EECE 229 Fall 2016

Assignment #2 Lab

Tim Hamling
Partner: Michelle Blum
November 3rd, 2016

Overview

As a continuation of Lab Assignment 2 Preparation, this assignment demonstrates the application of Verilog modules to Boolean functions. In this assignment, we will look at expressions, and, using our knowledge from the past few weeks of lectures, convert these expressions to K-Maps and Verilog modules. We will also implement different operation in Verilog, such as binary AND, OR, multiplexer, comparator, and subtractor modules. Finally, we will be looking at a design problem that deals with the engine speed control system of a car.

In Q1, we will evaluate two functions and then build a Karnaugh-map to minimize the solutions. A Verilog module will be created and tested in order to implement the two minimized solutions. For Q2, we will create two separate Verilog modules. The first will perform an 8-bit binary AND, and the second will perform an 8-bit binary OR. We then will test this function using the DE2-115 hardware.

In Q3, we will build a code that allows all fifteen of the hex digits to be represented by four input switches. The digits will then be displayed across the HEX display digits on the DE2-115 hardware. For Q4, we will use Verilog to create and test a 4-input multiplexer in order to demonstrate the $\log_2 n$ possibilities for the outcome of the circuit. Afterwards, we will create and test an 8-bit unsigned magnitude comparator to evaluate generated values. For Q5, we will continue to implement previously learned procedures in the Verilog language through creating and testing an 8-bit adder and an 8-bit subtractor. A process that would be time consuming to carryout on paper will become quick and easy in Verilog!

In Q6, we will create a module and BDF file that adds, subtracts, ANDs, and ORs two 8-bit values. The result will be displayed on the HEX displays. For Q7, we were asked to design a circuit for Toyota. We were given a circuit design in paragraph form, and asked to convert it to a Boolean expression. From this, we will create an SOP or POS expression to fit the system. Using this expression, we will construct a Verilog module and test the module using an LED as the output and switches as the inputs.

The work that will be performed in this assignment will be very helpful since it reinforces the concepts of Verilog implementation. Actually running the programs and ensuring that everything we have learned thus far has been interpreted appropriately is crucial to cementing our understanding of the programming language.

Solutions

Q1:

We were asked to evaluate the stated functions and construct Karnaugh-maps from our interpretation. From the Karnaugh-map, we were to minimize the functions F and G. After our final functions were established, we were asked to implement them in the Verilog software. Our only restriction was to use one module that had five inputs and two outputs. As shown in **Table 1** and **Table 2** below, the Karnaugh-maps we generated for functions F and G consisted of eight possible combinations. When grouping the places where minterms showed, we were able to form **Equation 3** for function F and **Equation 4** for function G. **Code Block 1** below demonstrates our implementation of the functions in Verilog. We chose a structural approach.

$$F = \sum\nolimits_{A,B,C}(2,6)$$

Equation 1– Function given for F

$$G = \sum\nolimits_{C,D,E}(0,2,3)$$

Equation 2– Function given for G

BC

A

	00	01	11	10
0	0	0	0	1
1	0	0	0	1

Table 1 – Karnaugh Map for Function F

$$\boxed{F = B \bullet C'}$$

Equation 3 – Minimized Expression for Function F

DE

C

Table 2 – Karnaugh Map for Function G

	00	01	11	10
0	1	0	1	1
1	0	0	0	0

$$\boxed{G = (C' \bullet D) + (E' \bullet C')}$$

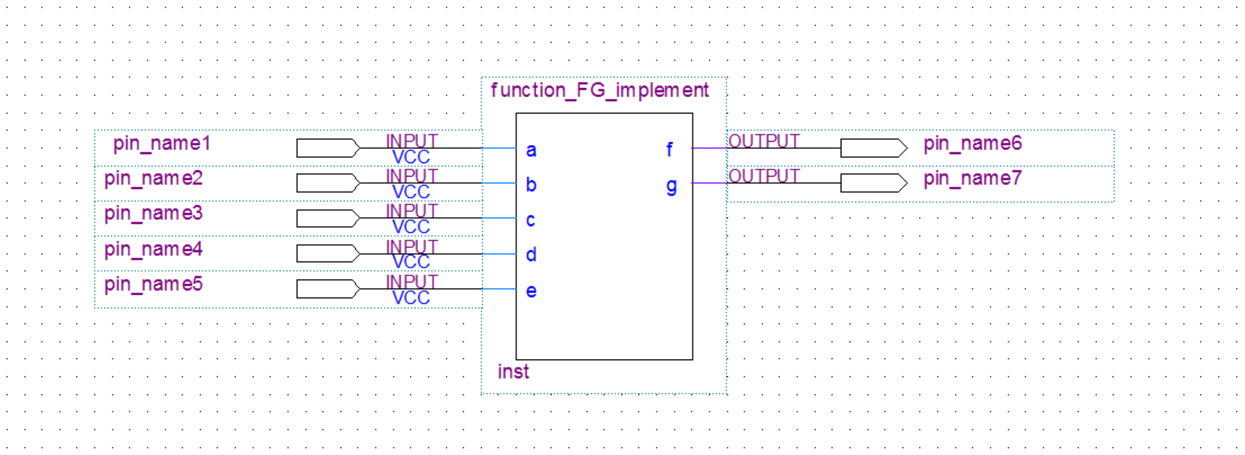
Equation 4 – Minimized Expression for Function G

```
module FG (input a,b,c,d,e, output f,g) ;
      and(f,b,~c);

      and(h,~c,d);
      and(i,~e,~c);
      or(g,h,i);

endmodule
```

Code Block 1 – Implementation of Functions F and G in Verilog

We tested the module by assigning each input to a switch and each of the outputs to an LED. The first LED would light up when the switches were turned on according to **Equation 3**, and the second LED would light up when the switches were turned on according to **Equation 4**. The BDF file can be seen in **Figure 1**.

Figure 1 – BDF File for functions F and G



Q2:

We were asked to create two modules using the Verilog language. The first module was to perform an 8-bit binary AND. The second module was to perform an 8-bit binary OR. This means that the bit in the 1's column of the first number would be compared to the bit in the 1's column of the second number, and they would either be compared with an AND or an OR operation. Then, the bits in the 2's columns would be compared the same way, then the bits in the 4's column, and so on. The Verilog module for these operations was very easy to create because Verilog has a built-in bitwise AND, as well as a bitwise OR operation included. **Code Block 2** below shows the Verilog code for a binary AND module.

Code Block 2 – BinaryAND Module

```
module binaryAND(input wire[7:0] inValA, inValB, output wire[7:0] outVal);
  assign outVal = inValA & inValB;
endmodule
```

The code for this is very simple; it takes in two 8-bit numbers, inValA and inValB, and performs a bitwise AND operation on them using the & operand. Then, it assigns the result of this to outVal. The 8-Bit binary OR module is identical, but replaces the & operand with an | operand because it will perform an OR comparison, not an AND comparison, on each bit. The code for this is shown in **Code Block 3**.

```
module binaryOR(input wire[7:0] inValA, inValB, output wire[7:0] outVal);
  assign outVal = inValA | inValB;
endmodule
```

Code Block 3 – BinaryOR Module

To test the binaryAND module, we used a bus to assign 8 switches to one binary input, and another bus to assign 8 switches to the second binary input. We displayed the output using 8 LEDs that turned on when the corresponding position in the binary number was 1, and off when the position in the number was 0. This procedure was repeated for the binaryOR module. The schematic for each of the tests can be seen in **Figure 2** and **Figure 3**.

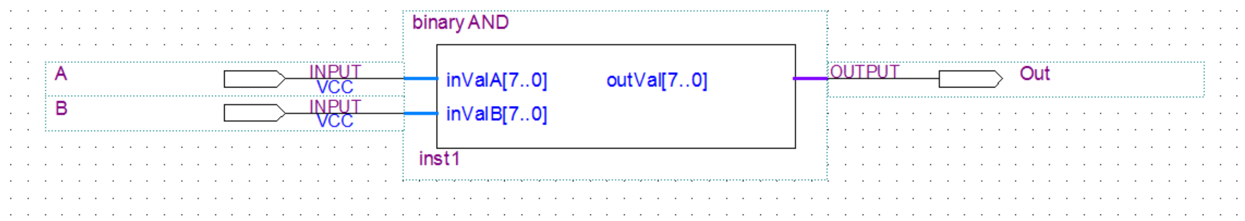


Figure 2 – BDF File for binaryAND

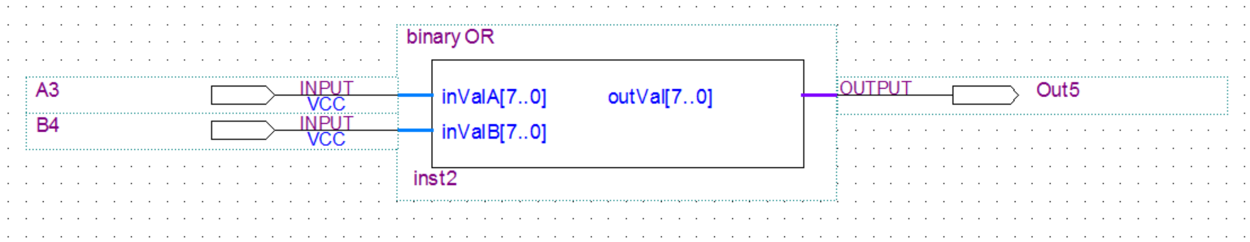


Figure 3 – BDF File for binaryOR

Q3:

For this question, we were asked to display hex numbers 0 through F on the HEX display digits using input switches. We used the first four switches: the first switch represented 1, the second switch represented 2, the third switch represented 4, and the fourth represented 8. These four numbers were able to add up to any of the 16 numbers using different combinations.
For our code, we had an input of size four, and an output of size 7. Using a case structure, we defined each case as a unique combination of the switches (or different amalgamations of 1, 2, 4 and 8 being added together). Whenever a case was met, the code instructed the HEX0 display to light up specific segments. The number 0 represented a segment that was to be lit, while the number 1 represented an unlit segment. The full code for HEX0 is shown below in **Code Block 4**.

```
module SevenSegment(SW, HEX0);
        input [3:0] SW;
        output reg [0:6] HEX0;

        always @(SW)
        case(SW)

                4’b0000: HEX0 = 7’b0000001;
                4'b0001: HEX0 = 7'b1001111;
                4'b0010: HEX0 = 7'b0010010;
                4'b0011: HEX0 = 7'b0000110;
                4'b0100: HEX0 = 7'b1001100;
                4'b0101: HEX0 = 7'b0100100;
                4'b0110: HEX0 = 7'b0100000;
                4'b0111: HEX0 = 7'b0001101;
                4'b1000: HEX0 = 7'b0000000;
                4'b1001: HEX0 = 7'b0001100;
                4'b1010: HEX0 = 7'b0001000;
                4'b1011: HEX0 = 7'b1100000;
                4'b1100: HEX0 = 7'b0110001;
                4'b1101: HEX0 = 7'b1000010;
                4'b1110: HEX0 = 7'b0110000;
                4'b1111: HEX0 = 7'b0111000;

     endcase

endmodule
```

Code Block 4 – HEX0 Display Module

In order to have the hex numbers display across all the HEX digit displays, we then copied this code, this time stating HEX1, HEX2, HEX3, etc. This concept is shown in the schematic of the full code, which is pictured in **Figure 4.** The code successfully compiled and ran appropriately, as demonstrated by **Figure 5.**

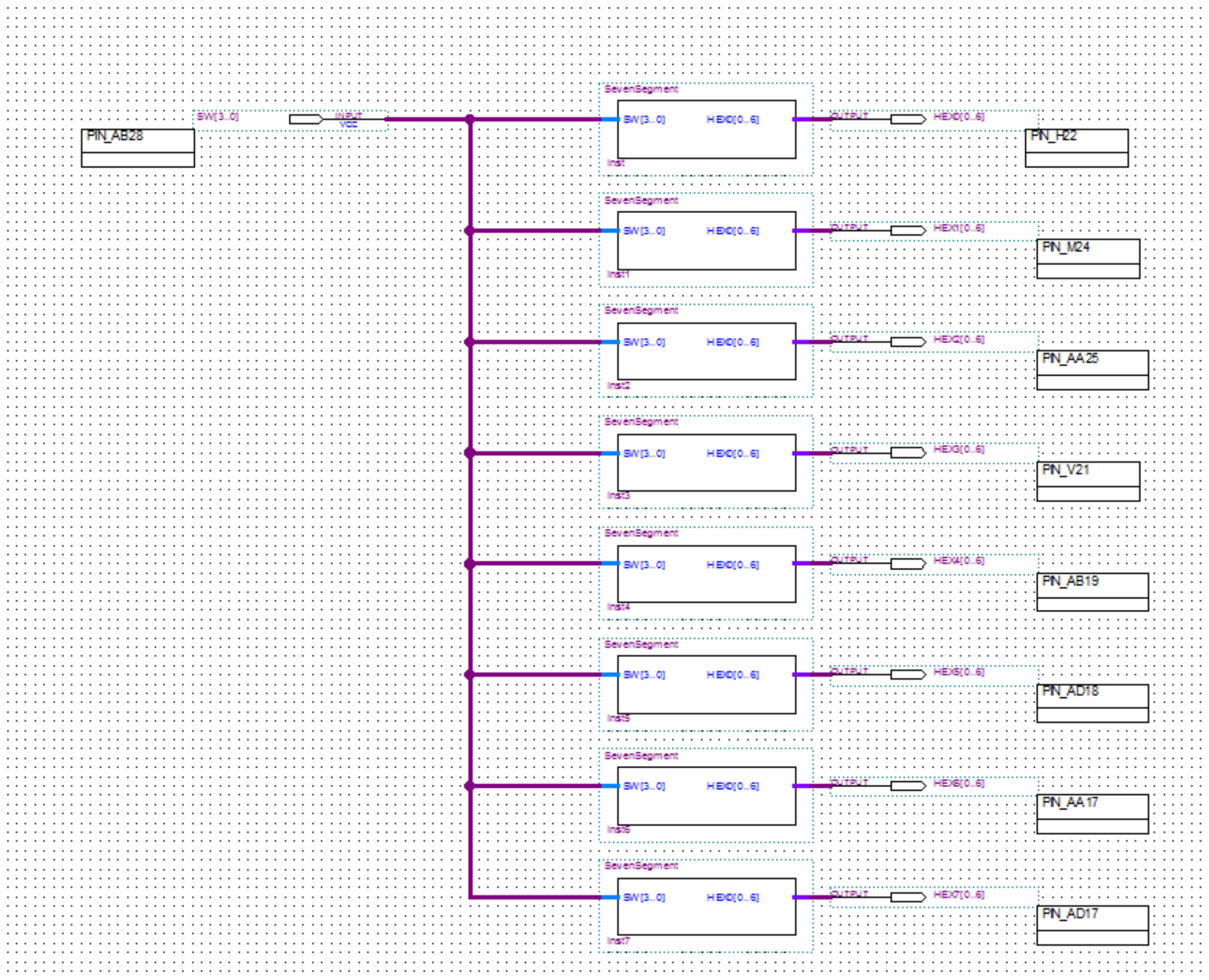
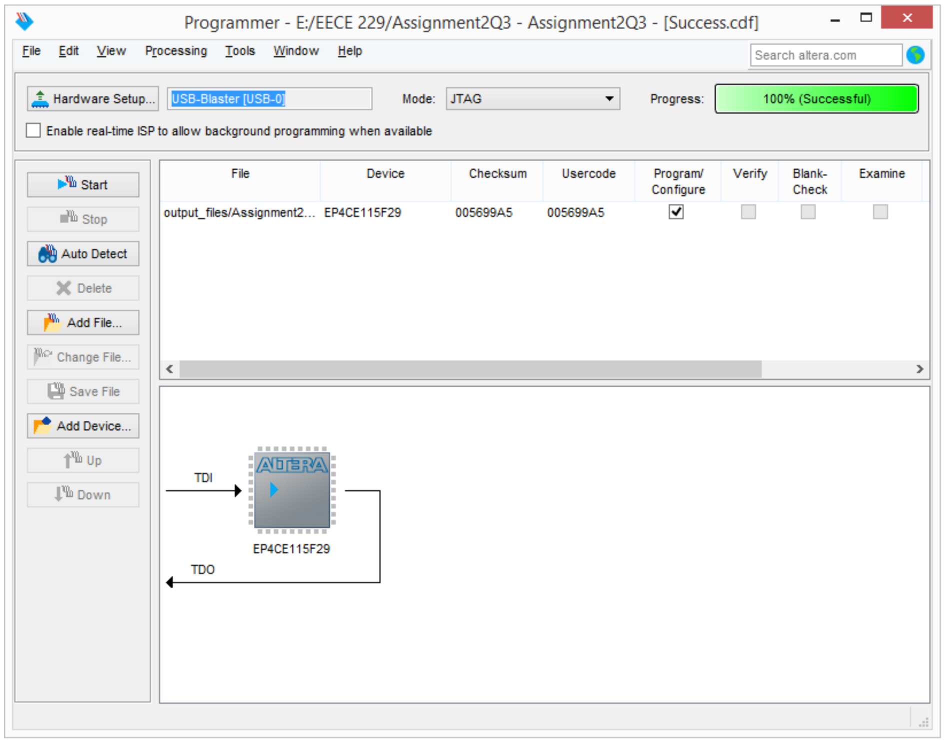


Figure 4 – HEX digit display BDF



Hardware Setup... USB-Blaster [USB-0]

Mode: JTAG

Progress: 100% (Successful)

Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine
output_files/Assignment2...	EP4CE115F29	005699A5	005699A5	✓			

Figure 5 – Successful Compilation Window

Q4:

This question asked us to create an 8-bit multiplexer with 4 inputs, as well as an 8-bit unsigned magnitude comparator. Both modules are going to be implemented in Verilog, and then tested on an Altera circuit board.

We started with the 8-bit multiplexer. The code is shown in **Code Block 5**. It takes in four input wires, each 8-bits as specified, and has a single 8-bit output. It also has a 2-bit selector, which will help the multiplexer choose which value to output. This is done within the case statement. Depending on the values given by the selector, the output value is assigned to one of the four corresponding input wires.

```
module FourInMultiplexer(a, b, c, d,
                            sel, z);
input wire[7:0] a, b, c, d;
input wire[1:0] sel;
output reg[7:0] z;

always @(sel or a or b or c or d)
  begin
    case(sel)
      2'b00 : z = a;
      2'b01 : z = b;
      2'b10 : z = c;
      2'b11 : z = d;
    endcase
  end
endmodule
```

Code Block 5 - Verilog module representing the 4-input Multiplexer

The next module had to be an 8-bit unsigned comparator. To solve this problem, three comparison operators were used: Greater Than, Less Than, and Equals To. These are represented in Verilog by the following signs, respectively: > < and ==. The two values that had to be inputs were supposed to be 8-bits, so they were made to be wire vectors. We had three outputs, one for each comparison, and assigned these to the resulting values of the two comparisons. The code for this module is shown in **Code Block 6.**

```
module UnsignedComparator(A, B, lt, gt, eq);
input wire[7:0] A, B;
output wire lt, gt, eq;

  assign lt = A < B;
  assign gt = A > B;
  assign eq = A == B;

endmodule
```

Code Block 6 - Verilog module representing the 8-bit Comparator

To test these modules, we compiled them in the Quartus sotware and created BSF files from the Verilog code. We then added them to a block diagram file, and assigned input and output pins to each of the necessary inputs and outputs. The screenshot of the two resulting boards are shown in **Figure 6** and **Figure 7**. Unfortunately, we were not able to assign pins or run our .bdf on the Altera Board because the number of inputs needed exceeded the number of switches available on the board.

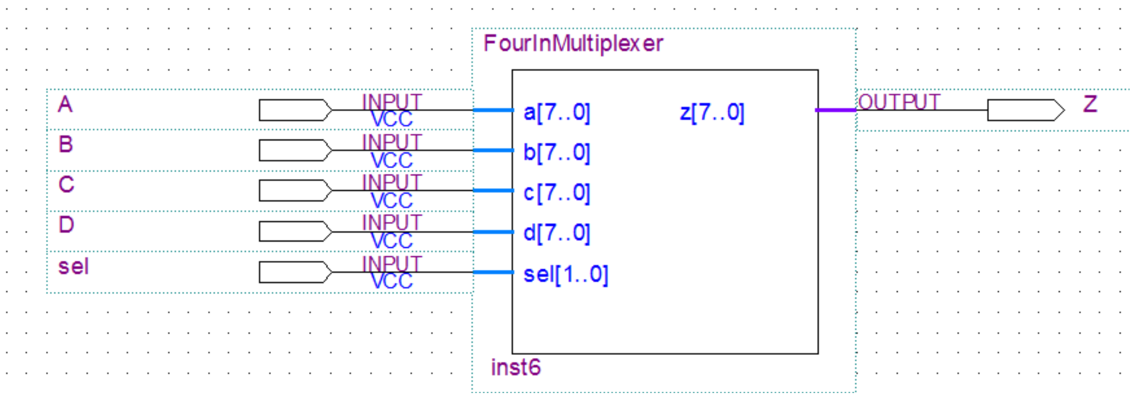


Figure 6 - BDF representation of the Multiplexer module

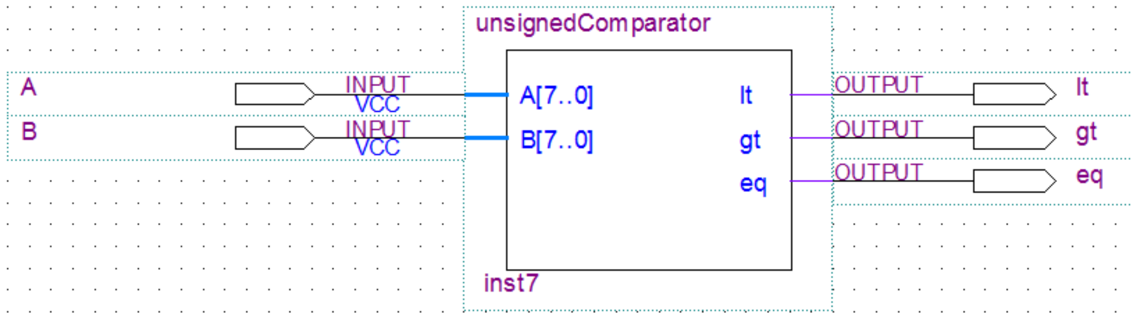


Figure 7 - BDF representation of the Unsigned Comparator module

Q5:

For this question, we had to create two more modules in Verilog and test them on the Altera board. One module had to be an 8-bit adder, and the other had to be an 8-bit subtractor.
These were both essentially the same modules, but had one change which operator was used. They both took in two 8-bit values, and had one 8-bit wire as an output. This output was assigned to be either the sum of the two input values, or the difference of the two inputs, depending on if it was the adder or the subtractor. The adder module is shown in **Code Block 7**, and the subtractor module is shown in **Code Block 8**.

```
module adder(A, B, sum);
input wire[7:0] A, B;
output wire[7:0] sum;

  assign sum = A + B;

endmodule
```

Code Block 7 - Verilog module representing the adder

```
module subtractor(A, B, diff);
input wire[7:0] A, B;
output wire[7:0] diff;

  assign diff = A - B;

endmodule
```

Code Block 8 - Verilog module representing the subtractor

To test these modules, we compiled the code in Quartus and created BSF files. We then put these files on a block diagram and attached input and output pins. Our diagram is shown below in **Figure 8**.

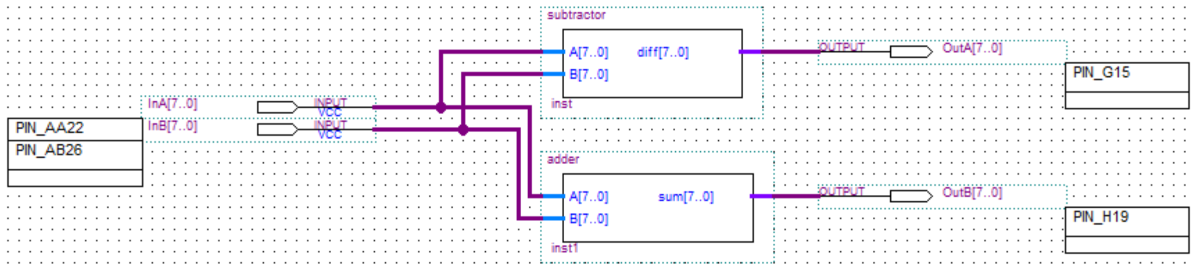


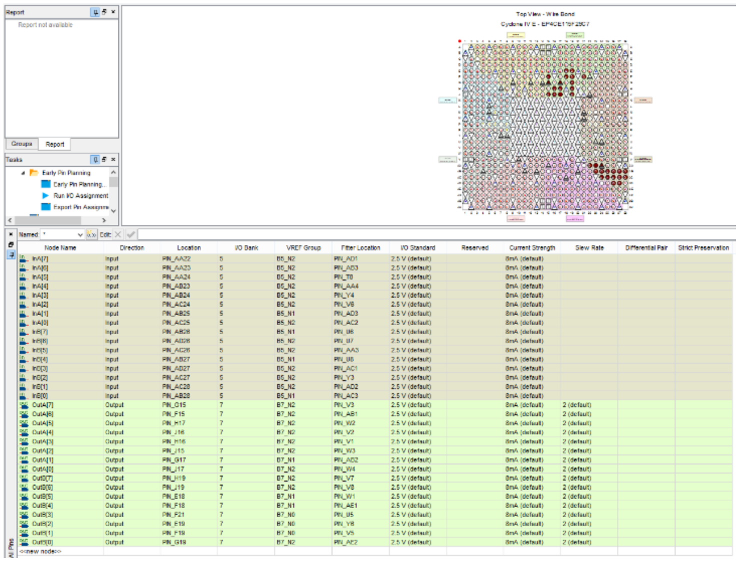
Figure 8 - Block Diagram of the Adder and Subtractor, with pin assignments

For the inputs, we used 8 switches to represent the first value, and 8 other switches to represent the other value. The inputs were then split to the two modules. We then had the output for the adder display across 8 LEDs, and the output for the subtractor display on 8 different LEDs.

Next, we assigned pins to each input and output. Since each used multiple values, we had to use a bus to assign 8 pins to each. The pins are shown on the BDF file above, and on the pin planner window shown in **Figure 9** to the right.

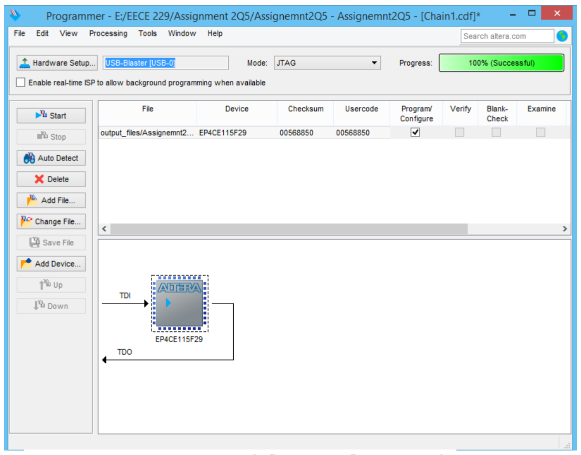
Finally, we could compile and run our modules on the Altera Board. The successful compilation report is shown in **Figure 10**.

When we ran it, we could use 16 switches to select two different 8-bit values. The sum of them was shown across 8 LEDs, while the difference of the two was shown on the other LEDs.



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservation
InA[7]	Input	PIN_AA22	5	B5_N2	PIN_AD1	2.5 V (default)		8mA (default)			
InA[6]	Input	PIN_AA23	5	B5_N2	PIN_AD3	2.5 V (default)		8mA (default)			
InA[5]	Input	PIN_AA24	5	B5_N2	PIN_T8	2.5 V (default)		8mA (default)			
InA[4]	Input	PIN_AB23	5	B5_N2	PIN_AA4	2.5 V (default)		8mA (default)			
InA[3]	Input	PIN_AB24	5	B5_N2	PIN_Y4	2.5 V (default)		8mA (default)			
InA[2]	Input	PIN_AC24	5	B5_N2	PIN_V6	2.5 V (default)		8mA (default)			
InA[1]	Input	PIN_AB25	5	B5_N1	PIN_AD3	2.5 V (default)		8mA (default)			
InA[0]	Input	PIN_AC25	5	B5_N2	PIN_AC2	2.5 V (default)		8mA (default)			
InB[7]	Input	PIN_AB26	5	B5_N1	PIN_U6	2.5 V (default)		8mA (default)			
InB[6]	Input	PIN_AD26	5	B5_N2	PIN_U7	2.5 V (default)		8mA (default)			
InB[5]	Input	PIN_AC26	5	B5_N2	PIN_AA3	2.5 V (default)		8mA (default)			
InB[4]	Input	PIN_AB27	5	B5_N1	PIN_U8	2.5 V (default)		8mA (default)			
InB[3]	Input	PIN_AD27	5	B5_N2	PIN_AC1	2.5 V (default)		8mA (default)			
InB[2]	Input	PIN_AC27	5	B5_N2	PIN_Y3	2.5 V (default)		8mA (default)			
InB[1]	Input	PIN_AC28	5	B5_N2	PIN_AD2	2.5 V (default)		8mA (default)			
InB[0]	Input	PIN_AB28	5	B5_N1	PIN_AC3	2.5 V (default)		8mA (default)			
OutA[7]	Output	PIN_G15	7	B7_N2	PIN_V3	2.5 V (default)		8mA (default)	2 (default)		
OutA[6]	Output	PIN_F15	7	B7_N2	PIN_AB1	2.5 V (default)		8mA (default)	2 (default)		
OutA[5]	Output	PIN_H17	7	B7_N2	PIN_W2	2.5 V (default)		8mA (default)	2 (default)		
OutA[4]	Output	PIN_J16	7	B7_N2	PIN_V2	2.5 V (default)		8mA (default)	2 (default)		
OutA[3]	Output	PIN_H16	7	B7_N2	PIN_V1	2.5 V (default)		8mA (default)	2 (default)		
OutA[2]	Output	PIN_J15	7	B7_N2	PIN_W3	2.5 V (default)		8mA (default)	2 (default)		
OutA[1]	Output	PIN_G17	7	B7_N1	PIN_AD2	2.5 V (default)		8mA (default)	2 (default)		
OutA[0]	Output	PIN_J17	7	B7_N2	PIN_W4	2.5 V (default)		8mA (default)	2 (default)		
OutB[7]	Output	PIN_H19	7	B7_N2	PIN_V7	2.5 V (default)		8mA (default)	2 (default)		
OutB[6]	Output	PIN_J19	7	B7_N2	PIN_V8	2.5 V (default)		8mA (default)	2 (default)		
OutB[5]	Output	PIN_E18	7	B7_N1	PIN_W1	2.5 V (default)		8mA (default)	2 (default)		
OutB[4]	Output	PIN_F18	7	B7_N1	PIN_AE1	2.5 V (default)		8mA (default)	2 (default)		
OutB[3]	Output	PIN_F21	7	B7_N0	PIN_U5	2.5 V (default)		8mA (default)	2 (default)		
OutB[2]	Output	PIN_E19	7	B7_N0	PIN_Y8	2.5 V (default)		8mA (default)	2 (default)		
OutB[1]	Output	PIN_F19	7	B7_N0	PIN_V5	2.5 V (default)		8mA (default)	2 (default)		
OutB[0]	Output	PIN_G18	7	B7_N2	PIN_AE2	2.5 V (default)		8mA (default)	2 (default)		
<<new node>>											

Figure 9 - Pin planner for the Adder and Subtractor board



Programmer - E:/EECE 229/Assignment 2Q5/Assignemnt2Q5 - Assignemnt2Q5 - [Chain1.cdf]*

Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress: 100% (Successful)

Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine
output_files/Assignemnt2...	EP4CE115F29	00568850	00568850	✓			

Figure 10 - Successful compilation of our two modules on the Altera Board

Q6:

This problem required us to create a circuit that could add/subtract/AND/OR two 8-bit values. The two inputted values should be displayed across four 7-segment displays, and the resulting value should be shown on two other 7-segment displays. Two switches should be used to select an operation from one of the four available operations.

We began by creating a multiplexer that took in two 8-bit values and a selector, and outputted an 8-bit result based on the selection. The Verilog code for this module is shown in **Code Block 9**. The selector could have four possible values: 00, which represented an AND operation, 01, which represented an OR operation, 10, which represented an

```
module Operation(inValA, inValB,
                 sel, outVal);

input wire[7:0] inValA, inValB;
input wire[1:0] sel;
output reg[7:0] outVal;

always @(sel)
  begin
    case(sel)
      2'b00 : outVal = inValA & inValB;
      2'b01 : outVal = inValA | inValB;
      2'b10 : outVal = inValA + inValB;
      2'b11 : outVal = inValA - inValB;
    endcase
  end
endmodule
```

Code Block 9 - Module that takes in values, and depending on the selector, evaluates the output addition operation, and 11, which represented a subtraction operation. The output variable was assigned to the resulting value produced by the selected operation.

Our next module was used to display a Hex value on a 7-segment display unit. The code for the module is shown in **Code Block 10**. It takes in a single hex value, which is 4 bits, and depending on which number it is, assigns the values of the 7 parts of the output to represent that number. Any segment that should be displayed is represented with a 0, and any segment that should not be lit up is represented with a 1. This module is identical to that of Question 3 above.

```
module SevenSegment(SW, HEX0);
input [3:0] SW;
output reg [0:6] HEX0;

  always @(SW)
    case(SW)
      4'b0000: HEX0 = 7'b0000001;
      4'b0001: HEX0 = 7'b1001111;
      4'b0010: HEX0 = 7'b0010010;
      4'b0011: HEX0 = 7'b0000110;
      4'b0100: HEX0 = 7'b1001100;
      4'b0101: HEX0 = 7'b0100100;
      4'b0110: HEX0 = 7'b0100000;
      4'b0111: HEX0 = 7'b0001101;
      4'b1000: HEX0 = 7'b0000000;
      4'b1001: HEX0 = 7'b0001100;
      4'b1010: HEX0 = 7'b0001000;
      4'b1011: HEX0 = 7'b1100000;
      4'b1100: HEX0 = 7'b0110001;
      4'b1101: HEX0 = 7'b1000010;
      4'b1110: HEX0 = 7'b0110000;
      4'b1111: HEX0 = 7'b0111000;
       endcase
endmodule
```

Code Block 10 - Module that displays a Hex value on a 7-segment display

The next two modules we created were used to combine or break apart the input to fit the modules we created. For example, the numbers were inputted using 8 bits, but we broke that into two 4-bit input busses. We did this because the SevenSegment module required inputs of 4 bits. However, the Operation module required an input of 8 bits. So, we created one module that took in two 4-bit values and outputted an 8-bit value representing the two, and another module that took in an 8-bit value and outputted two 4-bit values representing that 8-bit value. These two modules are shown in **Code Block 11**, and work by using the indexes of the input and output numbers to reorganize the needed result.

```
module Split8bit(In, Out0, Out1);
input wire[7:0] In;
output wire[3:0] Out0, Out1;

  assign Out0 = In[3:0];
  assign Out1 = In[7:4];

endmodule

module Combine8bit(In0, In1, Out);
input wire[3:0] In0, In1;
output wire[7:0] Out;

  assign Out[3:0] = In0;
  assign Out[7:4] = In1;

endmodule
```

Code Block 11 - One module that splits an 8-bit number, and one module that combines two 4-bit numbers

With all four modules created, we were ready to create the BDF representations of each of them. We then added them to a block diagram file, and connected input pins and output pins to each. Next, we ran wires that connected each module to each other module. The full board is shown below in **Figure 11**. We had four 4-bit input busses which would be controlled with switches. We then ran these values into four copies of our SevenSegment modules so that the inputted numbers would be displayed on the board. The output pins of the seven segment displays were assigned to the display segments on the Altera board. We also took the inputted values, and ran each pair of them into our Combine8Bit module to form an 8-bit number for each of them. The two resulting 8-bit values were run into the Operation module, along with a 2-bit input bus that would be controlled with 2 switches. This would be used to determine what operation would be performed. The output of the Operation module was fed into the Split8Bit module which returned two 4-bit values. These were then attached to the SevenSegment display module, which connected them to the two final 7-segment units on the Altera board.

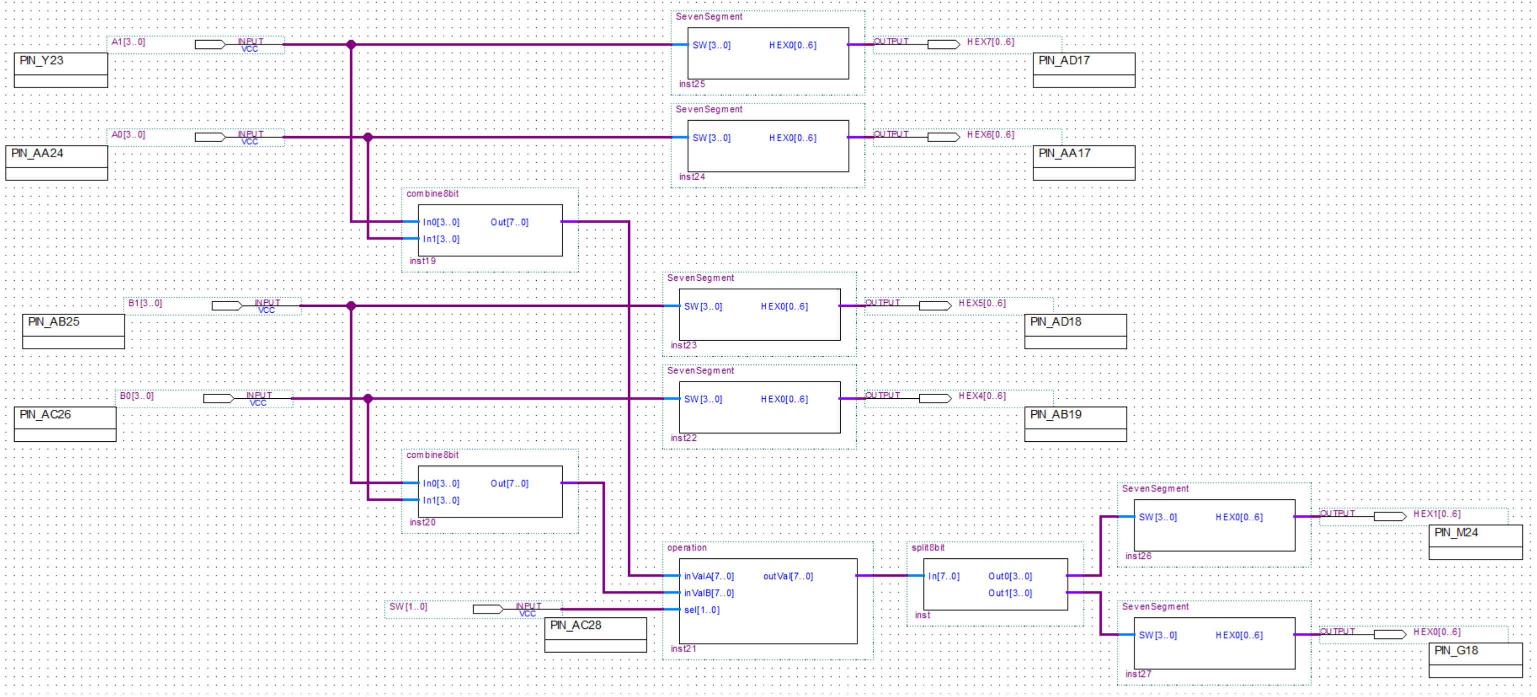
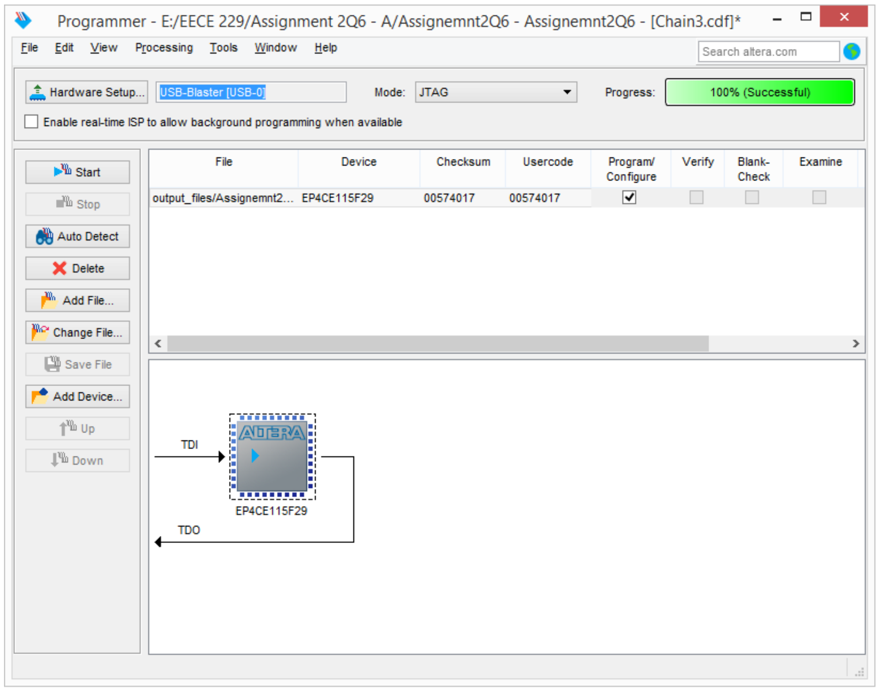


Figure 11 - Board representing our add/subtract/AND/OR circuit for two 8-bit values

Finally, after assigning all the input pins to corresponding switches and the output pins to corresponding segments of the displays, we compiled and ran our circuit on the Altera board. It worked perfectly, and we used an online calculator to verify all our answers. One issue we found was that the circuit could not accound for the overflow that resulted from the addition of two numbers that were above FF (or 255 in decimal) or a negative number that resulted from the subtraction of a large number from a smaller number. Aside from these issues, our circuit worked perfectly. The successful compilation is shown in **Figure 12**.


Programmer - E:/EECE 229/Assignment 2Q6 - A/Assignemnt2Q6 - Assignemnt2Q6 - [Chain3.cdf]*

Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress: 100% (Successful)

Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine
output_files/Assignemnt2...	EP4CE115F29	00574017	00574017	✓			

Figure 12 - Successful compilation of our circuit on the Altera Board.

Q7:

This final question was a schematic design problem. We had to design a circuit that represented the speed control system of a Toyota car. There were a few cases we had to follow for this system. First, the system should be disabled whenever the brake is applied, or whenever both the brake and throttle are not applied. The system should be enabled whenever both the throttle is engaged and the vehicle is in drive.

To begin, we decided to assign names to each of the input and output signals. We named the brake input B, the throttle input T, and the drive input D. We also named the output SC, for speed control. Now that all the signals were named, we were ready to construct a truth table. Because there are 3 inputs, there will be 8 rows to the table. We then filled in the output column with 1's wherever the system had to be engaged, and 0's wherever it had to be disabled. The specifications stated that it must be disabled whenever the brake is applied, so we put 0's in any column where B was 1. It must also be disabled where the brake and throttle are not applied, so we filled in the spaces where B and T are zero with a 0. The only case where the output should be 1 is when the throttle is applied and the car is in drive, so we put a 1 in the remaining column where T and D were both 1. This truth table is shown in **Table 3**.

B	T	D	SC
0	0	0	0
0	0	1	0
0	1	0	x
0	1	1	1
1	0	0	0
1	0	1	0

Table 3 – Truth table representing the speed control system circuit.

The remaining requirement of the design was to derive either a SOP or POS equation from the truth table. Because there was only one value of 1 in the table, it made more sense to create a SOP for this one maxterm on the table. The only working value was composed of having the brake off, throttle on, and car in drive. The representation of this equation is B'TD, and is shown in **Equation 5** below.

$$F = \sum_{B,T,D}(3) = B'TD$$

Equation 5 – Sum of Product expression for Table 4

Next, we had to translate this SoP expression into Verilog code so we could test it on the Altera board. Our code is shown in **Code Block 11** and is a simple visual representation of our SoP expression. We converted this module into a BDF file and attached input and output pins to the module. The resulting board is shown in **Figure 13**, and the successful compilation of the board is shown in **Figure 14**. We assigned the three inputs to three switches on the board, and the output to one LED on the board. When we compiled and ran the program on our board, we found that the only input combination of switches that turned the LED on was having the T and D switches on, and the B switch off. This simple test shows that our Verilog code and circuit indeed match the Truth Table we constructed.

```
module EngineControlSystem (B, T, D, SC);
input B, T, D;
output SC;
  assign SC = (~B & T & D);
endmodule
```

Code Block 12 – Verilog module representing the SoP expression

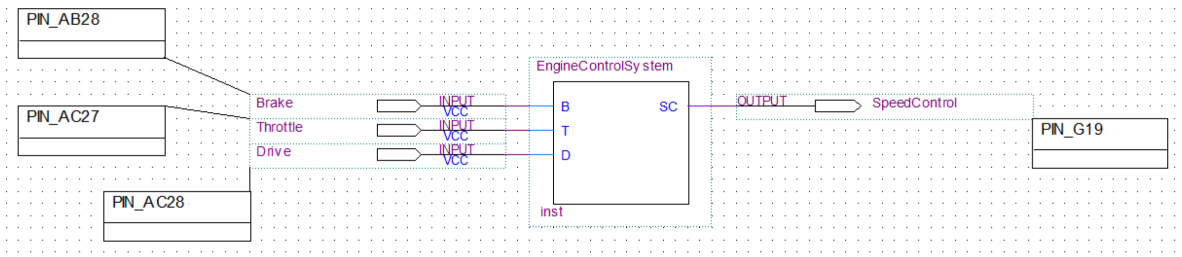
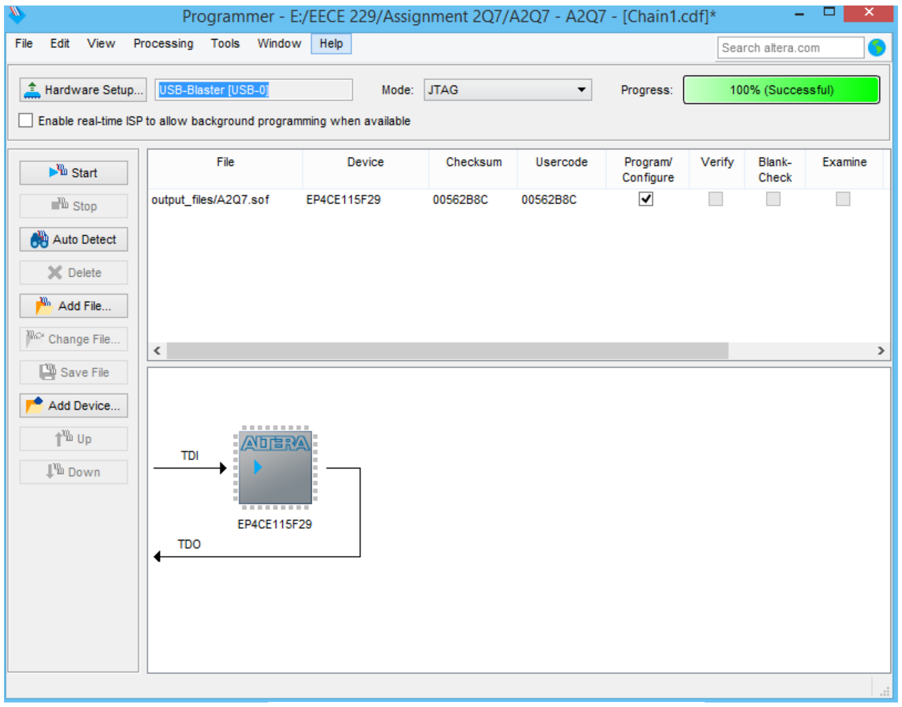


Figure 13 - Board diagram of our circuit



Programmer - E:/EECE 229/Assignment 2Q7/A2Q7 - A2Q7 - [Chain1.cdf]*

Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress: 100% (Successful)

☐ Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank- Check	Examine
output_files/A2Q7.sof	EP4CE115F29	00562B8C	00562B8C	✓	☐	☐	☐

Figure 14 - Successful compilation window