Lecture 4

Machine Language

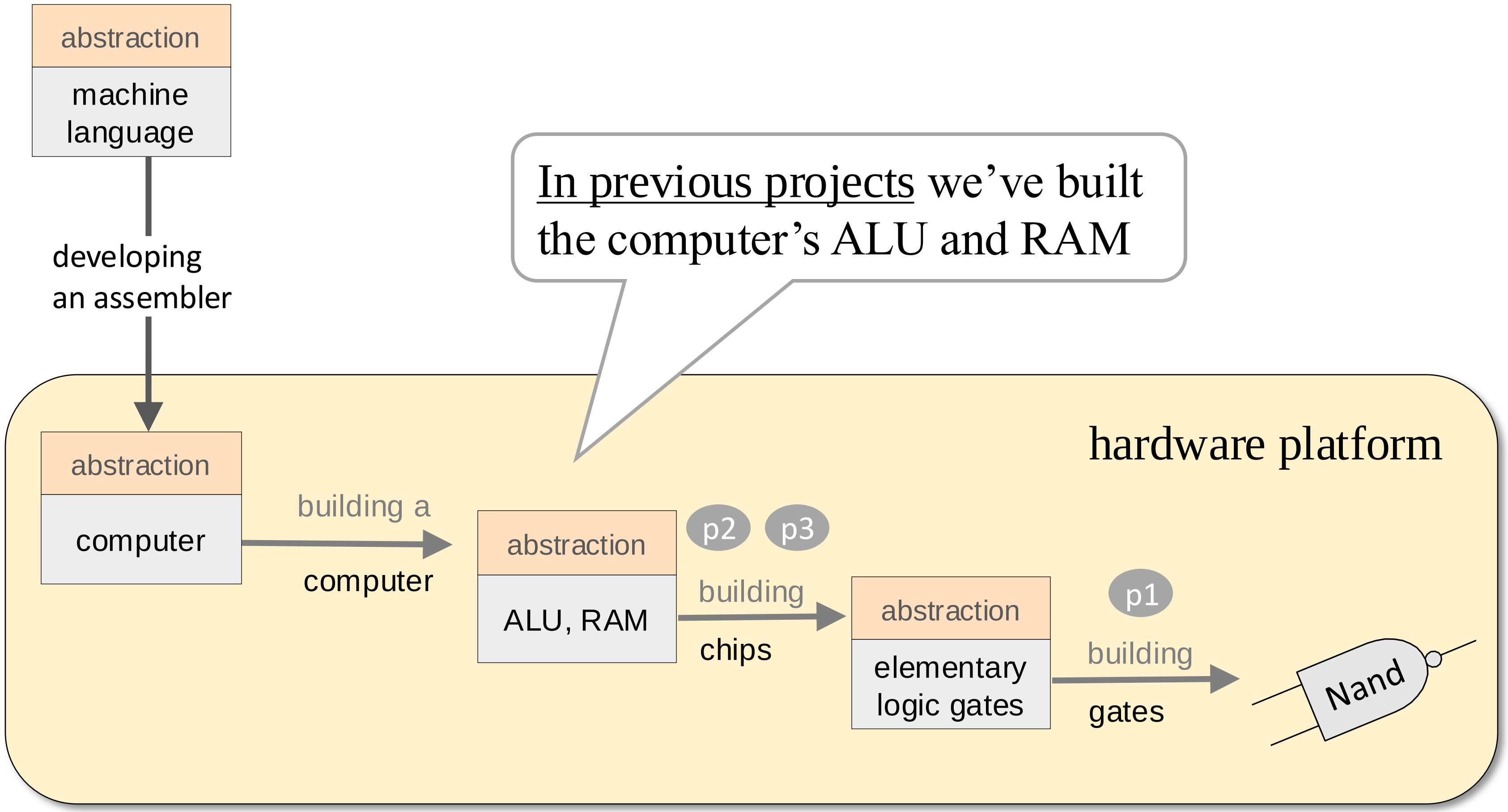
These slides support chapter 4 of the book

The Elements of Computing Systems

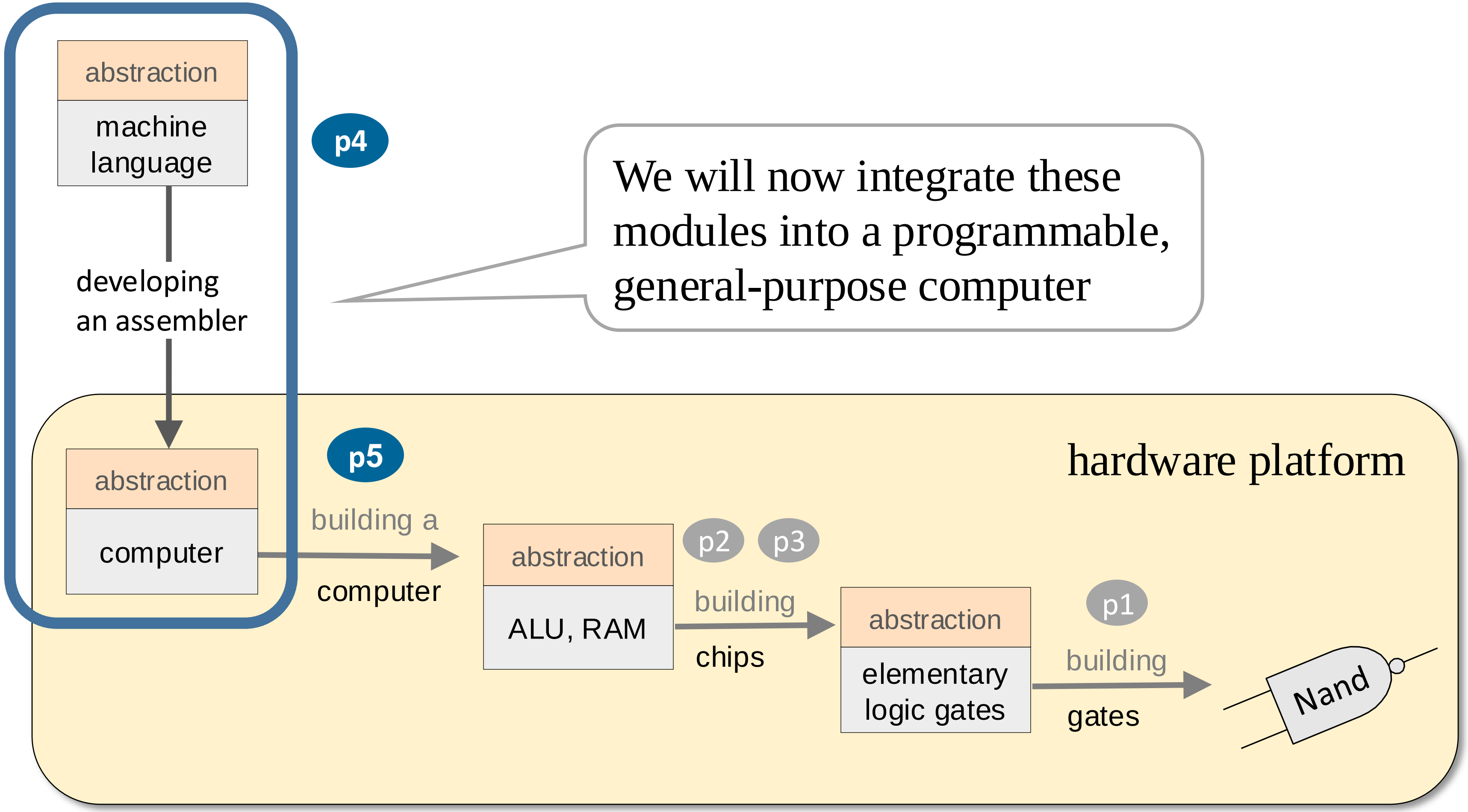
By Noam Nisan and Shimon Schocken

MIT Press, 2021

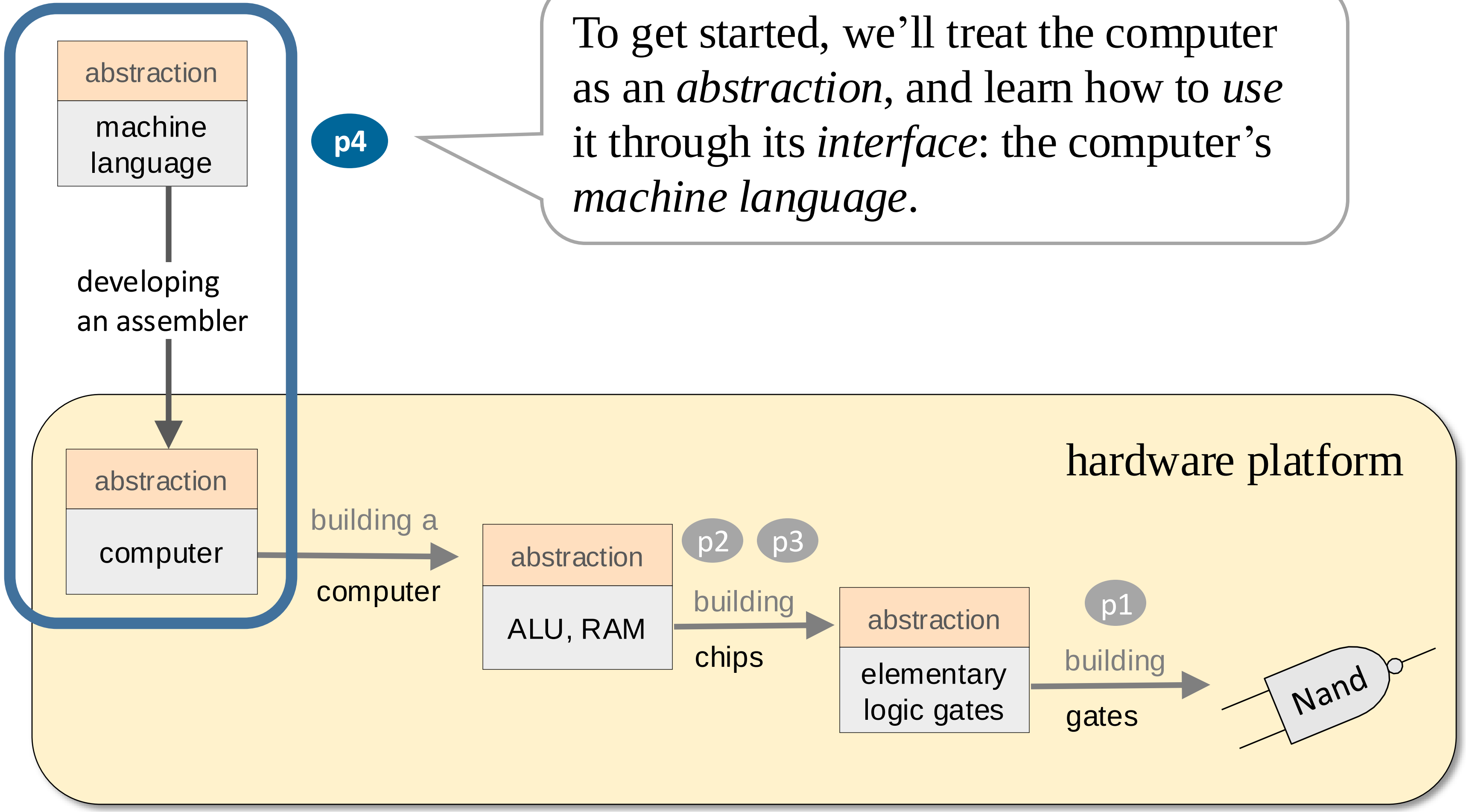
Nand to Tetris Roadmap: Hardware



Nand to Tetris Roadmap: Hardware

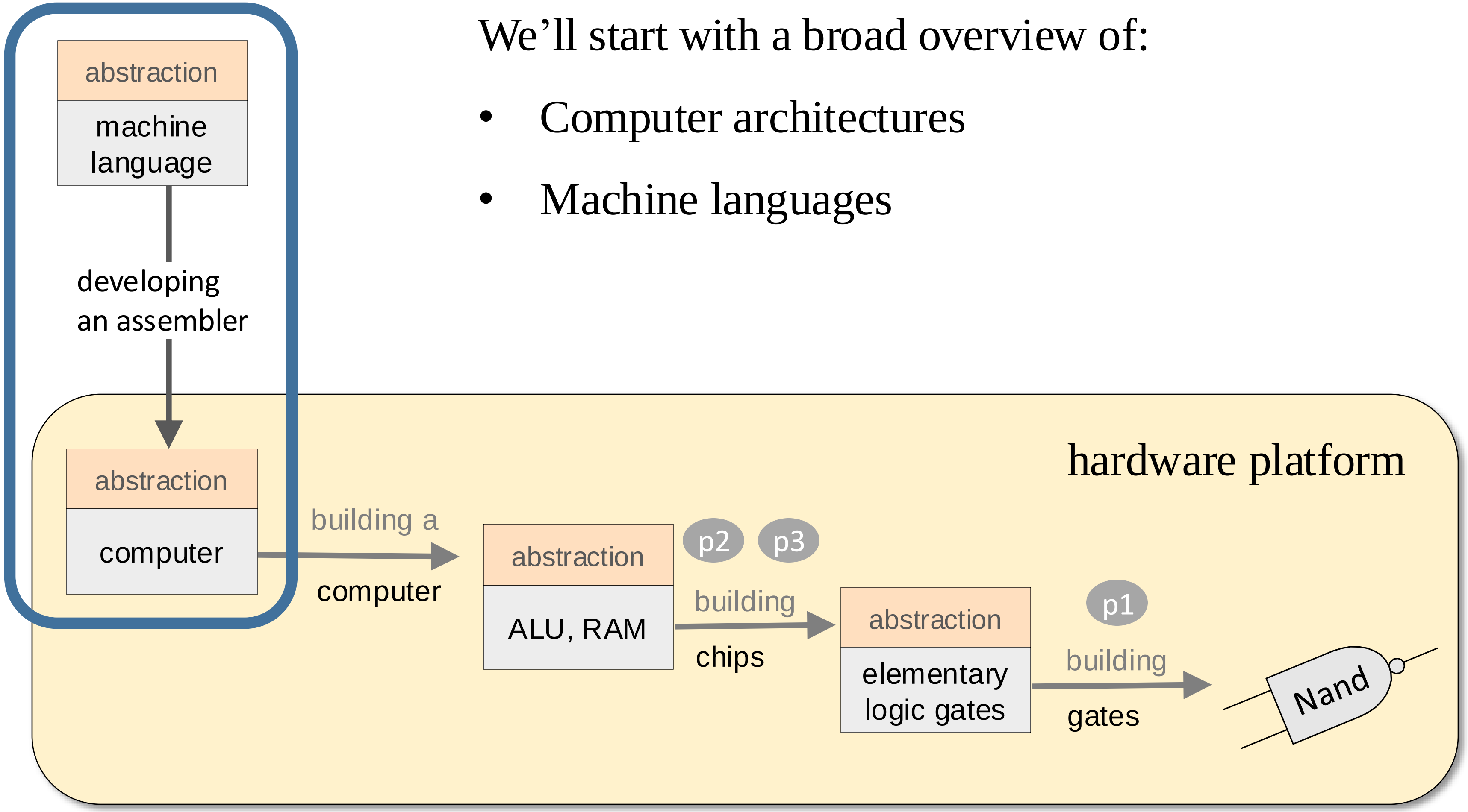


Nand to Tetris Roadmap: Hardware



To get started, we'll treat the computer as an *abstraction*, and learn how to *use* it through its *interface*: the computer's *machine language*.

Nand to Tetris Roadmap: Hardware



We'll start with a broad overview of:

- Computer architectures
- Machine languages

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

(1843)

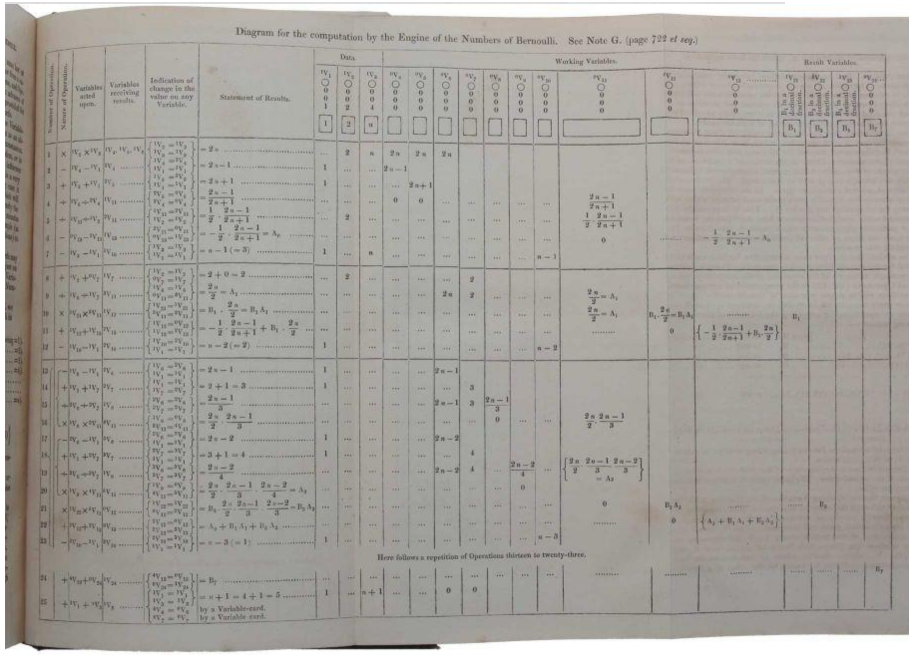


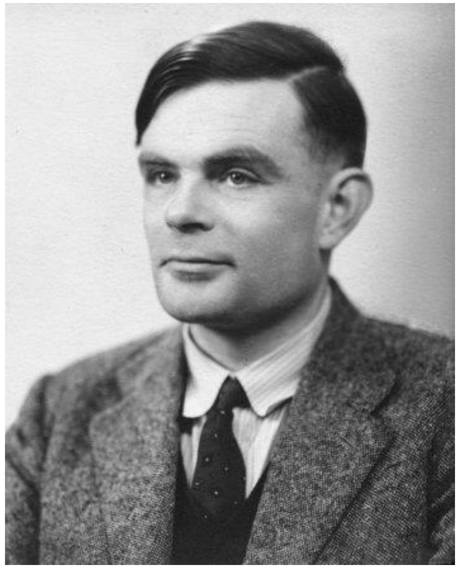
Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 et seq.)

Early symbolic program

Landmark "proof of concept" that a fixed computer can be programmed to perfom different tasks

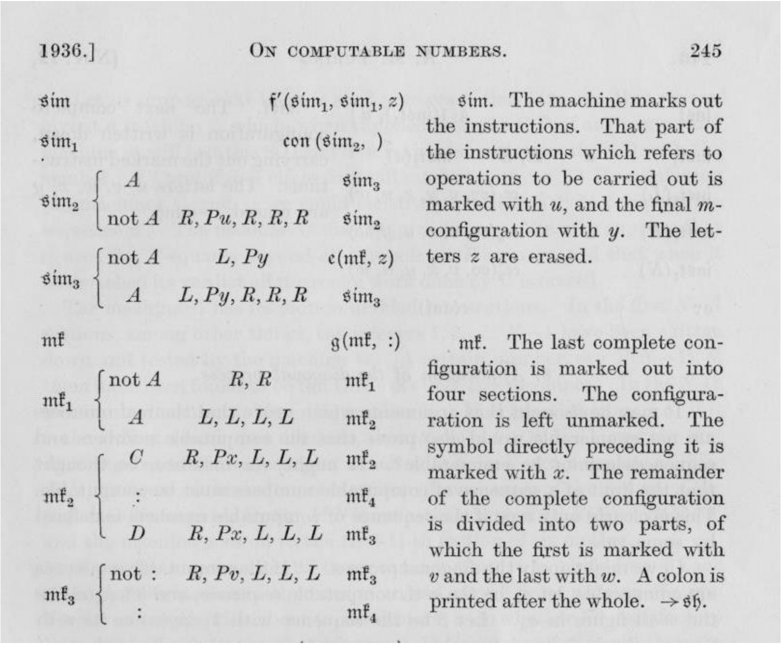
Computer systems are flexible and versatile

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. $\rightarrow \mathfrak{sh}$.

Universal Turing Machine

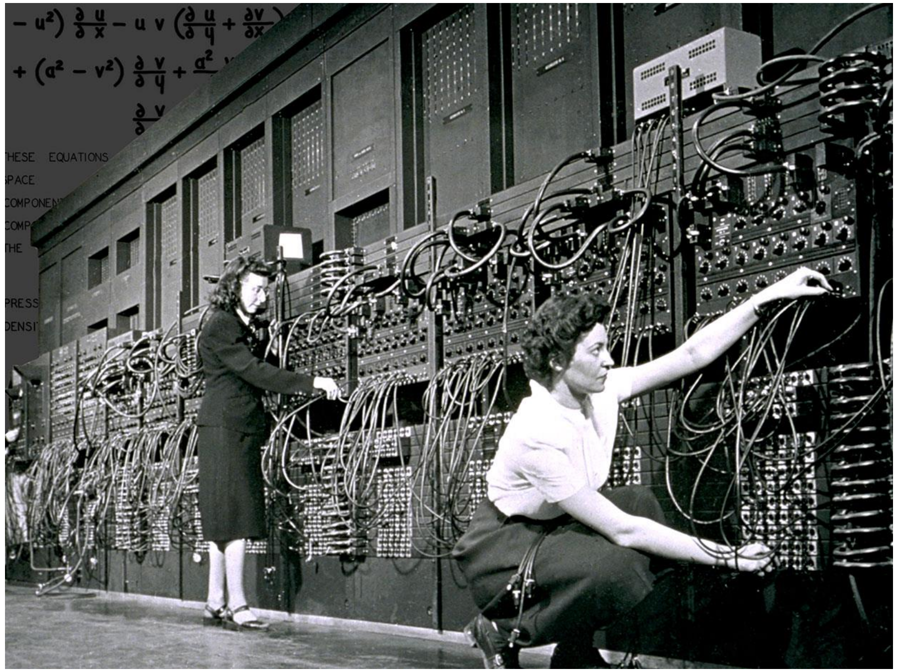
Landmark paper, describing a theoretical general-purpose computer

Computer systems are flexible and versatile

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

John Von Neumann

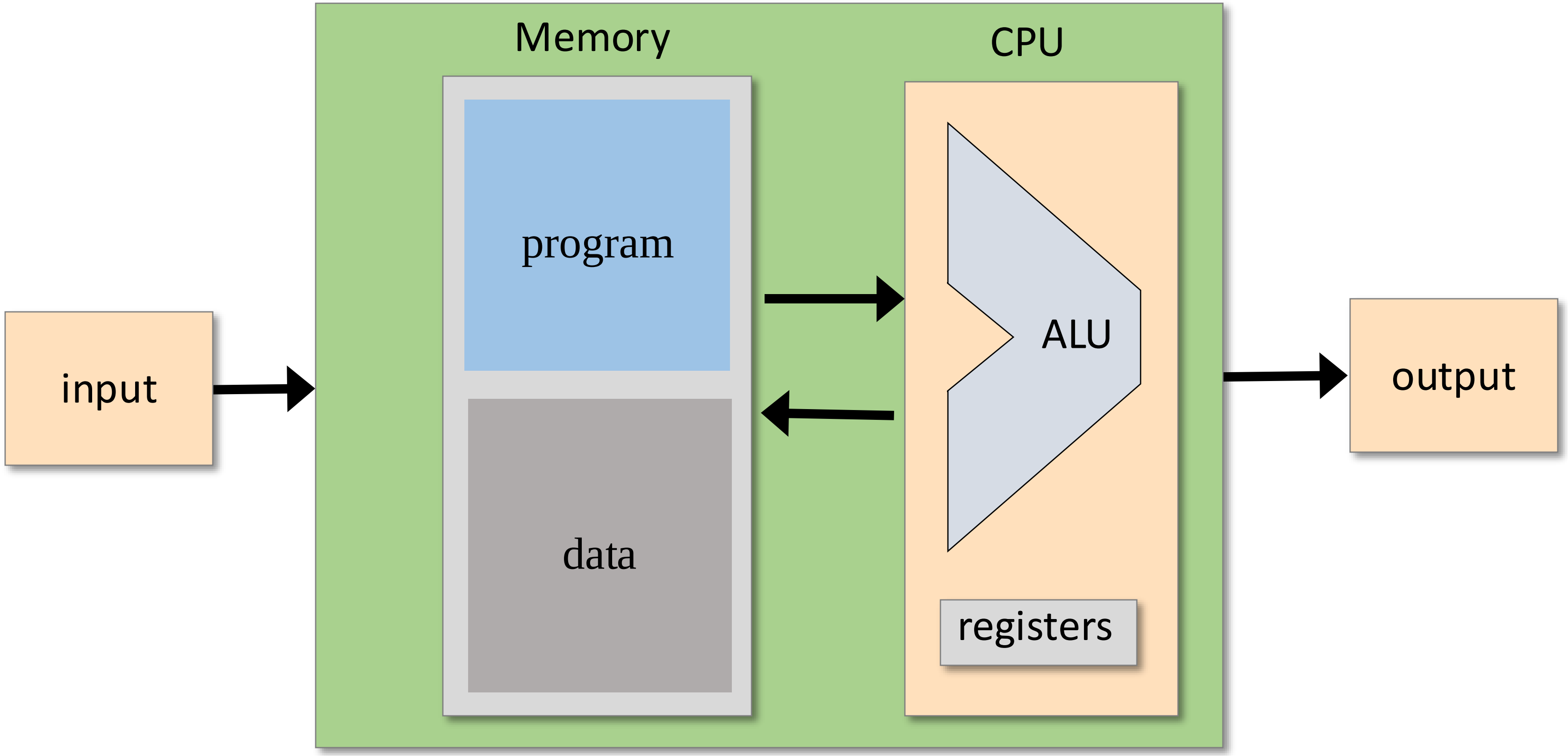
(1945)



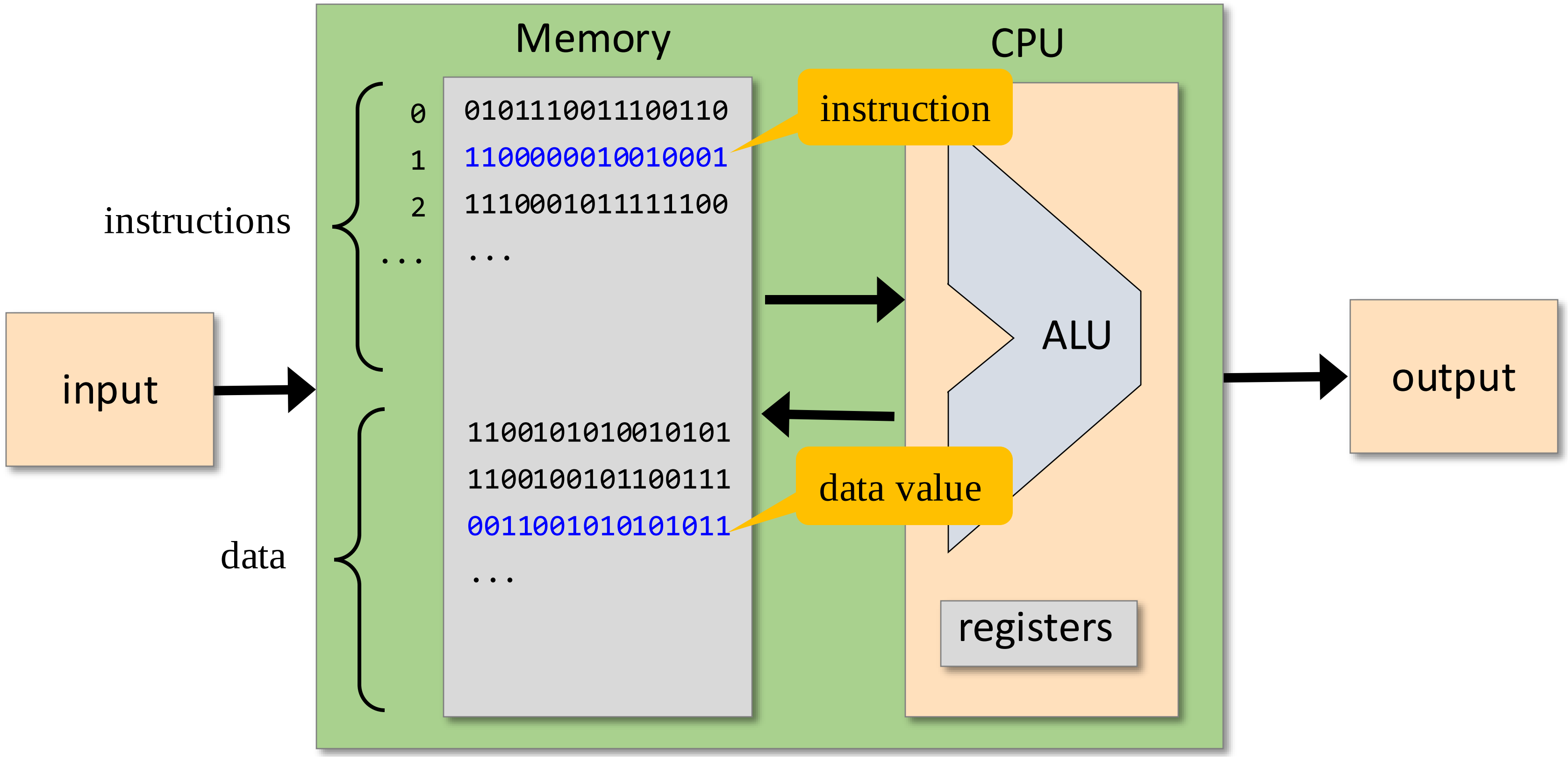
Landmark general-purpose computer

ENIAC, University of Pennsylvania

Computer architecture



Computer architecture



<u>Stored program concept</u>

- 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

Programming examples

- Basic
- Iteration
- Pointers

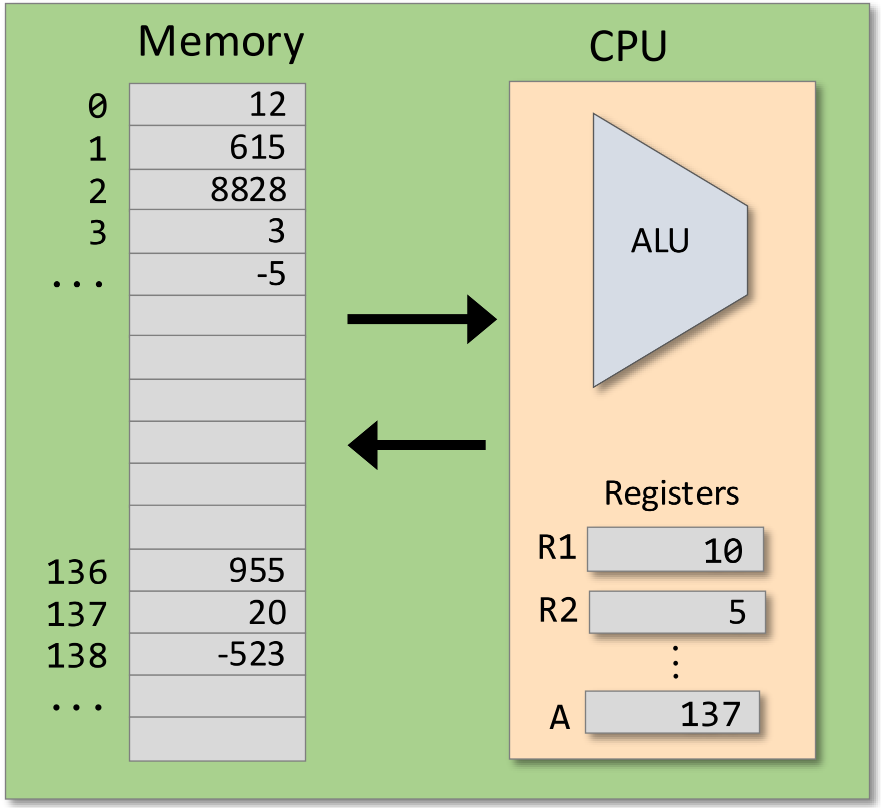
Symbolic programming

- Control
- Variables
- Labels

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

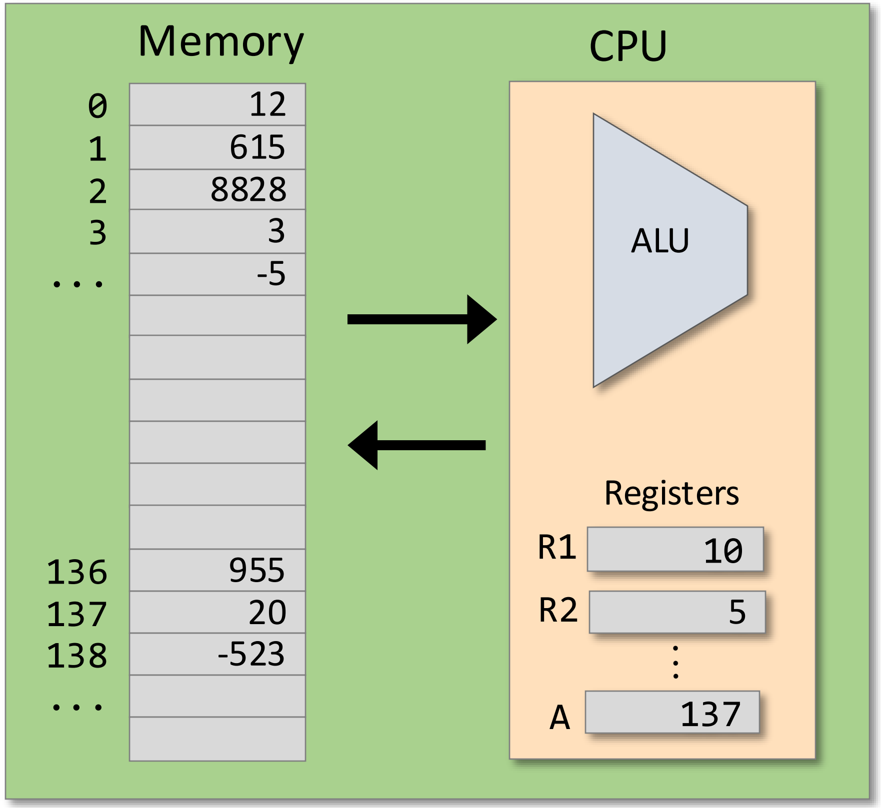
Hold data values

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 (using, for example, a RISC syntax)

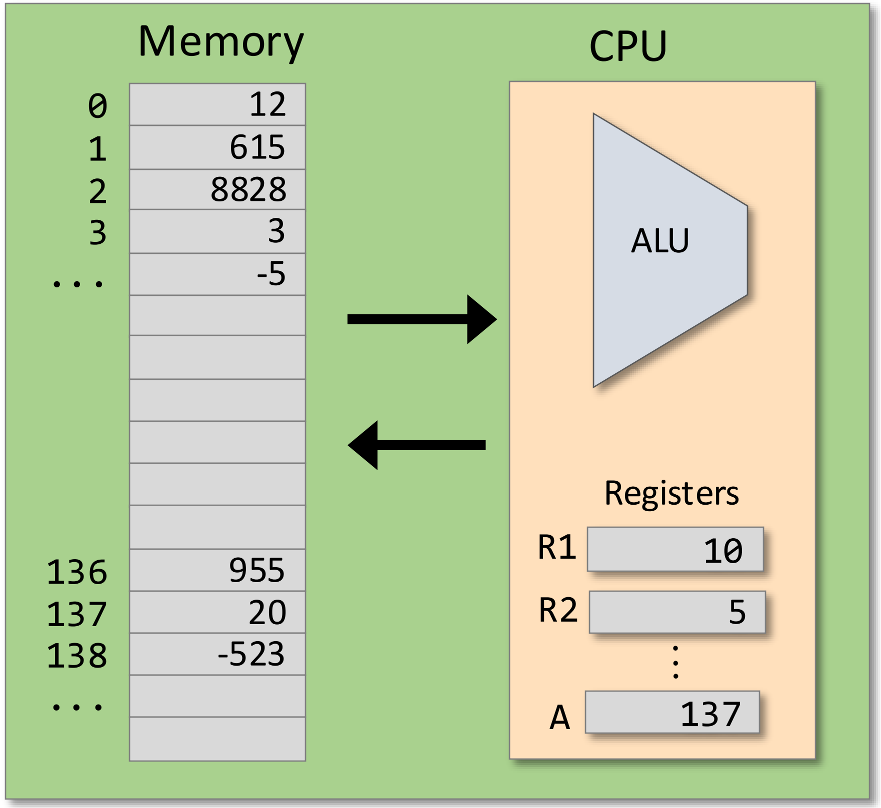
```
// R1 ← R1 + R2
add R1, R2

// 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
```



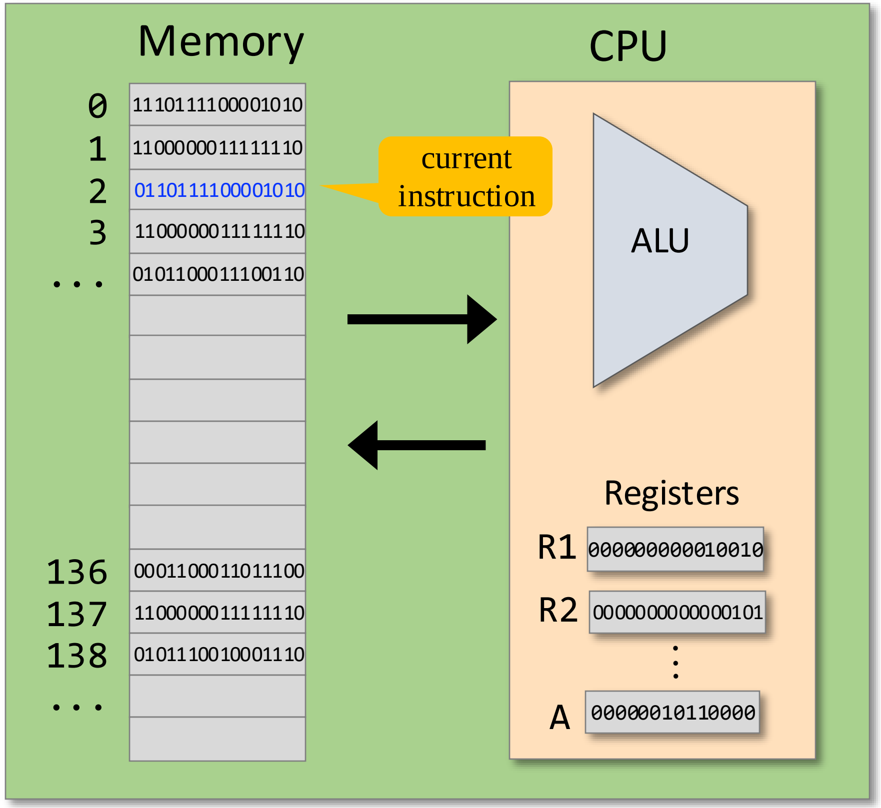
<u>The syntax</u> of machine languages varies across computers

<u>The semantics</u> is the same: Manipulating registers.

Typical operations

<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



Typical operations

Branching

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

Basic version

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

- Line numbers
- Physical addresses

Symbolic version

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

- No line numbers
- Symbolic addresses

Programs with symbolic references are …

- Easier to develop
- Readable
- Relocatable.

Typical operations

Conditional branching

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

Symbolic program

```
  // Sets R2 to abs(R1)
  // R2 ← R1
  mov R2,R1
  // if (R2 > 0) goto cont
  jgt R2,CONT
  // R2 ← -R1
  movi R2,0
  sub R2,R1
CONT:
  // Here R2 = abs(R1)
  ...
```

Program translation

Translation

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

Symbolic program

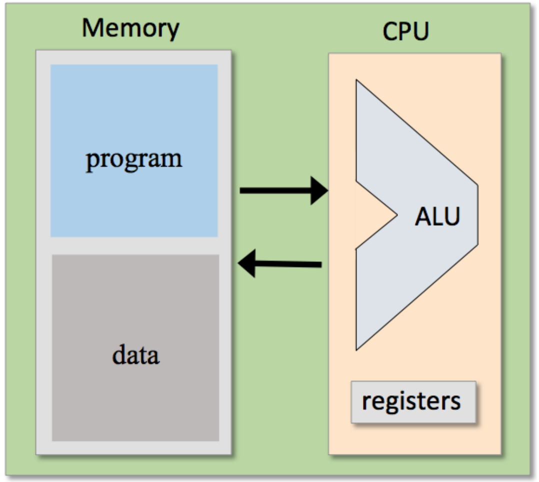
```
  // Sets R2 to abs(R1)
  // R2 ← R1
  mov R2,R1
  // if (R2 > 0) goto cont
  jgt R2,CONT
  // R2 ← −R1
  movi R2,0
  sub R2,R1
CONT:
  // Here R2 = abs(R1)
  ...
```

translate

Binary code

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

load and execute



Assembly (language)

Assembler (tool)

Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

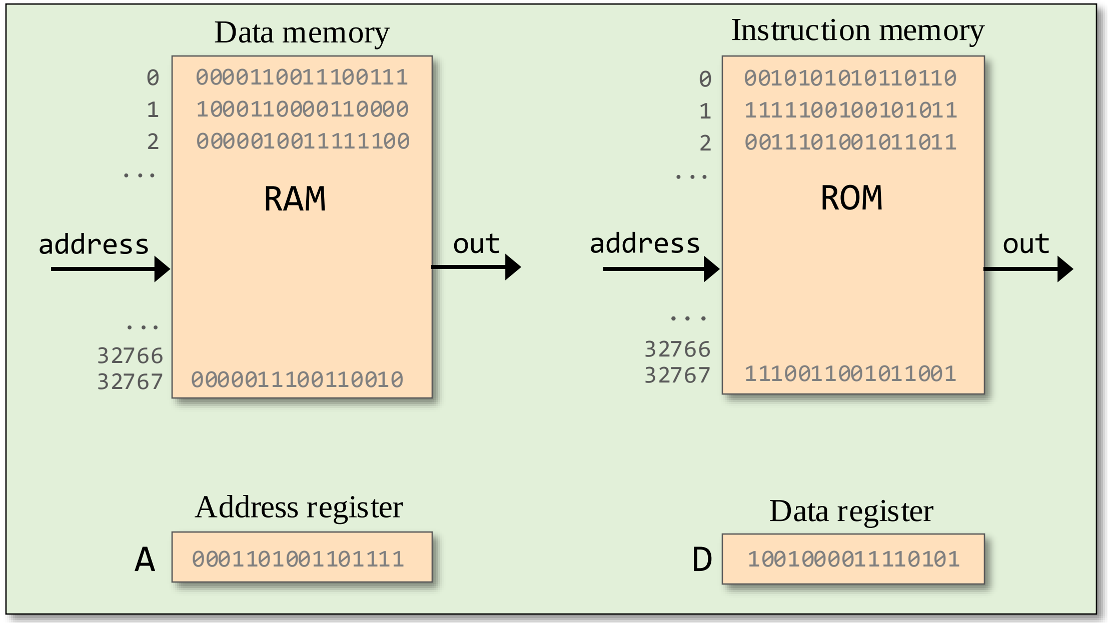
Programming examples

- Basic
- Iteration
- Pointers

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

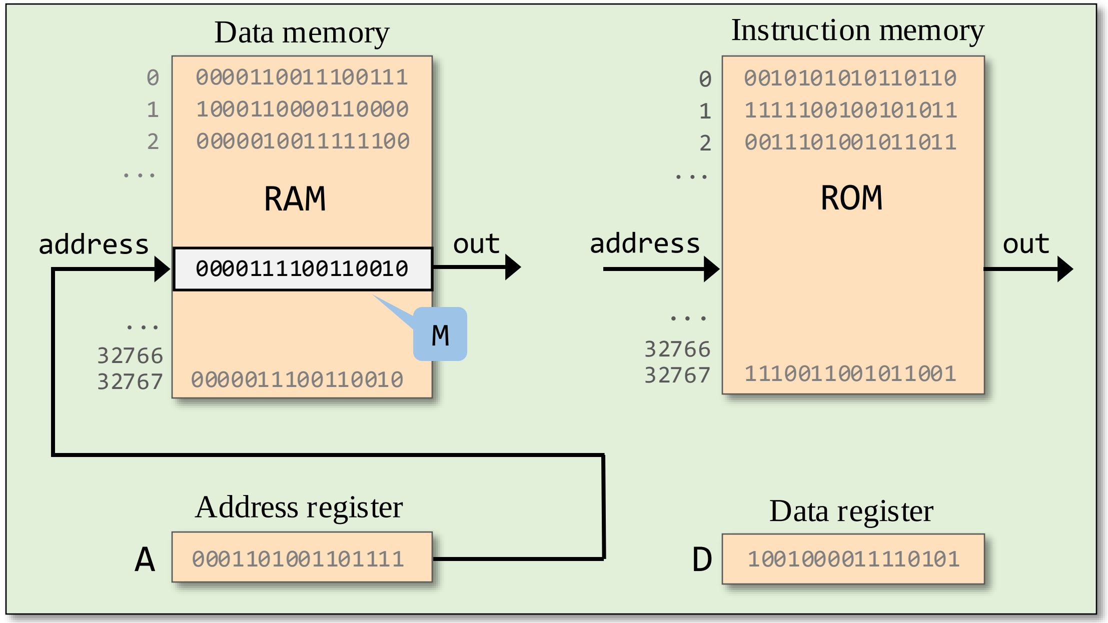
The Hack computer



(Conceptual, partial view of the Hack computer architecture)

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

Memory

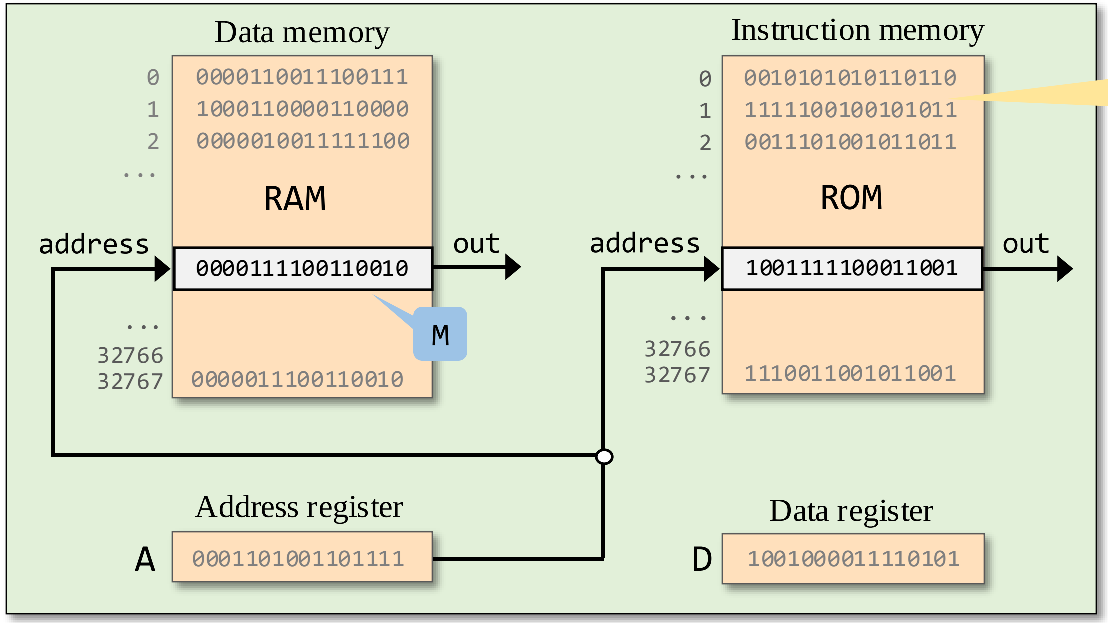


(Conceptual, partial view of the Hack computer architecture)

<u>RAM</u>

- Read-write data memory
- Addressed by the `A` register
- The selected memory location, `RAM[A]`, is referred to as `M`

Memory



Loaded with a sequence of 16-bit Hack instructions

(Conceptual, partial view of the Hack computer architecture)

RAM

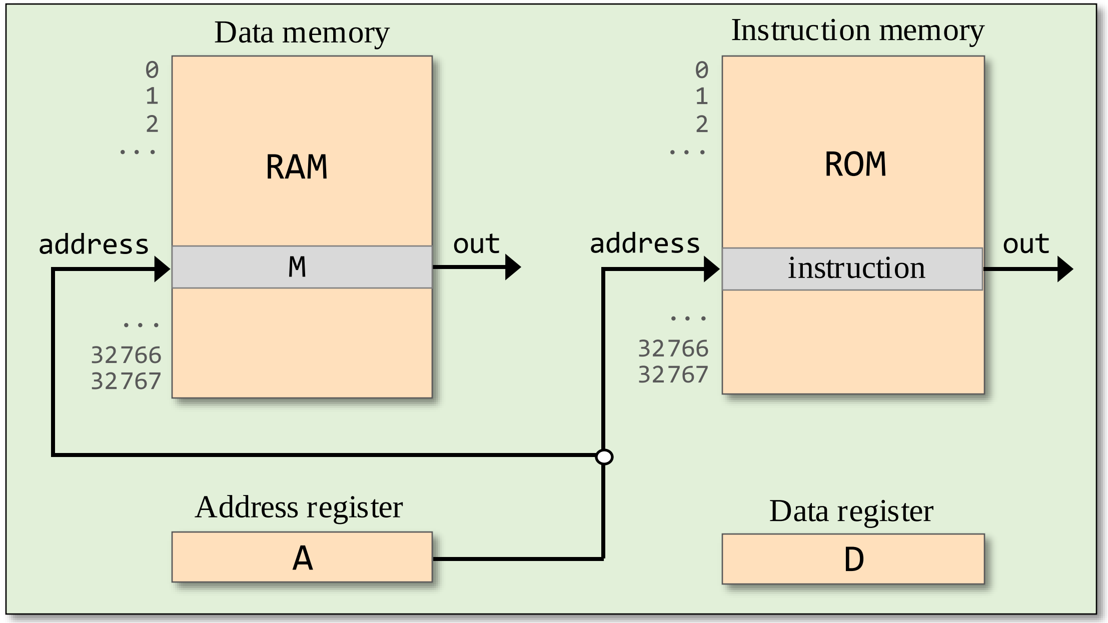
- Read-write data memory
- Addressed by the `A` register
- The selected memory location, `RAM[A]`, is referred to as `M`

ROM

- Read-only instruction memory
- Addressed by the (same) `A` register
- The selected memory location, `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`: selected RAM register

Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

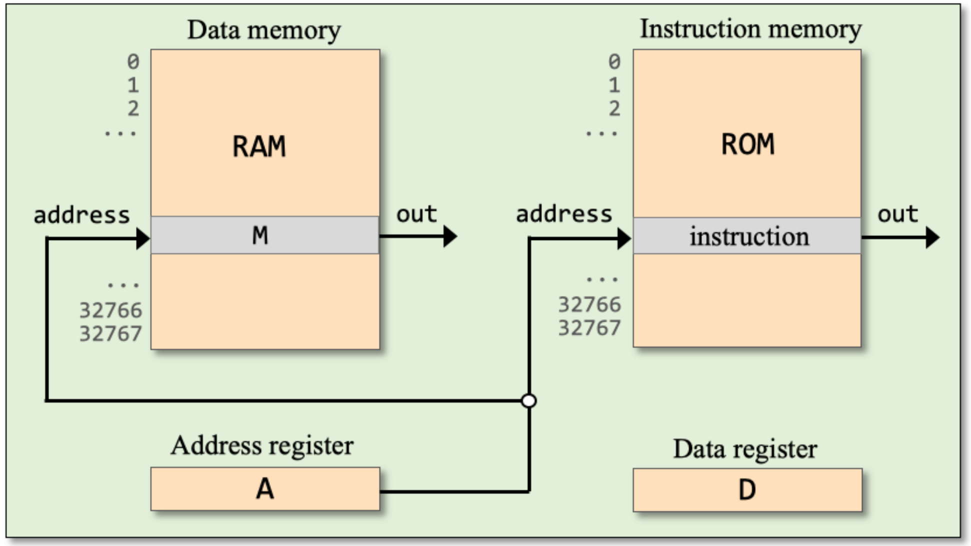
Programming examples

- Basic
- Iteration
- Pointers

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

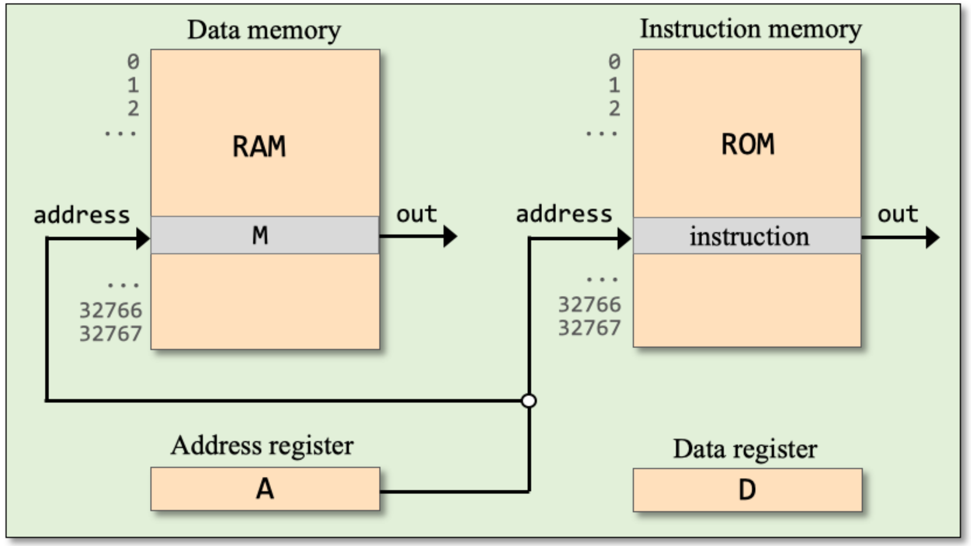
Hack instructions



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`

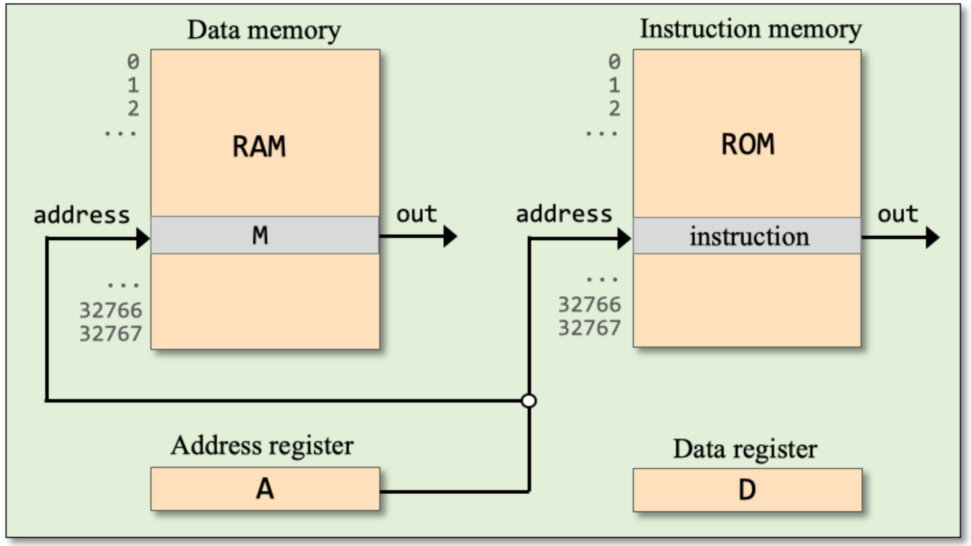
Semantics

A ← 19

Side effects:

- `RAM[A]` (denoted `M`) becomes selected
- `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 = {+|-|&|I}

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

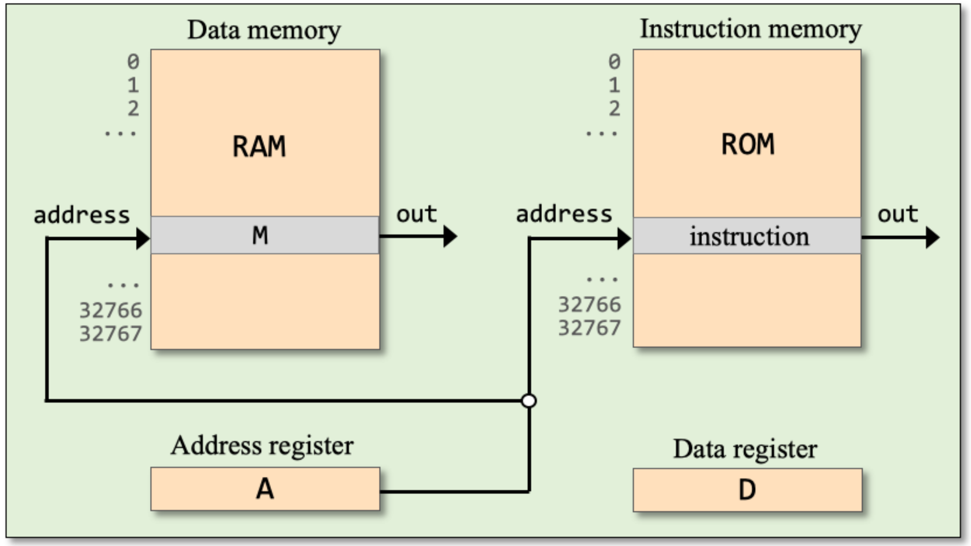
Typical instructions:

@*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
?
```

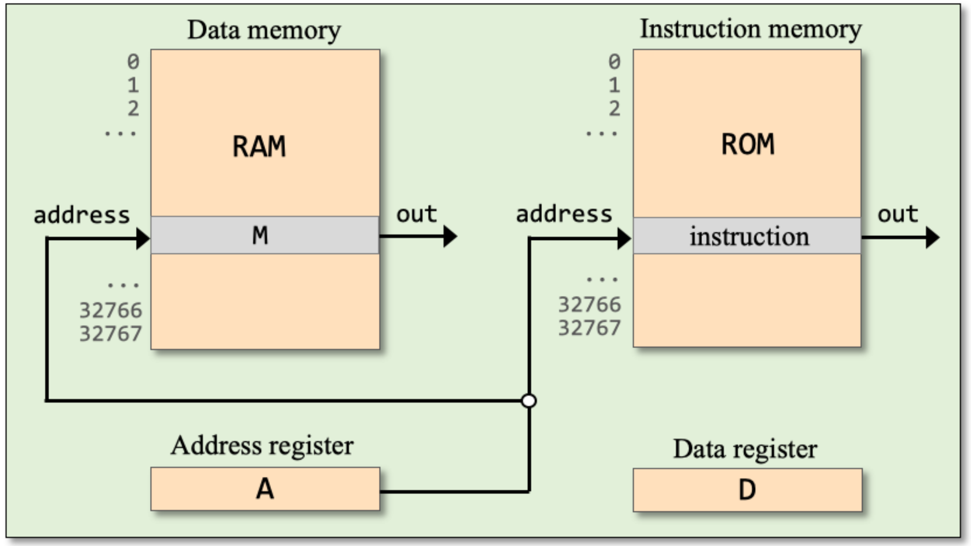
The game: We show a subset of Hack instructions (top left), and practice writing code examples that use these instructions.

Hack instructions

Typical instructions:

@*constant*	(A ← *constant*)

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



Use only the above instructions

Examples:

```
// D ← 2
D=1
D=D+1
```

```
// D ← 1954
?
```

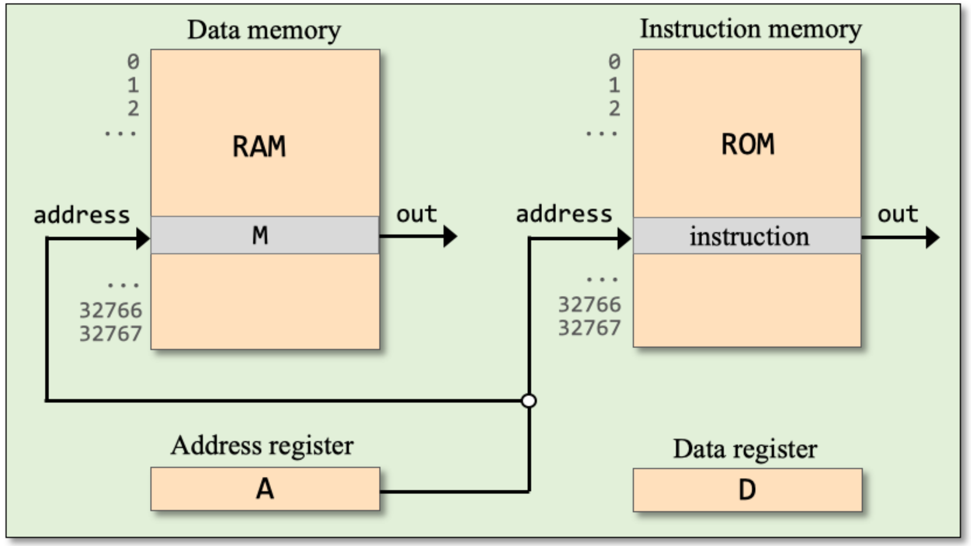
Use only the instructions shown above

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 above

Hack instructions

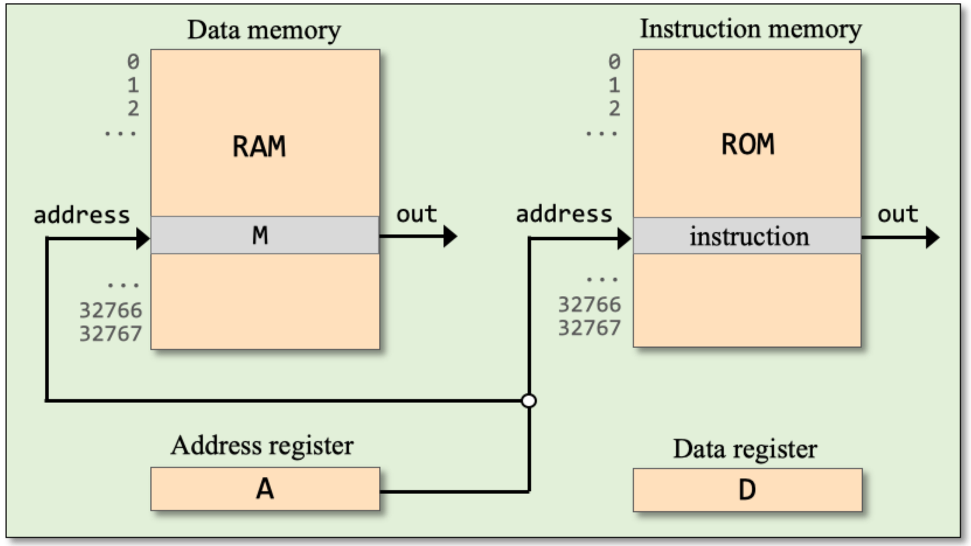
Typical instructions:

@*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 all these examples, we used both `D` and `A` as a *data registers*:

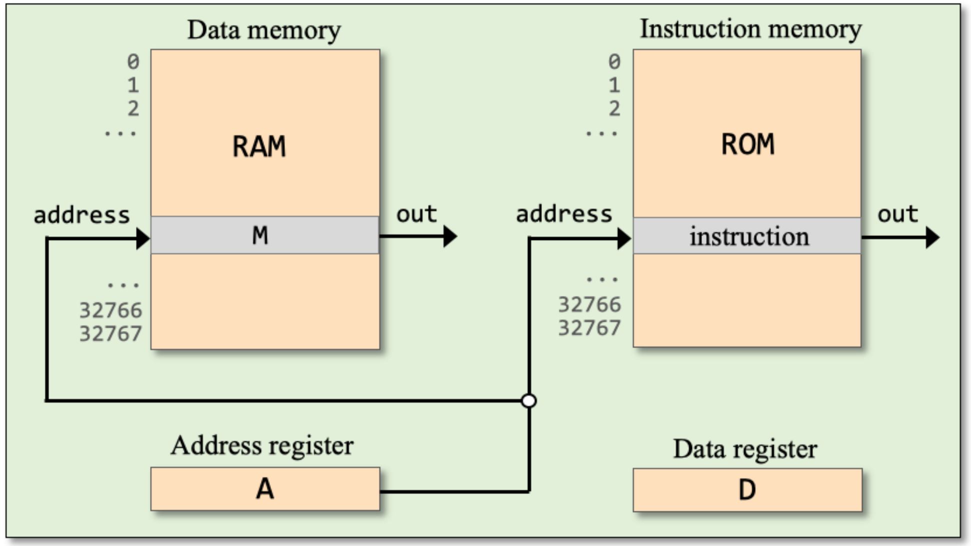
The addressing side-effects of `A` are ignored.

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

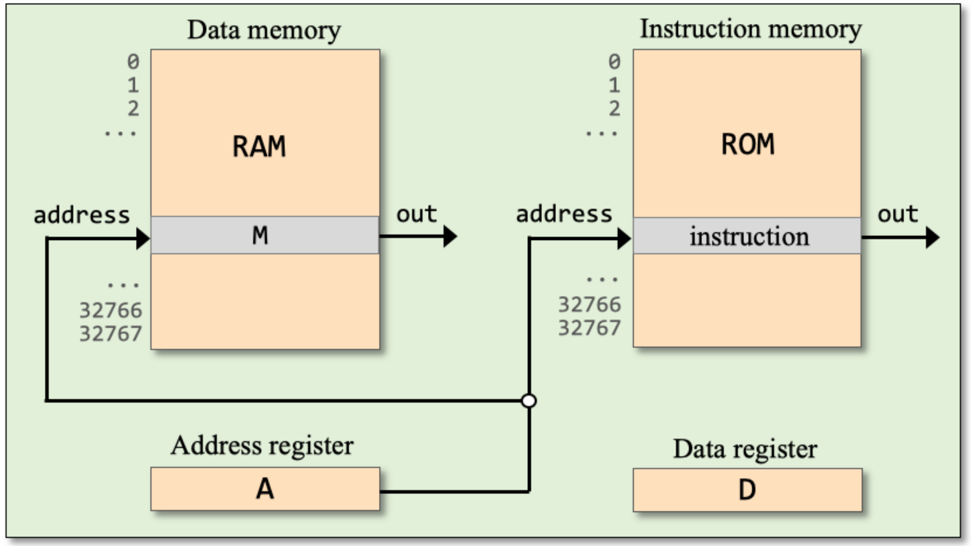
Typical instructions:

@*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

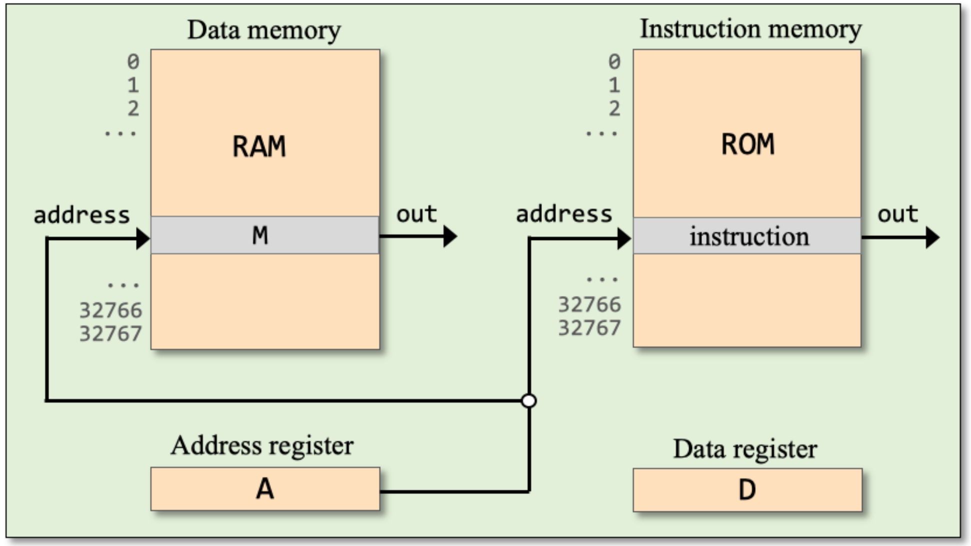
Typical instructions:

@*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
```

<u>When we want to operate on a memory location</u>, we use a pair of instructions:

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

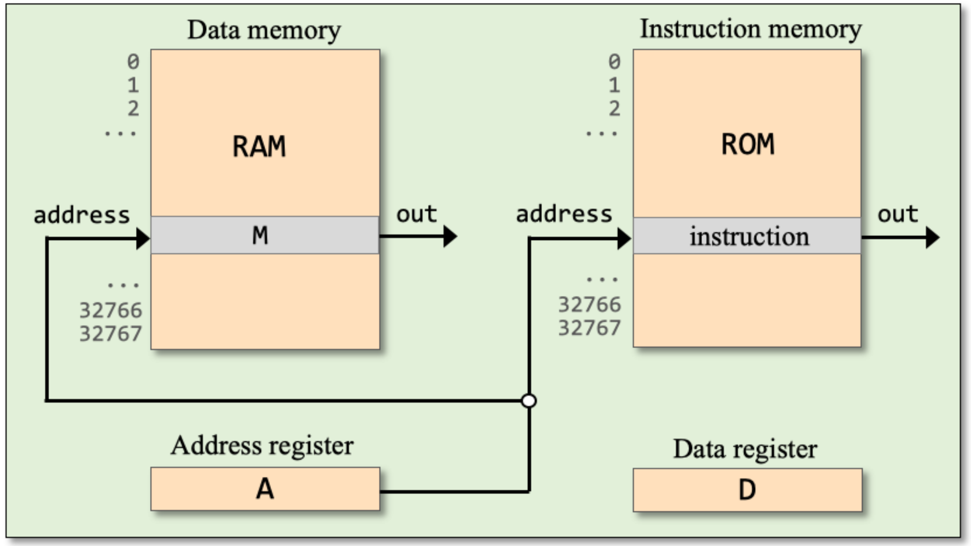
Hack instructions

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 ...

Use only the above instructions




```
// RAM[3] ← RAM[3] – 15
```

?

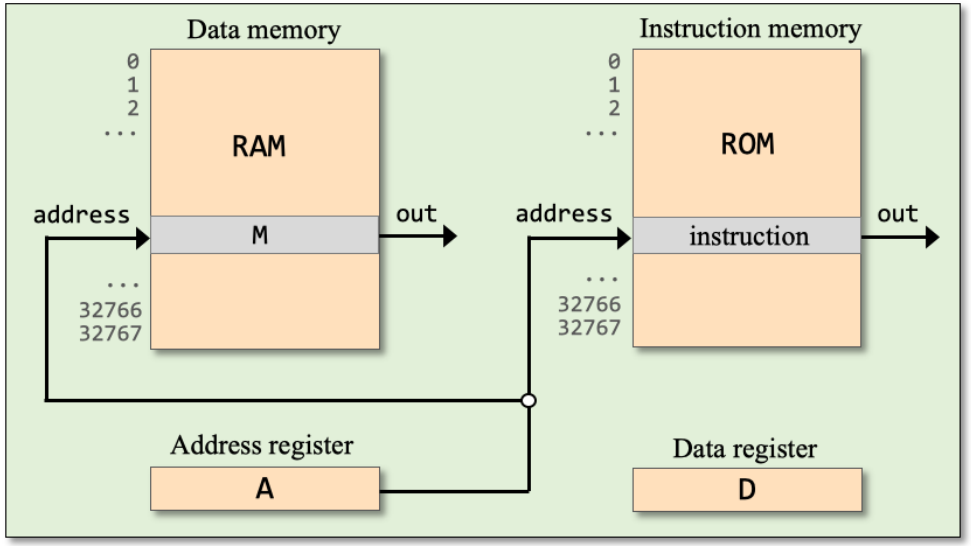
Hack instructions

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
...	...	...

Use only the above instructions




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

```
// RAM[3] ← RAM[4] + 1
?
```

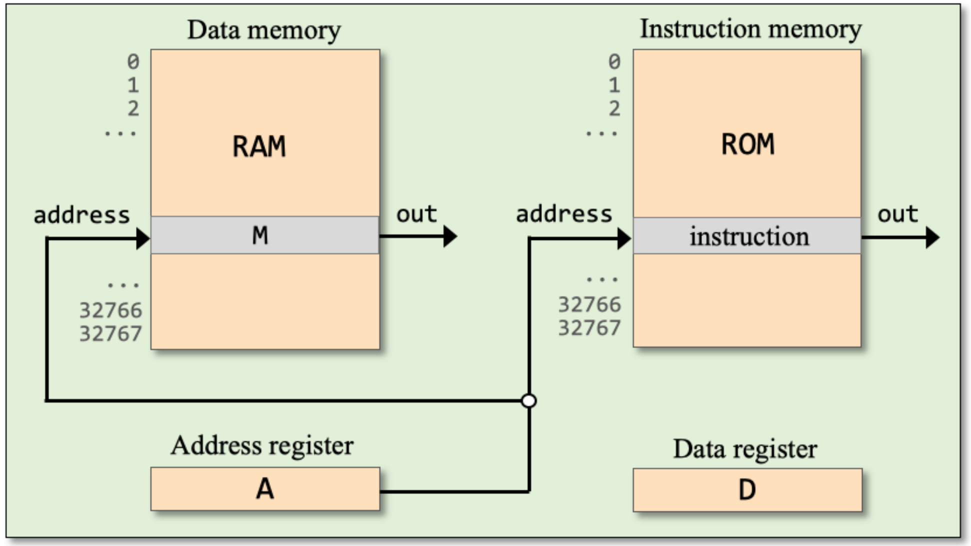
Hack instructions

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
...	...	...

Use only the above instructions




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

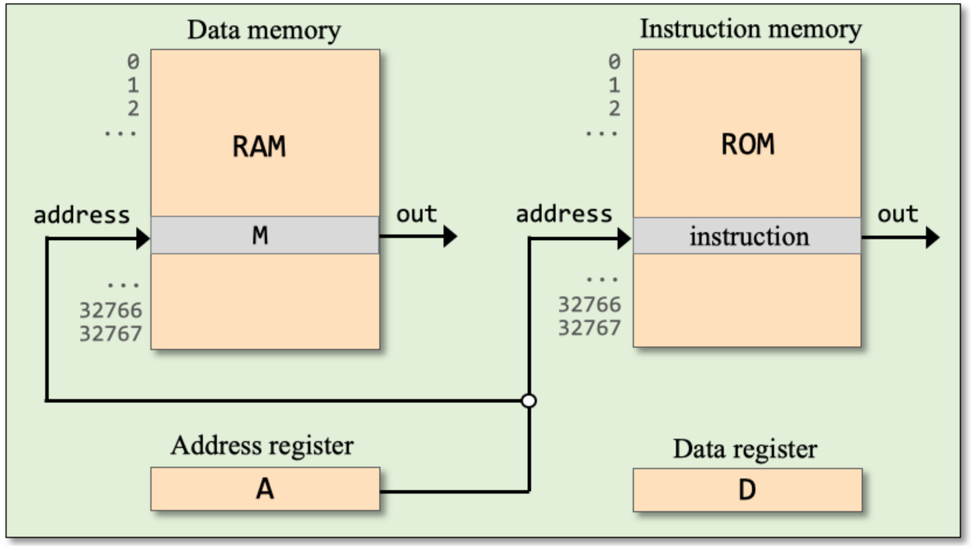
```
// RAM[3] ← RAM[4] + 1
@4
D=M+1
@3
M=D
```

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 above instructions

Hack instructions

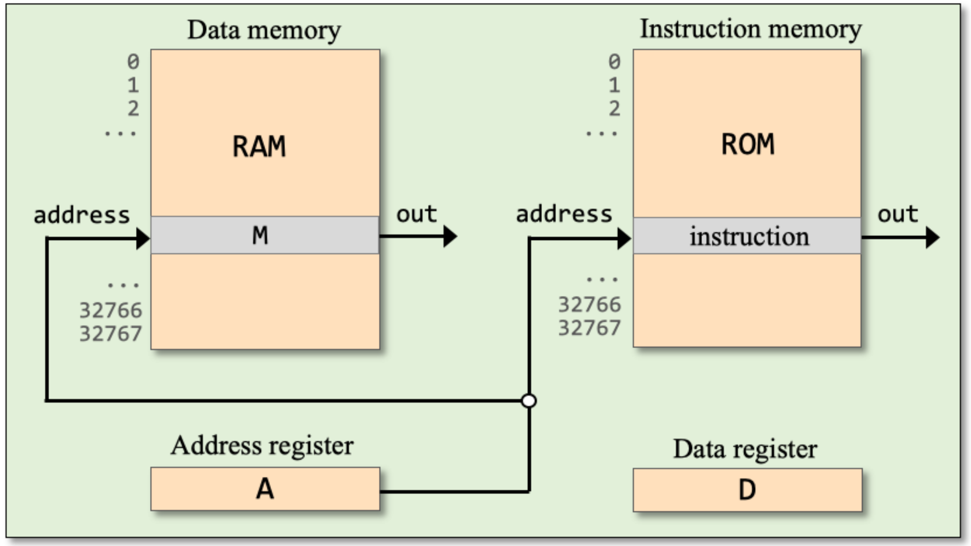
Typical instructions:

`@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
```

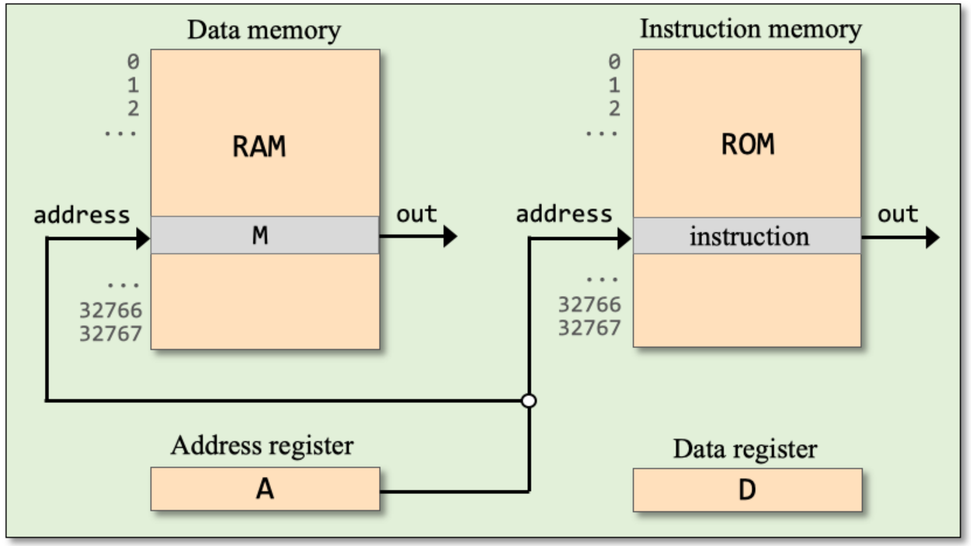
Use only the above instructions

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
```

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

→ 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

Programming examples

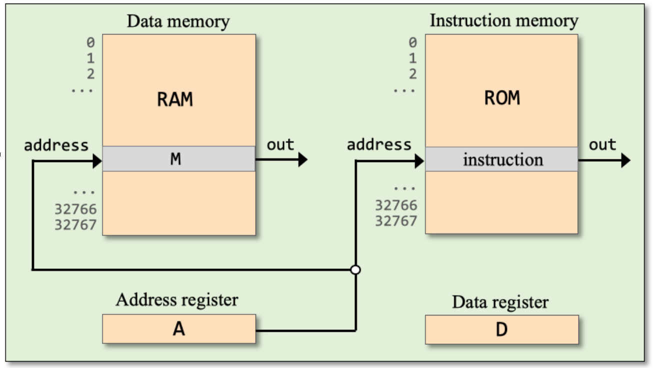
- Basic
- Iteration
- Pointers

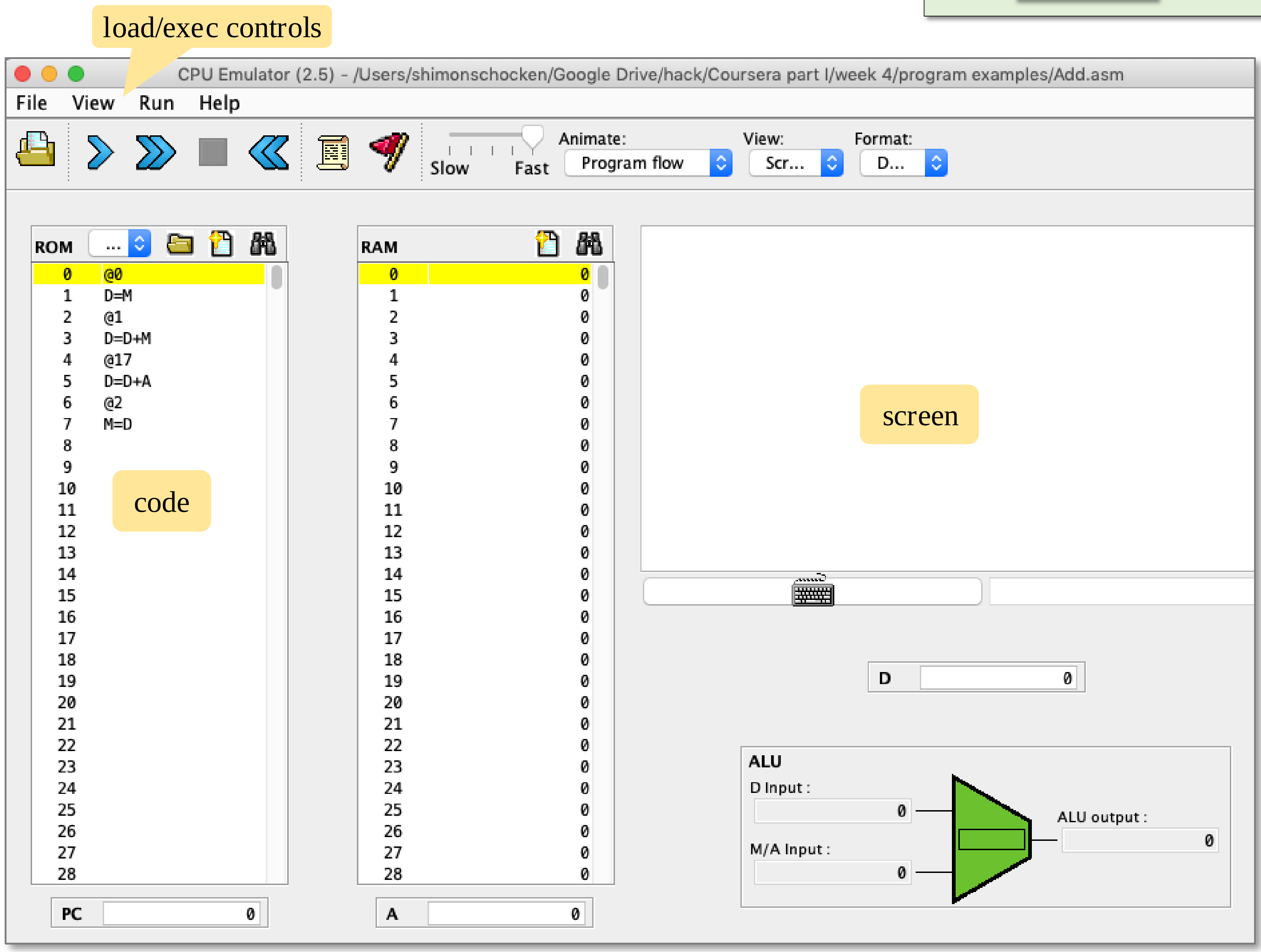
The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

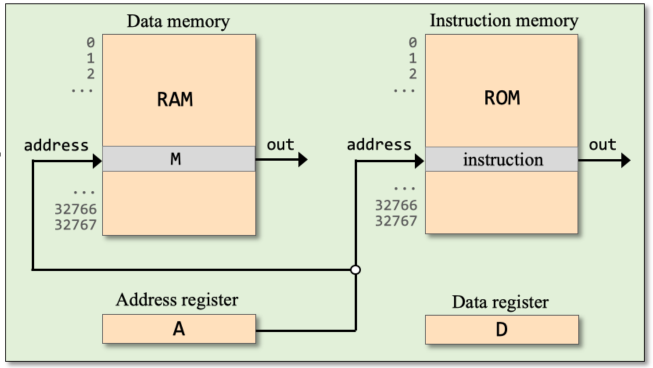
The CPU emulator

- Software that emulates the Hack CPU
- Part of the Nand to Tetris IDE





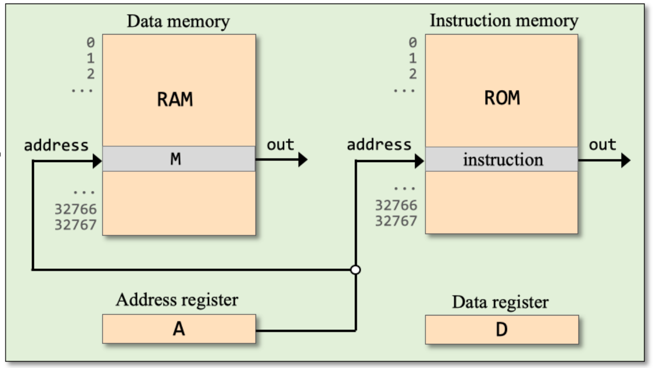
The CPU emulator



`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
```

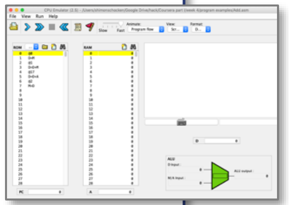
The CPU emulator



`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



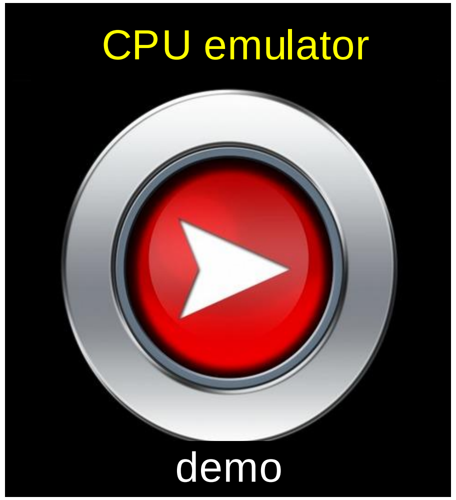
Binary

```
0000000000000000
1000010010001101
0000000000000001
1010011001100001
0000000000010001
1001111100110011
0000000000000010
1110010010010011
```

Execute 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

- Control
- Variables
- Labels

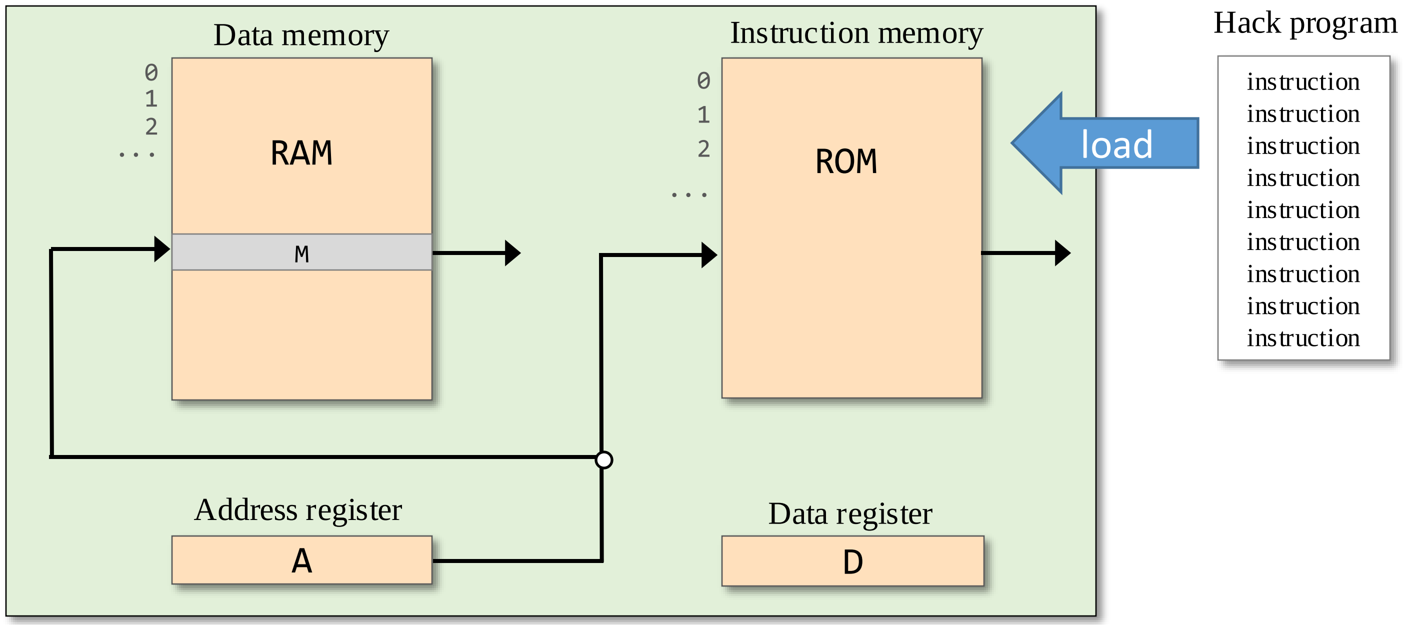
Programming examples

- Basic
- Iteration
- Pointers

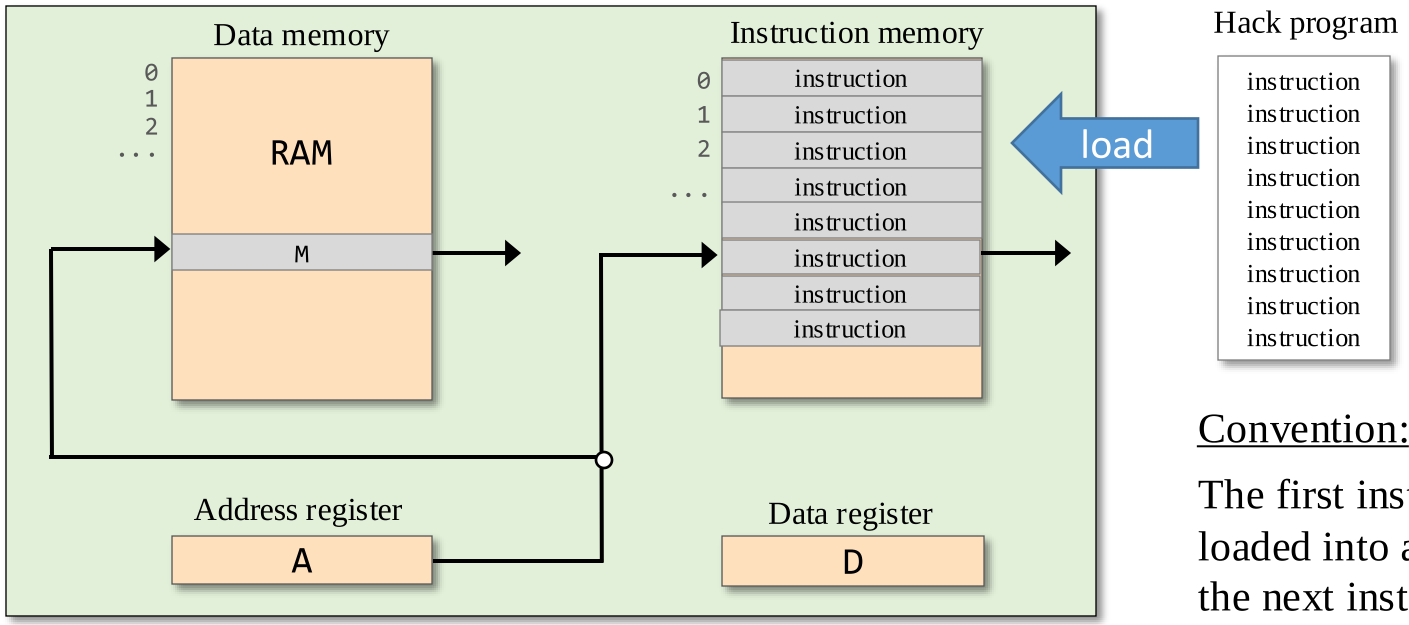
The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Loading a program



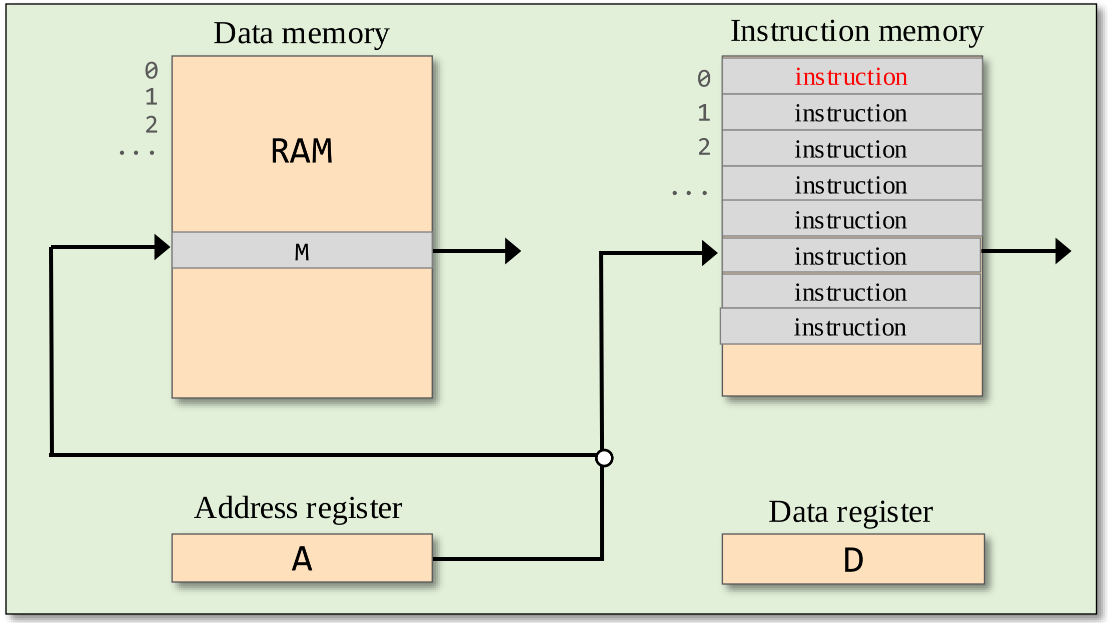
Loading a program



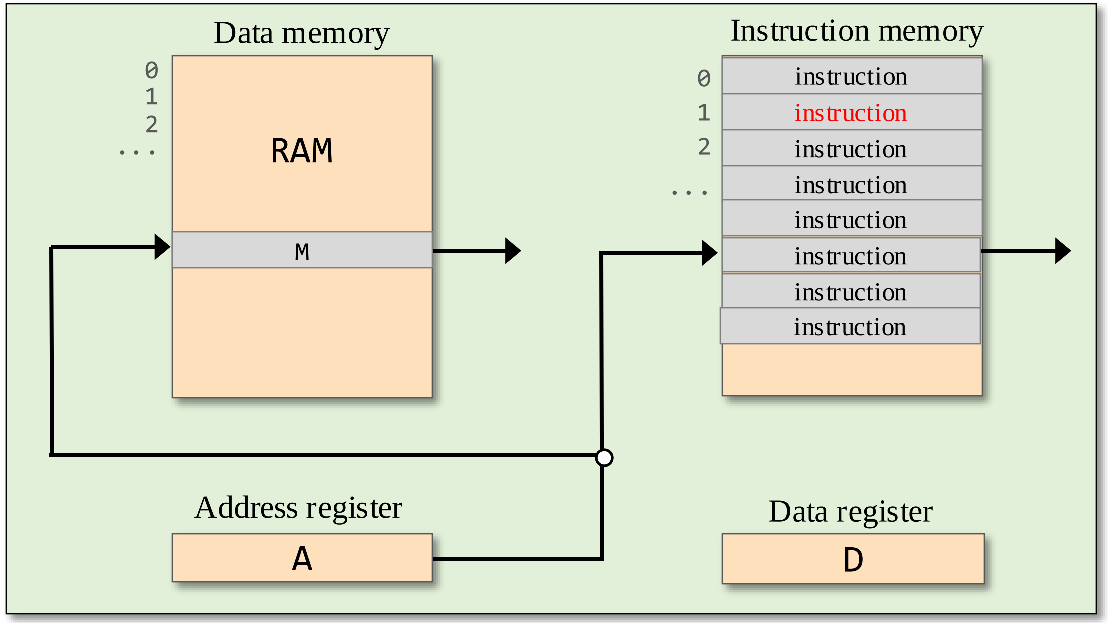
<u>Convention:</u>

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

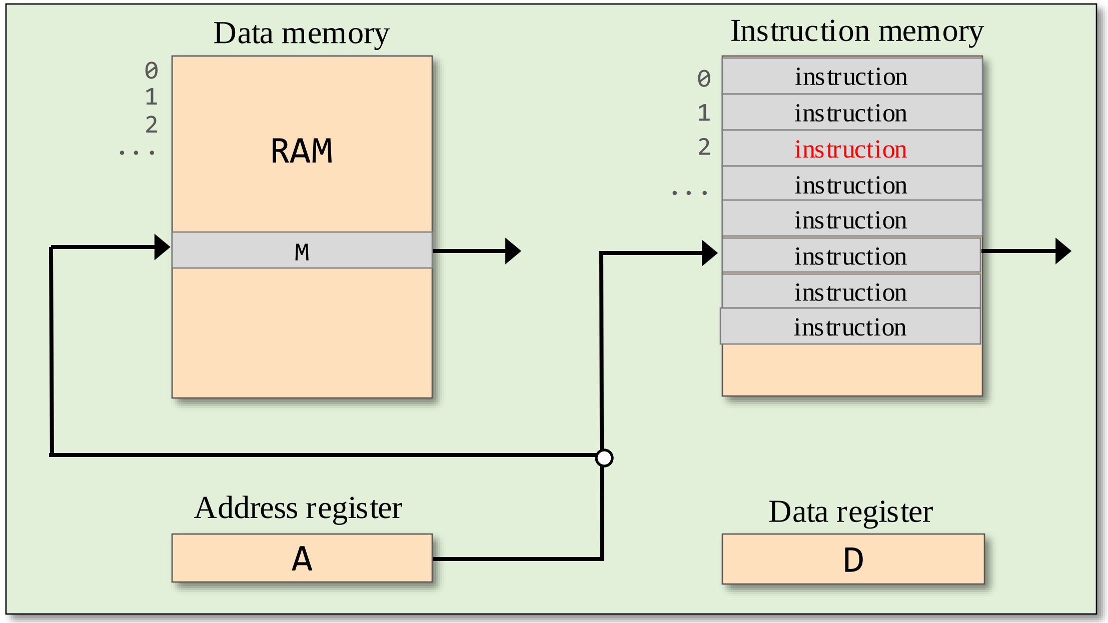
Executing a program



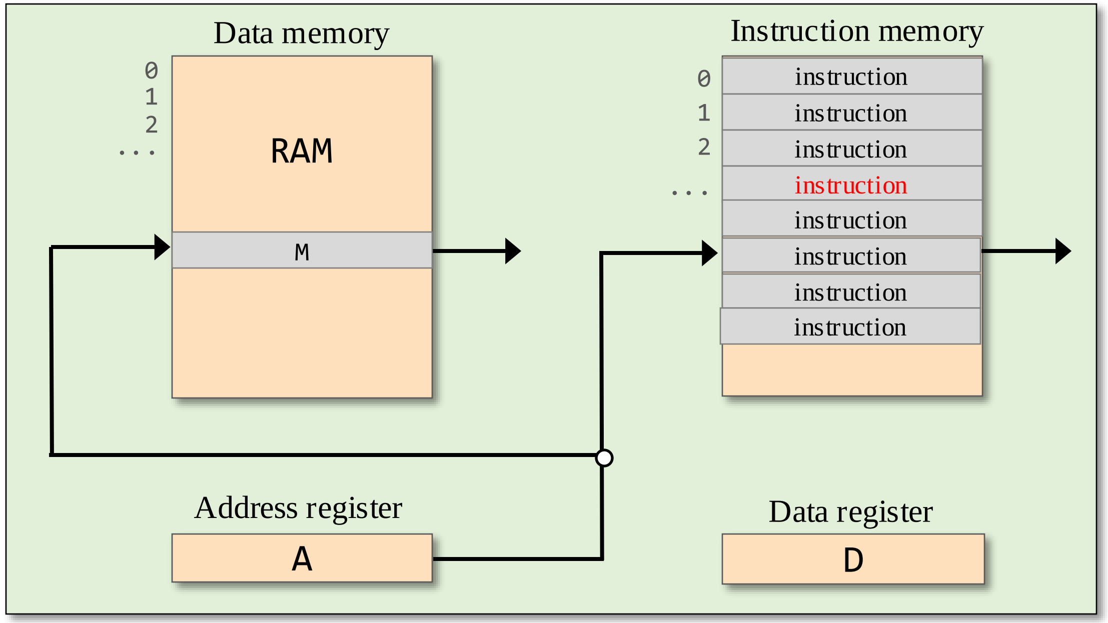
Executing a program



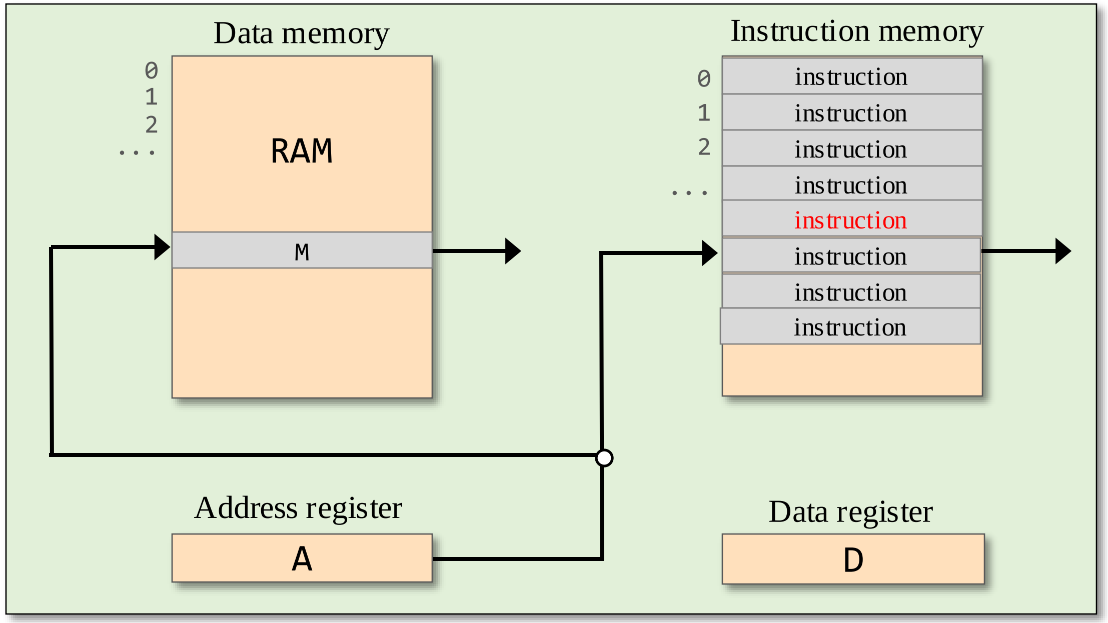
Executing a program



Executing a program



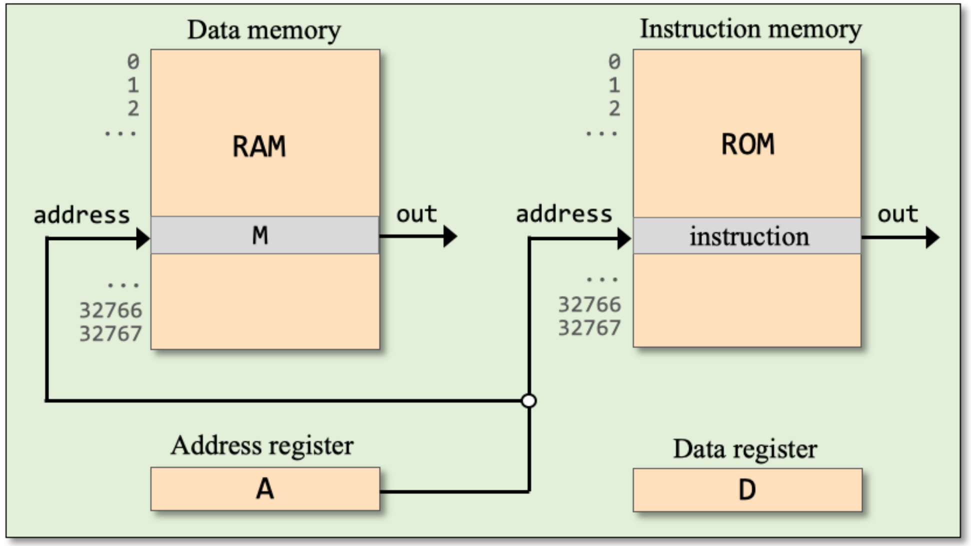
Executing a program



- The default: Execute the next instruction
- Suppose we wish to execute another instruction;

  How to specify *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,

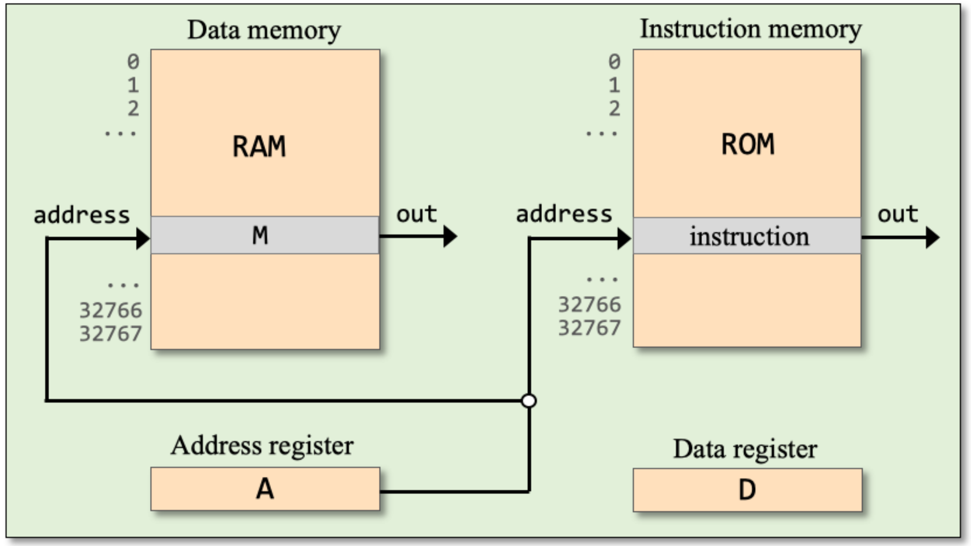
2,3,4,

7,8,9,

2,3,4,

...

Branching



Unconditional branching 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  ...
```

In Hack:

```
...
// goto 7
@7
0;JMP
...
```

Syntax:

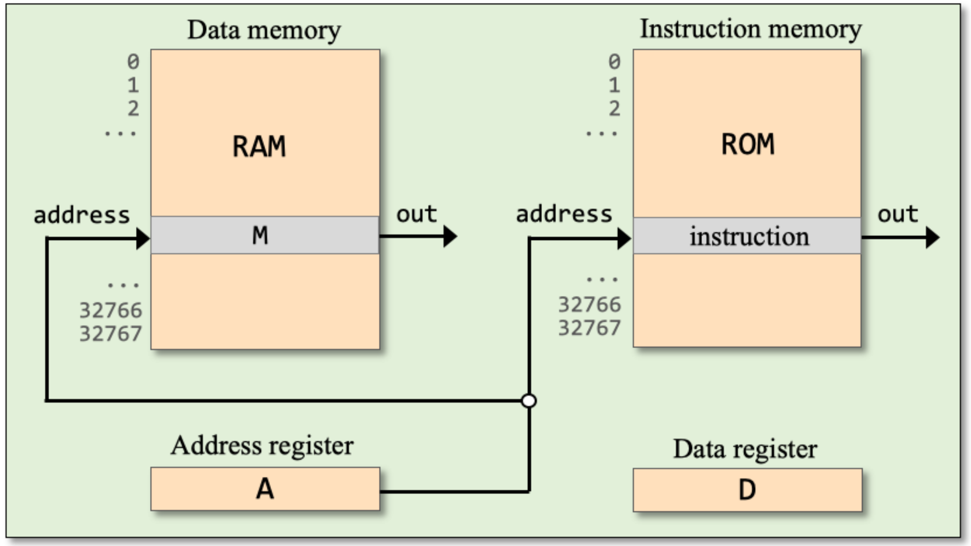
- Use an `A`-instruction to select an address
- Use a `C`-instruction to jump to that address

Semantics of `0;JMP`

Jump to execute the instruction stored in `ROM[A]`

(the `0;` prefix is a syntax convention)

Branching



Conditional branching example

Pseudocode

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

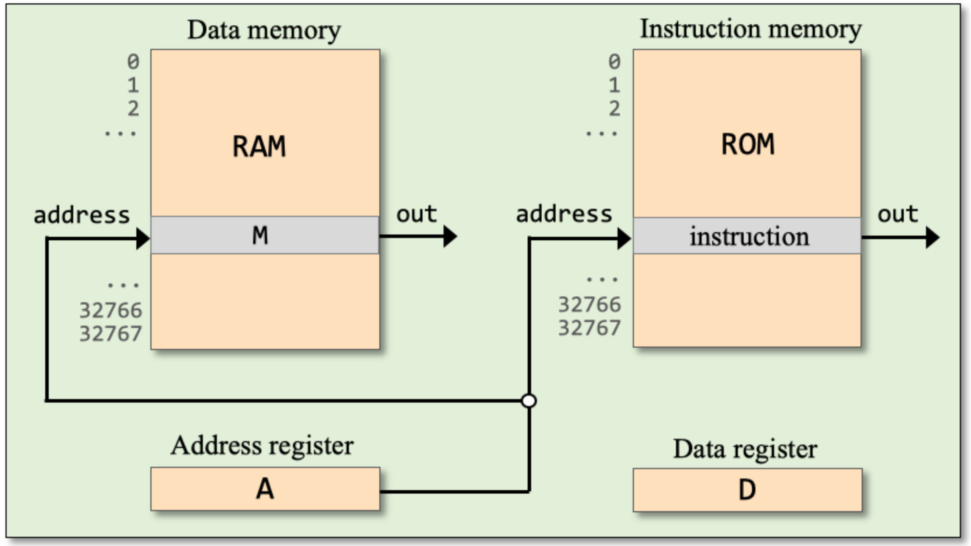
In Hack

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

Typical branching instructions:

`D;JGT` // if D > 0 jump to the instruction stored in `ROM[A]`

Branching



Conditional branching
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 ALU computation: `D+1`, `D-1`, etc.

Branching

Typical instructions:

@*constant* (A←*constant*)

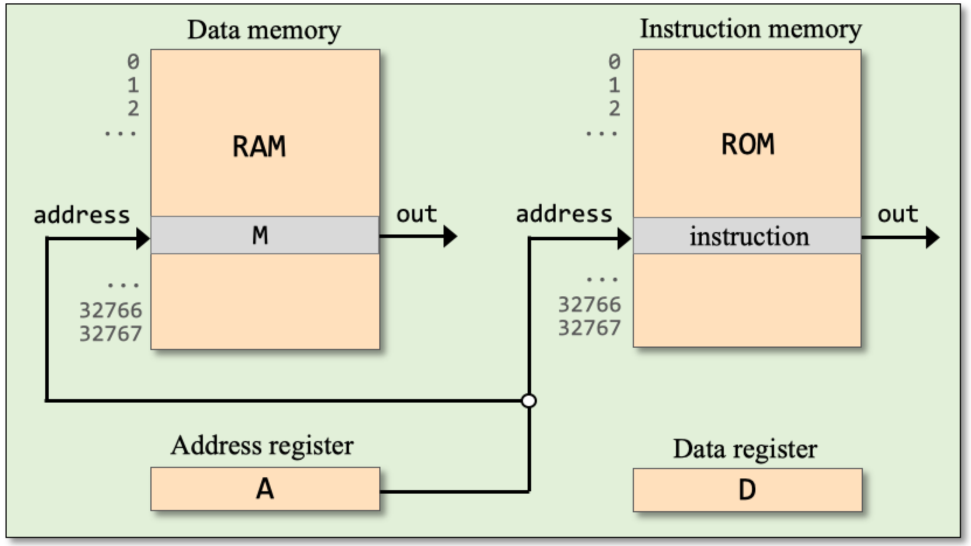
```
A=1
D=-1
M=0
...
```

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

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

Use only the above instructions

```
// if (D = 0) goto 300
?
```



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*)

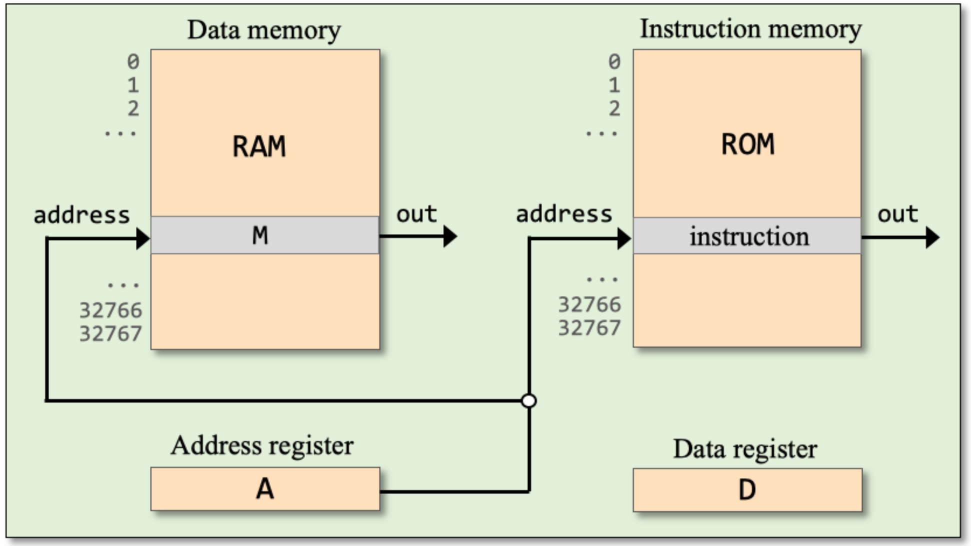
```
A=1
D=-1
M=0
...
```

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

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

Use only the above instructions

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



<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:

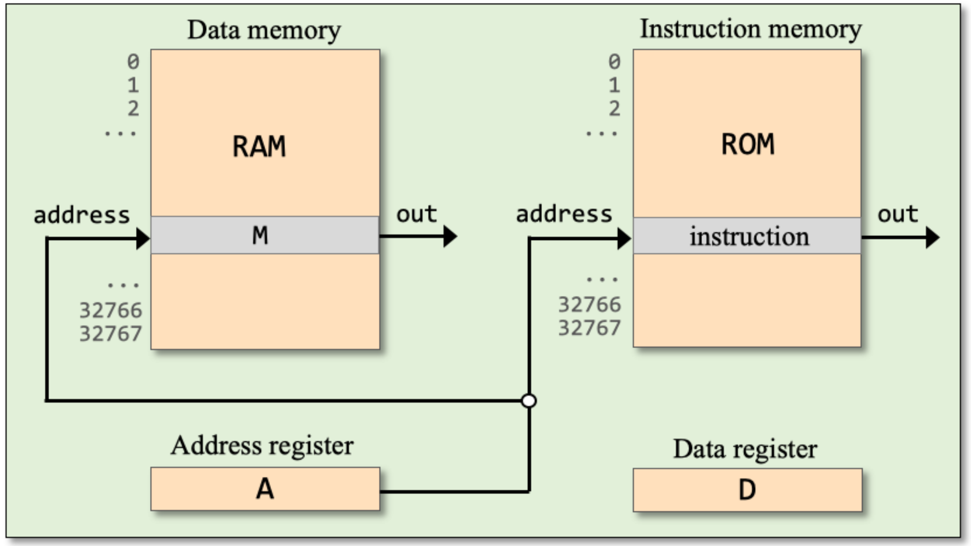
@*constant* (A←*constant*)

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

Use only the above instructions

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

?



<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

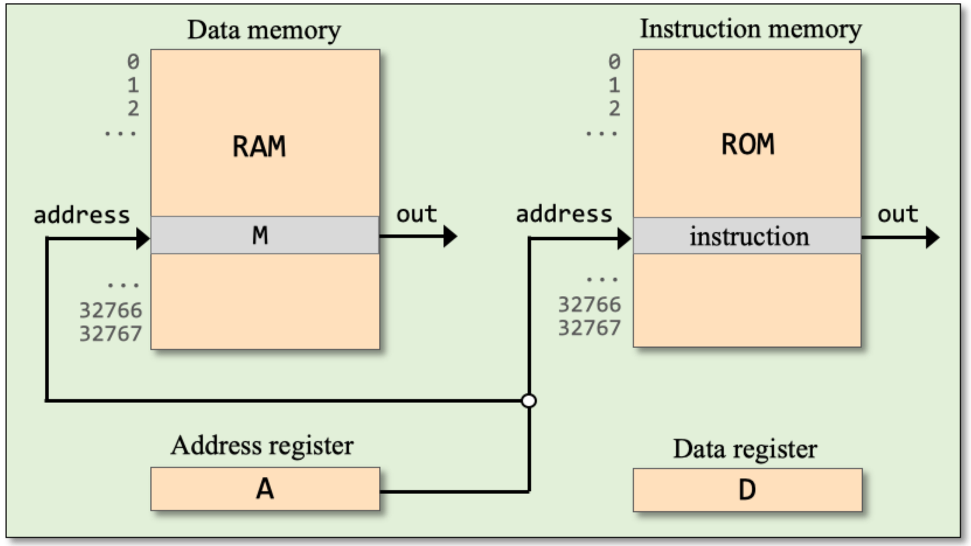
Typical instructions:

@*constant*	(A←*constant*)

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

Use only the above instructions

```
// 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]

Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

Programming examples

- Basic
- Iteration
- Pointers

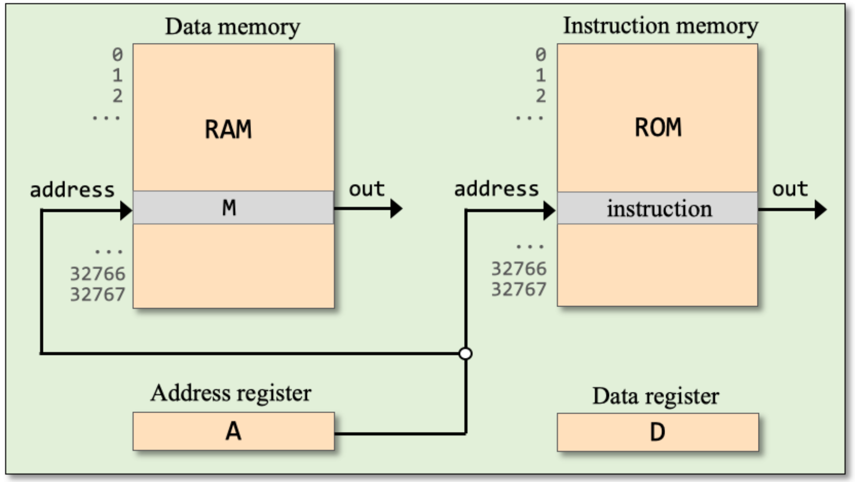
The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

The A-instruction

A-instruction

C-instruction



Syntax:

`@xxx`

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

Examples:	Semantics:
`@19`	A ← 19
`@sym`	A ← 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
```

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
...
```

<u>Symbolic programming</u>

- 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*

<u>For example</u>

- If the assembler will bind `i` and `sum` to, say, `16` and `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
...
```

Use only the above instructions

```
// sum = 0
?
```

```
// x = 512
?
```

```
// n = n - 1
?
```

```
// sum = sum + x
?
```

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
...
```

Use only the above instructions

```
// sum = 0
@sum
M=0
```

```
// x = 512
@512
D=A
@x
M=D
```

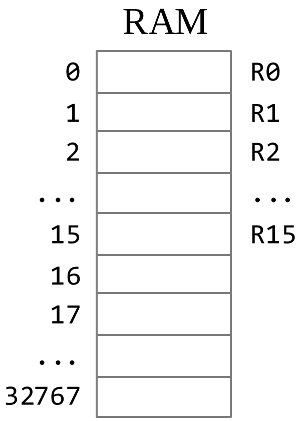
```
// n = n - 1
@n
M=M-1
```

```
// sum = sum + x
@sum
D=M
@x
D=D+M
@sum
M=D
```

Variables

Pre-defined symbols in the Hack language

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



16 "built-in variables" named `R0`…`R15`

Sometimes referred to as "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 more readable.

Machine Language

Overview

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

Symbolic programming

- ✓ Control
- ✓ Variables
- ➡ Labels

Programming examples

- 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:
   ...
```

write

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)
   ...
```

<u>Label declaration in the Hack assembly language:</u>

(*sym*)

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

<u>In this example:</u>

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

Programming examples

- 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:
```

`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, 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)
```

The assembler replaces all the symbols with physical addresses

Assembler / loader

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

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 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	
11	
12	
13	
14	
...	

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)
```

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 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)
```

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
```

Infinite loop

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	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
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)
   @END
   0;JMP
```

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	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
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)
   @END
   0;JMP
```

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	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
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)
     @END
     0;JMP
```

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	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
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)
   @END
   0;JMP
```

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	@10
11	0;JMP
12	0100100110011011
13	1110010011111111
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)
   @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.

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:
```

<u>Best practice</u>

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

Programming examples

- ✓ Basic
- ➡ Iteration
- Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Symbolic
- Binary
- 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
```

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
```

Machine Language

Overview

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

Programming examples

✓ Basic

✓ Iteration

➡ Pointers

Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Symbolic
- Binary
- Output
- Input
- Project 4

Pointers

<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
```

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	R0
1		R1
2		R2
...		...
15		R15
16		
17		
...		
255		
256		
...		
1012		
1013		
1014		
1015	-1	desired result
1016		
...		

example:
addr = 1015

Pointers

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	R0	
1	75	R1	desired result
2		R2	
...		...	
15		R15	
16			
17			
...			
255			
256			
...			
1012	512		
1013	75		
1014	19		
1015	-17		
1016	256		
...			

example:
addr = 1013

Pointers

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	R0	
1	75	R1	desired result
2		R2	
...		...	
15		R15	
16			
17			
...			
255			
256			
...			
1012	512		
1013	75		
1014	19		
1015	-17		
1016	256		
...			

example:
addr = 1013

Pointers

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

example:
base = 300
n = 5

Pointers

<u>Example 3</u>: Set the first *n* words of the memory block beginning in address *base* to `-1`

Inputs: `R0` (*base*) and `R1` (*n*)

RAM

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

example:
base = 300
n = 5

Pointers

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

Inputs: `R0` (*base*) and `R1` (*n*)

RAM

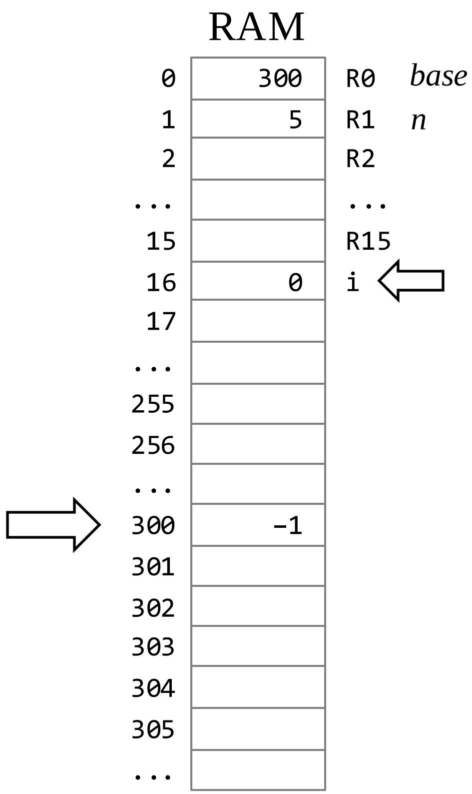
0	300	R0	*base*
1	5	R1	*n*
2		R2	
...		...	
15		R15	
16	0	i	
17			
...			
255			
256			
...			
300			
301			
302			
303			
304			
305			
...			

example:
base = 300
n = 5

Pointers

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

Inputs: `R0` (*base*) and `R1` (*n*)

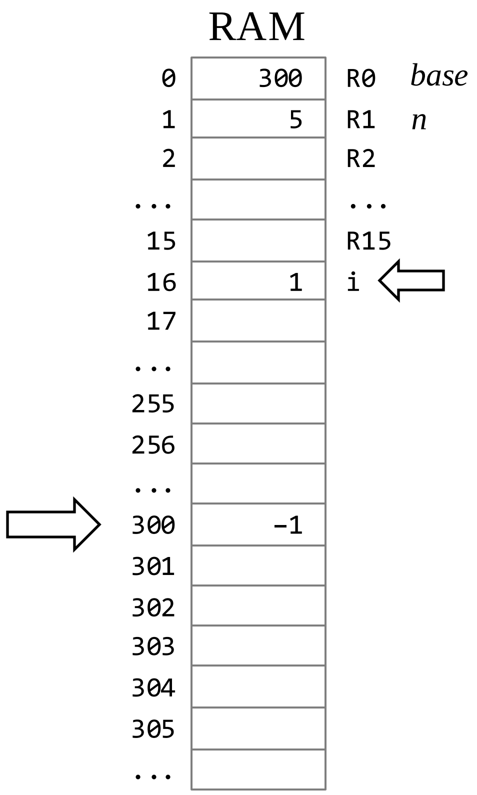


example:
base = 300
n = 5

Pointers

<u>Example 3</u>: Set the first *n* words of the memory block beginning in address *base* to `-1`

Inputs: `R0` (*base*) and `R1` (*n*)

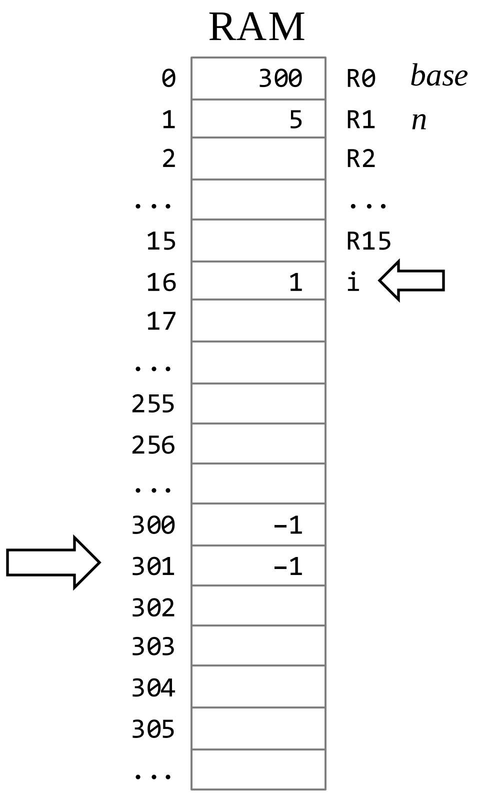


example:
base = 300
n = 5

Pointers

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

Inputs: `R0` (*base*) and `R1` (*n*)

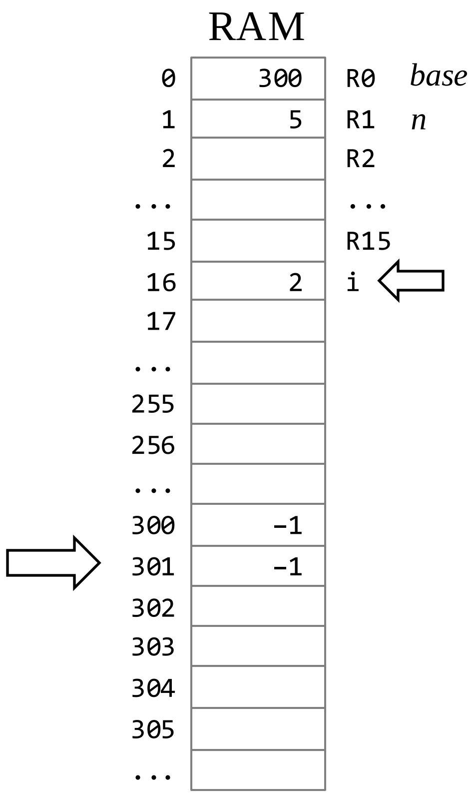


example:
base = 300
n = 5

Pointers

<u>Example 3</u>: Set the first *n* words of the memory block beginning in address *base* to `-1`

Inputs: `R0` (*base*) and `R1` (*n*)

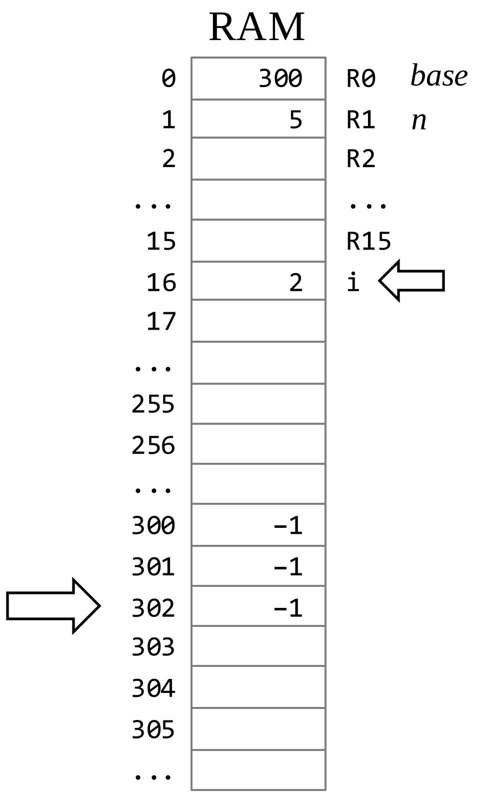


example:
base = 300
n = 5

Pointers

<u>Example 3</u>: Set the first *n* words of the memory block beginning in address *base* to `-1`

Inputs: `R0` (*base*) and `R1` (*n*)

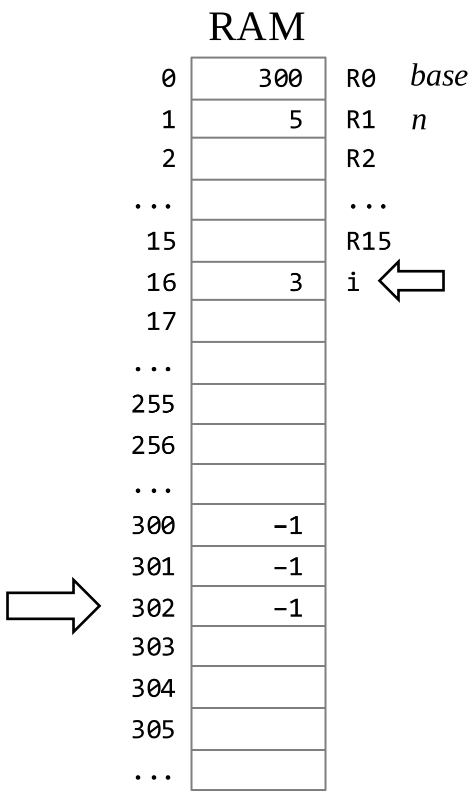


example:
base = 300
n = 5

Pointers

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

Inputs: `R0` (*base*) and `R1` (*n*)

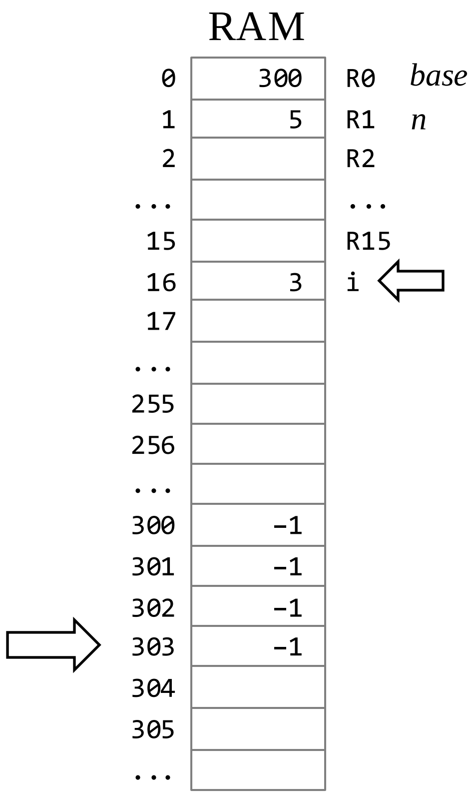


example:
base = 300
n = 5

Pointers

<u>Example 3</u>: 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

Pointers

<u>Example 3</u>: Set the first *n* words of the memory block beginning in address *base* to `-1`

Inputs: `R0` (*base*) and `R1` (*n*)

RAM

0	300	R0	*base*
1	5	R1	*n*
2		R2	
...		...	
15		R15	
16	4	i	
17			
...			
255			
256			
...			
300	-1		
301	-1		
302	-1		
303	-1		
304			
305			
...			

example:
base = 300
n = 5

Pointers

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

Inputs: `R0` (*base*) and `R1` (*n*)

RAM

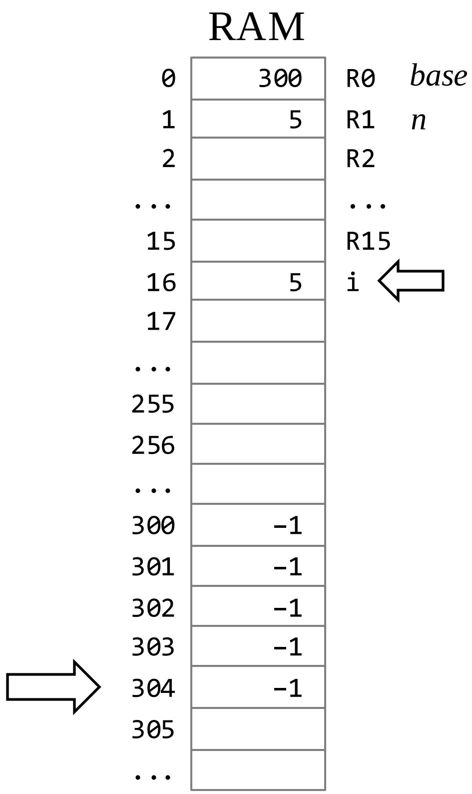
0	300	R0	*base*
1	5	R1	*n*
2		R2	
...		...	
15		R15	
16	4	i	
17			
...			
255			
256			
...			
300	-1		
301	-1		
302	-1		
303	-1		
304	-1		
305			
...			

example:
base = 300
n = 5

Pointers

<u>Example 3</u>: 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

Pointers

<u>Example 3</u>: 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:
```

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		
303	-1		
304	-1		
305			
...			

example:
base = 300
n = 5

Pointers

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

example:
base = 300
n = 5

Sneak preview to compilation: Handling arrays

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	arr
17	0	sum
...		
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	7	
5036		
...		

Sneak preview to compilation: Handling arrays

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	arr
17	359	sum
...	5	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	7	
5036		
...		

Sneak preview to compilation: Handling arrays

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];
}
...
```

Compiler

Hack assembly

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

RAM

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

Sneak preview to compilation: Handling arrays

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
}
...
```

Compiler

Hack assembly

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

RAM

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

Sneak preview to compilation: Handling arrays

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
}
...
```

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
?
```

RAM

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

Sneak preview to compilation: Handling arrays

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
}
...
```

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	359	sum
...	5	j
75		
76		
...		
255		
256		
...		
5034	100	
5035	50	
5036	200	
5037	2	
5038	7	
5036		
...		

Every high-level array access *arr*[*expression*] in any programming language 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

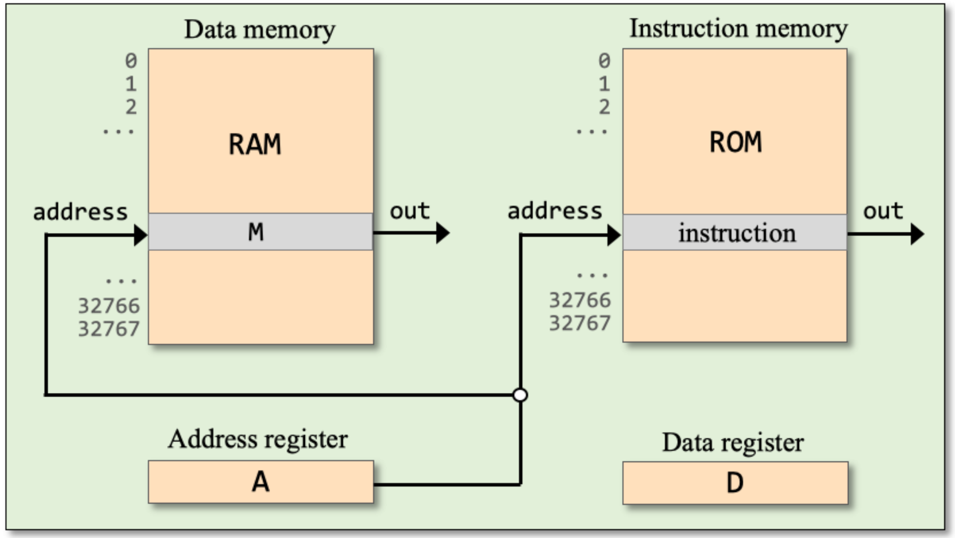
Programming examples

- 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

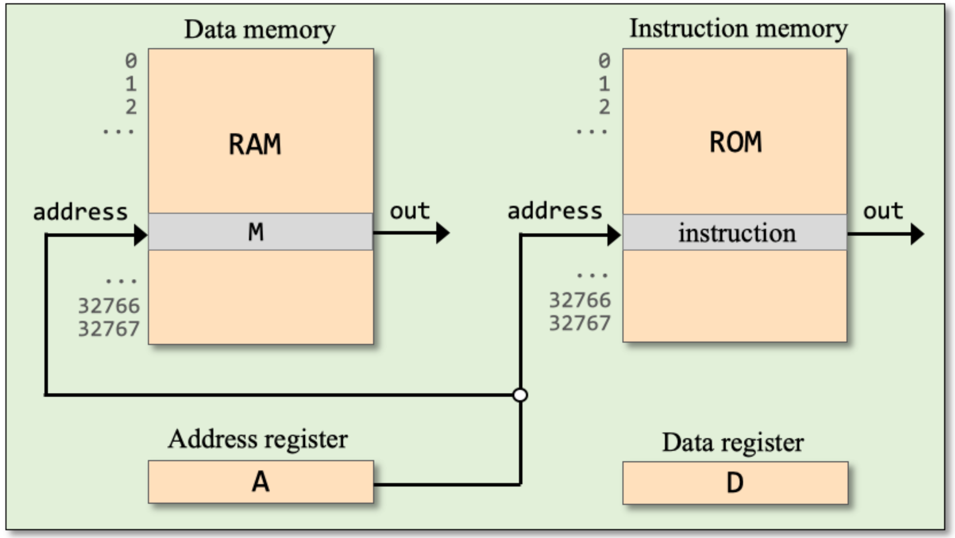
Semantics:

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

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

  `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 jump* `0`), branches to execute `ROM[A]`

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:

```
// 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 = `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:

```
// 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 = `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:

```
// 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

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

<u>Best practice rule</u>

A C-instruction should specify either `M`, or a jump directive, but not both

Syntax convention: A C-instruction that mentions M should not have
a jump directive, and vice versa

Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

Programming examples

- Basic
- Iteration
- Pointers

The Hack Language

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

Hack machine language specification

Two versions

- Symbolic
- Binary

The binary specification is not intended for writing *low-level programs*;
It is intended for writing *assemblers* (chapter 6).

We describe it here, for completeness.

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` *vvvvvvvvvvvvvvv* (*vv* ... *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: `111`*accccccdddjjj*

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

Machine Language

Overview

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

Symbolic programming

- Control
- Variables
- Labels

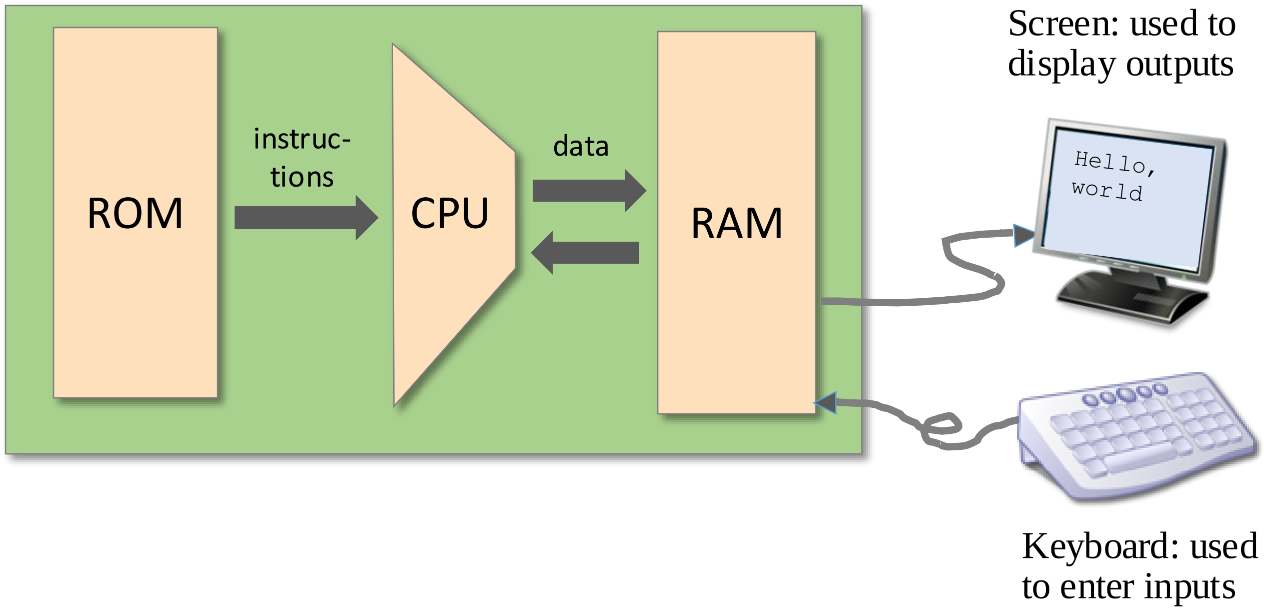
Programming examples

- Basic
- Iteration
- Pointers

The Hack Language

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

Input / output



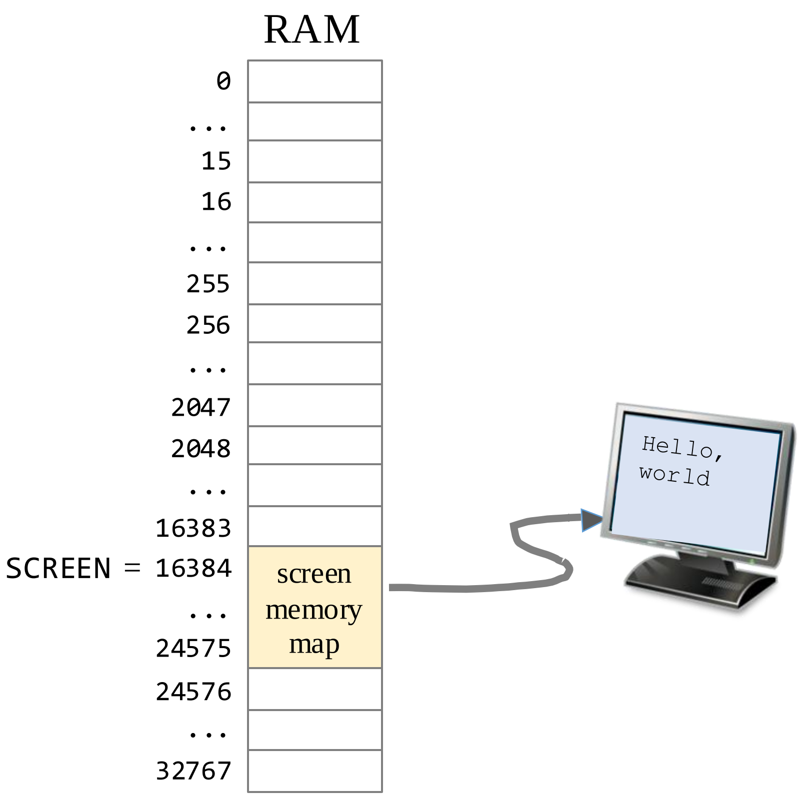
High-level I/O handling (later in the course):

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

Low-level I/O handling:

Manipulating bits directly, using memory resident *bitmaps*.

Bitmaps



<u>Screen memory map:</u>

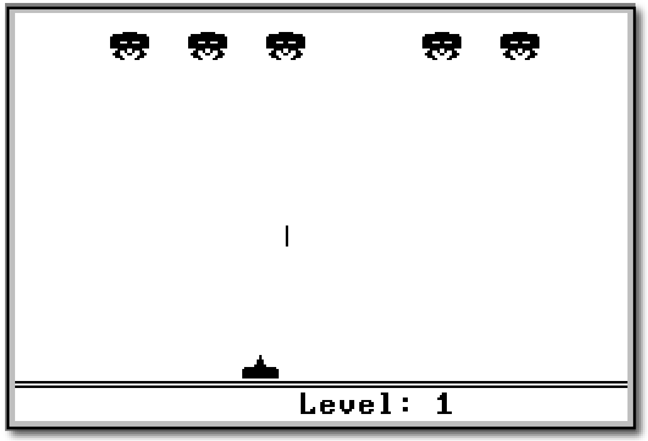
An 8K memory block, dedicated to representing a black-and-white display unit

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

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

Bitmaps

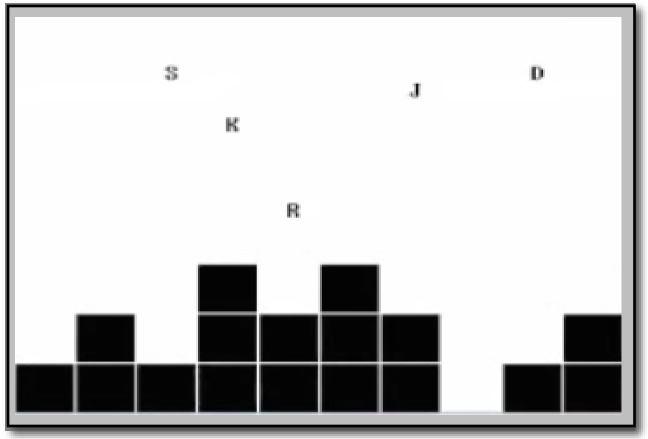
Physical screen



Screen shots of computer games
developed on the Hack computer

Bitmaps

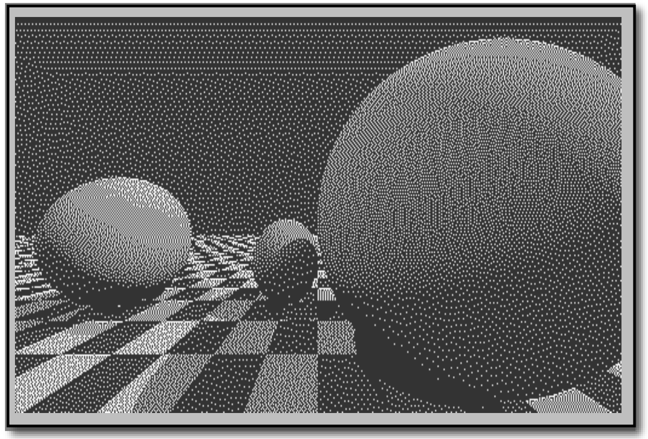
Physical screen



Screen shots of computer games developed on the Hack computer

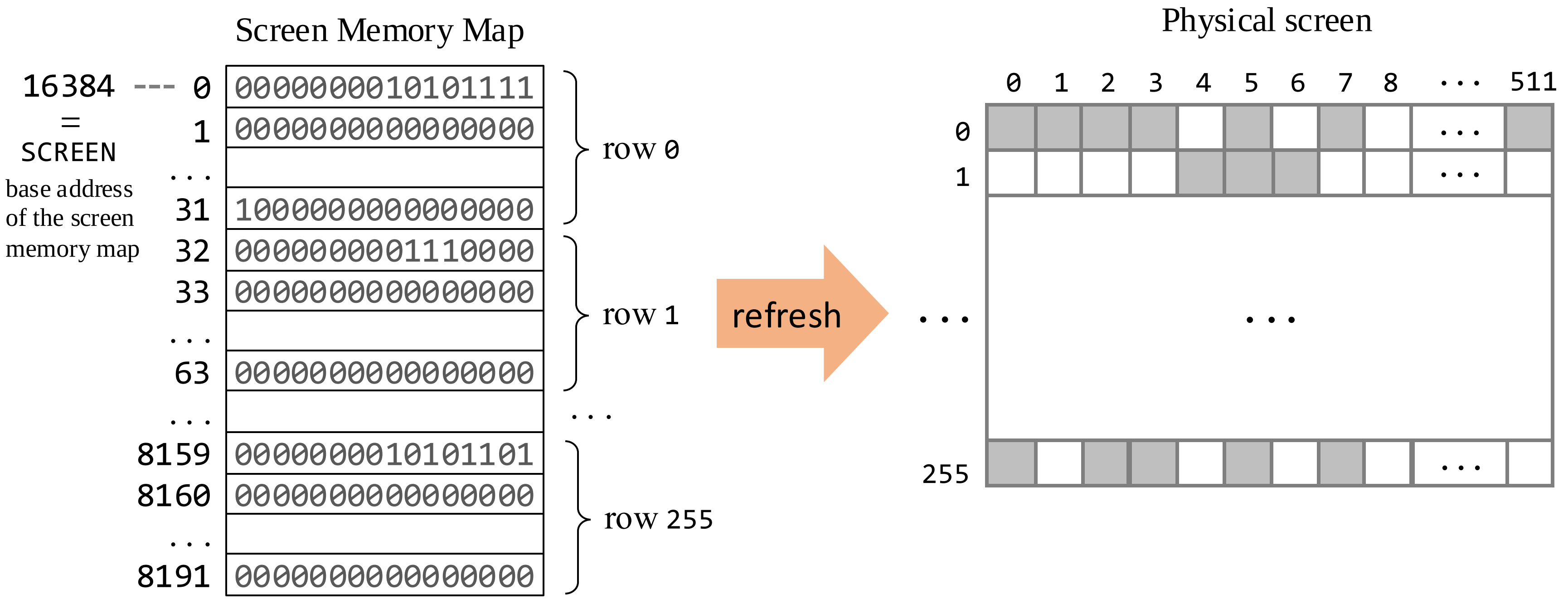
Bitmaps

Physical screen



Screen shots of computer games developed on the Hack computer

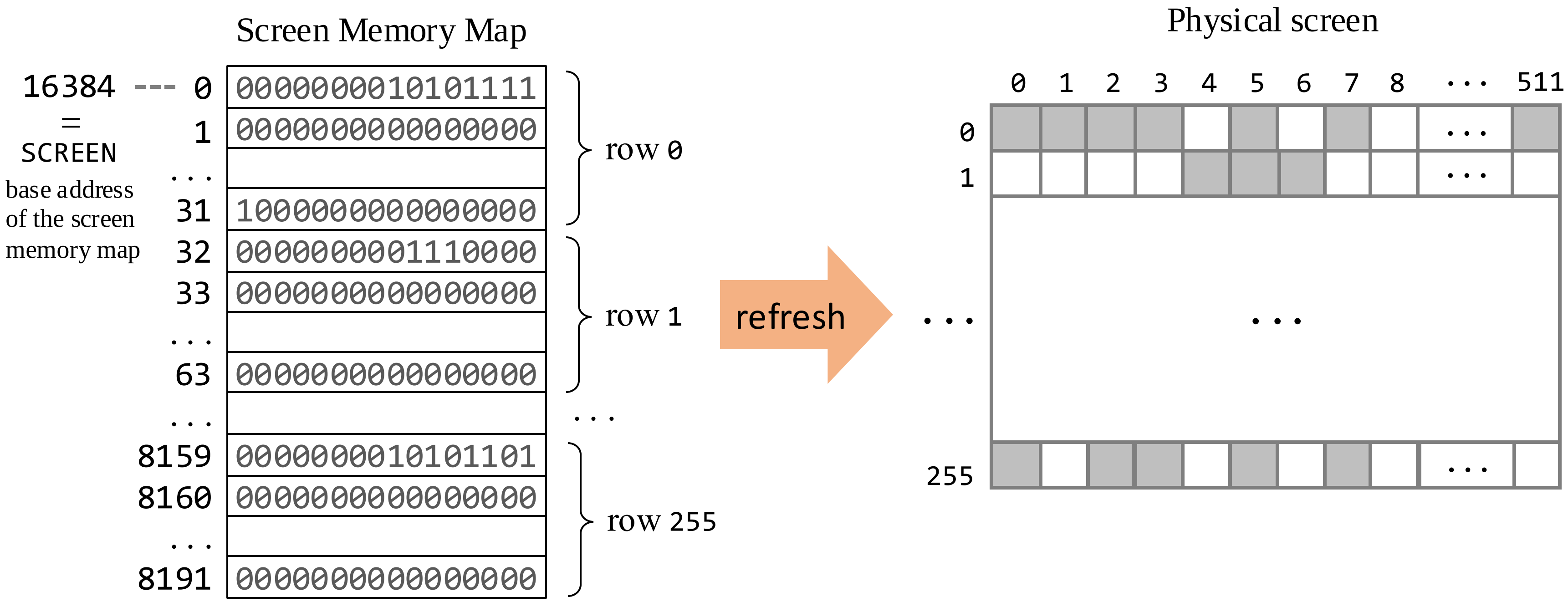
Bitmaps



<u>Mapping</u>

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

Bitmaps



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

addr ← SCREEN + 32 * *row* + *col* / 16

word ← RAM[*addr*]

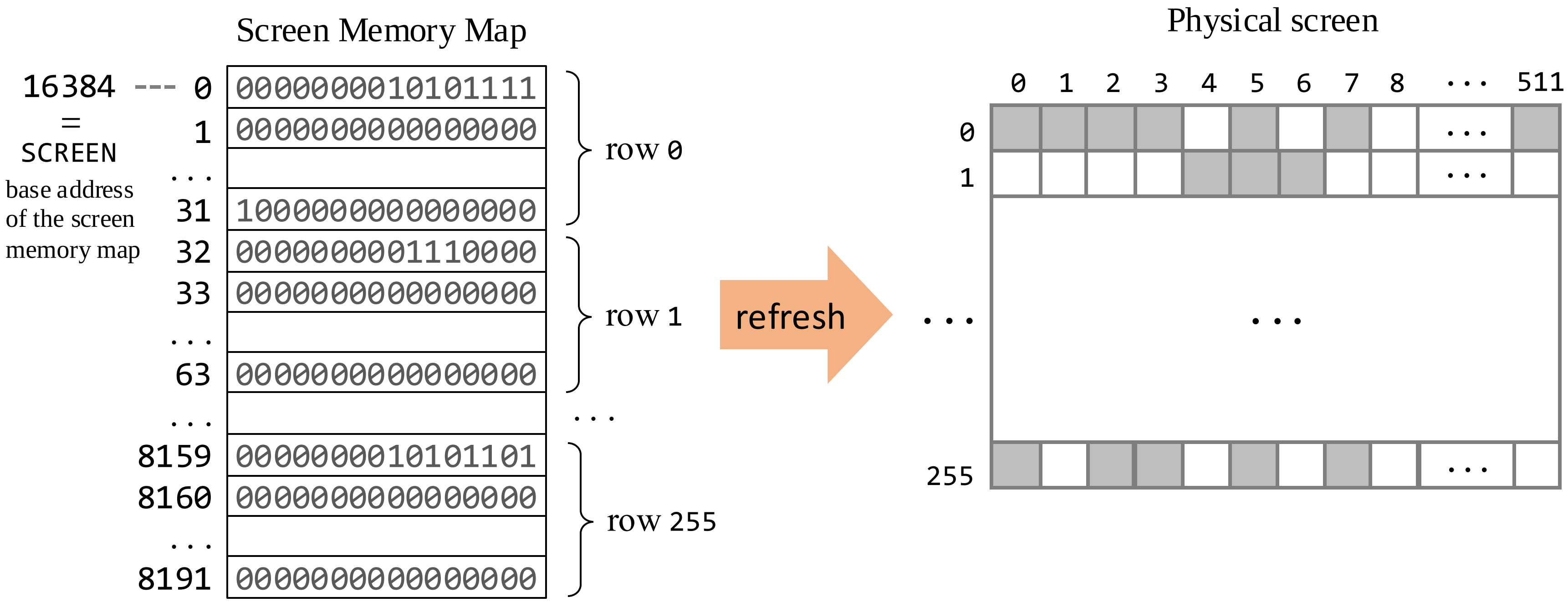
Set the (*col* % 16) *th* bit of *word* to 0 or 1

RAM[*addr*] ← *word*

Not to worry...

Cool Bitmap Editor coming up

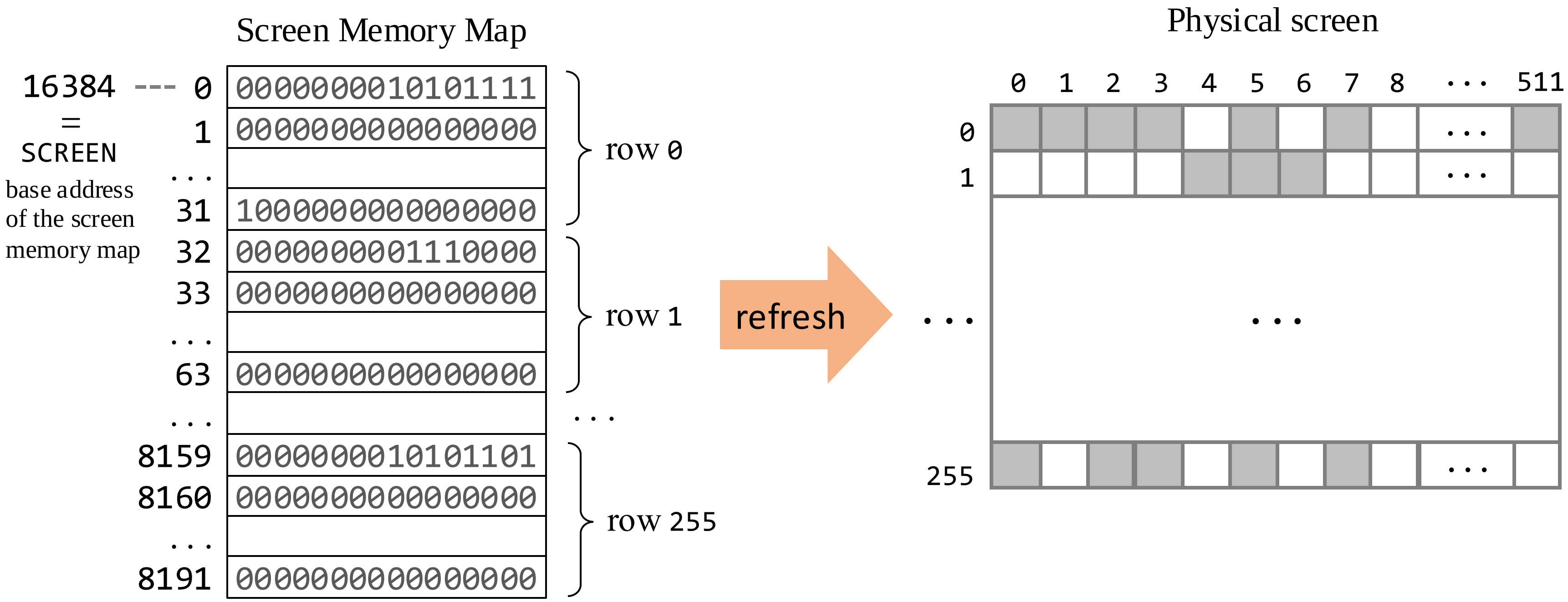
Bitmaps



Examples of simple patterns that can be easily drawn:

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

Bitmaps

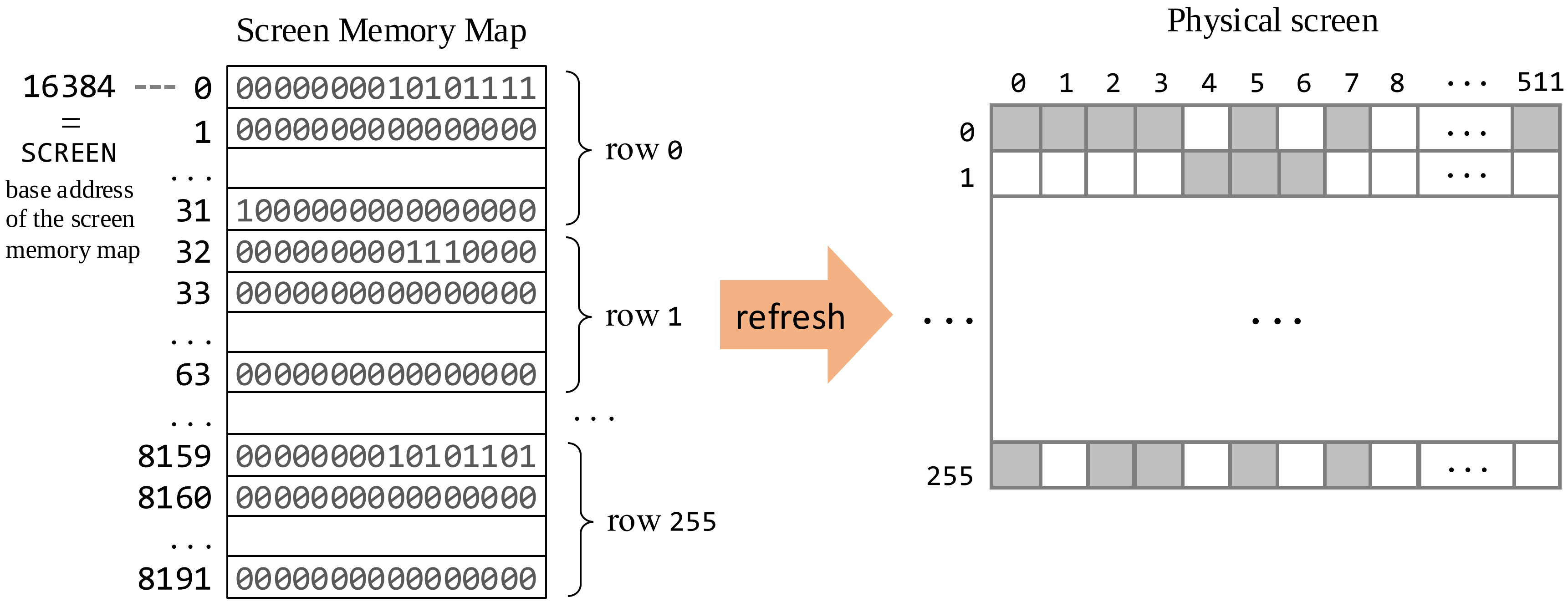


Examples of simple patterns that can be easily drawn:

```
// 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

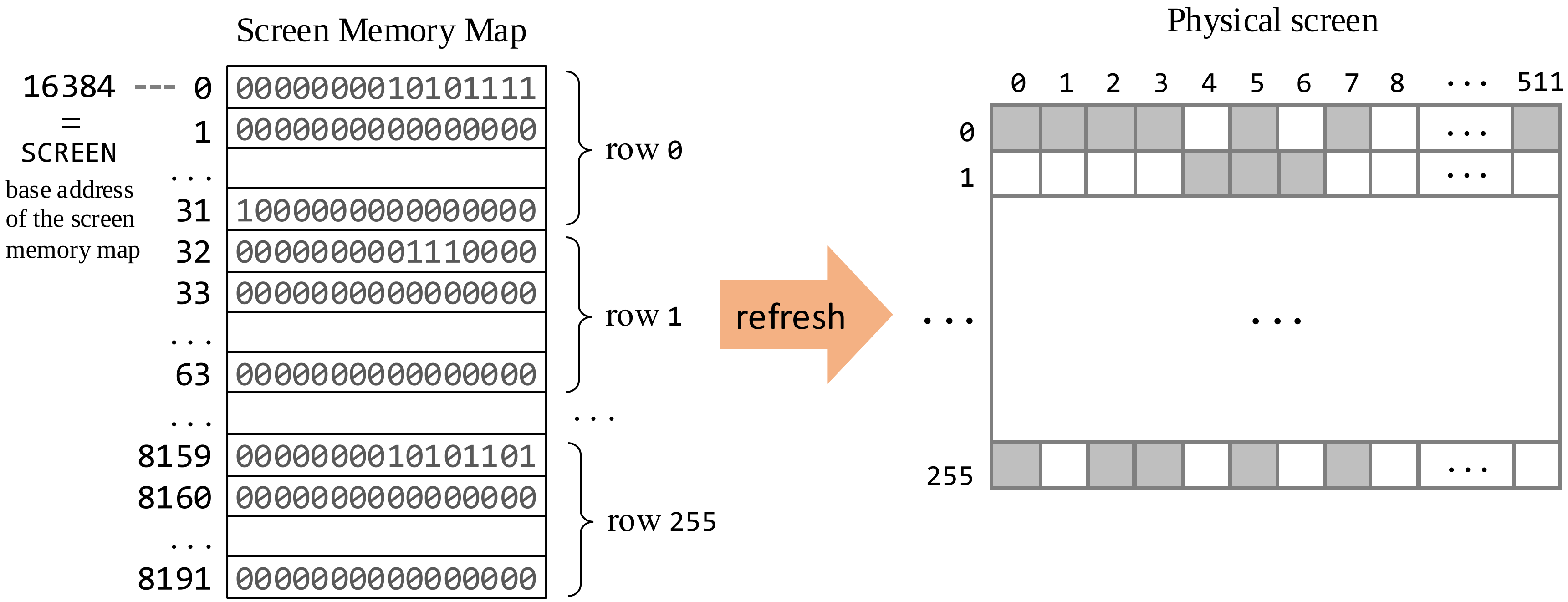


Examples of simple patterns that can be easily drawn:

```
// 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 easily drawn:

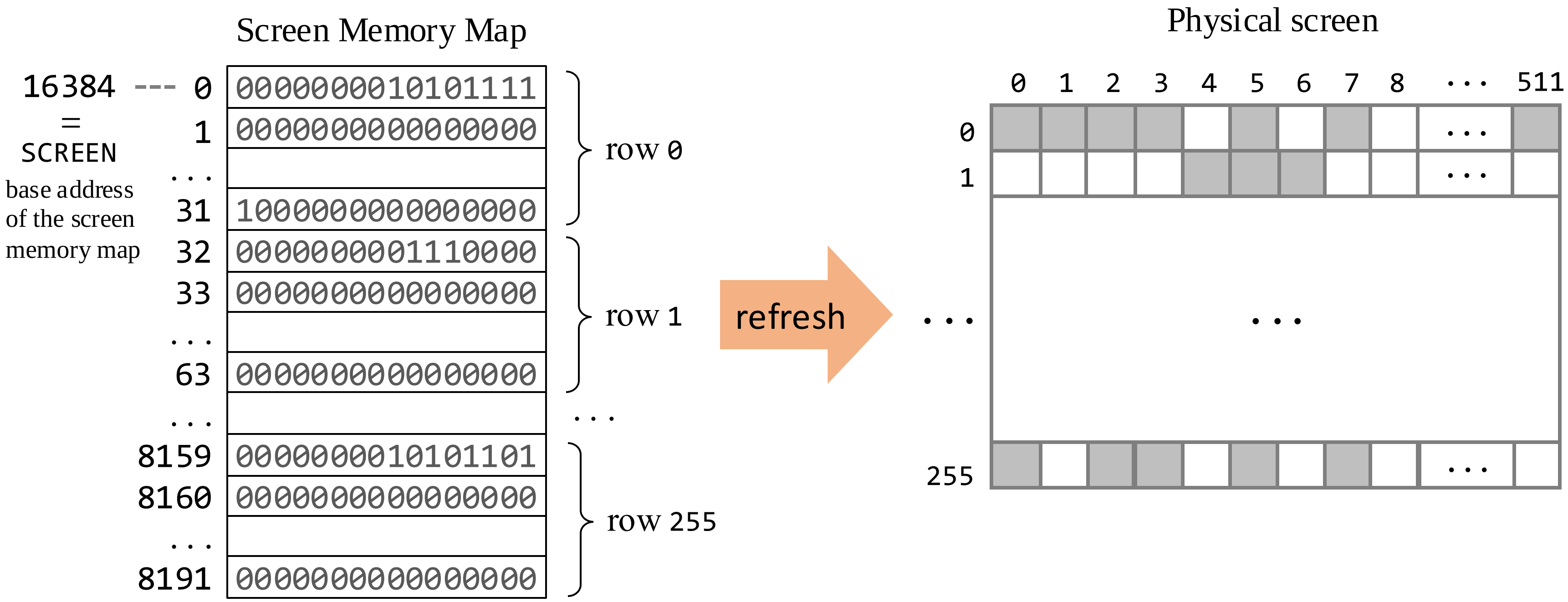
```
// Sets the first (left) 16 pixels
// of the top row to black
@SCREEN
M=-1        // -1 = 1111111111111111
```

```
// 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
```

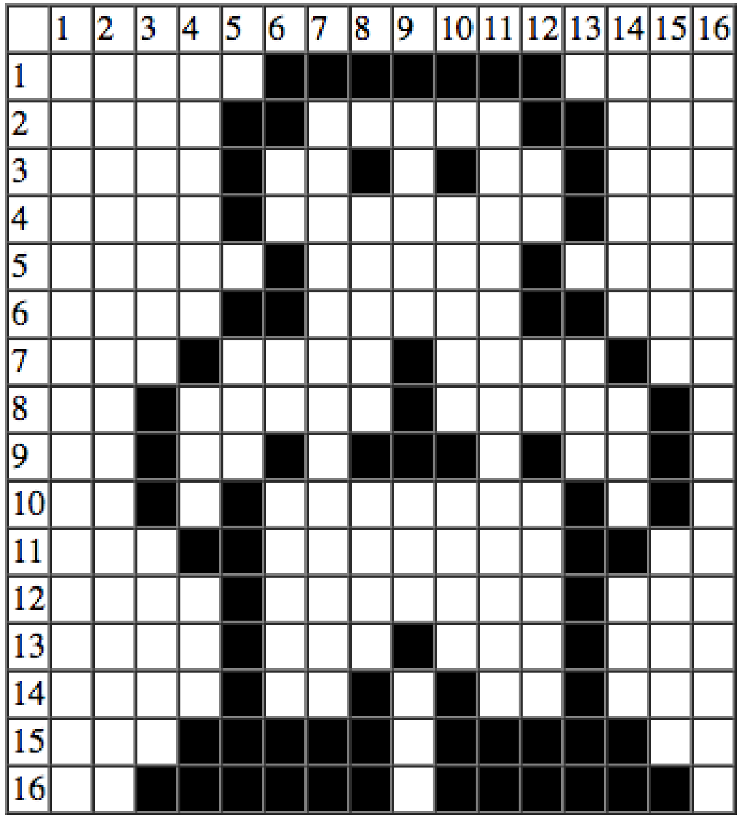
(Project 4)

Bitmaps



Simple graphics program:

Bitmap Editor




```
0000111111100000 = 4064
0001100000110000 = 6192
0001001010010000 = 4752
...
...
0111111011111100 = 32508
```

[Bitmap editor](#) (web-based tool)

The developer draws a pixeled image on a 2D grid;

The tool generates assembly code that draws the image in the RAM;

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

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

Programming examples

- Basic
- Iteration
- Pointers

The Hack Language

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

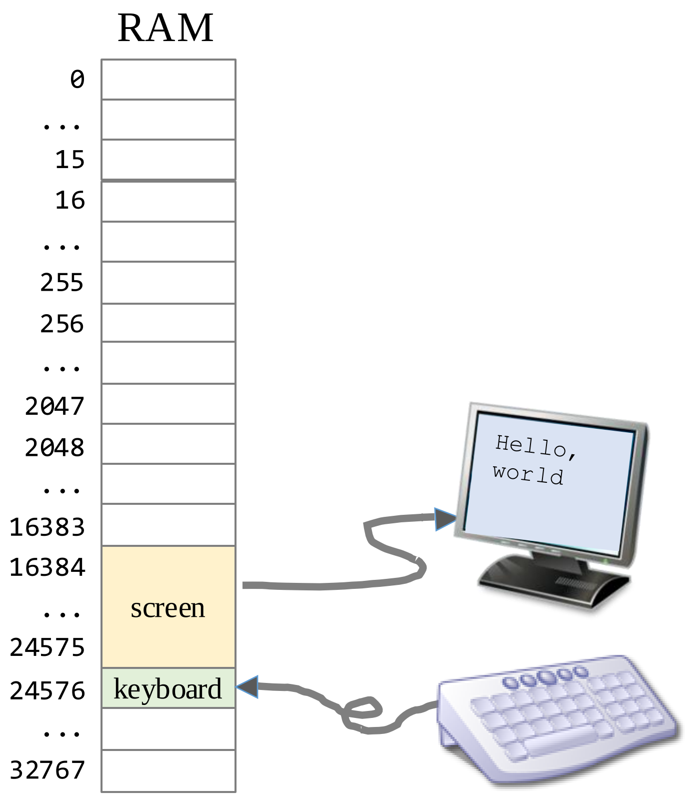
Input

High-level input handling (later in the course)

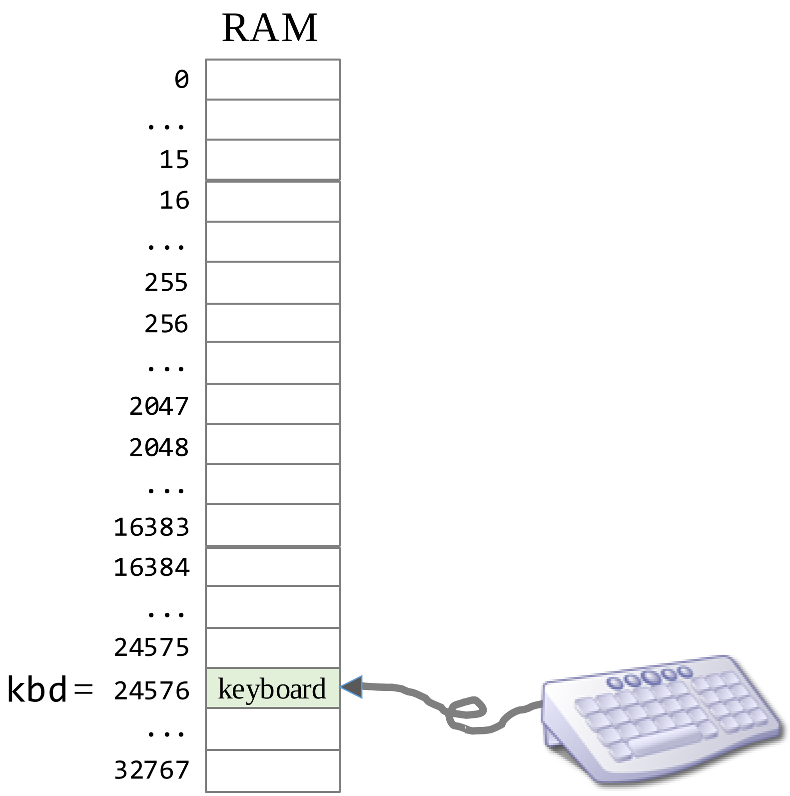
`readInt`, `readString`, ...

Low-level input handling

Read bits.



Input



Keyboard memory map

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

Base address: `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

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

key	code
A	65
B	66
C	…
…	…
Z	90

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

key	code
a	97
b	98
c	99
…	…
z	122

{	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

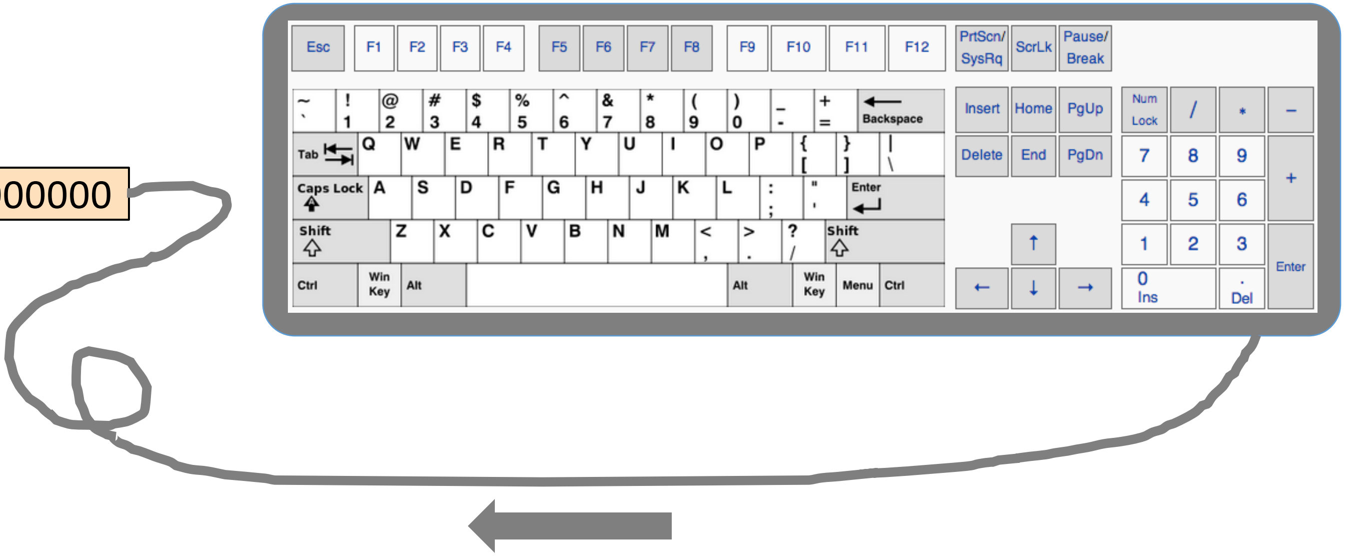
RAM

24576 | 0000000000000000

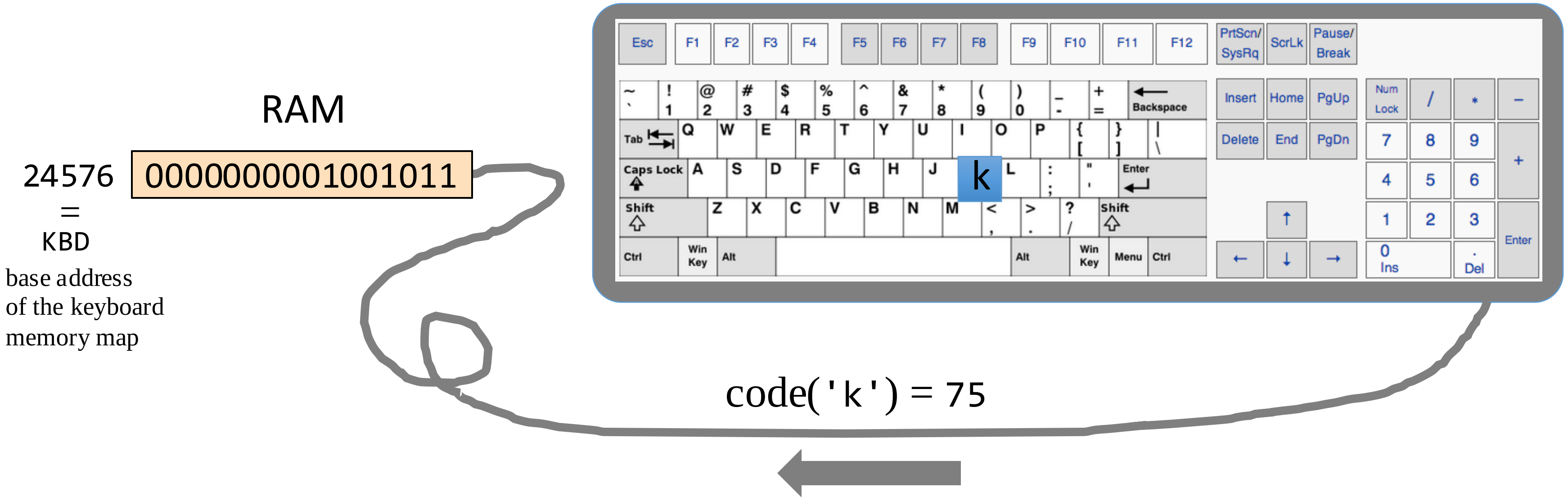
=

KBD

base address
of the keyboard
memory map



Memory mapped input



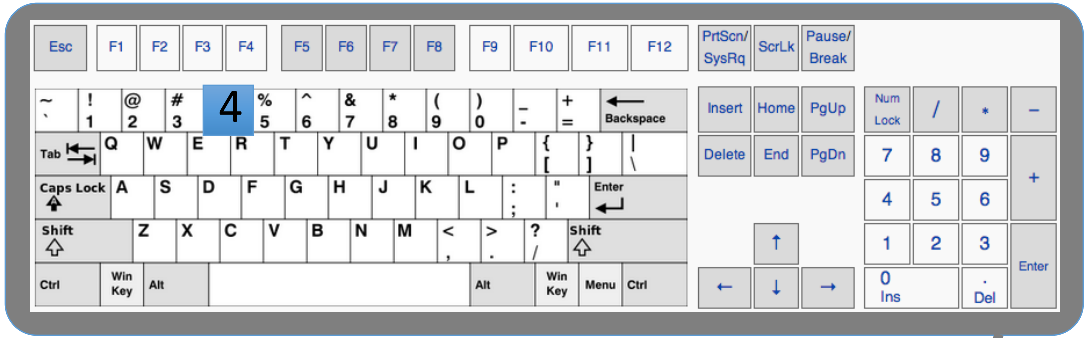
When a key is pressed on the keyboard,
the key’s character code appears in the keyboard memory map.

Memory mapped input

RAM

24576 | 0000000000110100
= KBD

base address of the keyboard memory map



code('4') = 52

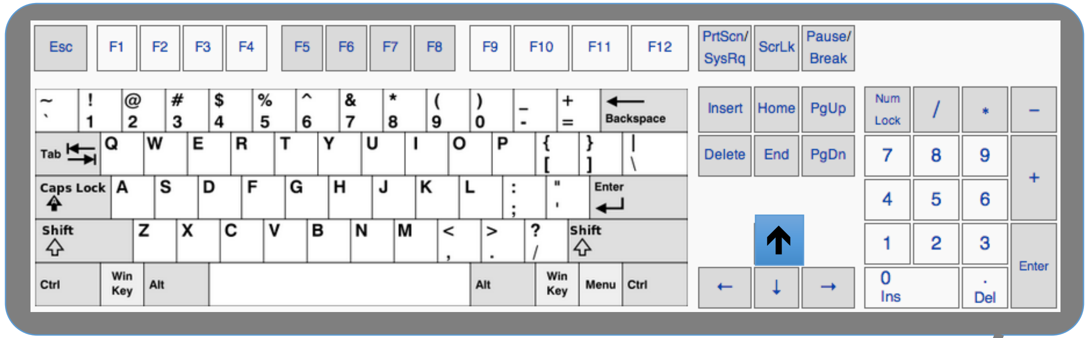
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input

RAM

24576 | 0000000010000011
=
KBD

base address
of the keyboard
memory map



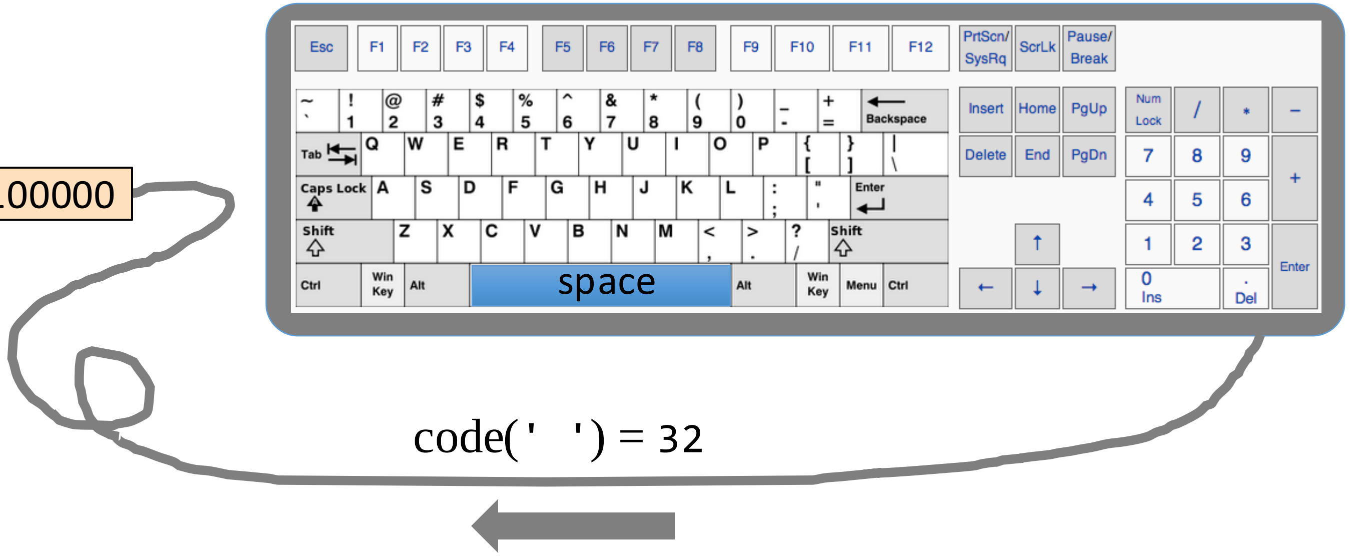
code('↑') = 131

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 | 0000000000100000



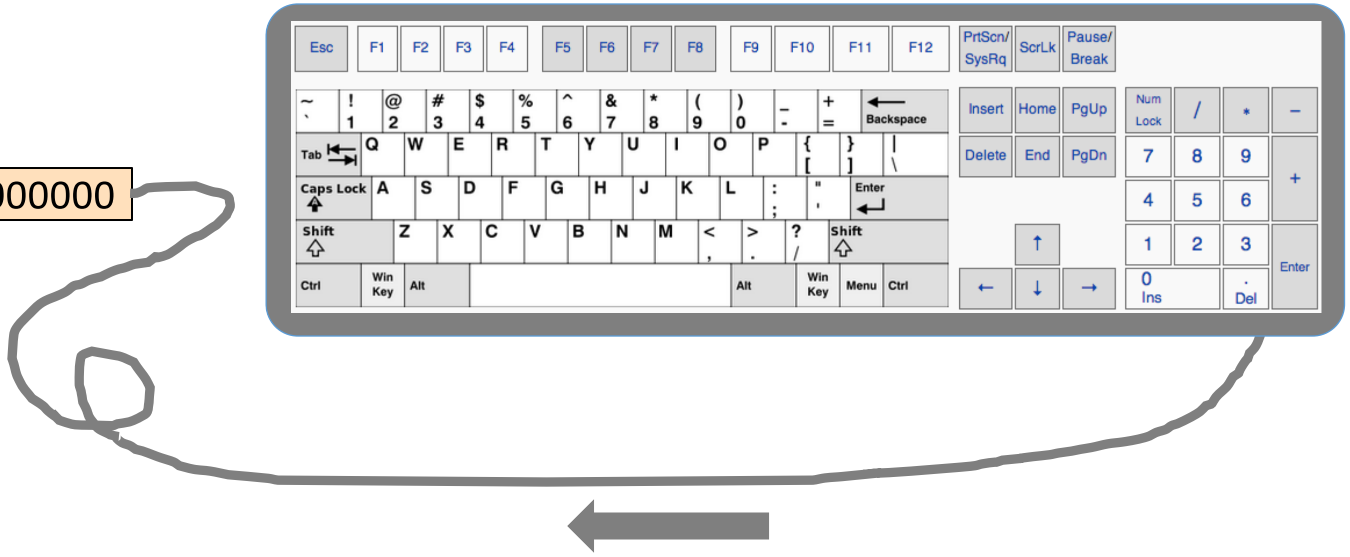
When a key is pressed on the keyboard,
the key's character code appears in the keyboard memory map.

Memory mapped input

RAM

24576 = KBD | 0000000000000000

base address
of the keyboard
memory map



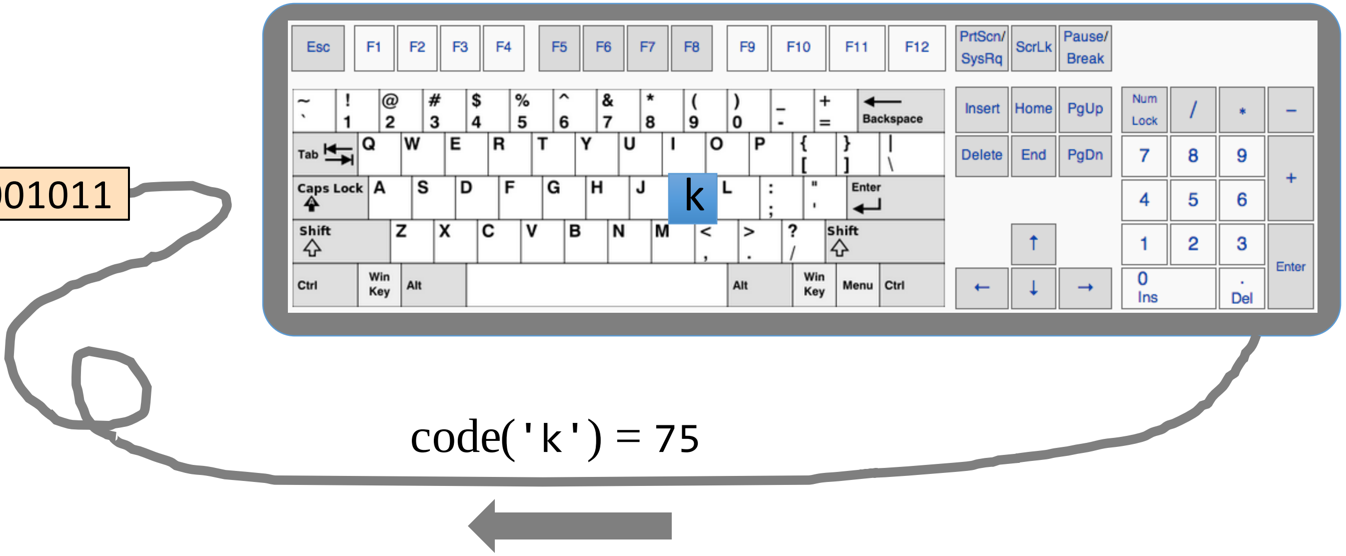
When no key is pressed, the resulting code is `0`.

Reading inputs

RAM

24576 | 0000000001001011
=
KBD

base address
of the keyboard
memory map



Examples:

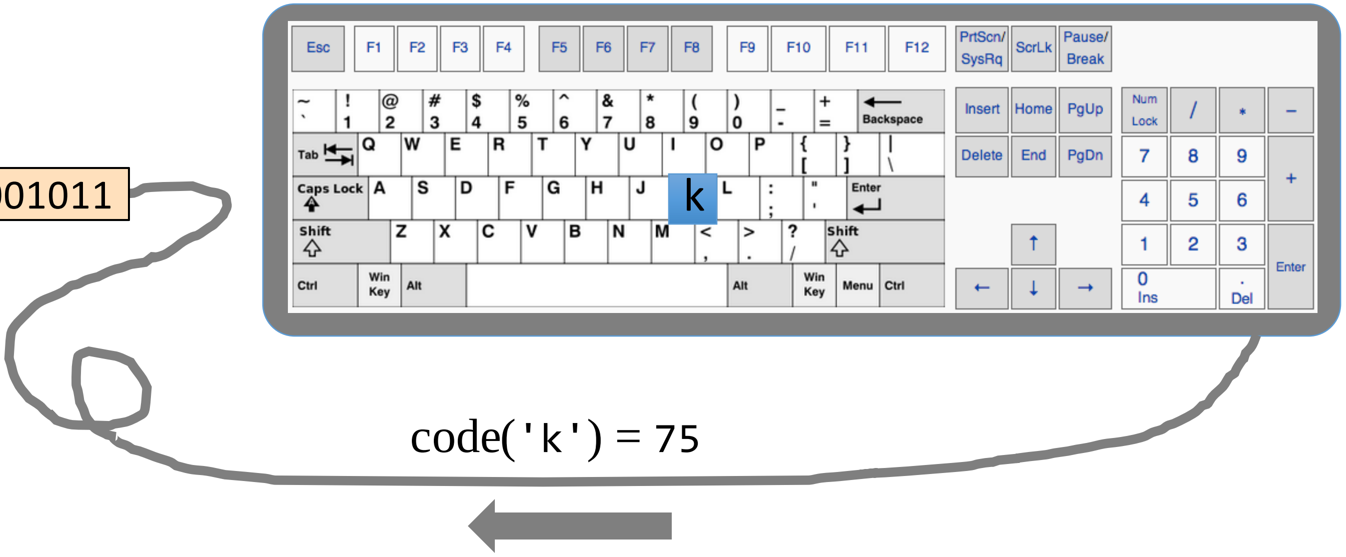
```
// Set D to the character code of
// the currently pressed key
```

Reading inputs

RAM

24576 = KBD — `0000000001001011`

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

Programming examples

- Basic
- Iteration
- Pointers

The Hack Language

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

Project 4

Objectives

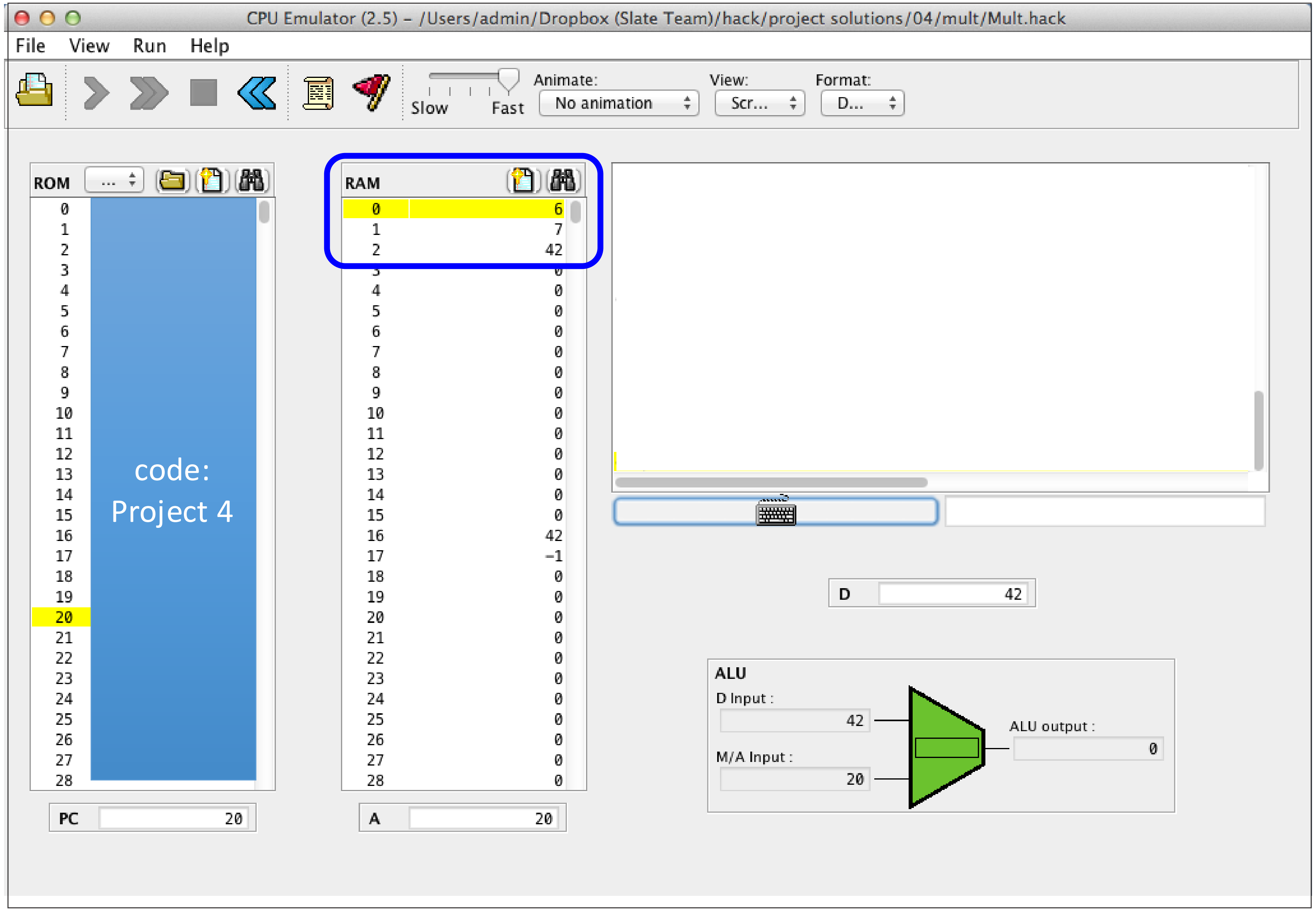
Gain a hands-on taste of:

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

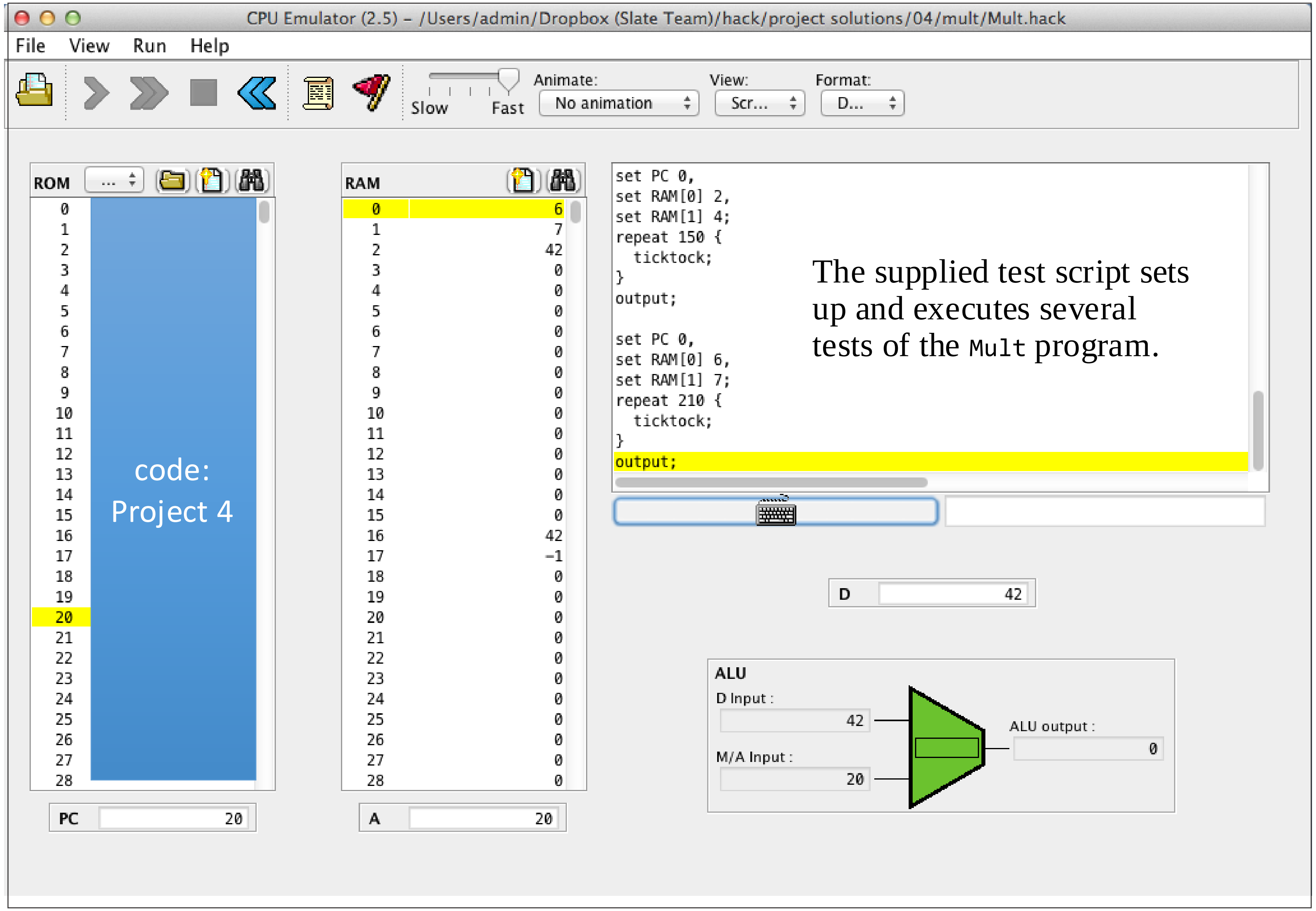
Tasks

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

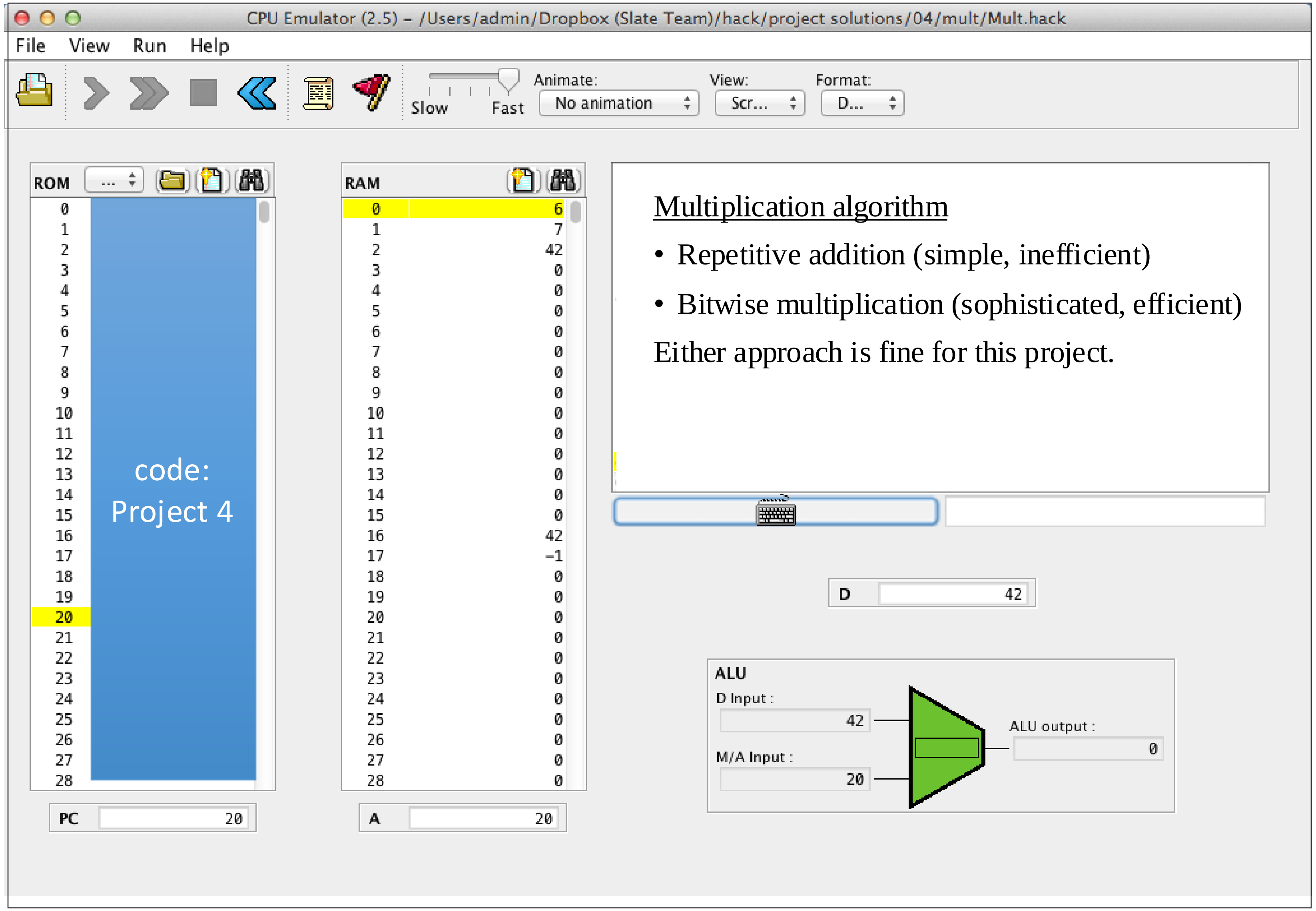
Mult: a program that computes R2 = R0 * R1



Mult: a program that computes R2 = R0 * R1



Mult: a program that computes R2 = R0 * R1

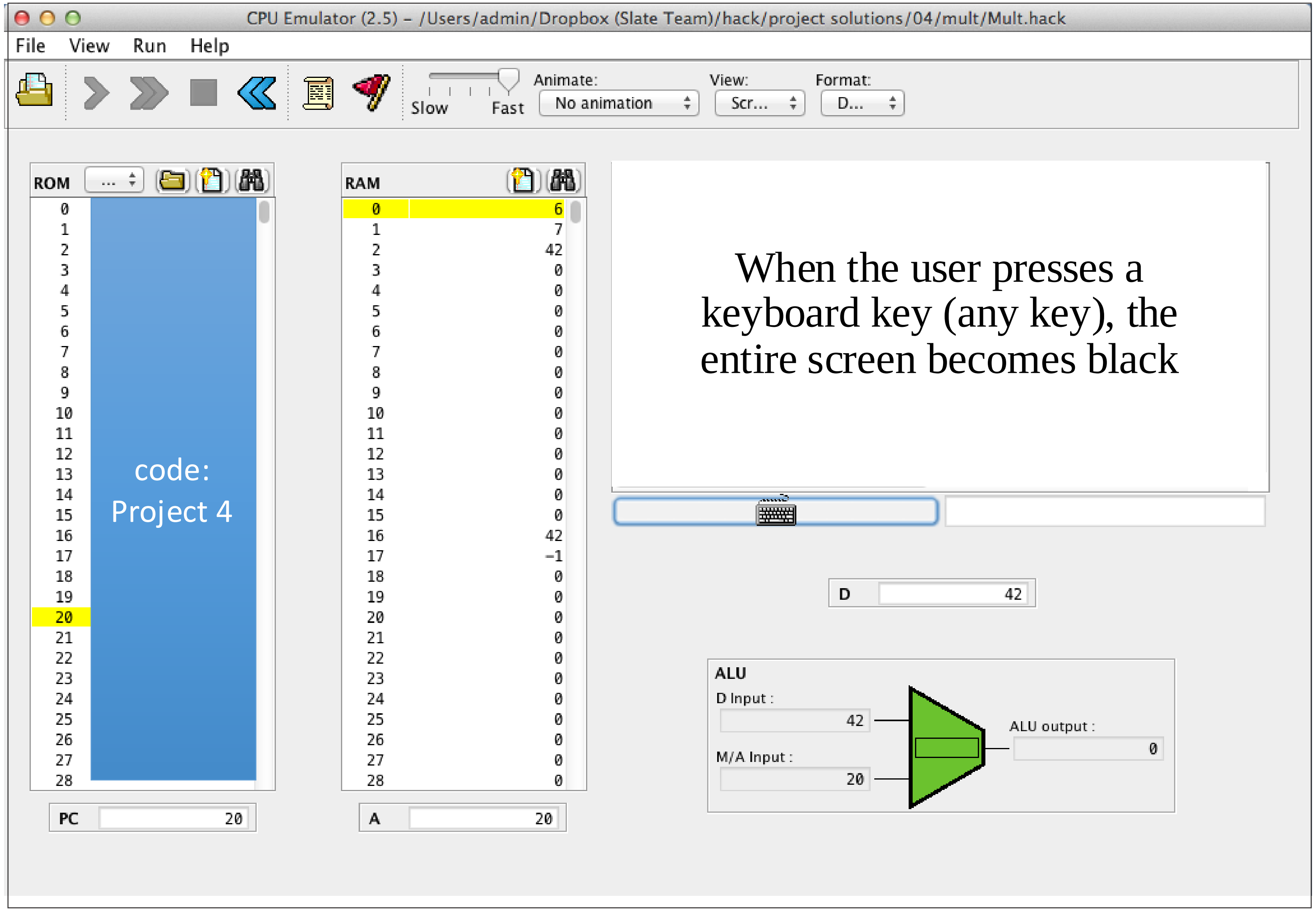


Multiplication algorithm

- Repetitive addition (simple, inefficient)
- Bitwise multiplication (sophisticated, efficient)

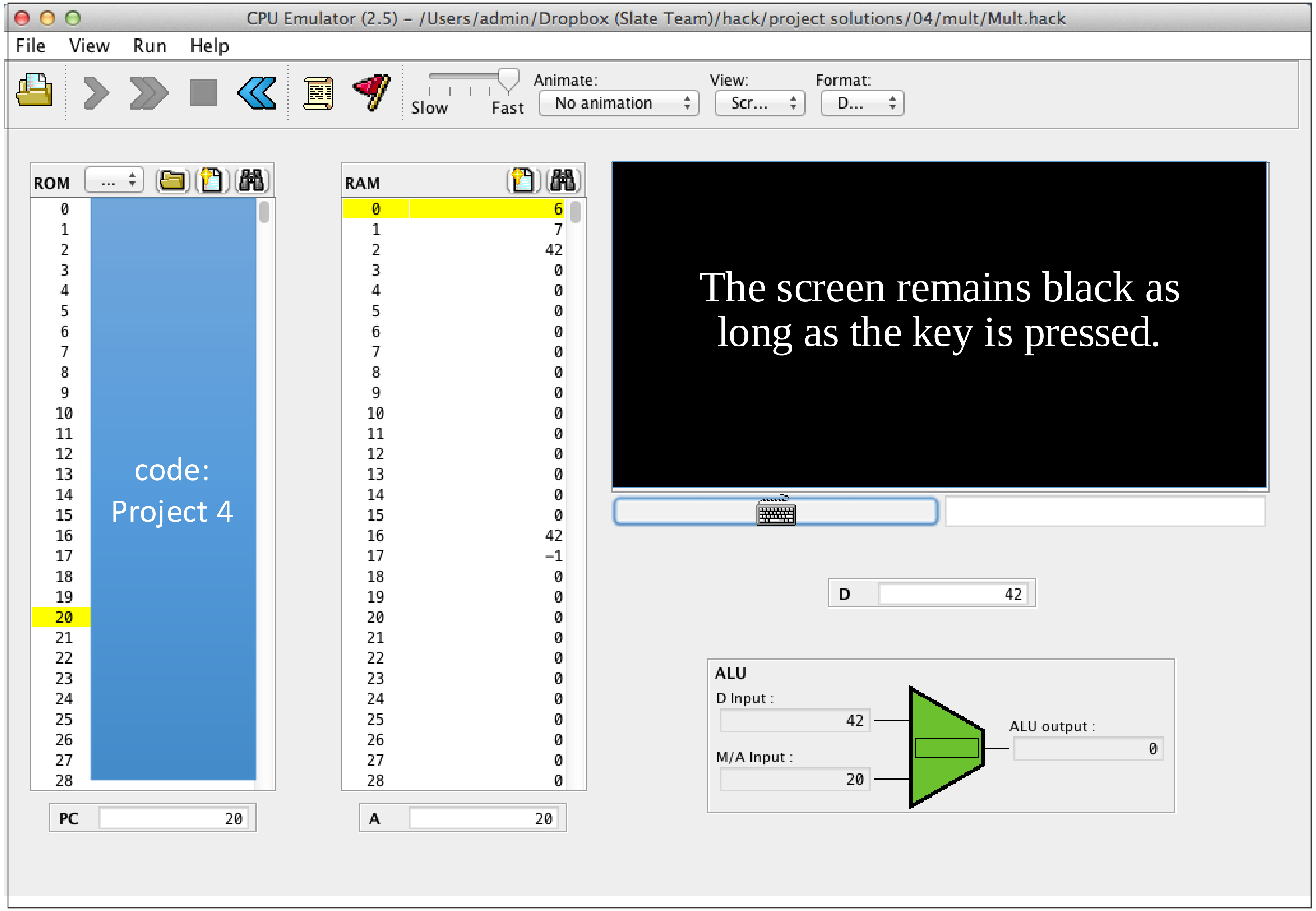
Either approach is fine for this project.

`Fill`: a simple interactive program

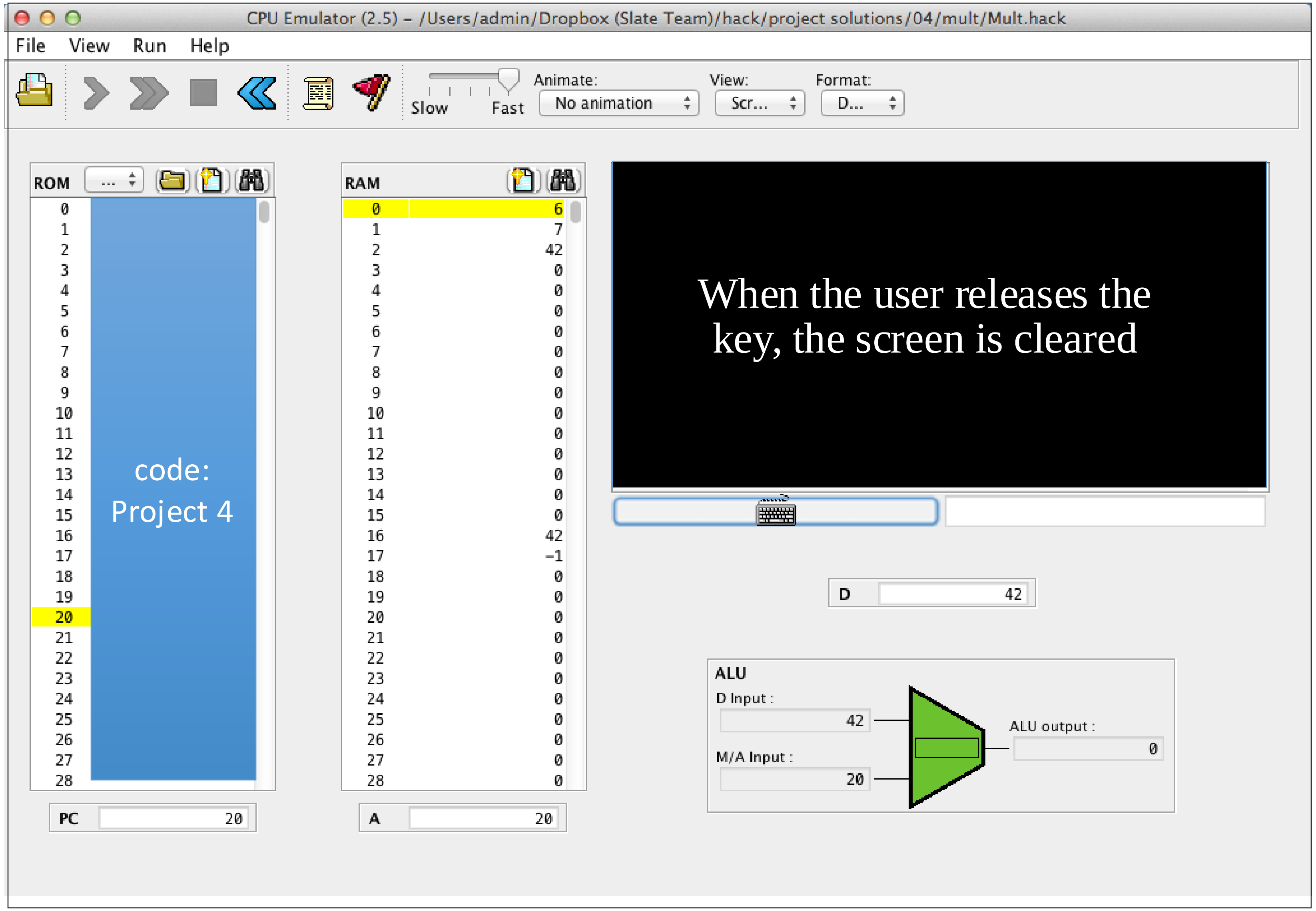


When the user presses a keyboard key (any key), the entire screen becomes black

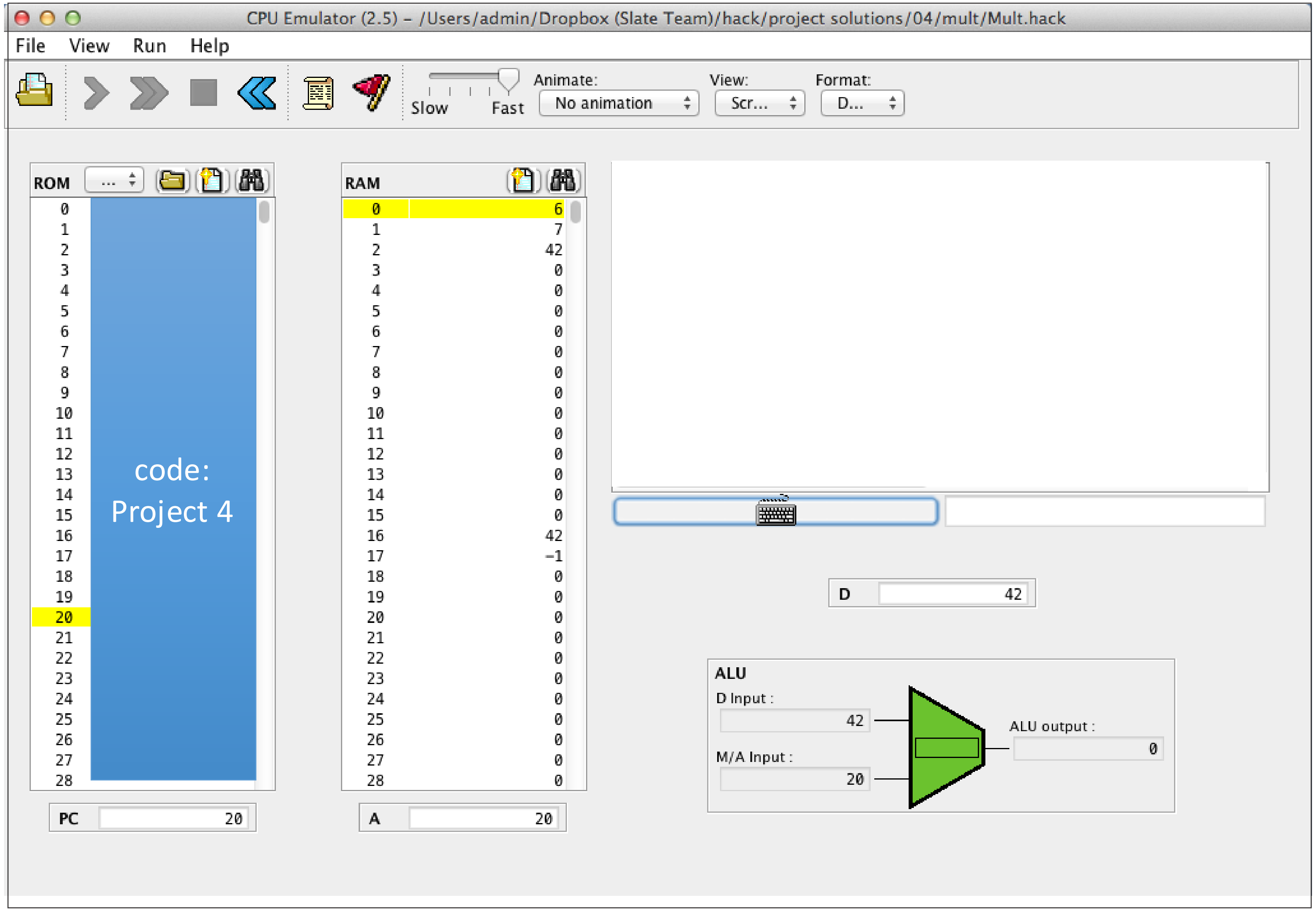
Fill: a simple interactive program



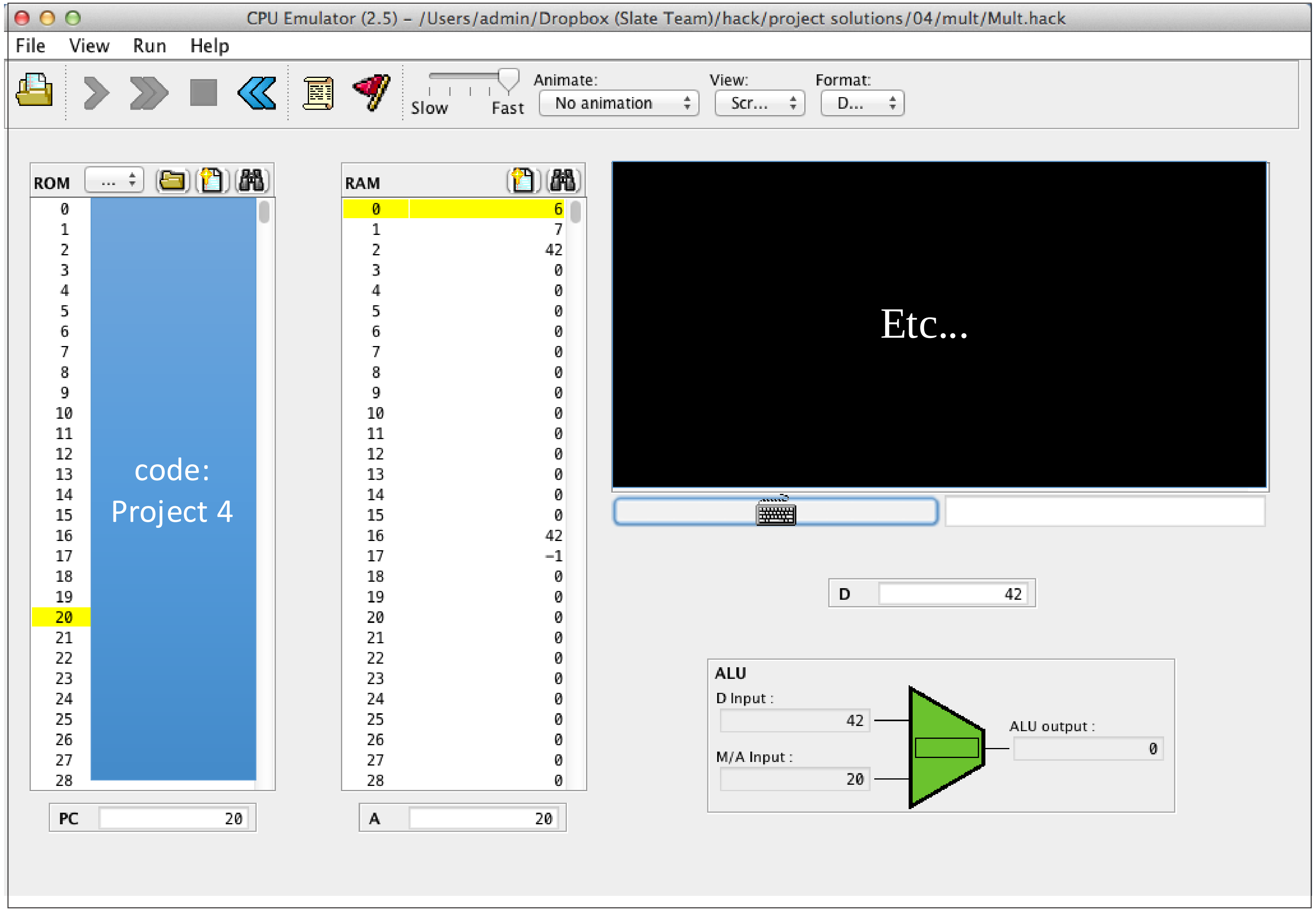
`Fill`: a simple interactive program



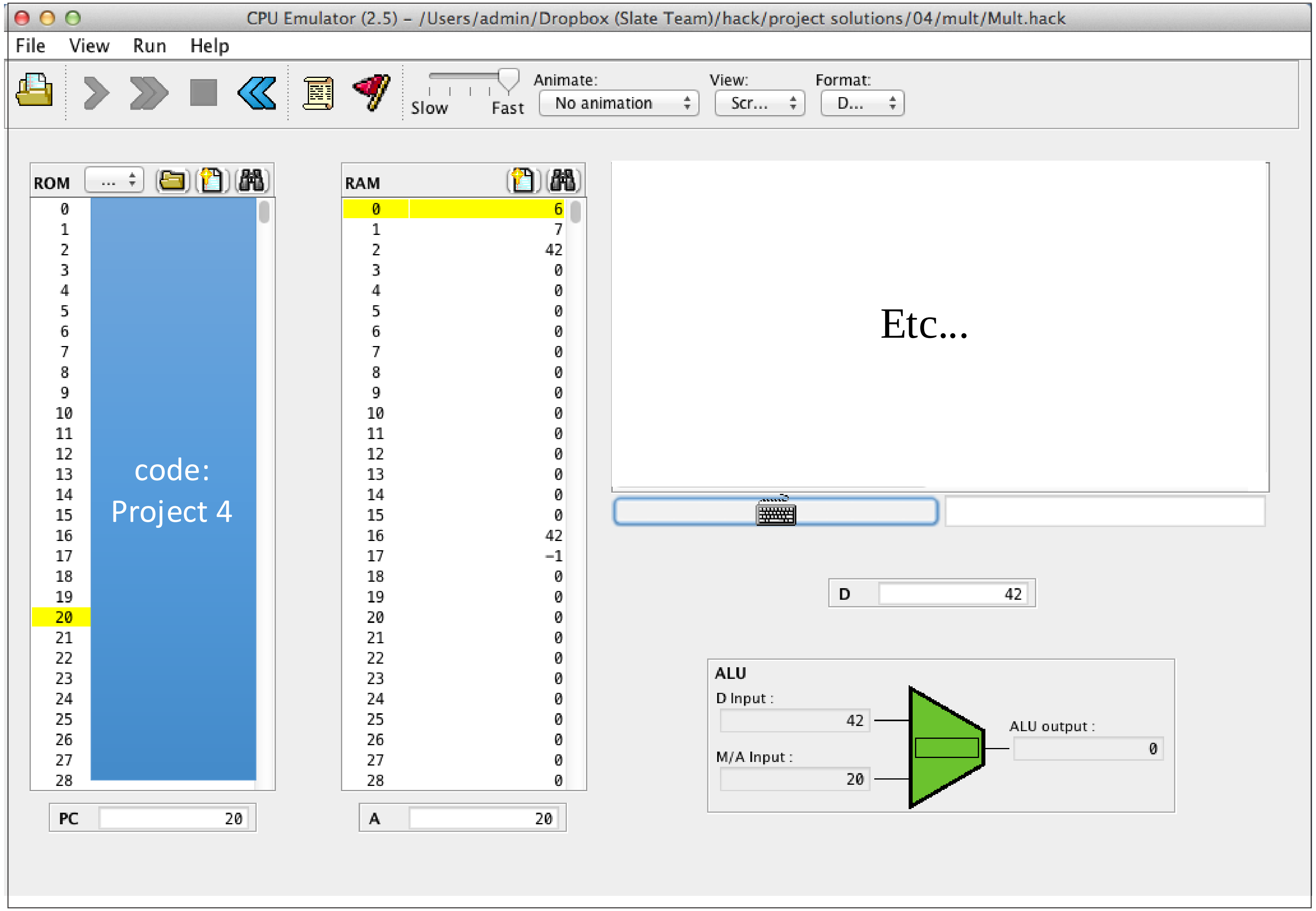
Fill: a simple interactive program



`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!

Implementation notes

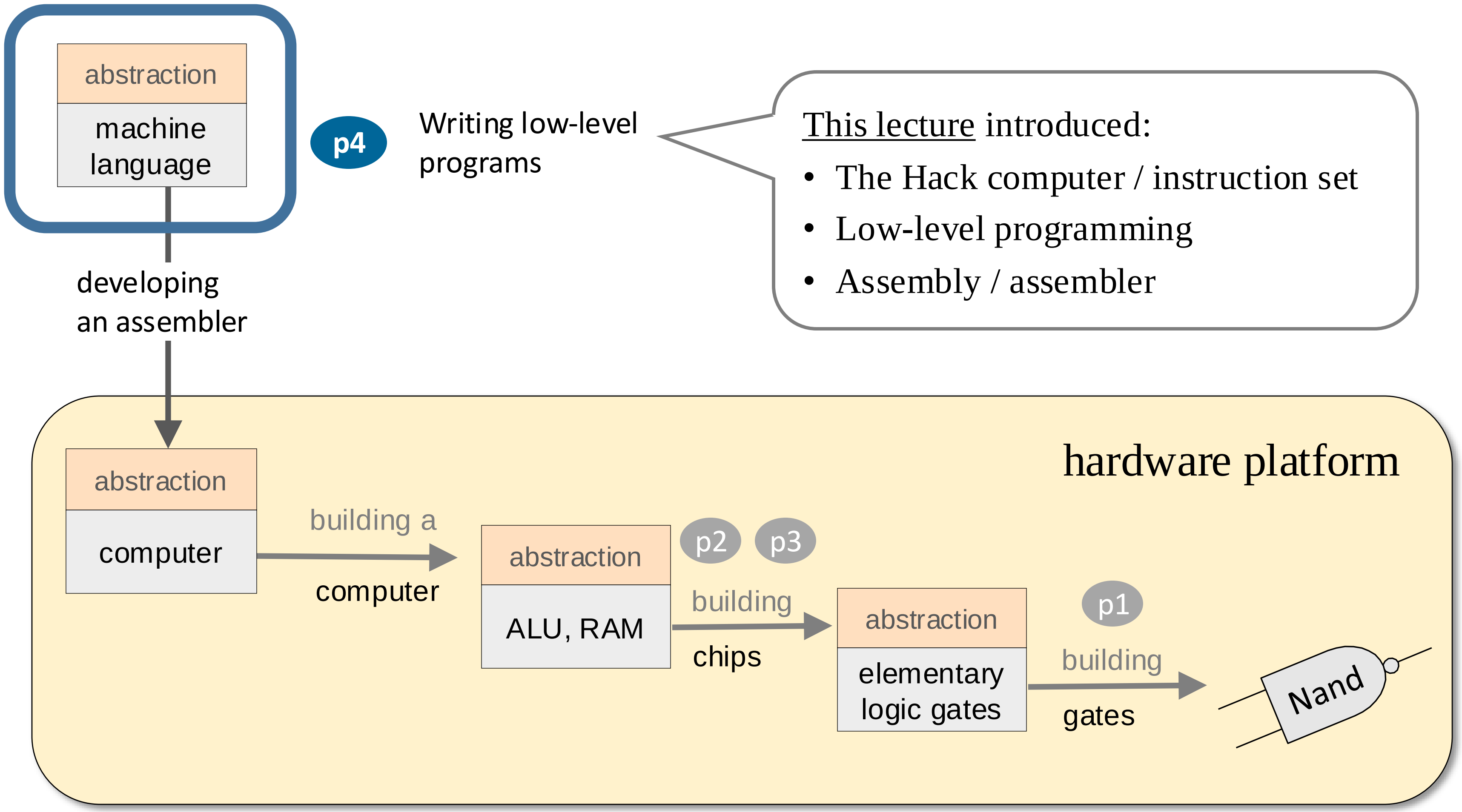
Well-written low-level code is

- 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



This lecture introduced:

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

Nand to Tetris Roadmap: Hardware