Lab Report #10

OBJECTIVE: The purpose of this lab was to construct a circuit which acted as a digital calculator. More specifically, the assignment was to create a circuit which was able to take two 8-bit numbers as input and have one of the basic arithmetic operations (addition, subtraction, division, multiplication) act upon those two numbers. The requirements stated that the output must be displayed on 7-segment displays located on the Altera FPGA board.

DESIGN: The general procedure for constructing the circuit is given by the following steps:

1. ***Black Box –*** The very first thing our group did was to draw a black box of the calculator. This showed simply the inputs and outputs from the board level. This is what gave us a basis to begin brainstorming the inside or the "brains" of the calculator. ***(Fig e-1)***

2. ***Brainstorming and the "Data Flow Model" –*** This part of the project involved conceptualizing the process of turning input into output. It was realized that defining this was crucial at the beginning because it would define how the block diagram would turn out.

   For example, we decided the way we wanted to calculator to perform was to calculate all arithmetic operations simultaneously, and then the user selected which information to display. Our block diagram may have turned out differently if we decided we wanted each operation selected first, that way only one operations output is ever being calculated at a time. This is what we mean when we say "Data Flow Model". ***(Fig e-2)***

3. ***Block Diagram –*** Once the "Data Flow Model" was tested and confirmed to work for one arithmetic operation (addition), the same procedure was carried out on the remaining three operations as well as creating other modules need to complete the design. ***(Fig e-3)***

4. ***Adder / Subtractor / Multiplier / Divider Module's –*** Each of these modules was implemented by using the built in arithmetic operators (+ , - , * , /) available in Quartus. ***(Fig e-4)***

5. ***Decoder Module's*** – Each of the decoders operated on the same similar principle with only minor specifics being altered for each arithmetic operation. Examples of differences in code between modules were the number of bits accepted for the input, and, the output configuration for each Hex display. ***(Fig e-5)*** ***Note* -- Because the code is so lengthy, to save space, only the adder decoder is shown in the report, but all the decoders operate on the same principle.***

6. ***Encoder Module*** – This module takes in input from the user through the use of the push keys on the board and translated it into information used by the multiplexers to select which operations output to push through to the 7-segment displays. The encoder operates on the same principle as the decoder, it just does it in the reverse fashion. ***(Fig e-6)***

7. ***Multiplexer Module***'s – These modules were all identical. They were used to accept information from the decoders, and also to pass information on the HEX displays once a selection was made by the user through the use of the push keys. ***(Fig e-7)***

8. ***Operation Selector Module*** – This is the module which houses all the "component modules" of the calculator, and is the module where all the "component modules" are wire to one another in order to create the functionality of a calculator. The Operation Selector module can be thought of as the digital calculator existing "digitally" in code, but not yet able to be implement on hardware. For hardware implementation, we needed to next connect the "digital" version of the calculator to the Altera board. For this reason, the Operation Selector module is the layer right below the pin assignments layer, which is the next module we will discuss. ***(Fig e-8)***

9. ***Pin Assignment Module –*** This is the top layer of code, its purpose is to wire (connect) the inputs and outputs of the Altera board, to the inputs and outputs of the operation select module. ***(Fig e-9)***

10. ***Pin Assignment Functional Simulation –*** This functional simulation shows that all functionalities of the calculator are working for all the different combinations of inputs and corresponding outputs. ***(Fig e-10)***

HARDWARE:

- Altera Quartus II – Programmable Logic Device Software
- Altera DE2-115 – FPGA Board

EXPERIMENT:

Brainstorming and the "Data Flow Model"

The very first thing our group did when presented with this project was to begin brainstorming and conceptualizing the project at a very "black box" level. ***(Fig e-1)*** This means to begin first by thinking in terms simply the inputs and outputs of calculator according the specification given by the lab sheet.

After creating the black box diagram, it was soon realized that the main challenge this project presented was the requirement that the HEX displays had to display different types of information at different time's. We had never been asked to do anything like this on any of our previous labs. After thinking about to tackle that particular problem, it was realized that something would have to "decide" or "select" which input would go to the HEX displays. It was then decided that multiplexers would be used to make this selection.

The next step was to decide exactly how many mux's there would be, and their position within the calculator. Several configurations were considered, with one being started on, but then discarded due to complications that were realized while building. Finally, what we are calling a "Data Flow Model" was created. ***(Fig e-2)***

A Data Flow Model, as we are defining it, means a specific way of handling the flow of information through the circuit. This model is then consistently implemented in the same fashion, across all

similar modules, and all similar groupings of modules. We thought it was very important to define this early on because we felt that the way data flows within the calculator is what essentially describes the requirements of which modules you would end up needing to implement. Following this logic, if one was able to figure out a "cookie cutter" method that handled data in the same way for all the operations, it would be very easy to then instantiate the same modules for each operation (addition, subtraction, multiplication, division).

With that all in mind, we decided to try and implement a method in which all operations were simply performed at the same time, and the user simply selected which operations output they would like to display. We were able to have this happen by using four 4-input Mux's (one to each hex display) to display the output of the circuit. Every time a user made a selection of an operation, let's say addition, each one of the Mux's would only output the information that was related to addition. In this way, the HEX displays only ever received information which was correlated with the KEY the user pressed.

The end result of the brainstorming and data flow model process led us to a top level block diagram ***(Fig e-3)*** of the calculator which included all the main modules which would enable the calculator to function. It should be noted that at this level, the inner workings of specific modules such as decoders, encoders, mux's or operations such as adding, dividing etc., were not realized yet.

Testing the Data Flow Model

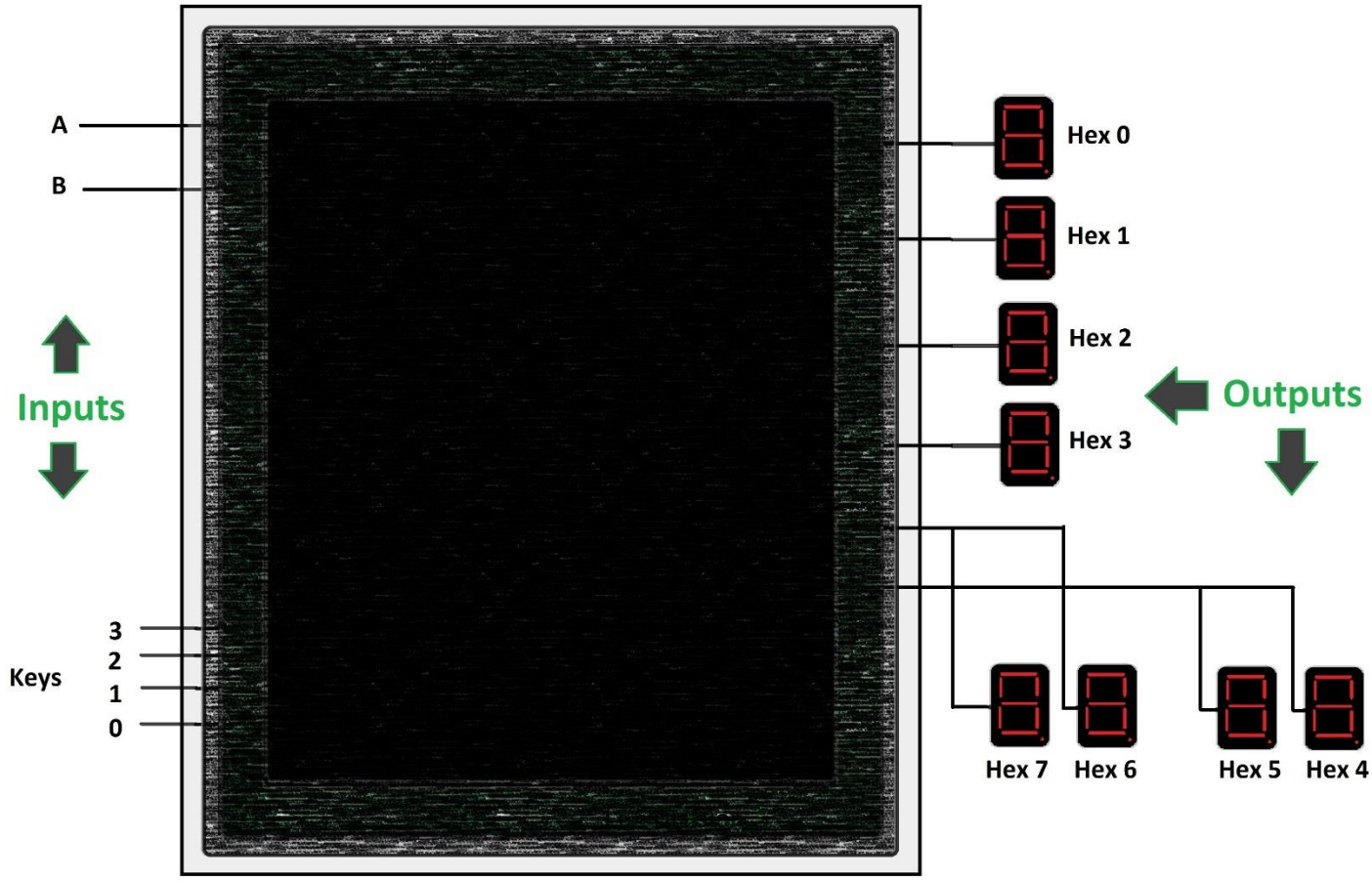
Once the Data Flow Model ***(Fig e-2)*** was decided on, and the block diagram was created, one full "flow" of data from input to output was tested in Verilog. This was done by creating only the adder module, its decoder, and the four MUX's. The purpose of doing this was to test if one decoders data would be able to be selected and then output appropriately across each MUX.

Verilog -- Board Testing -- Troubleshooting

Once the Data Flow Model was confirmed to work, the rest of the components had their modules coded in Verilog and then finally tested by a functional simulation. ***(Fig e-10)***

The entire process went quite smoothly with the only significant issue being that we forget that the Altera board is Active Low. This caused us to troubleshoot as to why the pushbutton keys were active the entire time even when we were not pressing them.

Fig e-1
Black Box



This illustration shows the initial Black Box representation of the calculator with only the external inputs and outputs visible.

Fig e-2
Brainstorming and the "Data Flow Model"

Data Flow Model

This illustration demonstrates the basis of operation of the calculator.

This test was done to ensure that when the user made a selction the multiplexers would output the information the user selected.

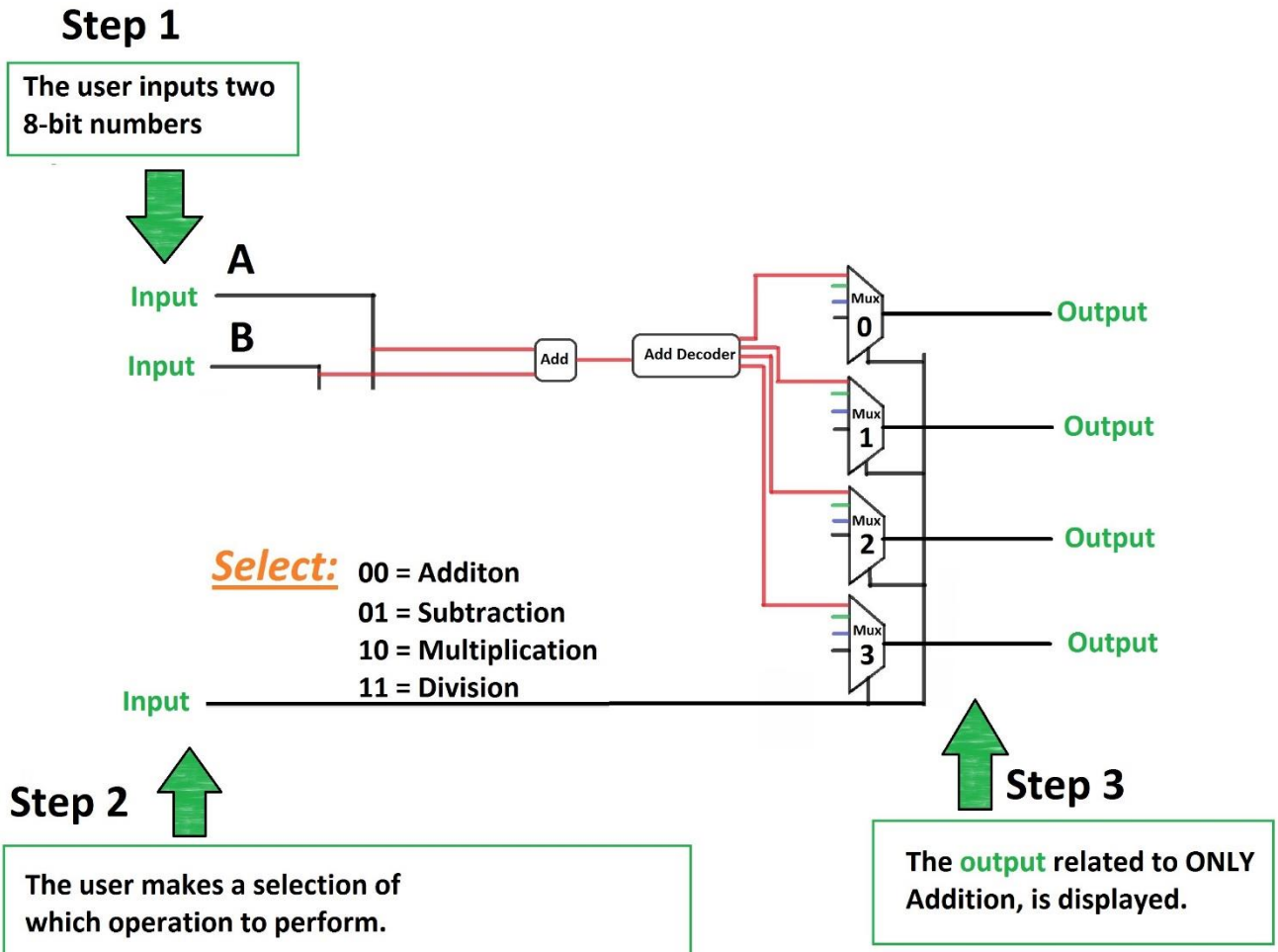
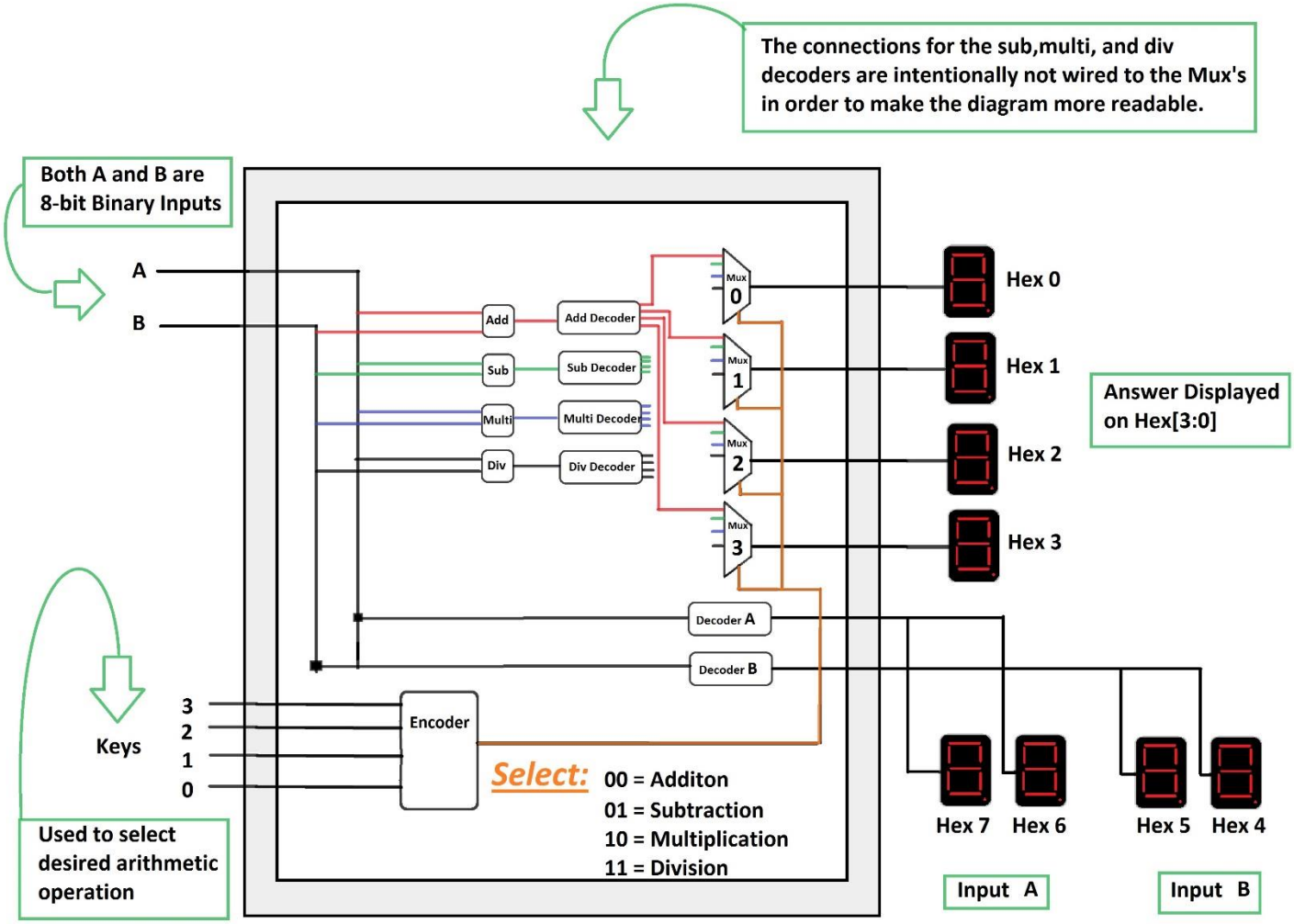


Fig e-3
Block Diagram



This illustration shows the fully developed block diagram for the entire calculator after the data flow model was tested.

Fig e-4
Adder / Subtractor / Multiplier / Divider -- Module's

```
1   module adder_simple (A,B,Sum);
2
3   input [7:0] A,B;
4   output [8:0] Sum;
5
6   assign Sum = A + B;
7
8   endmodule
9
```

```
1   module subtract_simple (A,B,difference);
2
3   input [7:0] A,B;
4   output reg[7:0] difference;
5
6   always
7      begin
8
9   if (A > B)
10        difference = A - B;
11
12  if (A < B)
13        difference = B - A;
14
15     end
16
17  endmodule
18
```

```
1   module multi_simple(A,B,product);
2
3   input [7:0]A,B;
4   output [15:0]product;
5
6   assign product = (A*B);
7
8   endmodule
9
```

```
1   module divider_simple(A,B,remainder,quotient);
2
3   input [7:0]A,B;
4   output [7:0]quotient,remainder;
5
6   //reg [15:0] out;
7   wire [7:0] quotient;
8
9   assign quotient = (A/B);
10  assign remainder = (((B*quotient) - A)*(-1));
11
12  endmodule
13
```

Fig e-5
Decoder Module's

```
module decoder_add(x,z,addhex0,addhex1,addhex2,addhex3);

input [8:0] x;
output [8:0] z;
assign z = x;

output [6:0]addhex0;
reg [6:0]addhex0;

output [6:0]addhex1;
reg [6:0]addhex1;

output [6:0]addhex2;
reg [6:0]addhex2;

output [6:0]addhex3;
reg [6:0]addhex3;

always
	begin
		case (x[3:0])
			4'b0000: addhex0 = 7'b1000000; // 0
			4'b0001: addhex0 = 7'b1111001; // 1
			4'b0010: addhex0 = 7'b0100100; // 2
			4'b0011: addhex0 = 7'b0110000; // 3
			4'b0100: addhex0 = 7'b0011001; // 4
			4'b0101: addhex0 = 7'b0010010; // 5
			4'b0110: addhex0 = 7'b0000010; // 6
			4'b0111: addhex0 = 7'b1111000; // 7
			4'b1000: addhex0 = 7'b0000000; // 8
			4'b1001: addhex0 = 7'b0011000; // 9
			4'b1010: addhex0 = 7'b0001000; // A
			4'b1011: addhex0 = 7'b0000011; // b
			4'b1100: addhex0 = 7'b1000110; // C
			4'b1101: addhex0 = 7'b0100001; // d
			4'b1110: addhex0 = 7'b0000110; // E
			4'b1111: addhex0 = 7'b0001110; // F
		endcase
	end

always
	begin
		case (x[7:4])
			4'b0000: addhex1 = 7'b1000000; // 0
			4'b0001: addhex1 = 7'b1111001; // 1
			4'b0010: addhex1 = 7'b0100100; // 2
			4'b0011: addhex1 = 7'b0110000; // 3
			4'b0100: addhex1 = 7'b0011001; // 4
			4'b0101: addhex1 = 7'b0010010; // 5
			4'b0110: addhex1 = 7'b0000010; // 6
			4'b0111: addhex1 = 7'b1111000; // 7
			4'b1000: addhex1 = 7'b0000000; // 8
			4'b1001: addhex1 = 7'b0011000; // 9
			4'b1010: addhex1 = 7'b0001000; // A
			4'b1011: addhex1 = 7'b0000011; // b
			4'b1100: addhex1 = 7'b1000110; // C
			4'b1101: addhex1 = 7'b0100001; // d
			4'b1110: addhex1 = 7'b0000110; // E
			4'b1111: addhex1 = 7'b0001110; // F
		endcase
	end

always
	begin
		case (x[8:8])
			4'b0000: addhex2 = 7'b1000000; // 0
			4'b0001: addhex2 = 7'b1111001; // 1
		endcase
	end
```

```
always
	begin
		case (x[3:0])
			4'b0000: addhex3 = 7'b1111111; // Blank
			4'b0001: addhex3 = 7'b1111111; // Blank
			4'b0010: addhex3 = 7'b1111111; // Blank
			4'b0011: addhex3 = 7'b1111111; // Blank
			4'b0100: addhex3 = 7'b1111111; // Blank
			4'b0101: addhex3 = 7'b1111111; // Blank
			4'b0110: addhex3 = 7'b1111111; // Blank
			4'b0111: addhex3 = 7'b1111111; // Blank
			4'b1000: addhex3 = 7'b1111111; // Blank
			4'b1001: addhex3 = 7'b1111111; // Blank
			4'b1010: addhex3 = 7'b1111111; // Blank
			4'b1011: addhex3 = 7'b1111111; // Blank
			4'b1100: addhex3 = 7'b1111111; // Blank
			4'b1101: addhex3 = 7'b1111111; // Blank
			4'b1110: addhex3 = 7'b1111111; // Blank
			4'b1111: addhex3 = 7'b1111111; // Blank
		endcase
	end

always
	begin
		case (x[7:4])
			4'b0000: addhex3 = 7'b1111111; // Blank
			4'b0001: addhex3 = 7'b1111111; // Blank
			4'b0010: addhex3 = 7'b1111111; // Blank
			4'b0011: addhex3 = 7'b1111111; // Blank
			4'b0100: addhex3 = 7'b1111111; // Blank
			4'b0101: addhex3 = 7'b1111111; // Blank
			4'b0110: addhex3 = 7'b1111111; // Blank
			4'b0111: addhex3 = 7'b1111111; // Blank
			4'b1000: addhex3 = 7'b1111111; // Blank
			4'b1001: addhex3 = 7'b1111111; // Blank
			4'b1010: addhex3 = 7'b1111111; // Blank
			4'b1011: addhex3 = 7'b1111111; // Blank
			4'b1100: addhex3 = 7'b1111111; // Blank
			4'b1101: addhex3 = 7'b1111111; // Blank
			4'b1110: addhex3 = 7'b1111111; // Blank
			4'b1111: addhex3 = 7'b1111111; // Blank
		endcase
	end
```

This illustration shows the Decoder Module for the Adder.

The decoder breaks the users 8-bit numbers into sections of 4 bit's.

Then, each of these 4 bit's are translated into a value to be displayed on a specific HEX.

In the case of the Adder, the maximum number generated from adding two 8-bit Binary numbers is 9-bits. When translated to hexidecimal, this value is 1FE. This only takes 3 displays to show that information so the third display is not needed and therefore blanked out.

This is done by sending 7b'1111111 to the HEX display, which causes all LED's to be turned off, since they are connect to the Altera Board, which is Active Low. Meaning a value of high (or 1) is equivalent to OFF.

Fig e-6
Encoder Module

```
1   module encoder_keys(KEY,choice_made);
2
3   input [3:0]KEY;
4   output [1:0]choice_made;
5   reg [1:0]choice_made;
6
7   always
8       begin
9           casex (KEY[3:0])
10              4'b1111: choice_made = 2'b00;
11              4'b1110: choice_made = 2'b00; // 0
12              4'b1101: choice_made = 2'b01; // 1
13              4'b1011: choice_made = 2'b10; // 2
14              4'b0111: choice_made = 2'b11; // 3
15          endcase
16      end
17
18
19  endmodule
20  //-------------------------------------------------------------------------------------------------
21              //  Important to remember :::: The Altera Board is ACTIVE LOW ::::
22              //  Meaning 4'b1111 = A value of ---- all Keys are OFF --------- is sent through
23              //  Meaning 4'b1110 = A value of ---- first key (or 0th key) --- is sent through
24  //-------------------------------------------------------------------------------------------------
25
```

The illustration above shows the Encoder Module.
It has the same functionality as the Decoder except in the reverse fashion whereby it takes information from the board and translates it into information to be used within the module.

Here, each KEY corresponds to a selection of an operation.

00 = Addition
01 = Subtraction
10 = Multiplication
11 = Division

Fig e-7
Multiplexer Module's

```
1   module MUX_FOUR_BY_ONE(select,b0,b1,b2,b3,out);
2
3   input [15:0] b0,b1,b2,b3; //Four inputs that accept 8 bit information
4   input [1:0]select;       // The select input is 2 bits
5
6   output reg [6:0] out; // One of the inputs is selected and
7                         // pushed to the output, so the output
8                         // must also be 8 bits
9
10  always
11    begin
12      case(select)
13
14        2'b00: out = b0;
15        2'b01: out = b1;
16        2'b10: out = b2;
17        2'b11: out = b3;
18
19      endcase
20    end
21  // The case statement is always executed
22  //  it causes the following desicion to take place
23
24  //                              if select = 00
25  //                                 output = b0
26  //                              if select = 01
27  //                                 output = b1  .. and so on
28
29  endmodule
30
31
32  // start   6:50 PM 5/3/2016
33  // end     5:52 PM 5/4/2016
```

Fig e-8
Operation Selector Module

```
1   module operation_selector(A,B,KEY,mux0,mux1,mux2,mux3,decAhex7,decAhex6,decBhex5,decBhex4);
2
3   input [7:0]A,B;
4   input [3:0]KEY;
5   output[6:0]mux0,mux1,mux2,mux3;
6   output[6:0]decAhex7,decAhex6,decBhex5,decBhex4;
7
8
9   //START---------------------------------------------- Decoders for Incoming Numbers A and B --
10  //          Group_A_Decoder
11      |   |   decoder_A (A,decAhex7,decAhex6);
12  //          decoder_A (A,decAhex7,decAhex6)
13      |   |   wire [6:0]decAhex7,decAhex6;
14      |   |
15  //          Group_B_Decoder
16      |   |   decoder_B (B,decBhex5,decBhex4);
17  //          decoder_B (B,decBhex5,decBhex4)
18      |   |   wire [6:0]decBhex5,decBhex4;
19  //END------------------------------------------------ Decoders for Incoming Numbers A and B ----
20
21
22
23  //START---------------------------------------------- Encoder for Incoming Keys 3,2,1,0 ------
24  //          Push_Button_Encoder
25      |   |   encoder_keys(KEY,choice_made);
26  //          encoder_keys(KEY,choice_made);
27      |   |   wire [1:0]choice_made;
28  //END------------------------------------------------ Encoder for Incoming Keys 3,2,1,0 --------
29
30
31
32
```

The Operation Selector Module is where all the individual modules which make up the calculator are "housed". It is the "code" representation of the interior graphics of Fig-3.

This module can be considered the digital representation of the entire functioning calculator, but because there is not module which interacts with the board yet, it cannot be tested on the FPGA yet. That is what the Pin Assignment Module is for.

Note* -- Code is continued on the next page.

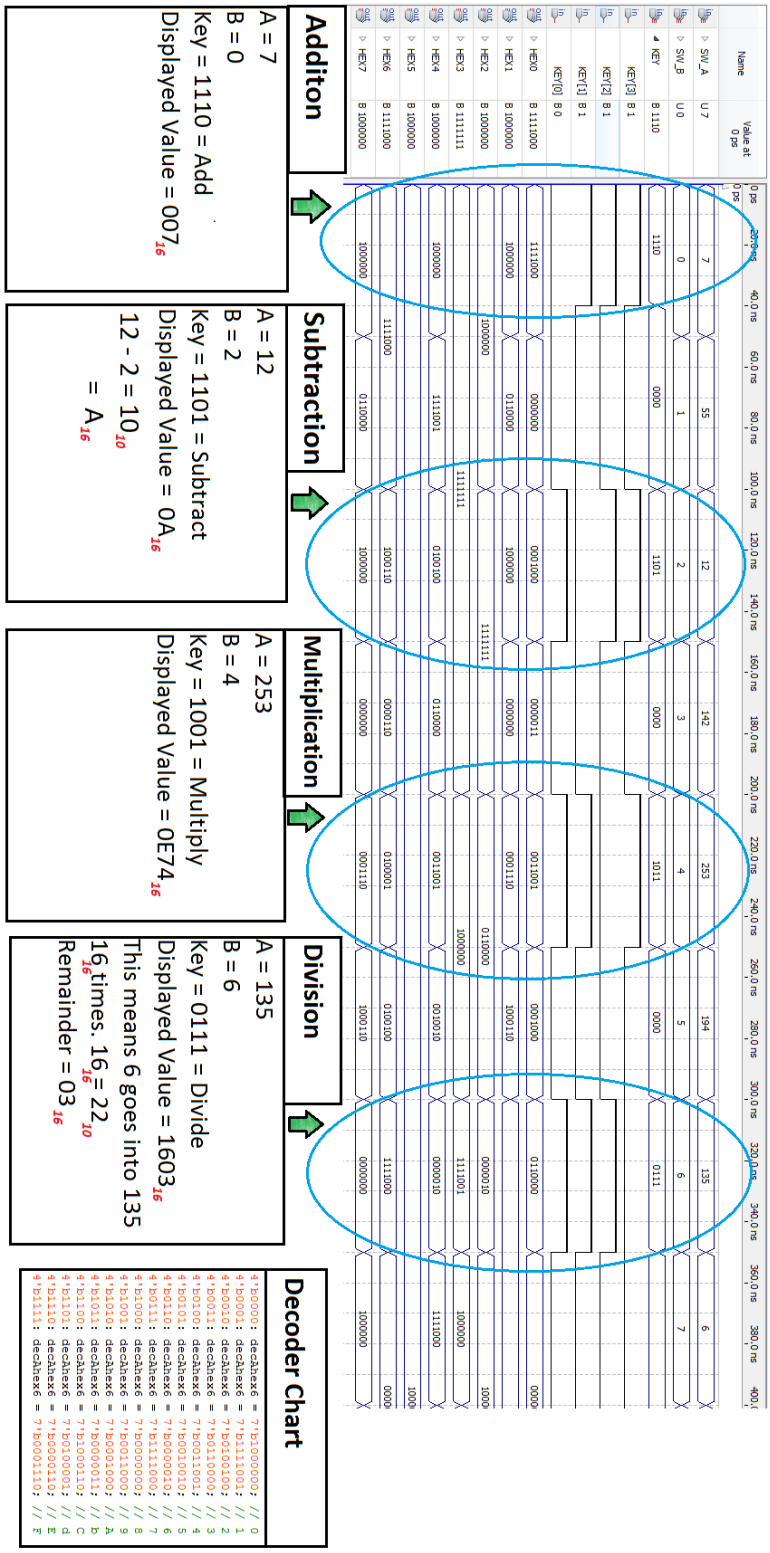
```
32
33  //START---------------------------------------------- Calls for Operations (add/sub/multi/div) Modules --
34  //------------------------------------------------------------- and thier associated Decoders ------------
35          //ADDER_Call
36                  adder_simple (A,B,Sum);
37                //adder_simple (A,B,Sum);
38                  wire [8:0]Sum;
39
40          //ADDER_Decoder
41                  decoder_add(Sum,z,addhex0,addhex1,addhex2,addhex3);
42                //decoder_add(x,z,addhex0,addhex1,addhex2,addhex3);
43                  wire [6:0]addhex0,addhex1,addhex2,addhex3;
44
45          //SUBTRACTOR_Call
46                  subtract_simple (A,B,difference);
47                //subtract_simple (A,B,difference);
48                  wire[7:0]difference;
49
50          //SUBTRACTOR_Decoder
51                  decoder_sub(A,B,difference,z,subhex0,subhex1,subhex2,subhex3);
52                //decoder_sub(A,B,x,z,subhex0,subhex1,subhex2,subhex3);
53                  wire [6:0]subhex0,subhex1,subhex2,subhex3;
54
55          //MULTIPLIER_Call
56                  multi_simple(A,B,product);
57                //multi_simple(A,B,product);
58                  wire[15:0]product;
59
60          //MULTIPLIER_Decoder
61                  decoder_multi(product,z,multihex0,multihex1,multihex2,multihex3);
62                //decoder_multi(x,z,multihex0,multihex1,multihex2,multihex3);
63                  wire [6:0]multihex0,multihex1,multihex2,multihex3;
64
65          //DIVIDER_Call
66                  divider_simple(A,B,remainder,quotient);
67                //divider_simple(A,B,remainder,quotient);
68                  wire[7:0]quotient,remainder;
69
70          //DIVIDER_Decoder
71                  decoder_div(quotient,remainder,divhex0,divhex1,divhex2,divhex3);
72                //decoder_div(quotient,remainder,divhex0,divhex1,divhex2,divhex3);
73                  wire [6:0]divhex0,divhex1,divhex2,divhex3;
74
75  //END------------------------------------------------ Calls for Operations (add/sub/multi/div) Modules --
76  //------------------------------------------------------------- and thier associated Decoders ------------
81    //START---------------------------------------------------------- Multiplexers:  ---------------------
82    //------------------------------------------------ Each of which output to one particular HEX Display -
83
84    //          MUX to hexZERO
85        MUX_FOUR_BY_ONE(choice_made,addhex0,subhex0,multihex0,divhex0,mux0);
86
87    //          MUX to hexONE
88        MUX_FOUR_BY_ONE(choice_made,addhex1,subhex1,multihex1,divhex1,mux1);
89
90    //          MUX to hexTWO
91        MUX_FOUR_BY_ONE(choice_made,addhex2,subhex2,multihex2,divhex2,mux2);
92
93    //          MUX to hexTHREE
94        MUX_FOUR_BY_ONE(choice_made,addhex3,subhex3,multihex3,divhex3,mux3);
95    //          MUX_FOUR_BY_ONE(select,b0,b1,b2,b3,out);
96
97    //END------------------------------------------------------------ Multiplexers:  ---------------------
98    //------------------------------------------------ Each of which output to one particular HEX Display -
99
100
101   endmodule
```

Fig e-9
Pin Assignment Module

```
1   module pin_assignments(SW,KEY,HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0,LEDG,LEDR);
2
3   input [15:0]SW;
4   input [3:0]KEY;
5   output[6:0]HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0;
6   output[6:0]LEDG;
7   output[15:0]LEDR;
8
9   assign LEDR[15:0] = SW[15:0];
10  assign LEDG[0] = !KEY[0];
11  assign LEDG[2] = !KEY[1];
12  assign LEDG[4] = !KEY[2];
13  assign LEDG[6] = !KEY[3];
14
15
16
17
18     operation_selector(SW[15:8],SW[7:0],KEY[3:0],HEX0,HEX1,HEX2,HEX3,HEX7,    HEX6,    HEX5,    HEX4);
19  // operation_selector(A,       B,      KEY,     mux0,mux1,mux2,mux3,decAhex7,decAhex6,decBhex5,decBhex4);
20     wire [15:0]SW;
21     wire[3:0]KEY;
22     wire[6:0]HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0;
23
24  endmodule
25
```

This is the top layer of code, its purpose is to wire (connect) the inputs and outputs of the Altera board, to the inputs and outputs of the operation select module.

Fig e-10
Pin Assignment Functional Simulation



POST LAB ANALYSIS:

Comment on the structure of your Verilog implementation. Check "Design Units" under Project Navigator to find all the units that are used in your design. What is each unit and what does it do?

MUX_FOUR_BY_ONE (Verilog HDL entity) – Module which implements a 4x1 Multiplexer

adder_simple (Verilog HDL entity) – Module which implements an adder

subtract_simple (Verilog HDL entity) – Module which implements the subtractor

divider_simple (Verilog HDL entity) – Module which implements the divider

multi_simple (Verilog HDL entity) – Module which implements the multiplier

decoder_A (Verilog HDL entity) – Module which decodes the group A numbers into Hex values

decoder_B (Verilog HDL entity) – Module which decodes the group B numbers into Hex values

decoder_add (Verilog HDL entity) – Module which decodes the addition answer into Hex values

decoder_div (Verilog HDL entity) – Module which decodes the division answer into Hex values

decoder_multi (Verilog HDL entity) – Module which decodes the multiplication answer into Hex values

decoder_sub (Verilog HDL entity) – Module which decodes the subtraction answer into Hex values

encoder_keys (Verilog HDL entity) – Module which encodes the push keys into 2 bit values for the MUX's select inputs.

multi_simple (Verilog HDL entity) – Module which implements the multiplier

operation_selector (Verilog HDL entity) – Module which "houses" all the component modules of the calculator

pin_assignments (Verilog HDL entity) – Module which connects the pins of the board to the inputs and outputs of the operation selector (digital calculator).

CONCLUSIONS:

As a group we feel that this lab challenged us to bring together every aspect of our knowledge we learned throughout the semester concerning building digital circuits and using Verilog. This project caused us to push our boundaries and come up with solutions for types of problems we had not encountered until now. Our main take away from this experience is the opportunity it granted us to enhance our understanding of how to approach building complex systems involving digital circuits.