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

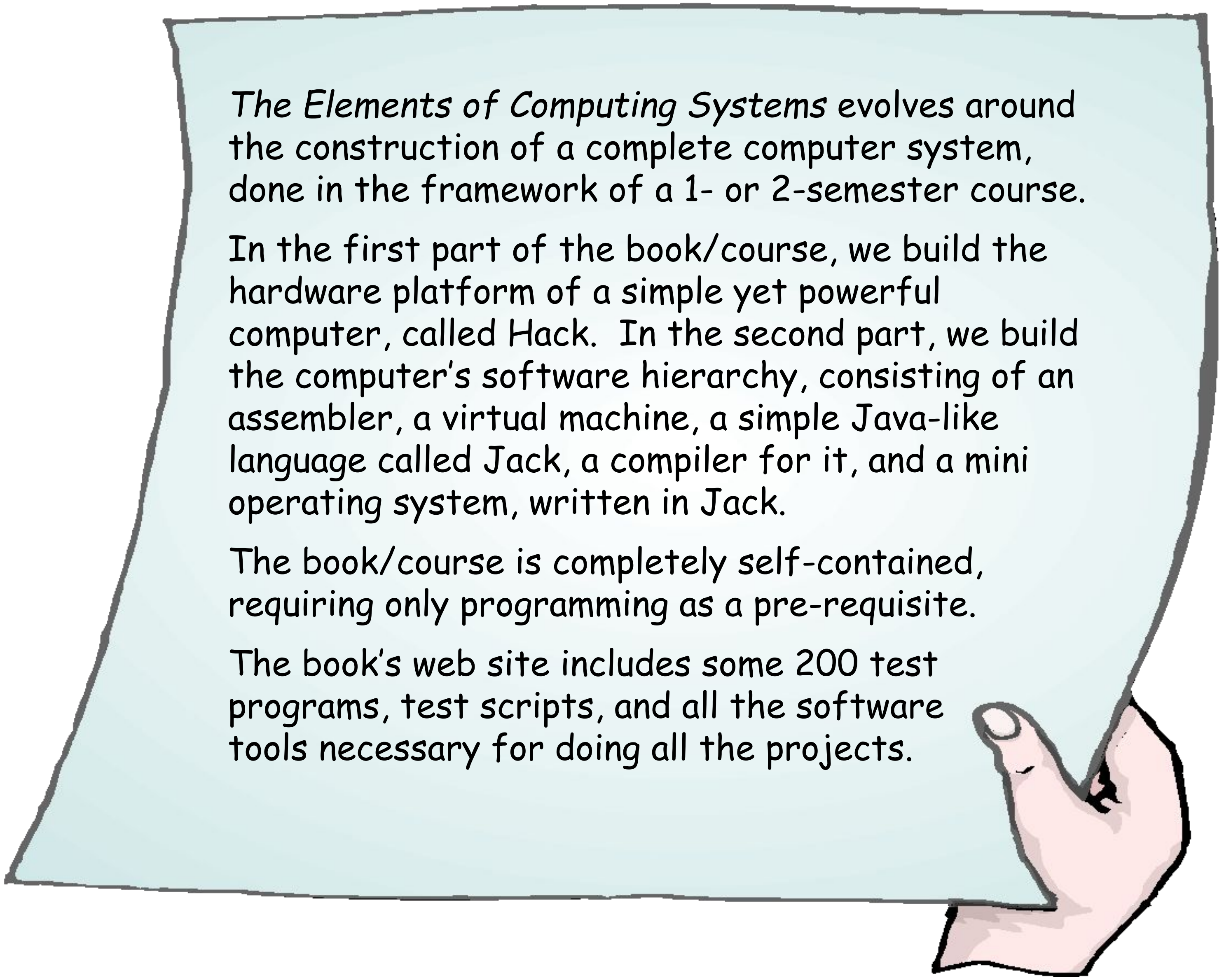
MIT Press

www.nand2tetris.org

Developed by students at the
Efi Arazi School of Computer Science Reichman University

Chief Software Architect: Yaron Ukrainitz

Background



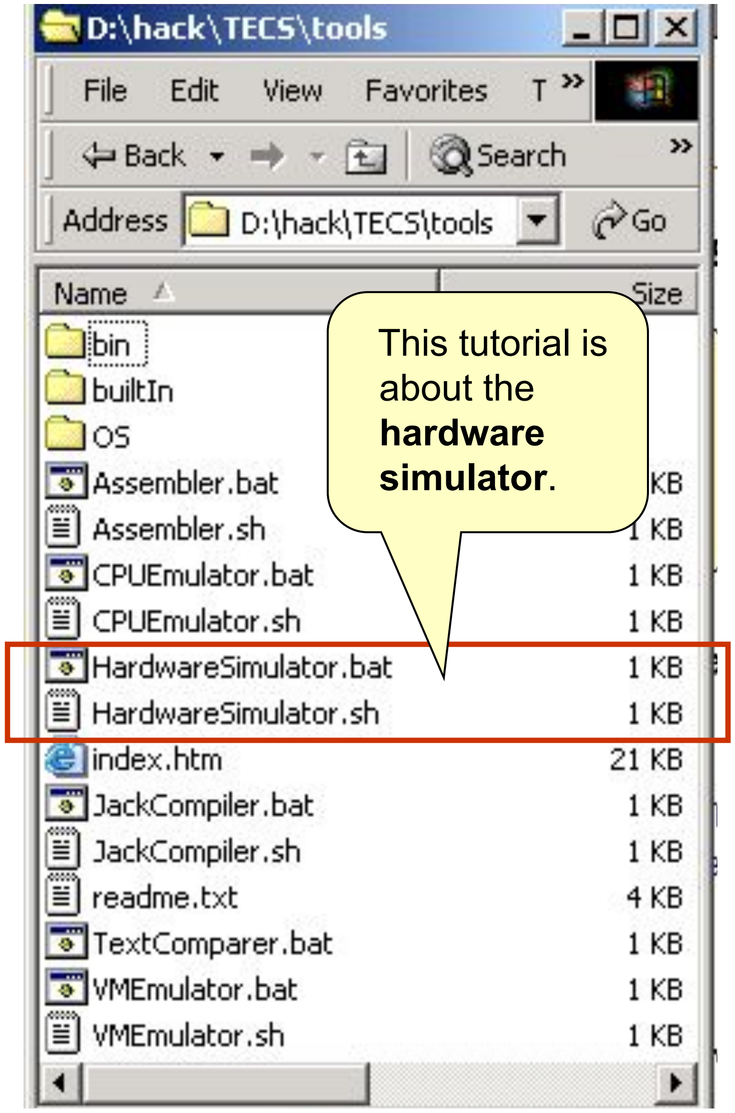
The Elements of Computing Systems evolves around the construction of a complete computer system, done in the framework of a 1- or 2-semester course.

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


D:\hack\TECS\tools
File Edit View Favorites T
Back Search
Address D:\hack\TECS\tools Go
This tutorial is about the **hardware simulator**.

Name	Size
bin	
builtIn	
OS	
Assembler.bat	KB
Assembler.sh	1 KB
CPUEmulator.bat	1 KB
CPUEmulator.sh	1 KB
HardwareSimulator.bat	1 KB
HardwareSimulator.sh	1 KB
index.htm	21 KB
JackCompiler.bat	1 KB
JackCompiler.sh	1 KB
readme.txt	4 KB
TextComparer.bat	1 KB
VMEmulator.bat	1 KB
VMEmulator.sh	1 KB

(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 is 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*

Hardware Simulation Tutorial

Chip Definition (`.hdl` file)

chip interface

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

    // Implementation missing.
}
```

- Chip interface:
  - 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)

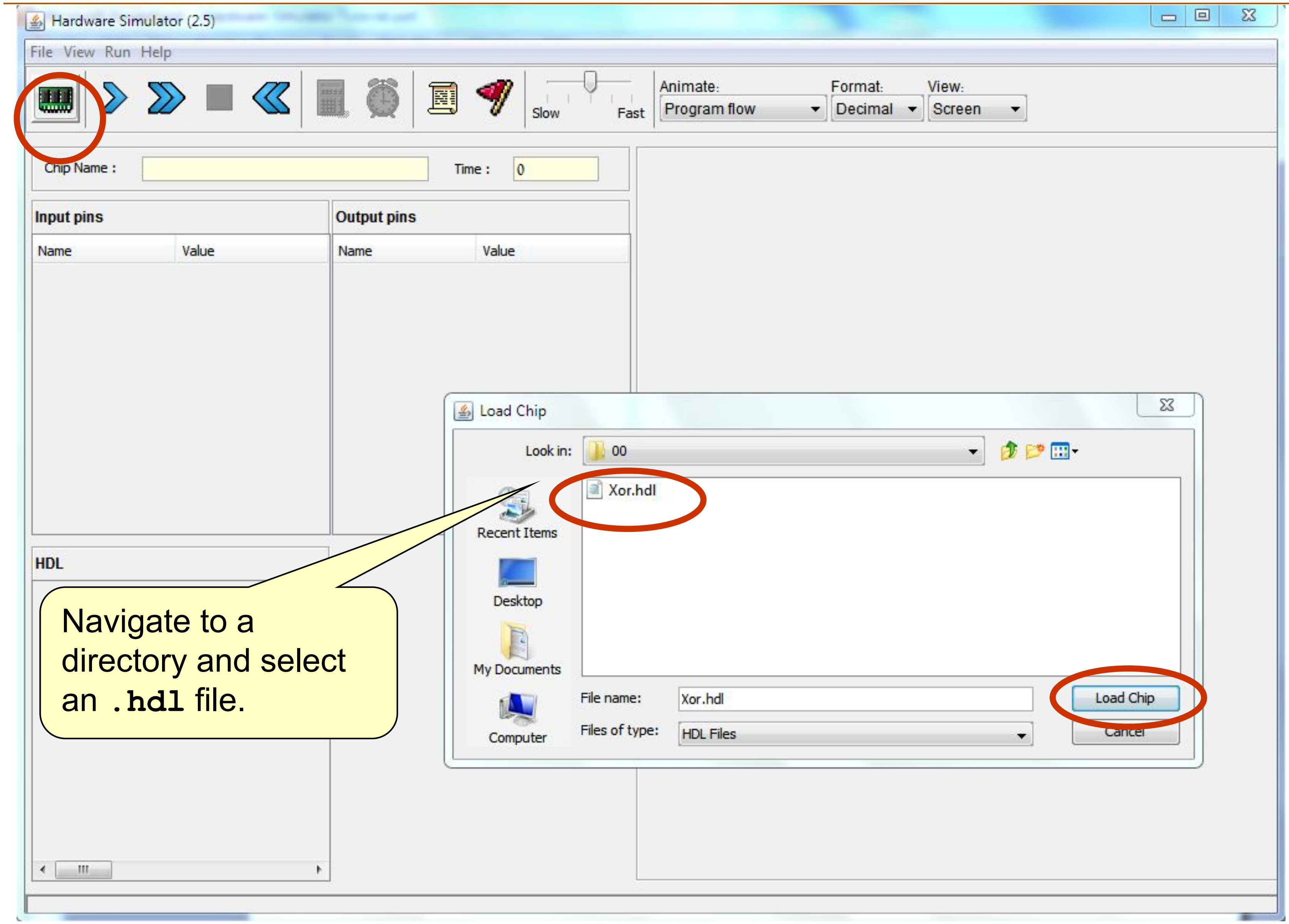
chip interface

chip implementation

```
/** 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

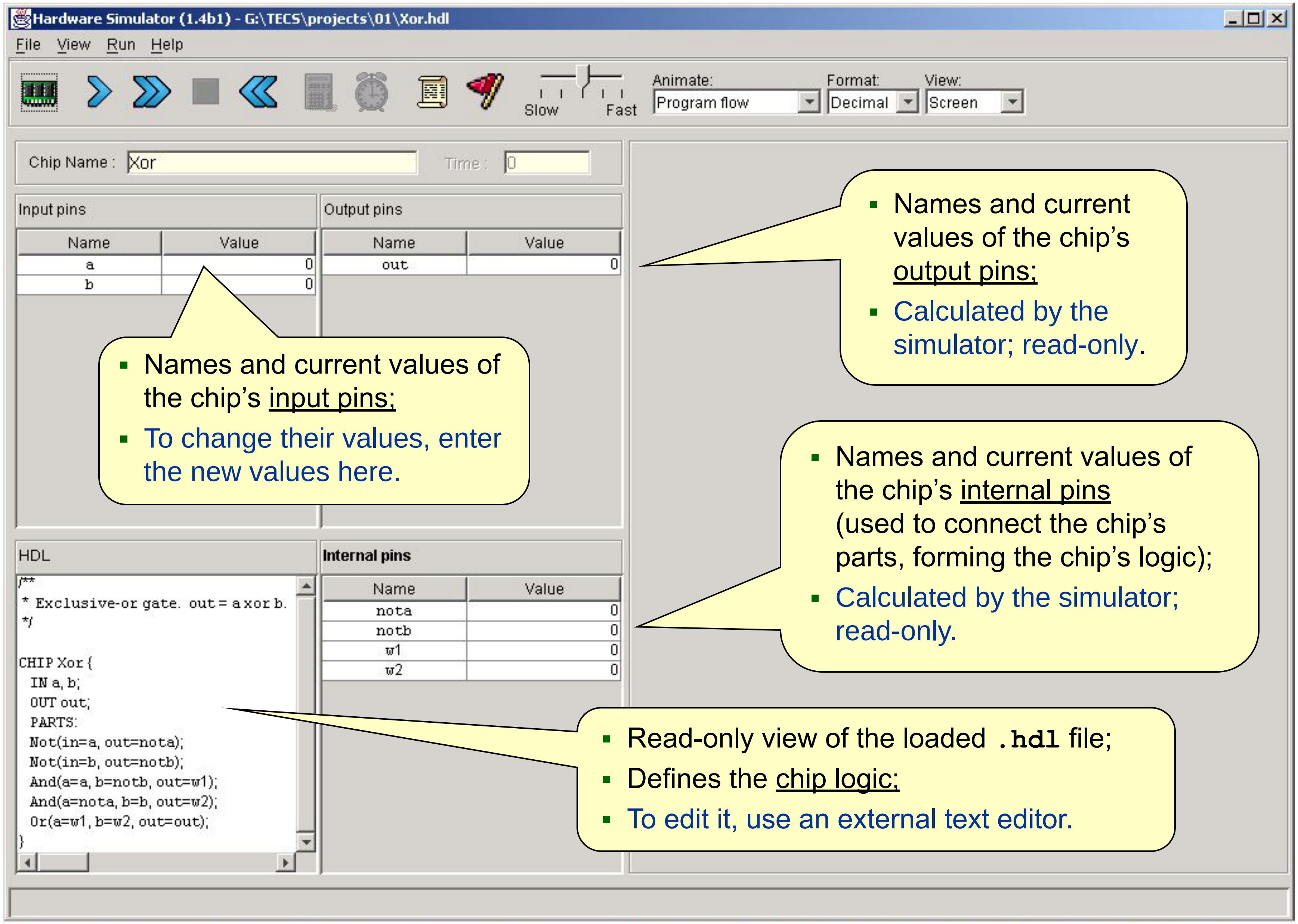


Animate: Program flow
Format: Decimal
View: Screen
Chip Name :
Time : 0

Look in: 00
Xor.hdl
File name: Xor.hdl
Files of type: HDL Files

Navigate to a directory and select an **`.hdl`** file.

Loading a Chip



Animate: Program flow
Format: Decimal
View: Screen

Chip Name : Xor
Time : 0

Input pins

Name	Value
a	0
b	0

Output pins

Name	Value
out	0

Internal pins

Name	Value
nota	0
notb	0
w1	0
w2	0

HDL

```
/**
 * 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);
}
```

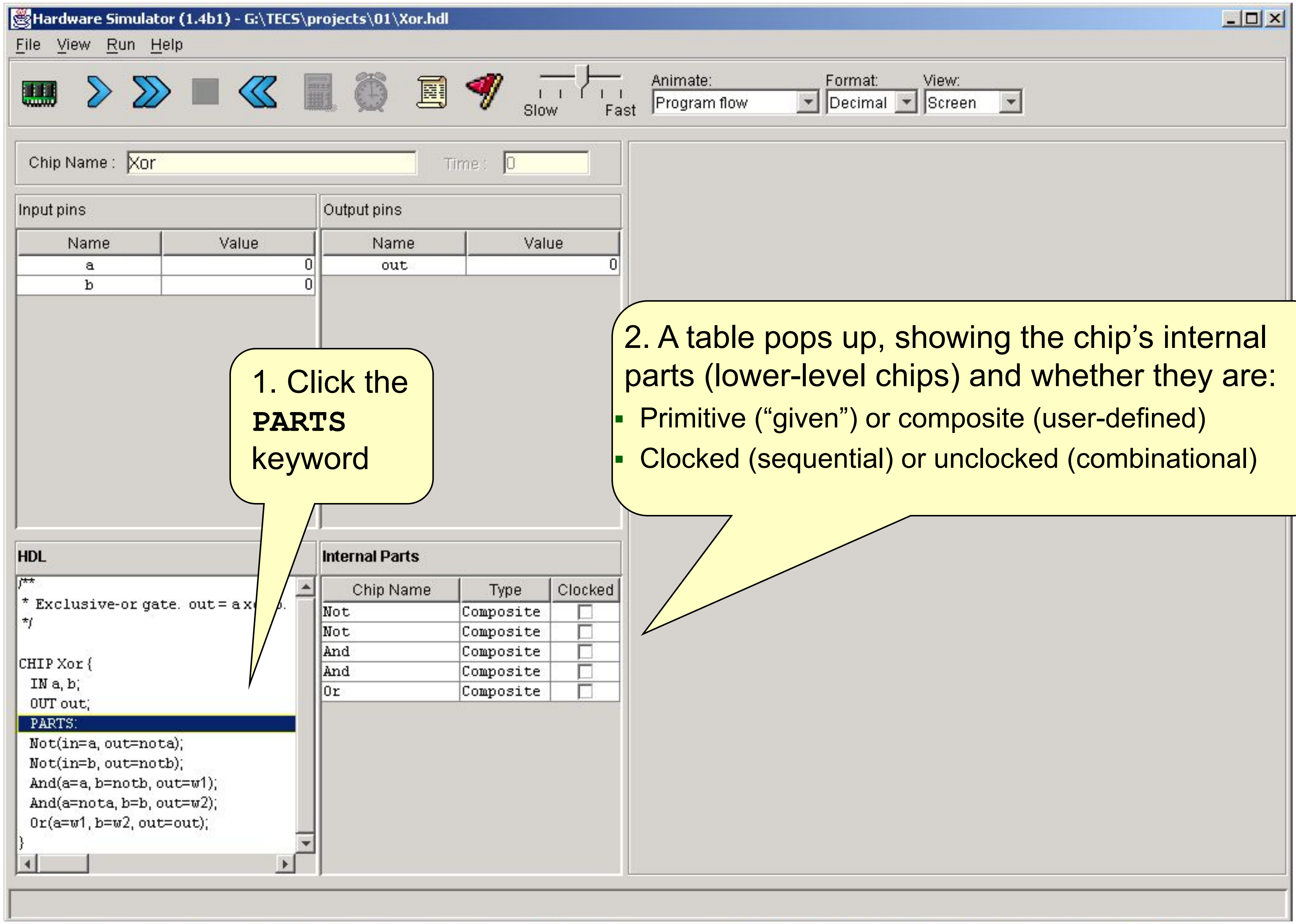
- Names and current values of the chip's <u>output pins;</u>
- Calculated by the simulator; read-only.

- Names and current values of the chip's <u>input pins;</u>
- To change their values, enter the new values here.

- Names and current values of the chip's <u>internal pins</u> (used to connect the chip's parts, forming the chip's logic);
- Calculated by the simulator; read-only.

- Read-only view of the loaded **.hdl** file;
- Defines the <u>chip logic;</u>
- To edit it, use an external text editor.

Exploring the Chip Logic



Chip Name : Xor Time : 0

Input pins

Name	Value
a	0
b	0

Output pins

Name	Value
out	0

HDL

```
/**
 * 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);
}
```

Internal Parts

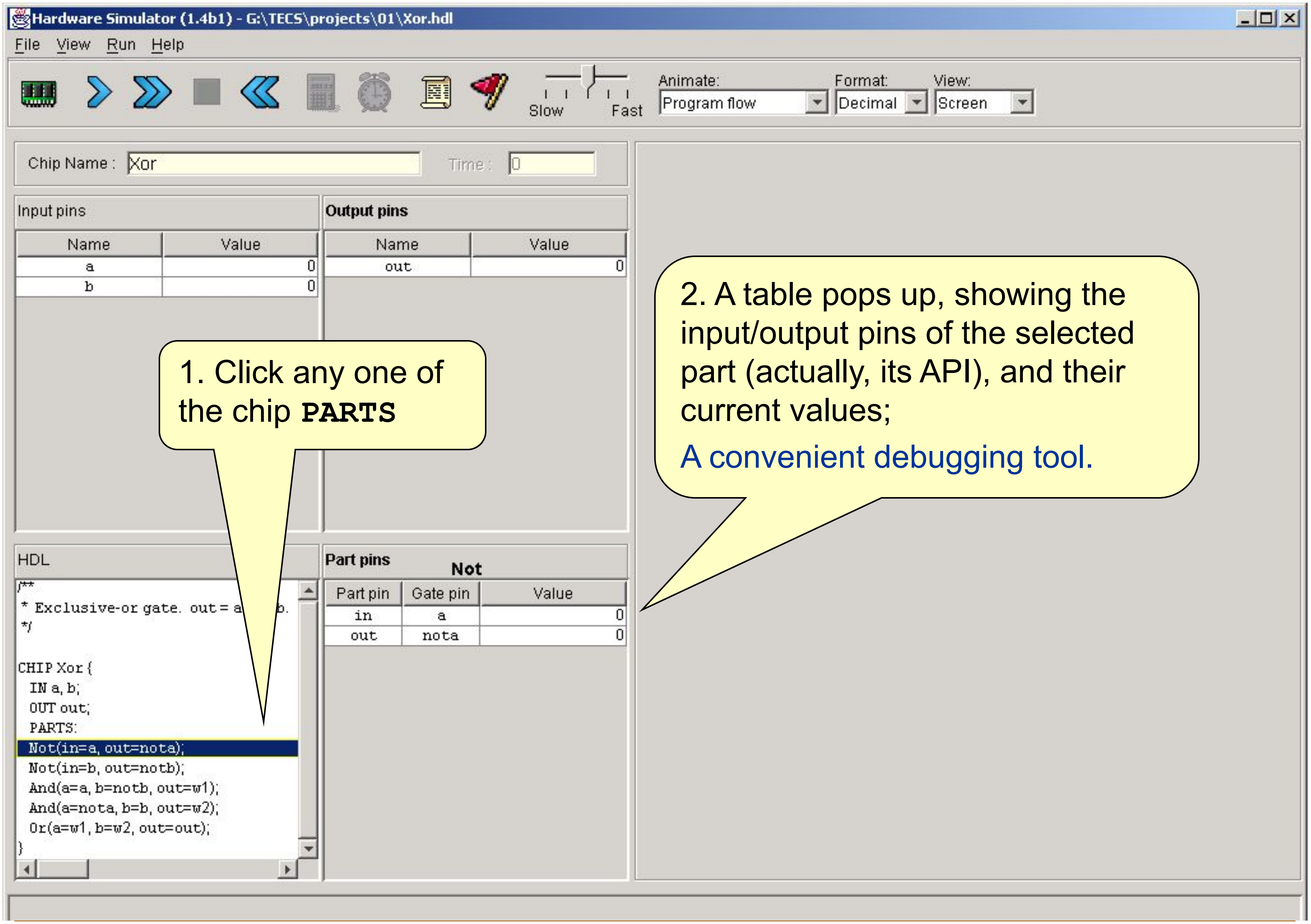
Chip Name	Type	Clocked
Not	Composite	☐
Not	Composite	☐
And	Composite	☐
And	Composite	☐
Or	Composite	☐

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



Chip Name : Xor   Time : 0

Input pins

Name	Value
a	0
b	0

Output pins

Name	Value
out	0

Part pins **Not**

Part pin	Gate pin	Value
in	a	0
out	nota	0

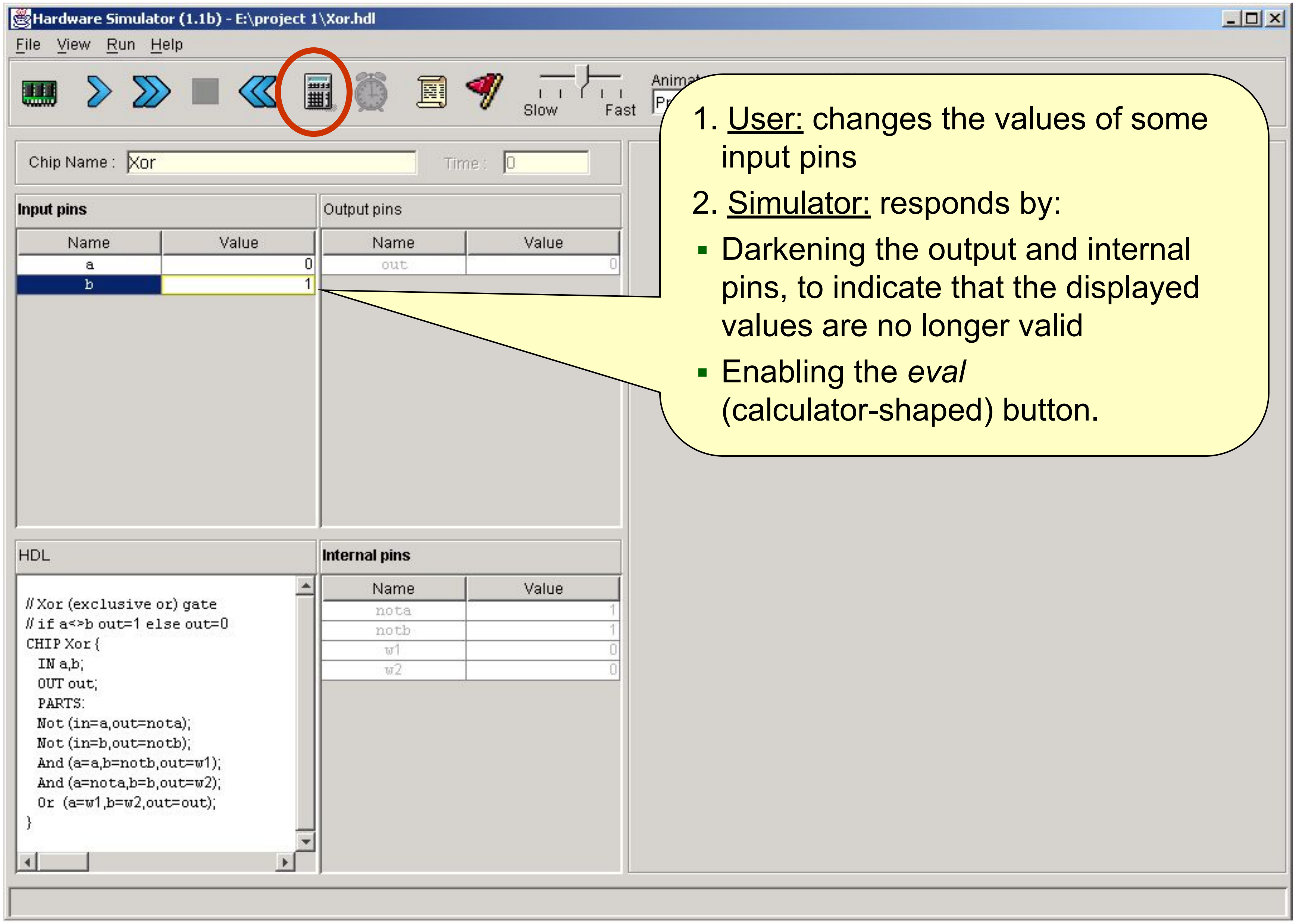
```
/**
 * Exclusive-or gate. out = a      b.
*/

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);
}
```

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



Chip Name : Xor   Time : 0

Input pins

Name	Value
a	0
b	1

Output pins

Name	Value
out	0

HDL

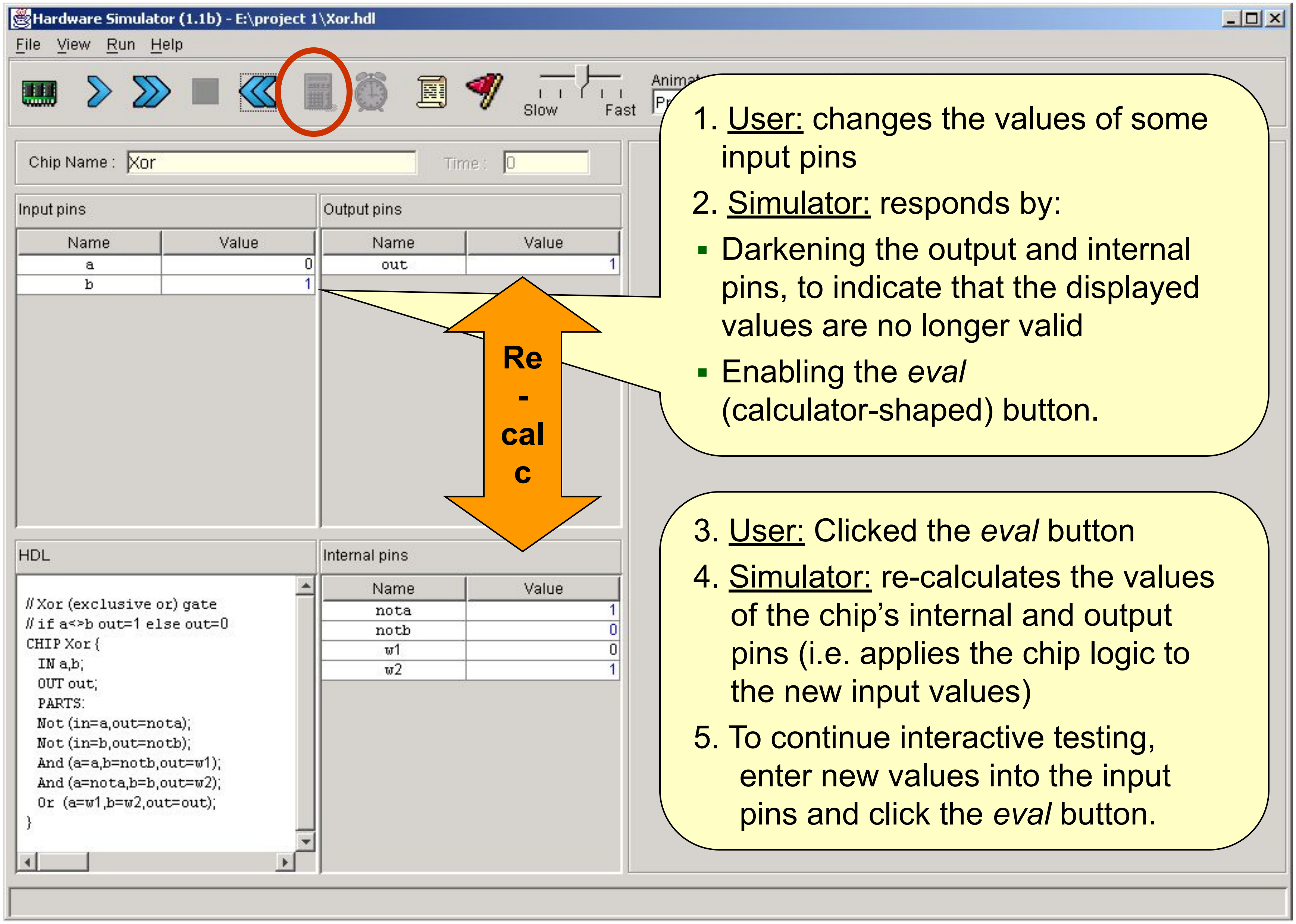
```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
  IN a,b;
  OUT out;
  PARTS:
  Not (in=a,out=nota);
  Not (in=b,out=notb);
  And (a=a,b=notb,out=w1);
  And (a=nota,b=b,out=w2);
  Or  (a=w1,b=w2,out=out);
}
```

Internal pins

Name	Value
nota	1
notb	1
w1	0
w2	0

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.

Interactive Chip Testing



Name	Value
a	0
b	1

Name	Value
out	1

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
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);
}
```

Name	Value
nota	1
notb	0
w1	0
w2	1

1. <u>User:</u> changes the values of some input pins
2. <u>Simulator:</u> 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. <u>User:</u> Clicked the *eval* button
4. <u>Simulator:</u> 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.

Hardware Simulation Tutorial

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`)

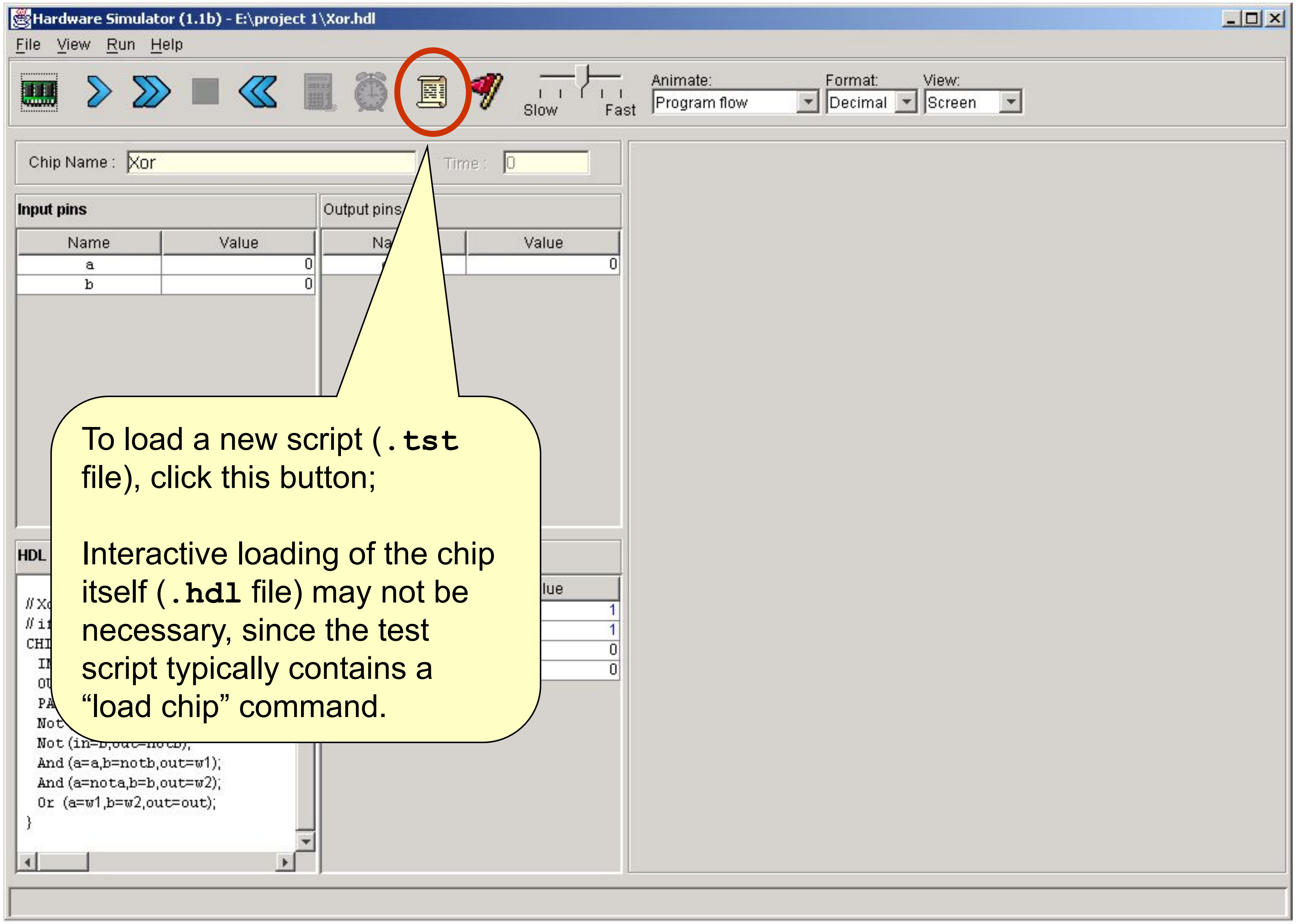
```
|   a   |   b   |  out  |
|   0   |   0   |   0   |
|   0   |   1   |   1   |
|   1   |   0   |   1   |
|   1   |   1   |   0   |
```

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



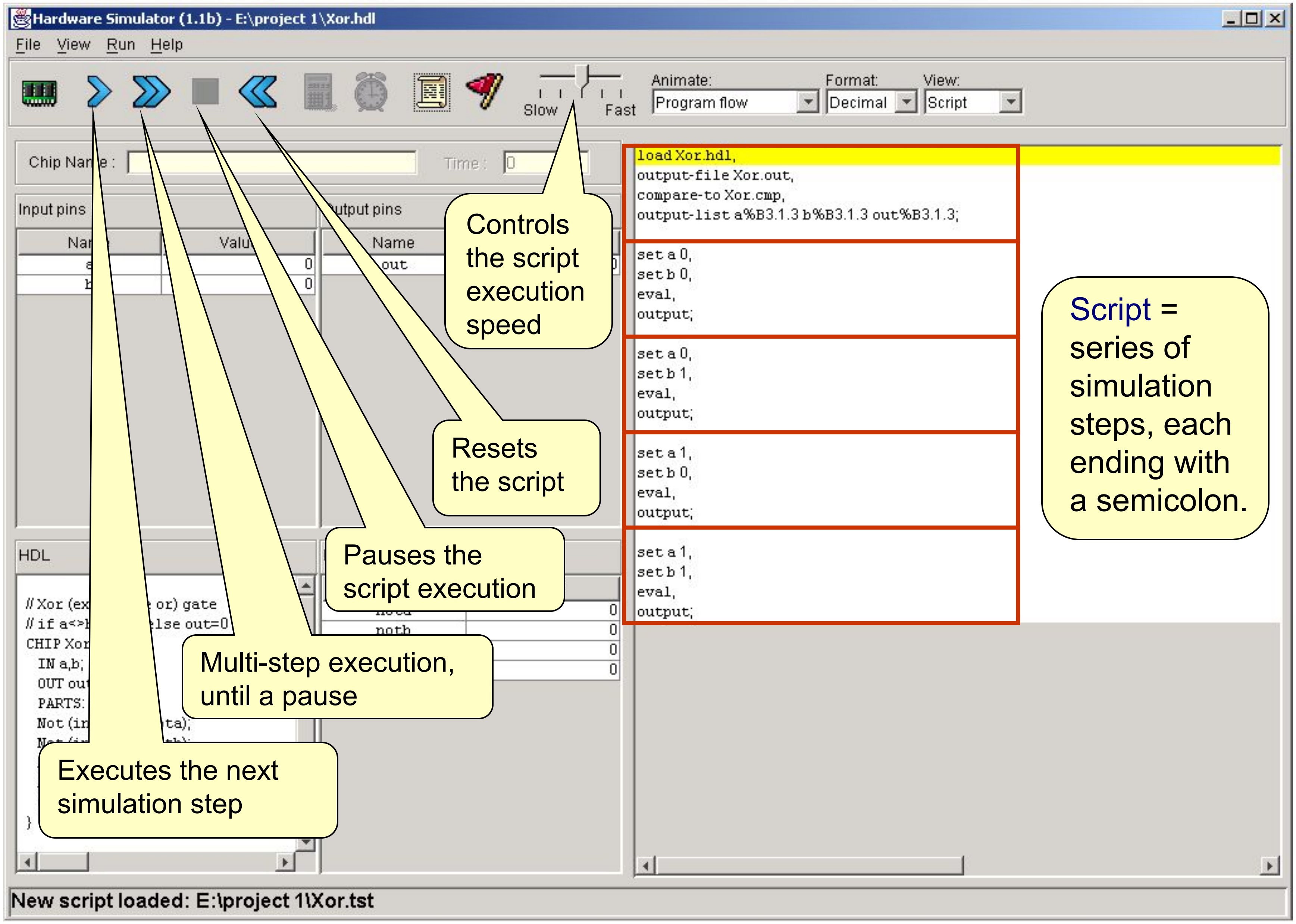
Name	Value
a	0
b	0

```
And (a=a,b=notb,out=w1);
And (a=nota,b=b,out=w2);
Or  (a=w1,b=w2,out=out);
}
```

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.

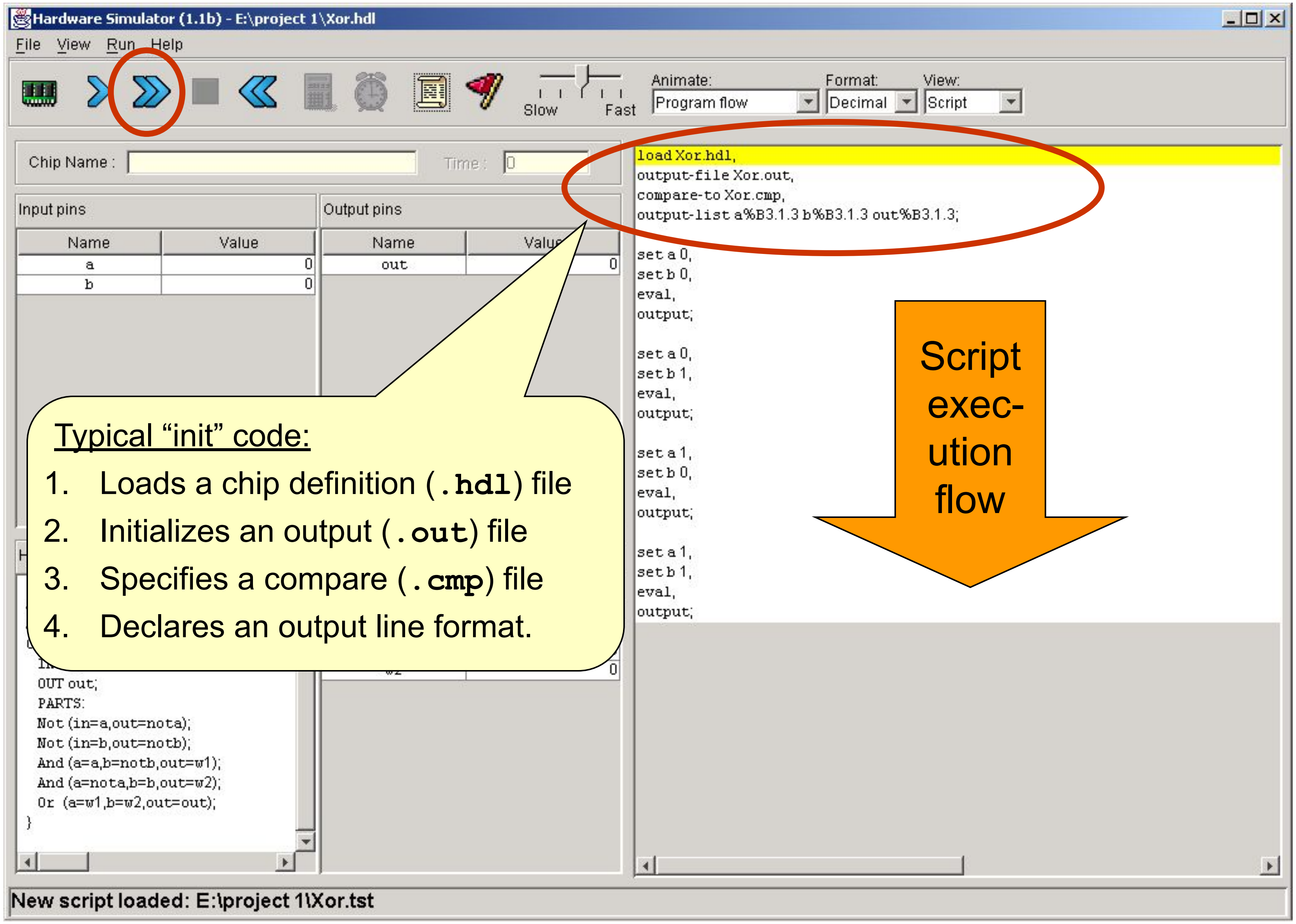
Script Controls


Hardware Simulator (1.1b) - E:\project 1\Xor.hdl
File View Run Help
Slow
Fast
Animate: Program flow
Format: Decimal
View: Script
Chip Name :
Time : 0
Input pins
Output pins
HDL
Controls the script execution speed
Resets the script
Pauses the script execution
Multi-step execution, until a pause
Executes the next simulation step
Script = series of simulation steps, each ending with a semicolon.

```
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;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

New script loaded: E:\project 1\Xor.tst

Running a Script



```
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;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

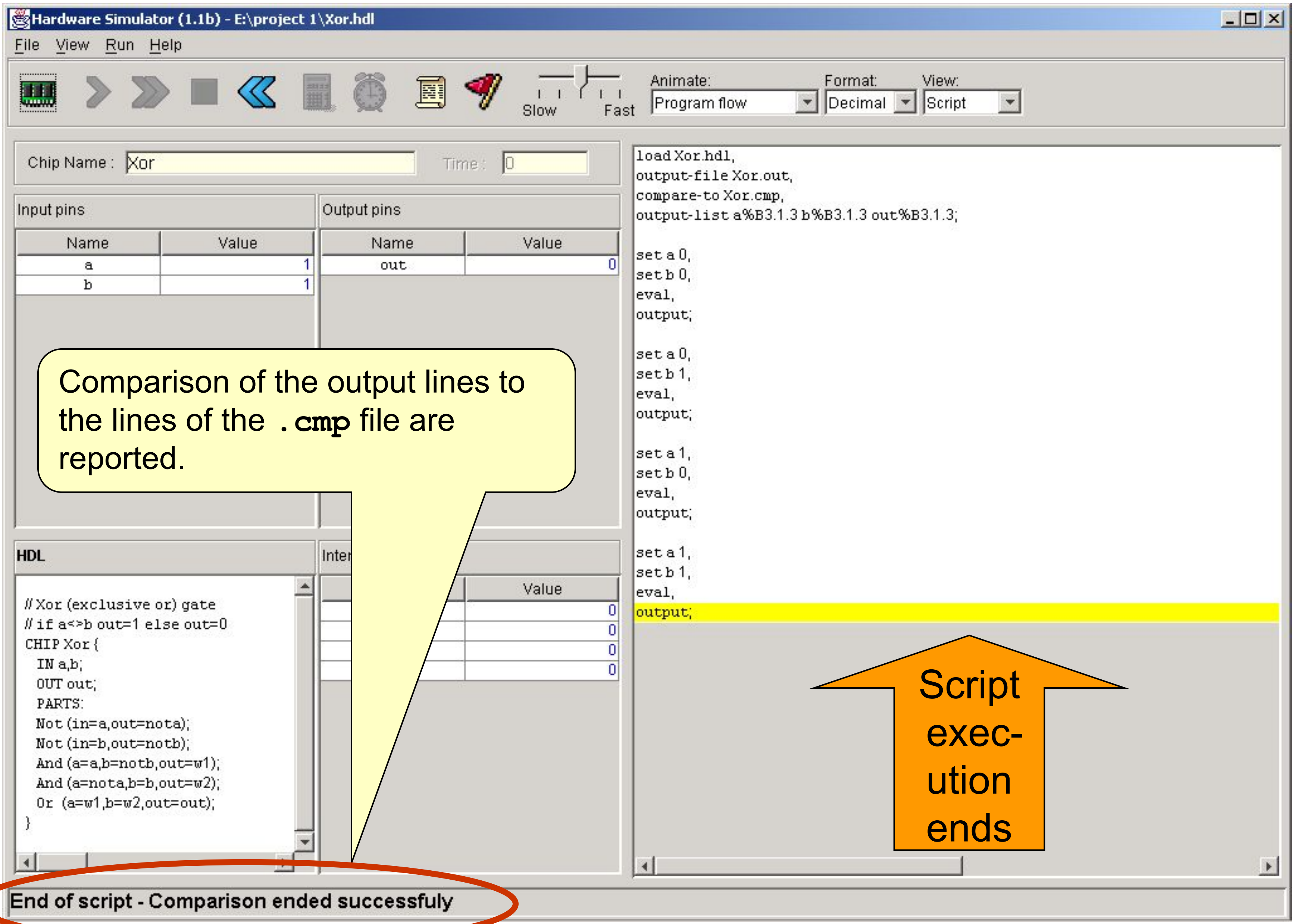
```
OUT out;
PARTS:
Not (in=a,out=nota);
Not (in=b,out=notb);
And (a=a,b=notb,out=w1);
And (a=nota,b=b,out=w2);
Or  (a=w1,b=w2,out=out);
}
```

New script loaded: E:\project 1\Xor.tst

Typical "init" code:

1. Loads a chip definition (`.hdl`) file
2. Initializes an output (`.out`) file
3. Specifies a compare (`.cmp`) file
4. Declares an output line format.

Running a Script



Name	Value
a	1
b	1

Name	Value
out	0

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
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);
}
```

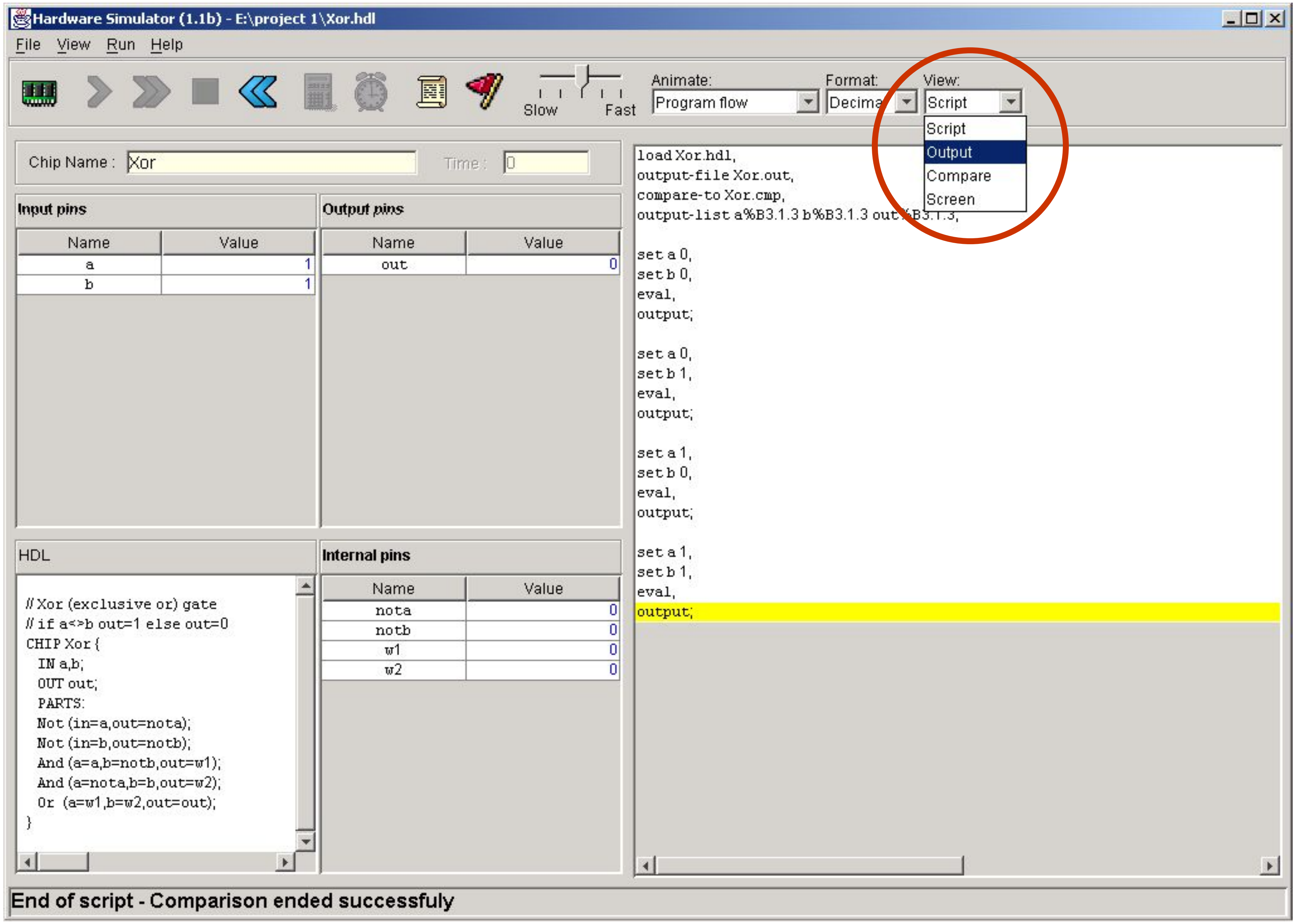
```
load Xor.hdl,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

End of script - Comparison ended successfuly

Comparison of the output lines to the lines of the **.cmp** file are reported.

Script exec-ution ends

Viewing Output and Compare Files



Animate: Program flow
Format: Decimal
View: Script

Chip Name : Xor
Time : 0

Input pins

Name	Value
a	1
b	1

Output pins

Name	Value
out	0

HDL

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
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);
}
```

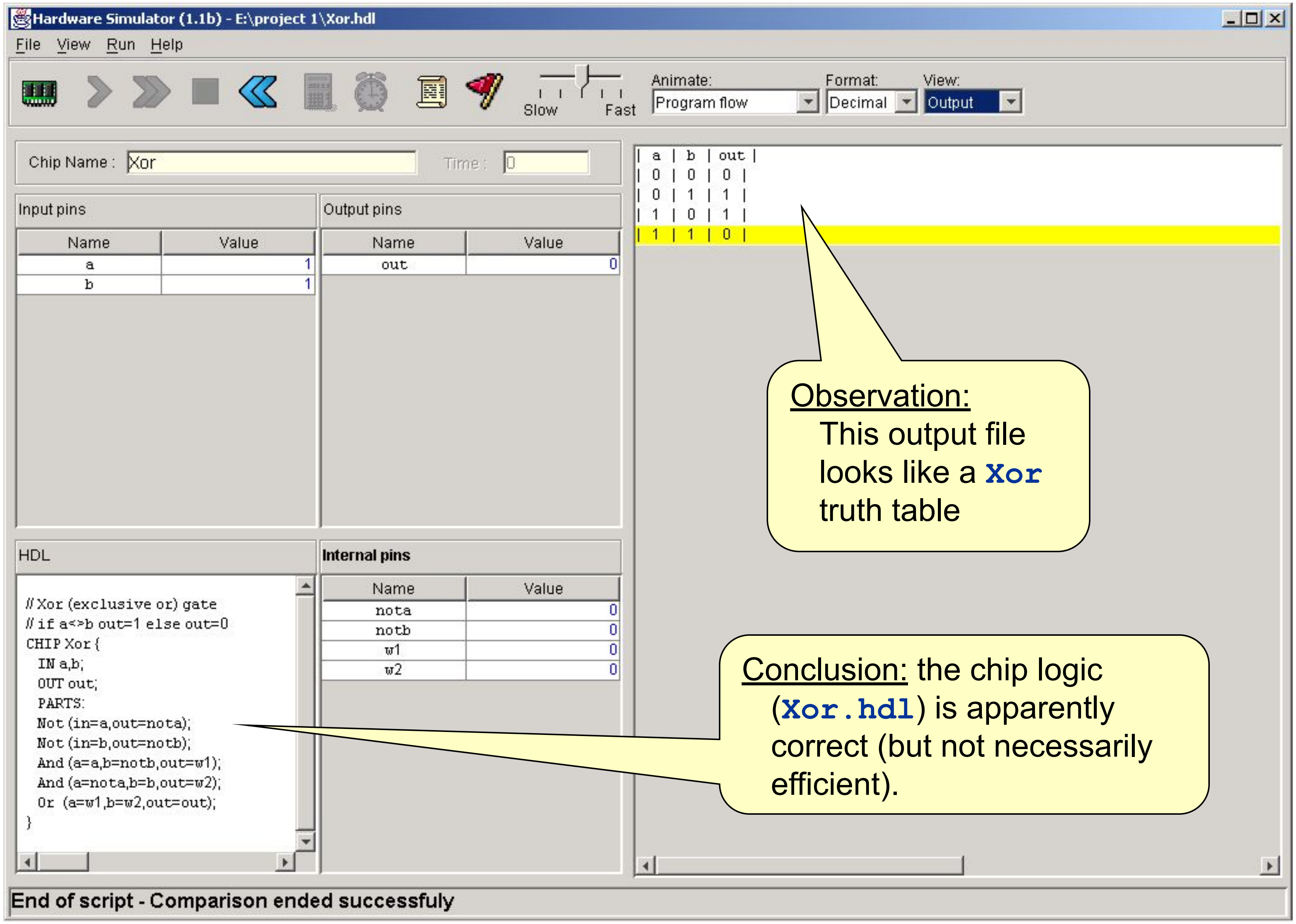
Internal pins

Name	Value
nota	0
notb	0
w1	0
w2	0

```
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;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

End of script - Comparison ended successfuly

Viewing Output and Compare Files



Name	Value
a	1
b	1

Name	Value
out	0

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
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);
}
```

Name	Value
nota	0
notb	0
w1	0
w2	0

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

End of script - Comparison ended successfuly

Observation:
This output file looks like a **Xor** truth table

Conclusion: the chip logic (**Xor.hdl**) is apparently correct (but not necessarily efficient).

Hardware Simulation Tutorial

Built-In Chips

General

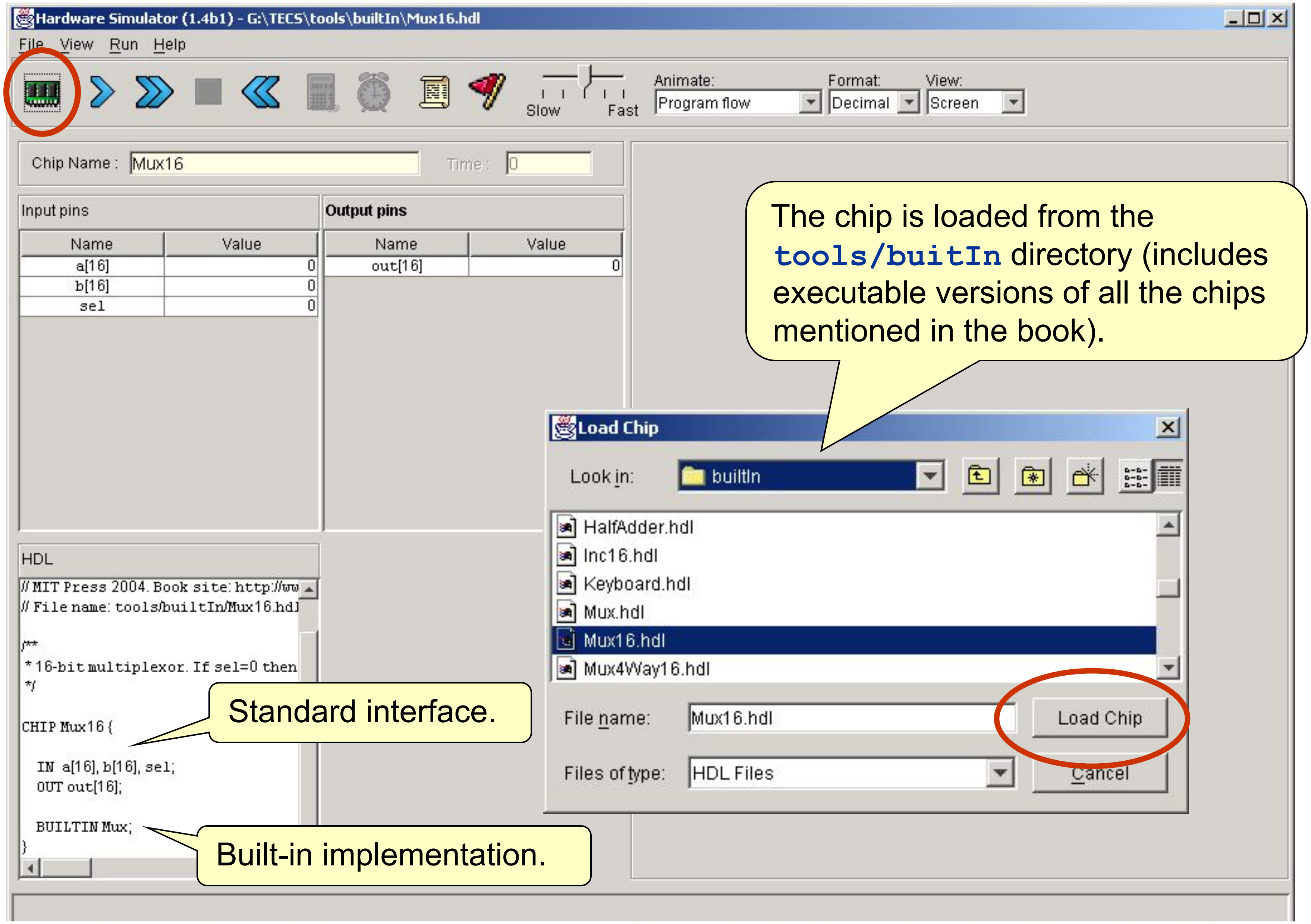
- A built-in chip has an HDL interface and a Java implementation (e.g. here: `Mux16.class`)
- The name of the Java class is specified following the `BUILTIN` keyword
- Built-In implementations of all the chips that appear in he book are supplied in the `tools/buitIn` directory.

```
// 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



Chip Name : Mux16   Time : 0

Input pins

Name	Value
a[16]	0
b[16]	0
sel	0

Output pins

Name	Value
out[16]	0

HDL

```
// MIT Press 2004. Book site: http://ww
// File name: tools/builtIn/Mux16.hdl

/**
 * 16-bit multiplexor. If sel=0 then
 */

CHIP Mux16 {

  IN  a[16], b[16], sel;
  OUT out[16];

  BUILTIN Mux;
}
```

HalfAdder.hdl
Inc16.hdl
Keyboard.hdl
Mux.hdl
Mux16.hdl
Mux4Way16.hdl

File name: Mux16.hdl
Files of type: HDL Files

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.

Hardware Simulation Tutorial

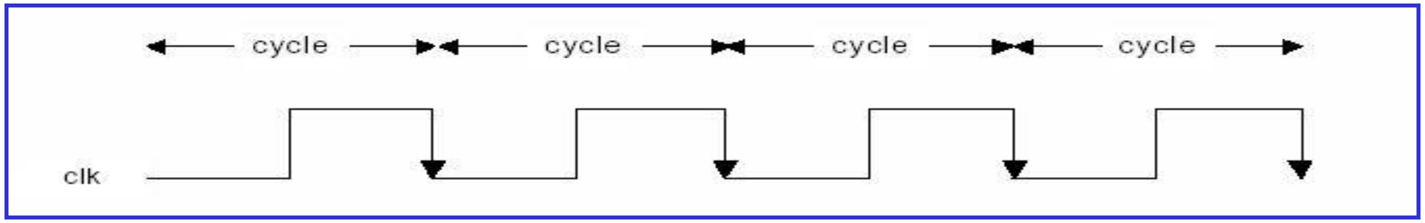
<u>Part IV:</u>

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;
}
```

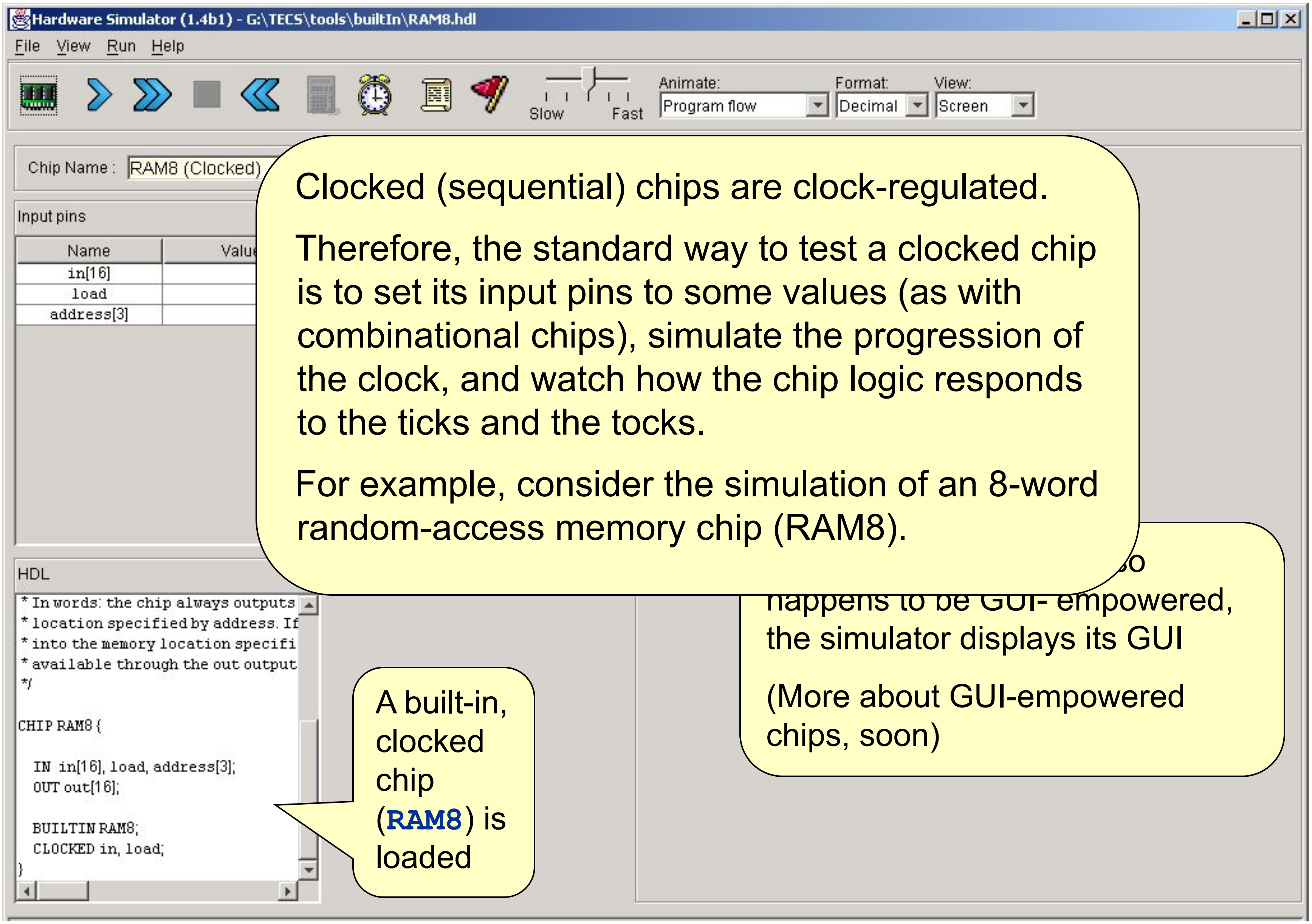
DFF:

- 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



```
* In words: the chip always outputs
* location specified by address. If
* into the memory location specifi
* available through the out output
*/

CHIP RAM8 {

  IN in[16], load, address[3];
  OUT out[16];

  BUILTIN RAM8;
  CLOCKED in, load;
}
```

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.

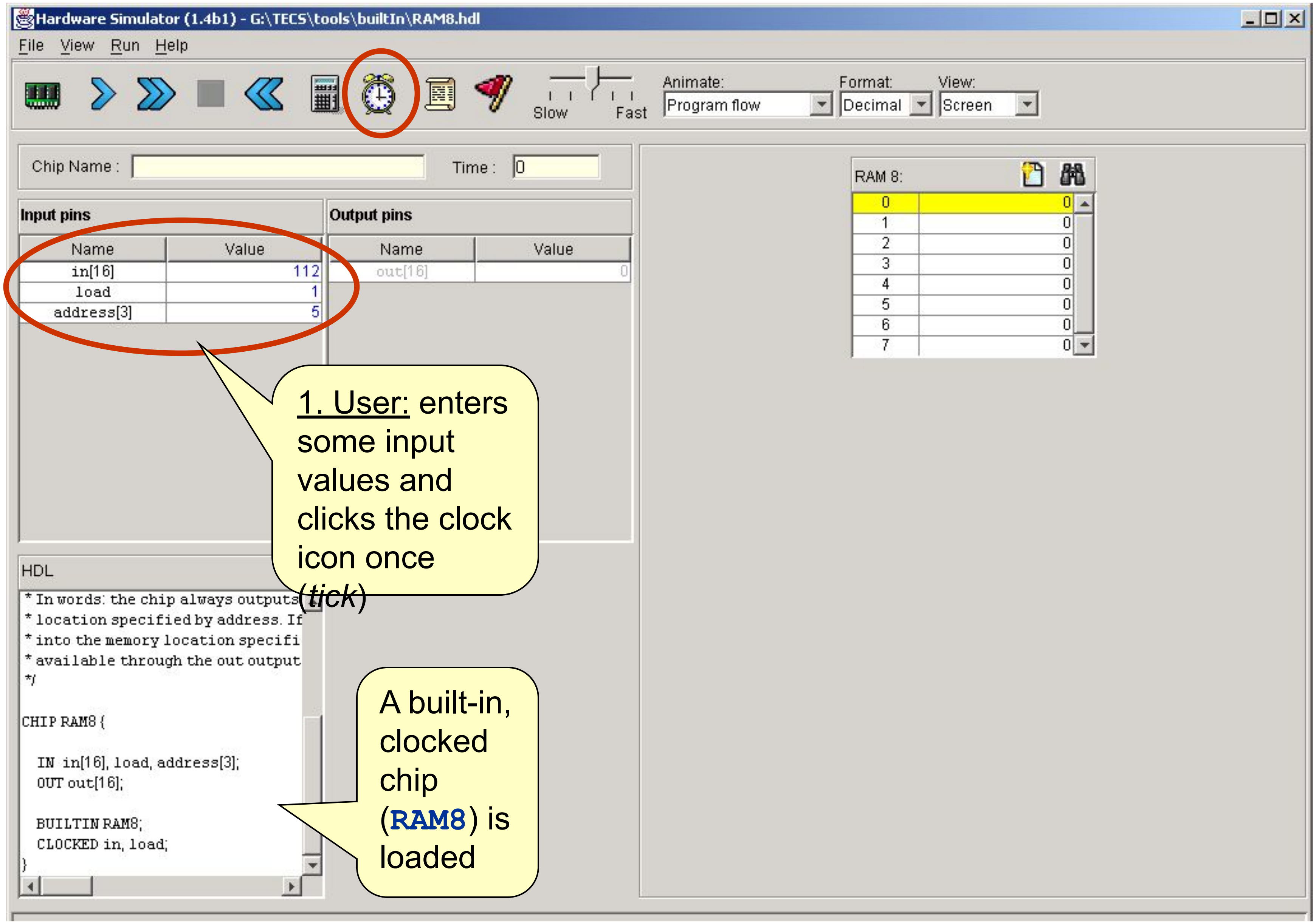
For example, consider the simulation of an 8-word random-access memory chip (RAM8).

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



Hardware Simulator (1.4b1) - G:\TECS\tools\builtIn\RAM8.hdl

Animate: Program flow   Format: Decimal   View: Screen

Chip Name :   Time : 0

Input pins

Name	Value
in[16]	112
load	1
address[3]	5

Output pins

Name	Value
out[16]	0

RAM 8:

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0

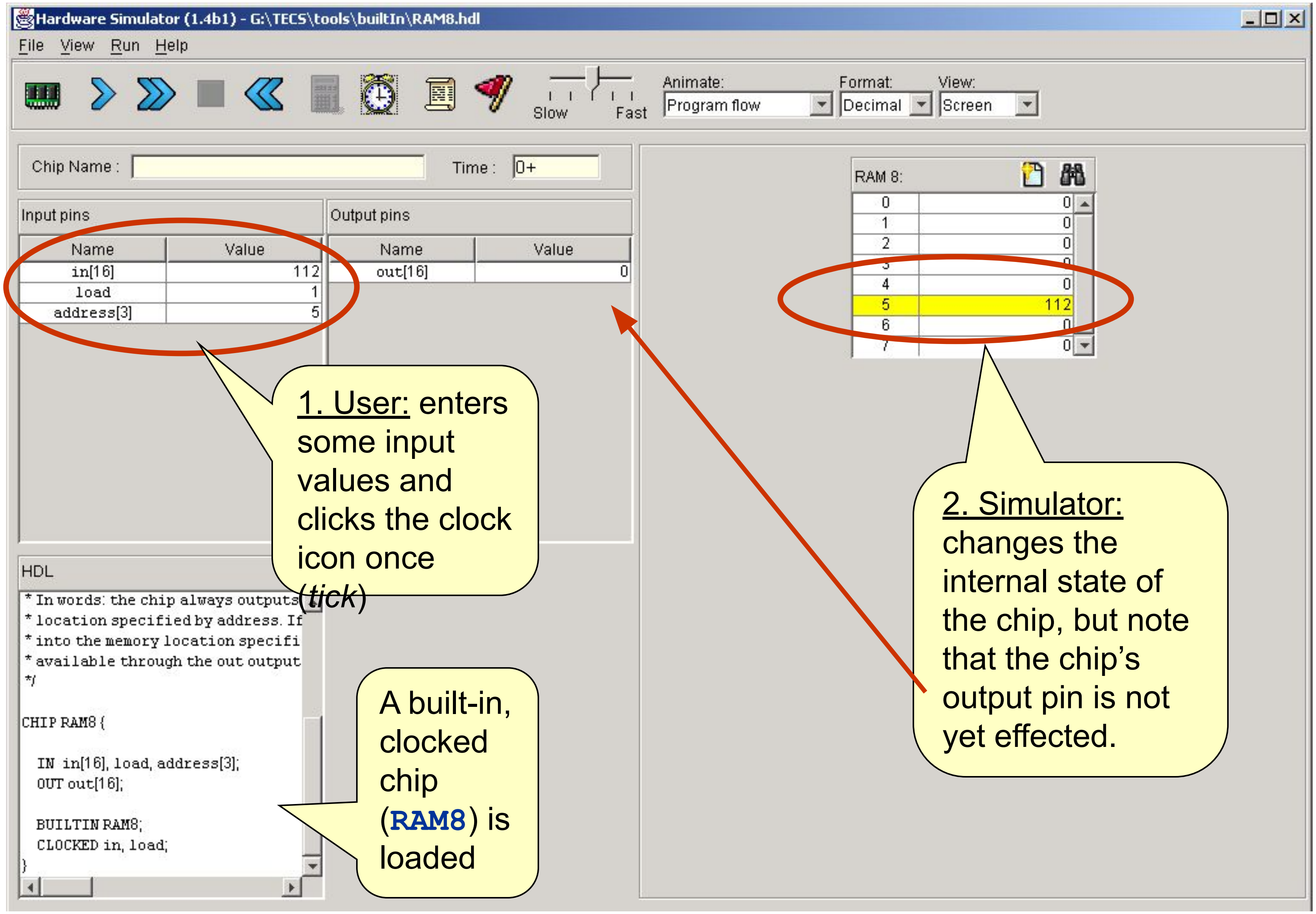
HDL

```
* In words: the chip always outputs
* location specified by address. If
* into the memory location specifi
* available through the out output
*/

CHIP RAM8 {

  IN  in[16], load, address[3];
  OUT out[16];

  BUILTIN RAM8;
  CLOCKED in, load;
}
```

1. User: enters some input values and clicks the clock icon once (*tick*)

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

Simulating Clocked Chips



Hardware Simulator (1.4b1) - G:\TECS\tools\builtIn\RAM8.hdl

Animate: Program flow
Format: Decimal
View: Screen

Chip Name :
Time : 0+

Input pins

Name	Value
in[16]	112
load	1
address[3]	5

Output pins

Name	Value
out[16]	0

RAM 8:

0	0
1	0
2	0
3	0
4	0
5	112
6	0
7	0

HDL

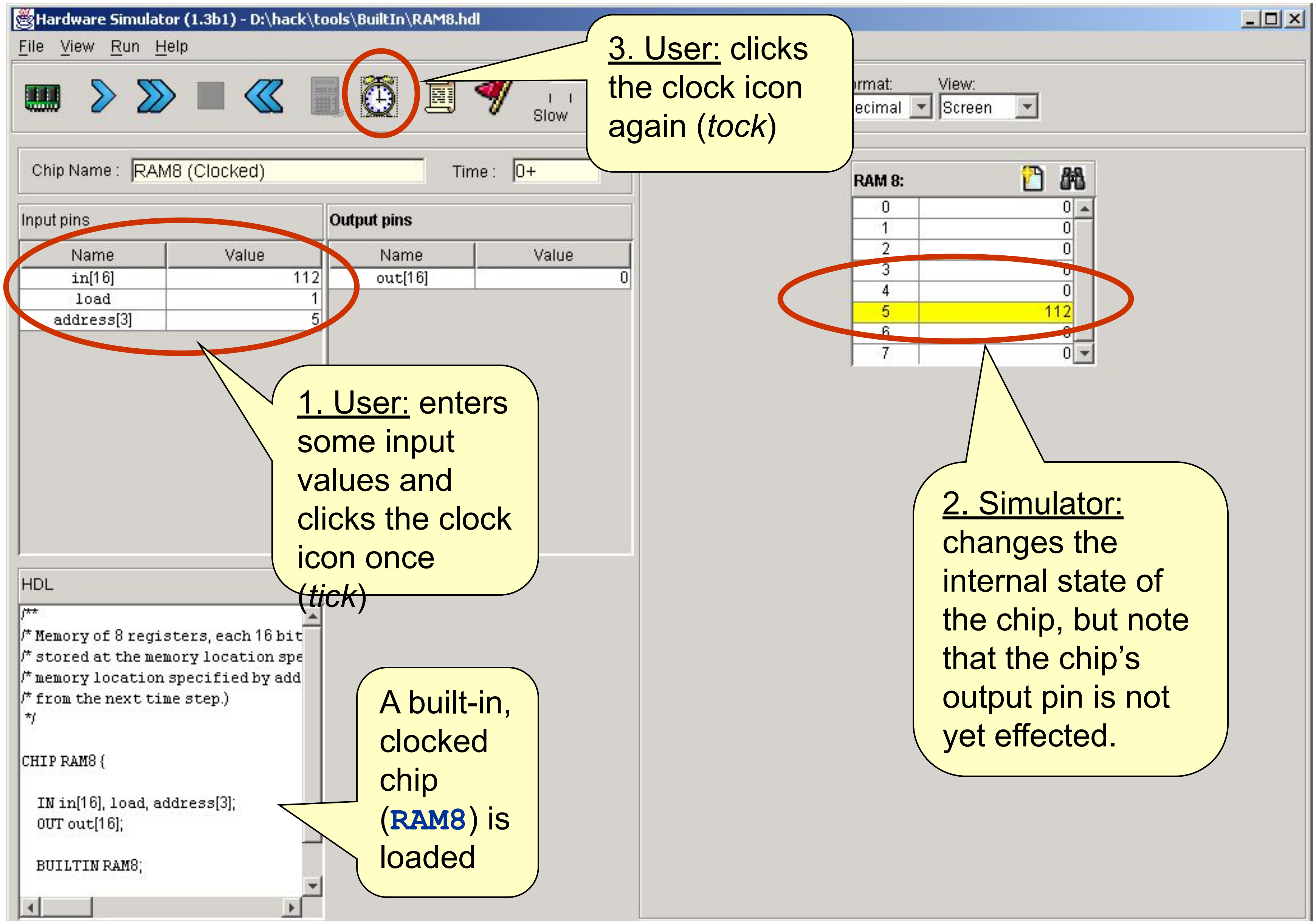
```
* In words: the chip always outputs
* location specified by address. If
* into the memory location specifi
* available through the out output
*/

CHIP RAM8 {

  IN  in[16], load, address[3];
  OUT out[16];

  BUILTIN RAM8;
  CLOCKED in, load;
}
```

1. User: enters some input values and clicks the clock icon once (*tick*)

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

2. Simulator: changes the internal state of the chip, but note that the chip's output pin is not yet effected.

Simulating Clocked Chips



Hardware Simulator (1.3b1) - D:\hack\tools\BuiltIn\RAM8.hdl

File View Run Help

Chip Name : RAM8 (Clocked)   Time : 0+

Format: Decimal   View: Screen

Input pins

Name	Value
in[16]	112
load	1
address[3]	5

Output pins

Name	Value
out[16]	0

RAM 8:

0	0
1	0
2	0
3	0
4	0
5	112
6	0
7	0

HDL

```
/**
/* Memory of 8 registers, each 16 bit
/* stored at the memory location spe
/* memory location specified by add
/* from the next time step.)
 */

CHIP RAM8 {

  IN in[16], load, address[3];
  OUT out[16];

  BUILTIN RAM8;
```

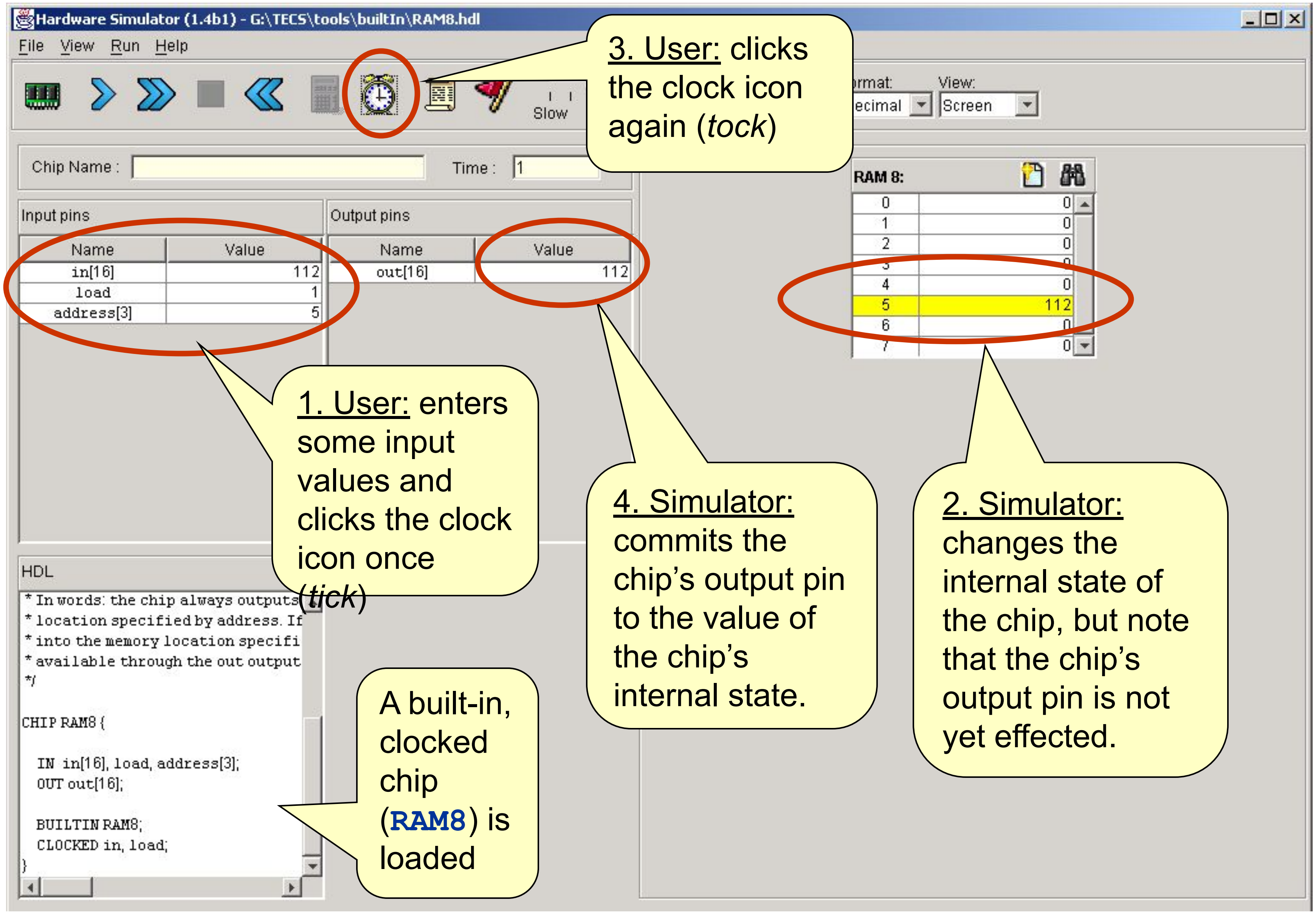
3. User: clicks the clock icon again (*tock*)

1. User: enters some input values and clicks the clock icon once (*tick*)

2. Simulator: changes the internal state of the chip, but note that the chip's output pin is not yet effected.

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

Simulating Clocked Chips



Hardware Simulator (1.4b1) - G:\TECS\tools\builtIn\RAM8.hdl

File View Run Help

Chip Name : Time : 1

Input pins

Name	Value
in[16]	112
load	1
address[3]	5

Output pins

Name	Value
out[16]	112

RAM 8:

0	0
1	0
2	0
3	0
4	0
5	112
6	0
7	0

HDL

```
* In words: the chip always outputs
* location specified by address. If
* into the memory location specifi
* available through the out output
*/

CHIP RAM8 {

  IN  in[16], load, address[3];
  OUT out[16];

  BUILTIN RAM8;
  CLOCKED in, load;
}
```

1. User: enters some input values and clicks the clock icon once (*tick*)

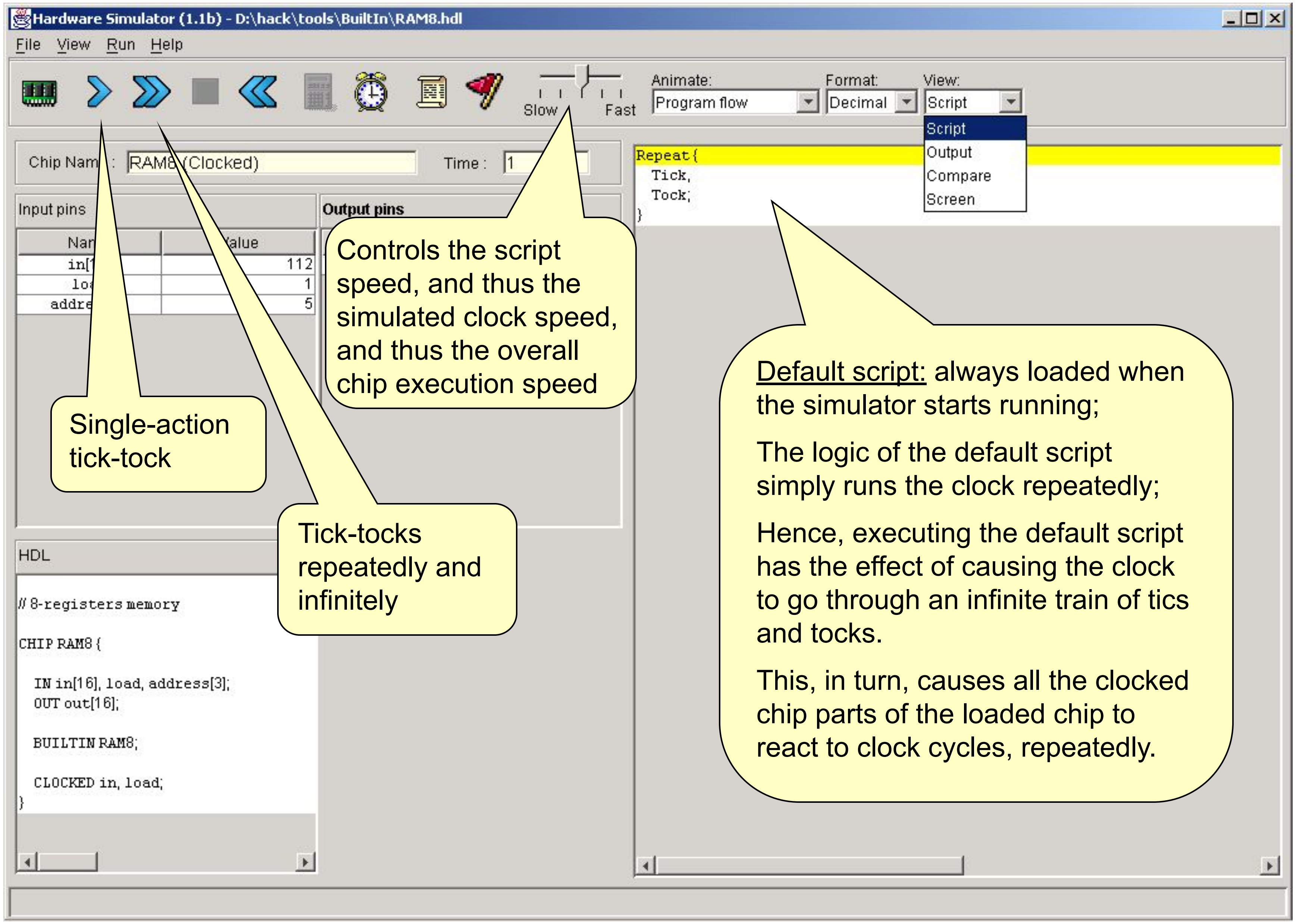
2. Simulator: changes the internal state of the chip, but note that the chip's output pin is not yet effected.

3. User: clicks the clock icon again (*tock*)

4. Simulator: commits the chip's output pin to the value of the chip's internal state.

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

Simulating Clocked Chips Using a Test Script



Name	Value
in[16]	112
load	1
address[3]	5

```
Repeat {
  Tick,
  Tock;
}
```

```
// 8-registers memory

CHIP RAM8 {

  IN in[16], load, address[3];
  OUT out[16];

  BUILTIN RAM8;

  CLOCKED in, load;
}
```

Single-action tick-tock

Tick-tocks repeatedly and infinitely

Controls the script speed, and thus the simulated clock speed, and thus the overall chip execution speed

Default script: always loaded when the simulator starts running;

The logic of the default script simply runs the clock repeatedly;

Hence, executing the default script has the effect of causing the clock to go through an infinite train of tics and tocks.

This, in turn, causes all the clocked chip parts of the loaded chip to react to clock cycles, repeatedly.

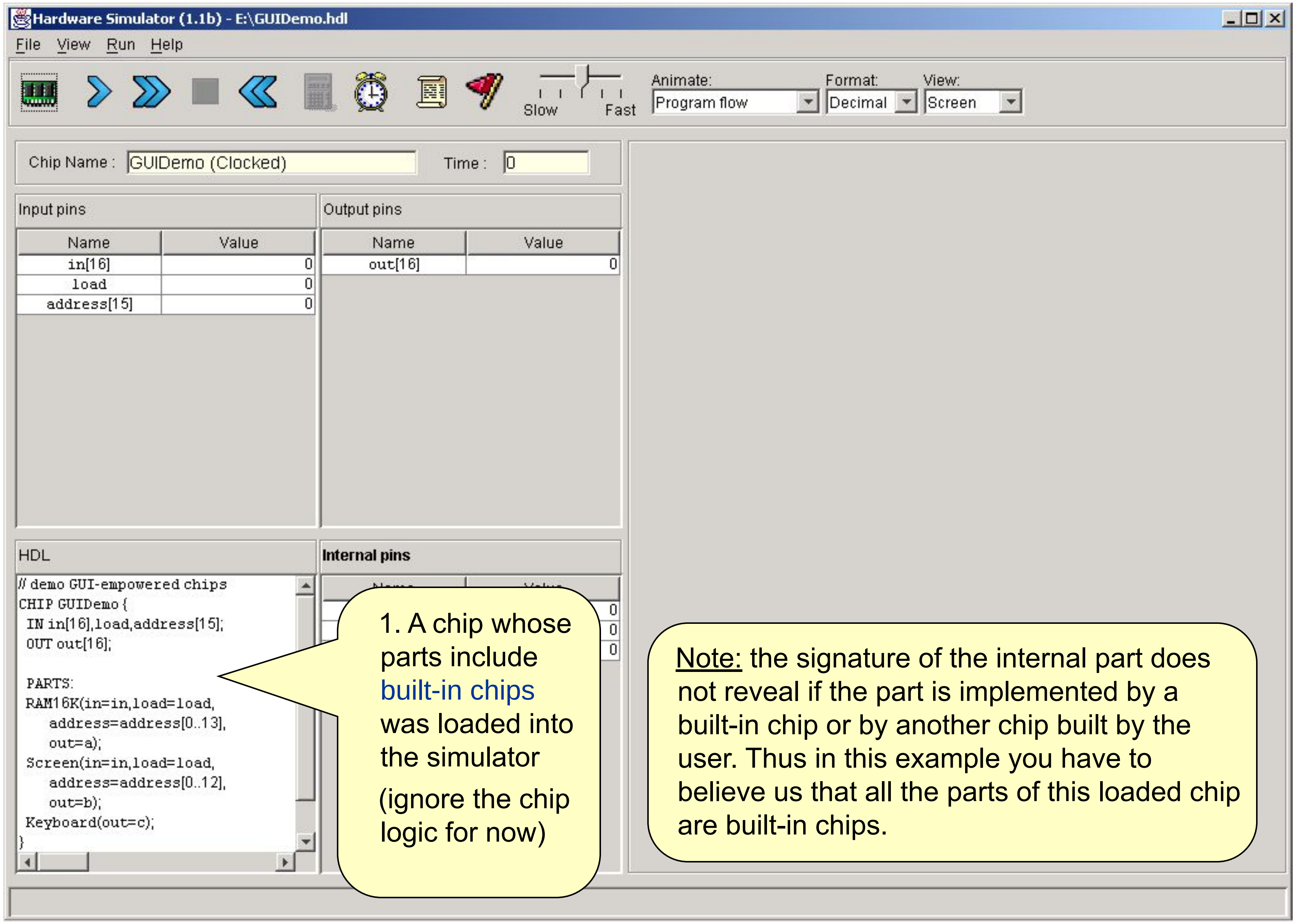
Hardware Simulation Tutorial

Part V:

GUI-Empowered

chips

Built-in Chips with GUI Effects



Chip Name : GUIDemo (Clocked)   Time : 0

Name	Value
in[16]	0
load	0
address[15]	0

Name	Value
out[16]	0

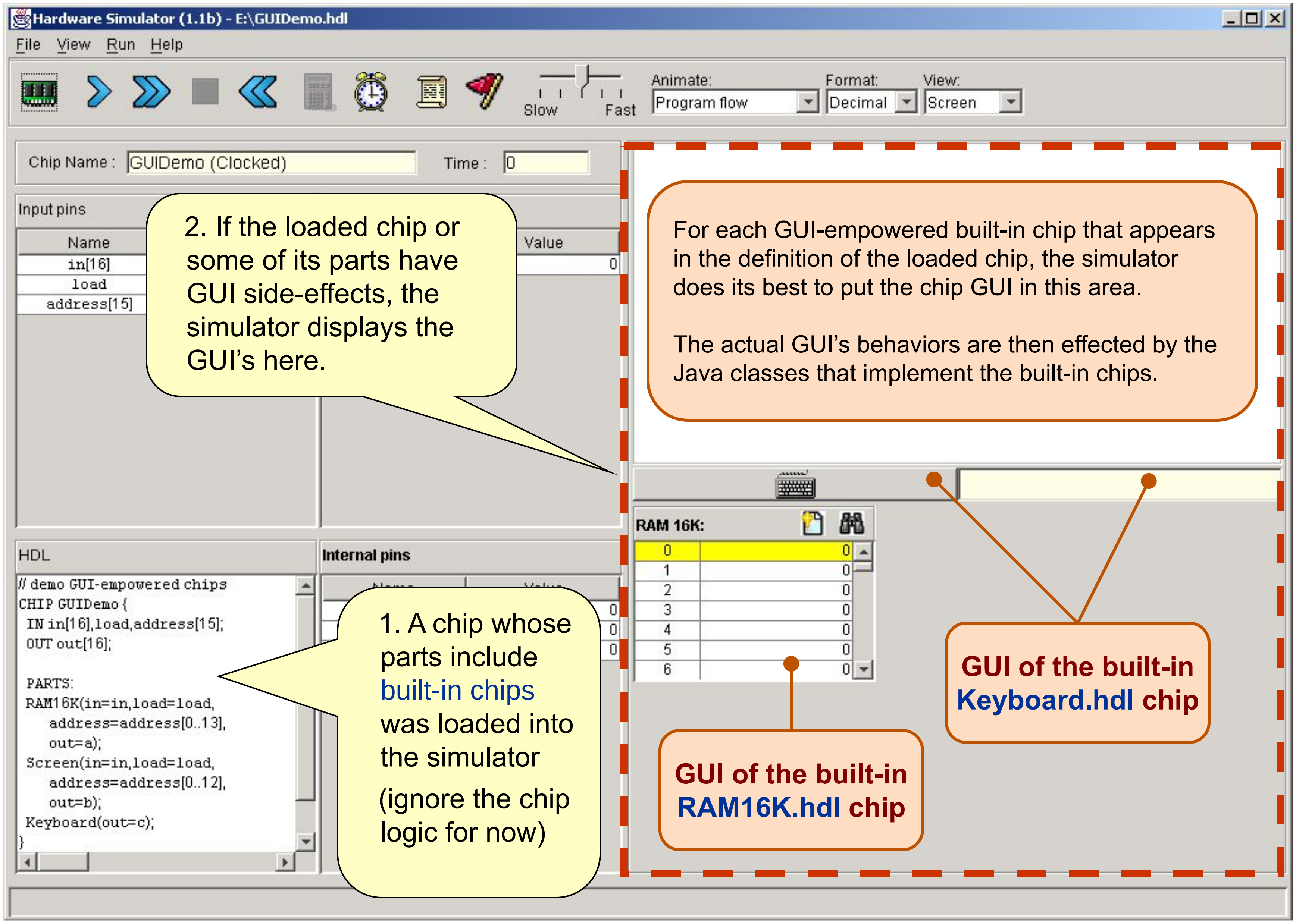
```
// demo GUI-empowered chips
CHIP GUIDemo {
 IN in[16],load,address[15];
 OUT out[16];

 PARTS:
 RAM16K(in=in,load=load,
     address=address[0..13],
     out=a);
 Screen(in=in,load=load,
     address=address[0..12],
     out=b);
 Keyboard(out=c);
}
```

1. A chip whose parts include built-in chips was loaded into the simulator

(ignore the chip logic for now)

Note: the signature of the internal part does not reveal if the part is implemented by a built-in chip or by another chip built by the user. Thus in this example you have to believe us that all the parts of this loaded chip are built-in chips.

Built-in Chips with GUI Effects



```
// demo GUI-empowered chips
CHIP GUIDemo {
 IN in[16],load,address[15];
 OUT out[16];

 PARTS:
 RAM16K(in=in,load=load,
    address=address[0..13],
    out=a);
 Screen(in=in,load=load,
    address=address[0..12],
    out=b);
 Keyboard(out=c);
}
```

0	0
1	0
2	0
3	0
4	0
5	0
6	0

1. A chip whose parts include built-in chips was loaded into the simulator

(ignore the chip logic for now)

2. If the loaded chip or some of its parts have GUI side-effects, the simulator displays the GUI's here.

For each GUI-empowered built-in chip that appears in the definition of the loaded chip, the simulator does its best to put the chip GUI in this area.

The actual GUI's behaviors are then effected by the Java classes that implement the built-in chips.

GUI of the built-in RAM16K.hdl chip

GUI of the built-in Keyboard.hdl chip

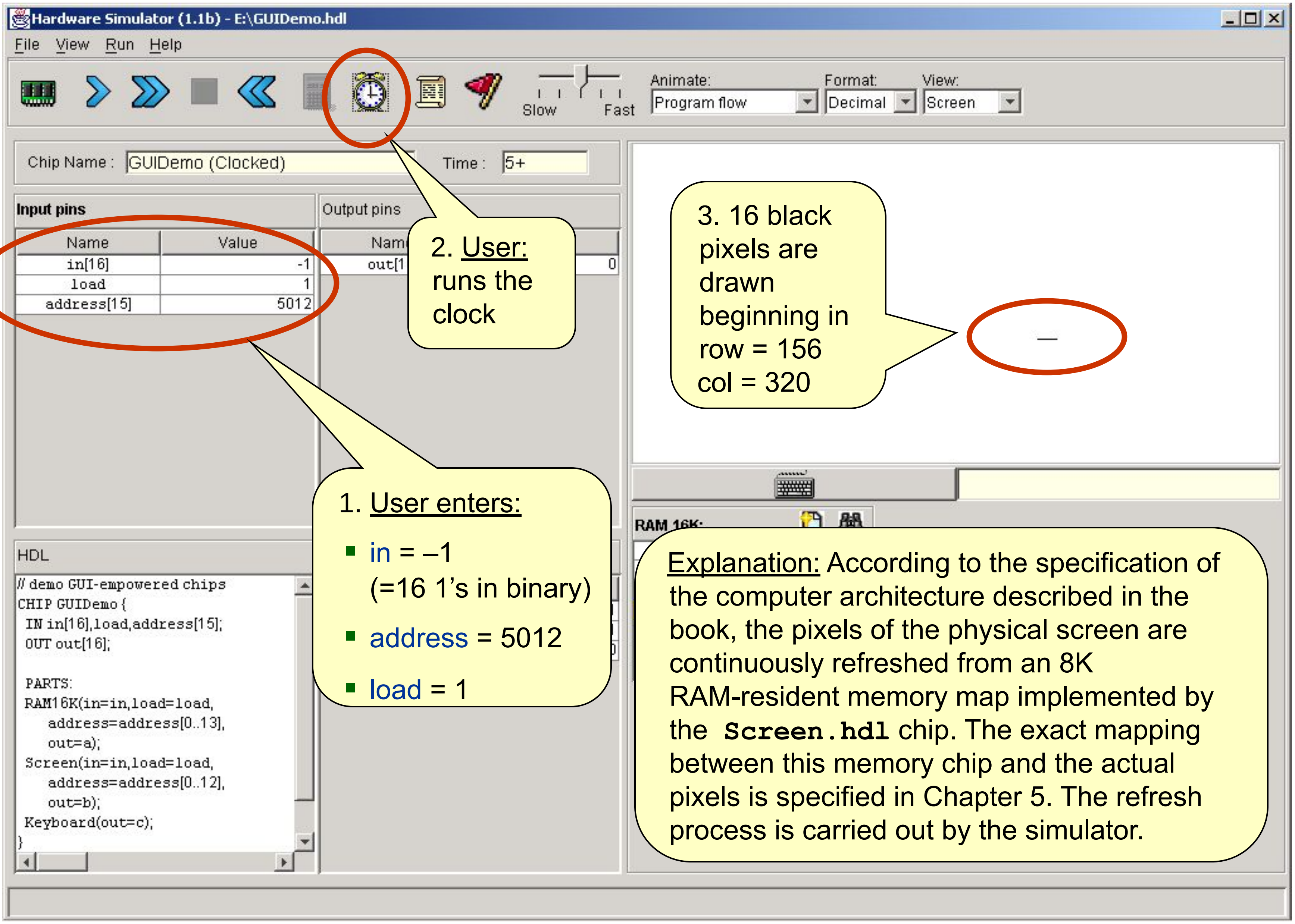
The Logic of the GUIDemo Chip

```
// 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



Chip Name : GUIDemo (Clocked)

Time : 5+

Name	Value
in[16]	-1
load	1
address[15]	5012

Name	Value
out[1	0

```
// demo GUI-empowered chips
CHIP GUIDemo {
 IN in[16],load,address[15];
 OUT out[16];

 PARTS:
 RAM16K(in=in,load=load,
     address=address[0..13],
     out=a);
 Screen(in=in,load=load,
     address=address[0..12],
     out=b);
 Keyboard(out=c);
}
```

1. User enters:
   - in = –1 (=16 1's in binary)
   - address = 5012
   - load = 1

2. User: runs the clock

3. 16 black pixels are drawn beginning in row = 156 col = 320

Explanation: According to the specification of the computer architecture described in the book, the pixels of the physical screen are continuously refreshed from an 8K RAM-resident memory map implemented by the **Screen.hdl** chip. The exact mapping between this memory chip and the actual pixels is specified in Chapter 5. The refresh process is carried out by the simulator.

Hardware Simulation Tutorial

<u>Part VI:</u>

Debugging tools

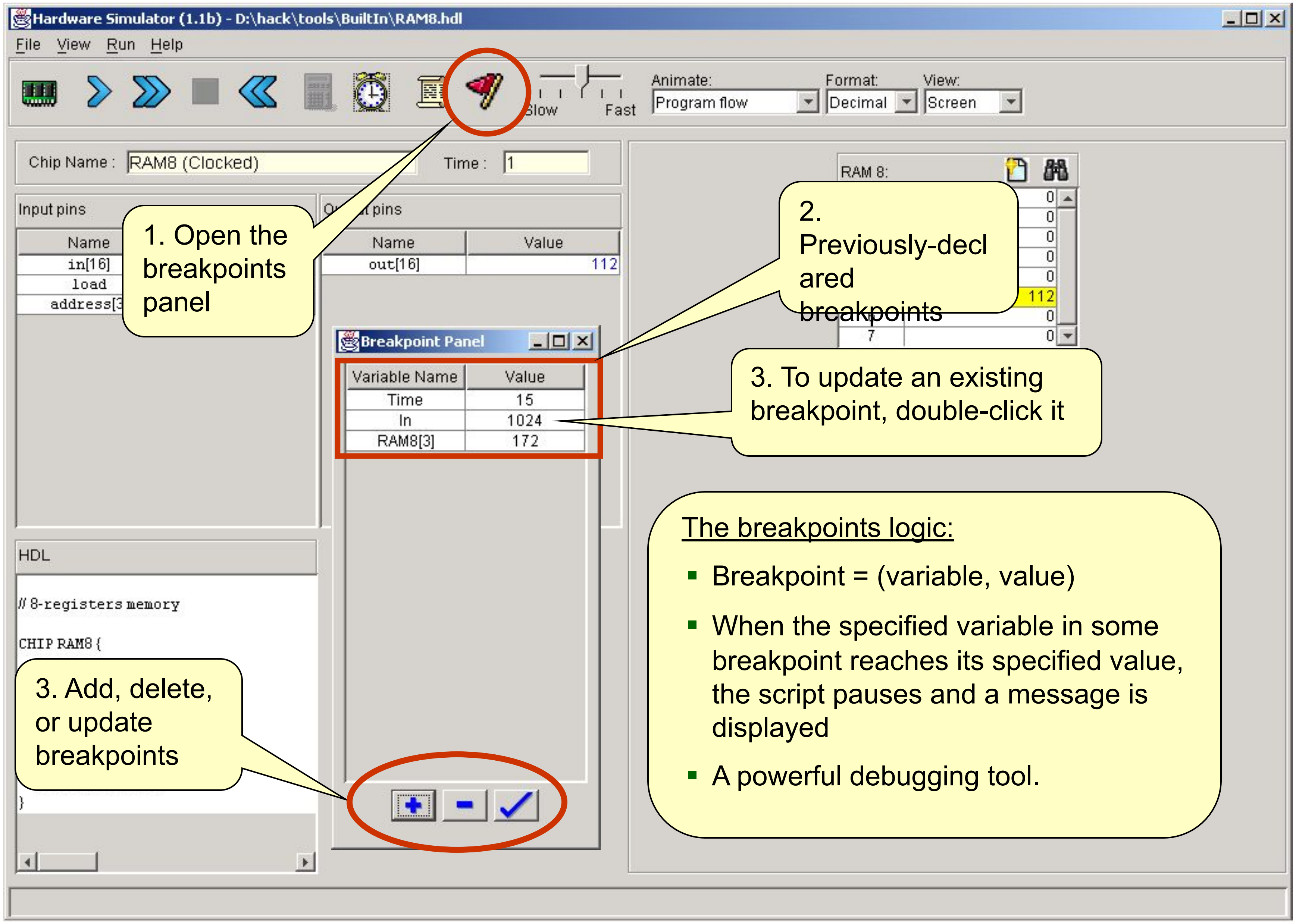
System Variables

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



Chip Name : RAM8 (Clocked)   Time : 1

Name	Value
in[16]	
load	
address[3	

Name	Value
out[16]	112

```
// 8-registers memory

CHIP RAM8 {
}
```

Variable Name	Value
Time	15
In	1024
RAM8[3]	172

1. Open the breakpoints panel

2. Previously-declared breakpoints

3. To update an existing breakpoint, double-click it

3. Add, delete, or update breakpoints

The breakpoints logic:

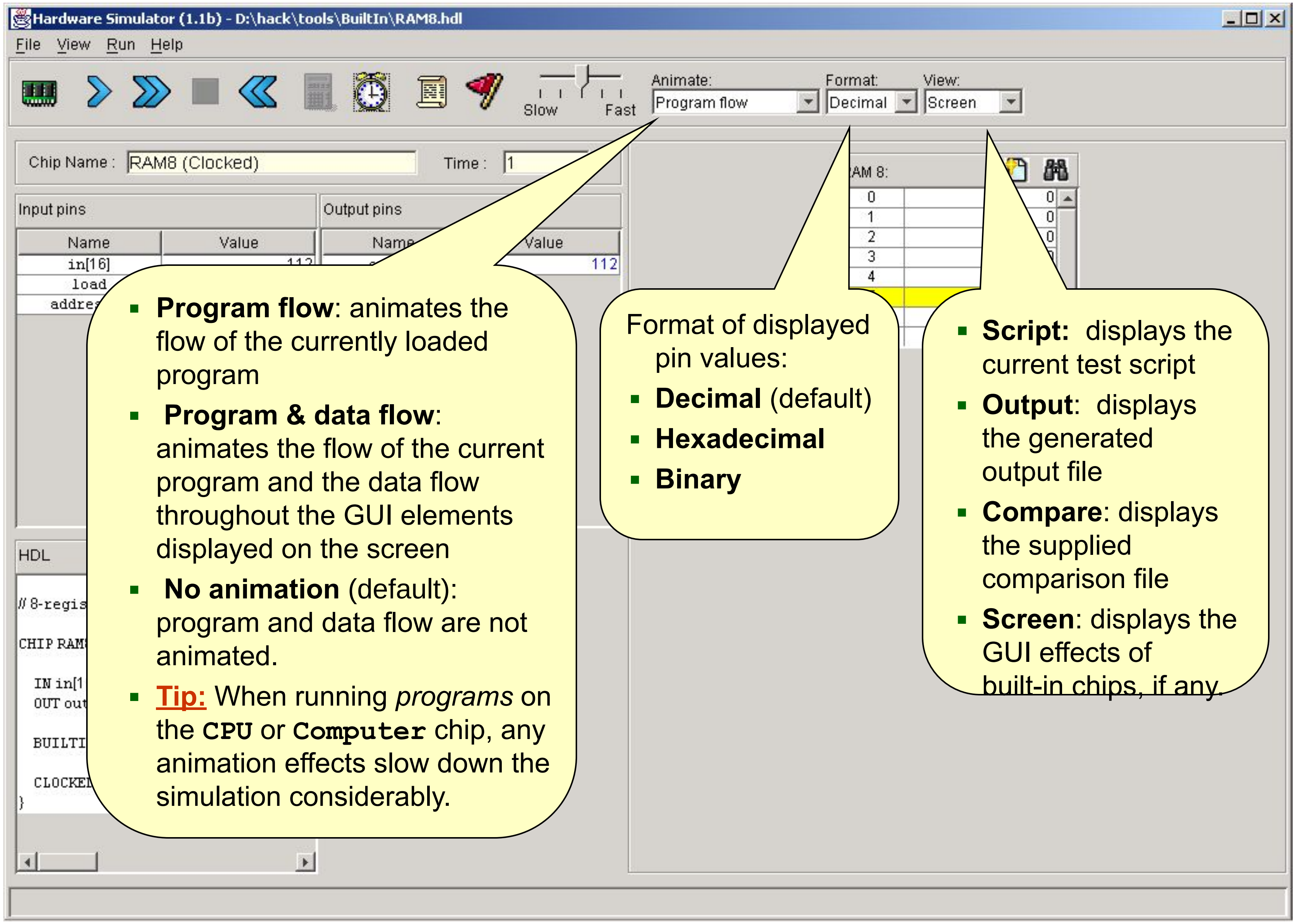
- Breakpoint = (variable, value)
- When the specified variable in some breakpoint reaches its specified value, the script pauses and a message is displayed
- A powerful debugging tool.

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.

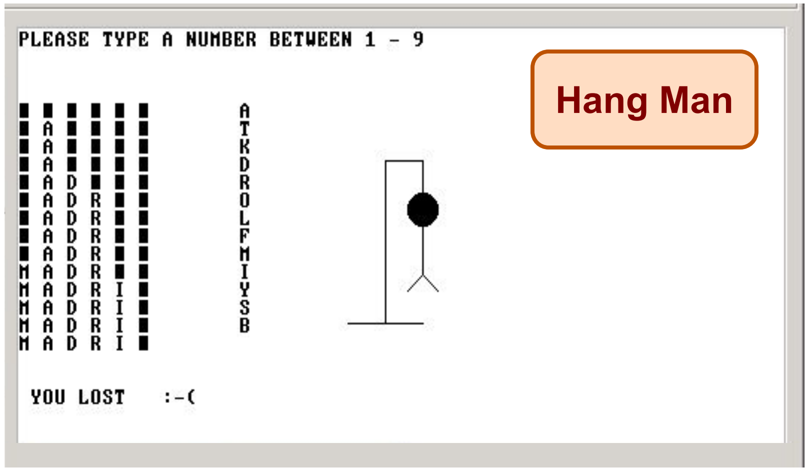
Hardware Simulation Tutorial

Part VII:

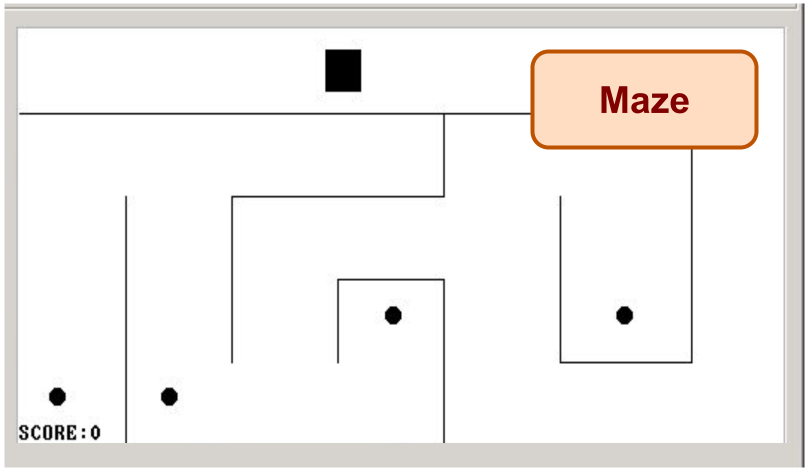
The Hack Hardware Platform

Hack: a General-Purpose 16-bit Computer

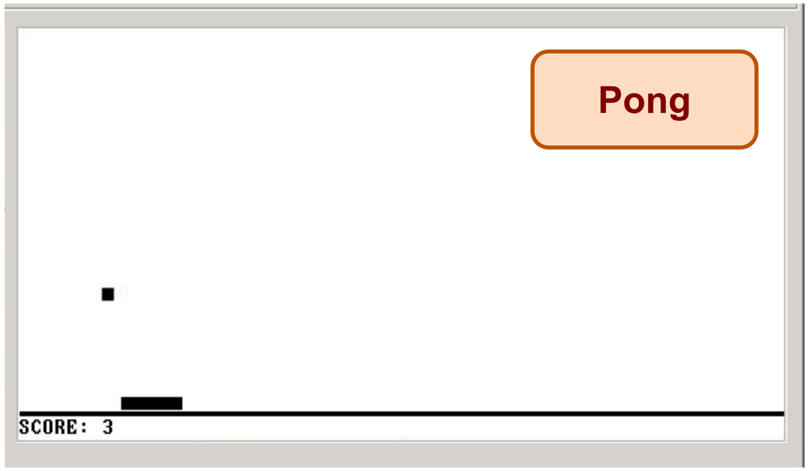
Sample applications running on the Hack computer:



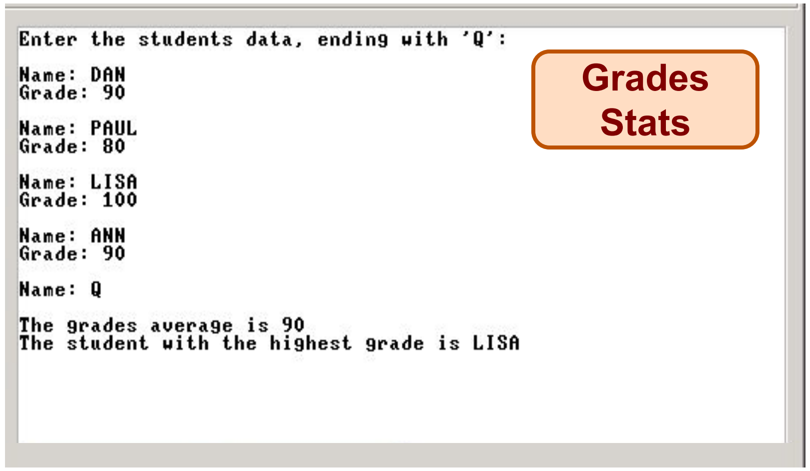
```
PLEASE TYPE A NUMBER BETWEEN 1 - 9

YOU LOST    :-(
```



```
SCORE:0
```



```
SCORE: 3
```



```
Enter the students data, ending with 'Q':

Name: DAN
Grade: 90

Name: PAUL
Grade: 80

Name: LISA
Grade: 100

Name: ANN
Grade: 90

Name: Q

The grades average is 90
The student with the highest grade is LISA
```

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.