



CPU Emulator Tutorial

This program is part of the software suite
that accompanies

The Elements of Computing Systems

by Noam Nisan and Shimon Schocken

MIT Press

www.nand2tetris.org

This software was developed by students at the
Efi Arazi School of Computer Science at IDC

Chief Software Architect: Yaron Ukrainitz

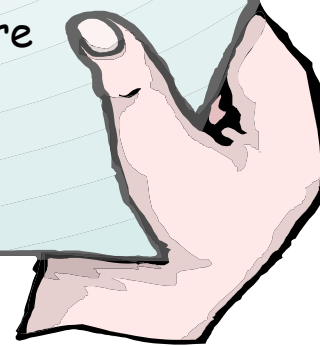
Background

The Elements of Computing Systems evolves around the construction of a complete computer system, done in the framework of a 1- or 2-semester course.

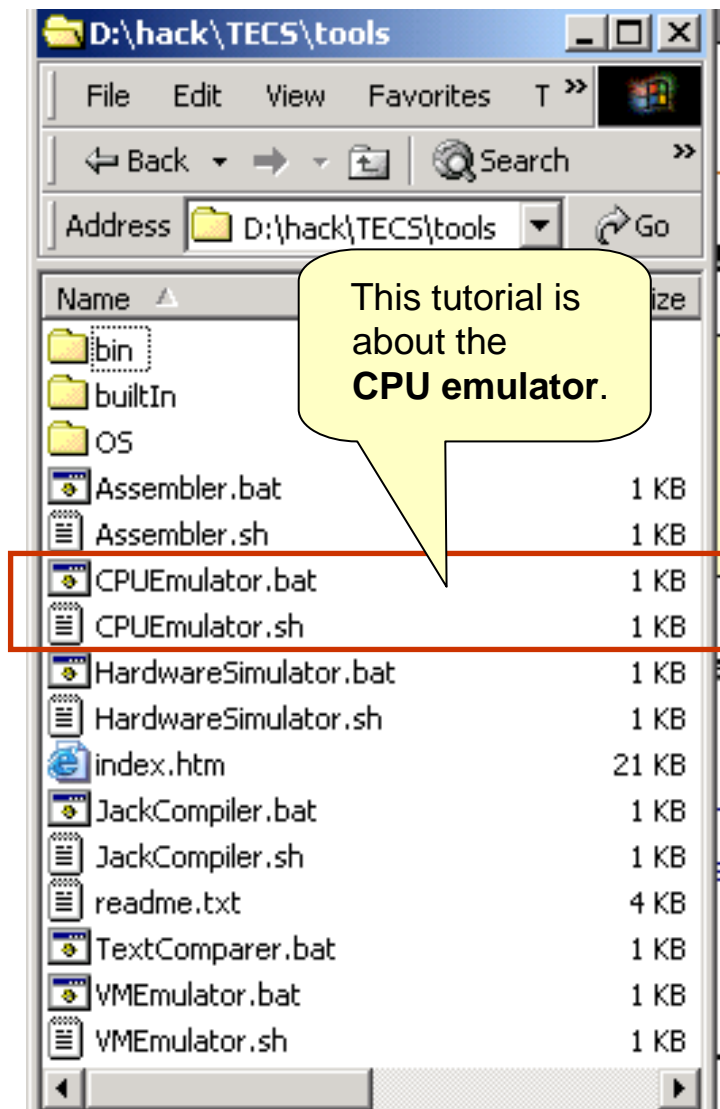
In the first part of the book/course, we build the hardware platform of a simple yet powerful computer, called Hack. In the second part, we build the computer's software hierarchy, consisting of an assembler, a virtual machine, a simple Java-like language called Jack, a compiler for it, and a mini operating system, written in Jack.

The book/course is completely self-contained, requiring only programming as a pre-requisite.

The book's web site includes some 200 test programs, test scripts, and all the software tools necessary for doing all the projects.



The book's software suite



(All the supplied tools are dual-platform: **xxx.bat** starts **xxx** in Windows, and **xxx.sh** starts it in Unix)

Simulators

(HardwareSimulator, CPUEmulator, VMEulator):

- Used to build hardware platforms and execute programs;
- Supplied by us.

Translators (Assembler, JackCompiler):

- Used to translate from high-level to low-level;
- Developed by the students, using the book's specs; Executable solutions supplied by us.

Other

- **bin**: simulators and translators software;
- **builtIn**: executable versions of all the logic gates and chips mentioned in the book;
- **os**: executable version of the Jack OS;
- **TextComparer**: a text comparison utility.

Tutorial Objective



The Hack computer

This CPU emulator simulates the operations of the Hack computer, built in chapters 1-5 of the book.

Hack -- a 16-bit computer equipped with a screen and a keyboard -- resembles hand-held computers like game machines, PDA's, and cellular telephones.

Before such devices are actually built in hardware, they are planned and simulated in software.

The CPU emulator is one of the software tools used for this purpose.



CPU Emulator Tutorial

- I. [Basic Platform](#)
- II. [I/O devices](#)
- III. [Interactive simulation](#)
- IV. [Script-based simulation](#)
- V. [Debugging](#)

Relevant reading (from “*The Elements of Computing Systems*”):

- Chapter 4: *Machine Language*
- Chapter 5: *Computer Architecture*
- Appendix B: *Test Scripting Language*

CPU Emulator Tutorial



The Hack Computer Platform (simulated)

CPU Emulator (1.4b3) - G:\examples\Rect.asm

File View Run Help

Animate: Program flow View: Screen Format: Decimal

ROM Asm

Address	Code
0	@0
1	D=M
2	@23
3	D; JLE
4	@16
5	M=D
6	@16384
7	D=A
8	@17
9	M=D
10	@17
11	A=M
12	M=-1
13	@17
14	D=M
15	@32
16	D=D+A
17	@17
18	M=D
19	@16
20	MD=M-1
21	@10
22	D; JGT
23	@23
24	0; JMP
25	
26	
27	
28	

RAM

Address	Value
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	

PC 15 A 17

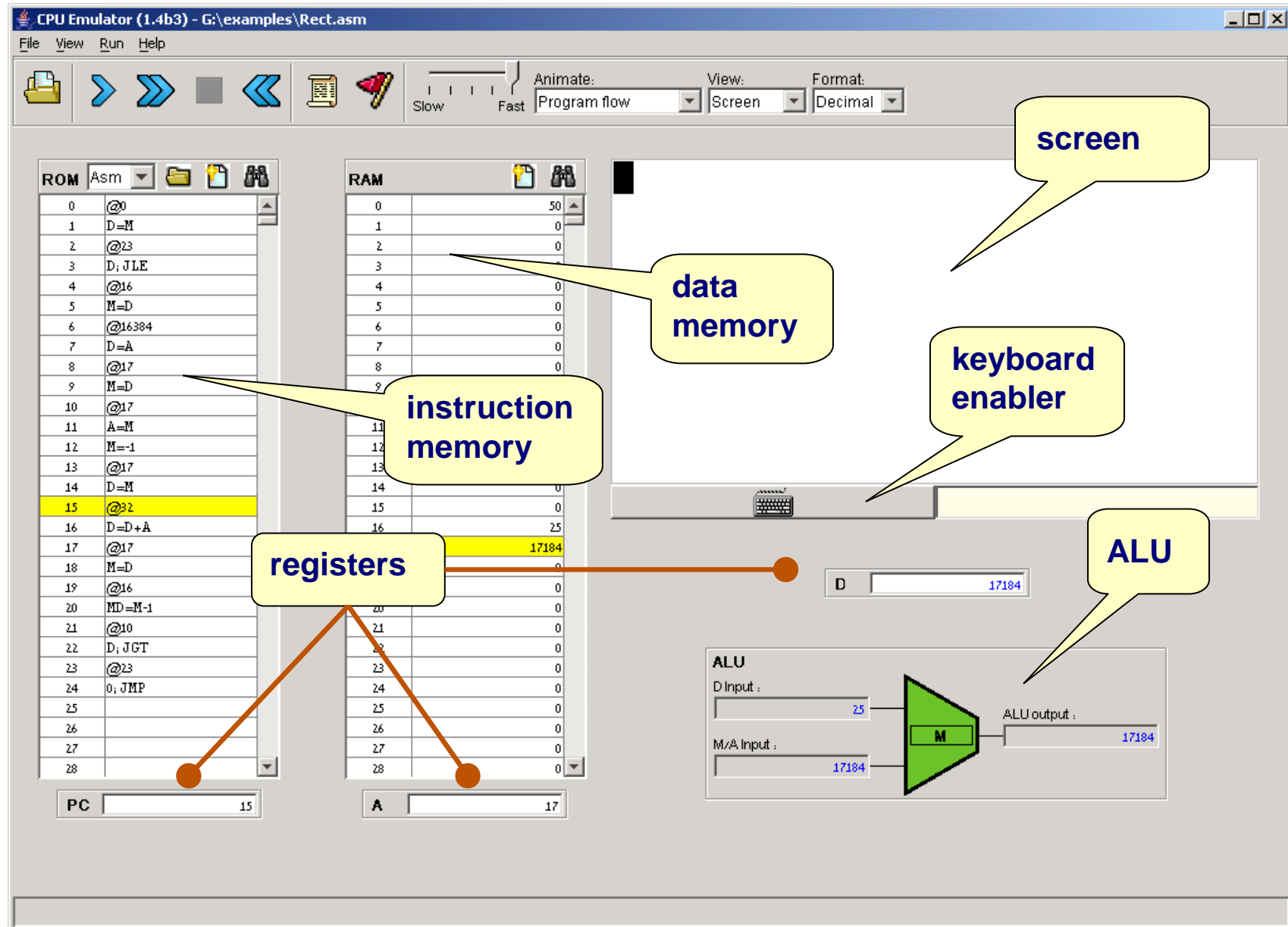
Travel Advice:

This tutorial includes some examples of programs written in the Hack machine language (chapter 4).

There is no need however to understand either the language or the programs in order to learn how to use the CPU emulator.

Rather, it is only important to grasp the *general logic* of these programs, as explained (when relevant) in the tutorial.

The Hack Computer Platform



Instruction memory

The screenshot shows the CPU Emulator (1.4b3) interface. The main window displays the instruction memory (ROM) and RAM. The ROM is currently set to 'Asm' view, showing a list of instructions. The RAM is shown in hexadecimal view. The Program Counter (PC) is set to 15, and the next instruction is highlighted. The ALU output is 17184.

Annotations:

- The loaded code can be viewed either in binary, or in symbolic notation (present view)** (points to the ROM view selector).
- Instruction memory (32K): Holds a machine language program** (points to the RAM area).
- Next instruction is highlighted** (points to the instruction at PC 15: @32).
- Program counter (PC) (16-bit): Selects the next instruction.** (points to the PC register).

ROM	Asm
0	@0
1	D=M
2	@23
3	D; JLE
4	@16
5	M=D
6	@16384
7	D=A
8	@17
9	M=D
10	@17
11	A=M
12	M=-1
13	@17
14	D=M
15	@32
16	D=D+A
17	@17
18	M=D
19	@16
20	MD=M-1
21	@10
22	D; JGT
23	@23
24	0; JMP
25	
26	
27	
28	

RAM	Hex
0	
1	
2	
3	
4	
5	
6	0
7	0
8	
9	
10	
11	
12	
13	
14	
15	0
16	25
17	17184
18	0
19	
20	
21	
22	
23	0
24	0
25	0
26	0
27	0
28	0

PC: 15

D: 17184

ALU Input: 25

M/A Input: M

ALU output: 17184

Data memory (RAM)

The screenshot shows the CPU Emulator (1.4b3) interface. The ROM window displays assembly code, with line 15 highlighted. The RAM window shows memory addresses 0 to 28, with address 17 highlighted. The PC register is 15, and the A register is 17. Two callouts provide additional information:

Data memory (32K RAM), used for:

- General-purpose data storage (variables, arrays, objects, etc.)
- Screen memory map
- Keyboard memory map

Address (A) register, used to:

- Select the current RAM location

OR

- Set the Program Counter (PC) for jumps (relevant only if the current instruction includes a jump directive).

Registers

Registers (all 16-bit):

- **D**: Data register
- **A**: Address register
- **M**: Stands for the memory register whose address is the current value of the Address register

M (=RAM[A])

A

ALU

D Input : 25

M/A Input : 17184

ALU output : 17184

PC: 15

A: 17

ROM	Asm
0	@0
1	D=M
2	@23
3	D; JLE
4	@16
5	M=D
6	@16384
7	D=A
8	@17
9	M=D
10	@17
11	A=M
12	M=-1
13	@17
14	D=M
15	@32
16	D=D+A
17	@17
18	M=D
19	@16
20	MD=M-1
21	@10
22	D; JGT
23	@23
24	0; JMP
25	
26	
27	
28	

RAM	
0	50
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
12	0
13	0
14	0
15	0
16	25
17	17184
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

Arithmetic/Logic Unit

The screenshot shows a CPU Emulator window titled "CPU Emulator (1.4b3) - G:\examples\Rect.asm". It features a menu bar (File, View, Run, Help) and a toolbar with icons for file operations, execution (single step, break, reset), and speed control (Slow, Fast). The main area is divided into three panes: ROM (Asm), RAM, and ALU.

ROM (Asm) Pane: A list of instructions. Instruction 14, "D=M", is highlighted in red and labeled "Current instruction". Instruction 15, "@32", is highlighted in yellow.

RAM Pane: A table of memory addresses and values. Address 17 contains the value 17184 and is highlighted in yellow. It is labeled "M (=RAM[A])".

ALU Pane: Shows the ALU operation. The "D Input" is 25 and the "M/A Input" is 17184. The ALU output is 17184. The operation is labeled "M".

Registers: The PC (Program Counter) is 15 and the A (Accumulator) register is 17.

Annotations: Orange callout boxes point to the "Current instruction" (D=M), the memory location M (=RAM[A]) at address 17, and the ALU operation.

Arithmetic logic unit (ALU)

- The ALU can compute various arithmetic and logical functions (let's call them f) on subsets of the three registers $\{M, A, D\}$
- All ALU instructions are of the form $\{M, A, D\} = f(\{M, A, D\})$ (e.g. $M=M-1$, $MD=D+A$, $A=0$, etc.)
- The ALU operation (LHS destination, function, RHS operands) is specified by the current instruction.



I/O devices: screen and keyboard

The screenshot shows the CPU Emulator (1.4b3) interface. On the left, there are two tables for ROM and RAM. The ROM table has columns for address (0-17) and assembly code. The RAM table has columns for address (0-17) and data (0). In the center, there is a large white rectangle representing the simulated screen. Below the screen is a simulated keyboard icon. To the right of the keyboard is a register labeled 'D' with a value of 0. At the bottom right, there is an ALU block with inputs for 'D Input' and 'M/A Input', both set to 0, and an 'ALU output' field showing 0. A status bar at the bottom left indicates 'Script restarted'.

Simulated screen: 256 columns by 512 rows, black & white memory-mapped device. The pixels are continuously refreshed from respective bits in an 8K memory-map, located at RAM[16384] - RAM[24575].

Simulated keyboard:
One click on this button causes the CPU emulator to intercept all the keys subsequently pressed on the real computer's keyboard; another click disengages the real keyboard from the emulator.

Screen action demo

Perspective: That's how computer programs put images (text, pictures, video) on the screen: they write bits into some display-oriented memory device.

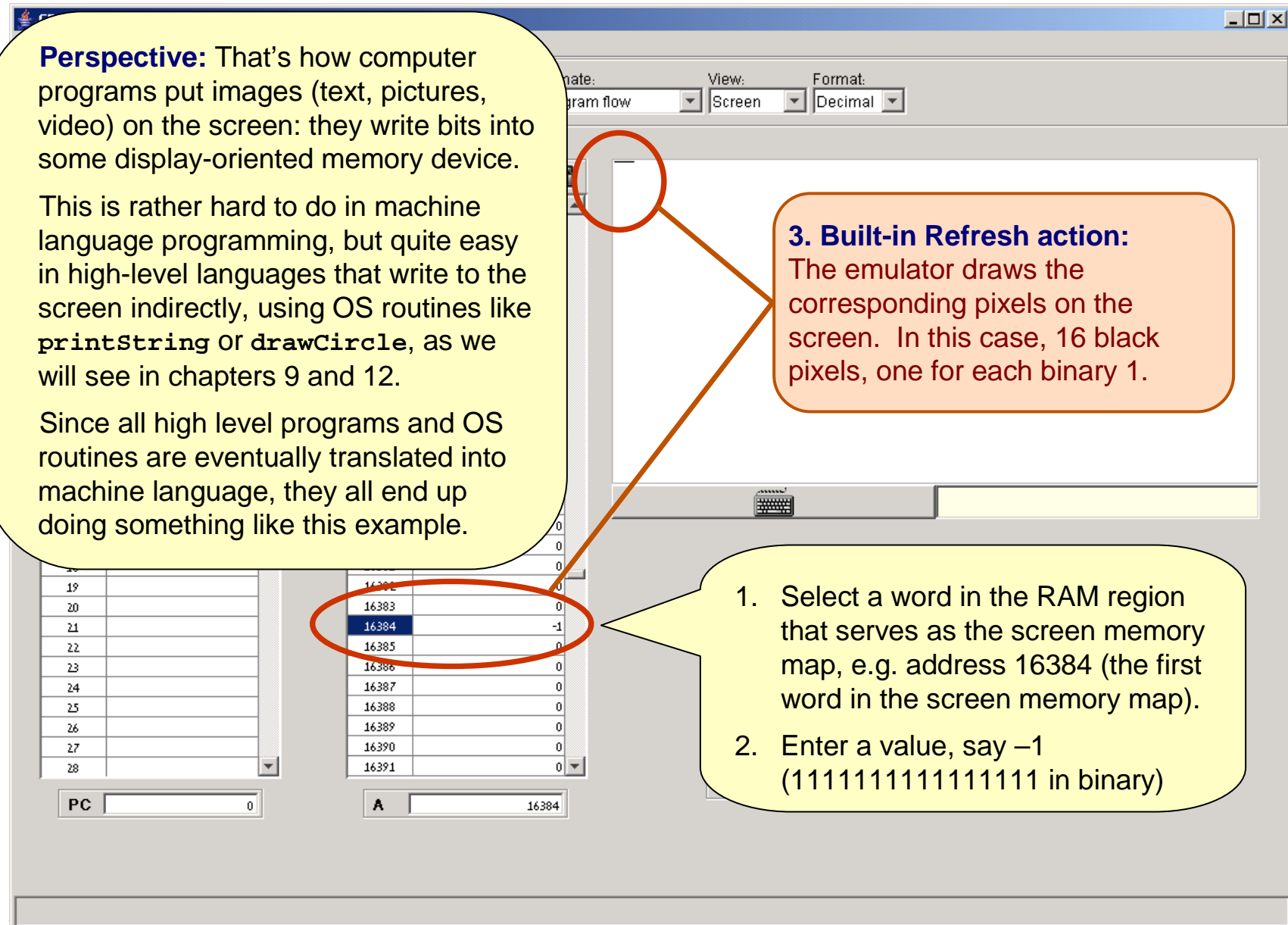
This is rather hard to do in machine language programming, but quite easy in high-level languages that write to the screen indirectly, using OS routines like `printString` or `drawCircle`, as we will see in chapters 9 and 12.

Since all high level programs and OS routines are eventually translated into machine language, they all end up doing something like this example.

3. Built-in Refresh action:

The emulator draws the corresponding pixels on the screen. In this case, 16 black pixels, one for each binary 1.

1. Select a word in the RAM region that serves as the screen memory map, e.g. address 16384 (the first word in the screen memory map).
2. Enter a value, say -1 (1111111111111111 in binary)



Keyboard action demo

The screenshot shows the CPU Emulator (1.4b3) interface. On the left, the ROM and RAM memory maps are displayed. The RAM map shows addresses from 24548 to 24576, with address 24576 highlighted. A yellow callout bubble points to address 24576, stating: "Keyboard memory map (a single 16-bit memory location)".

On the right, a large yellow callout bubble contains the following instructions:

1. Click the keyboard enabler
2. Press some key on the real keyboard, say "S"

Below these instructions, a keyboard icon is shown. A yellow bar highlights the keyboard icon, and a yellow callout bubble points to it, stating: "3. Watch here:".

Below the keyboard icon, a register D is shown with a value of 0. Below that, the ALU is shown with inputs D Input (0) and M/A Input (0), and an output ALU output (0).

At the bottom left, a status bar indicates "Script restarted".

Keyboard action demo

Perspective: That's how computer programs read from the keyboard: they peek some keyboard-oriented memory device, one character at a time.

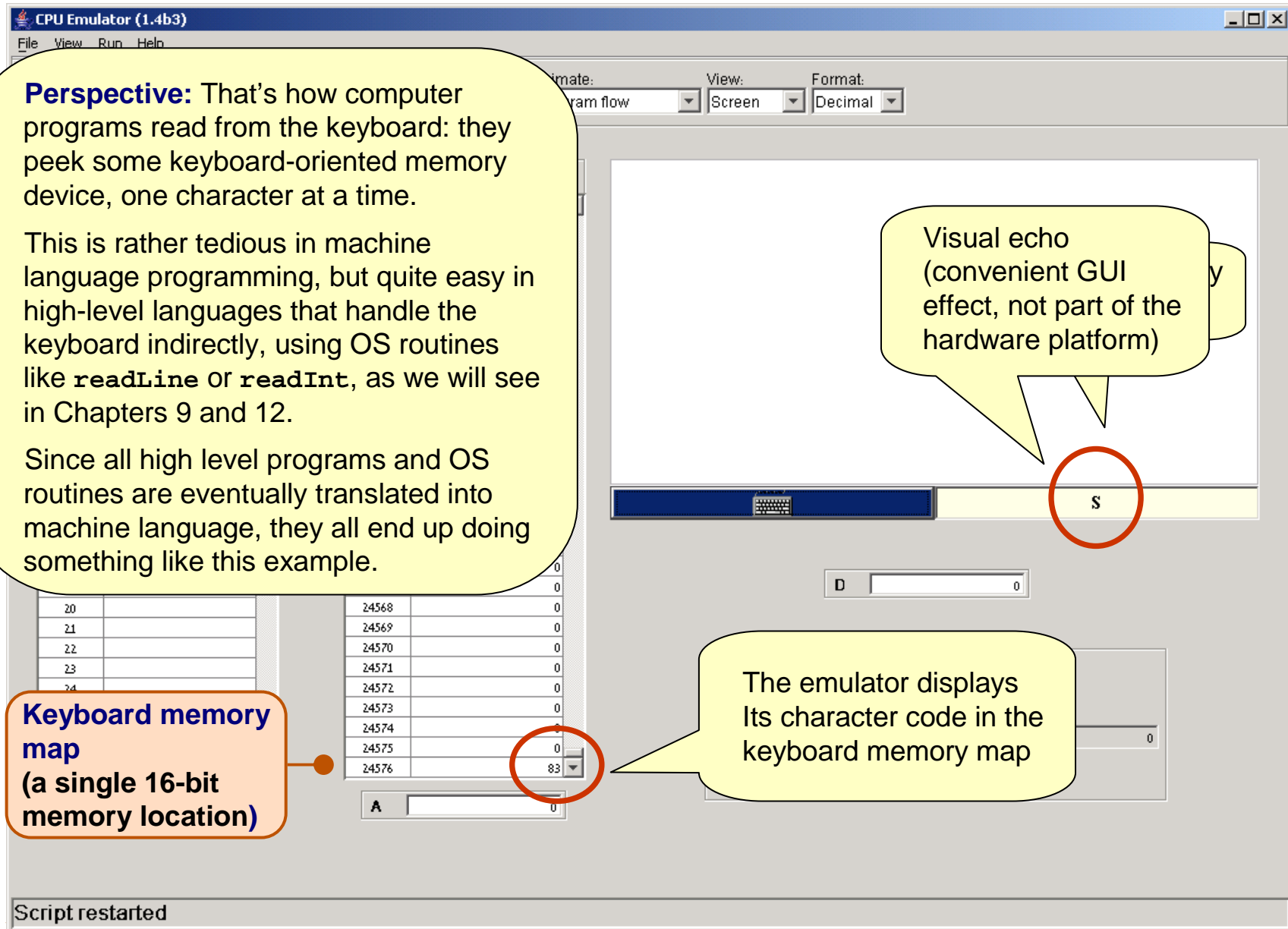
This is rather tedious in machine language programming, but quite easy in high-level languages that handle the keyboard indirectly, using OS routines like `readLine` or `readInt`, as we will see in Chapters 9 and 12.

Since all high level programs and OS routines are eventually translated into machine language, they all end up doing something like this example.

Keyboard memory map
(a single 16-bit memory location)

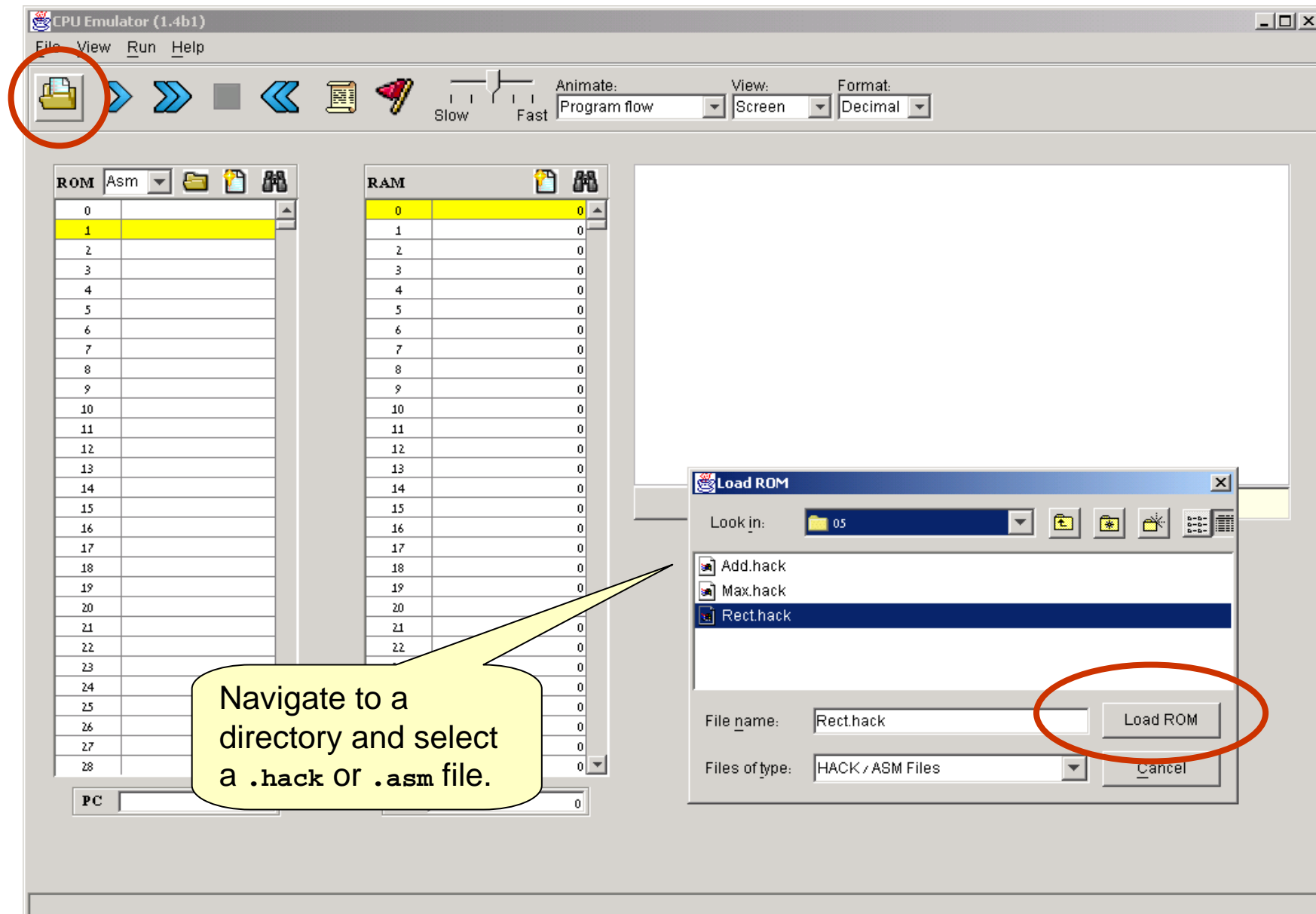
Visual echo
(convenient GUI effect, not part of the hardware platform)

The emulator displays its character code in the keyboard memory map

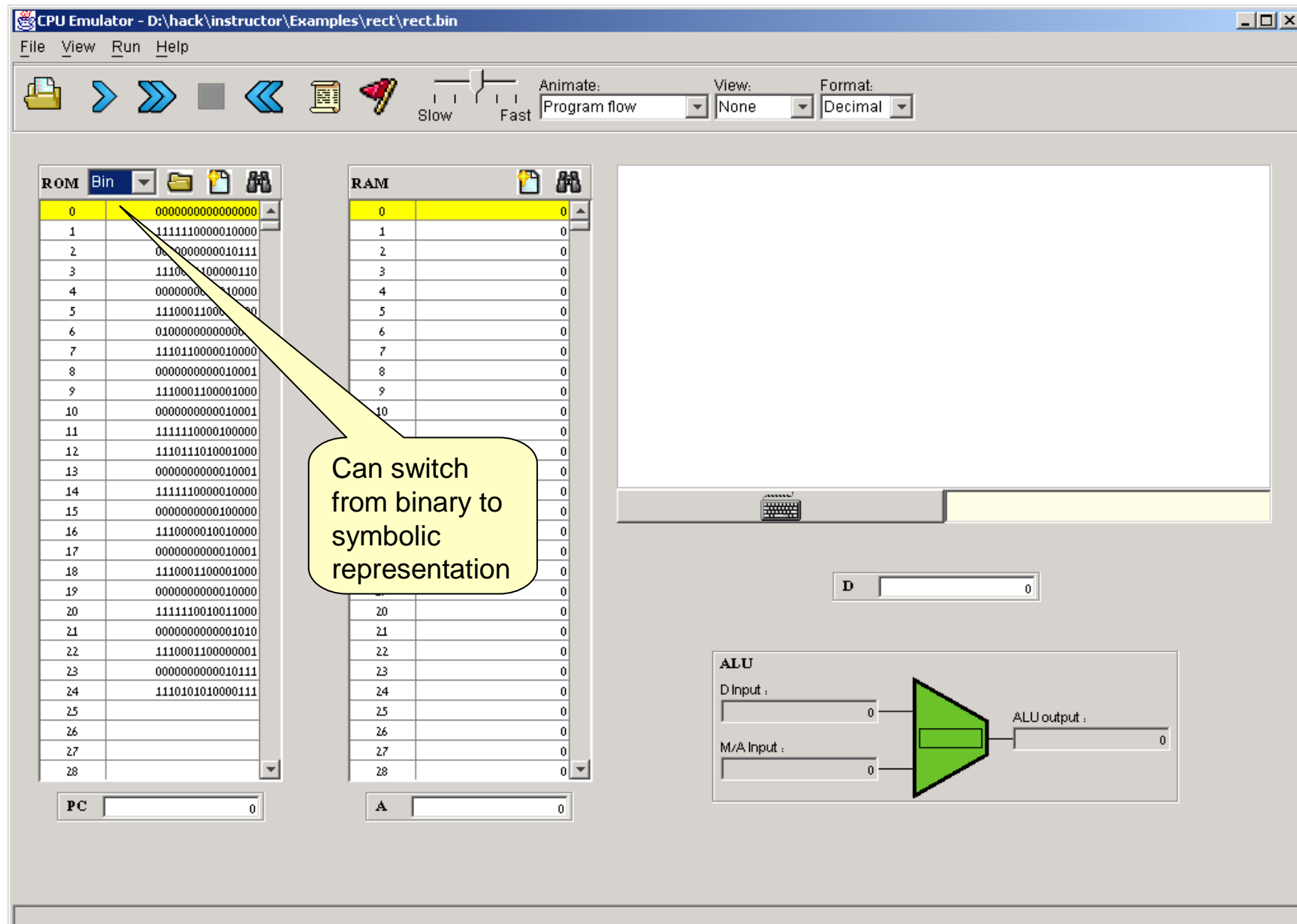




Loading a program



Loading a program



Running a program

The screenshot shows the CPU Emulator window titled "CPU Emulator - D:\hack\instructor\Examples\rect\rect.bin". The interface includes a menu bar (File, View, Run, Help), a toolbar with icons for file operations and execution, and a speed control slider. The main area is divided into three panes: ROM (Asm), RAM, and a large display window. The ROM pane shows assembly code, and the RAM pane shows memory values. The display window is currently blank. A keyboard icon is visible at the bottom of the display area.

2. Click the "run" button. (Callout pointing to the right-pointing arrow icon in the toolbar)

1. Enter a number, say 50. (Callout pointing to the RAM[0] field, which contains the value 50)

3. To speed up execution, use the speed control slider (Callout pointing to the slider between Slow and Fast)

4. Watch here (Callout pointing to the main display window)

Program's description: Draws a rectangle at the top left corner of the screen. The rectangle's width is 16 pixels, and its length is determined by the current contents of RAM[0].

Note: There is no need to understand the program's code in order to understand what's going on.

Running a program

The screenshot shows the CPU Emulator window with the following components and annotations:

- ROM Asm:** A list of assembly instructions. Instruction 12, `M=-1`, is highlighted in yellow. A yellow callout bubble points to it with the text: "2. Click the 'run' button."
- RAM:** A list of memory addresses and values. Address 17 contains the value 17536 and is highlighted in yellow. A yellow callout bubble points to it with the text: "1. Enter a number, say 50."
- Speed Control:** A slider between "Slow" and "Fast". An orange callout bubble points to it with the text: "3. To speed up execution, use the speed control slider"
- Execution Area:** A large white rectangle representing the screen. A small black rectangle is visible in the top-left corner. An orange callout bubble points to it with the text: "4. Watch here"
- Program's description:** A yellow callout bubble at the bottom right contains the text: "Program's description: Draws a rectangle at the top left corner of the screen. The rectangle's width is 16 pixels, and its length is determined by the current contents of RAM[0]."
- Note:** A yellow callout bubble at the bottom right contains the text: "Note: There is no need to understand the program's code in order to understand what's going on."
- PC:** A register showing the value 12.
- A:** A register showing the value 17536.
- D:** A register showing the value 14.

Hack programming at a glance (optional)

Next instruction is $M = -1$.

Since presently $A = 17536$, the next ALU instruction will effect $RAM[17536] = 1111111111111111$. The 17536 address, which falls in the screen memory map, corresponds to the row just below the rectangle's current bottom. In the next screen refresh, a new row of 16 black pixels will be drawn there.

Program action:

Since $RAM[0]$ happens to be 50, the program draws a 16X50 rectangle. In this example the user paused execution when there are 14 more rows to draw.

Program's description: Draws a rectangle at the top left corner of the screen. The rectangle's width is 16 pixels, and its length is determined by the current contents of $RAM[0]$.

Note: There is no need to understand the program's code in order to understand what's going on.

The screenshot shows a CPU emulator window. On the left, an assembly code list is displayed with instructions like `D=A`, `@17`, `M=D`, `A=M`, `M=-1` (highlighted), `@17`, `D=M`, `@32`, `D=D+A`, `@17`, `M=D`, `@16`, `MD=M-1`, `@10`, `D: JGT`, `@23`, and `0: JMP`. The PC register is 12, and the A register is 17536. On the right, a screen display shows a 16x50 rectangle of black pixels. A callout points to the bottom of the rectangle, indicating the next row to be drawn. Another callout points to the value 14 in the D register, indicating the current row count.

Animation options

The screenshot shows the CPU Emulator window with the title bar "CPU Emulator - D:\hack\instructor\Examples\rect\rect.asm". The menu bar includes "File", "View", "Run", and "Help". The toolbar contains icons for file operations and execution control, including a slider for "Slow" to "Fast" animation speed. The "Animate:" dropdown is set to "Program & data flow", "View:" is "None", and "Format:" is "Decimal".

The main window is divided into two panes: "ROM Asm" and "RAM". The "ROM Asm" pane shows assembly instructions, with instruction 13 (@17) highlighted in yellow. The "RAM" pane shows memory addresses and values, with address 17536 highlighted in yellow. A yellow callout bubble points to the "Slow" to "Fast" slider, stating: "Controls execution (and animation) speed." An orange callout bubble points to the highlighted instruction and RAM location, stating: "The simulator can animate both program flow and data flow". A large yellow callout bubble on the right contains the following text:

Animation control:

- **Program flow** (default): highlights the current instruction in the instruction memory and the currently selected RAM location
- **Program & data flow**: animates all program and data flow in the computer
- **No animation**: disables all animation

Usage tip: To execute any non-trivial program quickly, select *no animation*.



Interactive VS Script-Based Simulation

A program can be executed and debugged:

- **Interactively**, by ad-hoc playing with the emulator's GUI (as we have done so far in this tutorial)
- **Batch-ly**, by running a pre-planned set of tests, specified in a *script*.

Script-based simulation enables planning and using tests that are:

- Pro-active
- Documented
- Replicable
- Complete (as much as possible)

Test scripts:

- Are written in a *Test Description Language* (described in Appendix B)
- Can cause the emulator to do anything that can be done interactively, and quite a few things that cannot be done interactively.

The basic setting

The screenshot shows the CPU Emulator (1.4b1) interface. The title bar indicates the file path: G:\shimon progs\Max\Max.asm. The menu bar includes File, View, Run, and Help. The toolbar contains icons for file operations, execution (single step, step over, step into, run), and a speed slider (Slow to Fast). The 'Animate' dropdown is set to 'Program flow', 'View' is 'Script', and 'Format' is 'Decimal'.

On the left, the ROM window displays assembly code. A red box highlights the first 15 lines of code, which are also highlighted in yellow. A yellow callout bubble labeled 'tested program' points to this section. The RAM window on the right shows memory addresses 0 to 28, with values 47 at addresses 0 and 1, and 0 elsewhere. A yellow callout bubble labeled 'test script' points to the script window on the right.

The script window contains the following assembly code:

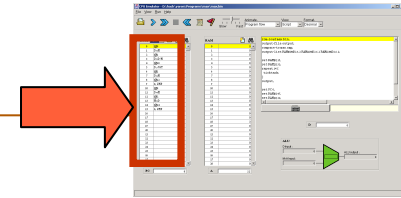
```
load Max.asm,  
output-file Max.out,  
// compare-to max.cmp,  
output-list RAM[0]%%D2.6.2  
    RAM[1]%%D2.6.2  
    RAM[2]%%D2.6.2;  
  
// test 1: max(15,32)  
set RAM[0]15,  
set RAM[1]32,  
repeat 14 {  
    ticktock;  
}  
output;
```

At the bottom, the PC register is 0, and the ALU window shows D Input: 0, M/A Input: 0, and ALU output: 0.

New script loaded: G:\shimon progs\Max\Max.tst

Example: Max.asm

Note: For now, it is not necessary to understand either the Hack machine language or the Max program. It is only important to grasp the program's logic. But if you're interested, we give a language overview on the right.



Hack language at a glance:

- **(label)** // defines a label
- **@xxx** // sets the **A** register
// to xxx's value
- The other commands are self-explanatory; Jump directives like **JGT** and **JMP** mean "Jump to the address currently stored in the **A** register"
- Before any command involving a RAM location (**M**), the **A** register must be set to the desired RAM address (**@address**)
- Before any command involving a jump, the **A** register must be set to the desired ROM address (**@label**).

```
// Computes M[2]=max(M[0],M[1]) where M stands for RAM
@0
D=M           // D = M[0]
@1
D=D-M         // D = D - M[1]
@FIRST_IS_GREATER
D;JGT         // If D>0 goto FIRST_IS_GREATER
@1
D=M           // D = M[1]
@SECOND_IS_GREATER
0;JMP         // Goto SECOND_IS_GREATER
(FIRST_IS_GREATER)
@0
D=M           // D=first number
(SECOND_IS_GREATER)
@2
M=D           // M[2]=D (greater number)
(INFINITE_LOOP)
@INFINITE_LOOP // Infinite loop (our standard
0;JMP         // way to terminate programs).
```

Sample test script: `Max.tst`

```
// Load the program and set up:
load Max.asm,
output-file Max.out,
compare-to Max.cmp,
output-list RAM[0]%D2.6.2
          RAM[1]%D2.6.2
          RAM[2]%D2.6.2;

// Test 1: max(15,32)
set RAM[0] 15,
set RAM[1] 32;
repeat 14 {
    ticktock;
}
output; // to the Max.out file

// Test 2: max(47,22)
set PC 0, // Reset prog. counter
set RAM[0] 47,
set RAM[1] 22;
repeat 14 {
    ticktock;
}
output;

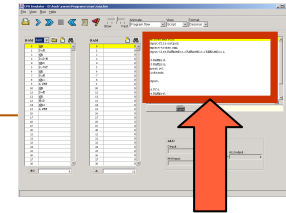
// test 3: max(12,12)
// Etc.
```

The scripting language
has commands for:

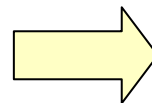
- Loading programs
- Setting up output and compare files
- Writing values into RAM locations
- Writing values into registers
- Executing the next command (“ticktock”)
- Looping (“repeat”)
- And more (see Appendix B).

Notes:

- As it turns out, the Max program requires 14 cycles to complete its execution
- All relevant files (`.asm`, `.tst`, `.cmp`) must be present in the same directory.

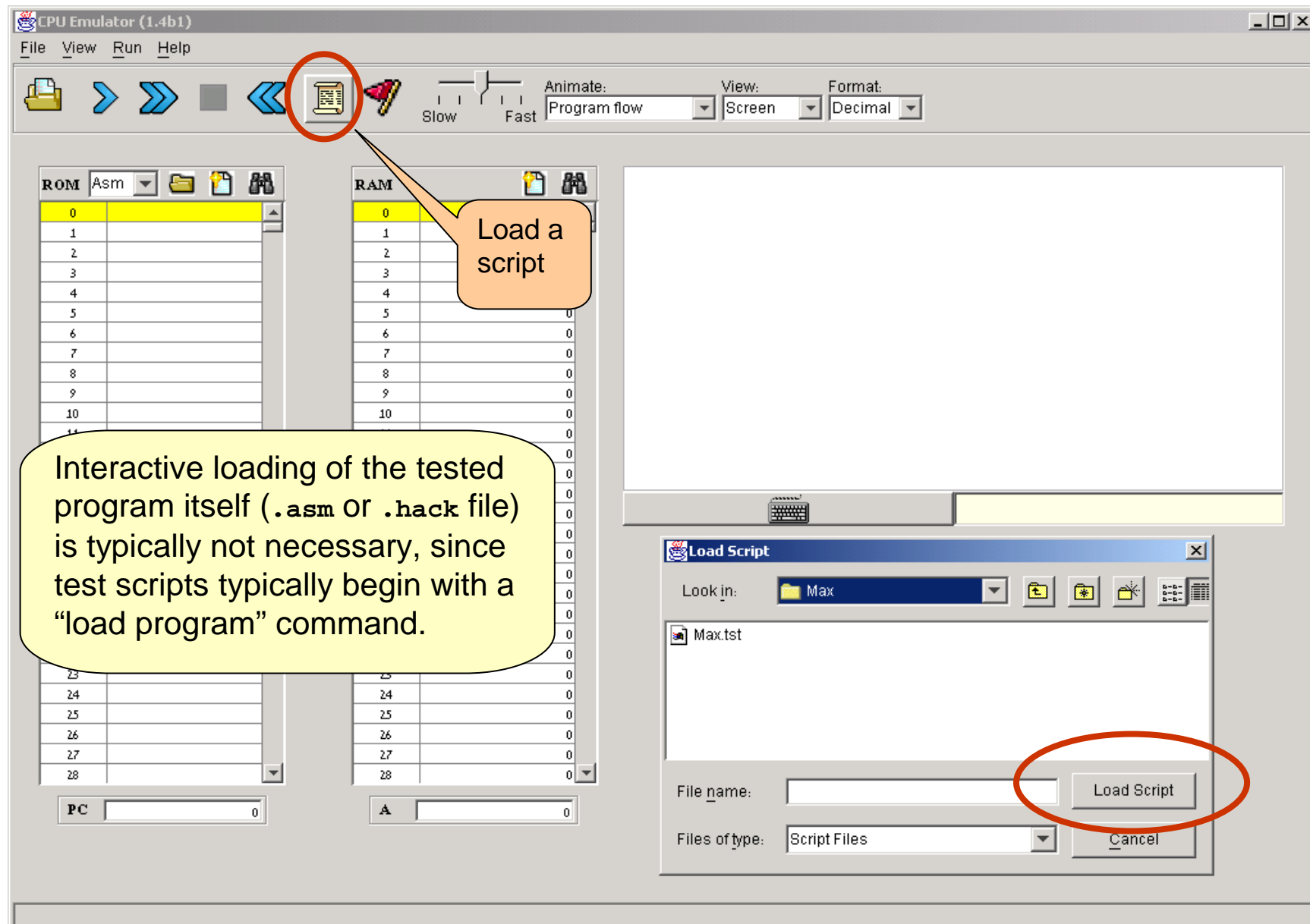


Output



RAM[0]	RAM[1]	RAM[2]
15	32	32
47	22	47

Using test scripts



Using test scripts

The screenshot shows the CPU Emulator (1.4b1) interface. The title bar reads "CPU Emulator (1.4b1) - G:\shimon progs\Max\Max.asm". The menu bar includes File, View, Run, and Help. The toolbar contains icons for file operations and execution, along with a speed control slider (Slow to Fast) and dropdown menus for Animate (Program flow), View (Script), and Format (Decimal).

On the left, there are two tables: ROM and RAM. The ROM table shows addresses 0 to 26 with corresponding assembly instructions. The RAM table shows addresses 0 to 11 with values. Below these are registers A and B, and an ALU output display.

On the right, a script editor window displays the following code:

```
load Max.asm,  
output-file Max.out,  
// compare-to max.asm,  
output-list RAM[0]%%D2.6.2  
    RAM[1]%%D2.6.2  
    RAM[2]%%D2.6.2;  
  
// test1: max(15,32)  
set RAM[0]15,  
set RAM[1]32;  
repeat 14 {  
    ticktock;  
}
```

Callouts provide additional information:

- Speed control:** Points to the speed slider in the toolbar.
- Load a script:** Points to the script editor window.
- Script = a series of simulation steps, each ending with a semicolon;** A general definition of a script.
- Important point:** Whenever an assembly program (.asm file) is loaded into the emulator, the program is assembled on the fly into machine language code, and this is the code that actually gets loaded. In the process, all comments and white space are removed from the code, and all symbols resolve to the numbers that they stand for.
- Execute step repeatedly:** Points to the "Execute step repeatedly" button in the toolbar.
- Execute the next simulation step:** Points to the "Execute next simulation step" button in the toolbar.

At the bottom, a status bar indicates: "New script loaded: G:\shimon progs\Max\Max.tst".

Using test scripts

CPU Emulator (1.4b1) - G:\shimon progs\Max\Max.asm

File View Run Help

Animate: Program flow View: Output Format: Decimal

ROM Asm

0	@0
1	D=M
2	@1
3	D=D-M
4	@10
5	D; JGT
6	@1
7	D=M
8	@12
9	0; JMP
10	@0
11	D=M
12	@2
13	M=D
14	@14
15	0; JMP
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	

RAM

0	47
1	47
2	47
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0

View options:

- **Script:** Shows the current script;
- **Output:** Shows the generated output file;
- **Compare:** Shows the given comparison file;
- **Screen:** Shows the simulated screen.

When the script terminates, the comparison of the script output and the compare file is reported.

ALU

D Input : 47

M/A Input : 14

ALU output : 0

PC: A: 14

End of script - Comparison ended successfully

The default script (and a deeper understanding of the CPU emulator logic)

The screenshot shows the CPU Emulator (1.4b1) interface. The title bar indicates the file path: G:\shimon progs\Max\Max.asm. The menu bar includes File, View, Run, and Help. The toolbar contains several icons, with a red box highlighting the run/stop buttons: a single right arrow, a double right arrow, a square, and a double left arrow. Below the toolbar are controls for 'Animate' (Program flow), 'View' (Script), and 'Format' (Decimal). The main area is divided into three sections: ROM, RAM, and a script editor.

ROM: A table showing memory addresses and instructions. Address 3 is highlighted with the instruction `D=D-M`.

Address	Instruction
0	@0
1	D=M
2	@1
3	D=D-M
4	@10
5	D; JGT
6	@1
7	D=M
8	@12
9	0; JM
10	@0
11	
24	
25	
26	
27	
28	

RAM: A table showing memory addresses and values. Address 0 contains the value 81.

Address	Value
0	81
1	907
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
24	0
25	0
26	0
27	0
28	0

Script Editor: Shows a script with a loop: `repeat { ticktock; }`. A callout points to this script, stating: "If you load a program file without first loading a script file, the emulator loads a default script (always). The default script consists of a loop that runs the computer clock infinitely."

Registers: The D register is shown with the value 81. The PC (Program Counter) is 3, and the A (Accumulator) is 1.

ALU: A diagram showing the ALU operation. The D Input is 0, and the M/A Input is 81. The ALU output is 81.

Callout 1: "Note that these run/stop buttons don't control the program. They control the script, which controls the computer's clock, which causes the computer hardware to fetch and execute the program's instructions, one instruction per clock cycle."



Breakpoints: a powerful debugging tool

The CPU emulator continuously keeps track of:

- **A**: value of the A register
- **D**: value of the D register
- **PC**: value of the Program Counter
- **RAM[i]**: value of any RAM location
- **time**: number of elapsed machine cycles

Breakpoints:

- A breakpoint is a pair <variable, value> where variable is one of {**A**, **D**, **PC**, **RAM[i]**, **time**} and **i** is between 0 and 32K.
- Breakpoints can be declared either interactively, or via script commands.
- For each declared breakpoint, when the variable reaches the value, the emulator pauses the program's execution with a proper message.

Breakpoints declaration

The screenshot shows the CPU Emulator interface with the following components:

- Toolbar:** Contains icons for file operations, execution (single step, step over, step into, run), and a red flag icon for breakpoints (circled in red).
- ROM Panel:** Displays assembly code. A yellow callout points to it with the text "1. Open the breakpoints panel".
- RAM Panel:** Displays memory contents. A yellow callout points to it with the text "2. Previously-declared breakpoints".
- Breakpoint Panel:** A floating window with a table of breakpoints. A red box highlights the table, and a yellow callout points to it with the text "2. Previously-declared breakpoints".
- Breakpoint Table:**

Variable Name	Value
A	2
RAM[20]	5
Time	12
- ALU Panel:** Shows the ALU output and inputs. A yellow callout points to it with the text "3. Add, delete, or update breakpoints".
- PC and A Registers:** Display the current program counter and accumulator values.

Yellow callouts provide instructions:

- 1. Open the breakpoints panel
- 2. Previously-declared breakpoints
- 3. Add, delete, or update breakpoints

Breakpoints declaration

The screenshot shows the CPU Emulator interface with the following components:

- ROM Asm:** A list of assembly instructions. Instruction 16, `D=D+A`, is highlighted.
- RAM:** A memory dump showing addresses 0 to 15.
- Breakpoint Panel:** A dialog box with a table of variables:

Variable Name	Value
A	2
RAM[20]	5
Time	12
- Breakpoint Variables:** A dialog box for setting a breakpoint. The 'Name' field is set to `RAM[21]` and the 'Value' field is set to `200`. A green checkmark button is circled in red.
- PC:** A field showing the current Program Counter value, 0.
- ALU:** A diagram showing the ALU output, with inputs for D and M/A.

Two callouts provide instructions:

1. Select the system variable on which you want to break
2. Enter the value at which the break should occur

Breakpoints usage

The screenshot shows the CPU Emulator interface with the following components:

- ROM Asm:** A list of assembly instructions. A yellow callout bubble points to the 'Run' button (a double right arrow) with the text "2. Run the program".
- RAM:** A list of memory addresses and values. A yellow callout bubble points to the 'Breakpoint Panel' window with the text "1. New breakpoint".
- Breakpoint Panel:** A window showing a table of variables and their values.

Variable Name	Value
A	2
RAM[20]	5
Time	12
RAM[21]	200

A yellow callout bubble points to the table with the text "3. When the A register will be 2, or RAM[20] will be 5, or 12 time units (cycles) will elapse, or RAM[21] will be 200, the emulator will pause the program's execution with an appropriate message."

A powerful debugging tool!

Postscript: Maurice Wilkes (computer pioneer) discovers debugging:

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

(Maurice Wilkes, 1949).

