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

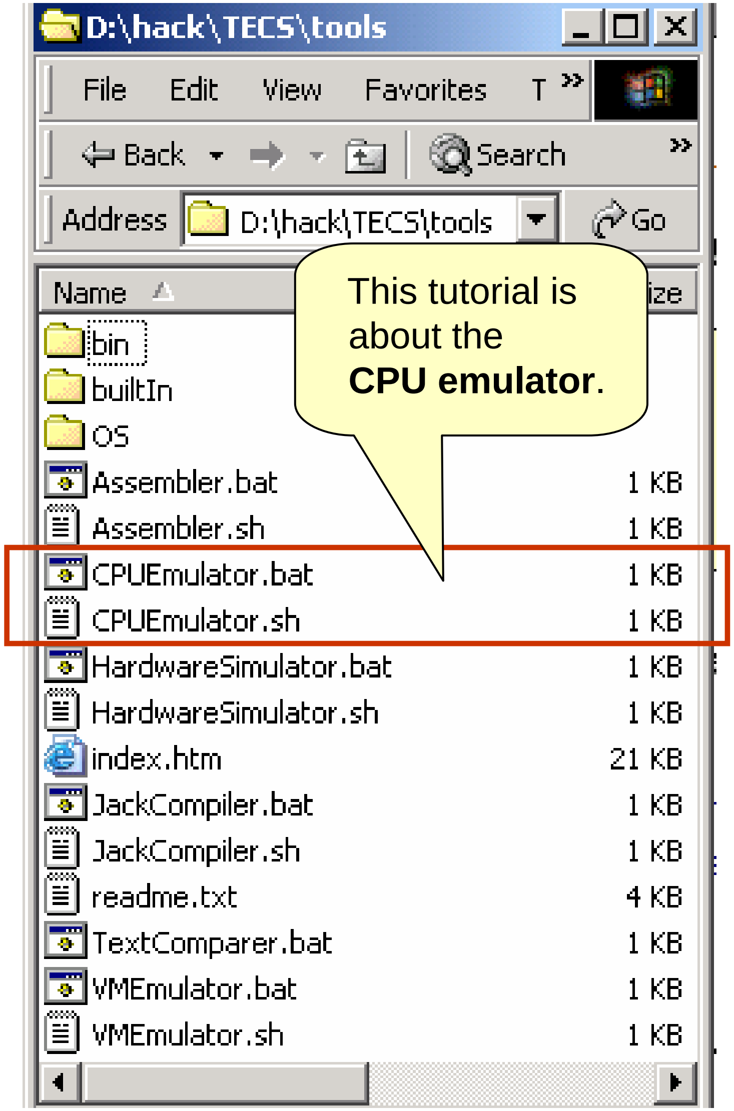
The Elements of Computing Systems evolves around the construction of a complete computer system, done in the framework of a 1- or 2-semester course.

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`**, **`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

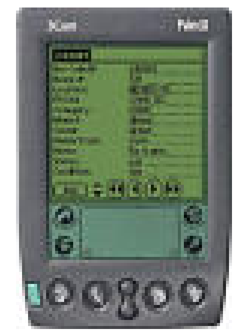
The Hack computer

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.

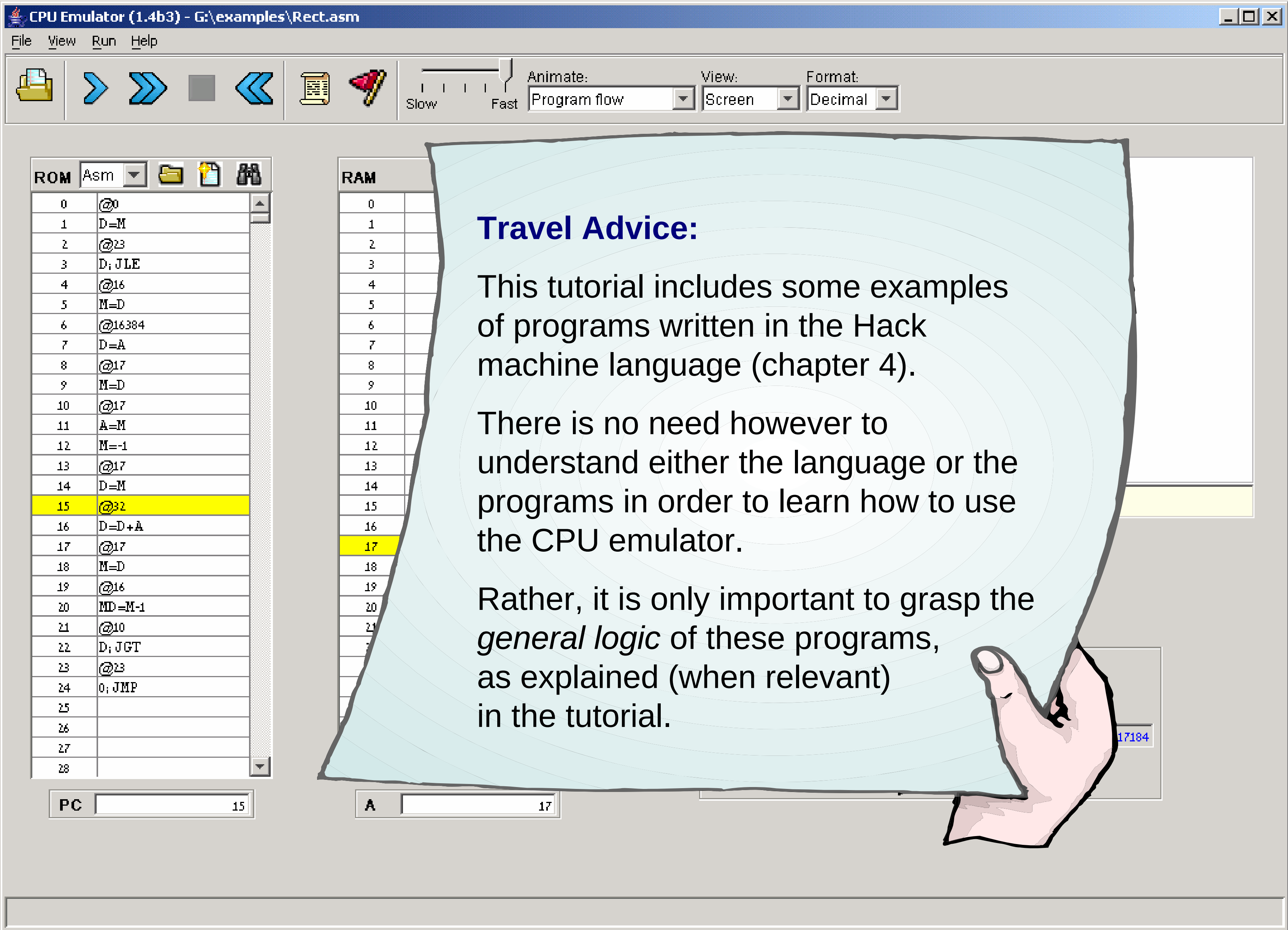


Relevant reading (from "*The Elements of Computing Systems*"):

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


Part I:
Basic Platform

The Hack Computer Platform (simulated)



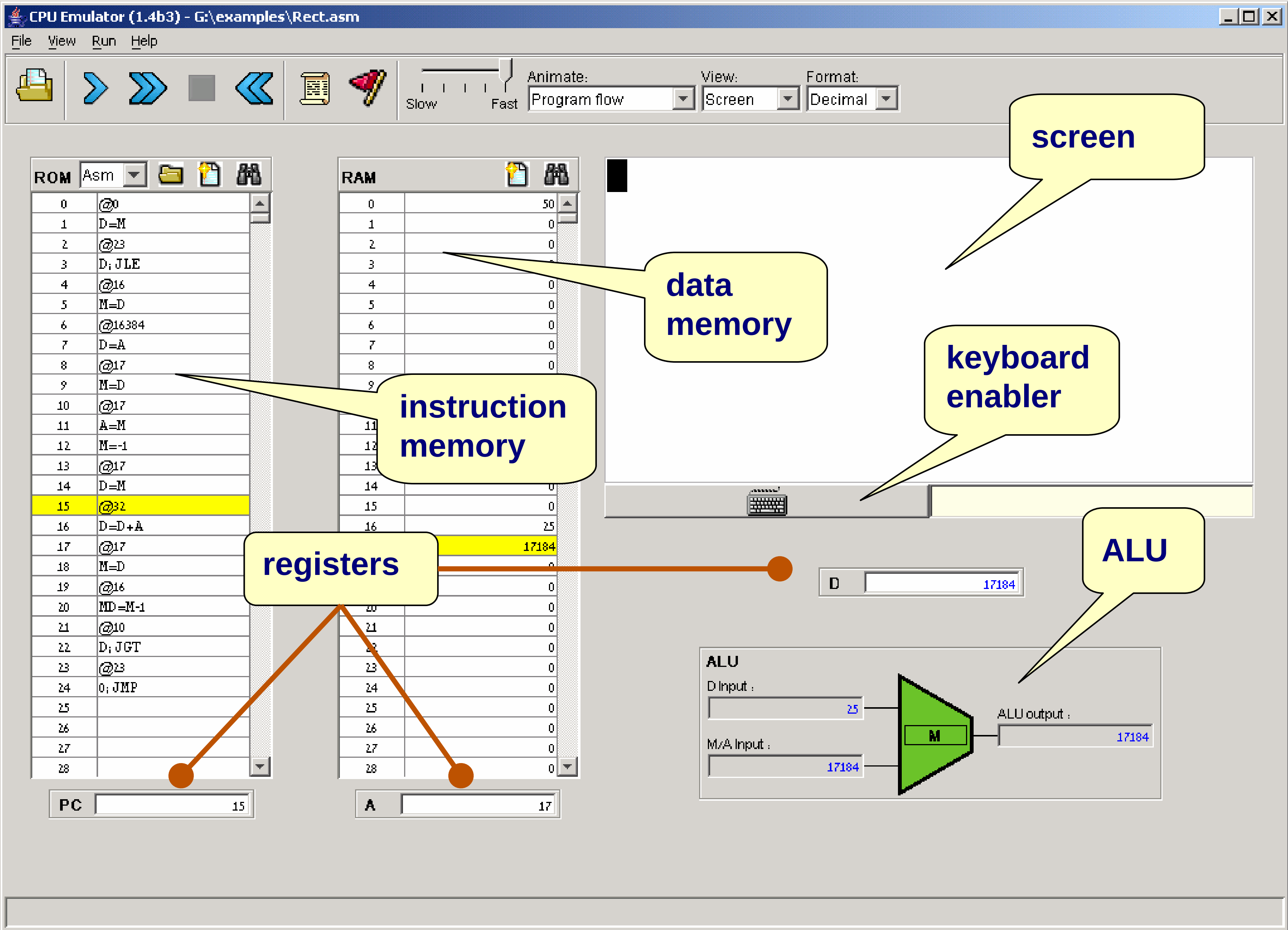
Travel Advice:

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

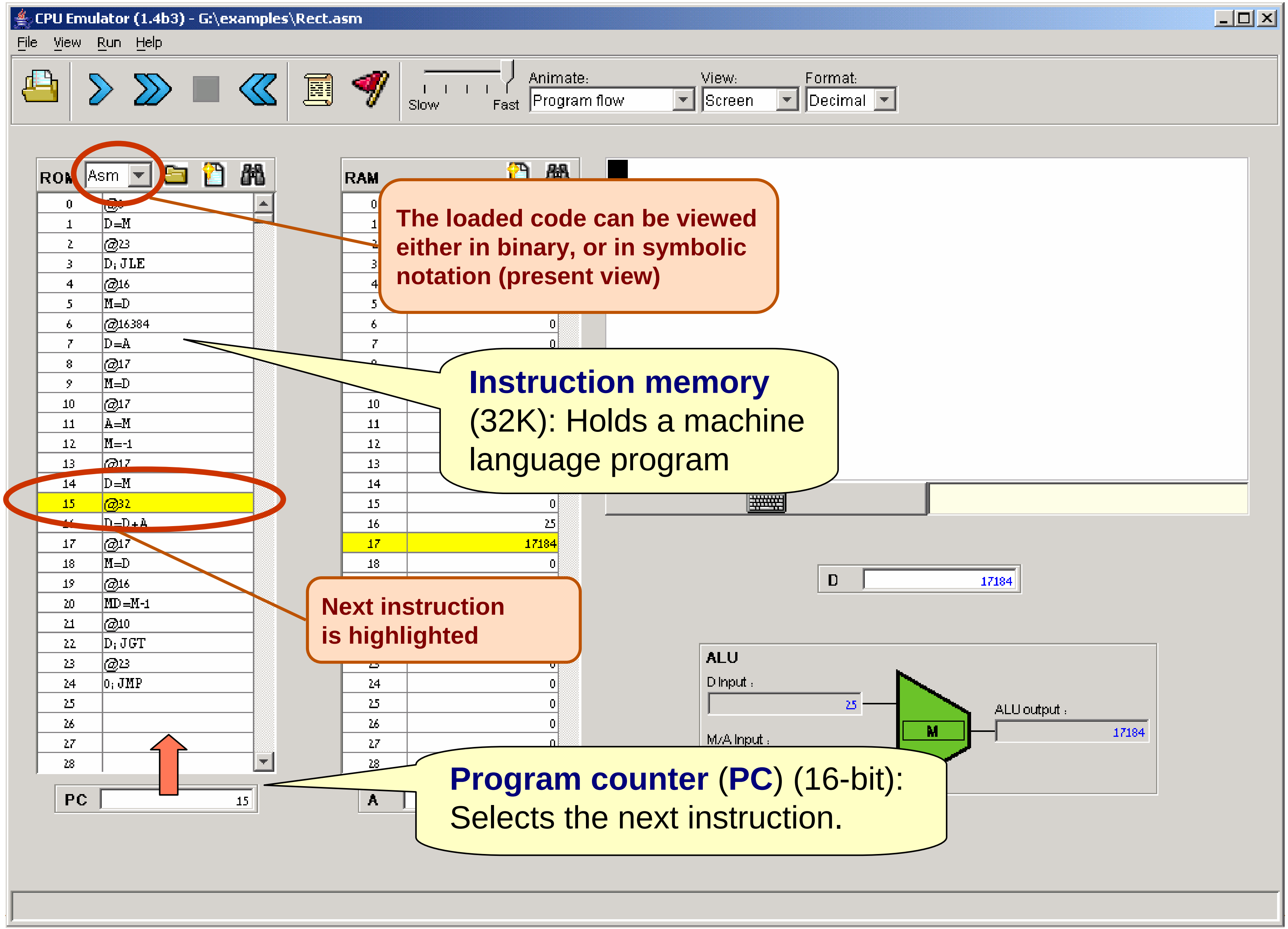
There is no need however to understand either the language or the programs in order to learn how to use the CPU emulator.

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

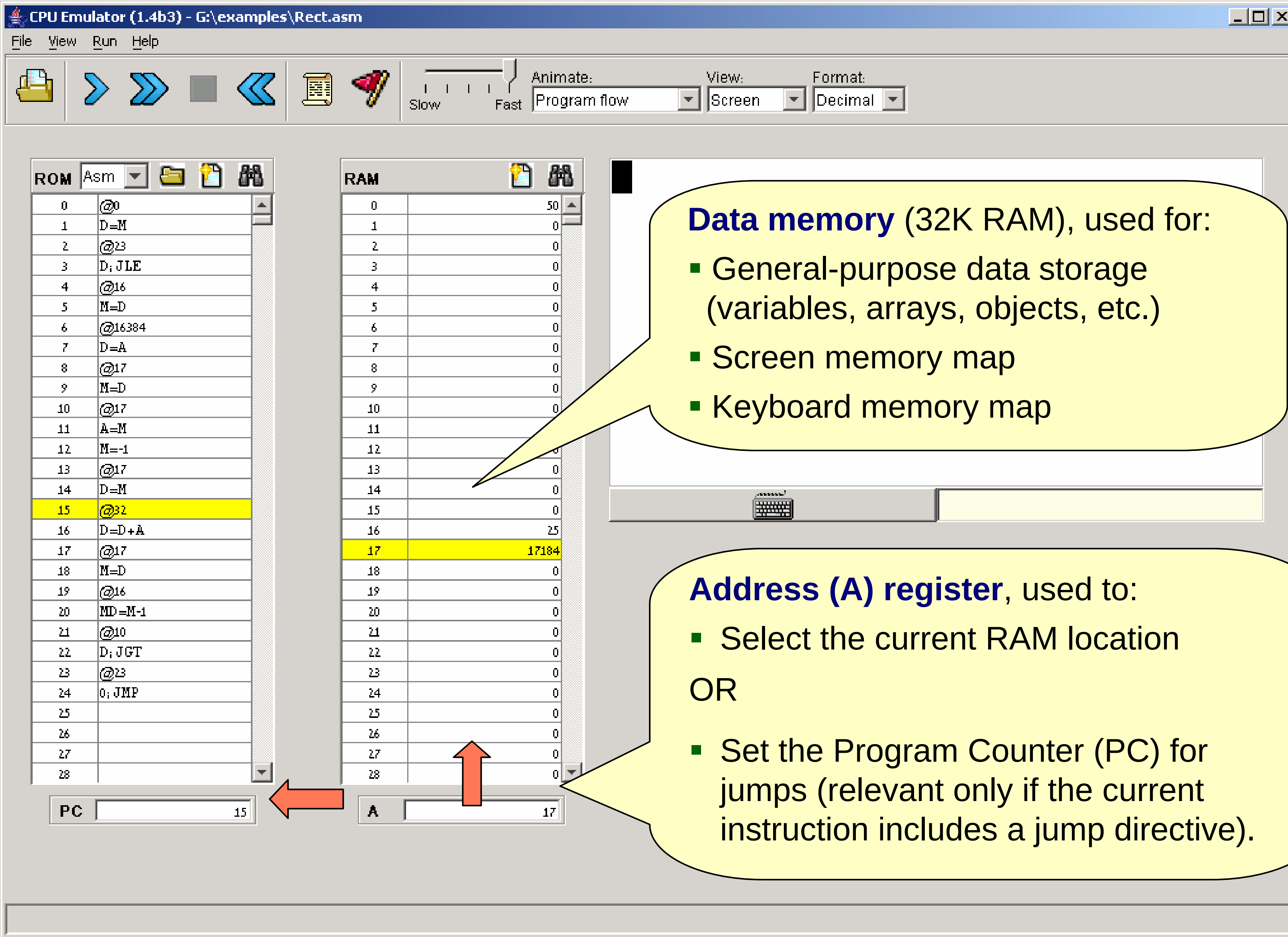
The Hack Computer Platform



Instruction memory



Data memory (RAM)



Data memory (32K RAM), used for:

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

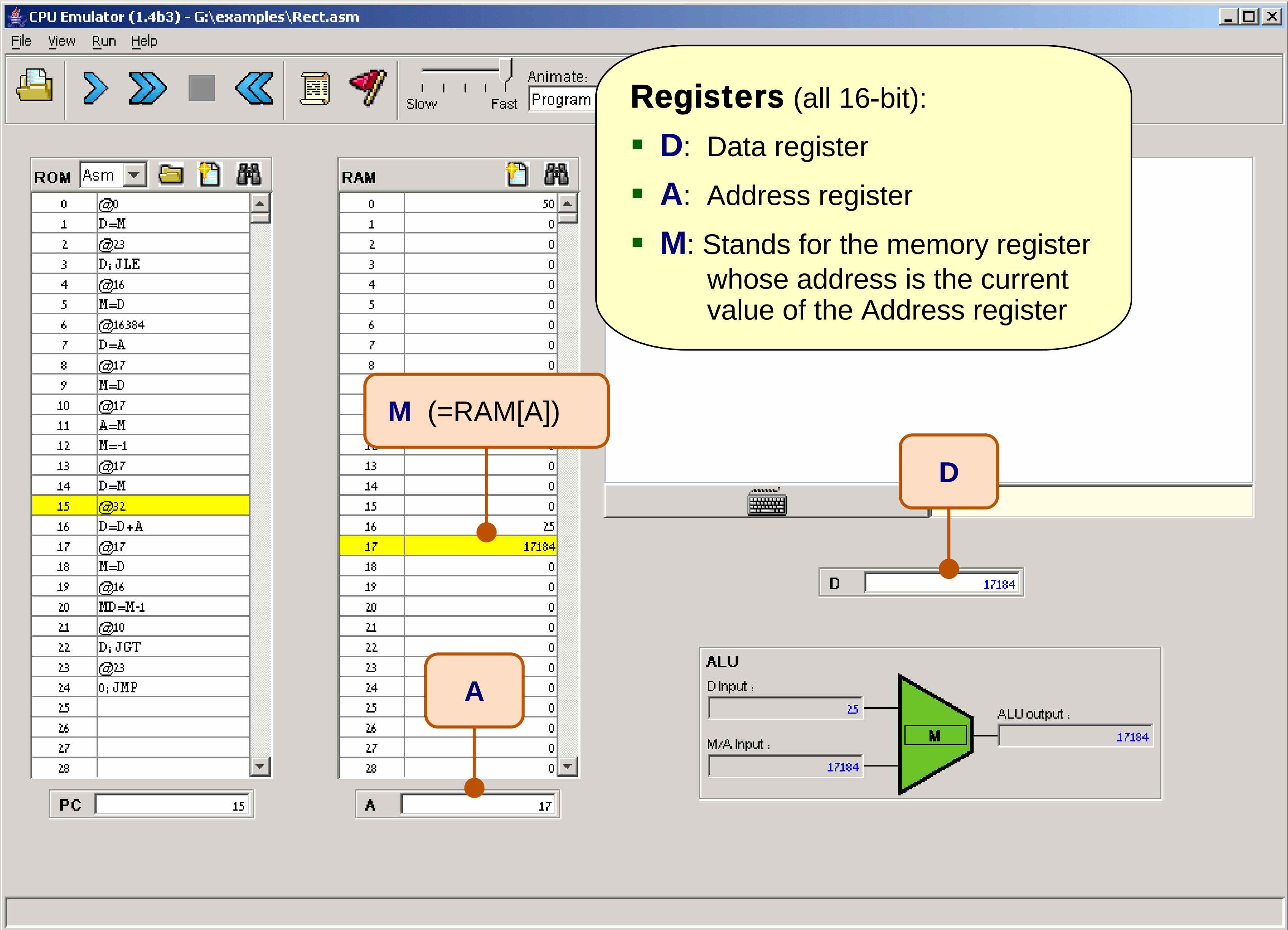
Address (A) register, used to:

- Select the current RAM location

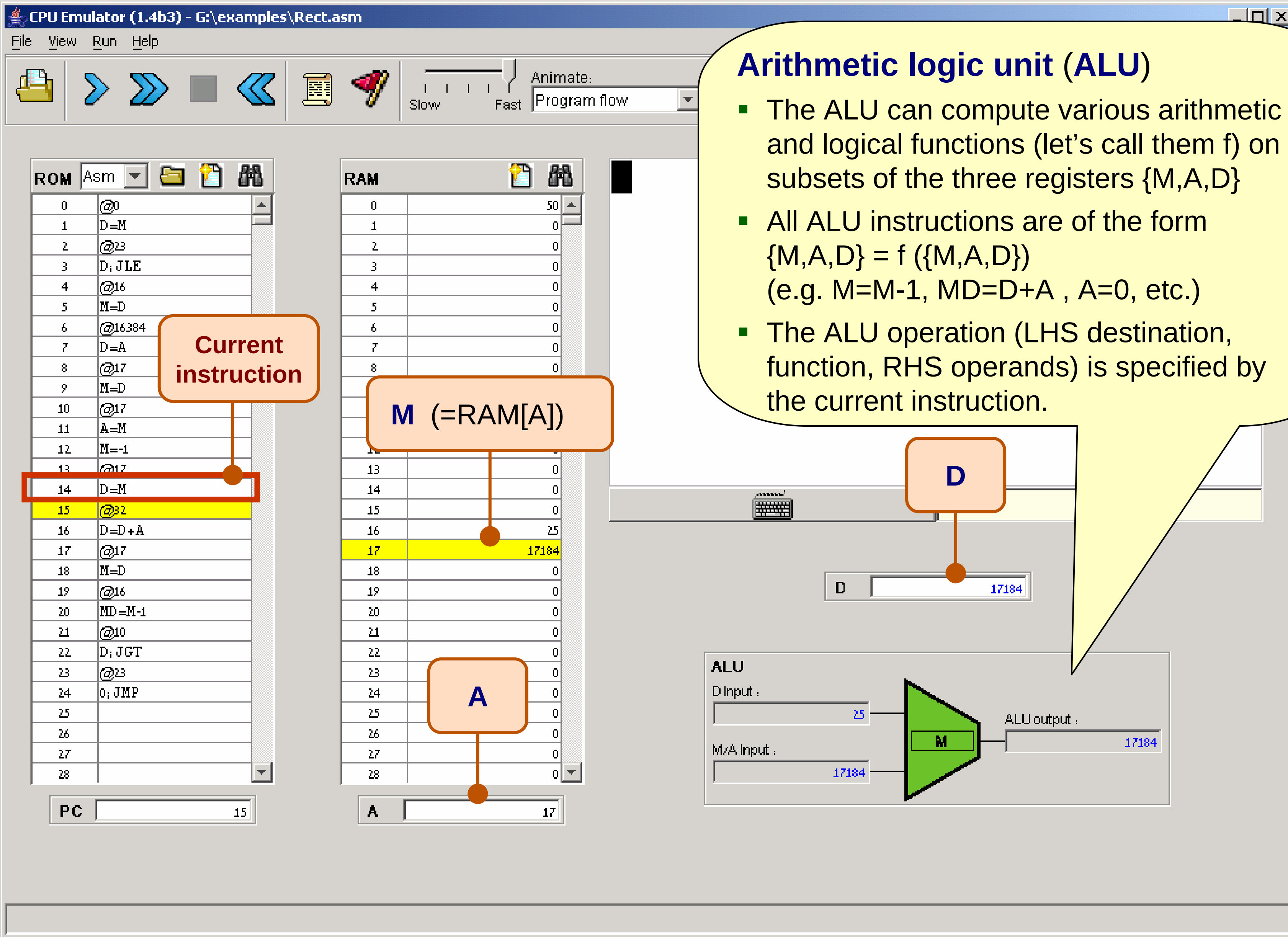
OR

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

Registers


CPU Emulator (1.4b3) - G:\examples\Rect.asm
File View Run Help
Animate:
Program
Slow
Fast
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])
D
A
ROM
Asm
RAM
PC
ALU
D Input :
M/A Input :
ALU output :

Arithmetic/Logic Unit

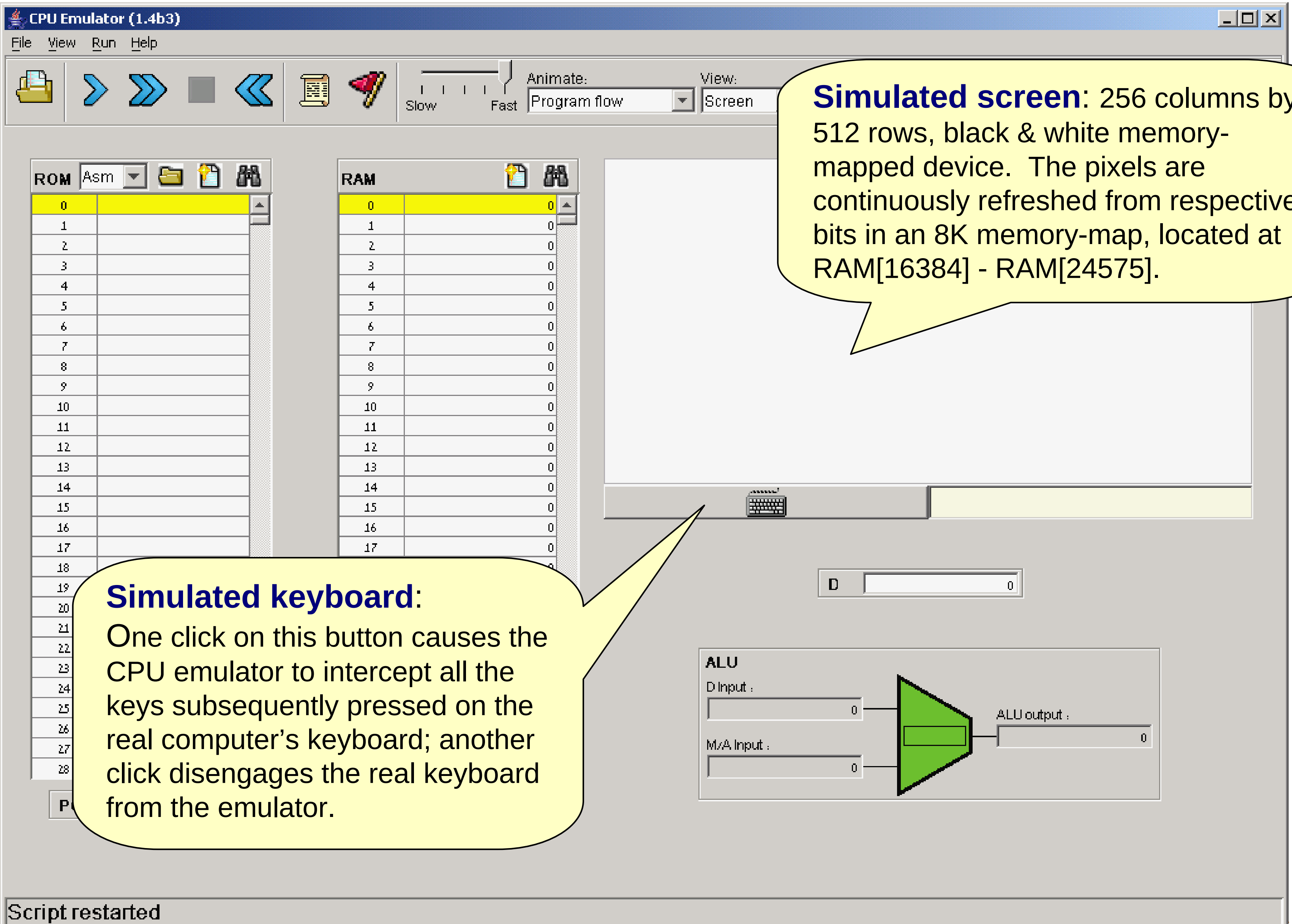


Arithmetic logic unit (ALU)

- 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.


Part II:
I/O Devices

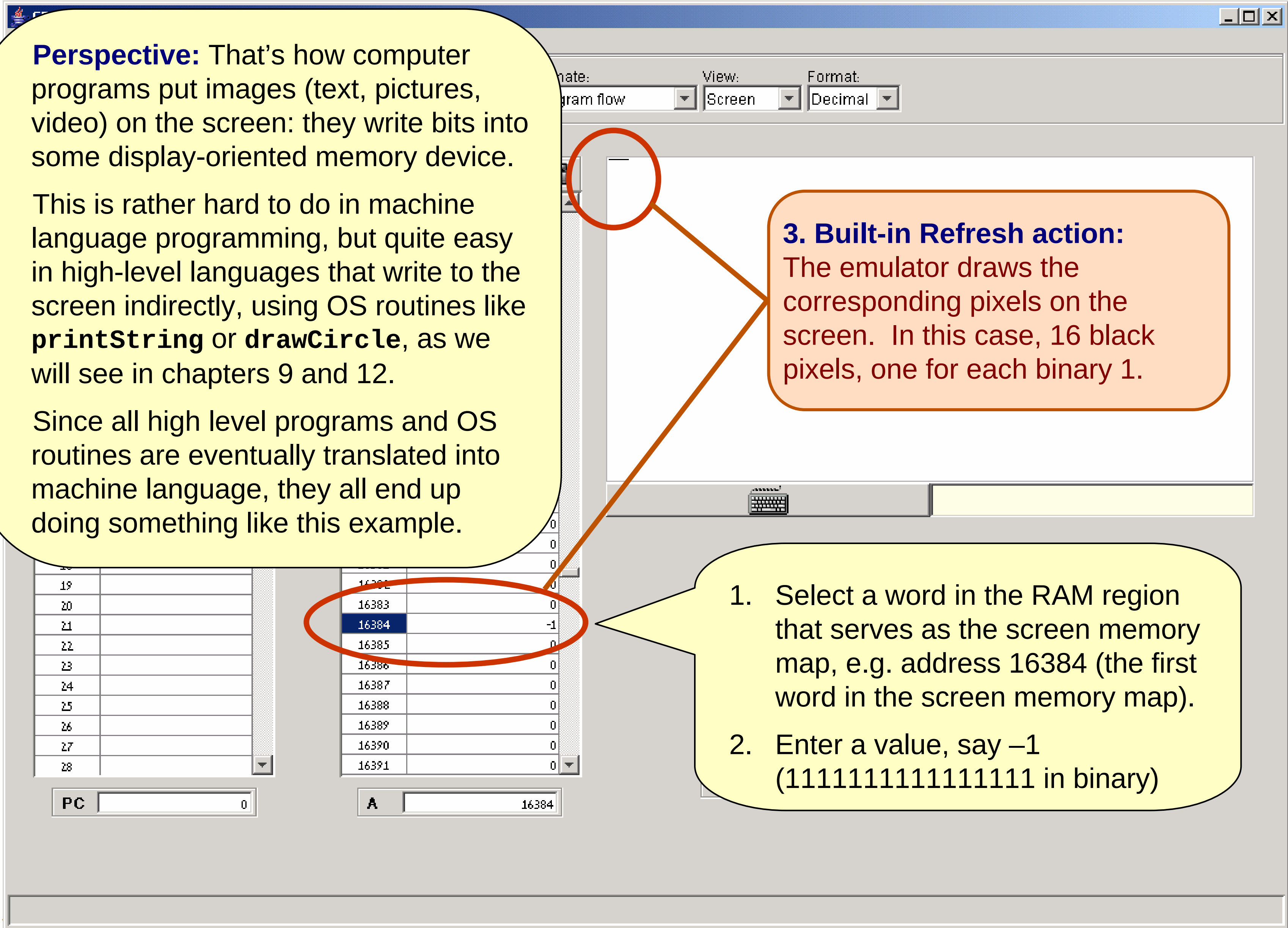
I/O devices: screen and keyboard



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.

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.

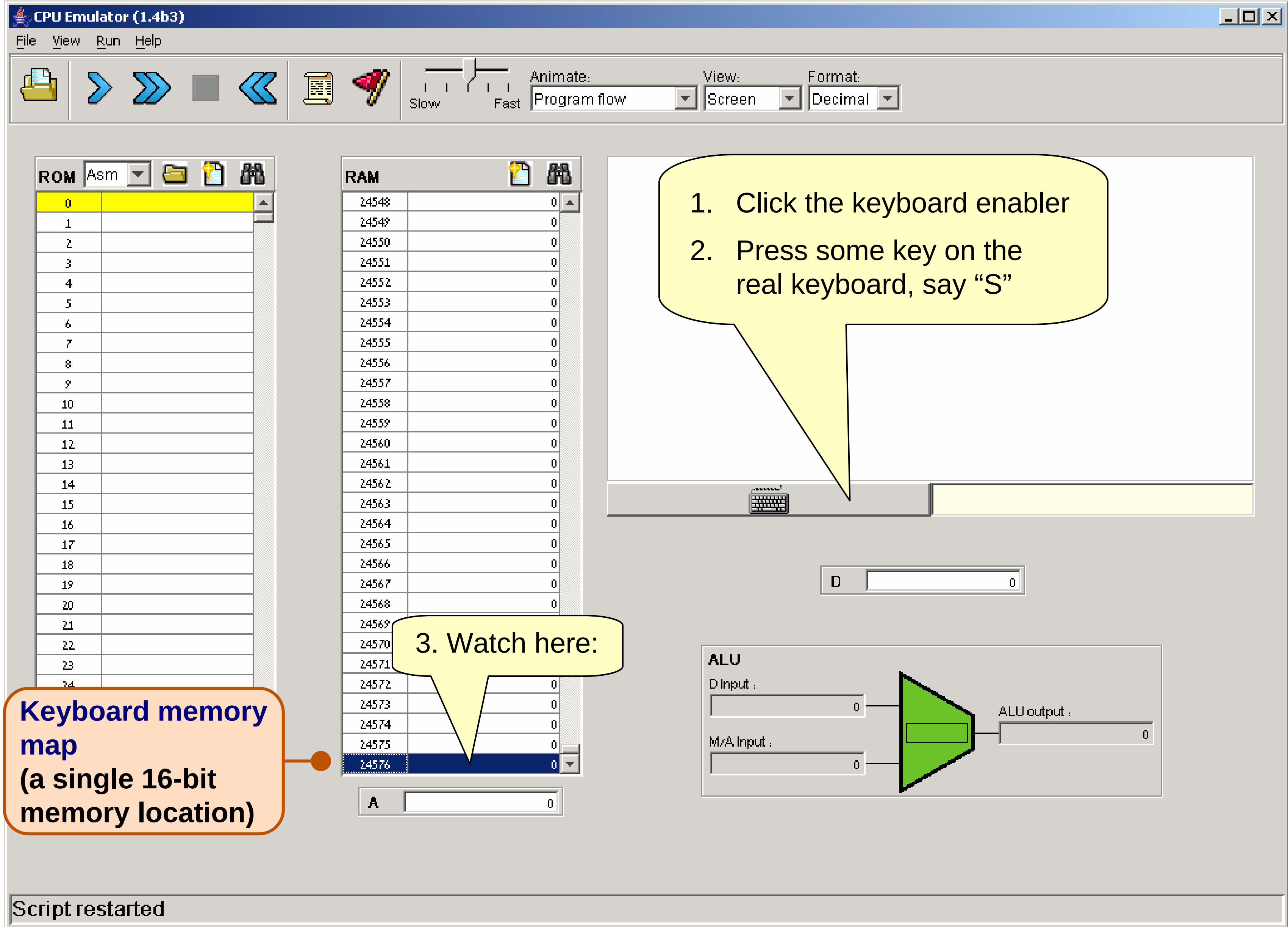
This is rather hard to do in machine language programming, but quite easy in high-level languages that write to the screen indirectly, using OS routines like **`printString`** or **`drawCircle`**, as we will see in chapters 9 and 12.

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

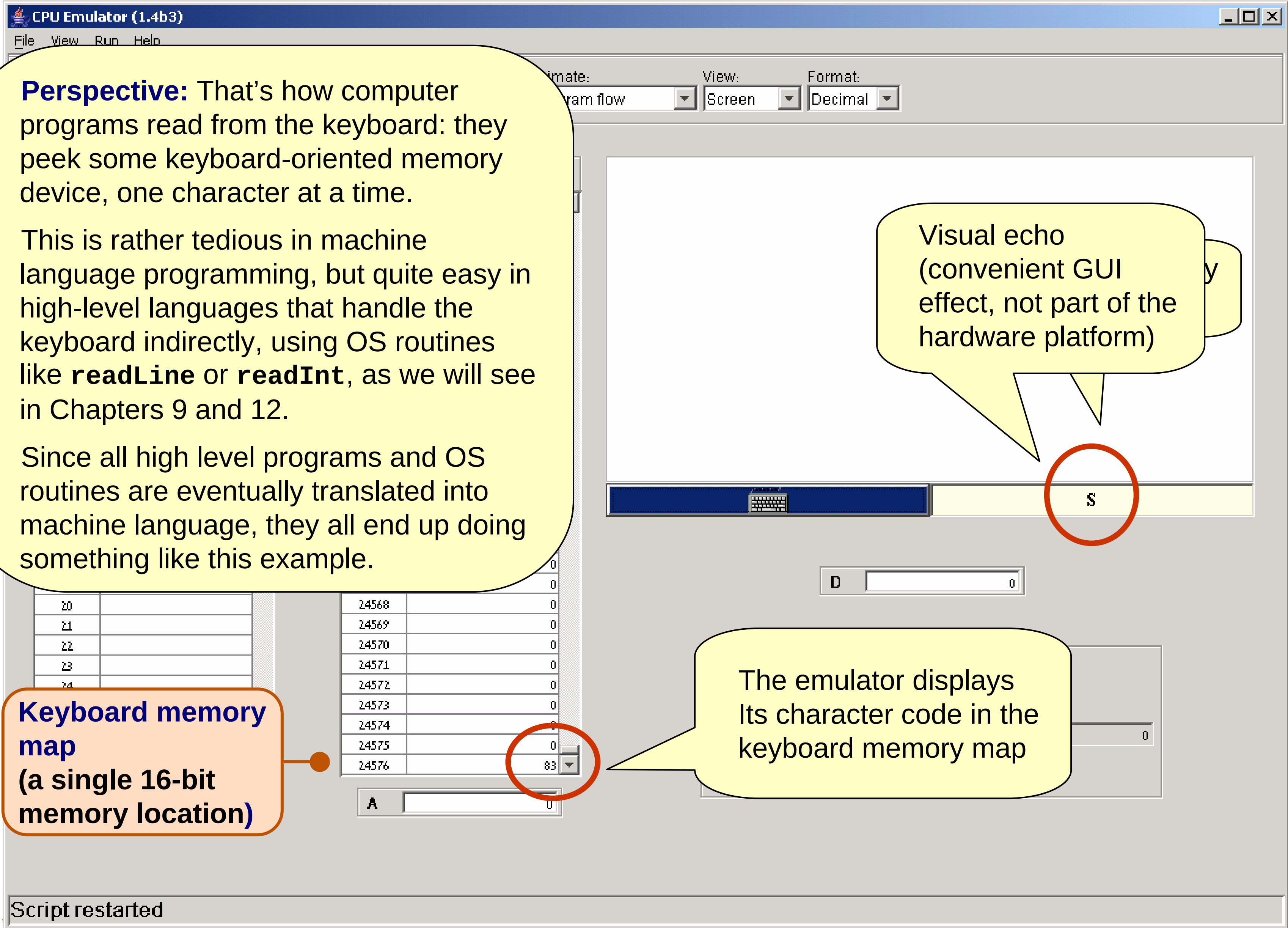
3. Built-in Refresh action: The emulator draws the corresponding pixels on the screen. In this case, 16 black pixels, one for each binary 1.

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



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.

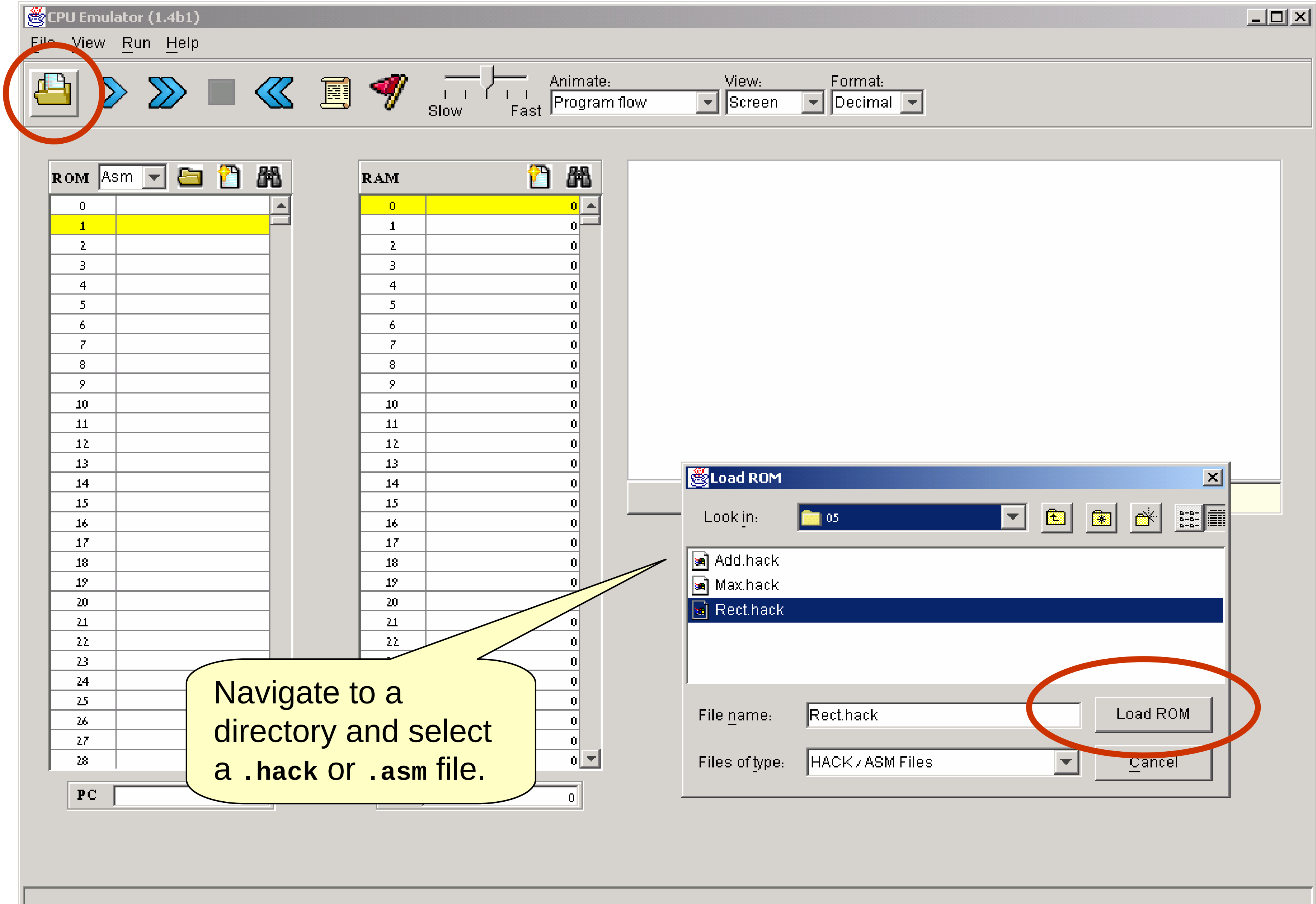
Visual echo (convenient GUI effect, not part of the hardware platform)

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

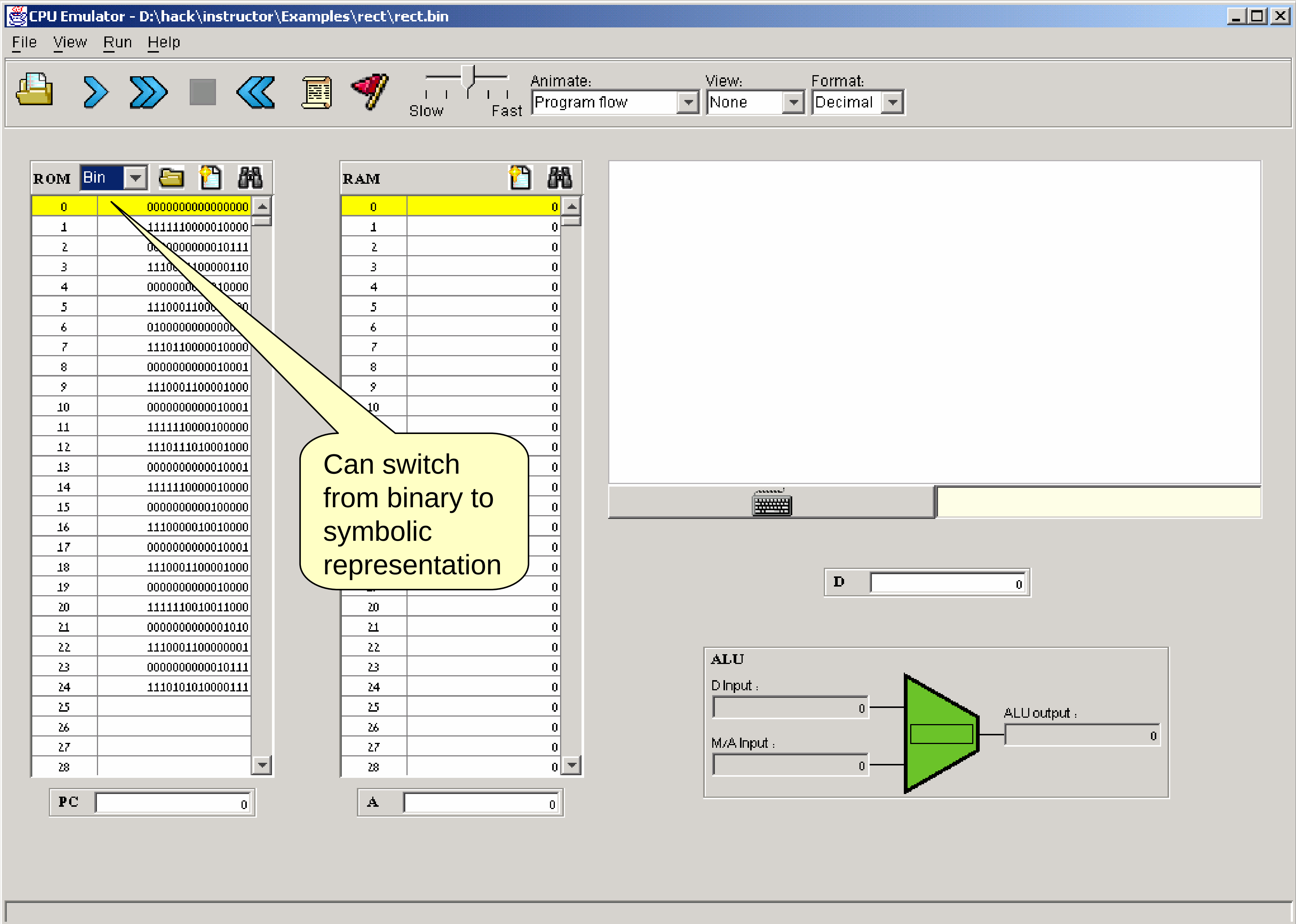
The emulator displays Its character code in the keyboard memory map

Part III:
Interactive
Simulation

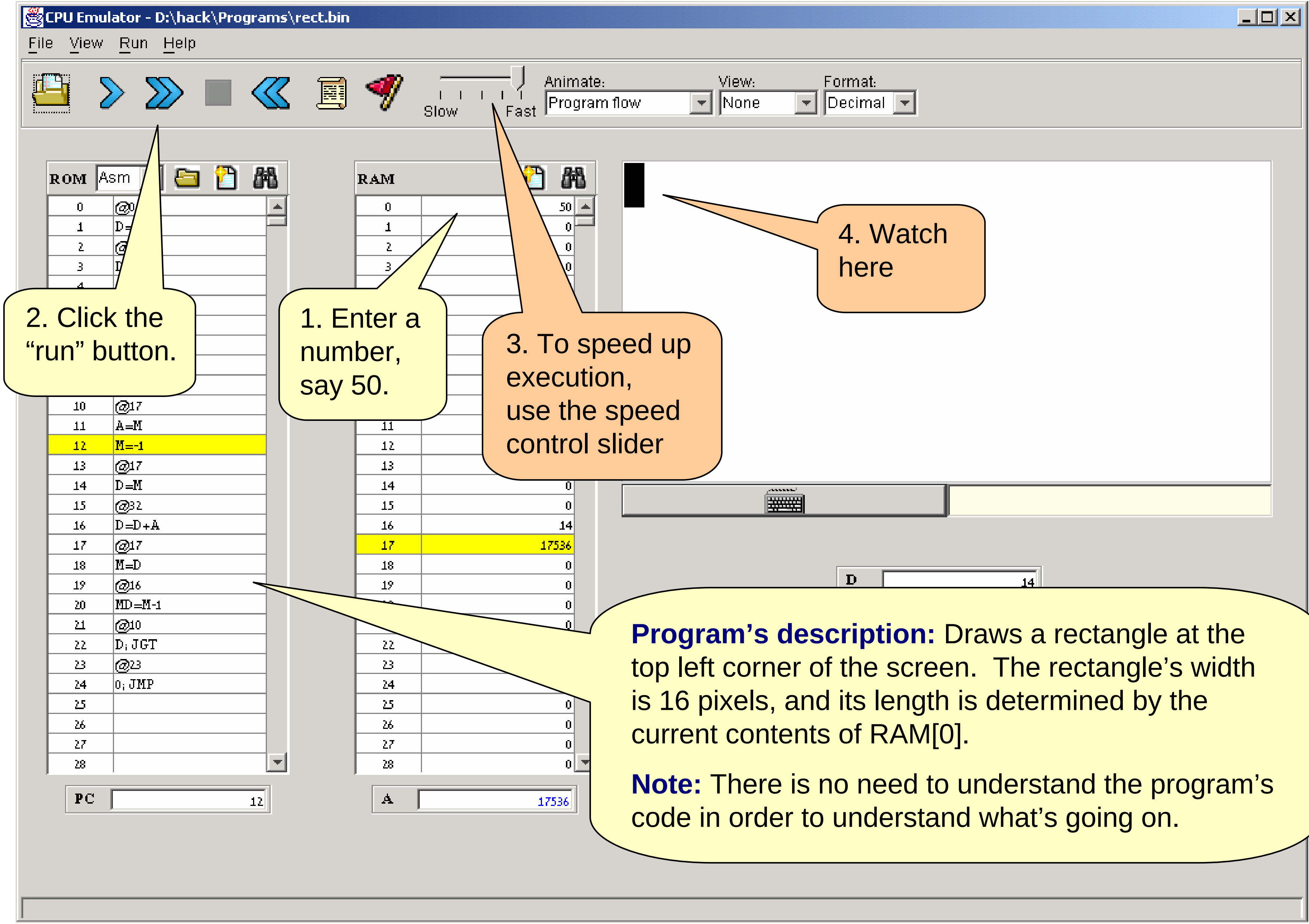
Loading a program



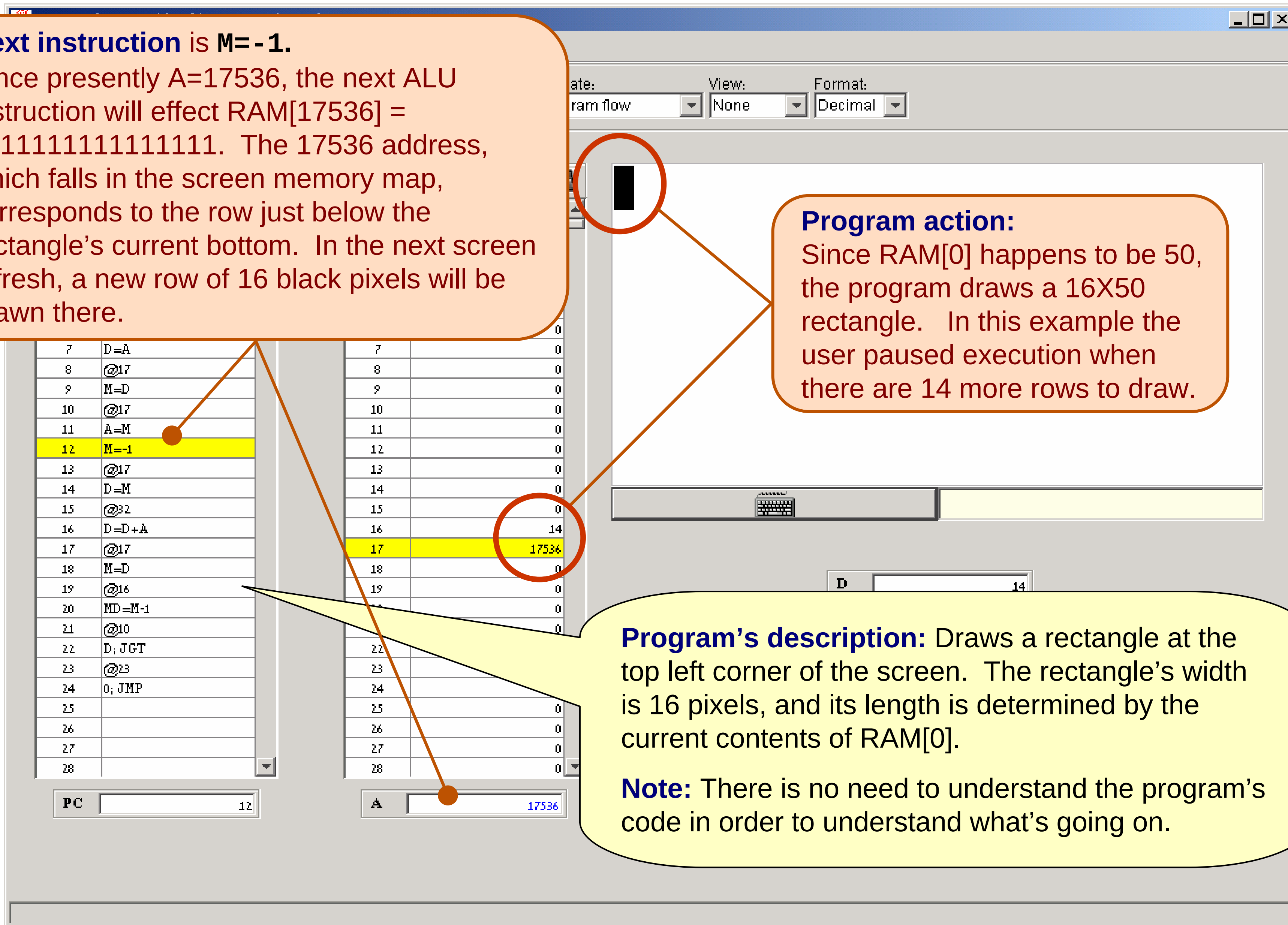
Loading a program



Running a program



Hack programming at a glance (optional)



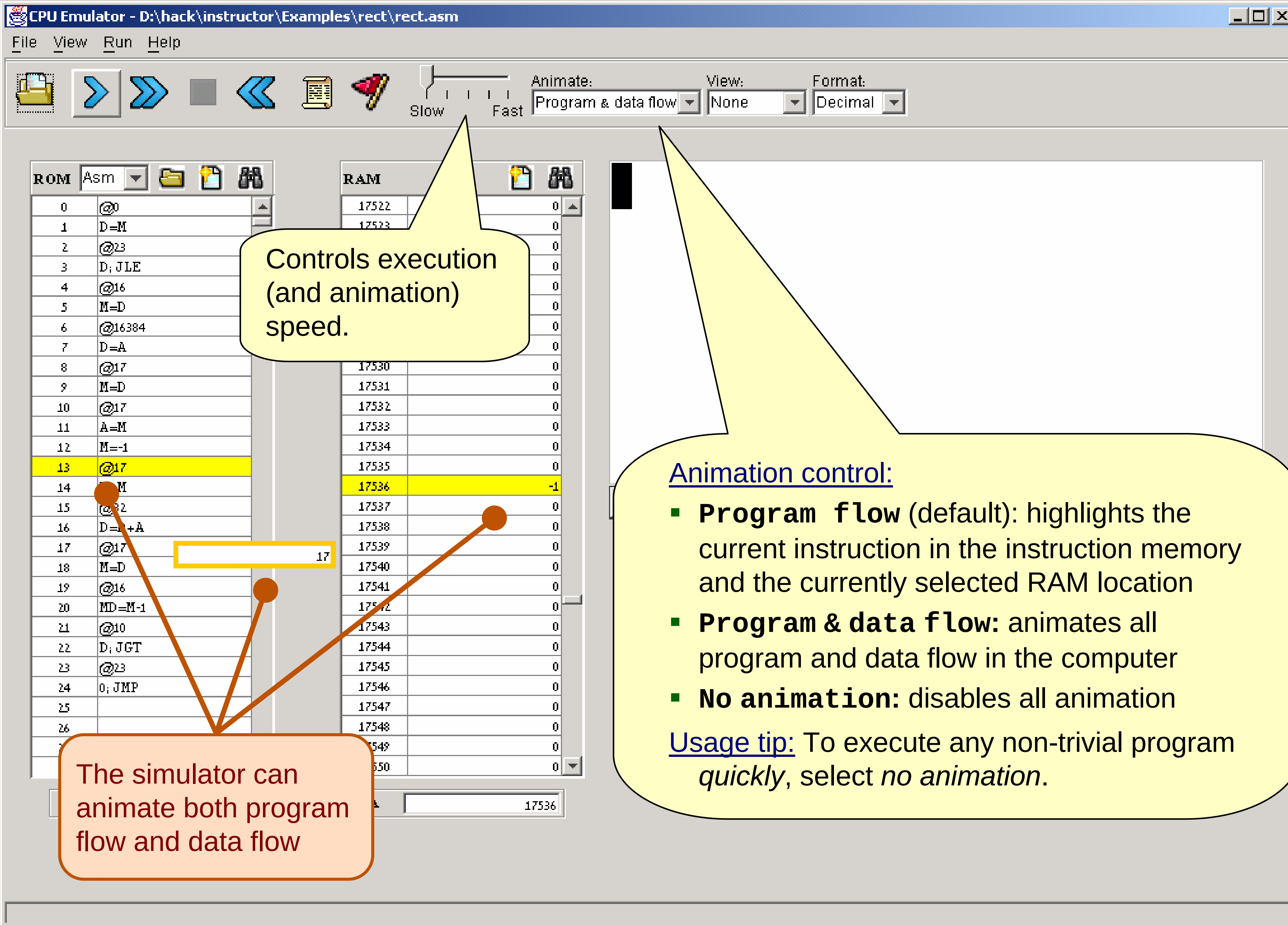
Next instruction is **`M=-1`.**
Since presently A=17536, the next ALU instruction will effect RAM[17536] = 1111111111111111. The 17536 address, which falls in the screen memory map, corresponds to the row just below the rectangle's current bottom. In the next screen refresh, a new row of 16 black pixels will be drawn there.

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



Controls execution (and animation) speed.

The simulator can animate both program flow and data flow

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*.

Part IV:

Script-Based Simulation

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*.

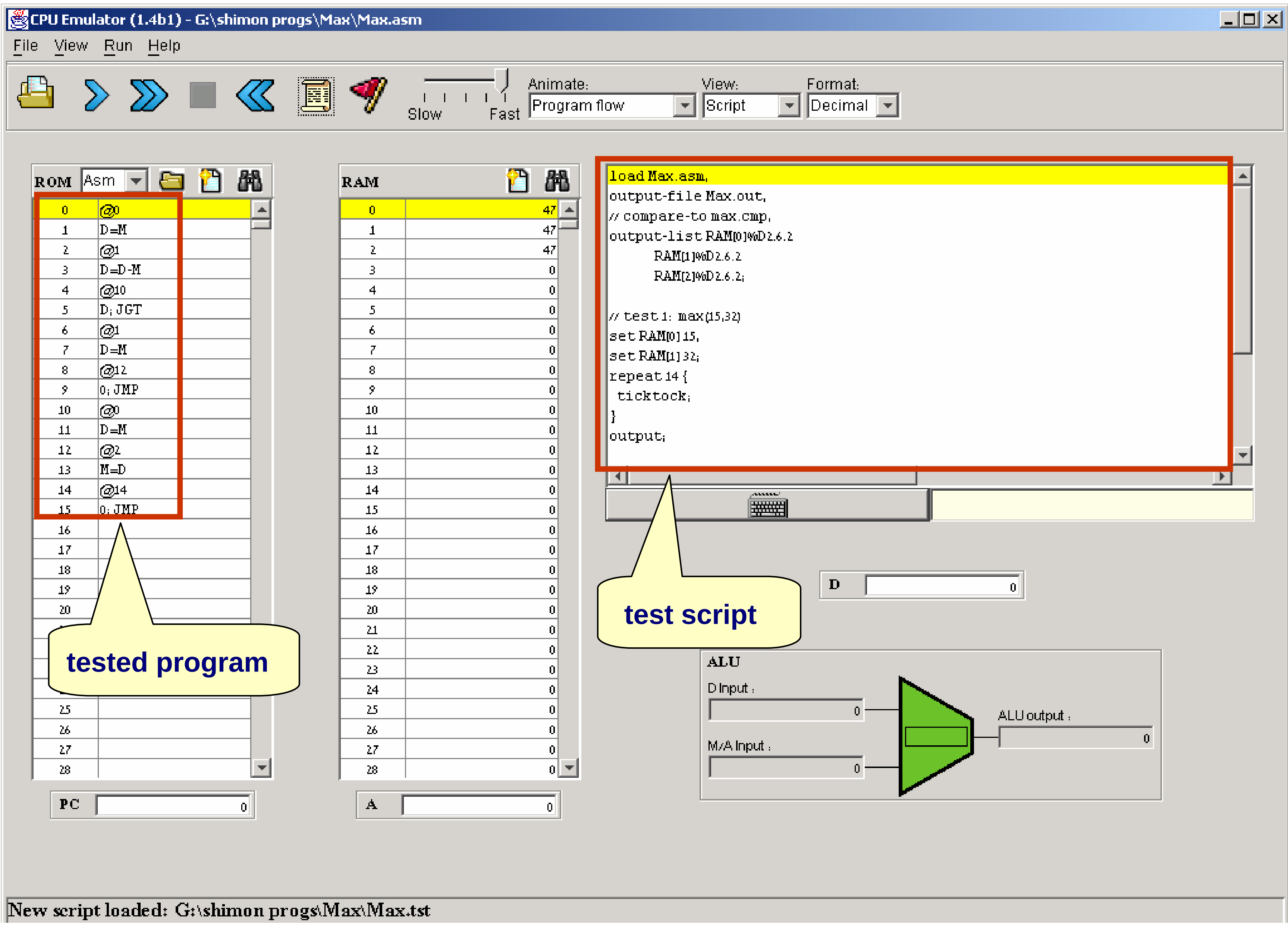
Script-based simulation enables planning and using tests that are:

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

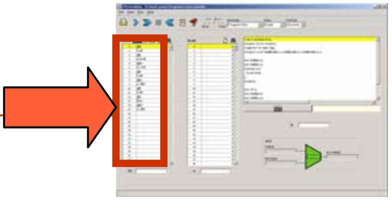
Test scripts:

- 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`**



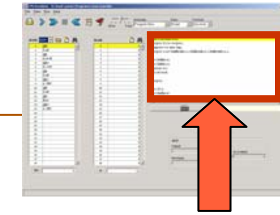
Note: For now, it is not necessary to understand either the Hack machine language or the Max program. It is only important to grasp the program's logic. But if you're interested, we give a language overview on the right.

```
// 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.
```

<u>The scripting language has commands for:</u>

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

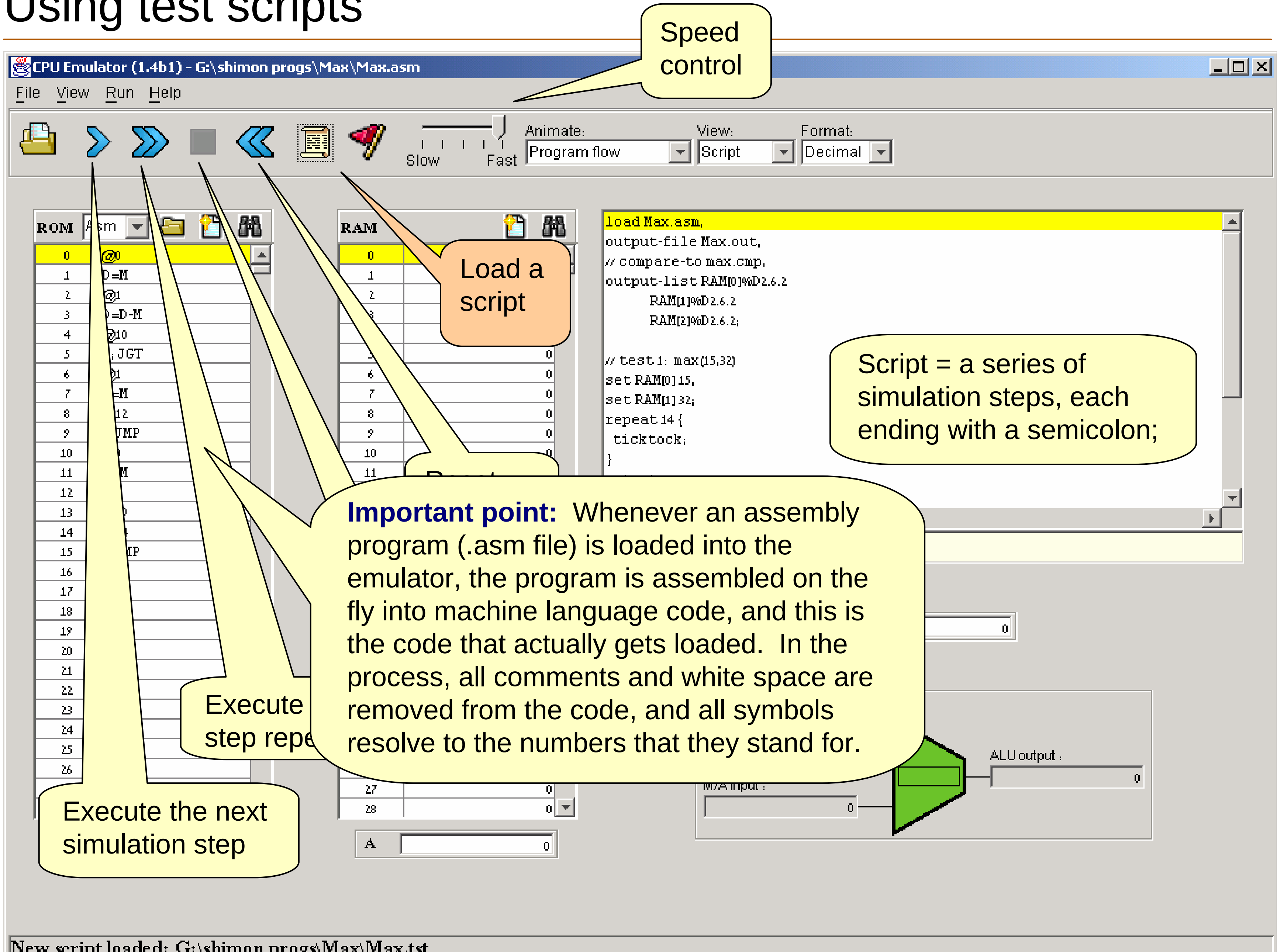
<u>Notes:</u>

- 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.

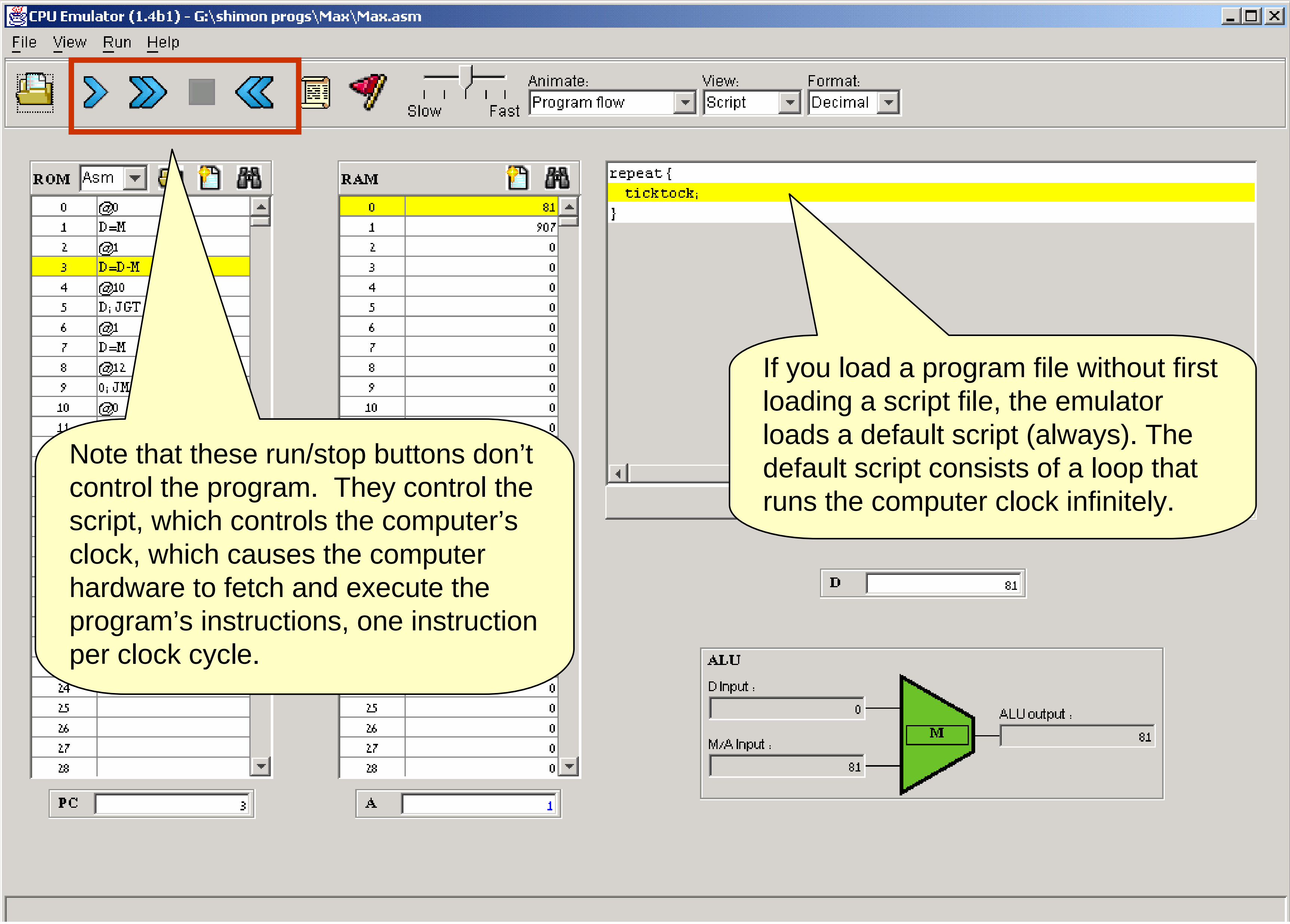
Output

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

Using test scripts



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



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.

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.


Part V:
Debugging

Breakpoints: a powerful debugging tool

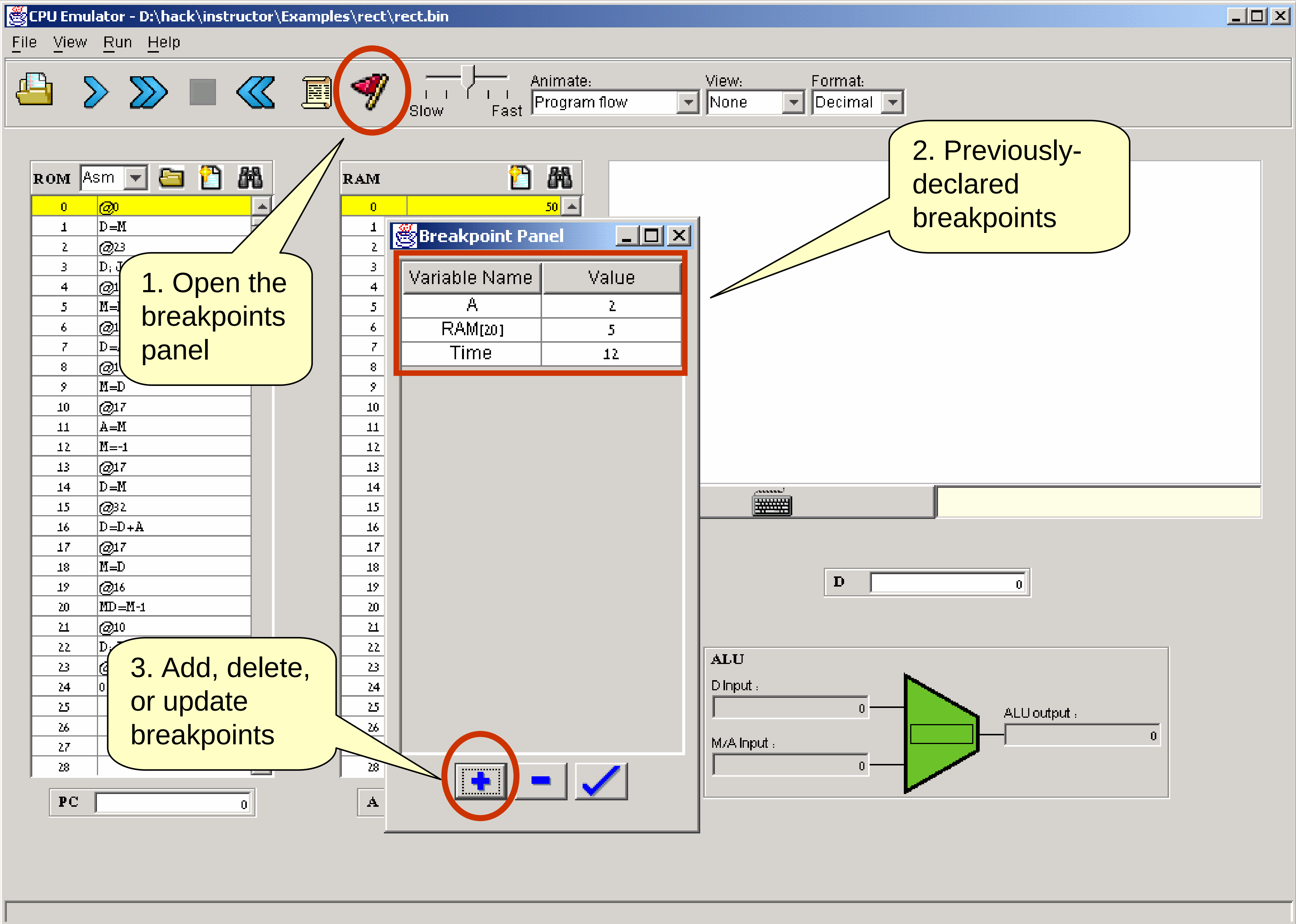
The CPU emulator continuously keeps track of:

- **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

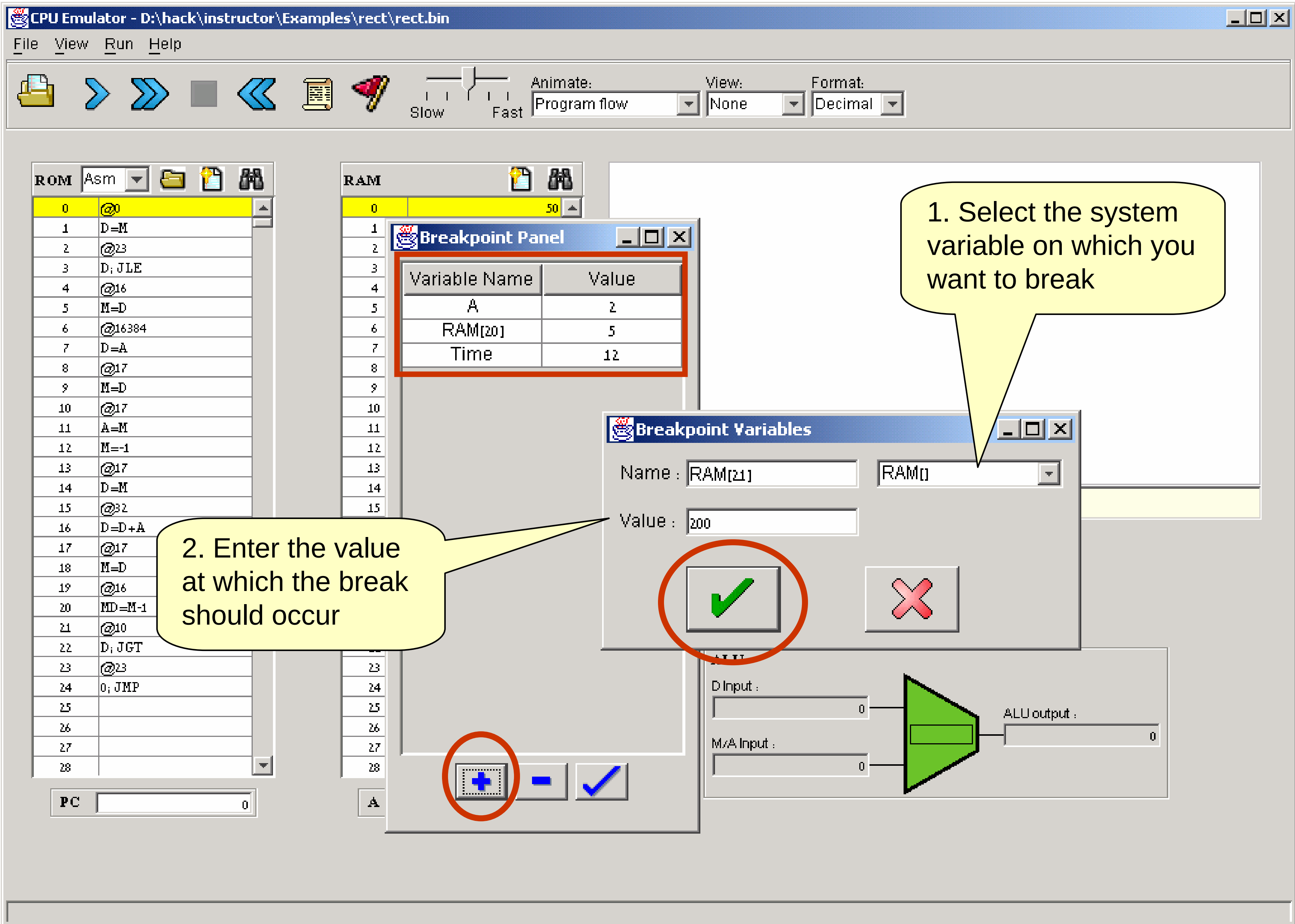
Breakpoints:

- 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.

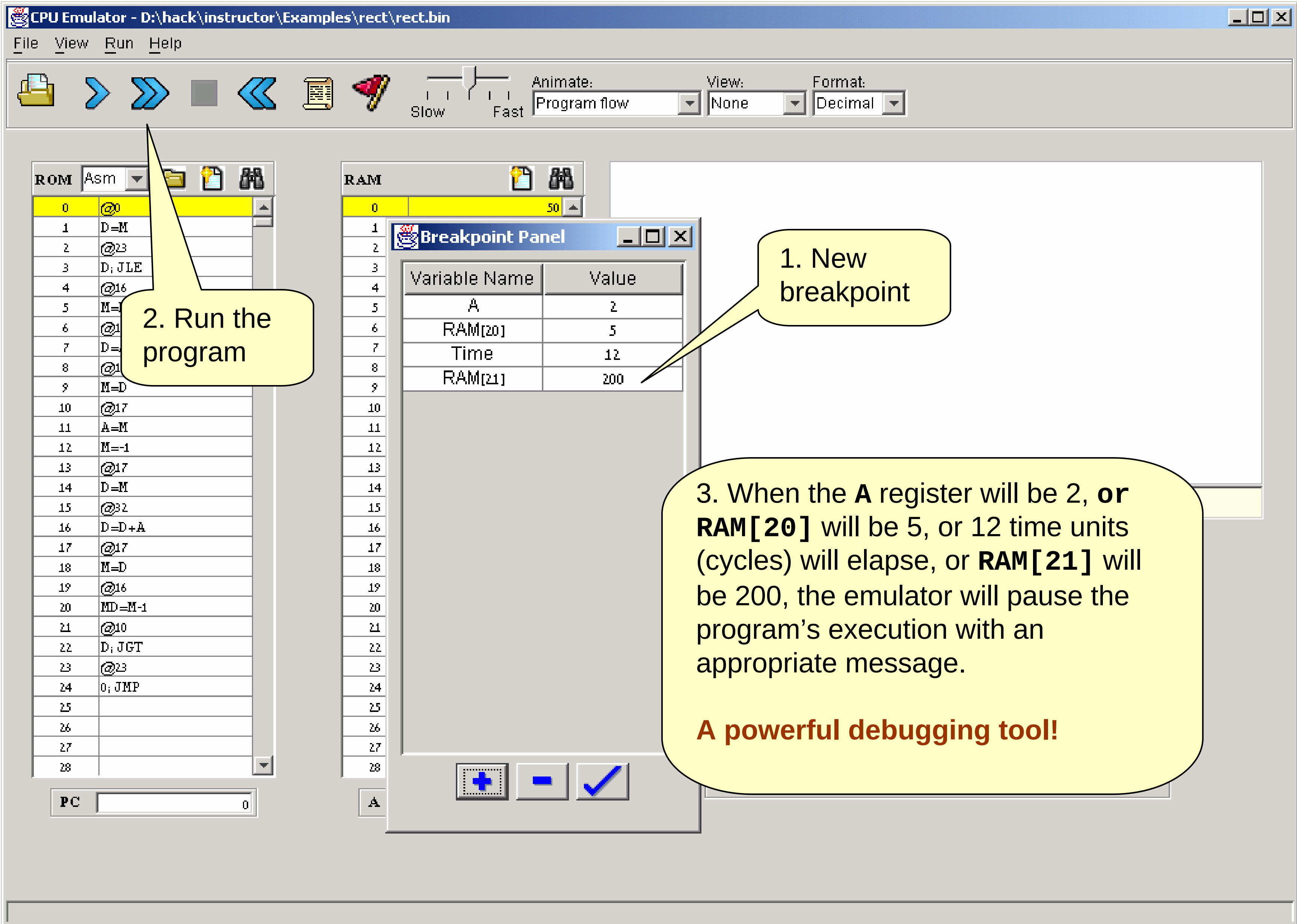
Breakpoints declaration



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).