Chapter 4

# Machine Language

These slides support chapter 4 of the book

*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press

# Usage Notice

You are welcome to use this presentation, or any part of it,
in Nand to Tetris courses, or in other courses, as you see fit.

Usage is free provided that it supports instruction in an educational,
non-profit setting.

Feel free to use the slides as-is, or modify them as needed.

We'll appreciate it if you will give credit somewhere to Nand to Tetris,
and put a reference to www.nand2tetris.org

You are welcome to remove this slide from the presentation. If you make
extensive changes to the slides, you can remove the copyright notice also.

Happy teaching!

Noam Nisam / Shimon Schocken

# Machine Language: lecture plan

- Machine languages
- Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
- Hack programming
- Project 4 overview

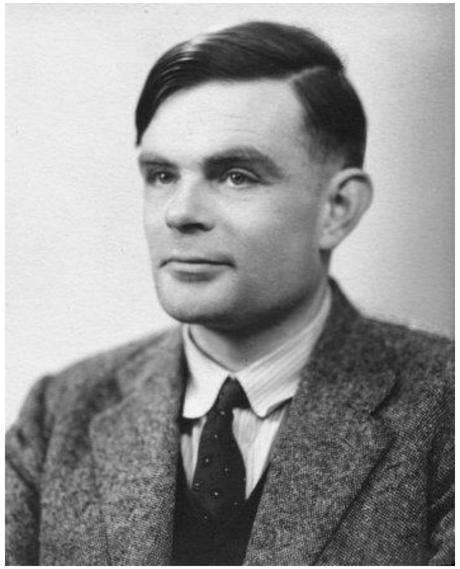
# Computers are flexible

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

# Universality

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

Theory



Alan Turing:

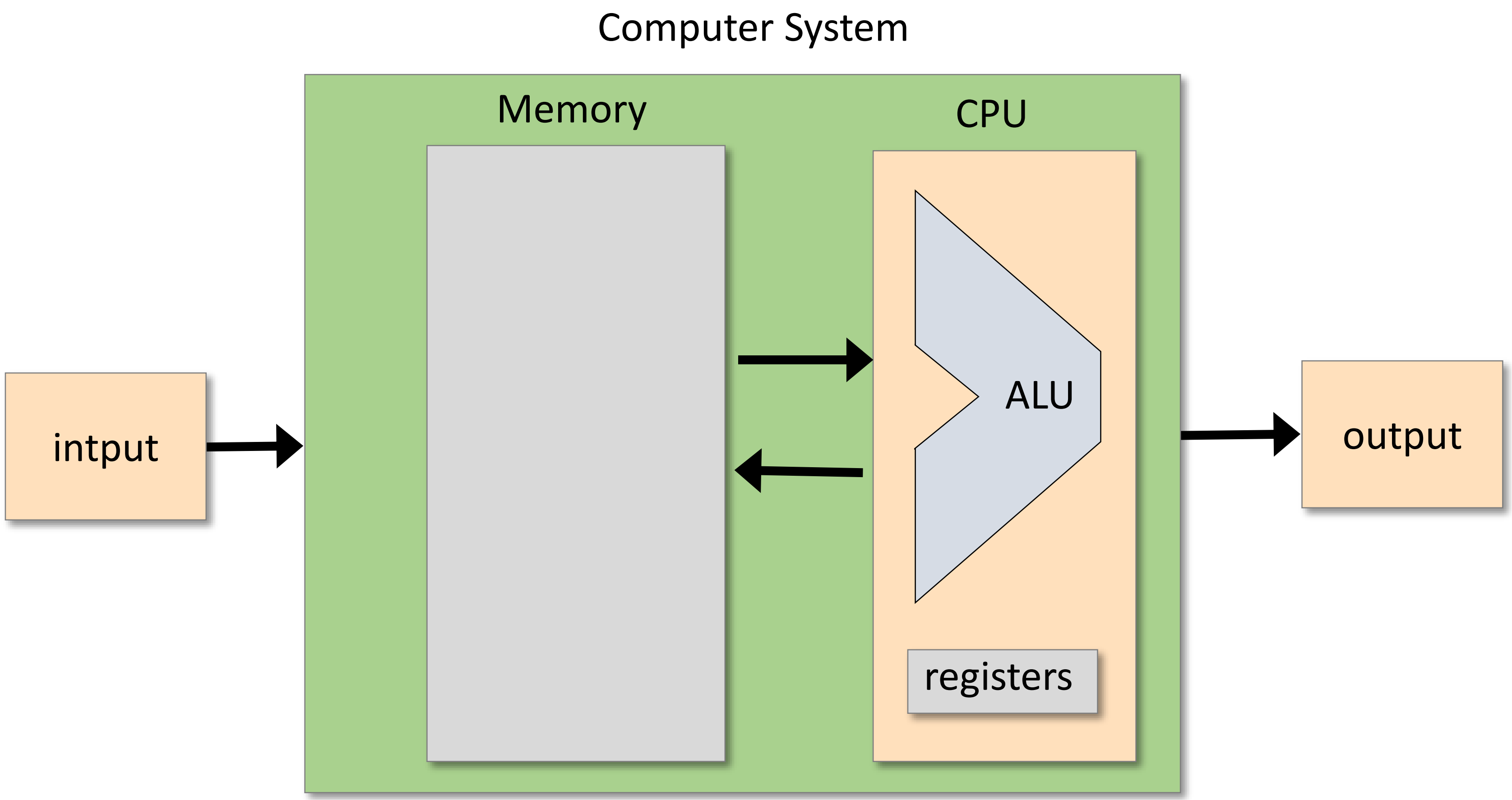
Universal Turing Machine

Practice

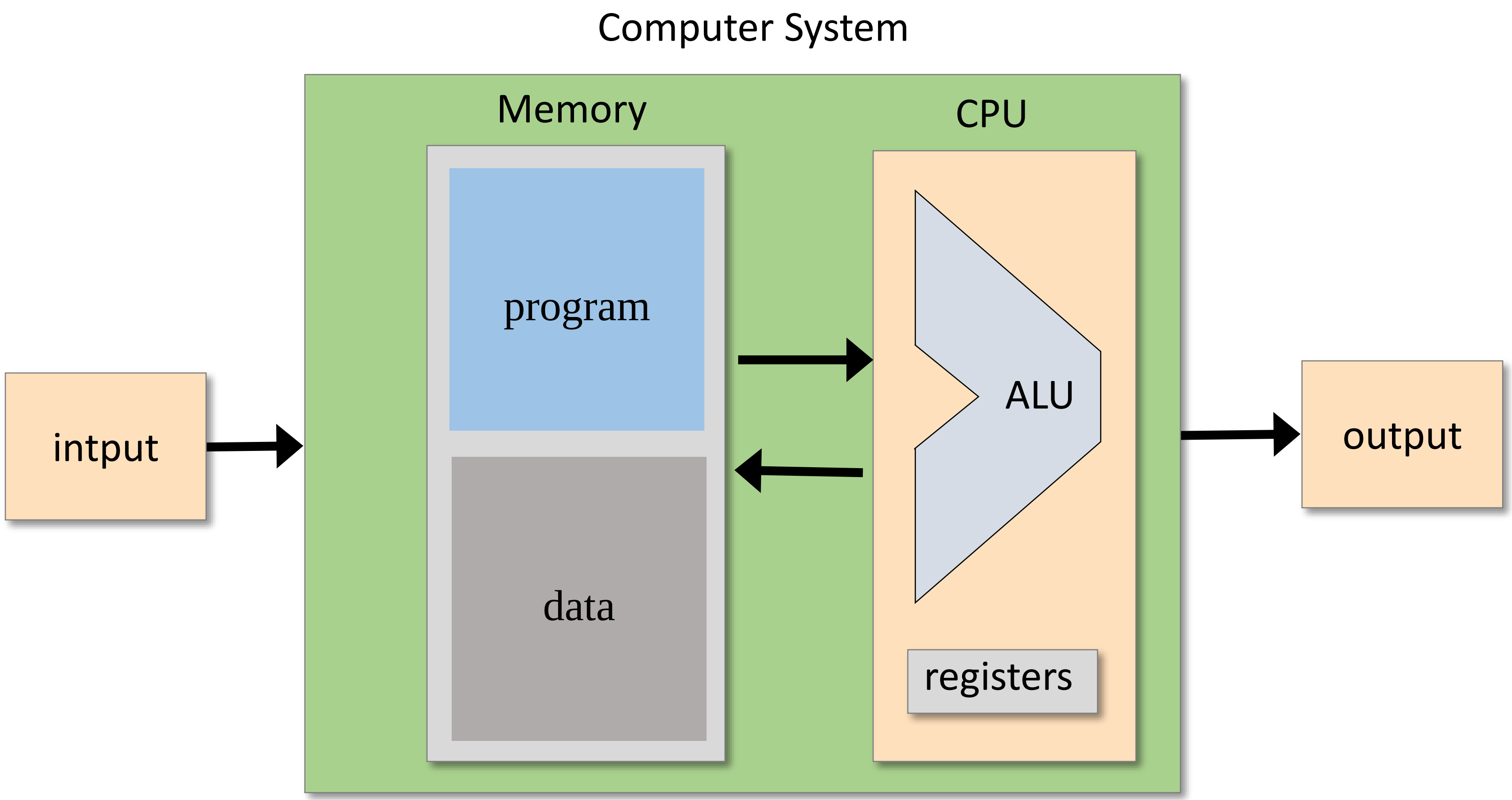
John Von Nuemann:

Stored Program Computer

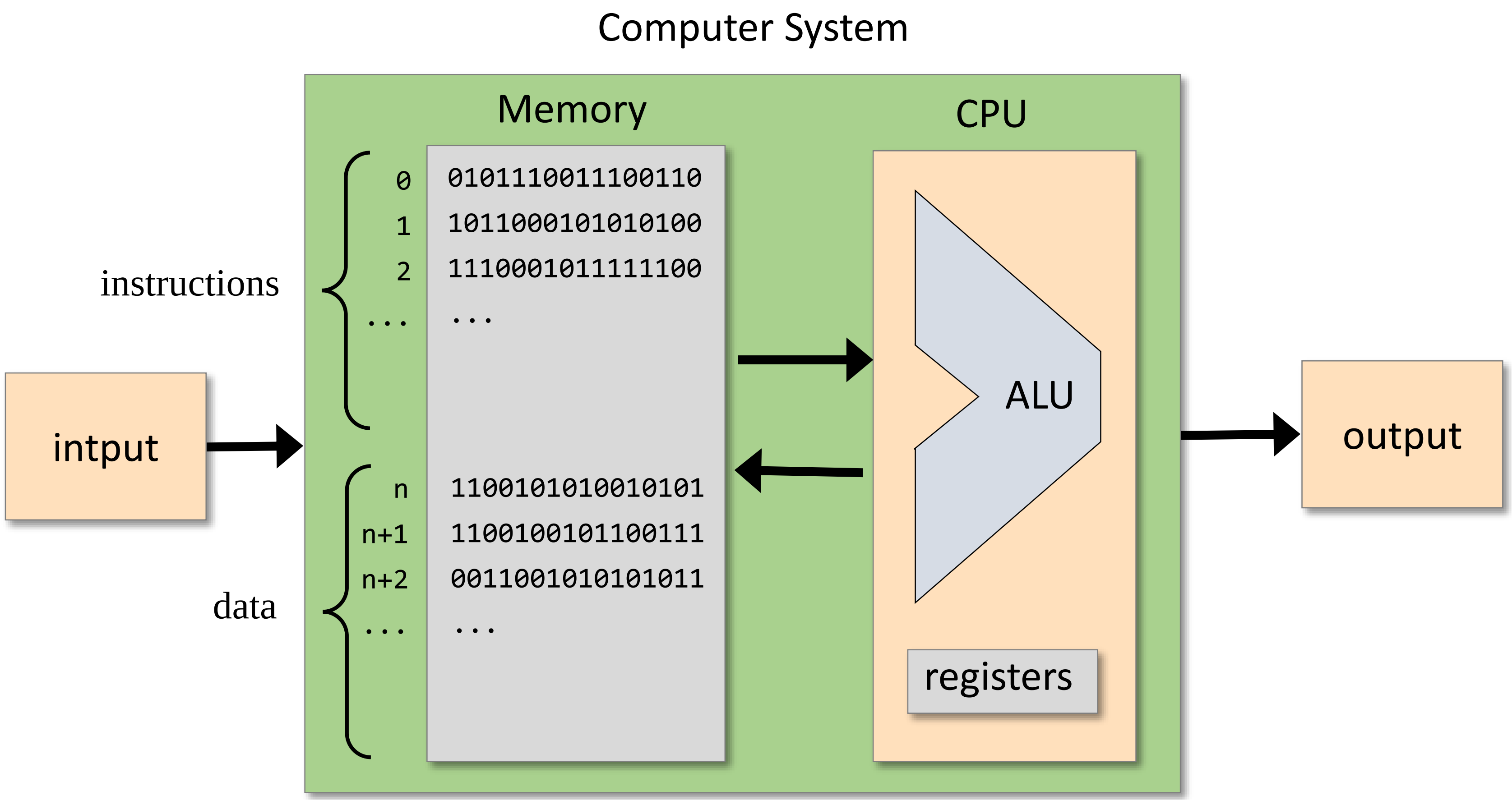
# Stored program concept



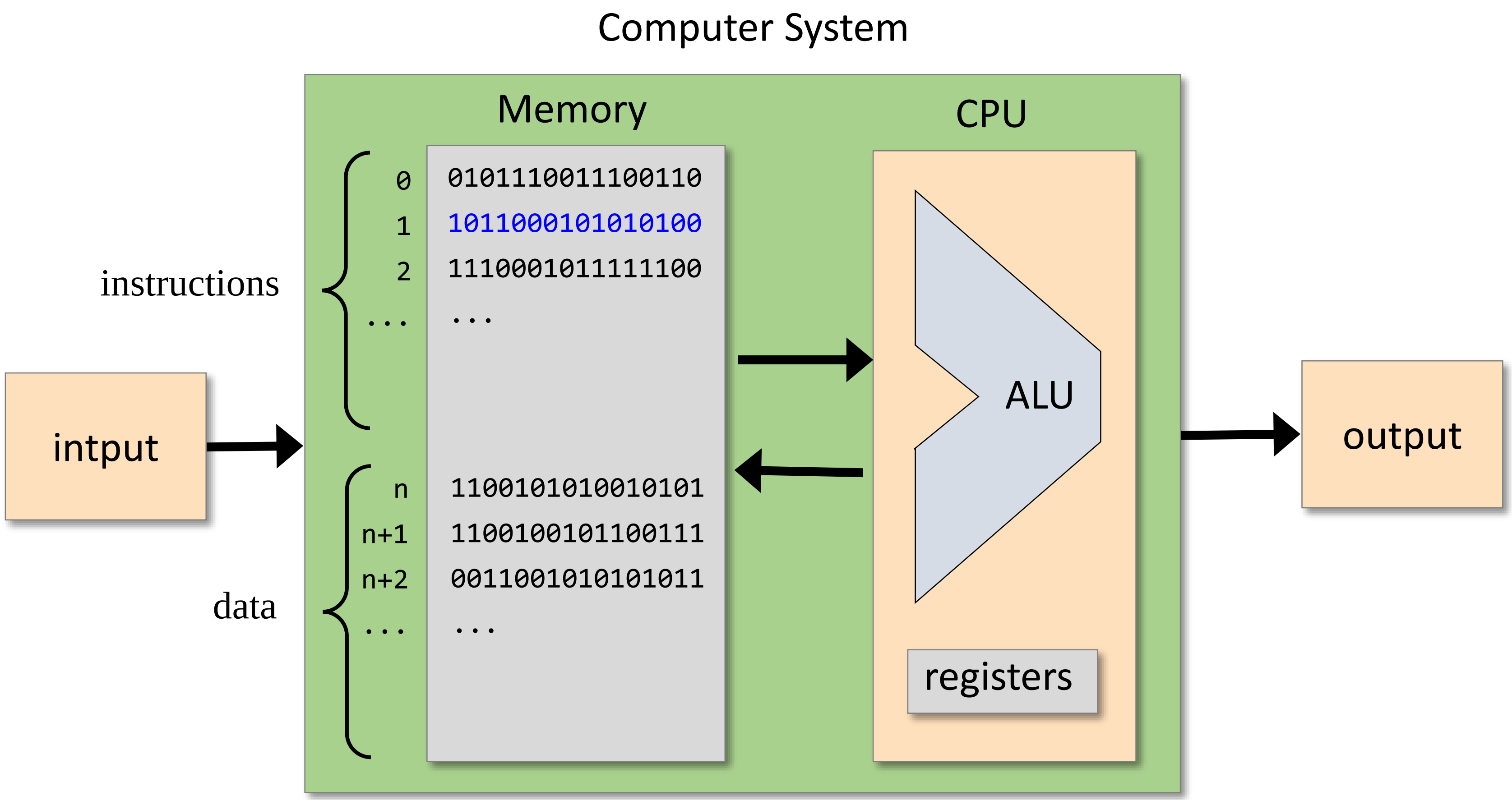
# Stored program concept



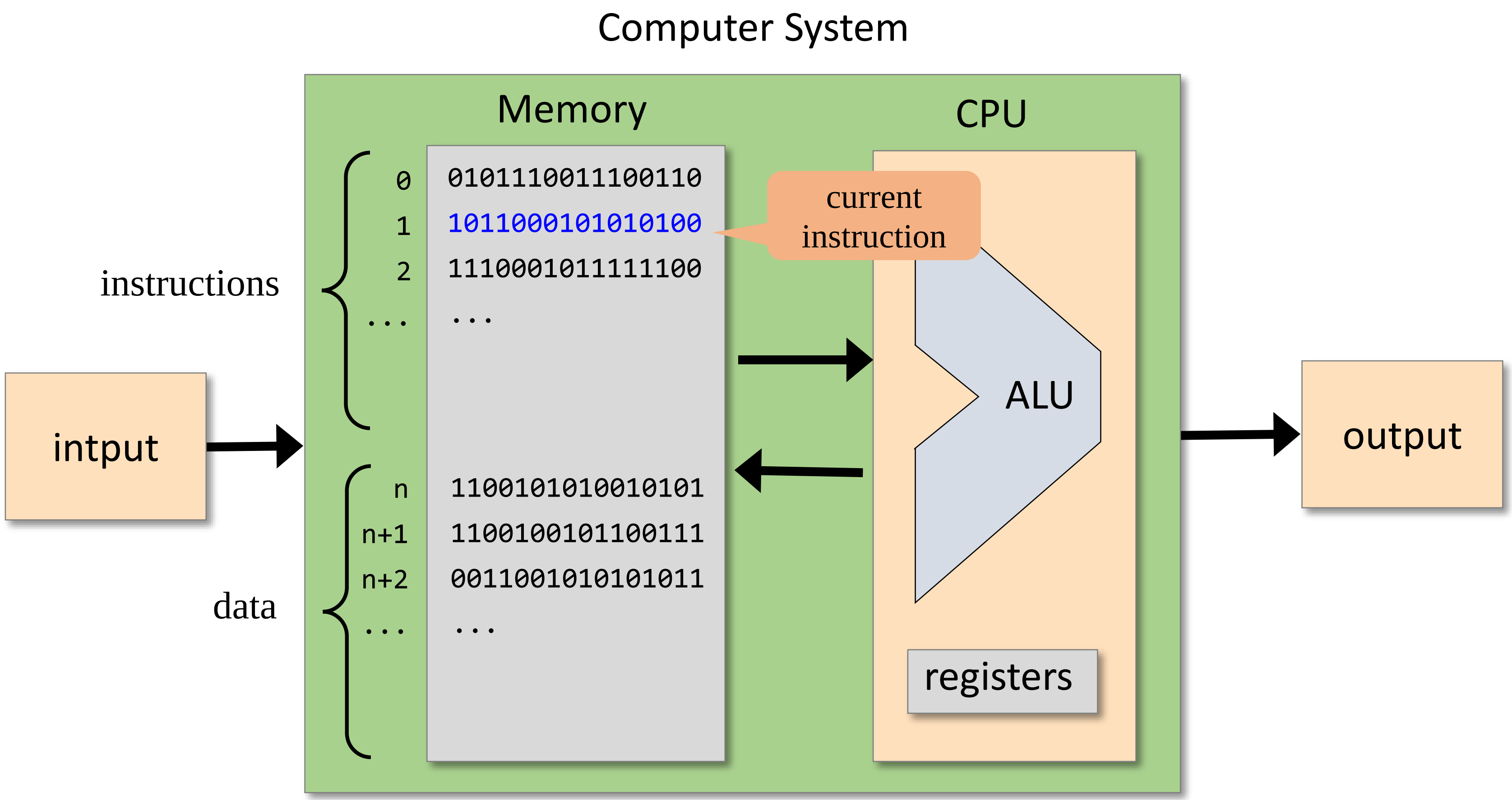
# Stored program concept



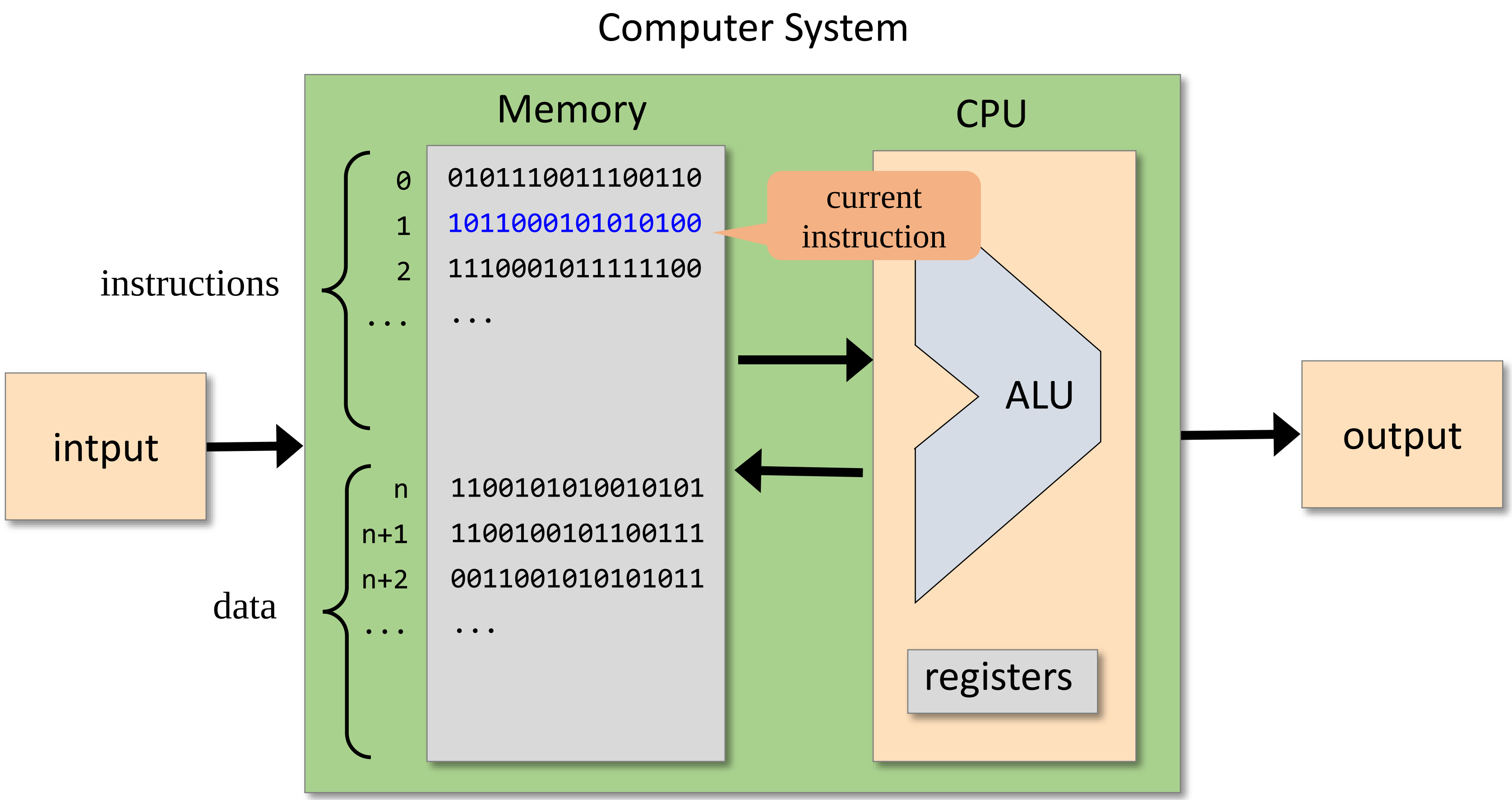
# Machine language



# Machine language

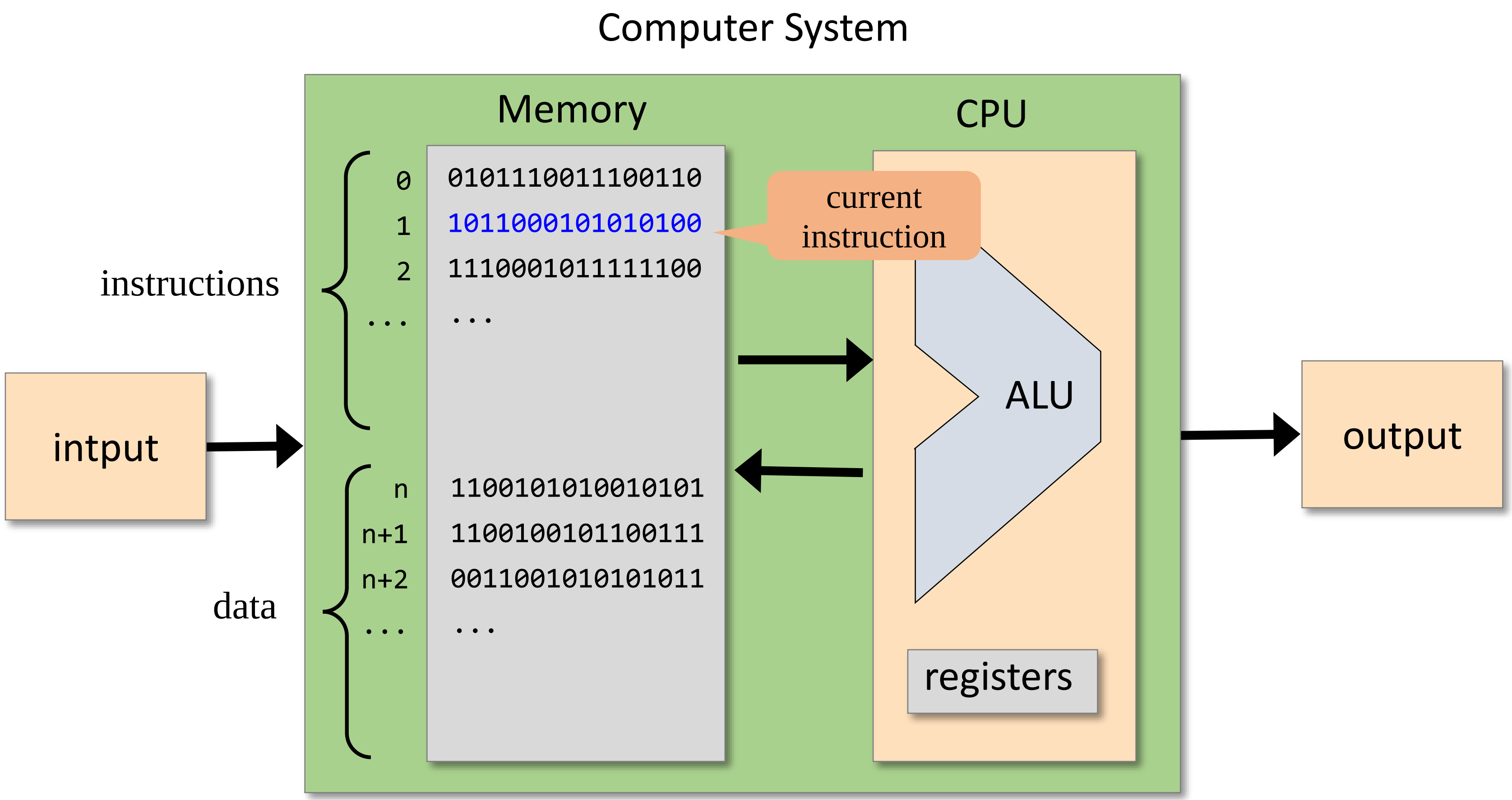


# Machine language




Handling instructions:

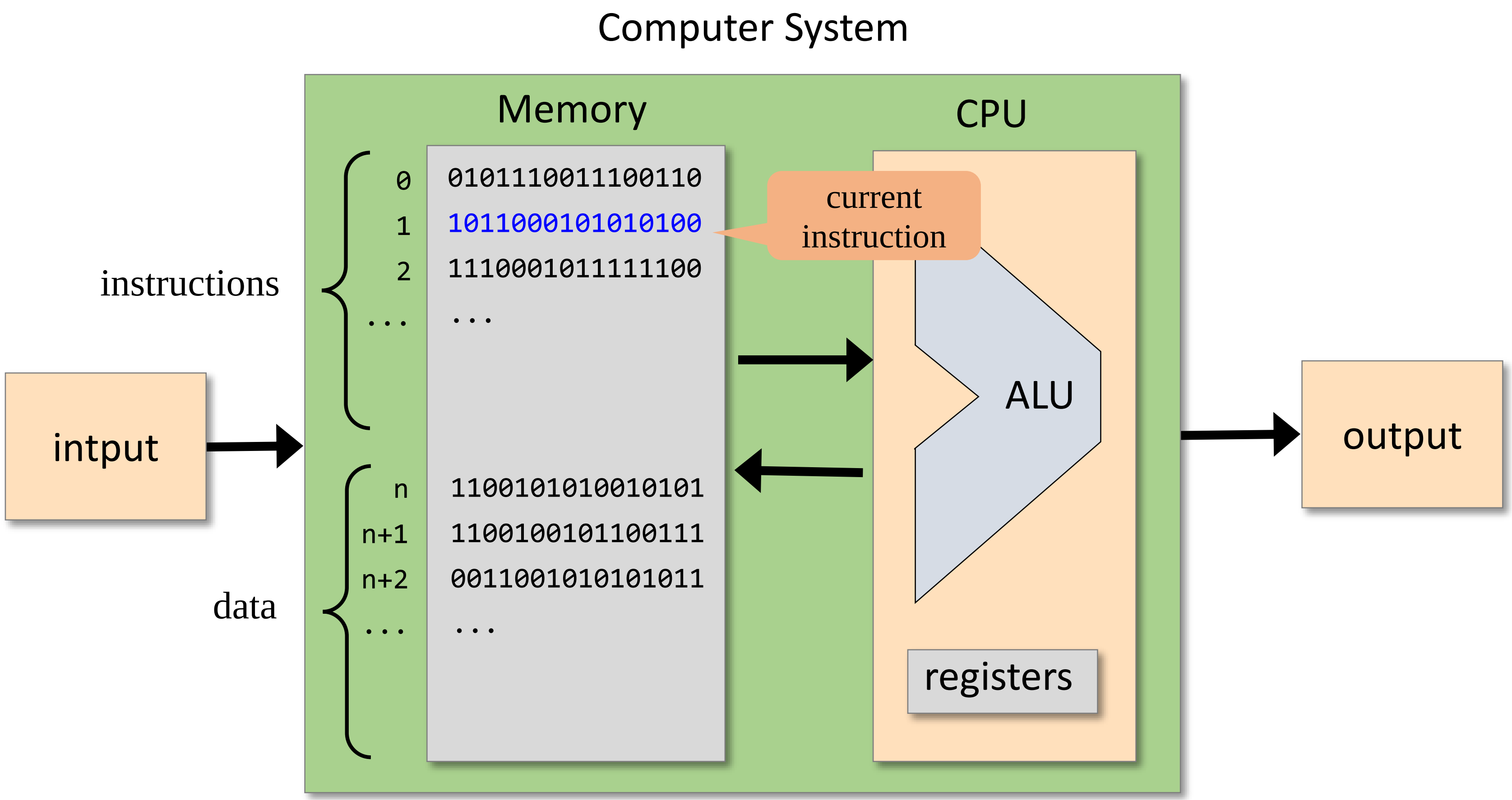
# Machine language




Handling instructions:

- `1011` means "addition"

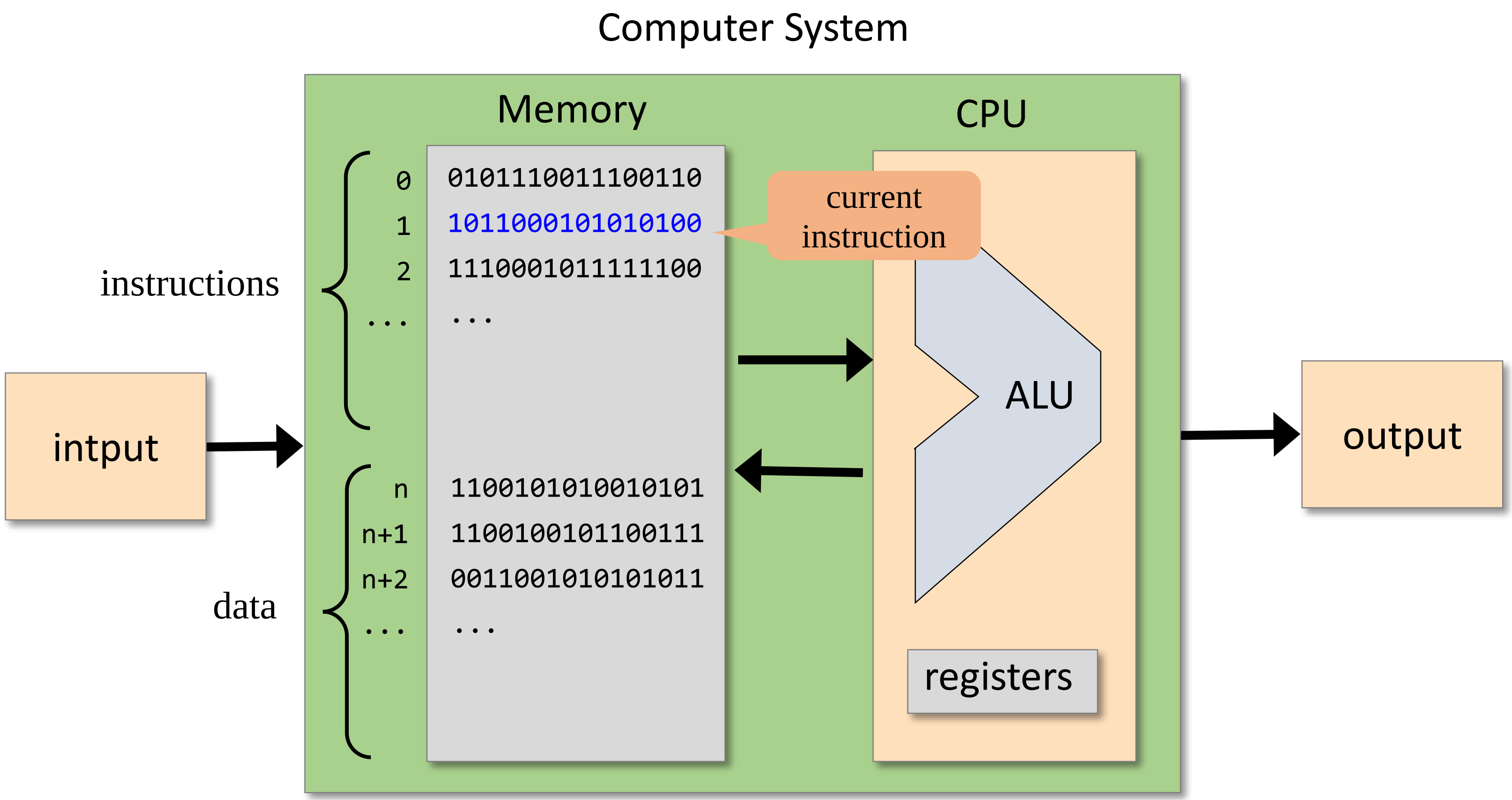
# Machine language




<u>Handling instructions:</u>

- `1011` means "addition" — operation
- `000101010100` means "operate on memory address 340" — addressing

# Machine language




<u>Handling instructions:</u>

- `1011` means "addition" — operation
- `000101010100` means "operate on memory address 340" — addressing
- Next we have to execute the instruction at address 2 — control

# Compilation

high-level program

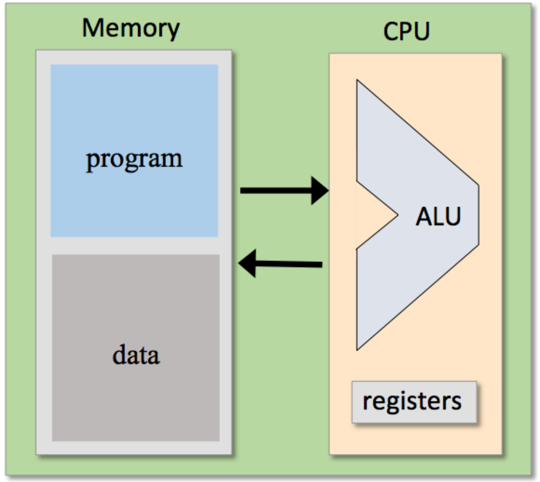
```
while (n < 100) {
    sum += arr[i];
    n++
}
```

compile

machine language

```
0101111100111100
1010101010101010
1101011010101010
1001101010010101
1101010010101010
1110010100100100
0011001010010101
1100100111000100
1100011001100101
0010111001010101
...
```

load and execute



# Mnemonics

Instruction:

| 1011 | 000011 | 000010 |
|---|---|---|
| add | R3 | R2 |

sample instruction

**Interpretation 1:**

- The symbolic form `add R3 R2` doesn't really exist
- It is just a convenient mnemonic that can be used to present machine language instructions to humans

# Mnemonics

Instruction:

| 1011 | 00001 1 | 000010 |
|---|---|---|
| add | R3 | R2 |

sample instruction

**Interpretation 2:**

- Allow humans to write symbolic machine language instructions, using *assembly language*
- Use an *assembler* program to translate the symbolic code into binary form.

# Symbols

Instruction:

`1011 0001 10000001`

add 1 Mem[129]

Assembly:

```
add 1, Mem[129]
```

```
add 1, index
```

Friendlier syntax:
we assume that `index`
stands for `Mem[129]`

The assembler will resolve the symbol `index` into a specific address.

# Machine Language: lecture plan

- ✓ Machine languages
- ➡ Basic elements
- The Hack computer and machine language
- The Hack language specification
- Input / Output
- Hack programming
- Project 4 overview

# Machine language

- Specification of the hardware/software interface:
  - What supported operations?
  - What do they operate on?
  - How is the program controlled?

- Usually in close correspondence to the hardware architecture
  - But not necessarily so

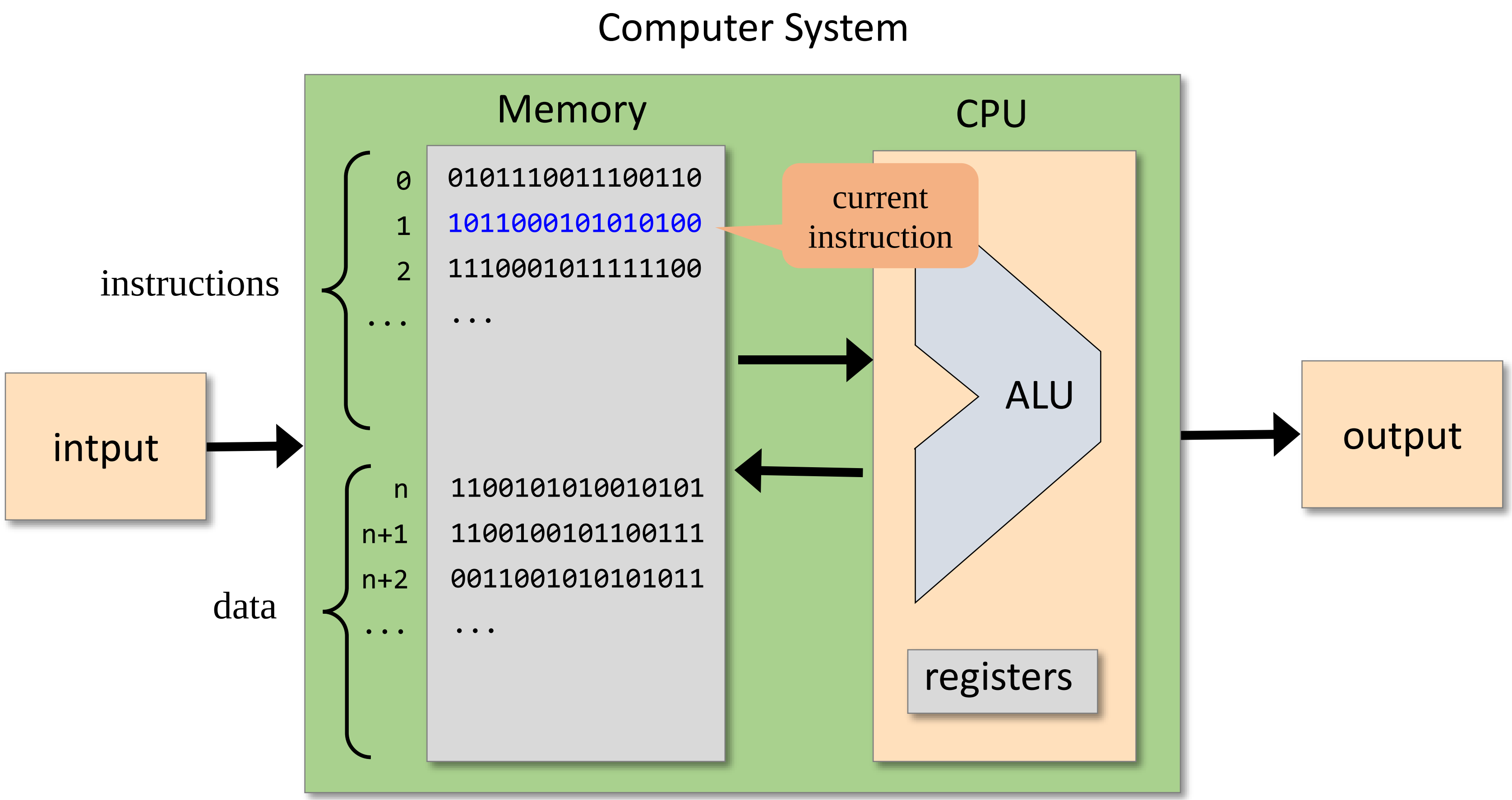
- Cost-performance tradeoffs:
  - Silicon area
  - Time to complete instruction.

# Machine operations

- Usually correspond to the operations that the hardware is designed to support:
  - Arithmetic operations: add, subtract, …
  - Logical operations: and, or, …
  - Flow control: “goto instruction *n*”
    “if (condition) then goto instruction *n*”

- Differences between machine languages:
  - Instruction set richness (division? bulk copy? …)
  - Data types (word width, floating point…).

# Addressing



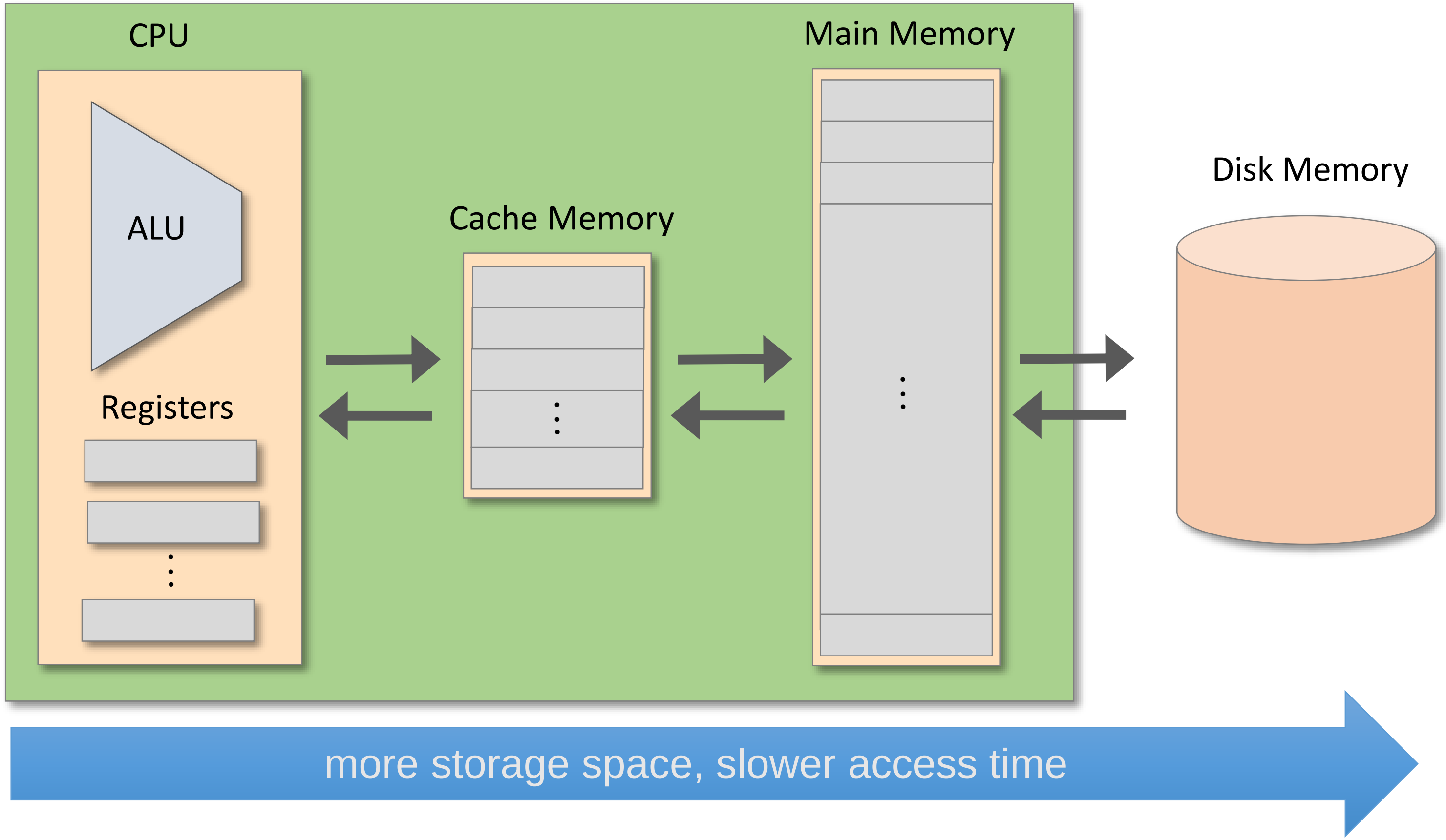

How does the language allow us to specify on which data the instruction should operate?

# Memory hierarchy

- Accessing a memory location is expensive
  - Need to supply a long address
  - Getting the memory contents into the CPU takes time

# Memory hierarchy

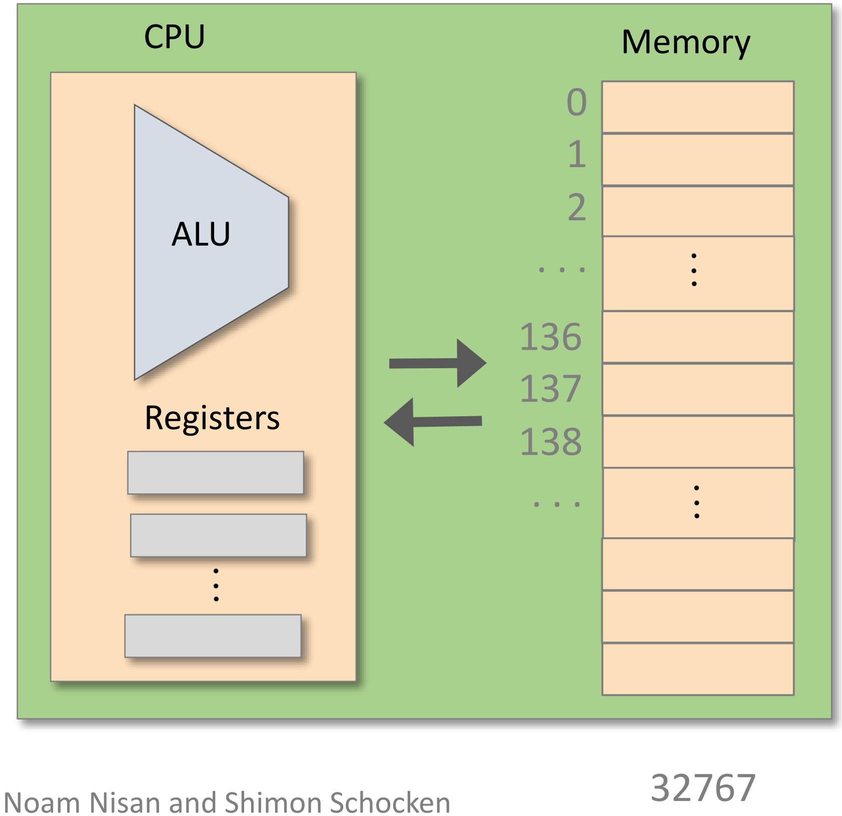
- Accessing a memory location is expensive:
  - Need to supply a long address
  - Getting the memory contents into the CPU takes time
- Solution: memory hierarchy:



# Registers

# Registers

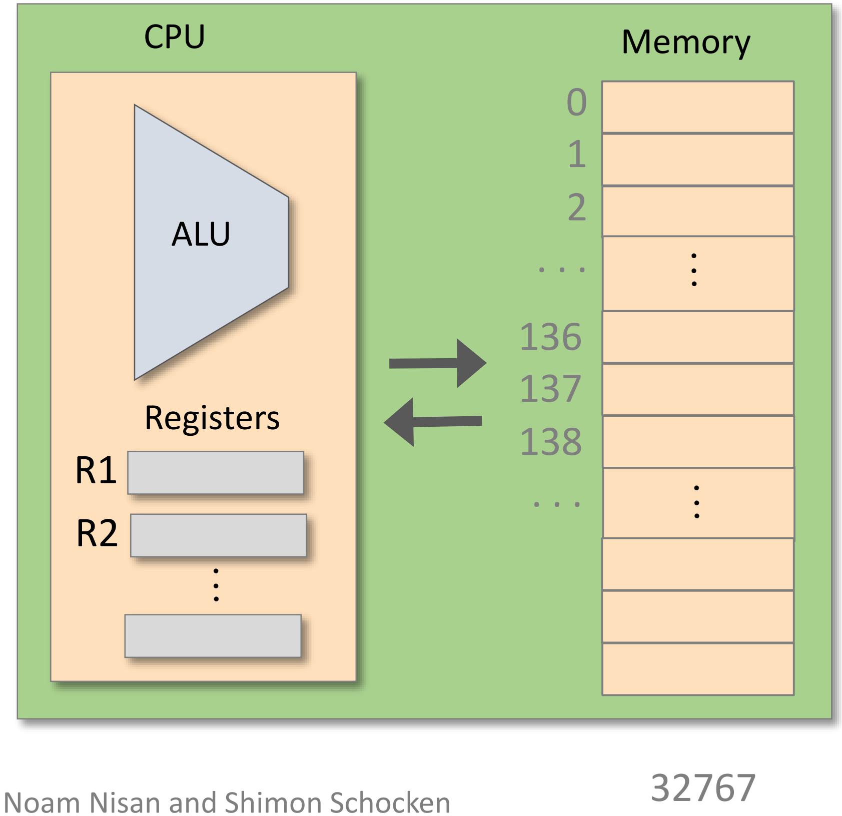
- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language



# Registers

- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

Data registers:

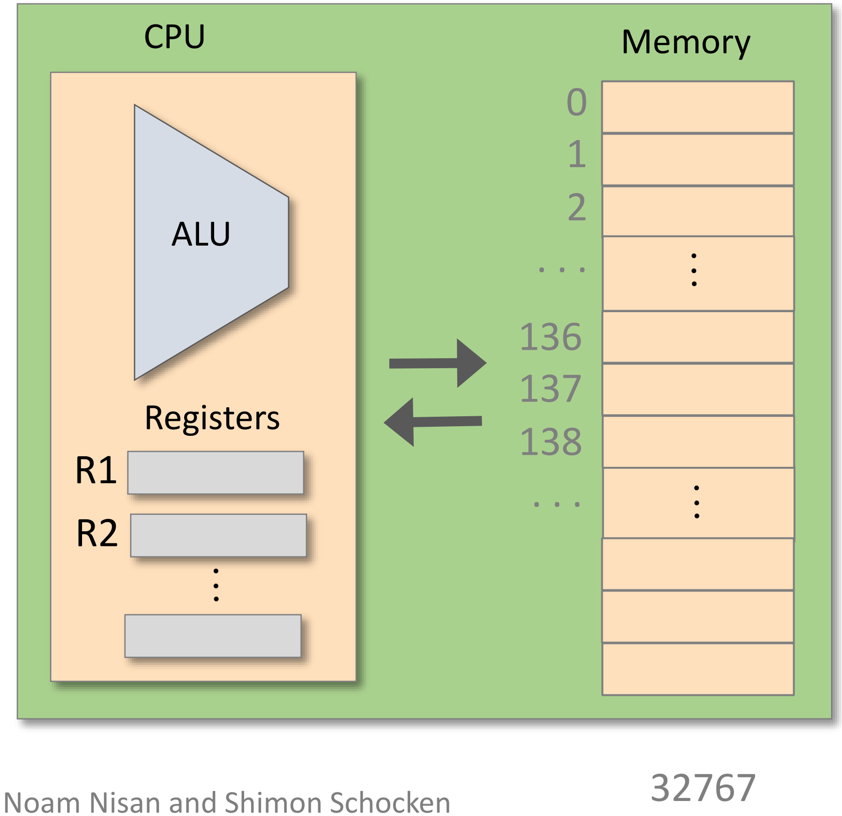


# Registers

- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

<u>Data registers:</u>

```
add R1, R2
```

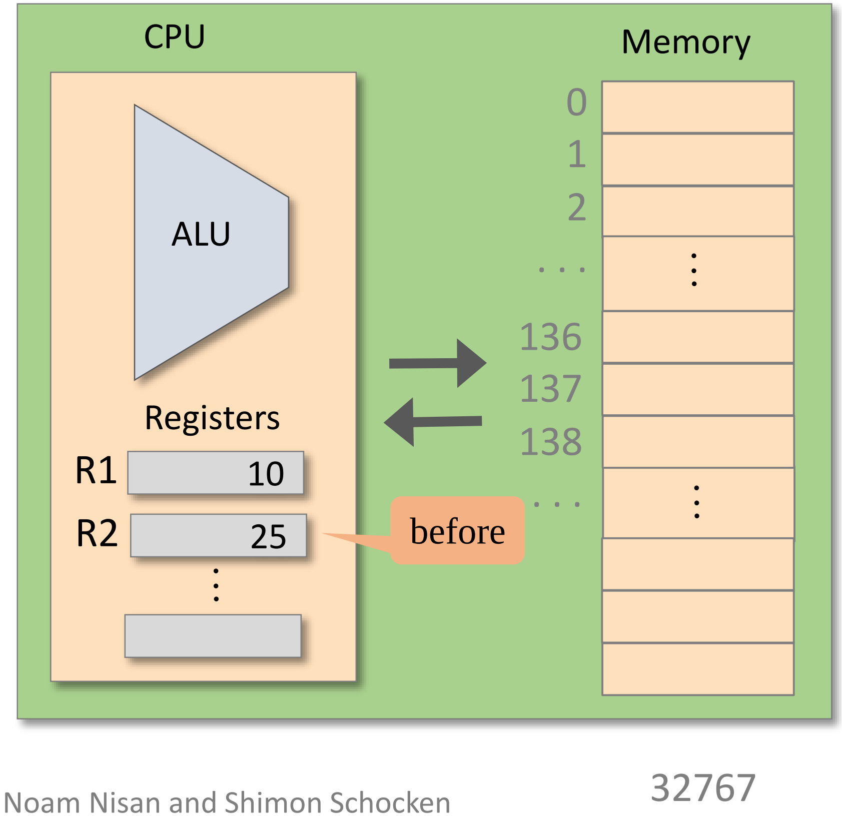


# Registers

- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

<u>Data registers:</u>

```
add R1, R2
```

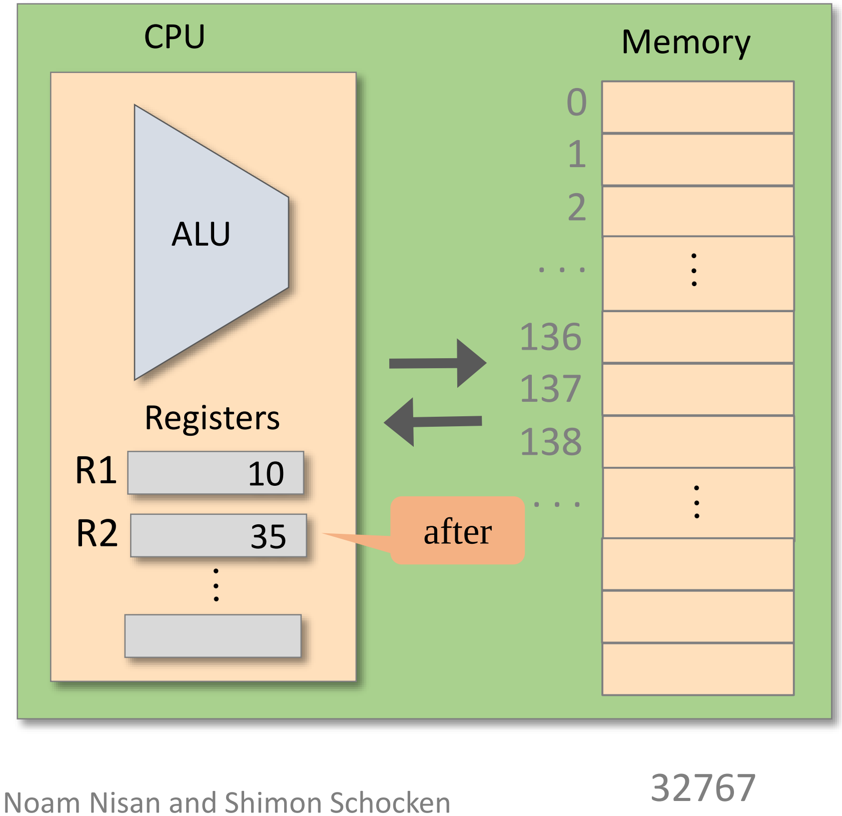


# Registers

- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

<u>Data registers:</u>

```
add R1, R2
```



# Registers

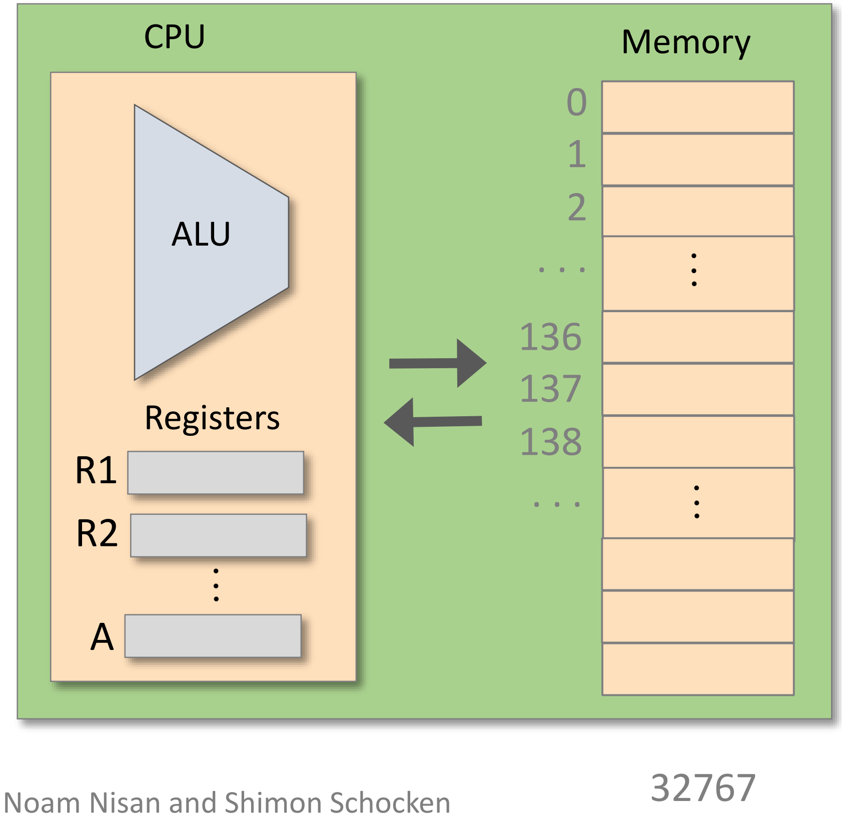
- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

<u>Data registers:</u>

```
add R1, R2
```

<u>Address registers:</u>

```
store R1, @A
```



# Registers

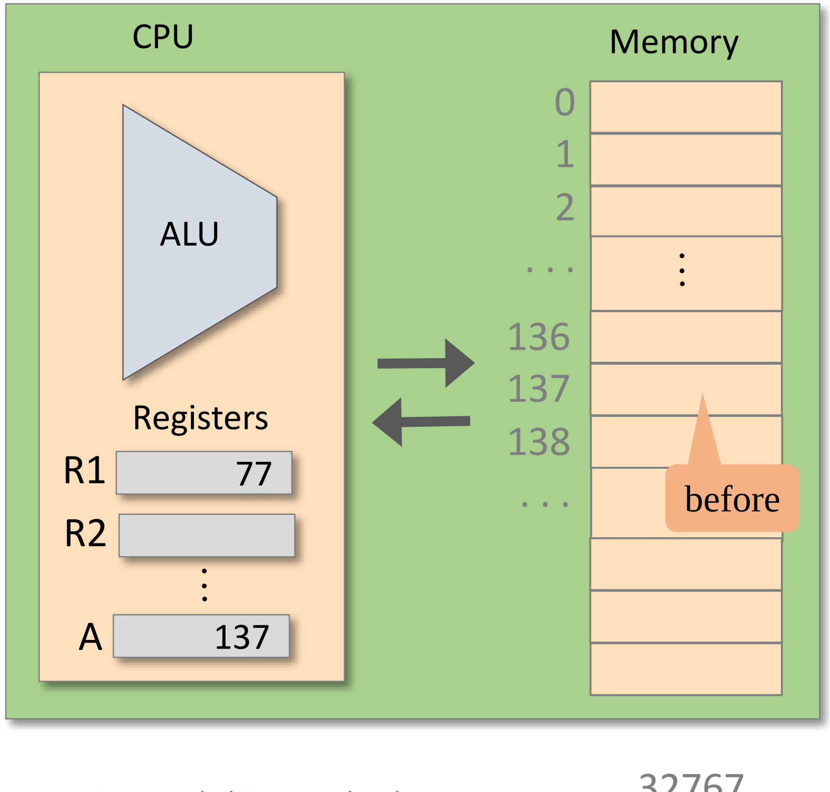
- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

<u>Data registers:</u>

```
add R1, R2
```

<u>Address registers:</u>

```
store R1, @A
```



# Registers

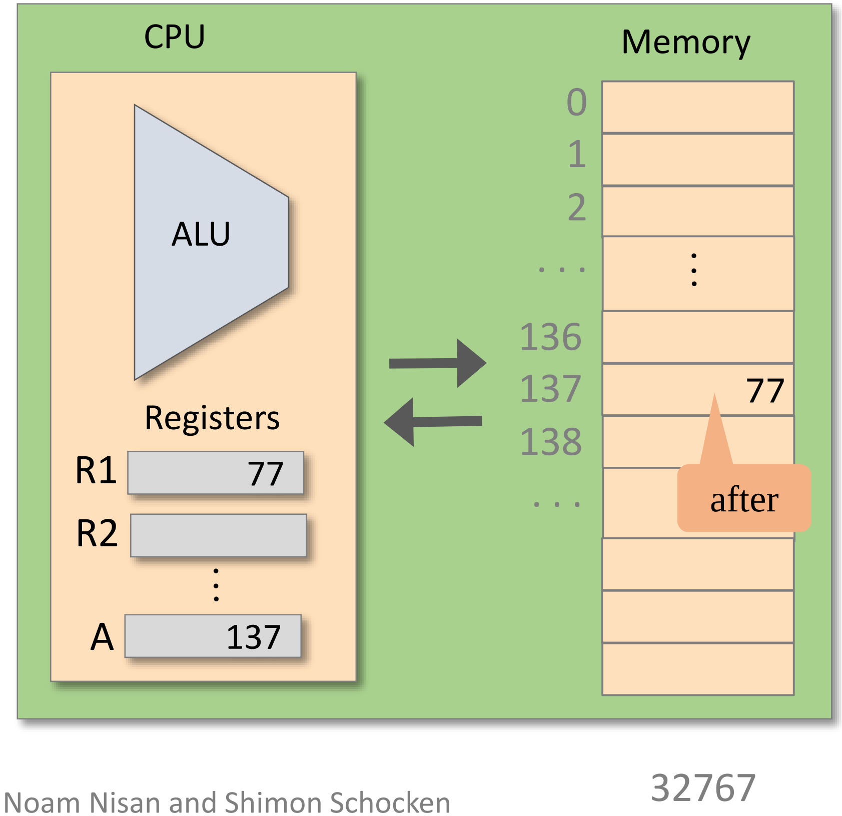
- The CPU typically contains a few, easily accessed, *registers*
- Their number and functions are a central part of the machine language

<u>Data registers:</u>

```
add R1, R2
```

<u>Address registers:</u>

```
store R1, @A
```



# Addressing modes

Register

```
add R1, R2        // R2 ← R2 + R1
```

Direct

```
add R1, M[200]    // Mem[200] ← Mem[200] + R1
```

Indirect

```
add R1, @A        // Mem[A] ← Mem[A] + R1
```

Immediate

```
add 73, R1        // R1 ← R1 + 73
```

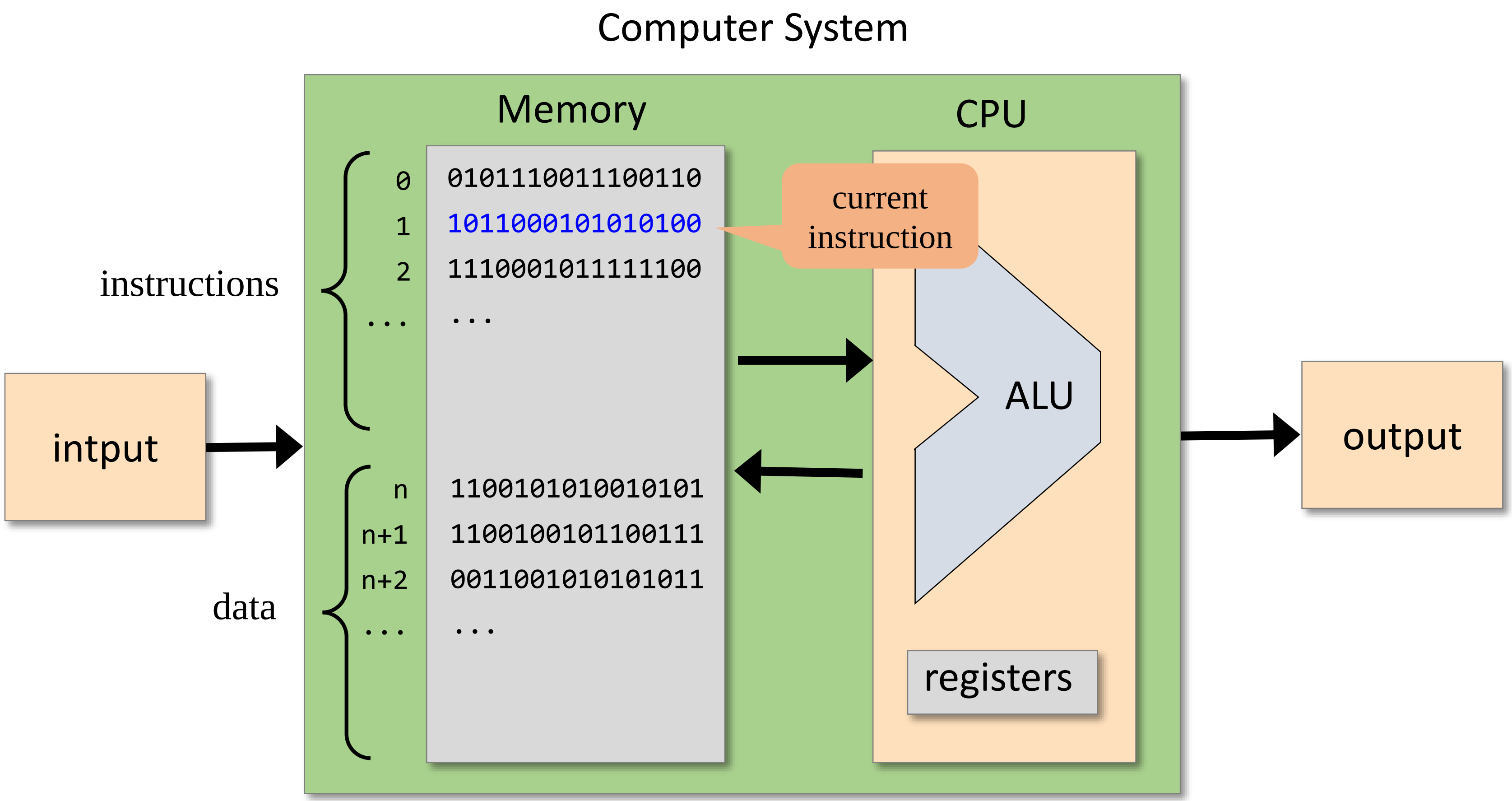
# Input / Output

- Many types of input and output devices:
    - Keyboard, mouse, camera, sensors, printers, screen, sound…

- The CPU needs some agreed-upon protocol to talk to each of them
    - Software drivers realize these protocols

- One general method of interaction uses *memory mapping*:
    - Memory location 12345 holds the direction of the last movement of the mouse
    - Memory location 45678 tells the printer to print single-side or double side
    - Etc.

# Flow control




How does the language allow us to decide, and specify, which instruction to process next?

# Flow control

- Usually the CPU executes machine instructions in sequence

# Flow control

- Usually the CPU executes machine instructions in sequence

- Sometimes we need to “jump” unconditionally to another location, e.g. in order to implement a loop:

Example:

```
101:  load R1,0
102:  add 1, R1
103:  ...
...   // do something with R1 value
...   ...
156:  jmp 102   // goto 102
```

Symbolic version:

```
   load R1,0
LOOP:
   add 1, R1
   ...
   // do something with R1 value
   ...
   jmp LOOP   // goto loop
```

# Flow control

- Usually the CPU executes machine instructions in sequence

- Sometimes we need to “jump” unconditionally to another location, e.g. in order to implement a loop

- Sometimes we need to jump only if some condition is met:

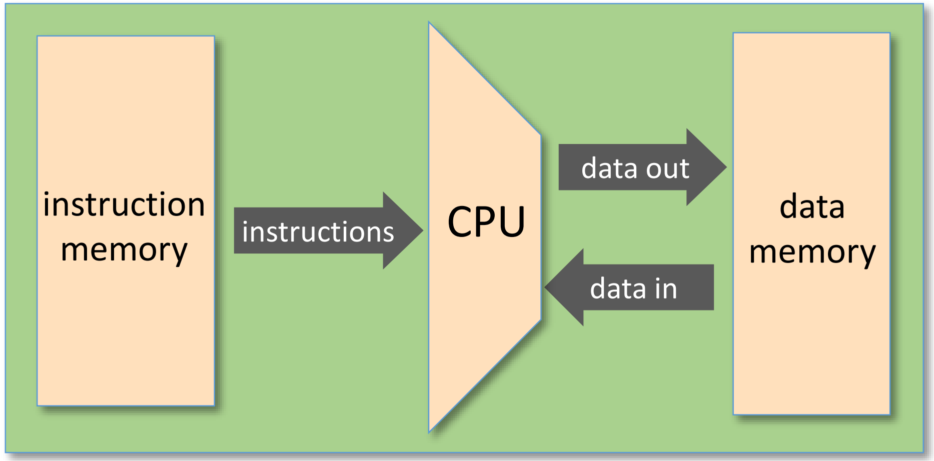
Example:

```
    jgt R1, 0, CONT   // if R1>0 jump to CONT
    sub R1, 0, R1     // R1 ← (0 - R1)
  CONT:
     ...
    // Do something with positive R1
```

# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ➡ The Hack computer and machine language
- The Hack language specification
- Input / Output
- Hack programming
- Project 4 overview

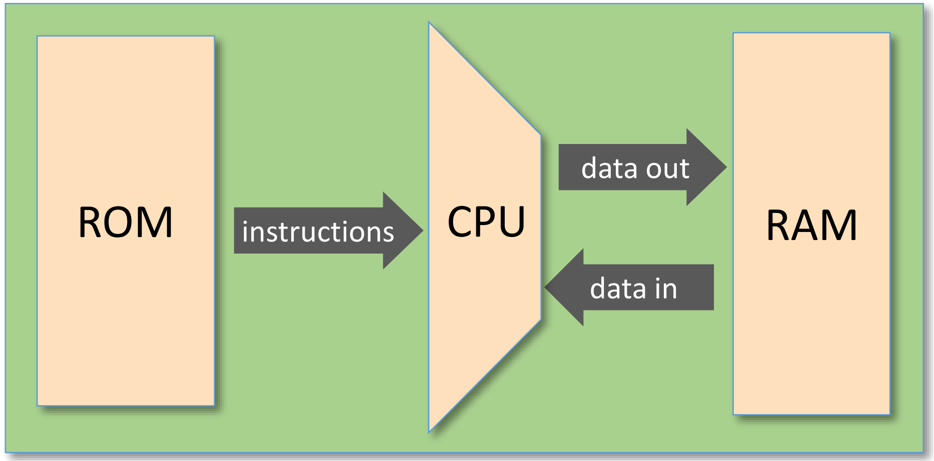
# Hack computer: hardware




A 16-bit machine consisting of:

- Data memory (`RAM`): a sequence of 16-bit registers:
  `RAM[0]`, `RAM[1]`, `RAM[2]`,…
- Instruction memory (`ROM`): a sequence of 16-bit registers:
  `ROM[0]`, `ROM[1]`, `ROM[2]`,…
- Central Processing Unit (`CPU`): performs 16-bit instructions
- Instruction bus / data bus / address buses.

# Hack computer: software

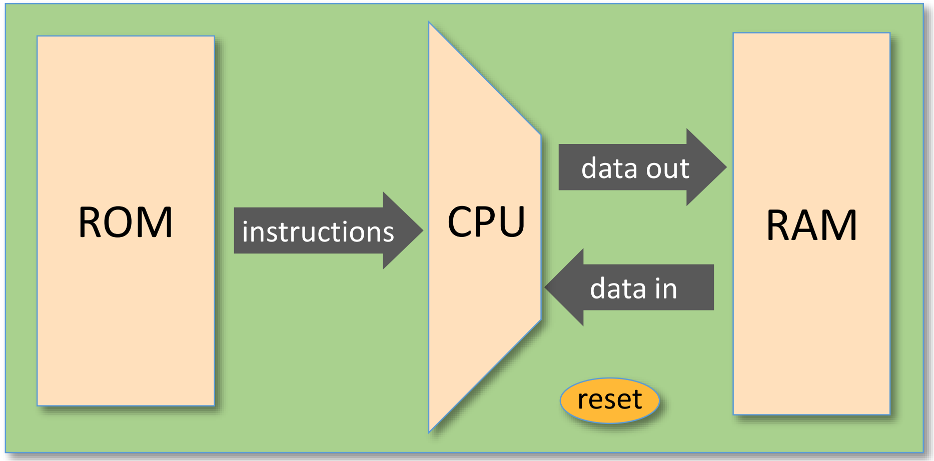



Hack machine language:

- 16-bit A-instructions
- 16-bit C-instructions

Hack program = sequence of instructions written in the
Hack machine language

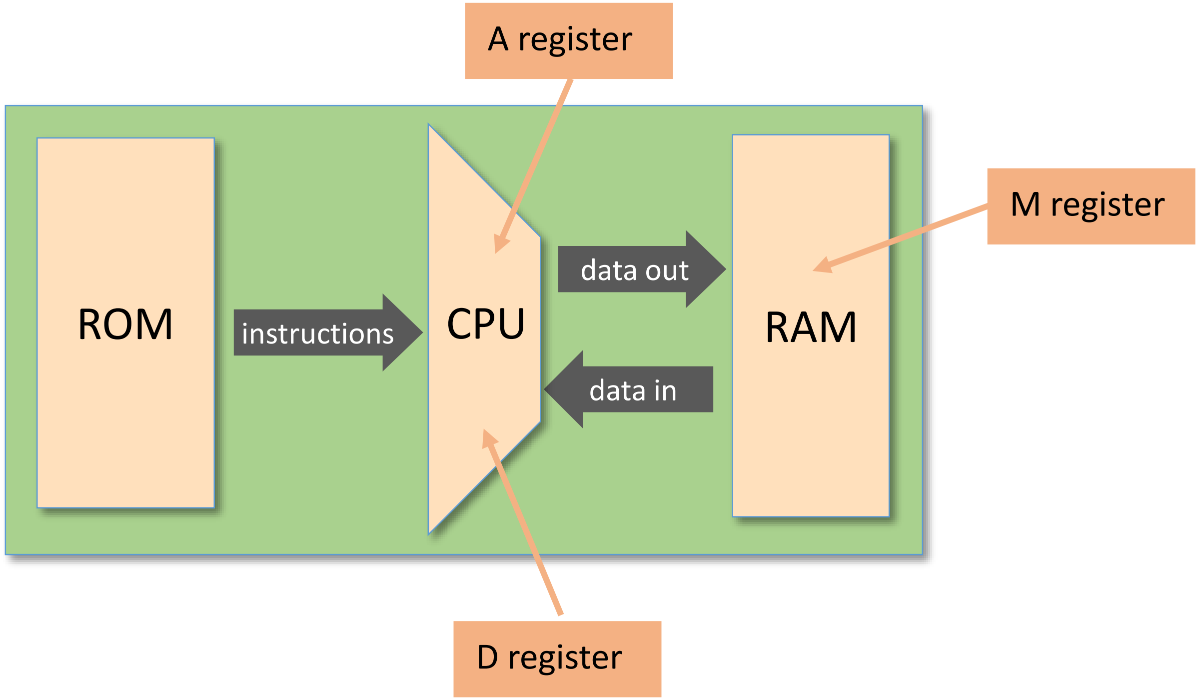
# Hack computer: control




Control:

- The ROM is loaded with a Hack program
- The *reset* button is pushed
- The program starts running

# Hack computer: registers




<u>The Hack machine language recognizes three 16-bit registers:</u>

- `D`: used to store data
- `A`: used to store data / address the memory
- `M`: represents the currently addressed memory register: `M = RAM[A]`

# The A-instruction

<u>Syntax:</u>

*@value*

Where *value* is either:

- a non-negative decimal constant or
- a symbol referring to such a constant (later)

<u>Semantics:</u>

- Sets the `A` register to *value*
- Side effects:
  - `RAM[A]` becomes the selected `RAM` register
  - `ROM[A]` becomes the selected `ROM` register

Example:

```
// Sets A to 17
```

# The A-instruction

Syntax:

```
@value
```

Where *value* is either:

- a non-negative decimal constant  or
- a symbol referring to such a constant (later)

Semantics:

- Sets the `A` register to *value*
- Side effects:
  - `RAM[A]` becomes the selected `RAM` register
  - `ROM[A]` becomes the selected `ROM` register

Example:

```
// Sets A to 17
@17
```

# 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, MD, A, AM, AD, AMD`   (`M` refers to `RAM[A]`)

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

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp jump* `0`) is true, jumps to execute the instruction at `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, MD, A, AM, AD, AMD` (`M` refers to `RAM[A]`)

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

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp jump* `0`) is true, jumps to execute the instruction at `ROM[A]`

Example:

```
// Sets the D register to -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, MD, A, AM, AD, AMD` (`M` refers to `RAM[A]`)

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

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp* *jump* `0`) is true, jumps to execute the instruction at `ROM[A]`

Example:

```
// Sets the D register to -1
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, MD, A, AM, AD, AMD
```

(`M` refers to `RAM[A]`)

*jump* =

```
null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP
```

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp jump* `0`) is true, jumps to execute the instruction at `ROM[A]`

Example:

```
// Sets RAM[300] to the value of the D register plus 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, MD, A, AM, AD, AMD
```

(`M` refers to `RAM[A]`)

*jump* =

```
null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP
```

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp jump* `0`) is true, jumps to execute the instruction at `ROM[A]`

Example:

```
// Sets RAM[300] to the value of the D register plus 1
@300    // A = 300
M=D+1   // RAM[300] = 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, MD, A, AM, AD, AMD
```

(`M` refers to `RAM[A]`)

*jump* =

```
null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP
```

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp jump* `0`) is true, jumps to execute the instruction at `ROM[A]`

Example:

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

# 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, MD, A, AM, AD, AMD
```
(`M` refers to `RAM[A]`)

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

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp jump* `0`) is true, jumps to execute the instruction at `ROM[A]`

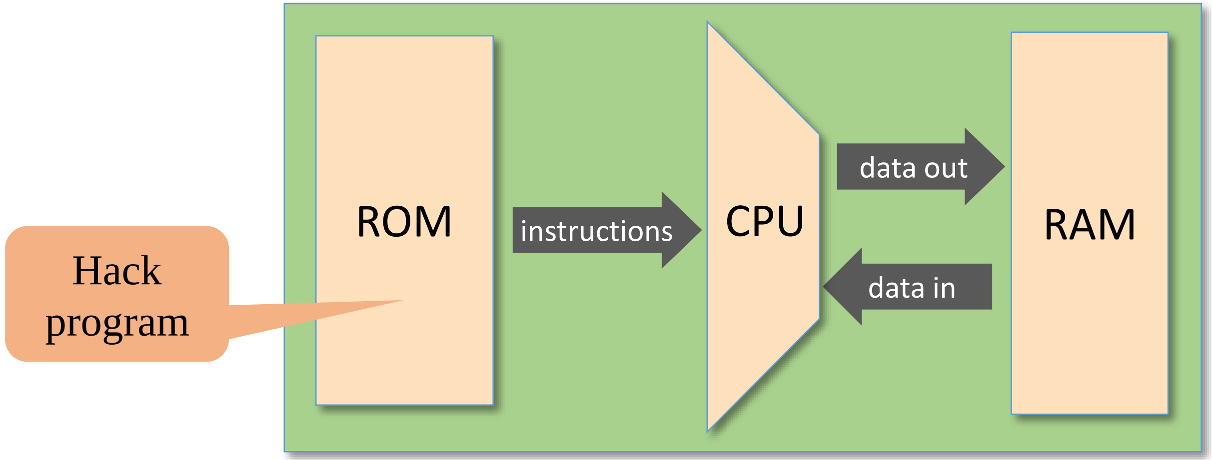
Example:

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

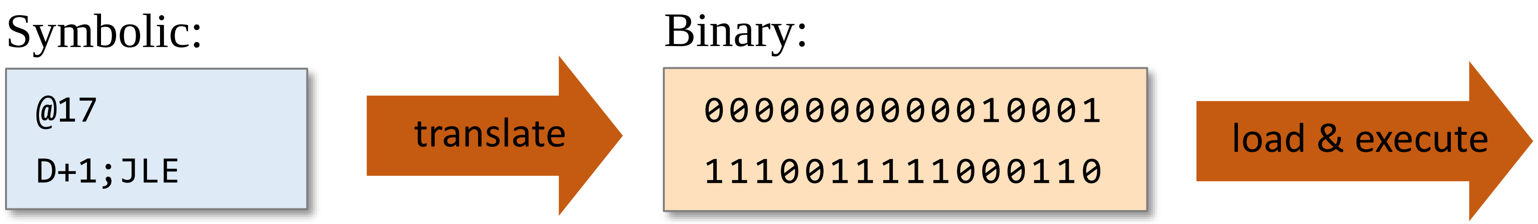
# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ✓ The Hack computer and machine language
- ➡ The Hack language specification
- Input / Output
- Hack programming
- Project 4 overview

# Hack machine language




Two ways to express the same semantics:



# A-instruction specification

Semantics: Sets the `A` register to *value*

Symbolic syntax:

`@`*value*

Example:

```
@21
```

sets `A` to 21

Where *value* is either:

- a non-negative decimal constant ≤ 65535 ($=2^{15}-1$) or
- a symbol referring to a constant (later)

Binary syntax:

`0`*value*

Example:

```
0000000000010101
```

opcode signifying an A-instruction

sets `A` to 21

Where *value* is a 15-bit binary constant

# C-instruction specification

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

Binary syntax: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

# C-instruction specification

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

Binary syntax: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

opcode

not used

*comp* bits

*dest* bits

*jump* bits

# C-instruction specification

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

Binary syntax: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

| *comp* | | c1 | c2 | c3 | c4 | c5 | c6 |
|---|---|---|---|---|---|---|---|
| 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 specification

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

Binary syntax: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

| *dest* | d1 d2 d3 | 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 |
| MD | 0 1 1 | RAM[A] and D register |
| A | 1 0 0 | A register |
| AM | 1 0 1 | A register and RAM[A] |
| AD | 1 1 0 | A register and D register |
| AMD | 1 1 1 | A register, RAM[A], and D register |

# C-instruction specification

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

Binary syntax: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

| *jump* | j1 j2 J3 | effect |
|---|---|---|
| null | 0 0 0 | no jump |
| JGT | 0 0 1 | if out>0 jump |
| JEQ | 0 1 0 | if out=0 jump |
| JGE | 0 1 1 | if out≥0 jump |
| JLT | 1 0 0 | if out<0 jump |
| JNE | 1 0 1 | if out≠0 jump |
| JLE | 1 1 0 | if out≤0 jump |
| JMP | 1 1 1 | unconditional jump |

# C-instruction specification

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

Binary syntax: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

| comp | | c1 | c2 | c3 | c4 | c5 | c6 |
|---|---|---|---|---|---|---|---|
| 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 | | | | | | |

| dest | d1 | d2 | d3 | 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 |
| MD | 0 | 1 | 1 | RAM[A] and D register |
| A | 1 | 0 | 0 | A register |
| AM | 1 | 0 | 1 | A register and RAM[A] |
| AD | 1 | 1 | 0 | A register and D register |
| AMD | 1 | 1 | 1 | A register, RAM[A], and D register |

| jump | j1 | j2 | j3 | effect: |
|---|---|---|---|---|
| null | 0 | 0 | 0 | no jump |
| JGT | 0 | 0 | 1 | if out > 0 jump |
| JEQ | 0 | 1 | 0 | if out = 0 jump |
| JGE | 0 | 1 | 1 | if out ≥ 0 jump |
| JLT | 1 | 0 | 0 | if out < 0 jump |
| JNE | 1 | 0 | 1 | if out ≠ 0 jump |
| JLE | 1 | 1 | 0 | if out ≤ 0 jump |
| JMP | 1 | 1 | 1 | Unconditional jump |

Examples:

Symbolic: `MD=D+1`

Binary: `1110011111011000`

# C-instruction specification

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

Binary syntax: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

| *comp* | | c1 | c2 | c3 | c4 | c5 | c6 |
|---|---|---|---|---|---|---|---|
| 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 | | | | | | |

| *dest* | d1 | d2 | d3 | 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 |
| MD | 0 | 1 | 1 | RAM[A] and D register |
| A | 1 | 0 | 0 | A register |
| AM | 1 | 0 | 1 | A register and RAM[A] |
| AD | 1 | 1 | 0 | A register and D register |
| AMD | 1 | 1 | 1 | A register, RAM[A], and D register |

| *jump* | j1 | j2 | j3 | effect: |
|---|---|---|---|---|
| null | 0 | 0 | 0 | no jump |
| JGT | 0 | 0 | 1 | if out > 0 jump |
| JEQ | 0 | 1 | 0 | if out = 0 jump |
| JGE | 0 | 1 | 1 | if out ≥ 0 jump |
| JLT | 1 | 0 | 0 | if out < 0 jump |
| JNE | 1 | 0 | 1 | if out ≠ 0 jump |
| JLE | 1 | 1 | 0 | if out ≤ 0 jump |
| JMP | 1 | 1 | 1 | Unconditional jump |

Examples:

Symbolic: M=1

Binary: 1110111111001000

# C-instruction specification

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

Binary syntax: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

| *comp* | | c1 | c2 | c3 | c4 | c5 | c6 |
|---|---|---|---|---|---|---|---|
| 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 | | | | | | |

| *dest* | d1 | d2 | d3 | 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 |
| MD | 0 | 1 | 1 | RAM[A] and D register |
| A | 1 | 0 | 0 | A register |
| AM | 1 | 0 | 1 | A register and RAM[A] |
| AD | 1 | 1 | 0 | A register and D register |
| AMD | 1 | 1 | 1 | A register, RAM[A], and D register |

| *jump* | j1 | j2 | j3 | effect: |
|---|---|---|---|---|
| null | 0 | 0 | 0 | no jump |
| JGT | 0 | 0 | 1 | if out > 0 jump |
| JEQ | 0 | 1 | 0 | if out = 0 jump |
| JGE | 0 | 1 | 1 | if out ≥ 0 jump |
| JLT | 1 | 0 | 0 | if out < 0 jump |
| JNE | 1 | 0 | 1 | if out ≠ 0 jump |
| JLE | 1 | 1 | 0 | if out ≤ 0 jump |
| JMP | 1 | 1 | 1 | Unconditional jump |

Examples:

Symbolic: D+1;JLE

Binary: 1110011111000110

# Hack program

Symbolic code

```
// Computes RAM[1] = 1+...+RAM[0]
// Usage: put a number in RAM[0]
@16   // RAM[16] represents i
M=1   // i = 1
@17   // RAM[17] represents sum
M=0   // sum = 0

@16
D=M
@0
D=D-M
@17    // if i>RAM[0] goto 17
D;JGT

@16
D=M
@17
M=D+M  // sum += i
@16
M=M+1  // i++
@4     // goto 4 (loop)
0;JMP

@17
D=M
@1
M=D    // RAM[1] = sum
@21    // program's end
0;JMP  // infinite loop
```

## Observations:

- Hack program:
  a sequence of Hack instructions
- White space is permitted
- Comments are welcome
- There are better ways to write symbolic Hack programs; stay tuned.

No need to understand ...
we'll review the code later in the lecture.

# Hack programs: symbolic and binary

Symbolic code

```
// Computes RAM[1] = 1+...+RAM[0]
// Usage: put a number in RAM[0]
@16   // RAM[16] represents i
M=1   // i = 1
@17   // RAM[17] represents sum
M=0   // sum = 0

@16
D=M
@0
D=D-M
@17    // if i>RAM[0] goto 17
D;JGT

@16
D=M
@17
M=D+M  // sum += i
@16
M=M+1  // i++
@4     // goto 4 (loop)
0;JMP

@17
D=M
@1
M=D    // RAM[1] = sum
@21    // program's end
0;JMP  // infinite loop
```

translate

Binary code

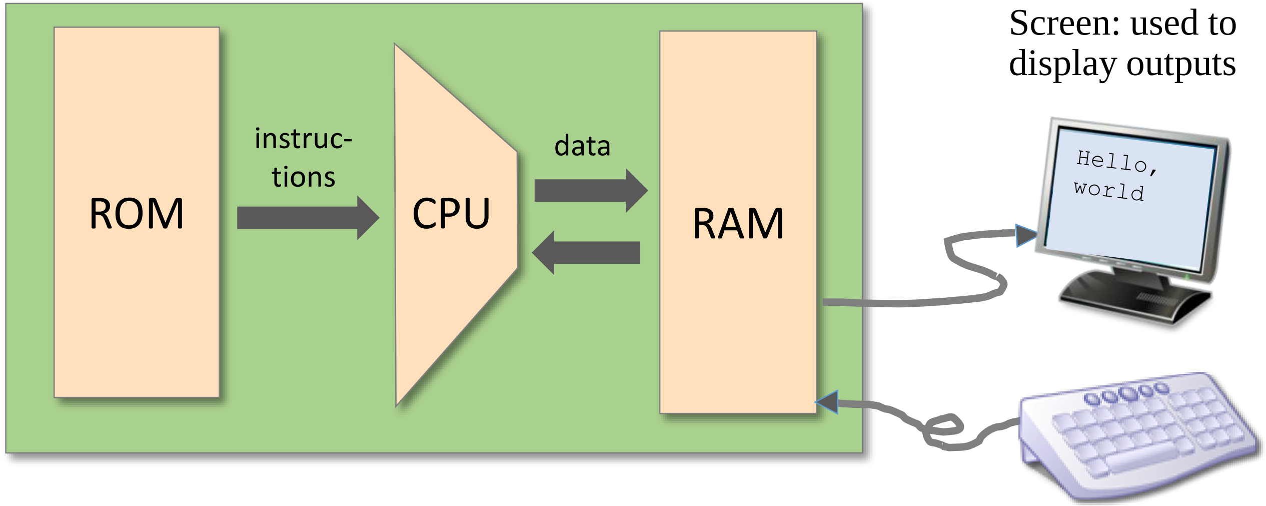
```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010001
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010101
1110101010000111
```

execute

# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ✓ The Hack computer and machine language
- ✓ The Hack language specification
- ➡ Input / Output
- Hack programming
- Project 4 overview

# Input / output




Screen: used to display outputs

Keyboard: used to enter inputs

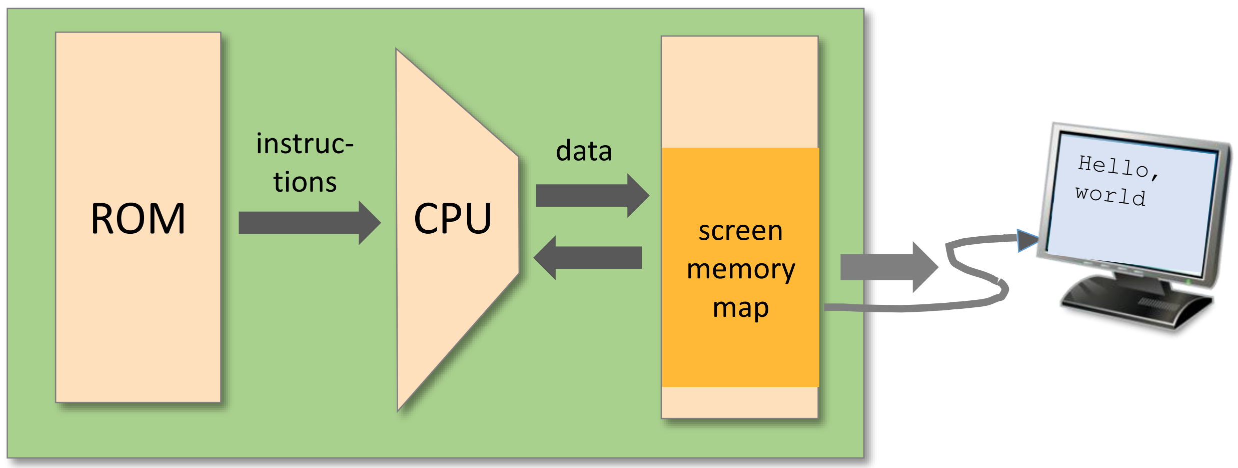
<u>I/O handling</u> (high-level):

Software libraries enabling text, graphics, audio, video, etc.

<u>I/O handling</u> (low-level):

Bits manipulation.

# Memory mapped output



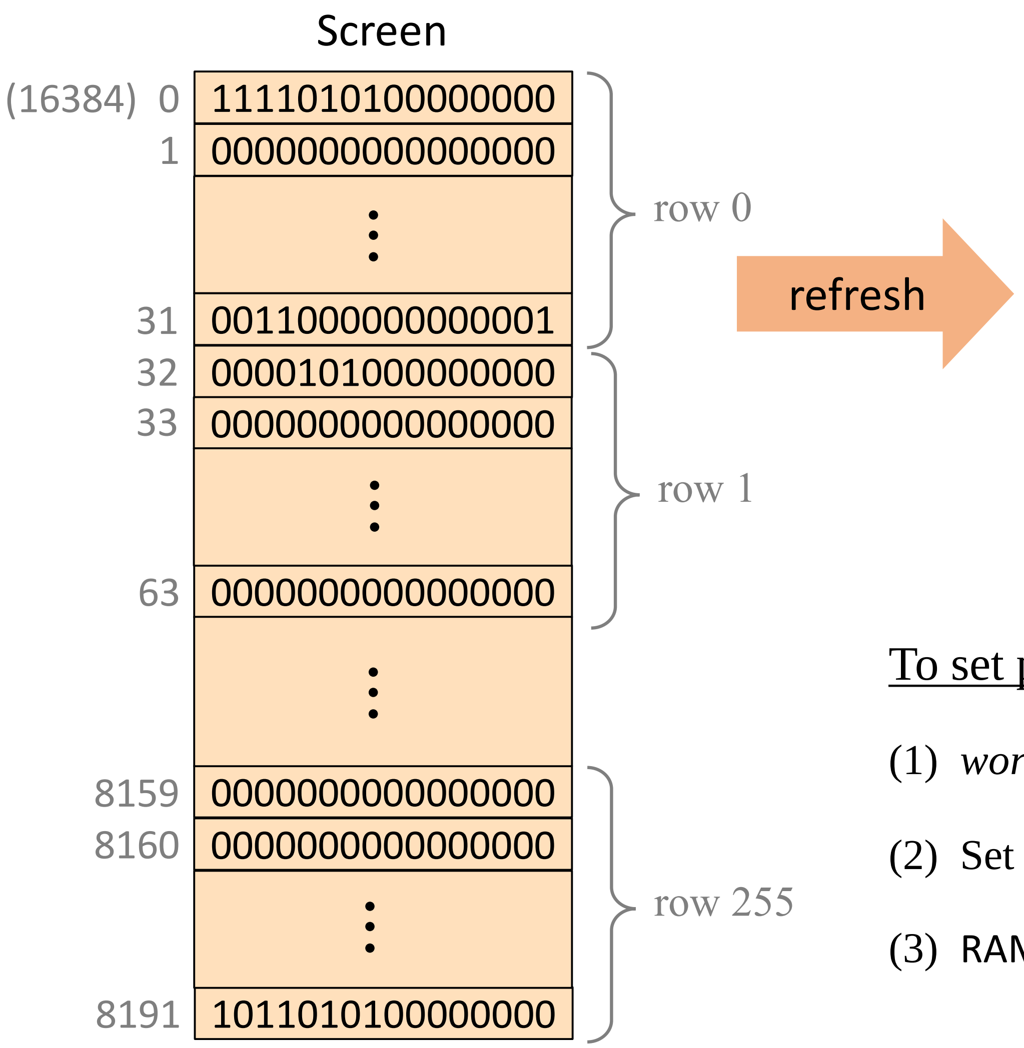

Memory mapped output

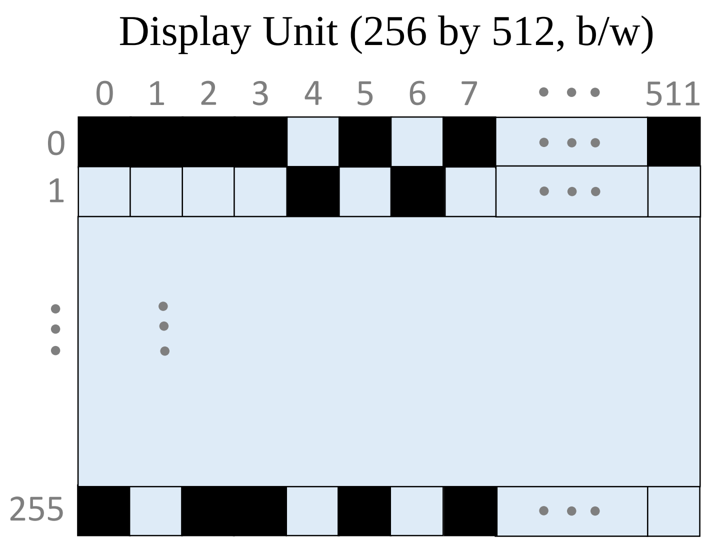
A designated memory area, dedicated to manage a display unit

The physical display is continuously *refreshed* from the memory map, many times per second

Output is effected by writing code that manipulates the screen memory map.

# Memory mapped output




<u>To set pixel (*row*,*col*) on/off:</u>

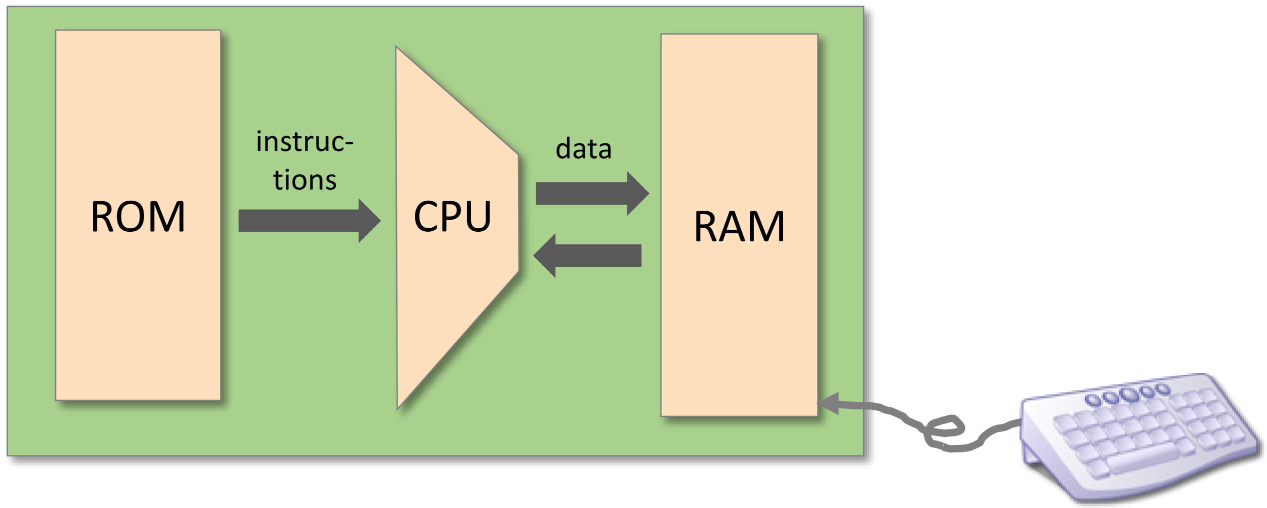
(1) *word* = RAM[16384 + 32**row* + *col*/16]

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

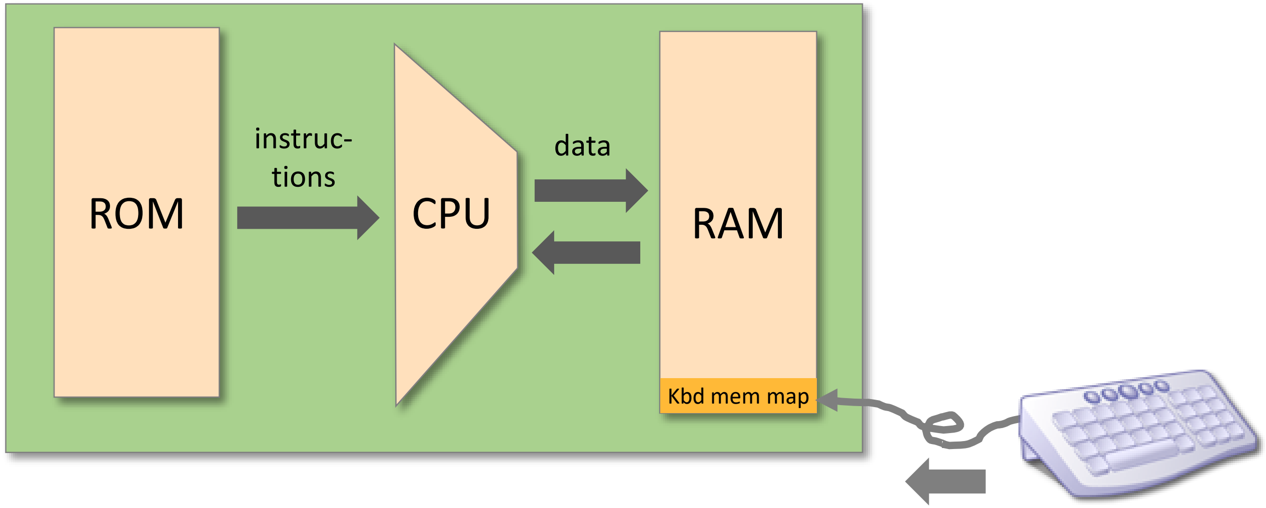
(3) RAM[ *i* ] = *word*

# Memory mapped output

# Input



# Input

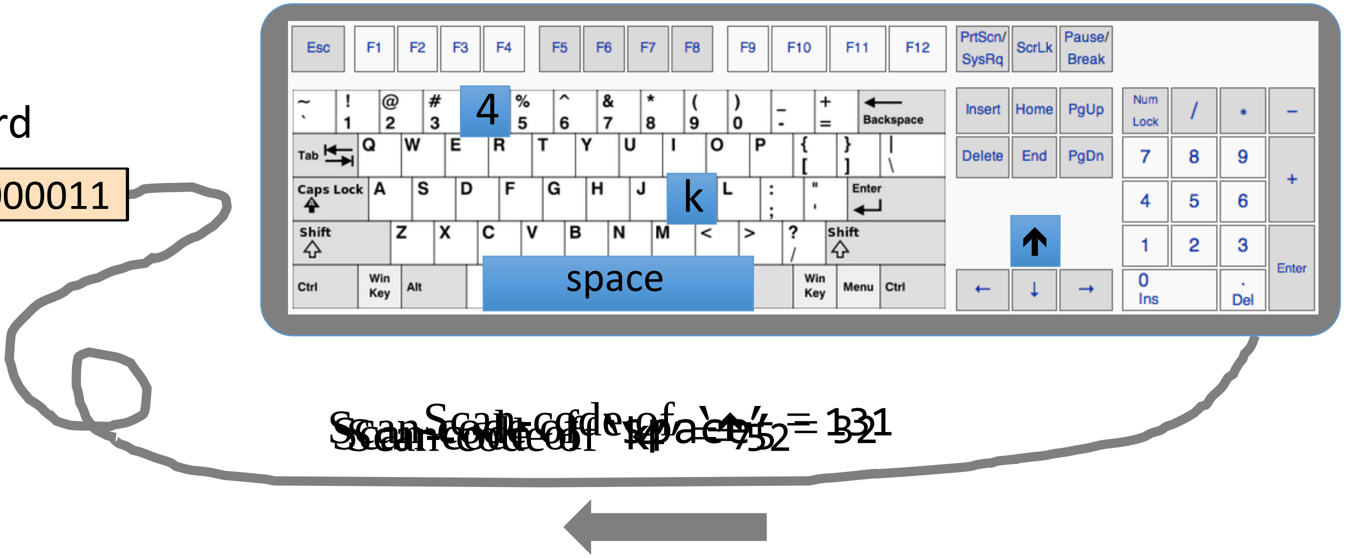



The physical keyboard is associated with a *keyboard memory map*.

# Memory mapped input

Keyboard

0000000010000011

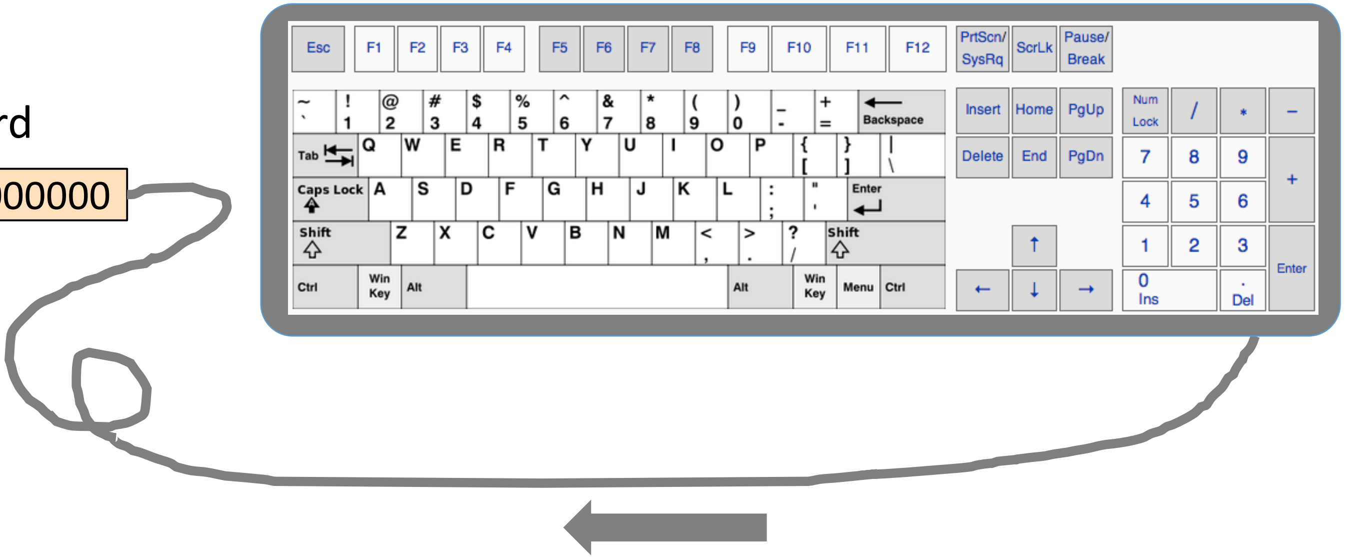


When a key is pressed on the keyboard, the key's *scan code* appears in the *keyboard memory map*

# Memory mapped input

Keyboard

0000000000000000



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

# 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 |

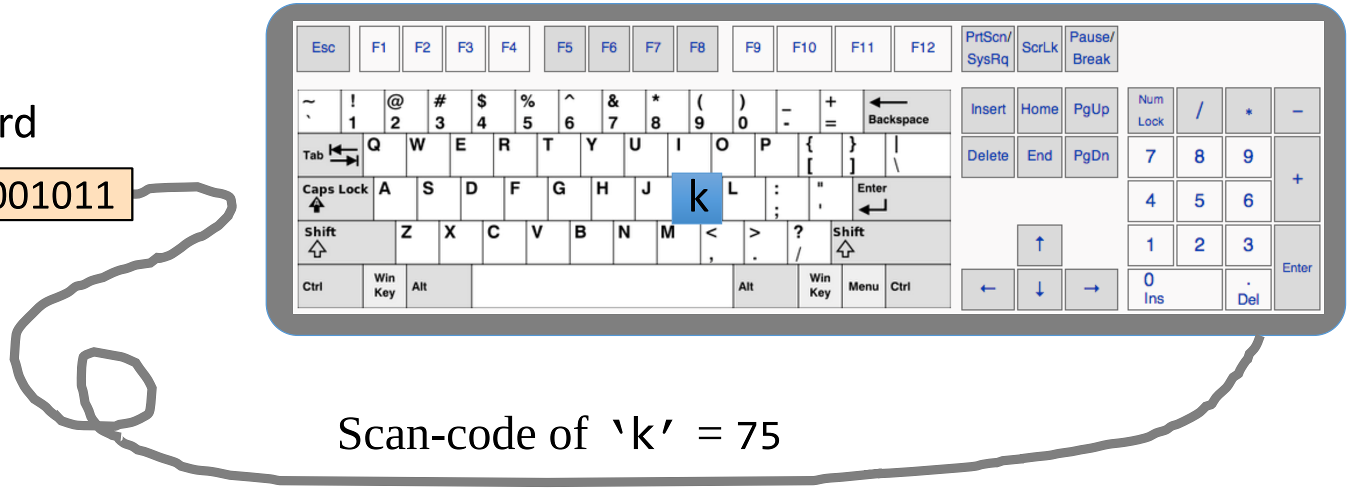
| key | code |
|---|---|
| a | 97 |
| b | 98 |
| c | 99 |
| … | … |
| z | 122 |
| { | 123 |
| \| | 124 |
| } | 125 |
| ~ | 126 |

| 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 |

# Handling the keyboard

Keyboard

0000000001001011




Scan-code of `'k'` = 75

<u>To check which key is currently pressed:</u>

- Probe the contents of the `Keyboard` chip
- In the Hack computer: probe the contents of `RAM[24576]`.

# Handling the keyboard

# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ✓ The Hack computer and machine language
- ✓ The Hack language specification
- ✓ Input / Output
- Hack programming
- Project 4 overview

# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ✓ The Hack computer and machine language
- ✓ The Hack language specification
- ✓ Input / Output
- Hack programming
  - ➡ Part 1: registers and memory
  - ❑ Part 2: branching, variables, iteration
  - ❑ Part 3: pointers, input/output
- Project 4 overview

# Hack assembly language (overview)

A-instruction:

```
@value    // A = value
```

where *value* is either a constant or a symbol referring to such a constant

C-instruction:

```
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, MD, A, AM, AD, AMD
```

(`M` refers to `RAM[A]`)

*jump* =

```
null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP
```

Semantics:

- Computes the value of *comp*
- Stores the result in *dest*
- If the Boolean expression (*comp jump* `0`) is true, jumps to execute the instruction at `ROM[A]`

# Hack assembler

Assembly program

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]
   @R1
   D=M
   @temp
   M=D     // temp = R1

   @R0
   D=M
   @R1
   M=D     // R1 = R0

   @temp
   D=M
   @R0
   M=D     // R0 = temp

(END)
   @END
   0;JMP
```

Hack assembler

Binary code

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
0000000000000000
1111110000010000
0000000000000001
1110001100001000
0000000000010000
1111110000010000
0000000000000000
1110001100001000
0000000000001100
1110101010000111
```

load & execute

We'll develop a Hack assembler later in the course.

# CPU Emulator

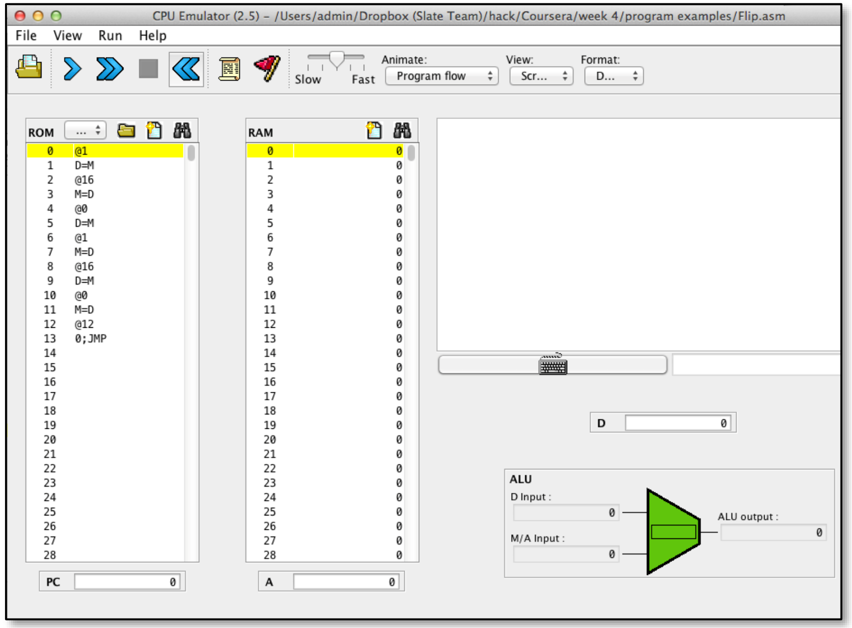
Assembly program

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]
   @R1
   D=M
   @temp
   M=D    // temp = R1

   @R0
   D=M
   @R1
   M=D    // R1 = R0

   @temp
   D=M
   @R0
   M=D    // R0 = temp

(END)
   @END
   0;JMP
```

load

(the simulator software translates from symbolic to binary as it loads)

CPU Emulator



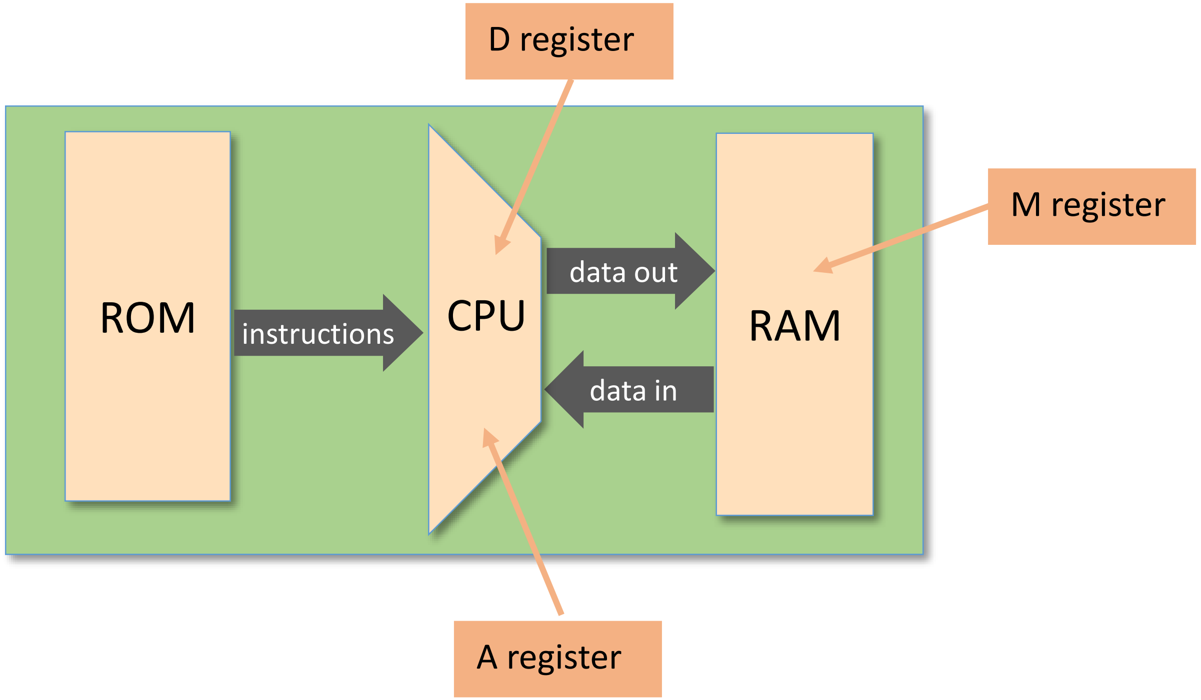

- A software tool
- Convenient for debugging and executing symbolic Hack programs.

# Registers and memory

D: data register

A: address / data register

M: the currently selected memory register: M = RAM[A]



# Registers and memory

`D`: data register

`A`: address / data register

`M`: the currently selected memory register: `M` = `RAM[A]`

Typical operations:

# Registers and memory

D: data register

A: address / data register

M: the currently selected memory register: M = RAM[A]

Typical operations:

```
// D=10
@10
D=A


// D++
D=D+1


// D=RAM[17]
@17
D=M


// RAM[17]=D
@17
M=D
```

# Registers and memory

D: data register

A: address / data register

M: the currently selected memory register: M = RAM[A]

Typical operations:

```
// D=10
@10
D=A


// D++
D=D+1


// D=RAM[17]
@17
D=M


// RAM[17]=D
@17
M=D
```

```
// RAM[17]=10
@10
D=A
@17
M=D


// RAM[5] = RAM[3]
@3
D=M
@5
M=D
```

# Program example: add two numbers

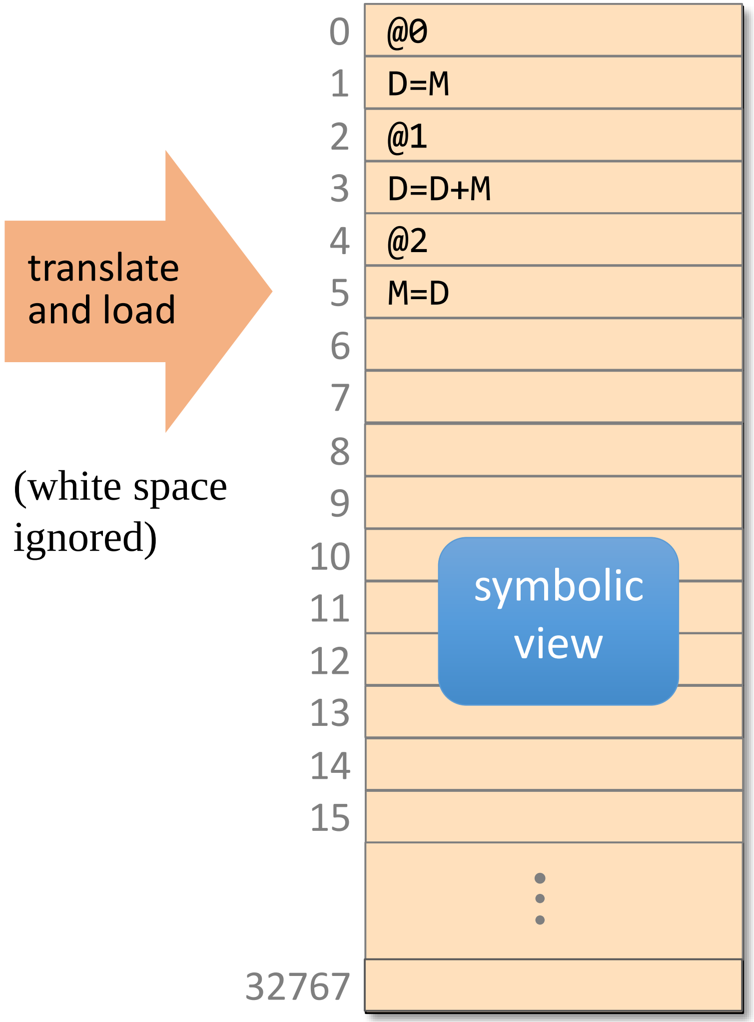
Hack assembly code

# Program example: add two numbers

Hack assembly code

```
    // Program: Add2.asm
    // Computes: RAM[2] = RAM[0] + RAM[1]
    // Usage: put values in RAM[0], RAM[1]

0   @0
1   D=M    // D = RAM[0]

2   @1
3   D=D+M // D = D + RAM[1]

4   @2
5   M=D    // RAM[2] = D
```

Memory (ROM)



# Program example: add two numbers

Hack assembly code

```
   // Program: Add2.asm
   // Computes: RAM[2] = RAM[0] + RAM[1]
   // Usage: put values in RAM[0], RAM[1]

0  @0
1  D=M    // D = RAM[0]

2  @1
3  D=D+M  // D = D + RAM[1]

4  @2
5  M=D    // RAM[2] = D
```

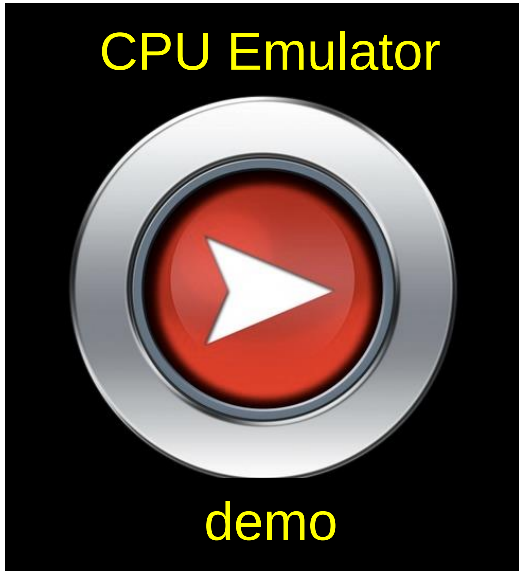
translate and load

Memory (ROM)

| | |
|---|---|
| 0 | 0000000000000000 |
| 1 | 1111110000010000 |
| 2 | 0000000000000001 |
| 3 | 1111000010010000 |
| 4 | 0000000000000010 |
| 5 | 1110001100001000 |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| ⋮ | |
| 32767 | |

binary view

# Program example: add two numbers



# Terminating a program

Hack assembly code

```
  // Program: Add2.asm
  // Computes: RAM[2] = RAM[0] + RAM[1]
  // Usage: put values in RAM[0], RAM[1]

0 @0
1 D=M    // D = RAM[0]

2 @1
3 D=D+M // D = D + RAM[1]

4 @2
5 M=D    // RAM[2] = D
```

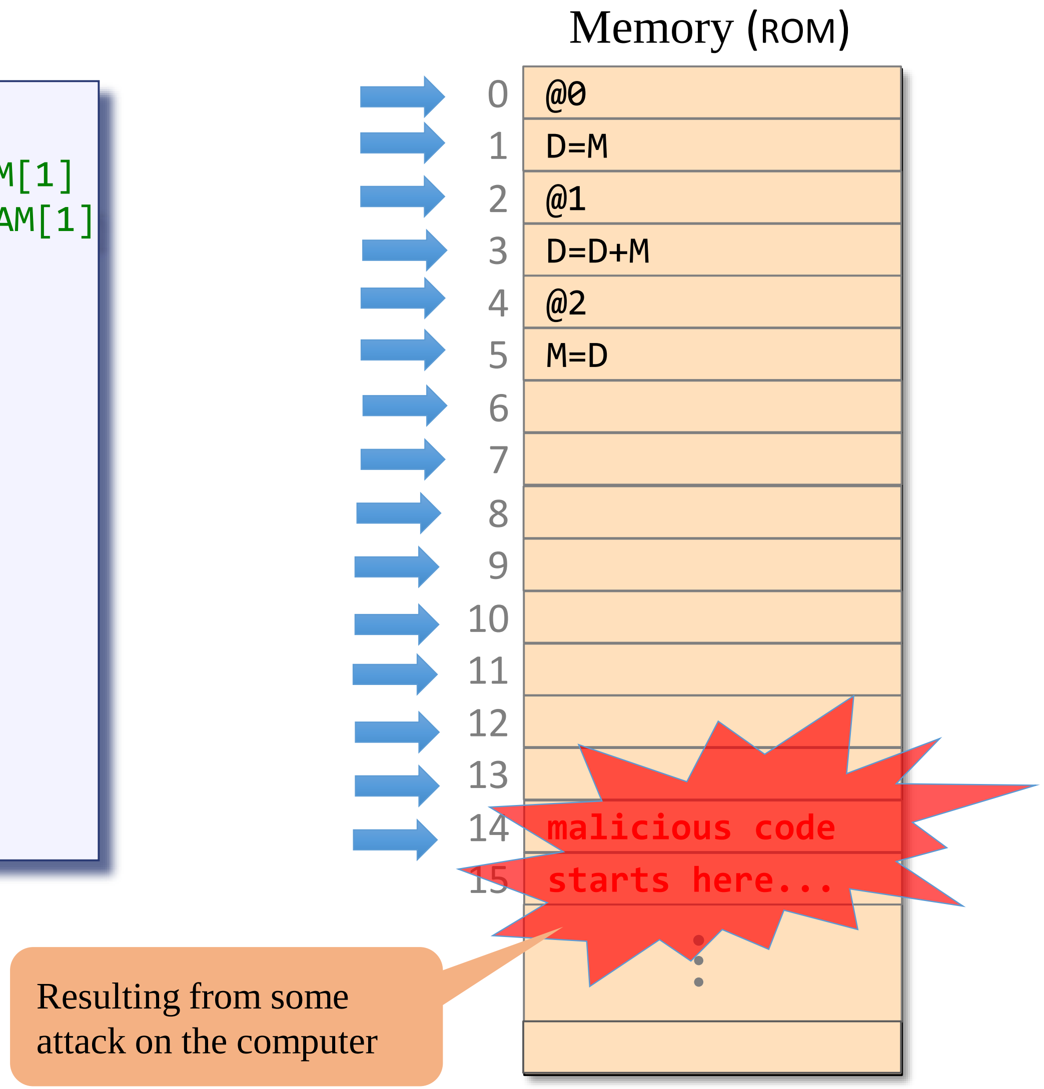
Memory (ROM)

| | |
|---|---|
| 0 | @0 |
| 1 | D=M |
| 2 | @1 |
| 3 | D=D+M |
| 4 | @2 |
| 5 | M=D |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| | ⋮ |
| | |

# Terminating a program

## Hack assembly code

```
    // Program: Add2.asm
    // Computes: RAM[2] = RAM[0] + RAM[1]
    // Usage: put values in RAM[0], RAM[1]

0   @0
1   D=M    // D = RAM[0]

2   @1
3   D=D+M // D = D + RAM[1]

4   @2
5   M=D    // RAM[2] = D
```



# Terminating a program

Hack assembly code

```
   // Program: Add2.asm
   // Computes: RAM[2] = RAM[0] + RAM[1]
   // Usage: put values in RAM[0], RAM[1]

0  @0
1  D=M    // D = RAM[0]

2  @1
3  D=D+M  // D = D + RAM[1]

4  @2
5  M=D    // RAM[2] = D
```

Memory (ROM)

| | |
|---|---|
| 0 | @0 |
| 1 | D=M |
| 2 | @1 |
| 3 | D=D+M |
| 4 | @2 |
| 5 | M=D |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| | ⋮ |
| | |

# Terminating a program

Hack assembly code

```
  // Program: Add2.asm
  // Computes: RAM[2] = RAM[0] + RAM[1]
  // Usage: put values in RAM[0], RAM[1]

0 @0
1 D=M    // D = RAM[0]

2 @1
3 D=D+M // D = D + RAM[1]

4 @2
5 M=D    // RAM[2] = D

6 @6
7 0;JMP
```

- Jump to instruction number A (which happens to be 6)
- `0`: syntax convention for `jmp` instructions

Memory (ROM)

| | |
|---|---|
| 0 | @0 |
| 1 | D=M |
| 2 | @1 |
| 3 | D=D+M |
| 4 | @2 |
| 5 | M=D |
| 6 | @6 |
| 7 | 0;JMP |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| | ⋮ |
| | |

<u>Best practice:</u>

To terminate a program safely, end it with an infinite loop.

# Built-in symbols

The Hack assembly language features *built-in symbols*:

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

Attention: Hack is case-sensitive!
R5 and r5 are different symbols.

These symbols can be used to denote "virtual registers"

Example: suppose we wish to use RAM[5] to represent some variable, say x, and we wish to let x=7

implementation:

```
// let RAM[5] = 7
@7
D=A

@5
M=D
```

better style:

```
// let RAM[5] = 7
@7
D=A

@R5
M=D
```

# Built-in symbols

The Hack assembly language features *built-in symbols*:

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

# Built-in symbols

The Hack assembly language features *built-in symbols*:

| symbol | value |
|---|---|
| R0 | 0 |
| R1 | 1 |
| R2 | 2 |
| ... | ... |
| R15 | 15 |
| SCREEN | 16384 |
| KBD | 24576 |

| symbol | value |
|---|---|
| SP | 0 |
| LCL | 1 |
| ARG | 2 |
| THIS | 3 |
| THAT | 4 |

- `R0`, `R1` ,…, `R15` : "virtual registers", can be used as variables
- `SCREEN` and `KBD` : base addresses of I/O memory maps
- Remaining symbols: used in the implementation of the Hack *virtual machine*, discussed in chapters 7-8.

# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ✓ The Hack computer and machine language
- ✓ The Hack language specification
- ✓ Input / Output
- Hack programming
  - ✓ Part 1: registers and memory
  - ➡ Part 2: branching, variables, iteration
  - ❑ Part 3: pointers, input/output
- Project 4 overview

# Branching

# Branching

example:

```
// Program: Signum.asm
// Computes: if R0>0
//                R1=1
//             else
//                R1=0
```

# Branching

example:

```
// Program: Signum.asm
// Computes: if R0>0
//                R1=1
//           else
//                R1=0
// Usage: put a value in RAM[0],
//        run and inspect RAM[1].
```

# Branching

example:

```
   // Program: Signum.asm
   // Computes: if R0>0
   //              R1=1
   //           else
   //              R1=0
   // Usage: put a value in RAM[0],
   //        run and inspect RAM[1].

0     @R0
1     D=M    // D = RAM[0]

2     @8
3     D;JGT  // If R0>0 goto 8

4     @R1
5     M=0    // RAM[1]=0
6     @10
7     0;JMP  // goto end

8     @R1
9     M=1    // R1=1

10    @10
11    0;JMP
```

# Branching

example:

```
   // Program: Signum.asm
   // Computes: if R0>0
   //                R1=1
   //             else
   //                R1=0
   // Usage: put a value in RAM[0],
   //        run and inspect RAM[1].

0     @R0
1     D=M     // D = RAM[0]

2     @8
3     D;JGT   // If R0>0 goto 8

4     @R1
5     M=0     // RAM[1]=0
6     @10
7     0;JMP   // goto end

8     @R1
9     M=1     // R1=1

10    @10
11    0;JMP
```

# Branching

example:

```
// Program: Signum.asm
// Computes: if R0>0
//                 R1=1
//              else
//                 R1=0
// Usage: put a value in RAM[0],
//        run and inspect RAM[1].

0      @R0
1      D=M     // D = RAM[0]

2      @8
3      D;JGT   // If R0>0 goto 8

4      @R1
5      M=0     // RAM[1]=0
6      @10
7      0;JMP   // goto end

8      @R1
9      M=1     // R1=1

10     @10
11     0;JMP
```

cryptic code

*"Instead of imagining that our main task as programmers is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."*

– Donald Knuth

# Branching

example:

```
   // Program: Signum.asm
   // Computes: if R0>0
   //                R1=1
   //             else
   //                R1=0
   // Usage: put a value in RAM[0],
   //        run and inspect RAM[1].

0     @R0
1     D=M     // D = RAM[0]

2     @POSITIVE
3     D;JGT   // If R0>0 goto 8

4     @R1
5     M=0     // RAM[1]=0
6     @10
7     0;JMP   // goto end

   (POSITIVE)
8     @R1
9     M=1     // R1=1

   (END)
10    @END
11    0;JMP
```

referring to a label

declaring a label

# Labels

example:

```
    // Program: Signum.asm
    // Computes: if R0>0
    //              R1=1
    //           else
    //              R1=0
    // Usage: put a value in RAM[0],
    //        run and inspect RAM[1].

0      @R0
1      D=M    // D = RAM[0]

2      @POSITIVE
3      D;JGT  // If R0>0 goto 8

4      @R1
5      M=0    // RAM[1]=0
6      @10
7      0;JMP  // goto end

    (POSITIVE)
8      @R1
9      M=1    // R1=1

    (END)
10     @END
11     0;JMP
```

referring to a label

declaring a label

resolving labels

## Label resolution rules:

- Label declarations generate no code
- Each reference to a label is replaced with a reference to the instruction number following that label's declaration.

Memory

| | |
|---|---|
| 0 | @0 |
| 1 | D=M |
| 2 | @8 // @POSITIVE |
| 3 | D;JGT |
| 4 | @1 |
| 5 | M=0 |
| 6 | @10 // @END |
| 7 | 0;JMP |
| 8 | @1 |
| 9 | M=1 |
| 10 | @10 // @END |
| 11 | 0;JMP |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| | ⋮ |
| 32767 | |

# Labels

example:

```
     // Program: Signum.asm
     // Computes: if R0>0
     //              R1=1
     //           else
     //              R1=0
     // Usage: put a value in RAM[0],
     //        run and inspect RAM[1].

0       @R0
1       D=M    // D = RAM[0]

2       @POSITIVE
3       D;JGT  // If R0>0 goto 8

4       @R1
5       M=0    // RAM[1]=0
6       @10
7       0;JMP  // goto end

     (POSITIVE)
8       @R1
9       M=1    // R1=1

     (END)
10      @END
11      0;JMP
```

referring to a label

declaring a label

resolving labels

Implications:

Memory

| | |
|---|---|
| 0 | @0 |
| 1 | D=M |
| 2 | @8 // @POSITIVE |
| 3 | D;JGT |
| 4 | @1 |
| 5 | M=0 |
| 6 | @10 // @END |
| 7 | 0;JMP |
| 8 | @1 |
| 9 | M=1 |
| 10 | @10 // @END |
| 11 | 0;JMP |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| | ⋮ |
| 32767 | |

# Labels

example:

```
     // Program: Signum.asm
     // Computes: if R0>0
     //                R1=1
     //              else
     //                R1=0
     // Usage: put a value in RAM[0],
     //        run and inspect RAM[1].

 0      @R0
 1      D=M     // D = RAM[0]

 2      @POSITIVE
 3      D;JGT   // If R0>0 goto 8

 4      @R1
 5      M=0     // RAM[1]=0
 6      @10
 7      0;JMP   // goto end

     (POSITIVE)
 8      @R1
 9      M=1     // R1=1

     (END)
10      @END
11      0;JMP
```

referring to a label

declaring a label

resolving labels

Memory

| | |
|---|---|
| 0 | @0 |
| 1 | D=M |
| 2 | @8 // @POSITIVE |
| 3 | D;JGT |
| 4 | @1 |
| 5 | M=0 |
| 6 | @10 // @END |
| 7 | 0;JMP |
| 8 | @1 |
| 9 | M=1 |
| 10 | @10 // @END |
| 11 | 0;JMP |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| ⋮ | |
| 32767 | |

## Implications:

- Instruction numbers no longer needed in symbolic programming
- The symbolic code becomes *relocatable*.

# Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]
```

# Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp

   @R1
   D=M
   @temp
   M=D    // temp = R1

   @R0
   D=M
   @R1
   M=D    // R1 = R0

   @temp
   D=M
   @R0
   M=D    // R0 = temp

(END)
   @END
   0;JMP
```

# Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp

   @R1
   D=M
   @temp
   M=D     // temp = R1

   @R0
   D=M
   @R1
   M=D     // R1 = R0

   @temp
   D=M
   @R0
   M=D     // R0 = temp

(END)
   @END
   0;JMP
```

symbol used for the first time

symbol used again

# Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp

   @R1
   D=M
   @temp
   M=D      // temp = R1

   @R0
   D=M
   @R1
   M=D      // R1 = R0

   @temp
   D=M
   @R0
   M=D      // R0 = temp

(END)
   @END
   0;JMP
```

symbol used for the first time

symbol used again

resolving symbols

Symbol resolution rules:

- A reference to a symbol that has no corresponding label declaration is treated as a reference to a variable
- If the reference @*symbol* occurs in the program for first time, *symbol* is allocated to address 16 onward (say *n*), and the generated code is @*n*
- All subsequencet @*symbol* commands are translated into @*n*

In other words: variables are allocated to `RAM[16]` onward.

Memory

| | |
|---|---|
| 0 | @1 |
| 1 | D=M |
| 2 | @16 // @temp |
| 3 | M=D |
| 4 | @0 |
| 5 | D=M |
| 6 | @1 |
| 7 | M=D |
| 8 | @16 // @temp |
| 9 | D=M |
| 10 | @0 |
| 11 | M=D |
| 12 | @12 |
| 13 | 0;JMP |
| 14 | |
| 15 | |
| | ⋮ |
| 32767 | |

# Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of
// RAM[0] and RAM[1]

// temp = R1
// R1 = R0
// R0 = temp

   @R1
   D=M
   @temp
   M=D    // temp = R1

   @R0
   D=M
   @R1
   M=D    // R1 = R0

   @temp
   D=M
   @R0
   M=D    // R0 = temp

(END)
   @END
   0;JMP
```

resolving symbols

Memory

| | |
|---|---|
| 0 | @1 |
| 1 | D=M |
| 2 | @16 // @temp |
| 3 | M=D |
| 4 | @0 |
| 5 | D=M |
| 6 | @1 |
| 7 | M=D |
| 8 | @16 // @temp |
| 9 | D=M |
| 10 | @0 |
| 11 | M=D |
| 12 | @12 |
| 13 | 0;JMP |
| 14 | |
| 15 | |
| | ⋮ |
| 32767 | |

Implications:

symbolic code is easy to read and debug

# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]
```

# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

LOOP:
   if i > n goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP

STOP:
   R1 = sum
```

# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

   ...
```

assembly code

```
// Program: Sum1toN.asm
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
```

# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

   ...
```

assembly code

```
// Program: Sum1toN.asm
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]

   @R0
   D=M
   @n
   M=D   // n = R0

   @i
   M=1   // i = 1

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

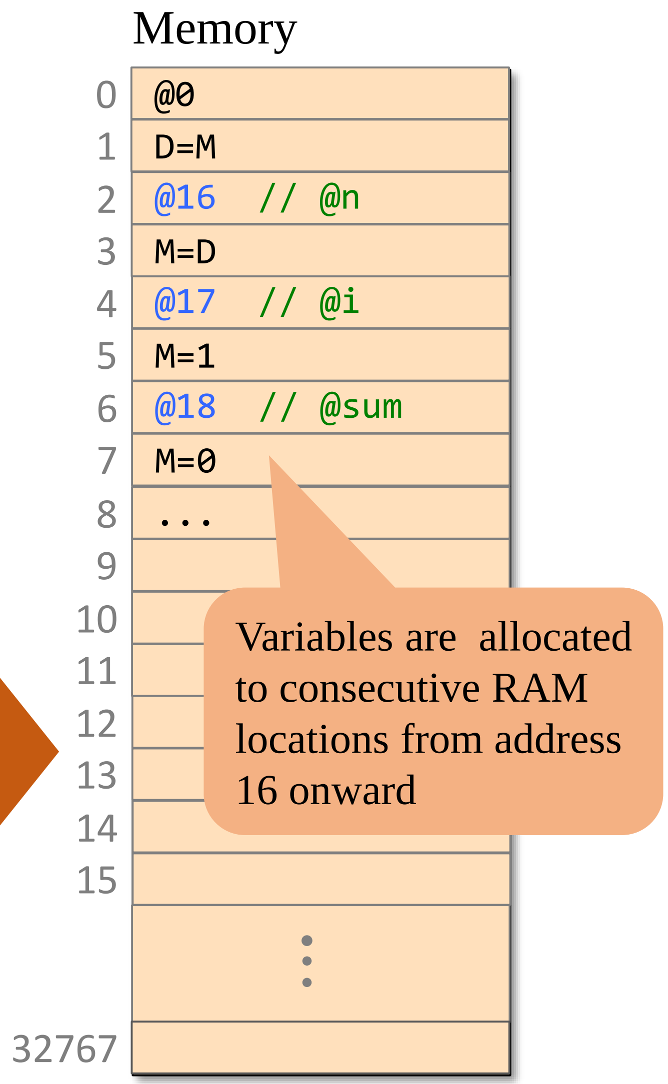
# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

   ...
```

assembly code

```
// Program: Sum1toN.asm
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]

   @R0
   D=M
   @n
   M=D    // n = R0

   @i
   M=1    // i = 1

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



# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

   ...
```

# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

LOOP:
   if i > n goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP

STOP:
   R1 = sum
```

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
```

# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

LOOP:
   if i > n goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP

STOP:
   R1 = sum
```

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
```

# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

LOOP:
   if i > n goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP

STOP:
   R1 = sum
```

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
```

# Iterative processing

pseudo code

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0

LOOP:
   if i > n goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP

STOP:
   R1 = sum
```

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP
```

# Program execution

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP
```

# Program execution

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP
```

iterations

| | 0 |
|---|---|
| RAM[0]: | 3 |
| n: | 3 |
| i: | 1 |
| sum: | 0 |

# Program execution

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP
```

iterations

| | 0 | 1 |
|---|---|---|
| RAM[0]: | 3 | |
| n: | 3 | |
| i: | 1 | 2 |
| sum: | 0 | 1 |

# Program execution

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP
```

iterations

| | 0 | 1 | 2 |
|---|---|---|---|
| RAM[0]: | 3 | | |
| n: | 3 | | |
| i: | 1 | 2 | 3 |
| sum: | 0 | 1 | 3 |

# Program execution

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP
```

iterations

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| RAM[0]: | 3 | | | |
| n: | 3 | | | |
| i: | 1 | 2 | 3 | 4 |
| sum: | 0 | 1 | 3 | 6 |

# Program execution

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP
```

iterations

| | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| RAM[0]: | 3 | | | | |
| n: | 3 | | | | |
| i: | 1 | 2 | 3 | 4 | ... |
| sum: | 0 | 1 | 3 | 6 | ... |

# Writing assembly programs

assembly program

```
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]
   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP
(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum
(END)
   @END
   0;JMP
```

<u>Best practice:</u>

- **Design** the program using pseudo code
- **Write** the program in assembly language
- **Test** the program (on paper) using a variable-value trace table

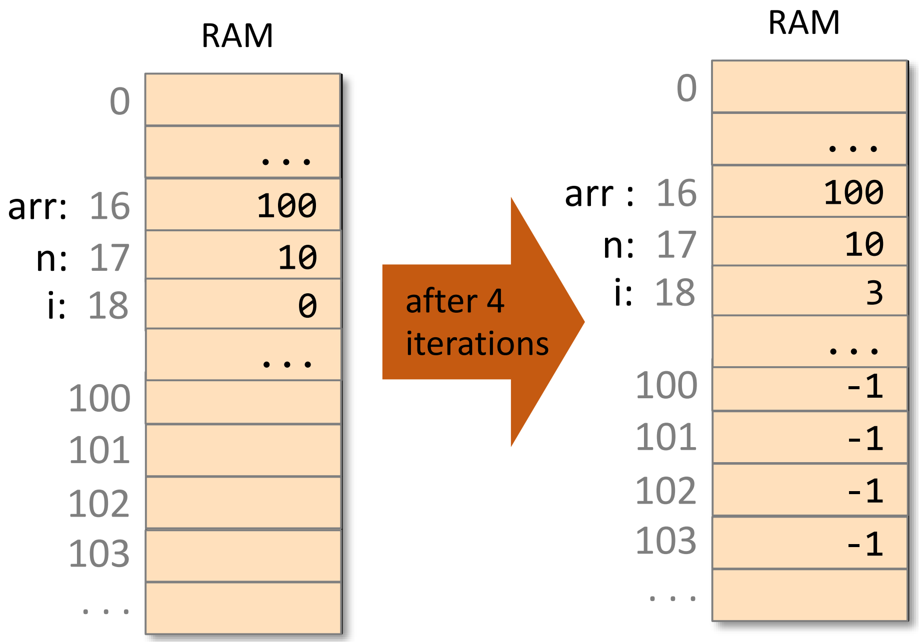
# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ✓ The Hack computer and machine language
- ✓ The Hack language specification
- ✓ Input / Output
- Hack programming
  - ✓ Part 1: registers and memory
  - ✓ Part 2: branching, variables, iteration
  - ➡ Part 3: pointers, input/output
- Project 4 overview

# Pointers

Example:

```
// for (i=0; i<n; i++) {
//       arr[i] = -1
// }
```



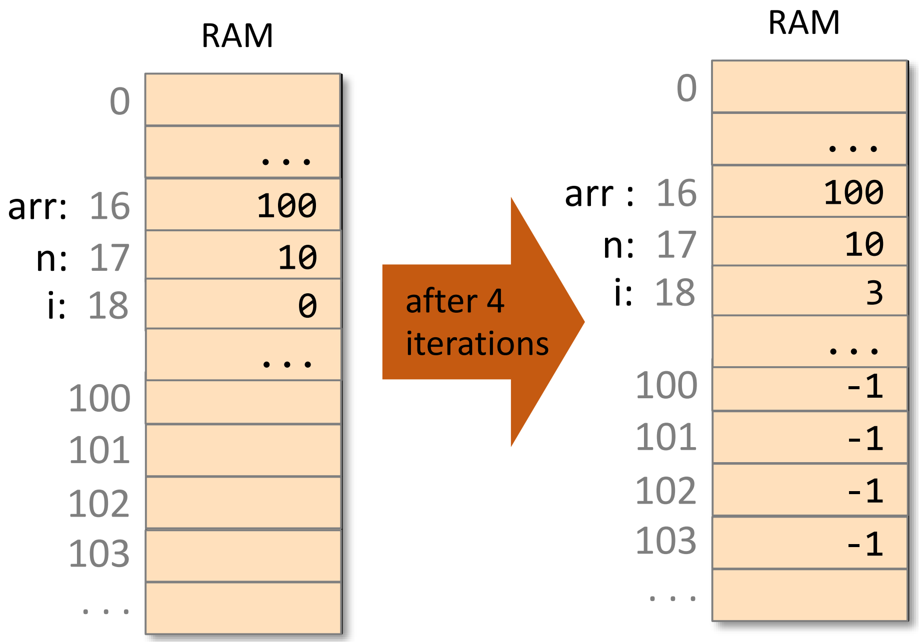

<u>Observations:</u>

- The array is implemented as a block of memory registers
- In order to access these memory registers one after the other, we need a variable that holds the current address
- Variables that represent addresses are called <u>pointers</u>
- There is nothing special about pointer variables, except that their values are interpreted as addresses.

# Pointers

Example:

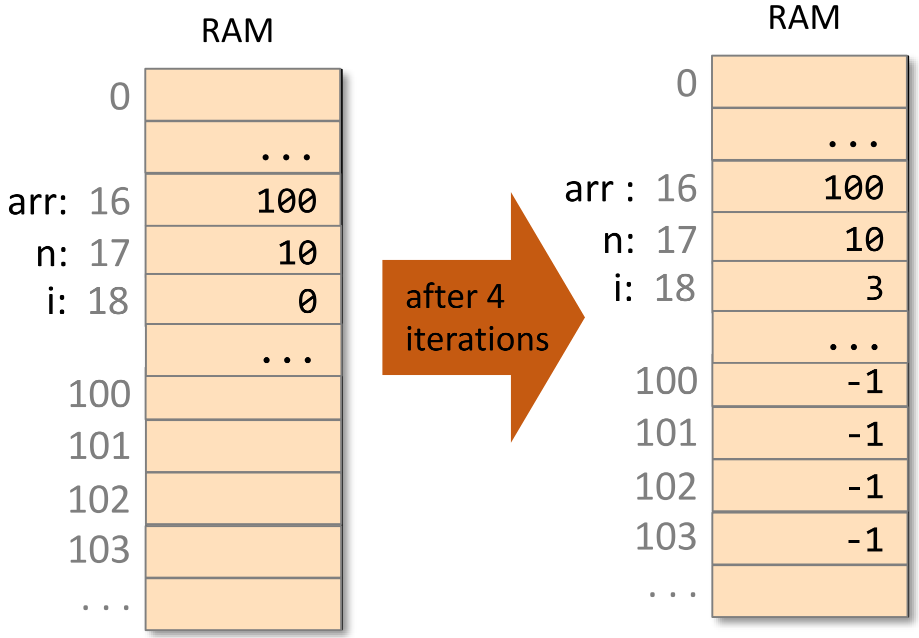
```
// for (i=0; i<n; i++) {
//       arr[i] = -1
// }

   // Suppose that arr=100 and n=10
```



# Pointers

Example:

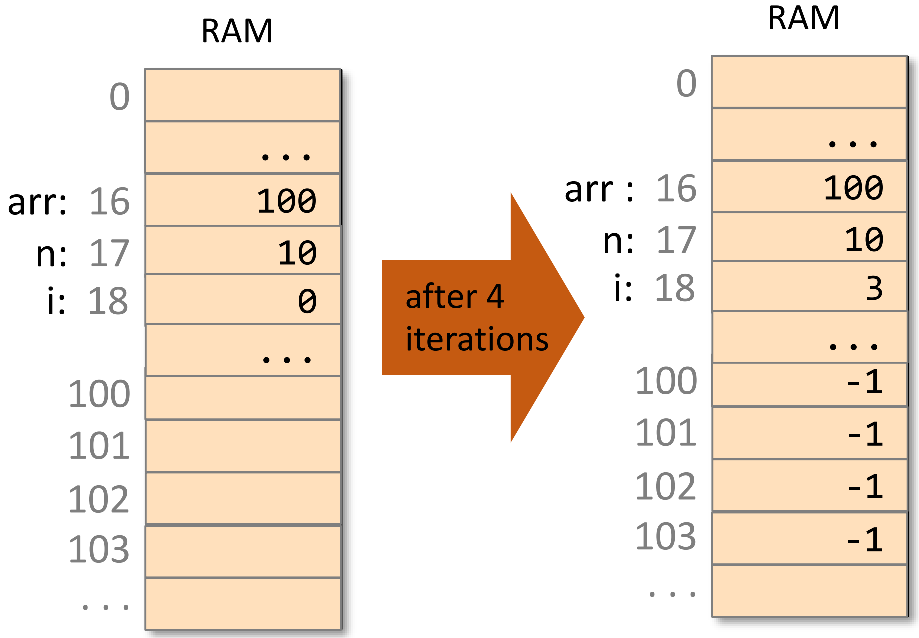
```
// for (i=0; i<n; i++) {
//       arr[i] = -1
// }

   // Suppose that arr=100 and n=10

   // Let arr = 100
   @100
   D=A
   @arr
   M=D
```



# Pointers

Example:

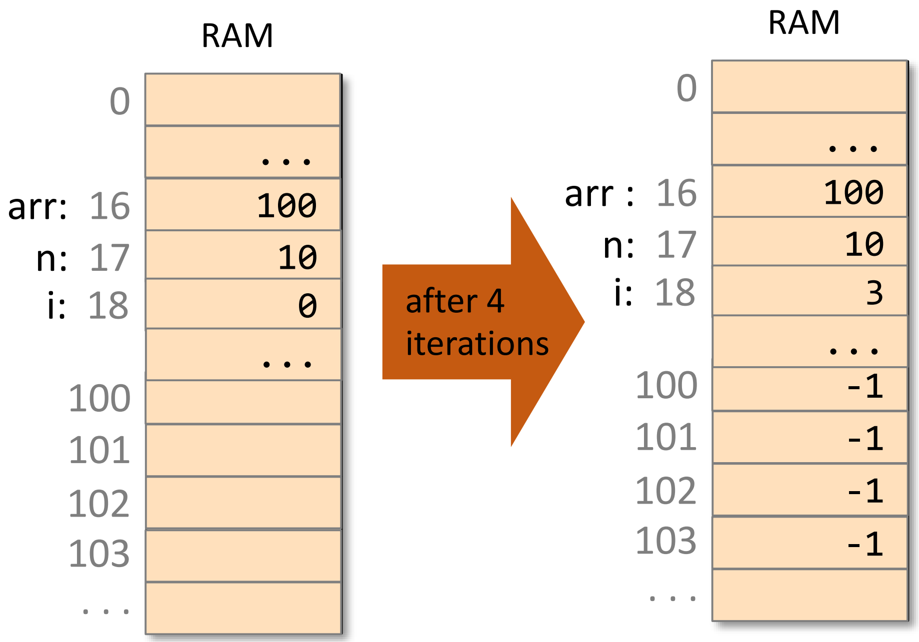
```
// for (i=0; i<n; i++) {
//      arr[i] = -1
// }

   // Suppose that arr=100 and n=10

   // Let arr = 100
   @100
   D=A
   @arr
   M=D

   // Let n = 10
   @10
   D=A
   @n
   M=D

   // Let i = 0
   @i
   M=0

   // Loop code continues
   // in next slide...
```



# Pointers

Example:

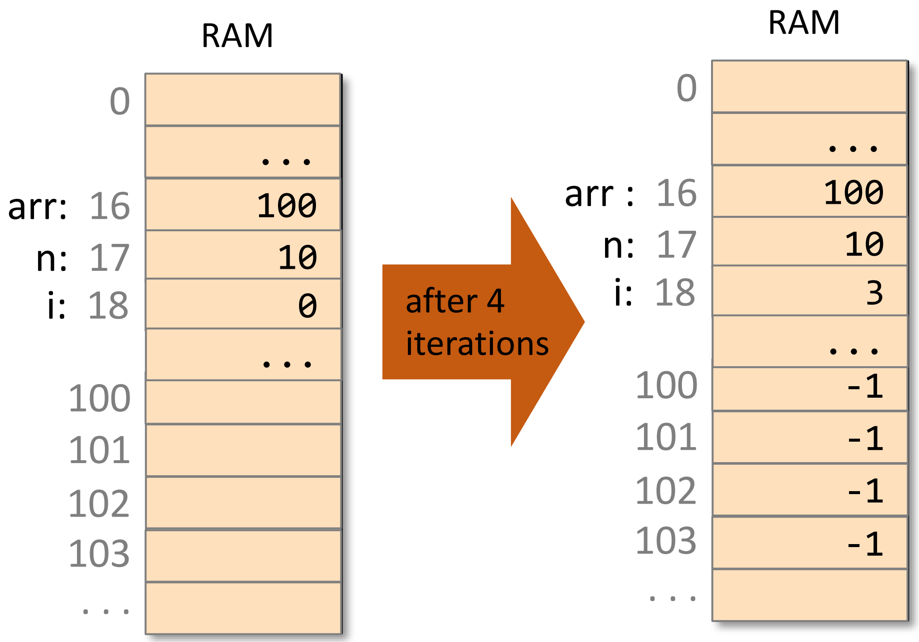
```
(LOOP)
   // if (i==n) goto END
   @i
   D=M
   @n
   D=D-M
   @END
   D;JEQ
```



# Pointers

Example:

```
(LOOP)
   // if (i==n) goto END
   @i
   D=M
   @n
   D=D-M
   @END
   D;JEQ

   // RAM[arr+i] = -1
   @arr
   D=M
   @i
   A=D+M
   M=-1
```

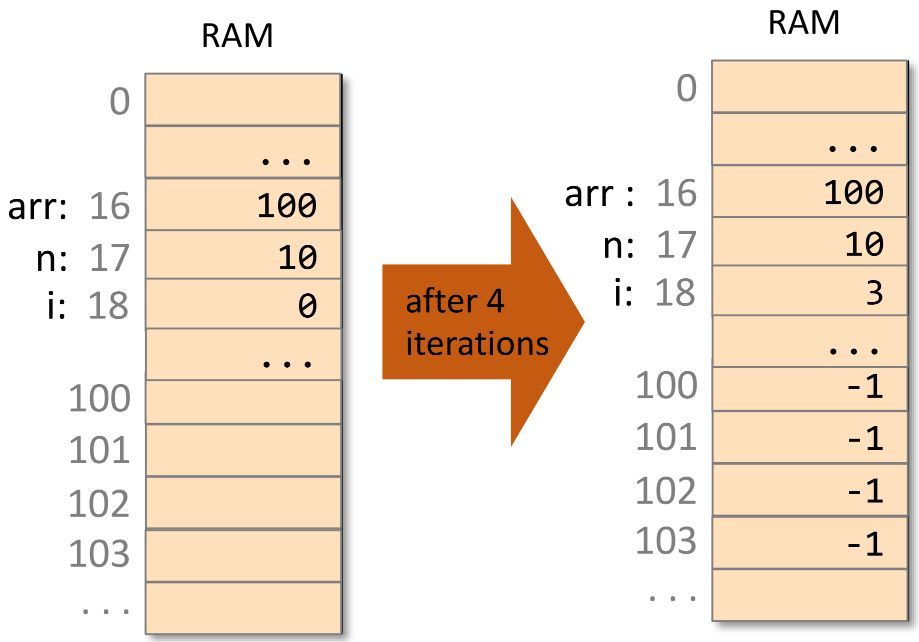


# Pointers

Example:

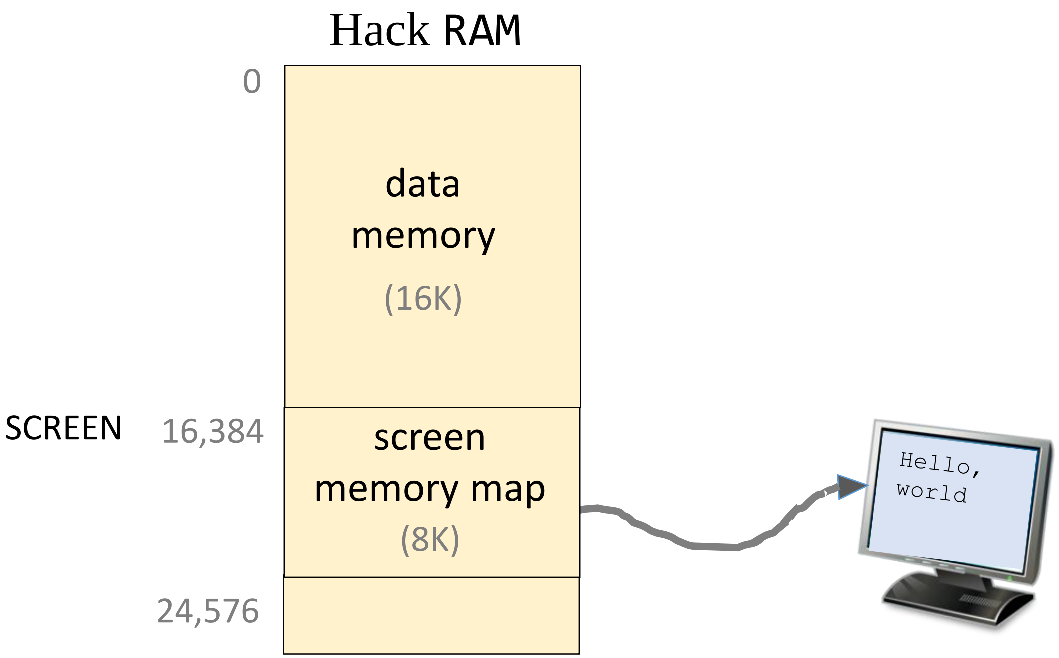
```
(LOOP)
   // if (i==n) goto END
   @i
   D=M
   @n
   D=D-M
   @END
   D;JEQ

   // RAM[arr+i] = -1
   @arr
   D=M
   @i
   A=D+M
   M=-1

   // i++
   @i
   M=M+1

   @LOOP
   0;JMP

(END)
   @END
   0;JMP
```

typical pointer manipulation




- Pointers: Variables that store memory addresses (like `arr`)
- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like `A`=*expression*
- Semantics:
  "set the address register to some value".

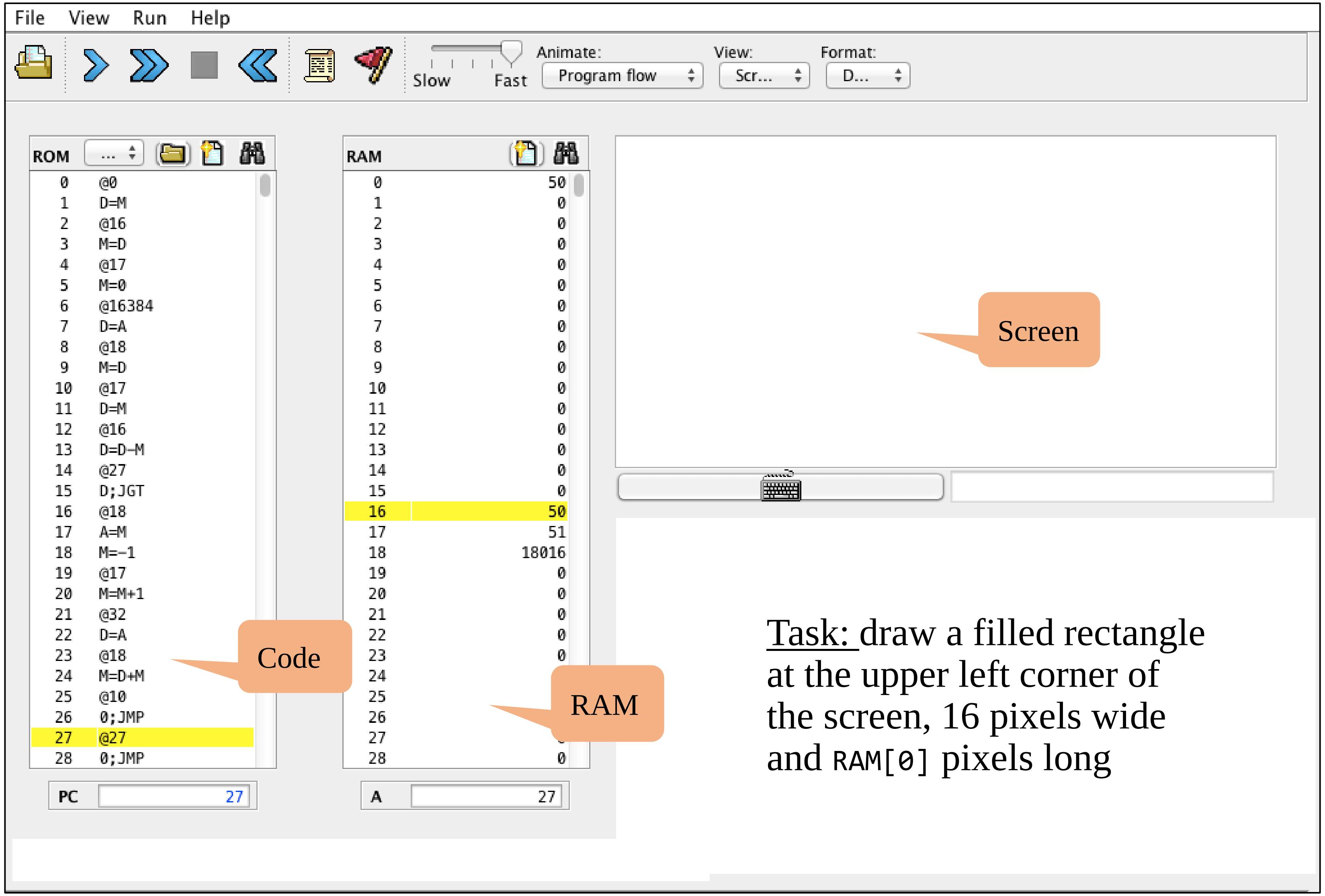
# Output




## Hack language convention:

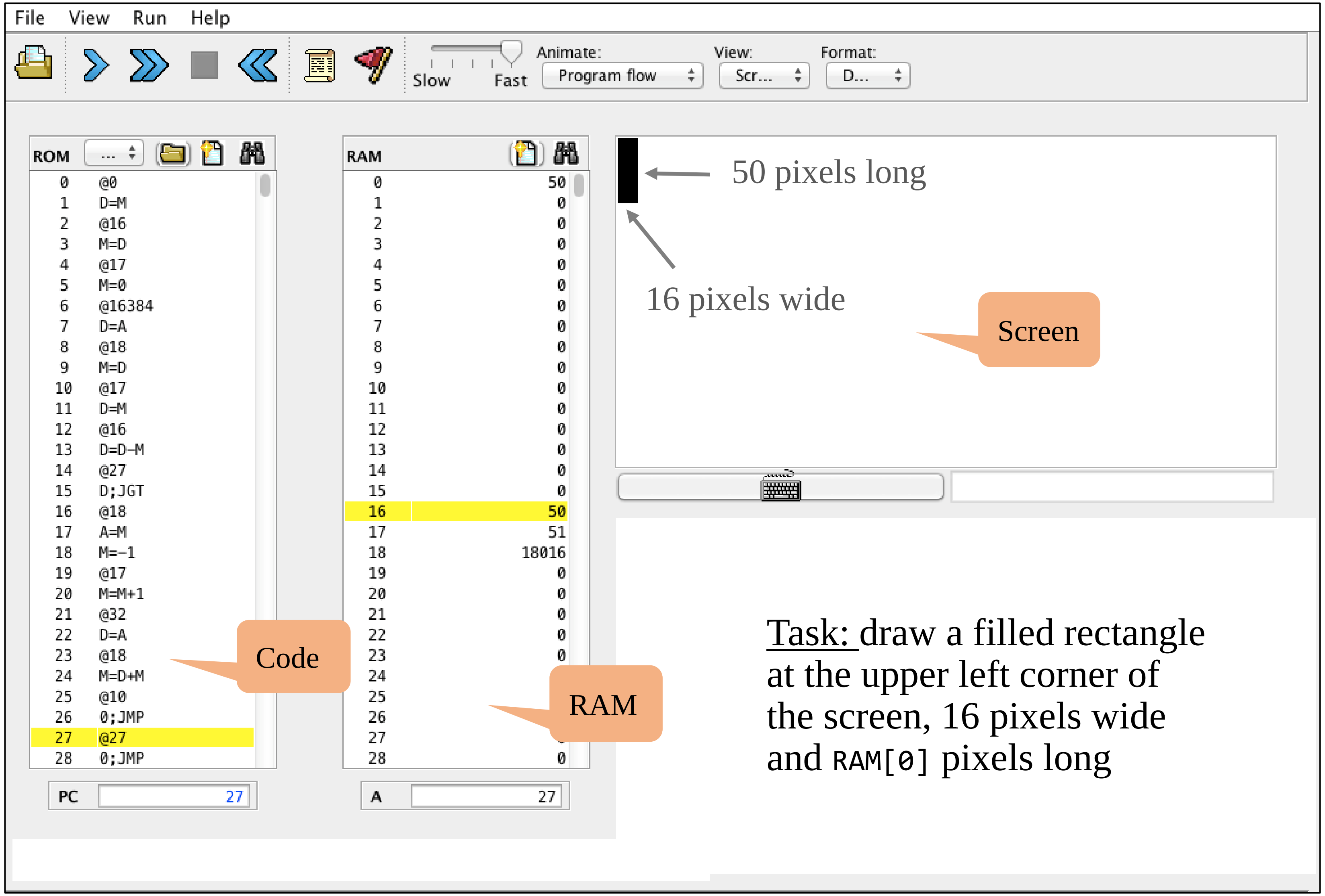
- SCREEN: base address of the screen memory map

# Handling the screen (example)



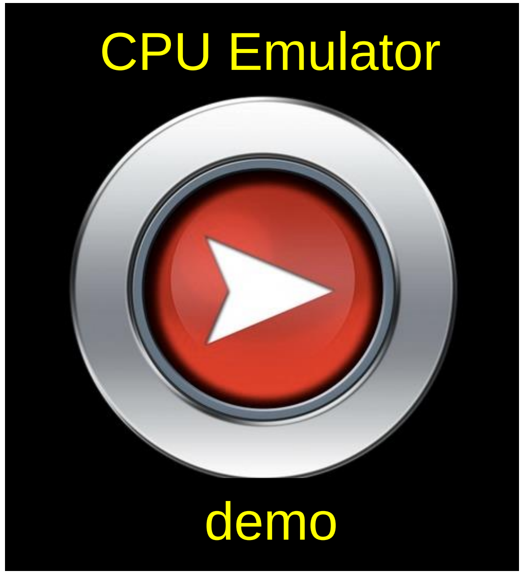

<u>Task:</u> draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and `RAM[0]` pixels long

# Handling the screen (example)

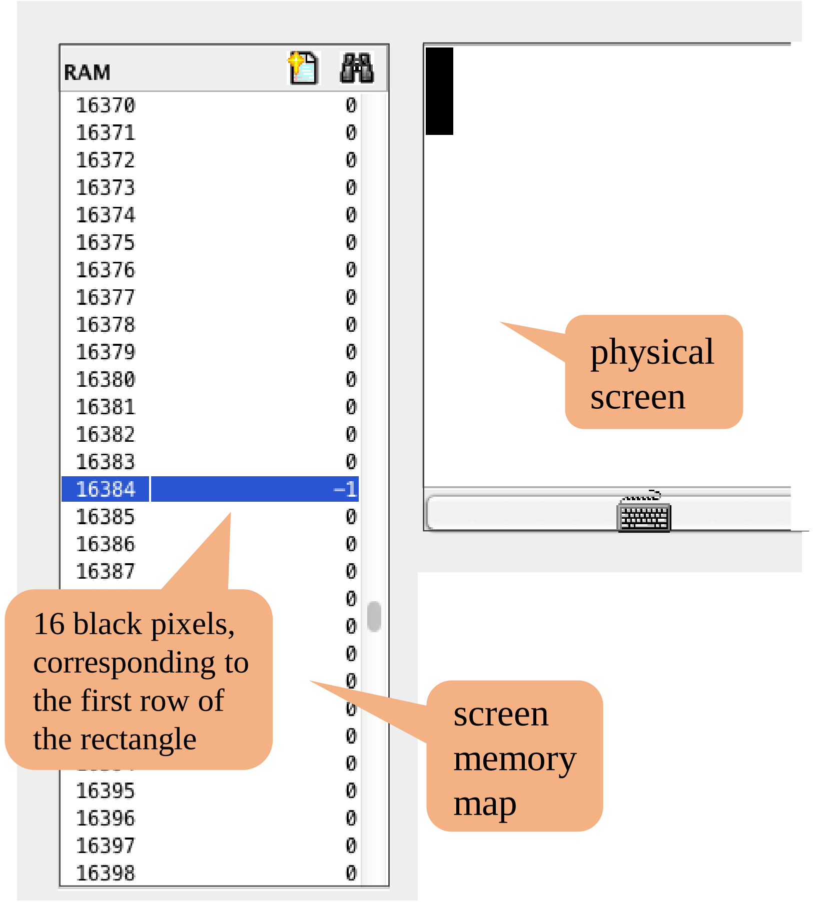



<u>Task:</u> draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and `RAM[0]` pixels long

# Handling the screen (example)



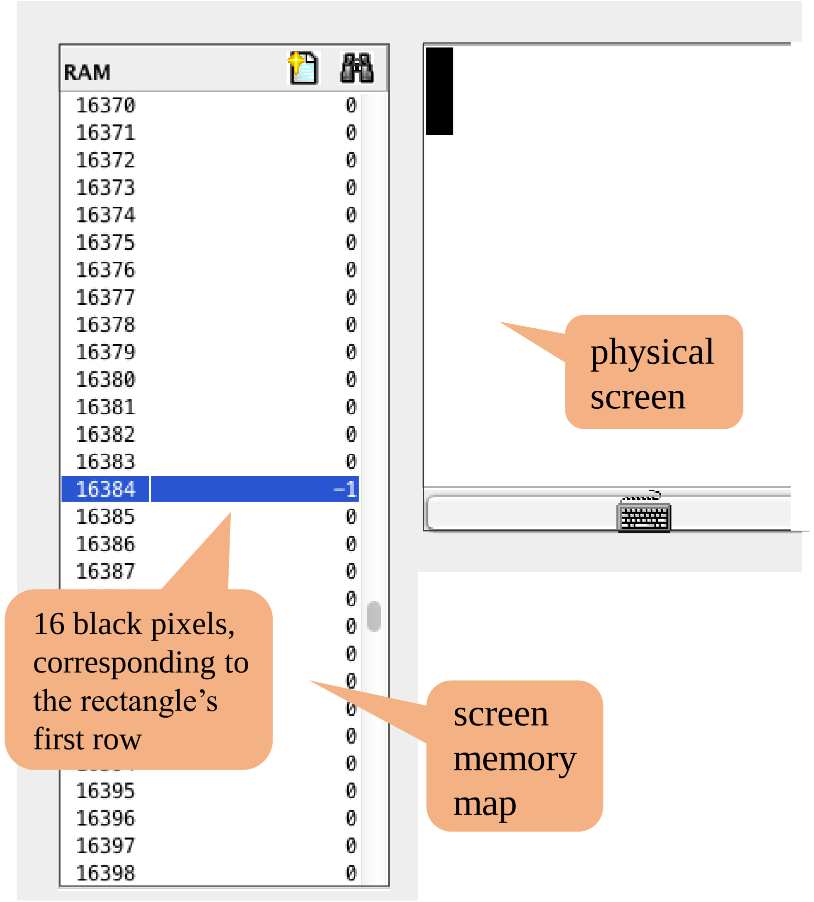
# Handling the screen (example)



# Handling the screen (example)

Pseudo code

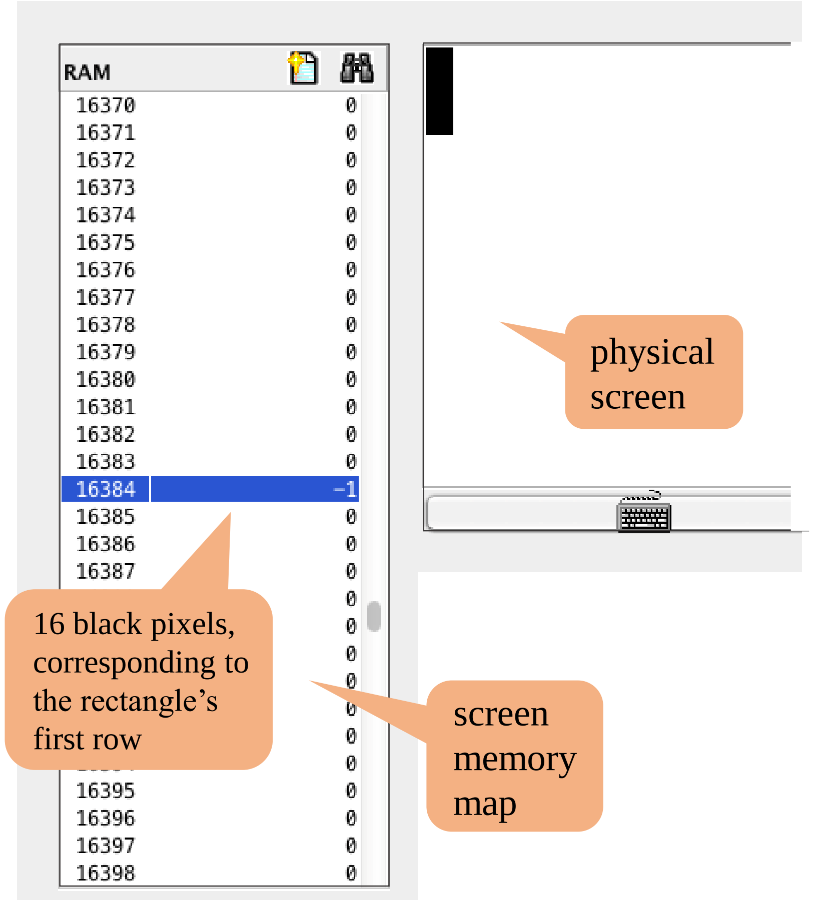
```
// for (i=0; i<n; i++) {
//     draw 16 black pixels at the
//     beginning of row i
// }
```



# Handling the screen (example)

Pseudo code

```
// for (i=0; i<n; i++) {
//     draw 16 black pixels at the
//     beginning of row i
// }

addr = SCREEN
n = RAM[0]
i = 0

LOOP:
   if i > n goto END
   RAM[addr] = -1  // 1111111111111111
   // advances to the next row
   addr = addr + 32
   i = i + 1
   goto LOOP

END:
   goto END
```



# Handling the screen (example)

Assembly code

```
// Program: Rectangle.asm
// Draws a filled rectangle at the
// screen's top left corner, with
// width of 16 pixels and height of
// RAM[0] pixels.
// Usage: put a non-negative number
// (rectangle's height) in RAM[0].

   @SCREEN
   D=A
   @addr
   M=D  // addr = 16384
        // (screen's base address)
   @0
   D=M
   @n
   M=D  // n = RAM[0]

   @i
   M=0  // i = 0
```

(continued)

```
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @END
   D;JGT  // if i>n goto END

   @addr
   A=M
   M=-1   // RAM[addr]=1111111111111111

   @i
   M=M+1  // i = i + 1
   @32
   D=A
   @addr
   M=D+M  // addr = addr + 32
   @LOOP
   0;JMP  // goto LOOP

(END)
   @END   // program's end
   0;JMP  // infinite loop
```

# Handling the screen (example)

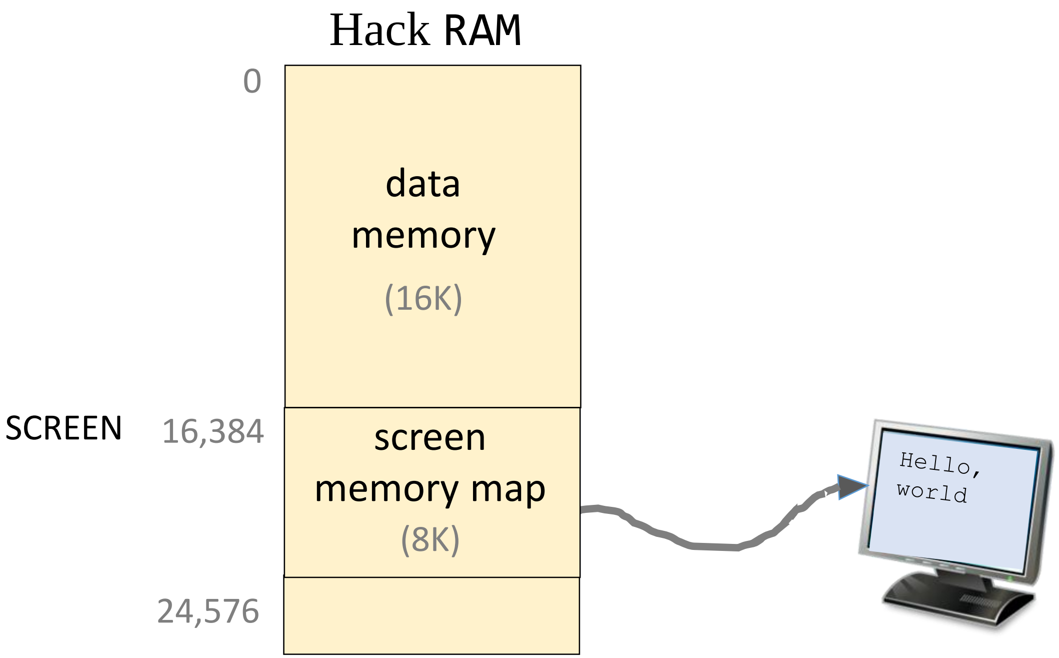
Assembly code

```
// Program: Rectangle.asm
// Draws a filled rectangle at the
// screen's top left corner, with
// width of 16 pixels and height of
// RAM[0] pixels.
// Usage: put a non-negative number
// (rectangle's height) in RAM[0].

   @SCREEN
   D=A
   @addr
   M=D  // addr = 16384
        // (screen's base address)
   @0
   D=M
   @n
   M=D  // n = RAM[0]

   @i
   M=0  // i = 0
```

(continued)

```
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @END
   D;JGT  // if i>n goto END

   @addr
   A=M
   M=-1   // RAM[addr]=1111111111111111

   @i
   M=M+1  // i = i + 1
   @32
   D=A
   @addr
   M=D+M  // addr = addr + 32
   @LOOP
   0;JMP  // goto LOOP

(END)
   @END   // program's end
   0;JMP  // infinite loop
```

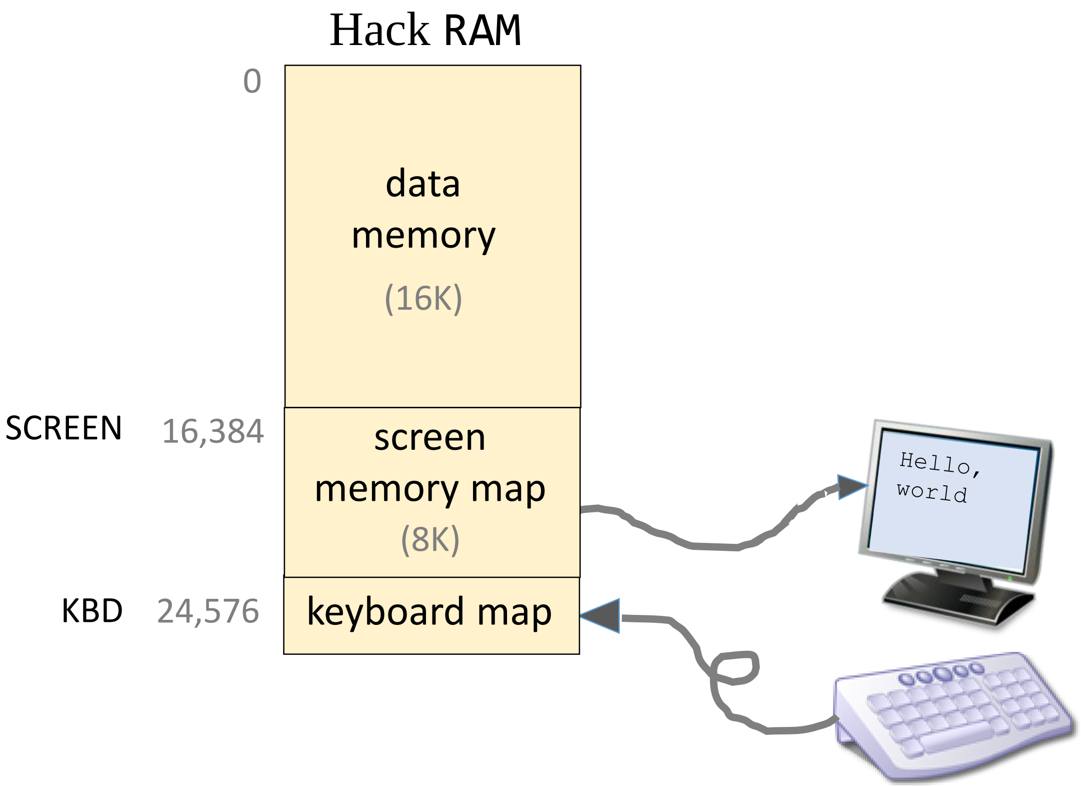
# Input




<u>Hack language convention:</u>

- SCREEN: base address of the screen memory map

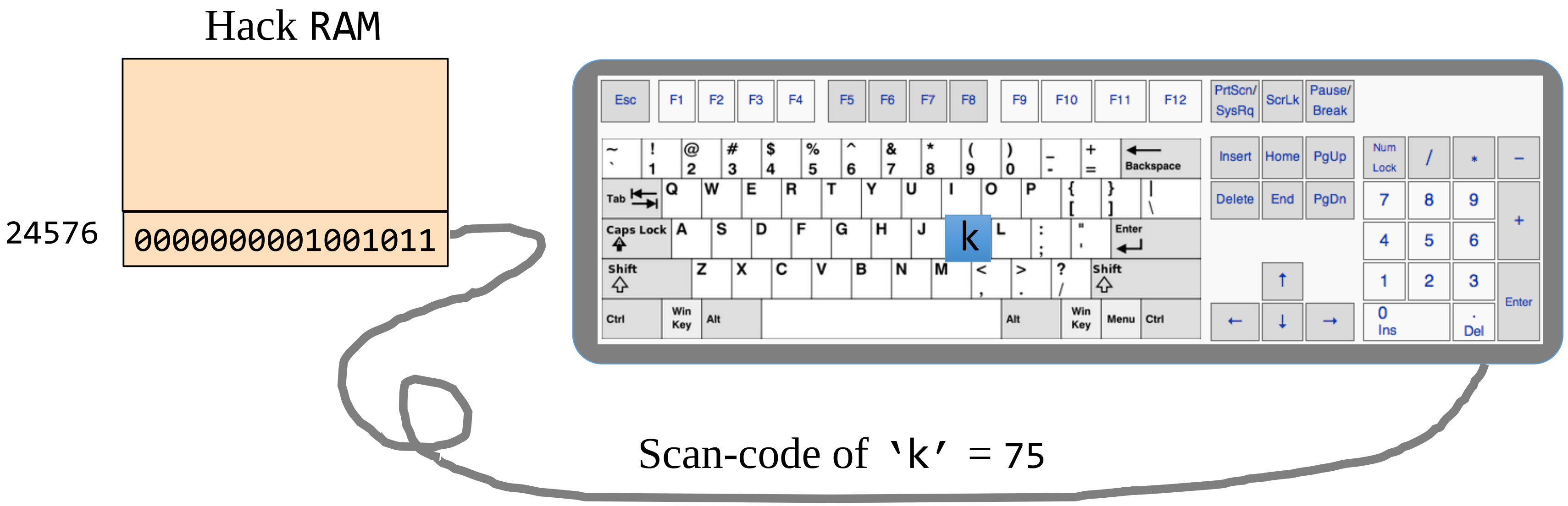
# Input




Hack language convention:

- SCREEN: base address of the screen memory map
- KBD: address of the keyboard memory map

# Handling the keyboard




<u>To check which key is currently pressed:</u>

- Read the contents of `RAM[24576]` (address `KBD`)
- If the register contains `0`, no key is pressed
- Otherwise, the register contains the scan code of the currently pressed key.

# End notes

High level code

```
for (i=0; i<n; i++) {
      arr[i] = -1
}
```

Compiler

Machine language

```
   ...
   @i
    M=0
(LOOP)
    @i
    D=M
    @n
    D=D-M
    @END
    D;JEQ
    @arr
    D=M
    @i
    A=D+M
    M=-1
    @i
    M=M+1
    @LOOP
    0;JMP
(END)
    @END
    0;JMP
```

Low-level programming is:

- Low level
- Profound
- Subtle
- Efficient (or not)
- Intellectually challenging.

# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ✓ The Hack computer and machine language
- ✓ The Hack language specification
- ✓ Input / Output
- ✓ Hack programming
  - Part 1: registers and memory
  - Part 2: branching, variable, iteration
  - Part 3: pointers, input/output
- ➡ Project 4 overview

# In a Nutshell

Project objectives

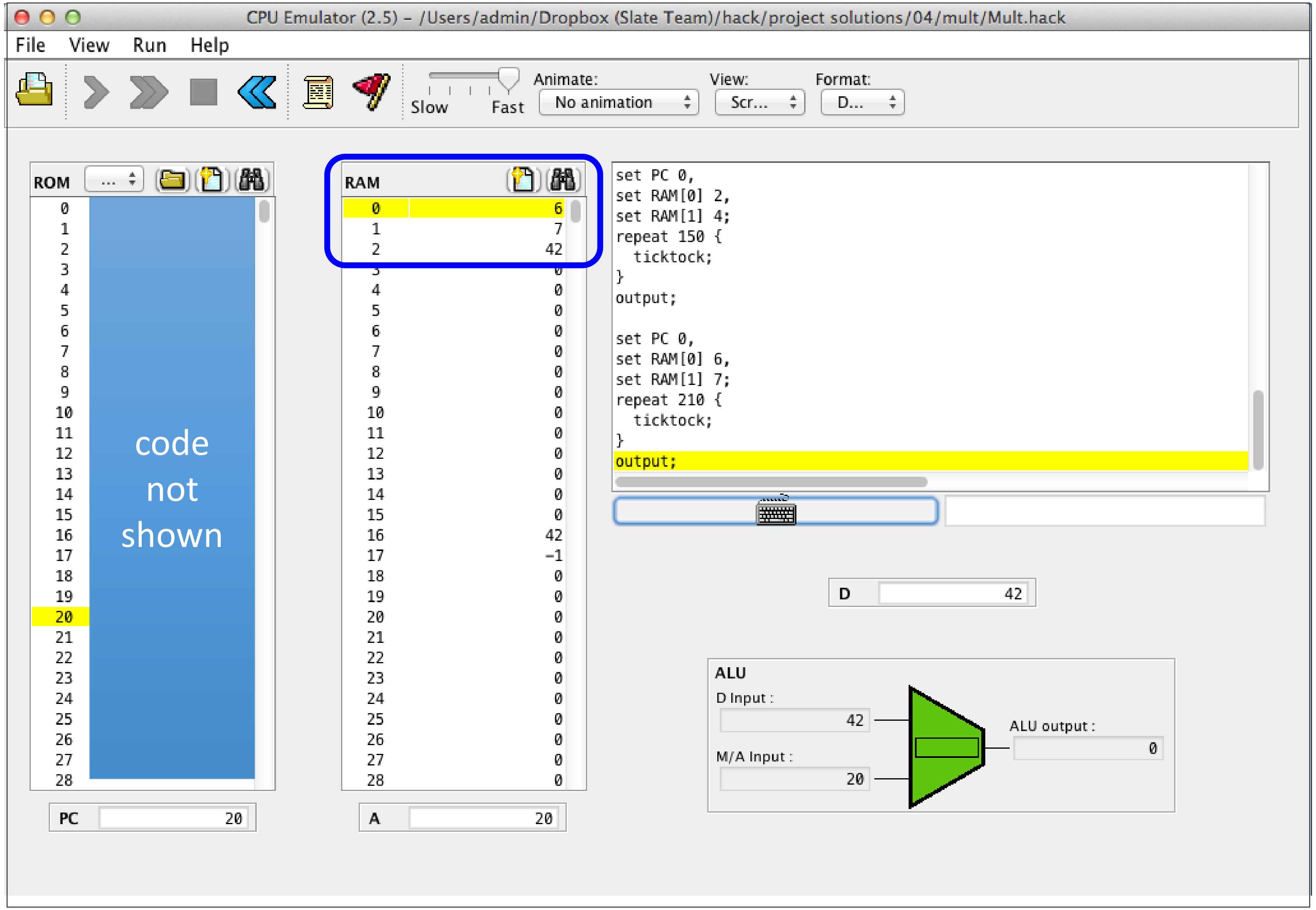
Have a taste of:

- low-level programming
- Hack assembly language
- Hack hardware

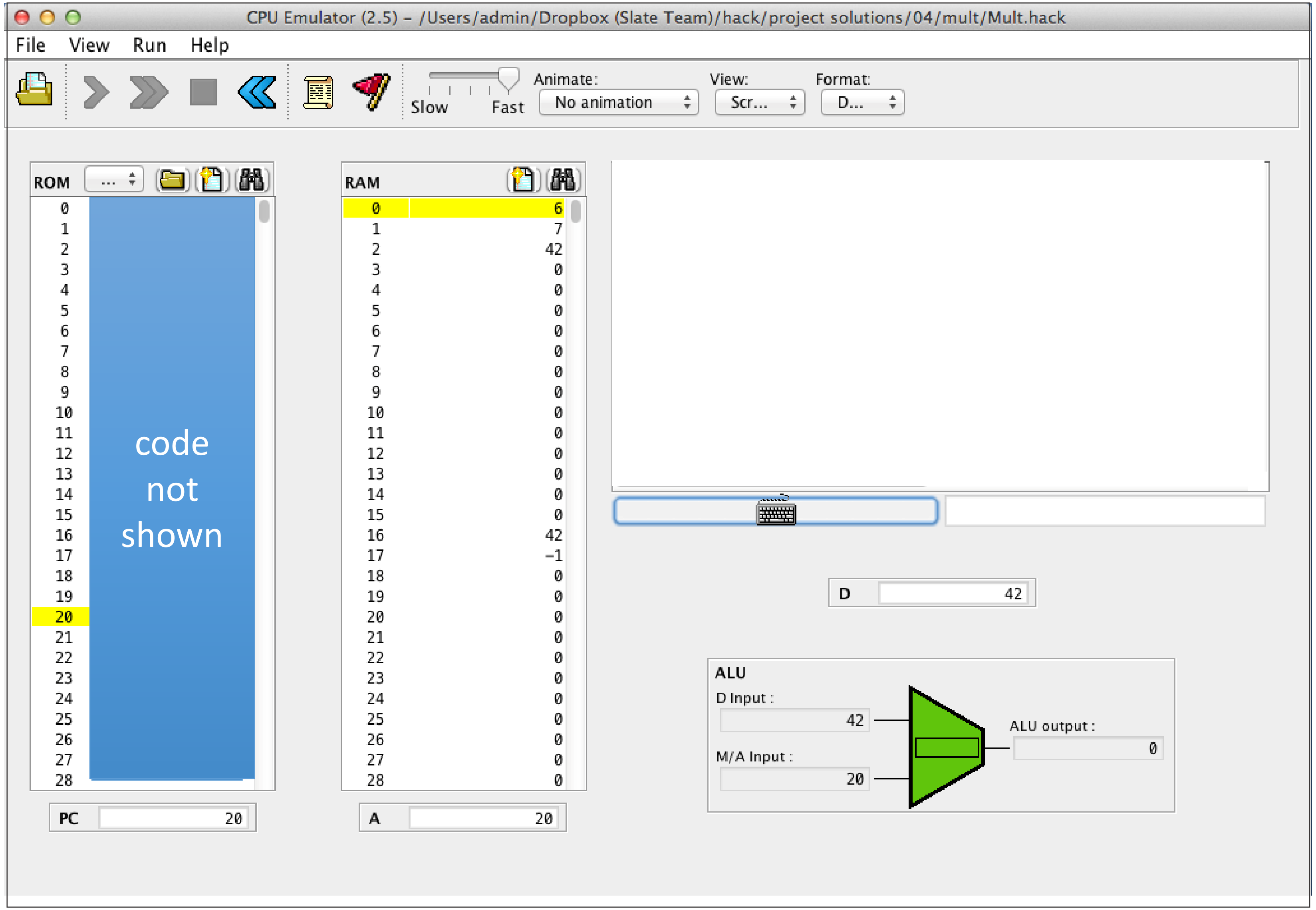
Tasks

- Write a simple algebraic program
- Write a simple interactive program.

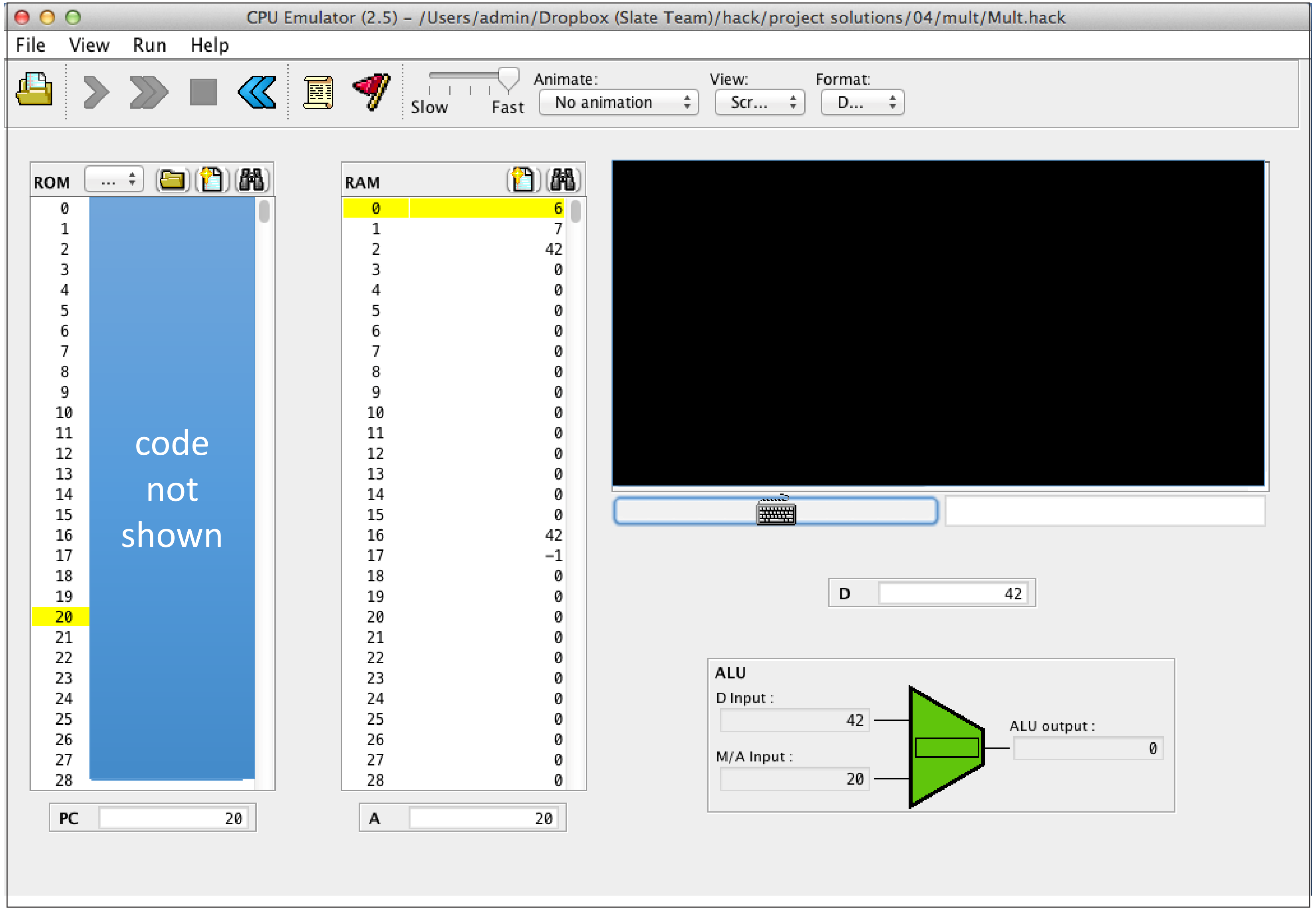
# `Mult`: a program performing `R2 = R0 * R1`



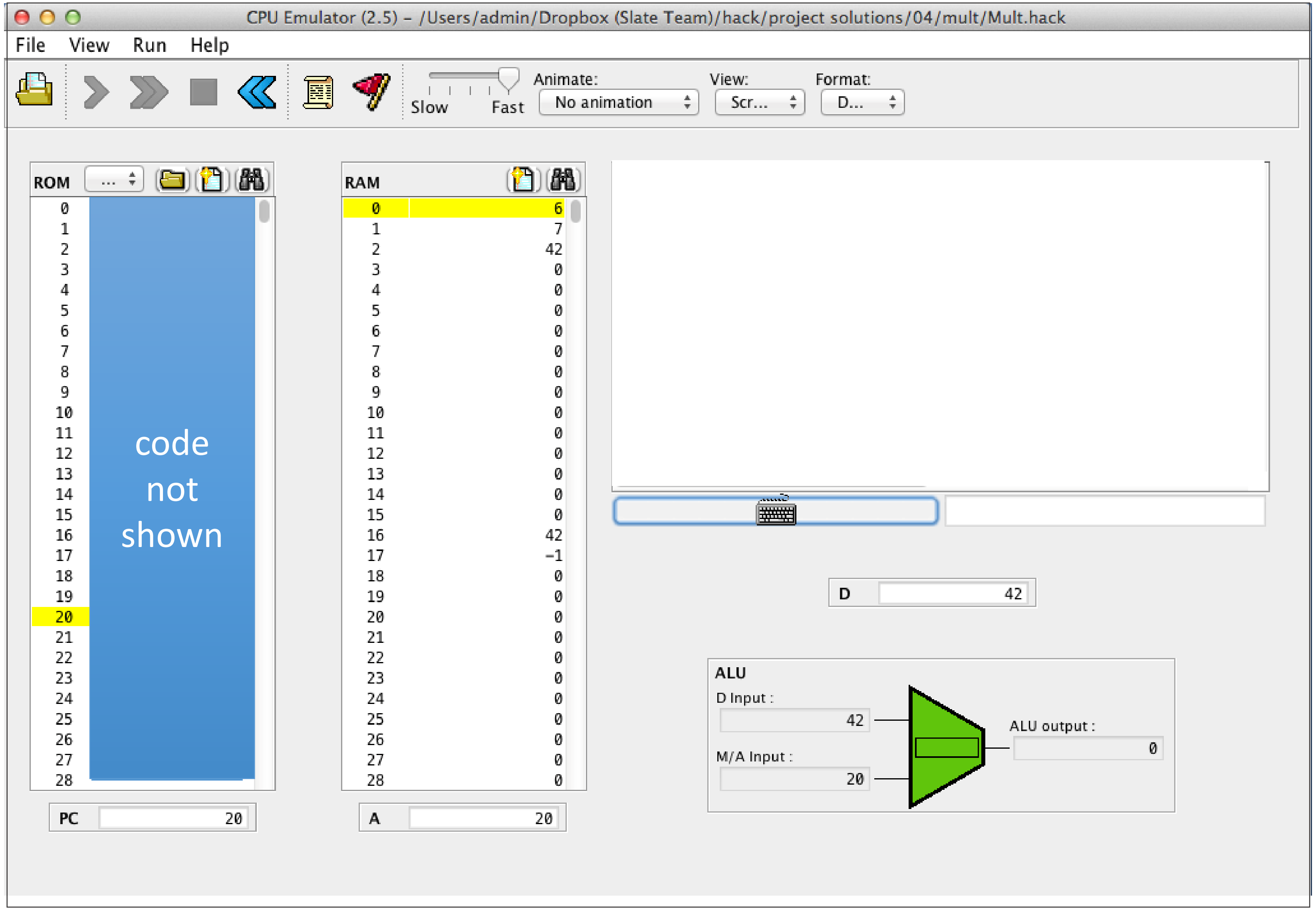
# `Fill`: a simple interactive program



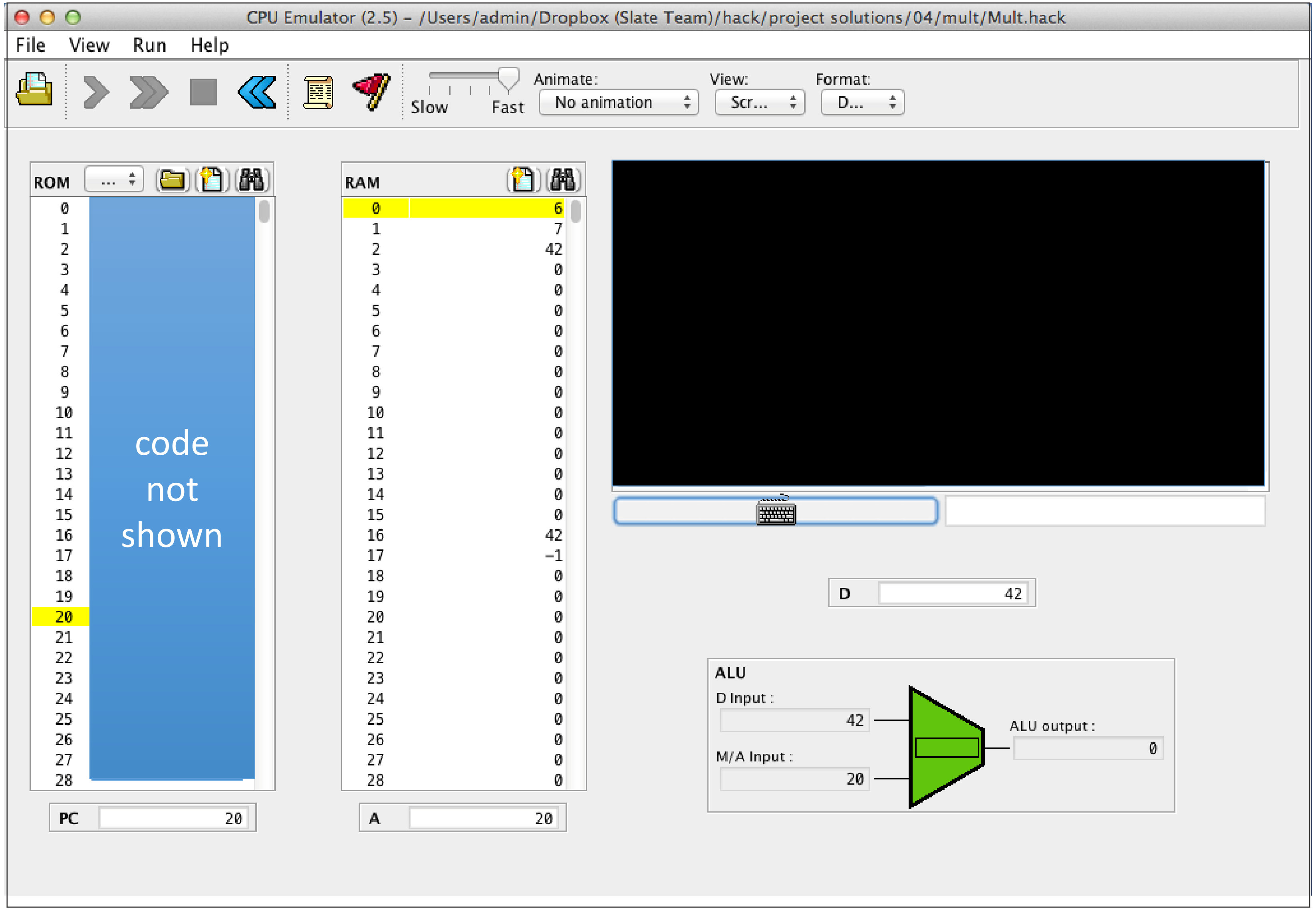
# `Fill`: a simple interactive program



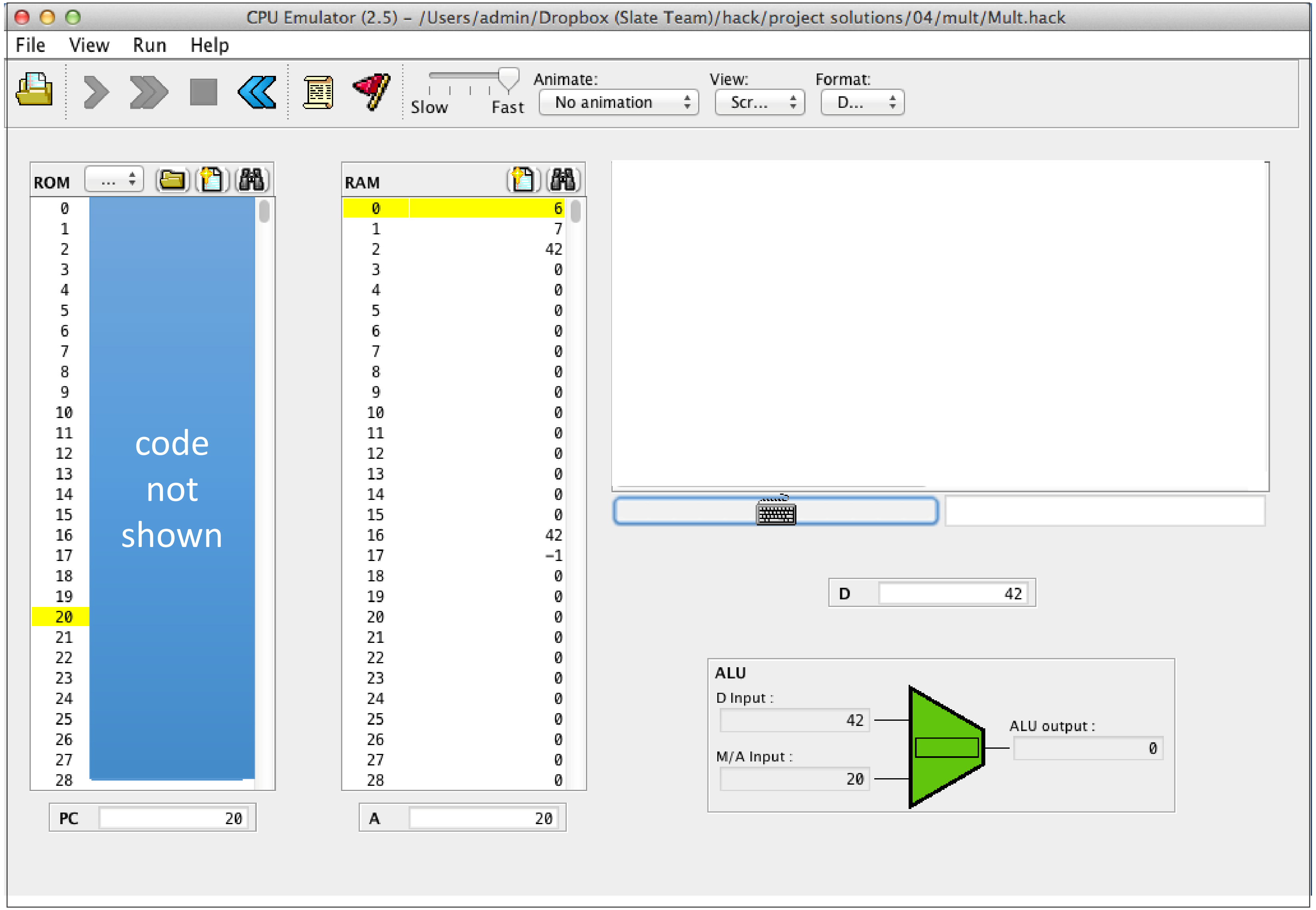
# `Fill`: a simple interactive program



# `Fill`: a simple interactive program



# `Fill`: a simple interactive program



# `Fill`: a simple interactive program

Implementation strategy

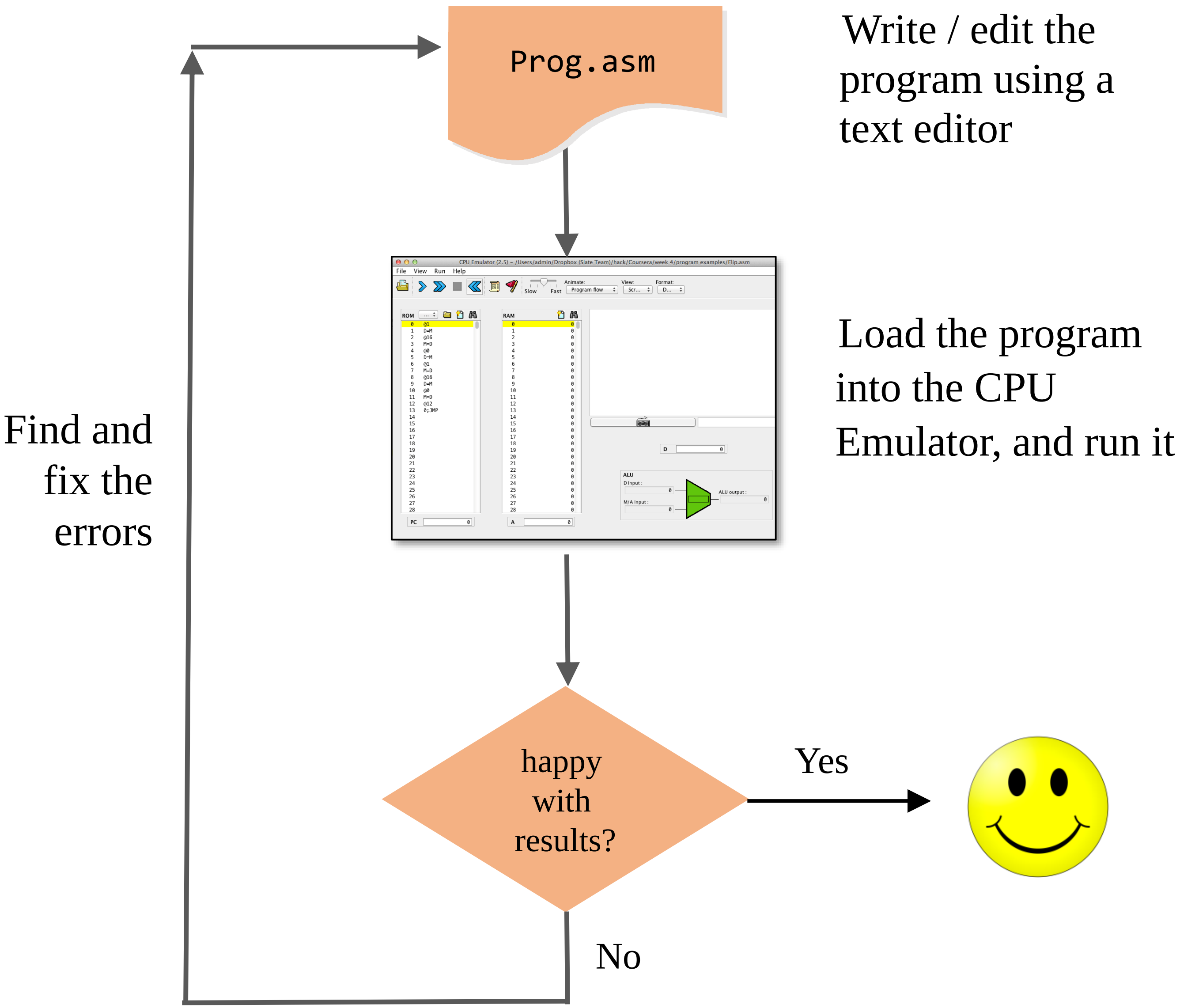
- Listen to the keyboard
- To blacken / clear the screen, write code that fills the entire screen memory map with either “white” or “black” pixels

  (Accessing the memory requires working with pointers)

Testing

- Select “no animation”
- Manual testing (no test scripts).

# Program development process



# Best practice

Well-written low-level code is

- Short
- Efficient
- Elegant
- Self-describing

Technical tips

- Use symbolic variables and labels
- Use sensible variable and label names
- Variables: lower-case
- Labels: upper-case
- Use indentation
- Start with pseudo code.

# Project 4 resources

**From NAND to Tetris**

*Building a Modern Computer From First Principles*

- Home
- Prerequisites
- Syllabus
- Course
- Book
- Software
- Terms
- Papers
- Talks
- Cool Stuff
- About
- Team
- Q&A

## Project 4: Machine Language Programming

### Background

Each hardware platform is designed to execute a certain machine language, expressed using agreed-upon binary codes. Writing programs directly in binary code is a possible, yet an unnecessary, tedium. Instead, we can write such programs in a low-level symbolic language, called *assembly*, and have them translated into binary code by a program called *assembler*. In this project you will write some low-level assembly programs, and will be forever thankful for high-level languages like C and Java. (Actually, assembly programming can be a lot of fun, if you are in the right mood; it's an excellent brain teaser, and it allows you to control the underlying machine directly and completely.

### Objective

To get a taste of low-level programming in machine language, and ... m. In the process of working on this project, you will become familiar w... ic language to machine-language - and you will appreciate visually h... vare platform. These lessons will be learned in the context of writing a... below.

All the necessary project 4 files are available in:

nand2tetris / projects / 04

### Programs

| Program | Description | Comments / Tests |
|---|---|---|
| Mult.asm | **Multiplication:** In the Hack framework, the top 16 RAM words (RAM[0] ... RAM[15]) are also referred to as the so-called *virtual registers* R0 ... R15. With this terminology in mind, this program computes the value R0*R1 and stores the result in R2. | For the purpose of this program, we assume that R0>=0, R1>=0, and R0*R1<32768 (you are welcome to ponder where this value comes from). Your program need not test these conditions, but rather assume that they hold. To test your program, put some values in RAM[0] and RAM[1], run the code, and inspect RAM[2]. The supplied Mult.tst script and Mult.cmp compare file are deigned to test your program "officially", running it on several representative values supplied by us. |

# Machine Language: lecture plan

- ✓ Machine languages
- ✓ Basic elements
- ✓ The Hack computer and machine language
- ✓ The Hack language specification
- ✓ Input / Output
- ✓ Hack programming
  - ❑ Part 1: registers and memory
  - ❑ Part 2: branching, variable, iteration
  - ❑ Part 3: pointers, input/output
- ✓ Project 4 overview

Chapter 4

# Machine Language

These slides support chapter 4 of the book

*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press