Project 3: Sequential Chips

CS 220

Background

The computer's main memory, also called Random Access Memory, or RAM, is an addressable sequence of n-bit registers, each designed to hold an n-bit value. In this project you will gradually build a RAM unit. This involves two main issues: (i) how to use gate logic to store bits persistently, over time, and (ii) how to use gate logic to locate ("address") the memory register on which we wish to operate.

Objective

Build all the chips described in Chapter 3 (see list below), leading up to a Random Access Memory (RAM) unit. The only building blocks that you can use are primitive DFF gates, chips that you will build on top of them, and chips described in previous chapters.

Chips

Chip (HDL)	Description	Test script	Compare file
DFF	Data Flip-Flop (primitive)		
Bit	1-bit register	Bit.tst	Bit.cmp
Register	16-bit register	Register.tst	Register.cmp
RAM8	16-bit / 8-register memory	RAM8.tst	RAM8.cmp
RAM64	16-bit / 64-register memory	RAM64.tst	RAM64.cmp
RAM512	16-bit / 512-register memory	RAM512.tst	RAM512.cmp
RAM4K	16-bit / 4096-register memory	RAM4K.tst	RAM4K.cmp
RAM16K	16-bit / 16384-register memory	RAM16K.tst	RAM16K.cmp
PC	16-bit program counter	PC.tst	PC.cmp

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 3 and Appendix A. Specifically, all the chips described in Chapter 3 should be implemented in the Hardware Description Language (HDL)

specified in Appendix A.

For each chip, we supply a skeletal .hdl file with 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.

The resources that you need for this project are the supplied Hardware Simulator and the files listed above. The project directory and files may be found in a ZIP file on the course web site. The project directory is further partitioned into two sub-directories, for reasons described below.

Tips

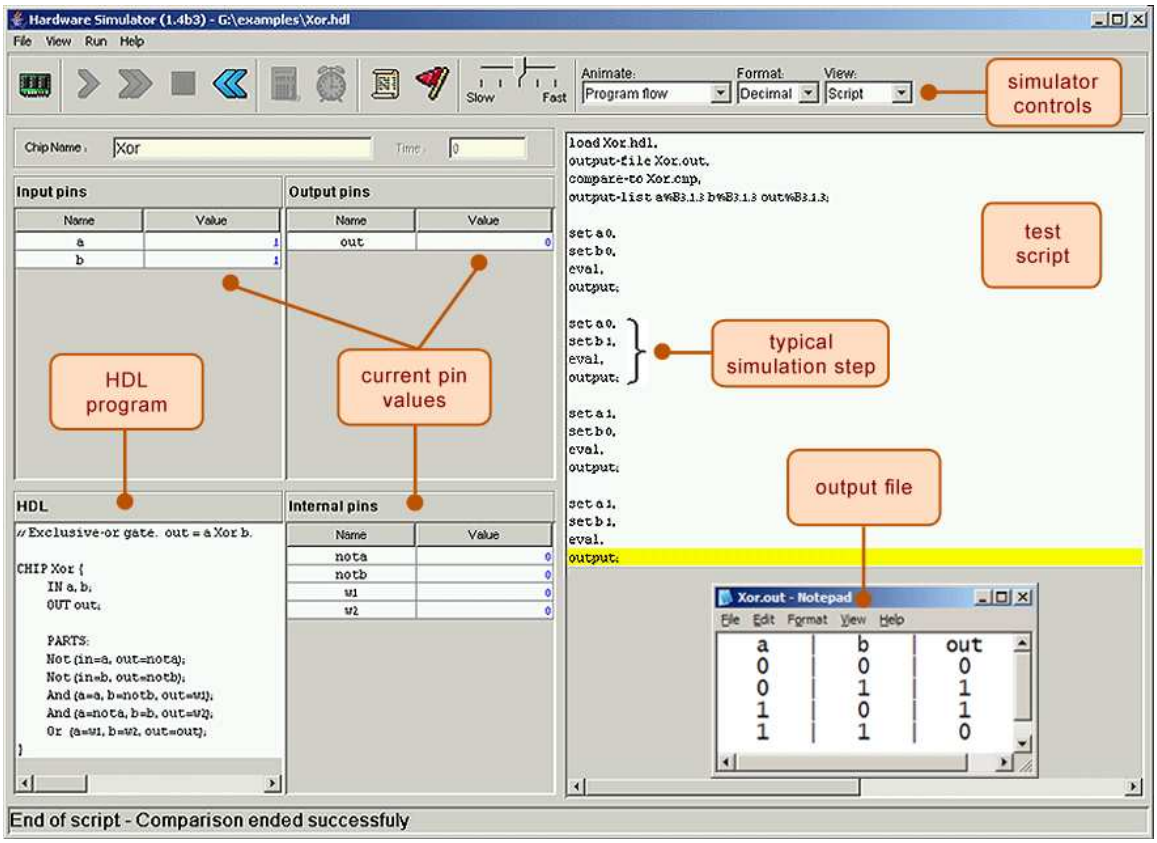
The Data Flip-Flop (DFF) gate is considered primitive and thus there is no need to build it: when the simulator encounters a DFF chip part in an HDL program, it automatically invokes the built-in implementation.

Built-in chips: When constructing RAM chips from lower-level RAM chip-parts, we recommend using built-in versions of the latter. Otherwise, the simulator will recursively generate numerous memory-resident software objects, one for each one of the many chip parts that make up a typical RAM unit. This may cause the simulator to run slowly, or, worse, out of memory. i.e. out of the memory of the computer on which the simulator is running.

To avert this problem, we've partitioned the RAM chips that you have to build in this project into two sub-directories, named `a` and `b`. This partition is superficial, and is done with one purpose only: when building the chips stored in `b`, the simulator is forced to use built-in implementations of the lower-level chip parts whose .hdl programs are stored in `a` but not in `b`.

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 built-in RAM8.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 3: Sequential Chips

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***
Bit	/ 7	/ 3	
Register	/ 7	/ 3	
RAM8	/ 13	/ 6	
RAM64	/ 13	/ 6	
RAM512	/ 5	/ 3	
RAM4K	/ 5	/ 3	
RAM16K	/ 5	/ 3	
PC	/ 12	/ 6	
Total	/ 67	/ 33	

Total grade: _____