ECE 3544: Digital Design I

Project 3 (Part A) – Design and Synthesis of a Parity-Checking System

*Read this specification in its entirety before you begin working on the project!*

**Honor Code Requirements**

You must do this assignment individually. The following represent unauthorized aids, as the term is used to define cheating in the Virginia Tech Honor System Constitution:

* Discussing *any aspect or result* of your design with *any person* other than your instructor and the ECE 3544 GTAs. This includes but is not limited to the implementation of Verilog code, as well as the supporting details for that code – truth tables, state diagrams, state tables, block diagrams, *etc.*
* Using *any design element* – including Verilog code – from *any printed or electronic source* other than your course textbook and those sources posted on the course Scholar site.

*It is permissible for you to re-use any of your own original Verilog code.*

All code submitted is subject to plagiarism checking by MOSS (theory.stanford.edu/~aiken/moss/). Any copying flagged by MOSS will be treated as Honor Code violations and submitted to the Virginia Tech Undergraduate Honor System.

**Project Objective**

In this project, you will implement an error *correction* system from new and existing design units. Since your design will rely largely upon code that you have already written, you will have the opportunity to test (and possibly improve upon) the synthesizability of your Verilog code.

In addition, you will implement your top-level module on the Altera DE1-SoC board, so you will learn how to assign pins of your FPGA to the module’s input and output ports, and how to synthesize the module.

**Requirements**

*The DE1-SoC board IS REQUIRED for this project.*

You must have the “current version” of ModelSim-Altera Starter Edition (10.5 or higher) and Quartus (Quartus II Web Edition 18.1 or higher) installed on your computer.

The instructions are done using an appropriate version of ModelSim and Quartus. While the directions may be consistent with other versions of ModelSim and Quartus, you must use the versions indicated above.

**Background**

First, let’s talk some more about parity bits. First, it’s worth noting that even though the 74280 circuit that you modeled in Project 2 is billed as a “parity generator/checker,” it’s arguably more a “parity checker” than it is a “parity generator.” Still, we were able to take advantage of its parity checking function to verify the integrity of a transmitted word. Put simply, the parity of the input word at transmission should match the parity of the output word at receipt.

If we compare the parity of the two words at those two points in time, they should be the same, and a comparator (the XNOR gate) will reveal this:

XNOR

9

odd

even

I[8:0]

odd

even

I[8:0]

1

1

1

(Transmitter)

(Receiver)

Figure 1: A 9-bit word containing an odd number of ones.

If a single bit error occurs – or for that matter, if an odd number of bit errors occur – the parity of the number of ones on the receive side will be different from the parity of the number of ones on the transmitter side will be different. A comparator will likewise reveal this:

XNOR

9

odd

even

I[8:0]

odd

even

I[8:0]

1

0

0

(Transmitter)

(Receiver)

Figure 2: During transmission, one of the bits is corrupted and has the wrong value.

If the value of only one bit changed, then the received value will end up with an even number of ones.

Let’s continue to operate on the assumption that a single bit error has occurred. The first logical question is “Which bit was corrupted?” The answer to this question is “We don’t know.” Knowing that an error happened is not enough to know which bit is erroneous. This is the crux of an *error detection* scheme – we can determine that certain kinds of errors happened, but we don’t have enough information to fix them.

The next logical question is “Is there a way to create an *error correction* scheme – one where we can not only know that an error occurred, but can also know where the error is?” The somewhat-nuanced answer is “Yes, but it depends on the kind of error and the number of errors.” In the case of single bit errors, we can create what is called a Hamming code.

*(The system that we will implement is different from Hamming’s original approach but is well-suited to the approach we have already used for parity checking in Project 2. Ultimately, the logic that I will describe here will be the same.)*

So, what is the key to creating a system that employs Hamming codes?

****

Consider a three-bit value that we want to transmit. Let’s name the bits as follow: {Bit3, Bit2, Bit1}. It might make you uncomfortable not to name the LSB as Bit0, but there is a logic to the convention. Now imagine that we are going to perform *two* parity checks on certain subsets of the bits. Study the following table:

|  |  |  |
| --- | --- | --- |
|  | Parity Check 2 | Parity Check 1 |
| Bit3 | 1 | 1 |
| Bit2 | 1 | 0 |
| Bit1 | 0 | 1 |

Figure 3: Assigning each bit to a parity check

Notice that all we did was to represent the value of each bit position in binary. So Parity Check 2 will only apply to Bits 3 and 2, while Parity Check 1 will only apply to Bits 3 and 1. What you should take away from this is that each bit has a unique set of parity checks that apply to it.

Imagine that as an example, we want to transmit the word {Bit3, Bit2, Bit1} = 010. Further suppose that – as we did in the lab – we use the *odd parity* output of parity checker. On the *transmit* side, Parity Checker 2 receives {7’b0, Bit3, Bit2} = 000000001, while Parity Checker 1 receives {7’b0, Bit3, Bit1} = 000000000. Thus, Parity Checker 2 outputs 1, and Parity Checker 1 outputs 0.

Assume that the same word 010 is what the receiver receives. Therefore, on the *receive* side, Parity Checker 2 receives {7’b0, Bit3, Bit2} = 000000001, while Parity Checker 2 receives 000000000. Thus, Parity Checker 2 outputs 1, and Parity Checker 1 outputs 0.

When we compare the corresponding Parity Checker outputs, we still want to know if the values are the same or not. But **unlike** Project 2, we want the result of a comparison to be 0 if the values are the same (to indicate “no error”) and 1 if they are not (to indicate “error”). Instead of an XNOR gate, we should use an XOR gate. This change makes sense if you now view the output as an “error” output instead of a “valid” output, as it was in Project 2

With this in mind, Comparator 2 will output 0 because its “transmit” and “receive” parity bits are the same. Comparator 1 will also output 0 for the same reason. What does this mean? In this case, it means that the word is genuine. If both comparators output 0, it generally means that there are no differences between the word that was transmitted and the one that was received.

**Now suppose** that during transmission, Bit2 is corrupted – that is, the receiver actually receives 000 instead of 010. What happens on the *transmit* side remains unchanged. On the *receive* side, Parity Checker 2 now gets 000000000, while Parity Checker 1 still gets 000000000. Therefore, Parity Checker 2 will output 0 (which is different from the previous case), and Parity Checker 1 will output 0 (as it did before).

In this situation, Comparator 2 will output 1, because its “transmit” and “receive” parity bits are now **different**. Comparator 1 will output 0, because its “transmit” and “receive” parity bits are the same.

But what does this mean? If we read the Comparator outputs as a binary number, the code “10” means that Bit2 must be the corrupt bit! And it was!

If you understand how the system operates – a single bit error causes a specific set of parity comparisons to fail in a way that identifies the corrupt bit position – then you can probably see that things can go wrong if more than one bit is corrupt. There are ways to add to the system to make it able to correct single errors *and* detect double errors. But in this project, we will only create and look for single bit errors.

Therefore, our system will need:

* Enough parity checkers to implement this system for a *seven*-bit input word, and
* A circuit that is capable of “corrupting” one of the bits.

**Project Description**

In this assignment, you will design and implement the system represented in the block diagram shown below so that it works on the DE1-SoC board.

odd

even

I[8:0]

odd

even

I[8:0]

odd

even

I[8:0]

?

?

?

SW[6:0]

XOR1

PC3Trans

PC2Trans

PC1Trans

Bit Corrupter

SW[9:7]

odd

even

I[8:0]

odd

even

I[8:0]

odd

even

I[8:0]

?

?

?

PC3Rec

PC2Rec

PC1Rec

LED[6:0]

LED[9]

LED[8]

LED[7]

XOR2

XOR3

74280

74280

74280

74280

74280

74280

Figure 4: Top-level block diagram of the Hamming code system

While this block diagram might seem complex, it actually represents a system that contains multiple instantiations of the same elements. Each of the blocks labeled **74280** is one of the Parity Checkers from Project 2. Each Parity Checker is further labeled to show its place in the system and whether it is on the Transmitter “side” or the Receiver “side” of the system. Following the logic in the example from the previous section, **PC3Trans** is Parity Checker 3 on the Transmitter side, while **PC3Rec** is Parity Checker 3 on the Receiver side. The outputs of those two Parity Checkers are being compared (by an XOR gate) to produce one of the three outputs in our Hamming code system. The three outputs should identify whether or not the received message is equal to the transmitted message, and if not, which of its seven bits is corrupt.

The blue clouds in the block diagram represent places where you will have to make implementation decisions. As you saw in the example from the previous section, not every parity checker will receive all seven bits of the input word. You will have to determine which bits of the input word should be connected to each of the parity checkers you instantiate. Please read the section “Implementing the Final Design” for more information on this and other implementation details.

You will receive a top-level module that defines the input and output interfaces of the FPGA and the DE1-SoC board. The basic top-level module also provides examples of module instantiation. Follow the instructions on pin assignment and programming your board to implement the existing behavior of the top-level module on your DE1-SoC board, then test the board to verify that that existing module performs as expected. *Do not program your board before you have performed the pin assignment.*

After you verify the functionality of the existing module and your board, implement the system described in the block diagram. Create a test-bench to simulate your design. After you verify the functionality of your design in simulation, program your board with the implementation of the final design.

**Implementing the Basic Module**

Make a working folder for this Project and copy the Quartus archive to that folder. You should use the same guidelines for locating and naming your working folder as have been indicated for previous assignments.

Start Quartus. Open the archive file in Quartus by selecting **Project** → **Restore Archived Project**. You will find an interface for the project. The existing interface does not implement the final design, but it does demonstrate the manner in which board-level inputs and outputs interface with instantiated modules. For now, you don’t need to modify any the code in the top-level module or in the modules that have been included for instantiation in the top-level module, but you should review all of the Verilog files to develop an understanding for what the system should do when you implement it on the DE1-SoC board.

**Pin Assignment**

Before you program the FPGA with the design contained in a Quartus project, you must perform a pin assignment. *This step is* ***important*** *– an incomplete or incorrect pin assignment could damage your board.*

The DE1-SoC Board has many peripherals, but we will not use all of them in this assignment. In general, you must assign each bit of the ports in your top-level module (slider switches, pushbuttons, clocks, LEDs, and hex display inputs – all as needed) to a physical pin on your FPGA. The existing top-level module uses the switches, the LEDs, and the hex displays. Your final design will use ten switches and ten LEDs. It will not use any of the hex displays. You may safely ignore the unused outputs, or you may insert code into the top-level module to account for their non-use, but since they are ports of the top-level module, you will still need to account for them in you pin assignment.

To make a pin assignment, you have to know which FPGA pins correspond with the board’s peripherals. You can find this information in the DE1-SoC manual, which I have posted for download along with the project files.

Follow the instructions in the Canvas document entitled “Pin Assignment for the DE1-SoC” board to assign pins for the top-level module. Remember that you must perform the analysis and elaboration step on an existing top-level module before you can assign pins for that module, and you must re-compile your project each time you make a change to the pin assignment. (You should make a habit of re-compiling your project each time you make a change to any element of your design.)

**Programming the FPGA with the Basic Module**

After you make a complete, correct pin assignment, you can program your board. Follow the instructions in the Canvas document entitled “Programming your DE1-SoC Board” to implement the design contained in the top-level module.

After you program your board, test the functionality of the existing design and your board by manipulating the input switches and observing the LEDs and the hex displays. *You should be able to verify that the design and your board are in working order based on your understanding of the behavior of the top-level module and its instantiated components.*

**Implementing the Final Design**

Make changes to the top-level module as described in the comments of the existing module. Your goal is to implement a system matching the block diagram in the “Project Description” section of this specification.

You have already implemented several of the blocks in the diagram in previous assignments. Use your 74280 module from Project 2. *You must use the version of the 74280 module that has no delays in it.* You may also describe the XOR gates in a manner that is similar to the one that you used in Project 2 to describe your XNOR gates. Add any files that you write or carry over from Project 2 to the existing project. I recommend that you copy your files to the working folder that you create for this project. In the **Files** tab of the **Project Navigator**, right-click on the Files folder and choose **Add/Remove Files in Project**. In the window that appears, choose the **…** button to browse for the files you want to add, then click **Add** once you have selected them.

This project would not do very much if we simply attached the same values on the Transmit side to parity checkers on the Receive side. Instead, we are going to write a module that can complement one of the bits of a seven-bit input word to create a “new” seven-bit output word. We will use this word on the Receive side of the system as though it were the same word that was transmitted, but we will be doing this with the knowledge that one of its bits might be corrupt.

Here is the module declaration that you must use for the Bit Corrupter module:

module bit\_corrupter(index, input\_word, output\_word);

input [2:0] index;

input [7:1] input\_word;

output [7:1] output\_word;

(Don’t let the indices of the input and output words worry you. The LSB is being called bit 1, and the MSB is being called bit 7. This is actually consistent with the way we are naming the bits in our system. You only have to remember that when you connect switches SW[6:0] as the input word in various places, that the value on SW[0] represents the LSB (which is our bit 1) and SW[6] represents the MSB (which is our bit 7).

In this module shown above, if index = 3’d0, the output word should equal the input word. If index equals a non-zero value (obviously, from 3’d1 to 3’d7) the value in that position of the input\_word should appear as its complement in the output\_word, with all other bits unchanged. This will ensure that at most, only one bit in the transmission is corrupted.

The Bit Corrupter module is a combinational circuit (as are all of the sub-modules in this system) so *if you choose to write a procedural model* *to describe it*, you should ensure that you use proper synthesizability principles for describing it.

To that point: since our goal this semester has been to write code that will synthesize correctly in addition to simulating correctly in ModelSim, this project will provide you with the opportunity to test the synthesizability of code that you have already written. If Quartus gives synthesis errors for any of your modules, you should determine what elements of your code have caused the synthesis errors and correct them.

When you write your modules (as you did in Project 2), you should write them as though they can take any input that is appropriate for the port declaration you made. But when you instantiate your modules in the top-level module, you are effectively connecting them to the switches and LEDs that you want to use to provide the I/O to your top-level module. Use the existing top-level module as an example of how to connect inputs and outputs to the instantiated module, then follow this logic when you do the same things in your actual implementation.

**Simulation**

The project files include the framework for a test bench. In general, you can create a test bench in Quartus by choosing **File** → **New**, and then choosing **Verilog HDL File** under **Design Files**. Remember to add the file to the existing project, and to repeat the Analysis and Synthesis step after you have finished making changes to the file that you add.

Follow the instructions in the document “Simulating Quartus Modules Using ModelSim” to simulate the behavior of the top-level module.

Write your test bench to simulate different operating cases of the comparator system. A sequence similar to the validation sequence (or the validation sequence itself) would be a good start. *The validation test sequences are not nearly exhaustive. While you need not produce an exhaustive set of test cases in your test bench, you should not rely on the validation sequence as the sole test of your system’s correctness.*

**Programming the FPGA with the Final Design**

After you make sure of your comparator’s functionality via simulation, you are ready to program the FPGA with your final design. You should not need to assign the pins a second time, but you might wish to open the Pin Planner to verify that your pin assignment remains unchanged. Follow the instructions in the document “Programming your DE1-SoC Board” to implement the final design on your FPGA.

After your program your board, test the functionality of your final design by manipulating the input switches and observing the LEDs and the hex displays. While the validation sheet contains one sequence of input tests that you can try, the validation test is not exhaustive. As you did with your test bench, you should perform an amount of testing that you consider adequate for verifying the working order of your design.

**Project Submission**

Write a report describing your design and implementation process. Your report should include waveforms showing the correct behavior of your design.

I have included the validation sheet that the GTAs will use to test your design after the submission deadline. *You do not have to go to the CEL to have your project validated.* Instead, you should use the information in the validation sheet as one basis for testing your design before you submit your project.

Your project submission on Scholar should include the following items:

1. Your project report in PDF
2. A Quartus Archive containing the source files for your top-level module, any modules that the top-level module requires to function, and your test benches for the top-level module.

To create the Quartus Archive, choose **Project > Archive Project** after you complete your implementation. When prompted for a name for your archive, the default archive name will be the same as the original archive. *Append your Virginia Tech PID to the end of the filename*. Make certain that you submit the archive that you create – the one containing your solution to the project – and not the one that I provided to you – the one that only contains the initial files.

1. The Verilog source files for:

* Your 74280 module
* Your bit corrupter module
* Your final top-level module

Your top-level module may be tested with our own secret test bench, so it is important that you use the module declaration provided with the archived project. You must also include the source files for every module required for your top-level module. Failure to do so will result in a grade of 0 for that portion of the project.

Grading for your submission will be as described on the cover sheet included with this description