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](http://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 hardware simulator is part of the software suite that accompanies the book “The Elements of Computing”. The book evolves around the construction of a complete computer system, done in the framework of a 1- or 2-semesters course.

In the first part of the book, we build all the logic gates and chips-set of a simple yet powerful computer, called Hack. This is done using HDL (Hardware Description Language) and the hardware simulator described in this tutorial.

In the second part of the book, 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 simple operating system, written in Jack.

The book is completely self-contained, requiring only programming as a pre-requisite.

The book’s software suite includes some 200 test programs, test scripts, and all the software tools necessary for doing the projects.
The book’s software suite

Simulators

(HardwareSimulator, CPUEmulator, VMEmulator):

- All the tools are dual-platform:
  - Xxx.bat: starts Xxx in Windows;
  - Xxx.sh: starts Xxx in Unix.

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.

This tutorial focuses on the Hardware Simulator.
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.

Indeed, that’s how hardware engineers build chips for real: first, the hardware is designed, tested, and optimized on a simulator. Only then, the resulting gate logic is committed to silicon.
I. Getting started

II. Test scripts

III. Built-in chips

IV. Clocked chips

V. GUI-empowered chips

VI. Debugging tools

VII. The Hack Platform

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)

- **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)

/** 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

1. To load a new chip description, click the Load Chip button.

2. Navigate to a directory and select an .hdl file.
Loading a chip

- Names and current values of the chip’s **input pins**;
- The pin values may be changed by the user.

- Names and current values of the chip’s **output pins**;
- Calculated by the simulator; read-only.

- Names and current values of the chip’s **internal pins** (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 chip logic;
- To edit it, use an external text editor.
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` button.
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` button.
3. **User**: Clicked the `eval` button
4. **Simulator**: re-calculates the values of the chip’s internal and output pins (done by applying 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

Test scripts:

- Are used for specifying, automating and replicating chip testing
- Are supplied for every chip mentioned in the book
- 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.

Test scripts:

```plaintext
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;
```

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

Generated output file (Xor.out)
Loading a test script

To load a script (.tst file), click the *Load Script* 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

- **Executes the next simulation step**
- **Executes the entire script, until a pause**
- **Pauses the script execution**
- **Resets the script**
- **Controls the script execution speed**

Logical simulation steps, each ending with a semicolon.
Running a script

Typical script’s “init” code:

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

When the script execution ends, the simulator reports the comparison results between the output file and the compare file.
Viewing output and compare files

To view the generated output (.out) file or the compare (.cmp) file, choose “output” or “compare” from the View button.
Observation: This output file looks like a Xor truth table.

Conclusion: The chip logic (Xor.hdl) is apparently correct (but not necessarily efficient).
Part III:
Built-in Chips
Built-In chips

General

- A built-in chip has an HDL interface and a Java implementation (e.g. `Mux16.class`)
- The name of the Java class is specified following the `BUILTIN` keyword
- Built-In versions of all the chips that appear in the book are supplied with the HW simulator, and stored in the `tools/builtIn` directory.

Built-in chips are used to:

- Implement primitive gates (in the computer built in the book: `Nand` and `DFF`)
- Implement chips that have peripheral side effects (like I/O devices)
- Implement chips that feature a GUI (for debugging)
- Replace (automatically) chips that the user didn’t implement for some reason
- Improve simulation speed and save memory (when used as parts in complex chips)
- Facilitate behavioral simulation of a chip prior to actually building it in HDL
- Built-in chips can be used either explicitly or implicitly, as we now turn to show.
Explicit use of built-in chips

Built-in chip logic. Can be used as-is or as a part in other chips, without having to implement it in HDL.

Executable versions of all the chips mentioned in the book are stored in the `tools/builtIn` directory.
Implicit use of built-in chips

```c
/** 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 an HDL file is loaded (e.g. `Xor.hdl`), the simulator parses its definition. For each internal chip `Xxx(...)` mentioned in the PARTS section, the simulator looks for a `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 (presumably, 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 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 the computer’s internal clock:

  ![Clock Cycle Diagram](image)

  - In our jargon, a clock cycle = *tick*-phase (down), followed by a *tock*-phase (up)
  - 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

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.

/** 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.
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, let us consider the simulation of an 8-word random-access memory chip (RAM8). A built-in, clocked chip (RAM8) is loaded. Since this built-in chip also happens to be GUI-empowered, the simulator displays its GUI (More about GUI-empowered chips, soon)
Simulating clocked chips

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

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

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

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)

<table>
<thead>
<tr>
<th>Input pins</th>
<th>Output pins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Value</td>
</tr>
<tr>
<td>in[16]</td>
<td>112</td>
</tr>
<tr>
<td>load</td>
<td>1</td>
</tr>
<tr>
<td>address[3]</td>
<td>5</td>
</tr>
</tbody>
</table>

A built-in, clocked chip (RAM8) is loaded.
Simulating clocked chips

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

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.
Simulating clocked chips using a test script

Single-action tick-tock

Tick-tocks repeatedly and infinitely

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 respond to clock cycles, repeatedly.
Part V:
GUI-Empowered chips
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.
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.

1. A chip whose parts include built-in chips was loaded into the simulator (ignore the chip logic for now)
The logic of the GUIDemo chip

```c
// 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);
}
```

- **Effect:** When the simulator evaluates this chip, it displays the GUI side-effects of all the 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.
1. User enters:
   - $\text{in} = -1$ (=16 1's in binary)
   - $\text{address} = 5012$
   - $\text{load} = 1$

2. User: runs the clock

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

A simulated 256 rows by 512 columns screen

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.
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

1. Open the breakpoints panel

2. Previously-declared breakpoints

3. Add, delete, or update breakpoints

The breakpoints logic:
- Breakpoint = variable name and declared value
- When the specified variable in some breakpoint reaches its specified value, the script pauses and a message is displayed
- A powerful debugging tool.

3. To update an existing breakpoint, double-click it
Test scripts of complex chips

Test scripts are normally written and supplied by the hardware platform architects

All the chips that you have to develop in the course have supplied test scripts

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 ROM 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, 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 is a general-purpose 16-bit computer

Sample applications running on the Hack computer:

- **Hangman**
- **Maze**
- **Pong**
- **Grades Stats**

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

**Combinational chips**
- HalfAdder
- FullAdder
- Add16
- Inc16
- ALU

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

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

Most of these chips are generic, meaning that you need them for the construction of any computer.

The Hack chip-set and hardware platform can be built using the HW simulator in recursive ascent, 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).