Project 1: Elementary Logic Gates

CS 220

Background

A typical computer architecture is based on a set of elementary logic gates like `And`, `Or`, `Mux`, etc., as well as their bit-wise versions `And16`, `Or16`, `Mux16`, etc. (assuming a 16-bit machine). This project engages you in the construction of a typical set of basic logic gates. These gates form the elementary building blocks from which more complex chips will be later constructed.

Objective

Build all the logic gates described in Chapter 1 (see list below), yielding a basic chip-set. The only building blocks that you can use in this project are primitive `Nand` gates and the composite gates that you will gradually build on top of them.

Contract

When loaded into the supplied Hardware Simulator, your chip design (modified `.hdl` program), tested on the supplied `.tst script`, should produce the outputs listed in the supplied `.cmp` file. If that is not the case, the simulator will let you know.

Resources

The relevant reading for this project is Chapter 1 and Appendix A, sections 1 through 6. Specifically, all the chips described in Chapter 1 should be implemented in the Hardware Description Language (HDL) specified in Appendix A. Another resource that you will find handy in this and in all subsequent hardware projects is this *HDL Survival Guide*, written by Mark Armbrust.

For each chip, we supply a skeletal `.hdl` file with a place holder for a missing implementation part. In addition, for each chip we supply a `.tst` script that instructs the hardware simulator how to test it, and a `.cmp` ("compare file") containing the correct output that this test should generate. Your job is to complete and test the supplied skeletal `.hdl` files.

You'll find the necessary project directory and files in a ZIP file on the course web site. To get acquainted with the Hardware Simulator, go through parts I through III of the supplied *Hardware Simulator Tutorial*.

Chips

Chip (HDL)	Description	Test Script	Compare File
Nand	Nand gate (primitive)		
Not	Not gate	Not.tst	Not.cmp
And	And gate	And.tst	And.cmp
Or	Or gate	Or.tst	Or.cmp
Xor	Xor gate	Xor.tst	Xor.cmp
Mux	Mux gate	Mux.tst	Mux.cmp
DMux	DMux gate	DMux.tst	DMux.cmp
Not16	16-bit Not	Not16.tst	Not16.cmp
And16	16-bit And	And16.tst	And16.cmp
Or16	16-bit Or	Or16.tst	Or16.cmp
Mux16	16-bit multiplexor	Mux16.tst	Mux16.cmp
Or8Way	Or(in0,in1,...,in7)	Or8Way.tst	Or8Way.cmp
Mux4Way16	16-bit/4-way mux	Mux4Way16.tst	Mux4Way16.cmp
Mux8Way16	16-bit/8-way mux	Mux8Way16.tst	Mux8Way16.cmp
DMux4Way	4-way demultiplexor	DMux4Way.tst	DMux4Way.cmp
DMux8Way	8-way demultiplexor	DMux8Way.tst	DMux8Way.cmp

Tips

Built-in chips: The `Nand` gate is considered primitive and thus there is no need to implement it: whenever a `Nand` chip-part is encountered in your HDL code, the simulator automatically invokes the built-in `tools/builtInChips/Nand.hdl` implementation. We recommend implementing the other gates in this project in the order in which they appear in Chapter 1. However, note that the simulator's environment includes a library with built-in versions of all these chips. Therefore, you can use any one of these chips before implementing it: the simulator will automatically invoke their built-in versions.

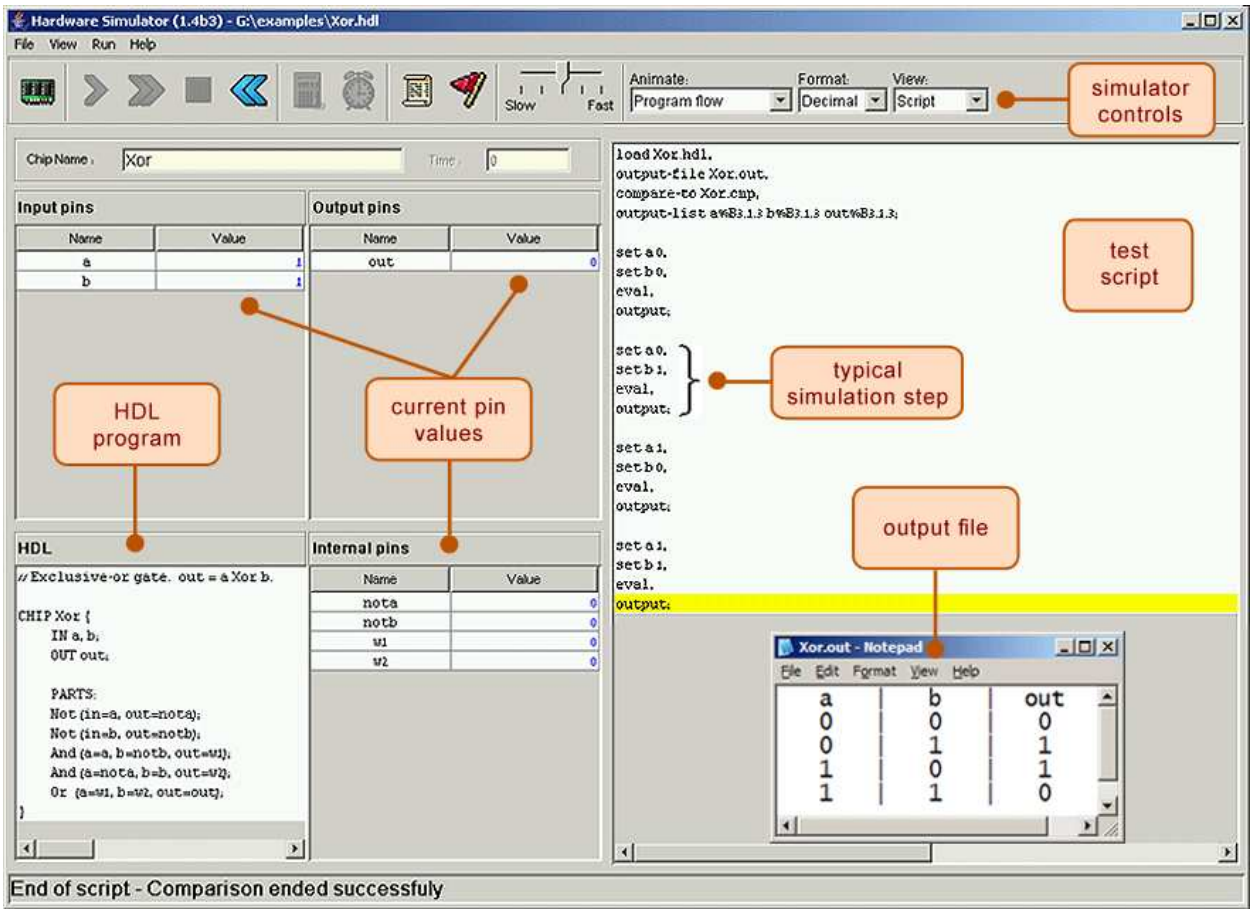
For example, consider the supplied skeletal `Mux.hdl` program. Suppose that for one reason or another you did not complete the implementation of `Mux`, but you still want to use `Mux` chips as internal parts in other chip designs. You can easily do so, thanks to the following convention. If the simulator fails to find a `Mux.hdl` file in the current directory, it automatically invokes a built-in `Mux` implementation, which is part of the supplied simulator's environment. This built-in `Mux` implementation has the same interface and functionality as those of the `Mux` chip described in the book. Thus, if you want the simulator to ignore one or more of your chip implementations, simply

rename the corresponding `chipname.hdl` file, or remove it from the directory. When you are ready to develop this chip in HDL, put the file `chipname.hdl` back in the directory, and proceed to edit it with your HDL code.

To simplify the chip coding process, it can be quite useful to define and build gates not listed above. For example, a `Mux4` chip can come in handy for building the `Mux4Way16` chip. It can also come in handy for building a `Mux8` chip which in turn is handy for building the `Mux8Way16` chip.

Tools

All the chips mentioned in Projects 1–5 can be implemented and tested using the supplied *Hardware Simulator*. Here is a screen shot of testing a `Xor.hdl` chip implementation on the *Hardware Simulator*:



Submission and Assessment

If you can't finish the project on time, submit what you've managed to do, and relax. All the projects in this course are highly modular, with incremental test files. Each hardware project consists of many chip modules (`*.hdl` programs), and each software project consists of many software modules (classes and methods). It is best to treat each project as a modular problem set, and try to work out as many problems as you can. You will get partial credit for partial work.

What if your chip or program is not working? It's not the end of the world. Hand in whatever you did, and explain what works and what doesn't in a README file. If you want, you can also

supply test files that you developed, to demonstrate working and non-working parts of your project. Instead of trying to hide the problem, be explicit and clear about it. You will get partial credit for your work.

See the next page for the assessment rubric. Submit the following as a single ZIP archive in Canvas:

1. A README file containing the names of all group members. This file may also contain other information, as described above.
2. All your HDL files.
3. Nothing else.

Project 1: Logic Gates

Student name(s): _____

Grading method: The implementation of some chips was described in the book, and some chips are simpler than others. The different weights assigned to the chips below reflect this variance. If the chip passes *all* the tests specified in the supplied test script, it receives two thirds of its allotted points. The remaining third reflects our evaluation of the way the chip is built.

Generally speaking, we prefer implementations that *use as few chip parts as possible,* even if it implies a less efficient chip design.

Chip	***Working?***	***Well built?***	***Comments***
Not	/ 3	/ 1	
And	/ 3	/ 1	
Or	/ 4	/ 2	
Xor	/ 4	/ 2	
Mux	/ 5	/ 3	
DMux	/ 5	/ 3	
Not16	/ 3	/ 2	
And16	/ 3	/ 2	
Or16	/ 3	/ 2	
Mux16	/ 3	/ 2	
Or8Way	/ 5	/ 3	
Mux4Way16	/ 6	/ 3	
Mux8Way16	/ 6	/ 3	
DMux4Way	/ 6	/ 3	
DMux8Way	/ 6	/ 3	
Total	/ 65	/ 35	

Total grade: _____