

Lecture 4

Machine Language

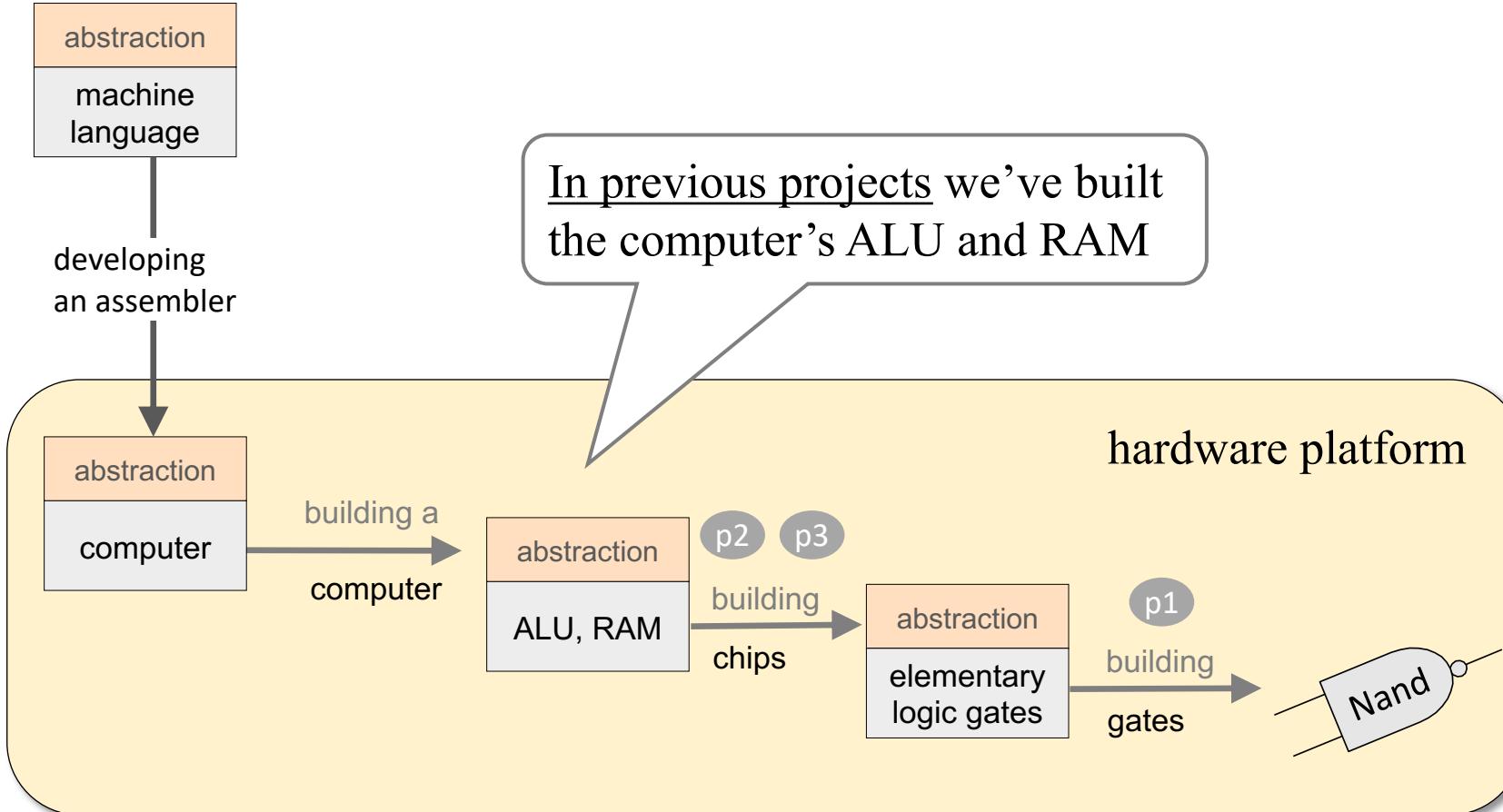
These slides support chapter 4 of the book

The Elements of Computing Systems

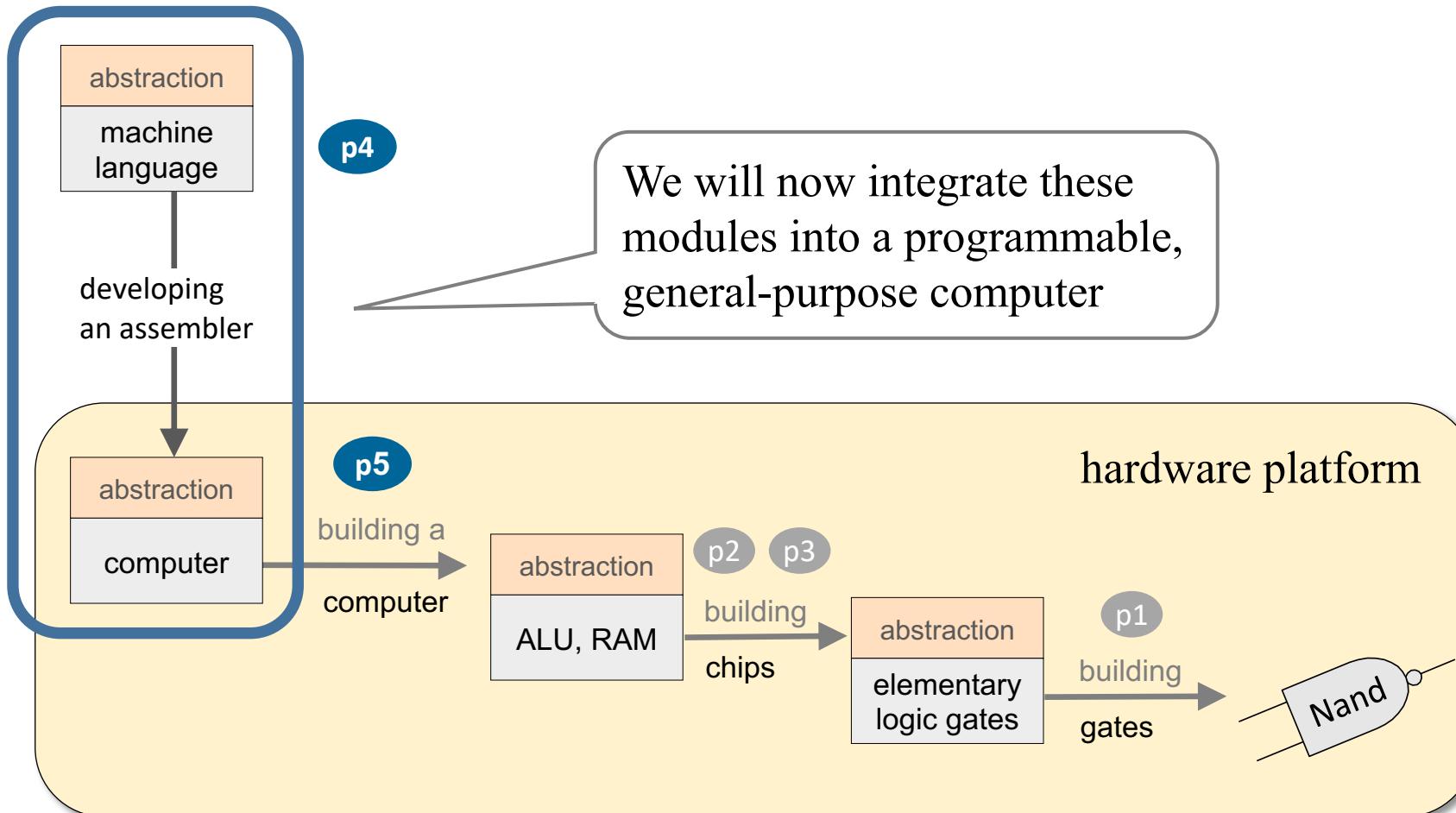
By Noam Nisan and Shimon Schocken

MIT Press, 2021

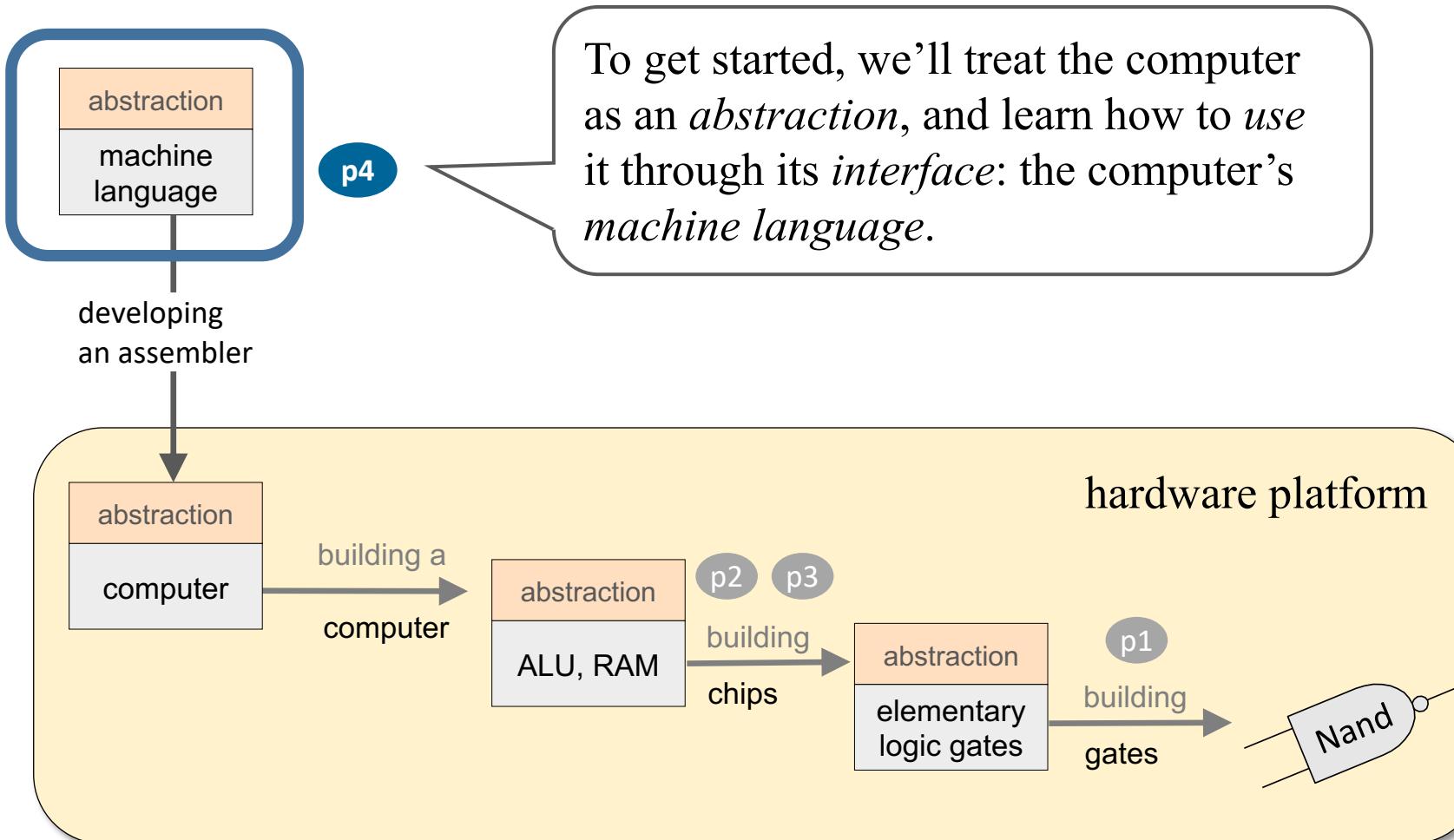
Nand to Tetris Roadmap: Hardware



Nand to Tetris Roadmap: Hardware



Nand to Tetris Roadmap: Hardware



Computer systems are flexible and versatile

Same **hardware** can run many different programs (**software**)



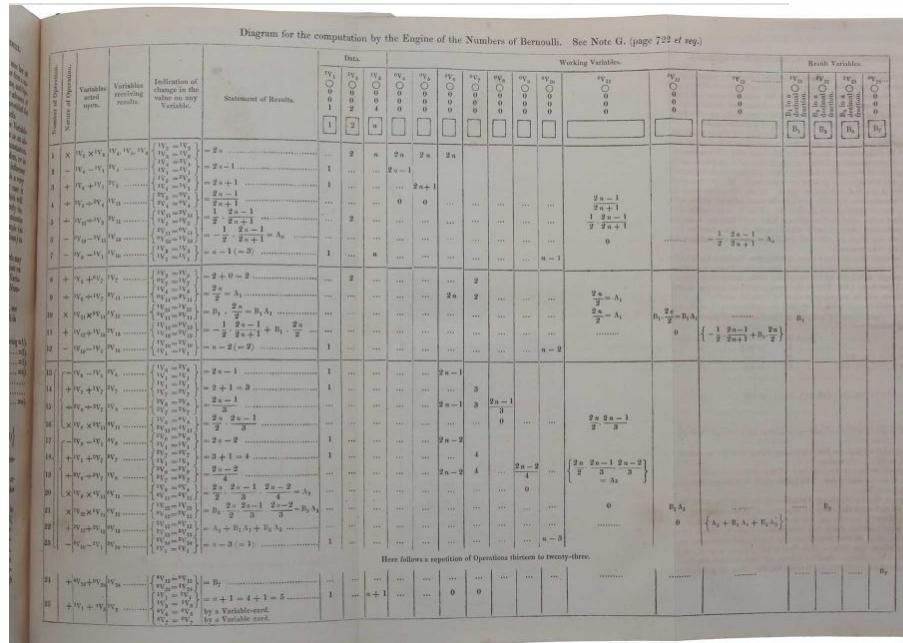
Computer systems are flexible and versatile

Same **hardware** can run many different programs (**software**)



Ada Lovelace

Early symbolic program



Computer systems are flexible and versatile

Same **hardware** can run many different programs (**software**)



Alan Turing
(1936)

1936.]	ON COMPUTABLE NUMBERS.	245
\$im	$f'(\text{sim}_1, \text{sim}_1, z)$	\$im. The machine marks out the instructions. That part of the instructions which refers to operations to be carried out is marked with u , and the final m -configuration with y . The letters z are erased.
\$im ₁	con (\$im ₂ ,)	
\$im ₂	$\begin{cases} A & \text{$im}_3 \\ \text{not } A & R, Pu, R, R, R \end{cases}$	\$im ₂
\$im ₃	$\begin{cases} \text{not } A & L, Py \\ A & L, Py, R, R, R \end{cases}$	e(mf, z)
mf	g(mf, :)	mf. The last complete configuration is marked out into four sections. The configuration is left unmarked. The symbol directly preceding it is marked with x . The remainder of the complete configuration is divided into two parts, of which the first is marked with v and the last with w . A colon is printed after the whole. → \$y.
mf ₁	$\begin{cases} \text{not } A & R, R \\ A & L, L, L, L \end{cases}$	mf ₁
mf ₂	$\begin{cases} C & R, Px, L, L, L \\ : & \end{cases}$	mf ₂
mf ₃	$\begin{cases} D & R, Px, L, L, L \\ \text{not } : & R, Pv, L, L, L \end{cases}$	mf ₃
	:	mf ₄

Universal Turing Machine

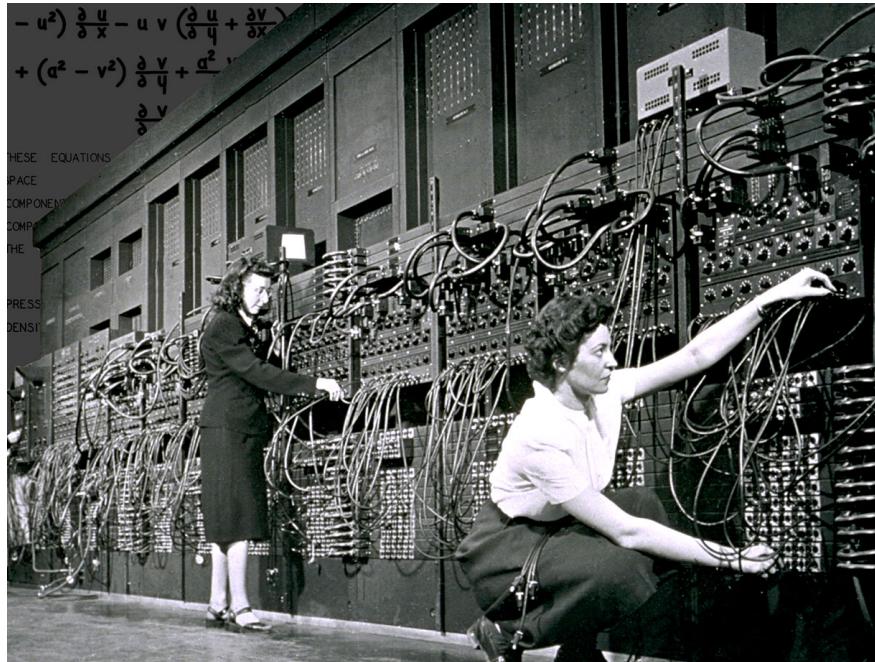
Landmark paper, describing a theoretical general-purpose computer

Computer systems are flexible and versatile

Same **hardware** can run many different programs (**software**)

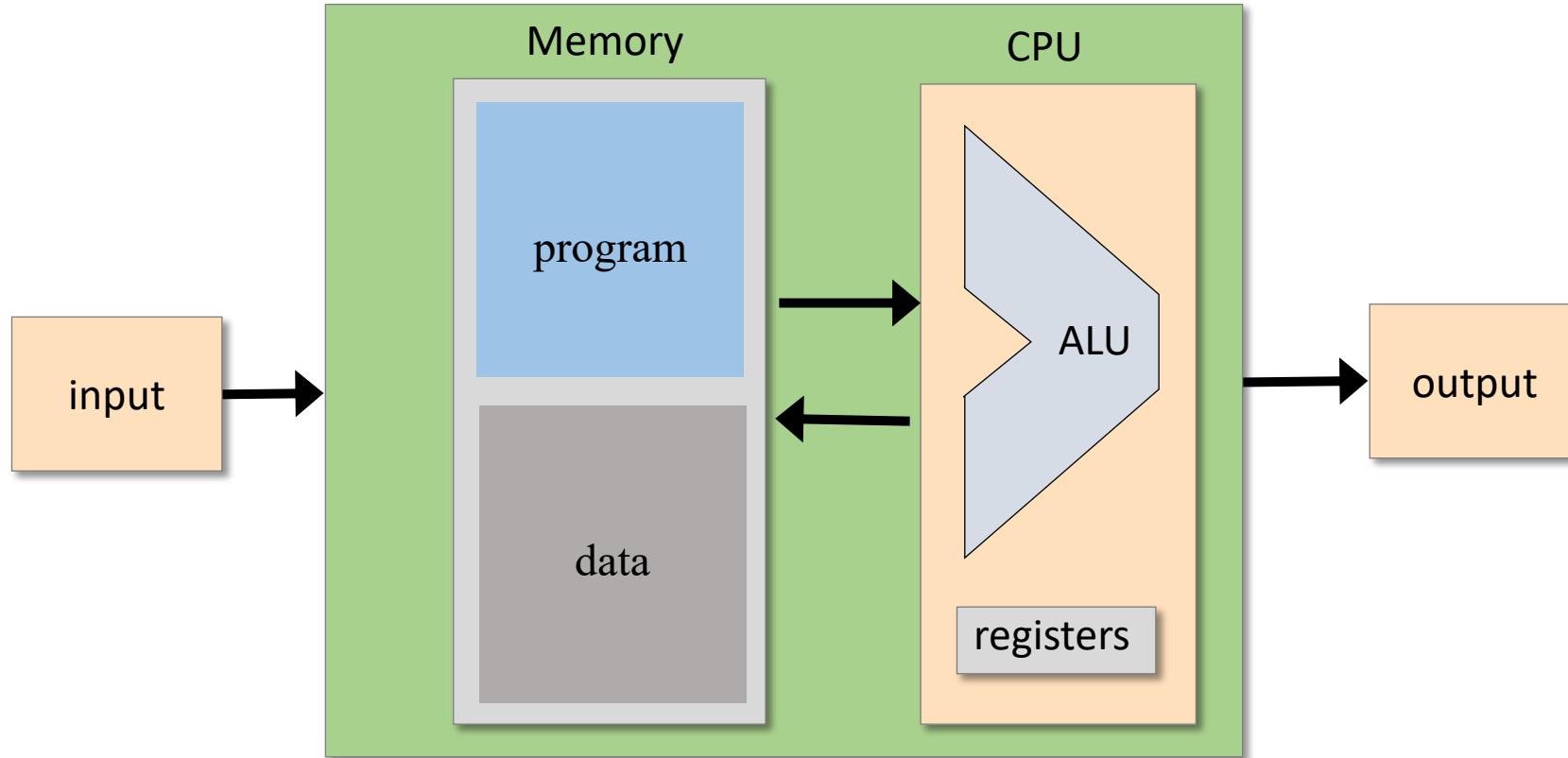


John Von Neumann
(1945)

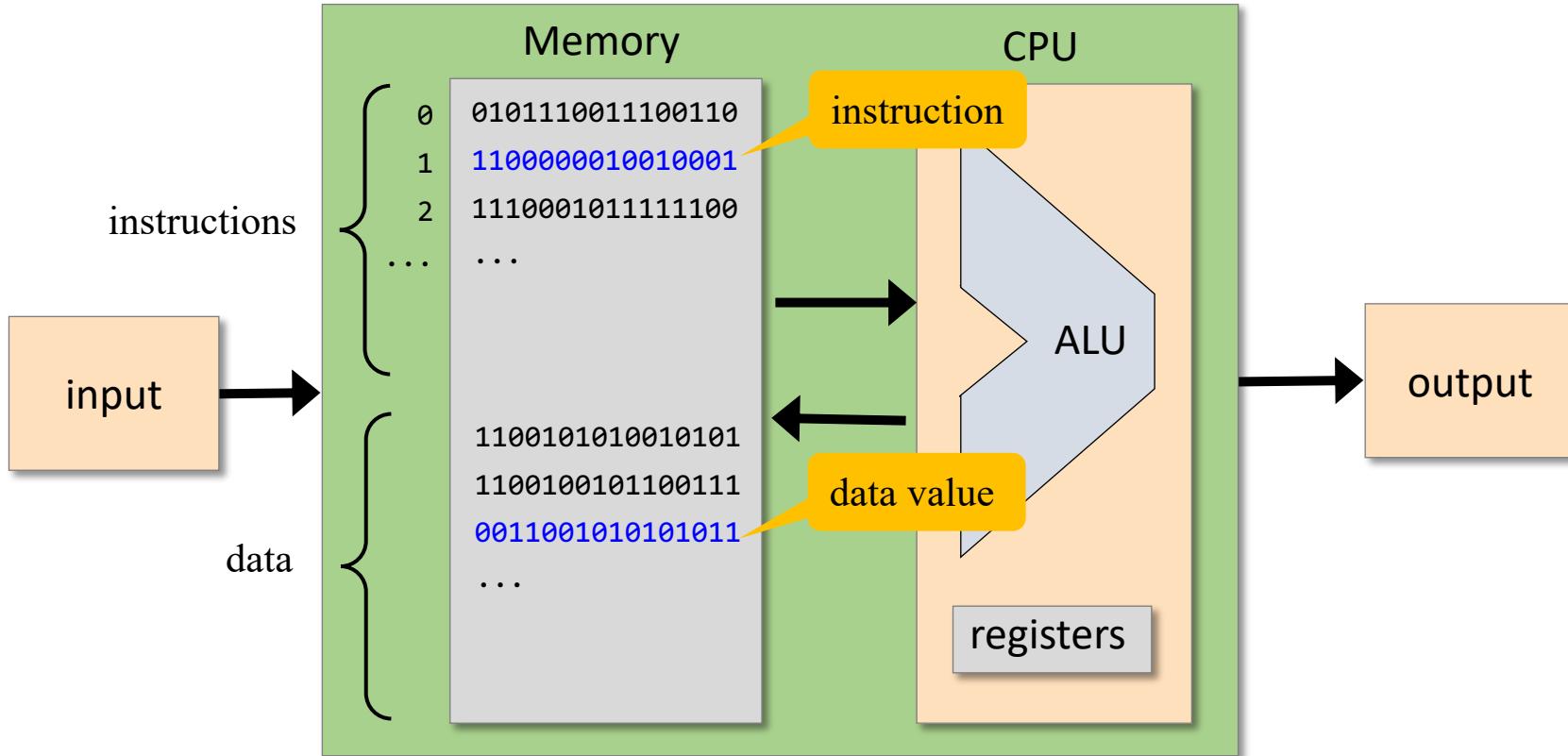


Landmark general-purpose computer
ENIAC, University of Pennsylvania

Computer architecture



Computer architecture



Stored program concept

- The computer memory can store programs, just like it stores data
- Programs = data.

A fundamental idea in the history of computer science

Lecture plan

Overview



Machine language

- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

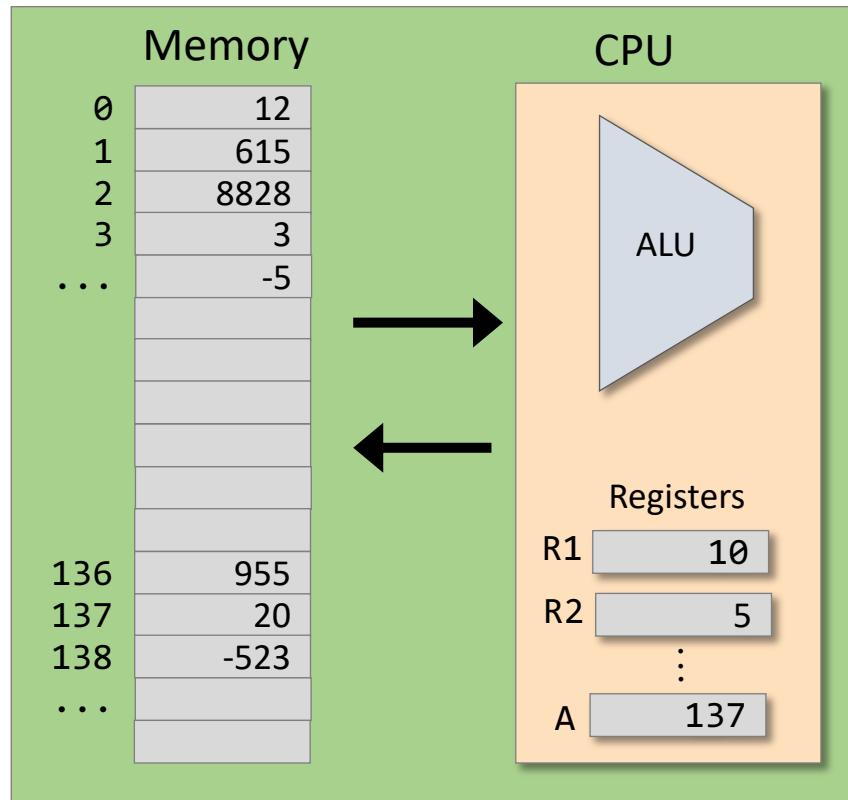
Machine Language

Computer

(conceptual definition):

A processor (CPU) that manipulates a set of *registers*:

- CPU-resident registers
(few, accessed directly, by name)
- Memory-resident registers
(many, accessed by address)



Machine language

A formalism for accessing and manipulating registers.

Registers

Data registers

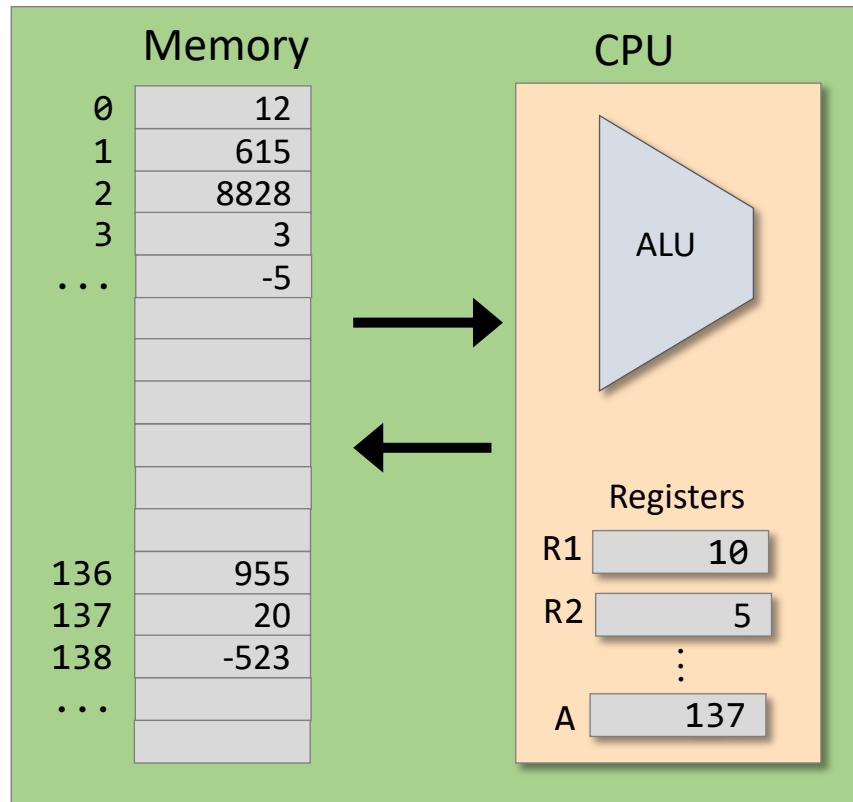
Hold data values

Address register

Holds an address

Instruction register

Holds an instruction



- All these registers are... registers (containers that hold bits)
- The number and bit-width of the registers vary from one computer to another.

Typical operations

// R1 ← R1 + R2

add R1, R2

// R1 ← R1 + 73

addi R1, 73

// R1 ← R2

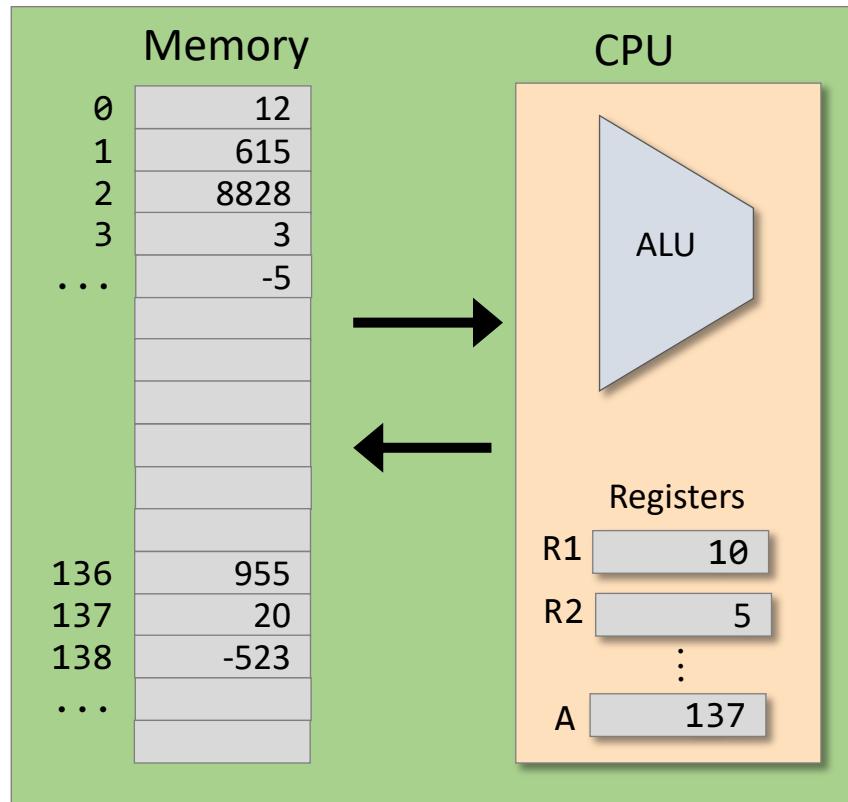
mov R1, R2

// R1 ← Memory[137]

load R1, 137

// if R1>0 goto 15

jgt R1, 15



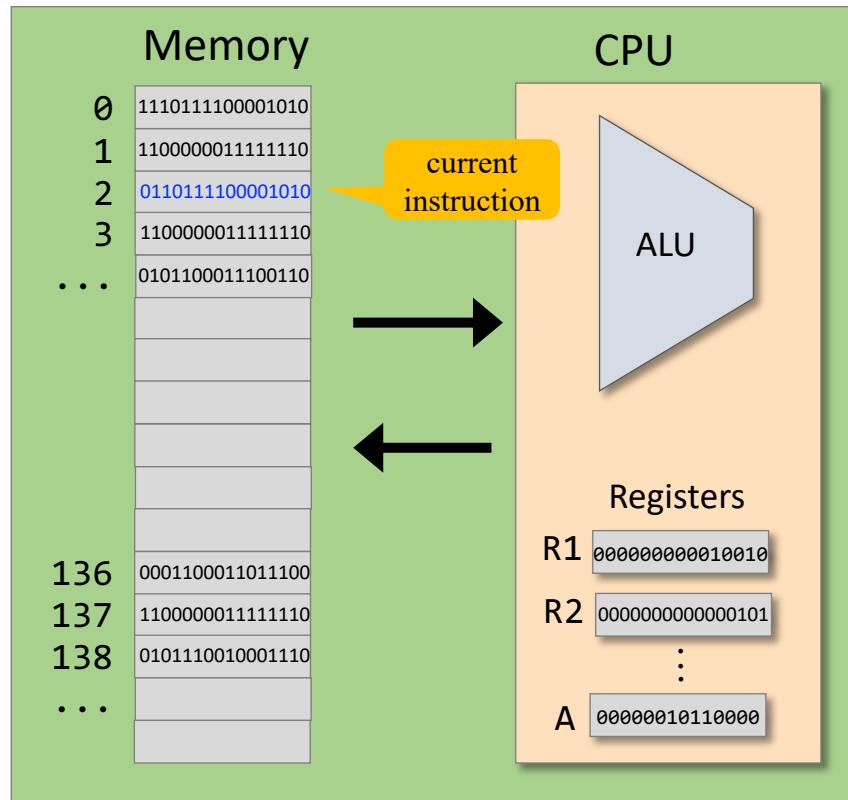
The syntax of machine languages varies across computers

The semantics is the same: Manipulating registers.

Typical operations

Which instruction should be executed next?

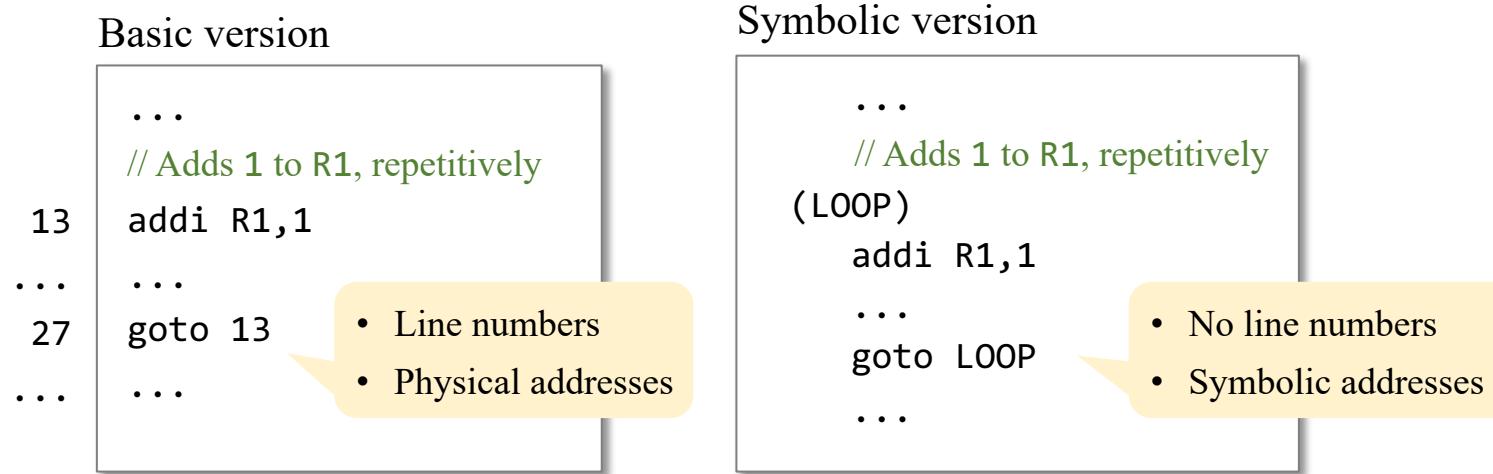
- By default, the CPU executes the *next instruction*
- Sometimes we want to “jump” to execute another instruction



Typical operations

Branching

- Execute an instruction other than the next one
- Example: Embarking on a new iteration in a loop



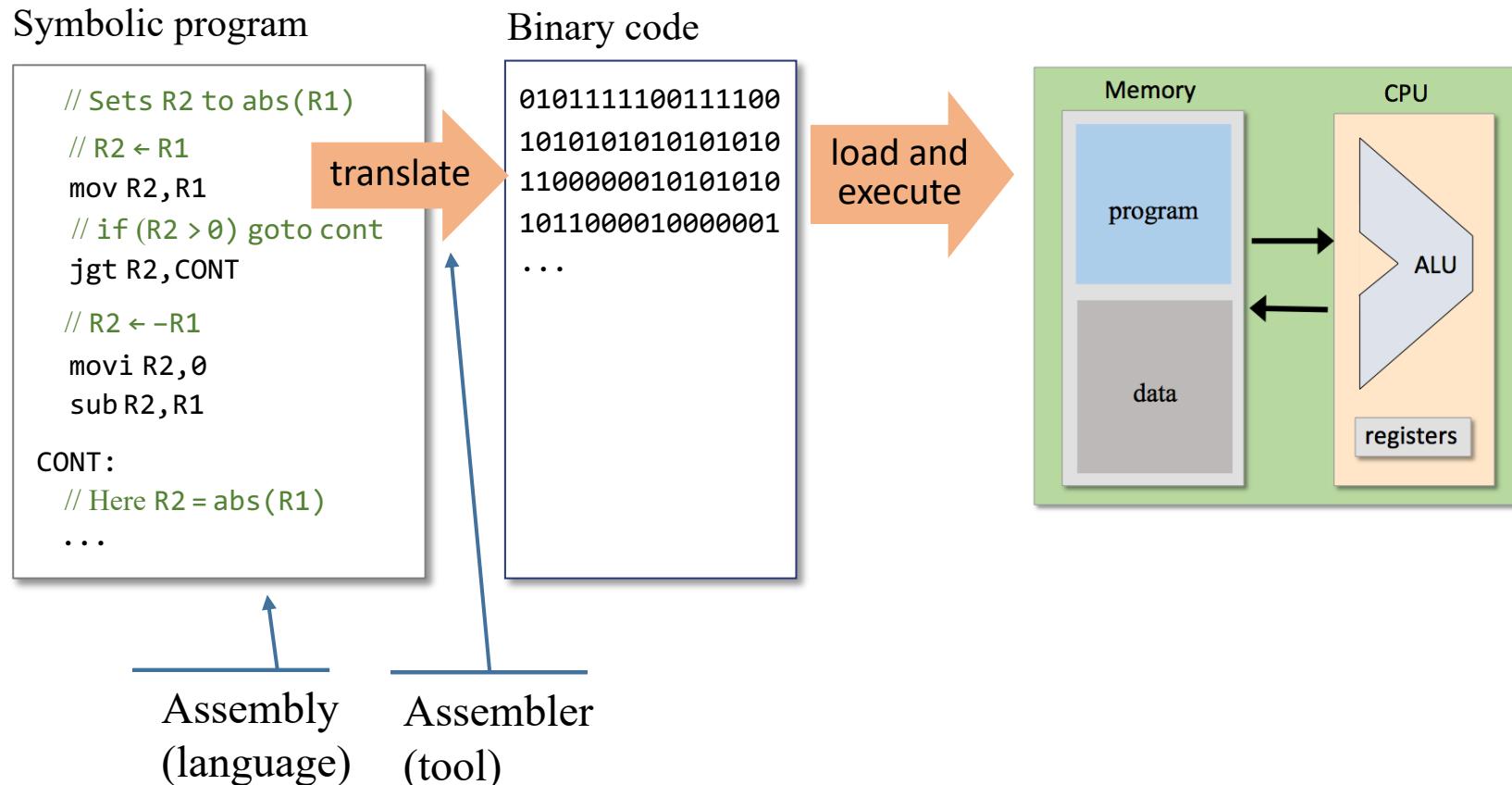
Programs with symbolic references are ...

- Easier to develop
- Readable
- Relocatable.

Program translation

Translation

Before it can be executed, a symbolic program must be translated into instructions that the computer can decode and execute.



Machine Language

Overview

✓ Machine language

→ The Hack computer

- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

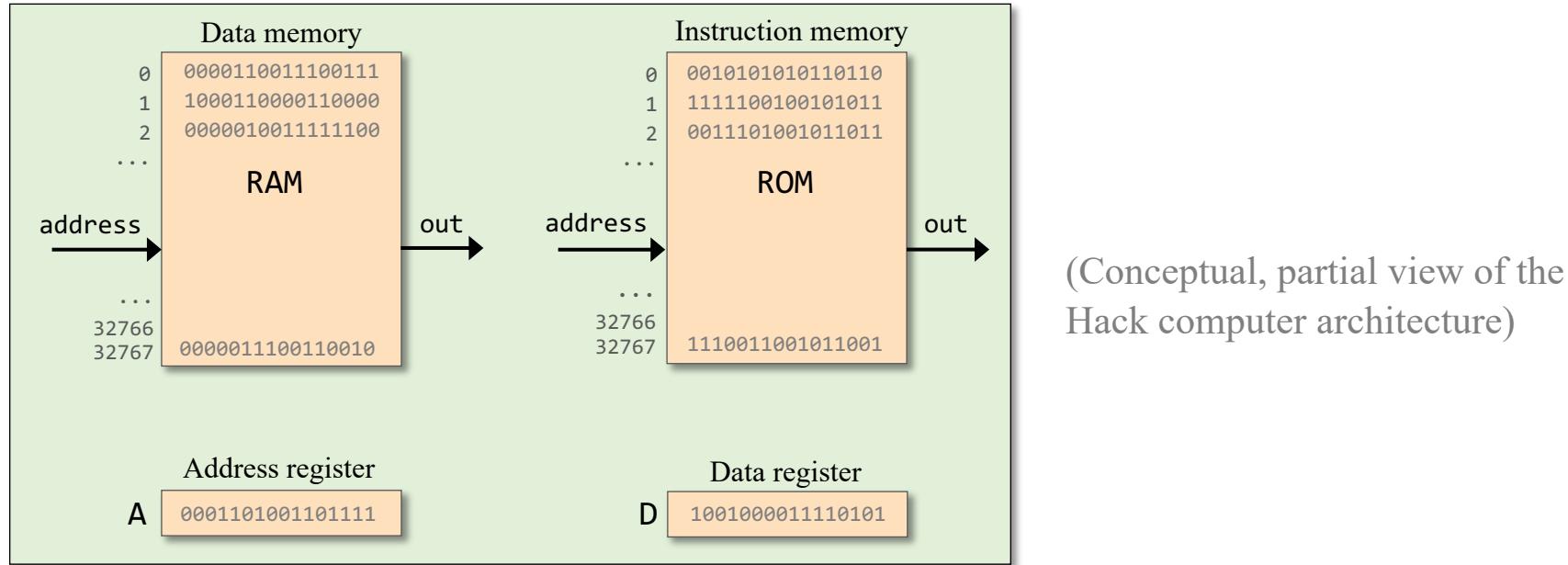
Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

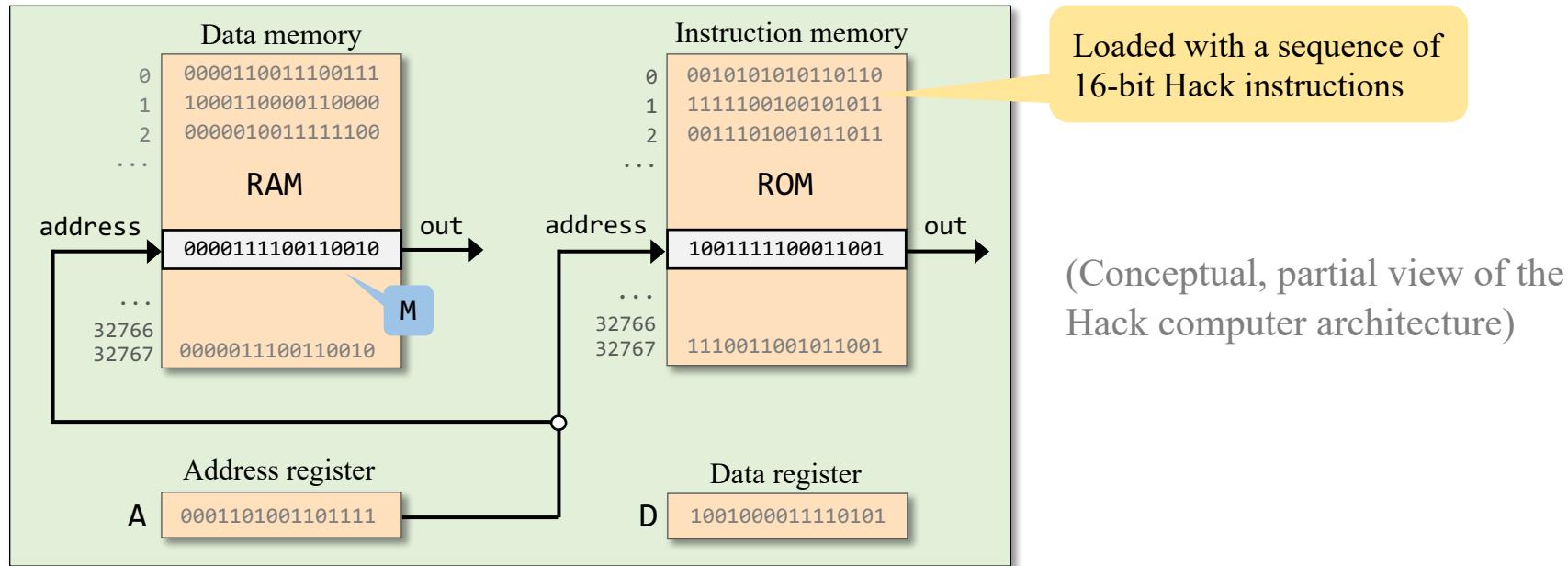
The Hack computer



(Conceptual, partial view of the
Hack computer architecture)

Hack: a 16-bit computer, featuring two memory units

Memory



RAM

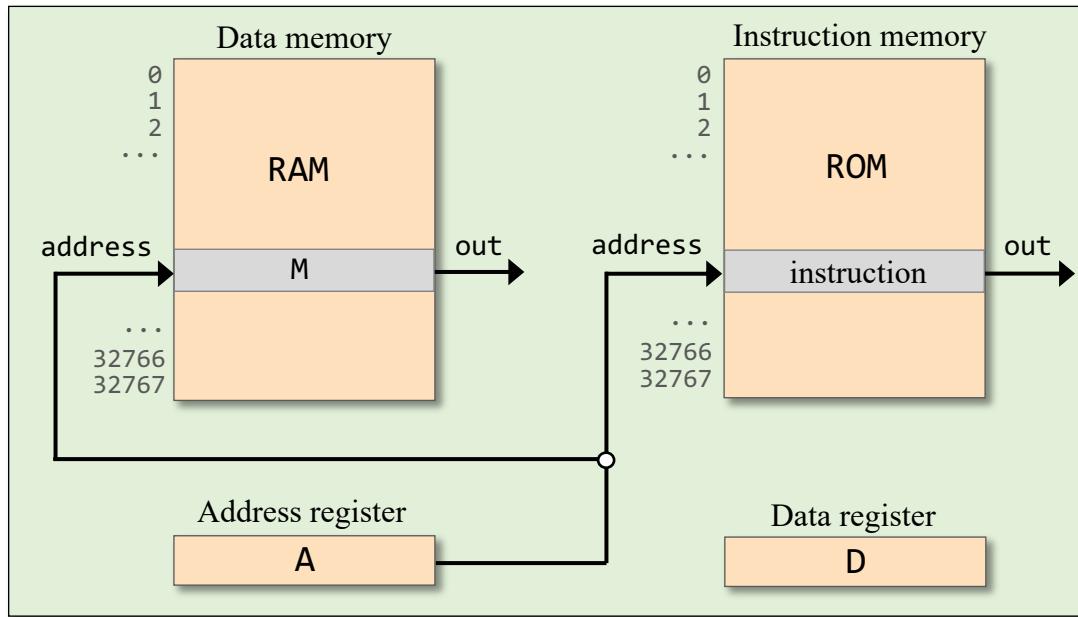
- Read-write data memory
- Addressed by the A register
- The selected memory location, $\text{RAM}[A]$, is referred to as M

ROM

- Read-only instruction memory
- Addressed by the (same) A register
- The selected memory location, $\text{ROM}[A]$, contains the *current instruction*

Should we focus on $\text{RAM}[A]$, or on $\text{ROM}[A]$?
Depends on the *current instruction* (later)

Registers



(Conceptual, partial view of the
Hack computer architecture)

D: data register

A: address register

M: selected RAM location

Machine Language

Overview

- ✓ Machine language
- ✓ The Hack computer
- The Hack instruction set
 - The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming

- Basic
- Iteration
- Pointers

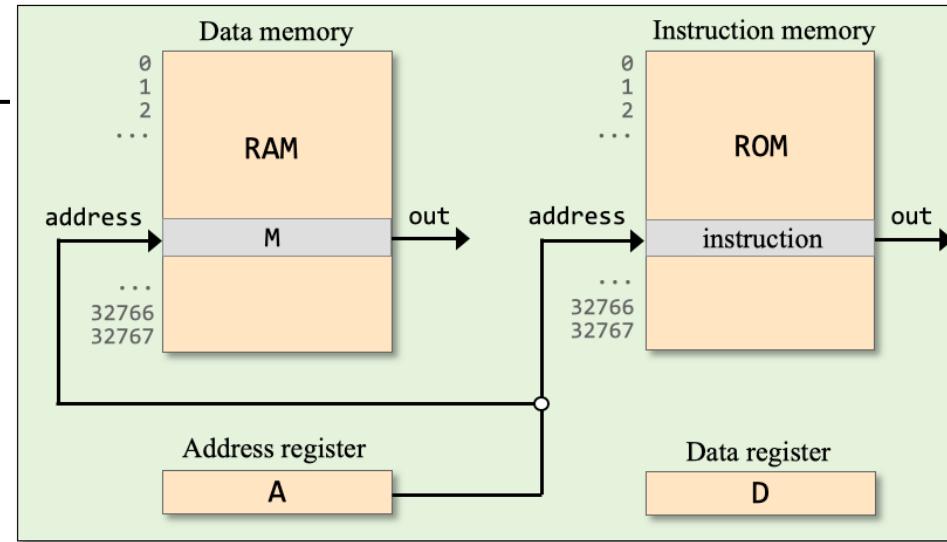
The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Hack instructions

Instruction set

- A - instruction (*address*)
- C - instruction (*compute*)

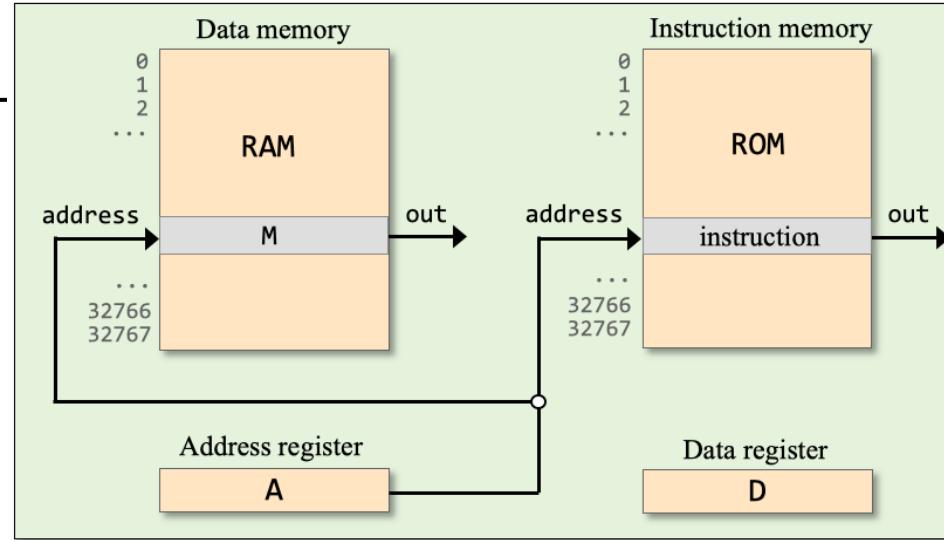


Hack instructions

Instruction set

→ A - instruction (*address*)

- C - instruction (*compute*)



Syntax:

`@xxx`

where *xxx* is a non-negative integer

Example

`@19`

Semantics

$A \leftarrow 19$

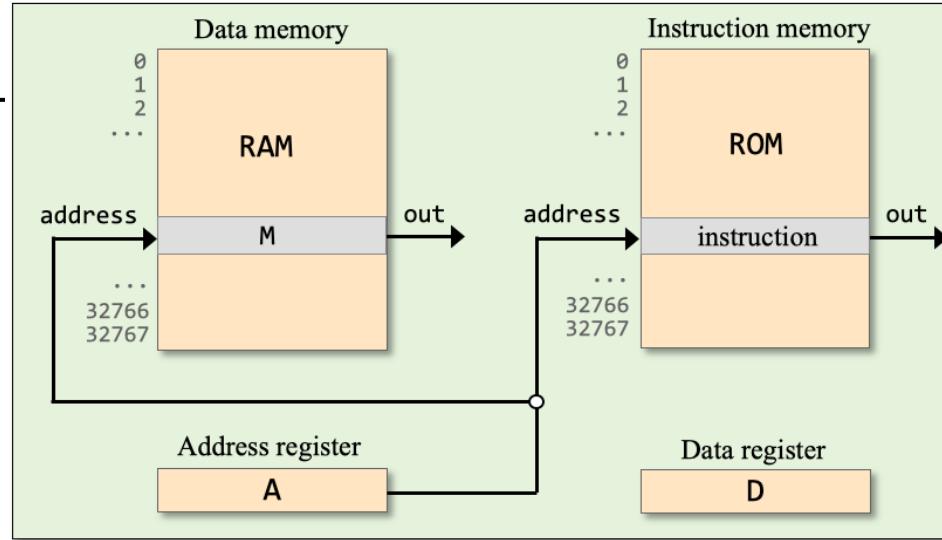
Side effects:

- $\text{RAM}[A]$ (denoted M) becomes selected
- $\text{ROM}[A]$ becomes selected

Hack instructions

Instruction set

- A - instruction (*address*)
- C - instruction (*compute*)



Syntax:

$$reg = \{0|1|-1\}$$

$$reg_1 = reg_2$$

$$reg = reg_1 op reg_2$$

where $reg = \{A|D|M\}$

where $reg_1 = \{A|D|M\}$
 $reg_2 = [-] \{A|D|M\}$

where $reg, reg_1 = \{A|D|M\}$
 $reg_2 = \{A|D|M|1\}$
 $op = \{+|-|\&| |\}$

Examples:

D=0
A=-1
M=1
...

D=A
D=M
M=-M
...

D=D+M
A=A-1
M=D+1
...

(Complete / formal syntax, later).

Hack instructions

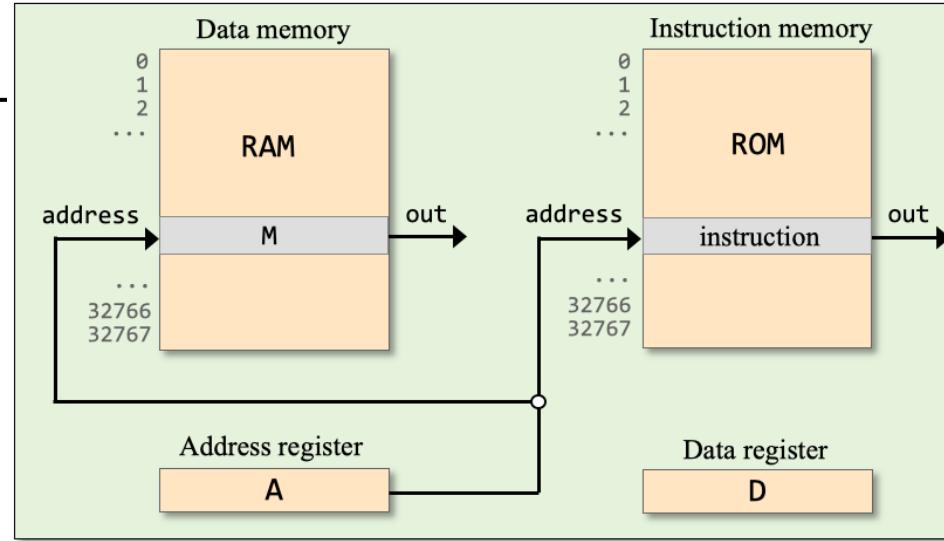
Typical instructions:

`@constant` $(A \leftarrow constant)$

$D=1$
 $D=A$
 $D=D+1$
 \dots

$D=D+A$
 $D=M$
 $D=D+A$
 $M=0$
 \dots

$M=D$
 $D=D+A$
 $M=M-D$
 \dots



Examples:

// $D \leftarrow 2$

?

The game: We show a subset of Hack instructions (top left), and practice writing code examples that use these instructions.

Hack instructions

Typical instructions:

$@constant$ $(A \leftarrow constant)$

$D=1$

$D=A$

$D=D+1$

\dots

$D=D+A$

$D=M$

$M=0$

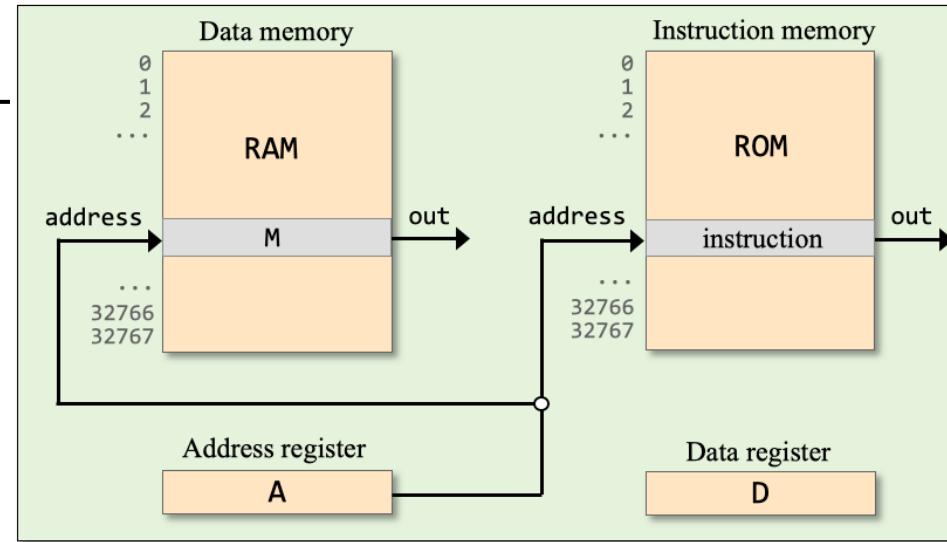
\dots

$M=D$

$D=D+A$

$M=M-D$

\dots



Examples:

// $D \leftarrow 2$
 $D=1$
 $D=D+1$

// $D \leftarrow 1954$
?

Use only the instructions
shown above

Hack instructions

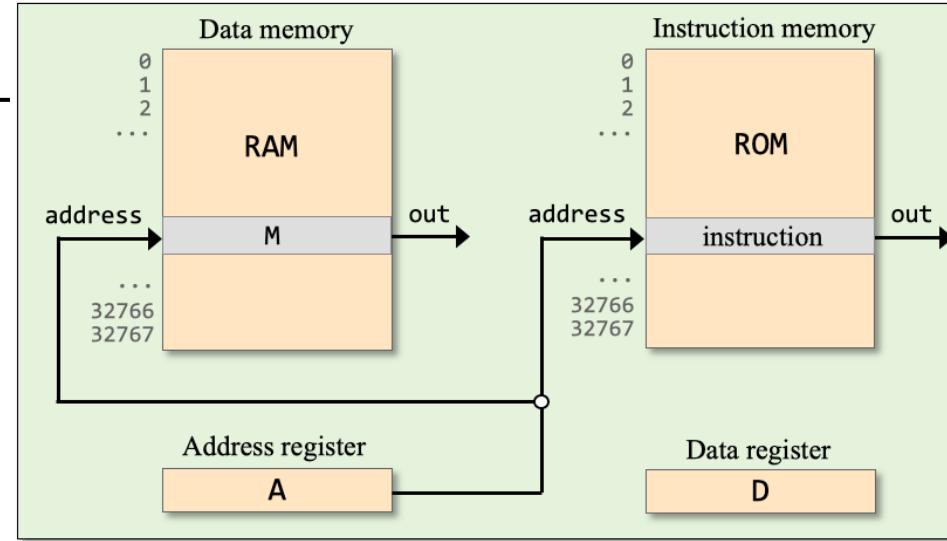
Typical instructions:

`@constant (A \leftarrow constant)`

`D=1`
`D=A`
`D=D+1`
`...`

`D=D+A`
`D=M`
`M=D`
`M=0`
`...`

`M=D`
`D=D+A`
`M=M-D`
`...`



Examples:

// $D \leftarrow 2$
`D=1`
`D=D+1`

// $D \leftarrow 1954$
`@1954`
`D=A`

// $D \leftarrow D + 23$
?

Use only the instructions shown above

Hack instructions

Typical instructions:

$@constant$ ($A \leftarrow constant$)

$D=1$

$D=A$

$D=D+1$

\dots

$D=D+A$

$D=M$

$M=0$

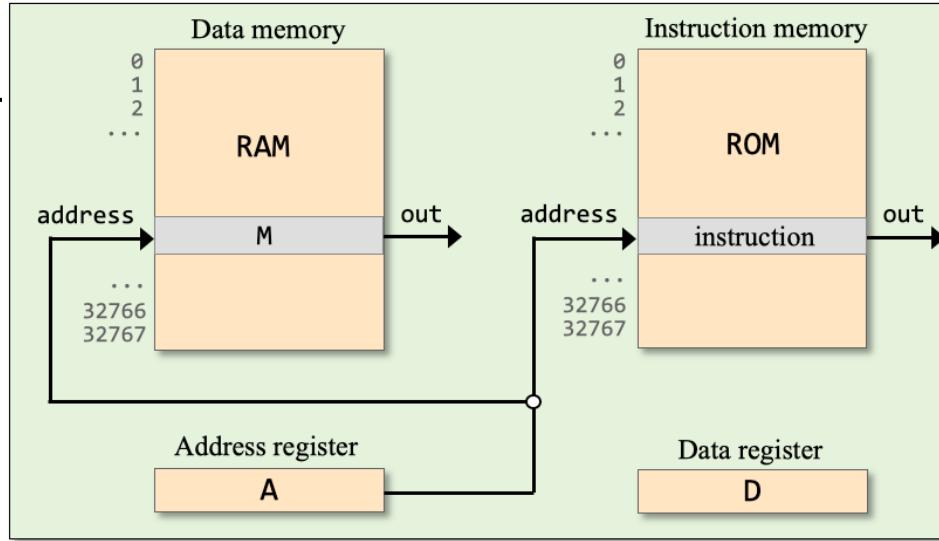
\dots

$M=D$

$D=D+A$

$M=M-D$

\dots



Examples:

// $D \leftarrow 2$
 $D=1$
 $D=D+1$

// $D \leftarrow 1954$
 $@1954$
 $D=A$

// $D \leftarrow D + 23$
 $@23$
 $D=D+A$

Observation

In all these examples, we used A as a *data register*:

The addressing side-effects of A are ignored.

Hack instructions

Typical instructions:

`@constant` $(A \leftarrow constant)$

$D=1$

$D=A$

$D=D+1$

\dots

$D=D+A$

$D=M$

$M=0$

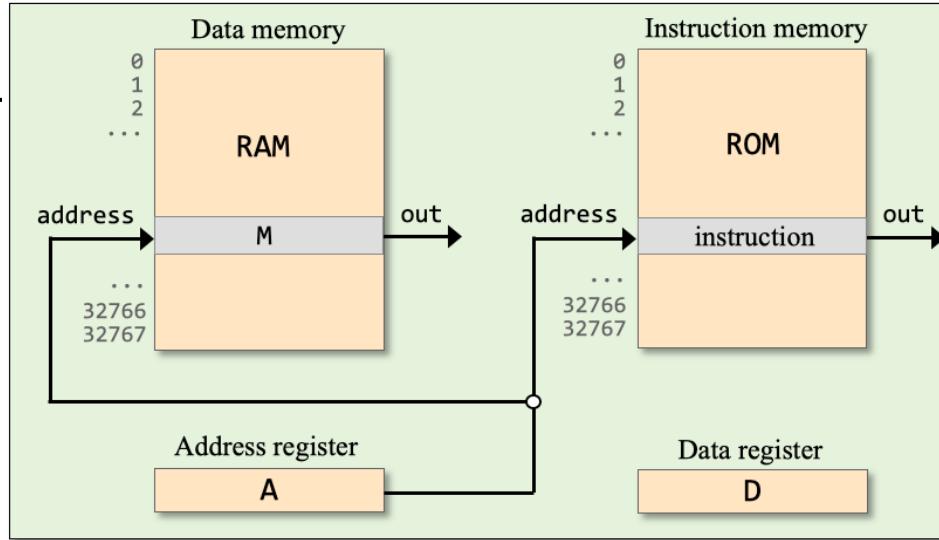
\dots

$M=D$

$D=D+A$

$M=M-D$

\dots



More examples:

// $RAM[100] \leftarrow 0$
`@100`
 $M=0$

// $RAM[100] \leftarrow 17$
`@17`
 $D=A$
`@100`
 $M=D$

- First pair of instructions:
A is used as a *data register*
- Second pair of instructions:
A is used as an *address register*

Hack instructions

Typical instructions:

`@constant (A \leftarrow constant)`

`D=1`

`D=A`

`D=D+1`

`...`

`D=D+A`

`D=M`

`M=0`

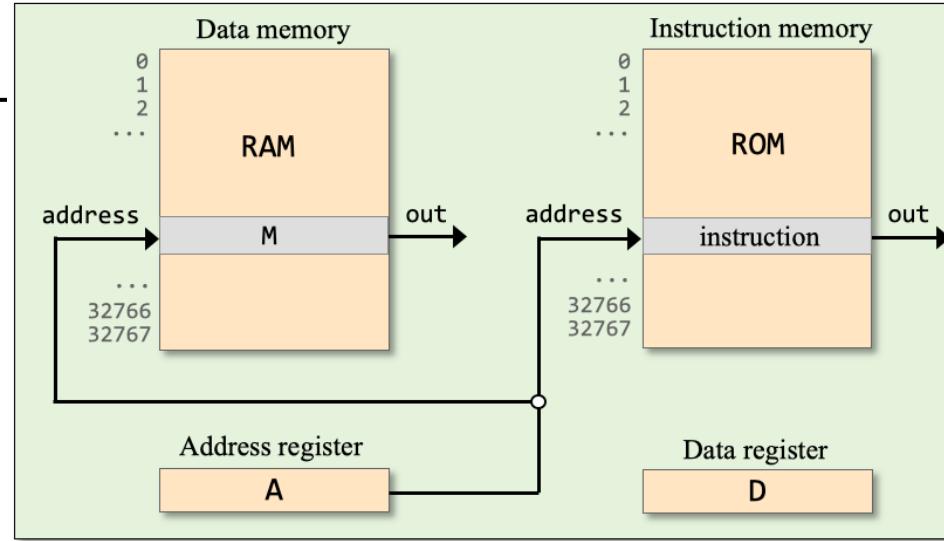
`...`

`M=D`

`D=D+A`

`M=M-D`

`...`



More examples:

// RAM[100] \leftarrow 0
`@100`
`M=0`

// RAM[100] \leftarrow 17
`@17`
`D=A`
`@100`
`M=D`

// RAM[100] \leftarrow RAM[200]
?

Hack instructions

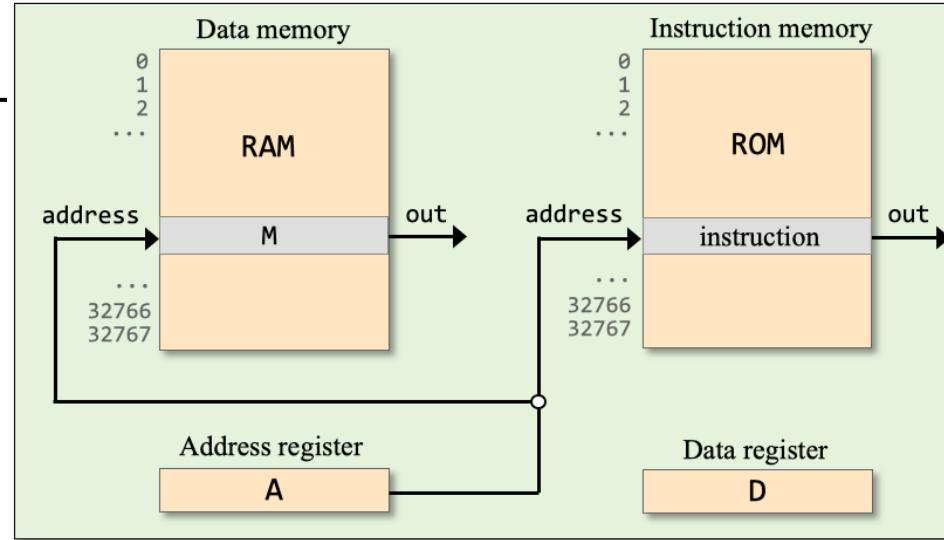
Typical instructions:

`@constant` $(A \leftarrow constant)$

$D=1$
 $D=A$
 $D=D+1$
 \dots

$D=D+A$
 $D=M$
 $D=D+A$
 $M=0$
 \dots

$M=D$
 $D=D+A$
 $M=M-D$
 \dots



More examples:

// $RAM[100] \leftarrow 0$
`@100`
 $M=0$

// $RAM[100] \leftarrow 17$
`@17`
 $D=A$
`@100`
 $M=D$

// $RAM[100] \leftarrow RAM[200]$
`@200`
 $D=M$
`@100`
 $M=D$

When we want to operate on a memory location, we typically need a pair of instructions:

- A-instruction: Selects a memory location
- C-instruction: Operates on the selected location.

Hack instructions

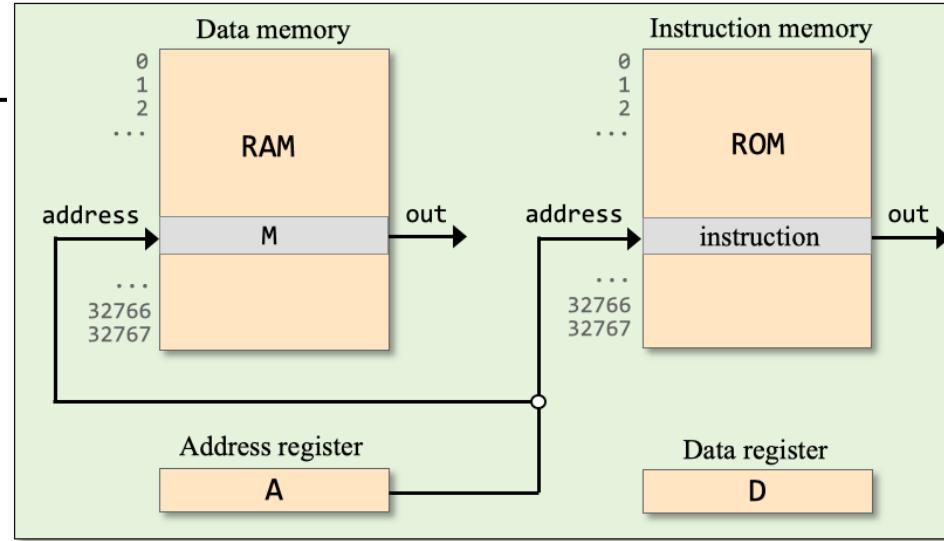
Typical instructions:

$@constant$ $(A \leftarrow constant)$

$D=1$
 $D=A$
 $D=D+1$
 \dots

$D=D+A$
 $D=M$
 $D=D+A$
 $M=0$
 \dots

$M=D$
 $D=D+A$
 $M=M-D$
 \dots



// $RAM[3] \leftarrow RAM[3] - 15$

?

Use only the instructions shown above

Hack instructions

Typical instructions:

$@constant$ $(A \leftarrow constant)$

$D=1$

$D=A$

$D=D+1$

\dots

$D=D+A$

$D=M$

$M=D$

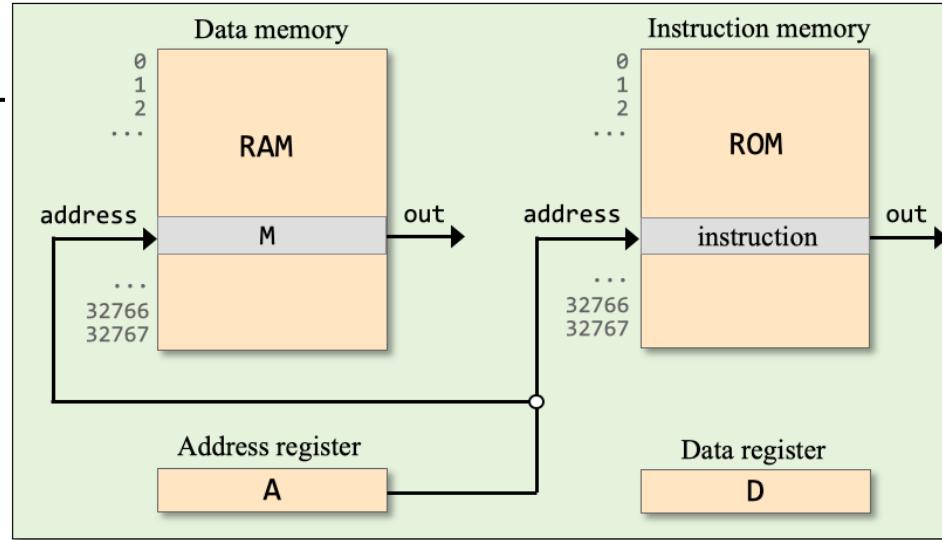
\dots

$M=D$

$D=D+A$

$M=0$

\dots



```
// RAM[3] ← RAM[3] - 15  
@15  
D=A  
@3  
M=M-D
```

Use only the instructions shown above

```
// RAM[3] ← RAM[4] + 1
```

?

Hack instructions

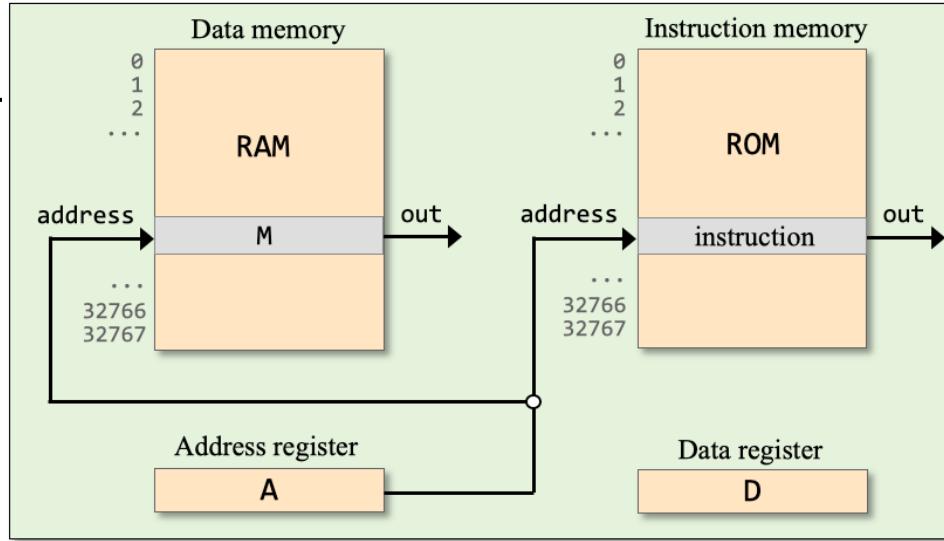
Typical instructions:

$@constant$ $(A \leftarrow constant)$

$D=1$
 $D=A$
 $D=D+1$
 \dots

$D=D+A$
 $D=M$
 $D=D+A$
 $M=0$
 \dots

$M=D$
 $D=D+A$
 $M=M-D$
 \dots



```
// RAM[3] ← RAM[3] - 15  
@15  
D=A  
@3  
M=M-D
```

Use only the instructions shown above

```
// RAM[3] ← RAM[4] + 1  
@4  
D=M+1  
@3  
M=D
```

Hack instructions

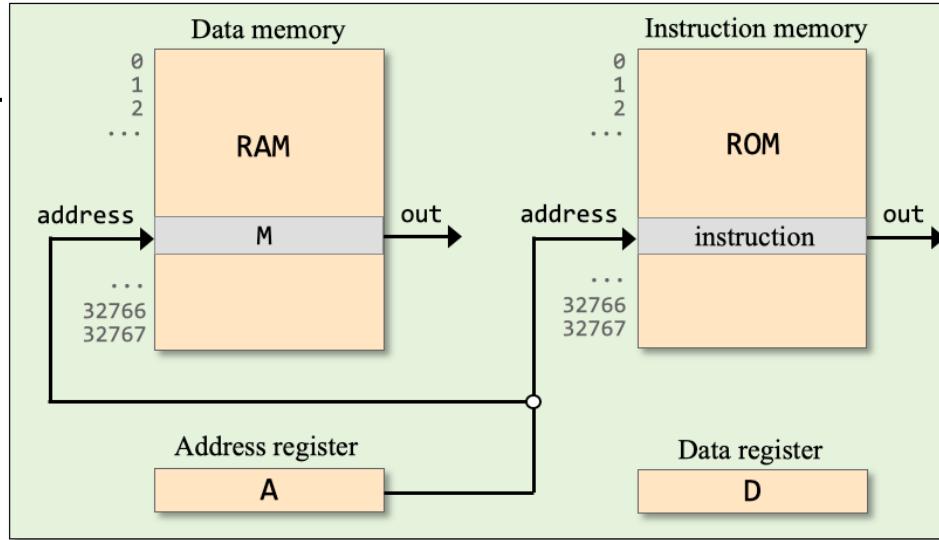
Typical instructions:

$@constant$ $(A \leftarrow constant)$

$A=1$
 $D=-1$
 $M=0$
 \dots

$A=M$
 $D=M$
 $M=D$
 \dots

$A=D-A$
 $D=D+A$
 $M=D$
 \dots



Add.asm

```
// Computes: RAM[2] = RAM[0] + RAM[1] + 17
```

?

Use only the instructions
shown above

Hack instructions

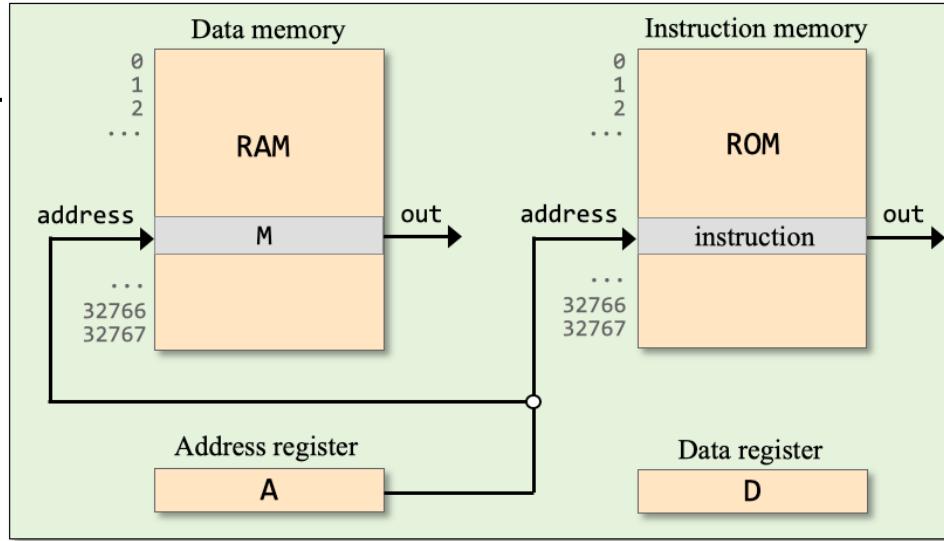
Typical instructions:

$@constant$ $(A \leftarrow constant)$

$A=1$
 $D=-1$
 $M=0$
 \dots

$A=M$
 $D=M$
 $M=D$
 \dots

$A=D-A$
 $D=D+A$
 $M=D$
 $D=D+M$
 \dots



Add.asm

```
// Computes: RAM[2] = RAM[0] + RAM[1] + 17  
  
// D = RAM[0]  
@0  
D=M  
  
// D = D + RAM[1]  
@1  
D=D+M  
  
// D = D + 17  
@17  
D=D+A  
  
// RAM[2] = D  
@2  
M=D
```

Hack instructions

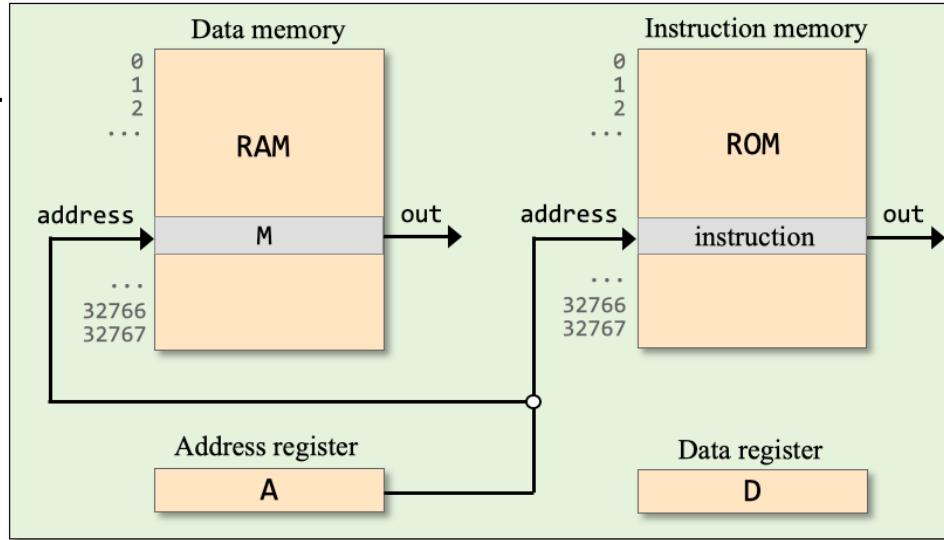
Typical instructions:

$@constant$ $(A \leftarrow constant)$

$A=1$
 $D=-1$
 $M=0$
 \dots

$A=M$
 $D=M$
 $M=D$
 \dots

$A=D-A$
 $D=D+A$
 $M=D$
 $D=D+M$
 \dots



Add.asm

```
// Computes: RAM[2] = RAM[0] + RAM[1] + 17  
  
// D = RAM[0]  
@0  
D=M  
  
// D = D + RAM[1]  
@1  
D=D+M  
  
// D = D + 17  
@17  
D=D+A  
  
// RAM[2] = D  
@2  
M=D
```

How can we tell that a given program
actually works?

→ Testing / simulating

- Formal verification

Machine Language

Overview

- ✓ Machine language
 - ✓ The Hack computer
 - ✓ The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming

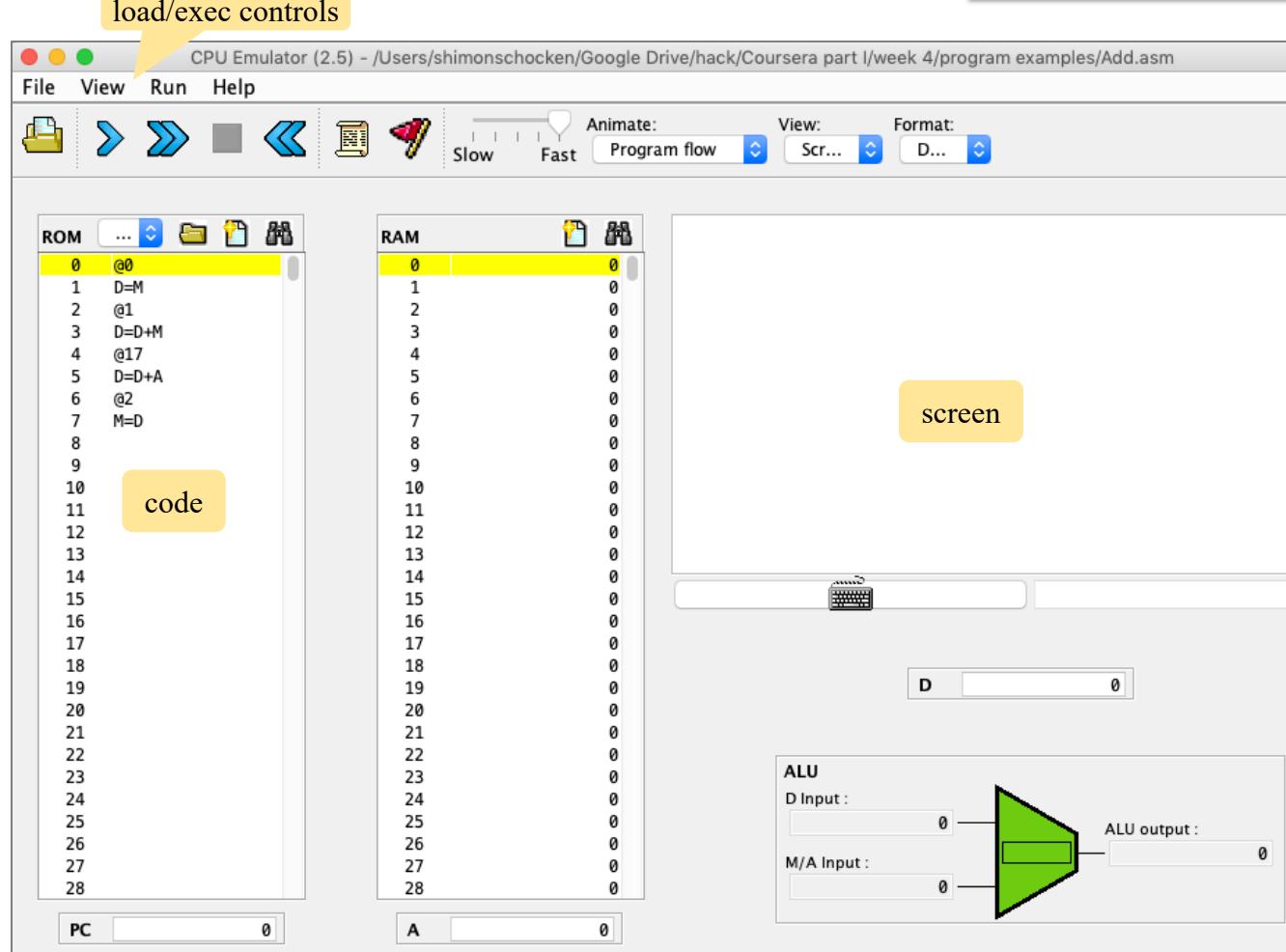
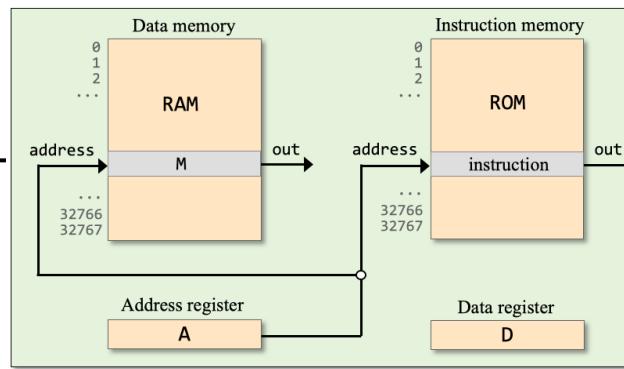
- Basic
- Iteration
- Pointers

The Hack Language

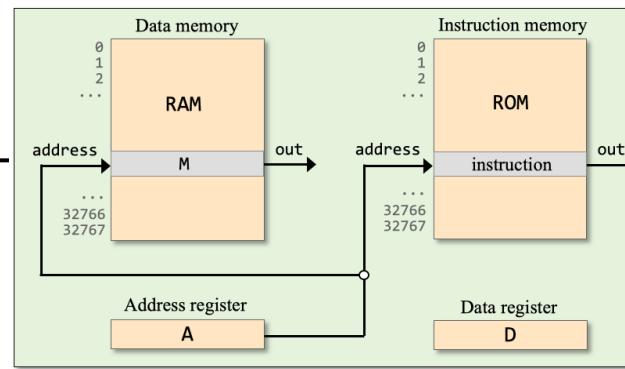
- Symbolic
- Binary
- Output
- Input
- Project 4

The CPU emulator

- A Java program that emulates the Hack CPU
- On your PC (nand2tetris/tools)



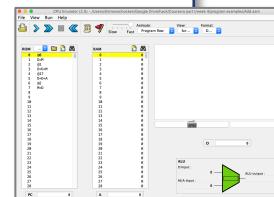
The CPU emulator



Add.asm (example)

```
// Computes: RAM[2] = RAM[0] + RAM[1] + 17  
// D = RAM[0]  
@0  
D=M  
// D = D + RAM[1]  
@1  
D=D+M  
// D = D + 17  
@17  
D=D+A  
// RAM[2] = D  
@2  
M=D
```

Load into the
CPU emulator



Binary

```
0000000000000000  
1000010010001101  
0000000000000001  
1010011001100001  
0000000000100001  
1001111100110011  
0000000000000010  
1110010010010011
```

Execute in the
CPU emulator

When loading a symbolic program into our CPU emulator, the emulator translates it into binary code (using a built-in assembler).

The CPU emulator



Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator



Symbolic programming



- Variables
- Labels

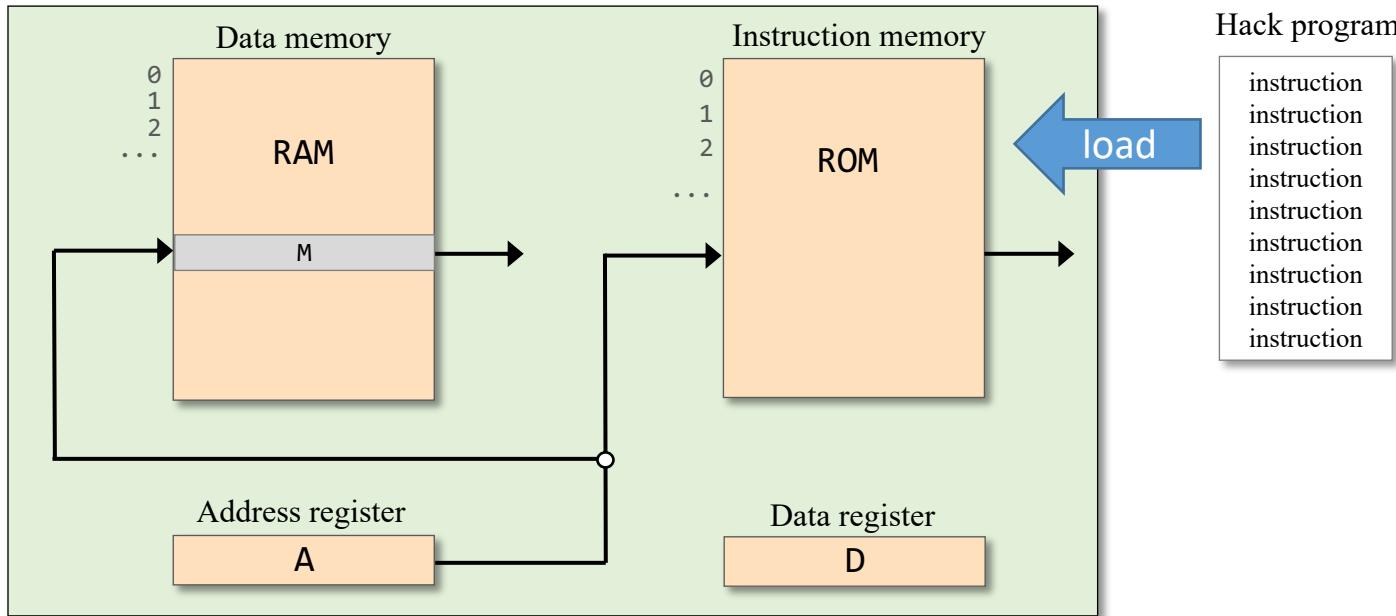
Low Level Programming

- Basic
- Iteration
- Pointers

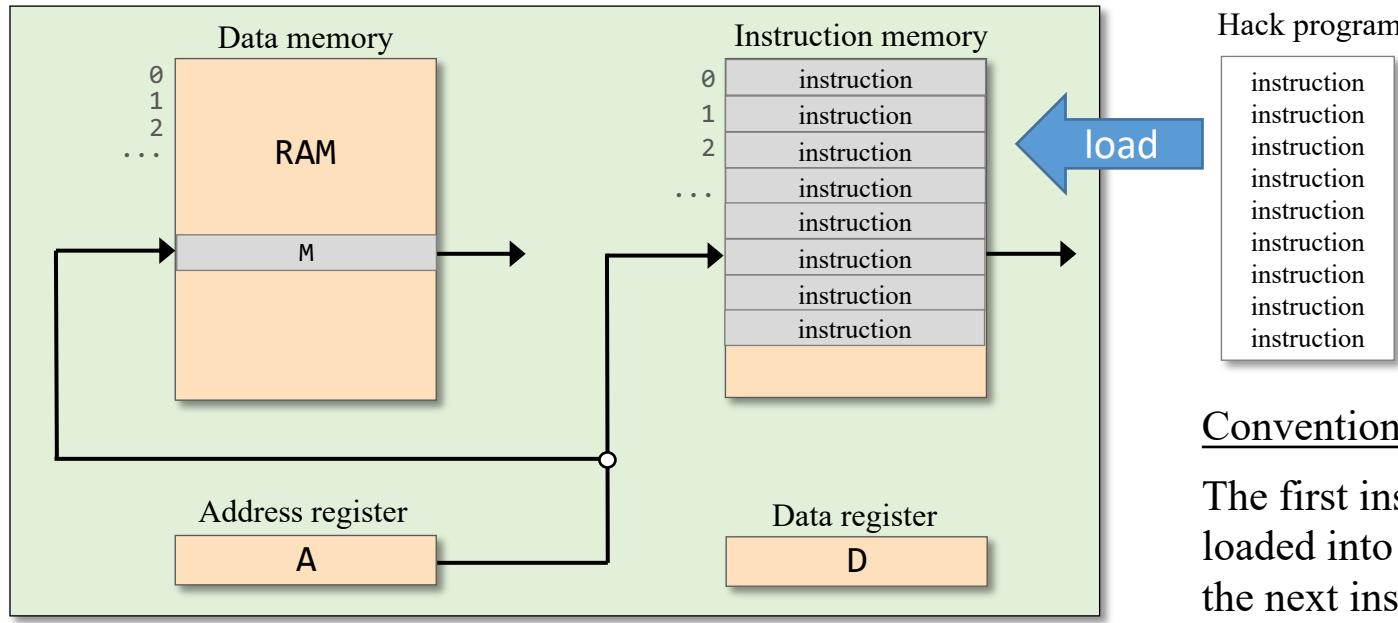
The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Loading a program



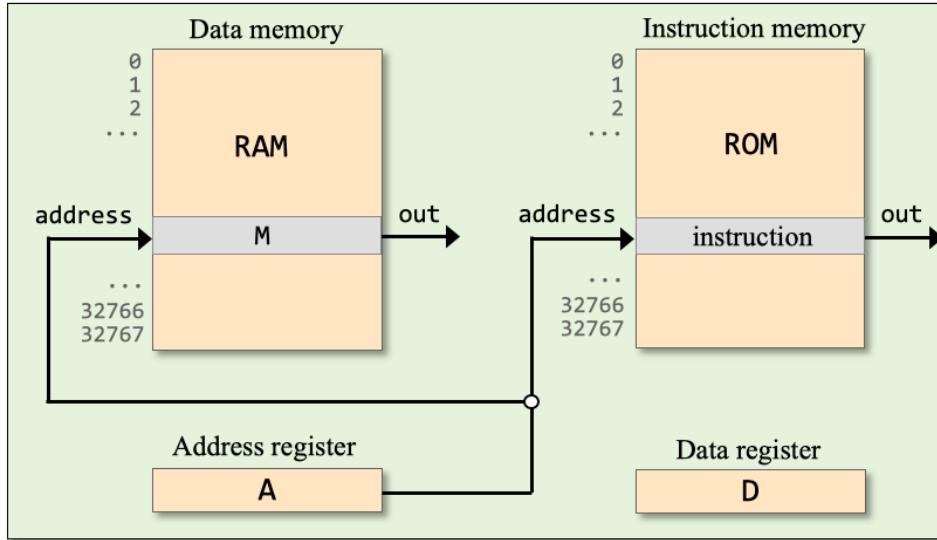
Loading a program



Convention:

The first instruction is loaded into address 0, the next instruction into address 1, and so on.

Branching



Unconditional branching
example (pseudocode)

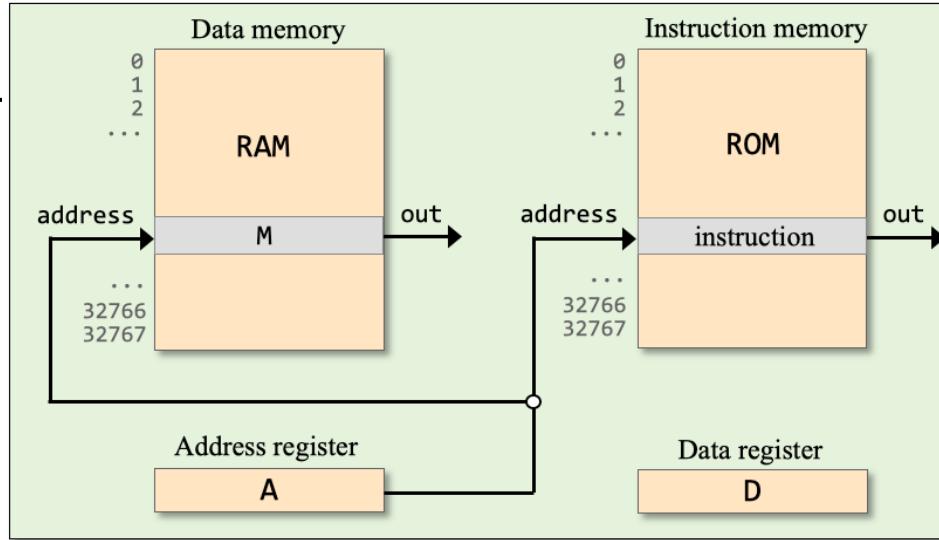
```
0  instruction
1  instruction
2  instruction
3  instruction
4  instruction
5  goto 7
6  instruction
7  instruction
8  instruction
9  goto 2
10 instruction
11 ...
```

Flow of control:
0,1,2,3,4,
7,8,9,
2,3,4,
7,8,9,
2,3,4,
...

Branching

Conditional branching
example (pseudocode)

```
0 instruction
1 instruction
2 instruction
3 instruction
4 if (condition) goto 7
5 instruction
6 instruction
7 instruction
8 instruction
9 instruction
...
...
```

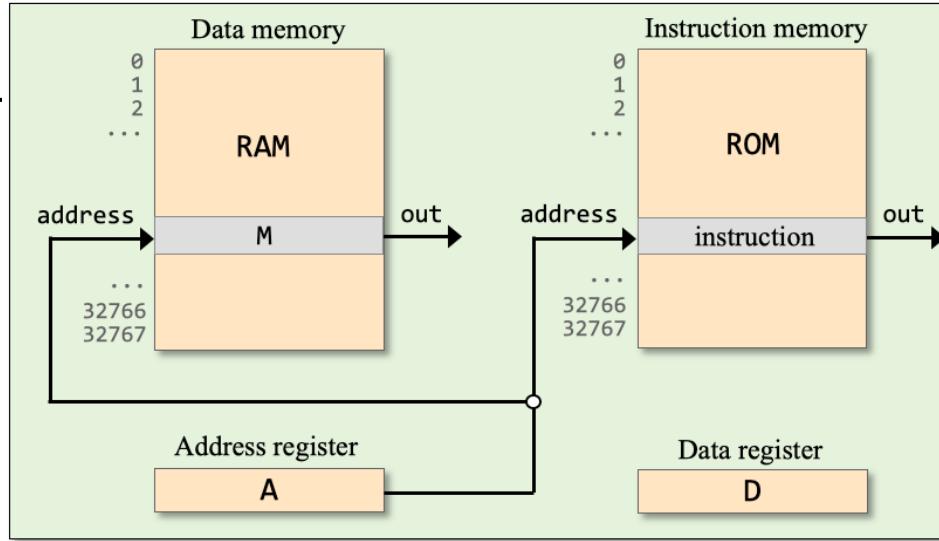


Flow of control:

0,1,2,3,4,
if *condition* is true
 7,8,9,...
else
 5,6,7,8,9,...

Branching

Branching in the Hack language:



Example (pseudocode):

```
0 instruction  
1 instruction  
2 goto 6  
3 instruction  
4 instruction  
5 instruction  
6 instruction  
7 instruction  
...  
...
```

In Hack:

```
...  
// goto 6  
@6  
0;JMP  
...
```

Syntax:

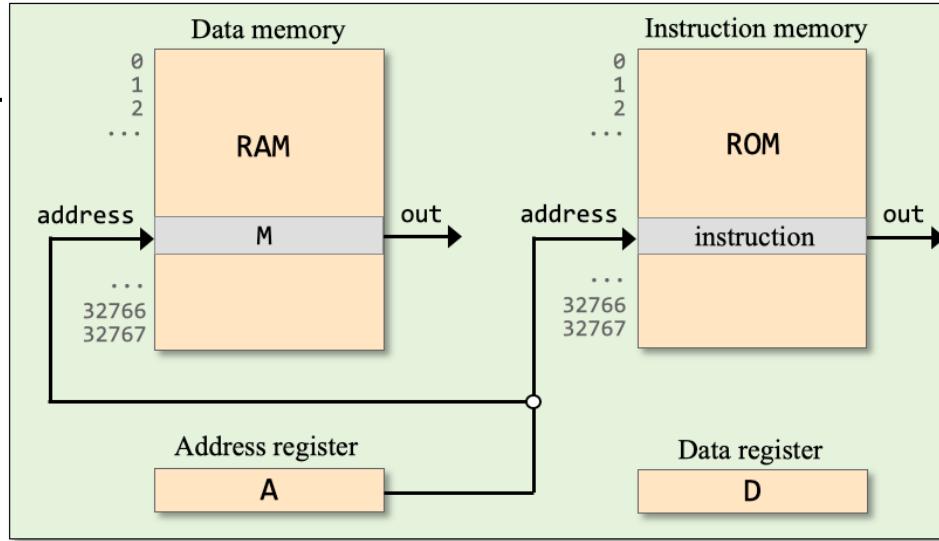
- Use an A-instruction to select an address
- Use a C-instruction to jump to that address

Semantics of 0;JMP

Jump to execute the instruction stored in ROM[A]
(the 0; prefix is a meaningless syntax convention)

Branching

Branching in the Hack language:



Example (pseudocode):

```
0 instruction
1 instruction
2 if (D>0) goto 6
3 instruction
4 instruction
5 instruction
6 instruction
7 instruction
...
```

In Hack:

```
...
// if (D > 0) goto 6
@6
D;JGT
...
```

Typical branching instructions:

D;JGT // if D > 0 jump
D;JGE // if D ≥ 0 jump
D;JLT // if D < 0 jump
D;JLE // if D ≤ 0 jump
D;JEQ // if D = 0 jump
D;JNE // if D ≠ 0 jump
0;JMP // jump

to the
instruction
stored in
ROM[A]

D can be replaced with any expression that the ALU computes: D+1, M, M-1, etc.

Branching

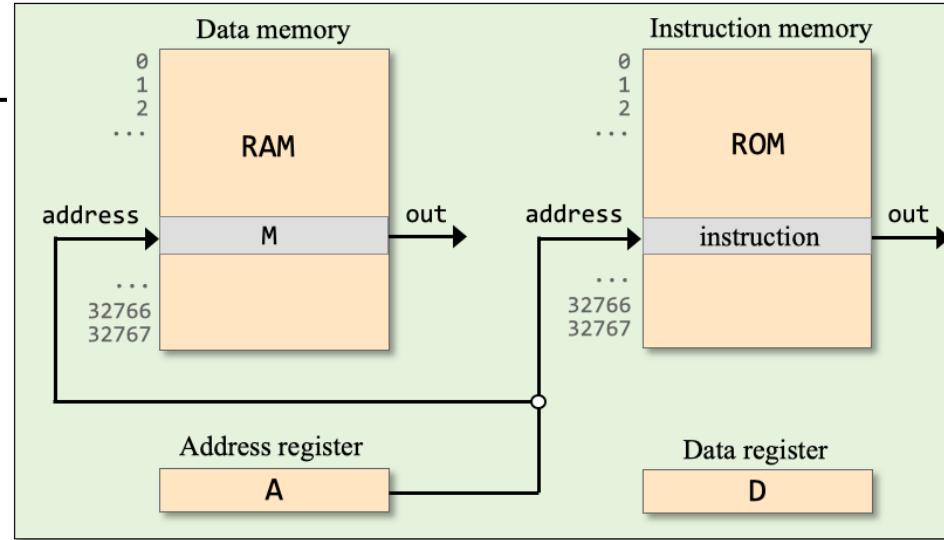
Typical instructions:

`@constant` $(A \leftarrow \text{constant})$

$A = 1$
 $D = -1$
 $M = 0$
...

$A = M$
 $D = A$
 $M = D$
...

$D = D - A$
 $A = A - 1$
 $M = D + 1$
...



// if ($D = 0$) goto 300

?

Typical branching instructions:

$D; JGT$ // if $D > 0$ jump
 $D; JGE$ // if $D \geq 0$ jump
 $D; JLT$ // if $D < 0$ jump
 $D; JLE$ // if $D \leq 0$ jump
 $D; JEQ$ // if $D = 0$ jump
 $D; JNE$ // if $D \neq 0$ jump
 $0; JMP$ // jump

to the instruction stored in $\text{ROM}[A]$

Use only the instructions shown in this slide

Branching

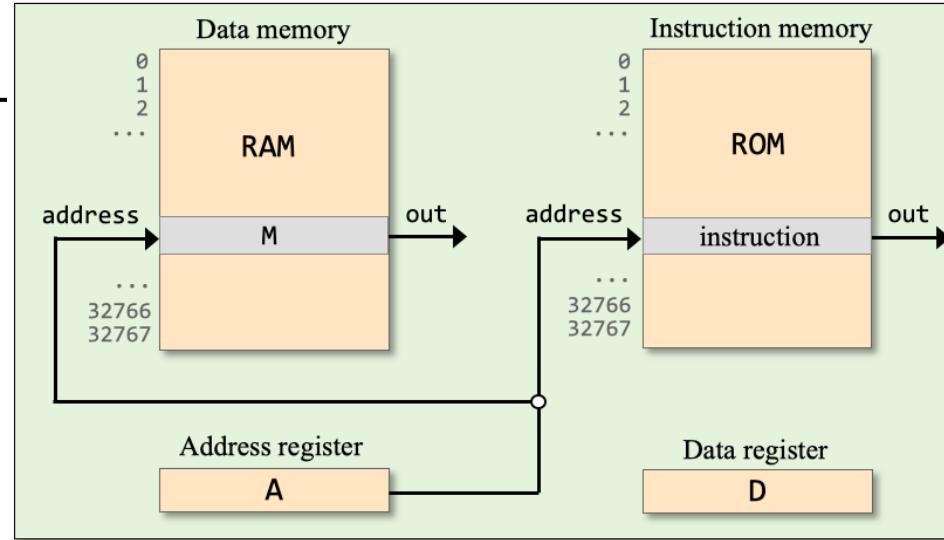
Typical instructions:

`@constant` $(A \leftarrow constant)$

$A = 1$
 $D = -1$
 $M = 0$
...

$A = M$
 $D = A$
 $M = D$
...

$D = D - A$
 $A = A - 1$
 $M = D + 1$
...



```
// if (D = 0) goto 300
@300
D;JEQ
```

Typical branching instructions:

`D;JGT // if $D > 0$ jump`
`D;JGE // if $D \geq 0$ jump`
`D;JLT // if $D < 0$ jump`
`D;JLE // if $D \leq 0$ jump`
`D;JEQ // if $D = 0$ jump`
`D;JNE // if $D \neq 0$ jump`
`0;JMP // jump`

to the
instruction
stored in
ROM[A]

Use only the instructions shown in this slide

Branching

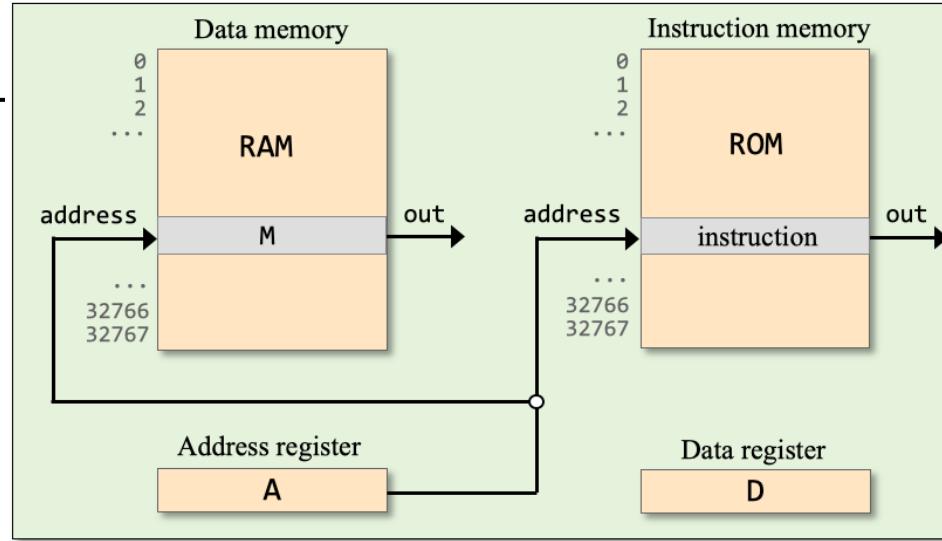
Typical instructions:

$@constant$ $(A \leftarrow constant)$

$A=1$
 $D=-1$
 $M=0$
 \dots

$A=M$
 $D=A$
 $M=D$
 \dots

$D=D-A$
 $A=A-1$
 $M=D+1$
 \dots



```
// if (RAM[3] < 100) goto 12
```

?

Typical branching instructions:

$D;JGT$ // if $D > 0$ jump
 $D;JGE$ // if $D \geq 0$ jump
 $D;JLT$ // if $D < 0$ jump
 $D;JLE$ // if $D \leq 0$ jump
 $D;JEQ$ // if $D = 0$ jump
 $D;JNE$ // if $D \neq 0$ jump
 $0;JMP$ // jump

to the instruction stored in ROM[A]

Use only the instructions shown in this slide

Branching

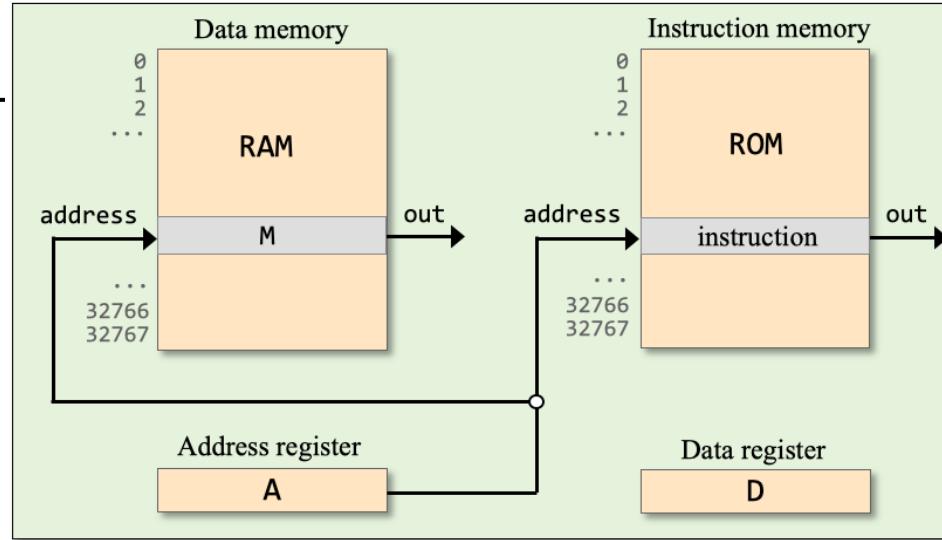
Typical instructions:

`@constant` $(A \leftarrow constant)$

$A=1$
 $D=-1$
 $M=0$
...

$A=M$
 $D=A$
 $M=D$
...

$D=D-A$
 $A=A-1$
 $M=D+1$
...



```
// if (RAM[3] < 100) goto 12
// D = RAM[3] - 100
@3
D=M
@100
D=D-A
// if (D < 0) goto 12
@12
D;JLT
```

Typical branching instructions:

`D;JGT // if $D > 0$ jump`

`D;JGE // if $D \geq 0$ jump`

`D;JLT // if $D < 0$ jump`

`D;JLE // if $D \leq 0$ jump`

`D;JEQ // if $D = 0$ jump`

`D;JNE // if $D \neq 0$ jump`

`0;JMP // jump`

to the
instruction
stored in
ROM[A]

Use only the instructions shown in this slide

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming



- Labels

Low Level Programming

- Basic
- Iteration
- Pointers

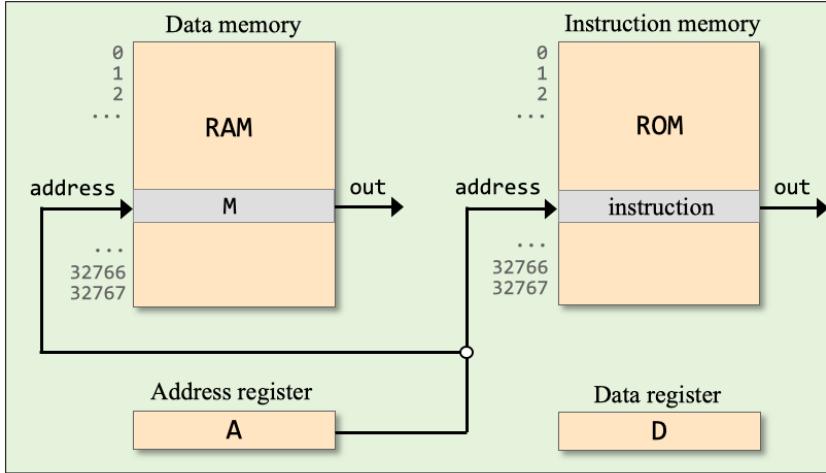
The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Hack instructions (review)

→ A - instruction

- C - instruction



Syntax:

`@xxx`

where *xxx* is either a constant, or a symbol bound to a constant

Examples:

`@19`

`@sym`

Semantics:

$A \leftarrow 19$

$A \leftarrow$ the number that *sym* is bound to

This idiom can be used for realizing:

- Variables
- Labels

Variables

Pseudocode (example)

```
...  
i = 1  
sum = 0  
...  
sum = sum + i  
i = i + 1  
...
```

write



Hack assembly

```
...  
// i = 1  
@i  
M=1  
// sum = 0  
@sum  
M=0  
...  
  
// sum = sum + i  
@i  
D=M  
@sum  
M=D+M  
// i = i + 1  
@i  
M=M+1  
...
```

Symbolic programming

- The code writer is allowed to create and use symbolic variables, as needed
- We assume that there is an agent who knows how to bind these symbols to sensible RAM addresses

This agent is the *assembler*

For example

- If the assembler will bind *i* to 16 and *sum* to 17, every instruction *@i* and *@sum* will end up selecting *RAM[16]* and *RAM[17]*
- Invisible to the code writer
- The result: a powerful, low-level, *variables abstraction*.

Variables

Typical instructions:

`@constant`

`A ← constant`

`@symbol`

`A ← the constant which is bound to symbol`

`D=0`

`M=1`

`D=-1`

`M=0`

`...`

`D=M`

`A=M`

`M=D`

`D=A`

`...`

`D=D+A`

`D=A+1`

`D=D+M`

`M=M-1`

`...`

`// sum = 0`

?

`// x = 512`

?

`// n = n - 1`

?

`// sum = sum + x`

?

Using only the instructions
shown above

Variables

Typical instructions:

`@constant`

`A ← constant`

`@symbol`

`A ← the constant which is bound to symbol`

`D=0`

`M=1`

`D=-1`

`M=0`

`...`

`D=M`

`A=M`

`M=D`

`D=A`

`...`

`D=D+A`

`D=A+1`

`D=D+M`

`M=M-1`

`...`

`// sum = 0`

`@sum`

`M=0`

`// x = 512`

`@512`

`D=A`

`@x`

`M=D`

`// n = n - 1`

`@n`

`M=M-1`

`// sum = sum + x`

`@sum`

`D=M`

`@x`

`D=D+M`

`@sum`

`M=D`

Using only the instructions
shown above

Variables

Pre-defined symbols in the Hack language

<u>symbol</u>	<u>value</u>	RAM
R0	0	0
R1	1	1
R2	2	2
...
R15	15	15
		16
		17
		...
		32767

As if we have 16 “built-in variables” named R0...R15
We sometimes call them “virtual registers”

Example:

```
// Sets R1 to 2 * R0
// Usage: Enter a value in R0

@R0
D=M
@R1
M=D
M=D+M
```

The use of R0, R1, ... (instead of physical addresses 0, 1, ...) makes Hack code easier to document, write, and debug.

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- ✓ Control
- ✓ Variables



Labels

Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Labels

Example (pseudocode)

```
i = 1000
LOOP:
    if (i = 0) goto CONT
    i = i - 1
    goto LOOP
CONT:
    ...
    ...
```



Hack assembly

```
// i = 1000
@1000
D=A
@i
M=D
(LOOP)
// if(i = 0) goto CONT
@i
D=M
@CONT
D;JEQ
// i = i - 1
@i
M=M-1
// goto LOOP
@LOOP
0;JMP
(CONT)
...
```

write

Label declaration in the Hack assembly language:

(sym)

Results in binding *sym* to the address of the next instruction

In this example:

LOOP is bound to 4

CONT is bound to 12

(done by the assembler;
The code writer doesn't care
about these numbers)

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels



Low Level Programming



Basic

- Iteration
- Pointers

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Program example 1: Add

Add.asm

```
// Sets R2 to R0 + R1 + 17
// D = R0
@R0
D=M
// D = D + R1
@R1
D=D+M
// D = D + 17
@17
D=D+A
// R2 = D
@R2
M=D
```

Program example 2: signum

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
  
END:
```

write



Signum.asm

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
  
// if R0 >= 0 goto POS  
@R0  
D=M  
@POS  
D;JGE  
// R1 = -1  
@R1  
M=-1  
// goto END  
@END  
0;JMP  
  
(POS)  
// R1 = 1  
@R1  
M=1  
  
(END)
```

Best practice

When writing a (non-trivial) assembly program, start by writing pseudocode;

Then translate the pseudo instructions into assembly.

Program translation

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
  
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
// if R0 >= 0 goto POS  
@R0  
D=M  
@POS  
D;JGE  
// R1 = -1  
@R1  
M=-1  
// goto END  
@END  
0;JMP  
  
(POS)  
// R1 = 1  
@R1  
M=1  
  
(END)
```

Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	
11	
12	
13	
14	
...	

The assembler replaces all the symbols with physical addresses

Assembler / loader

(the assembler generates binary instructions;
Here we show their symbolic versions,
for readability)

Watch out for loose ends

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
  
// if R0 >= 0 goto POS  
@R0  
D=M  
@POS  
D;JGE  
// R1 = -1  
@R1  
M=-1  
// goto END  
@END  
0;JMP  
  
(POS)  
// R1 = 1  
@R1  
M=1  
  
(END)
```

Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	0111111000111110
11	1010101001011110
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

The memory is
never empty

Watch out for loose ends

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
  
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
// if R0 >= 0 goto POS  
@R0  
D=M  
@POS  
D;JGE  
// R1 = -1  
@R1  
M=-1  
// goto END  
@END  
0;JMP  
  
(POS)  
// R1 = 1  
@R1  
M=1  
  
(END)
```

Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	0111111000111110
11	1010101001011110
12	Malicious
13	Code
14	0101011100110111
...	

Program execution:



Watch out for loose ends

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1=-1  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
  
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1  
// else R1=-1  
// if R0 >= 0 goto POS  
@R0  
D=M  
@POS  
D;JGE  
// R1 = -1  
@R1  
M=-1  
// goto END  
@END  
0;JMP  
  
(POS)  
// R1 = 1  
@R1  
M=1  
  
(END)
```

Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	0111111000111110
11	1010101001011110
12	Malicious
13	Code
14	0101011100110111
...	

Program execution:

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
  
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
// if R0 >= 0 goto POS  
@R0  
D=M  
@POS  
D;JGE  
// R1 = -1  
@R1  
M=-1  
// goto END  
@END  
0;JMP  
  
(POS)  
// R1 = 1  
@R1  
M=1  
  
(END) ←
```

Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
  
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
// if R0 >= 0 goto POS  
@R0  
D=M  
@POS  
D;JGE  
// R1 = -1  
@R1  
M=-1  
// goto END  
@END  
0;JMP  
  
(POS)  
// R1 = 1  
@R1  
M=1  
  
(END)  
@END  
0;JMP
```



Terminating programs properly

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
END:
```

Signum.asm

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
  
// if R0 >= 0 goto POS  
@R0  
D=M  
@POS  
D;JGE  
// R1 = -1  
@R1  
M=-1  
// goto END  
@END  
0;JMP  
  
(POS)  
// R1 = 1  
@R1  
M=1  
  
(END)  
@END  
0;JMP
```

Memory

0	@0
1	D=M
2	@8
3	D;JGE
4	@1
5	M=-1
6	@10
7	0;JMP
8	@1
9	M=1
10	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
14	0101011100110111
...	

Best practice

Terminate every assembly program with an infinite loop.

By the way...

Pseudocode

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
if (R0 ≥ 0) goto POS  
R1 = -1  
goto END  
  
POS:  
    R1 = 1  
  
END:
```

Better:

```
// if R0 >= 0 then R1 = 1  
// else R1 = -1  
R1 = -1  
if (R0 < 0) goto END  
R1 = 1  
  
END:
```

Best practice

Optimize your pseudocode before writing it in machine language.

Program example 3: Max

Pseudocode

```
// R2 = max(R0, R1)  
// if (R0 > R1) then R2 = R0  
// else           R2 = R1  
...
```

write



Max2.asm

```
/// You do it
```

- Start by writing the pseudocode
- Write the assembly code in a text file named `Max2.asm`
- Load `Max2.asm` into the CPU emulator
- Put some values in `R0` and `R1`
- Run the program, one instruction at a time
- Inspect the result, `R2`.

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming



Basic



Iteration

- Pointers

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Iterative processing

Example: Compute $1 + 2 + 3 + \dots + N$

Pseudocode

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >=1 in R0
i = 1
sum = 0
LOOP:
if (i > R0) goto STOP
sum = sum + i
i = i + 1
goto LOOP
STOP:
R1 = sum
```

Hack assembly

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >=1 in R0
// i = 1
@i
M=1
// sum = 0
@sum
M=0
(LOOP)
// if(i > R0) goto STOP
@i
D=M
@R0
D=D-M
@STOP
D;JGT
// sum= sum + i
@sum
D=M
@i
D=D+M
@sum
M=D
// i = i + 1
@i
M=M+1
// goto LOOP
@LOOP
0;JMP
```

(code continues here)

```
(STOP)
// R1 = sum
@sum
D=M
@R1
M=D
// infinite loop
(END)
@END
0;JMP
```

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming



Basic



Iteration



Pointers

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Pointer-based processing

Example 1: Set the register at address *addr* to -1

Input: R0 holds *addr*

```
// Sets RAM[R0] to -1
// Usage: Put some non-negative value in R0

@R0
A=M
M=-1
```

In Hack, pointer-based access is realized by setting the address register to the address that we want to access, using the instruction:

A = ...

RAM	
0	1015
1	
2	
...	
15	
16	
17	
...	
255	
256	
...	
1012	
1013	
1014	
1015	-1
1016	
...	

desired result

example:
addr = 1015

Pointer-based processing

Example 2: Get the value of the register at address *addr*

Input: R0 holds *addr*

```
// Gets R1 = RAM[R0]  
// Usage: Put some non-negative value in R0
```

?

RAM	
0	1013
1	75
2	
...	
15	
16	
17	
...	
255	
256	
...	
1012	512
1013	75
1014	19
1015	-17
1016	256
...	

R0
R1 desired
R2 result
...
R15

example:
addr = 1013

Pointer-based processing

Example 2: Get the value of the register at address *addr*

Input: R0 holds *addr*

```
// Gets R1 = RAM[R0]
// Usage: Put some non-negative value in R0

@R0
A=M
D=M
@R1
M=D
```

RAM	
0	1013
1	75
2	
...	
15	
16	
17	
...	
255	
256	
...	
1012	512
1013	75
1014	19
1015	-17
1016	256
...	

example:
addr = 1013

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

RAM	
0	300
1	5
2	
...	
15	
16	
17	
...	
255	
256	
...	
300	-1
301	-1
302	-1
303	-1
304	-1
305	
...	

desired output

example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: R0 ($base$) and R1 (n)

RAM	
0	300
1	5
2	
...	
15	
16	
17	
...	
255	
256	
...	
300	
301	
302	
303	
304	
305	
...	

example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

RAM	
0	300
1	5
2	
...	
15	
16	0
17	
...	
255	
256	
...	
300	
301	
302	
303	
304	
305	
...	

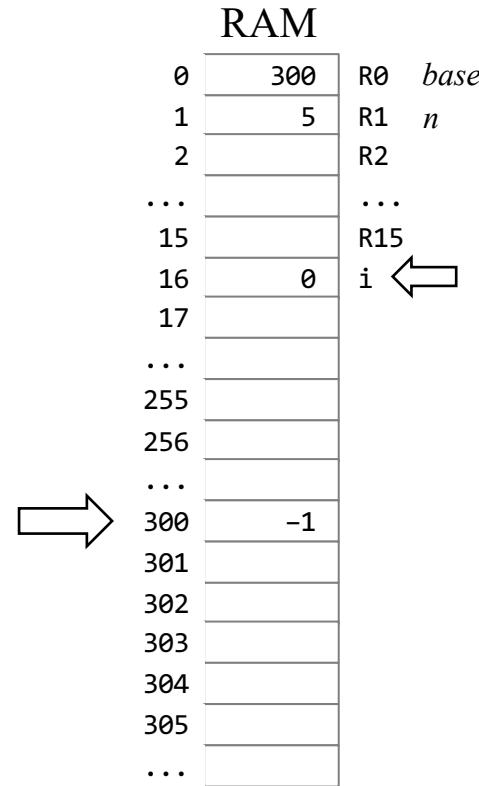
$i \leftarrow$

example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

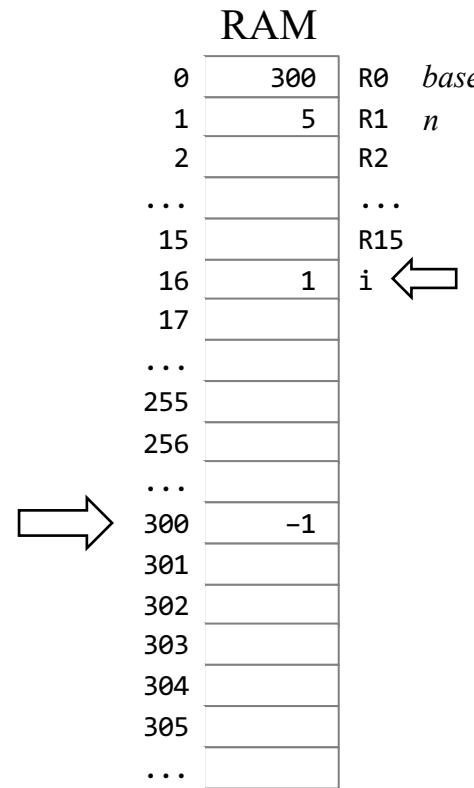


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

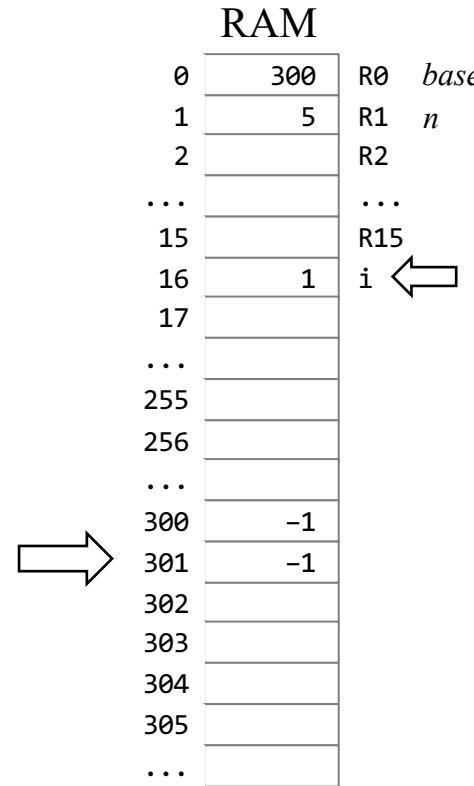


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

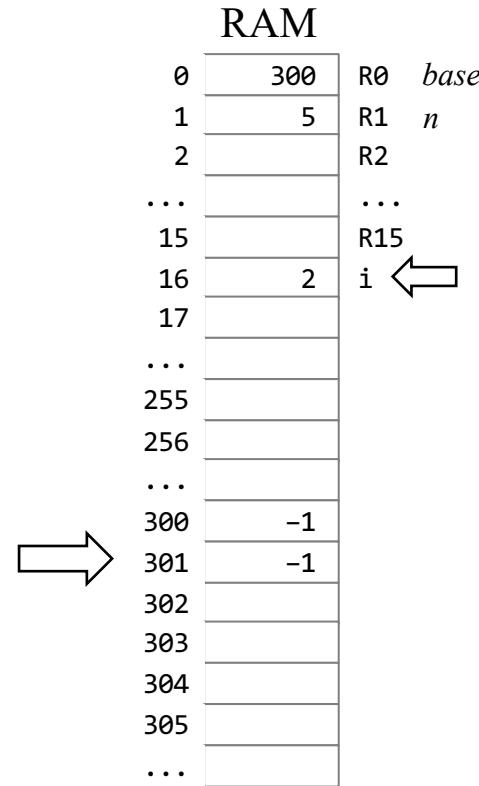


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

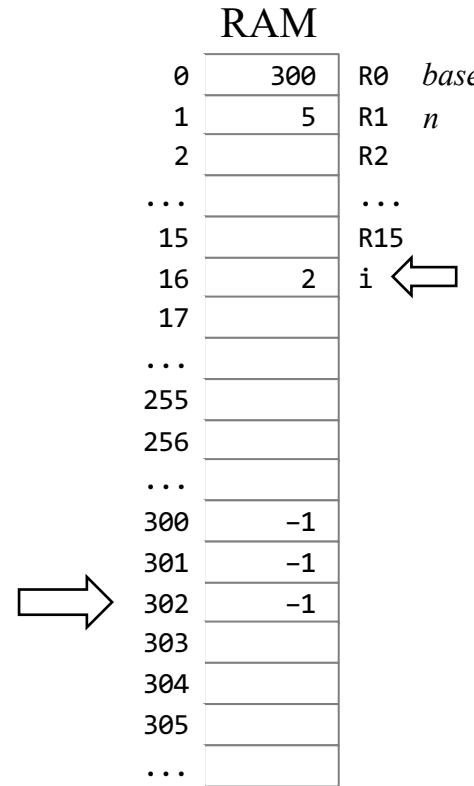


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

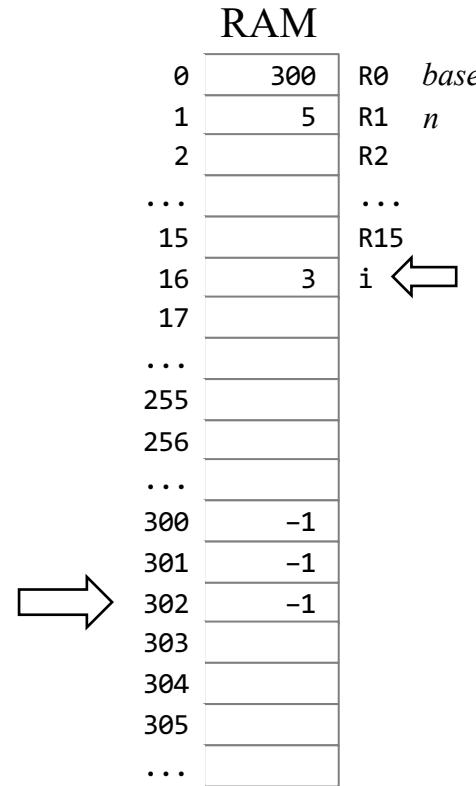


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

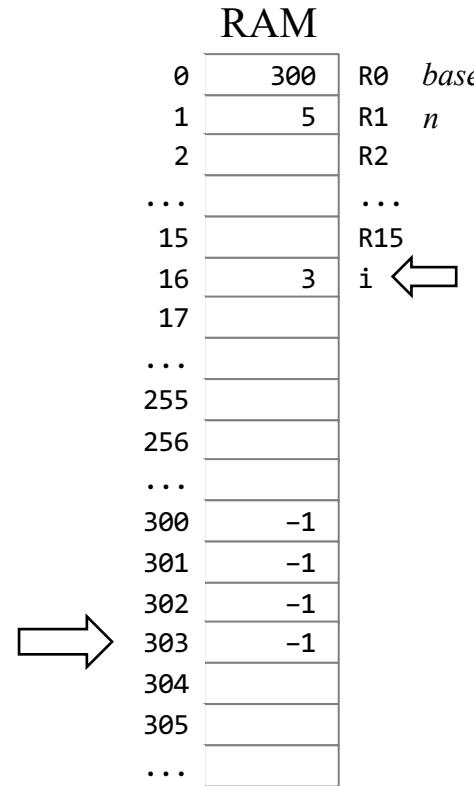


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

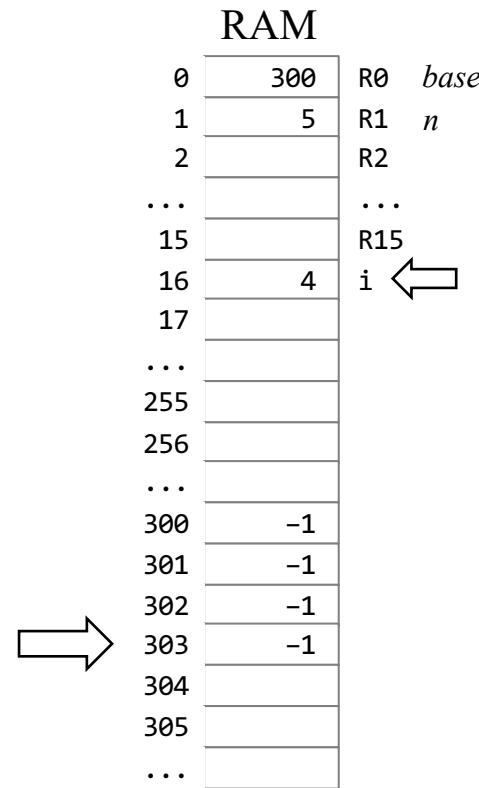


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

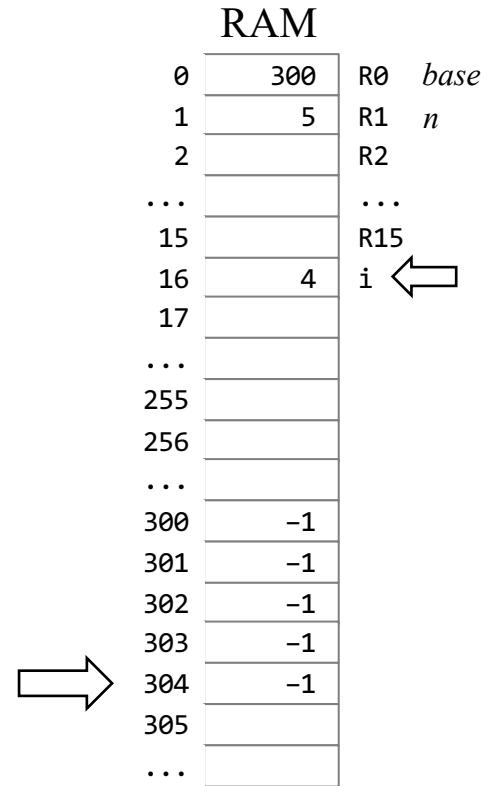


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

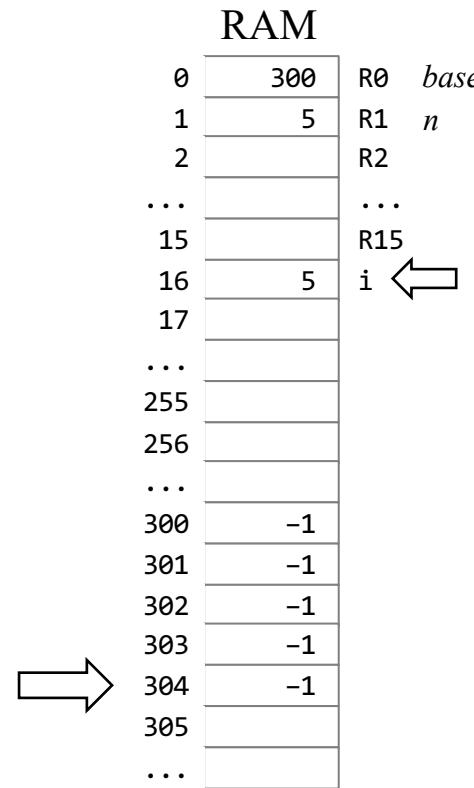


example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)



example:
 $base = 300$
 $n = 5$

Pointer-based processing

Example 3: Set the first n words of the memory block beginning in address $base$ to -1

Inputs: $R0$ ($base$) and $R1$ (n)

Pseudocode

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1

    i = 0
LOOP:
    if (i == R1) goto END
    RAM[R0+i] = -1
    i = i+1
    goto LOOP
END:
```

Assembly code

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1

    // i = 0
    @i
    M=0

(LOOP)
    // if (i == R1) goto END
    @i
    D=M
    @R1
    D=D-M
    @END
    D;JEQ
    // RAM[R0 + i] = -1
    @R0
    D=M
    @i
    A=D+M
    M=-1
    // i = i + 1
    @i
    M=M+1
    // goto LOOP
    @LOOP
    0;JMP

(END)
    @END
    0;JMP
```

RAM	
0	300
1	5
2	
...	
15	
16	5
17	
...	
255	
256	
...	
300	-1
301	-1
302	-1
303	-1
304	-1
305	
...	

example:
 $base = 300$
 $n = 5$

Array processing

High-level code (Java example)

```
...
// Variable declarations
int[] arr = new int[5];
int sum = 0;
...

// Enters some values into the array
//(code omitted)
...

// Sums up the array elements
for (int j=0; j<5; j++) {
    sum = sum + arr[j];
}
...
```

Memory state just before executing the for loop:

RAM	
0	R0
1	R1
2	R2
...	...
15	R15
16	5034
17	arr
...	sum
75	
76	
...	
255	
256	
...	
5034	100
5035	50
5036	200
5037	2
5038	7
5036	
...	

Array processing

High-level code (Java example)

```
...
// Variable declarations
int[] arr = new int[5];
int sum = 0;
...
// Enters some values into the array
//(code omitted)
...
// Sums up the array elements
for (int j=0; j<5; j++) {
    sum = sum + arr[j];
}
...
```

Memory state just
after executing the
for loop:

RAM	
0	R0
1	R1
2	R2
...	...
15	R15
16	5034
17	359
...	sum
5	j
75	
76	
...	
255	
256	
...	
5034	100
5035	50
5036	200
5037	2
5038	7
5036	
...	

Array processing

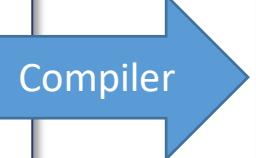
High-level code (Java example)

```
...
// Variable declarations
int[] arr = new int[5];
int sum = 0;
...

// Enters some values into the array
//(code omitted)
...

// Sums up the array elements
for (int j=0; j<5; j++) {
    sum = sum + arr[j];
}

// Increments each array element
for (int j=0; j<5; j++) {
    arr[j] = arr[j]+1
}
```



Hack assembly

```
...
// sum = sum + arr[j]
@arr
D=M
@j
A=D+M
D=M
@sum
M=M+D
...
// arr[j] = arr[j] + 1
?
```

RAM

0	R0
1	R1
2	R2
...	...
15	R15
16	5034
17	359
...	...
5	sum
75	j
76	
...	
255	
256	
...	
5034	100
5035	50
5036	200
5037	2
5038	7
5036	
...	

Array processing

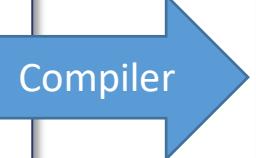
High-level code (Java example)

```
...
// Variable declarations
int[] arr = new int[5];
int sum = 0;
...

// Enters some values into the array
//(code omitted)
...

// Sums up the array elements
for (int j=0; j<5; j++) {
    sum = sum + arr[j];
}

// Increments each array element
for (int j=0; j<5; j++) {
    arr[j] = arr[j]+1
}
```



Hack assembly

```
...
// sum = sum + arr[j]
@arr
D=M
@j
A=D+M
D=M
@sum
M=M+D
...
// arr[j] = arr[j] + 1
@arr
D=M
@j
A=D+M
M=M+1
...
```

RAM

0	R0
1	R1
2	R2
...	...
15	R15
16	5034
17	359
...	...
5	sum
75	j
76	
...	
255	
256	
...	
5034	100
5035	50
5036	200
5037	2
5038	7
5036	
...	

Every high-level array access $arr[expression]$ can be compiled into Hack code that realizes the access using the low-level idiom
 $A = arr + expression$

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming



- Basic
- Iteration
- Pointers

The Hack Language



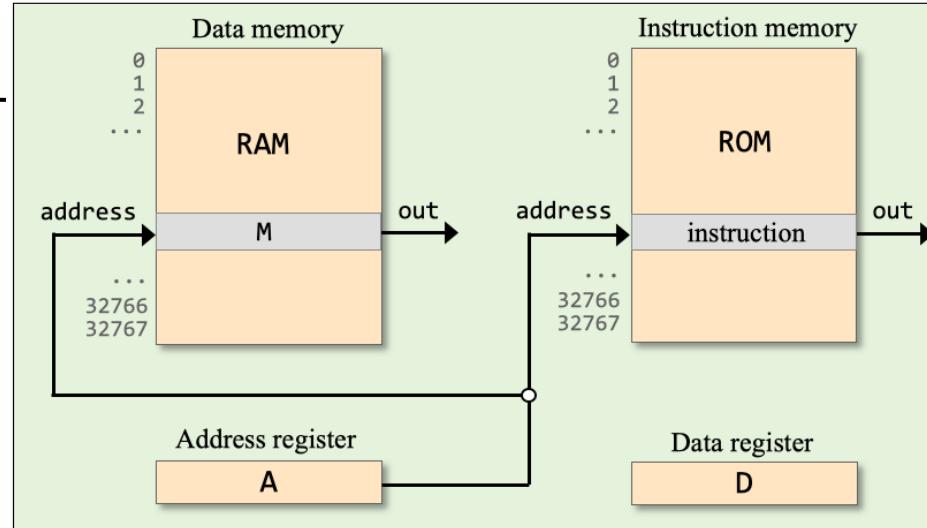
- Symbolic
- Binary
- Output
- Input
- Project 4

The A-instruction

Instruction set

→ A - instruction

- C - instruction



Syntax:

`@xxx`

where *xxx* is either a constant, or
a symbol bound to a constant

Semantics:

- Sets the A register to the *xxx*
- Side effects:

$\text{RAM}[A]$ becomes the selected RAM location

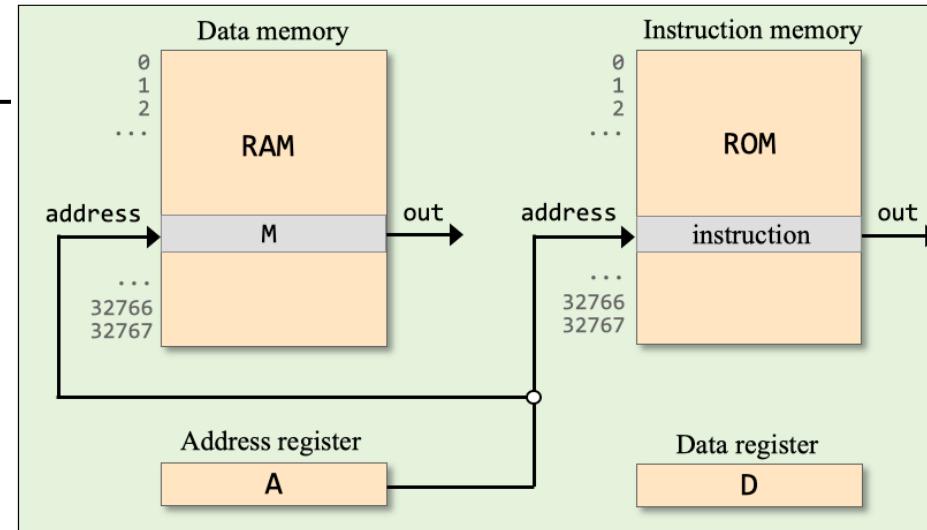
$\text{ROM}[A]$ becomes the selected ROM location

The C-instruction

Instruction set

- A - instruction

→ C - instruction



The C-instruction

Syntax: $dest = comp ; jump$ “ $dest =$ ” and “ $; jump$ ” are optional

where:

$comp =$ $0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A$
 $M, !M, -M, M+1, M-1, D+M, D-M, M-D, D\&M, D|M$

$dest =$ null, M, D, DM, A, AM, AD, ADM M stands for RAM[A]

$jump =$ null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP

Semantics:

Computes the value of $comp$ and stores the result in $dest$.

If ($comp \neq 0$), branches to execute ROM[A]

Examples:

```
// Sets the D register to -1
```

The C-instruction

Syntax: $dest = comp ; jump$ “ $dest =$ ” and “ $; jump$ ” are optional

where:

$comp =$ $0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A$
 $M, !M, -M, M+1, M-1, D+M, D-M, M-D, D\&M, D|M$

$dest =$ null, M, D, DM, A, AM, AD, ADM M stands for RAM[A]

$jump =$ null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP

Semantics:

Computes the value of $comp$ and stores the result in $dest$.

If ($comp \neq 0$), branches to execute ROM[A]

Examples:

```
// Sets the D register to -1  
D=-1
```

The C-instruction

Syntax: $dest = comp ; jump$ “ $dest =$ ” and “ $; jump$ ” are optional

where:

$comp = 0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A$
 $M, !M, -M, M+1, M-1, D+M, D-M, M-D, D\&M, D|M$

$dest = \text{null}, M, D, DM, A, AM, AD, ADM$ M stands for RAM[A]

$jump = \text{null}, JGT, JEQ, JGE, JLT, JNE, JLE, JMP$

Semantics:

Computes the value of $comp$ and stores the result in $dest$.

If $(comp \neq 0)$, branches to execute ROM[A]

Examples:

```
// Sets D and M to the value of the D register, plus 1  
DM=D+1
```

The C-instruction

Syntax: $dest = comp ; jump$ “ $dest =$ ” and “ $; jump$ ” are optional

where:

$comp = 0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A$
 $M, !M, -M, M+1, M-1, D+M, D-M, M-D, D\&M, D|M$

$dest = \text{null}, M, D, DM, A, AM, AD, ADM$ M stands for RAM[A]

$jump = \text{null}, JGT, JEQ, JGE, JLT, JNE, JLE, JMP$

Semantics:

Computes the value of $comp$ and stores the result in $dest$.

If ($comp \neq 0$), branches to execute ROM[A]

Examples:

```
// If (D-1 = 0) jumps to execute the instruction stored in ROM[56]
@56
D-1;JEQ
```

The C-instruction

Syntax: $dest = comp ; jump$ “ $dest =$ ” and “ $; jump$ ” are optional

where:

$comp =$ $0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A$
 $M, !M, -M, M+1, M-1, D+M, D-M, M-D, D\&M, D|M$

$dest =$ **null, M, D, DM, A, AM, AD, ADM** M stands for RAM[A]

$jump =$ **null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP**

Semantics:

Computes the value of $comp$ and stores the result in $dest$.

If $(comp \ jump \ 0)$, branches to execute ROM[A]

Examples:

```
// goto LOOP  
@LOOP  
0;JMP // The 0; prefix is a syntax convention
```

Recap: A-instructions and C-instructions

They normally come in pairs:

```
// RAM[5] = RAM[5] - 1  
@5  
M=M-1
```

To set up for a C-instruction that operates on M,
Use an A-instruction to select the target address

```
// if D=0 goto 100  
@100  
D;JEQ
```

To set up for a C-instruction that specifies a jump,
Use an A-instruction to select the target address

It makes no sense that a C-instruction will use the same address to access the data memory and the instruction memory simultaneously;

Best practice rule

A C-instruction should specify ...

- Either M
- Or a jump directive

But not both.

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- ✓ Symbolic
- Binary
- Output
- Input
- Project 4

The Hack language specification

A instruction

Symbolic:	$@xxx$	(xxx is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)
Binary:	$0\ vvvvvvvvvvvvvvvv$	($vv \dots v$ = 15-bit value of xxx)

C instruction

Symbolic:	$dest = comp ; jump$	($comp$ is mandatory. If $dest$ is empty, the $=$ is omitted; If $jump$ is empty, the $;$ is omitted)
Binary:	$111accccccdddjjj$	

Predefined symbols:

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
SCREEN	16384
KBD	24576

$comp$	c	c	c	c	c	c
0		1	0	1	0	1
1		1	1	1	1	1
-1		1	1	1	0	1
D		0	0	1	1	0
A	M	1	1	0	0	0
!	D	0	0	1	1	0
!	A	!M	1	1	0	0
-	D	0	0	1	1	1
-	A	-M	1	1	0	1
-	-	D+1	0	1	1	1
-	-	A+1	M+1	1	1	0
-	-	D-1	0	0	1	1
-	-	A-1	M-1	1	1	0
-	-	D+A	D+M	0	0	0
-	-	D-A	D-M	0	1	0
-	-	A-D	M-D	0	0	1
-	-	D&A	D&M	0	0	0
-	-	D A	D M	0	1	0

$a == 0$ $a == 1$

$dest$	d	d	d	Effect: store $comp$ in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register (reg)
DM	0	1	1	RAM[A] and D reg
A	1	0	0	A reg
AM	1	0	1	A reg and RAM[A]
AD	1	1	0	A reg and D reg
ADM	1	1	1	A reg, D reg, and RAM[A]

$jump$	j	j	j	Effect:
null	0	0	0	no jump
JGT	0	0	1	if $comp > 0$ jump
JEQ	0	1	0	if $comp = 0$ jump
JGE	0	1	1	if $comp \geq 0$ jump
JLT	1	0	0	if $comp < 0$ jump
JNE	1	0	1	if $comp \neq 0$ jump
JLE	1	1	0	if $comp \leq 0$ jump
JMP	1	1	1	unconditional jump

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

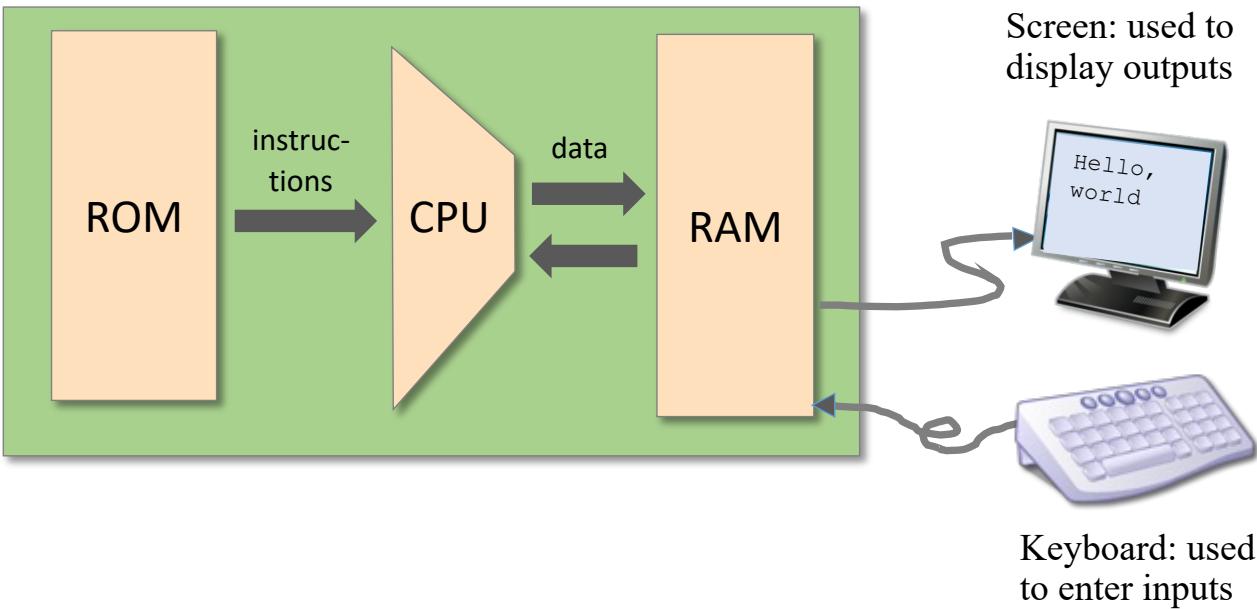
Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- ✓ Symbolic
- ✓ Binary
- Output
- Input
- Project 4

Input / output



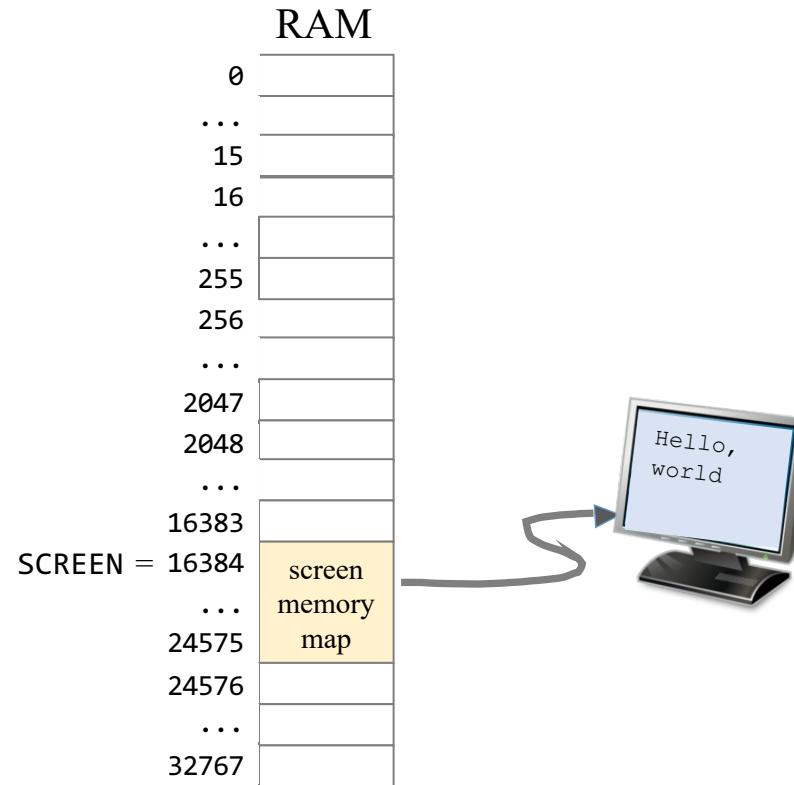
High-level I/O handling (later in the course):

I/O libraries for handling text, graphics, audio, video, ...

Low-level I/O handling:

Manipulating bits directly, using memory resident *bitmaps*.

Bitmaps



Screen memory map:

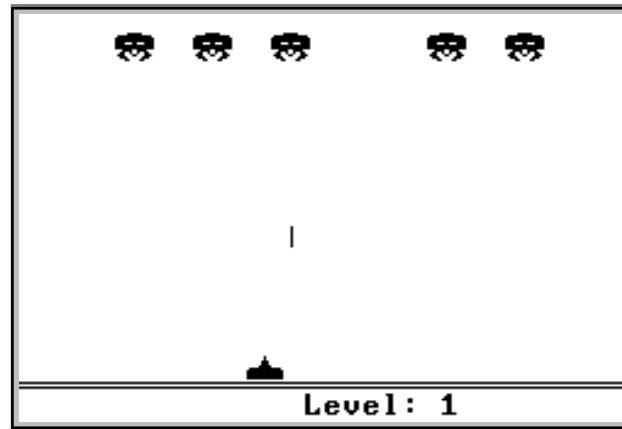
An 8K memory block, dedicated to representing a black-and-white display unit

Base address: `SCREEN = 16384` (predefined symbol)

Output is rendered by writing bits in the screen memory map.

Bitmaps

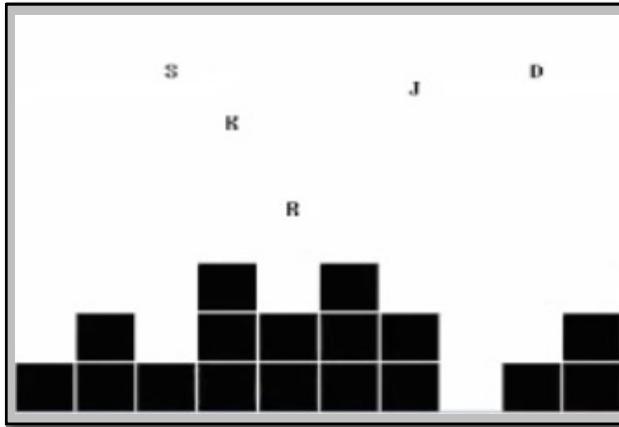
Physical screen



Screen shots of computer games
developed on the Hack computer

Bitmaps

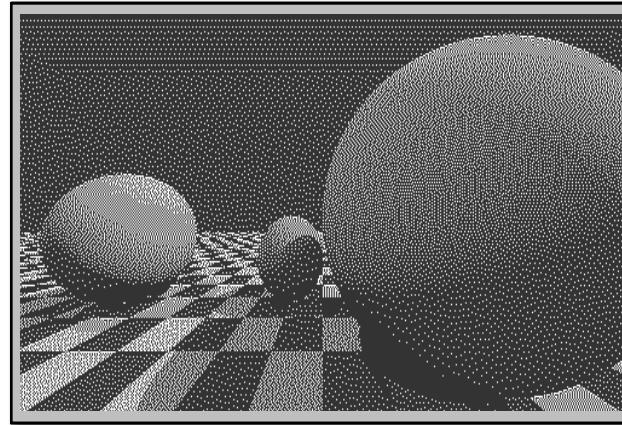
Physical screen



Screen shots of computer games
developed on the Hack computer

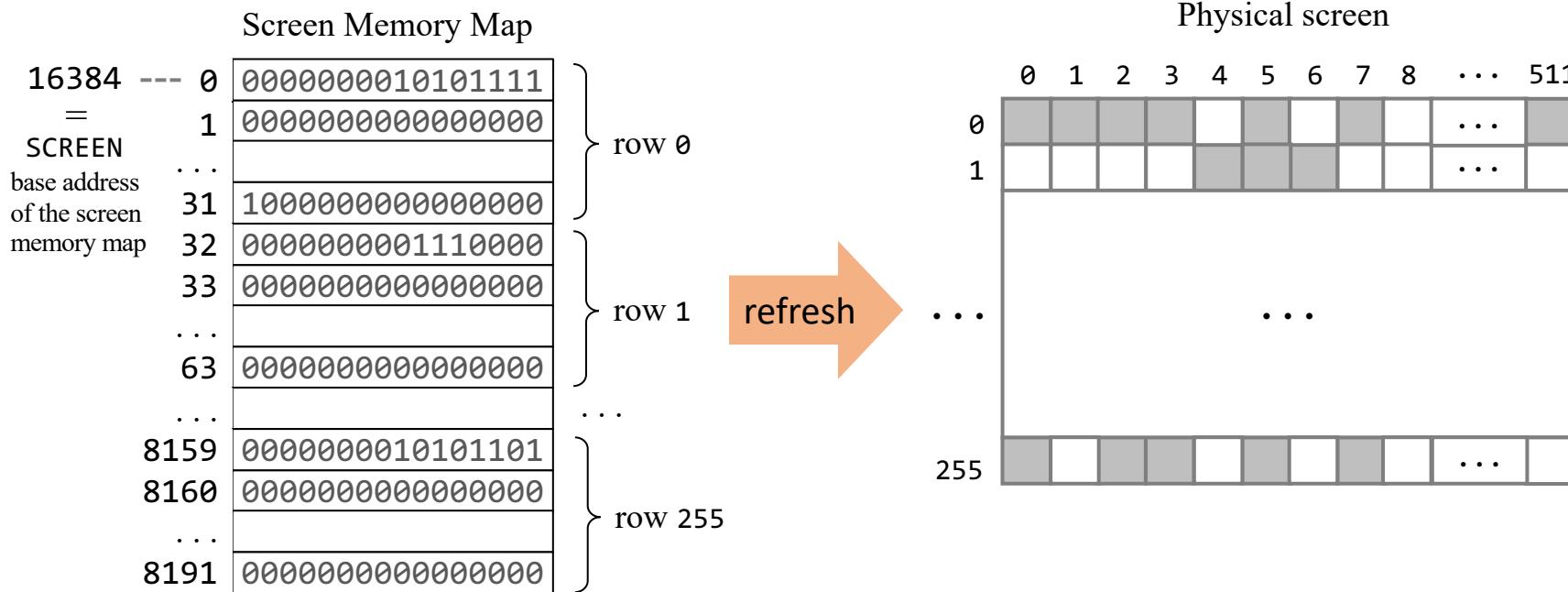
Bitmaps

Physical screen



Screen shots of computer games
developed on the Hack computer

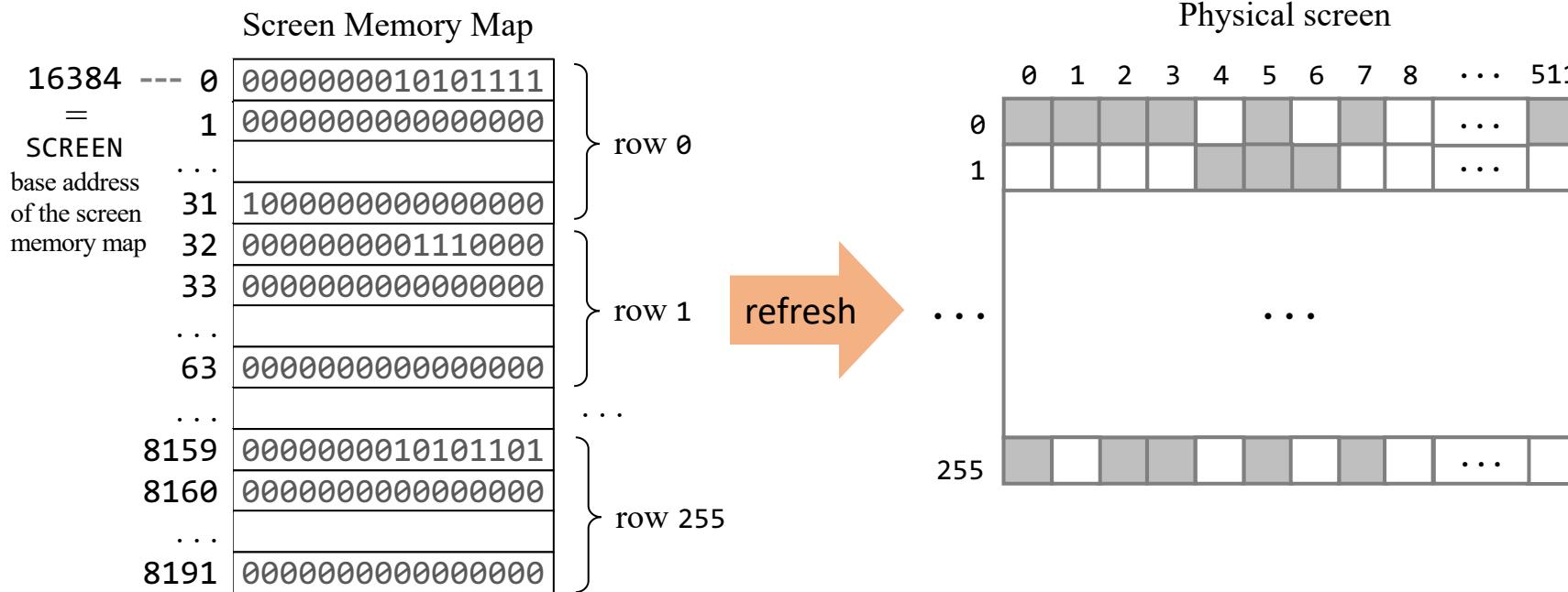
Bitmaps



Mapping

The (row, col) pixel in the physical screen is represented by the $(col \% 16)th$ bit in RAM address $\text{SCREEN} + 32 * \text{row} + \text{col} / 16$

Bitmaps



To set the (row, col) pixel to black or white:

$addr \leftarrow SCREEN + 32 * row + col / 16$

$word \leftarrow RAM[addr]$

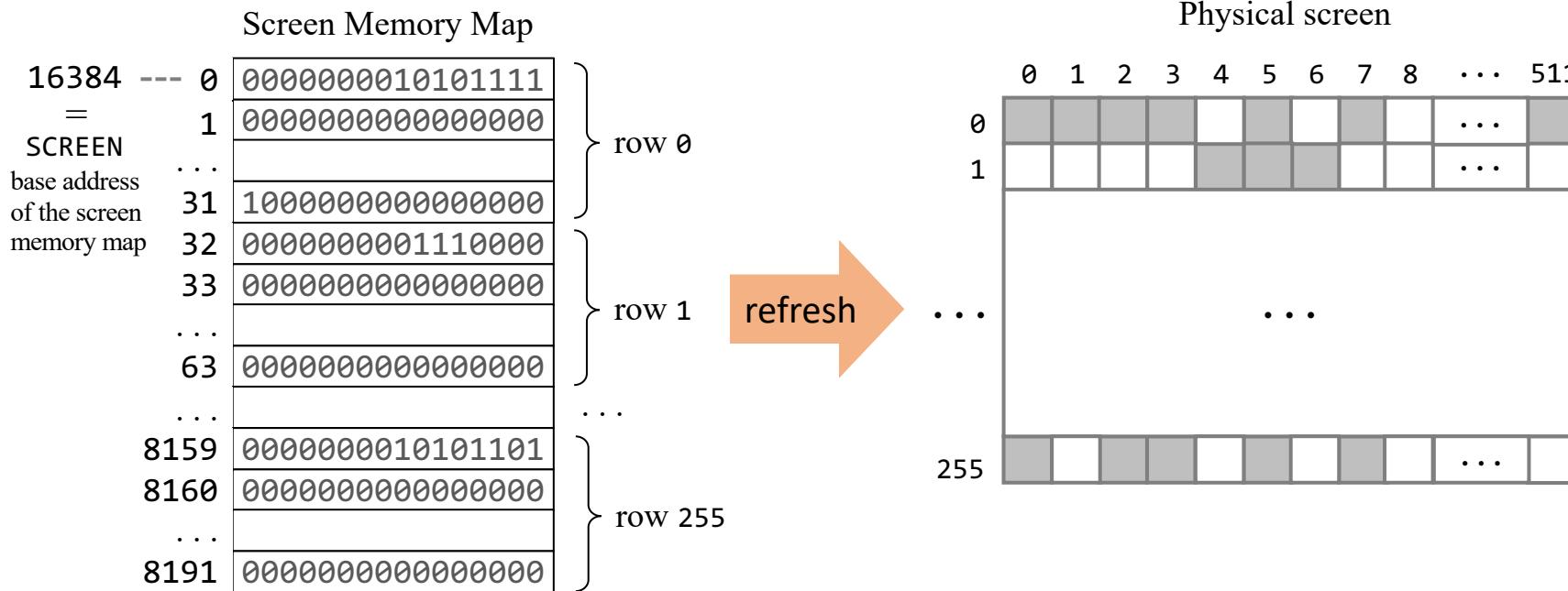
Set the $(col \% 16)$ th bit of $word$ to 0 or 1

$RAM[addr] \leftarrow word$

Not to worry...

Cool Bitmap Editor coming up

Bitmaps



Examples of simple patterns that can be drawn relatively easily:

```
// Sets the first (left) 16 pixels
// of the top row to black

@SCREEN
M=-1      // -1 = 1111111111111111
```

Bitmaps

Screen Memory Map

16384	---	0	0000000010101111
=	SCREEN	1	0000000000000000
base address of the screen memory map	...	31	1000000000000000
32	0000000001110000	33	0000000000000000
...	...	63	0000000000000000
...	...	8159	0000000010101101
8160	0000000000000000	8160	0000000000000000
...	...	8191	0000000000000000

} row 0

} row 1

} row 255

refresh

Physical screen



Examples of simple patterns that can be drawn relatively easily:

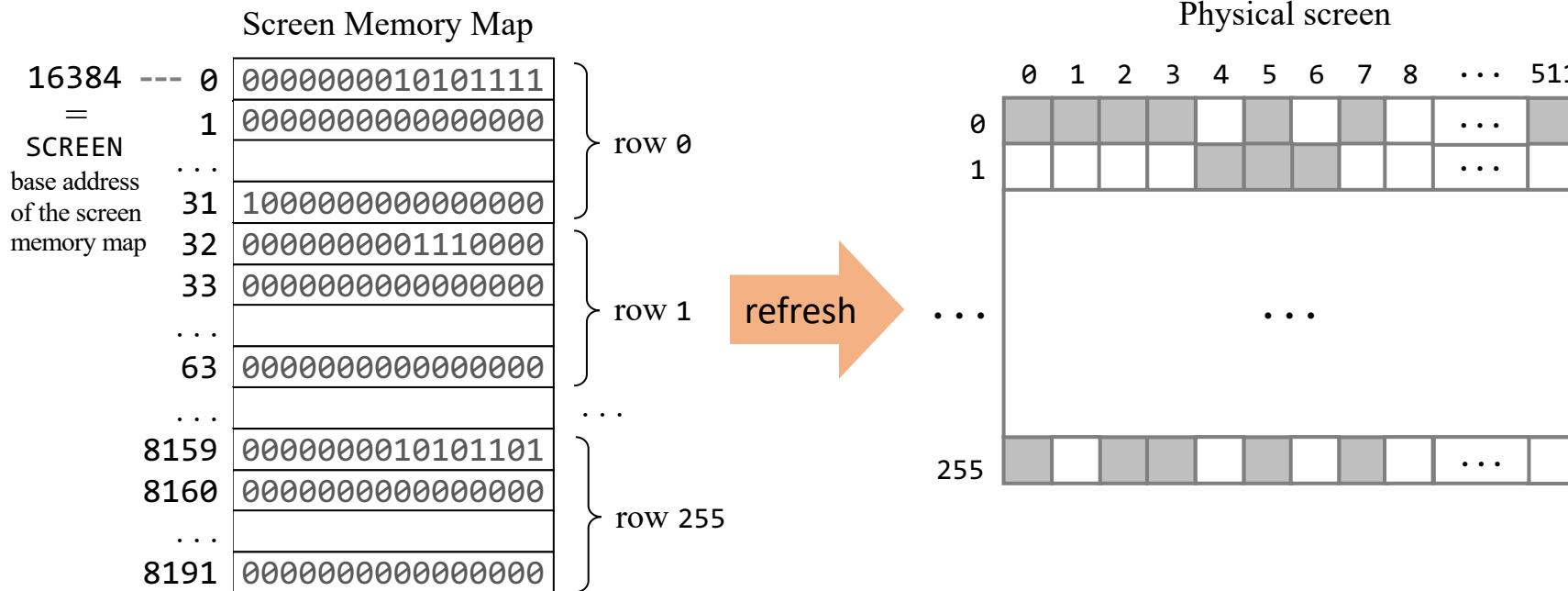
```
// Sets the first (left) 16 pixels
// of the top row to black

@SCREEN
M=-1      // -1 = 1111111111111111
```

```
// Sets the first 16 pixels
// of row 2 to black
```



Bitmaps



Examples of simple patterns that can be drawn relatively easily:

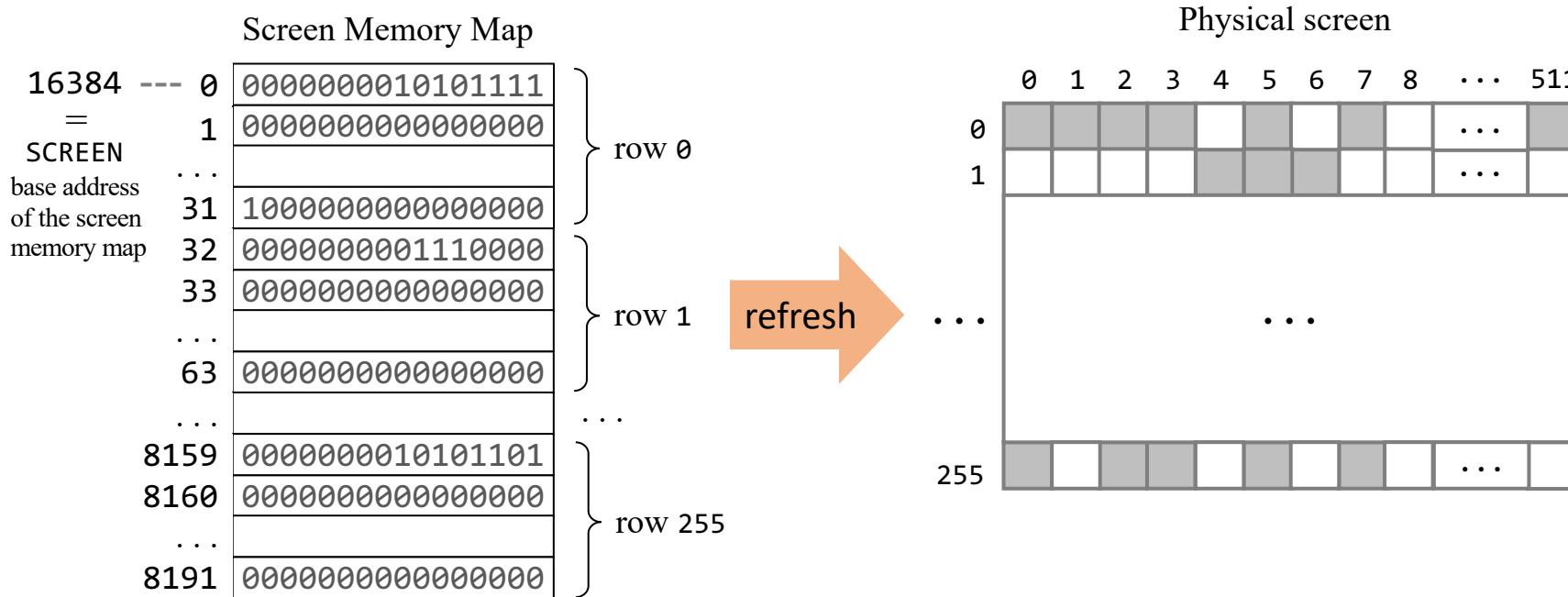
```
// Sets the first (left) 16 pixels
// of the top row to black

@SCREEN
M=-1      // -1 = 1111111111111111
```

```
// Sets the first 16 pixels
// of row 2 to black

@64
D=A
@SCREEN
A=A+D
M=-1
```

Bitmaps



Examples of simple patterns that can be drawn relatively easily:

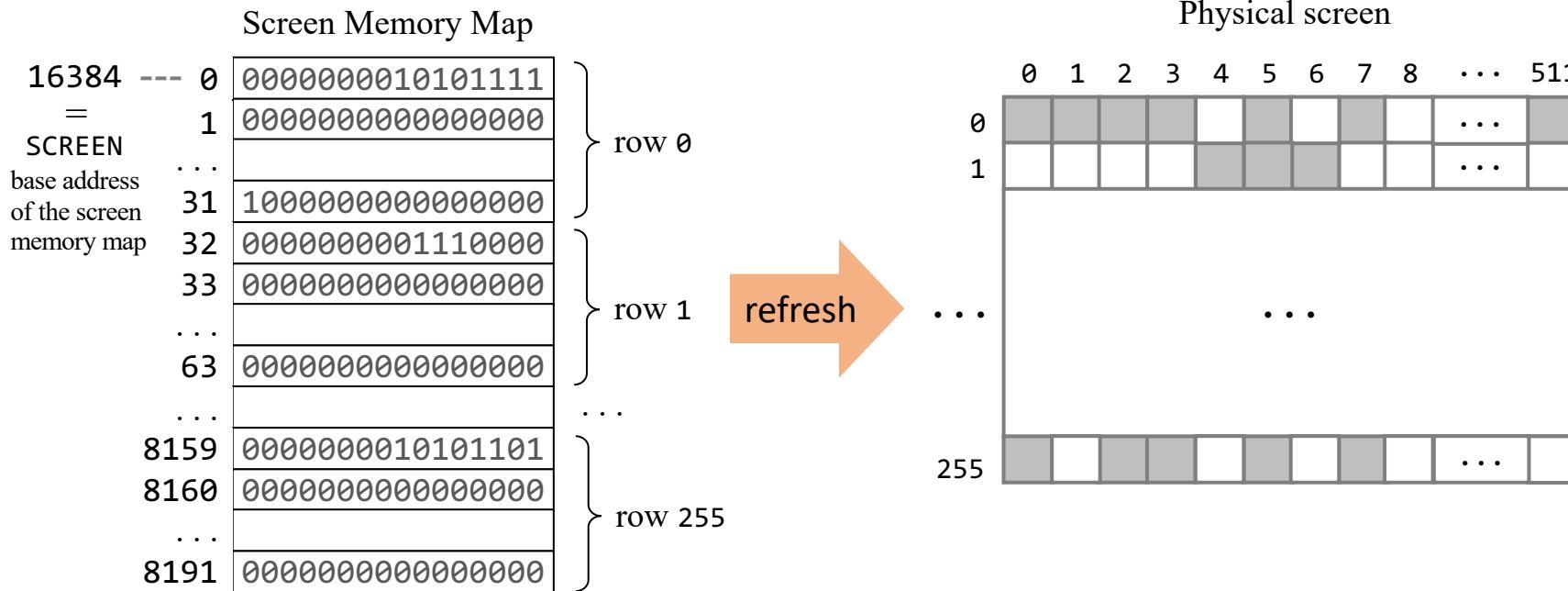
```
// Sets the first (left) 16 pixels  
// of the top row to black  
  
@SCREEN  
  
M=-1          // -1 = 1111111111111111
```

```
// Sets the first 16 pixels  
// of row 2 to black  
  
@64  
D=A  
  
@SCREEN  
A=A+D  
  
M=-1
```

```
// Sets the entire screen  
// to black / white
```

(Project 4)

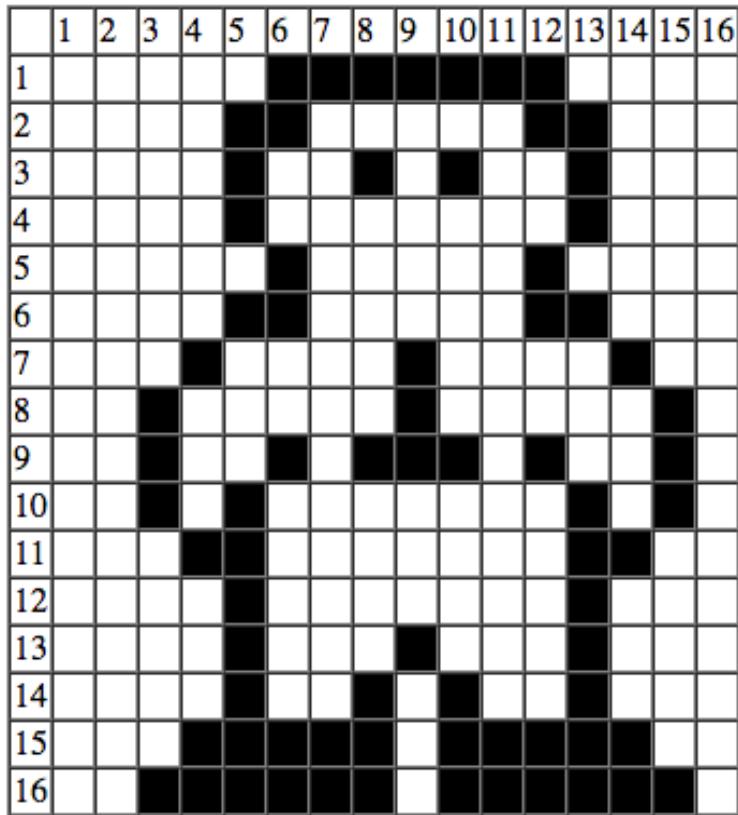
Bitmaps



Simple
graphics
program:



Bitmap Editor



000011111100000 = 4064

0001100000110000 = 6192

0001001010010000 = 4752

...

Bitmap editor

The user draws a pixeled image on a 2D grid

The tool generates code that draws the image in the RAM

The generated code can be copy-pasted into the code.

...

011111011111100 = 32508

Located in this [Git project](#)

Note: The editor generates either Jack code or Hack assembly code – see the radio buttons at the very bottom of the editor's GUI.

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- ✓ Symbolic
- ✓ Binary
- ✓ Output
- Input
- Project 4

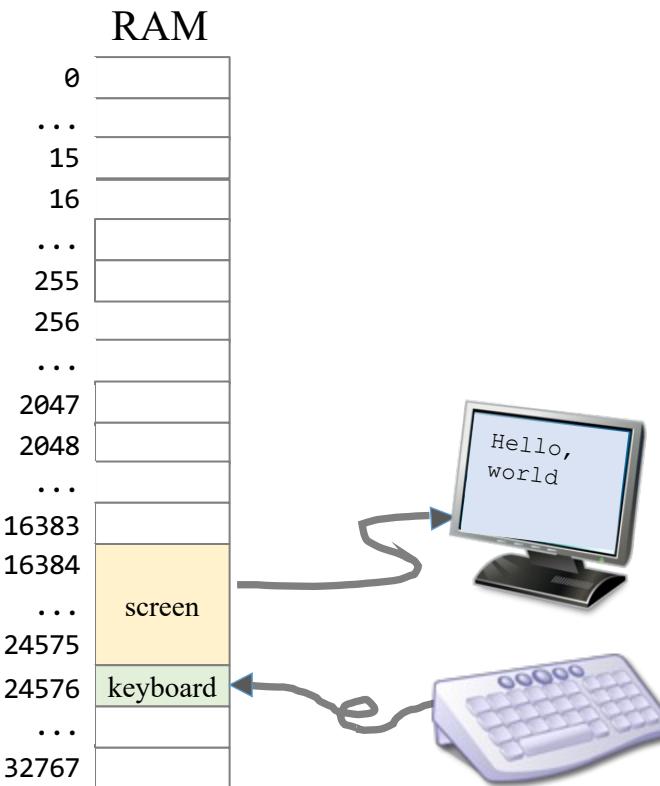
Input

High-level input handling (later in the course)

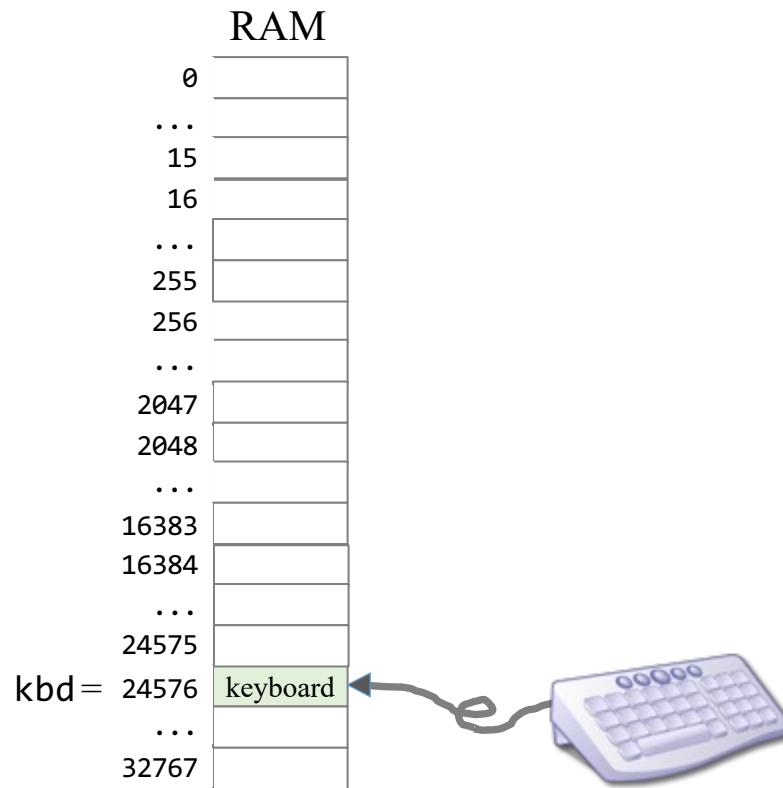
`readInt`, `readString`, ...

Low-level input handling

Read bits.



Input



Keyboard memory map

A single 16-bit memory location, dedicated to representing the keyboard.

Base address: $\text{KBD} = 24576$ (predefined symbol)

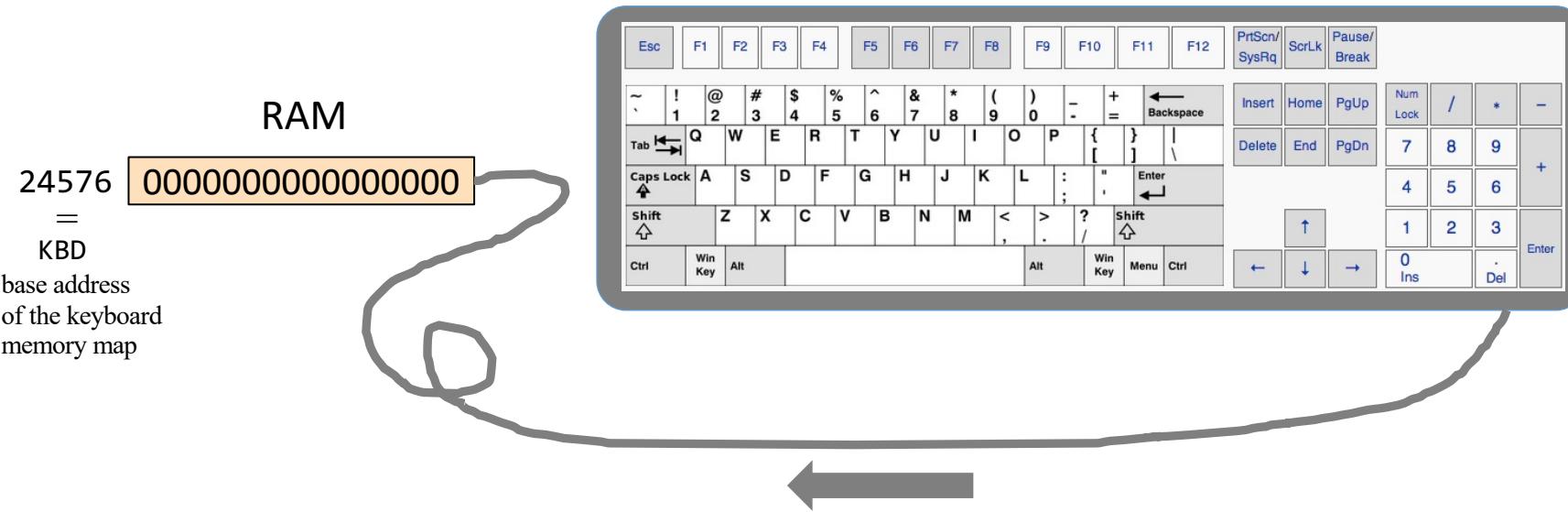
Reading inputs is affected by probing this register.

The Hack character set

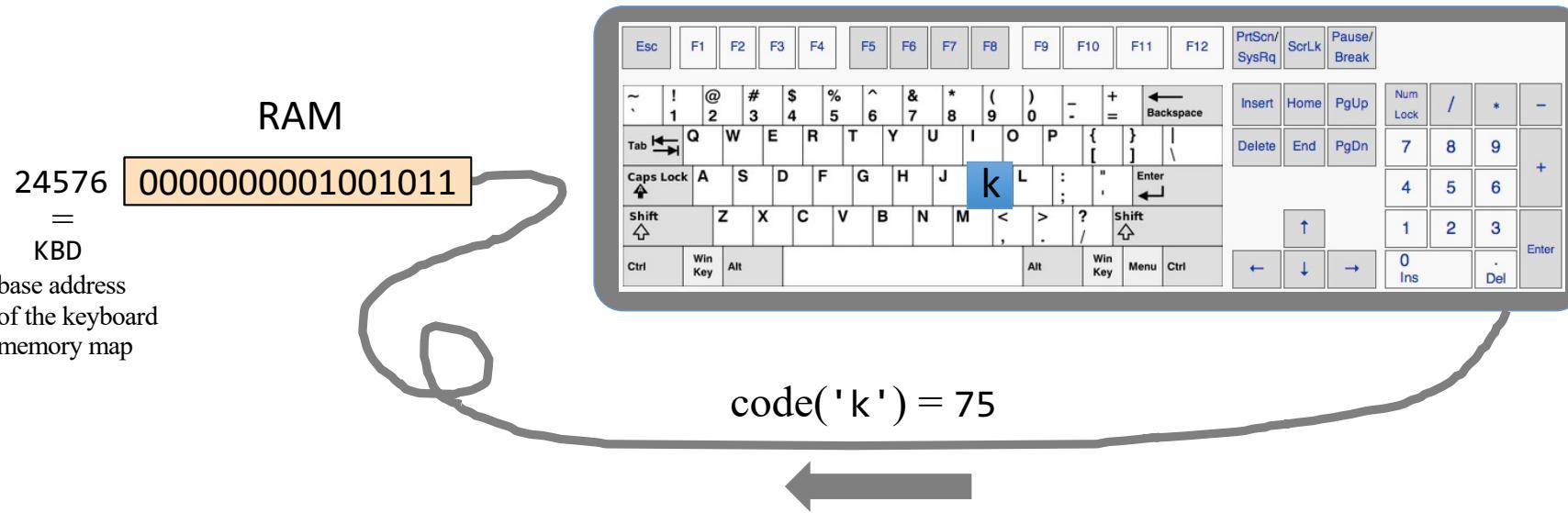
key	code	key	code	key	code	key	code	key	code
(space)	32	0	48	A	65	a	97	newline	128
!	33	1	49	B	66	b	98	backspace	129
"	34	C	...	c	99	left arrow	130
#	35	9	57	up arrow	131
\$	36	:	58	Z	90	z	122	right arrow	132
%	37	;	59	[91	{	123	down arrow	133
&	38	<	60	/	92		124	home	134
'	39	=	61]	93	}	125	end	135
(40	>	62	^	94	~	126	Page up	136
)	41	?	63	-	95			Page down	137
*	42	@	64	`	96			insert	138
+	43							delete	139
,	44							esc	140
-	45							f1	141
.	46						
/	47							f12	152

(Subset of Unicode)

Memory mapped input

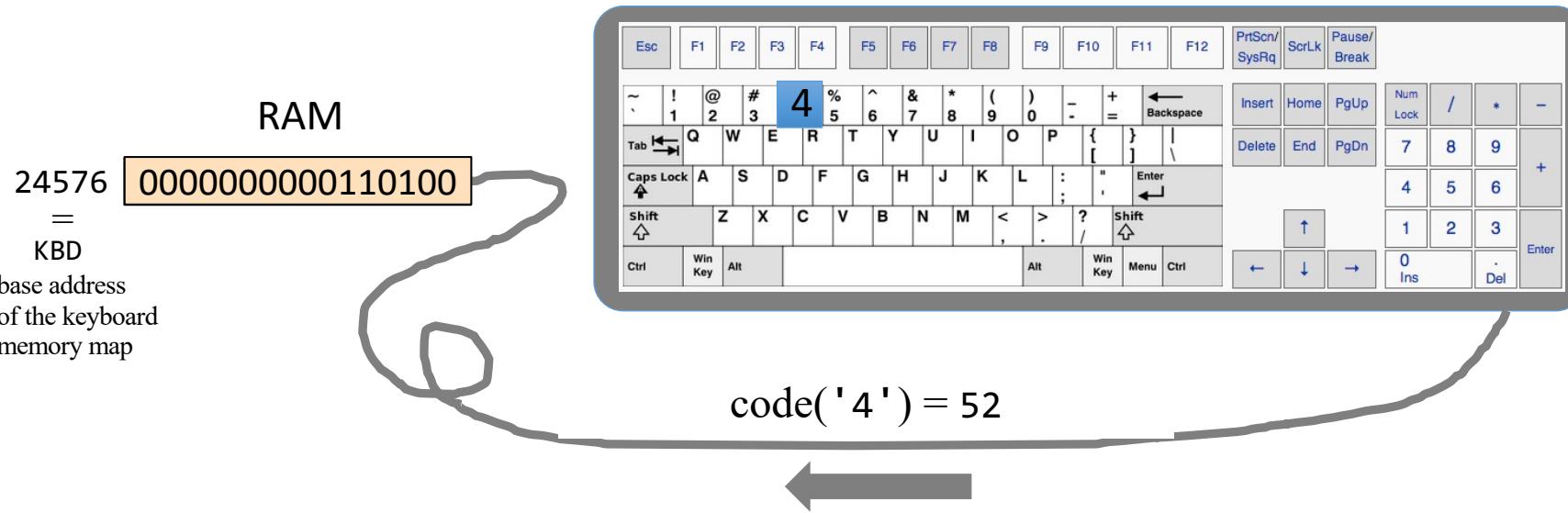


Memory mapped input



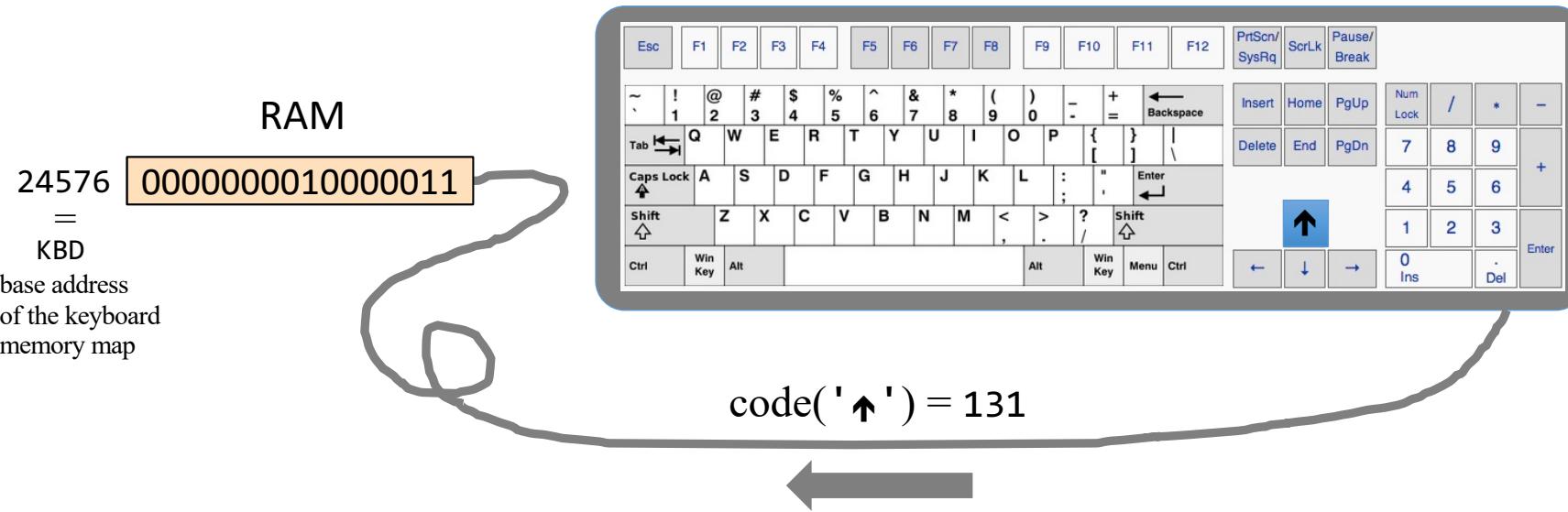
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input



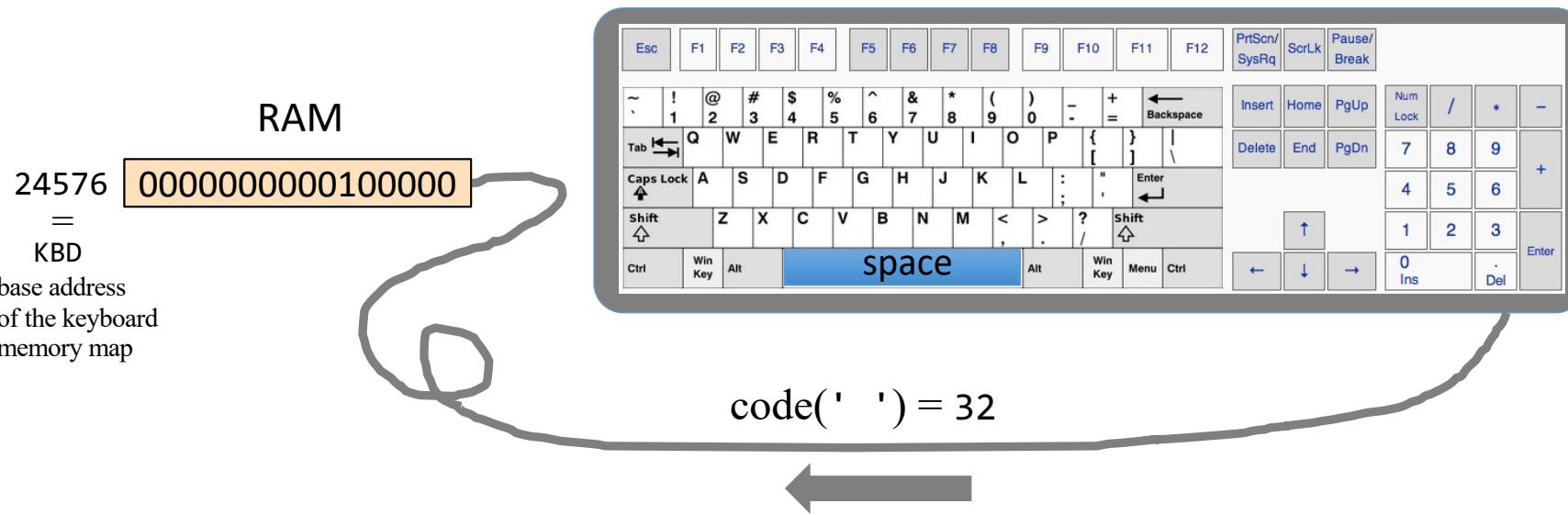
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input



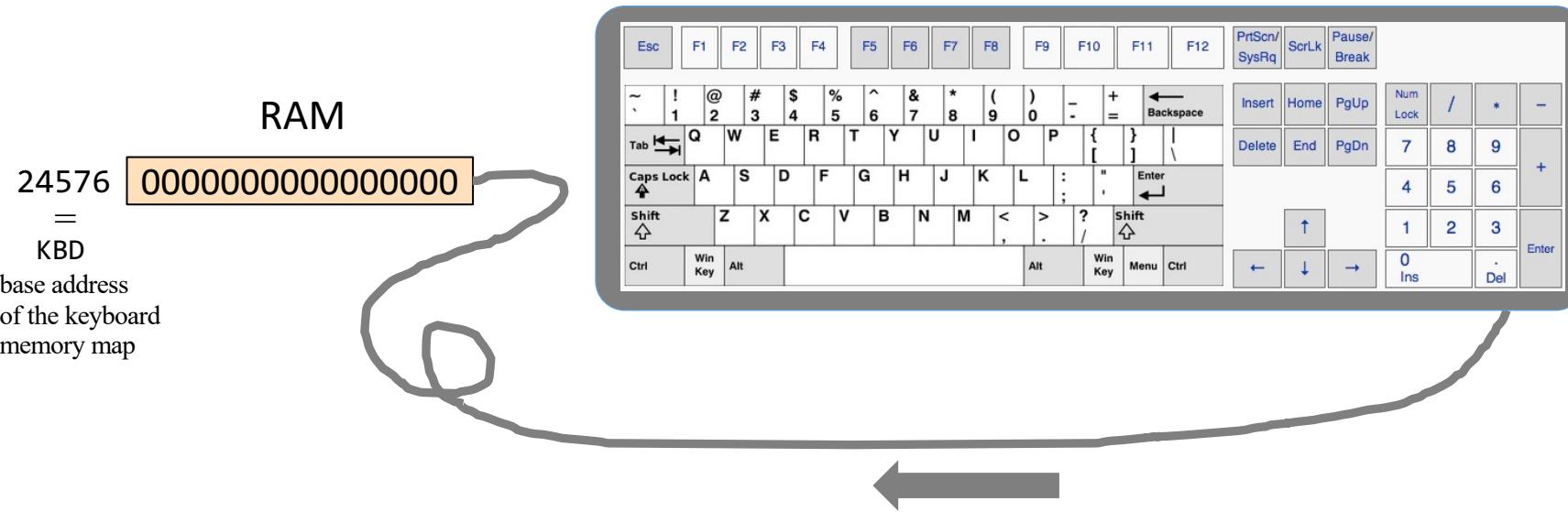
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input



When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input

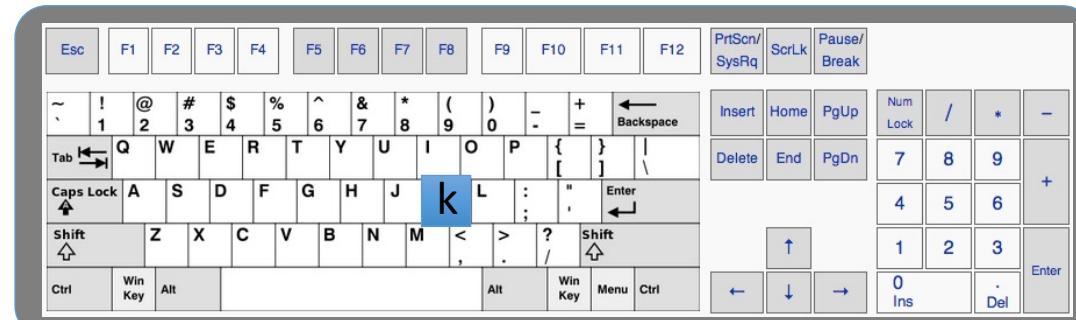


When no key is pressed, the resulting code is 0.

Reading inputs

RAM

24576 [0000000001001011]
= KBD
base address
of the keyboard
memory map



code('k') = 75

Examples:

```
// Set D to the character code of
// the currently pressed key
@KBD
D=M
```

```
// If the currently pressed key is 'q', goto END
@KBD
D=M
@113 // 'q'
D=D-A
@END
D;JEQ
```

Machine Language

Overview

- Machine language
- The Hack computer
- The Hack instruction set
- The Hack CPU Emulator

Symbolic programming

- Control
- Variables
- Labels

Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- ✓ Symbolic
- ✓ Binary
- ✓ Output
- ✓ Input

 Project 4

Project 4

Objectives

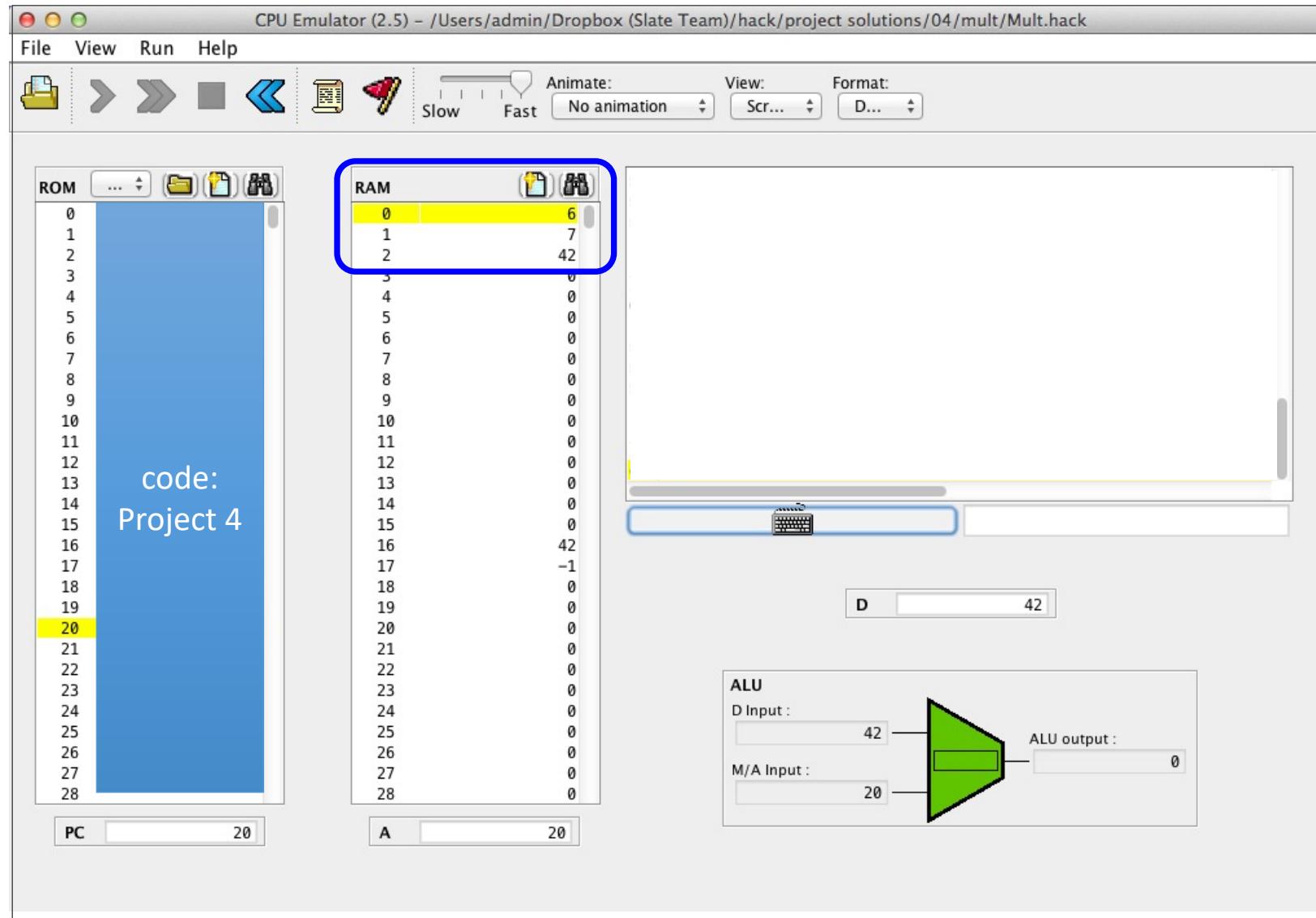
Gain a hands-on taste of:

- Low-level programming
- Assembly language
- The Hack computer

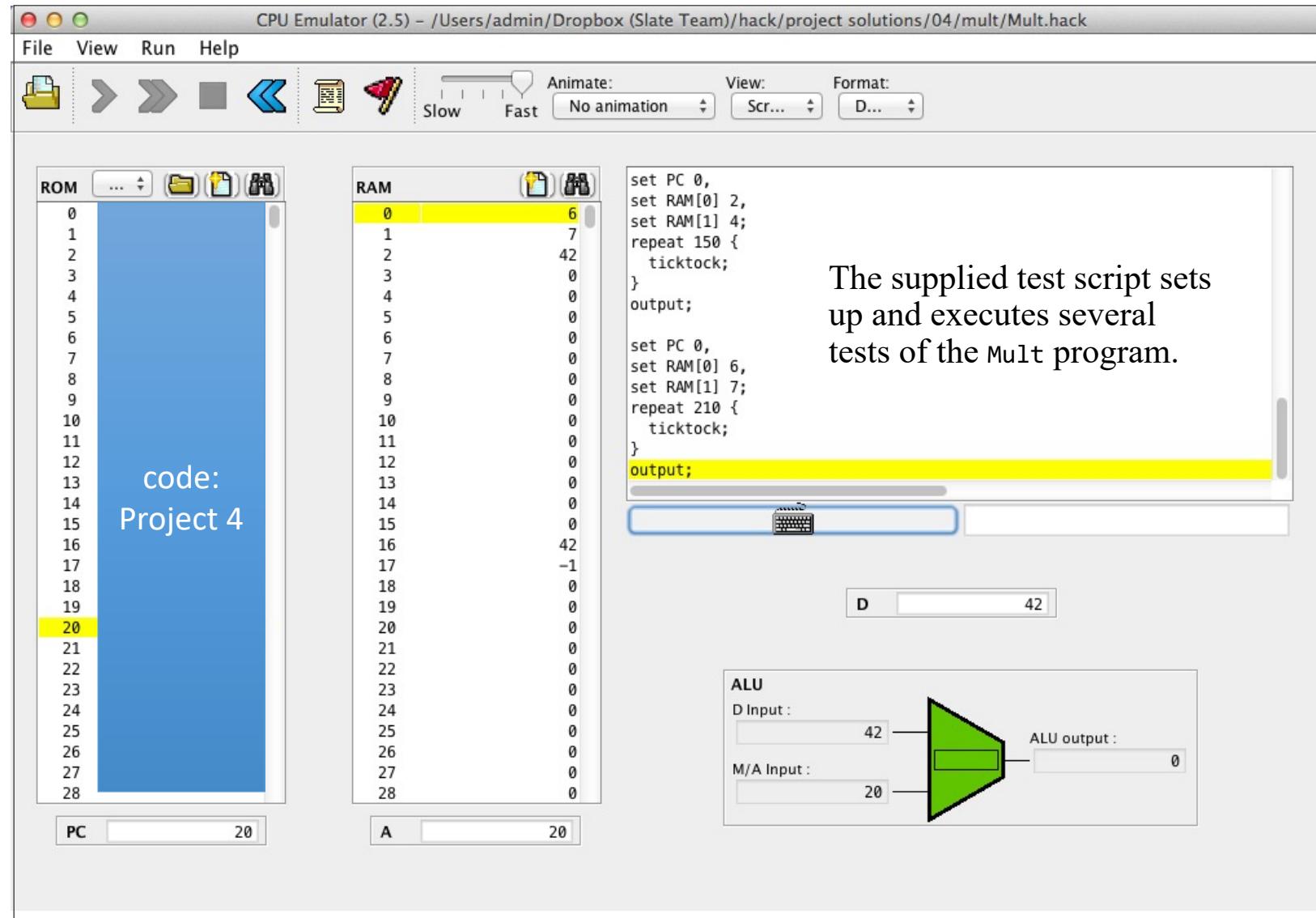
Tasks

- Write a simple algebraic program: `Mult`
- Write a simple interactive program: `Fill`
- Get creative: Define and write some program of your own (optional).

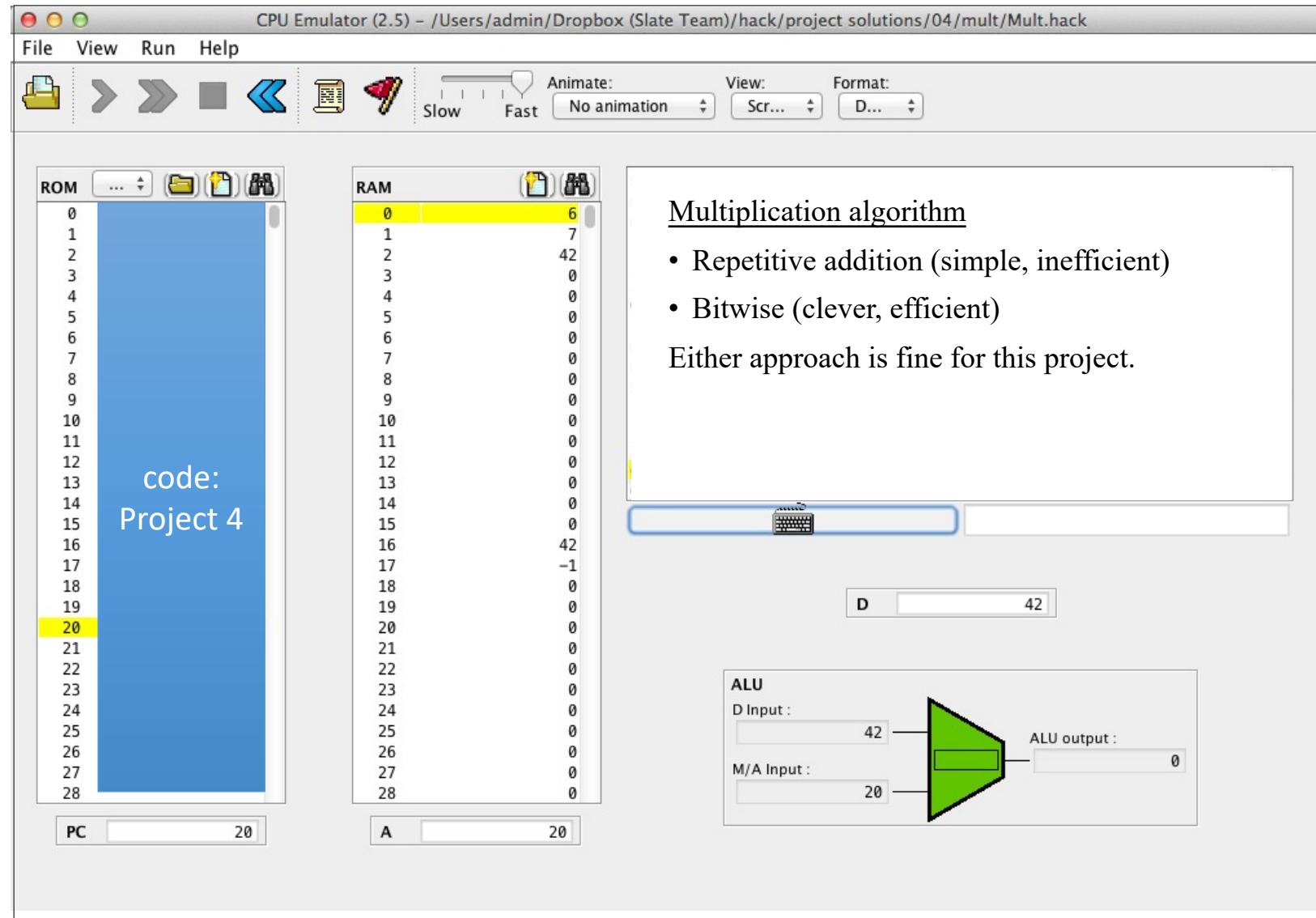
Mult: a program that computes $R2 = R0 * R1$



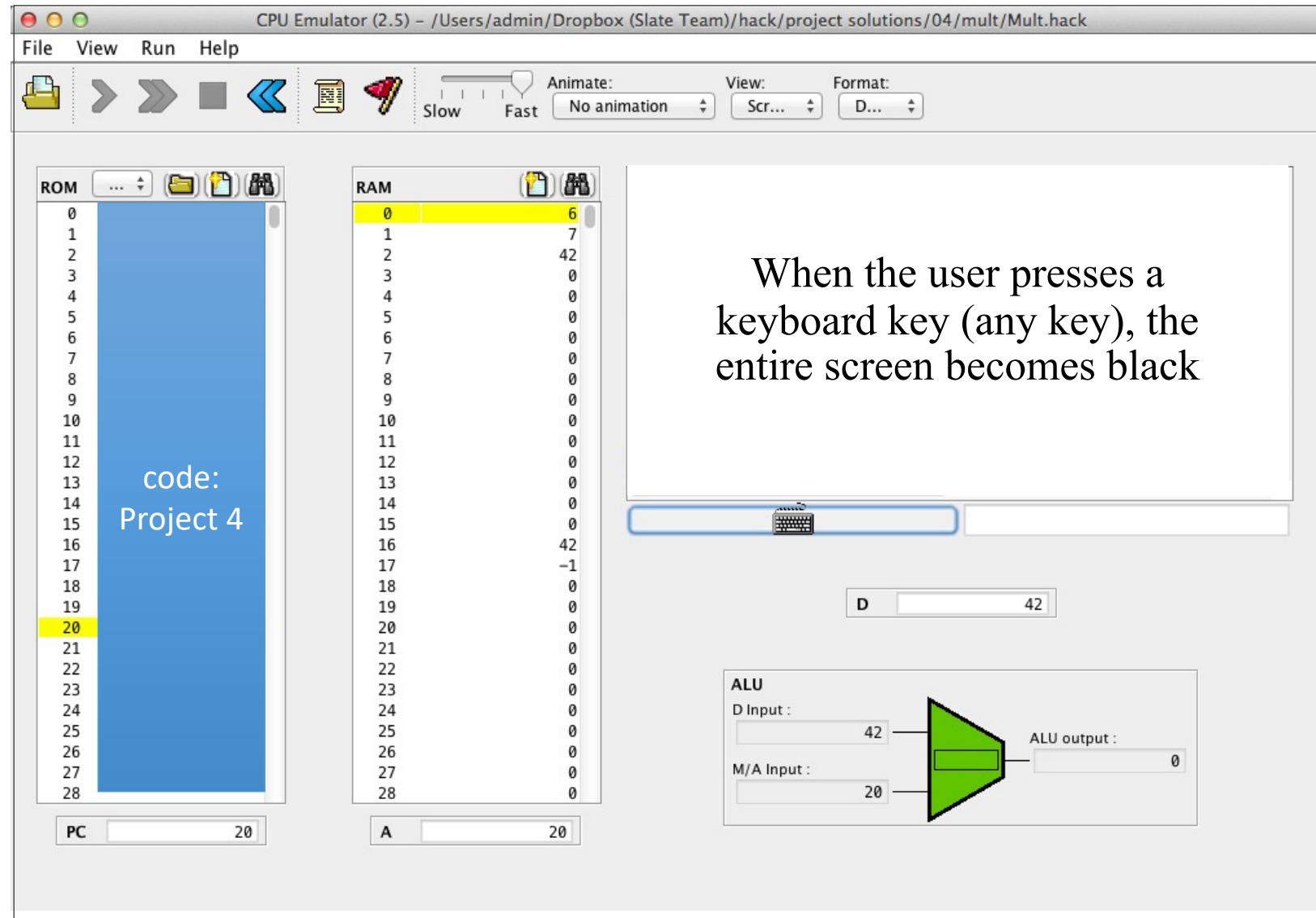
Mult: a program that computes $R2 = R0 * R1$



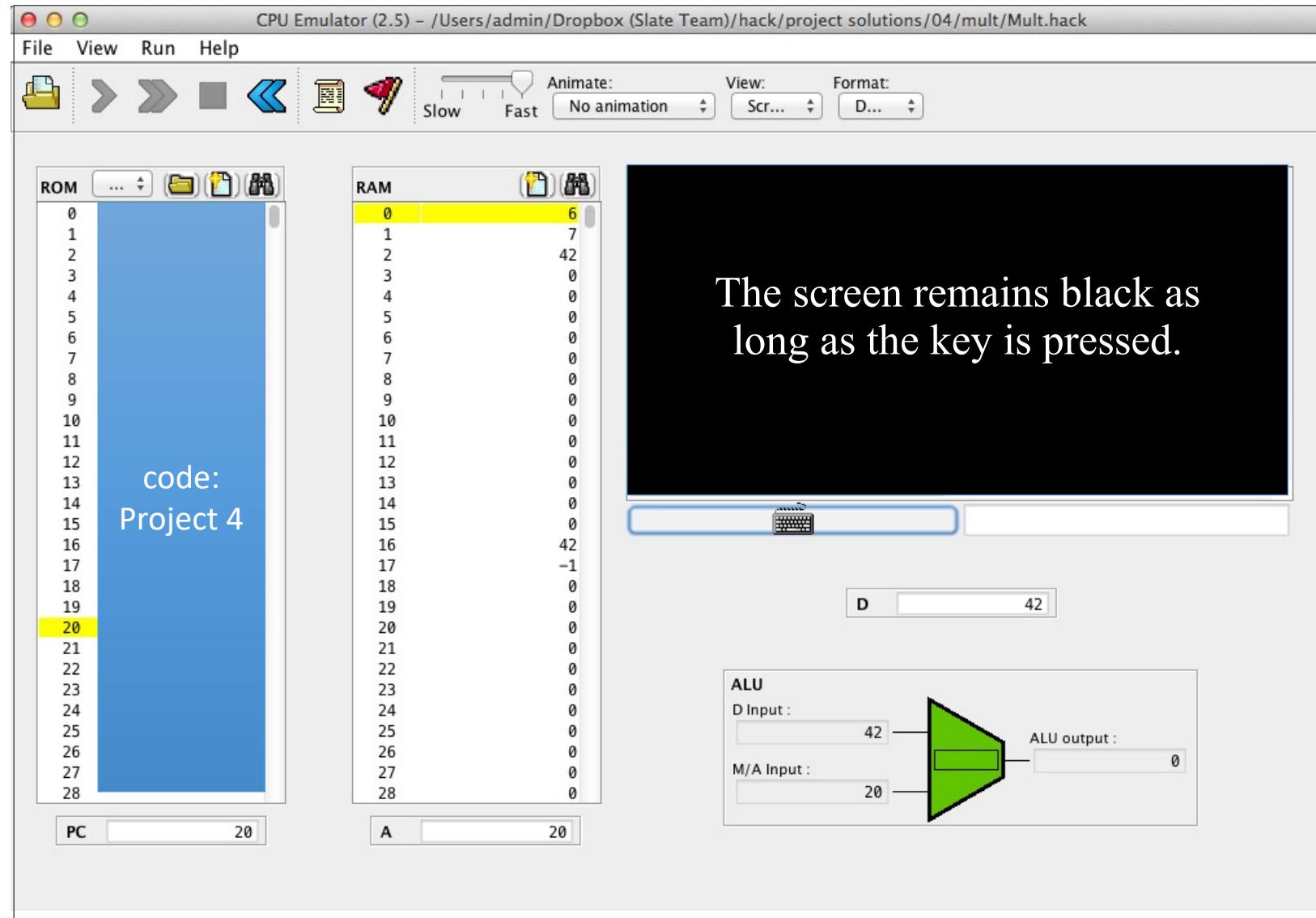
Mult: a program that computes $R2 = R0 * R1$



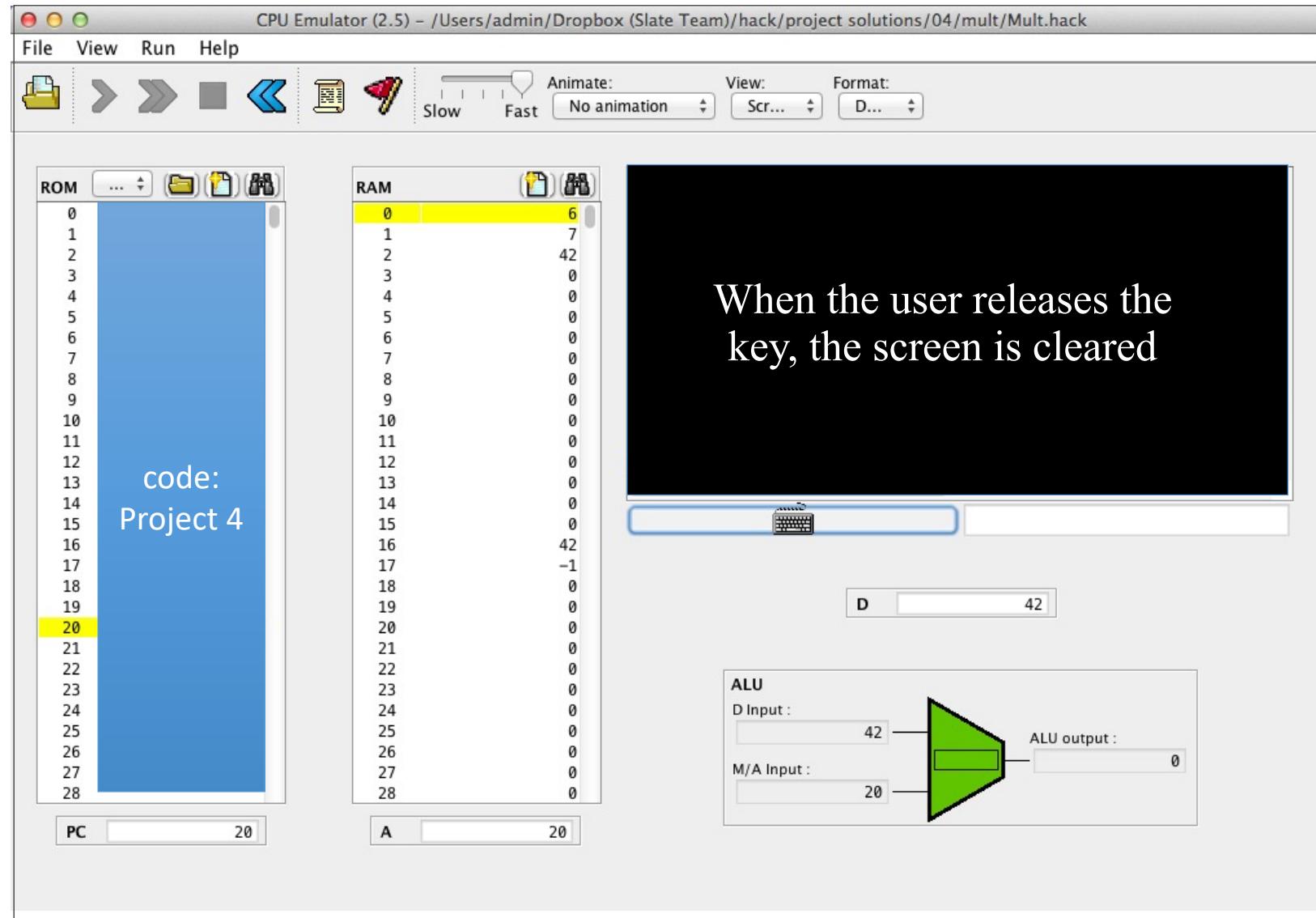
Fill: a simple interactive program



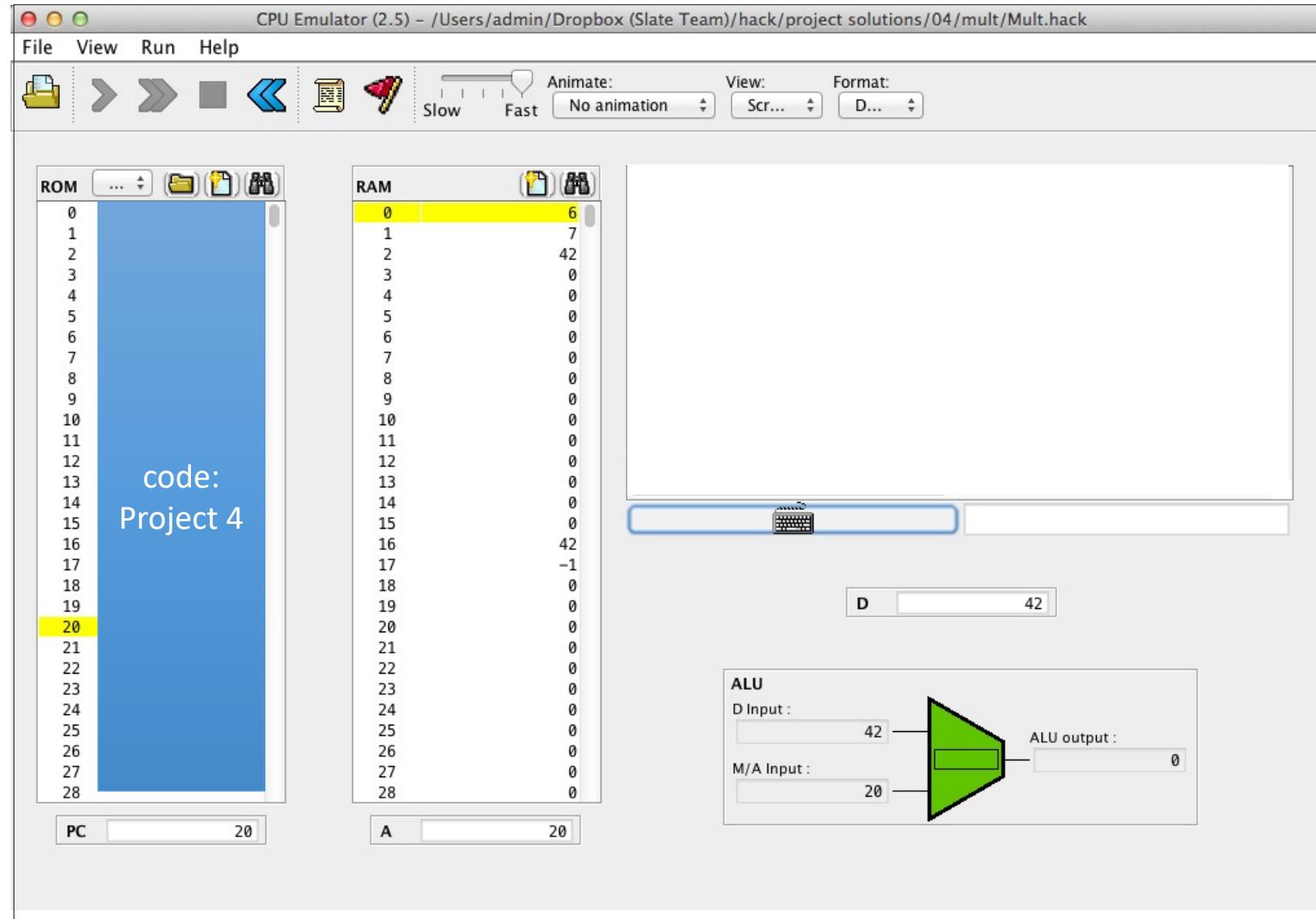
Fill: a simple interactive program



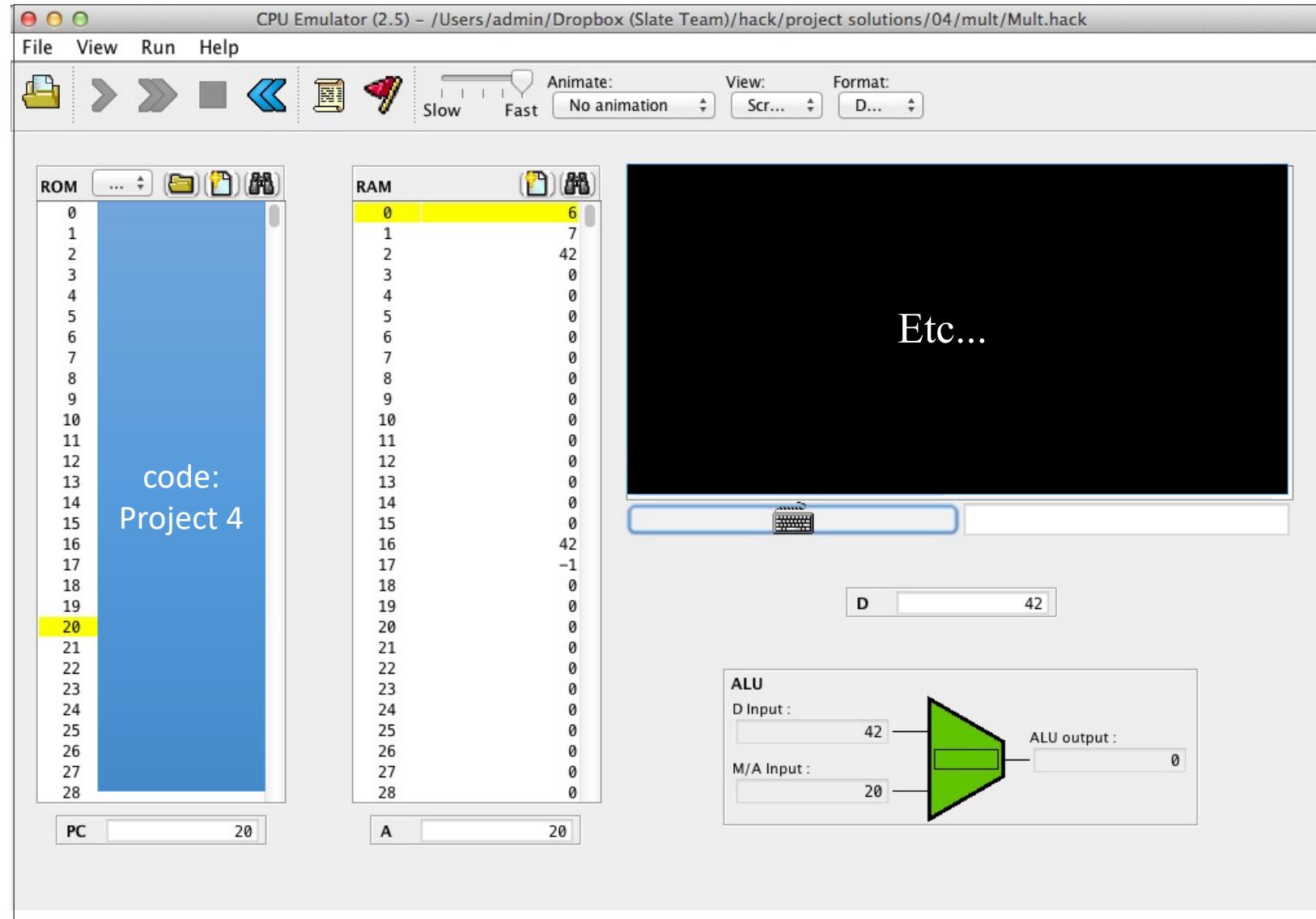
Fill: a simple interactive program



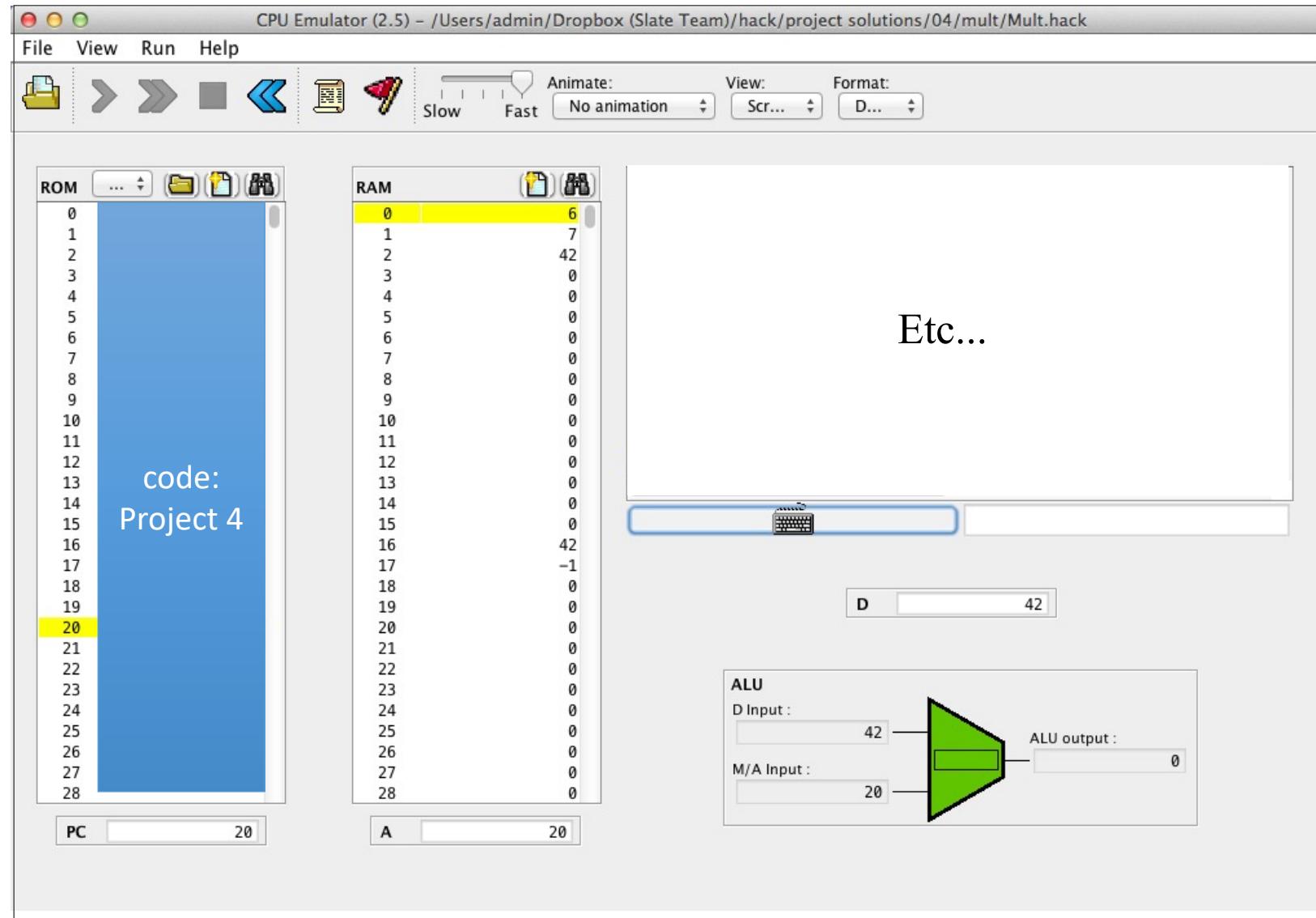
Fill: a simple interactive program



Fill: a simple interactive program



Fill: a simple interactive program



Fill: a simple interactive program

Algorithm

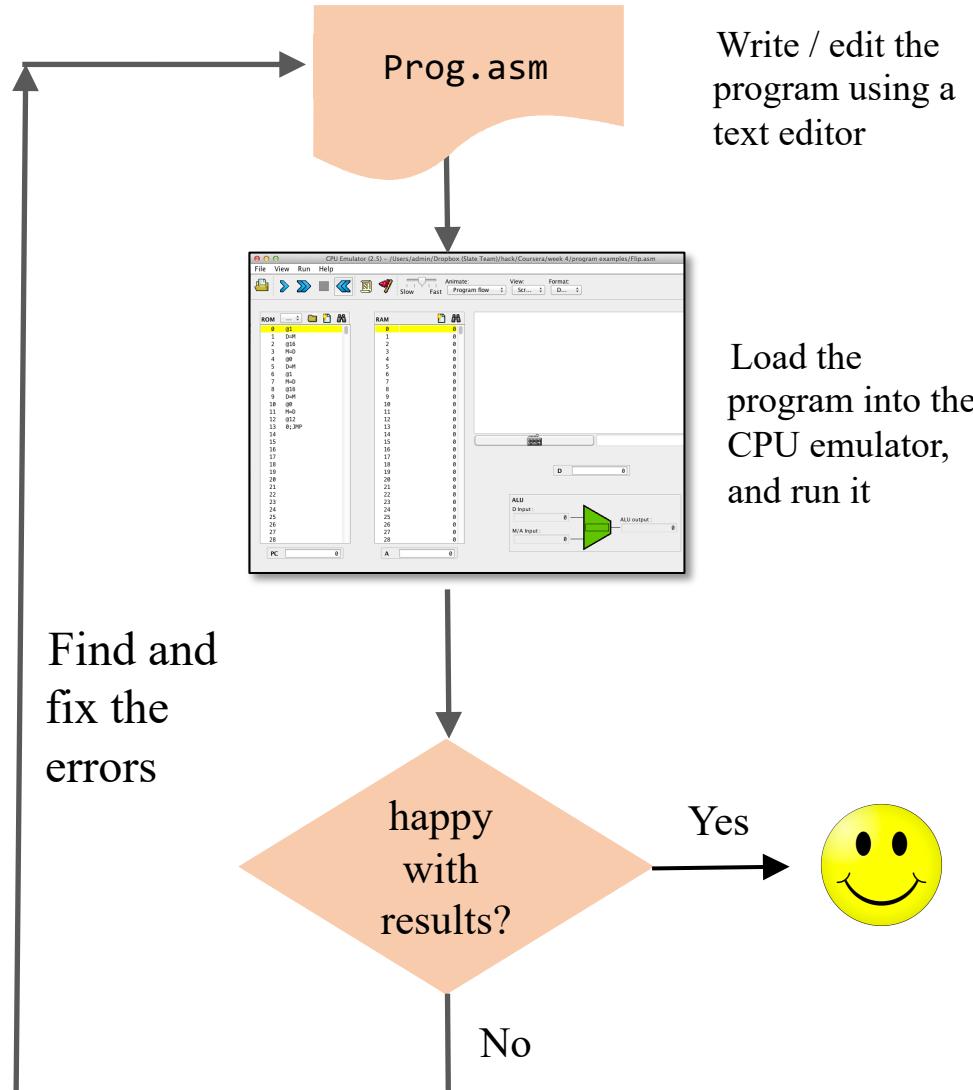
- Execute an infinite loop that listens to the keyboard input
- When a key is pressed (any key),
execute code that writes "black" in every pixel
- When no key is pressed, execute code that writes "white" in every pixel

Tip: This program requires working with pointers.

Task 3: Define and write a program of your own

Any ideas?
It's your call!

Program development process



Write / edit the program using a text editor

Load the program into the CPU emulator, and run it

Find and fix the errors

happy with results?

No

Translation options

1. Let the CPU emulator translate into binary code (as seen on the left)
2. Use the supplied assembler:
 - Find it on your PC in nand2tetris/tools
 - See the *Assembler Tutorial* in Project 6 (www.nand2tetris.org)

Implementation notes

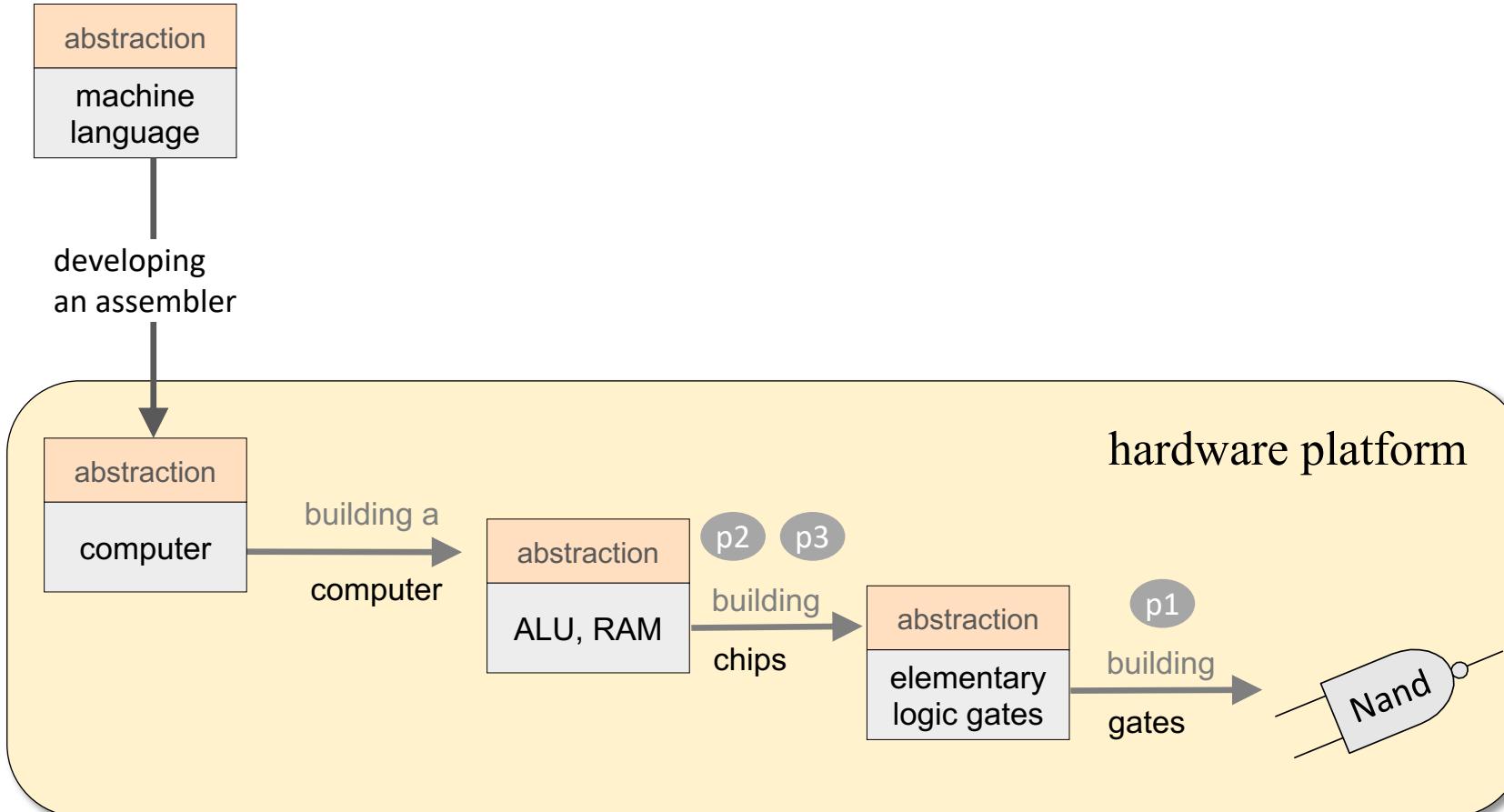
Well-written low-level code is

- Compact
- Efficient
- Elegant
- Self-describing

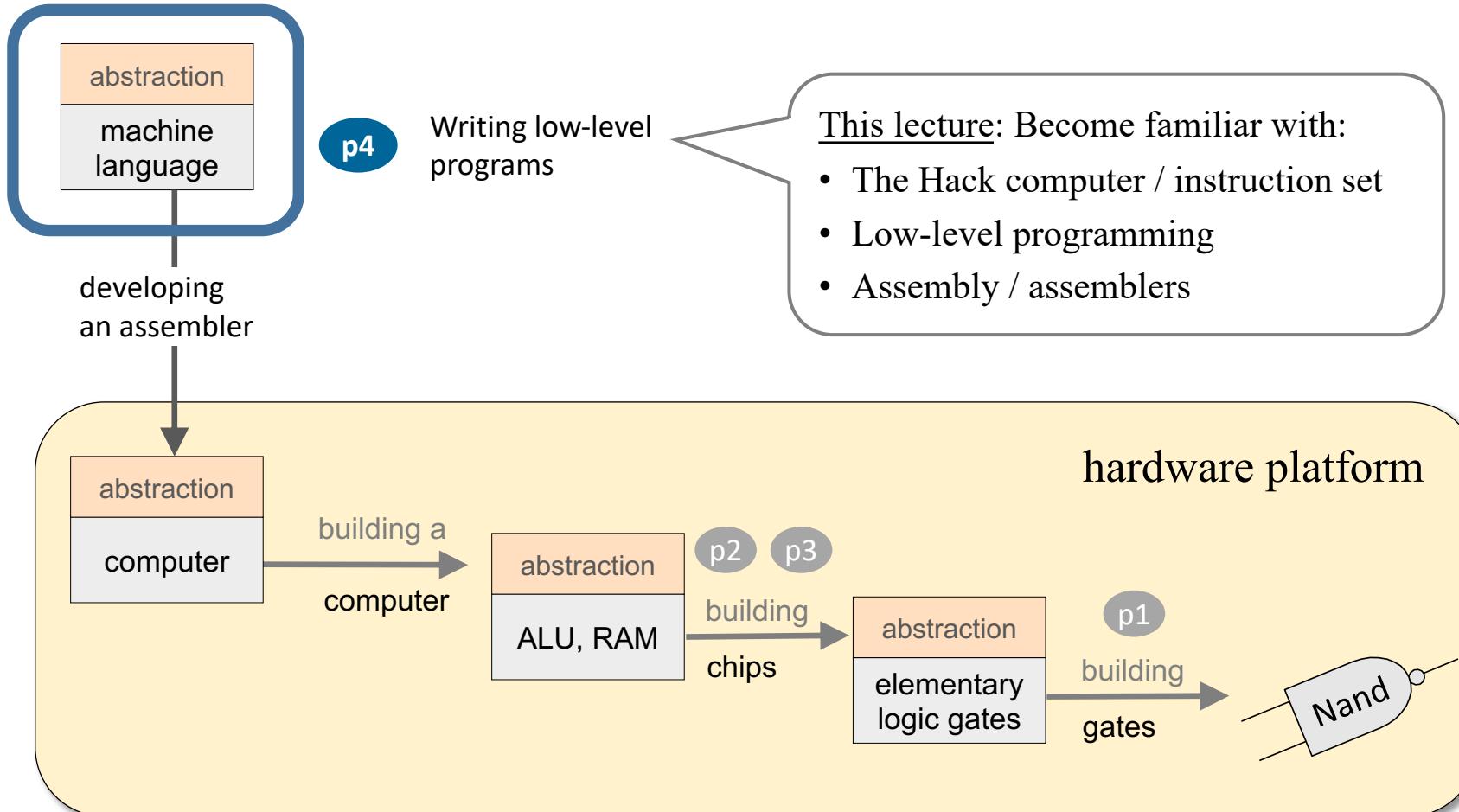
Tips

- Use symbolic variables and labels
- Use sensible documentation
- Use sensible variable and label names
- Variables: lower-case
- Labels: upper-case
- Use indentation
- Start by writing pseudocode.

Nand to Tetris Roadmap: Hardware



Nand to Tetris Roadmap: Hardware



Nand to Tetris Roadmap: Hardware

