



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

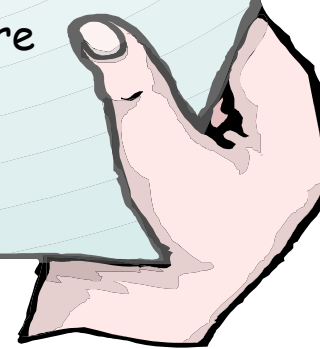
Background

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.

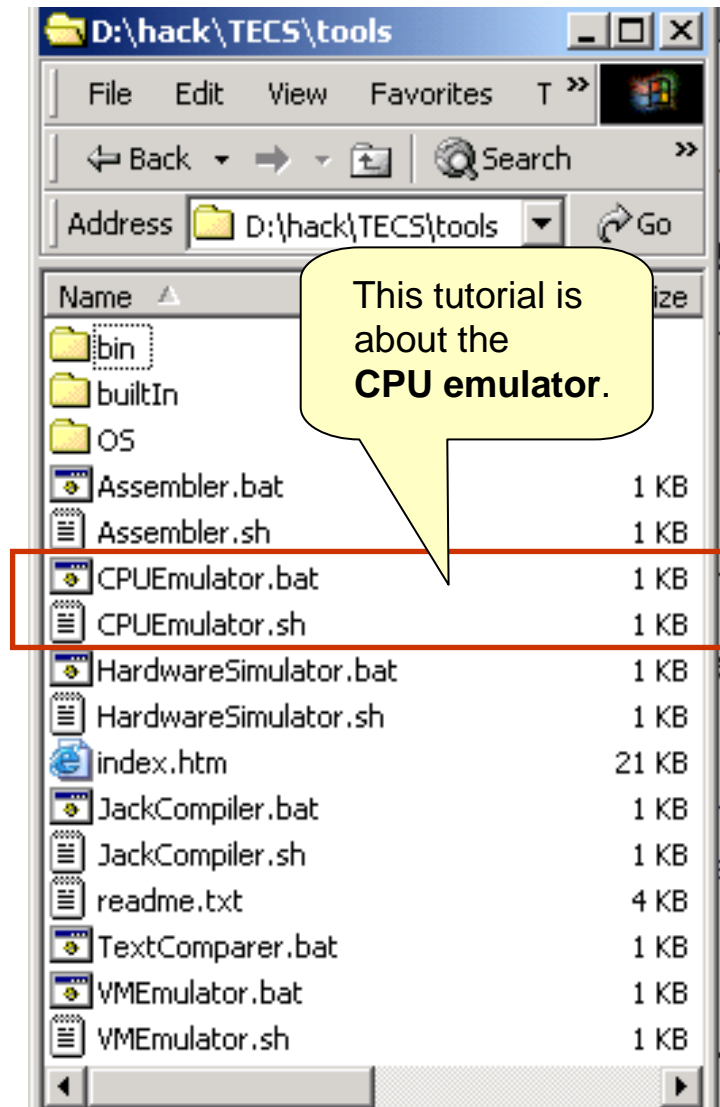
In the first part of the book/course, we build the hardware platform of a simple yet powerful computer, called Hack. In the second part, we build the computer's software hierarchy, consisting of an assembler, a virtual machine, a simple Java-like language called Jack, a compiler for it, and a mini operating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



The Hack Computer Platform (simulated)

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

- ROM Asm:** A list of memory addresses and instructions. Address 15 is highlighted in yellow.
- RAM:** A list of memory addresses. Address 17 is highlighted in yellow.
- PC:** Program Counter, value 15.
- A:** Accumulator, value 17.
- Hand-drawn Note:** A light blue note with a hand holding it, containing travel advice.

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

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

- ROM Asm:** A list of assembly instructions. Instruction 15, `@32`, is highlighted in yellow.
- RAM:** A memory dump showing values. Address 17 contains the value 17184.
- Registers:** The PC register is 15 and the A register is 17. Callouts point to these registers and the RAM value at address 17.
- ALU:** Shows D Input: 25, M/A Input: 17184, and ALU output: 17184. A callout points to the ALU output.
- Keyboard Enabler:** A callout points to the keyboard icon at the bottom of the screen.
- Screen:** A callout points to the main display area.
- Data Memory:** A callout points to the RAM area.
- Instruction Memory:** A callout points to the ROM area.

Instruction memory

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

- ROM Asm:** A list of instructions. The instruction at address 15, `@32`, is highlighted in yellow. A red circle highlights the dropdown menu above it.
- RAM:** A list of memory addresses and values. Address 17 contains the value 17184.
- PC (Program Counter):** Located at the bottom left, showing the value 15. A red arrow points to it.
- ALU:** Located at the bottom right, showing the D Input as 25 and the ALU output as 17184.

Callouts provide additional information:

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

Data memory (RAM)

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

File View Run Help

Slow Fast Animate: Program flow View: Screen Format: Decimal

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
9	0
10	0
11	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

PC 15 A 17

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

The screenshot shows a CPU Emulator window titled "CPU Emulator (1.4b3) - G:\examples\Rect.asm". The interface includes a menu bar (File, View, Run, Help), a toolbar with navigation and execution icons, and a control panel with "Slow", "Fast", and "Animate: Program" options. The main area is divided into three panes: ROM, RAM, and ALU.

ROM Pane: A list of assembly instructions. Instruction 15, "@32", is highlighted in yellow. The PC register is shown at the bottom with the value 15.

RAM Pane: A table of memory addresses and values. Address 17 is highlighted in yellow and contains the value 17184. An annotation "M (=RAM[A])" points to this address.

ALU Pane: A diagram of the ALU. The "D Input" is 25, and the "M/A Input" is 17184. The ALU output is 17184. A register "D" is shown with the value 17184, and an annotation "D" points to it.

Annotations: A yellow callout box titled "Registers (all 16-bit):" lists:

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

Orange callout boxes label the "M (=RAM[A])" and "D" registers.

Arithmetic/Logic Unit

Current instruction

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	

M (=RAM[A])

RAM		
0		50
1		0
2		0
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		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

D

D: 17184

A

A: 17

ALU

D Input: 25

M/A Input: 17184

ALU output: 17184

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.

CPU Emulator Tutorial



I/O devices: screen and keyboard

The screenshot shows the CPU Emulator (1.4b3) interface. On the left, there are two memory maps: ROM and RAM. The ROM map shows addresses 0 to 28, and the RAM map shows addresses 0 to 17. The main window displays a simulated screen, a simulated keyboard, and an ALU diagram. The ALU diagram shows D Input and M/A Input both set to 0, resulting in an ALU output of 0. A status bar at the bottom 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.

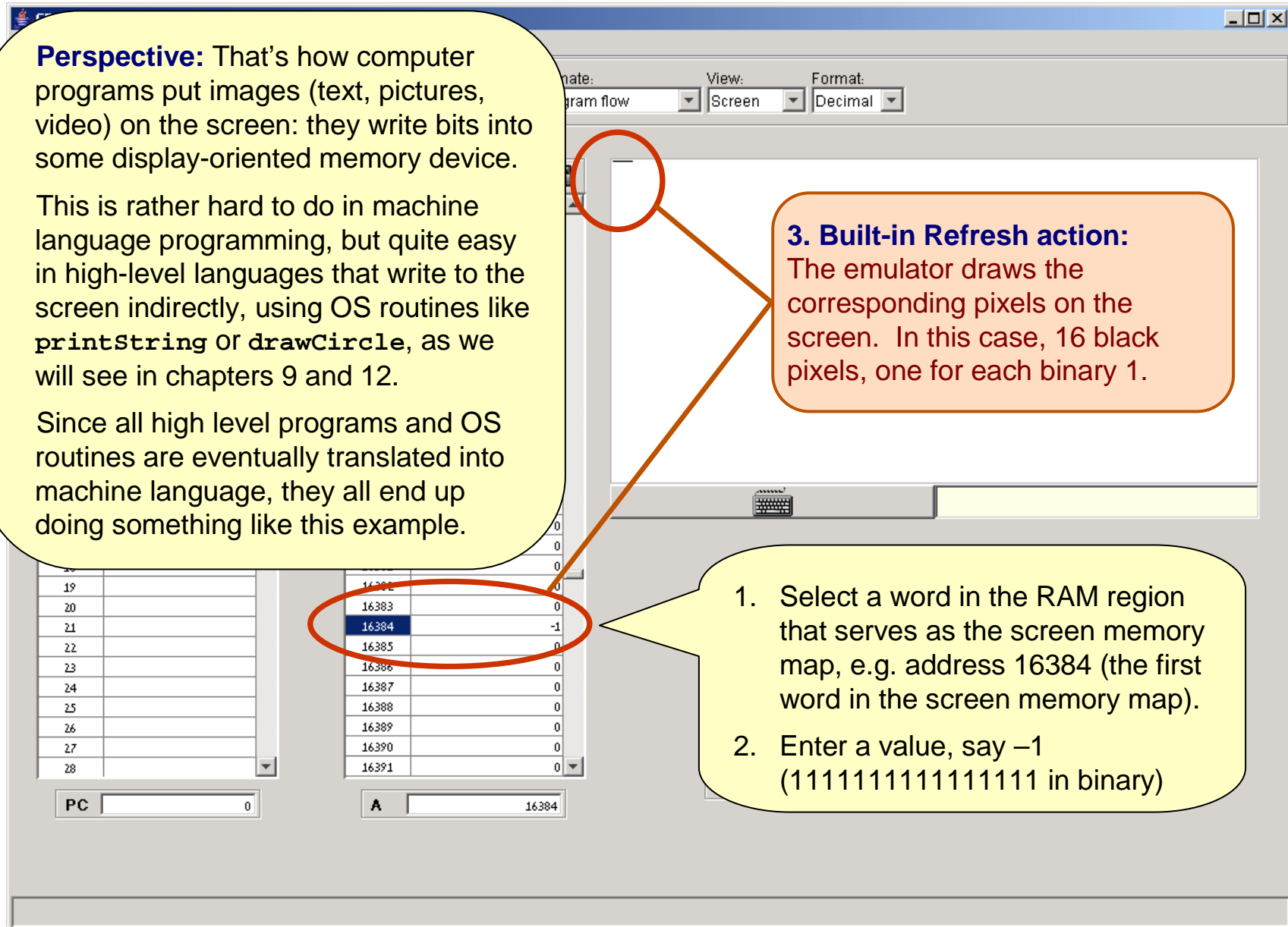
This is rather hard to do in machine language programming, but quite easy in high-level languages that write to the screen indirectly, using OS routines like `printString` or `drawCircle`, as we will see in chapters 9 and 12.

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, there are two memory maps: ROM (Asm) and RAM. The RAM map shows addresses from 24548 to 24576, with the value 0 in each cell. The ALU output is shown as 0. A yellow speech bubble contains the instructions: 1. Click the keyboard enabler, 2. Press some key on the real keyboard, say "S". A yellow speech bubble points to the RAM address 24576, with the text "3. Watch here:". An orange callout box points to the RAM address 24576, with the text "Keyboard memory map (a single 16-bit memory location)".

ROM Asm

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	

RAM

24548	0
24549	0
24550	0
24551	0
24552	0
24553	0
24554	0
24555	0
24556	0
24557	0
24558	0
24559	0
24560	0
24561	0
24562	0
24563	0
24564	0
24565	0
24566	0
24567	0
24568	0
24569	0
24570	0
24571	0
24572	0
24573	0
24574	0
24575	0
24576	0

ALU

D Input : 0

M/A Input : 0

ALU output : 0

3. Watch here:

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

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

The screenshot shows the CPU Emulator (1.4b3) interface. At the top, there are menu options (File, View, Run, Help) and settings for 'Program flow', 'View' (Screen), and 'Format' (Decimal). The main display area is divided into several sections. On the left, there is a 'Keyboard memory map' table with two columns of memory addresses and values. The value '83' at address 24576 is circled in red. Below this table is a register 'A' with a value of 0. In the center, there is a 'Visual echo' area showing the character 'S' on a keyboard graphic, which is also circled in red. Below the keyboard graphic is a register 'D' with a value of 0. At the bottom, there is a status bar that says 'Script restarted'.

20		24568	0
21		24569	0
22		24570	0
23		24571	0
24		24572	0
		24573	0
		24574	0
		24575	0
		24576	83



Loading a program

The screenshot shows the CPU Emulator (1.4b1) interface. The 'Load ROM' dialog box is open, showing a file list with 'Rect.hack' selected. The 'Load ROM' button is circled in red. A yellow callout bubble points to the dialog with the text: 'Navigate to a directory and select a .hack or .asm file.'

ROM	Asm
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	

RAM	
0	0
1	0
2	0
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
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

PC: 0

Loading a program

The screenshot shows a 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 navigation and execution controls, and a control panel with "Animate:" (Program flow), "View:" (None), and "Format:" (Decimal) dropdowns. The main area is divided into several sections:

- ROM:** A table showing memory addresses 0 to 28 with their corresponding binary values. The first row (address 0) is highlighted in yellow.
- RAM:** A table showing memory addresses 0 to 28, all currently containing the value 0.
- PC (Program Counter):** A register showing the value 0.
- A (Accumulator):** A register showing the value 0.
- Keyboard:** A small keyboard icon and a yellow highlighted area representing the keyboard buffer.
- D (Data Register):** A register showing the value 0.
- ALU:** A diagram of an ALU with two inputs: "D Input" and "M/A Input", both showing 0. The output is "ALU output" showing 0.

A yellow callout bubble points to the ROM table with the text: "Can switch from binary to symbolic representation".

Running a program

The screenshot shows a 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 navigation and execution buttons, and two data tables: ROM (Asm) and RAM. The ROM table lists assembly instructions, and the RAM table shows memory addresses and values. A speed control slider is set to "Fast". A large callout box at the bottom right provides a program description and a note.

2. Click the "run" button.

1. Enter a number, say 50.

3. To speed up execution, use the speed control slider

4. Watch here

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 a CPU Emulator window titled "CPU Emulator - D:\hack\Programs\rect.bin". The interface includes a menu bar (File, View, Run, Help), a toolbar with navigation buttons (back, forward, stop, home, end), a speed control slider (Slow to Fast), and dropdown menus for Animate (Program flow), View (None), and Format (Decimal). On the left, the ROM memory view shows assembly code, with line 12 highlighted in yellow. On the right, the RAM memory view shows values, with address 17 highlighted in yellow. A speed control slider is positioned between the memory views and the main display. The main display area is mostly blank, with a small black rectangle at the top left. Callout boxes provide instructions: "2. Click the 'run' button." points to the right arrow in the toolbar; "1. Enter a number, say 50." points to the RAM view; "3. To speed up execution, use the speed control slider" points to the slider; "4. Watch here" points to the main display area. A large callout box at the bottom right contains the program's description and a note.

2. Click the "run" button.

1. Enter a number, say 50.

3. To speed up execution, use the speed control slider

4. Watch here

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.

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 with the following components:

- Assembly Code:** A list of instructions from address 7 to 28. Instruction 12, $M=-1$, is highlighted in yellow. Instruction 17, $@17$, is also highlighted in yellow, with the value 17536 shown in the right column.
- Registers:** The PC register is 12, and the A register is 17536.
- Screen Display:** A window showing a screen with a single black pixel at the top left corner. A keyboard icon is visible at the bottom of the screen.
- Annotations:** Three callout boxes provide context. One points to instruction 12, another to instruction 17, and a third to the single black pixel on the screen.

Animation options

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

- Toolbar:** Includes icons for file operations, execution (single step, multi-step, stop, back, forward), and a speed slider from Slow to Fast.
- Animation Controls:** A dropdown menu set to "Program & data flow", a "View" dropdown set to "None", and a "Format" dropdown set to "Decimal".
- ROM Asm:** A list of instructions with addresses 0 to 26. Instruction 13 is highlighted in yellow.
- RAM:** A memory dump showing addresses 17522 to 17536. Address 17536 is highlighted in yellow.
- Annotations:**
 - A yellow callout points to the speed slider: "Controls execution (and animation) speed."
 - An orange callout points to the ROM and RAM windows: "The simulator can animate both program flow and data flow"
 - A large yellow callout points to the animation dropdowns: "Animation control:"

ROM	Asm	RAM
0	@0	17522 0
1	D=M	17523 0
2	@23	0 0
3	D; JLE	0 0
4	@16	0 0
5	M=D	0 0
6	@16384	0 0
7	D=A	0 0
8	@17	17530 0
9	M=D	17531 0
10	@17	17532 0
11	A=M	17533 0
12	M=-1	17534 0
13	@17	17535 0
14	M	17536 -1
15	@12	17537 0
16	D=M+A	17538 0
17	@17	17539 0
18	M=D	17540 0
19	@16	17541 0
20	MD=M-1	17542 0
21	@10	17543 0
22	D; JGT	17544 0
23	@23	17545 0
24	0; JMP	17546 0
25		17547 0
26		17548 0
		17549 0
		17550 0

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 ROM window displays assembly code, with lines 0 through 15 highlighted in yellow and enclosed in a red box. A yellow callout bubble labeled "tested program" points to this box. The RAM window shows memory addresses 0 through 28, with address 0 containing the value 47. The script window, also highlighted in yellow and enclosed in a red box, 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;
```

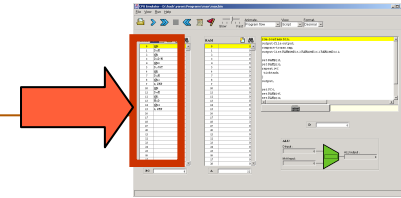
A yellow callout bubble labeled "test script" points to the script window. Below the script window is a keyboard icon and a register window showing D = 0. At the bottom right, there is an ALU diagram with D Input and M/A Input both set to 0, and an ALU output of 0.

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.

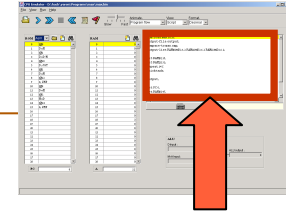
```
// Computes M[2]=max(M[0],M[1]) where M stands for RAM
@0
D=M           // D = M[0]
@1
D=D-M        // D = D - M[1]
@FIRST_IS_GREATER
D;JGT        // If D>0 goto FIRST_IS_GREATER
@1
D=M           // D = M[1]
@SECOND_IS_GREATER
0;JMP        // Goto SECOND_IS_GREATER
(FIRST_IS_GREATER)
@0
D=M           // 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).
```



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`).

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.
```

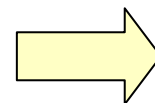
The scripting language has commands for:

- 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

The screenshot shows the CPU Emulator (1.4b1) interface. The 'Load Script' dialog box is open, displaying the 'Max' folder and a file named 'Max.tst'. The 'Load Script' button is circled in red. A yellow callout box contains the following text:

Interactive loading of the tested program itself (.asm or .hack file) is typically not necessary, since test scripts typically begin with a "load program" command.

The emulator interface includes a menu bar (File, View, Run, Help), a toolbar with navigation buttons, and two memory windows (ROM and RAM) showing hexadecimal and decimal values. The 'Load Script' dialog box has a 'Look in:' field set to 'Max', a file list containing 'Max.tst', and buttons for 'Load Script' and 'Cancel'.

Using test scripts

Speed control

Load a script

Script = a series of simulation steps, each ending with a semicolon;

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 the next simulation step

Execute step repeatedly

```
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;  
  
// test 1: max(15,32)  
set RAM[0]15,  
set RAM[1]32,  
repeat 14 {  
    ticktock;  
}
```

ROM

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

RAM

Address	Value
0	
1	
2	
3	
4	
5	0
6	0
7	0
8	0
9	0
10	0
11	

ALU output: 0

max input: 0

A: 0

File View Run Help

Animate: Program flow View: Script Format: Decimal

Slow Fast

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

Using test scripts

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.

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

RAM[0] | RAM[1] | RAM[2]

15	32	32
47	22	47
47	47	47

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 interface with the following components:

- ROM Asm:** A table of instructions. Row 3 is highlighted with the instruction `D=D-M`.
- RAM:** A table of memory addresses and values. Address 0 contains the value 81.
- Script Editor:** A code editor showing a `repeat { ticktock; }` loop, which is highlighted in yellow.
- ALU:** A diagram of an ALU with a green trapezoidal shape labeled 'M'. It shows 'D Input' as 0 and 'M/A Input' as 81, resulting in an 'ALU output' of 81.
- PC and A:** Registers showing PC=3 and A=1.
- Control Panel:** Includes a red-bordered toolbar with run/stop buttons, a speed slider (Slow/Fast), and dropdown menus for Animate (Program flow), View (Script), and Format (Decimal).

Callout 1 (left): 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.

Callout 2 (right): 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.



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 three callouts explaining breakpoint declaration steps:

- 1. Open the breakpoints panel:** A red circle highlights the breakpoint icon in the toolbar.
- 2. Previously-declared breakpoints:** A yellow callout points to the 'Breakpoint Panel' window, which contains a table of declared breakpoints.
- 3. Add, delete, or update breakpoints:** A red circle highlights the '+' button in the breakpoint panel's toolbar.

The 'Breakpoint Panel' window contains the following table:

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

The emulator interface also shows ROM and RAM memory windows, a keyboard input field, and an ALU diagram.

Breakpoints declaration

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

- ROM Asm:** A list of assembly instructions from address 0 to 28. Address 0 is highlighted in yellow.
- RAM:** A memory dump showing addresses 0 to 28. Address 0 is highlighted in yellow.
- 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 contains 'RAM[21]' and the 'Value' field contains '200'. A green checkmark button is circled in red.
- ALU:** A diagram of an ALU with 'D Input' and 'M/A Input' both set to 0, and 'ALU output' set to 0.
- PC:** A field showing the current Program Counter value as 0.

Two callout boxes 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 from address 0 to 28. Instruction 11 is `A=M`.
- RAM:** A memory dump showing addresses 0 to 28. Address 20 contains the value 5, and address 21 contains the value 200.
- Breakpoint Panel:** A window titled "Breakpoint Panel" containing a table of variables and their values.

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

Callouts in the image provide the following instructions:

1. New breakpoint
2. Run the program
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).

