



CPU Emulator Tutorial

This program is part of the software suite
that accompanies the book

The Elements of Computing Systems

by Noam Nisan and Shimon Schocken

MIT Press

www.idc.ac.il/tecs

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

Chief Software Architect: Yaron Ukrainitz



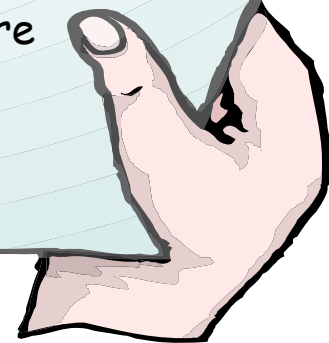
Background

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`):

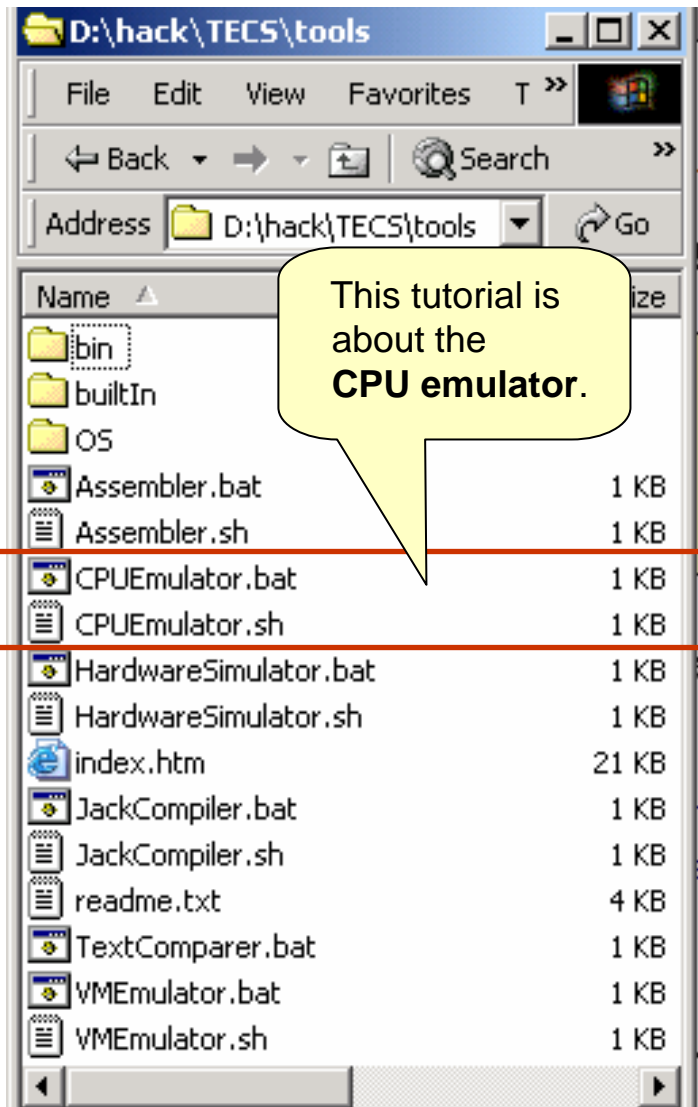
- Used to build hardware platforms and execute programs;
- Supplied by us.

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

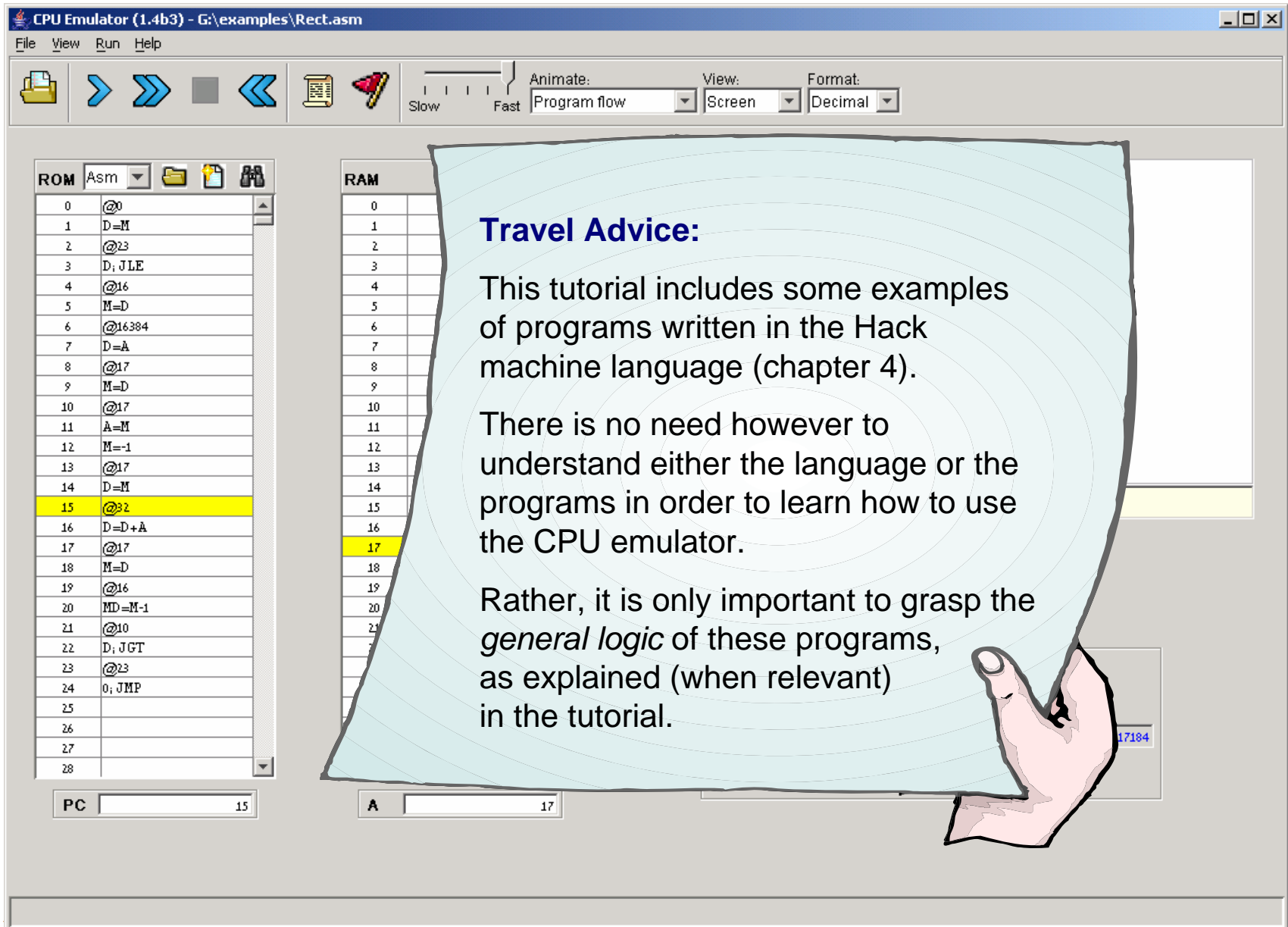
- I. [Basic Platform](#)
- II. [I/O devices](#)
- III. [Interactive simulation](#)
- IV. [Script-based simulation](#)
- V. [Debugging](#)

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

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



The Hack Computer Platform (simulated)



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
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

17184

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 displays the CPU Emulator interface with the following components and callouts:

- ROM Asm:** A list of instructions from address 0 to 28. Instruction 15, `@32`, is highlighted in yellow. A callout labeled "instruction memory" points to this instruction.
- RAM:** A list of memory addresses from 0 to 28. Address 15 contains the value 17184. A callout labeled "data memory" points to this value.
- Registers:** The PC register is at 15 and the A register is at 17. A callout labeled "registers" points to these two registers.
- ALU:** A detailed view of the ALU shows the D Input as 25 and the M/A Input as 17184. The ALU output is 17184. A callout labeled "ALU" points to this component.
- Keyboard Enabler:** A callout labeled "keyboard enabler" points to a keyboard icon on the screen.
- Screen:** A callout labeled "screen" points to the main display area.

Address	Instruction	Value
0	@0	50
1	D=M	0
2	@23	0
3	D; JLE	0
4	@16	0
5	M=D	0
6	@16384	0
7	D=A	0
8	@17	0
9	M=D	0
10	@17	0
11	A=M	0
12	M=-1	0
13	@17	0
14	D=M	0
15	@32	17184
16	D=D+A	25
17	@17	0
18	M=D	0
19	@16	0
20	MD=M-1	0
21	@10	0
22	D; JGT	0
23	@23	0
24	0; JMP	0
25		0
26		0
27		0
28		0

Instruction memory

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

- ROM Asm:** A list of instructions. Callout: "The loaded code can be viewed either in binary, or in symbolic notation (present view)".
- RAM:** A list of memory addresses and values. Callout: "Instruction memory (32K): Holds a machine language program".
- Next instruction:** Instruction 15, "@32", is highlighted in yellow. Callout: "Next instruction is highlighted".
- Program Counter (PC):** A register showing the value 15. Callout: "Program counter (PC) (16-bit): Selects the next instruction." An arrow points to the PC register.
- ALU:** A component showing the ALU output as 17184. Callout: "ALU output : 17184".

ROM Asm	RAM
0	0
1 D=M	1
2 @23	2
3 D;JLE	3
4 @16	4
5 M=D	5
6 @16384	6 0
7 D=A	7 0
8 @17	8
9 M=D	9
10 @17	10
11 A=M	11
12 M=-1	12
13 @17	13
14 D=M	14
15 @32	15 0
16 D=D+A	16 25
17 @17	17 17184
18 M=D	18 0
19 @16	
20 MD=M-1	
21 @10	
22 D;JGT	
23 @23	
24 0;JMP	
25	
26	
27	
28	

Data memory (RAM)

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

File View Run Help

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 the CPU Emulator interface with the following components and values:

- ROM:** A list of instructions. Instruction 15 is highlighted: `@32`.
- RAM:** A list of memory addresses. Address 17 is highlighted and contains the value 17184.
- PC (Program Counter):** 15
- A (Address Register):** 17
- D (Data Register):** 17184
- ALU:** Shows the current operation. D Input is 25, M/A Input is 17184, and the ALU output is 17184.

Annotations in the image:

- 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
- An orange box labeled "M (=RAM[A])" points to the value 17184 in RAM address 17.
- An orange box labeled "D" points to the value 17184 in the Data Register.
- An orange box labeled "A" points to the value 17 in the Address Register.

Arithmetic/Logic Unit

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 icons, and a control panel with "Slow" and "Fast" buttons and an "Animate: Program flow" dropdown. The main area is divided into three sections: ROM, RAM, and ALU.

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

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

ALU: A diagram of the Arithmetic Logic Unit. It shows a green trapezoidal block labeled "M" (Multi-plier/Divider). The "D Input" is 25 and the "M/A Input" is 17184. The "ALU output" is 17184. A box labeled "D" with the value 17184 is also shown.

Registers: At the bottom, the "PC" register is 15 and the "A" register is 17.

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

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.

Script restarted

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.

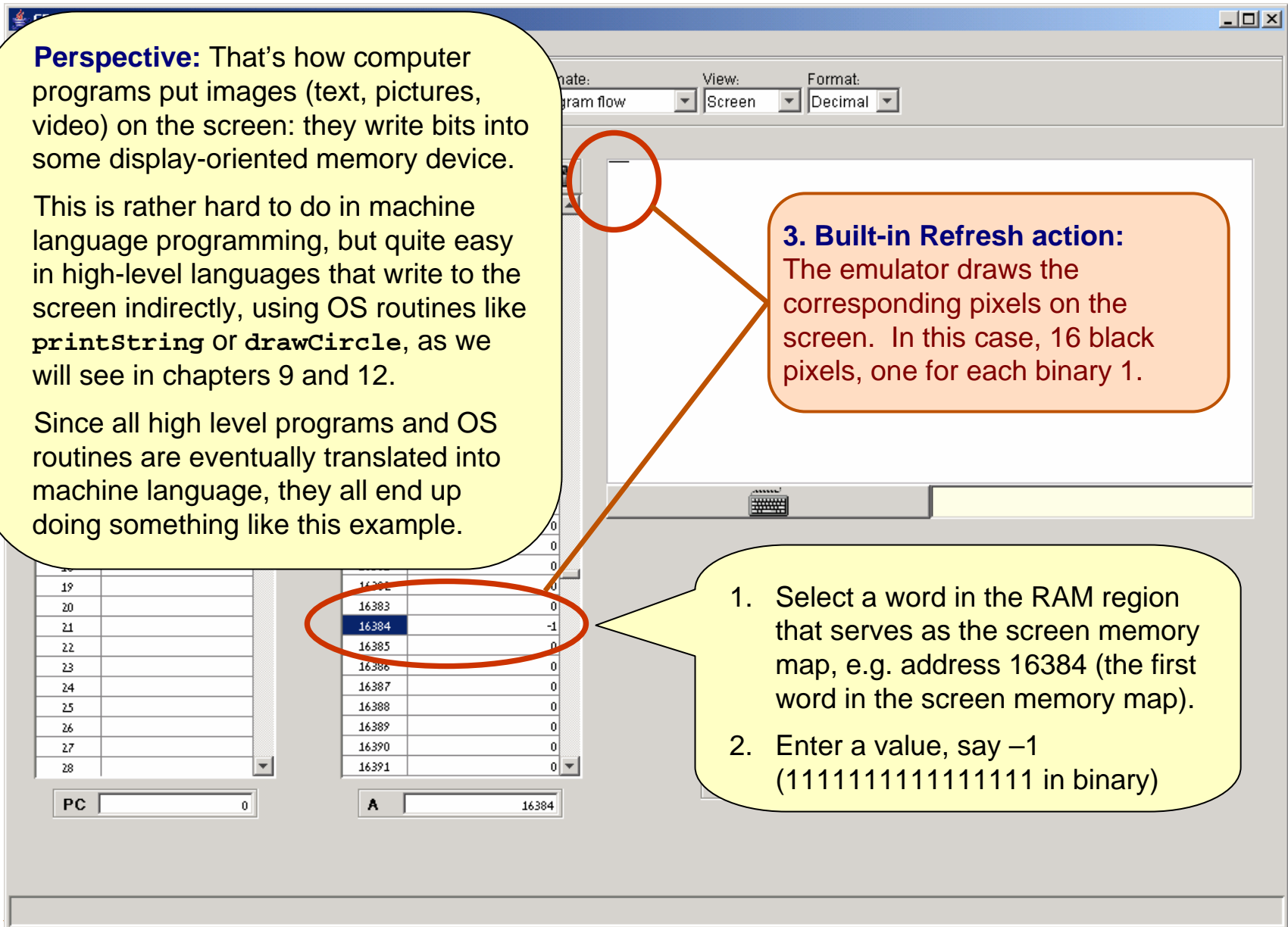
This is rather hard to do in machine language programming, but quite easy in high-level languages that write to the screen indirectly, using OS routines like `printString` or `drawCircle`, as we will see in chapters 9 and 12.

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. A callout box points to address 24576, stating it is the keyboard memory map. The center features a keyboard icon and a 'D' register showing 0. The bottom right shows an ALU diagram with D Input and M/A Input both at 0, and an ALU output of 0. A status bar at the bottom indicates 'Script restarted'.

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

3. Watch here:

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

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)

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

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

The emulator displays its character code in the keyboard memory map

Script restarted



Loading a program

The screenshot shows the CPU Emulator (1.4b1) interface. The 'File' menu is open, and the 'Load ROM' dialog box is displayed. The dialog box shows the current directory as '05' and lists three files: 'Add.hack', 'Max.hack', and 'Rect.hack'. The 'Rect.hack' file is selected. The 'File name:' field contains 'Rect.hack'. The 'Load ROM' button is circled in red. A yellow callout bubble points to the dialog box 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 three dropdown menus: "Animate:" (Program flow), "View:" (None), and "Format:" (Decimal). The main area is divided into several sections:

- ROM:** A table with 29 rows (0-28) and 2 columns. The first row (0) is highlighted in yellow and contains the value "0000000000000000". A callout bubble points to this row with the text "Can switch from binary to symbolic representation".
- RAM:** A table with 29 rows (0-28) and 2 columns. The first row (0) is highlighted in yellow and contains the value "0".
- Keyboard:** A small keyboard icon is visible below the RAM table.
- D Register:** A register labeled "D" with a value of "0".
- ALU:** A diagram of an ALU with two inputs: "D Input" and "M/A Input", both with a value of "0". The output is "ALU output" with a value of "0".
- PC and A Registers:** At the bottom, there are two registers: "PC" and "A", both with a value of "0".

Running a program

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

- ROM Asm:** A table of assembly instructions. Row 12 is highlighted in yellow.
- RAM:** A table of memory addresses and values. Row 17 is highlighted in yellow.
- PC:** A register showing the value 12.
- A:** A register showing the value 17536.
- Speed Control:** A slider between 'Slow' and 'Fast'.
- Buttons:** A 'run' button (a blue double arrow) and other navigation buttons.
- Callouts:** Four callout boxes with numbers 1-4 explaining the steps to run the program.
- Program Description:** A large yellow callout box at the bottom right describing the program's function.
- Note:** A smaller yellow callout box at the bottom right providing a note about understanding the code.

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.

Animation options

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

- Toolbar:** Contains icons for file operations, navigation (back, forward, stop), and a speed slider between 'Slow' and 'Fast'. The 'Animate' dropdown is set to 'Program & data flow', 'View' is 'None', and 'Format' is 'Decimal'.
- ROM Asm Table:**

Address	Instruction
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	M
15	@2
16	D=M+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	
- RAM Table:**

Address	Value
17522	0
17523	0
	0
	0
	0
	0
	0
17530	0
17531	0
17532	0
17533	0
17534	0
17535	0
17536	-1
17537	0
17538	0
17539	0
17540	0
17541	0
17542	0
17543	0
17544	0
17545	0
17546	0
17547	0
17548	0
17549	0
17550	0
- Callouts:**
 - A yellow callout points to the speed slider: "Controls execution (and animation) speed."
 - An orange callout points to the ROM table: "The simulator can animate both program flow and data flow"
 - A large yellow callout on the right contains the "Animation control" and "Usage tip" 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 interface with the following components:

- ROM Asm:** A table of memory addresses and assembly instructions. A red box highlights the first 16 rows, which are also highlighted in yellow in the original image. A yellow callout bubble labeled "tested program" points to this area.
- RAM:** A table of memory addresses and values. The value at address 0 is 47.
- Script Window:** A text area containing assembly code. A red box highlights the entire script, and a yellow callout bubble labeled "test script" points to it. The script content is:

```
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;

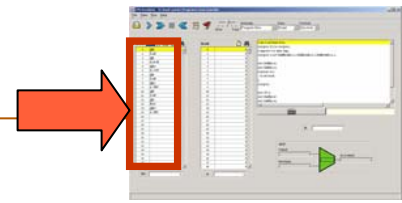
// test1: max(15,32)
set RAM[0]15,
set RAM[1]32;
repeat 14 {
    ticktock;
}
output;
```
- ALU Diagram:** A green trapezoidal ALU with two inputs labeled "D Input" and "M/A Input", both set to 0, and one output labeled "ALU output" set to 0.
- Registers:** "PC" and "A" registers are shown at the bottom, both with a value of 0.
- Toolbar:** Includes navigation buttons (back, forward, stop, home), speed controls (Slow, Fast), and dropdown menus for "Animate" (Program flow), "View" (Script), and "Format" (Decimal).

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.

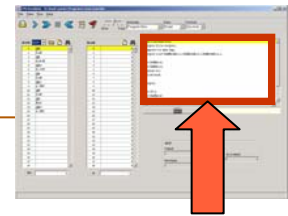
```
// Computes M[2]=max(M[0],M[1]) where M stands for RAM
@0
D=M          // D = M[0]
@1
D=D-M        // D = D - M[1]
@FIRST_IS_GREATER
D;JGT        // If D>0 goto FIRST_IS_GREATER
@1
D=M          // D = M[1]
@SECOND_IS_GREATER
0;JMP        // Goto SECOND_IS_GREATER
(FIRST_IS_GREATER)
@0
D=M       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

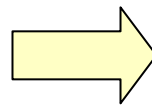
The scripting language has commands for:

- 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 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, navigation (back, forward, stop), and execution (single step, repeat step). A speed control slider is set between "Slow" and "Fast". The "Animate" dropdown is set to "Program flow", "View" is "Script", and "Format" is "Decimal".

On the left, the ROM and RAM memory windows are visible. The ROM window shows assembly code with addresses 0 to 26. The RAM window shows addresses 0 to 11. A callout "Load a script" points to the RAM window.

The main window displays a script with the following content:

```
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;  
  
// test1: max(15,32)  
set RAM[0]15,  
set RAM[1]32;  
repeat 14 {  
    ticktock;  
}
```

A callout "Script = a series of simulation steps, each ending with a semicolon;" points to the script content.

At the bottom, the ALU output and MUX input are shown, both with a value of 0. A callout "Execute step repeatedly" points to the repeat step icon in the toolbar. Another callout "Execute the next simulation step" points to the single step icon.

A large callout box contains the following text:

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.

At the bottom of the window, the status bar reads "New script loaded: G:\shimon progs\Max\Max.tst".

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 step repeatedly

Execute the next simulation step

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. Address 0 contains the value 81.
- Script:** A code editor showing a `repeat { ticktock; }` loop, with the `ticktock;` line highlighted in yellow.
- ALU:** A diagram of an ALU with a green triangle labeled 'M'. The D Input is 0 and the M/A Input is 81, resulting in an ALU output of 81.
- Registers:** PC is 3 and A is 1.
- Control Panel:** Includes run/stop buttons (highlighted in a red box), speed controls (Slow/Fast), and dropdown menus for Animate (Program flow), View (Script), and Format (Decimal).

Callout 1 (Yellow): 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 (Yellow): 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 the following components:

- Toolbar:** Contains icons for file operations, navigation (back, forward, stop), and execution (play, pause). A red circle highlights the breakpoint icon (a red flag).
- ROM Asm:** A list of assembly instructions with addresses from 0 to 28. A yellow callout bubble points to this area with the text "1. Open the breakpoints panel".
- RAM:** A list of memory addresses from 0 to 28. A yellow callout bubble points to this area with the text "2. Previously-declared breakpoints".
- Breakpoint Panel:** A window titled "Breakpoint Panel" with a table of declared breakpoints. A red box highlights the table content.
- ALU:** A diagram of an ALU with inputs for D, M/A, and output.
- PC:** A register labeled "PC" with a value of 0.
- Bottom Controls:** A red circle highlights a button with a plus sign in a dashed box, with a yellow callout bubble pointing to it containing the text "3. Add, delete, or update breakpoints".

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

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 15. Address 50 is highlighted in yellow.
- Breakpoint Panel:** A table listing system variables and their values:

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 the 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 yellow 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

At the bottom of the emulator window, a red circle highlights the '+' button in the breakpoint control area.

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 1 is highlighted in yellow.
- RAM:** A memory dump showing addresses 0 to 28. Address 0 contains the value 50.
- Breakpoint Panel:** A dialog box with a table of variables and their values.
- PC:** Program Counter register showing value 0.
- A:** Accumulator register showing value 2.

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

2. Run the program

1. New breakpoint

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).

