EECS 31L Midterm Project: Enhanced Arithmetic Logic Unit Implementation in SystemVerilog

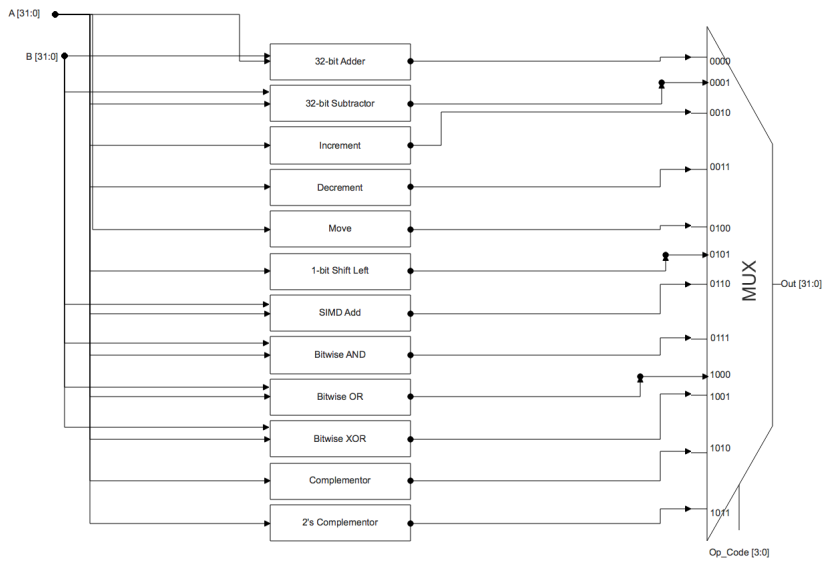
Group: Um
Kathy Nguyen (49131012), Josh Park (25729974), Aina Tancinco (66726318), Nicole Thai (55729939), Amy Yee (55246967)

30 October 2017

1 Introduction

Our Enhanced Arithmetic Logic Unit (ALU) consists of 12 operations that are selected through a multiplexer: addition, subtraction, increment, decrement, move, 1-bit logical shift left of A, SIMD Add (four 8-bits add), bitwise AND, bitwise OR, bitwise XOR, complement, and 2's complement. The multiplexer implements a conditional statement to properly determine which module to call on depending on the opcode selected. All 12 operations selected are passed to determine the proper call to its corresponding module implementation of the operation. This design and simulation of an Enhanced ALU will be connected to the FPGA board for analysis of the functions on the board.

1.1 32-Bit Enhanced ALU Block Diagram

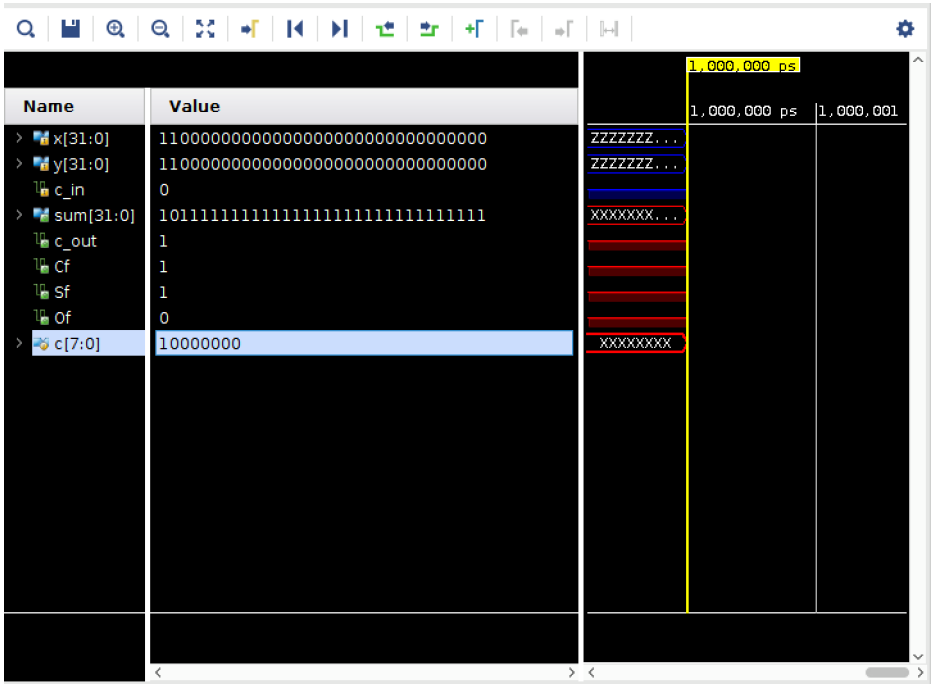


2 Process

Through effective collaboration, the Um group divided the task of implementation of an ALU into twelve modules among five members of the group. Individually the UM group struggled; however the UM group would collaborate on specific submodules that were a bit more challenging such as the addition and subtraction module when handling errors mentioned in more details below. Collectively, we were able to collaborate to increase efficiency and catch potential errors in the code. The multiplexer connects 12 submodules by selecting them based on the conditional statement that would call the specific submodule.

2.1 Addition

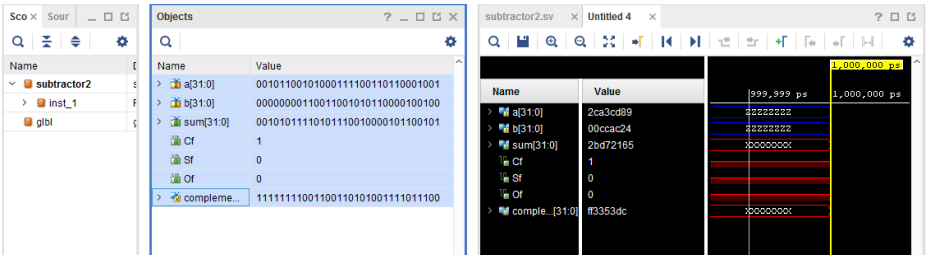
```
22
23  module FA_32bitv2(x, y, sum, c_out, Cf, Sf, Of);
24
25      input [31:0] x, y;
26      output [31:0] sum;
27      output c_out;
28      output Cf;
29      output Sf;
30      output Of;
31
32      logic c_in=0;
33      logic [7:0] c;
34
35
36      // 32 = 8 * 4
37      FA_4bit inst_1(x[3:0], y[3:0], c_in, sum[3:0], c[0]);
38      FA_4bit inst_2(x[7:4], y[7:4], c[0], sum[7:4], c[1]);
39      FA_4bit inst_3(x[11:8], y[11:8], c[1], sum[11:8], c[2]);
40      FA_4bit inst_4(x[15:12], y[15:12], c[2], sum[15:12], c[3]);
41      FA_4bit inst_5(x[19:16], y[19:16], c[3], sum[19:16], c[4]);
42      FA_4bit inst_6(x[23:20], y[23:20], c[4], sum[23:20], c[5]);
43      FA_4bit inst_7(x[27:24], y[27:24], c[5], sum[27:24], c[6]);
44      FA_4bit inst_8(x[31:28], y[31:28], c[6], sum[31:28], c[7]);
45      assign c_out = c[7];
46
47      // assign the flags
48      assign Cf = c_out;
49      assign Sf = sum[31];
50      assign Of = (x[31]==y[31]) ? ( (x[31] != sum[31]) ? 1 : 0 ) : 0;
51
52  endmodule
```



The Um group initially began typing out the bit-by-bit equation that we were provided in lab for our 4 bit full adder. However, the module was seemingly inefficient therefore module instantiation was utilized instead. The 32 bit adder module utilizes the 4 bit adder module from lab 2 and instatiates the 4 bit adder 8 times. Although the mistype of an equation in the 4 bit adder module prevented us from executing the 32 bit adder successfully, the answer was made clear eventually. This error made module instantiation the obvious approach as it made miniscule typing errors less likely. In addition, the importance of keeping track of the carry outs and being able to track it from the least significant bit all the way to the most significant bit were made obvious when we realized we needed to determine the different flags. The overflow flag was determined using conditional statements according to the logical interpretation for overflow cases. To test the flags, forced constants such as {0110,28'b0} and {1100,28'b0} were added to ensure that the carry and overflow flags were triggered for appropriate cases. We continued to test values in the lower range including unsigned decimals less than 100. After multiple successful executions, the 32 bit adder was determined to be complete.

2.2 Subtraction

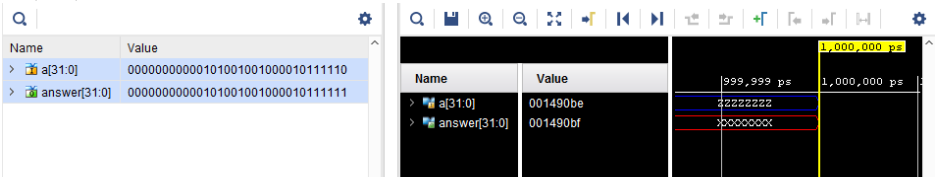
```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 10/28/2017 03:25:09 PM
7  // Design Name:
8  // Module Name: subtractor2
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module subtractor2(a,b, sum, c_out, Cf, Sf, Of);
24
25     input [31:0] a;
26     input [31:0] b;
27     output [31:0] sum;
28     output c_out;
29     output Cf;
30     output Sf;
31     output Of;
32
33     logic [31:0] complement;
34
35     assign complement=~b+(1'b1);
36
37     FA_32bitv2 inst_1(a, complement, sum, c_out, Cf, Sf, Of);
38
39
40 endmodule
```



Because subtraction is just the addition of the first given number and the two's complement of the other, using module instantiation of the previous 32 bit adder seemed more efficient. Two's complement was performed on the second input value for the subtraction operation. This value was stored into a separate variable called complement since this made it easier to see if the initial value properly underwent the two's complement operation. After this, the two values, A and complement were added using an instantiation of the 32-bit adder. The final result would be the different between A and B, and is stored into the "Out" return variable in the main module. Carry and Sign Flags were determined the same way it was in the 32-bit Adder, and was tested with the following cases: 00000101110000001111100001011000 and 00010001000100010001000100010001, 32'hffffffff and 32'hffffffff, 32'h0000eeee and 32'heeee0000.

2.3 Increment

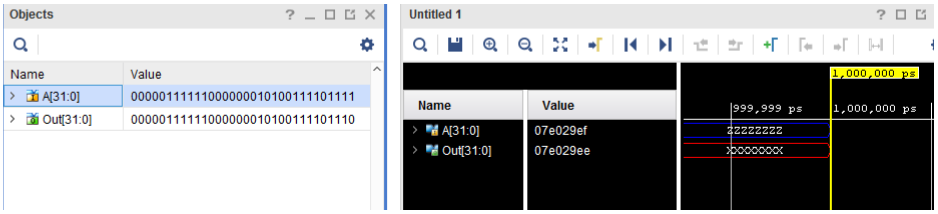
```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date: 10/28/2017 12:29:32 PM
7   // Design Name:
8   // Module Name: increment
9   // Project Name:
10  // Target Devices:
11  // Tool Versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////
21
22
23  module increment(input [31:0] a,
24      output [32:0] answer
25      );
26
27      assign answer=a+1;
28
29  endmodule
30
```



Increment adds 1 to the 32-bit user input and returns the value after incrementing by 1.In SystemVerilog, the input [31:0] A the user entered is 00000037, so the expected output [32:0] answer would be 00000038. The reason the array goes from [31:0] to [32:0] is to handle overflow that would occur when adding 1. We made it more efficient by making it one line.

2.4 Decrement

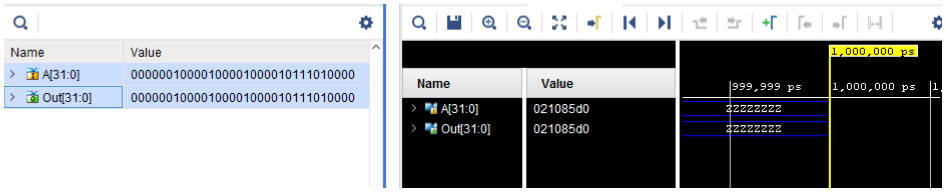
```
23  module decrement(
24      input [31:0] A,
25      output [31:0] Out
26      );
27
28      assign Out = A - 1;
29
30  endmodule
31
```



Decrement subtracts user input by 1 and returns the decrement. In SystemVerilog, the hexadecimal input [31:0] A the user entered was 0000000f, so the expected output [31:0] Out would be 0000000e when subtracting by 1.

2.5 Move

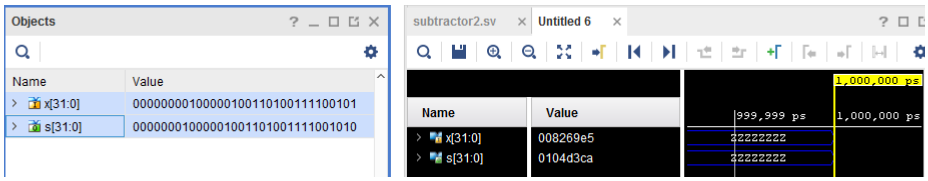
```
23    module move(
24        input [31:0] A,
25        output [31:0] Out
26
27        );
28
29        assign Out = A[31:0];
30
31    endmodule
```



The Move function takes an input and moves it to the output. This is implemented in SystemVerilog by defining an input port and output port. The input receives a 32-bit binary number then assigns it to the output port.

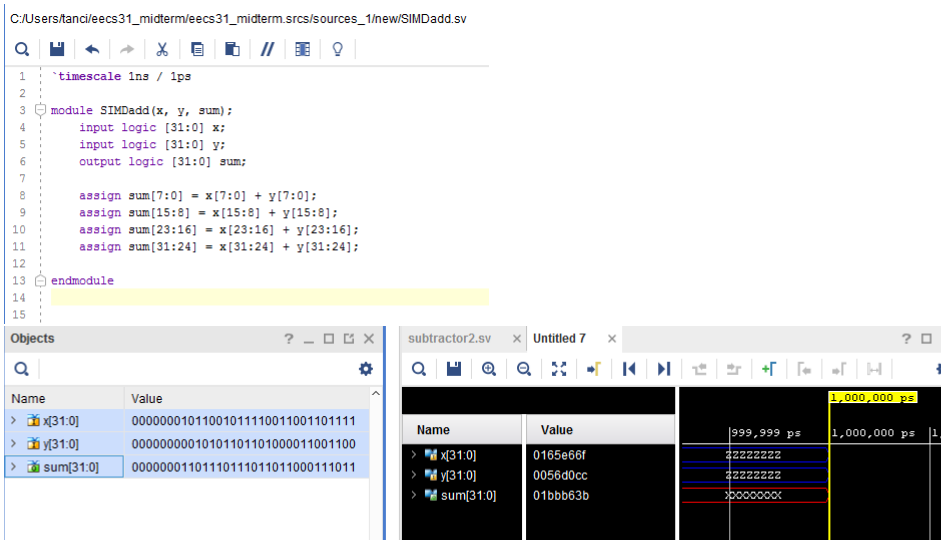
2.6 1-bit Logical Shift Left of A

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date: 10/25/2017 08:57:19 PM
7   // Design Name:
8   // Module Name: ShLeft
9   // Project Name:
10  // Target Devices:
11  // Tool Versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////
21
22
23  module ShLeft(x, s);
24
25      input logic [31:0]x;
26      output logic [31:0]s;
27      assign s = x << 1;
28
29  endmodule
```



The 1-bit logical shift process takes in A as a 32-bit input and performs a 1-bit shift left on A. The result of the shift is then stored into the return output S, which in turn would be the final return value for Out in the main module calling this ShiftLeft module.

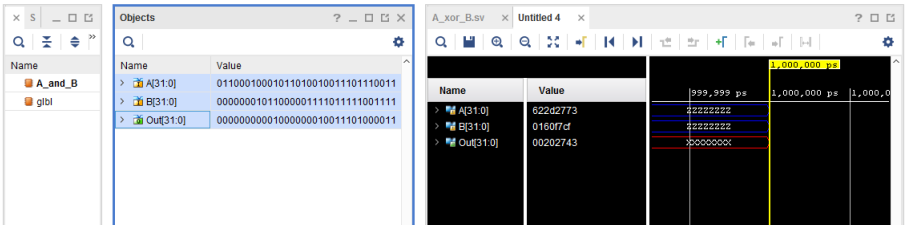
2.7 SIMD Add: Four 8-bits Add



The SIMD Add module takes in two input 32-bit values x and y, and one 32-bit output return sum. Every 8-bits of x and y are added and stored into the corresponding 8-bit location in sum, carry outs ignored as it will not be stored into the resultant sum. Concatenation was an initial idea to saving the result of every 8-bit addition into the resultant output variable, but saving the values directly into an allocated 8-bit area would already directly alter the values in the output variable, therefore a direct assignment was used instead of concatenation for ease in the code. Carry out is ignored because it is out of bounds for the storage space. This module was tested using values where carry out was ensured but not carried over at every 4 bits: x and y as 84'b1000, resulting in 00010000000100000000100000010000, and x = 10110111 00111010 11100010 01101100, y = 10010010 10010010 00101001 10100010 resulting in 01001001110011000000101100001110.

2.8 Bitwise AND

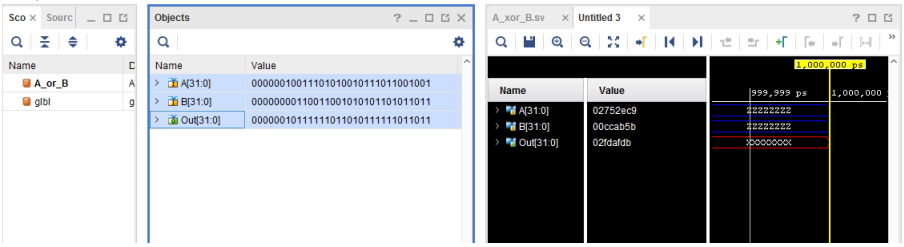
```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 10/25/2017 08:28:14 PM
7  // Design Name:
8  // Module Name: A_and_B
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////
21
22
23 module A_and_B(A,B,Out);
24     input logic [31:0] A, B;
25     output logic [31:0] Out;
26     assign Out=A&B;
27 endmodule
28
```



The bitwise AND module compares two inputted 32 bit numbers and returns either 1 or 0 for each compared bit depending on the given values by using the "&" operator. If there is a 0 the module it returns 0 but otherwise returns 1, following the digital logic AND operation described in class. The code was tested with multiple values including 32'b0 and 1011, 28'b1 before concluding it was accurate.

2.9 Bitwise OR

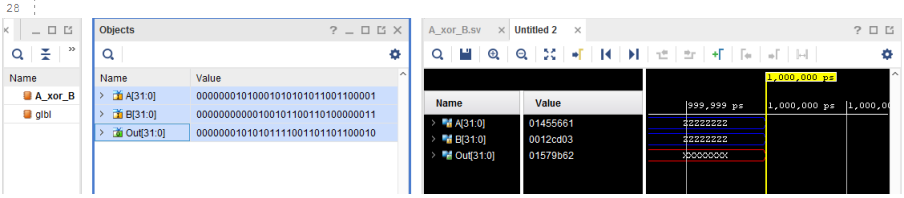
```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 10/25/2017 08:35:02 PM
7  // Design Name:
8  // Module Name: A_or_B
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module A_or_B(A,B,Out);
24     input logic [31:0] A, B;
25     output logic [31:0] Out;
26
27     assign Out=A|B;
28 endmodule
29
```



The bitwise OR module compares two inputted 32 bit numbers and returns either 1 or 0 for each compared bit depending on the given values. By using "|", the operator returned 1 if the two bits being compared had a 1 present, otherwise it returned 0. This followed the OR digital logic definition explained in class and after testing the code with both basic values like 32'b0 and 32'b1 as well as 111,29b'0 and 1001, 28b'1 the code was considered successful.

2.10 Bitwise XOR

```
21
22
23 module A_xor_B(A,B,Out);
24     input logic [31:0] A,  B;
25     output logic [31:0] Out;
26     assign Out=A^B;
27 endmodule
28
```



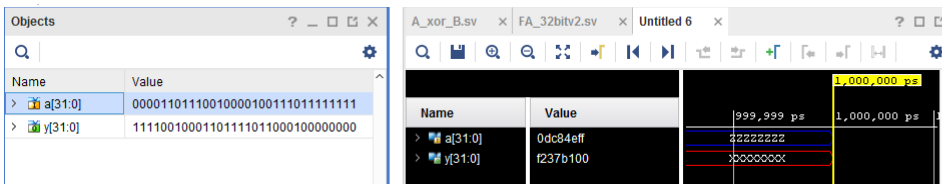
According to the definition of XOR, if the two bits being compared are the same (either both 1 or both 0) then it returns 0. Otherwise, if only one of

the bits is 1, XOR returns 1. By using the XOR operator, ^ the module traverses the index of the first given value with the corresponding index of the second value. After testing values, like 25b'1, 0101110 and 29b'0, 3b'1, the code seemingly accomplished it's goal.

2.11 Complement

C:/Users/tanci/eecs31_midterm/eecs31_midterm.srcs/sources_1/new/complement.sv

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 10/11/2017 08:15:43 PM
7  // Design Name:
8  // Module Name: Complementer
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module Complementer(
24     input logic [31:0] a, //32 bit input
25     output logic [31:0] y //32 bit output
26     );
27
28
29    initial begin
30    assign y = ~a;
31
32
33    $display("a= %32b",a);
34    $display ("y= %32b", y);
35    $finish;
36    end
37
38 endmodule
```



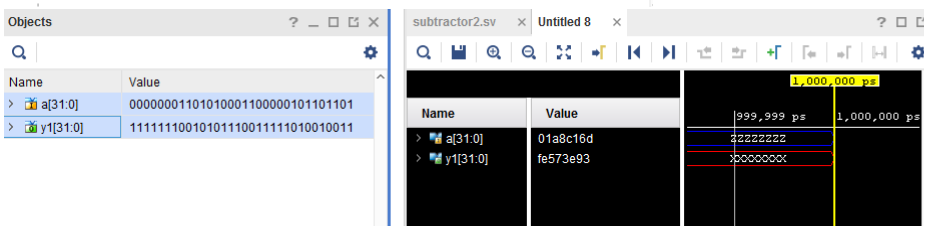
The Complement process takes the user 32 bit input and inverts the inputs. For instance, if the user inputs a binary number for a[31:0] with 00000000000000010000000000000001, we would expect the output of y[31:0] the complement to be 11111111111111101111111111111110. This is implemented in SystemVerilog by inverting the 1's to 0's and vice versa as show in the image below. To implement this in SystemVerilog, we define one input port and one output port. We assign the output y as the NOT of the input a.

2.12 2's Complement

twoscomplement.sv

C:/Users/tanci/eecs31_midterm/eecs31_midterm.srcs/sources_1/new/twoscomplement.sv

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 10/28/2017 12:02:51 PM
7  // Design Name:
8  // Module Name: twoscomplement
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module twoscomplement(
24     input logic [31:0] a,
25     output logic[31:0] y1 //twos complement
26
27     );
28
29     assign y1=~a+1'b1;
30
31
32 endmodule
33
```



The 2's Complement process takes the 32 bit input to get the complement and then add 1 to get 2's complement. For instance, if the user inputs 00000000000000000000000000001111 then the complement would be 11111111111111111111111111110000. With the complement, we would add 1 to the complement to get 11111111111111111111111111110001 which is what we would expect the output would be when implementing the code on SystemVerilog. To implement this in SystemVerilog, we define one input port and one output port. We assign the output as the NOT of the input plus 1 to simulate the two's complement.

2.13 Main Function and Multiplxer

In order to fully impliment the code to simiulate as one unit, the UM group created a main source code that will instantiate all of the previously explained functions, along with a multiplexer(MUX). The inputs of this function are two 32 bit numbers. The outputs are a 32 bit final output, 1 bit carry flag, 1 bit sign flag, 1 bit overflow flag, and a 1 bit zero flag. In addition, there are also 12 temporary value holders that correspond the outputs of each function of the Enhanced ALU and 2 temporoary values for the carry and sign flag. When run, the main fucntion will instatiate all functions at once, the final outputs and flags will be determined through the use of the multiplexer.

The multiplexer functions as a case assignment statement, using 12 temporary varibales that correspond to each function of the Enhanced ALU, the MUX uses the value of the selector bits to assign the appropriate value to the final output. There is another case assignment statement that is used to assign the correct values for the overflow and sign flag. Previously there was a problem found in the project that would produce an "x" value. This was found to be from the assignment of the same variable to two differnet instatiantions. This flag case assignment will only produce a value for the selected addition or subtraction function, otherwise it will be "x".

3 Implementation and Bit-stream transfer to FPGA

Since the FPGA board does not possess the ability to appropriately display 32 bits, it becomes necessary to alter the inputs in a way that the board will work appropriatley with the Enhanced ALU. This was done by redefining the inputs and output as a 6 bit number instead of a 32 bit number. In the main function code, there are 2 new values that would be defined as 32 bit numbers and take in the values of the inputs which are concatenated with twenty-six 0's. These new values place hold the values of the 6 bit inputs so that the rest of the modules would not have to be altered. A new variable was also assigned and would be the output variable in all the instatiations. The 31 bit to the 26 bit of this number would then be assigned to the 6 bit main output.

Once the board was synthesized and the bit stream was generated, it was observed that the FPGA was able to diplay most of the functions. Some functions that could not be appropriatley dispalayed on the FPGA are the increment and decrement functions. This was found to not be an error in the code, but a lack of defined ability. The increment and decrement cannot be ovbserved on the FPGA using this code, however simulations state that it is functional.

4 Conclusion

Ultimately, the Enhanced ALU was designed and simulated with concepts and formulated equations learned previously through block diagrams and Karnaugh Maps. For much of this project, trial and error helped guide the approach for each ALU operation towards efficiency and modularity. Building on the concepts known already about multiplexers, ALUs, bit-manipulation, and other arithmetic functions, we learned to implement this to the Enhanced ALU and practiced in SystemVerilog.