

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

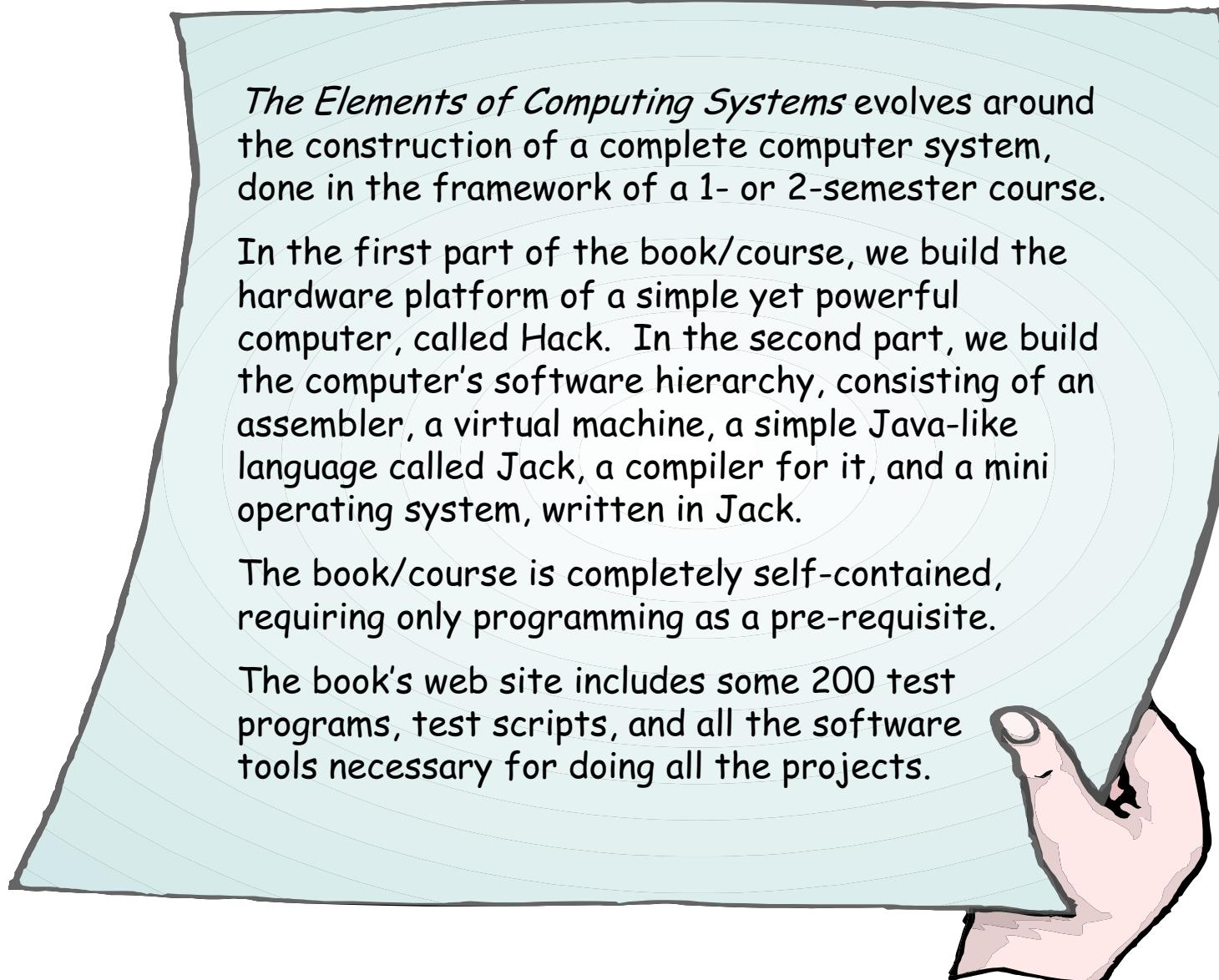
Background

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.

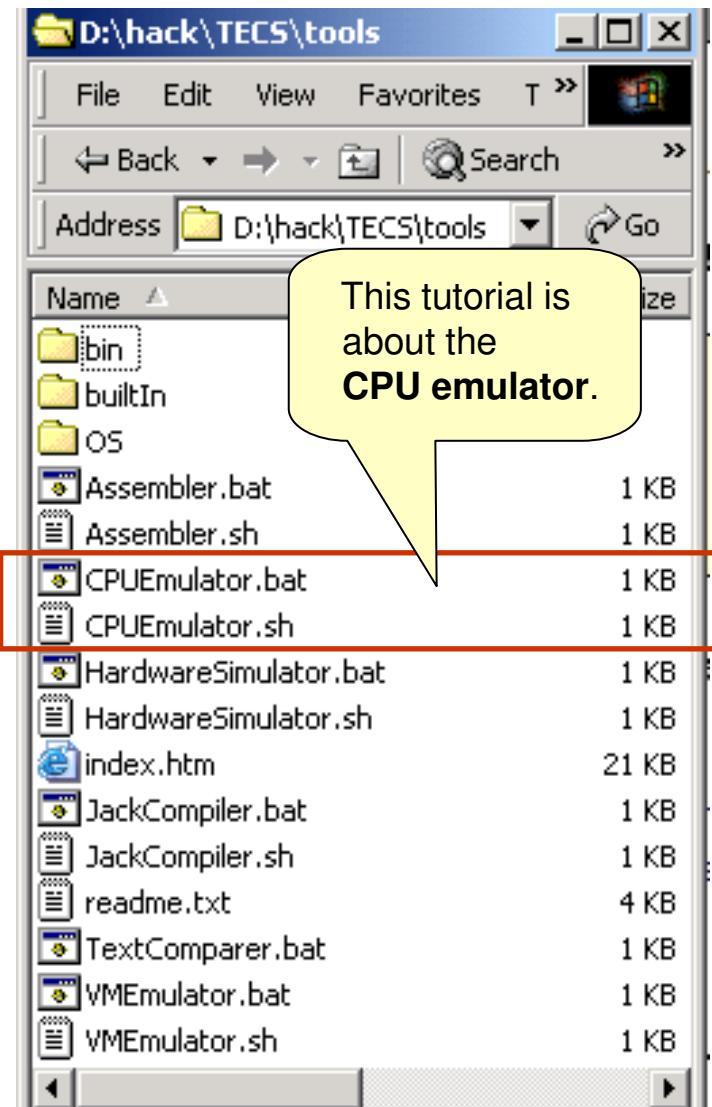
In the first part of the book/course, we build the hardware platform of a simple yet powerful computer, called Hack. In the second part, we build the computer's software hierarchy, consisting of an assembler, a virtual machine, a simple Java-like language called Jack, a compiler for it, and a mini operating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

- 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.

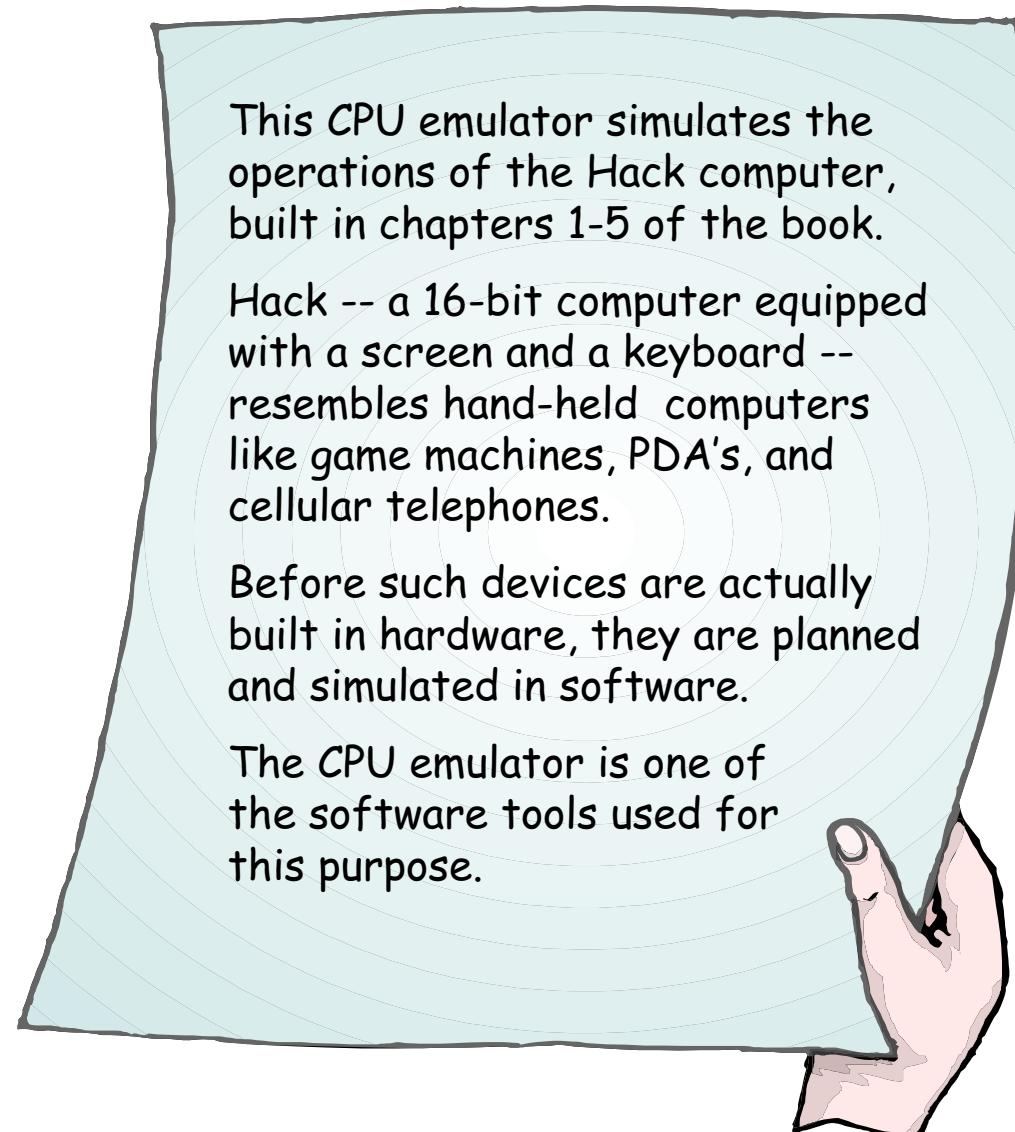
Other

- 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



CPU Emulator Tutorial

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

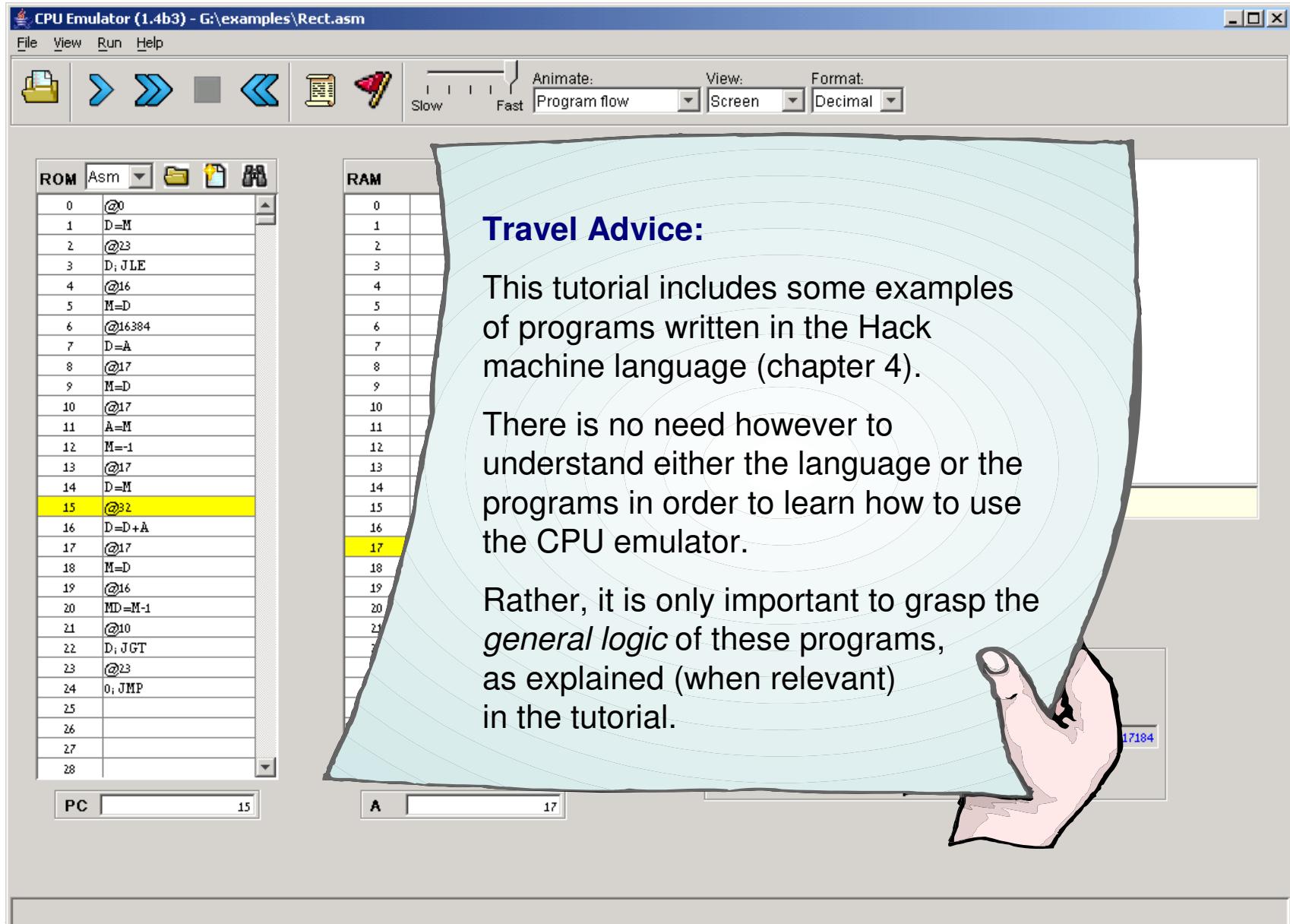
Relevant reading (from “*The Elements of Computing Systems*”):

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

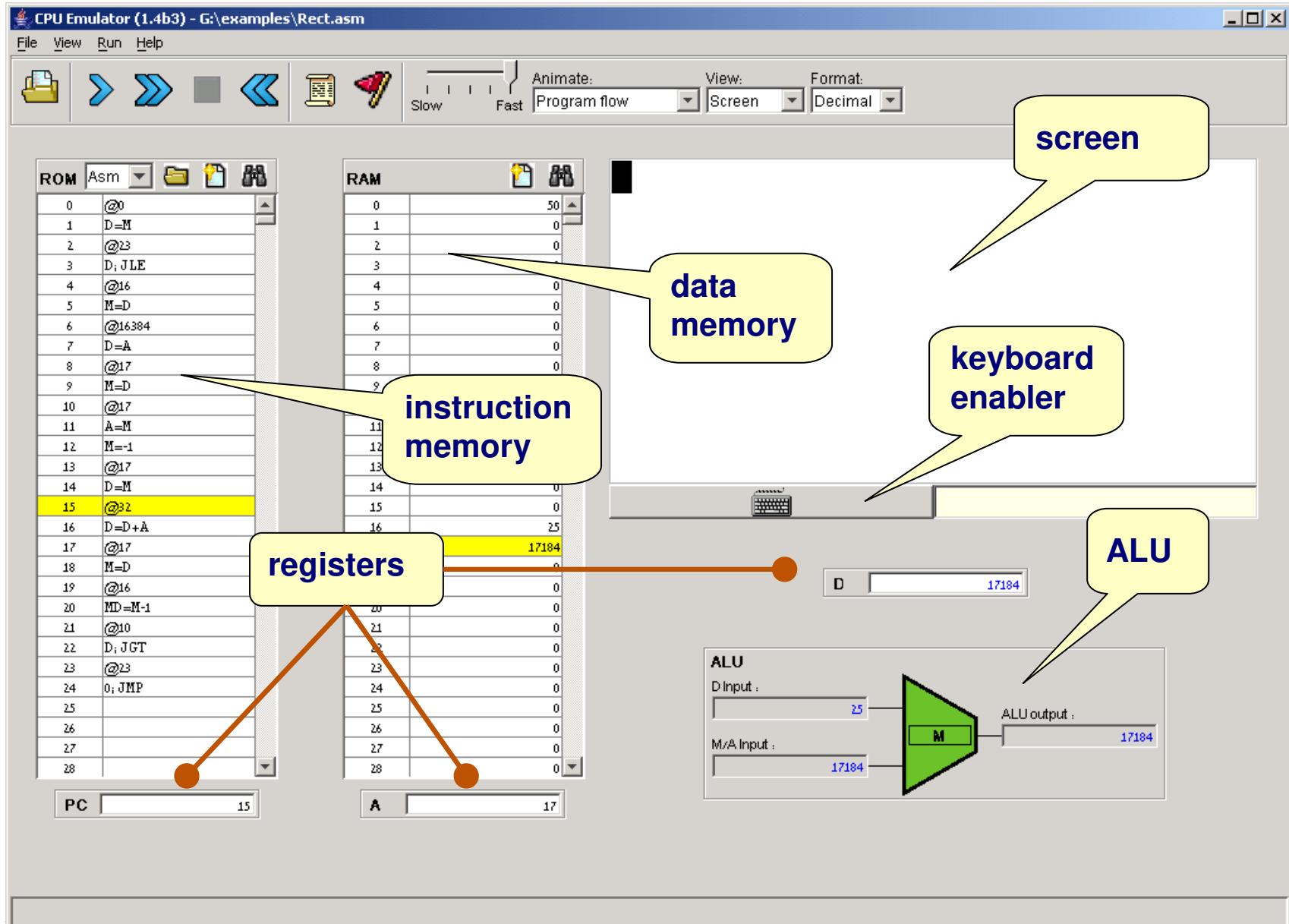
CPU Emulator Tutorial



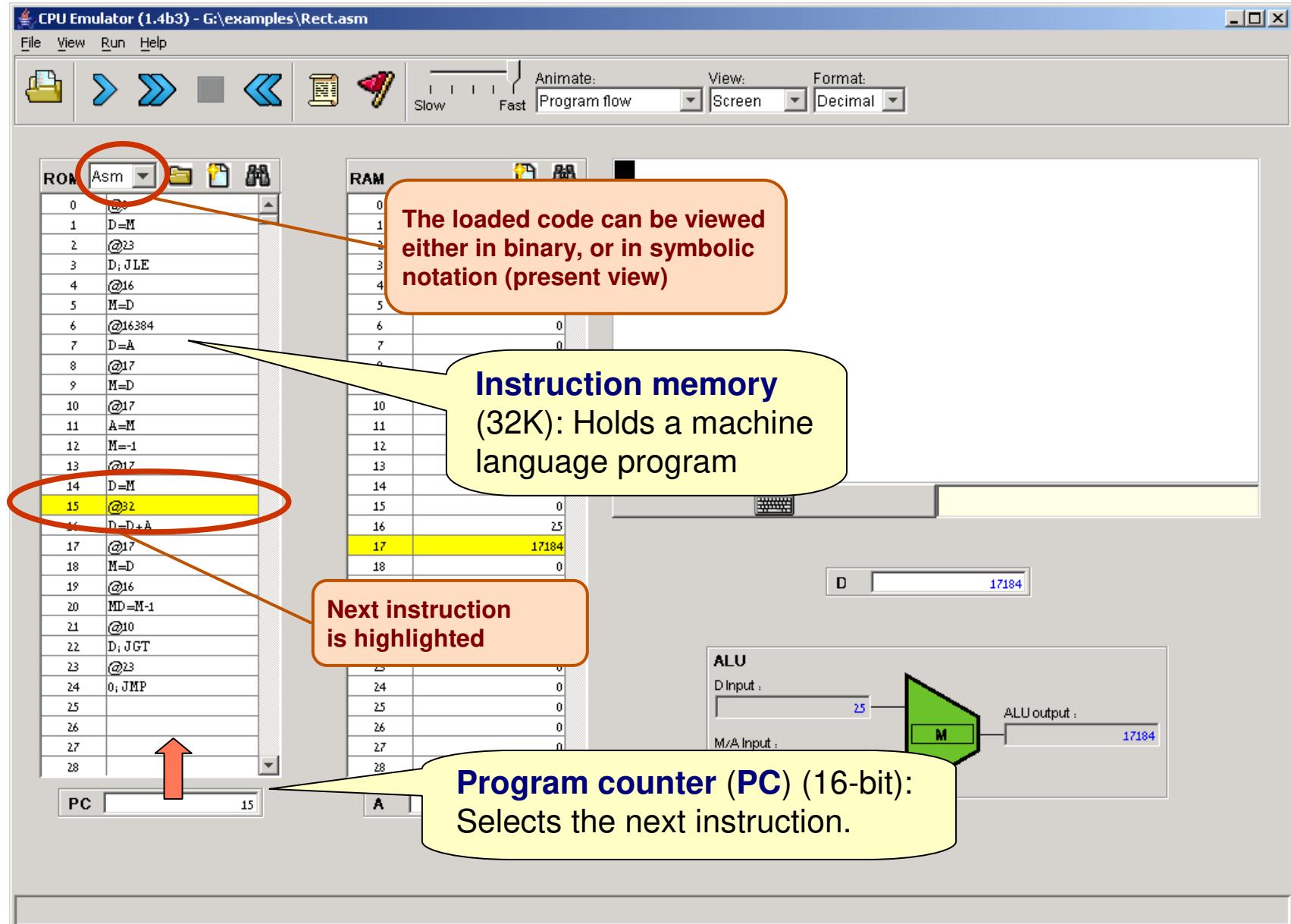
The Hack Computer Platform (simulated)



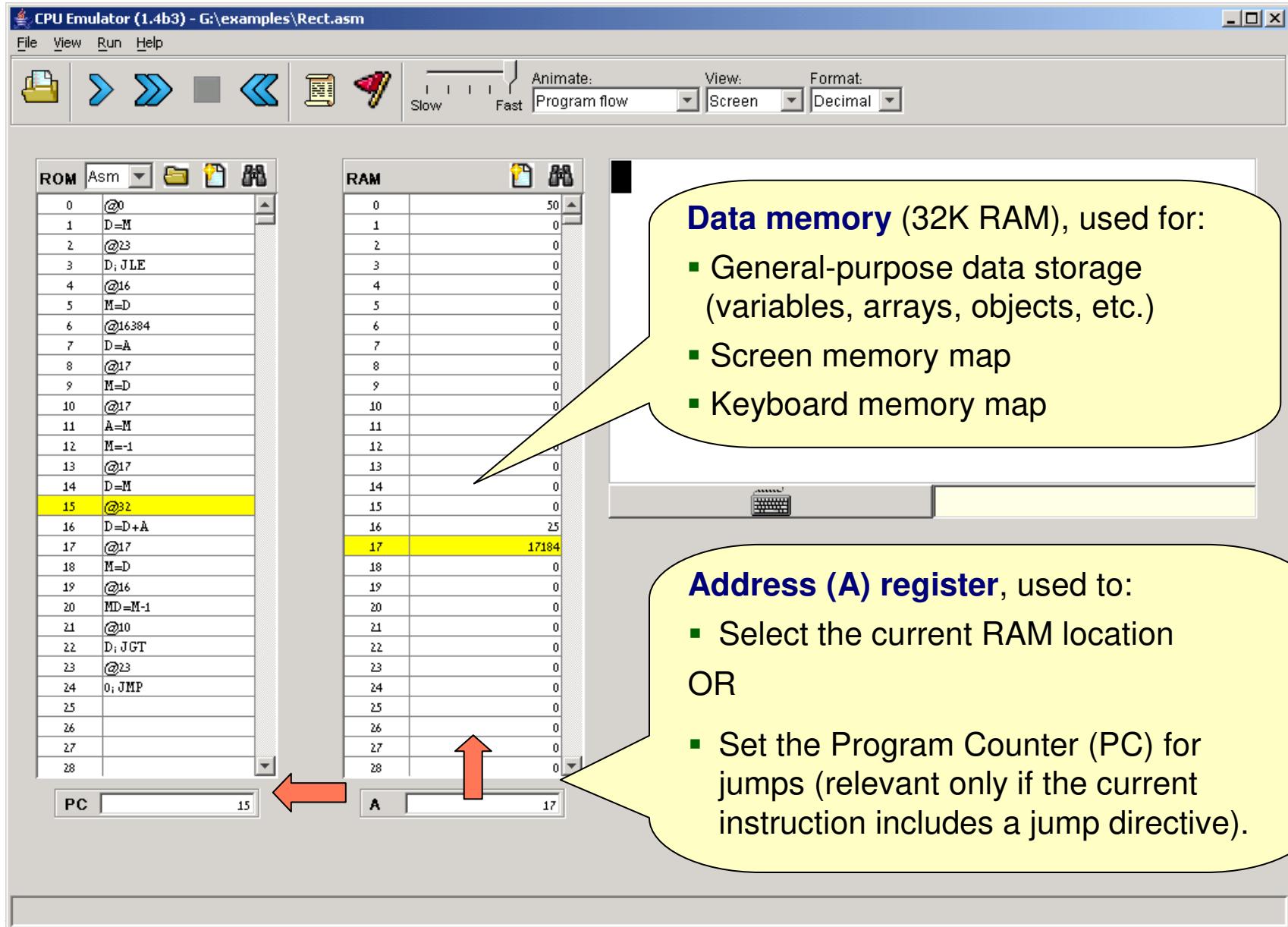
The Hack Computer Platform



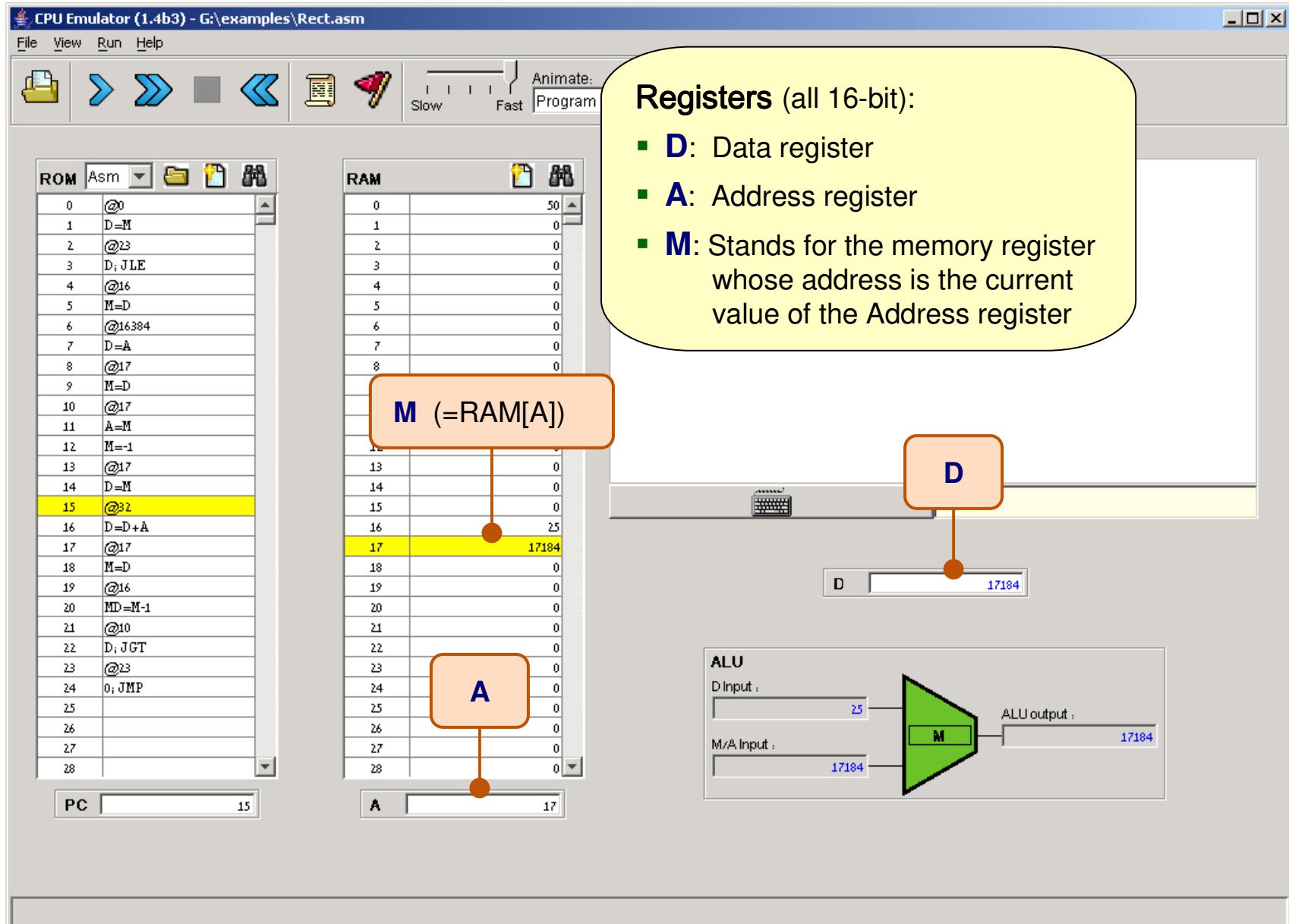
Instruction memory



Data memory (RAM)



Registers



Arithmetic/Logic Unit

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

- ROM/Asm View:** Displays assembly code. The current instruction at PC 15 is highlighted in yellow and is labeled "D=M". Other instructions include @23, D; JLE, @16, M=D, @16384, D=A, @17, M=D, @17, A=M, @17, M=-1, @17, @32, D=D+A, @17, M=D, @16, MD=M-1, D; JGT, @23, 0; JMP.
- RAM View:** Shows memory locations from 0 to 28. Address 17 contains the value 17184, which is highlighted in yellow and labeled "M (=RAM[A])". Address 25 contains the value 25, labeled "A". Address 15 contains the value 0, labeled "D".
- ALU Diagram:** An ALU block diagram with inputs D (25) and M (17184). The ALU output is labeled 17184.
- PC:** Program Counter value 15.

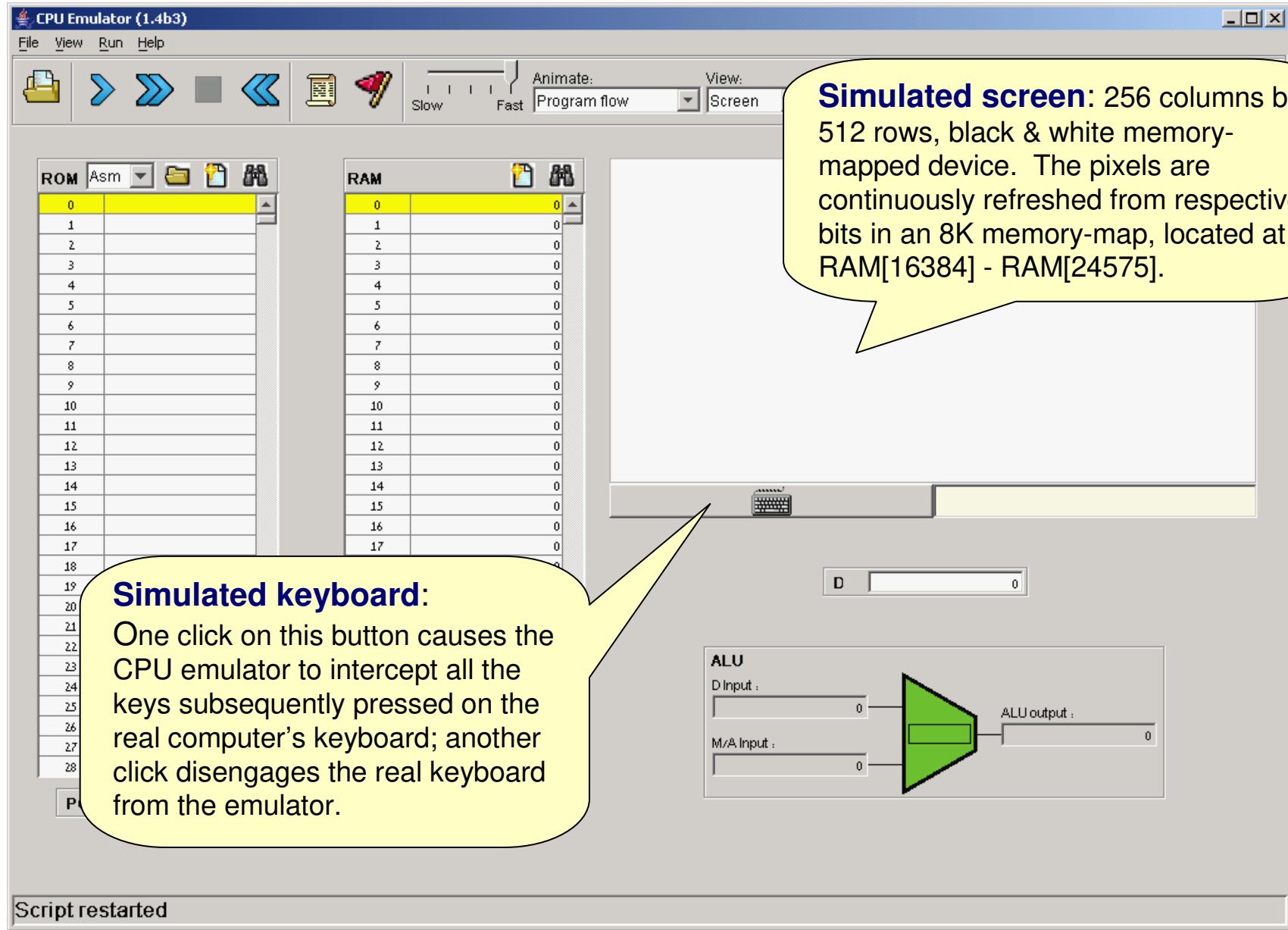
A callout bubble provides the following information about the ALU:

- The ALU can compute various arithmetic and logical functions (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



Screen action demo

The screenshot shows a CPU emulator interface. At the top, there's a menu bar with "File", "Edit", "Run", "Break", "Step", "View", and "Format". The "Format" dropdown is set to "Decimal". Below the menu is a toolbar with icons for "Program flow", "Screen", and "Decimal". The main area has three panes: a left pane showing memory addresses from 19 down to 28, a middle pane showing the screen memory map with address 16384 highlighted, and a right pane showing the screen output. A red circle highlights the screen output area. A large yellow callout bubble points to the screen output, containing the following text:

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.

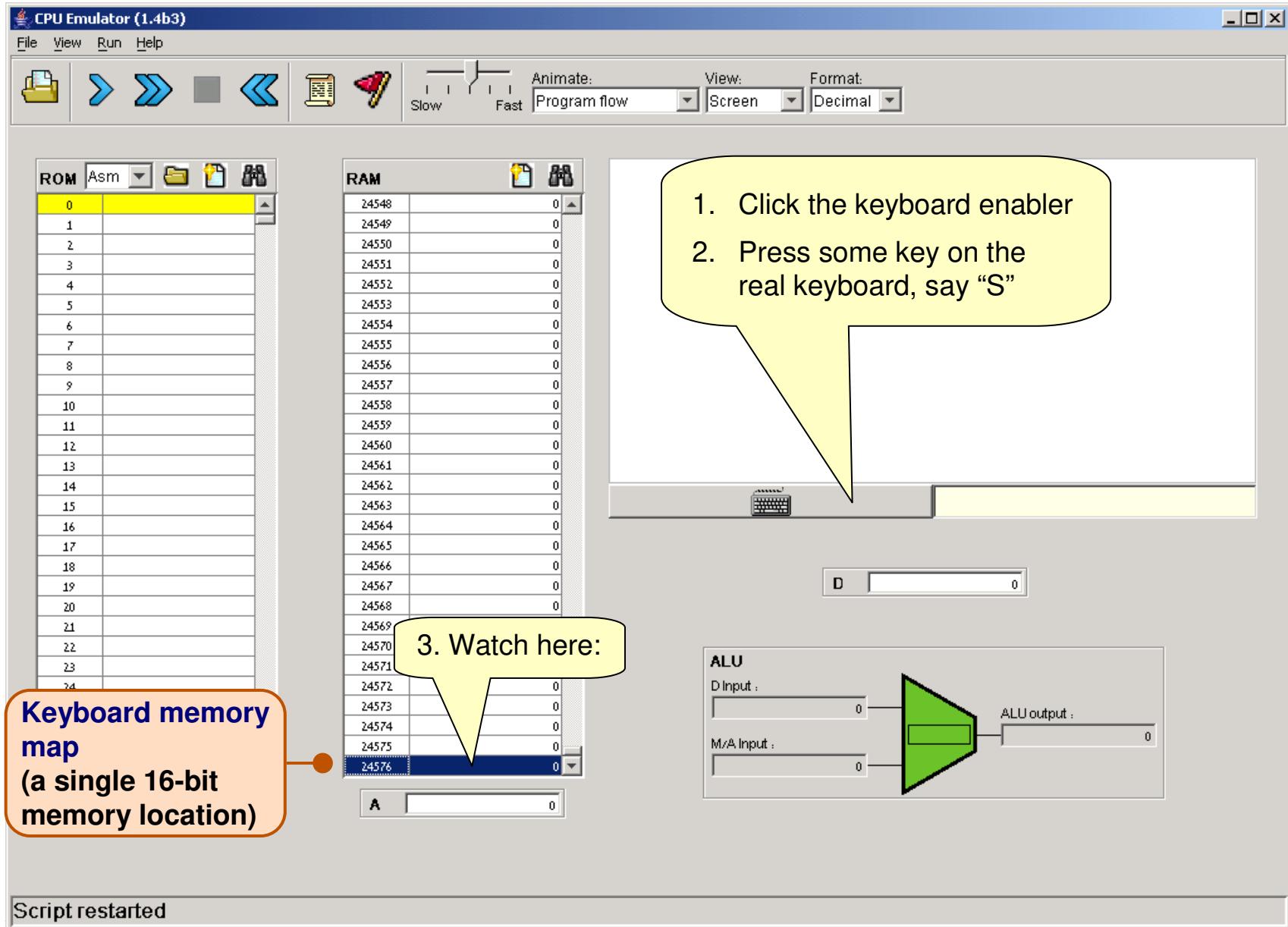
Since all high level programs and OS routines are eventually translated into machine language, they all end up doing something like this example.

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)

Address	Value
16382	0
16383	0
16384	-1
16385	0
16386	0
16387	0
16388	0
16389	0
16390	0
16391	0

Keyboard action demo



Script restarted

Keyboard action demo

The screenshot shows the CPU Emulator interface. The top menu bar includes File, View, Run, and Help. The toolbar has options for Trace, View, and Format, with Screen selected. The main window has three panes: a memory map on the left, a screen display in the center, and a keyboard input area at the bottom.

- 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

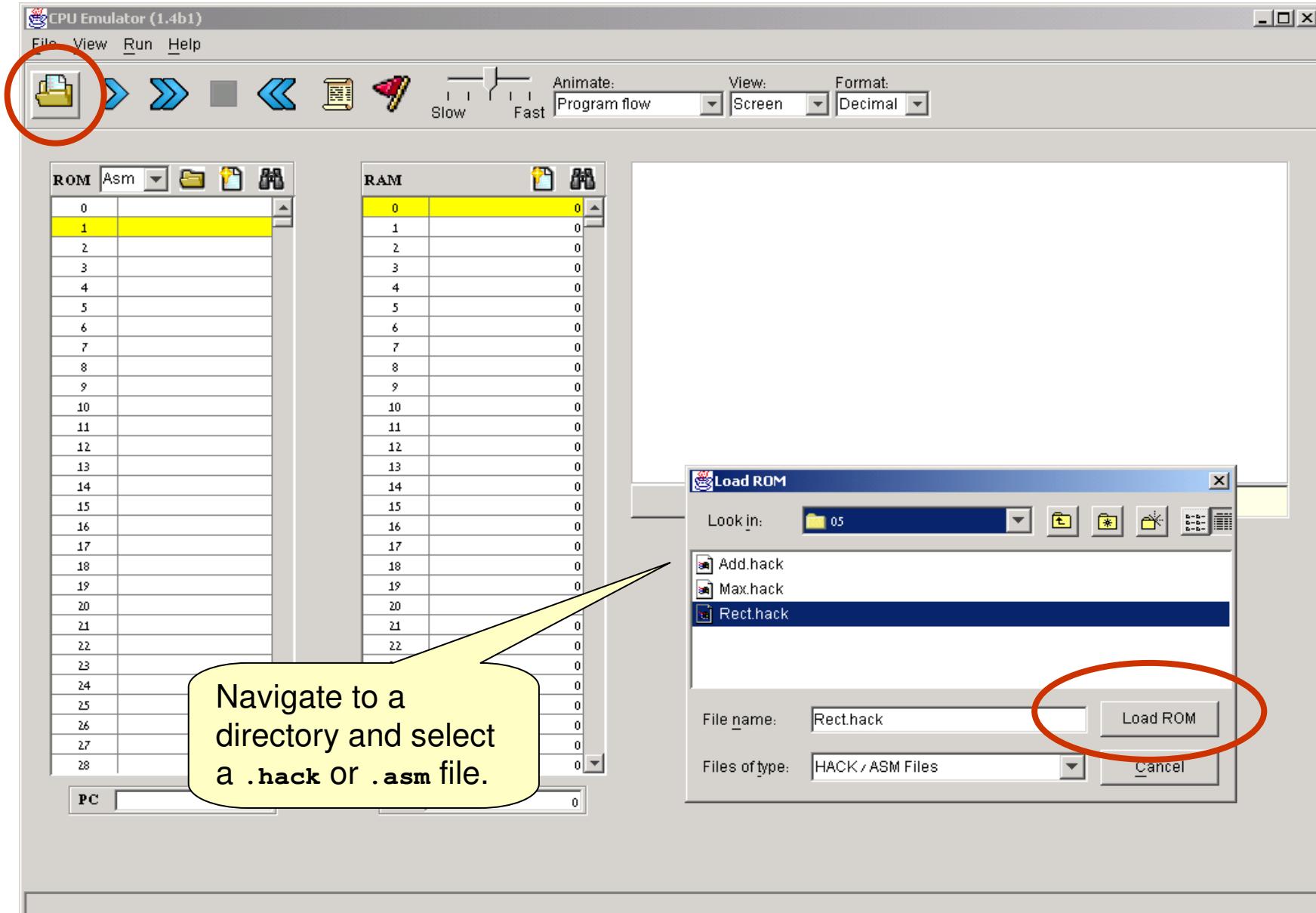
Address	Value
20	0
21	0
22	0
23	0
24	0
24568	0
24569	0
24570	0
24571	0
24572	0
24573	0
24574	0
24575	0
24576	83

Script restarted

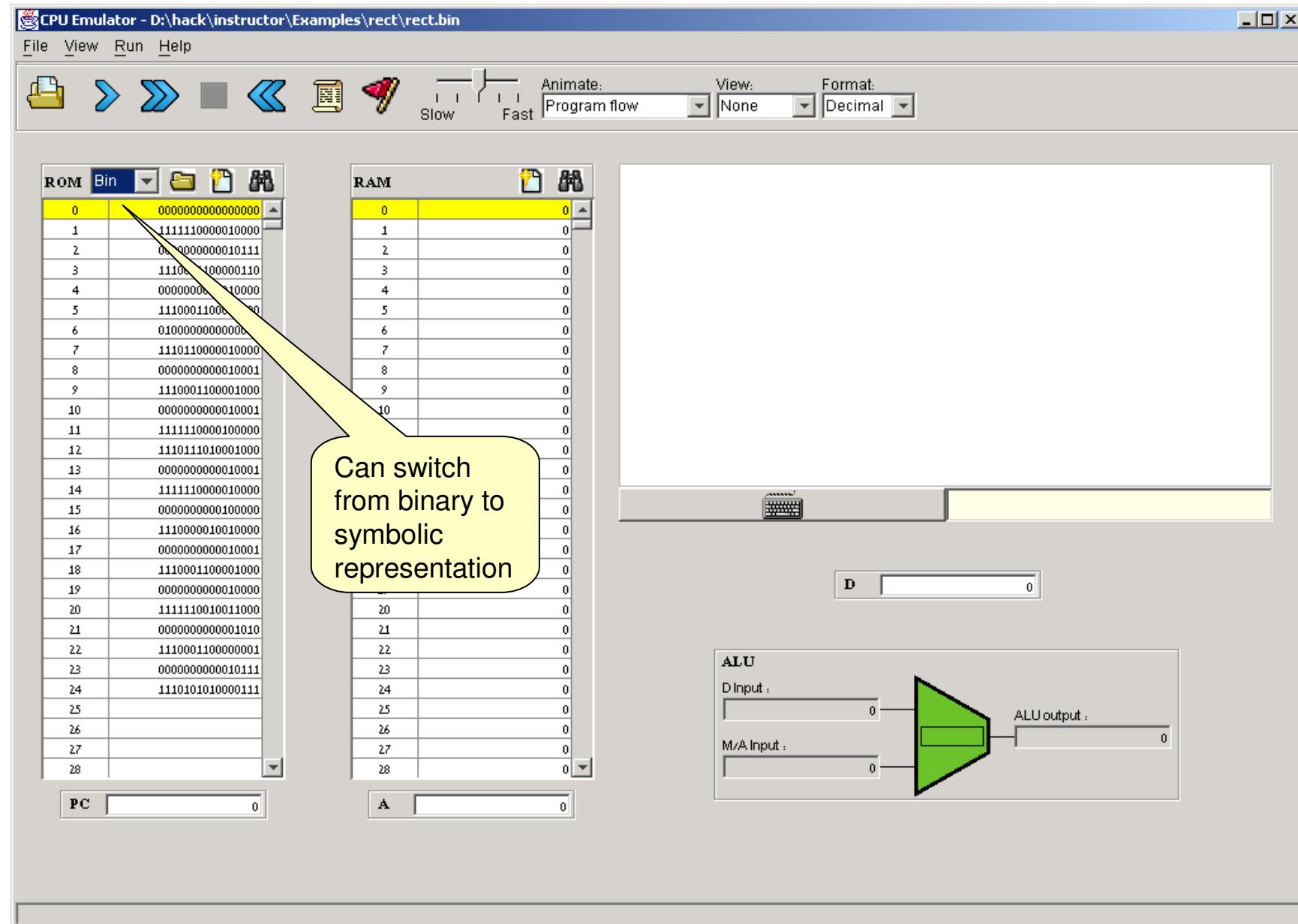
CPU Emulator Tutorial



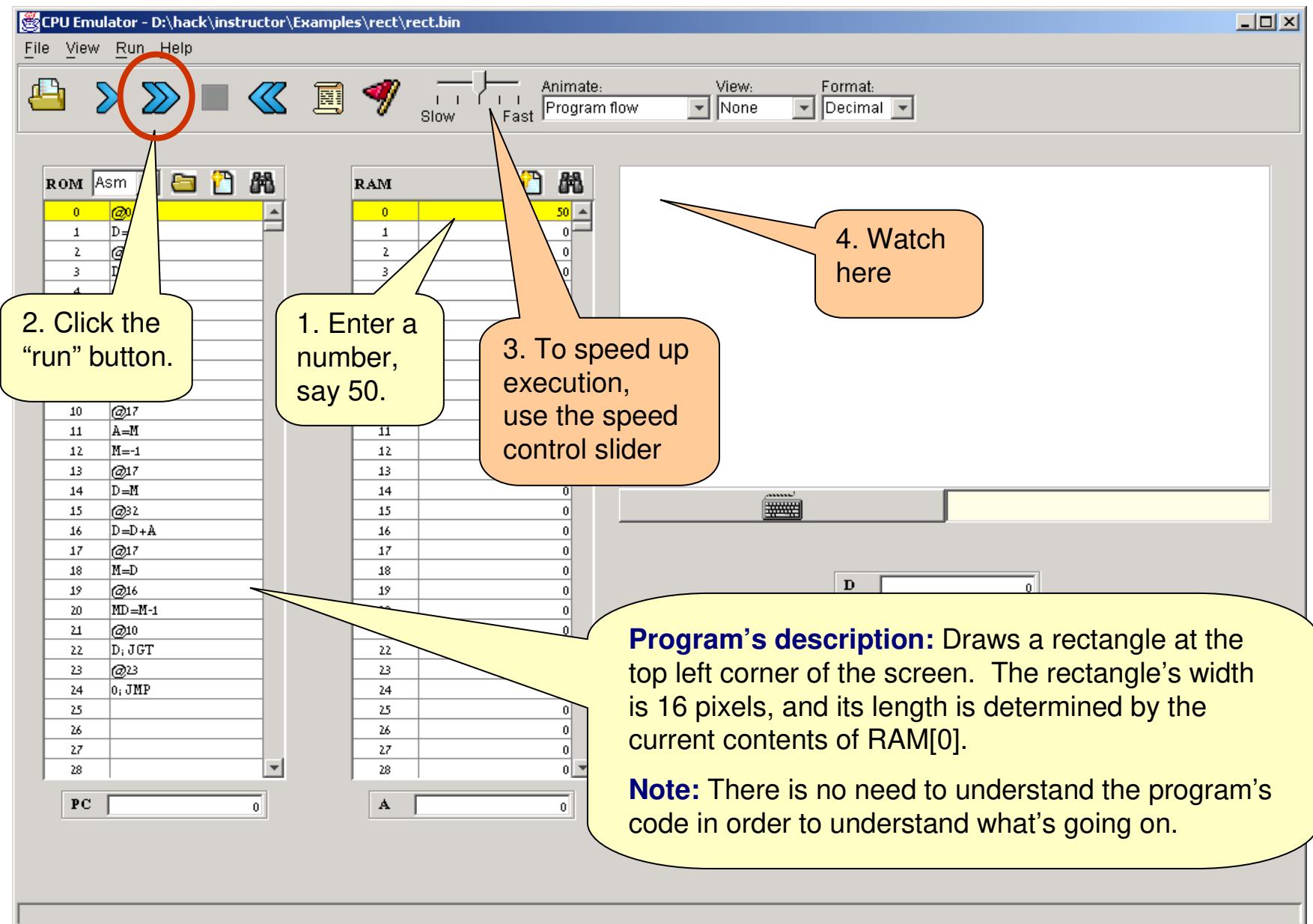
Loading a program



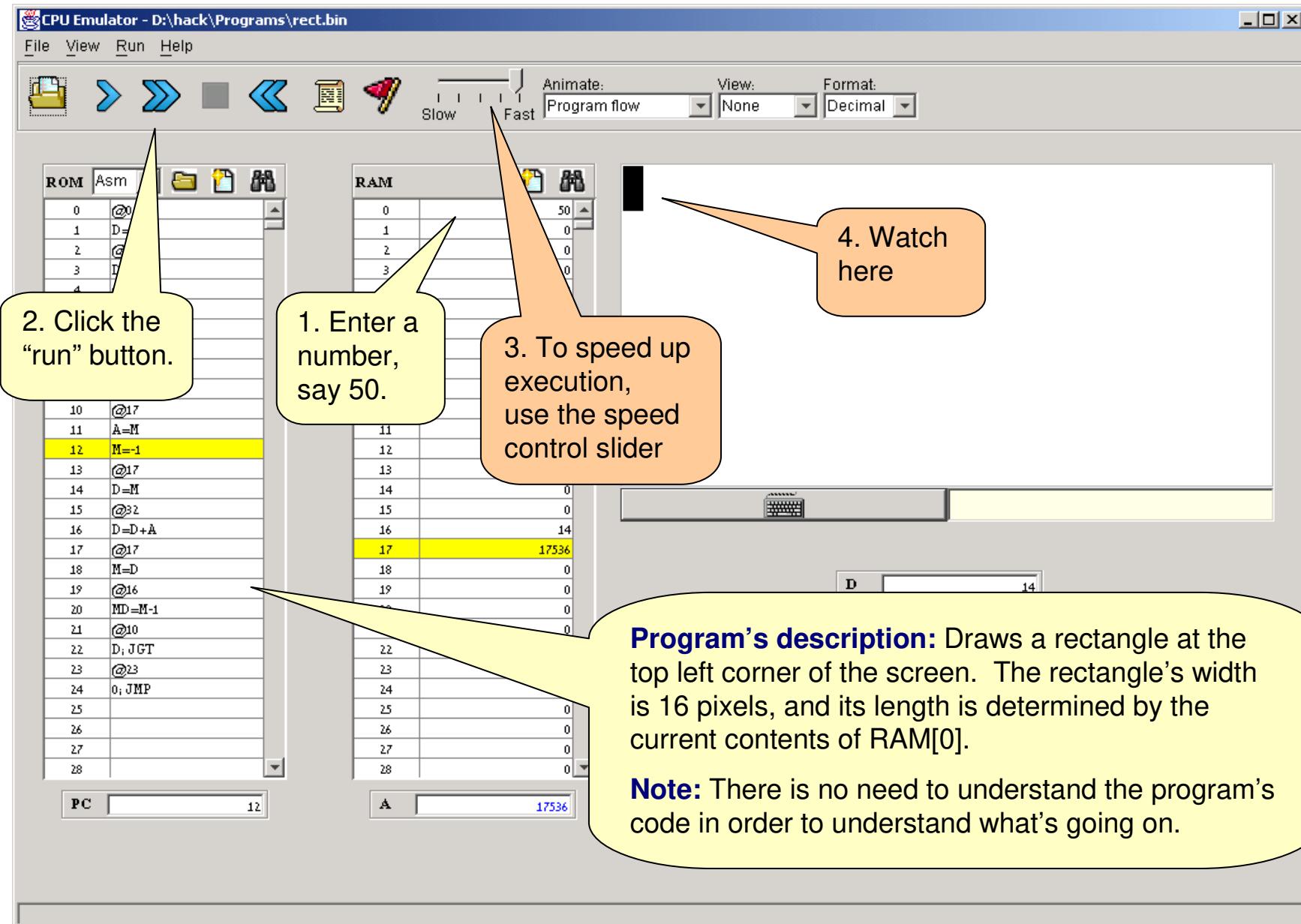
Loading a program



Running a program



Running a program



Hack programming at a glance (optional)

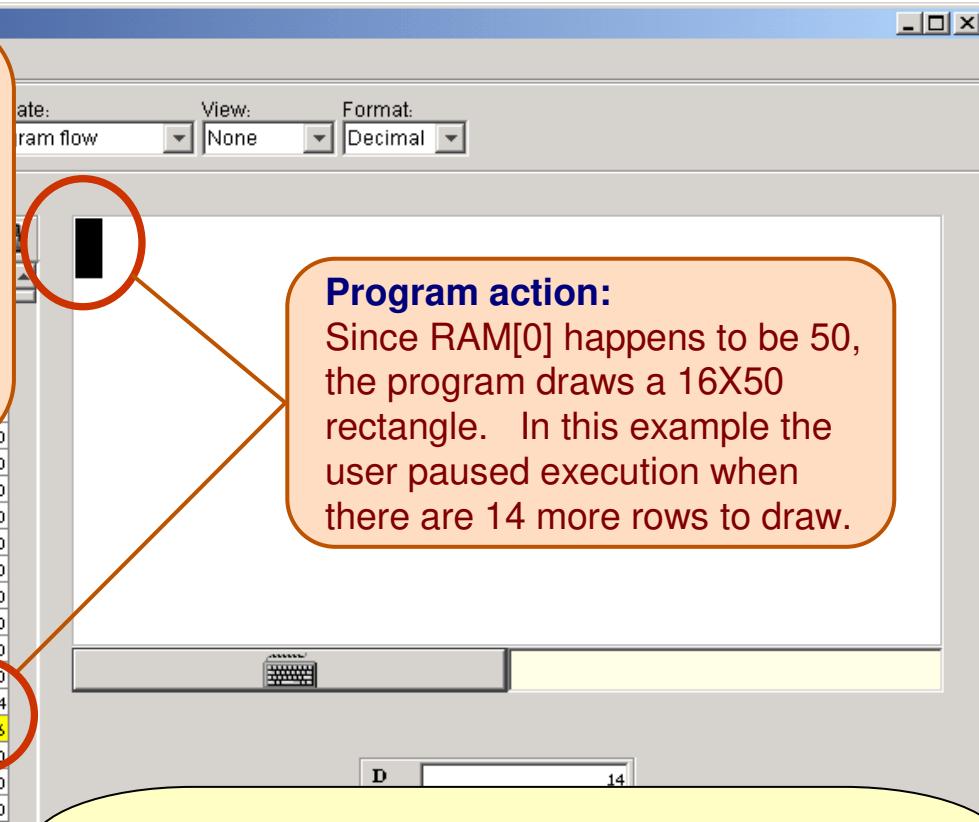
Next instruction is M=-1.

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.

The screenshot shows a CPU emulator interface. On the left, the assembly code window displays the following instructions:

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	

The PC register (bottom left) shows the value 12. The A register (bottom right) shows the value 17536. The D register (bottom center) shows the value 14. A callout points from the M=-1 instruction to the A register, indicating that the next instruction will modify the A register.



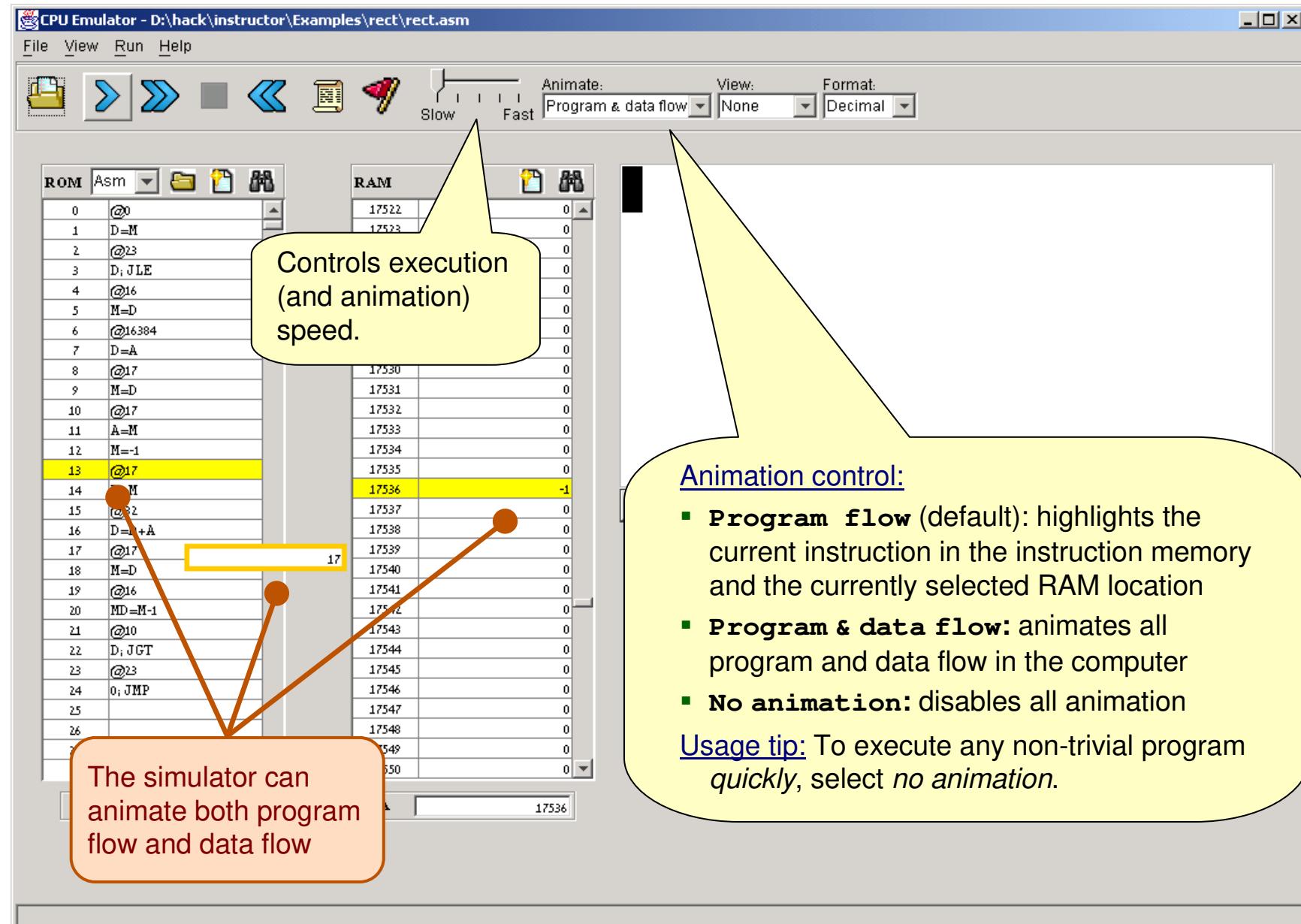
Program action:

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





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*.

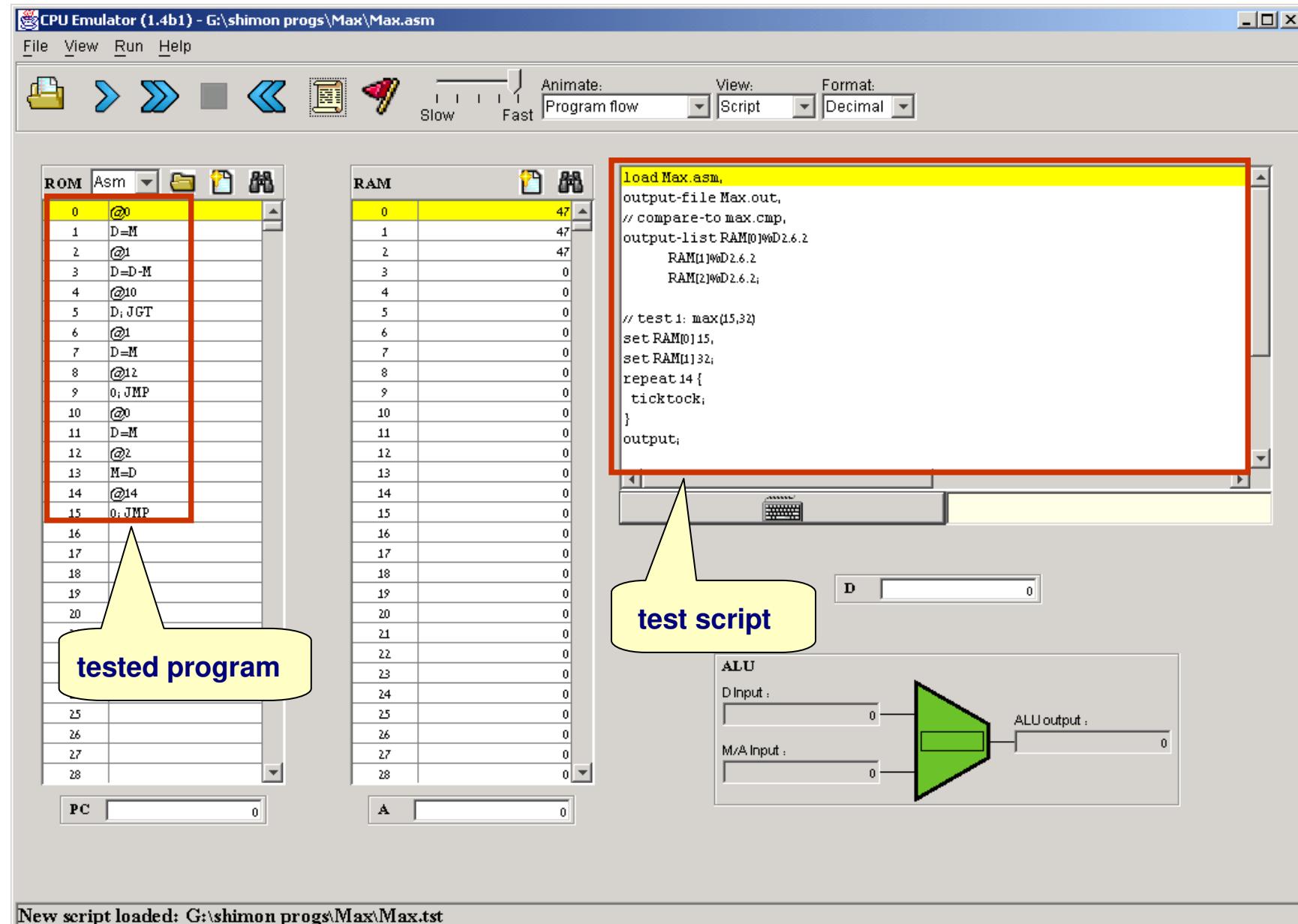
Script-based simulation enables planning and using tests that are:

- 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.

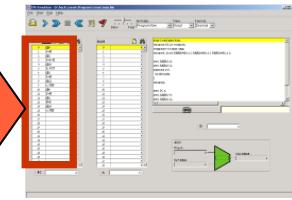
The basic setting



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:

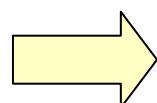
- Loading programs
- Setting up output and compare files
- Writing values into RAM locations
- Writing values into registers
- Executing the next command (“**ticktock**”)
- Looping (“**repeat**”)
- And more (see Appendix B).



Notes:

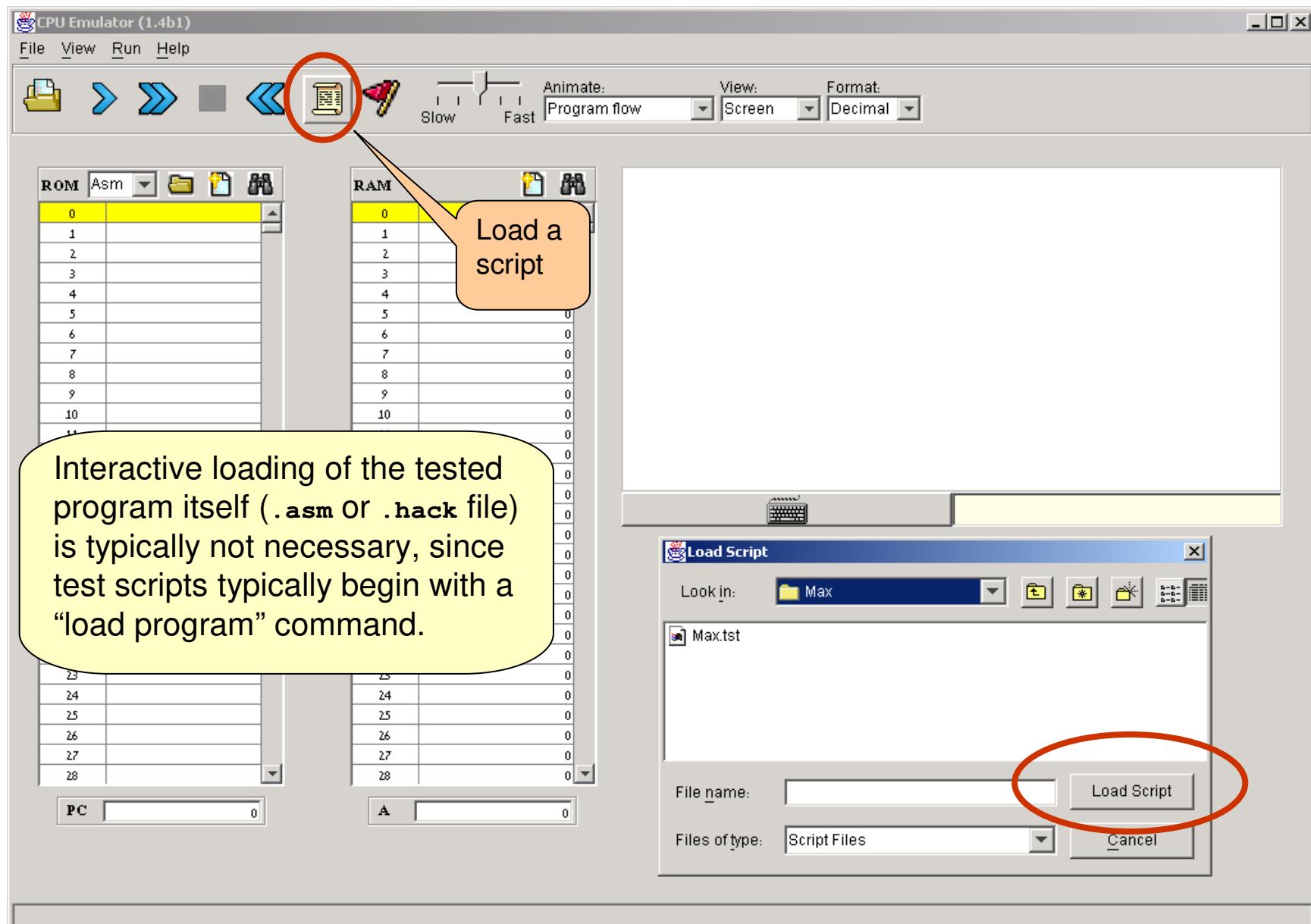
- As it turns out, the Max program requires 14 cycles to complete its execution
- All relevant files (**.asm**, **.tst**, **.cmp**) must be present in the same directory.

Output

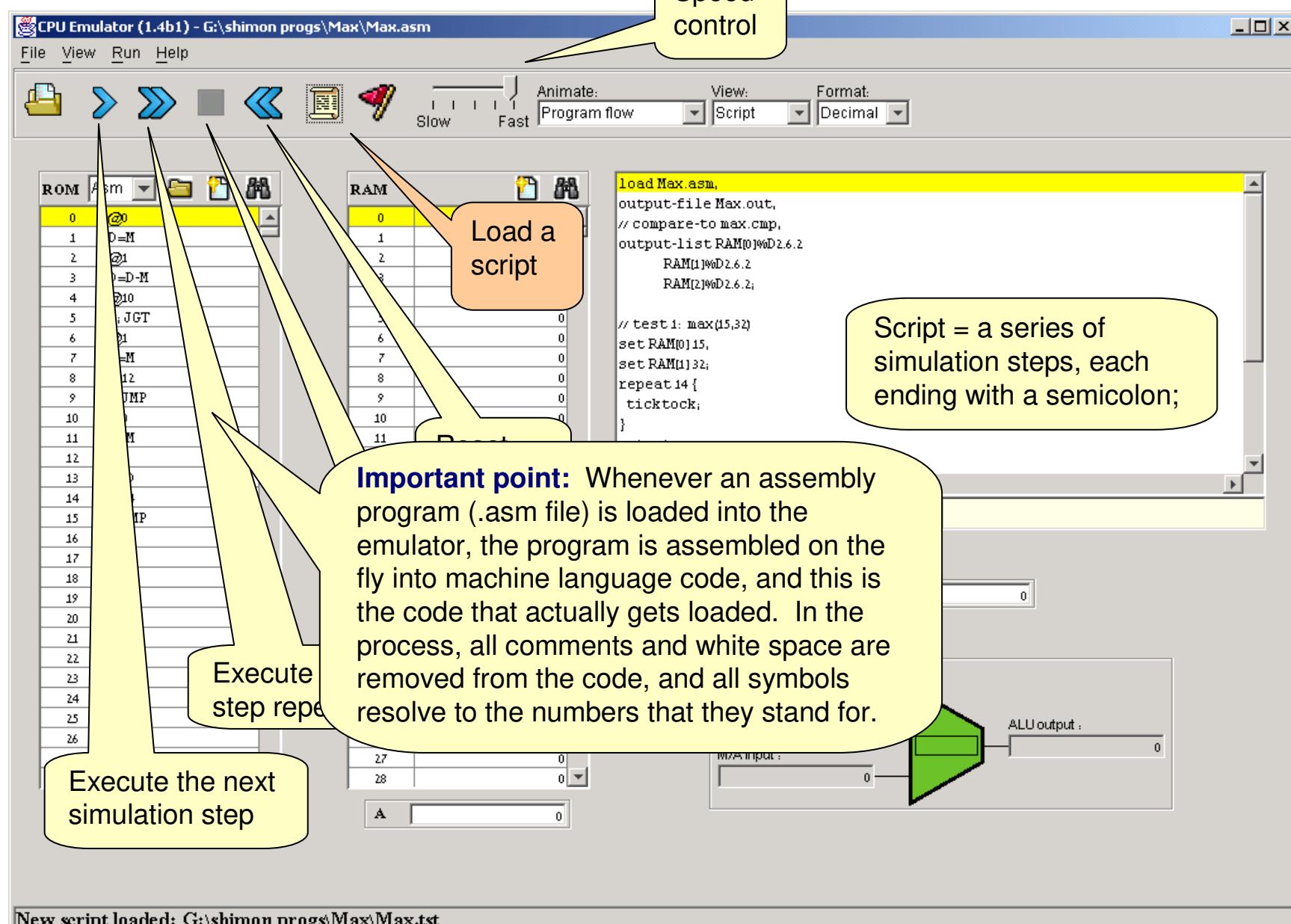


	RAM[0]	RAM[1]	RAM[2]	
	15	32	32	
	47	22	47	

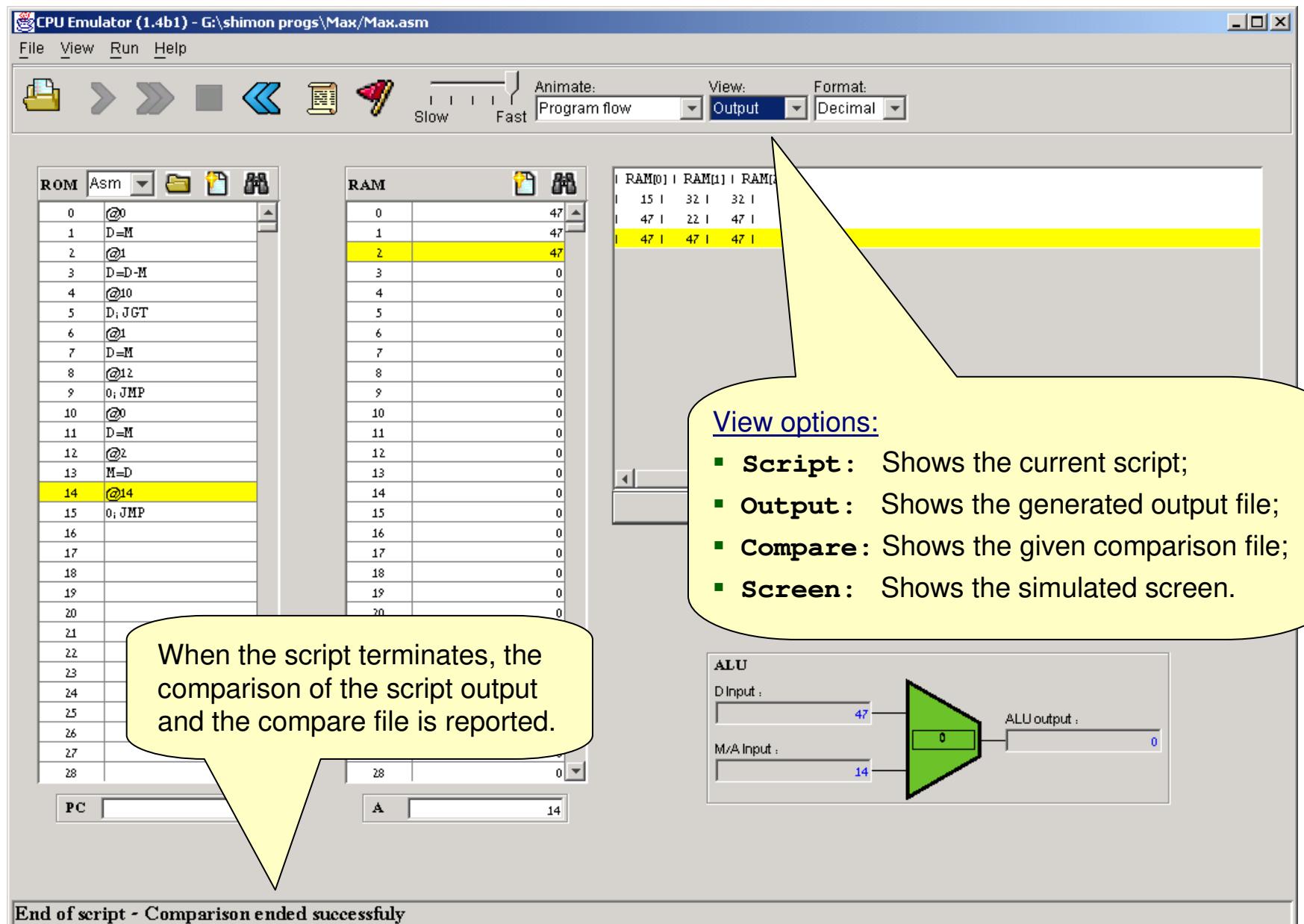
Using test scripts



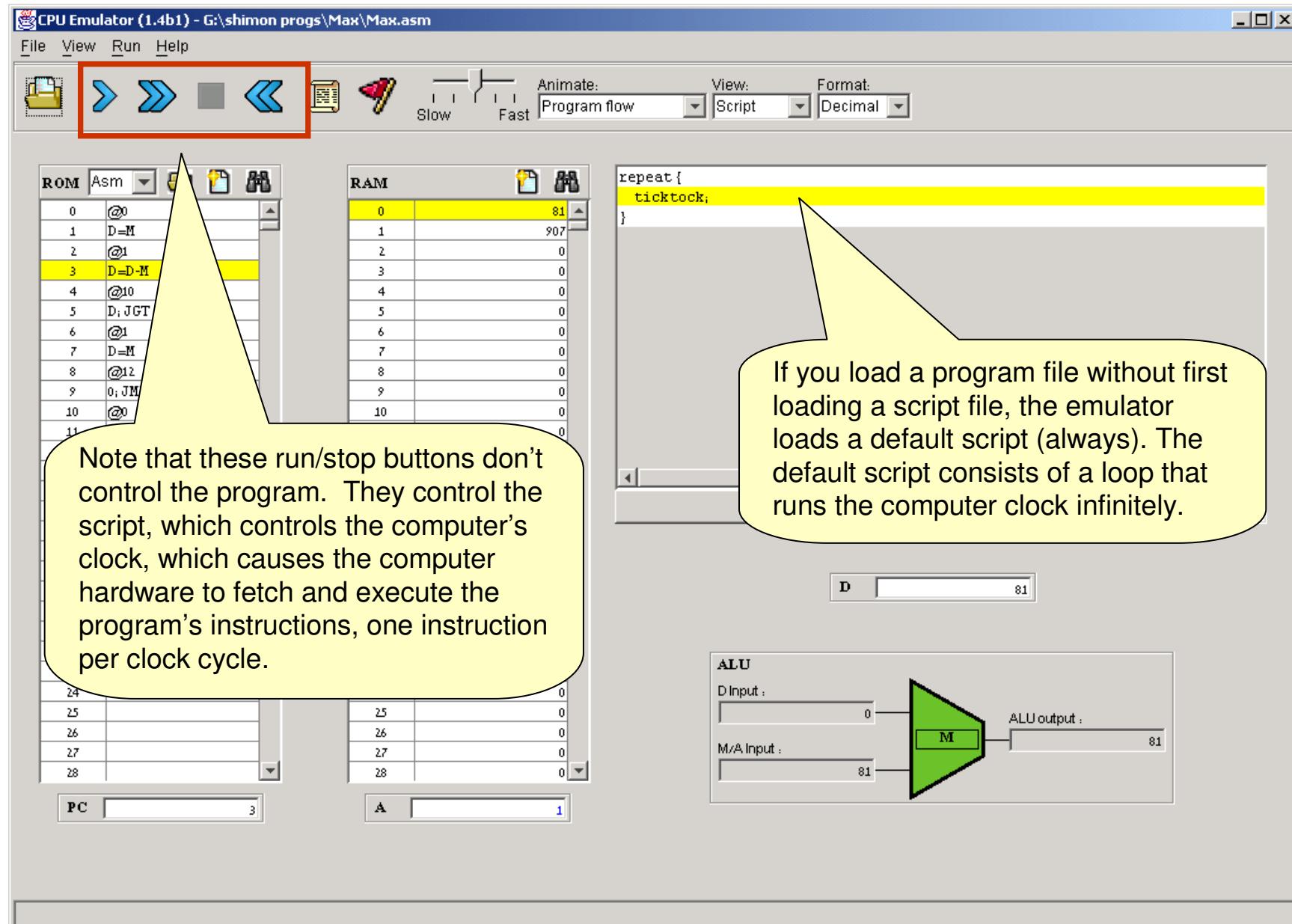
Using test scripts



Using test scripts



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



CPU Emulator Tutorial



Breakpoints: a powerful debugging tool

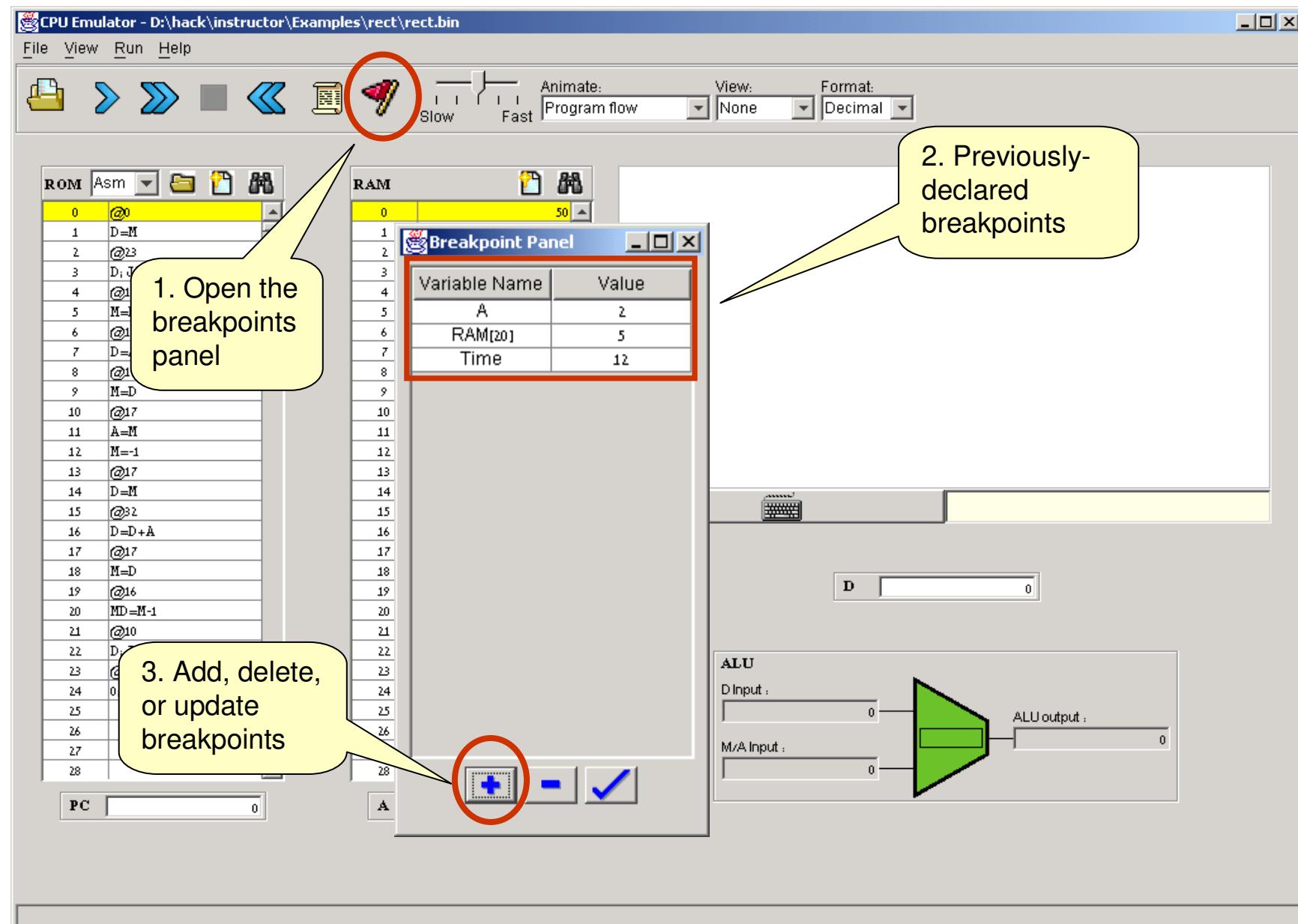
The CPU emulator continuously keeps track of:

- **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

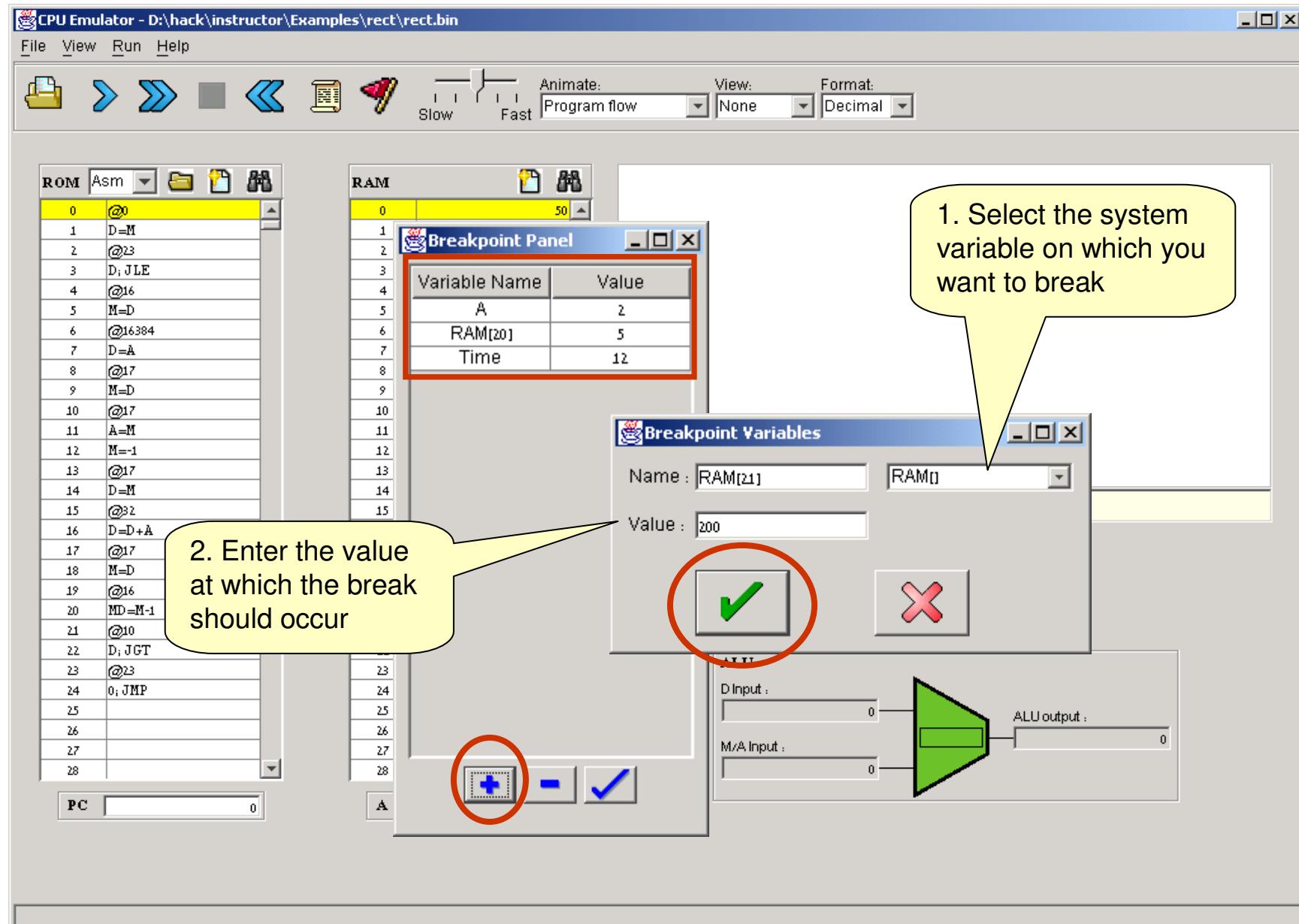
Breakpoints:

- 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.

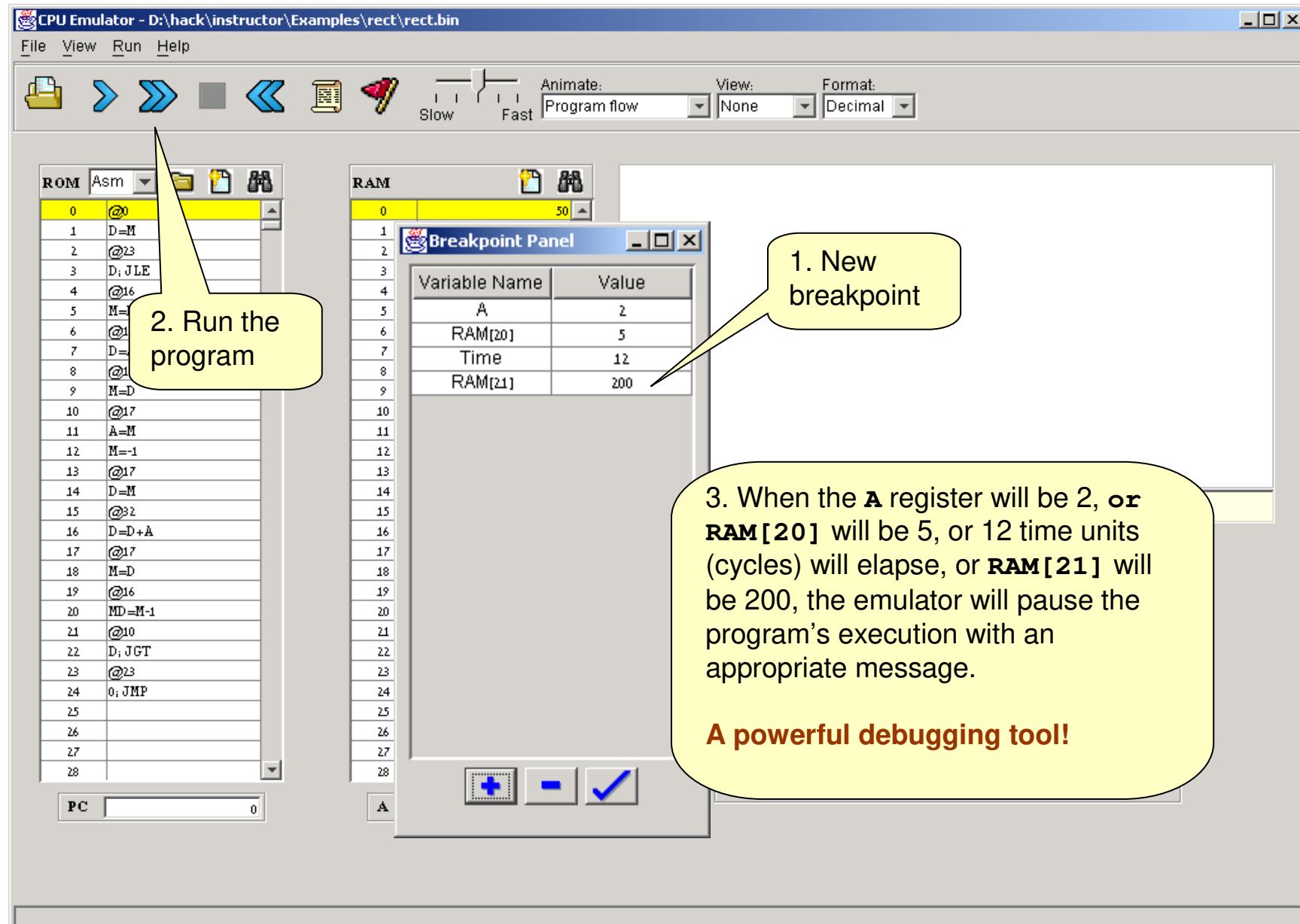
Breakpoints declaration



Breakpoints declaration



Breakpoints usage



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).

