APPLIED HOMEWORK #1

Design Using FPGAs

1 Description and Homework Objectives

AHW1 is to be submitted by the report deadline listed on the course webpage. You need to **first view two** videos (**Structural Verilog & ECE352 CAD Tools**) in that order. Please note that, although this applied homework is significantly shorter than the remaining four, it still will require some time to complete. The work for this report should be completed **individually**. You will be paired with a partner from the same "lab" section and will perform the next 4 AHW's as a team of two.

After this homework, you should be able to:

1. Launch ModelSim and compile Verilog design files
2. Debug Verilog compilation errors
3. Simulate a test bench in ModelSim
4. Debug Errors, recompile and re-simulate
5. Write structural Verilog for a simple unit

2 Background: FPGAs and Quartus

A designer can choose from many different implementation technologies for a given application. The most basic choice is between using a microprocessor and a custom hardware circuit. In programming courses, your instructor essentially decided in advance that the implementation of your project was best done using a microprocessor running your software. In this course, we teach the fundamentals of digital design, and implement applications in *hardware*.

An engineer can implement hardware in several ways: full custom, semi-custom (standard cell), and field-programmable. In a full custom design, the engineer designs the entire circuit from the transistor level using no pre-designed elements. This method has the most potential performance but is the most expensive both in time to design and initial cost for one chip. In a semi-custom design, the designer uses a library of standard cells (e.g. logic gates, memory, etc) combined with automated tools to create the circuit. This method produces designs much more quickly than full custom, but is still expensive for initial cost (still need the factory for fabrication). The final method is to use field-programmable chips. In this method, the designer uses a pre-made chip able to implement many different circuits, such as a field-programmable gate array (FPGA). The designer creates their circuit and uses automated tools to map the circuit to the FPGA. This method is the quickest and has the smallest initial cost; however, it also has the worst potential for performance. In this class, because we want to implement many different small circuits in a short period of time, we will use FPGAs.

2.1 ECE352 CAD Tools

2.1.1 Verilog & ModelSim

We will implement our designs in a **H**ardware **D**escription **L**anguage (HDL) called Verilog which is commonly used in industry. We will try to keep to a simple subset of Verilog as to not confuse you too much. Of course Verilog code needs to be tested. This is done by placing your design (DUT = Device Under Test) in a test bench. The test bench is also written in Verilog, however, we will provide the test bench to you. The Verilog simulator we use is called ModelSim, and is installed on all CAE machines and can be accessed by XenApp. The best option, if you have a Windows machine, is to install ModelSim on your machine (see the link under "Applied Homework Resources" to download). A final option is to remote login into a CAE linux machine and run ModelSim (called vsim on linux) there.

2.1.2 Quartus & Altera Cyclone IV FPGA

During a "normal" semester you would Demo your AHW (done in verilog) in "lab". There you would see your code in action on hardware. This portion of the class does not really add to the instructional value, but does help students really see that verilog is not a "programming" language but actually a Hardware Description Language (HDL). This point was driven home by the fact that you were configuring hardware (an FPGA) with your HDL and seeing the consequences of your design (and bugs). This connection is lost with the online version of the course. We are supposed to tell you the online version is just as good, but it simply isn't. Somethings need to be done in person, and it is not cost effective to have all students buy their own FPGA development board at the cost of several hundred $$ each.

Design Entry

For this course, we will provide Verilog shells of the designs we wish you to implement. You will complete the "guts" of the Verilog to implement the design. Design entry can be done in any text editor. You can use the editor built into ModelSim or an external editor of your choice. Notepad++ is not a bad choice. We will use the latest version of Verilog referred to as **S**ystem **V**erilog. All of our design files will end with a **.sv** extension.

Functional Simulation

Once our design is created we have to simulate it to ensure it is functionally correct. A self checking Verilog test bench will be provided. You will launch ModelSim and simulate the test bench. Invariably your design will have bugs and you will have to view waveforms and figure out why your design is not working properly. You will fix the bugs and iterate until the self checking test bench says your DUT is good. In the course ECE551 (352 is a prerequisite for it) you will use Verilog in more depth and write all blocks and test benches from scratch.

Synthesis (Implementation)

Once we have a functionally correct design we can then perform implementation, which produces a file that can be loaded into an Altera FPGA to prototype the design (ie, actually implement the hardware circuit). The new file describes our circuit in terms of the FPGA's internal logic elements. Implementation consists of five main steps:

Map: The tool optimizes our circuit (eliminating unnecessary logic), and then breaks the circuit into pieces that can fit into a logic element (LE)

Place: The pieces from the mapping step are assigned to specific LEs and other FPGA resources. Placement tries to simplify the next step (routing) by assigning pieces with many interconnections to LEs that are near each other.

Route: All interconnections between the pieces of the circuit and the inputs/outputs are now assigned to specific routing resources of the FPGA. In other words, we connect the pieces that were placed in the previous step.

Timing: After mapping and place and route, locations of the look-up tables, flip-flops, input, outputs and all of the interconnection paths are known. Based on this (particularly the number of transistors along each interconnection path), delay estimates for various paths through the circuit and delay statistics for the FPGA can be calculated. These values are placed in a file which is used to perform timing simulation.

Configuration: The final implementation step is the generation of the configuration file—the sets of 1s and 0s to load into the FPGA configuration SRAM that implements the circuit. The file we use is an SOF (*SRAM object file*). When this file is loaded into the FPGA's configuration SRAM, the look-up tables are filled, the multiplexers are set, and the routing transistors turned ON or OFF to create the interconnections.

All of these steps produce report files that record what happened during that step, related statistics, and whatever warning or error messages were output (if any). A report viewer is provided to look at the reports. Another file produced by place and route is the Pin file that indicates which pin is assigned to each of the FPGA inputs and outputs. This file is useful for making board-level connections to the FPGA chip and knowing which pins were used for what purpose..

Prototyping

Despite the fact that we are not mapping our designs to hardware this semester we would still like you to read the following section on the FPGA we would use.

2.2 Prototyping Hardware

We will use an Altera Cyclone IV FPGA board for our hardware demonstrations. The boards are powered by an external power supply (and two on-board voltage regulators), and include many resources attached to the FPGA for use in a wide variety of projects. This includes things like LEDs, switches, 7-segment displays, LCD displays, …

How does a signal name in your verilog get mapped to a specific pin on the FPGA? This is done through a .qsf (Quartus Settings File) file. Normally this file is provided by the instructor, but in reality is just a simple text file that maps signal names to FPGA pin locations.

Altera Cyclone IV Series FPGAs

The information in this box is not needed to complete this homework. It is provided in case you are interested.

Look-up tables (LUTs) implement combinational logic functions in most current FPGA architectures. A LUT is a very small static random-access memory (SRAM) that holds the truth table for the function it implements. The inputs of the LUT act as an address for the SRAM, and the output is the value held at that address (that line of the truth table). A complete logic circuit is formed by programming the contents of the LUTs as needed, then setting other SRAM bits that control how the LUTs are connected to one another through tristates and multiplexers. These values are loaded during FPGA configuration (generally each time the FPGA is powered on), which can take 100s of milliseconds, depending on the size of the FPGA.

The LUTs in the Cyclone FPGA are 4-LUTs: they have four inputs. Any logic function with 4 inputs and 1 output can be implemented using a single 4-LUT. The fundamental structure of the Cyclone FPGA is known as a logic element (LE), shown below in Figure 1, and each LE contains 4-input LUT and a flip-flop (a storage element we'll discuss later).

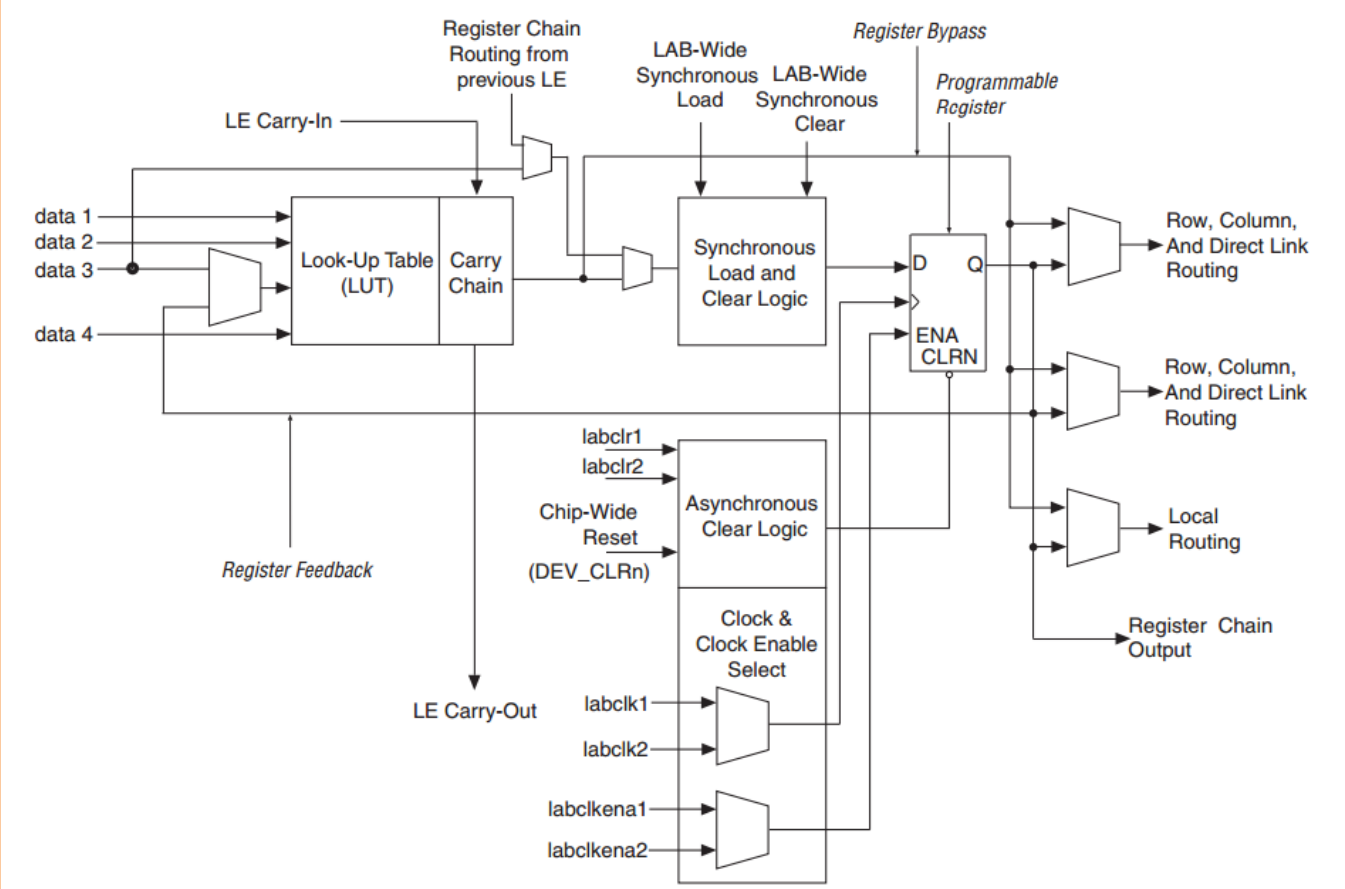


Figure 1: Altera Cyclone IV Logic Element (LE) [1]

The process of deciding which LUT in the FPGA should implement which logic gate, and which wires should implement which signals between gates is very complex. These choices affect whether or not the circuit fits in the FPGA, and if it does, how fast it can run. Fortunately, this is done automatically for us by the Quartus development tool. In a realistic commercial design, you would probably have to "tweak" certain tool parameters to meet your particular design requirements and produce an "optimal" design.

[Remember "optimal" will mean different things for different designs and situations (minimal delay, minimal area, minimal power, minimal skew, etc). This is why companies hire highly-paid hardware designers!]

3 Applied Homework Tasks

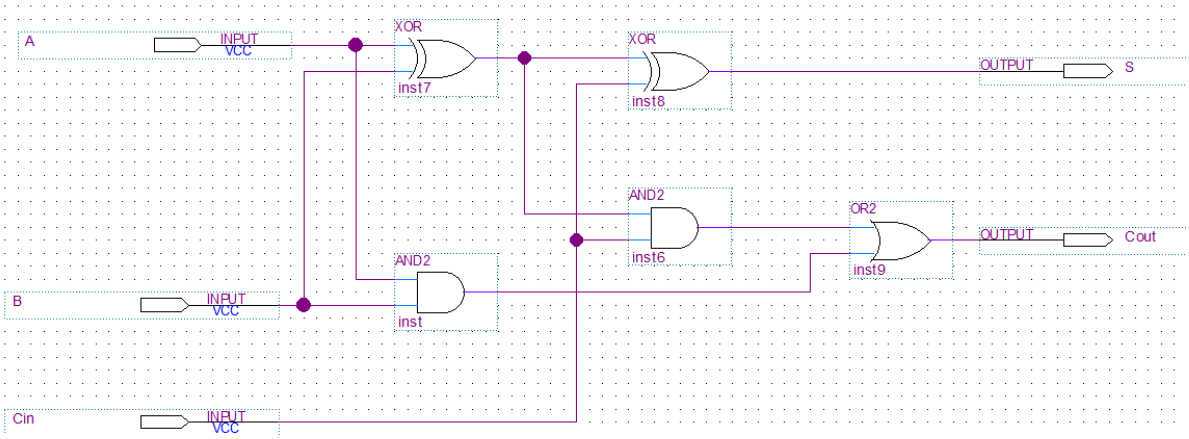
This must be done ***individually***. Every student needs to know how to use ModelSim, and write basic verilog. You will eventually have a AHW partner, but do not rely on them to perform oll the tasks. Hoffman has a nasty habit of sticking AHW related question on the exams just to catch the slackers…Don't be a slacker. The skills you gain here also come to play for some of the later written homeworks throughout the semester. All work submitted for AHW1 **must be your own work**, based on working through the steps in this document.

Be sure you have watched the two videos (Structural Verilog & ECE352 CAD Tools) **before attempting this AHW.**

In this experiment, you will design and test an FPGA-based "full adder" – a circuit that adds two bits and produces a sum bit and a carry bit, and use it to create a larger adder.

3.1 Full Adder Design/Simulation/Debug/Submission

1) Download the AHW1.zip file. Put is someplace logical…don't be a lazy unorganized slacker that just downloads and unzips everything into their ***Downloads*** folder. Instead create **ece352** folder and an **AHW1** sub-folder in a logical spot on your laptop.
2) Launch ModelSim
    a. Perform: **File → New → Project** to create a new project
    b. In the resulting **"Create Project"** form:
        i. Give the project the name: **AHW1**
        ii. Hit **"Browse"** under **Project Location** and change location to folder where you extracted the provided files to.
        iii. Take an **"OK"** on the **"Create Project"** form.
    c. The project will be create and an **"Add Times to the Project"** form will appear:
        i. Click **"Add Existing File"**
        ii. Click **"Browse"** and select all the provided **.sv** files. *(press and hold Cntrl while selecting files)*
        iii. Take an "OK" on the **"Add Files to Project"** form.
        iv. Click **"Close"** on the **"Add Items to the Project"** form
3) The first circuit you will be editing is **FA.sv**. You can either open in using the text editor built into ModelSim, or the text editor of your choice. I personally like Notepad++. *Notepad++ does have different .XML files you can download and install for syntax highlighting of various languages. There is one available for System Verilog.*
4) **FA.sv** contains a shell of a full adder circuit. A full adder will add two operand bits (**A** & **B**) and a possible carry in from a lower order bit position (**Cin**) and produces a sum (**S**) for that bit position and a carry out (**Cout**) that can ripple to a higher order bit position.
5) Complete the code for **FA.sv** by filling in any needed intermediate signals, and writing the structural Verilog to implement the functionality. The picture below shows how a full adder is constructed



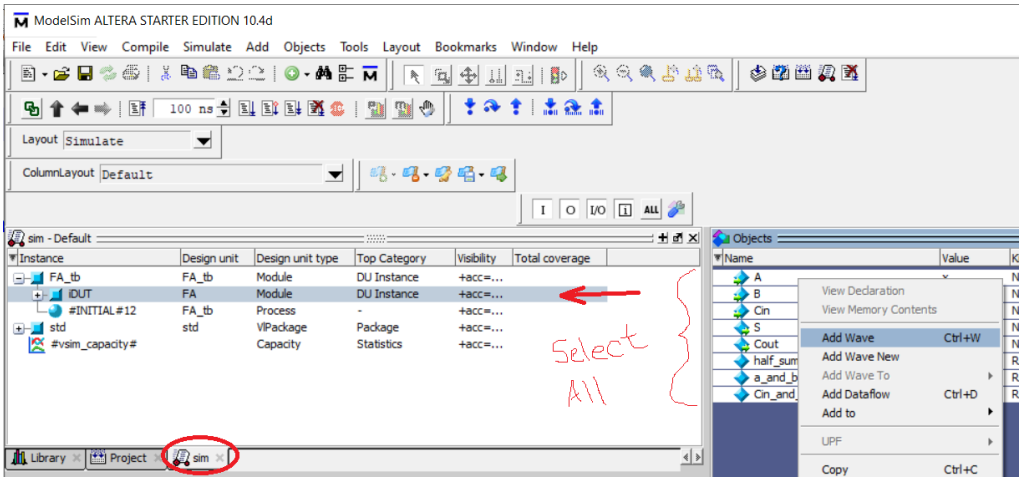
6) When you have finished save the file and compile it within ModelSim. Does it compile (green check mark) or does it have errors (red X). If it has errors double click on the red text in the "Transcript" window and fix the errors that are mentioned.

7) If you have not already, also compile the provided test bench (**FA_tb.sv**).

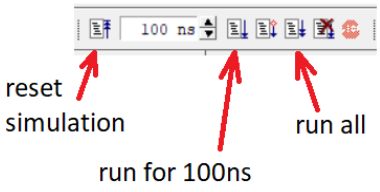
8) Now simulate the test bench

   a. Start the simulator by either Simulate=>Start Simulation and expanding the **work** library and choosing **FA_tb**. Or…my preferred way by going to the **"Library"** tab, expanding the **work** library selecting **FA_tb** and right clicking to launch.

   b. Once the simulation tab is up select the DUT (iDUT) then click and shift click to select all the signals of your DUT. Finally right click and select "Add Wave" (see figure below).



   c. Now run the entire simulation using the simulation control in either the waveform window or the main ModelSim window.

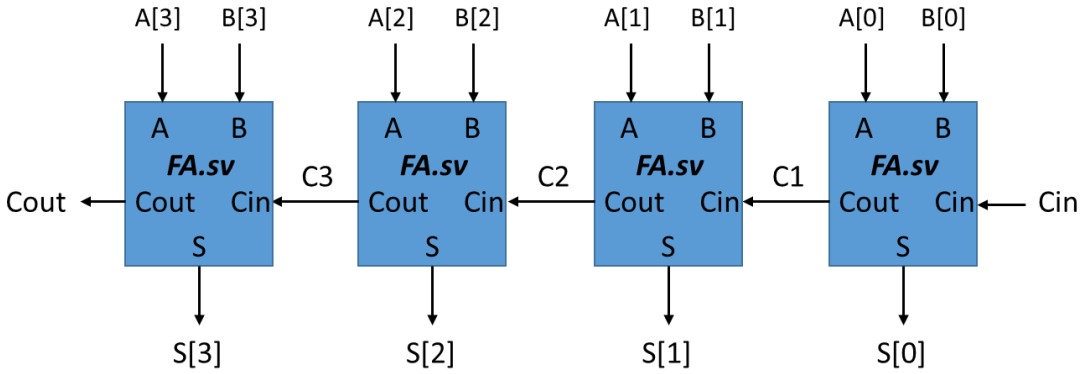


   d. Did your design run error free? If so you got lucky this time. If not…start debugging. Don't like debugging? … pick a different career path. When debugging remember you do **not** have to stop simulation in ModelSim after editing your verilog. You can edit and save the verilog, recompile, reset the simulation (see graphic above) and then re-run the simulation.

9) Once your simulation of **FA.sv** runs with no errors take a screen image of the waveforms. Call this **FA_sim.jpg**. Submit **FA_sim.jpg** and **FA.sv** to the dropbox for AHW1.

3.2 Ripple Carry Adder Design/Simulation/Debug/Submission

Next you will create a 4-bit ripple carry adder (RCA) by instantiating 4 copies of your newly created FA module and threading the carry chain. This ripple carry adder accepts two 4-bit vectors (**A[3:0]** &**B[3:0]**) and a carry in (**Cin**). It produces a 4-bit sum (**S[3:0]**) and a carry out (see image below).



1. A shell called **RCA4.sv** is available. Edit it to flush out the ripple carry adder implementation. Implement it as shown with 4 discrete instantiations of **FA**, and 3 intermediate nets (C1,C2,C3). Your instantiations of FA should use connect by name, not connect by reference order. Meaning…connections like **FA iFA0(.A(A[0]), .B(B[0]), …**
2. Compile **RCA4.sv** in ModelSim (fix any compilation errors)
3. Compile the provided test bench **RCA4_tb.sv**
4. Simulate **RCA4_tb.sv**. Does your DUT perform correct? If not start debugging.
5. Once your **RCA4.sv** design is passing the test bench perform a screen capture of the waveforms for all the signals of the DUT for the entire length of this simulation. Call it **RCA4_sim.jpg**. Also capture an image of the "YAHOO! test passed" message in the transcript window (**RCA4_yahoo.jpg**). Submit: **RCA4.sv**, **RCA4_sim.jpg** and **RCA4_yahoo.jpg** to the dropbox for AHW1.

3.3 Vectored Instantiation…A smarter way

1. Imagine if I had asked you to implement a 16-bit RCA adder using 16 copies of your **FA** cell? Would be pretty tedious right?
2. Prof. Lipasti presented a smarter way of doing multiple instantiations of a block. A "vectored" way of doing it.
3. Copy your **RCA4.sv** block and call it **RCA16.sv**. Modify this block to use the smarter vectored way of instantiating the 16 copies of **FA**. Don't forget to modify the **module name** (on top of file) to call it **RCA16** instead of **RCA4**. Otherwise when you compile in ModelSim it will overwrite the compiled version of **RCA4**. ModelSim does not give a rat's butt what the file name is. Just because the file is called **RCA16.sv** does not matter. When it compiles into the work library it will use the module name. Easy to forget this, and hard to figure out what is happening.
4. Compile **RCA16.sv** in ModelSim (fix any compilation errors)
5. Compile the provided test bench **RCA16_tb.sv**
6. Simulate **RCA16_tb.sv**. Does your DUT perform correct? If not start debugging.
7. Once your **RCA16.sv** design is passing the test bench perform a screen capture of its waveforms. Call it **RCA16_sim.jpg**. Also capture an image of the "YAHOO! test passed" message in the transcript window (**RCA16_yahoo.jpg**). Submit: **RCA16.sv**, **RCA16_sim.jpg** and **RCA16_yahoo.sv** to the dropbox for AHW1.

4 Applied Homework Submission

You will upload a number of files to the AHW1 dropbox be sure that your files are named correctly.

Files to be submitted in advance of the demo:

File Name:	Description:
FA.sv	Structural Verilog of full adder
FA_sim.jpg	Waveforms showing functionally correct simulation of full adder
RCA4.sv	Structural Verilog of 4-bit ripple carry adder
RCA4_sim.jpg	Waveforms showing functionally correct simulation of 4-bit adder
RCA16_yahoo.jpg	Capture of “Yahoo Test Passed!” message from transcript window
RCA16.sv	Structural Verilog of 16-bit Ripple Carry Adder using vectored instantiation.
RCA16_sim.jpg	Waveforms showing functionally correct simulation of 16-bit adder
RCA16_yahoo.jpg	Capture of “Yahoo Test Passed!” message from transcript window