Experiment #2: Introduction to Programmable Logic and VHDL

Team A-Shred
Steve Comer
Stuart Larsen

Implementation Due: 18 Sept 2012
Report Due: 1 Oct 2012

_____
Steve Comer

_____
Stuart Larsen

Table of Contents

1.0 Introduction

This lab consisted of three subsections. All three involved using VHDL programming. VHDL is a nested acronym that stands for Very High Speed Integrated Circuit (VHISIC) Hardware Description Language (VHDL). The first task was to create a driver for a 7-segment Hexadecimal display in VHDL. The second task was to implement our random sequence counter from Lab 1 in VHDL. The third task was to create a master file which made use of the 7-segment driver to display values provided by the random sequence counter. The sequence to be displayed was {3, 1, 6, 4, 2, 0, 7, 5}.

2.0 Method

2.1 Background Information

2.1.1 FPGAs

As we experienced in the last lab, constructing circuits on a breadboard with individual chips and wires is a time-consuming and error-prone process. Physical circuits can be difficult to debug as well. A Field Programmable Gate Array (FPGA) offers a different method for implementing circuits. FPGAs contain many different logic gates. Although more expensive than the individual chips it contains, an FPGA is very versatile, as it can be customized and reused. The user tells the FPGA which gates to use and in what order by way of a software programming language known as a hardware description language.

2.1.2 Hardware Description Languages

The focus of this lab was learning to use an FPGA in conjunction with a hardware description language to produce a useful circuit. Hardware description languages, such as VHDL, are used to specify the behavior of a circuit. We used VHDL code to configure an FPGA with a circuit that performed the actions necessary for the lab.

VHDL programming has two modes: structural and behavioral. Structural programming is low-level and specifies gates and connections to be used, similar to breadboard implementation. Behavioral programming is several levels of abstraction higher than structural programming. As the name suggests, behavioral programming specifies the behavior of the circuitry in high-level terms. The compiler, in this case Quartus II, uses complex algorithms to build an efficient implementation of the circuit based on the specified behavior. The programmer does not see the actual hardware-level circuitry.

At compile time, the compiler displays statistics about the implementation, such as number of logic elements and number of registers. The programmer loads the implementation onto the FPGA via a wired connection and can immediately begin testing the circuit. Behavioral programming is beneficial because the user can maintain an intuitive view of the circuit's operation without the burden of low-level details.

2.2 Procedure

2.2.1 Design

Most of the design phase revolved around the Hex Driver. We decided it would be best to build this first, as it would allow for simple testing of the Counter component.

A hexadecimal digit can describe any one of sixteen possible values. The possible values are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. Each hexadecimal digit can be converted into a 4-bit binary value. For example, $(0)_{16} = (0000)_2$ and $(F)_{16} = (1111)_2$. The toggle switches on the FPGA board map easily to binary inputs. We used four toggle switches (see Appendix B) as the 4-bit input. These controlled one hexadecimal display chip.

The next step in the design was determining how to describe the 4-bit input in a way that the seven segment display could interpret it correctly. The seven segment display takes a 7-bit active-low input which turns on and off the individual segments of the display. The seven segments are referred to as segments 'a' through 'g' (see Figure 2.1). The input to the display is in the form "gfedcba" (see Figure 2.2).

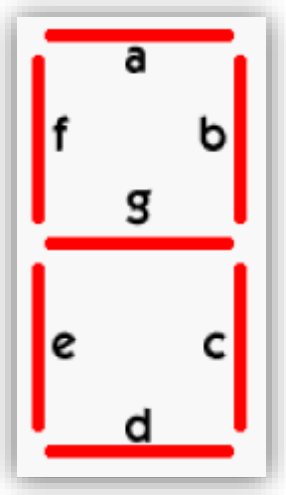

Figure 2.1 Seven Segment Display
source: https://wiki.ittc.ku.edu/ittc/EECS_140_Seven_Segment_Display

Hex Digit	4-bit input	7-bit output (gfedcba)
0	0000	1000000
1	0001	1111001
2	0010	0100100
3	0011	0110000
4	0100	0011001
5	0101	0010010
6	0110	0000010
7	0111	1111000
8	1000	0000000
9	1001	0001000
A	1010	0000011
B	1011	1000110
C	1100	0100001
D	1101	0100001
E	1110	0000110
F	1111	0001110

Figure 2.2 Seven Segment Display Input/Output

In summary, the input to the display driver is a 4-bit vector. This 4-bit input is mapped to a 7-bit vector which turns the individual segments on and off. The visible output is the result of the 7-bit vector, in

which a '0' turns on a given segment and a '1' turns off a given segment. The next phase of the design was VHDL coding.

2.2.2 VHDL Coding

For this experiment, we used the behavioral style of programming in VHDL. See Appendix A for code used in our experiment. The general structure of a VHDL program is as follows:

Library Files:	additional functionality to import
Entity:	defines inputs and outputs for the file (Port)
Component(s) (optional):	declaration of imported data structures used in the program
Architecture:	specifies type of the program as behavioral, includes processes
Process(es):	specific behaviors performed by circuit

2.2.2.1 Hex Driver

The purpose of this program is to output a hexadecimal character on a seven segment display. The input for the file is a 4-bit binary vector. The output is a 7-bit binary vector which turns specific segments in the display chip on or off.

There is one process in the file, which contains a CASE-WHEN structure with 17 WHEN statements. The sensitivity list for the process triggers on numberToDisplay. It only checks for updates when the input changes. There is one statement to handle each possible 4-bit input vector and give the corresponding 7-bit output vector. The seventeenth statement is WHEN OTHERS, which is redundant because all 16 combinations of 4-bit input are covered. However, it catches any unknown inputs which could potentially cause problems.

2.2.2.2 Counter

This program is a random sequence counter finite state machine. It generates the next state based on the current state. The sequence produced by the program is {3, 1, 6, 4, 2, 0, 7, 5}. The only formal input for the file is a clock signal. There is also a synchronous reset option, which only works if reset is high at the same time a rising clock edge occurs. The primary output is Q, a 4-bit vector representing the 3-bit current state of random sequence counter. We extended Q to 4-bits by prepending an additional 0 most significant bit of the 3-bit vector. This makes communication of values between the Counter and Hex Driver files much more simple. The program also outputs clockLED and resetLED which show the current values of clock and reset respectively.

There is one process in the file, which contains an outer IF-ELSE structure which determines if a rising edge of the clock has occurred. In order to prevent useless checking of the IF-ELSE condition, the process is triggered by changes in the value of the clock. The IF-ELSE structure also accounts for the reset condition. If there is a rising clock edge and reset is high, then the state goes to 3. If there is a rising clock edge and reset is low, then the next state depends on the current state.

Within the IF-ELSE structure is a CASE-WHEN structure to account for each state of the counter. Similarly to the Hex Driver file, there is a redundant WHEN OTHERS statement to prevent unintended behavior of the circuit. Within each case, the output Q and the current state are updated.

In the PORT section of the code, it is important to note that we set the initial state of the counter to be 3. If this is not done, the counter defaults to 0 initially. The circuit still proceeds through the sequence correctly if it starts at 0, but our sequence starts with 3.

2.2.2.3 Master File

This program combines the functionality of the Hex Driver and Counter to make a random sequence generator with a seven segment output. The inputs for the master file are reset and clock. The outputs are sevenSegmentOut, which is sent to Hex Driver to be displayed, and resetLED and clockLED. This program includes the functionality of Counter and Hex Driver, added as components. They communicate via the signal binaryNumberOut which takes the current 4-bit state vector from Counter and sends it to Hex Driver to be displayed. The reset input works as previously described in Counter.

2.2.3 Pin Assignments

The full list of pin assignments is given in Appendix B. Pin assignments are explained in the order in which they appear in the appendix.
numberToDisplay[3 DOWNTO 0]: These four toggle switches were used as inputs to test the Hex Driver file to make sure that the 16 possible hexadecimal values could be displayed correctly. Note that numberToDisplay[x] corresponds to SW[x] on the DE2 board.

sevenSegmentOut[6 DOWNTO 0]: These seven outputs are segments 'a' through 'g', forming one hexadecimal character on the DE2 board. These segments are active low because they turn on when given a value of 0. Note that sevenSegmentOut[x] corresponds to HEX0[x] on the DE2 board.

Clock: This is a push-button input to the circuit. Pushing this button down and then releasing it corresponds to one clock cycle for the circuit. The button gives a high signal when it is pressed. This corresponds to KEY[0] on the DE2 board.

Reset: This is a push-button input to the circuit. If this button is held down at the same time a rising edge of the clock cycle occurs, the circuit resets to its initial value of 3. This corresponds to KEY[1] on the DE2 board.

clockLED: This is a green LED which shows the current value of the clock signal. A high clock signal turns on the LED. This LED was chosen because it is physically close to the push-button input used for clock. This corresponds to LEDG[0] on the DE2 board.

resetLED: This is a green LED which shows the current value of the reset signal. A high reset signal turns on the LED. This LED was chosen because it is physically close to the push-button input used for reset. This corresponds to LEDG[1] on the DE2 board.

Q[3 DOWNTO 0]: These are four red LEDs used to show the 4-bit binary value being passed from the Counter component to the Hex Driver component within the Lab 2 master file. This was used as way to confirm that the two components were communicating correctly. Note that an extra bit (Q[4]) was added to the 3-bit value generated by the counter. This bit is always 0 and is used to fit the 4-bit requirement of our Hex Driver. Note that Q[x] corresponds to LEDR[x] on the DE2 board.

2.3 State Transition Diagram

The transition diagram in Figure 2.3 models the high-level behavior of the circuit. Our sequence consisted of 8 numbers corresponding to a permutation of the binary values $(000)_2$ through $(111)_2$. Our specific sequence was {3, 1, 6, 4, 2, 0, 7, 5}. The extra state bubble in the middle is an explicit labeling of the Reset condition. It was added for clarity. The transitions which end here immediately and always go to "3", the beginning of the sequence.

2.4 Problems and Solutions

The first problem we encountered involved the 7-bit vector used to control the seven segment display. At first, we assumed that the display segments were active high like an LED. After realizing that the display was showing the exact opposite segments that we had intended to turn on, we discovered that the display segments are active high.

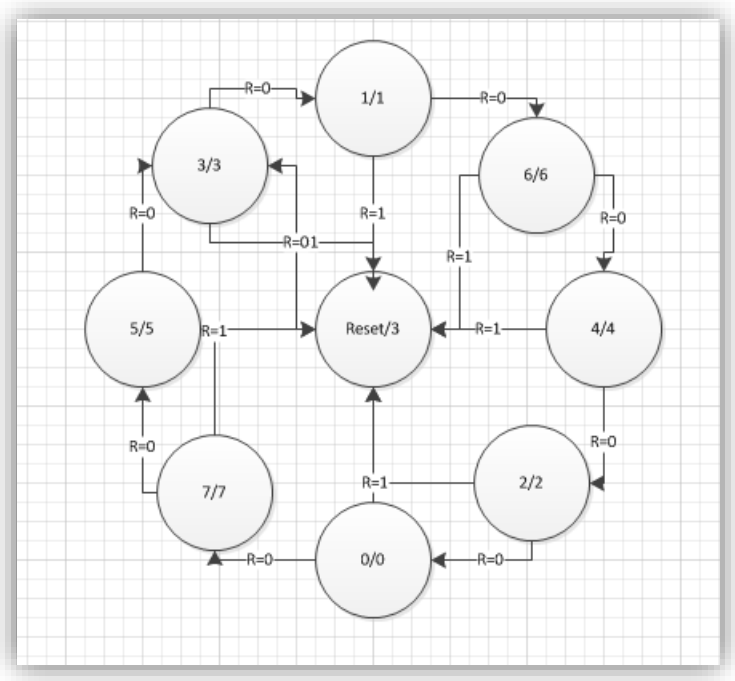

Figure 2.3: State Transition Diagram for our circuit

The second problem we had also involved the seven segment display. We realized and then confirmed by researching online that the display segments are active high. We then flipped all the input vectors (for example, 0011001 to 1100110). This should have solved the problems we were having. We loaded the updated program onto the DE2 board. However, nothing changed in the physical display. After some struggling, we eventually closed and reopened Quartus II. It was only at this point that we realized we had changed the path to our lab folder from the desktop to a different folder. For this reason, our old, flawed program was loaded onto the board every time we tried to update it to the new code. This was frustrating because our code was correct, but still was not working correctly.

The third problem again involved the seven segment display. We originally entered the input vector in the order "abcdefg". This error was more difficult to find than the active high versus active low problem because the result was not exactly the opposite of what we intended. We eventually found that the vector should be loaded in the reverse order "gfedcba".

2.5 Materials and Tools Used

The materials and tools used in this experiment are as follows:

1. Quartus II Software
2. Altera DE 2 Board with Cyclone II – EP2C35F672C6 FPGA

3.0 Results

3.1 Observations

Since the project was mainly a programming assignment, there wasn't much to observe besides our finished result. Our finished project functioned correctly and we were able to observe the project working as expected.

3.2 Testing Procedure

The testing procedure for lab 2 went smoothly for our board. We were able to demonstrate full functionality and correctness for all input combinations. The following list documents the testing procedure that was used:

1. Power on the circuit. Initial state is 3.
2. Power on the circuit. Hold reset and then cycle the clock to demonstrate that the reset state holds.
3. Power on the circuit. Cycle the clock until the sequence repeats.
4. Power on the circuit. For each state, show that reset works correctly, and that the sequence works correctly after exiting the reset state.

To assist with the testing procedure, two LEDs were added, one to display the clock value and the other to show the reset value.

3.3 Results

Experiment 2 was successful for Team A-Shred. While there were multiple problems during the execution and design of experiment 2, Team A-Shred completed all objectives without any problems.

3.4 Warning Explanations

Quartus gives four main types of messages when compiling code: info, warnings, critical warnings, and errors.

3.4.1 Errors

Errors are the most important type of message. They explicitly tell you where in the code or design Quartus was unable to understand. It is essential that all errors are removed from a project, since the code will not compile correctly unless the errors are fixed. It's also important to realize that just because all the errors are removed from a project doesn't mean the code works.

3.4.2 Critical Warnings

The second most important message type is a critical warning. While these don't explicitly mean that there's something wrong with the code, there still exist a high change that something isn't right. Consider the following C++ example:

```
for (int x = 0; x < 10; ++x);
        std::cout << "Hello world!" << std::endl;
```

Figure 3.4.2

There is a semicolon at the end of the For Loop, meaning that the loop will repeat ten times, and then print "Hello world!". While the code is perfectly valid, and it will compile correctly, it probably isn't what the programmer intended, and most C++ compilers will throw a critical warning.

Team A-Shred received two critical warnings. The two critical warnings involved time requirements. Since we were not explicitly using the Timing Analyzer it was not a major concern, but the warnings are explained below:

Critical Warning 1:
Critical Warning (332012): Synopsys Design Constraints File file not found: 'Lab1_new.sdc'. A Synopsys Design Constraints File is required by the TimeQuest Timing Analyzer to get proper timing constraints. Without it, the Compiler will not properly optimize the design.

Reason:
Team A-shred did not supply a timing constraint file, so Quartus was unable to test the timing of our circuit to see if it'll work in its environment.

Critical Warning 2:
Critical Warning (332148): Timing requirements not met

Reason:
Critical warning 2 is related to critical warning 1, in that since a timing constraints file was not supplied, Quartus was unable to do timing analysis, so the timing requirements were not met.

3.4.3 Warnings

Warnings are fairly common in Quartus projects. Due to the fact that Quartus compiles to an industry standard, it is very particular about little details. Quartus warnings can be considered gentle reminders that inform the programmer that it finds something peculiar. A common example in programming languages is unused variables. Declaring a variable and not using it generally doesn't cause any harm, but at the same time, the variable has no use and should be removed or used at some point.

Team A-Shred received five different types of warnings. They are described below.

Warning 1:
Warning (10492): VHDL Process Statement warning at counter.vhd(34): signal "Reset" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

Reason:
Quartus is stating that the sensitivity list for the main process in the counter module did not contain Reset, even though Reset was used inside of the process. This warning is superfluous due to our design. Since the circuit only updates on a clock event, there's no need to do anything when only the value of Reset is modified.

Warning 2:
Warning (15705): Ignored locations or region assignments to the following nodes…

Reason:
Quartus is stating that the parameters to our seven segment display driver are not specifically bound to anything physical. This is due to the fact that we are treating those specific variables as signals.

Warning 3:
Warning (306006): Found 9 output pins without output pin load capacitance assignment

Reason:

This warning states that the reported timing of these pins will be faster than in reality because load capacitance is unknown.

Warning 4:
Warning (171167): Found invalid Fitter assignments. See the Ignored Assignments panel in the Fitter Compilation Report for more information.

Reason:
Since we tested each component individually with specific pin assignments, and then used different pin assignments late when combining all the modules together, some filters will be invalid depending on the context.

Warning 5:
Warning (169174): The Reserve All Unused Pins setting has not been specified, and will default to 'As output driving ground'.

Reason:
Since it was not specified what to do with the unused pins, they will be grounded.

3.4.4 Info
Info messages are purely for the programmers benefit. They let the programmer know what Quartus is doing with the project during compile time. They do not need to be checked every compile cycle.

3.5 Quartus Flow Summary
During compilation of a project, Quartus outputs a flow summary for the project. The flow summary list properties about the project such as if the compilation was successful, the top level entity, the number of logical elements, the number of pins, and other such information. It's important to review this information to verify if the project is working correctly. Figure 3.5 is the flow summary for our project.

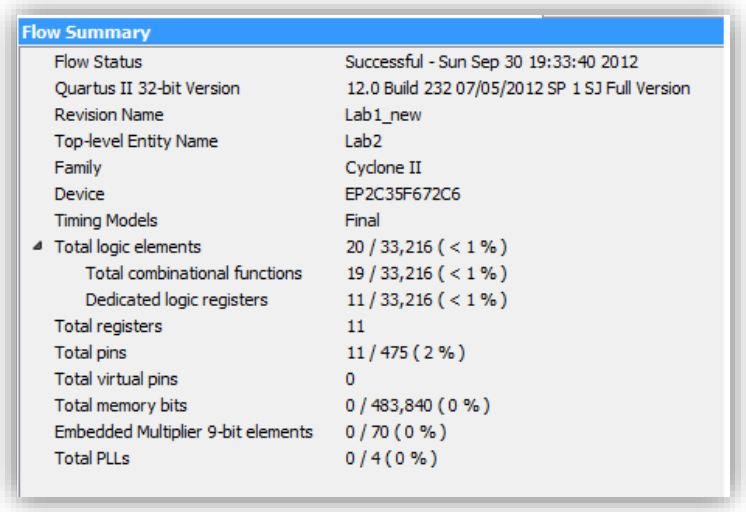

Figure 3.5: Quartus Flow Summary

The flow summary gives a lot of information about our lab. It states that our top-level entity is Lab2, and that we were using a Cyclone II EP2C35F672C6 board. We used 20 total logic elements, 11 of them were dedicated logic registers, and 19 total combinational functions.

The logic elements were close to what we expected based off of Lab 1, but not exact. The 11 dedicated logic registers originated from the hex driver port mappings. In lab 1 we had a total of 11 combinational functions after we optimized it. We are assuming that Quartus converted some of our logic to NAND or NOR logic for speed issues.

4.0 Conclusion

In conclusion, the experiment went exceedingly well. The objective of the experiment was to implement Lab 1 in Quartus. We split the project into three modules: the counter, hex driver, and the master file. Each module works according to specifications and the correct results were achieved.

During the lab we learned a lot of new techniques. The biggest thing we learned was how to use Quartus. Neither of us had much prior experience in VHDL. Working through an entire project in Quartus was a great learning experience. We also learned the basics of VHDL programming, such as how to create entities, assign port maps, establish processes, and instantiate variables. Within Quartus we specifically learned how to create projects, set modules as top level entities, assign ports, and compile complete projects. The lessons we learned in Lab 2 should be applicable to the rest of the labs this semester.

A. Appendix A – Code

A.1 HexDriver.vhd

```
-- Lab 2
-- Hex Driver
-- Steve Comer
-- Stuart Larsen
-- Team A-Shred
-- Updated 18 Sept 2012
--		This entity displays a hexadecimal character based on a 4-bit input vector.

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY HexDriver IS

PORT(numberToDisplay  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
		sevenSegmentOut  : OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
END HexDriver;

ARCHITECTURE Behavioral OF HexDriver IS
	BEGIN

		HexDriver : PROCESS(numberToDisplay)
			BEGIN
				CASE numberToDisplay IS
			-- Disp
					WHEN "0000" => sevenSegmentOut <= "1000000";		-- 0
					WHEN "0001" => sevenSegmentOut <= "1111001";		-- 1
					WHEN "0010" => sevenSegmentOut <= "0100100";		-- 2
					WHEN "0011" => sevenSegmentOut <= "0110000";		-- 3
					WHEN "0100" => sevenSegmentOut <= "0011001";		-- 4
					WHEN "0101" => sevenSegmentOut <= "0010010";		-- 5
					WHEN "0110" => sevenSegmentOut <= "0000010";		-- 6
					WHEN "0111" => sevenSegmentOut <= "1111000";		-- 7
					WHEN "1000" => sevenSegmentOut <= "0000000";		-- 8
					WHEN "1001" => sevenSegmentOut <= "0010000";		-- 9
					WHEN "1010" => sevenSegmentOut <= "0001000";		-- A
					WHEN "1011" => sevenSegmentOut <= "0000011";		-- b
					WHEN "1100" => sevenSegmentOut <= "1000110";		-- C
					WHEN "1101" => sevenSegmentOut <= "0100001";		-- d
					WHEN "1110" => sevenSegmentOut <= "0000110";		-- E
					WHEN "1111" => sevenSegmentOut <= "0001110";		-- F
					WHEN OTHERS => sevenSegmentOut <= "1111111";		-- null
				END CASE;

			END PROCESS HexDriver;

	END Behavioral;
```

A.2 Counter.vhd

```
-- Lab 2
-- Sequence Counter
-- Steve Comer
-- Stuart Larsen
-- Team A-Shred
-- Updated 18 Sept 2012
--		This entity outputs a sequence of states, based on the previous state. The next state is changed on clock
--		high. The sequence can be reset back to its initial state with the synchronous reset pin.

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY counter IS
PORT(Reset		: IN  STD_LOGIC;
		Clock		: IN	STD_LOGIC;
		Q			: OUT STD_LOGIC_VECTOR(0 TO 3) := "0011"; -- set initial value to 3
		clockLED	: OUT STD_LOGIC;
		resetLED	: OUT STD_LOGIC);

TYPE State IS (S0, S1, S2, S3, S4, S5, S6, S7);
SIGNAL cs : State := S3; -- initial state set to 3

END counter;

ARCHITECTURE Behavioral OF counter IS
	BEGIN
		clockLED <= NOT Clock;
		resetLED <= NOT Reset;

		counter : PROCESS(Clock)
			BEGIN
				IF (Clock'EVENT and Clock = '0' and Reset = '0') THEN
					Q <= "0011";
					cs <= S3;
				ELSIF (Clock'EVENT and Clock = '0' and Reset = '1') THEN
					CASE cs IS
						WHEN S3 =>
											-- if 3 -> 1
							cs <= S1;
							Q <= "0001";
						WHEN S1 =>
											-- if 1 -> 6
							cs <= S6;
							Q <= "0110";
						WHEN S6 =>
											-- if 6 -> 4
							cs <= S4;
							Q <= "0100";
						WHEN S4 =>
											-- if 4 -> 2
							cs <= S2;
							Q <= "0010";
						WHEN S2 =>
											-- if 2 -> 0
							cs <= S0;
							Q <= "0000";
```

```
                                        WHEN S0 =>
                                                                -- if 0 -> 7
                                            cs <= S7;
                                            Q <= "0111";
                                        WHEN S7 =>
                                                                -- if 7 -> 5
                                            cs <= S5;
                                            Q <= "0101";
                                        WHEN S5 =>
                                                                -- if 5 -> 3
                                            cs <= S3;
                                            Q <= "0011";
                                        WHEN OTHERS =>
                                                                -- if other -> 3
                                            cs <= S3;
                                            Q <= "0011";
                                END CASE;
                        END IF;
                END PROCESS counter;
END Behavioral;
```

A.3 Lab2.vhd

```
-- Lab 2
-- Master File (Uses Counter & Hex Driver)
-- Steve Comer
-- Stuart Larsen
-- Team A-Shred
-- Updated 18 Sept 2012
--		Combines both Counter and HexDriver to create a sequence generator with a hexadecimal display.

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Lab2 IS

PORT(Reset 			: IN  STD_LOGIC;
	Clock 			: IN 	STD_LOGIC;
	sevenSegmentOut : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
	clockLED 		: OUT STD_LOGIC;
	resetLED 		: OUT STD_LOGIC);

END Lab2;

ARCHITECTURE Behavioral OF Lab2 IS

COMPONENT counter
	PORT(Reset  : IN  STD_LOGIC;
		Clock    : IN 	STD_LOGIC;
		Q 		 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
		clockLED 		: OUT STD_LOGIC;
		resetLED 		: OUT STD_LOGIC);
END COMPONENT;

COMPONENT HexDriver
	PORT(numberToDisplay  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
	   sevenSegmentOut 	  : OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
END COMPONENT;

SIGNAL binaryNumberOut : STD_LOGIC_VECTOR(3 DOWNTO 0);
	BEGIN

		counter1 : counter PORT MAP(Reset, Clock, binaryNumberOut, clockLED, resetLED);
		hexDriver1 : HexDriver PORT MAP(binaryNumberOut, sevenSegmentOut);

	END Behavioral;
```

B. Appendix B – Quartus II Pinout

To	Direction	Location	I/O Bank	VREF Group	Fitter Location
numberToDisplay[3]	Input	PIN_AE14	7	B7_N1	PIN_AE14
numberToDisplay[2]	Input	PIN_P25	6	B6_N0	PIN_P25
numberToDisplay[1]	Input	PIN_N26	5	B5_N1	PIN_N26
numberToDisplay[0]	Input	PIN_N25	5	B5_N1	PIN_N25
sevenSegmentOut[6]	Output	PIN_V13	8	B8_N0	PIN_V13
sevenSegmentOut[5]	Output	PIN_V14	8	B8_N0	PIN_V14
sevenSegmentOut[4]	Output	PIN_AE11	8	B8_N0	PIN_AE11
sevenSegmentOut[3]	Output	PIN_AD11	8	B8_N0	PIN_AD11
sevenSegmentOut[2]	Output	PIN_AC12	8	B8_N0	PIN_AC12
sevenSegmentOut[1]	Output	PIN_AB12	8	B8_N0	PIN_AB12
sevenSegmentOut[0]	Output	PIN_AF10	8	B8_N0	PIN_AF10
Clock	Unknown	PIN_G26	5	B5_N0	
Reset	Unknown	PIN_N23	5	B5_N1	
clockLED	Unknown	PIN_AE22	7	B7_N0	
resetLED	Unknown	PIN_AF22	7	B7_N0	
Q[0]	Unknown	PIN_AC22	7	B7_N0	
Q[1]	Unknown	PIN_AB21	7	B7_N0	
Q[2]	Unknown	PIN_AF23	7	B7_N0	
Q[3]	Unknown	PIN_AE23	7	B7_N0	