CSCE-312 | Spring 2019

Project 1

Building Boolean Logic Gates

Due Date: Submit on eCampus by **Saturday, Feb 2nd, 11:00 PM**

Grading

(A) Project Demo [70%]: Logistics to be provided soon

You will be graded for correctness of the chips (hdl) you have designed and coded. You will be running **live** test of all your HDL codes using Nand2tetris software (Hardware Simulator) with TA/PT. The same simulator you would have used to check your chips in the project. So, make sure to test and verify your codes before finally submitting on eCampus.

Rubric: *PriorityEncoder83.hdl* is worth **10 points** and the others are worth **4 points each**. Each circuit needs to pass all its test cases to get the points, else you will receive **0** on that circuit.

(B) Code Review/Q&A [30%]: To be held with the LIVE Demo

Code review of randomly selected chips. The questions can involve drawing circuit diagram of randomly selected chips. Should not be difficult for you if you have understood the core inner workings of your project.

Deliverables & Submission

You need to turn in only the HDL files for all the chips implemented. Put your full name in the introductory comment present in each HDL code. Use relevant code comments and indentation in your code. Also, include this cover sheet with your signature below. Zip all the required HDL files and the signed cover sheet into a compressed file *FirstName-LastName-UIN.zip* (TA will show the students an example for this). Submit this zip file on eCampus.

Late Submission Policy: Refer to the Syllabus

First Name: Last Name: UIN:

Any assignment turned in without a fully completed cover page will NOT BE GRADED.
Please list all below all sources (people, books, web pages, etc) consulted regarding this assignment:

CSCE 312 Students	Other People	Printed Material	Web Material (URL)	Other
1.	1.	1.	1.	1.
2.	2.	2.	2.	2.
3.	3.	3.	3.	3.
4.	4.	4.	4.	4.
5.	5.	5.	5.	5.

Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Submission Date: _____

Printed Name (in lieu of a signature): _____

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

1. Build all the logic gates described below, yielding a basic chip-set. The **only** building blocks that you can use in this project are primitive **NOR gate** (HDL provided) and the following composite gates that you will gradually build on top of them.

2. After implementing all elementary logic gates, use them to implement chips for the following logic scenarios:

- **Exercise 1:** A student would fail an exam if he spent the previous night studying for the exam, or if he has not had breakfast before the exam.
- **Exercise 2:** You cannot get onto the ride if you are too young and too short, or too old and have heart disease.
  Hint: Convert the sentence to Boolean algebra equation and draw the truth table of each scenario before implementation.

Here are the Chips you are required to implement:

Chip Name	File Name	Description
Not	Not.hdl	Not Gate
Or	Or.hdl	Or Gate
And	And.hdl	And Gate
Xor	Xor.hdl	Exclusive Or Gate
Mux	Mux.hdl	Multiplexer Gate
DMux	DMux.hdl	Demultiplexer Gate
Not16	Not16.hdl	16-bit Not Gate
And16	And16.hdl	16-bit And Gate
Or16	Or16.hdl	16-bit Or Gate
Mux16	Mux16.hdl	16-Bit Multiplexer
Or8Way	Or8Way.hdl	8-bit input Or Gate
Mux4Way16	Mux4Way16.hdl	4-way 16-bit input Multiplexer
DMux4Way	DMux4Way.hdl	4-way 16-bit input Demultiplexer
Exercise1	Exercise1.hdl	Exercise 1 based circuit
Exercise2	Exercise2.hdl	Exercise 2 based circuit
8 to 3 Priority Encoder	PriorityEncoder83.hdl	**Attend Next Week Lab (Mon/Tue)**

Contract

Download and uncompress the *P1Codes.zip* file in your local nand2tetris folder. You should have already deleted the default project directory already present in nand2tetris folder, as announced on the Piazza post.

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 of the textbook.
Specifically, all the chips described in **Chapter 1**
https://docs.wixstatic.com/ugd/44046b_f2c9e41f0b204a34ab78be0ae4953128.pdf
should be implemented in the Hardware Description Language (HDL) specified in **Appendix A**
https://docs.wixstatic.com/ugd/44046b_2cc5aac034ae49f4bf1650a3d31df32c.pdf
Another resource that you will find handy in this and in all subsequent hardware projects is this **HDL Survival Guide** written by Mark Armbrust https://www.nand2tetris.org/hdl-survival-guide

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.**

Tips

Prerequisite: If you haven't done it yet,

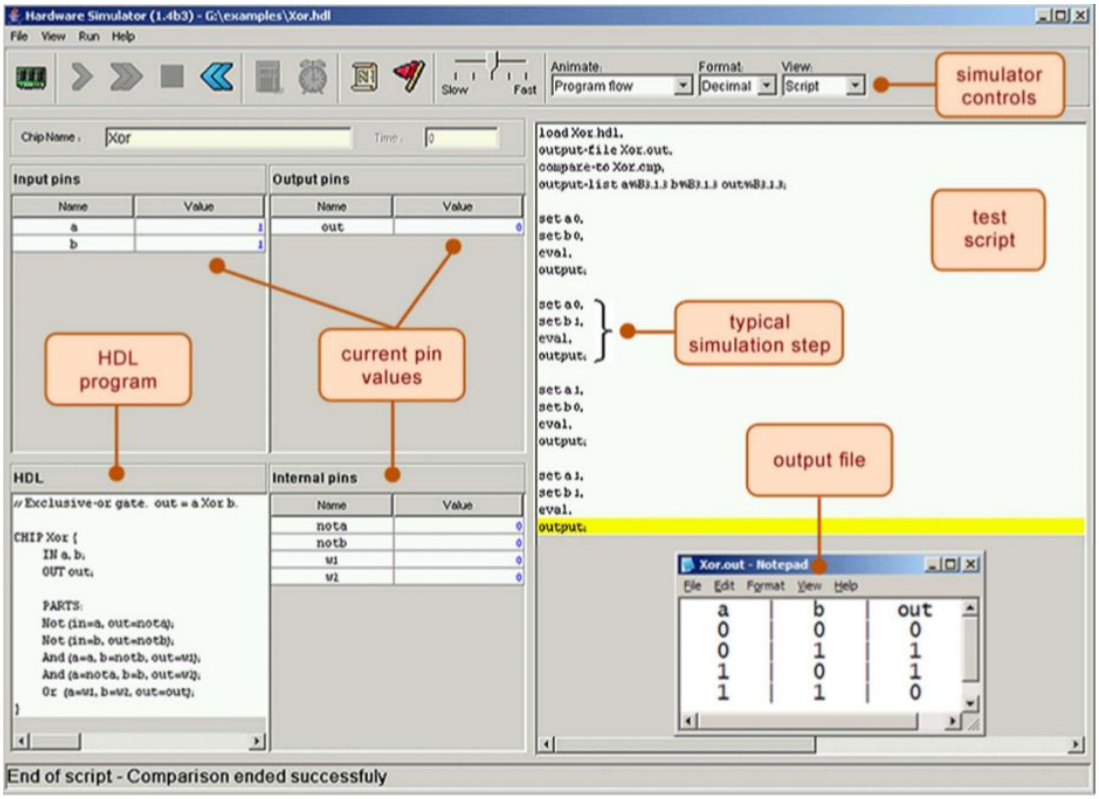
1. Go over Piazza post https://piazza.com/class/jkmb76dcbzw2u4?cid=12
2. Read Chapter 1 and Appendix A, and go through parts I-II-III of the Hardware Simulator Tutorial ppt (see Piazza Resources section) , before starting to work on this project.
3. Built-in chips: The NOR gate is considered primitive and thus there is no need to implement it: whenever a NOR chip-part is encountered in your HDL code, the simulator automatically invokes the built-in tools/builtInChips/Nor.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 builtin 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.

Tools

All the chips mentioned in Projects 1-3 and 6 can be implemented and tested using the supplied Hardware Simulator. Here is a screenshot of testing a Xor.hdl chip implementation on the Hardware Simulator:



Tips/FAQs:

Here are some useful tips for writing your HDL codes.

1. Input = 0
    1. To input zero to a wire, use the "false" keyword.
    2. And(a=false, b=false, out=o1); would always compute "0 and 0"
2. Sub busing
    1. Sub busing is your tool to references a slice of the wire. The syntax is wire_name[i..j]. (This references the slice of wire_name from i to j inclusive)

2. Example: Something4 (in=in, out=my_out[0..3]); this Something4's "out" wire to indexes 0 through 3 inclusive of "my_out".
3. Example: Something4 (a[0..1]=my_a, out=out); this wires the 2 bit wire "my_a" to indexes 0 through 1 inclusive of Something4's "a" wire.
4. Example: Something4 (a[0..1]=my_a[4..5], out=out); wires indexes 4 through 5 inclusive of "my_a" to indexes 0 through 1 inclusive of Something4's "a" wire.

This is all discussed in the HDL survival guide available at https://phoenix.goucher.edu/~kelliher/f2015/cs220/hdlSurvivalGuide.pdf if you have further questions.

From a Truth Table to a Simplified Boolean expression for a chip

Boolean function synthesis requires us to first identify the cases in the Truth Table which have output logic as TRUE (1). These are the different scenarios of input values which correspond to output of boolean function to be true. By creating an expression for these TRUE output cases, it is ensured that FALSE output cases will be automatically taken care of as the output itself is binary and hence complement of these TRUE cases.

With that in mind, we try to express each TRUE case as "Product of Input variables" i.e. combining all the inputs through logical AND. Why? Since, all these inputs will need to hold their respective boolean values, simultaneously, for the function output of that case to be TRUE. Eg: Let's take a scenario where we need to derive expression for f(a,b)

a	b	f(a,b)
0	0	0
0	1	1
1	0	1
1	1	1

Step 1: f(a, b) = 1 (TRUE) when inputs are as follows:

(i) a=0 and b=1 => NOT(a) AND b => **a'.b** *{Always complement the input variable if it is FALSE}*

(ii) a=1 and b=0 => a AND NOT(b) => **a.b'**

(iii) a=1 and b=1 => a AND b => **a.b**

These product terms, namely, **a'b , ab' , ab** are called minterms. These are used when we focus just on output cases when TRUE.

*(Likewise, there are also maxterms which is alternate way of observing the truth table in terms output logic FALSE, i.e., we focus on creating "complement" of the boolean function. Maxterm involves "sum of input variables" for a given **FALSE** output case, which is later multiplied as "PRODUCT of SUMs" across different FALSE cases).* **In this course, we will follow minterm notation only**.

Step 2: Since, cases (i), (ii) and (iii) will not happen simultaneously as boolean inputs in digital circuits can either be 1 or 0 but not both (that's quantum, folks, when any state can exhibit both dual values 0<=>1 at the same time, which is being used to build quantum computer), there is notion of OR across these three TRUE output cases. So, we create a "SUM of PRODUCTS" expression with the minterms , i.e., **f(a.b) = (a'b) OR (ab') OR (ab) = a'b + ab' + ab** (AND/. has higher precedence than OR/+)

Step 3: Simplify the boolean expression using boolean identities.

f(a.b) = a'b + ab' + ab

= a'b + a(b'+b)

= a'b + a(1)

= a'b + a

= a + a'b {by commutative property}

= (a+a')(a+b) {by Distributive property}

= (1)(a+b)

= a+b

= a OR b

Hence, we have successfully derived the simplified boolean function f(a, b).

How do we get mathematical form of Boolean expressions: eg: x ∨ y = x OR y = x + y ? Which one should we learn/use ? / Using . and + seem less intuitive than logical or set notation ∧ , ∨, ¬

In terms of mathematical logic and set theory, which came earlier in literature and were in a way to express logical human thought, it is absolutely correct to use notations such as ∧ , ∨, ¬ for logical AND (intersection), logical OR (union), logical NOT (complement). Hence, in the slides you will find the logical names (AND, OR, NOT) being used in all the examples and expressions. In boolean algebra, we as computer scientists abuse this notation little bit to make our lives easier in terms of modeling these logics as mathematical expressions to better understand the boolean identities. On a personal level, I found expressing AND as . (multiplication) and OR as + (addition) helps me to remember and understand the boolean identities as some mathematical property.
To answer why OR is represented/denoted as + or AND as . => Well, if we look at the truth table of these logics, we find that all the outputs are either non-zero or zero for any given logic (OR/AND) and are also well preserved under mathematical operations of addition (+) or multiplication(.) over all possible values of inputs for that logic. eg: 0 OR 0 = 0 can be thought of as 0+0 = 0, similarly,
0 OR 1 = 1 OR 0 = 1 is same as if 0+1 = 1+0 = 1, and 1 OR 1 = 1+1 = 2 which is again non-zero, hence, logic HIGH or Logic 1 (Recall, the expressions used in if() in C++programming, where non-zero

evaluations are treated as logic TRUE). So, I will suggest use notations that best help you to understand and to remember the identities and simplifying boolean expressions.