Lab 1: 16-Bit Adder Quartus

Brandon Chin

CSC343 - Instructor: Prof. Izidor Gertner

3/6/2015

Objective

In Lab 1, we designed a 16-Bit Adder/Subtractor in ModelSim. Now we must modify the lab to work in Quartus II. Then we must assign pins in order to display the functionality of our design on a DE2 Circuit Board. Due to the Board not having enough input or output ports, we must sign extend the first 8th bit of each input in order to simulate a 16-bit adder/subtractor. We must also display the output of our adder/subtractor through the seven-segment display in hexadecimal notation.

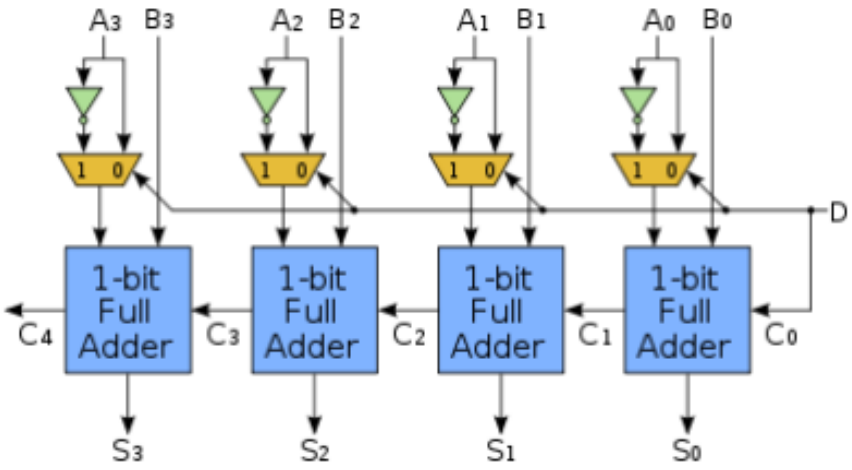
Functionality and Specifications

A bit adder is a digital circuit that is designed to take two binary string inputs, and return the sum of those inputs.

Component: Full Bit Adder

To design a 16-bit adder, we port map individual 1-bit adders together in sequence. If we recall from our previous lab, the 1-bit full adder VHDL entity is defined below:

```
-- FULL BIT ADDER
library ieee;
use ieee.std_logic_1164.all;

entity full_adder is
   port( x, y, cin : in std_logic;
         sum, cout : out std_logic);
end full_adder;

architecture ARCH of full_adder is
begin
   sum <= (x xor y) xor cin;
   cout <= (x and (y or cin)) or (cin and y);
end ARCH;
```



(4-bit Adder-Subtractor Digital Circuit Design)

Subtraction (w/ Two's Complement)

If we look at the above circuit design, we will see an example of a 4-bit adder-subtractor that we can use to help understand what we will be doing in our 16-bit adder-subtractor. In order to perform addition as well as subtraction, we have to use a method called two's complement, which allows us to represent negative integers in binary. Positive numbers are represented as themselves, however negative numbers are represented by the two's complement of their absolute value. In other words, take the absolute value of the negative number, invert each bit (0's become 1's and 1's become 0's), then add one to the string. This new form is your negative number. Something to notice -- strings whose most significant bit are a 0 are positive, and those that are a 1 are negative.

This can make it easier to perform subtraction -- In the diagram, there is an input called D (in our addsub16, it is called mode), and this is used to determine the operation. D is fed into two ports, one is the first 1-bit full adder as the cin, and the other is into a series of multiplexors (in our addsub16, we use XOR gates). If D is 0, the multiplexor allows the two inputs to enter the adders as normal, and sets the first cin to 0 as well. However, if D is 1, this inverts each of the input bits before it is passed into the adder, and the cin is set to 1, which adds 1 to our result. In other words, we are handling subtraction by essentially **adding A to the negative of B**.

Component: XOR Gate

In order to apply the functionality of subtraction, we will define an XOR gate component, similar to the multiplexor in the above example, which will achieve the same thing.

```
-- XOR gate
library ieee;
use ieee.std_logic_1164.all;

entity xorGate is
   port( a, b : in std_logic;
             f : out std_logic);
end xorGate;
--
architecture ARCH of xorGate is
begin
   f <= a xor b;
end ARCH;
```

Component: Sign Extension

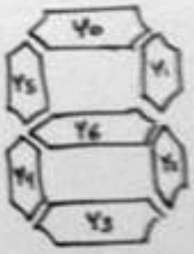
Our inputs for this lab will only be 8 bits long, however, we still want to simulate a 16-bit adder/subtractor. In order to do this, we will take the most significant bit of our inputs, and extend them until we reach a total of 16 bits. (for example, "*1111 0001*" becomes "*1111 1111 1111 0001*"). The sign extension entity is quite simple, it is simply assigning the input bit to the output bit. However, we will see how this will be used later in the design when we begin port mapping everything together.

```
-- SIGN EXTENSION
library ieee;
use ieee.std_logic_1164.all;

entity ext is
   port( a, b         : in std_logic;
         newA, newB : out std_logic);
end ext;

architecture ARCH of ext is
begin
   newA <= a;
   newB <= b;
end ARCH;
```

Component: Seven Segment Hex Converter

The result of our operation is to be displayed through the seven segment display window. In order to do this, we must decode, or convert, the output of our circuit. Originally, our output is a string of length 16. However, we can subdivide this into 4 sections, each being 4 bits long. Each section represents one hexadecimal number. Each hexadecimal number will be displayed as a figure of seven segments. The truth table is shown below:

LETTER	INPUT $x_3\ x_2\ x_1\ x_0$	OUTPUT $Y_6\ Y_5\ Y_4\ Y_3\ Y_2\ Y_1\ Y_0$
0	0 0 0 0	1 0 0 0 0 0 0
1	0 0 0 1	1 1 1 1 0 0 1
2	0 0 1 0	0 1 0 0 1 0 0
3	0 0 1 1	0 1 1 0 0 0 0
4	0 1 0 0	0 0 1 1 0 0 1
5	0 1 0 1	0 0 1 0 0 1 0
6	0 1 1 0	0 0 0 0 0 1 0
7	0 1 1 1	1 1 1 1 0 0 0
8	1 0 0 0	0 0 0 0 0 0 0
9	1 0 0 1	0 0 1 1 0 0 0
A	1 0 1 0	0 0 0 1 0 0 0
b	1 0 1 1	0 0 0 0 0 1 1
C	1 1 0 0	1 0 0 0 1 1 0
d	1 1 0 1	0 1 0 0 0 0 1
E	1 1 1 0	0 0 0 0 1 1 0
F	1 1 1 1	0 0 0 1 1 1 0



0 = ON
1 = OFF

The Seven Segment Hex Converter VHDL entity:

```
-- SEVEN SEGMENT HEX CONVERTER
library ieee;
use ieee.std_logic_1164.all;

entity convert is
   port( X          : in std_logic_vector(3 downto 0);
         Y          : out std_logic_vector(6 downto 0));
end convert;

architecture ARCH of convert is
begin
  process(X)         -- 0 = on, 1 = off
    begin
      case X is
        when "0000" => Y <= "1000000";      -- 0
        when "0001" => Y <= "1111001";      -- 1
        when "0010" => Y <= "0100100";      -- 2
        when "0011" => Y <= "0110000";      -- 3
        when "0100" => Y <= "0011001";      -- 4
        when "0101" => Y <= "0010010";      -- 5
        when "0110" => Y <= "0000010";      -- 6
        when "0111" => Y <= "1111000";      -- 7
        when "1000" => Y <= "0000000";      -- 8
        when "1001" => Y <= "0011000";      -- 9
        when "1010" => Y <= "0001000";      -- A
        when "1011" => Y <= "0000011";      -- B
        when "1100" => Y <= "1000110";      -- C
        when "1101" => Y <= "0100001";      -- D
        when "1110" => Y <= "0000110";      -- E
        when "1111" => Y <= "0001110";      -- F
      end case;
  end process;

end ARCH;
```

Component: 16-Bit Adder/Subtractor

Now that we have defined all of our entities, we can define our 16-bit adder/subtractor entity, and include all of the other entities as components to this larger circuit.

```
-- 16-bit Adder/Subtractor
library ieee;
use ieee.std_logic_1164.all;

entity addsub16 is
   port( mode                 : in std_logic;
      A                       : in std_logic_vector(7 downto 0);
      B                       : in std_logic_vector(7 downto 0);
      S                       : buffer std_logic_vector(15 downto 0);
      cout, overflow          : out std_logic;
      HEX0, HEX1, HEX2, HEX3 : out std_logic_vector(6 downto 0));
end addsub16;
```

Our mode input is one bit, and will determine which operation will be performed (0 = addition, 1 = subtraction). A and B will be our 8-bit input strings, and S will be the sum of A and B, which will be 16-bits in length. S is of type buffer because we will use it as both input and output later when we display S through the seven segment display. Cout and Overflow are output bits which will represent whether or not our result had carry out or had overflow detected, respectively (0 = false, 1 = true). (Overflow will be dealt with later, during the port mapping). Finally, our HEX outputs will be each hexadecimal number, each being 7-bits, representing the seven segments of each display.

```
   component xorGate is              --XOR component
       port( a, b : in std_logic;
                 f : out std_logic);
   end component;

   component full_adder is           --FULL ADDER component
      port( x, y, cin : in std_logic;
            sum, cout : out std_logic);
   end component;

   component ext is                  -- SIGN EXTENSION component
       port( a, b         : in std_logic;
         newA, newB     : out std_logic);
   end component;

   component convert is              -- HEX_CONVERTER component
       port(X : in std_logic_vector(3 downto 0);
            Y : out std_logic_vector(6 downto 0));
   end component;
```

Here, we are including all of the previously defined entities as components to our 16-bit adder/subtractor.

Port Mapping

Before we beginning port mapping our components, we must define a few signals first.

```
signal C :std_logic_vector(15 downto 0);     -- intermediate carries
signal X : std_logic_vector(15 downto 0);    -- xor outputs
signal newA, newB : std_logic_vector(15 downto 0);       -- extended bits
```

Signal C will represent the intermediate carries between our 16 full bit adders. Signal X will represent the output of each XOR gate belonging to each of our 16 full bit adders. Finally, signals newA and newB will represent the inputs A and B after they have been extended as a result of our sign extension component.

```
begin
   extend0: ext port map(A(0), B(0), newA(0), newB(0));
   extend1: ext port map(A(1), B(1), newA(1), newB(1));
   extend2: ext port map(A(2), B(2), newA(2), newB(2));
   extend3: ext port map(A(3), B(3), newA(3), newB(3));
   extend4: ext port map(A(4), B(4), newA(4), newB(4));
   extend5: ext port map(A(5), B(5), newA(5), newB(5));
   extend6: ext port map(A(6), B(6), newA(6), newB(6));
   extend7: ext port map(A(7), B(7), newA(7), newB(7));
   extend8: ext port map(A(7), B(7), newA(8), newB(8));
   extend9: ext port map(A(7), B(7), newA(9), newB(9));
   extend10: ext port map(A(7), B(7), newA(10), newB(10));
   extend11: ext port map(A(7), B(7), newA(11), newB(11));
   extend12: ext port map(A(7), B(7), newA(12), newB(12));
   extend13: ext port map(A(7), B(7), newA(13), newB(13));
   extend14: ext port map(A(7), B(7), newA(14), newB(14));
   extend15: ext port map(A(7), B(7), newA(15), newB(15));
```

Here, we are connecting the sign extension components -- passing in the first 8 bits of A and B unaltered, but the last bit (most significant bit) of A and B are passed multiple times for the remaining 8 most significant bits of the newA and newB strings. This is how we would achieve the sign extension.

```
GX0: xorGate port map(mode, newB(0), X(0));
GX1: xorGate port map(mode, newB(1), X(1));
GX2: xorGate port map(mode, newB(2), X(2));
GX3: xorGate port map(mode, newB(3), X(3));
GX4: xorGate port map(mode, newB(4), X(4));
GX5: xorGate port map(mode, newB(5), X(5));
GX6: xorGate port map(mode, newB(6), X(6));
GX7: xorGate port map(mode, newB(7), X(7));
GX8: xorGate port map(mode, newB(8), X(8));
GX9: xorGate port map(mode, newB(9), X(9));
GX10: xorGate port map(mode, newB(10), X(10));
GX11: xorGate port map(mode, newB(11), X(11));
GX12: xorGate port map(mode, newB(12), X(12));
GX13: xorGate port map(mode, newB(13), X(13));
GX14: xorGate port map(mode, newB(14), X(14));
GX15: xorGate port map(mode, newB(15), X(15));
```

This is where we connect the XOR gates, thus allowing us to control which operation we would perform. This is affecting the input of newB. If mode is 1, we negate the bit of newB, otherwise we leave it alone. This new result becomes the value of our signal X.

```
FA0: full_adder port map(newA(0), X(0), mode,  S(0), C(0));
FA1: full_adder port map(newA(1), X(1), C(0),  S(1), C(1));
FA2: full_adder port map(newA(2), X(2), C(1),  S(2), C(2));
FA3: full_adder port map(newA(3), X(3), C(2),  S(3), C(3));
FA4: full_adder port map(newA(4), X(4), C(3),  S(4), C(4));
FA5: full_adder port map(newA(5), X(5), C(4),  S(5), C(5));
FA6: full_adder port map(newA(6), X(6), C(5),  S(6), C(6));
FA7: full_adder port map(newA(7), X(7), C(6),  S(7), C(7));
FA8: full_adder port map(newA(8), X(8), C(7),  S(8), C(8));
FA9: full_adder port map(newA(9), X(9), C(8),  S(9), C(9));
FA10: full_adder port map(newA(10), X(10), C(9),  S(10), C(10));
FA11: full_adder port map(newA(11), X(11), C(10),  S(11), C(11));
FA12: full_adder port map(newA(12), X(12), C(11),  S(12), C(12));
FA13: full_adder port map(newA(13), X(13), C(12),  S(13), C(13));
FA14: full_adder port map(newA(14), X(14), C(13),  S(14), C(14));
FA15: full_adder port map(newA(15), X(15), C(14),  S(15), C(15));
```

Now we can send these new values through our full bit adders. We can see mode is one of the inputs of our first adder, and C(0) is the carry out of our first adder. We can then use C(0) as the carry in input of our next adder, and so on. These adders are now connected in sequence, meaning the output of one adder affects the input of the successive adder.

```
DISPLAY0: convert port map(S(3)&S(2)&S(1)&S(0), HEX0);
DISPLAY1: convert port map(S(7)&S(6)&S(5)&S(4), HEX1);
DISPLAY2: convert port map(S(11)&S(10)&S(9)&S(8), HEX2);
DISPLAY3: convert port map(S(15)&S(14)&S(13)&S(12), HEX3);
```

Finally, we can use our converter to read our sum, and create the display of each hexadecimal number.

Overflow Detection

In addition to all of this functionality, we still want our design to be able to detect the overflow cases.

Overflow is defined as a condition that occurs when the result of an operation is greater in value than the given register it must be stored in. In our example, there are two cases when overflow can occur. One, when we add more than the largest positive 16-bit signed integer, or two, when we subtract more than the most negative 16-bit signed integer. Now how can we detect this in our hardware design?

We must look at the two most significant carry out bits of our circuit, *i.e.* the carry in and carry out of the last full adder of our circuit. Let the carry out of the full adder adding the least significant bit be called C(0), then the carry out of the full adder adding the two most significant bits are C(14) and C(15). On our circuit, we define overflow to be the result of the following equation:

Overflow = C(14) XOR C(15)

The XOR of the C(14) and C(15) differ if there's either a 1 being carried in and a 0 being carried out, or if there's a 0 being carried in and a 1 being carried out.

Case 1: If a 1 is carried in and a 0 is carried out, then A(15) = 0 and B(15) = 0, and the sum is 1. This is the case when two non-negative numbers are added, and yet a negative number becomes the result.

Case 2: If a 0 is carried in and a 1 is carried out, then A(15) = 1 and B(15) = 1, and the sum is 0. This is the case when two negative numbers are added, and yet a non-negative number becomes the result.

In either case, the result is incorrect, and overflow returns a 1, indicating overflow has been detected.

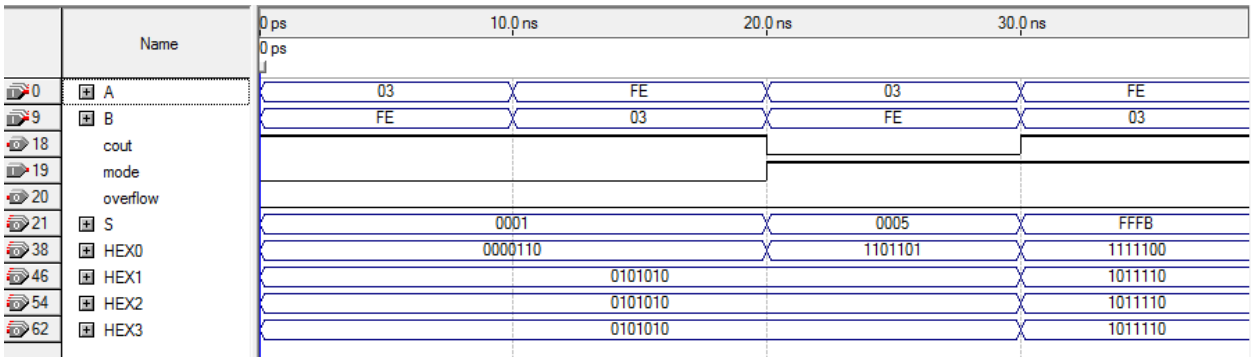
We can achieve this with the following piece of code:

```
overflow_out: xorGate port map(C(14), C(15), overflow);
cout <= C(15);
```

Simulation

Vector Waveform Simulation

In order to test the 16-bit adder/subtractor, we created a vector waveform file (.vwf) for the circuit. Then we ran this simulation, and verified the results.



Looking at the simulation, the following cases were tested (values are represented in hexadecimal notation -- with the exception of the HEX outputs, which are in binary so that we can see the value of each of the seven segments):

Case 1: *Hex:* *03 + FE = 0001*

Decimal: *3 + (-2) = 1*

Cout = 1 *Overflow = 0*

Case 2: *Hex:* *FE + 03 = 0001*

Decimal: *(-2) + 3 = 1*

Cout = 1 *Overflow = 0*

Case 3: *Hex:* *03 - FE = 0005*

Decimal: *3 - (-2) = 5*

Cout = 0 *Overflow = 0*

Case 4: *Hex:* *FE - 03 = FFFB*

Decimal: *(-2) - 3 = -5*

Cout = 1 *Overflow = 0*

DE2 Circuit Board Test

Before connecting to the DE2 board, we must first assign the correct pins to each component of the circuit. The inputs (A, B, mode) will be assigned to the board's toggle switches, and the outputs (S, cout, overflow) will be assigned to the red LED lights. The HEX outputs will be assigned to the Seven Segment Digits.

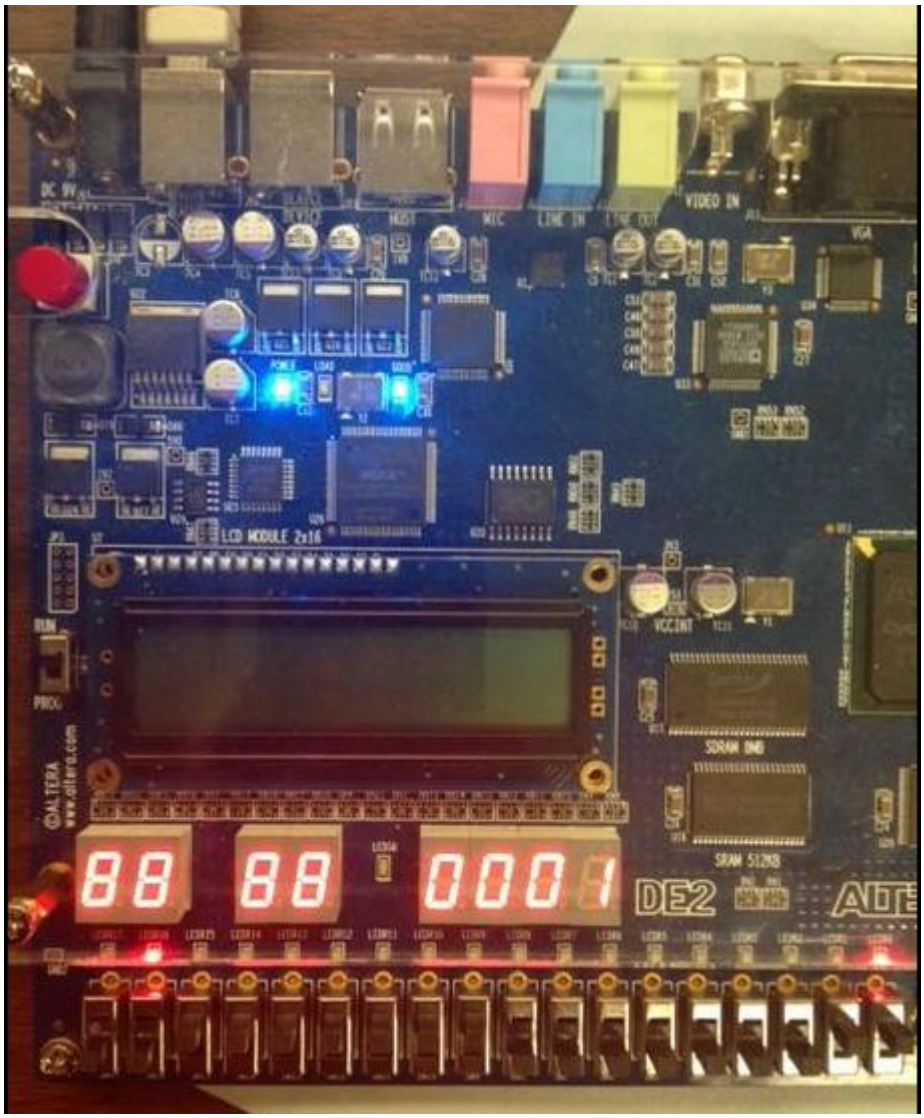
Pin assignments text file:

```
to, location

A[0], PIN_N25
A[1], PIN_N26
A[2], PIN_P25
A[3], PIN_AE14
A[4], PIN_AF14
A[5], PIN_AD13
A[6], PIN_AC13
A[7], PIN_C13

B[0], PIN_B13
B[1], PIN_A13
B[2], PIN_N1
B[3], PIN_P1
B[4], PIN_P2
B[5], PIN_T7
B[6], PIN_U3
B[7], PIN_U4

mode, PIN_V2

S[0], PIN_AE23
S[1], PIN_AF23
S[2], PIN_AB21
S[3], PIN_AC22
S[4], PIN_AD22
S[5], PIN_AD23
S[6], PIN_AD21
S[7], PIN_AC21
S[8], PIN_AA14
S[9], PIN_Y13
S[10], PIN_AA13
S[11], PIN_AC14
```

```
35  S[12], PIN_AD15
36  S[13], PIN_AE15
37  S[14], PIN_AF13
38  S[15], PIN_AE13
39
40  cout, PIN_AE12
41  overflow, PIN_AD12
42
43  HEX0(0), PIN_AF10
44  HEX0(1), PIN_AB12
45  HEX0(2), PIN_AC12
46  HEX0(3), PIN_AD11
47  HEX0(4), PIN_AE11
48  HEX0(5), PIN_V14
49  HEX0(6), PIN_V13
50
51  HEX1(0), PIN_V20
52  HEX1(1), PIN_V21
53  HEX1(2), PIN_W21
54  HEX1(3), PIN_Y22
55  HEX1(4), PIN_AA24
56  HEX1(5), PIN_AA23
57  HEX1(6), PIN_AB24
58
59  HEX2(0), PIN_AB23
60  HEX2(1), PIN_V22
61  HEX2(2), PIN_AC25
62  HEX2(3), PIN_AC26
63  HEX2(4), PIN_AB26
64  HEX2(5), PIN_AB25
65  HEX2(6), PIN_Y24
66
67  HEX3(0), PIN_Y23
68  HEX3(1), PIN_AA25
69  HEX3(2), PIN_AA26
70  HEX3(3), PIN_Y26
71  HEX3(4), PIN_Y25
72  HEX3(5), PIN_U22
73  HEX3(6), PIN_W24
```

Now we can begin board testing:

Test1:

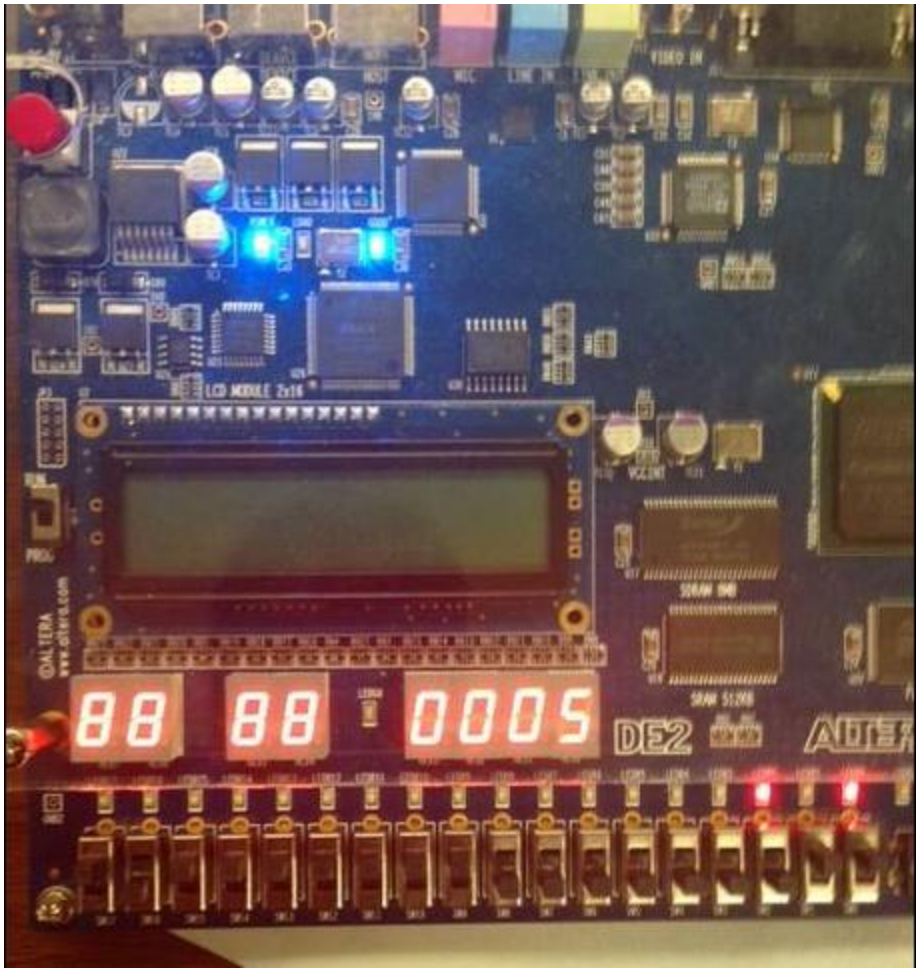


Hex: *03 + FE = 0001*

Decimal: *3 + (-2) = 1*

Cout = 1 *Overflow = 0*

Test2:

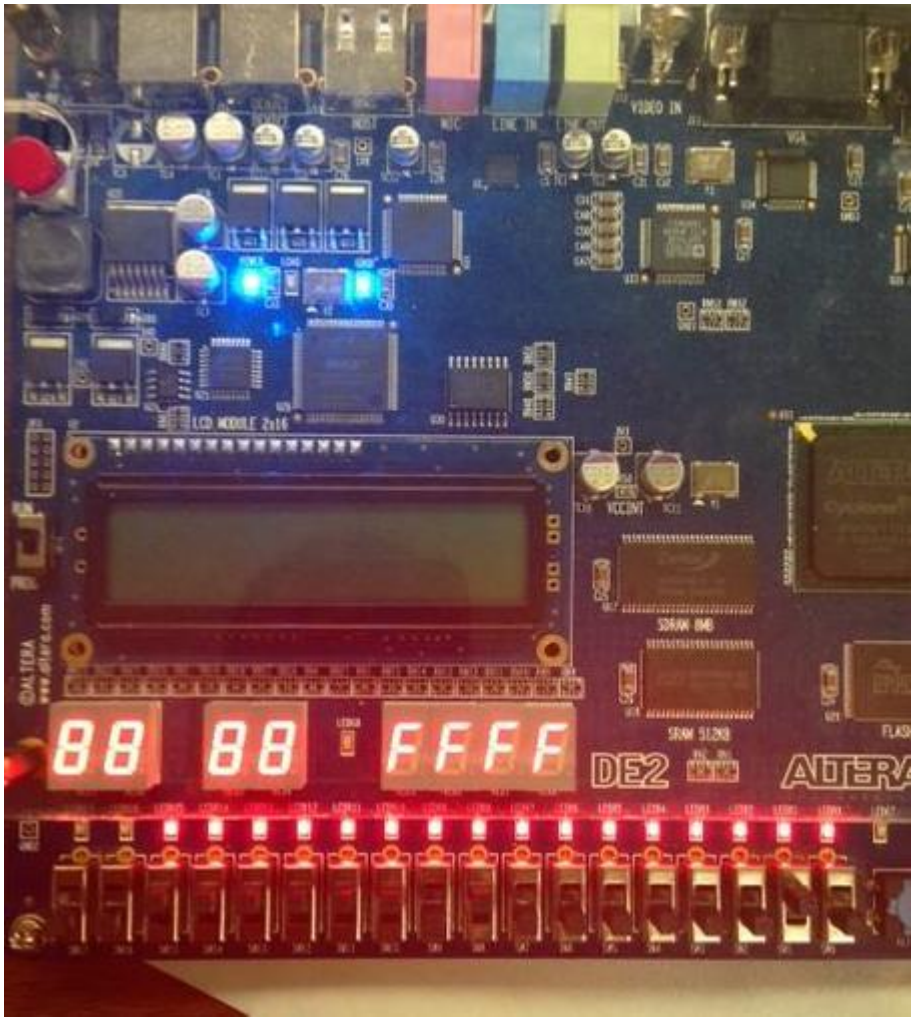


Hex: *03 - FE = 0005*

Decimal: *3 - (-2) = 5*

Cout = 0 *Overflow = 0*

Test3:



Hex: *02 - 03 = FFFF*

Decimal: *2 - 3 = -1*

Cout = 0 *Overflow = 0*

Conclusion:

Beginning with a basic component, we continued to extend the functionality of our circuit. The full adder is very important in digital circuits because of how common it is used. Our lab connected multiple full adders to create adders of larger magnitudes, able to add strings of multiple bits. Then, we took this concept, and allowed our circuit to not only add, but subtract as well. This was essentially achieved by adding A to the negative of B. We also enabled our circuit to detect overflow occurrences by observing the carry ins and carry outs of the most significant full adder.

We then further modified our design by extending the 8th most significant bit over a 16-bit string in order simulate a 16-bit adder with only 8-bit string inputs on the DE2 board. Finally, we enabled our design to display the sum of our circuit in hexadecimal onto the seven segment display window.