Chapter 4

Machine Language

These slides support chapter 4 of the book

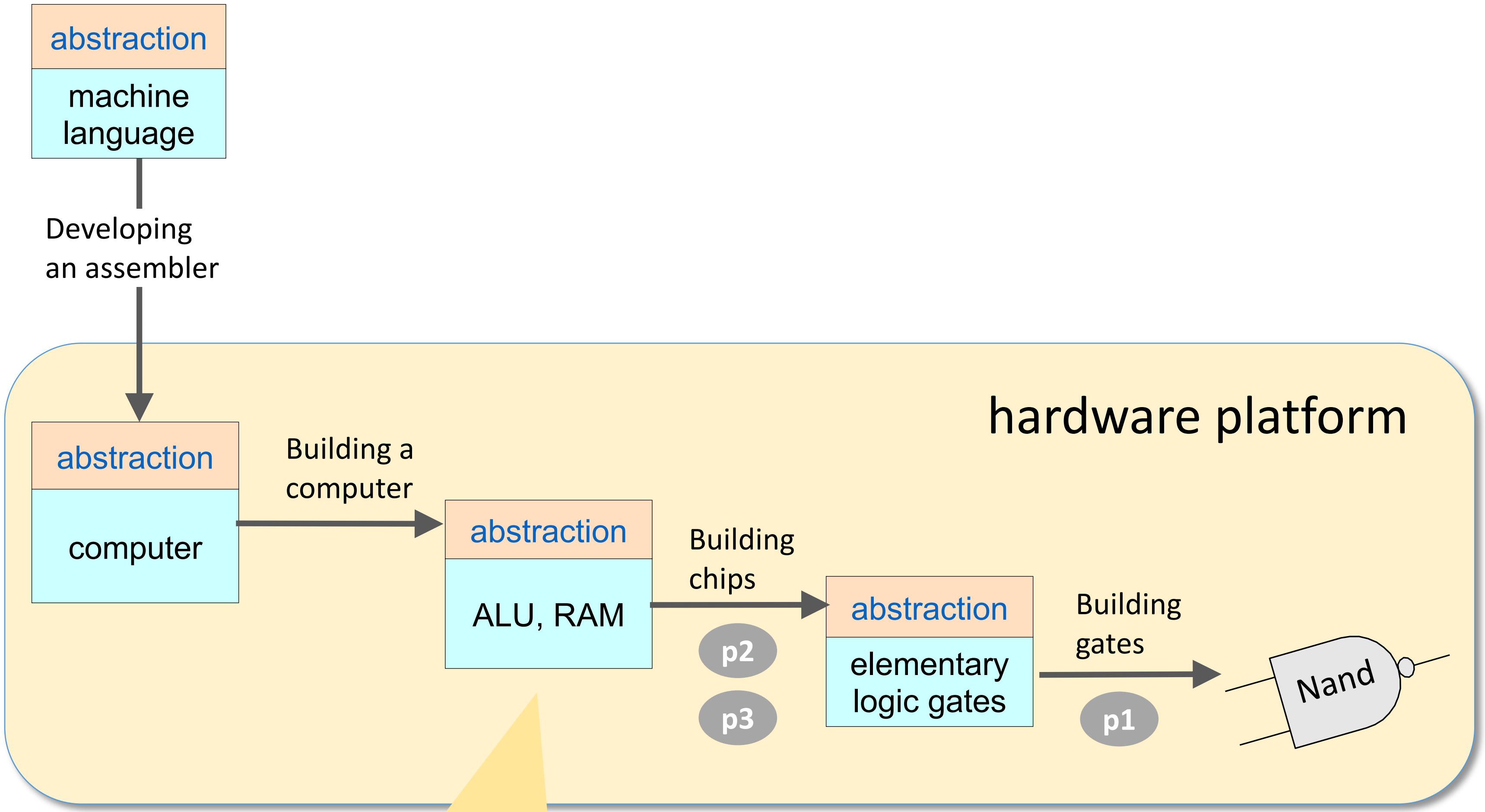
The Elements of Computing Systems

(1st and 2nd editions)

By Noam Nisan and Shimon Schocken

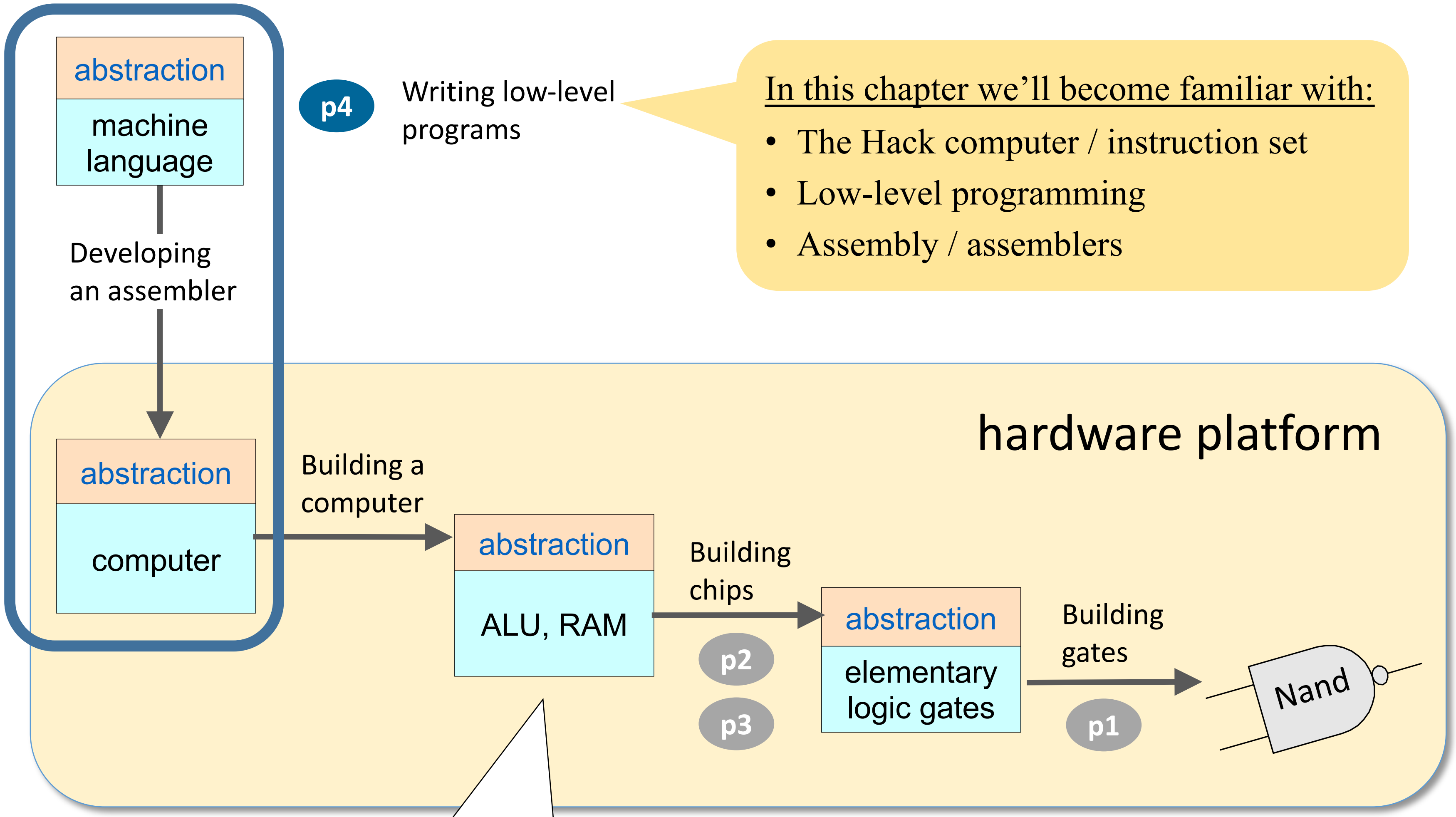
MIT Press

Nand to Tetris Roadmap (Part I: Hardware)



In previous chapters we've built the computer's ALU and RAM

Nand to Tetris Roadmap (Part I: Hardware)



In this chapter we'll become familiar with:

- The Hack computer / instruction set
- Low-level programming
- Assembly / assemblers

In previous chapters we've built the computer's ALU and RAM

Computers are flexible and versatile

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

Computers are flexible and versatile

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

Ada Lovelace

(1843)

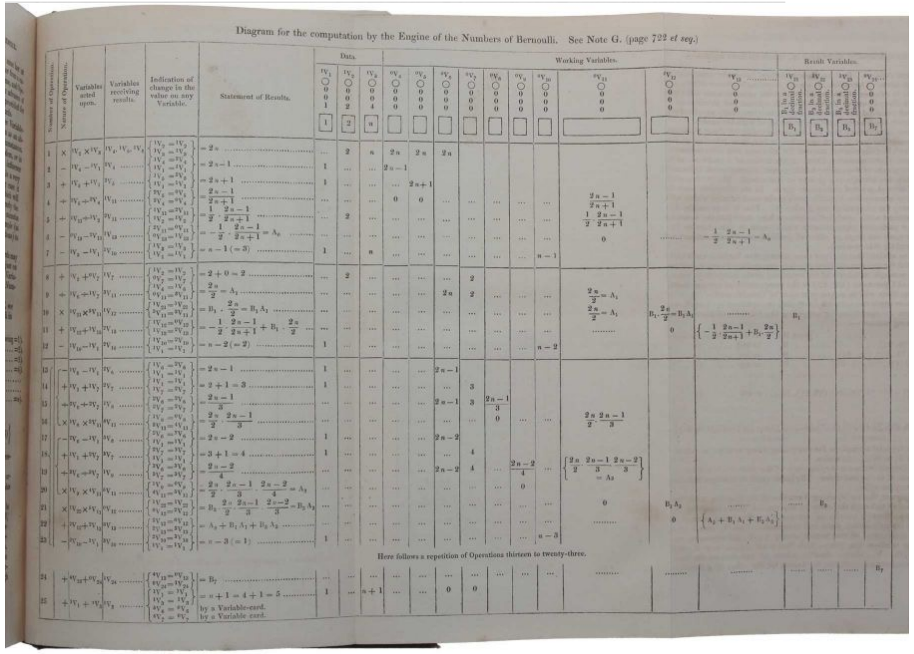


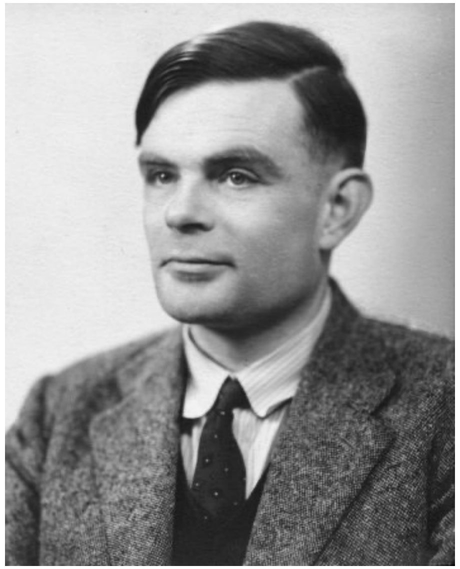
Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 et seq.)

Early symbolic program

Landmark “proof of concept” that computers can be programmed

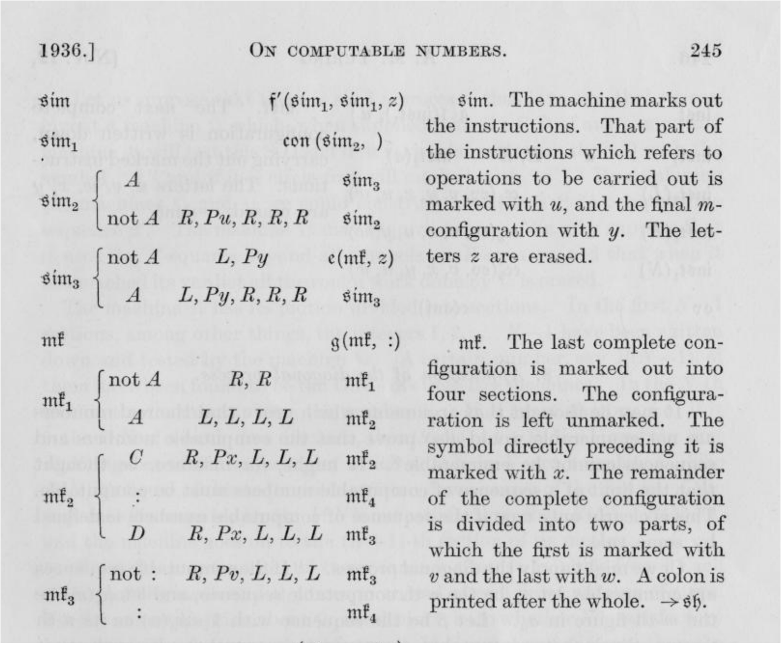
Computers are flexible and versatile

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



Alan Turing

(1936)



1936.] ON COMPUTABLE NUMBERS. 245

$\mathfrak{sim}$			$\mathfrak{f}'(\mathfrak{sim}_1, \mathfrak{sim}_1, z)$
$\mathfrak{sim}_1$			$\mathfrak{con}(\mathfrak{sim}_2, \ )$
$\mathfrak{sim}_2$	A		$\mathfrak{sim}_3$
	not A	R, Pu, R, R, R	$\mathfrak{sim}_2$
$\mathfrak{sim}_3$	not A	L, Py	$\mathfrak{e}(\mathfrak{mk}, z)$
	A	L, Py, R, R, R	$\mathfrak{sim}_3$
$\mathfrak{mk}$			$\mathfrak{g}(\mathfrak{mk}, :)$
$\mathfrak{mk}_1$	not A	R, R	$\mathfrak{mk}_1$
	A	L, L, L, L	$\mathfrak{mk}_2$
$\mathfrak{mk}_2$	C	R, Px, L, L, L	$\mathfrak{mk}_2$
	:		$\mathfrak{mk}_4$
	D	R, Px, L, L, L	$\mathfrak{mk}_3$
$\mathfrak{mk}_3$	not :	R, Pv, L, L, L	$\mathfrak{mk}_3$
	:		$\mathfrak{mk}_4$

$\mathfrak{sim}$. 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.

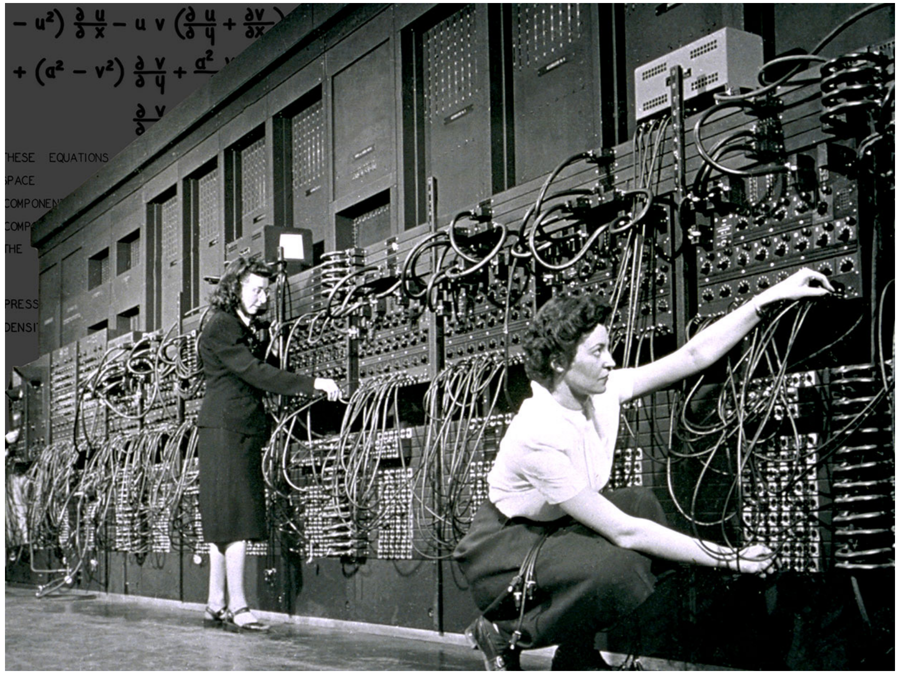
$\mathfrak{mk}$. The last complete configuration is marked out into four sections. The configuration is left unmarked. The symbol directly preceding it is marked with x. The remainder of the complete configuration is divided into two parts, of which the first is marked with v and the last with w. A colon is printed after the whole. $\rightarrow \mathfrak{sh}$.

<u>Universal Turing Machine</u>

Landmark article, describing a theoretical general-purpose computer

Computers are flexible and versatile

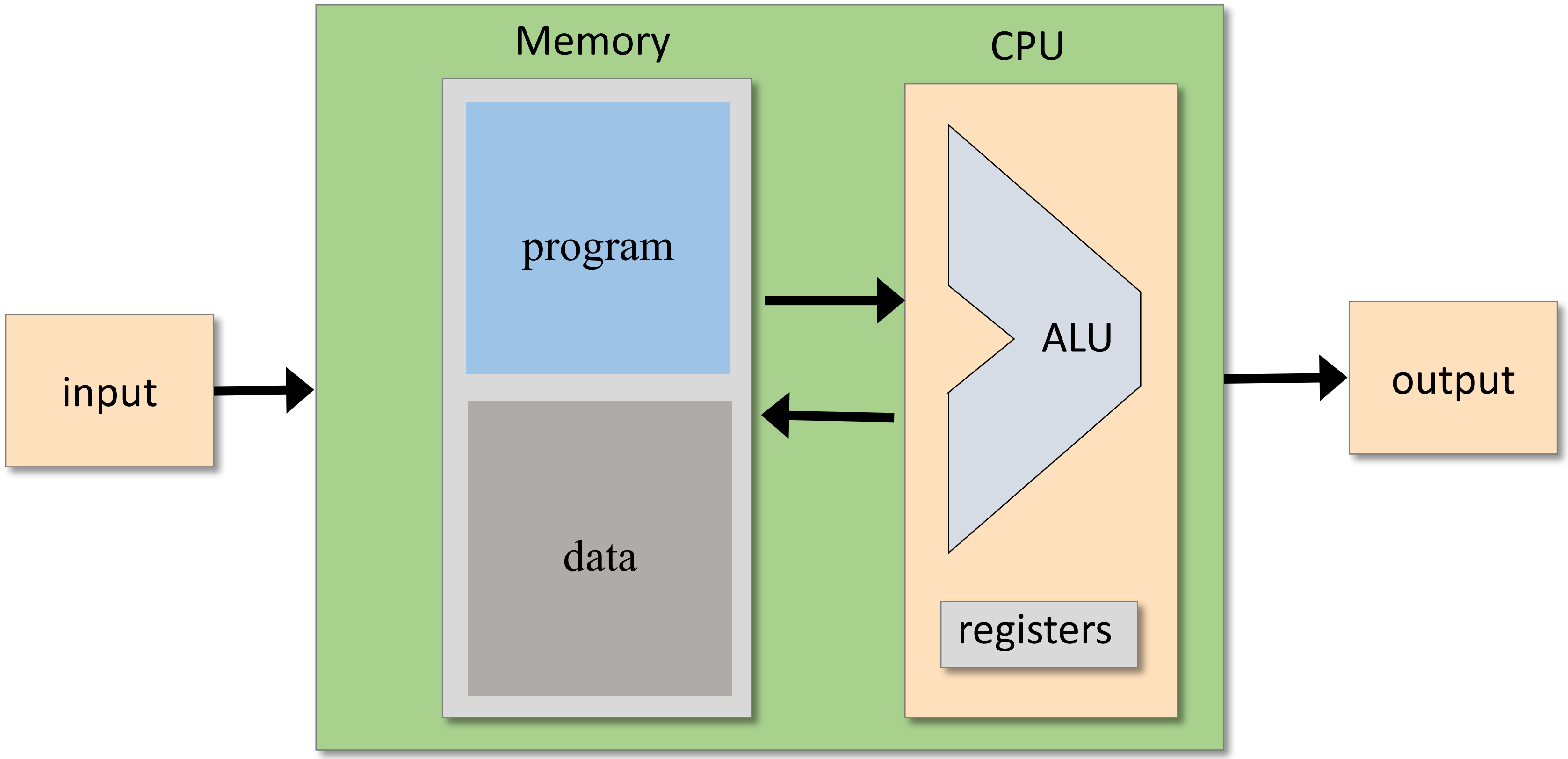
Same **hardware** can run many different programs (**software**)



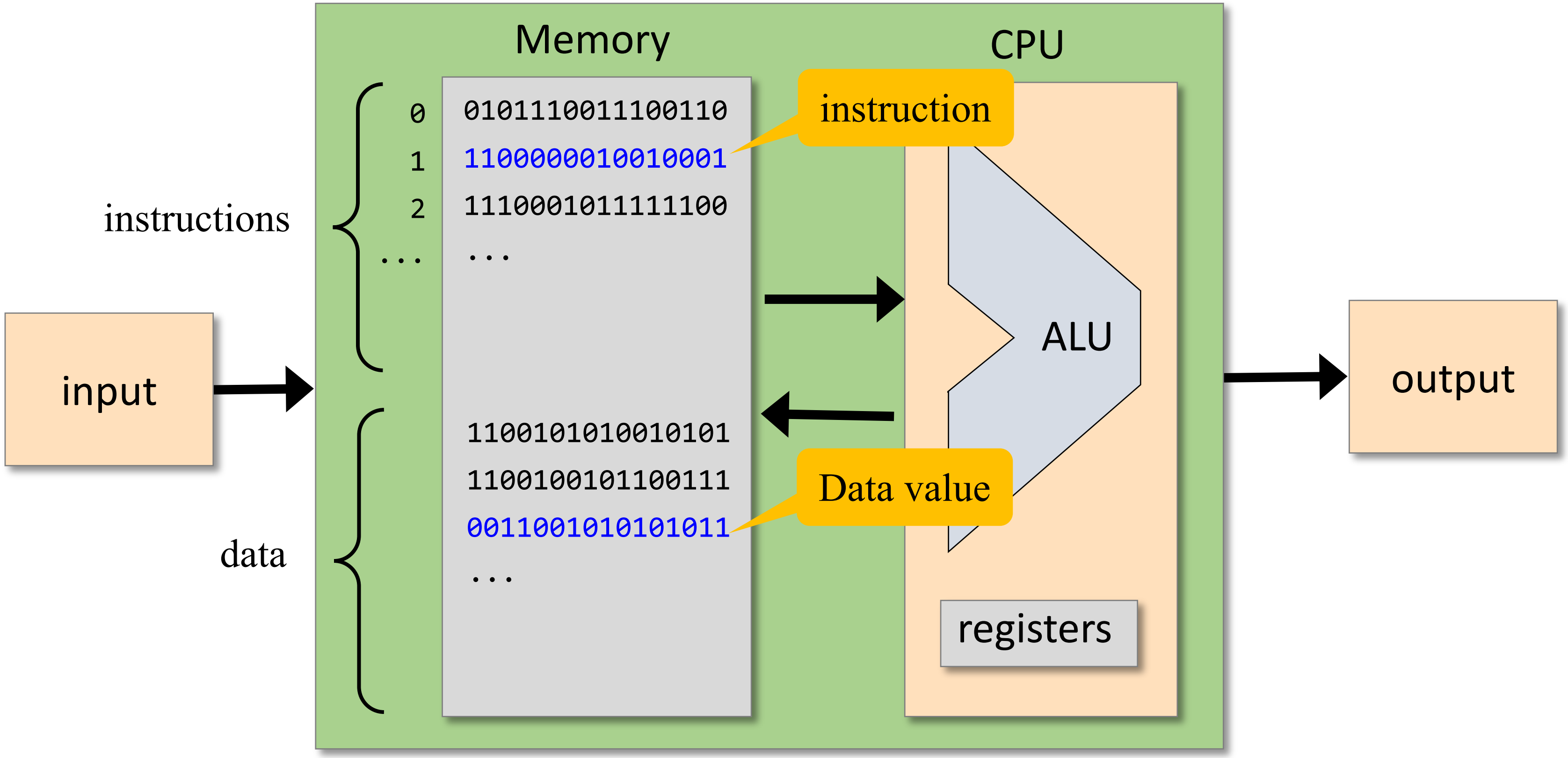
First general-purpose computer

Eniac, University of Pennsylvania, 1945

Computer architecture



Computer architecture



Stored program concept

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

One of the most important ideas in the history of computer science

Chapter 4: Machine Language

Overview

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

Low Level Programming

- Basic
- Iteration
- Pointers

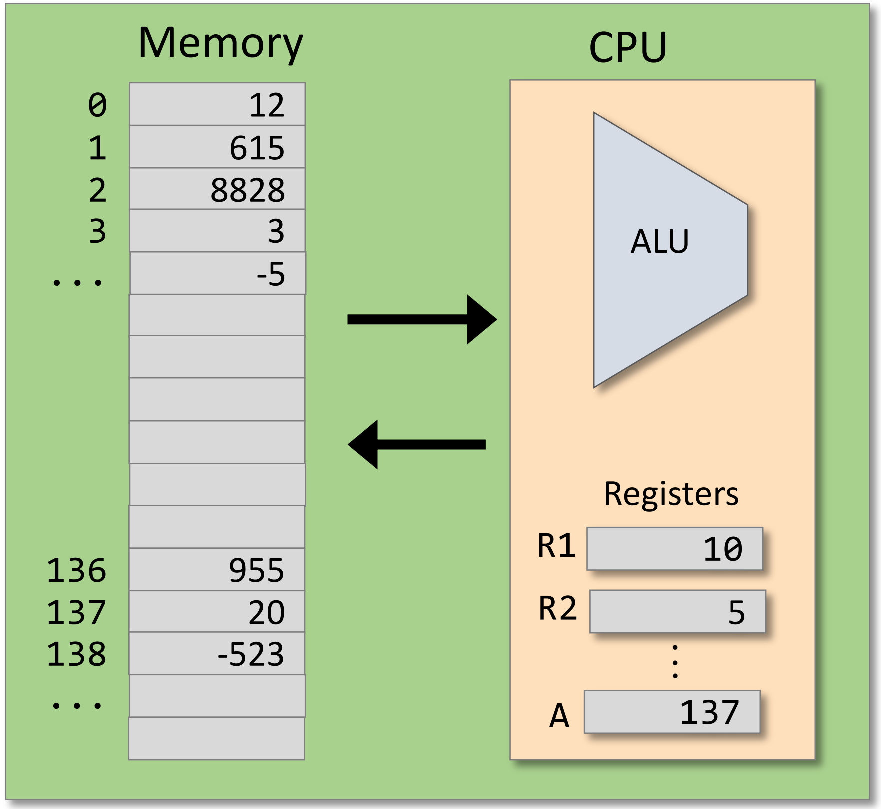
Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- 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 supplying an address)

Machine language

A formalism specifying how to access and manipulate registers.

Registers

<u>Data registers:</u>

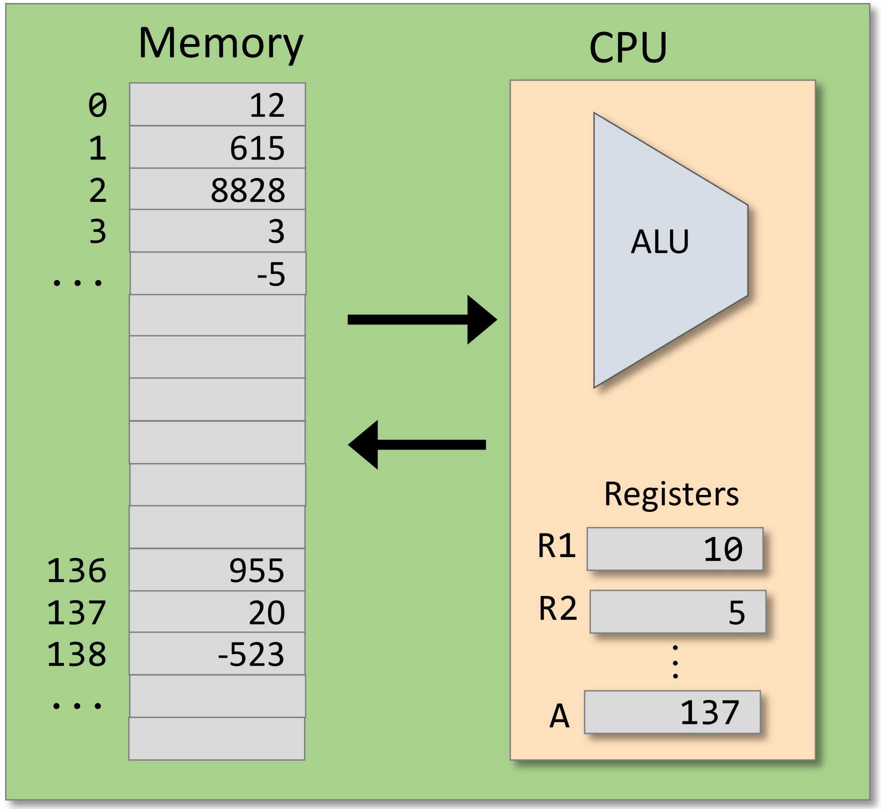
Hold data values

<u>Address register:</u>

Holds an address

<u>Instruction register:</u>

Holds an instruction



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

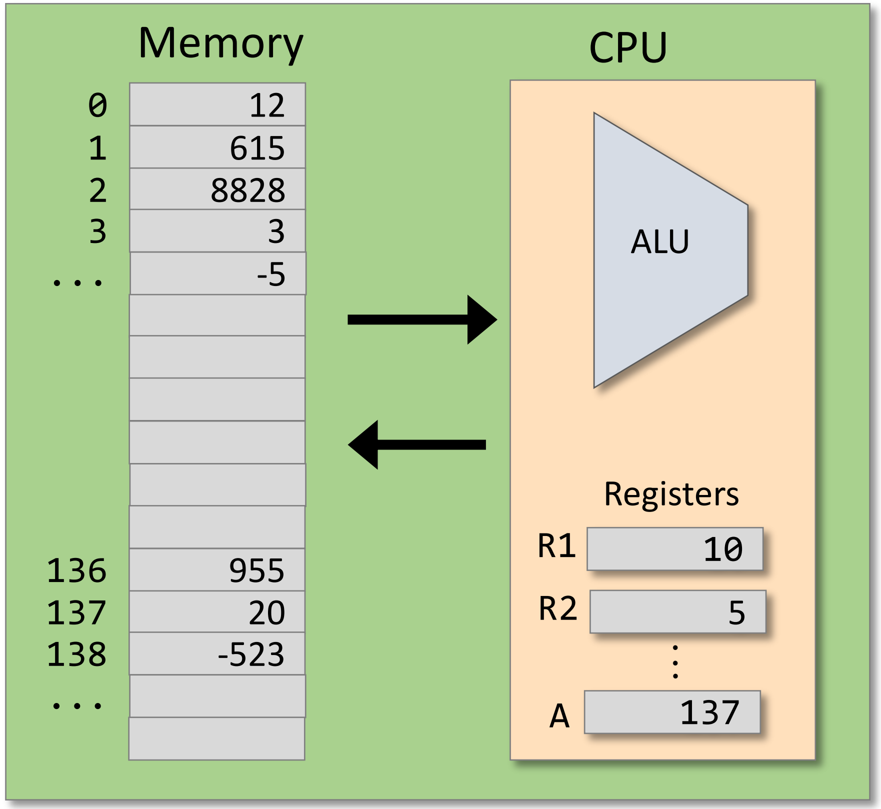
Typical operations

```
// R1 ← 73
load R1, 73

// R1 ← R1 + R2
add R1, R2

// R1 ← R1 + Memory[137]
add R1, M[137]

// R1 ← Memory[A]
load R1, @A
```

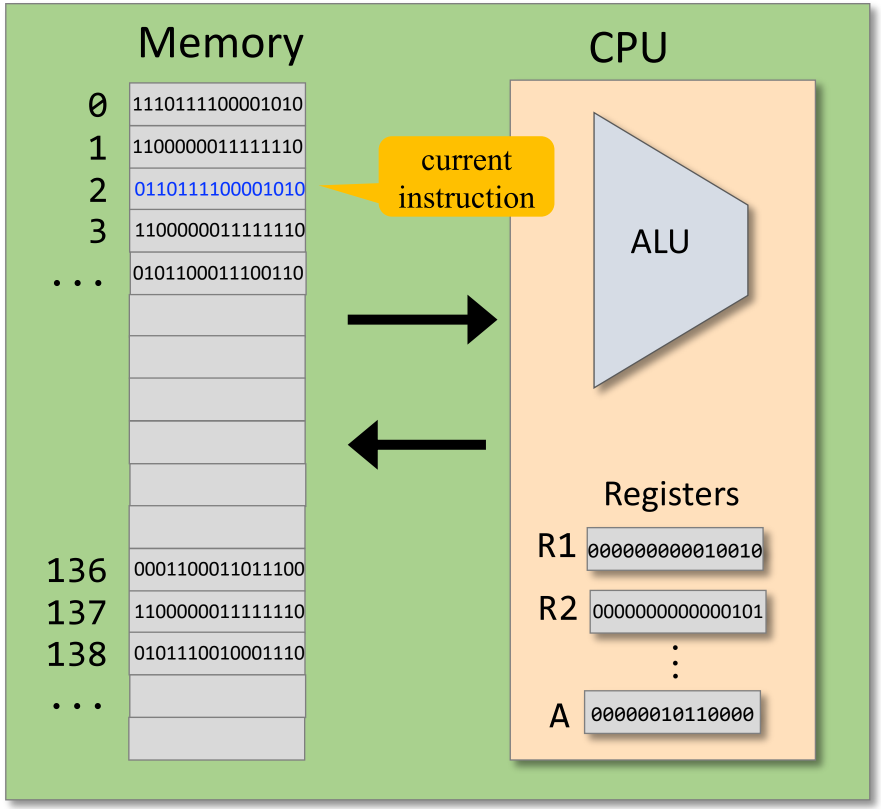


<u>The syntax</u> of machine languages varies greatly from one computer to another, but all of them are designed to do the same thing: Manipulate registers.

Control

<u>Which instruction should be executed next?</u>

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



Control

<u>Unconditional branching</u>

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

Basic version

```
    ...
    // Adds 1 to R1, repetitively
13  add R1,1
... ...
27  goto 13
... ...
```

- Line numbers
- Physical addresses

Symbolic version

```
    ...
    // Adds 1 to R1, repetitively
(LOOP)
    add R1,1
    ...
    goto LOOP
    ...
```

- No line numbers
- Symbolic addresses

<u>Programs with symbolic references are</u> …

- Easier to develop
- Readable
- Relocatable.

Control

Conditional branching

Sometimes we want to "jump" to execute another instruction, but only if a certain condition is met

Symbolic program

```
  // Set R1 to abs(R1).

  // if R1 > 0 goto CONT
  jgt R1,CONT

  // R1 ← -R1
  store R2,R1
  store R1,0
  subt R1,R2

CONT:
  // Here R1 is non-negative
  ...
```

How can we actually execute the program?

Control

Conditional branching

Sometimes we want to "jump" to execute another instruction,
but only if a certain condition is met

Symbolic program

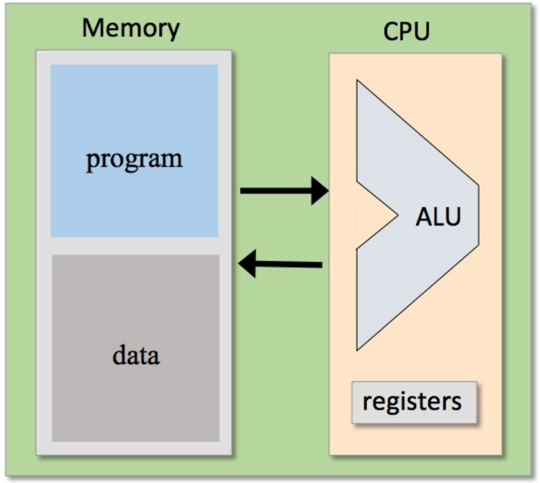
```
  // Set R1 to abs(R1).
  // if R1 > 0 goto CONT
  jgt R1,CONT
  // R1 ← −R1
  store R2,R1
  store R1,0
  subt R1,R2
CONT:
  // Here R1 is non-negative
  ...
```

translate

Binary code

```
0101111100111100
1010101010101010
1100000010101010
1011000010000001
...
```

load and execute



Assembly

Assembler

Chapter 4: Machine Language

Overview

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

Low Level Programming

- Basic
- Iteration
- Pointers

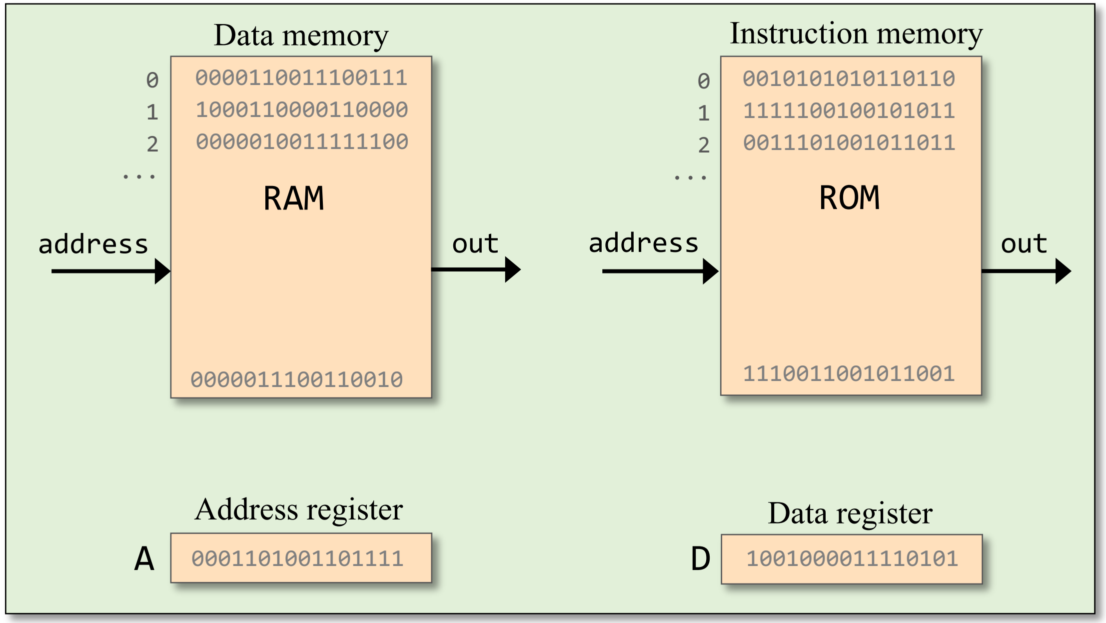
Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

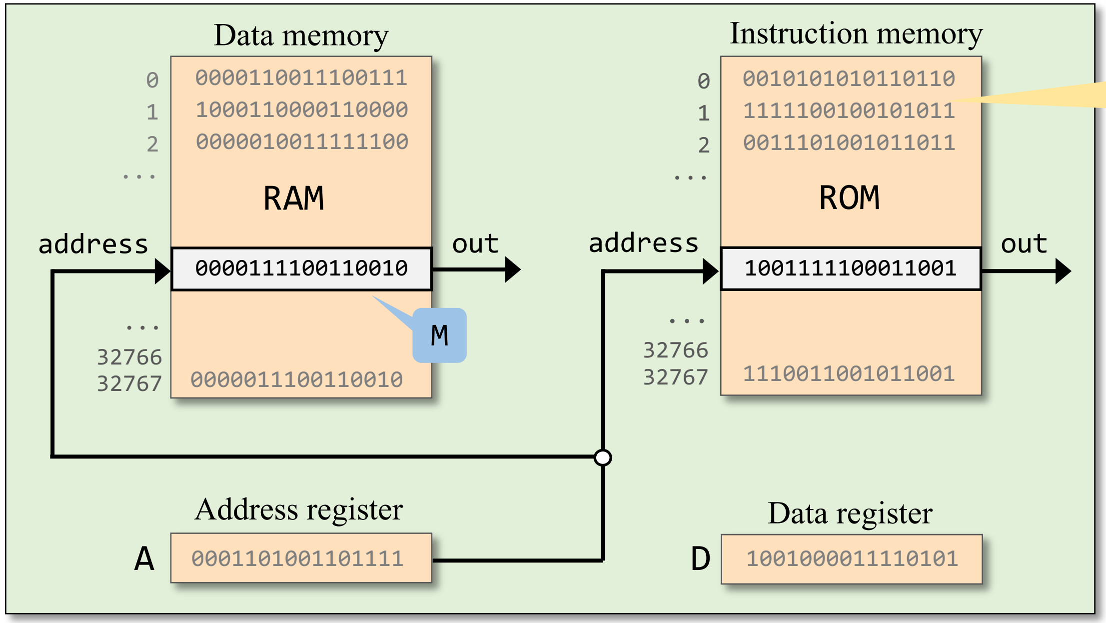
The Hack computer



(Conceptual, partial view of the Hack computer architecture)

- Hack is a 16-bit computer, featuring two memory units
- The address input of each memory unit is 15-bit wide
- **Question:** How many words can each memory unit have?
- **Answer:** The *address space* of each memory unit is 2^{15} = 32K words.

Memory



Loaded with a sequence of 16-bit Hack instructions

(Conceptual, partial view of the Hack computer architecture)

RAM

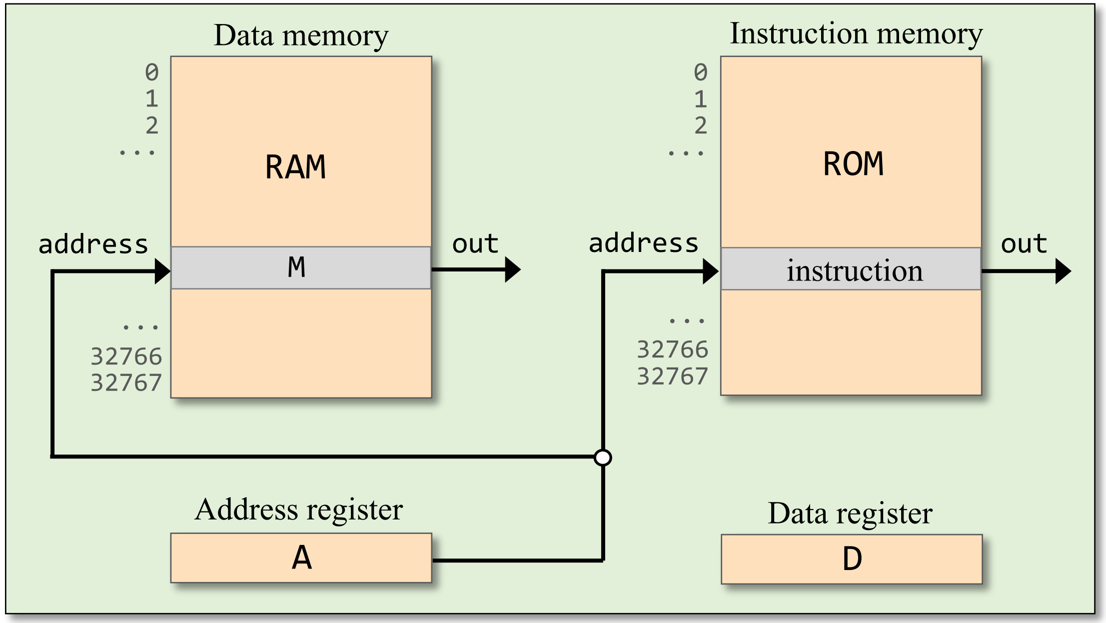
- Read-write data memory
- Addressed by the `A` register
- The selected register, `RAM[A]`, is represented by the symbol `M`

ROM

- Read-only instruction memory
- Addressed by the (same) `A` register
- The selected register, `ROM[A]`, contains the "current instruction"

- Should we focus on `RAM[A]`, or on `ROM[A]`?
- Depends on the current instruction (later)

Registers



(Conceptual, partial view of the Hack computer architecture)

`D`: data register

`A`: address register

`M`: the selected RAM register

Chapter 4: Machine Language

Overview

- ✓ Machine languages
- ✓ The Hack computer
- ➡ The Hack instruction set
- The Hack CPU Emulator

Low Level Programming

- Basic
- Iteration
- Pointers

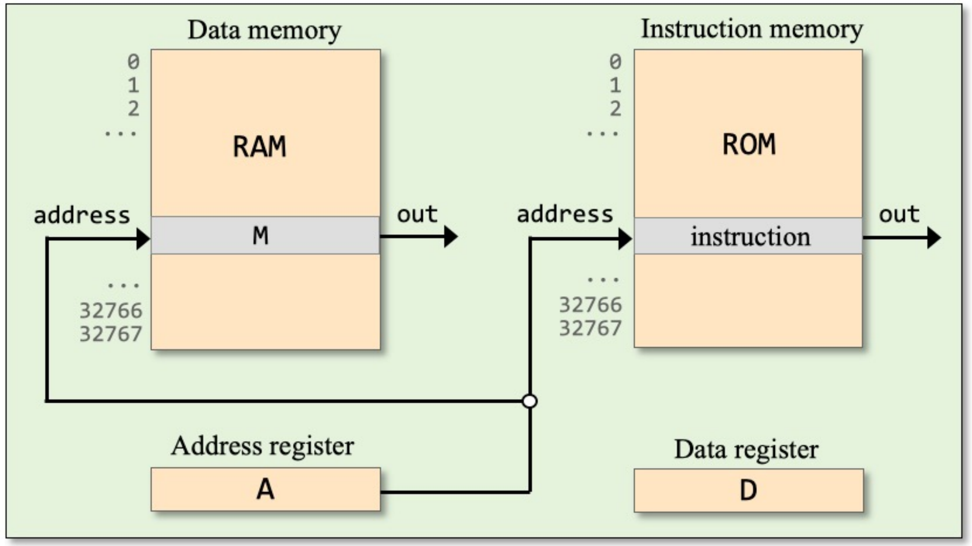
Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Hack instructions



Instruction set

- A instruction
- C instruction

Syntax:

`@const`

where *const* is a constant

(Complete / formal syntax, later).

Example:

`@19`

Semantics:

A ← 19

Side effects:

- `RAM[A]` (called `M`) becomes selected
- `ROM[A]` becomes selected

Hack instructions

Instruction set

- `A` instruction
- `C` instruction

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`$\}$, $op = \{$`+`$|$`-`$\}$, and $reg_2 = \{$`A`$|$`D`$|$`M`$|$`1`$\}$ and $reg_1 \neq reg_2$

Examples:

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

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

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

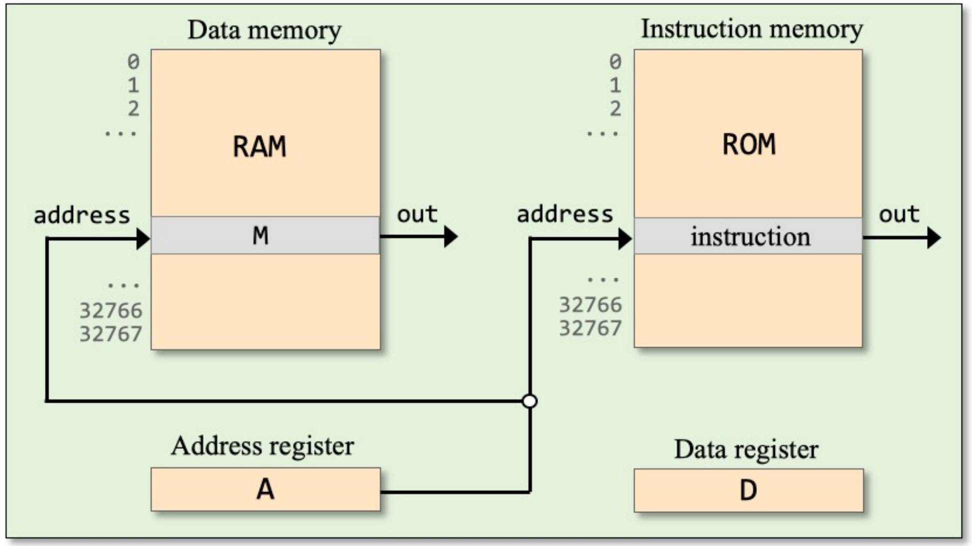
(Complete / formal syntax, later).

Hack instructions

Typical instructions:

@*constant*	(A ← *constant*)

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



Examples:

```
// D ← 2
?
```

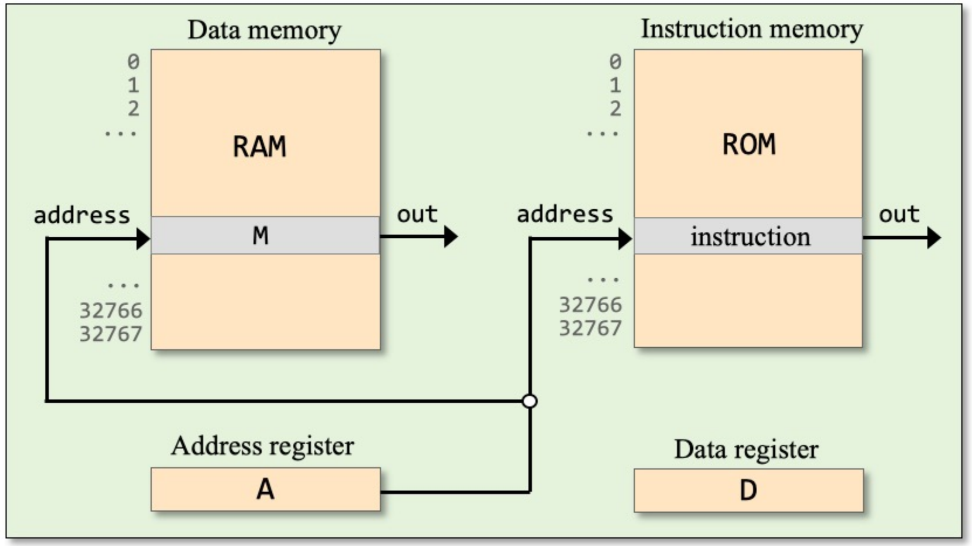
<u>The game:</u> We show some typical Hack instructions (top left), and practice writing code examples that use subsets of these instructions.

Hack instructions

Typical instructions:

@*constant*	(A←*constant*)

D=1 D=A D=D+1 ...	D=D+A D=M M=0 ...	M=D D=D+A M=M–D ...



Examples:

```
// D ← 2
D=1
D=D+1
```

```
// D ← 1954
?
```

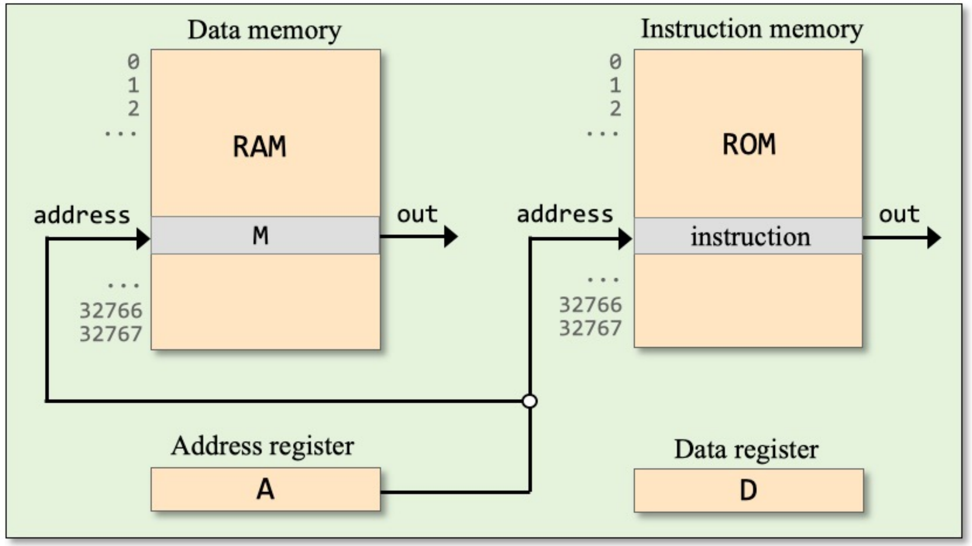
Use only the instructions shown in this slide

Hack instructions

Typical instructions:

@*constant*	(A←*constant*)

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



Examples:

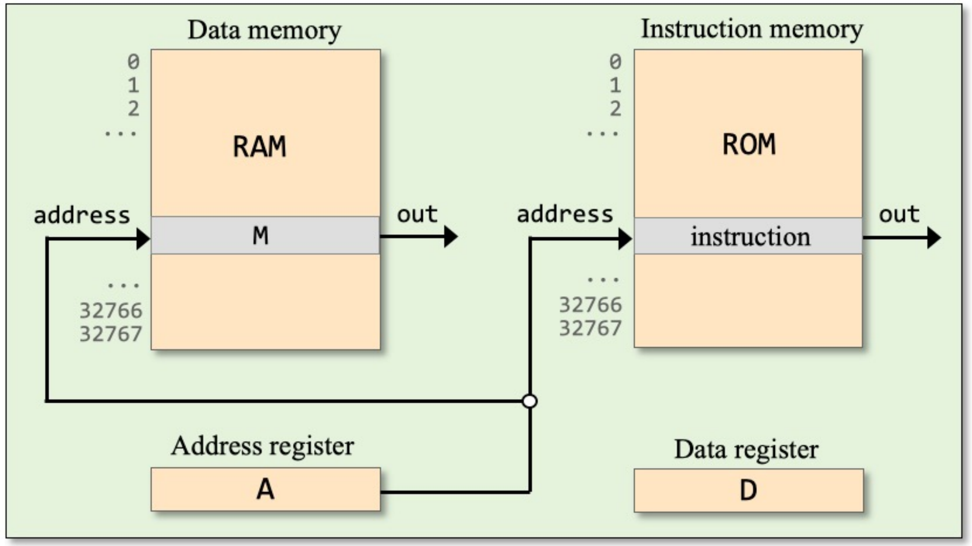
```
// D ← 2
D=1
D=D+1
```

```
// D ← 1954
@1954
D=A
```

```
// D ← D + 23
?
```

Use only the instructions shown in thus slide

Hack instructions



Typical instructions:

`@constant` (A ← *constant*)

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

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

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

Examples:

```
// D ← 2
D=1
D=D+1
```

```
// D ← 1954
@1954
D=A
```

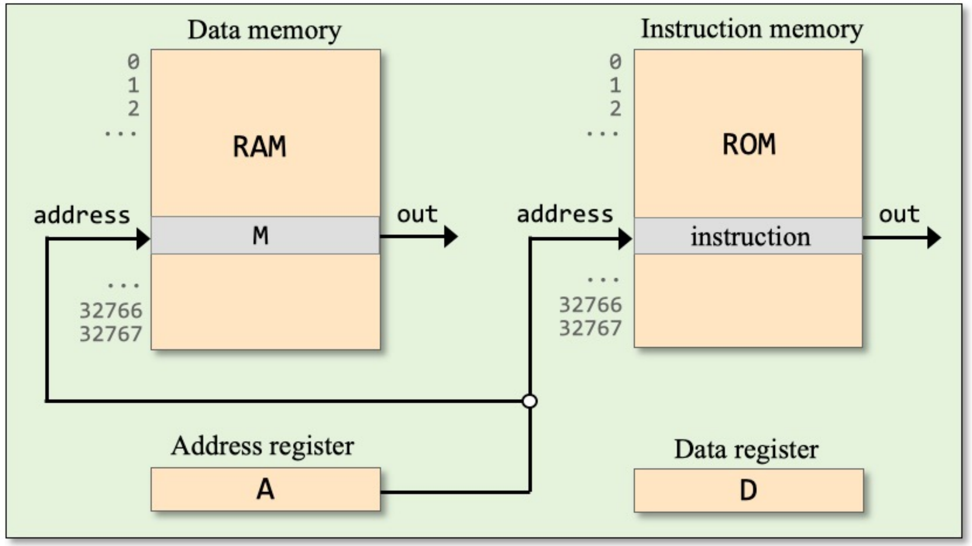
```
// D ← D + 23
@23
D=D+A
```

<u>Observation</u>

In these examples we use the address register `A` as a *data register*:

The addressing side-effects of `A` are ignored.

Hack instructions



Typical instructions:

`@constant`   (A ← *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] ← 0
@100
M=0
```

```
// RAM[100] ← 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

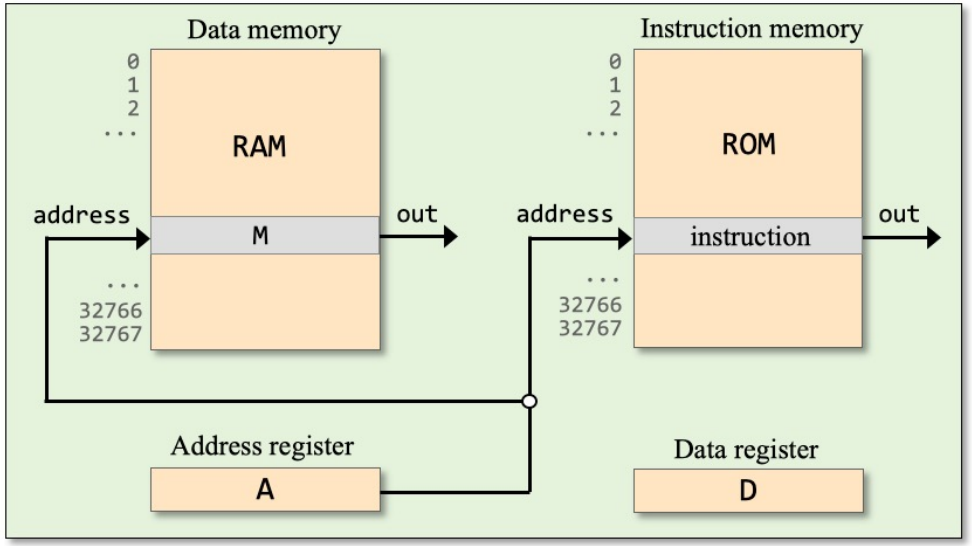
Typical instructions:

@*constant*	(A ← *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] ← 0
@100
M=0
```

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

```
// RAM[100] ← RAM[200]
?
```

Hack instructions

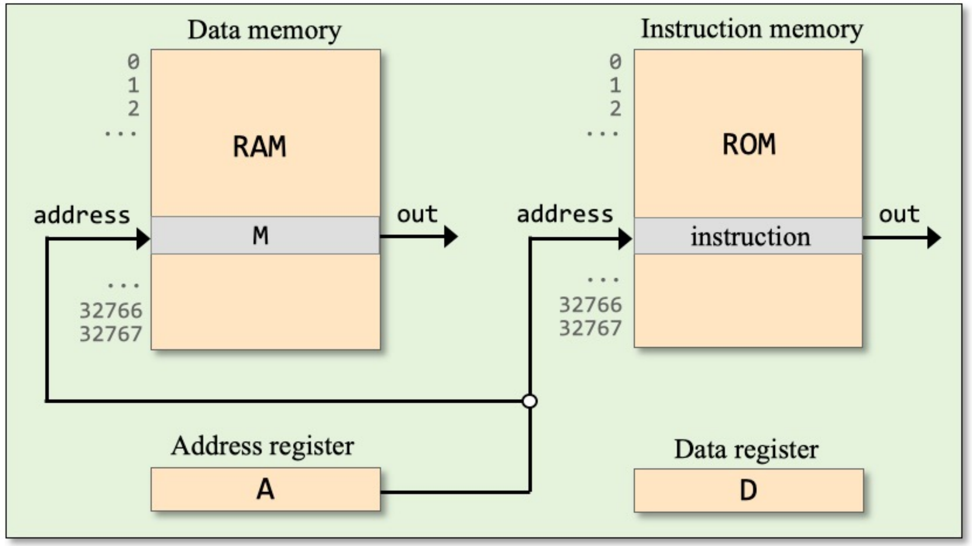
Typical instructions:

@*constant*	(A ← *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] ← 0
@100
M=0
```

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

```
// RAM[100] ← RAM[200]
@200
D=M
@100
M=D
```

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

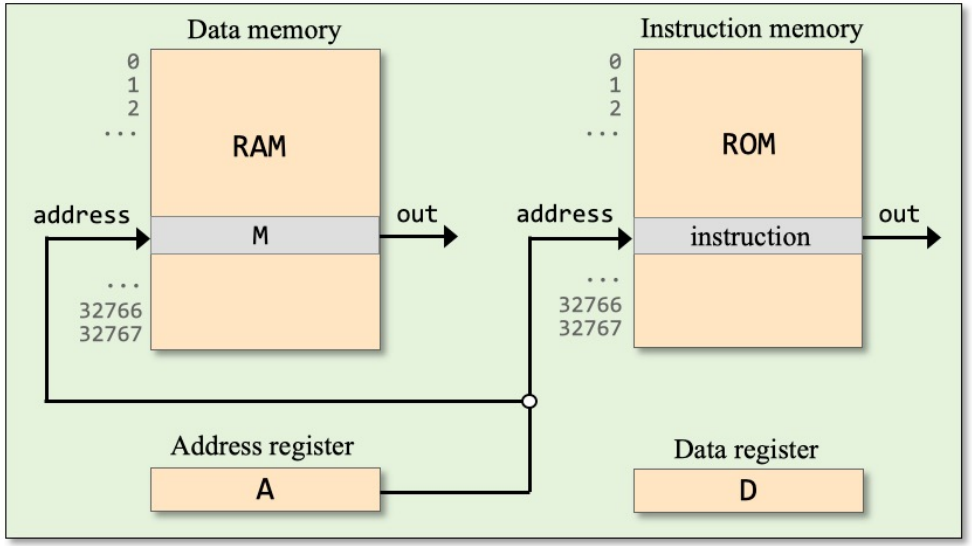
- A instruction: Selects a memory register
- C instruction: Operates on the selected register.

Hack instructions

Typical instructions:

@*constant*	(A ← *constant*)

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




```
// RAM[3] ← RAM[3] - 15
```

?

Use only the instructions shown in the current slide

Hack instructions

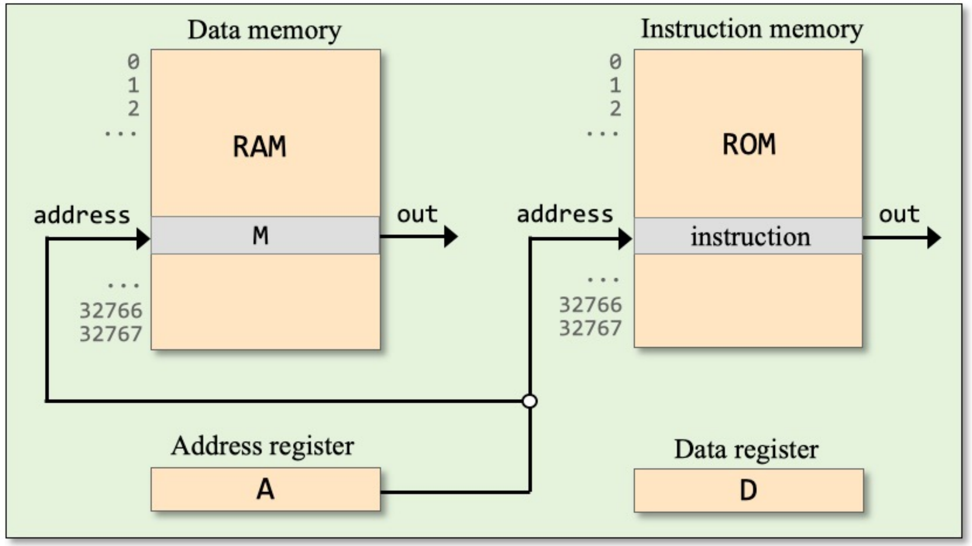
Typical instructions:

@ *constant*	(A←*constant*)

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

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

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




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

Use only the instructions shown in the current slide

```
// RAM[3] ← RAM[4] + 1
```

?

Hack instructions

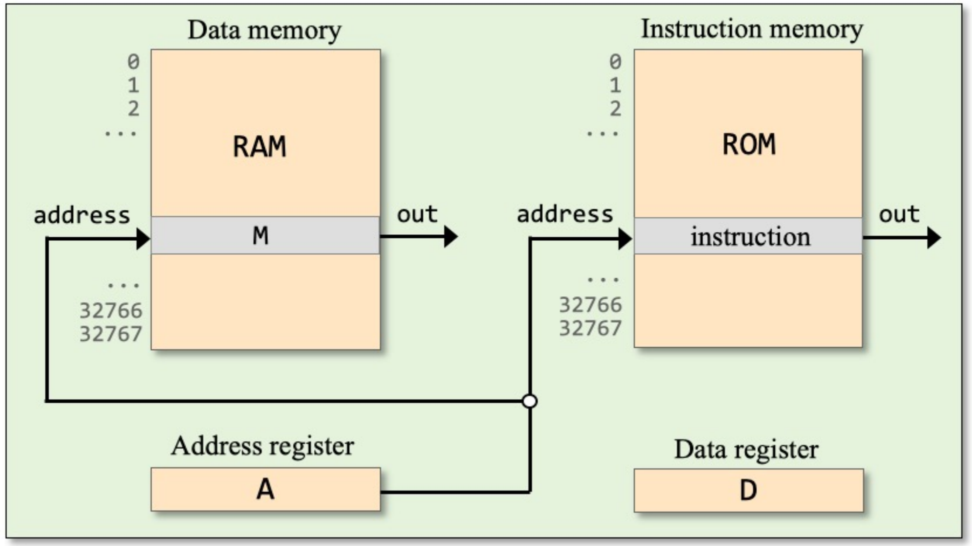
Typical instructions:

@*constant*	(A ← *constant*)

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

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

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




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

Use only the instructions shown in the current slide

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

Hack instructions

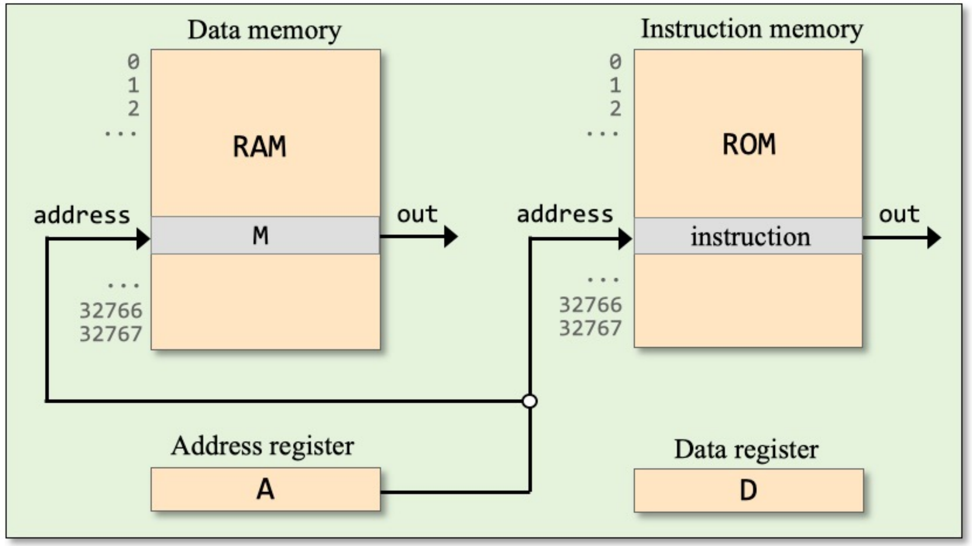
Typical instructions:

@*constant*	(A←*constant*)

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

```
A=M
D=M
M=D
...
```

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



Add.asm

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

?
```

Use only the instructions shown in the current slide

Hack instructions

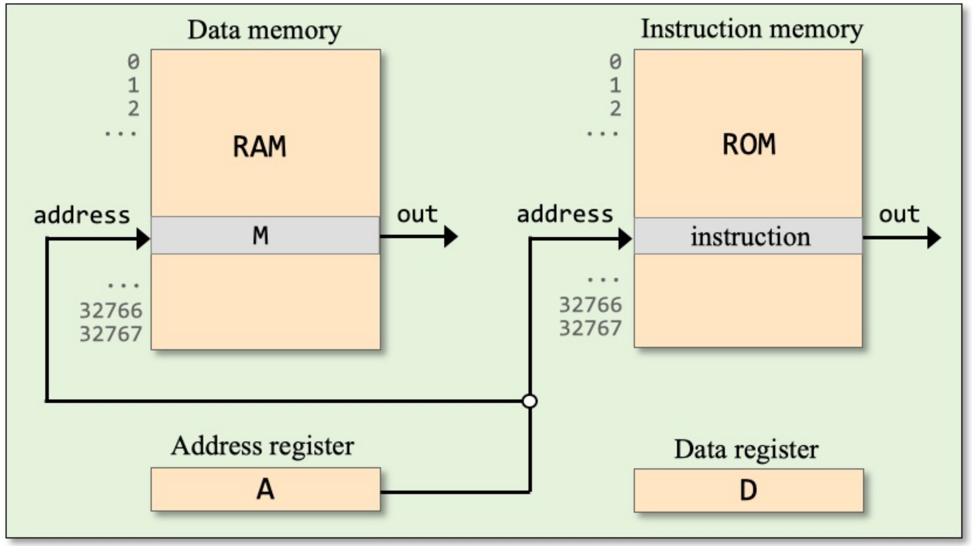
Typical instructions:

@ *constant*	(A←*constant*)

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

```
A=M
D=M
M=D
...
```

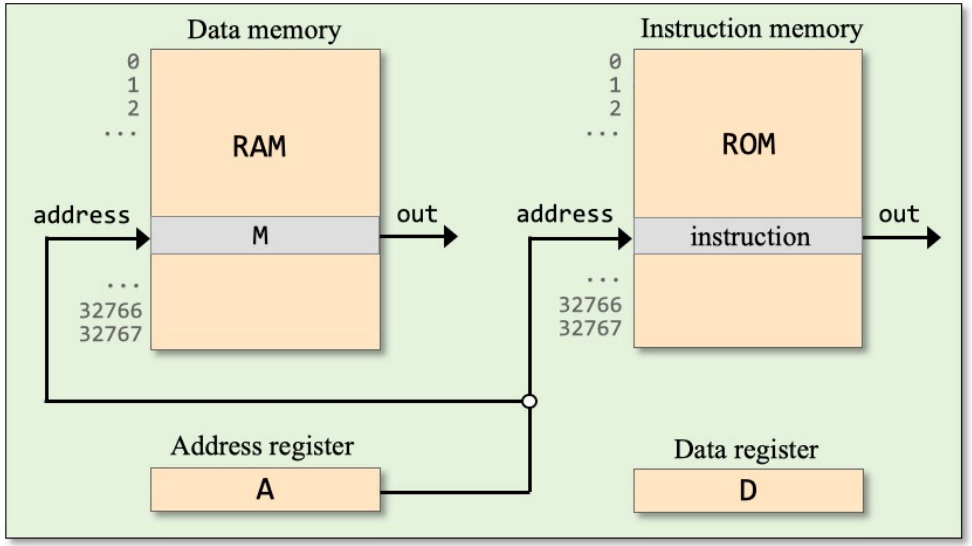
```
A=D-A
D=D+A
D=D+M
...
```



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←*constant*)

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

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
```

<u>How can we tell that a given program *actually works*?</u>

- ➡ Testing / simulating
- Formal verification

Chapter 4: Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

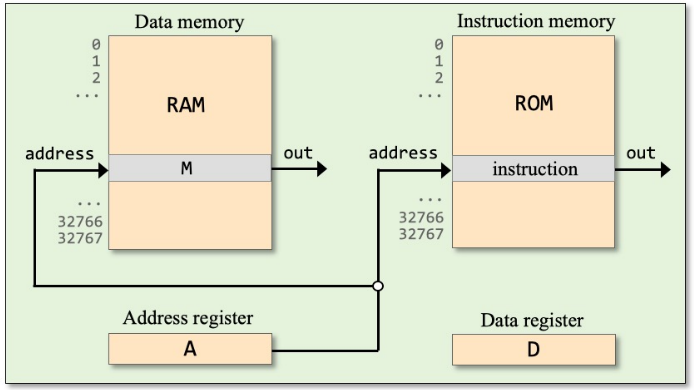
Low Level Programming

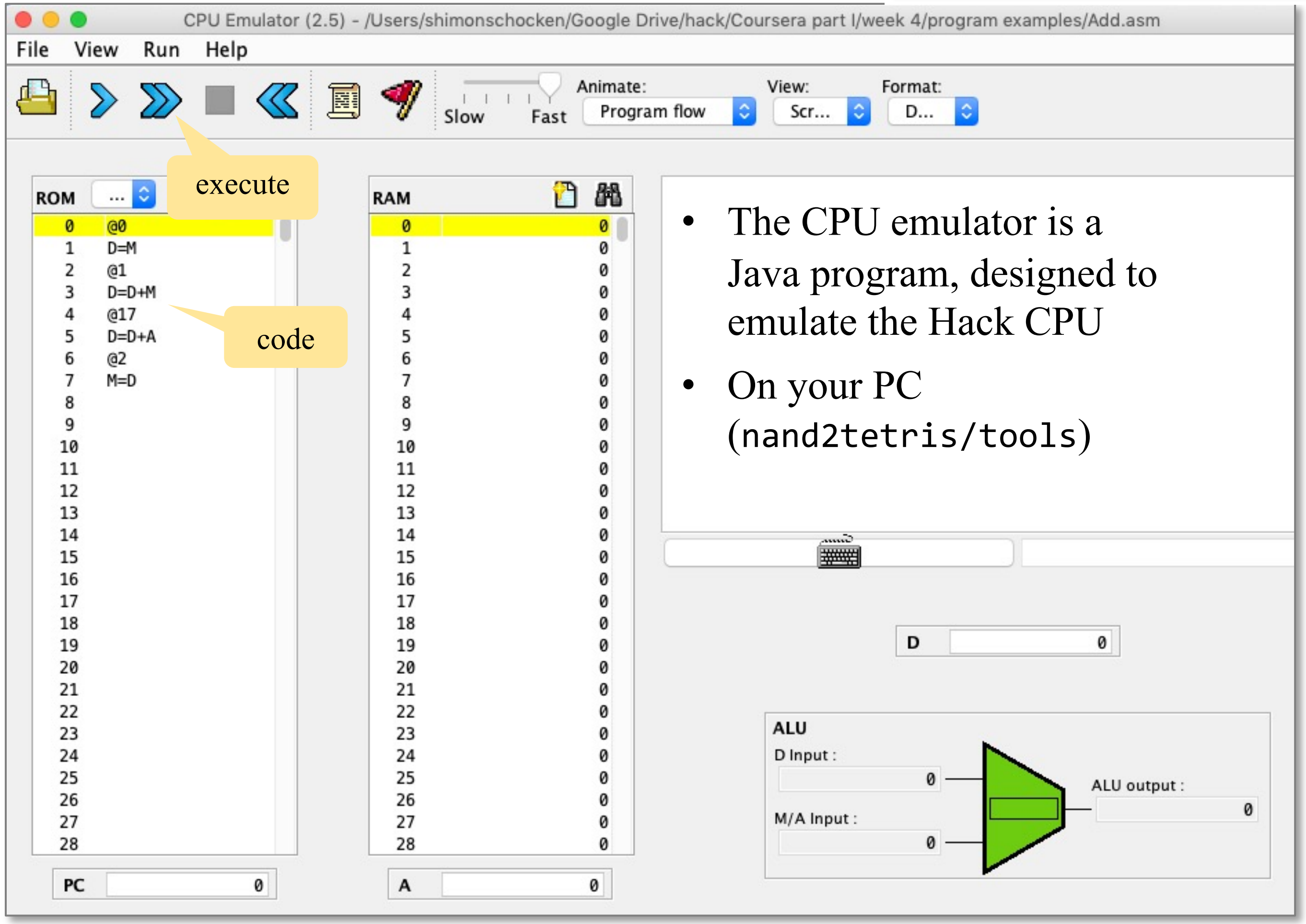
- Basic
- Iteration
- Pointers

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

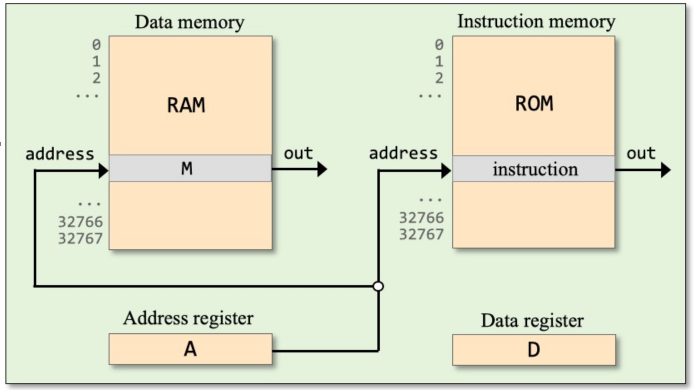
The CPU emulator





- The CPU emulator is a Java program, designed to emulate the Hack CPU
- On your PC (`nand2tetris/tools`)

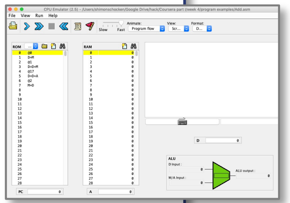
The CPU emulator



`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
0000000000010001
1001111100110011
0000000000000010
1110010010010011
```

Execute

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

The CPU emulator

Chapter 4: Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

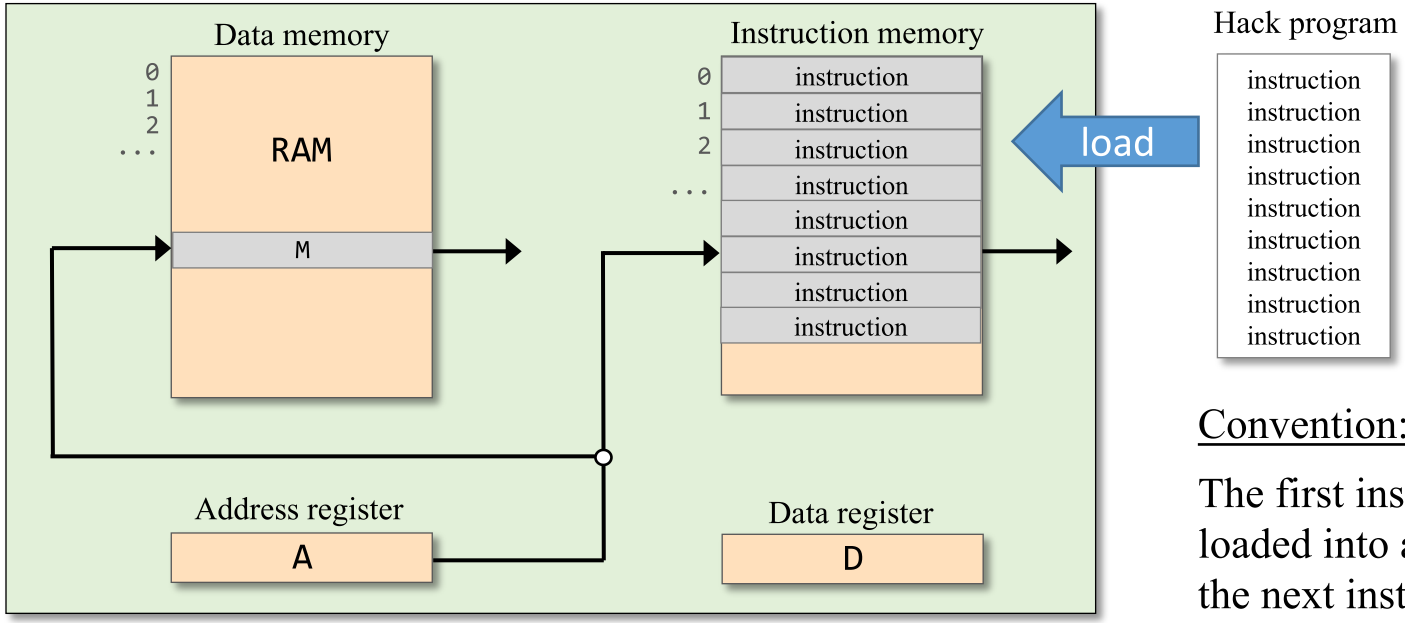
Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

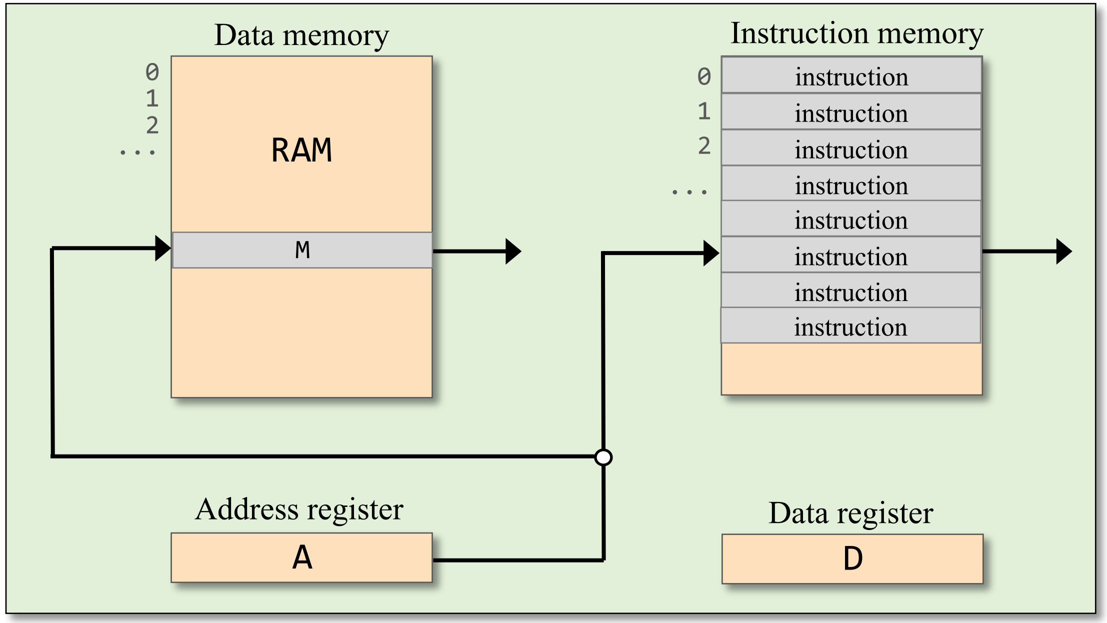
Loading a program



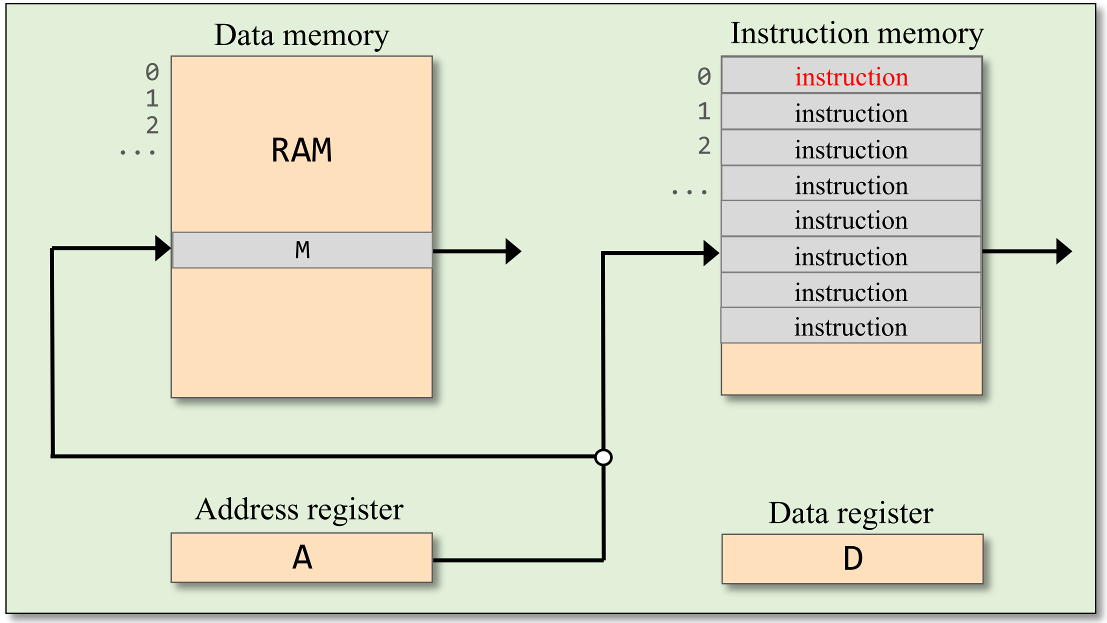
Convention:

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

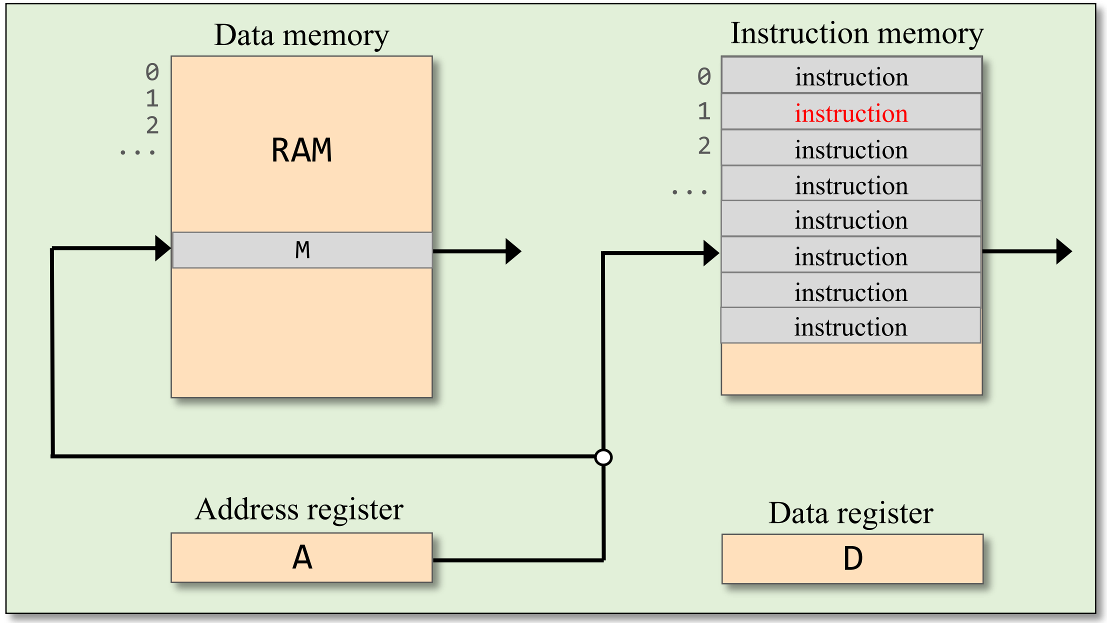
Executing a program



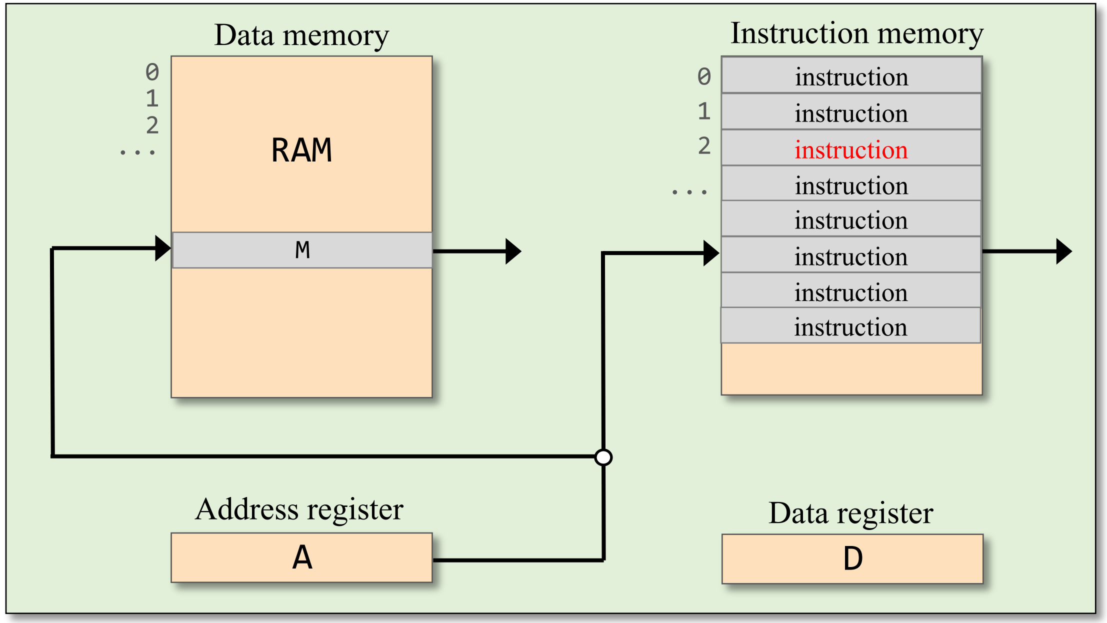
Executing a program



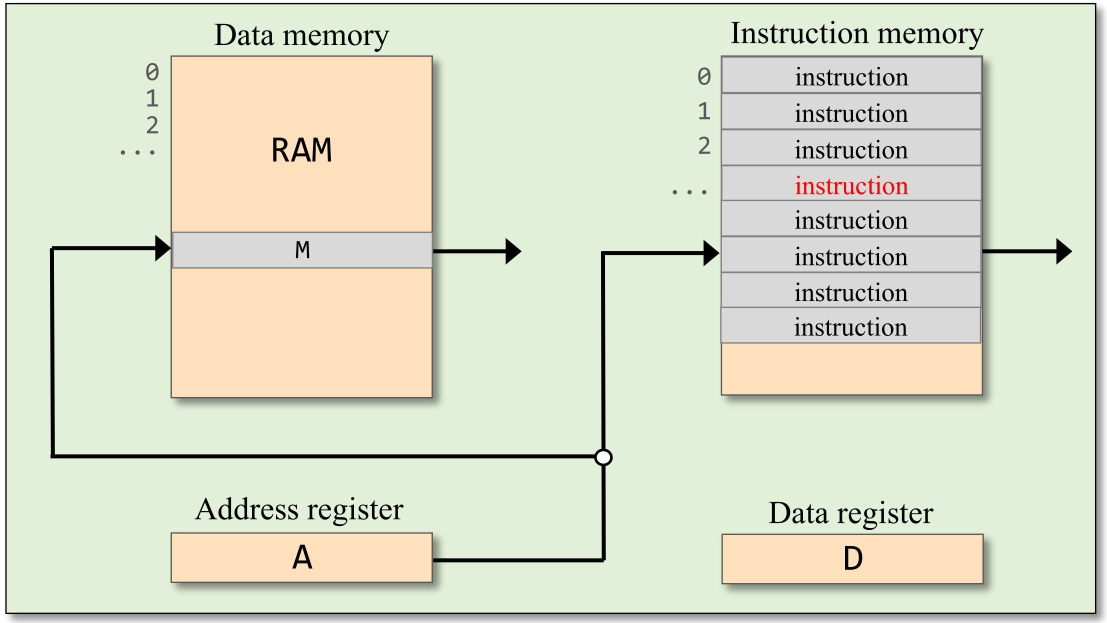
Executing a program



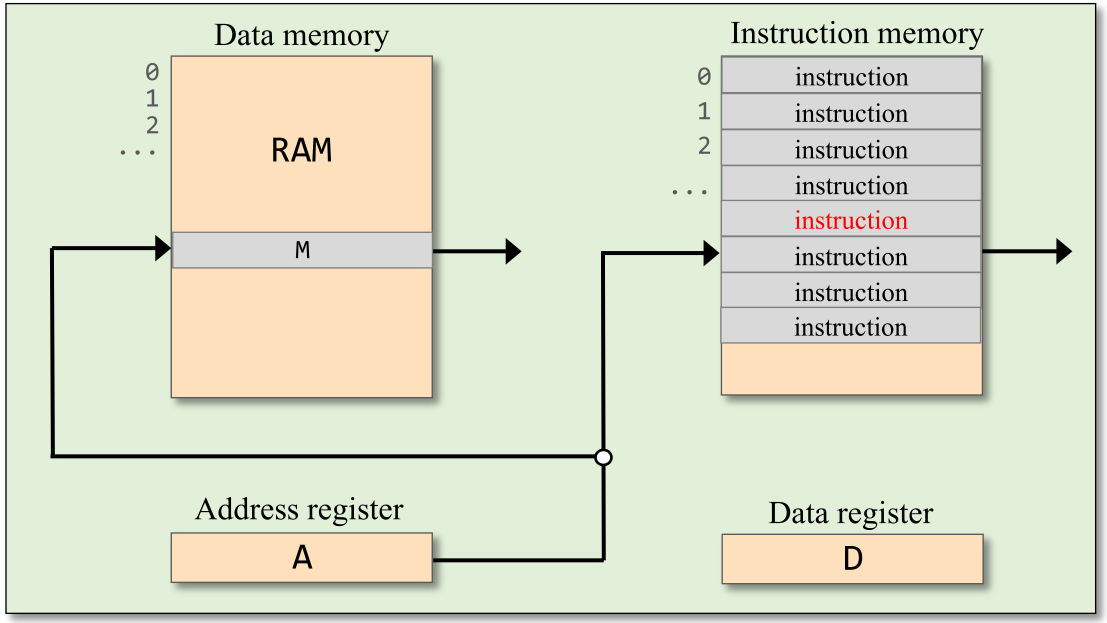
Executing a program



Executing a program

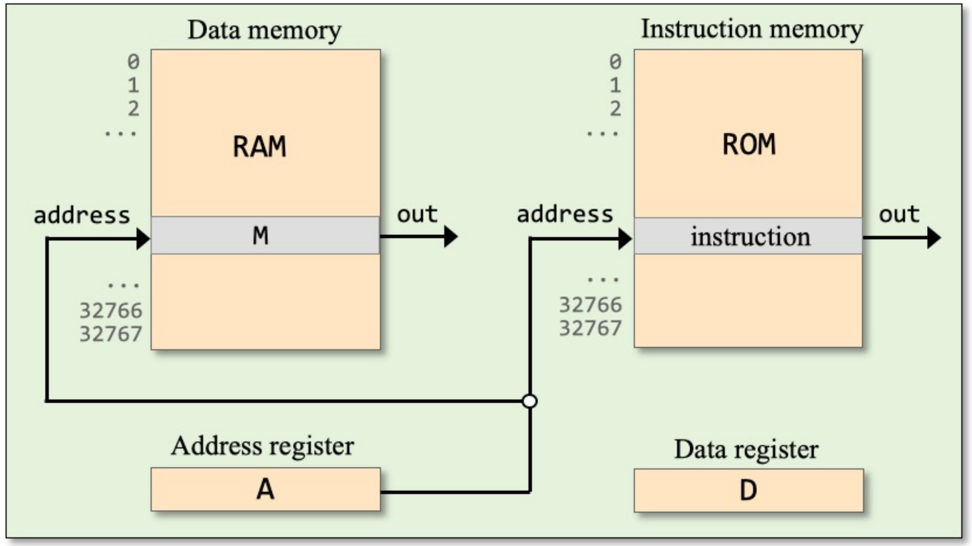


Executing a program



- The default: Execute the next instruction
- Suppose we wish to execute another instruction
- How to specify this *branching*?

Branching



<u>Unconditional branching</u>
example (pseudocode)

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

<u>Flow of control:</u>

0,1,2,3,4,

7,8,9,

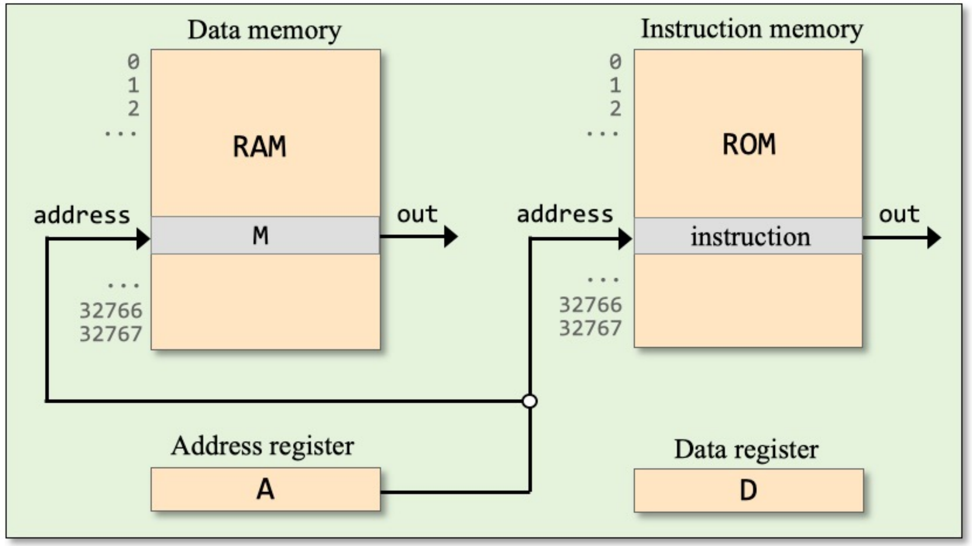
2,3,4,

7,8,9,

2,3,4,

...

Branching



<u>Conditional branching</u>
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
...  ...
```

<u>Flow of control:</u>

`0,1,2,3,4,`

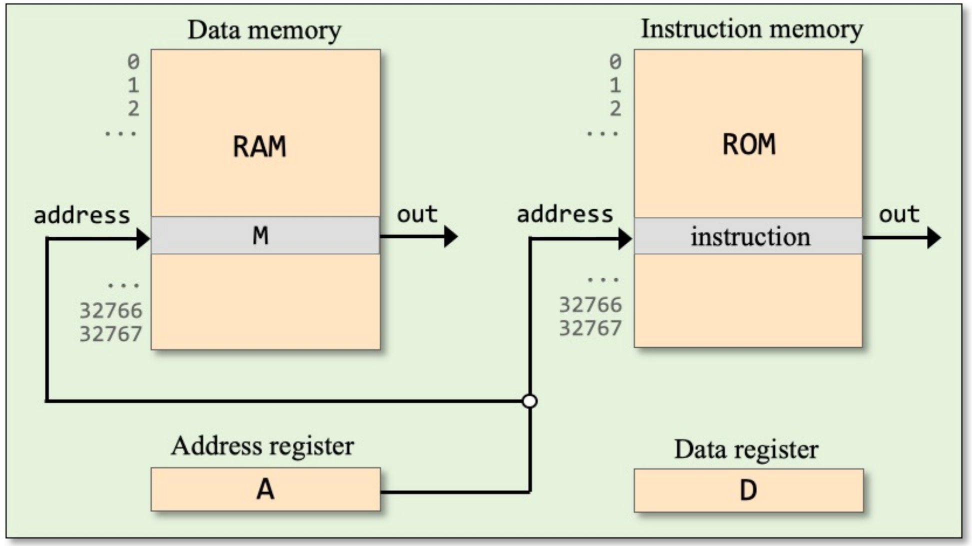
if *condition* is true

`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:

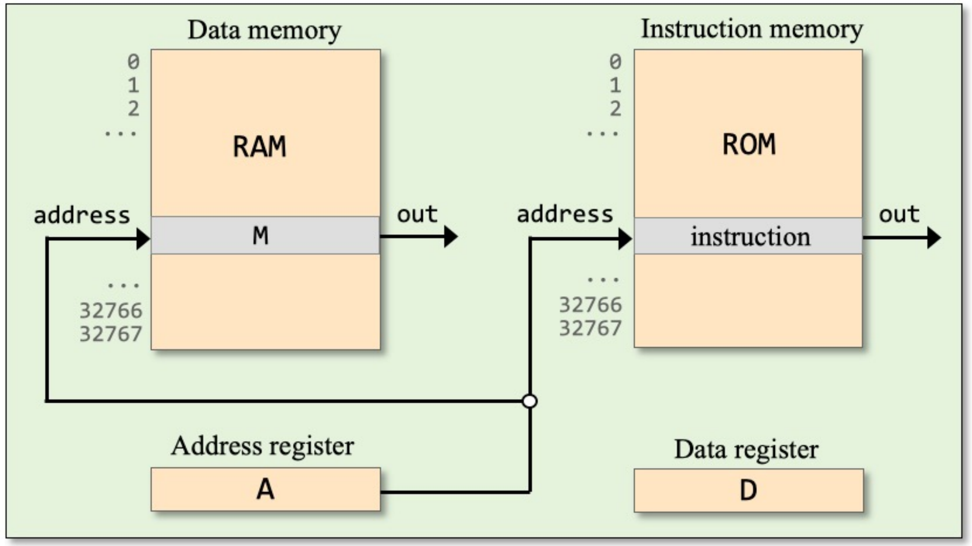
```
...
// goto 6
@6
0;JMP
...
```

Semantics of `0;JMP`

Jump to the instruction stored in the register selected by `A`

(the "`0;`" prefix will be explained later)

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
...
```

<u>Typical branching instructions:</u>

```
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]

Branching

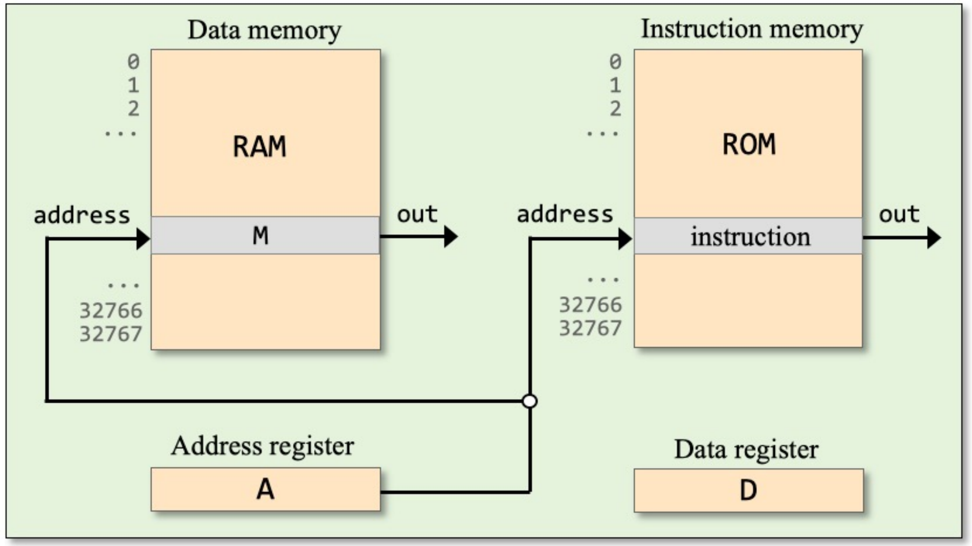
Typical instructions:

@*constant* (A←*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

?

Use only the instructions shown in the current slide

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
```

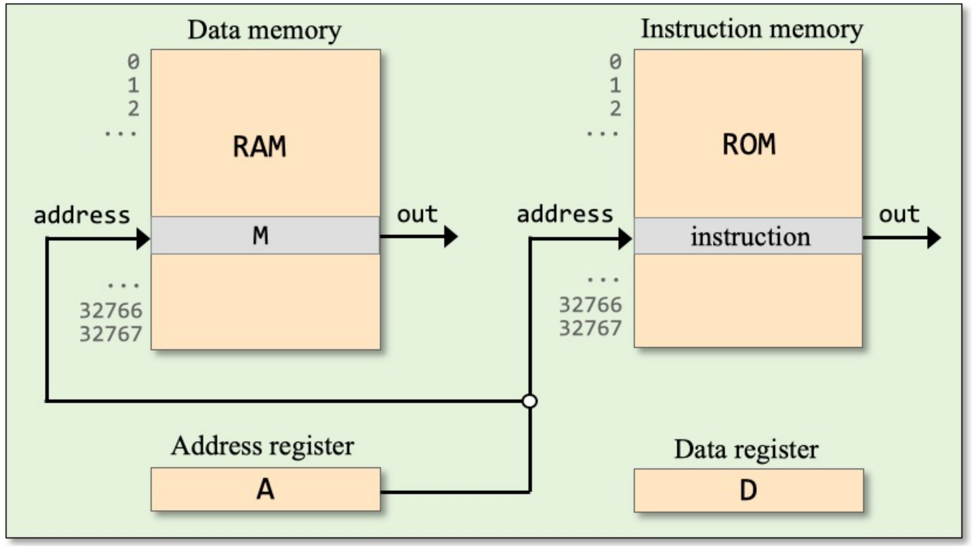
to the instruction stored in ROM[A]

Branching

Typical instructions:

`@constant`	(A←*constant*)

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




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

Use only the instructions shown in the current slide

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
```

$\Big\}$ to the instruction stored in `ROM[A]`

Branching

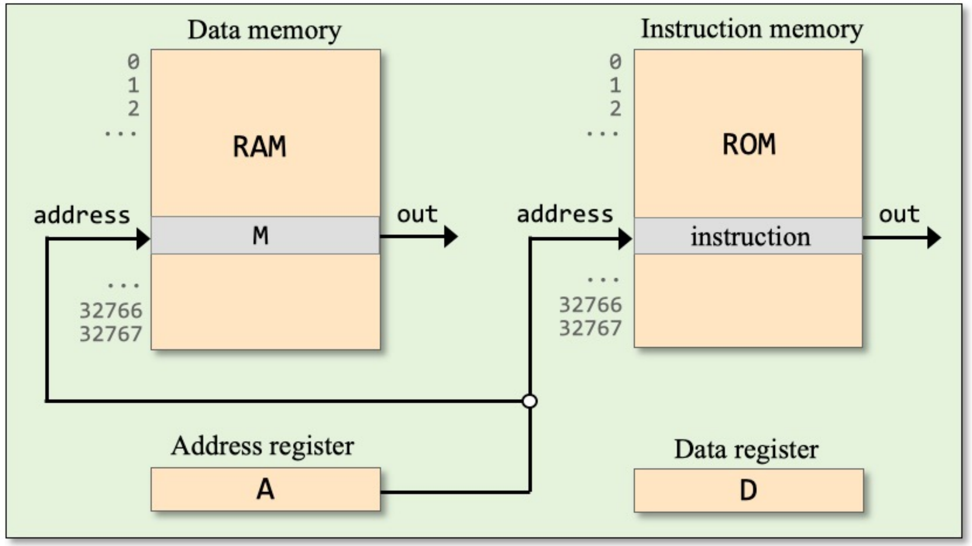
Typical instructions:

@*constant* (A←*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
?
```

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
```

$\left.\right\}$ to the instruction stored in ROM[A]

Use only the instructions shown in the current slide

Branching

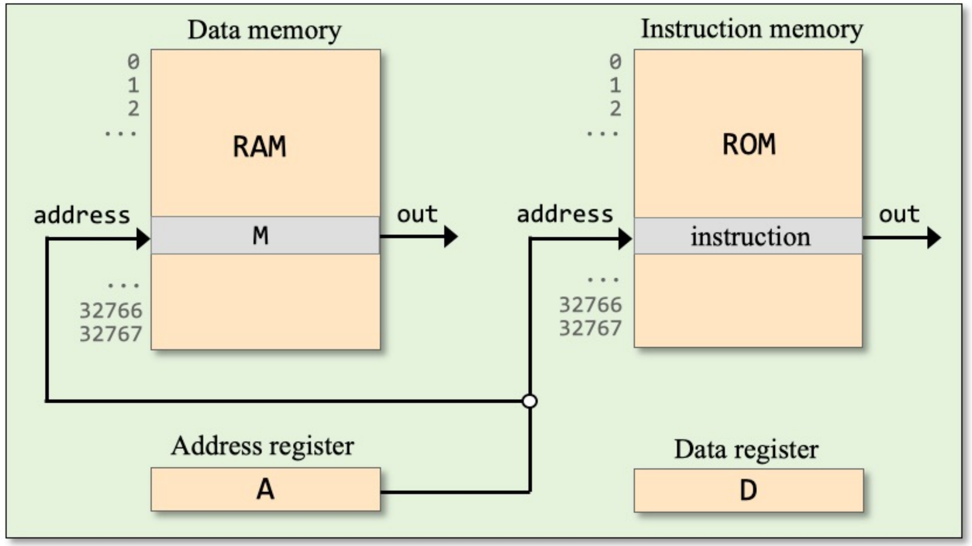
Typical instructions:

@*constant* (A←*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 ≥ 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]`

Use only the instructions shown in the current slide

Chapter 4: Machine Language

Overview

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

Low Level Programming

- Basic
- Iteration
- Pointers

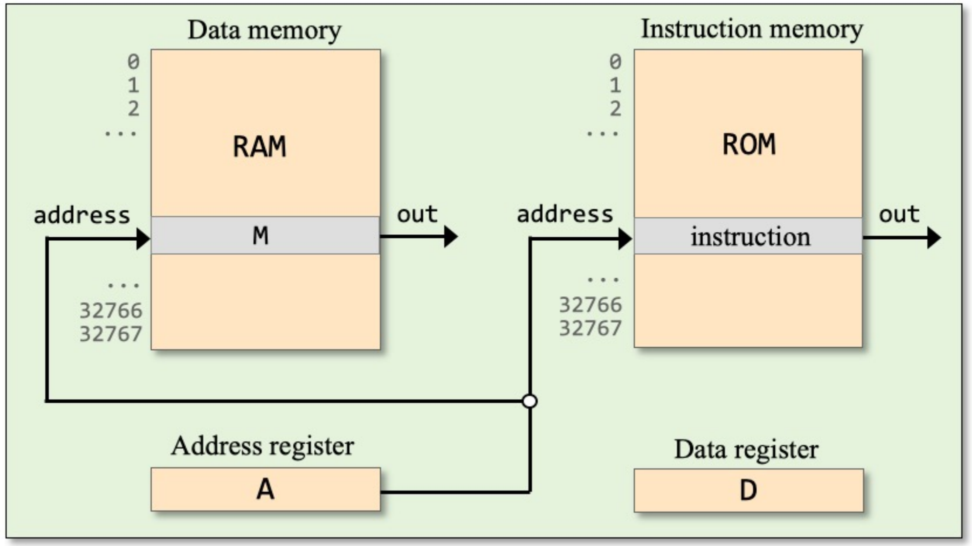
Symbolic programming

- ✓ Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Hack instructions



- A instruction
- C instruction

Syntax:

`@const`

where *const* is a constant

Example:

```
@19   // A ← 19
```

`@sym`

where *sym* is a symbol bound to a constant

```
@x
```

For example, if x is bound to 21, this instruction will set A to 21

This idiom can be used for realizing:

- Variables
- Labels

Variables

Pseudocode (example)

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

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 use symbolic variables, as needed
- We assume that there is an agent who knows how to bind these symbols to selected 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]`
- Should the code writer worry about what is the actual bindings? No
- The result: a low-level model for representing *variables*.

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
?
```

Use only the instructions shown in the current slide

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
```

Use only the instructions shown in the current slide

Variables

Pre-defined symbols

symbol	value
`R0`	`0`
`R1`	`1`
`R2`	`2`
...	...
`R15`	`15`

RAM

address		symbol
0		R0
1		R1
2		R2
...		...
15		R15
16		
17		
...		
32767		

- As if you 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 it easier to document, write, and debug Hack code.

Chapter 4: Machine Language

Overview

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

Symbolic programming

- ✓ Control
- ✓ Variables
- Labels

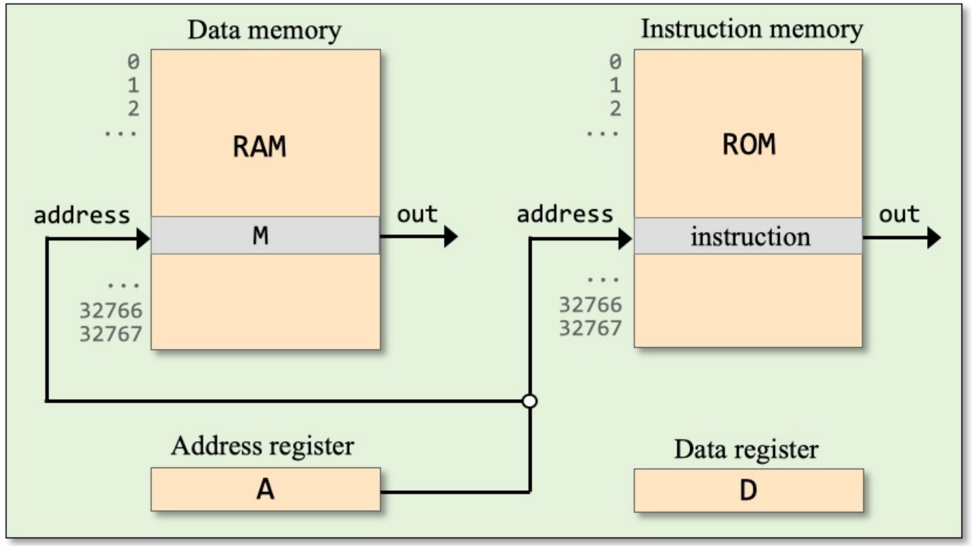
Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Labels



Typical branching instructions:

`D;JGT`	// if D > 0 jump	to ROM[A]
`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	

Examples (similar to what we did before):

```
// goto 48
?
```

```
// if (D > 0) goto 21
?
```

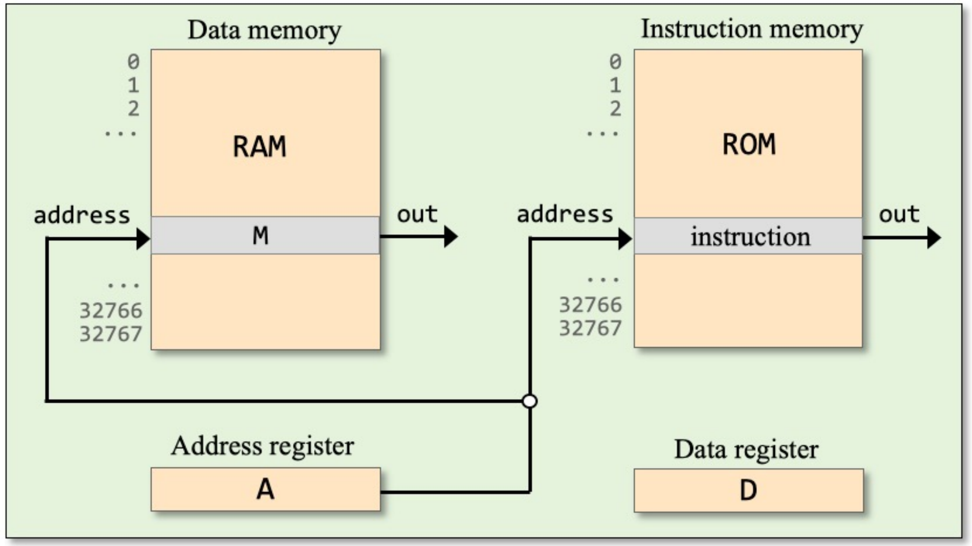
```
// if (RAM[100] < 0) goto 35
?
```

Labels

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 ROM[A]



Examples (similar to what we did before):

```
// goto 48
@48
0;JMP
```

```
// if (D > 0) goto 21
@21
D;JGT
```

```
// if (RAM[100] < 0) goto 35
@100
D=M
@35
D;JLT
```

Same examples, using *labels*

```
// goto LOOP
@LOOP
0;JMP
```

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

```
// if (x < 0) goto NEG
@x
D=M
@NEG
D;JLT
```

Hack convention:

Variables: lower-case symbols

Labels: upper-case symbols

Labels

Example (pseudocode)

```
  ...
LOOP:
   if (i = 0) goto CONT
   do this
   ...
   goto LOOP
CONT:
   do that
   ...
```

Hack assembly

```
   ...
(LOOP)
   // if (i = 0) goto CONT
   @i
   D=M
   @CONT
   D;JEQ
   do this
   ...
   // goto LOOP
   @LOOP
   0;JMP
(CONT)
   do that
   ...
```

Hack assembly syntax:

- A label *sym* is declared using (*sym*)
- Any label *sym* declared somewhere in the program can appear in a @*sym* instruction
- The assembler resolves the labels to actual addresses.

Programs that use symbolic labels and variables are...

- Easy to write / translate from pseudocode
- Readable
- Relocatable.

Chapter 4: Machine Language

Overview

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

Low Level Programming

- Basic
- Iteration
- Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Program example 1: Add

Task: R2 ← R0 + R1 + 17

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:
```

`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)
```

<u>Best practice</u>

When writing a (non-trivial) assembly program, always start with writing pseudocode.

Then translate the pseudo instructions into assembly, line by line.

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:
```

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)
```

Assembler / loader

(Note how the assembler mapped all the symbols on physical addresses)

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	
...	

Watch out: Security breach

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)
```

Assembler / loader

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: Security breach

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)
```

Program execution:

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
...	

Watch out: Security breach

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)
```

Program execution:

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
...	

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
```

Infinite loop

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
...	

<u>Best practice:</u>

Terminate every assembly program with an infinite loop.

Program example 3: `Max`

<u>Task</u>: Set `R0` to *max*(`R1`, `R2`)

Examples: *max*(5,3) = 5, *max*(2,7) = 7, *max*(4,4) = 4

Pseudocode

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

write

`Max2.asm`

```
// You do it
```

- Write the pseudocode
- Translate and write the assembly code in a text file named `Max2.asm`
- Load `Max2.asm` into the CPU emulator
- Put some values in `R1` and `R2`
- Run the program, one instruction at a time
- Make sure that the program puts the correct value in `R0`.

Chapter 4: Machine Language

Overview

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

Low Level Programming

- Basic
- Iteration
- Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Iterative processing

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

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
```

Chapter 4: Machine Language

Overview

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

Low Level Programming

- ✓ Basic
- ✓ Iteration
- ➡ Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Pointer-based processing

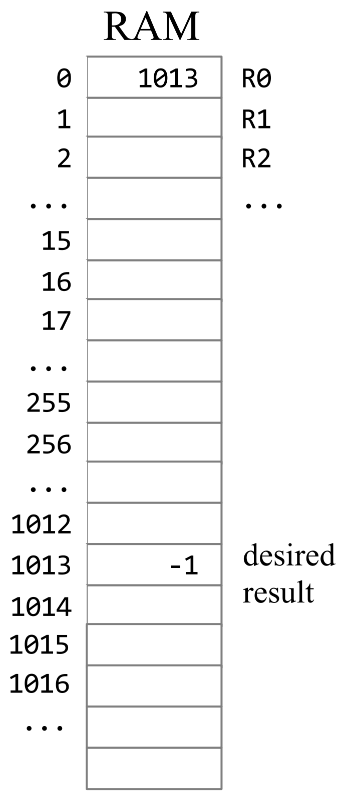
<u>Example 1</u>: Set the register at address *addr* to –1

Input: `R0`: Holds *addr*

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

<u>The key instruction:</u>

In the Hack machine language, pointer-based processing is realized by setting the address register to the address that we want to access, using the instruction `A=` ...



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
```

Example 2:

```
// Sets RAM[R0] to R1
// Usage: Put some non-negative value in R0,
//        and some value in R1.
@R1
D=M
@R0
A=M
M=D
```

RAM

0	1015	R0
1	-17	R1
2		R2
...		...
15		
16		
17		
...		
255		
256		
...		
1012		
1013		
1014		
1015	-17	desired result
1016		
...		

Pointer-based processing

<u>Example 3</u>: Get the value of the register at *addr*

Input: `R0`: Holds *addr*

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

?
```

RAM

0	1013	R0	
1	75	R1	desired result
2		R2	
...		...	
15			
16			
17			
...			
255			
256			
...			
1012	512		
1013	75		
1014	19		
1015	-17		
1016	256		
...			

Pointer-based processing

Example 3: Get the value of the register at *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	R0	
1	75	R1	desired result
2		R2	
...		...	
15			
16			
17			
...			
255			
256			
...			
1012	512		
1013	75		
1014	19		
1015	-17		
1016	256		
...			

Pointer-based processing

Example 4: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
...
// RAM[R0 + i] = -1
?
```

The key operation

RAM

0	300	R0	*base*
1	5	R1	*n*
2		R2	
...		...	
15		R15	
16	5	i	
17			
...			
255			
256			
...			
300	-1		
301	-1		
302	-1	desired output	
303	-1		
304	-1		
305			
...			

Pointer-based processing

<u>Example 4</u>: Set the first *n* entries of the memory block beginning in address *base* to `-1`

Inputs: `R0`: *base*
`R1`: *n*

Example: *base* = 300, *n* = 5

```
// Program: PointerDemo.asm
// Starting at the address stored in R0,
// sets the first R1 words to -1
...
// RAM[R0 + i] = -1
@R0
D=M
@i
A=D+M
M=-1
...
```

The key operation

RAM

0	300	R0	*base*
1	5	R1	*n*
2		R2	
...		...	
15		R15	
16	5	i	
17			
...			
255			
256			
...			
300	-1		desired output
301	-1		
302	-1		
303	-1		
304	-1		
305			
...			

Pointer-based processing

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
```

?

RAM

0	300	R0
1	5	R1
2		R2
...		...
15		R15
16		i
17		
...		
255		
256		
...		
300	-1	
301	-1	
302	-1	desired output
303	-1	
304	-1	
305		
...		

Pointer-based processing

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	R0
1	5	R1
2		R2
...		...
15		R15
16		i
17		
...		
255		
256		
...		
300	-1	
301	-1	
302	-1	desired output
303	-1	
304	-1	
305		
...		

Array processing

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	R0
1	5	R1
2		R2
...		...
15		R15
16		i
17		
...		
255		
256		
...		
300	-1	
301	-1	
302	-1	desired
303	-1	output
304	-1	
305		
...		

Array processing

Is done similarly, using pointer-based access to the memory block that represents the array.

Array processing

High-level code (e.g. Java)

```
...
// Declares variables
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 after executing this code:

RAM

0		R0
1		R1
2		R2
...		...
15		R15
16	5034	arr
17	357	sum
...	4	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	5	
5036		
...		

Array processing

High-level code (e.g. Java)

```
...
// Declares variables
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
}
...
```

Compiler

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	arr
17	357	sum
...	4	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	5	
5036		
...		

Every high-level array access operation involving *arr*[*expression*] can be compiled into Hack code that realizes the operation using the low-level memory access instruction A = *arr* + *expression*

Chapter 4: Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

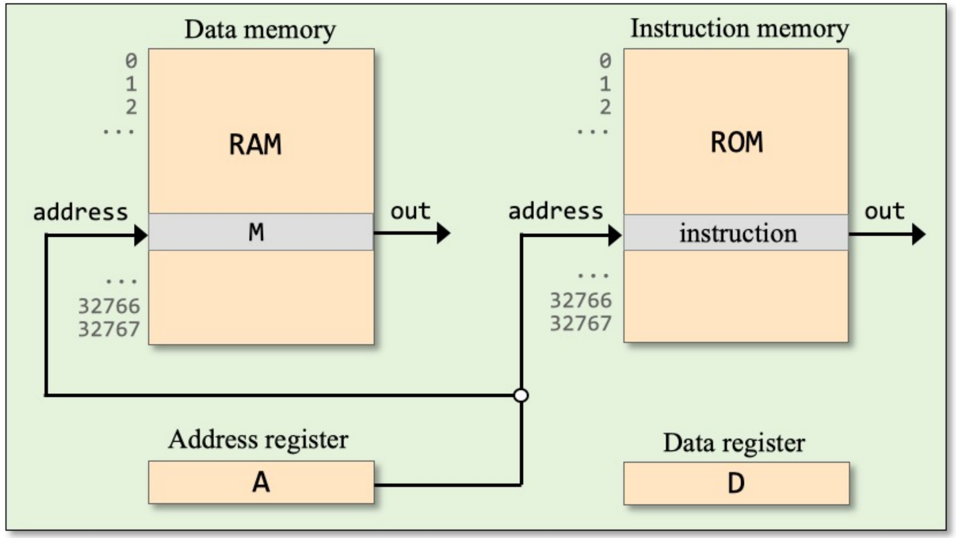
Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

The A instruction



Instruction set

- A instruction
- C instruction

Syntax: *@value*

Where *value* is either:

- a constant, or
- a symbol bound to a constant

Semantics:

- Sets the `A` register to the constant
- Side effects:

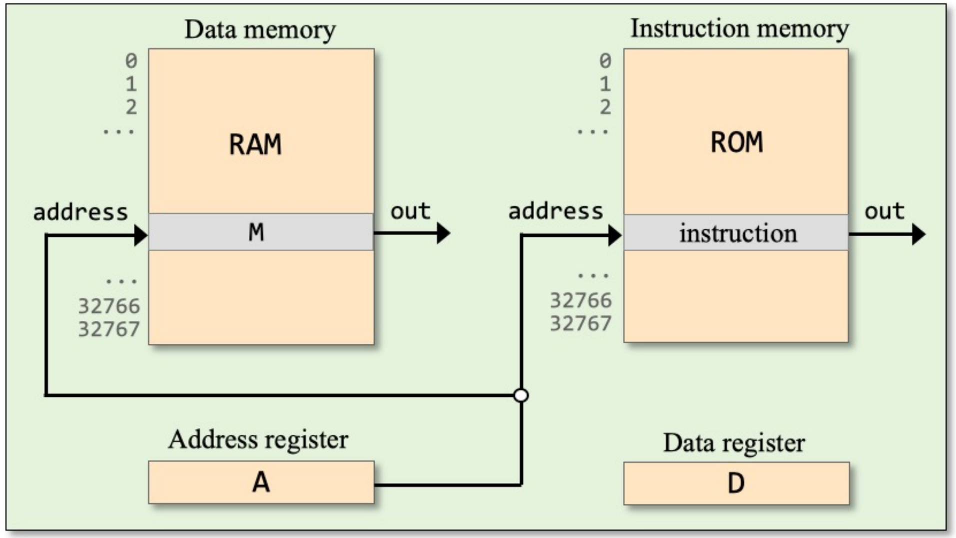
  `RAM[A]` becomes the selected `RAM` register

  `ROM[A]` becomes the selected `ROM` register

The C instruction

Instruction set

- A instruction
- C instruction



The C instruction

<u>Syntax:</u>

```
dest = comp ; jump
```

both *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
```

<u>Semantics:</u>

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

If (*comp jump* `0`), jumps to execute `ROM[A]`

The C instruction

Syntax: *dest* = *comp* ; *jump*   both *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`), jumps to execute `ROM[A]`

Examples:

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

The C instruction

<u>Syntax:</u> `dest = comp ; jump` both *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`

<u>Semantics:</u>

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

If (*comp jump* `0`), jumps 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* both *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`), jumps to execute `ROM[A]`

Examples:

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

Recap: A instructions and C instructions

<u>They normally come in pairs:</u>

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

To set up for a C instruction that mentions `M`,
Use an A instruction that selects the memory address on which you want to operate

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

To set up for a C instruction that specifies a jump,
Use an A instruction that selects the memory address to which you want to jump

Observation: C instructions that include *both* `M` *and* a jump directive *make no sense*

<u>Best practice rule</u>: Each C instruction should …

- Either have a reference to `M`
- Or have a jump directive

But not both.

Chapter 4: Machine Language

Overview

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

Low Level Programming

- Basic
- Iteration
- Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

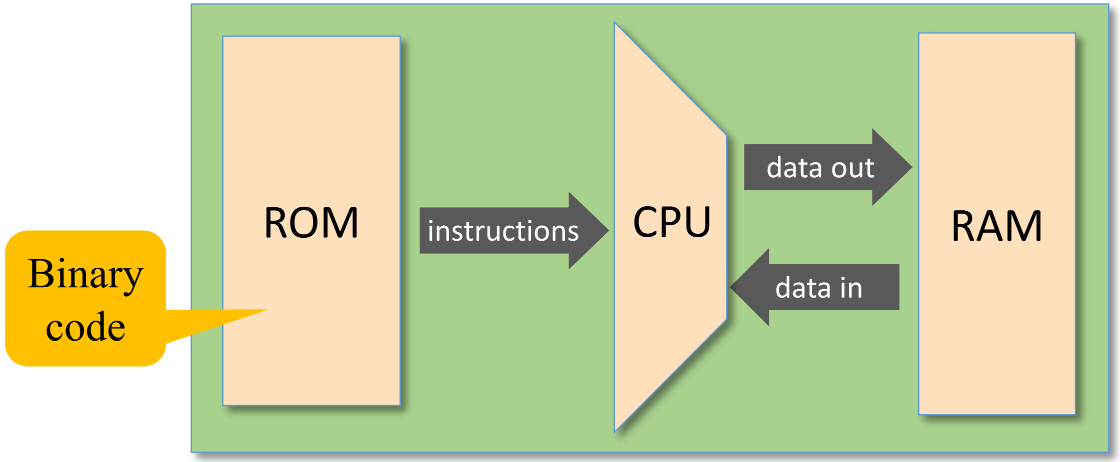
- Usage
- Specification
- Output
- Input
- Project 4

The Hack machine language

So far, we illustrated the Hack language using examples.

We now turn to give a complete language specification.

Hack machine language



The Hack language:

Symbolic: (example)

```
...
@17
D;JLE
...
```

translate

Binary:

```
...
0000000000010001
1110001100000110
...
```

load & execute

- The *binary version* of the language is not essential for low-level programming
- We present it anyway, for completeness
- The binary version will come to play when we'll build the computer architecture (chapter 5) and the assembler (chapter 6)

A instruction

Action: Sets the `A` register to a constant

Symbolic syntax:

`@`*value*

Binary syntax:

`0` *v v v v v v v v v v v v v v v*

Where *value* is either:

a non-negative decimal constant ≤ 65535 ($= 2^{16} - 1$)

or a symbol bound to a constant

Where:

`0` is the A instruction op-code

v v v ... *v* is the 15-bit binary representation of the constant

Example:

Symbolic:

```
@6
```

Binary:

```
0000000000000110
```

C instruction

Symbolic syntax: *dest* = *comp* ; *jump*

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

Binary syntax:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*j*	*j*	*j*

C instruction op-code

not used

comp bits

dest bits

jump bits

C instruction

Symbolic syntax: *dest* = *comp* ; *jump*

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

Binary syntax:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*j*	*j*	*j*

comp bits

comp (*a*==0)	*comp* (*a*==1)	*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a==0	*a*==1						

C instruction

Symbolic syntax: *dest* = *comp* ; *jump*

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*j*	*j*	*j*

Binary syntax: 1 1 1 *a c c c c c c d d d j j j*

$\underbrace{d\ d\ d}_{dest\text{ bits}}$

dest	*d*	*d*	*d*	effect: the value is stored in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
DM	0	1	1	D register and RAM[A]
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
ADM	1	1	1	A register, D register, and RAM[A]

C instruction

Symbolic syntax: *dest* = *comp* ; *jump*

comp is mandatory.
If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted

Binary syntax:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*d*	*d*	*d*	*j*	*j*	*j*

jump bits

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* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	Unconditional jump

The Hack language specification

<u>A instruction</u>

Symbolic: @*xxx* (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

Binary: 0 *vvvvvvvvvvvvvvv* (*vv ... v* = 15-bit value of *xxx*)

<u>C instruction</u>

Symbolic: *dest* = *comp* ; *jump* (*comp* is mandatory. If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted)

Binary: 111*acccccc ddd jjj*

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 (*a* == 0)	*comp* (*a* == 1)	*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	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* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	unconditional jump

Chapter 4: Machine Language

✓ Overview

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

✓ Low Level Programming

- Basic
- Iteration
- Pointers

✓ Symbolic programming

- Control
- Variables
- Labels

The Hack Language

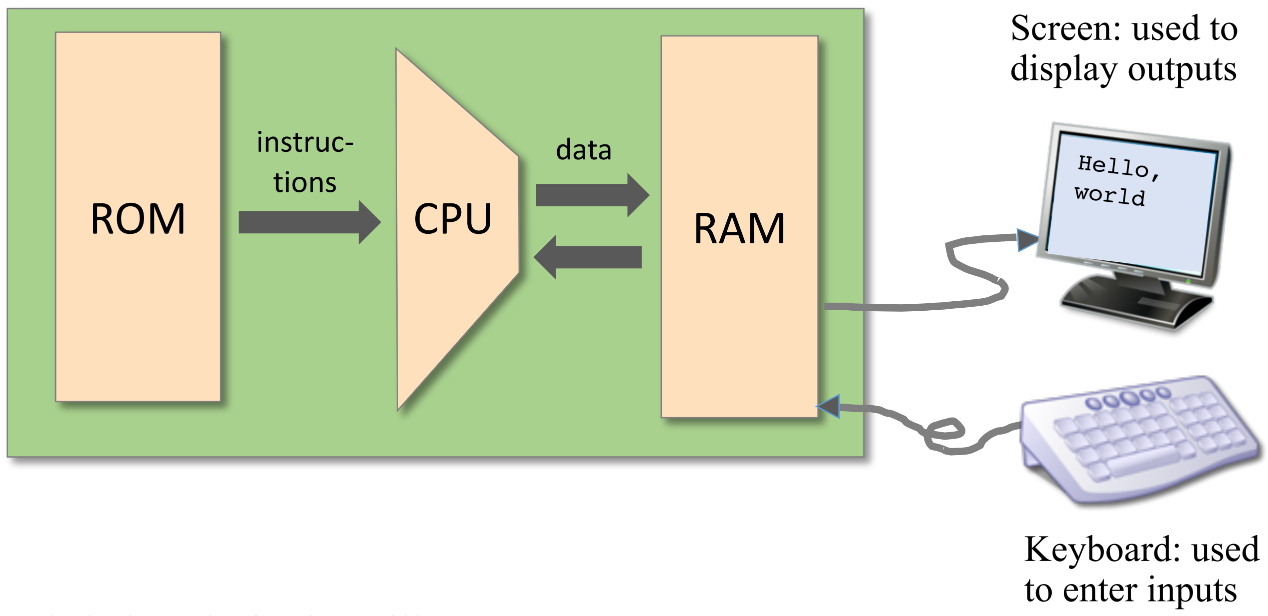
✓ Usage

✓ Specification

➡ Output

- Input
- Project 4

Input / output



High-level I/O handling:

Software libraries for inputting / outputting text, graphics, audio, video, …

Low-level I/O handling:

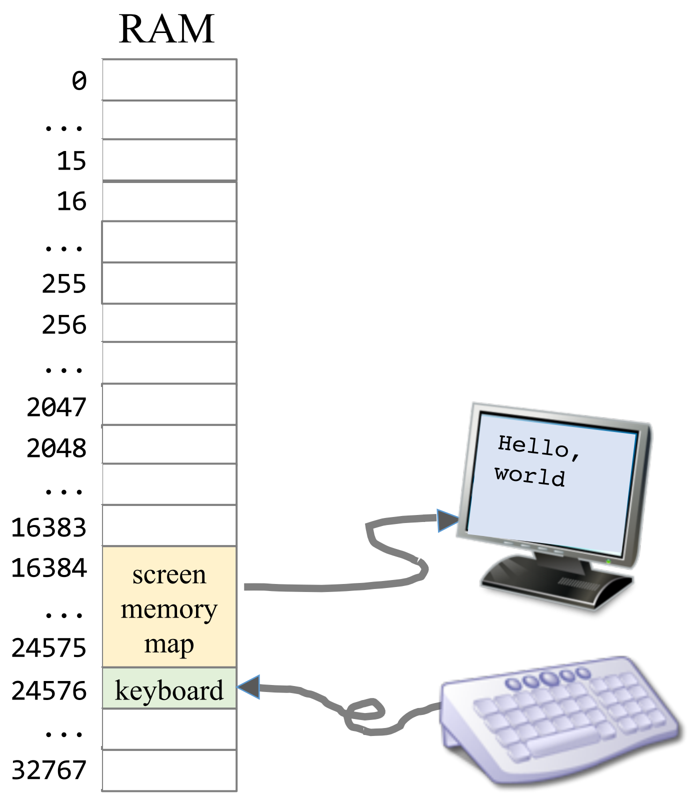
Manipulating bits in memory resident *bitmaps*.

Bitmaps

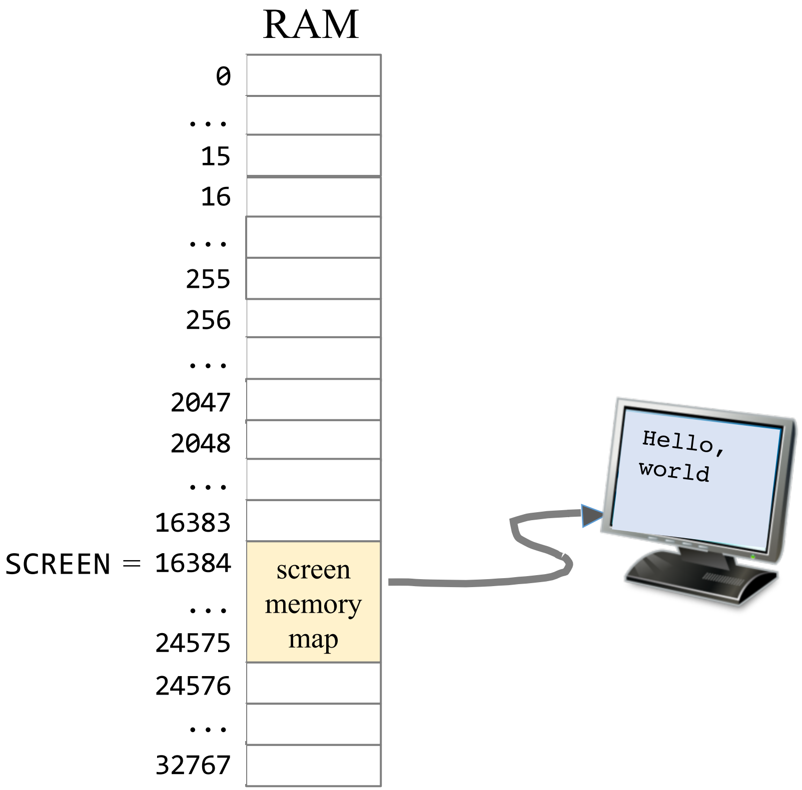
RAM

0
...
15
16
...
255
256
...
2047
2048
...
16383
16384
...
24575
24576
...
32767

Bitmaps


RAM
0
...
15
16
...
255
256
...
2047
2048
...
16383
16384
...
24575
screen memory map
24576
keyboard
...
32767
Hello,
world

Bitmaps



<u>Screen memory map:</u>

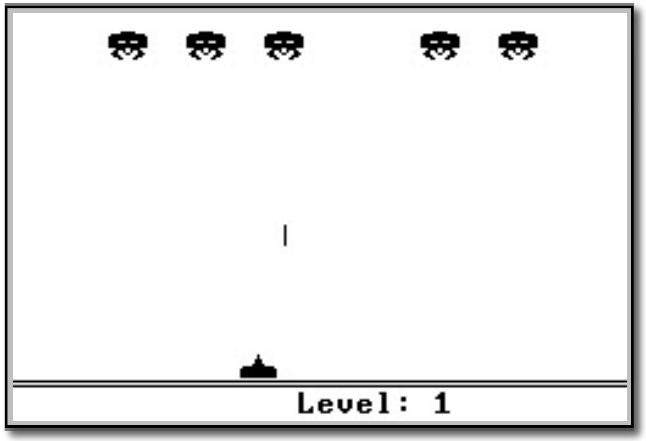
An 8K memory block, dedicated to representing a black-and-white display unit

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

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

Bitmaps

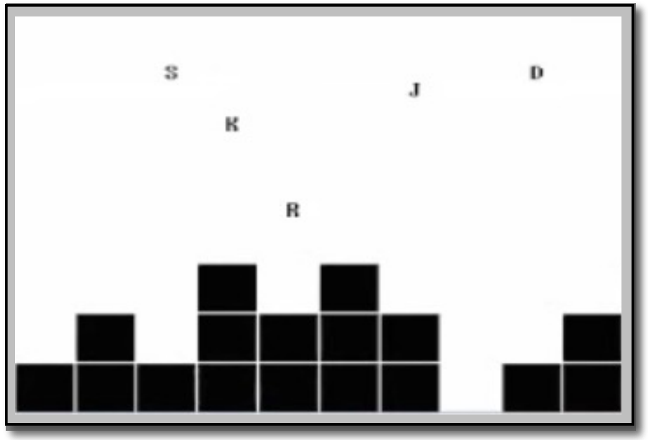
Physical screen



Screen shots of computer games
developed on the Hack computer

Bitmaps

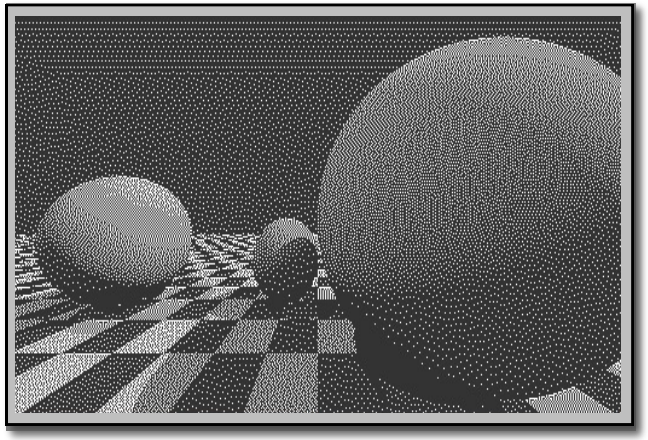
Physical screen



Screen shots of computer games developed on the Hack computer

Bitmaps

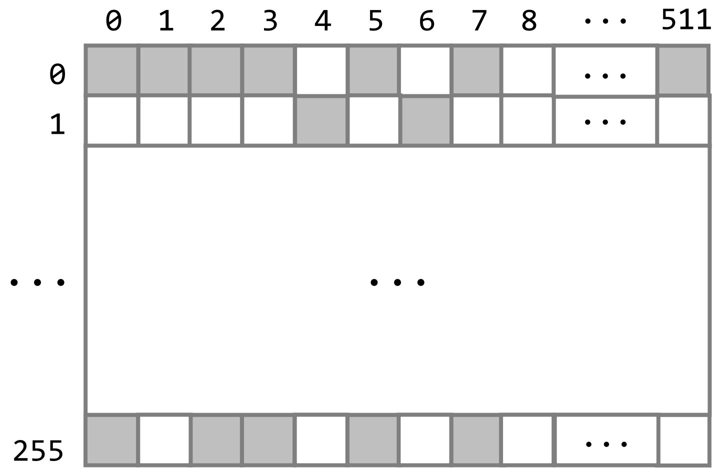
Physical screen



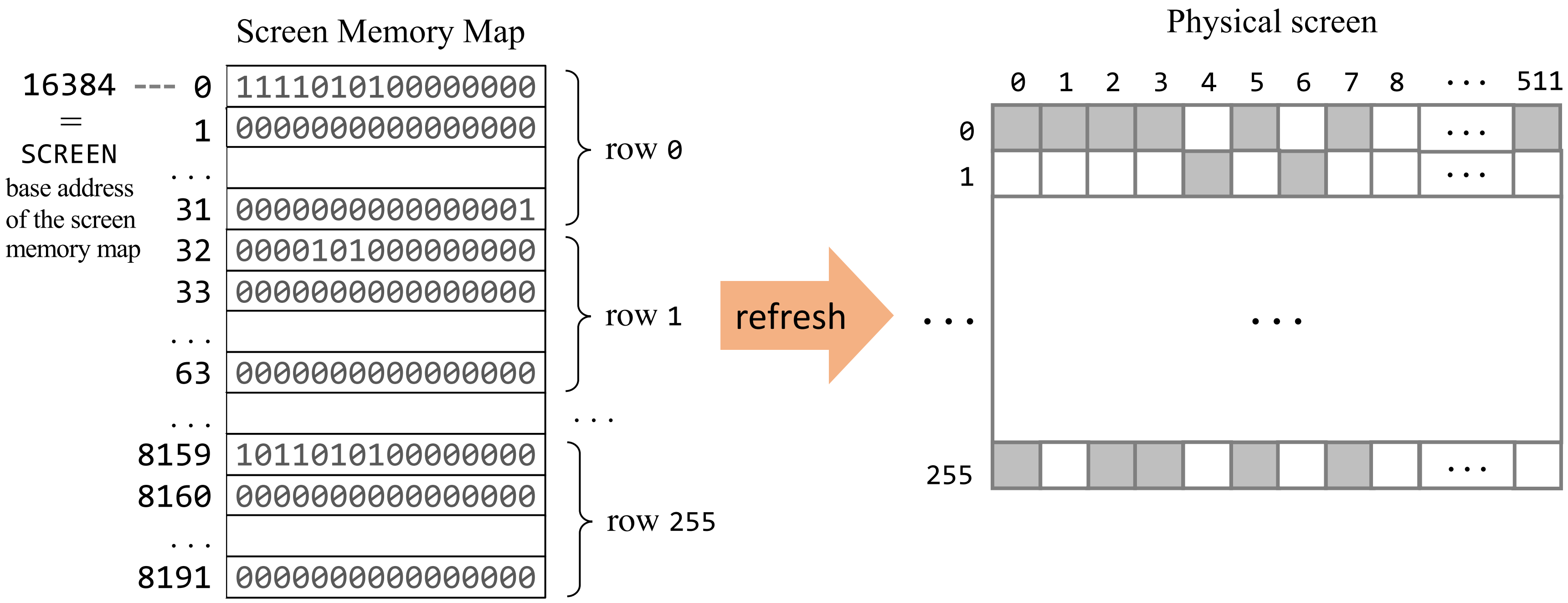
Screen shots of computer games developed on the Hack computer

Bitmaps

Physical screen



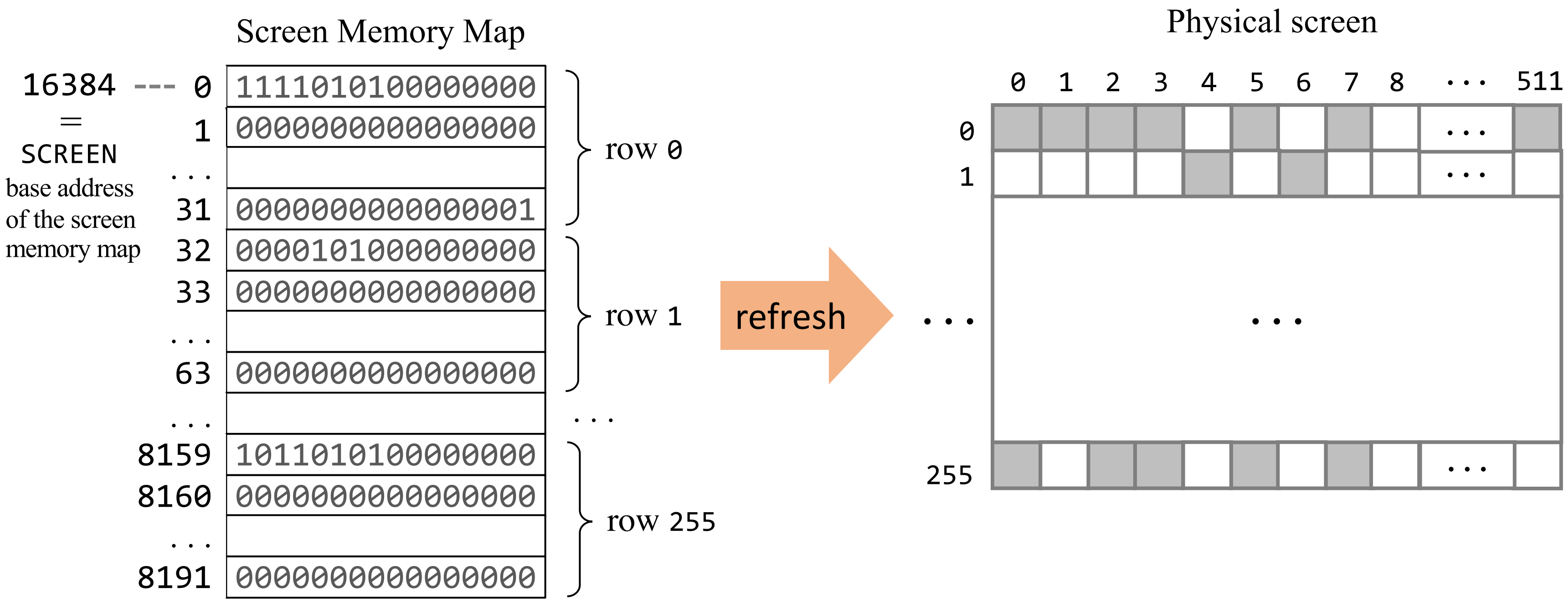
Bitmaps



<u>Mapping:</u>

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

Bitmaps



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

(1) *addr* ← SCREEN + 32 * *row* + *col* / 16

(2) *word* ← RAM[*addr*]

(2) Set the (*col* % 16) *th* bit of *word* to 0 or 1

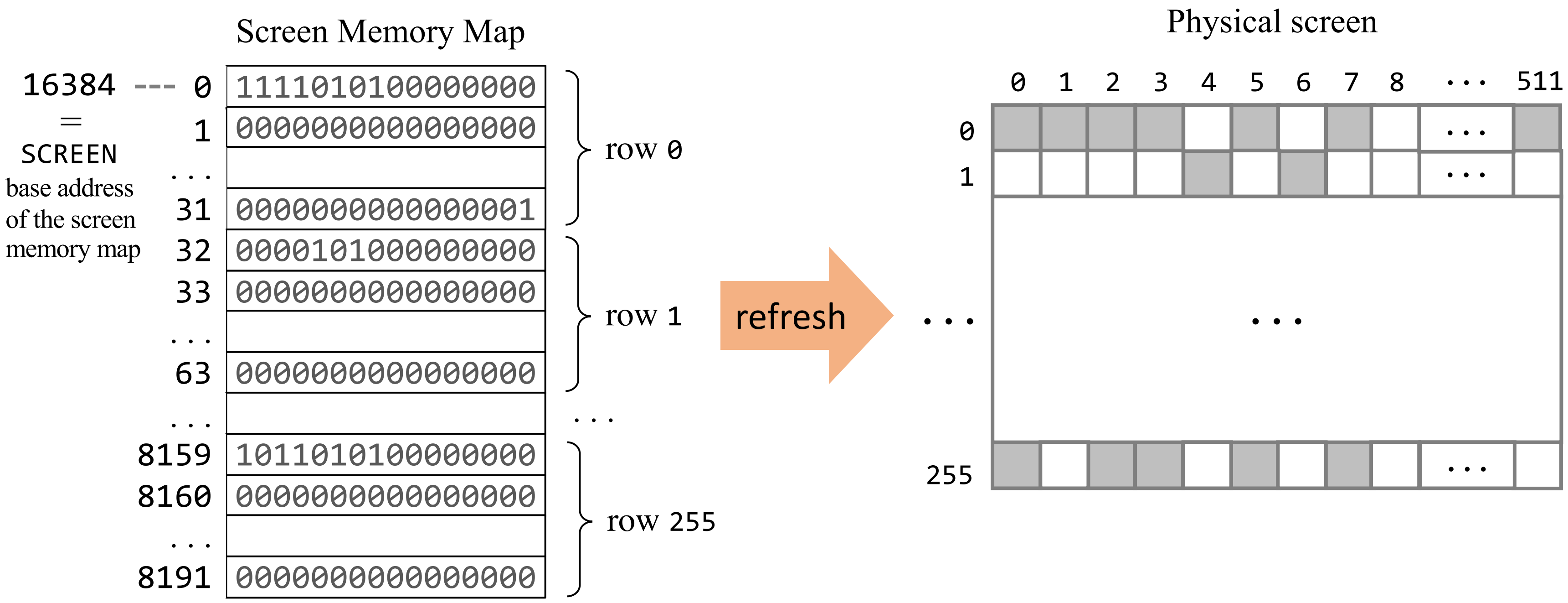
(3) RAM[*addr*] ← *word*

Not to worry:

Nice workarounds coming up

(Bitmap Editor)

Bitmaps



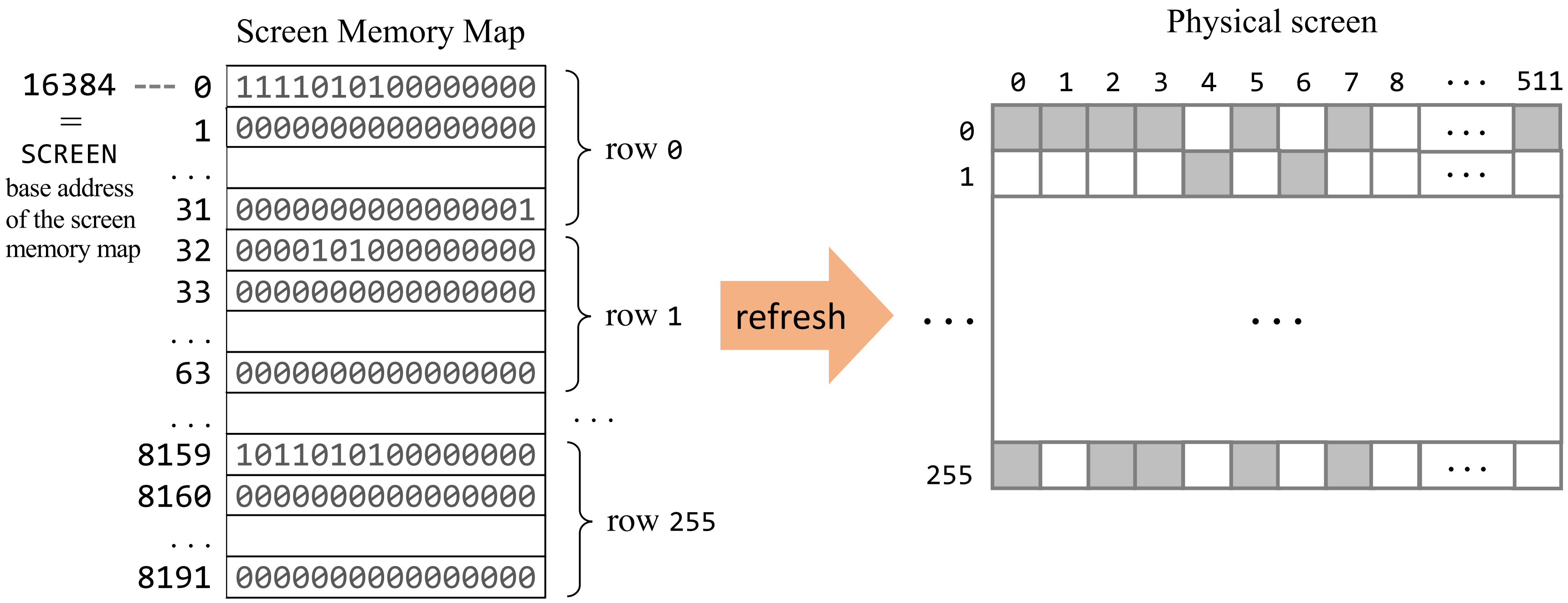
Examples of simple patterns that can be drawn relatively easily:

```
// 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
```

?

Bitmaps



Examples of simple patterns that can be drawn relatively easily:

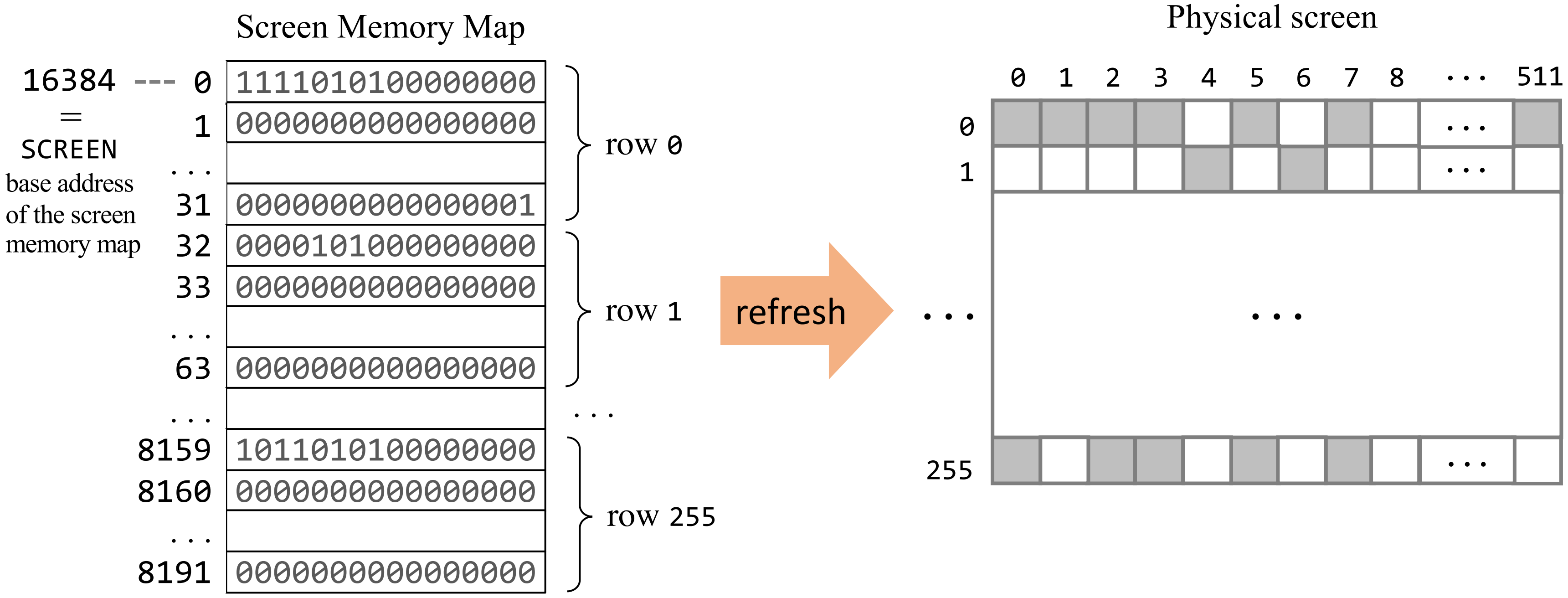
```
// Sets the first (left) 16 pixels
// of the top row to black
@SCREEN
M=-1        // -1 = 1111111111111111
```

```
// 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



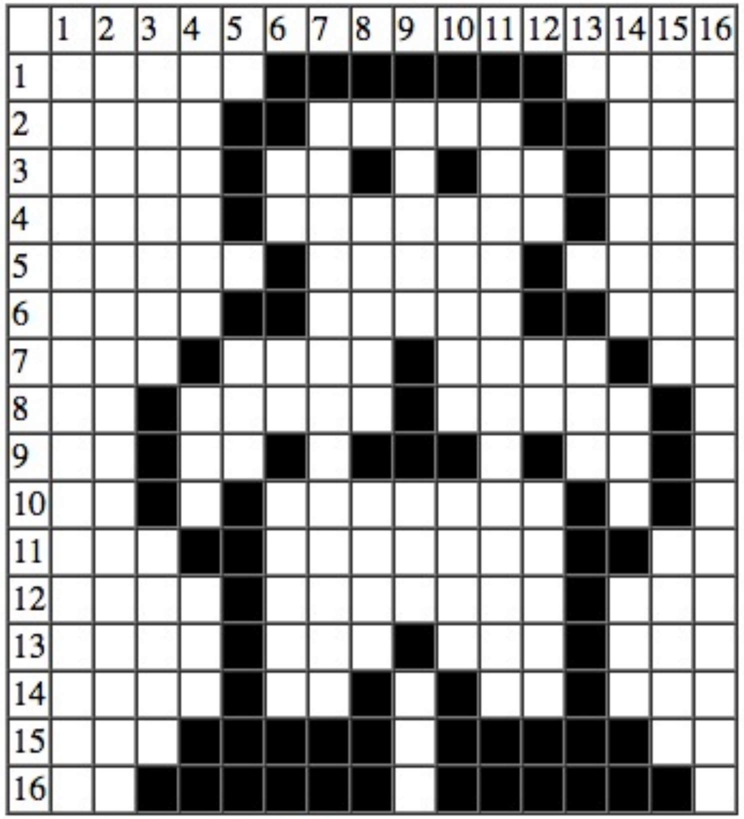
Examples of simple patterns that can be drawn relatively easily:

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

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

Simple graphics program:

Bitmap Editor




```
0000111111100000 = 4064
0001100000110000 = 6192
0001001010010000 = 4752
. . .
```

<u>Bitmap editor</u>: A productivity tool for developers.

The developer draws a pixled image on a 2D grid, and the program generates code that draws the image in the RAM.

The generated code can be copy-pasted into the developer's assembly code.

```
. . .
0111111011111100 = 32508
```

<u>The Nand to Tetris Bitmap Editor</u> is available 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.

Chapter 4: Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

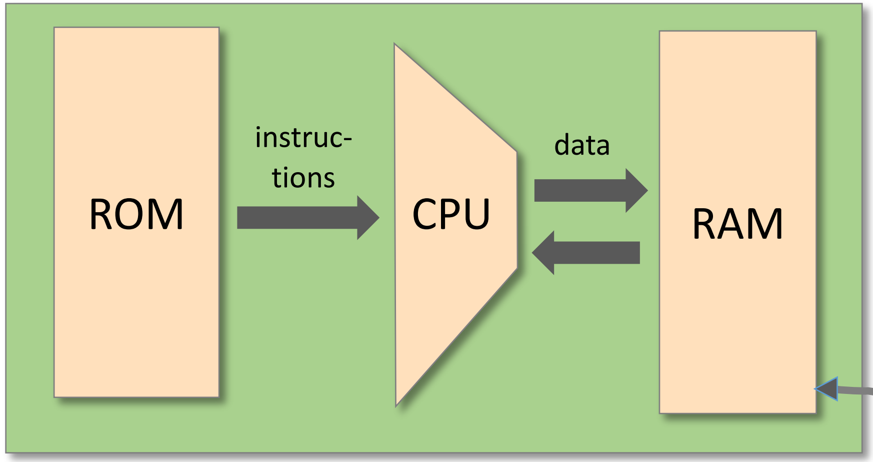
Low Level Programming

- Basic
- Iteration
- Pointers

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

Input



Keyboard: used
to enter inputs

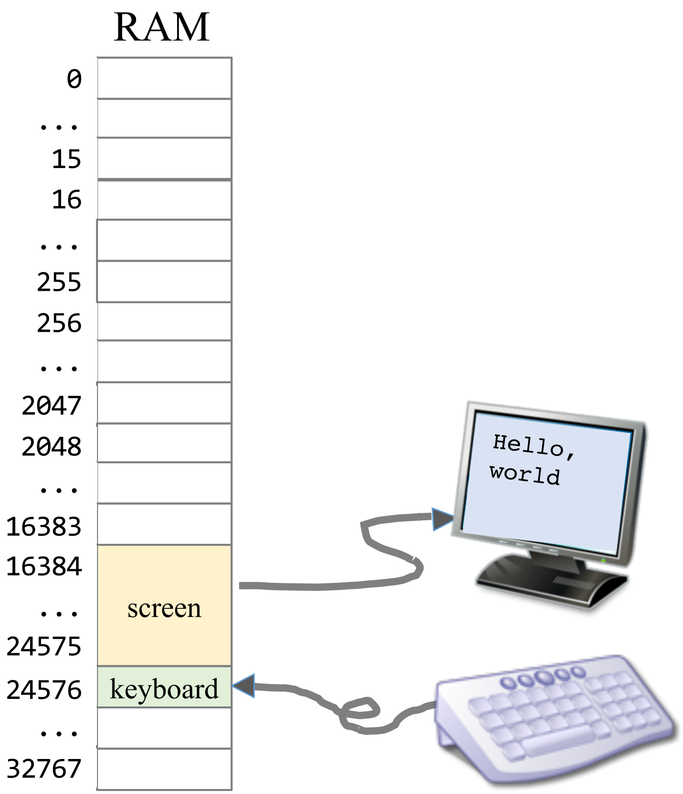
<u>High-level input handling</u>

`readInt`, `readString`, ...

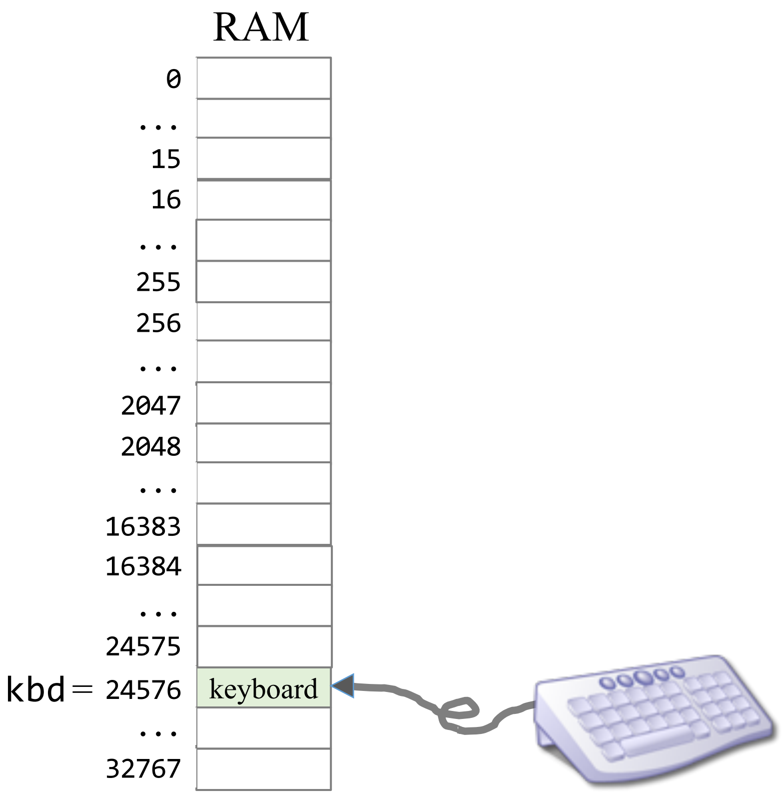
<u>Low-level input handling</u>

Read bits.

Hack RAM


RAM
0
...
15
16
...
255
256
...
2047
2048
...
16383
16384
...
24575
24576
...
32767
screen
keyboard
Hello,
world

Hack RAM



Keyboard memory map

A single 16-bit register, dedicated to representing the keyboard

Base address: KBD = 24576 (predefined symbol)

Reading inputs is affected by probing this register.

The Hack character set

key	code
(space)	32
!	33
“	34
#	35
$	36
%	37
&	38
‘	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

key	code
0	48
1	49
…	…
9	57

key	code
:	58
;	59
<	60
=	61
>	62
?	63
@	64

key	code
A	65
B	66
C	…
…	…
Z	90

key	code
[	91
/	92
]	93
^	94
_	95
`	96

key	code
a	97
b	98
c	99
…	…
z	122

key	code
{	123
|	124
}	125
~	126

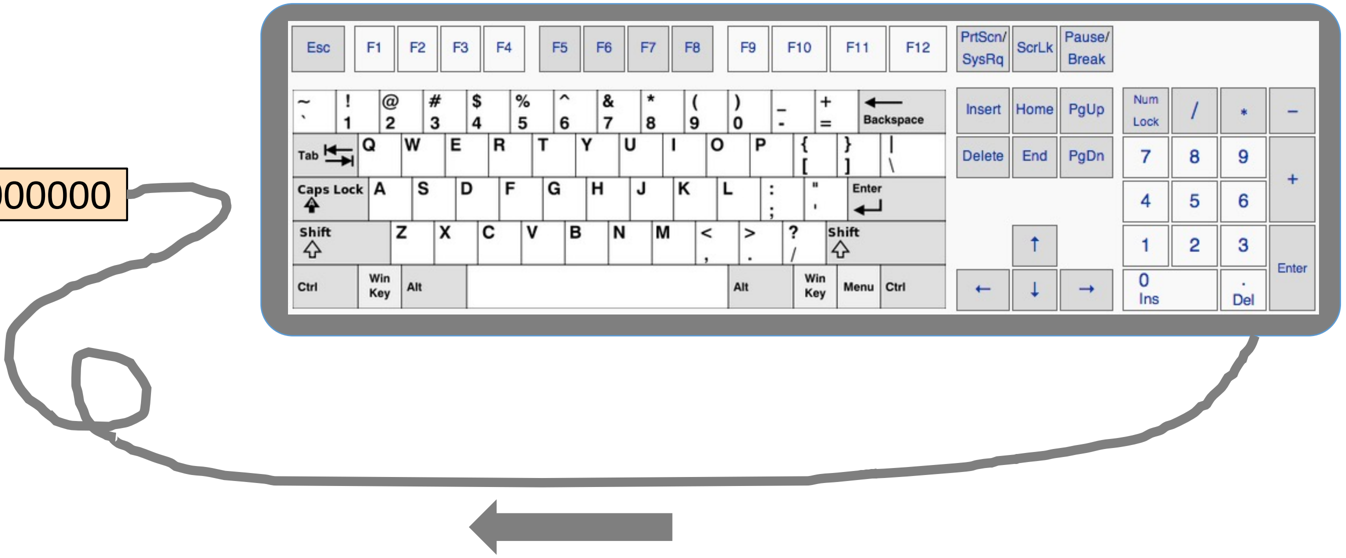
key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
. . .	. . .
f12	152

(Subset of Unicode)

Memory mapped input

RAM

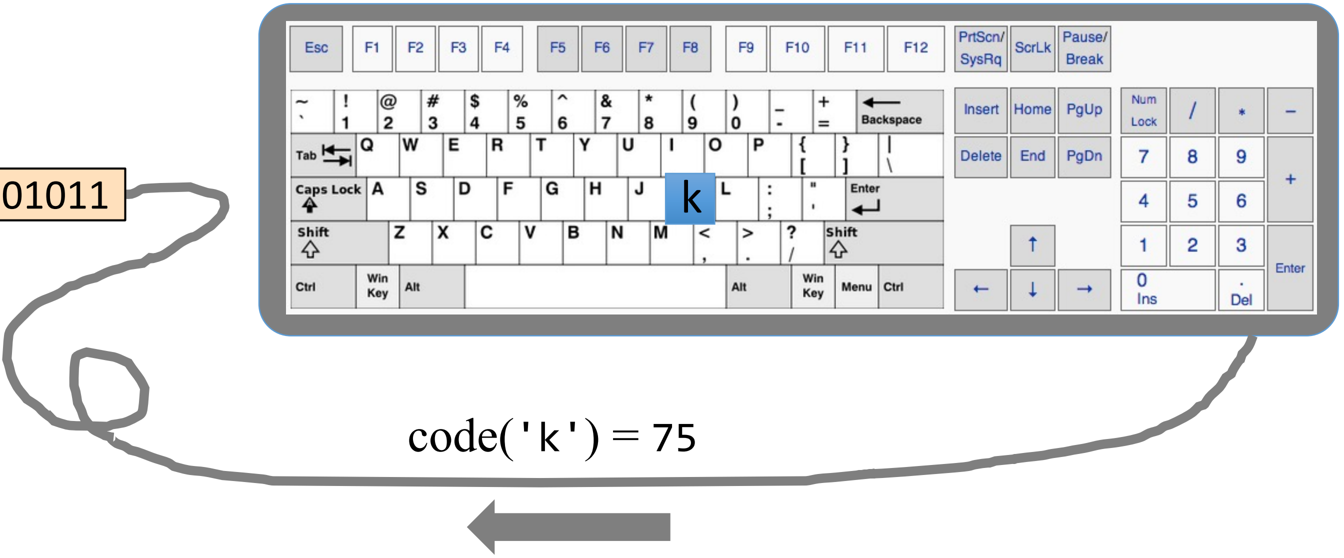
24576 | 0000000000000000
=
KBD
base address
of the keyboard
memory map



Memory mapped input

RAM

24576 = KBD base address of the keyboard memory map | 0000000001001011



code('k') = 75

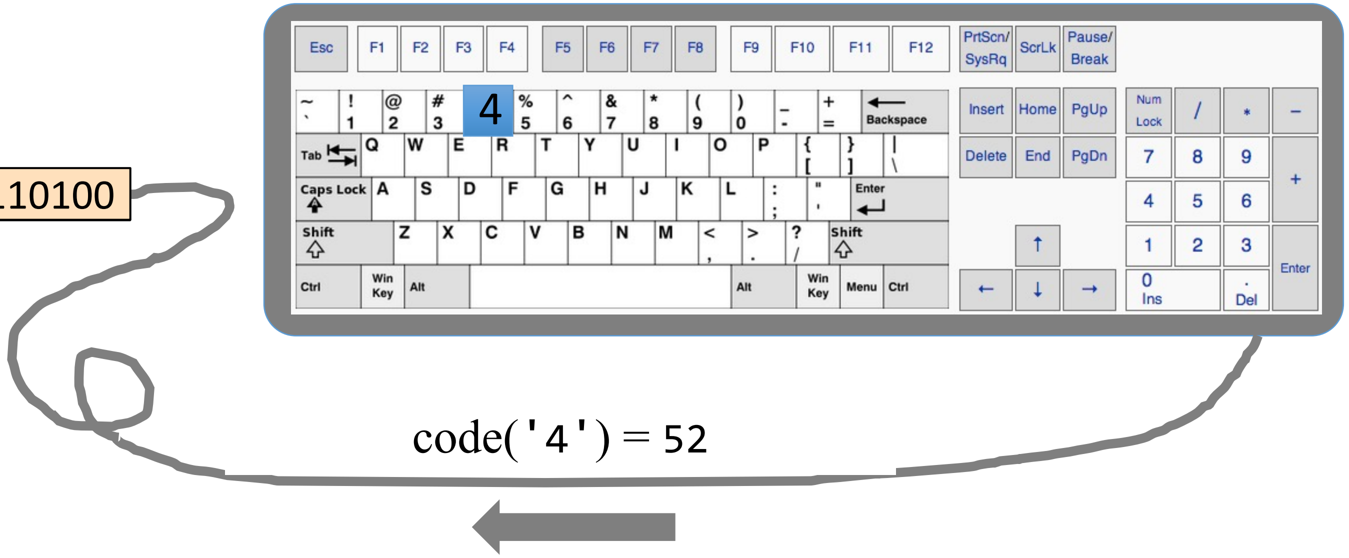
When a key is pressed on the keyboard,
the key’s character code appears in the keyboard memory map.

Memory mapped input

RAM

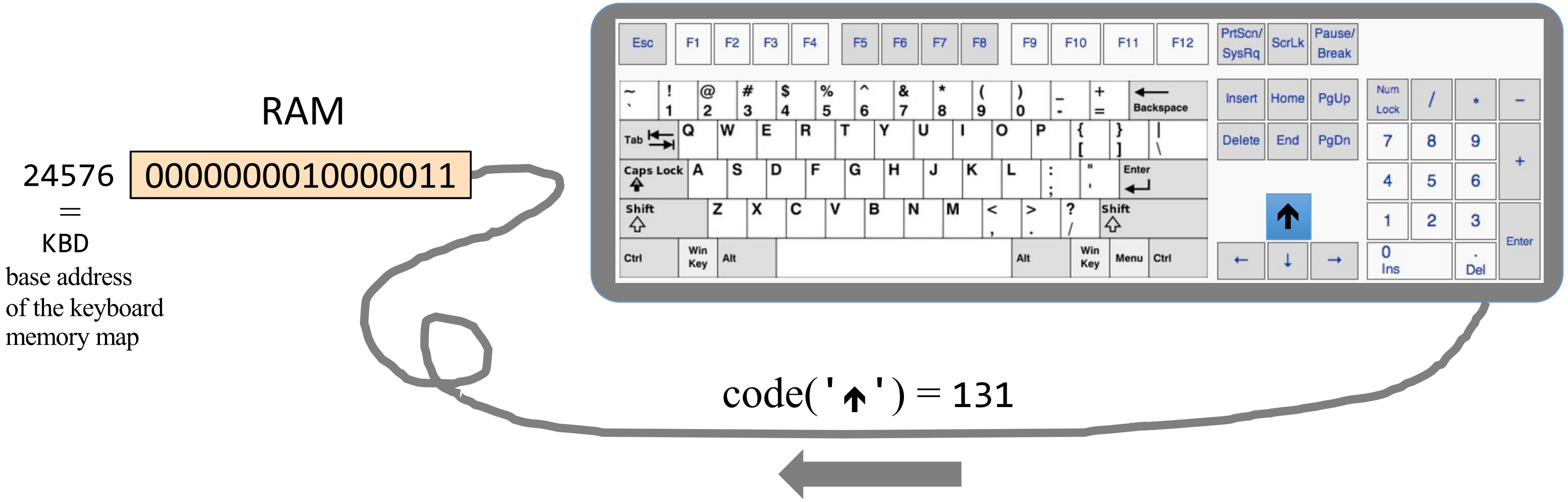
24576 = KBD base address of the keyboard memory map

24576	0000000000110100



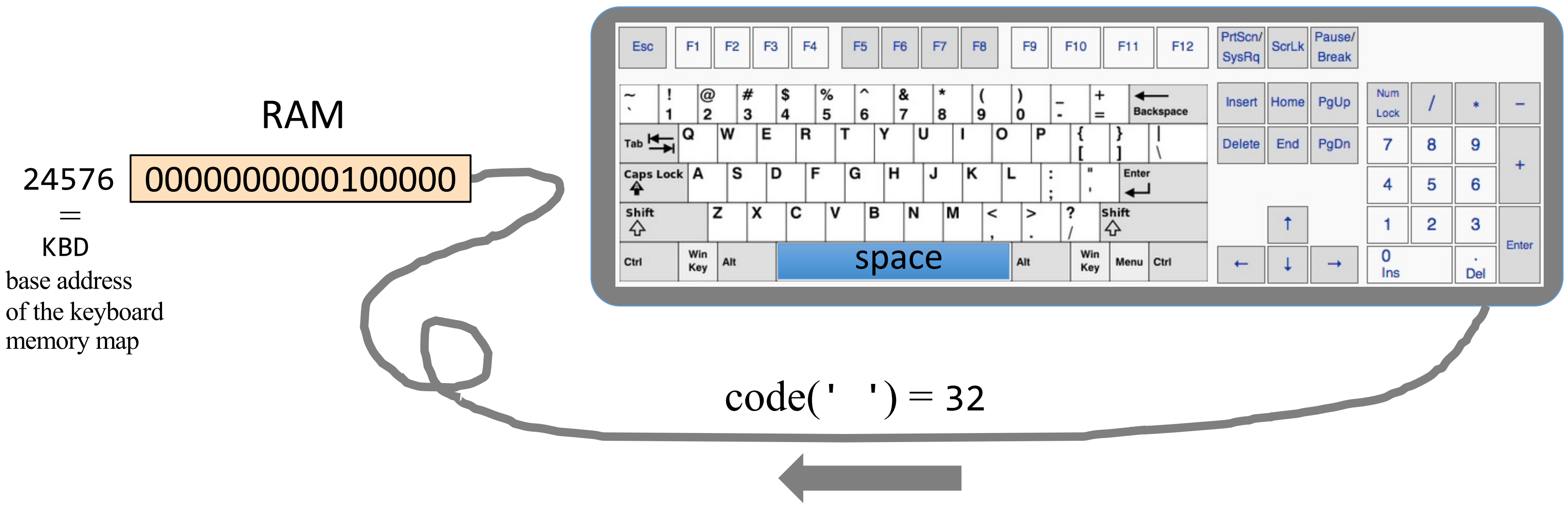
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

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



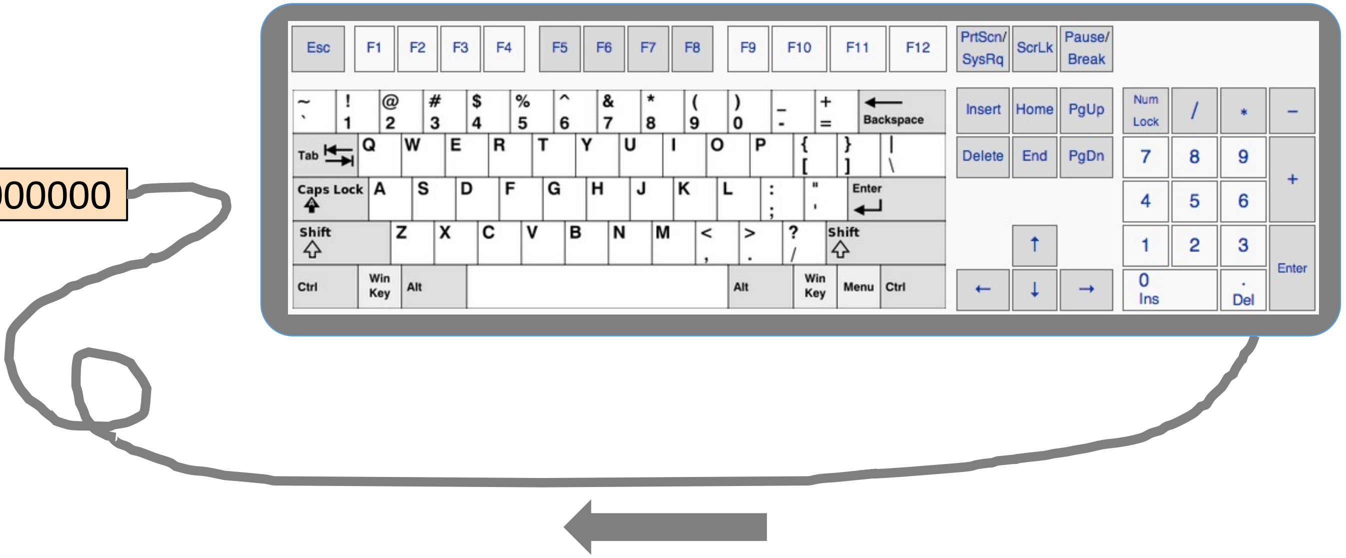
When a key is pressed on the keyboard,
the key’s character code appears in the keyboard memory map.

Memory mapped input

RAM

24576 = KBD base address of the keyboard memory map

24576	0000000000000000



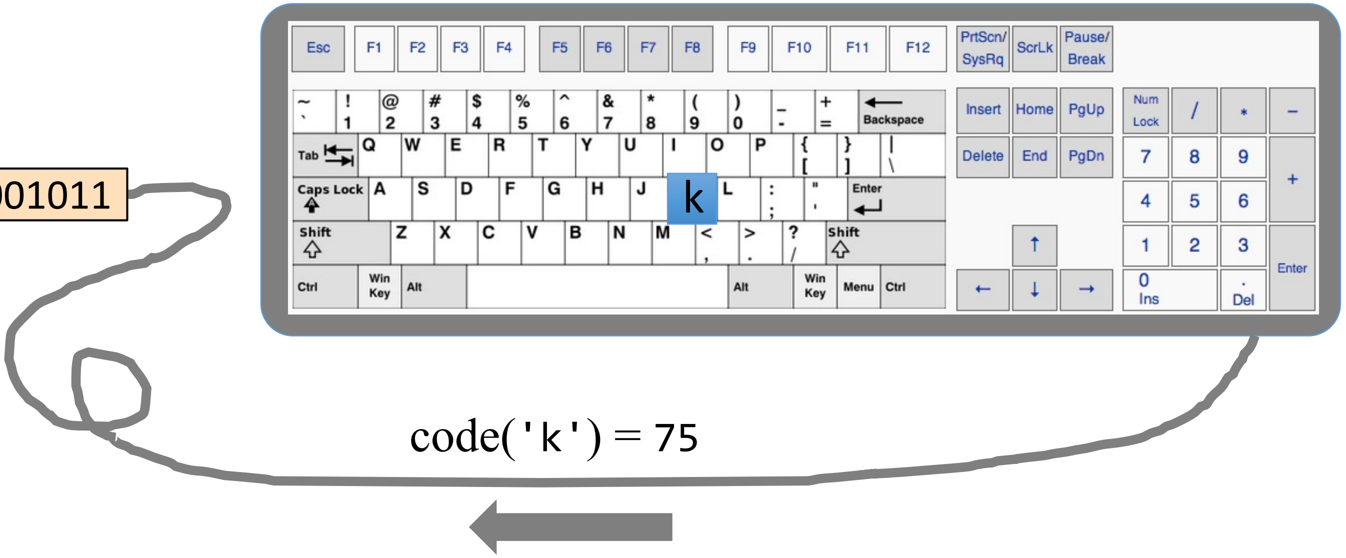
When no key is pressed, the resulting code is `0`.

Reading inputs

RAM

24576 = KBD — `0000000001001011`

base address of the keyboard memory map



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
```

Chapter 4: Machine Language

Overview

- Machine languages
- 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

- ✓ Usage
- ✓ Specification
- ✓ Output
- ✓ Input
- Project 4

Project 4

<u>Objectives</u>

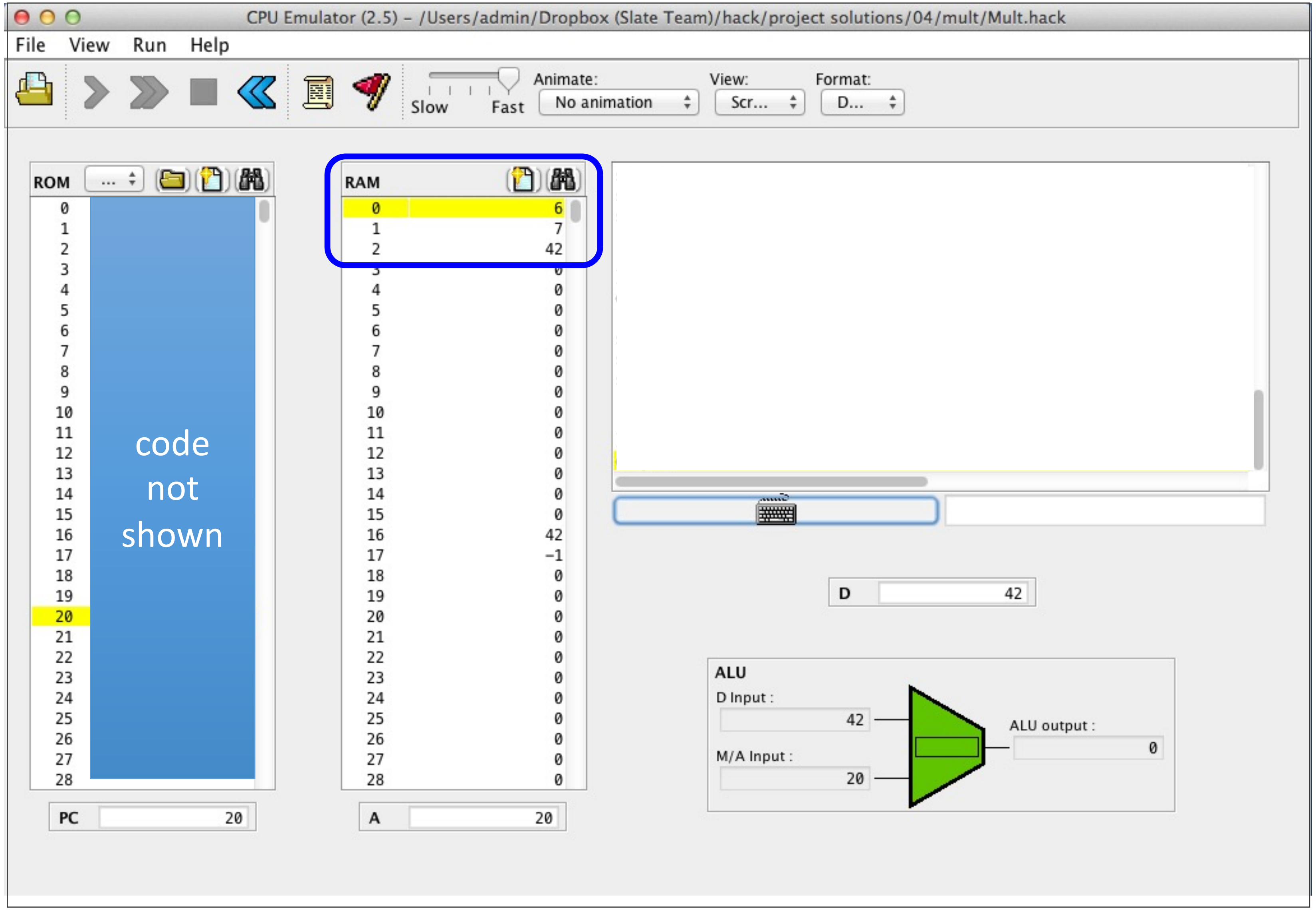
Gain a hands-on taste of:

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

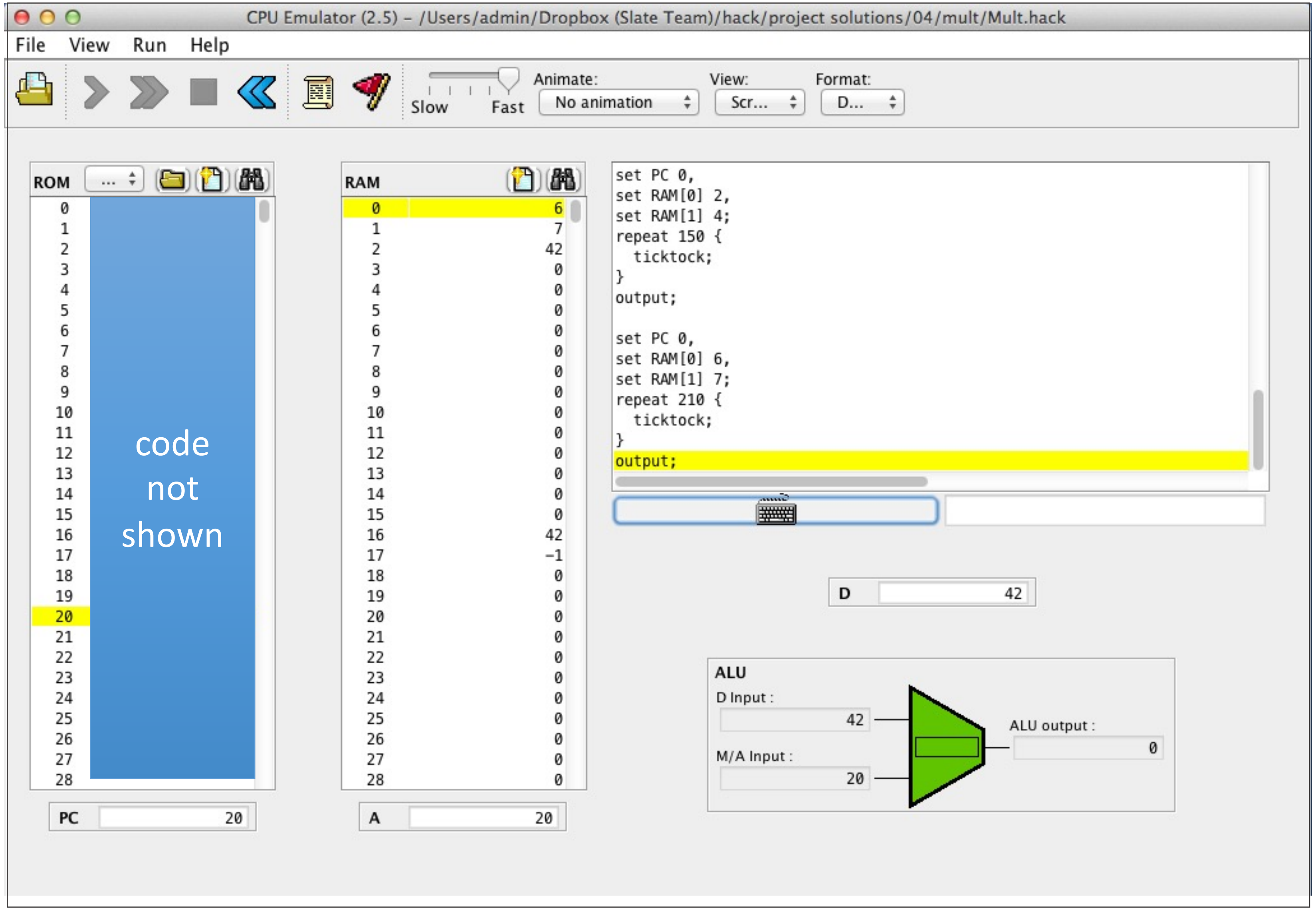
<u>Tasks</u>

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

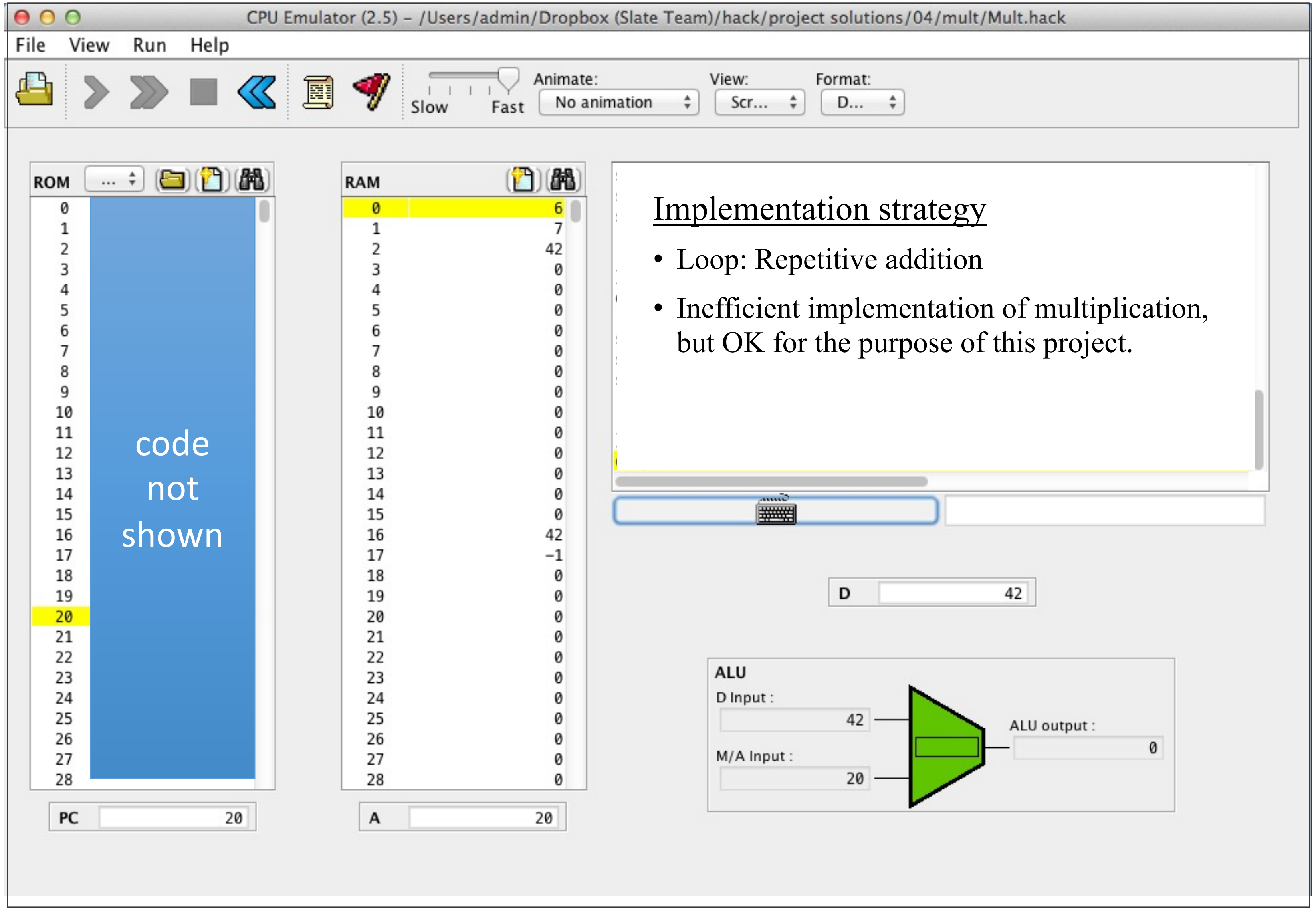
`Mult`: a program that computes `R2 = R0 * R1`



Mult: a program that computes R2 = R0 * R1



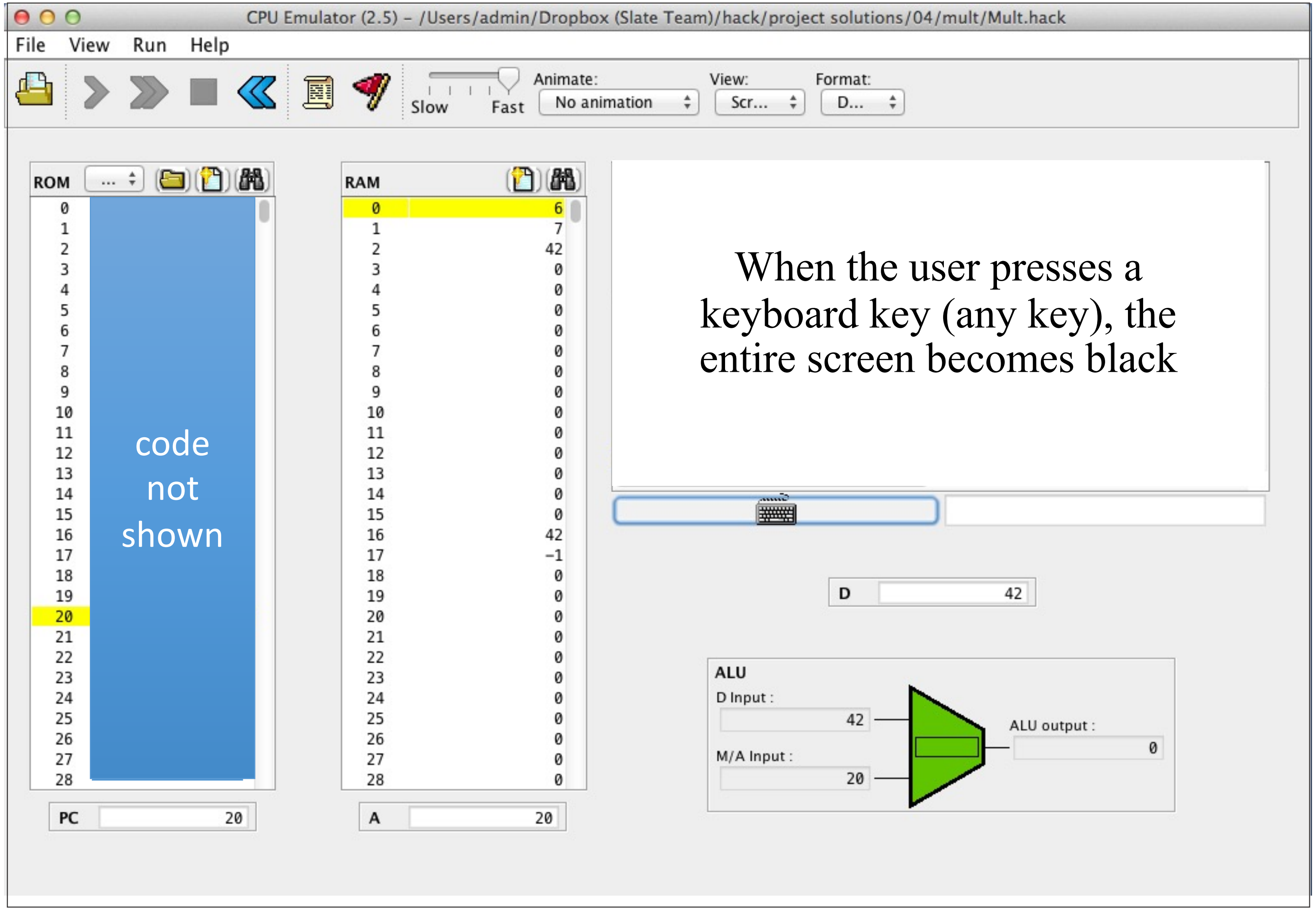
`Mult`: a program that computes `R2 = R0 * R1`



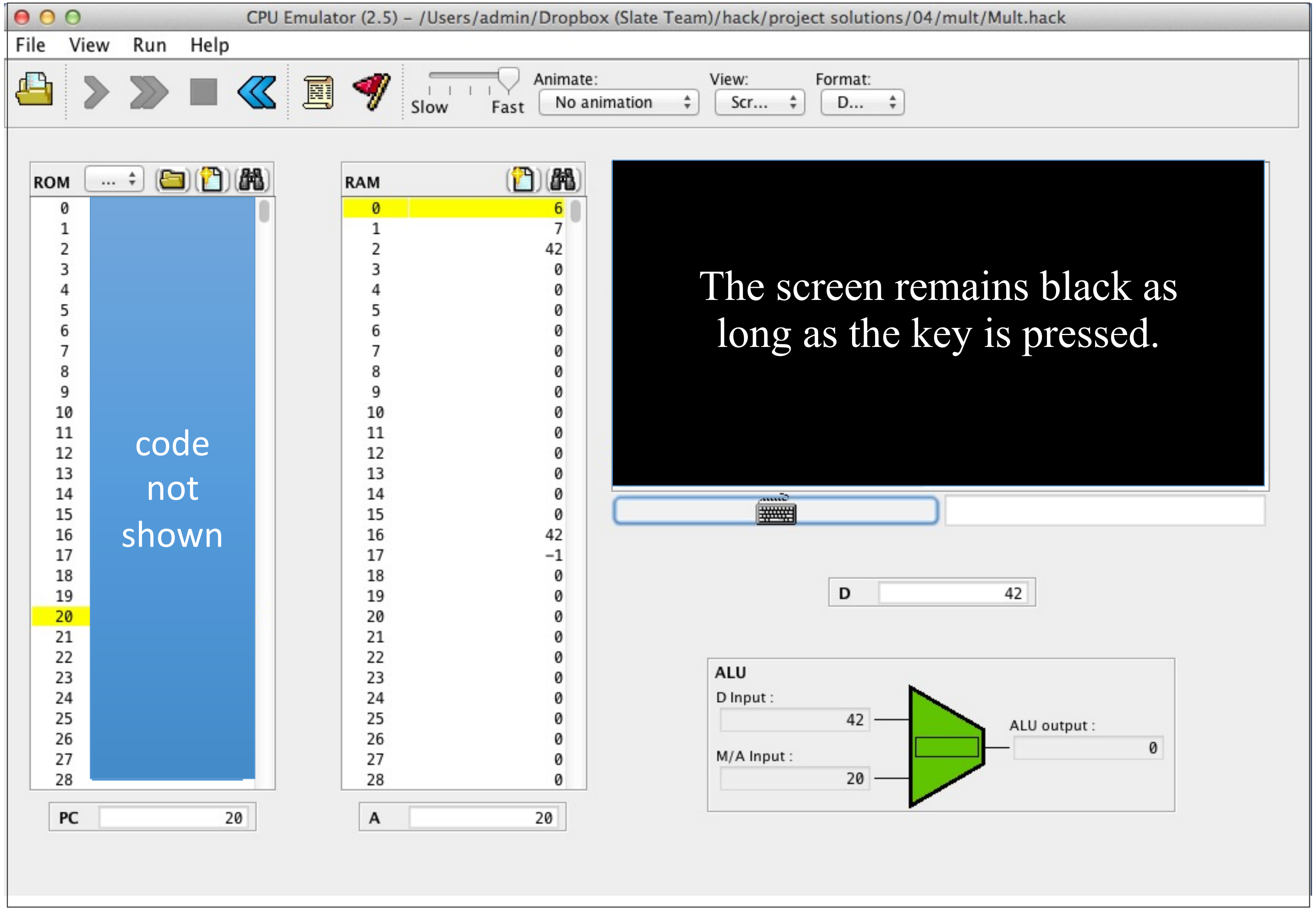
Implementation strategy

- Loop: Repetitive addition
- Inefficient implementation of multiplication, but OK for the purpose of this project.

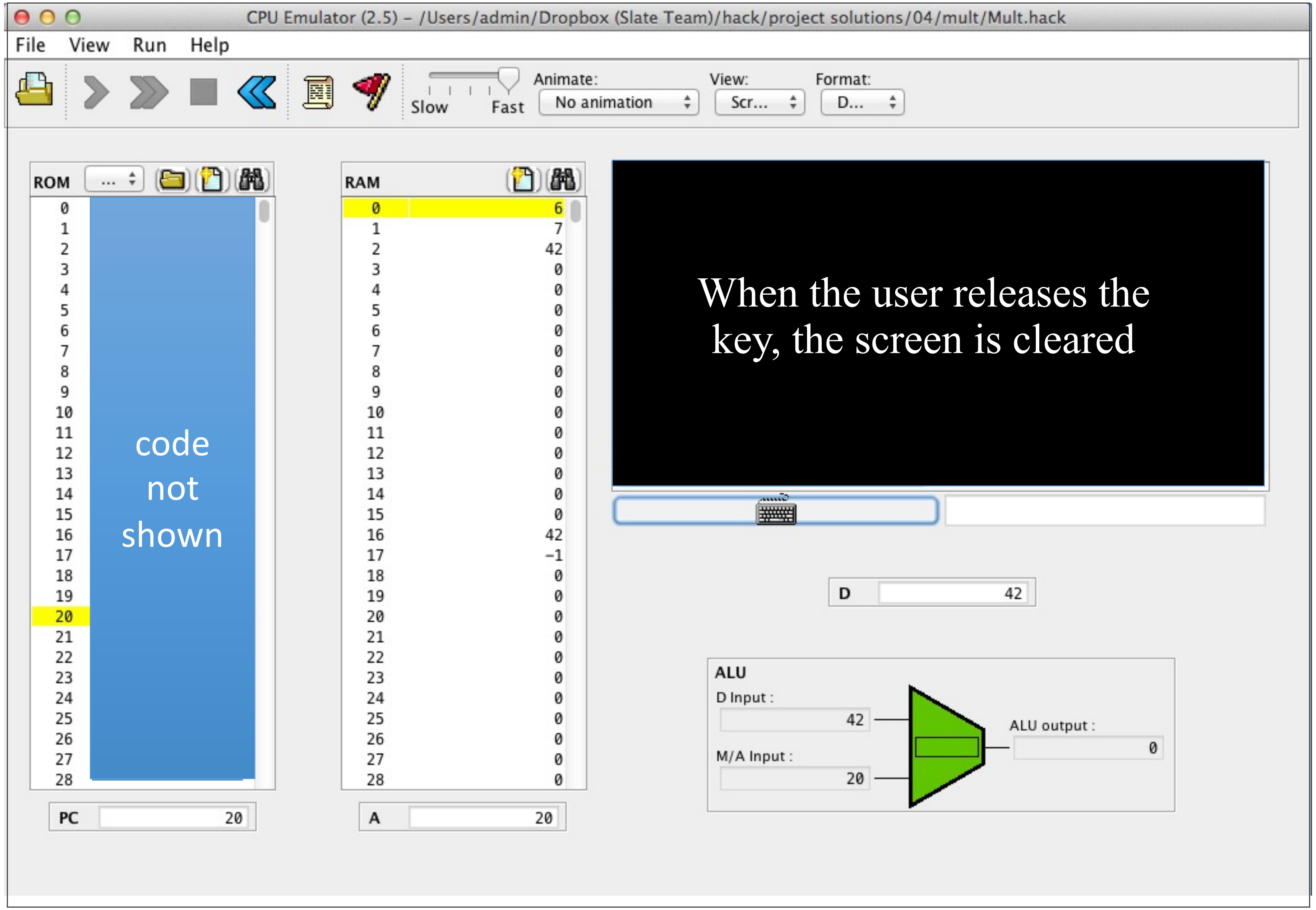
`Fill`: a simple interactive program



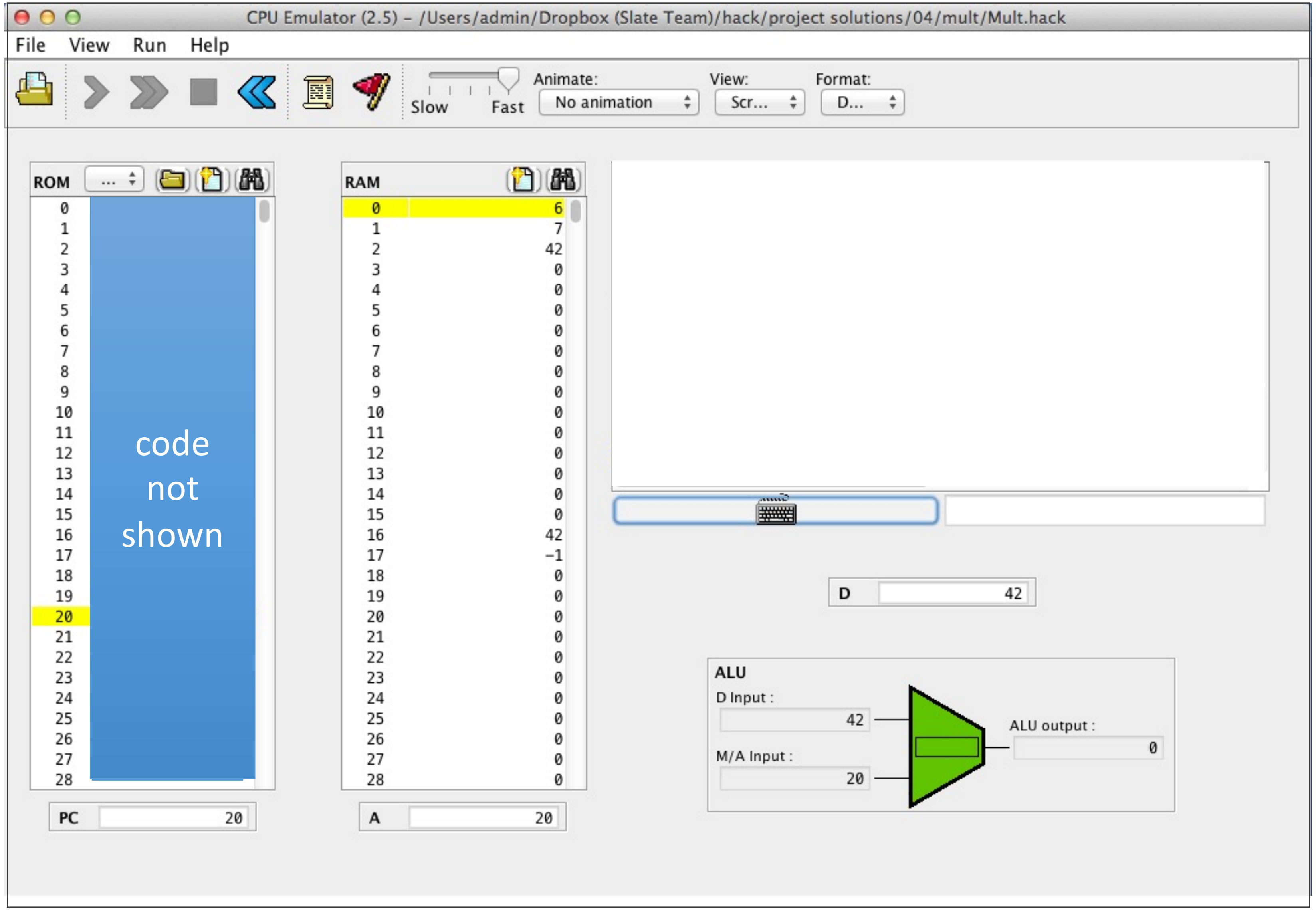
Fill: a simple interactive program



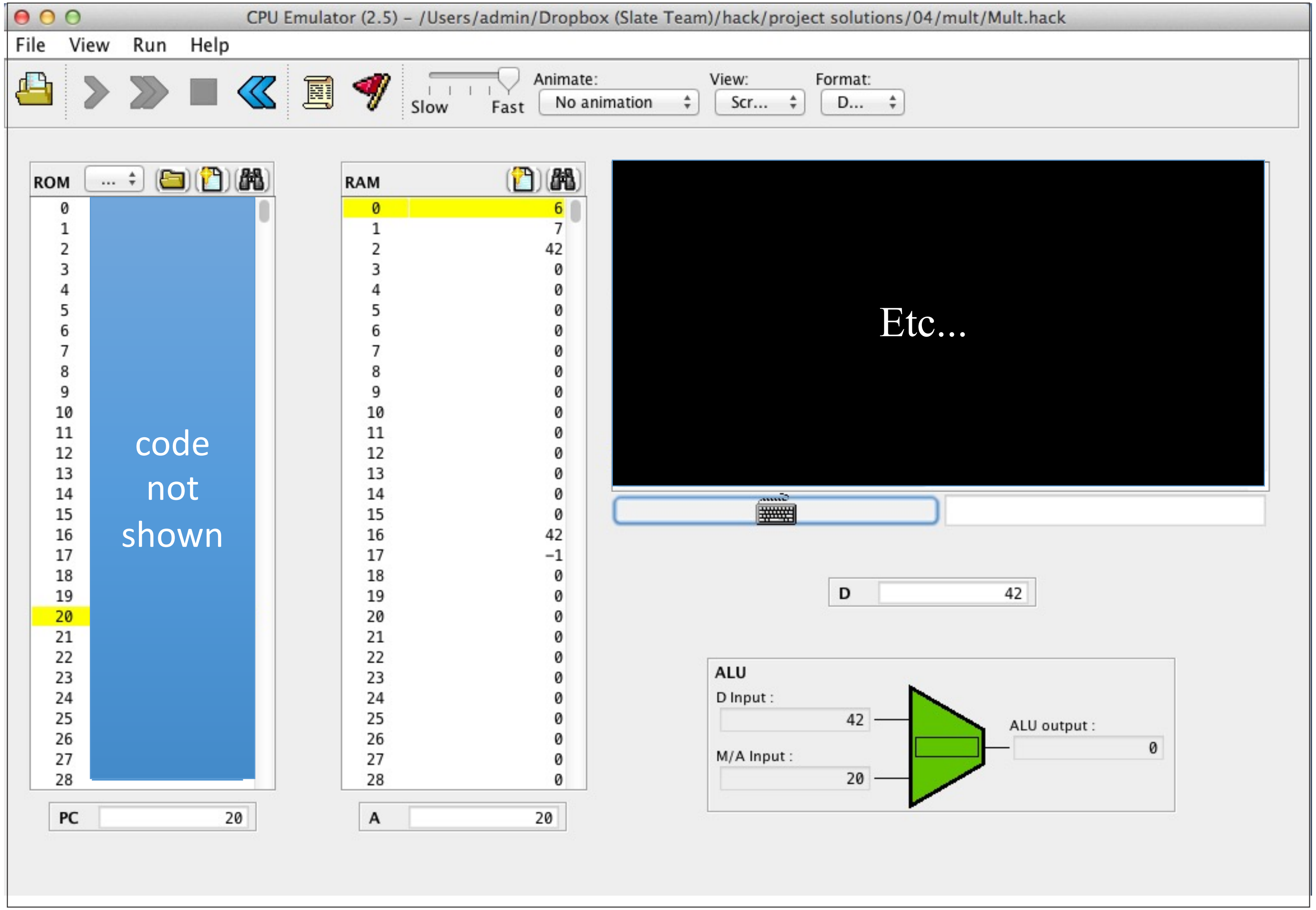
`Fill`: a simple interactive program



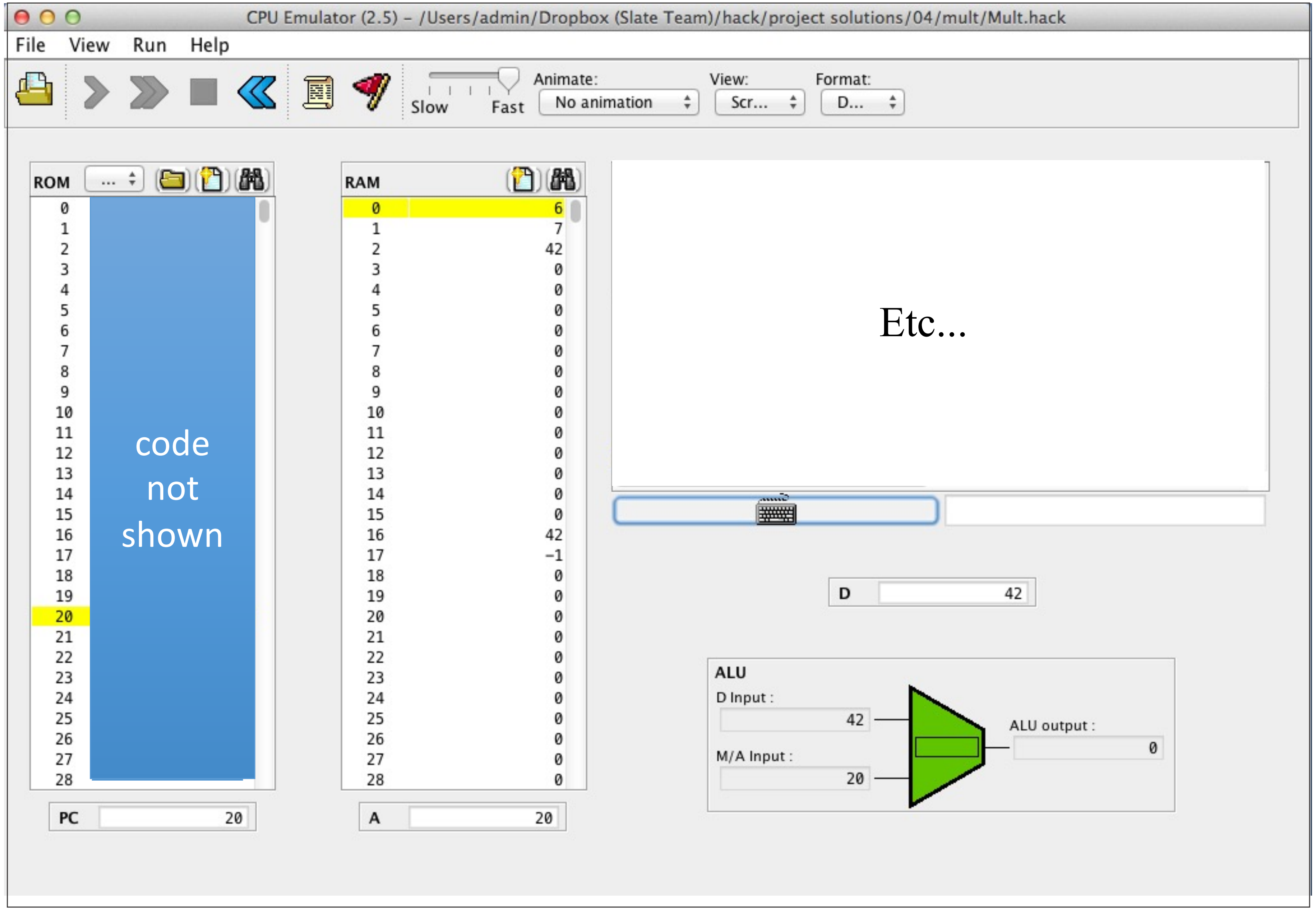
`Fill`: a simple interactive program



`Fill`: a simple interactive program



`Fill`: a simple interactive program



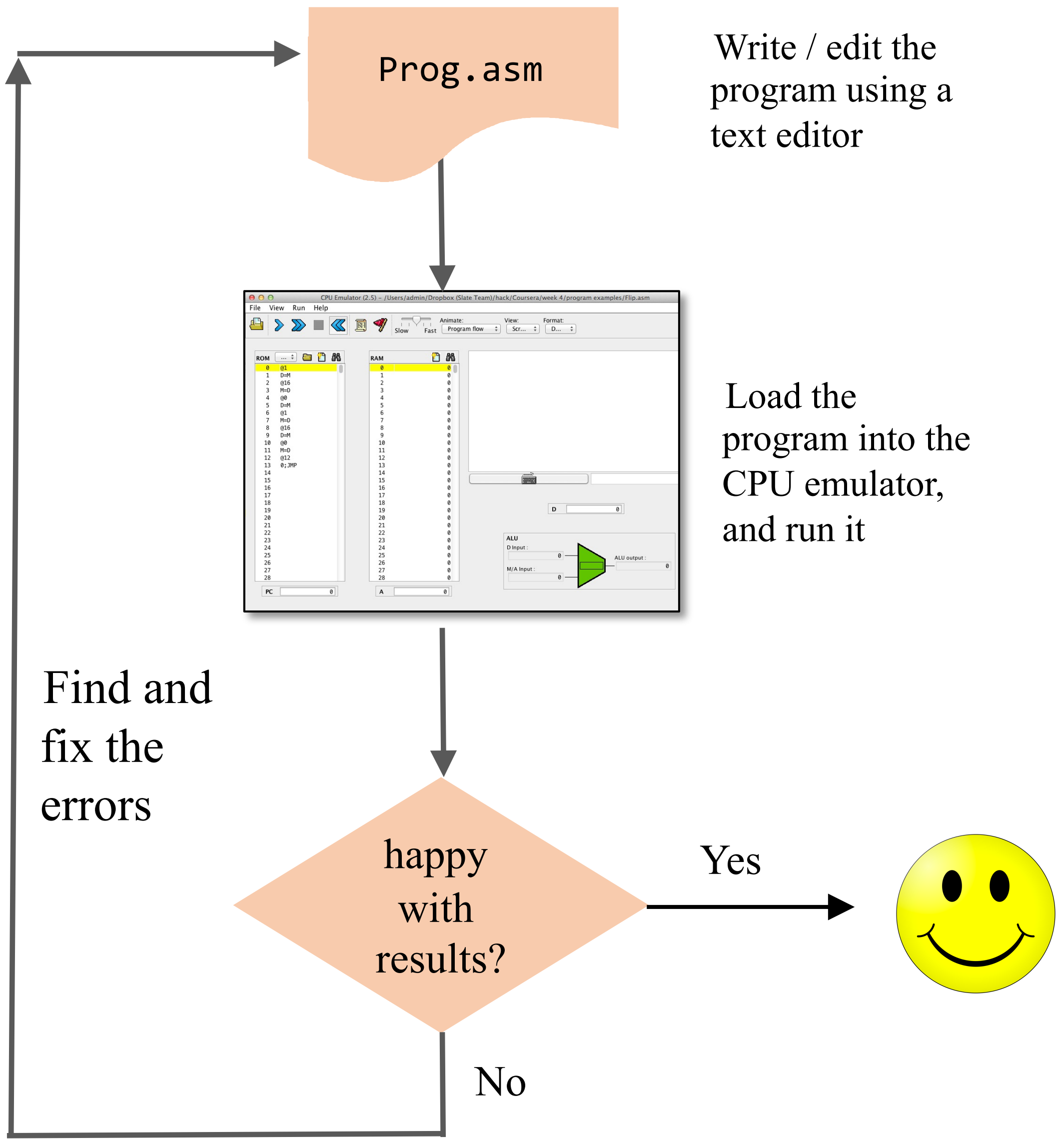
`Fill`: a simple interactive program

Implementation strategy

- 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.

Program development process



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

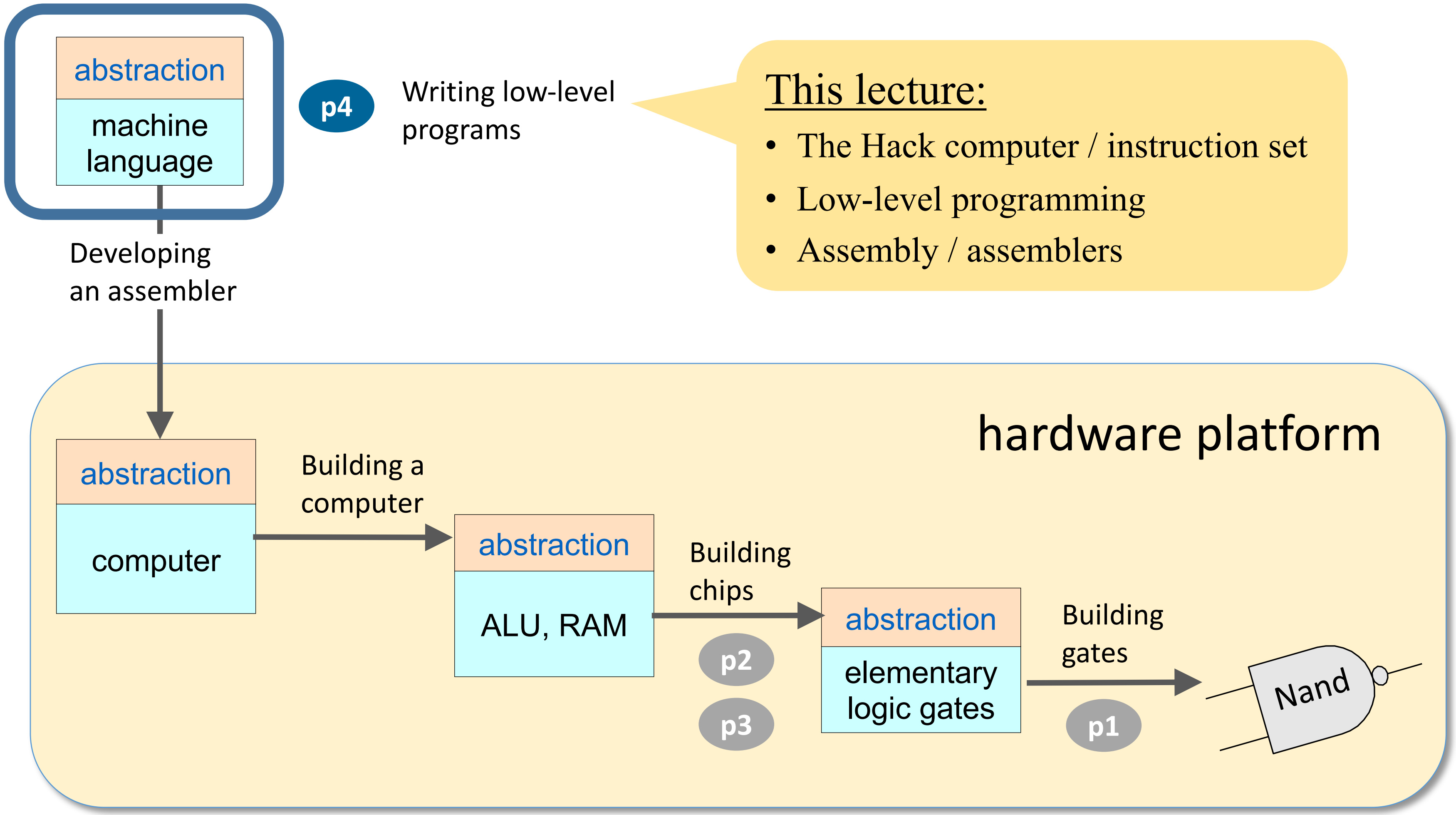
- Use symbolic variables and labels
- Use sensible variable and label names
- Variables: lower-case
- Labels: upper-case
- Use indentation
- Start by writing pseudocode.

Task 3:Define and write a program of your own

Any ideas?

It's your call!

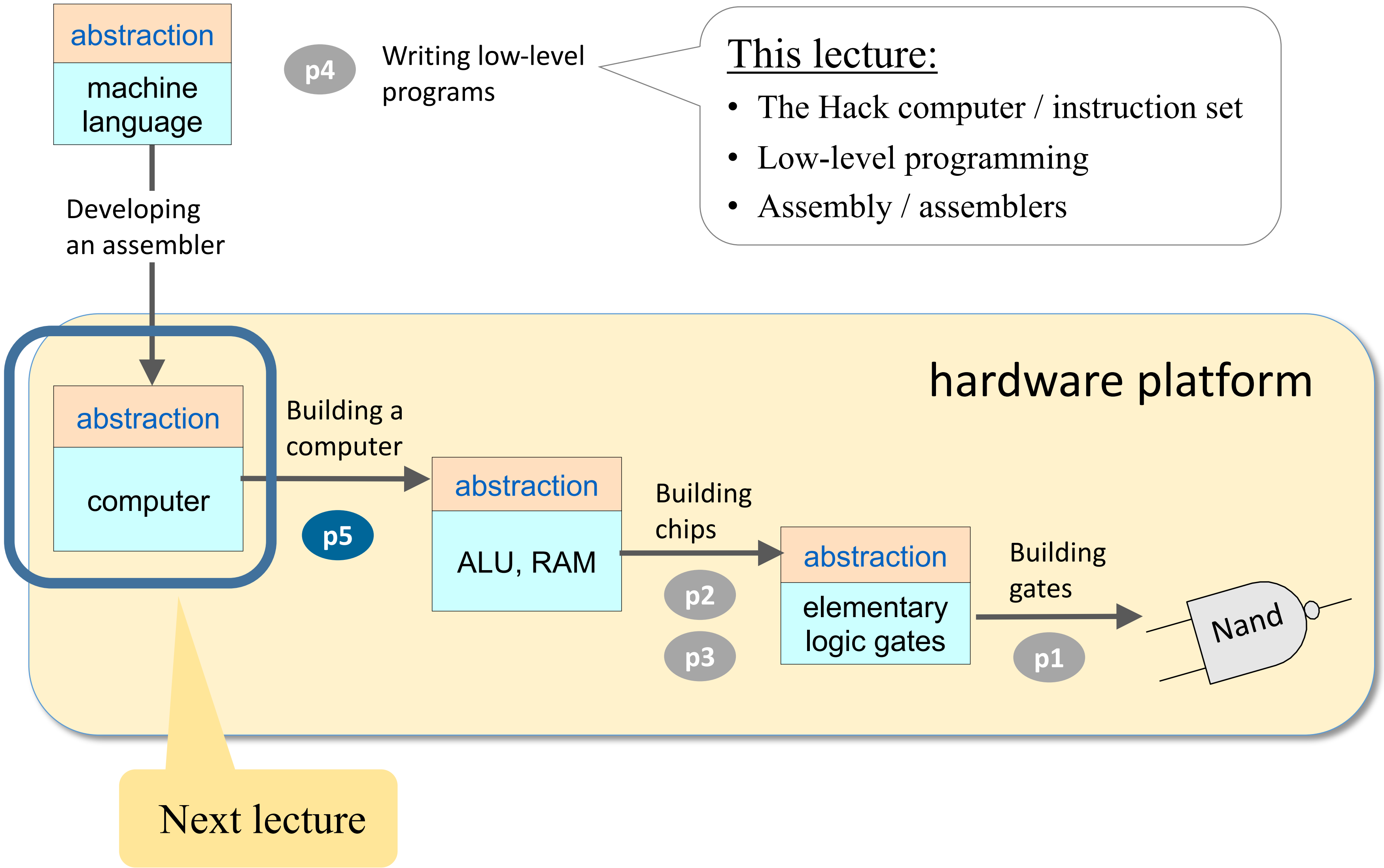
Nand to Tetris Roadmap (Part I: Hardware)



This lecture:

- The Hack computer / instruction set
- Low-level programming
- Assembly / assemblers

Nand to Tetris Roadmap (Part I: Hardware)



This lecture:

- The Hack computer / instruction set
- Low-level programming
- Assembly / assemblers