Lab 1: (15,11) Hamming Code Encoder/Decoder

*This lab implements the (15,11) Hamming Code on the BASYS 3 Artix-7 Board through VHDL code written in Vivado. The resulting code allows two functions, encoding and decoding. In encode mode, the VHDL code will take the 11 input data bits, compute the corresponding parity bits, and display them on the LEDs. In decode mode, the BASYS 3 board will display the error syndrome (corrupted bit) on the LEDs and the 7 segment display.*

**Mark Lee**

**ECE 357**

**September 25, 2019**

**TA Date:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**TA Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Introduction**

When electronic information -- usually consisting of binary values -- is transmitted across space and time, it is susceptible to data corruption. Radiation and other factors affecting the quantum world can flip zeros to ones and ones to zeros, introducing error into the data. The receiver of that information does not want erroneous data, and if possible would like to know which particular bits are corrupted. Hamming Codes offer the ability to detect up to 2 bit errors and correct a single erroneous bit. By assigning unique binary indexes to each bit, and strategically placing parity bits (each covering a place value of the indexes - least significant bit, most significant bit, etc.) within the data, the receiver of the data can know by looking at the parity bits whether or not an error occurred. If the XOR of the data bits that any parity bit covers is not consistent with the value of the parity bit, then that parity bit is in error. The binary value of the parity bits (1 meaning the parity bit is wrong, 0 meaning it’s correct) indicates the index of the flipped bit, and this can be rectified. If two bits are flipped, the Hamming Code can detect that the data is wrong but not where the erroneous bits are, so the data must be discarded or re-transmitted. If several bits are flipped, the Hamming Code can not tell whether it’s corrupted or not. Therefore, Hamming Codes are useful only when the error rate is low, such as in normal computer memory.

This lab experiment implements a (15,11) Hamming Code decoder/encoder in VHDL code programmed to the BASYS3 board. The (15,11) specifies a 15 bit value composed of 4 parity bits and 11 data bits, and has an efficiency of Since parity bits cover unique place values when they are set (equal to 1), they are located at powers of 2, and so in the (15,11) code are located at positions 1, 2, 4, 8. By creating a Hamming Code encoder/decoder, students will become better at working with VHDL, interfacing with the hardware peripherals on the BASYS3 board (such as the 7 segment display and the LED lights), and teach them with a core concept in the field of error correction.

**Procedure**

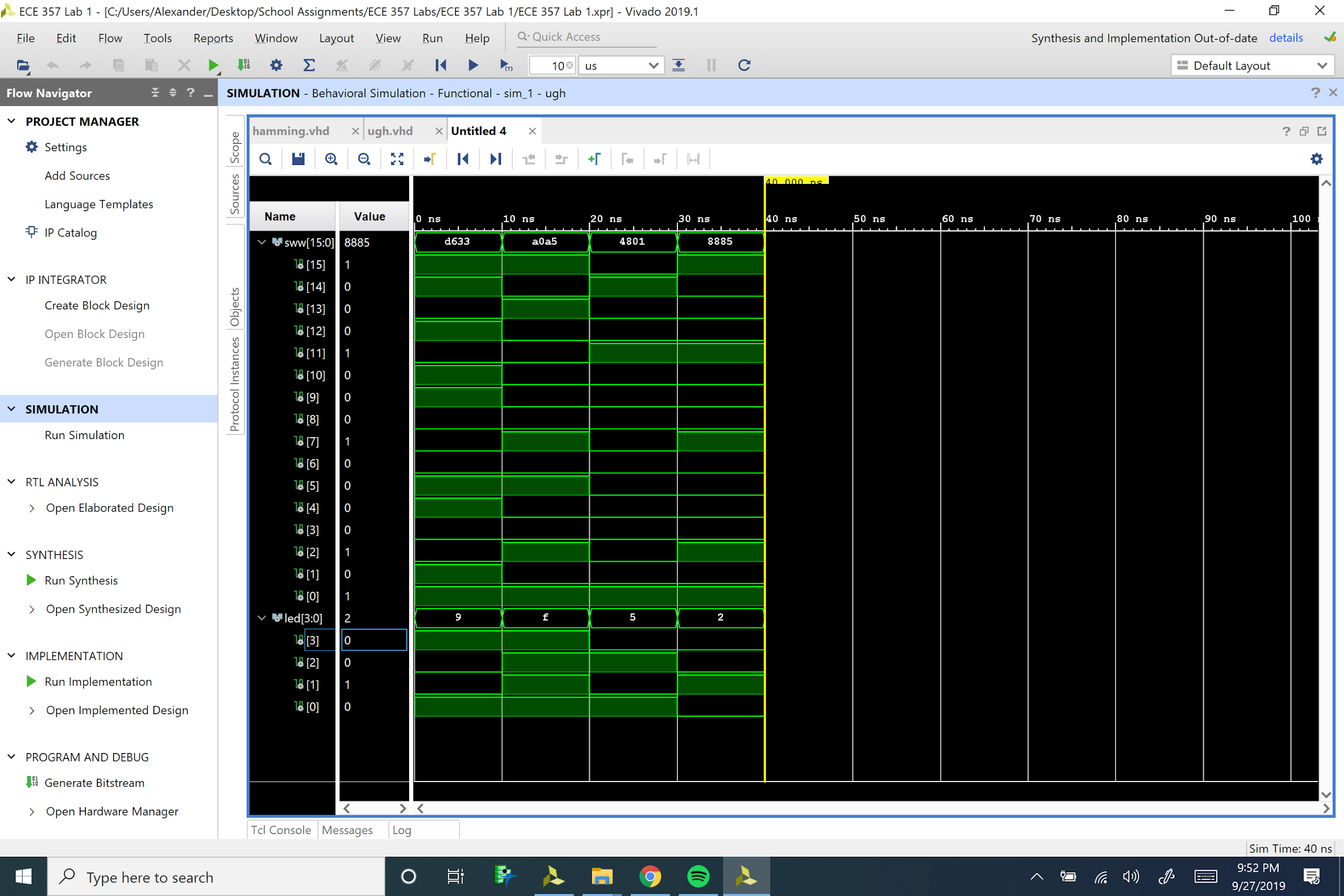
For Task 1, I implemented the (15,11) Hamming Code encoder/decoder by writing the appropriate VHDL code and uploading it to the BASYS3 board. Encode means computing and displaying the parity bits for the 11 data bits; decode means computing and displaying the error syndrome for the 15 input bits. My implementation takes 16 bits as input, 15 bits corresponding to the 4 parity bits and 11 data bits of the Hamming Code value, and the 16th bit corresponding to an encode/decode mode (encode = ‘1’, decode = ‘0’). I represent these bits as the 16 switches on the BASYS3 board, SW0-SW15, with SW0 being the encode/decode mode and SW1-SW15 being the 15 value input bits (SW15 most significant bit, SW1 least significant bit). When in SW0 = 1 (encode mode), I ignore the parity bits as input and only look at the 11 data bits. I XOR the appropriate data bits together, based on which positions the particular parity bit encodes, and compute what the parity bits should be. If the number of 1s is odd, the parity bit is 1, and if the number of 1s is even, the parity bit is 0. I do this for the four parity bits and display the results on the LEDs LD0-LD3 (LD0 corresponding to P1, LD3 to P8). Lit LEDs correspond to a 1, darkened to 0. When SW0 = 0 (decode mode), I determine from all 15 input bits where the erroneous bit was located. I do this by again XORing the bits encoded by a particular parity bit, but this time I check to see if they match. If they do, the parity is correct (0), but if they don’t it is wrong (1). I again display the parity bits on the LEDs, but this time they represent to the index of the erroneous bit (the error syndrome).

I am going to briefly summarize how I implement this in actual VHDL code. I input the switches as a 16-bit bus, create signals to store parity bit values P1-P8, and then manually XOR all the appropriate bits together. I then use a bunch of if/else statements (since this is inside a process) to determine which LEDs to turn on, and finally display the results on the LEDs through a 4 bit output bus.

For Task 2, I expand my code from Task 1 to have the same functionality, but add the 7-segment display to show the error syndrome (in decimal) while in decode mode. In encode mode, I just display ‘E’ for encode. I had to add a lot of VHDL code to accomplish this, much of it inspired by the reference resources attached to the assignment. To display two segments simultaneously, I alternate between two 7-segments (the right two when viewing the board) at a rate of about 10ms per refresh. I utilize the clock (CLK in my VHDL code) to achieve the timing, but have to divide it by a 20 bit bus (signal counter in my VHDL code) to get 10ms because the natural refresh rate of 100MHz is way too fast. I added some logic to my existing if/else statements to copy the 4-bit parity error syndrome value into a 4-bit signal (called error\_syndrome in my VHDL code). Then, depending on whether error\_syndrome is greater or less than 10, I update an 8-bit signal called “displayed\_number” that keeps track of the values to be displayed (the most significant 4 bits correspond to the tens place, which is either 0 or 1, and the least significant 4 bits correspond to the ones places, which can be 0-9). When the program is in decode mode and is alternating displays, I alternate the number displayed as well. My output to light up the 7-segment displays consists of the 7-bit bus “Cathodes” and the 4-bit bus “Anodes” -- their functions should be obvious.

**Results**

The simulation results for encoding are shown below. From 0ns to 40ns I ran the four test



cases that came with the assignment and which were used in the lab demo. The first input, 110101100011001, is on the left from 0 to 10 ns, and the output is 1001 (9 in decimal) which is correct. The second test case, 101000001010010, is shown from 10ns to 20ns, and the output for the LEDs is 15, as shown by all the values for led[3:0] being 1. This is correct as well, as are the other test cases. This simulation data verifies the validity of my implementation.

Additionally, I went and tested my project in lab, so I am confident of the results that it provides.

**Conclusion**

As the results and my lab demo show, my code works perfectly. Every test case I have tried is correct. I have to admit this lab took me a long time to complete, partly because I haven’t used VHDL/Vivado in a year, and partly because I didn’t understand how the Hamming Code worked until I worked through some examples. Additionally, I had a couple strange errors that held me up for a while. One such error occurred when I tried to implement my design in Vivado, where I had to go to a source file in my project directory and change a line of code to allow some apparently “unadvised” routing. I also ran into a lot of syntax errors, and it took me forever to learn how to correctly alternate 7-segment displays, even with the reference guide. Nevertheless I got my code working and I’m quite pleased with the result. All of the test cases work, and I feel quite refreshed in my VHDL knowledge.

I could probably make my code a lot cleaner, but I have already invested several hours into this lab, so I won’t go for perfectionism this time. For example, I think I could create an exquisite for loop that does all the XORing for the parity bits. It would be a nice way to practice new VHDL skills that I’ve learned in class, but like I said I’ll save that for next lab. Once I figured out what the Hamming Code does I began to really like it, and it got me interested in other error detection methods, and I figured out it is a whole field of study. Overall I’m quite pleased with my results and with the outcome of the lab.

**Code**

