Hardware Simulator 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

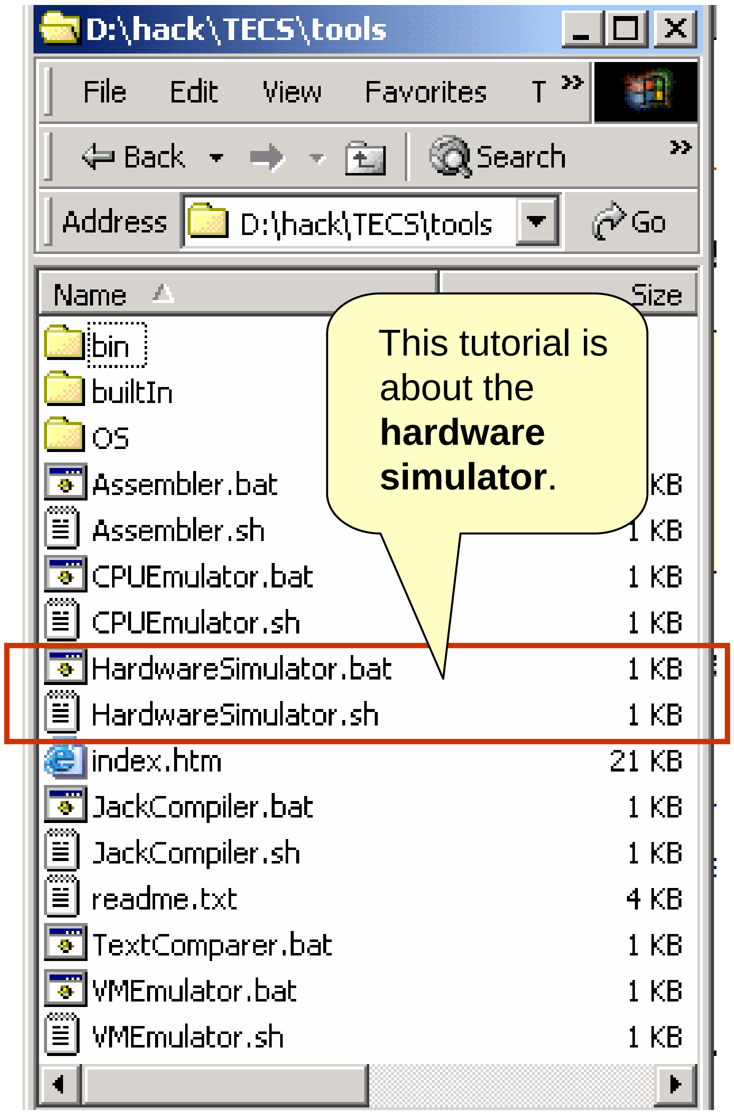
The Elements of Computing Systems evolves around the construction of a complete computer system, done in the framework of a 1- or 2-semester course.

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.

The Hack computer

The hardware simulator described in this tutorial can be used to build and test many different hardware platforms. In this book, we focus on one particular computer, called Hack.

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

The first 5 chapters of the book specify the elementary gates, combinational chips, sequential chips, and hardware architecture of the Hack computer.

All these modules can be built and tested using the hardware simulator described in this tutorial.

That is how hardware engineers build chips for real: first, the hardware s designed, tested, and optimized on a software simulator. Only then, the resulting gate logic is committed to silicon.

Hardware Simulation Tutorial

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

- Chapter 1: *Boolean Logic*
- Appendix A: *Hardware Description Language*
- Appendix B: *Test Scripting Language*


Part I:
Getting Started

Chip Definition (**`.hdl`** file)

chip interface

```
/** Exclusive-or gate. out = a xor b */
CHIP Xor {
    IN a, b;
    OUT out;

    // Implementation missing.
}
```

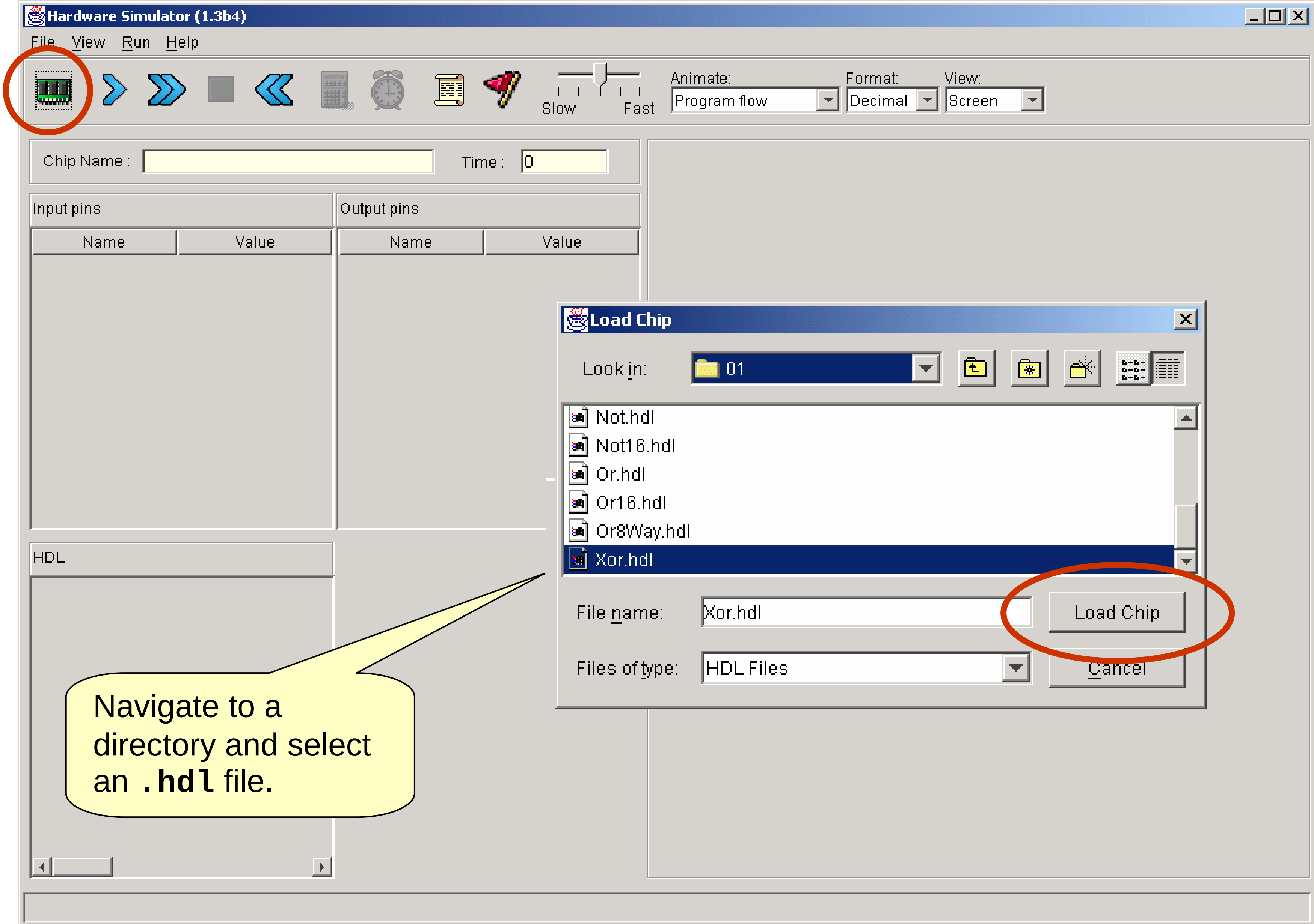
- <u>Chip interface:</u>
  - Name of the chip
  - Names of its input and output pins
  - Documentation of the intended chip operation
- Typically supplied by the chip architect; similar to an API, or a contract.

Chip Definition (**.hdl** file)

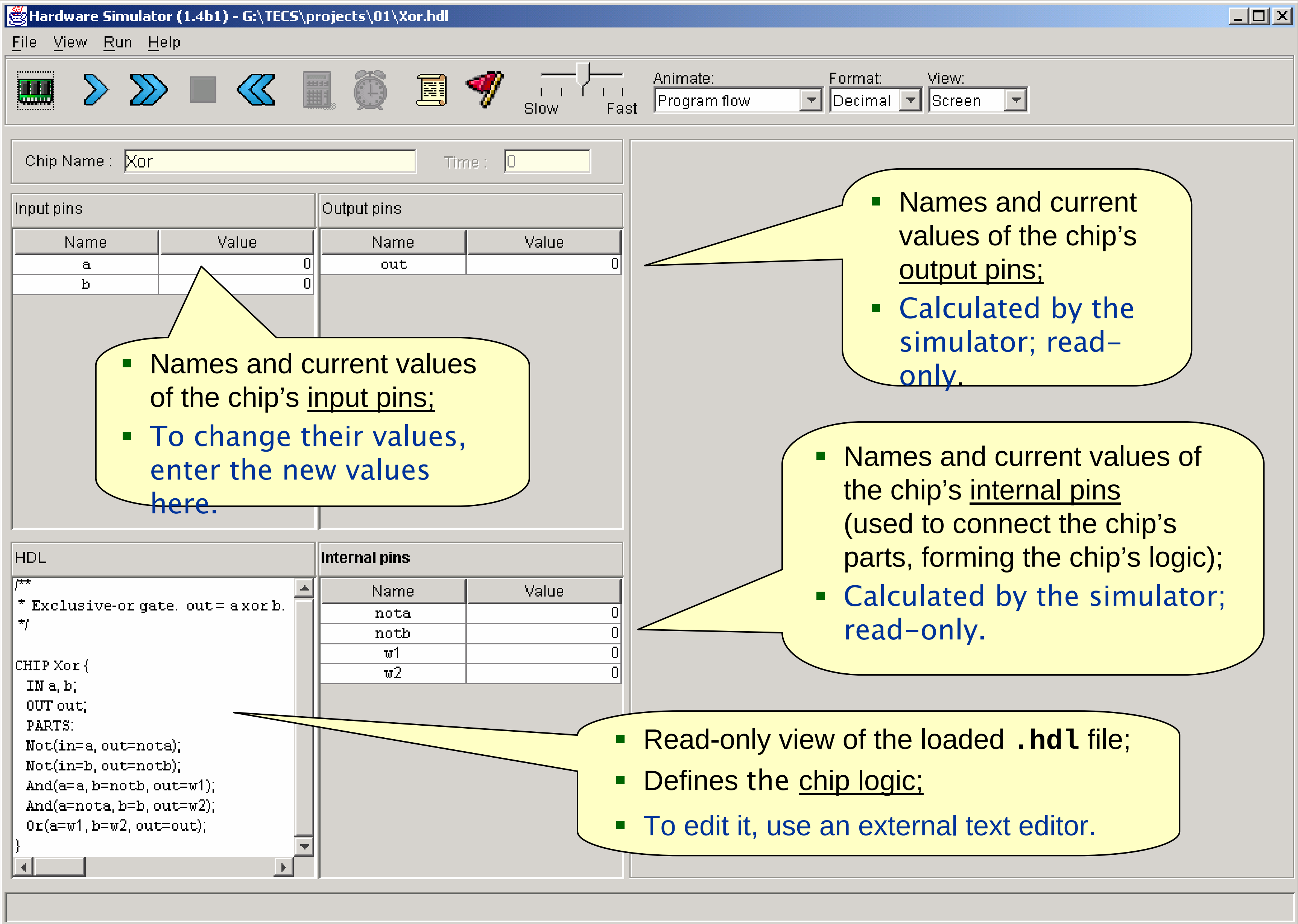

```
/** Exclusive-or gate. out = a xor b */
CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    Not(in=a, out=nota);
    Not(in=b, out=notb);
    And(a=a, b=notb, out=w1);
    And(a=nota, b=b, out=w2);
    Or(a=w1, b=w2, out=out);
}
```

- Any given chip can be implemented in several different ways. This particular implementation is based on: Xor(a,b) = Or(And(a,Not(b)), And(b,Not(a)))
- **`Not`**, **`And`**, **`Or`**: *Internal parts* (previously built chips), invoked by the HDL programmer
- **`nota`**, **`notb`**, **`w1`**, **`w2`**: *internal pins*, created and named by the HDL programmer; used to connect internal parts.

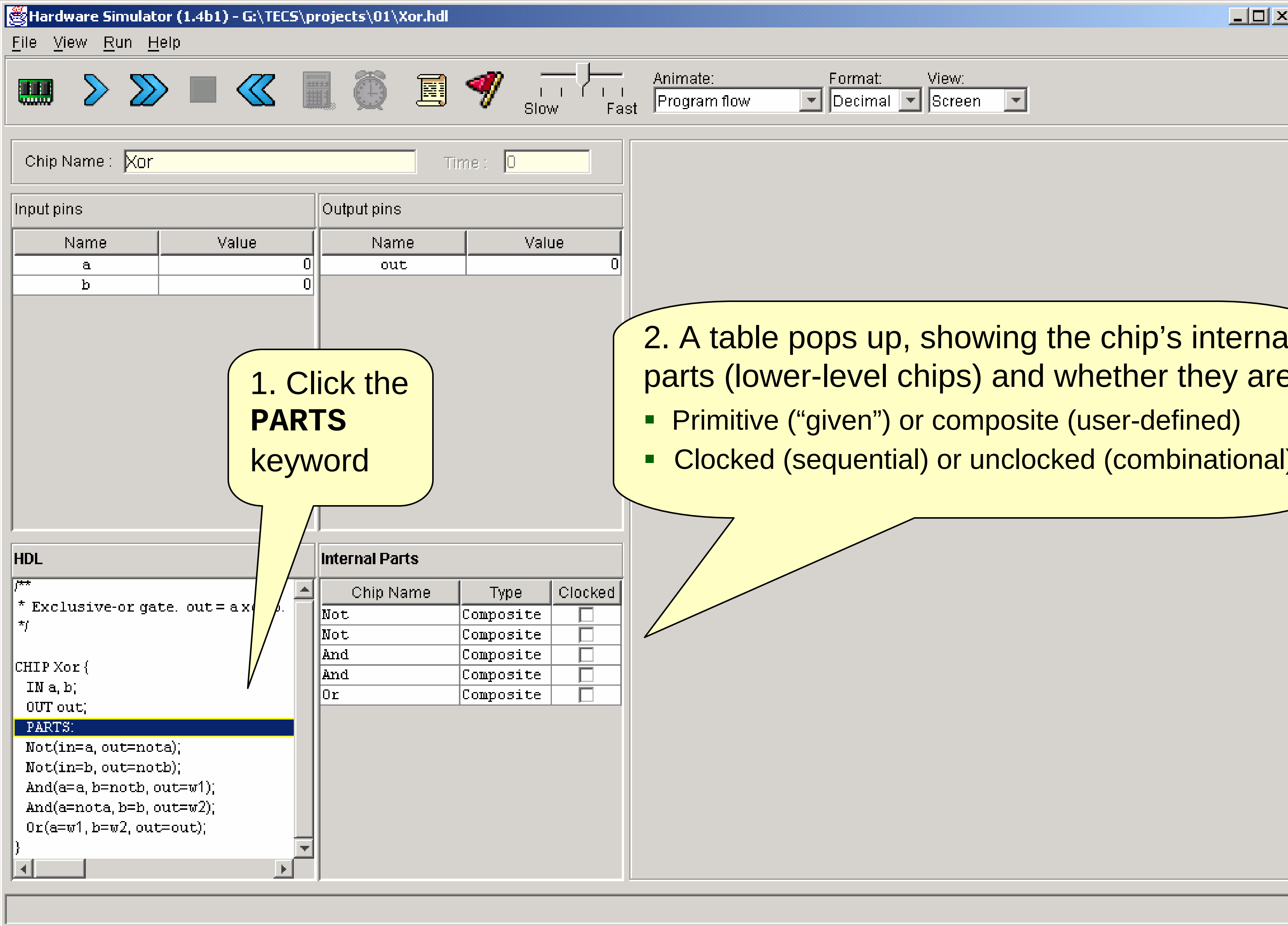
Loading a Chip



Loading a Chip



Exploring the Chip Logic

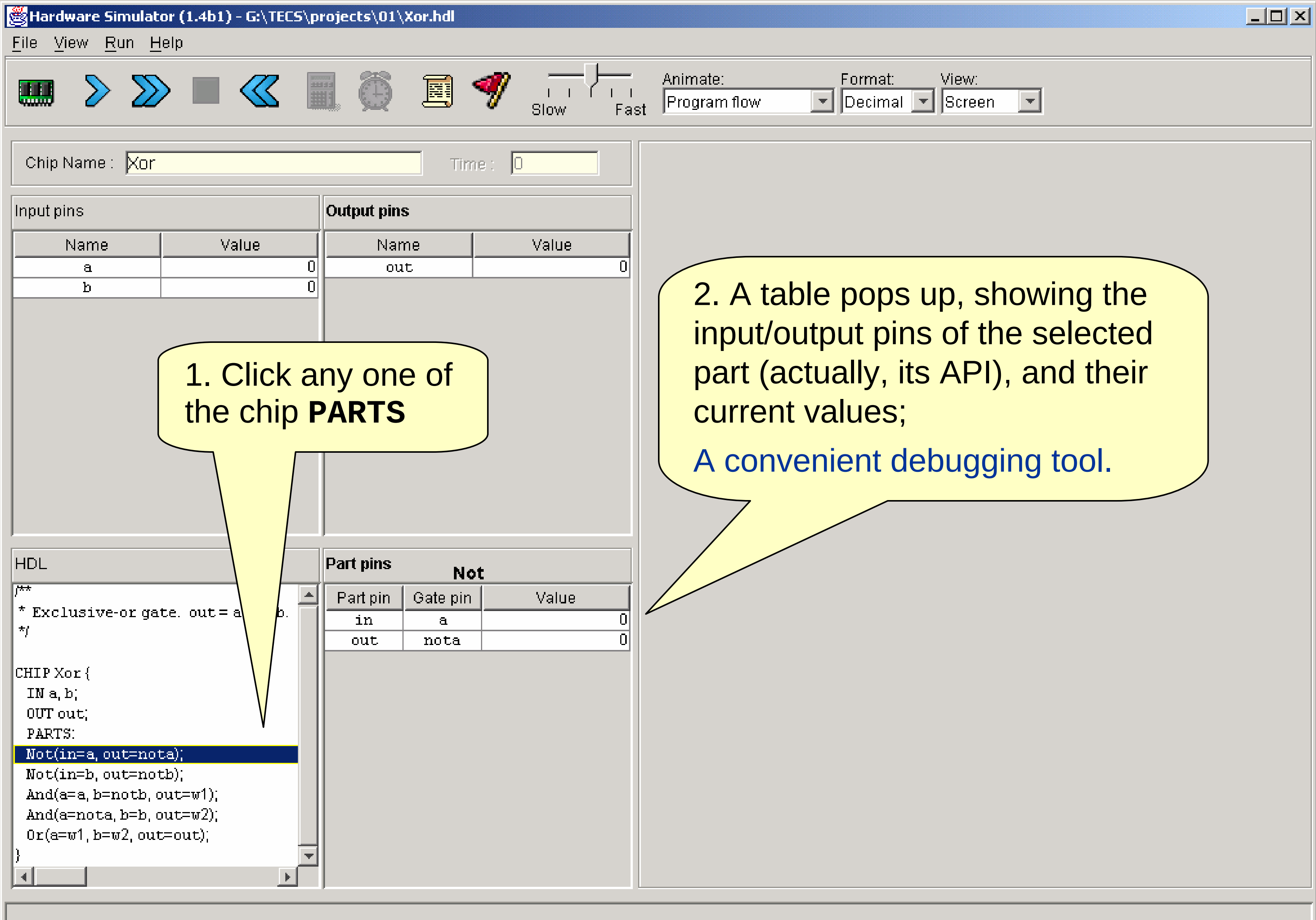


1. Click the **PARTS** keyword

2. A table pops up, showing the chip's internal parts (lower-level chips) and whether they are:

- Primitive ("given") or composite (user-defined)
- Clocked (sequential) or unclocked (combinational)

Exploring the Chip Logic

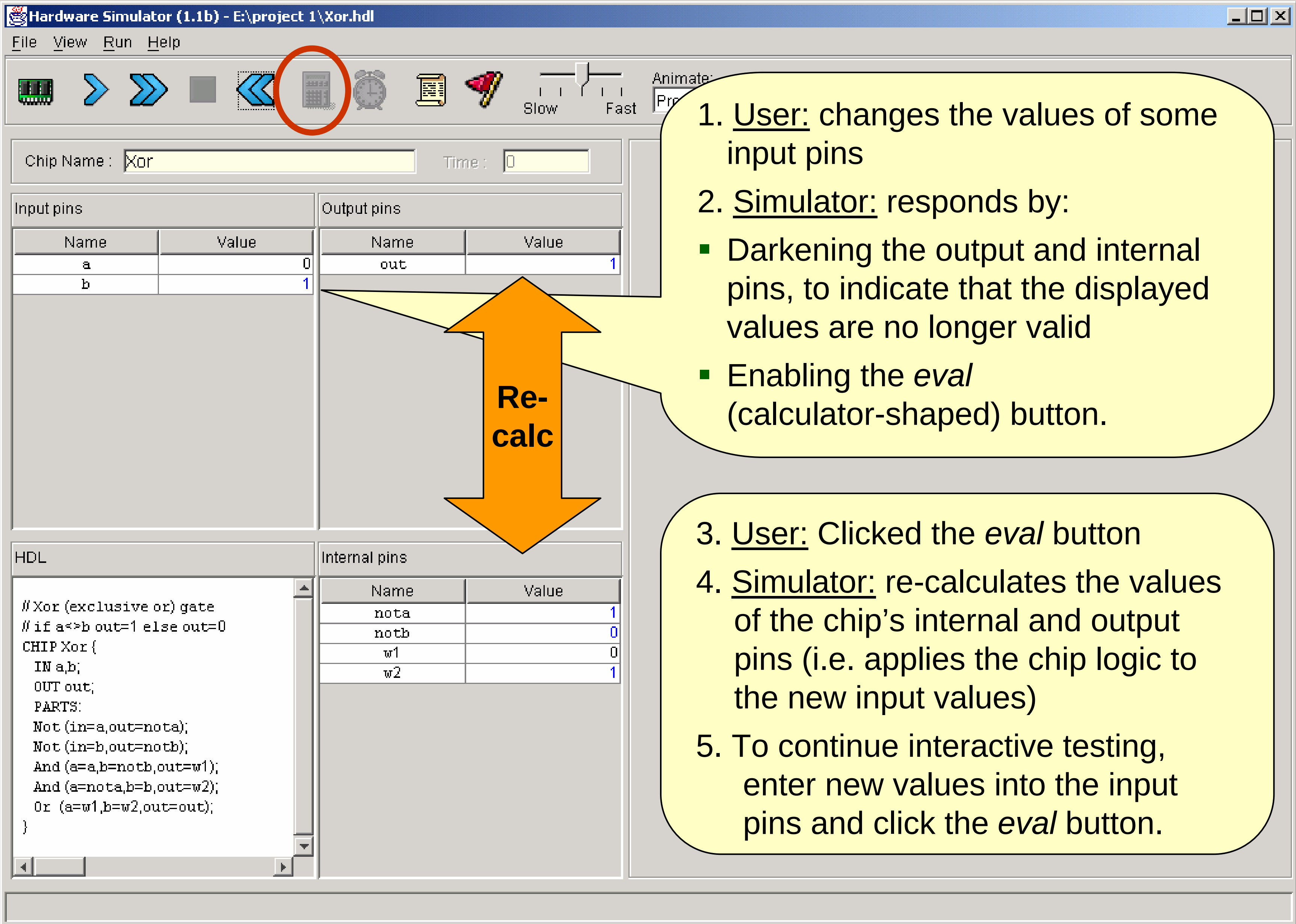


1. Click any one of the chip **PARTS**

2. A table pops up, showing the input/output pins of the selected part (actually, its API), and their current values;

A convenient debugging tool.

Interactive Chip Testing



1. User: changes the values of some input pins
2. Simulator: responds by:
   - Darkening the output and internal pins, to indicate that the displayed values are no longer valid
   - Enabling the *eval* (calculator-shaped) button.
3. User: Clicked the *eval* button
4. Simulator: re-calculates the values of the chip's internal and output pins (i.e. applies the chip logic to the new input values)
5. To continue interactive testing, enter new values into the input pins and click the *eval* button.


Part II:
Test Scripts

Test Scripts

```
load Xor.hdl,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3
            b%B3.1.3
            out%B3.1.3;
set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;
Etc.
```

Generated output file (`Xor.out`)

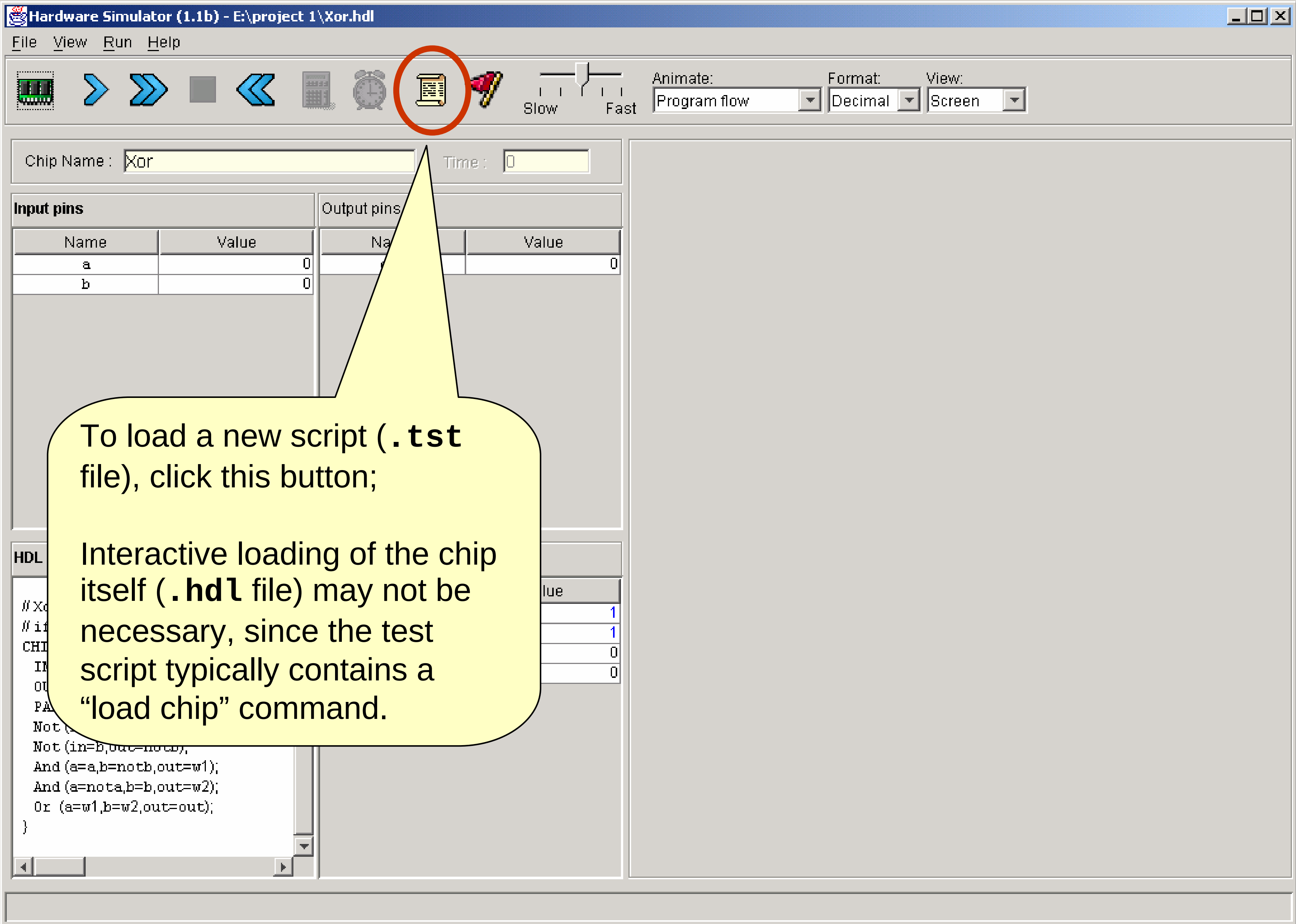
```
|   a   |   b   |  out  |
|   0   |   0   |   0   |
|   0   |   1   |   1   |
|   1   |   0   |   1   |
|   1   |   1   |   0   |
```

Test scripts:

- Are used for specifying, automating and replicating chip testing
- Are supplied for every chip mentioned in the book (so you don't have to write them)
- Can effect, batch-style, any operation that can be done interactively
- Are written in a simple language described in Appendix B of the book
- Can create an output file that records the results of the chip test

- If the script specifies a compare file, the simulator will compare the **.out** file to the **.cmp** file, line by line.

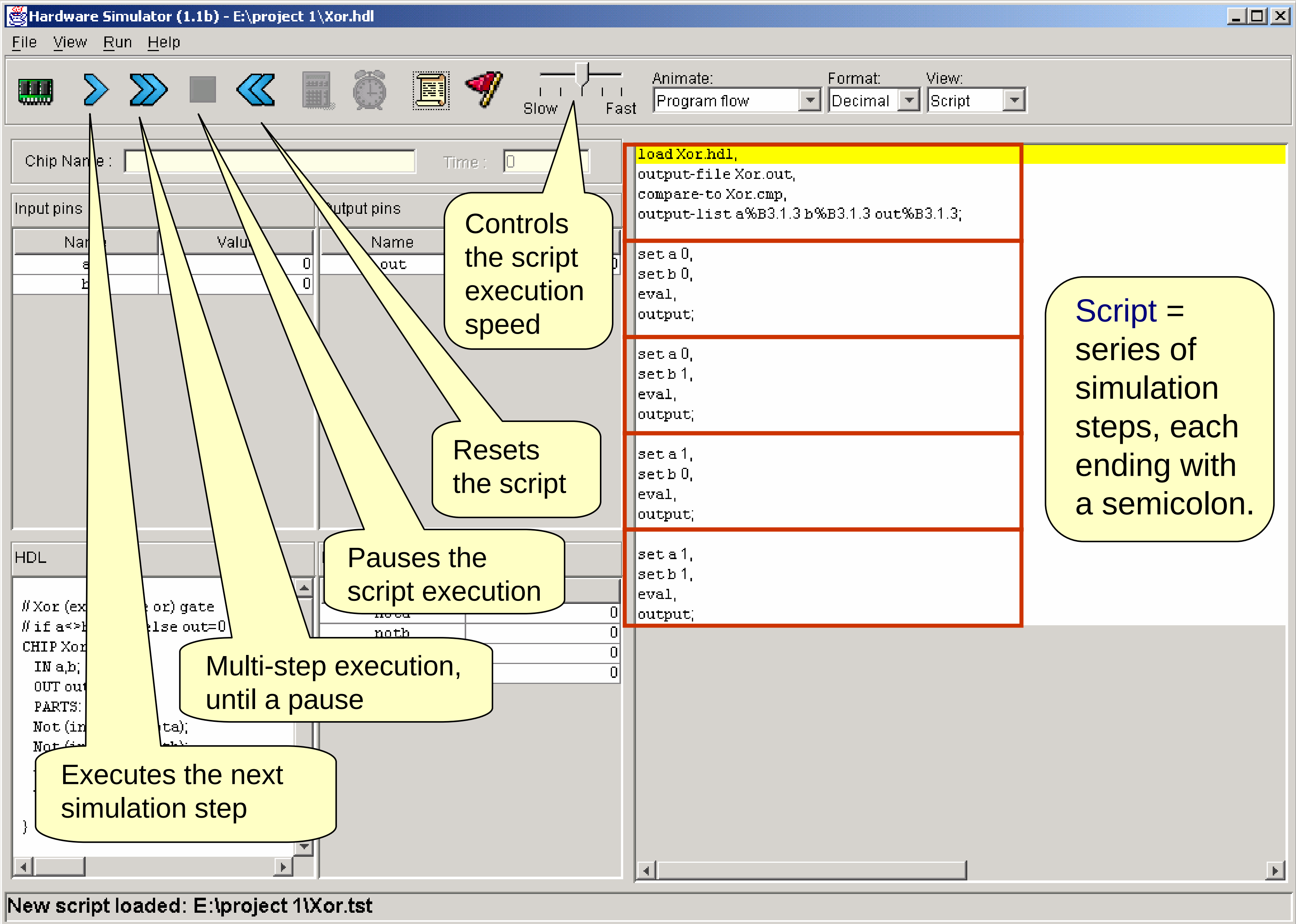
Loading a Script



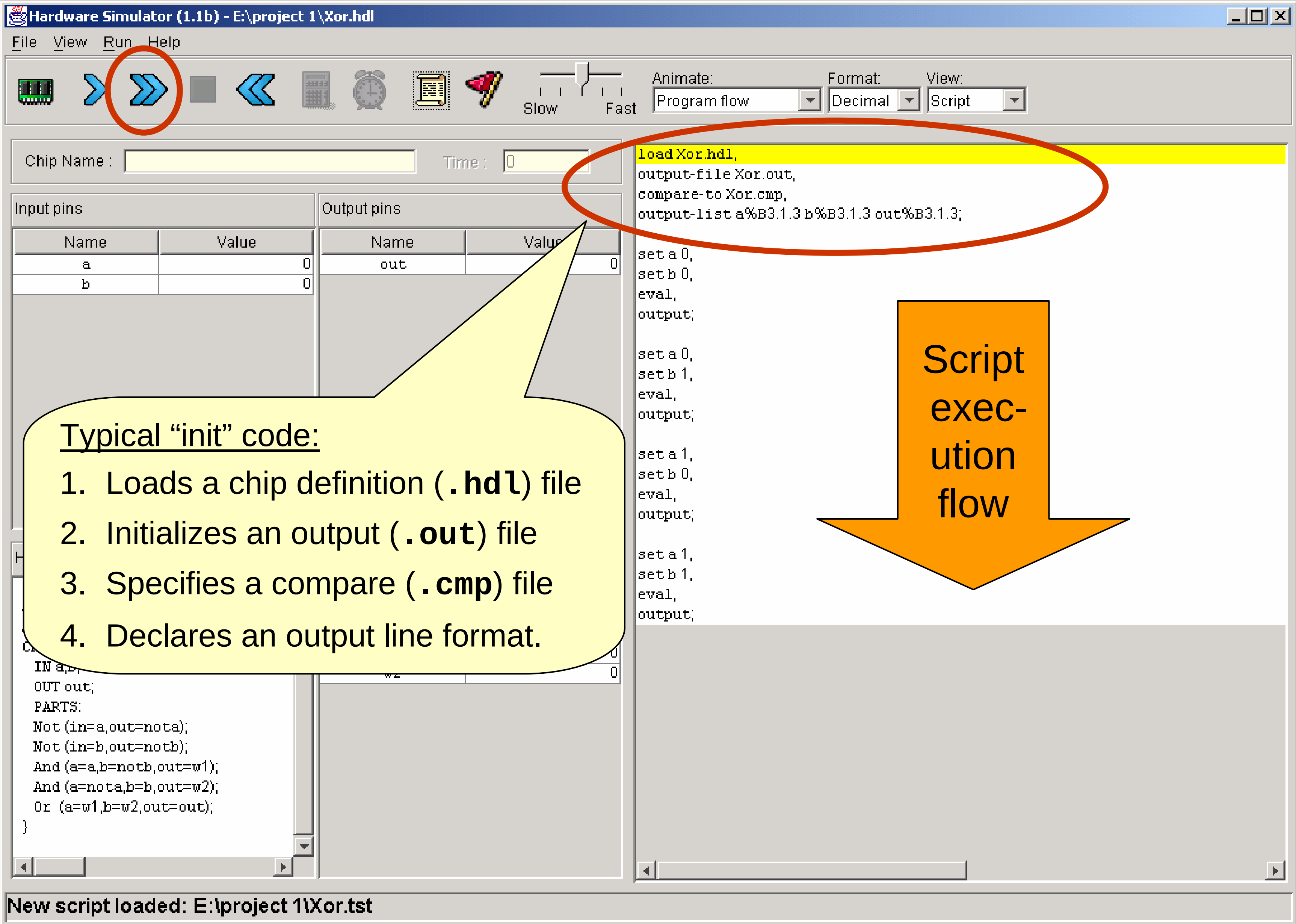
To load a new script (**`.tst`** file), click this button;

Interactive loading of the chip itself (**`.hdl`** file) may not be necessary, since the test script typically contains a "load chip" command.

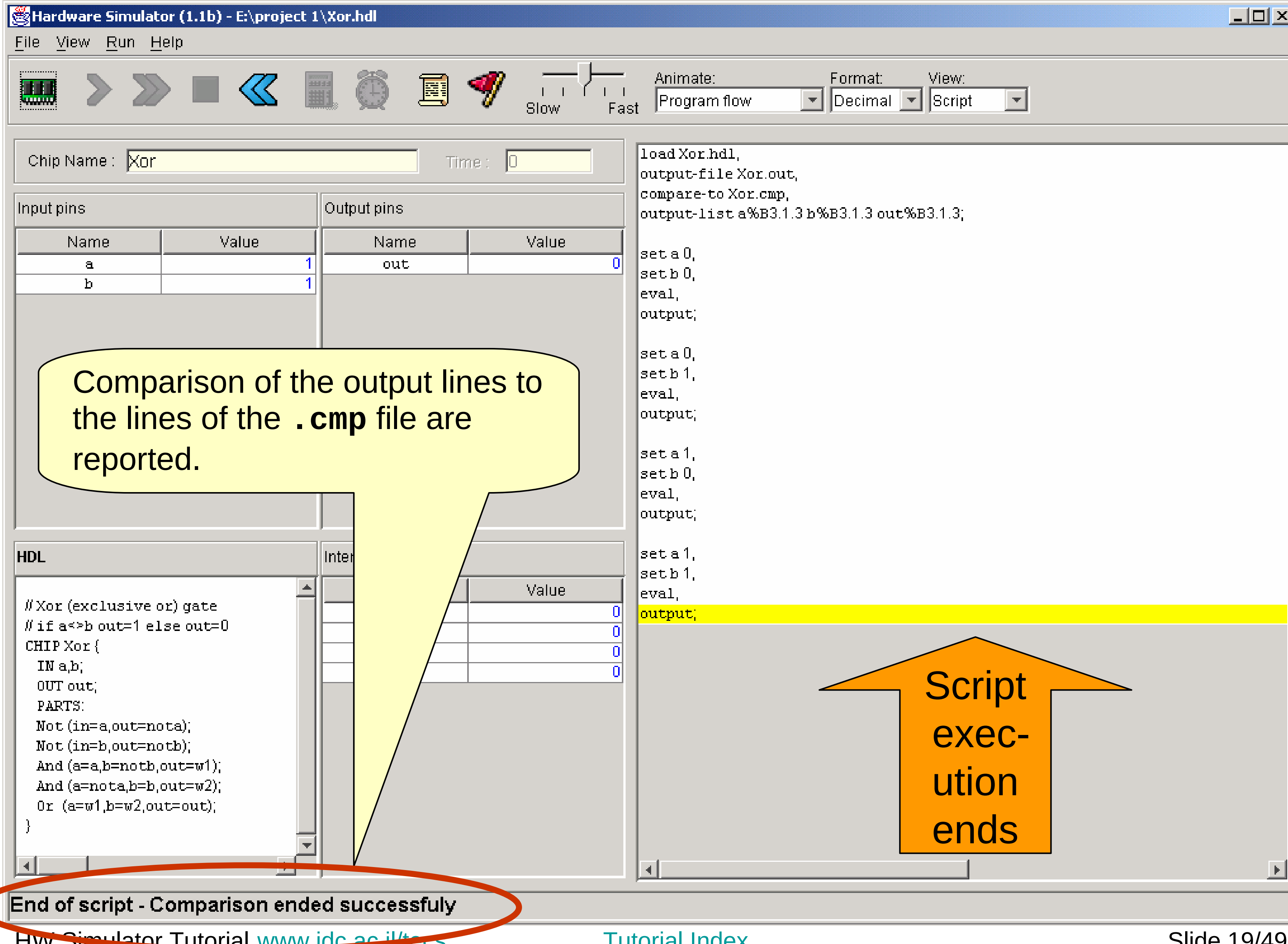
Script Controls



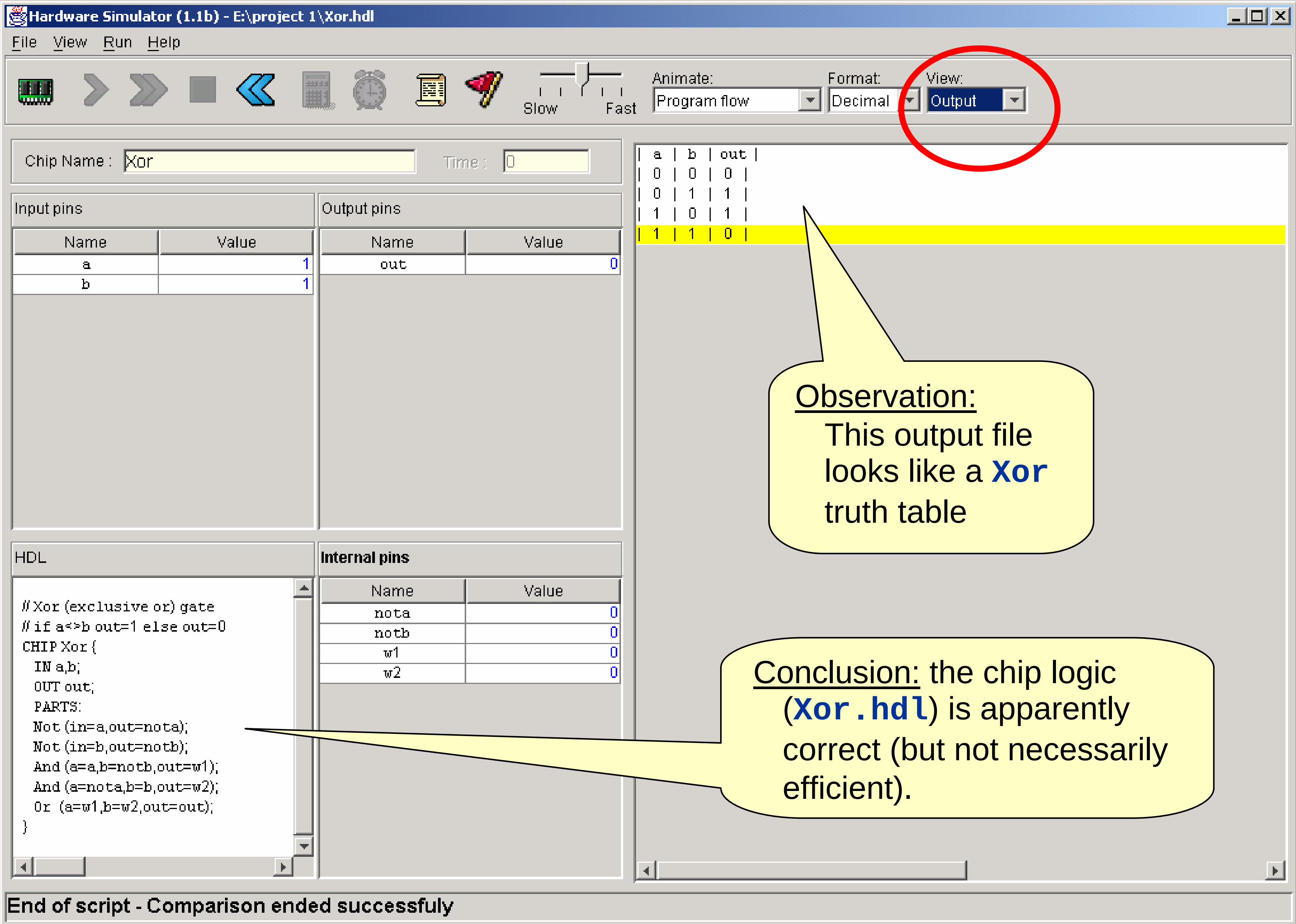
Running a Script



Running a Script



Viewing Output and Compare Files




Part III:
Built-in Chips

Built-In Chips

General

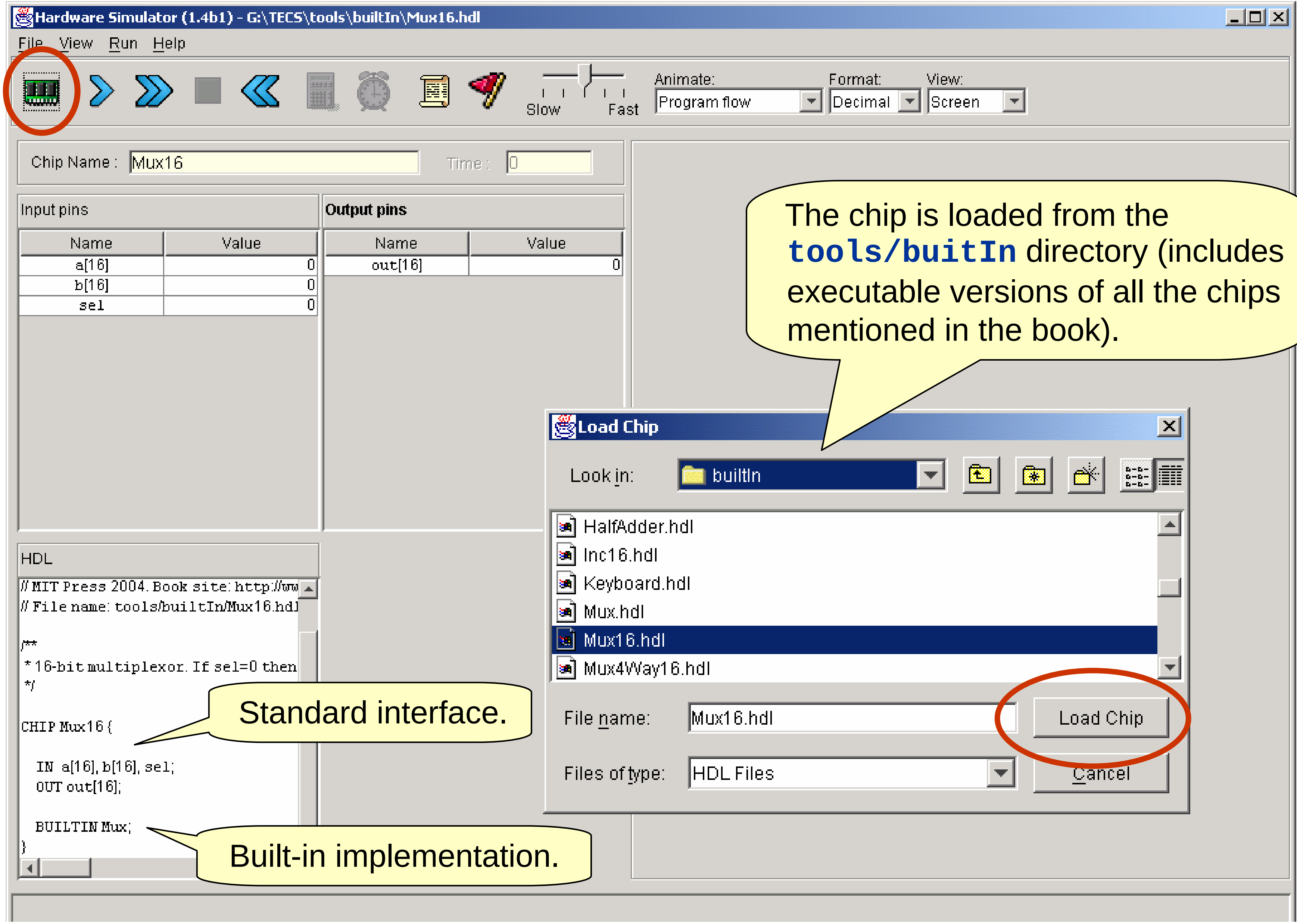
- A built-in chip has an HDL interface and a Java implementation (e.g. here: **`Mux16.class`**)
- The name of the Java class is specified following the **`BUILTIN`** keyword
- Built-In implementations of <u>all</u> the chips that appear in he book are supplied in the **`tools/buitIn`** directory.

```
// Mux16 gate (example)
CHIP Mux16 {
    IN a[16],b[16],sel;
    OUT out[16];
    BUILTIN Mux16;
}
```

Built-in chips are used to:

- Implement primitive gates (in the computer built in this book: **`Nand`** and **`DFF`**)
- Implement chips that have peripheral side effects (like I/O devices)
- Implement chips that feature a GUI (for debugging)
- Provide the functionality of chips that the user did not implement for some reason
- Improve simulation speed and save memory (when used as parts in complex chips)
- Facilitate behavioral simulation of a chip before actually building it in HDL
- Built-in chips can be used either *explicitly*, or *implicitly*.

Explicit Use of Built-in Chips



The chip is loaded from the **tools/buitIn** directory (includes executable versions of all the chips mentioned in the book).

Standard interface.

Built-in implementation.

Implicit Use of Built-in Chips

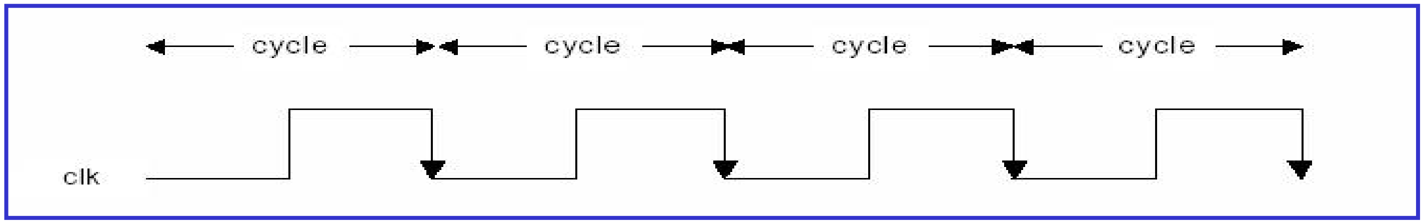
```
/** Exclusive-or gate. out = a xor b */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not(in=a,out=Nota);
    Not(in=b,out=Notb);
    And(a=a,b=Notb,out=aNotb);
    And(a=Nota,b=b,out=bNota);
    Or(a=aNotb,b=bNota,out=out);
}
```

- When any HDL file is loaded, the simulator parses its definition. For each internal chip `Xxx(...)` mentioned in the PARTS section, the simulator looks for an `Xxx.hdl` file in the same directory (e.g. `Not.hdl`, `And.hdl`, and `Or.hdl` in this example).
- If `Xxx.hdl` is found in the current directory (e.g. if it was also written by the user), the simulator uses its HDL logic in the evaluation of the overall chip.
- If `Xxx.hdl` is not found in the current directory, the simulator attempts to invoke the file `tools/builtIn/Xxx.hdl` instead.
- And since `tools/builtIn` includes executable versions of all the chips mentioned in the book, it is possible to build and test any of these chips before first building their lower-level parts.


Part IV:
Clocked Chips
(Sequential Logic)

Clocked (Sequential) Chips

- The implementation of clocked chips is based on *sequential logic*
- The operation of clocked chips is regulated by a master clock signal:



- In our jargon, a clock cycle = *tick*-phase (low), followed by a *tock*-phase (high)
- During a *tick-tock*, the internal states of all the clocked chips are allowed to change, but their outputs are “latched”
- At the beginning of the next *tick*, the outputs of all the clocked chips in the architecture commit to the new values
- In a real computer, the clock is implemented by an oscillator; in simulators, clock cycles can be simulated either manually by the user, or repeatedly by a test script.

The D-Flip-Flop (DFF) Gate

```
/** Data Flip-flop:
 *   out(t)=in(t-1)
 *   where t is the time unit.
 */
CHIP DFF {
    IN in;
    OUT out;

    BUILTIN DFF;
    CLOCKED in, out;
}
```

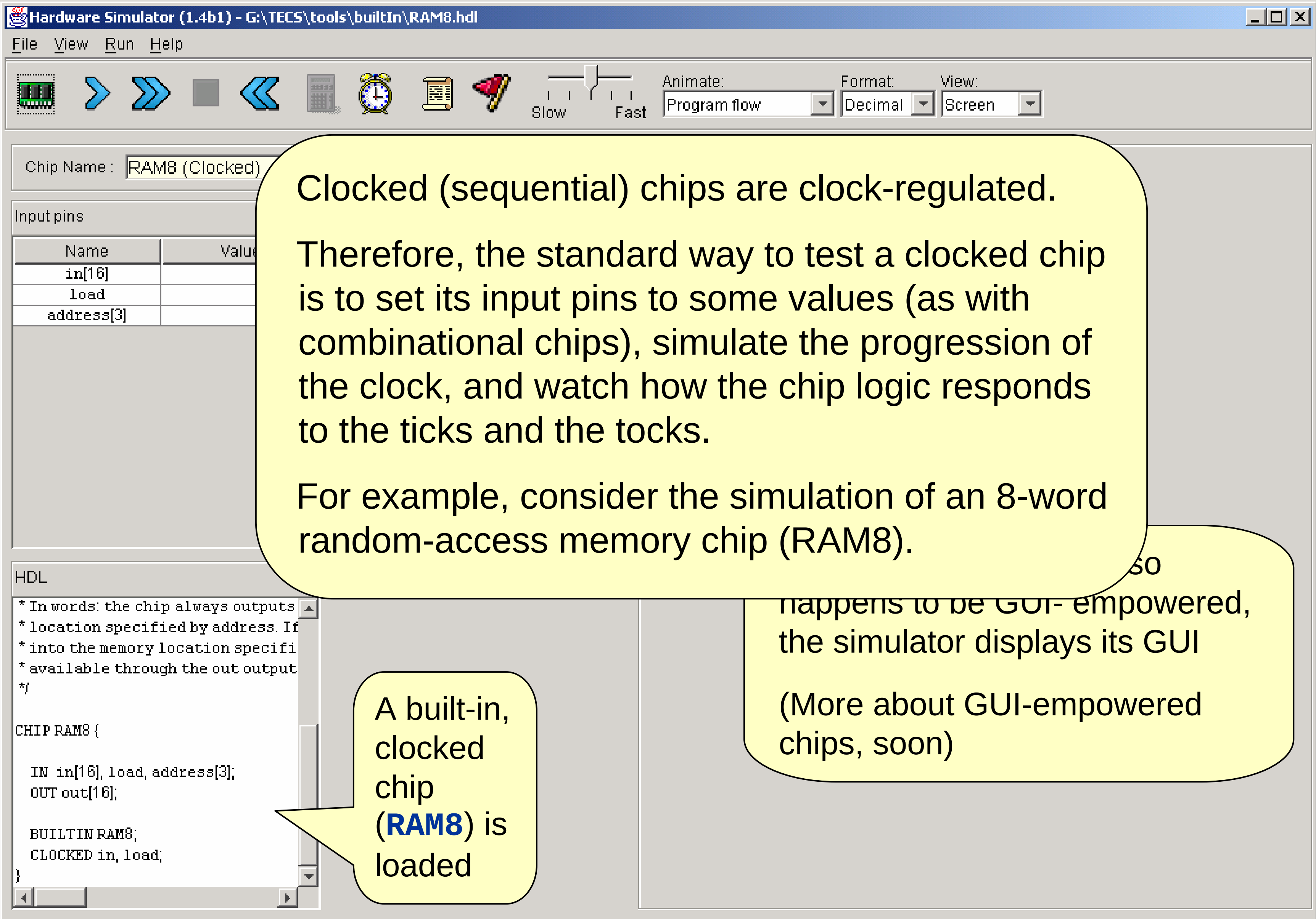
DFF:

- A primitive memory gate that can “remember” a state over clock cycles
- Can serve as the basic building block of all the clocked chips in a computer.

Clocked chips

- Clocked chips include registers, RAM devices, counters, and the CPU
- The simulator knows that the loaded chip is clocked when one or more of its pins is declared “clocked”, or one or more of its parts (or sub-parts, recursively) is a clocked chip
- In the hardware platform built in the book, all the clocked chips are based, directly or indirectly, on (many instances of) built-in DFF gates.

Simulating Clocked Chips



Clocked (sequential) chips are clock-regulated.

Therefore, the standard way to test a clocked chip is to set its input pins to some values (as with combinational chips), simulate the progression of the clock, and watch how the chip logic responds to the ticks and the tocks.

For example, consider the simulation of an 8-word random-access memory chip (RAM8).

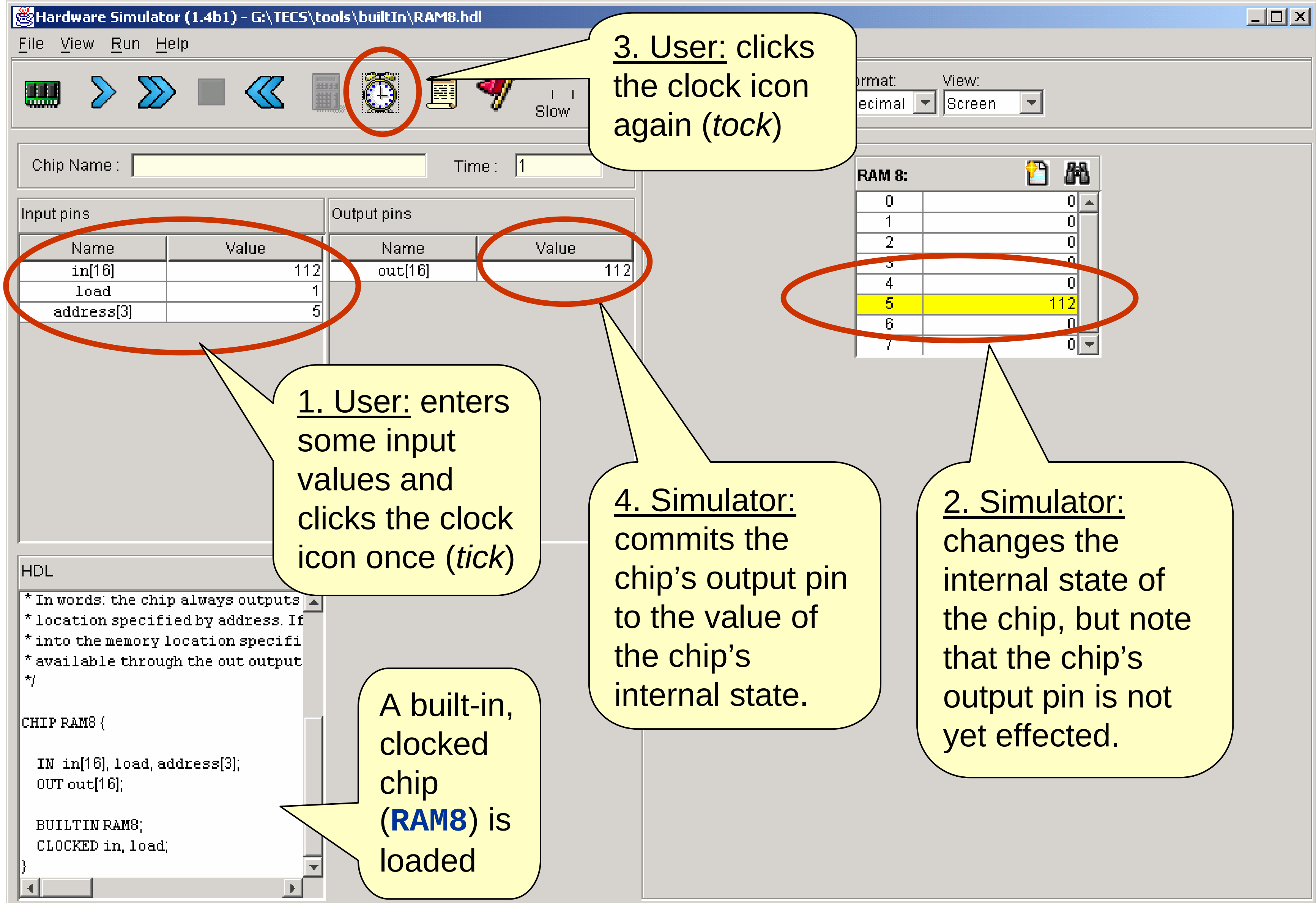
A built-in, clocked chip (**RAM8**) is loaded

so

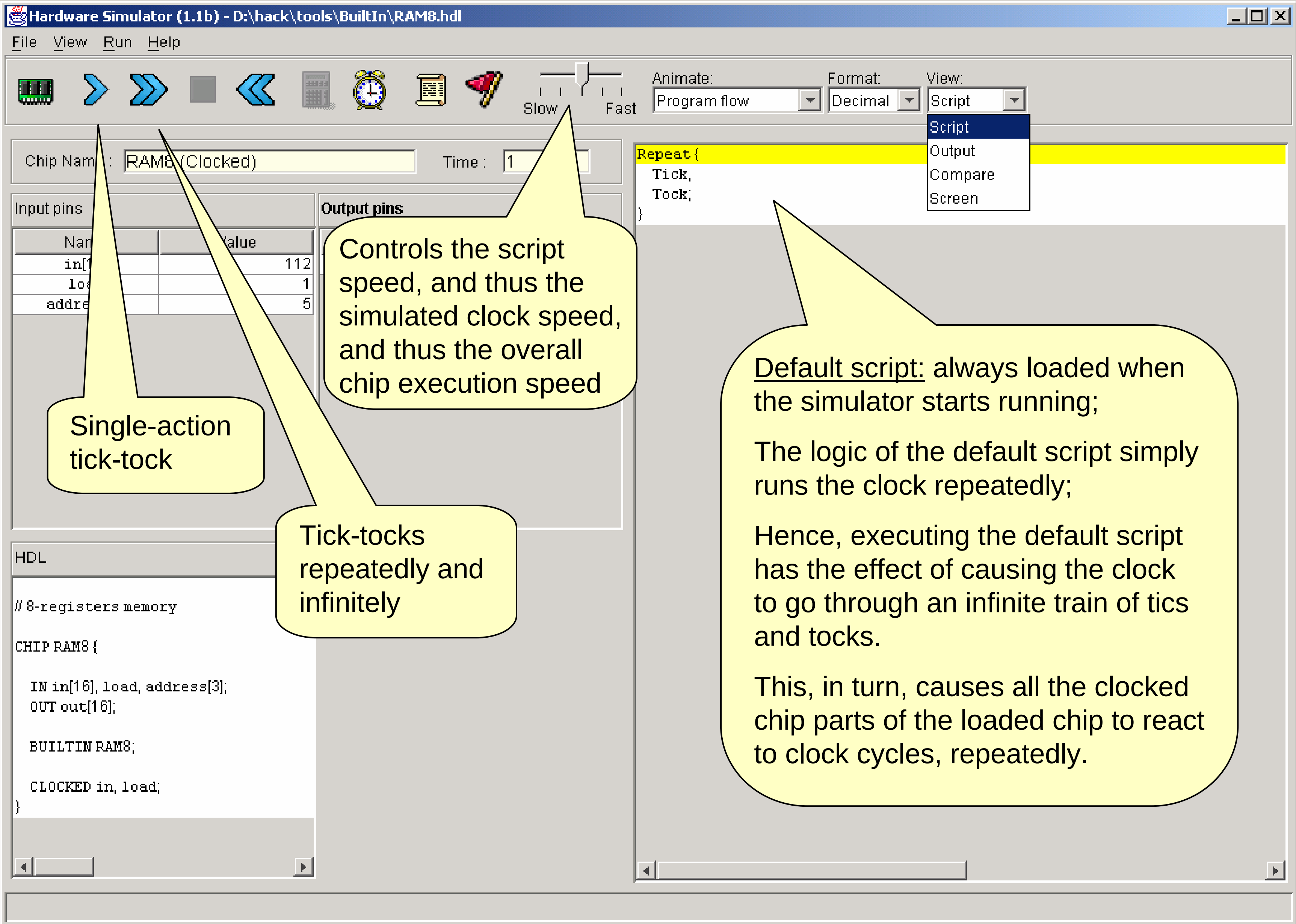
happens to be GUI- empowered, the simulator displays its GUI

(More about GUI-empowered chips, soon)

Simulating Clocked Chips

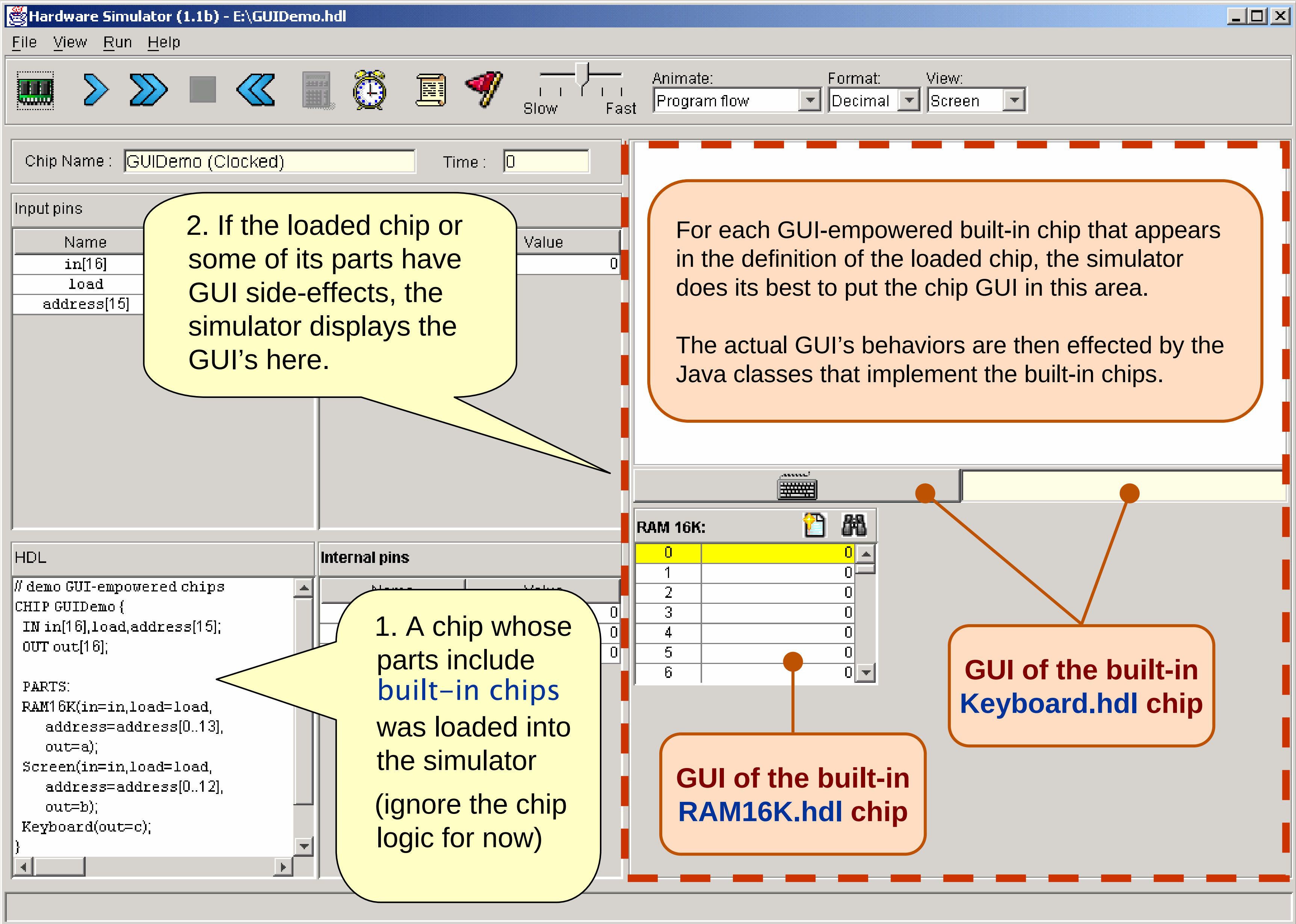


Simulating Clocked Chips Using a Test Script




Part V:
GUI-Empowered
chips

Built-in Chips with GUI Effects



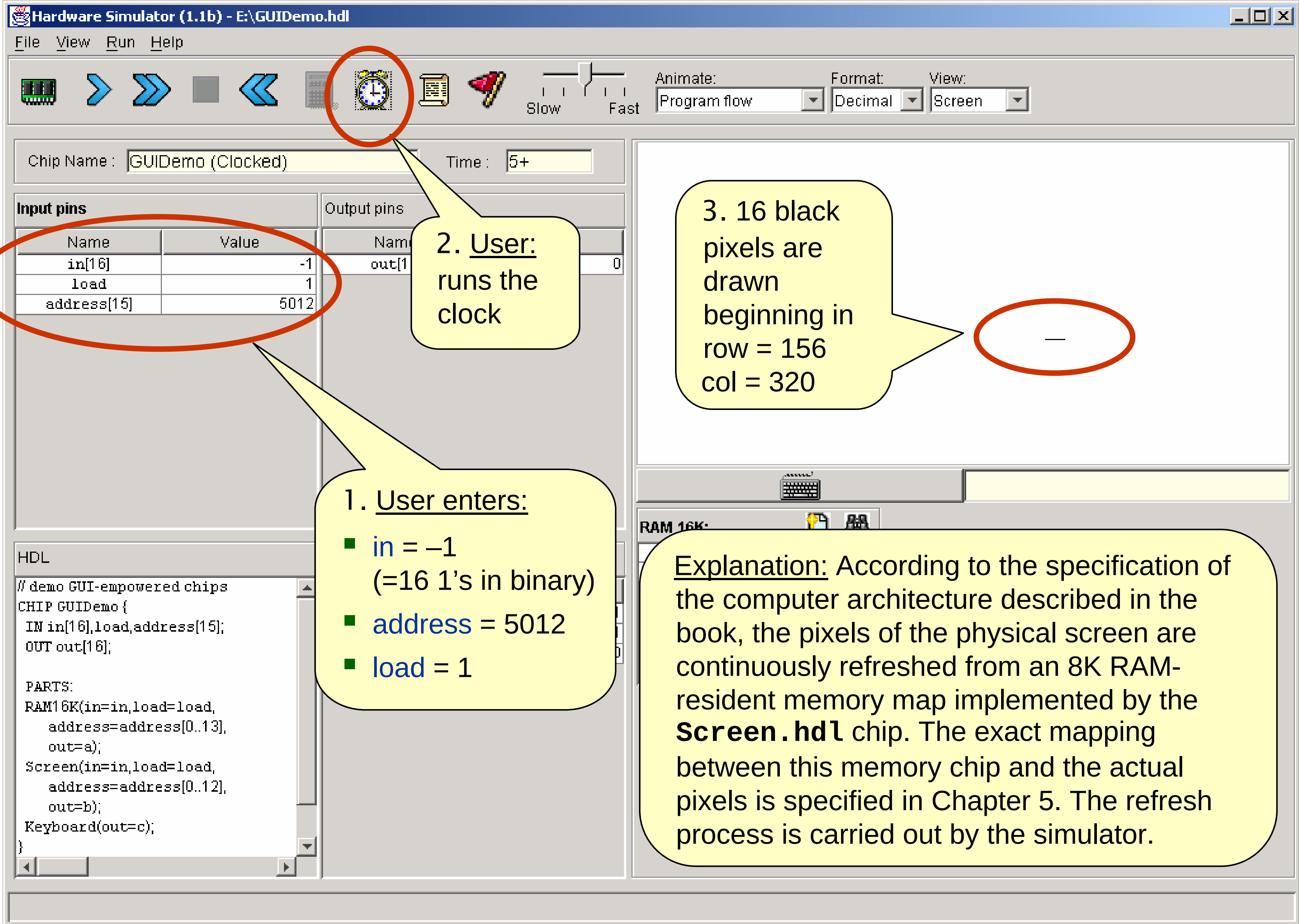
The Logic of the GUIDemo Chip

```
// Demo of built-in chips with GUI effects
CHIP GUIDemo {
  IN in[16],load,address[15];
  OUT out[16];
  PARTS:
  RAM16K(in=in,load=load,address=address[0..13],out=null);
  Screen(in=in,load=load,address=address[0..12],out=null);
  Keyboard(out=null);
}
```

RAM16K, **Screen**, & **Keyboard** are built-in chips with GUI side-effects

- **Effect:** When the simulator evaluates this chip, it displays the GUI side-effects of its built-in chip parts
- **Chip logic:** The only purpose of this demo chip is to force the simulator to show the GUI of some built-in chips. Other than that, the chip logic is meaningless: it simultaneously feeds the 16-bit data input (**in**) into the **RAM16K** and the **Screen** chips, and it does nothing with the keyboard.

GUIDemo Chip in Action




Part VI:
Debugging tools

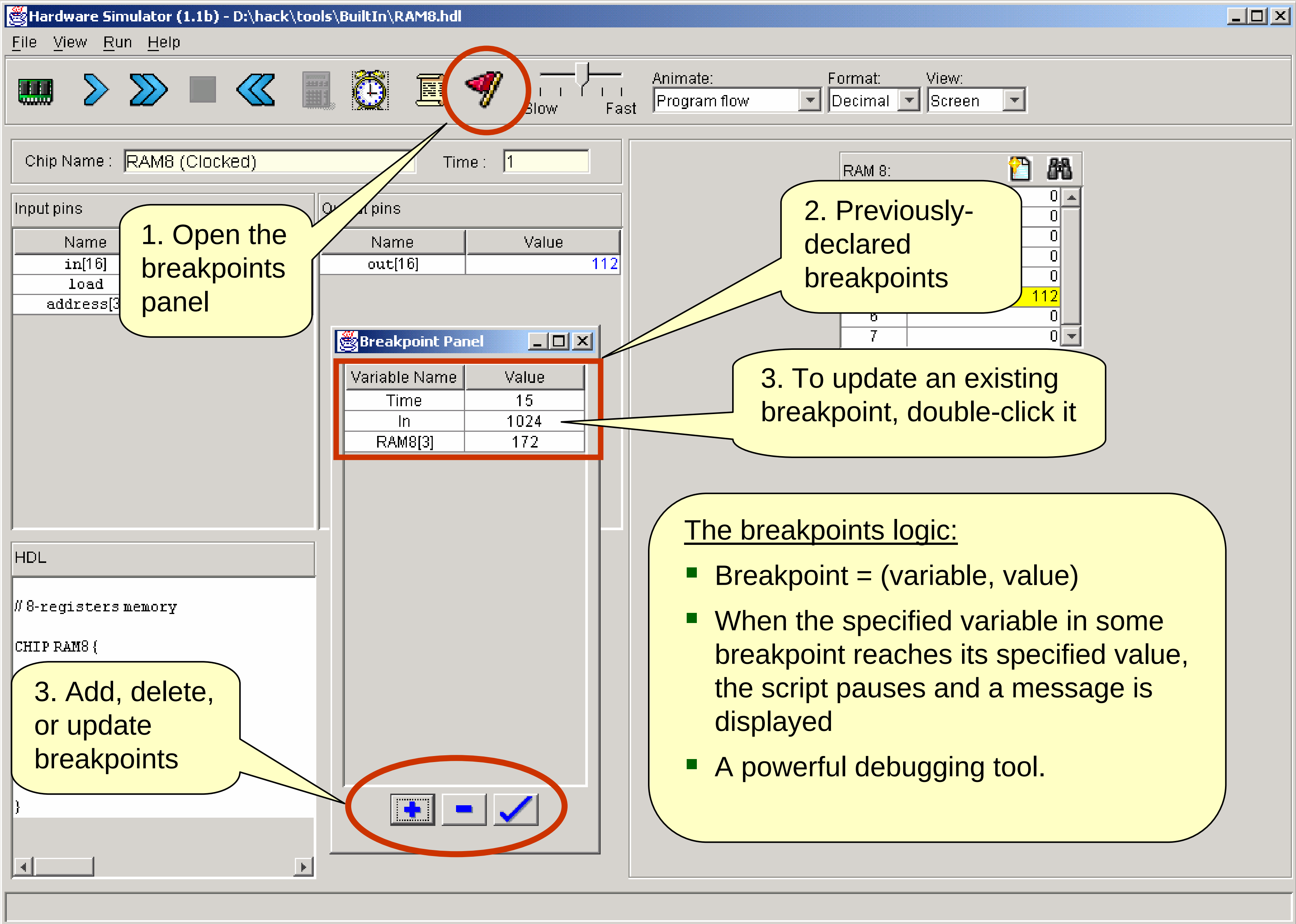
System Variables

The simulator recognizes and maintains the following variables:

- Time: the number of time-units (clock-cycles) that elapsed since the script started running is stored in the variable **`time`**
- Pins: the values of all the input, output, and internal pins of the simulated chip are accessible as variables, using the names of the pins in the HDL code
- GUI elements: the values stored in the states of GUI-empowered built-in chips can be accessed via variables. For example, the value of register 3 of the **`RAM8`** chip can be accessed via **`RAM8[3]`**.

All these variables can be used in scripts and *breakpoints*, for debugging.

Breakpoints



The breakpoints logic:

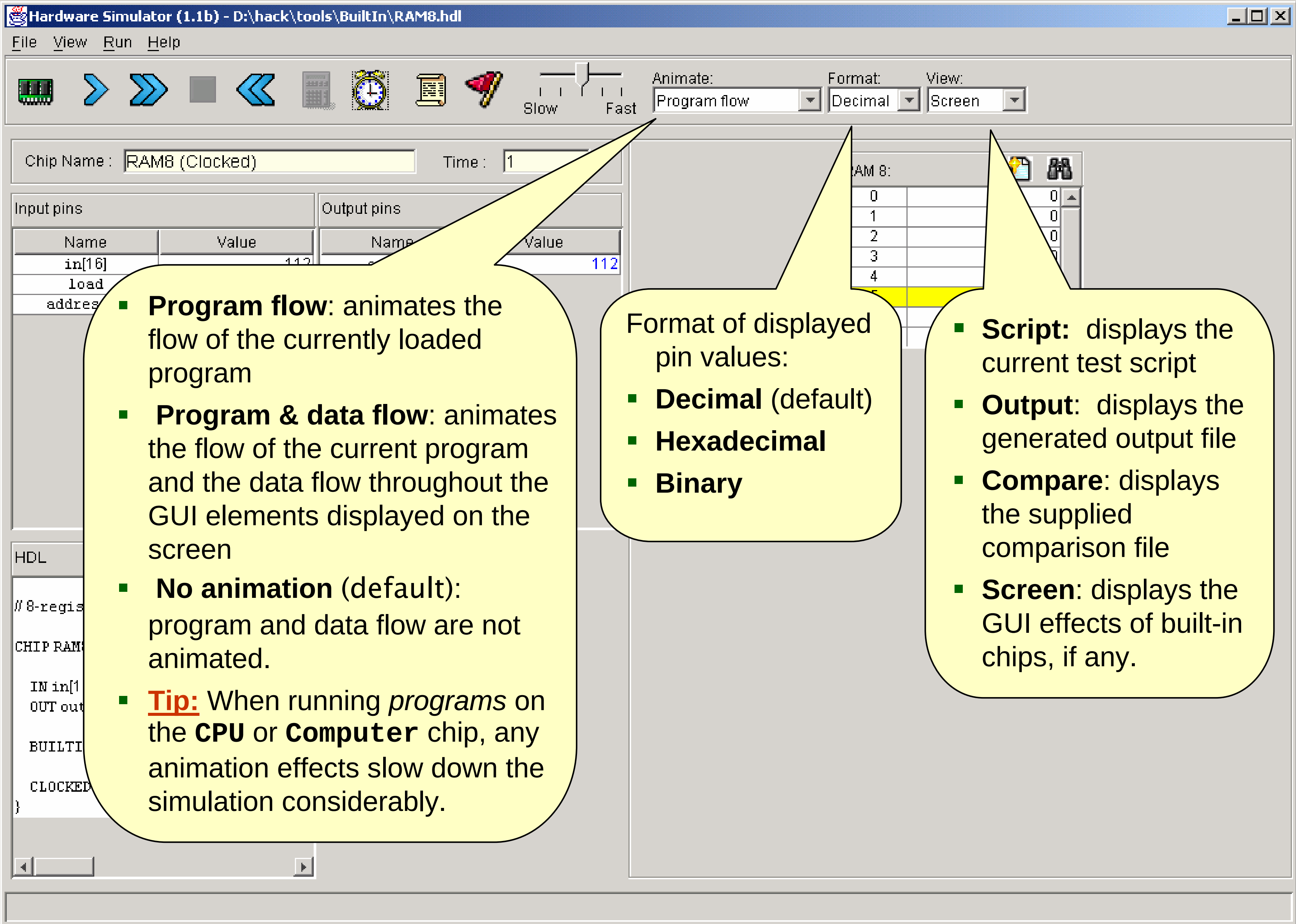
- Breakpoint = (variable, value)
- When the specified variable in some breakpoint reaches its specified value, the script pauses and a message is displayed
- A powerful debugging tool.

Scripts for Testing the Topmost **`Computer`** chip

```
load Computer.hdl
ROM32K load Max.hack,
output-file ComputerMax.out,
compare-to ComputerMax.cmp,
output-list time%S1.4.1
         reset%B2.1.2
         ARegister[]%D1.7.1
         DRegister[]%D1.7.1
         PC[]%D0.4.0
         RAM16K[0]%D1.7.1
         RAM16K[1]%D1.7.1
         RAM16K[2]%D1.7.1;
breakpoint PC 10;
// First run: compute max(3,5)
set RAM16K[0] 3,
set RAM16K[1] 5,
output;
repeat 14 {
    tick, tock, output;
}
// Reset the PC (preparing for
// second run)
set reset 1,
tick, tock, output;
// Etc.
clear-breakpoints;
```

- Scripts that test the **`CPU`** chip or the **`Computer`** chip described in the book usually start by loading a machine-language program (**`.asm`** or **`.hack`** file) into the **`ROM32K`** chip
- The rest of the script typically uses various features like:
  - Output files
  - Loops
  - Breakpoints
  - Variables manipulation
  - tick, tock
  - Etc.
- All these features are described in Appendix B of the book (*Test Scripting Language*).

Visual Options



- **Program flow**: animates the flow of the currently loaded program
- **Program & data flow**: animates the flow of the current program and the data flow throughout the GUI elements displayed on the screen
- **No animation** (default): program and data flow are not animated.
- Tip: When running *programs* on the **`CPU`** or **`Computer`** chip, any animation effects slow down the simulation considerably.

Format of displayed pin values:

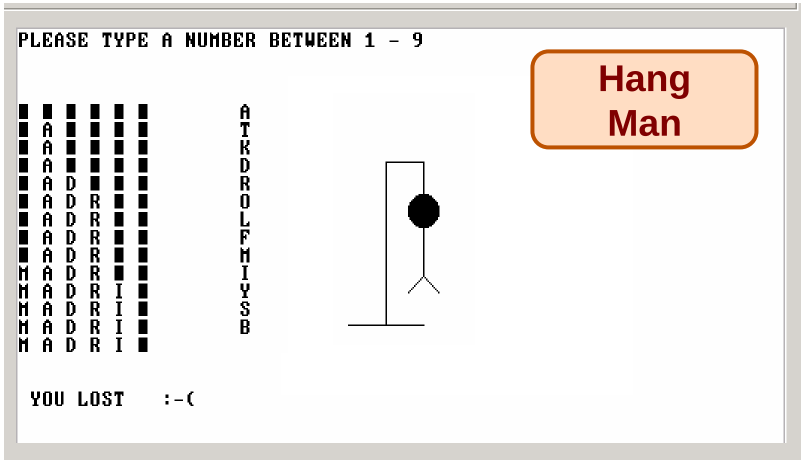
- **Decimal** (default)
- **Hexadecimal**
- **Binary**

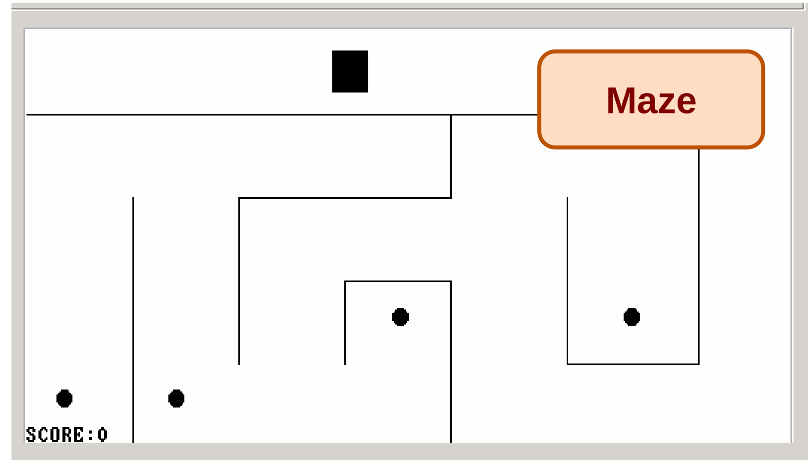
- **Script:** displays the current test script
- **Output**: displays the generated output file
- **Compare**: displays the supplied comparison file
- **Screen**: displays the GUI effects of built-in chips, if any.

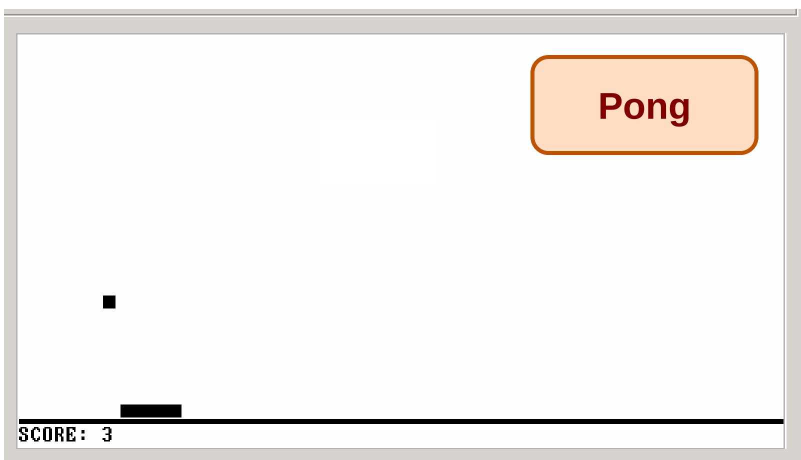

Part VII:
The Hack
Hardware Platform

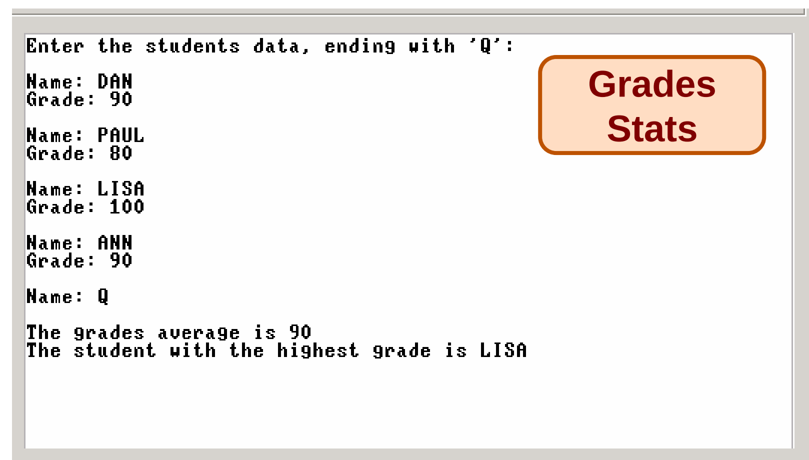
Hack: a General-Purpose 16-bit Computer

Sample applications running on the Hack computer:









These programs (and many more) were written in the Jack programming language, running in the Jack OS environment over the Hack hardware platform. The hardware platform is built in chapters 1-5, and the software hierarchy in chapters 6-12.

The Hack Chip-Set and Hardware Platform

Elementary logic gates

(**Project 1**):

- **Nand** (primitive)
- **Not**
- **And**
- **Or**
- **Xor**
- **Mux**
- **Dmux**
- **Not16**
- **And16**
- **Or16**
- **Mux16**
- **Or8Way**
- **Mux4Way16**
- **Mux8Way16**
- **DMux4Way**
- **DMux8Way**

Combinational chips

(**Project 2**):

- **HalfAdder**
- **FullAdder**
- **Add16**
- **Inc16**
- **ALU**

Sequential chips

(**Project 3**):

- **DFF** (primitive)
- **Bit**
- **Register**
- **RAM8**
- **RAM64**
- **RAM512**
- **RAM4K**
- **RAM16K**
- **PC**

Computer Architecture

(**Project 5**):

- **Memory**
- **CPU**
- **Computer**

Most of these chips are generic, meaning that they can be used in the construction of many different computers.

The Hack chip-set and hardware platform can be built using the hardware simulator, starting with primitive **Nand.hdl** *and* **DFF.hdl** *gates and culminating in the* **Computer.hdl** *chip.*

This construction is described in chapters 1,2,3,5 of the book, and carried out in the respective projects.

Aside: H.D. Thoreau about chips, bugs, and close observation:

I was surprised to find that the chips were covered with such combatants, that it was not a duellum, but a bellum, a war between two races of ants, the red always pitted against the black, and frequently two red ones to one black. The legions of these Myrmidons covered all the hills and vales in my wood-yard, and the ground was already strewn with the dead and dying, both red and black.

It was the only battle which I have ever witnessed, the only battlefield I ever trod while the battle was raging; internecine war; the red republicans on the one hand, and the black imperialists on the other. On every side they were engaged in deadly combat, yet without any noise that I could hear, and human soldiers never fought so resolutely.... The more you think of it, the less the difference. And certainly there is not the fight recorded in Concord history, at least, if in the history of America, that will bear a moment's comparison with this, whether for the numbers engaged in it, or for the patriotism and heroism displayed.

From "Brute Neighbors," Walden (1854).