm program that reads a positive integer value, then writes the factorial of that value.

Use only arithmetic, assignments, and jumps.

Why is there a `nop` instruction?

Can you come up with some good strategies for writing Hmmm programs?

tinyurl.com/hmc-hmmm

Factorial (iterative version)

```
# get the input (r1) from the user
0 read r1

# The program works by multiplying r1 * (r1 - 1) * (r1 - 2) * ... * 1,
# storing the result in r2, then printing r2
# (We'll assume, rather than check, that r1 is non-negative.)

# initialize answer (r2) to be 1
1 setn r2 1

# while r1 > 0:
#   multiply the result (r2) by the current value of the counter (r1)
#   decrement r1
2 jeqzn r1 6   # loop condition: enter loop if r1 != 0
3 mul r2 r2 r1
4 addn r1 -1
5 jumpn 2      # go back to the top of the loop

# write the result
6 write r2
7 halt
```