ECEN2350 Digital Logic - Project 1

Priyanka Makin
Savio Tran

03/05/2017

Introduction

Project 1 was a good introductory project to using Verilog Hardware Design Language. We explored coding arithmetic, logical, and comparison modules. For this, we had to learn how to derive logical equations, decide inputs and outputs, and assign wires and output registers. Also, we had to learn the syntax for writing continuous and procedural assignments, “always”, “genvar” (which involve “for” statements), and “if” statements.

This project has 12 different functions sorted into three main categories. The Arithmetic module includes add, subtract, multiply by two, and divide by two operations. The operations for the Logical module are AND, OR, XOR, and NOT. The Comparison module operations are Equal to , Greater than, Less than, and Max value. The two buttons, Key1 and Key0 on the DE10-Lite select the module and Switches 8 and 9 select between the operations within a module. Figures 1 through 4 below show a block diagram representation of our project.

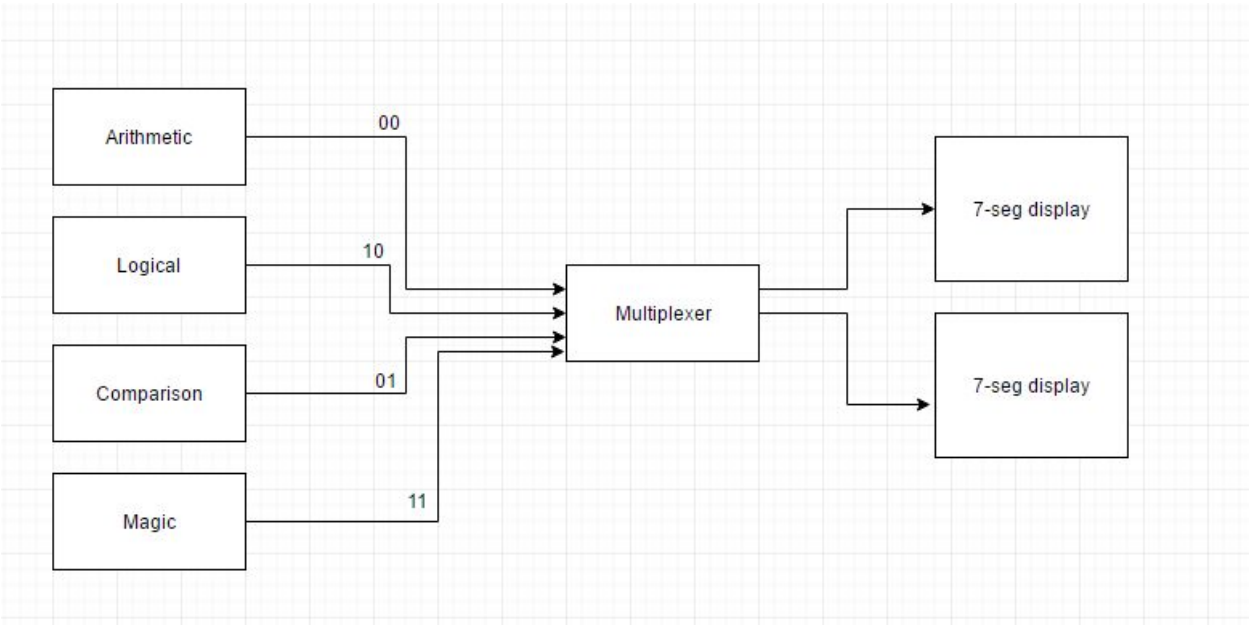

Figure 1: Project1_top block diagram

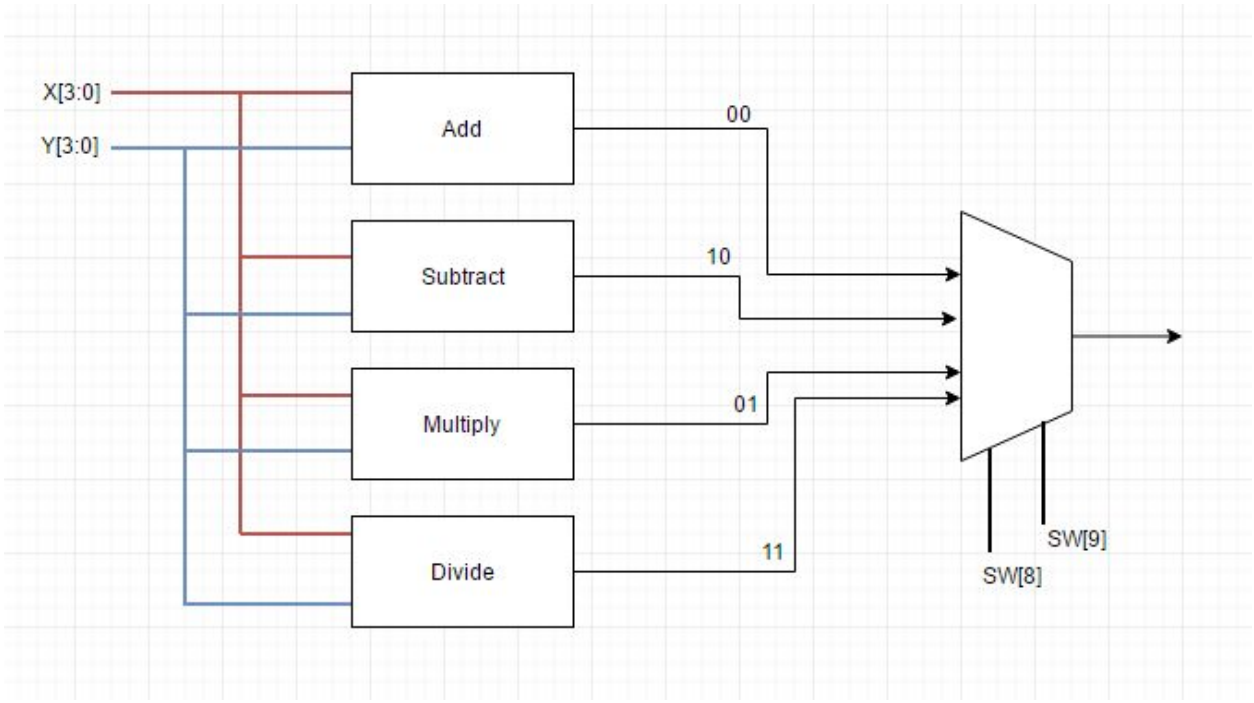

Figure 2: arithmetic.v block diagram

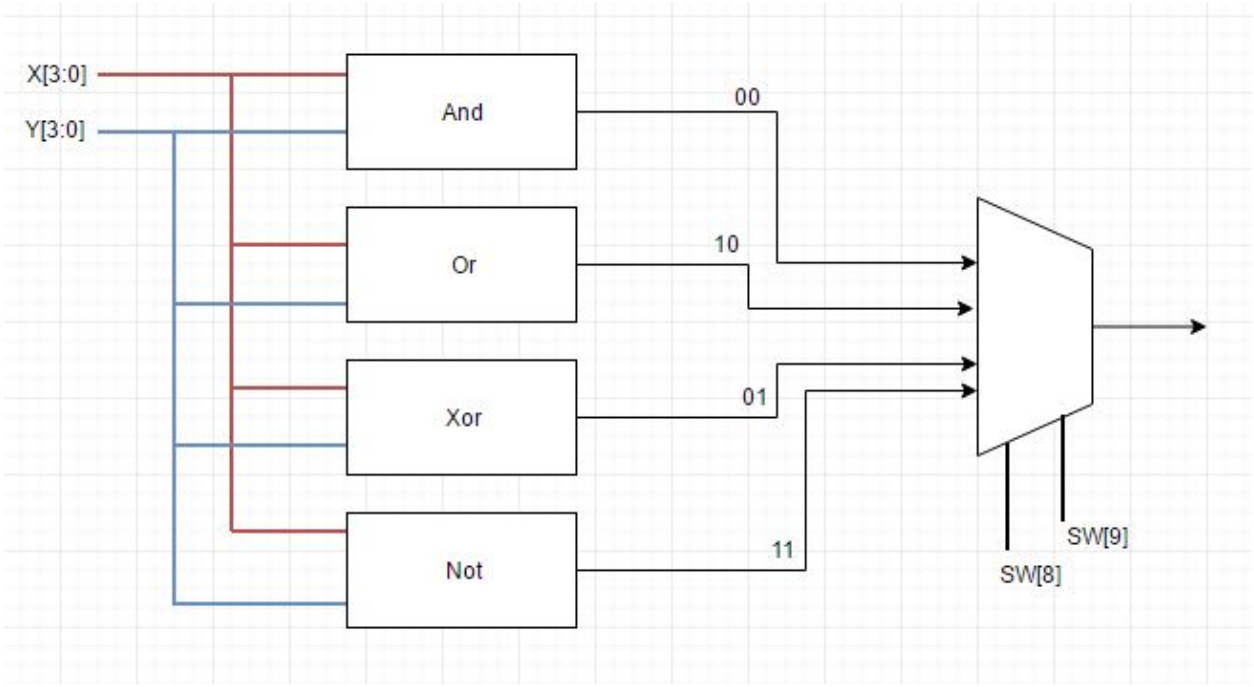

Figure 3: logical.v block diagram

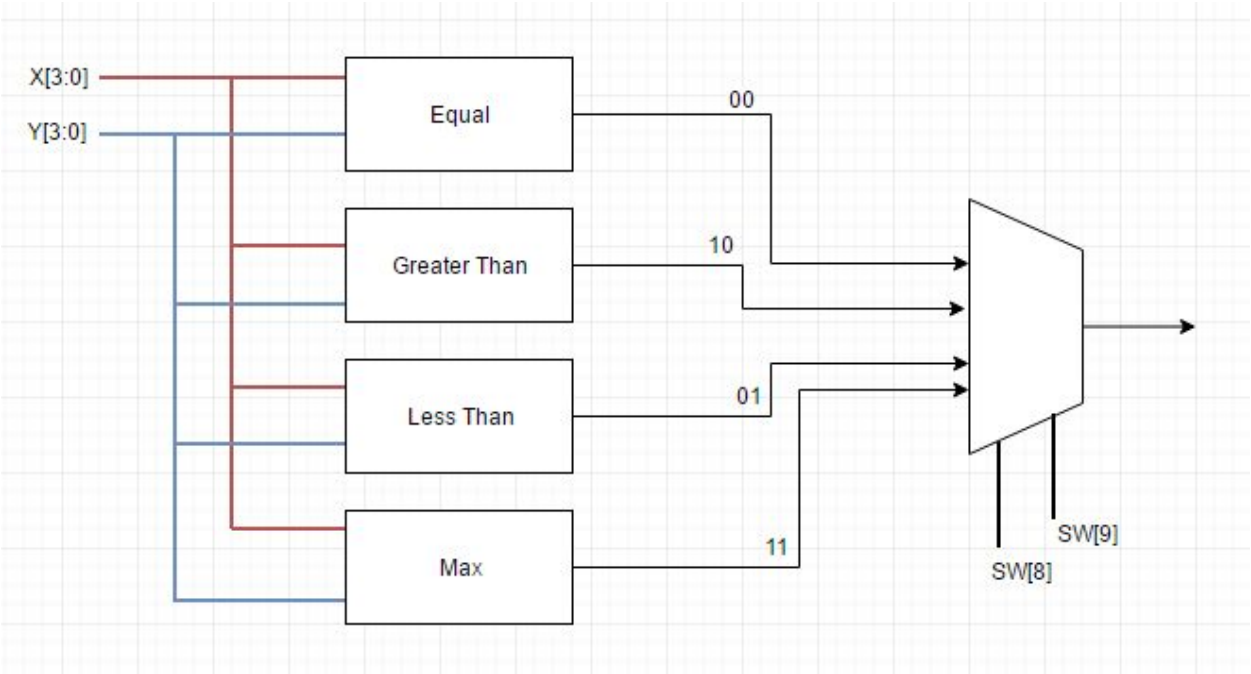

Figure 4: Compare2.v block diagram

How we encoded the modes in this project can be seen below in Table 1. Once again, the buttons Key0 and Key1 specifies the module and Switch8 and Switch9 specify the operation within the module. This scheme has room for one more major category, Magic, that we did not complete.

	Key 1	Key 0	Switch 9	Switch 8
Arithmetic	0	0	N/A	N/A
Add	0	0	0	0
Subtract	0	0	1	0
Multiply	0	0	0	1
Divide	0	0	1	1
Logical	1	0	N/A	N/A
And	1	0	0	0
Or	1	0	1	0
Exor	1	0	0	1
Not	1	0	1	1
Comparison	0	1	N/A	N/A
Equal	0	1	0	0
Greater Than	0	1	1	0
Less Than	0	1	0	1
Max	0	1	1	1
Magic	1	1	N/A	N/A

Table 1: Mode Encoding

So, for example, if Keys 1 and 0 are both pressed, the output would be in Arithmetic mode. Then, if switch 9 is up and switch 8 is down, the output of Arithmetic Subtract will appear on the 7 segment display. So, if a button is held down, then Key = 0, and if a switch is up, then Switch = 1.

Each of these modes take in either 2 4-bit inputs X and Y, or 1 8 bit input, W. The values of these inputs are determined with switches 7 through 0. Y is made up of Switches [7:4] while X is made up of switches [3:0]. An 8 bit input W is the concatenation of Y and X, or Switches [7:0]. Looking at the board, these bits can be read from most significant bit on the left to the least significant bit to the right. So, the most significant bit of X would be the value of Switch[3], and the least significant bit of Y would be the value of Switch[4]. In the case of W, the most significant bit would be the value of Switch[7], and the least significant bit of W would be the value of Switch[0]. Figure 5 is a visual representation of how we configured the DE10-Lite's peripherals.

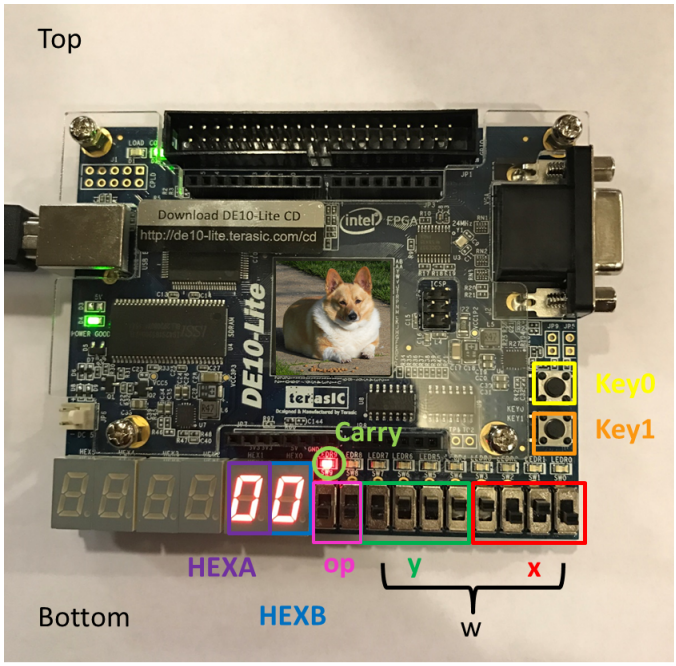

Figure 5: DE10-Lite peripheral configuration

For further clarity, if the switches have the following values shown in Table 2, X would have a value of 0b1101, Y would have a value of 0b0001, and W would have a value of 0b11010001.

Switch	Sw 7	Sw 6	Sw 5	Sw 4	Sw 3	Sw 2	Sw 1	Sw 0
Up/Down	Up	Up	Down	Up	Down	Down	Down	Up
Binary Value	1	1	0	1	0	0	0	1

Table 2: Example Switch Encoding

The outputs of all the modes are represented in hexadecimal on the seven segment display with the exception of the extra carry out or borrow bits in Arithmetic mode. The Arithmetic module displays the output on the seven segment display, and if there is a carry bit or borrow bit, LED 9 turns on. If the carry bit is equal to 0, the LED turns off.

Lab Procedure/Results

Multiplexer Modules

This project uses 3 different multiplexer modules. The first is a 2-1 multiplexer module that takes in a selector and two inputs, and outputs one of the inputs. This is done with a continuous assignment of (x & selector) | (y & selector). This multiplexer is only used in the comparison module to find the maximum value.

The next multiplexer module is 4-1 which will be referred to as one bit multiplexer in this report. This takes in four one bit inputs as well as a two-bit selector input. It outputs one bit. This was done using a continuous assignment. So, the multiplexer takes inputs a, b, c, and d, with selectors s1 and s2. Each of the inputs is anded with either s1s2, ~s1s2, s1~s2, or ~s1~s2. So, the output will hold the value of one of the four inputs a, b, c, or d. The values of the two selectors determine the output.

The last multiplexer module takes in 4, eight-bit vectors and a two-bit selector. This will output one eight-bit vector. The module uses “generate” and a “for loop” to instantiate the 4-1, one-bit multiplexer 8 times. So, this module essentially creates eight 4-1 multiplexers. All of them take in the same selectors. Each time the for loop runs, it instantiates a multiplexer with bits a[i], b[i], c[i], and d[i] as inputs, so that each wire is taken into account.

Arithmetic Module

Arithmetic mode consists of four functions: add, subtract, multiply by two, and divide by two. The add and subtract operations both take two four bit numbers, performs an operation on them, and then outputs a four bit number with a carry. Our code concatenates this four bit number with four zeros so that it can be put through the eight-bit multiplexer, to choose between operations, and represented on the HEXB seven segment display.

Arithmetic Add

Add takes two 4 bit numbers, X and Y and adds them. The seven segment display shows the first four bits of the sum as one hexadecimal digit, and uses the LED to represent a carry out bit. The ripple carry adder works using a full adder module that is instantiated multiple times using a generate loop. Each adder takes the previous adder’s carry out as a carry in value. Below, Figure 6 shows the summation of X = 0xA and Y = 0x7. The output would be a 0x11, but since only 4 bits are accounted for, the output is a 1 and the carry out light is on.

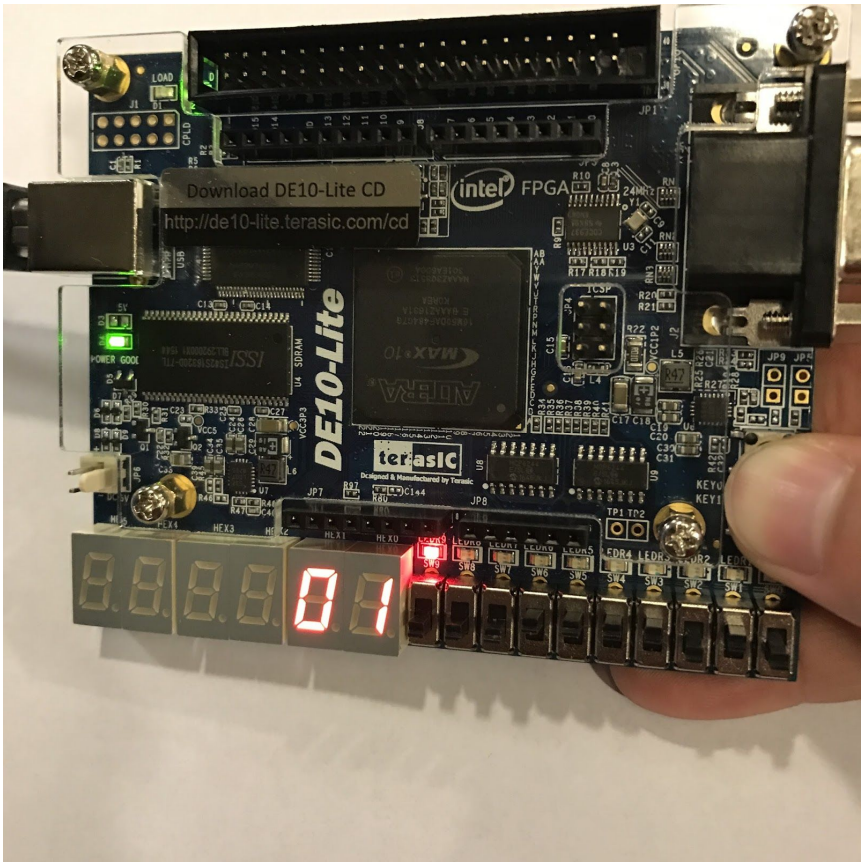

Figure 6: Test output arithmetic add

Arithmetic Subtract

Arithmetic Subtract takes the difference X-Y. Like Arithmetic add, this is a ripple subtractor. It uses a full subtractor module, and in the Arithmetic module, the Subtractor operation is instantiated multiple times using the previous Subtractor's borrow out as a borrow in. The LED turns on for a borrow out. Figure 7 shows Arithmetic Subtract taking the difference of 0xA - 0x7, which should be 3, no borrow needed.

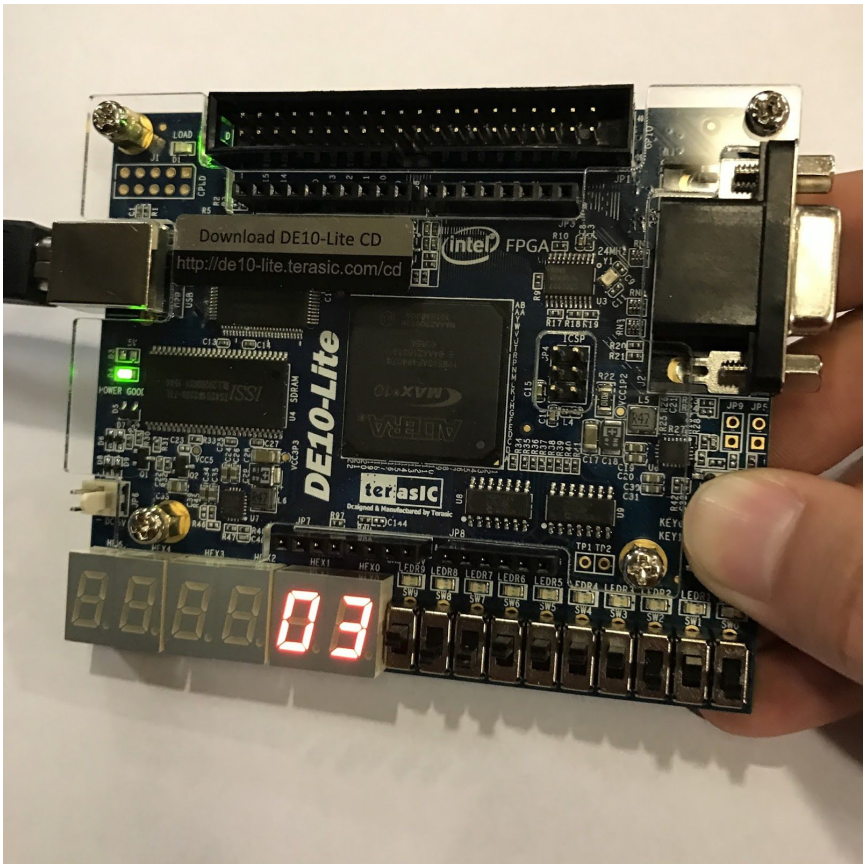

Figure 7: Test output arithmetic subtract

Arithmetic Multiply

Arithmetic Multiply uses bit shifting to multiply by 2. Multiplying by 2 takes the input W, and performs 1 left bitshift. The MSB of W becomes the carry out, so if W[7] was 1, then the LED turns on. In this case, the carry out means that the next most significant bit would have the value of the carry out if there was space for it. Below in, Figure 8, is the Arithmetic Multiply with an input W = 0xA7. So, W = 167 in decimal, and 167*2 = 334, which is 0x14E, so the output of 0x4E with the carry light is agrees with the solution.

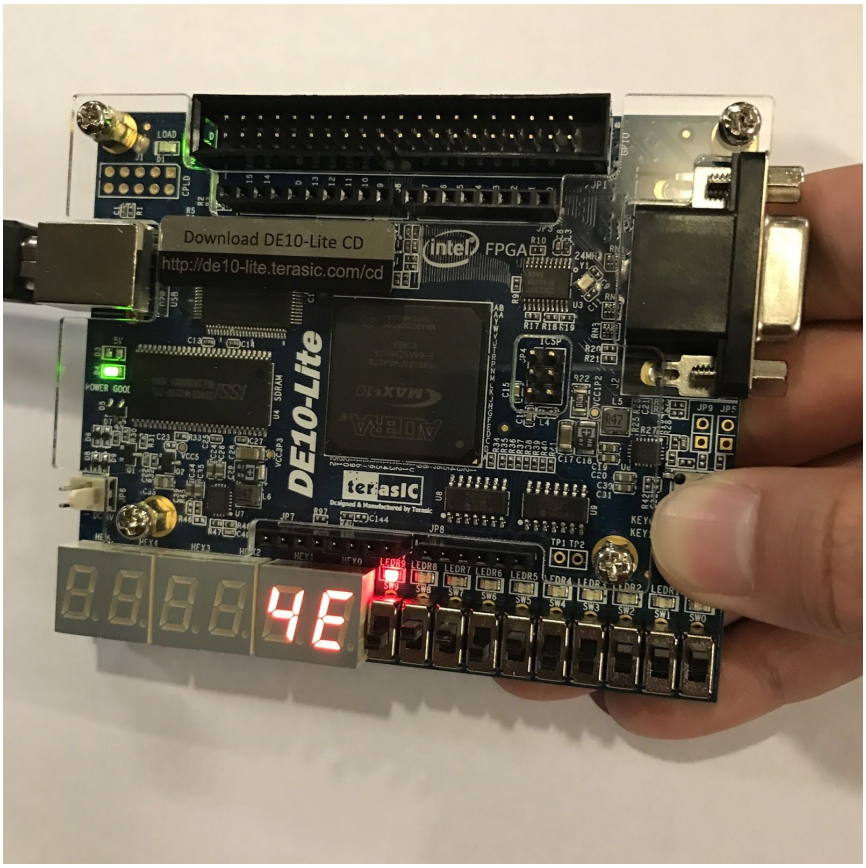

Figure 8: Test output for arithmetic multiply

Arithmetic Divide

Arithmetic divide divides W by 2 using 1 right bitshift. The remainder is equal to the LSB lost in the bitshift. So, W[0] is the remainder. Below is an example of the divide mode with an input of W = 0xA7 = 0b10100111 = 0d167. Then, W / 2 = 0b01010011 = 0d83 = 0x53 with a carry out of the one bit shifted off.

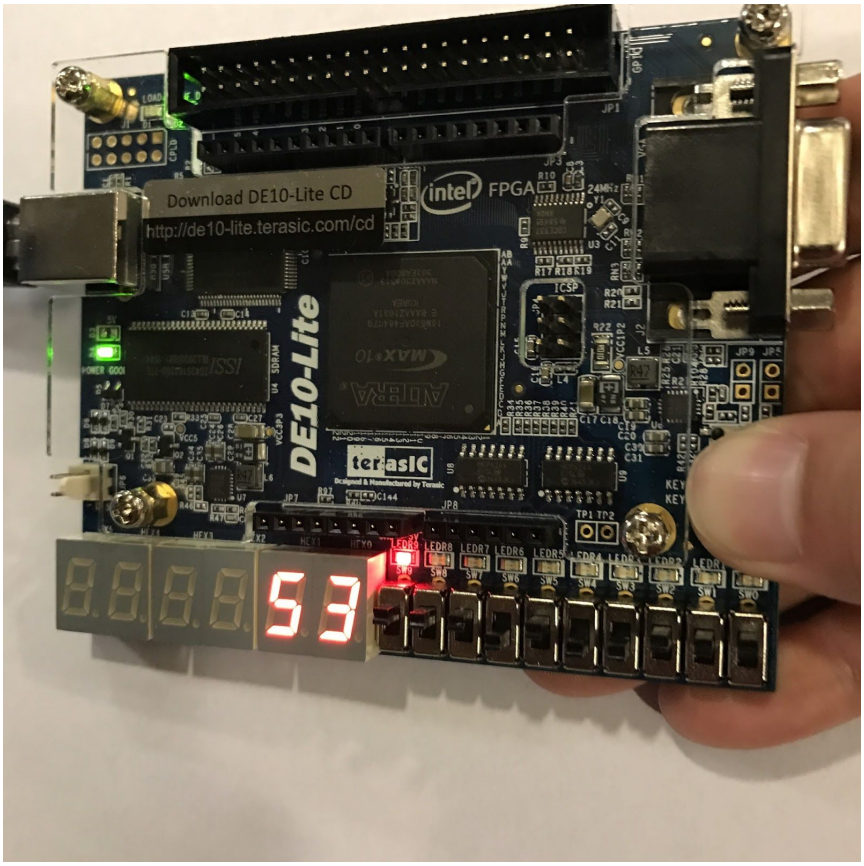

Figure 9: Test output for arithmetic divide

Logical Module

The Logical module also has four main operations: AND, OR, XOR, and NOT. For the first three operations, AND, OR, and XOR, four bit numbers X and Y are operated on. Their output is also a four bit number. Because of this, we have to concatenate four zeros to the four bit output so that the total is eight bits. The NOT operation takes an input of eight bits and inverts them. Since all the operations yield an eight bit number, a multiplexer can be applied to choose between them.

Logical AND

Logical AND does a bitwise AND of two four bit numbers, X and Y. This is done in verilog with continuous assignment of the equation x[3:0] & y[3:0]. Below, in Figure 10 is an example of Logical AND with inputs X = 0xA and Y= 0x7. 0b1010 & 0b0111 should and to 0b0010, which is 0x2. This agrees with the board's output.

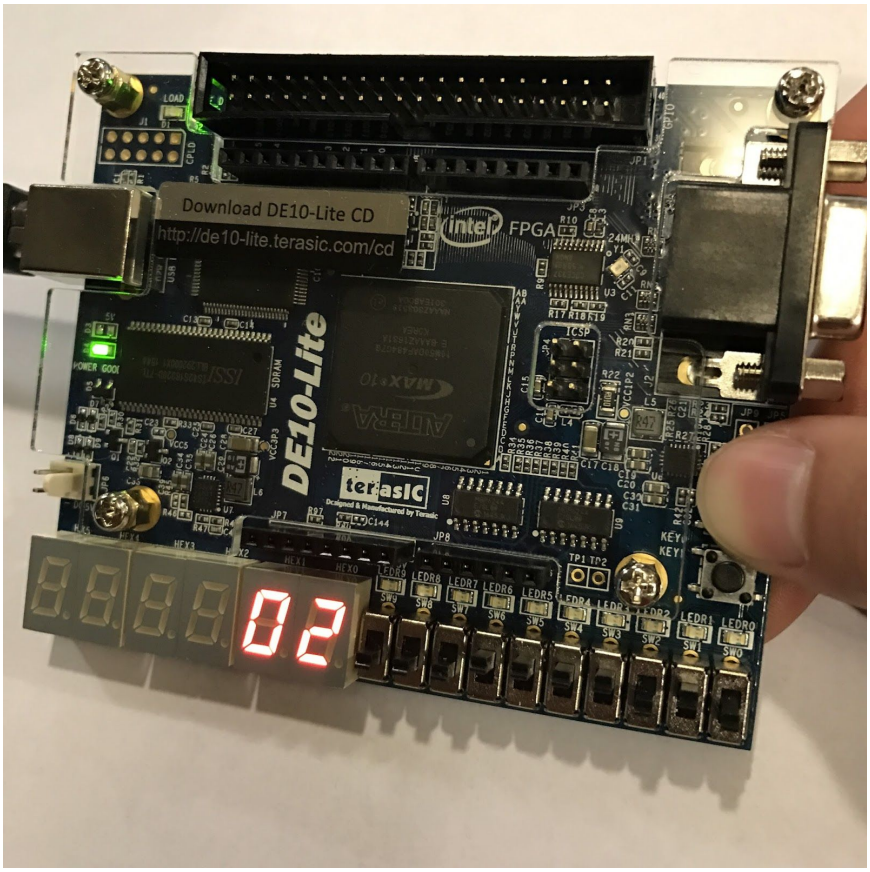

Figure 10: Test output for logical AND

Logical OR

Logical OR works in a similar way as Logical And. It also uses a continuous assignment, only the inputs X and Y are OR'd together. So, the output is assigned to equal x[3:0] | y[3:0]. Below is an example with inputs X = 0xA and Y= 0x7. 0b1010 | 0b0111 = 0b1111, which is 0xF, which agrees with the board's output.

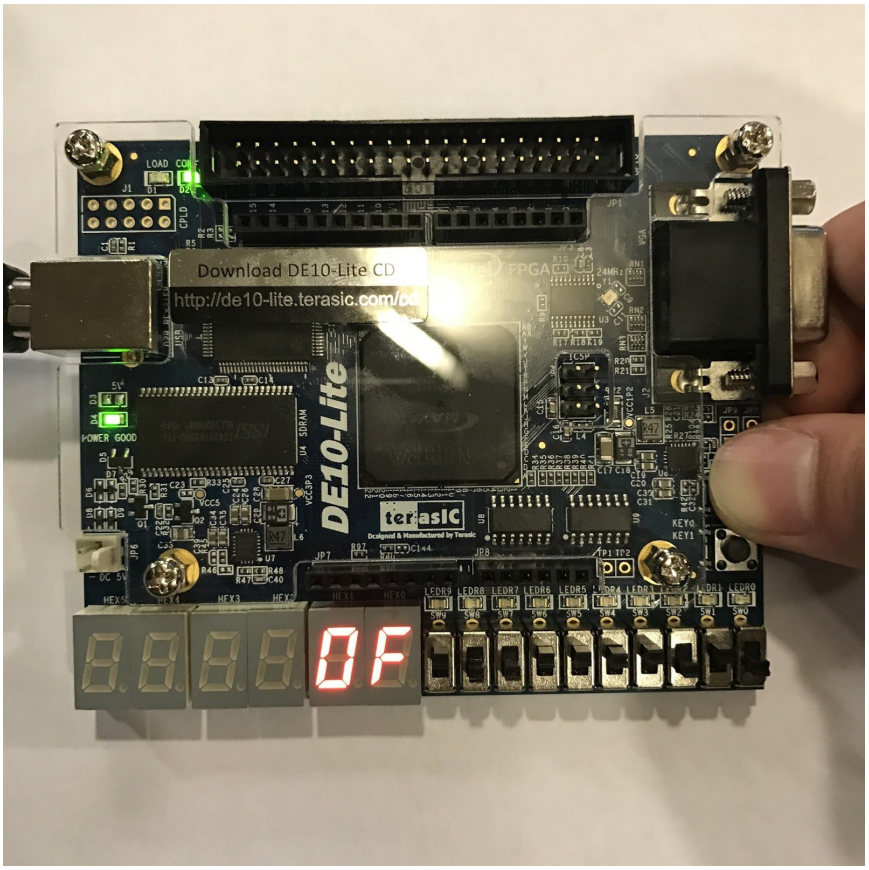

Figure 11: Test output for logical OR

Logical XOR

Logical XOR is also a continuous assignment. The inputs X and Y are XOR'd together using the carrot symbol. So, the output is assigned to equal x[3:0] ^ y[3:0]. Below is an example with inputs X = 0xA = 0b1010 and Y= 0x7 = 0b0111. Thus, X and Y XOR'd with each other becomes 0b1101 = 0x0D.

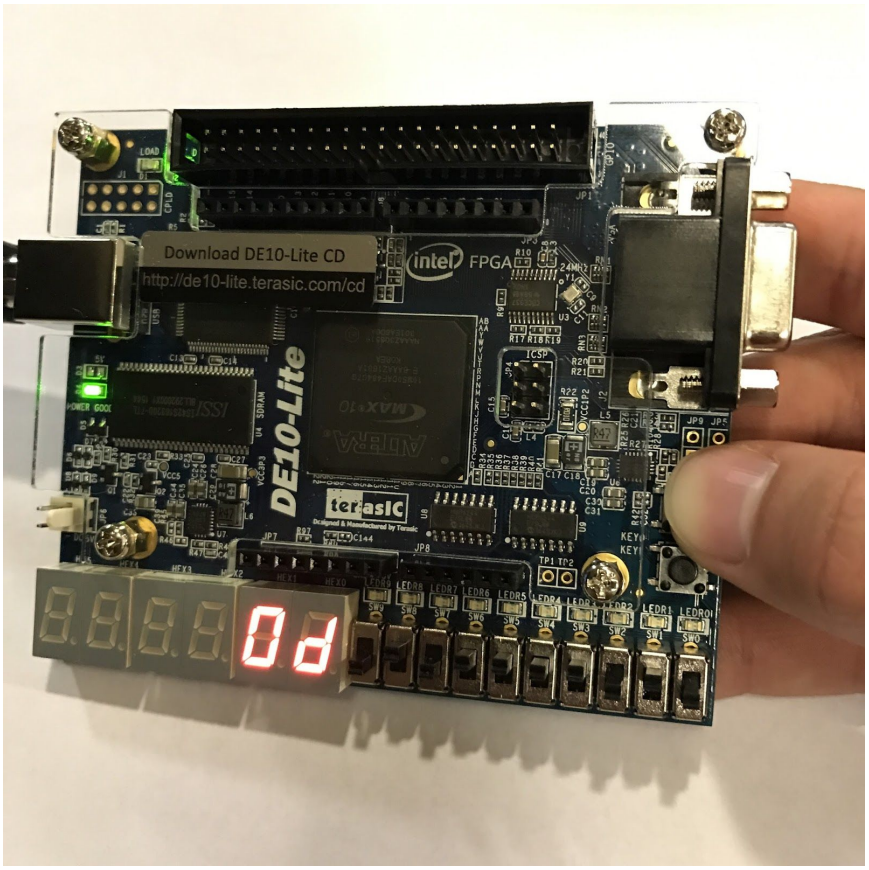

Figure 12: Test output for logical XOR

Logical NOT

Logical NOT takes an eight bit input W and inverts each bit. That is done with a continuous assignment where the output is equal to ~W[7:0]. Below is an example of Logical NOT with an input of W = 0x A7. W = 0b10100111, so ~W = 0b01011000, which is 0x58, which agrees with the board's output.

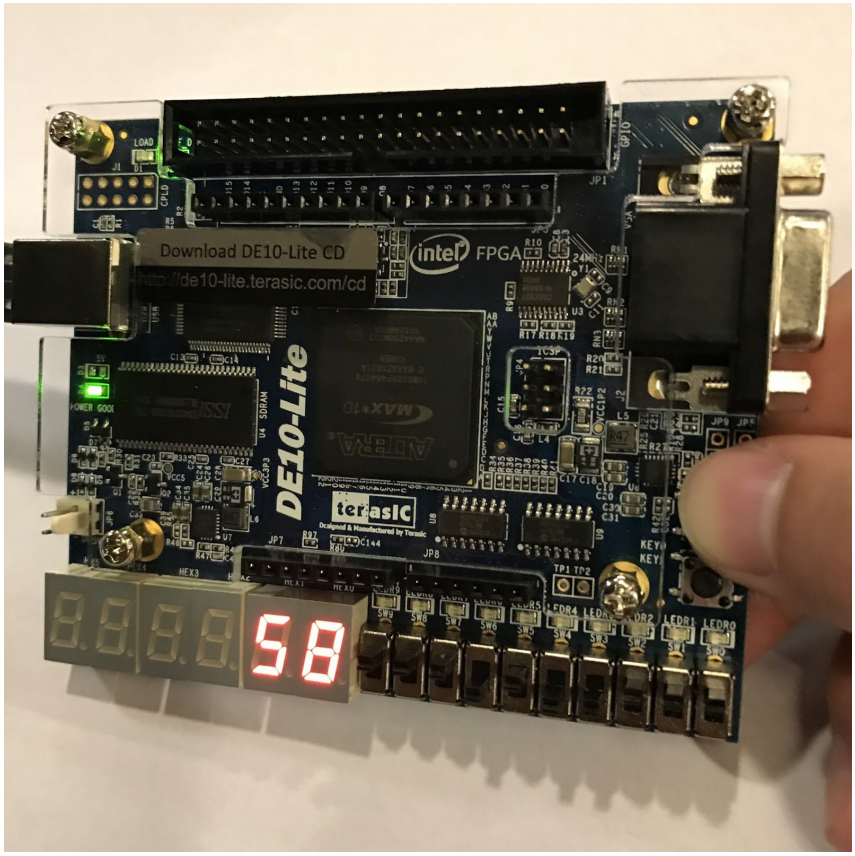

Figure 13: Test output for logical NOT

Comparison Module

This module has functions equal, greater than, less than, and maximum. Coding this module went a bit differently because we had to use registers and procedural assignment. Within an always block, if statements are used to assign values to these registers. If X>Y, then greater = 1 while equal and less are set to 0. Similarly, if X=Y, equal is set to 1 while greater and less are set to 0. Finally, if X<Y, then less is set to 1 while equal and greater are set to 0. Eight wires are created for Greater than mode, Less than Mode, and Equal Mode. Since each of these will only display a logical 1 or 0 on the sevens segment display, their outputs only need one bit. Therefore, wires[7:1] are set to zero. Then, wire[0] is assigned to the value of its corresponding register. So, for example, wire_greater[0] is assigned to equal the register greater. Then, the eight wires in each of these mode are passed into an eight-bit 4-1 multiplexer. Max needs to output a four-bit value, so maxwire[7:4] are set to zero, and max[3:0] will be set as seen below. Each of these modes has eight wires in a vector that pass into the eight-bit mux. This is the output of the comparison module.

Comparison Equal

As said above, for the equal wire, Cequal, Cequal[7:1] are set to zero. Cequal[0] is assigned to the value of the register, equal. Below is an example of Comparison Equal for X = 0x8 and Y= 0x7. 0x8 is not equal to 0x7, so the output of 0 from the board makes sense.

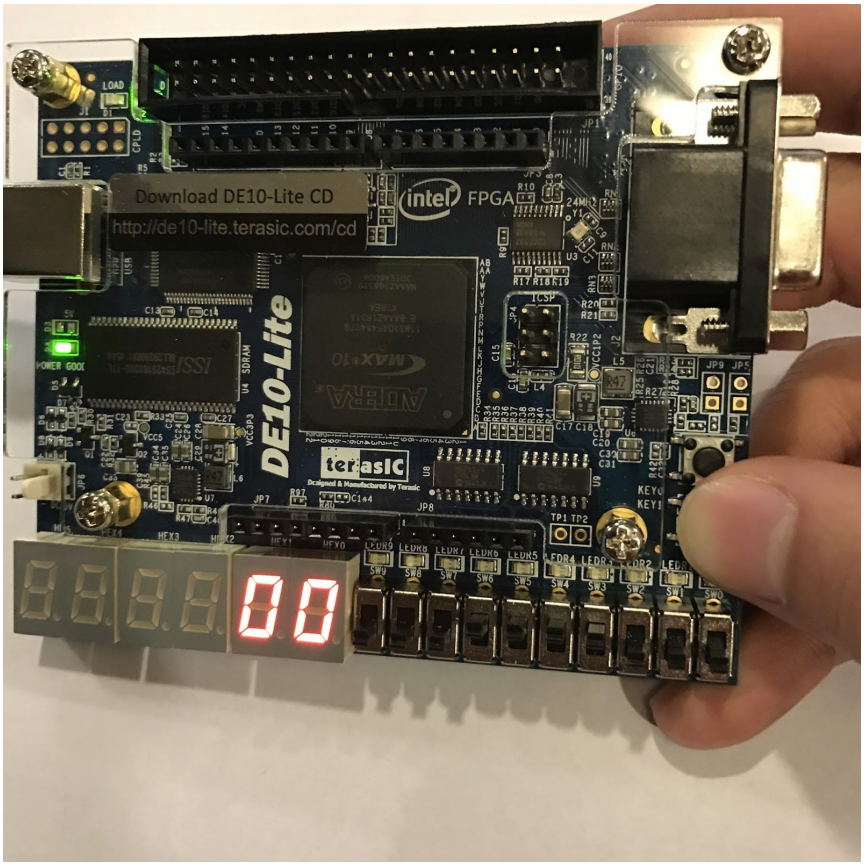

Figure 14: Test output for comparison equal

Comparison Greater Than

Comparison Greater Than works the same way as Comparison Equal. However, the last bit of the Comparison wire is set to equal the register greater. Below is an example of Comparison Equal for X = 0x8 and Y= 0x7. 0x8 is greater than 0x7, the output of 0x1 on the board makes sense.

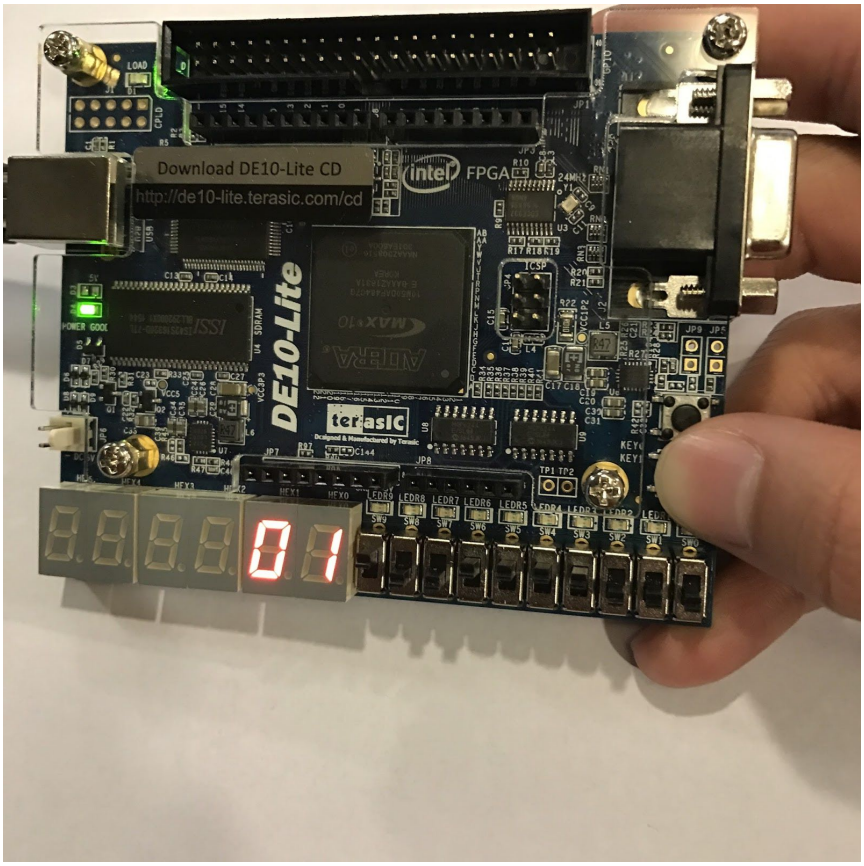

Figure 15: Test output for comparison greater than

Comparison Less Than

Comparison Less Than works similar to the other operations, however, the last bit of the Comparison Less Than wire is set to equal the register less. Below is an example of Comparison Less Than for X = 0x8 = 0d8 and Y= 0x7 = 0d7. So, X is not less than Y, so the output is zero.

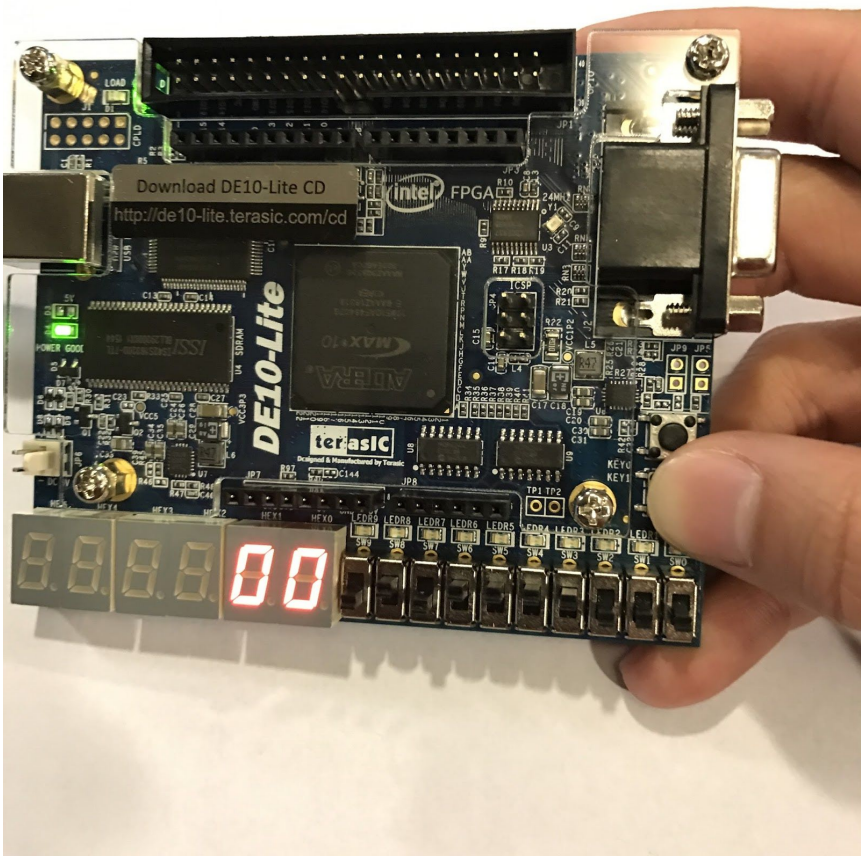

Figure 16: Test output for comparison less than

Comparison Max

Max works similarly to the eight bit multiplexer. Max uses a generate for loop to instantiate multiple 2-1 multiplexers. Each iteration of the loop instantiates the 2-1 bit multiplexer using inputs X[i], Y[i] and the register greater as a selector. This runs from i=0 to i=3, and outputs 4 bits that will either equal the value of X or Y depending on which was greater. Below is an example of Comparison Max with X = 0x8, and Y = 0x7. 0x8 is greater than 0x7, so the board's output of 0x8 is valid.

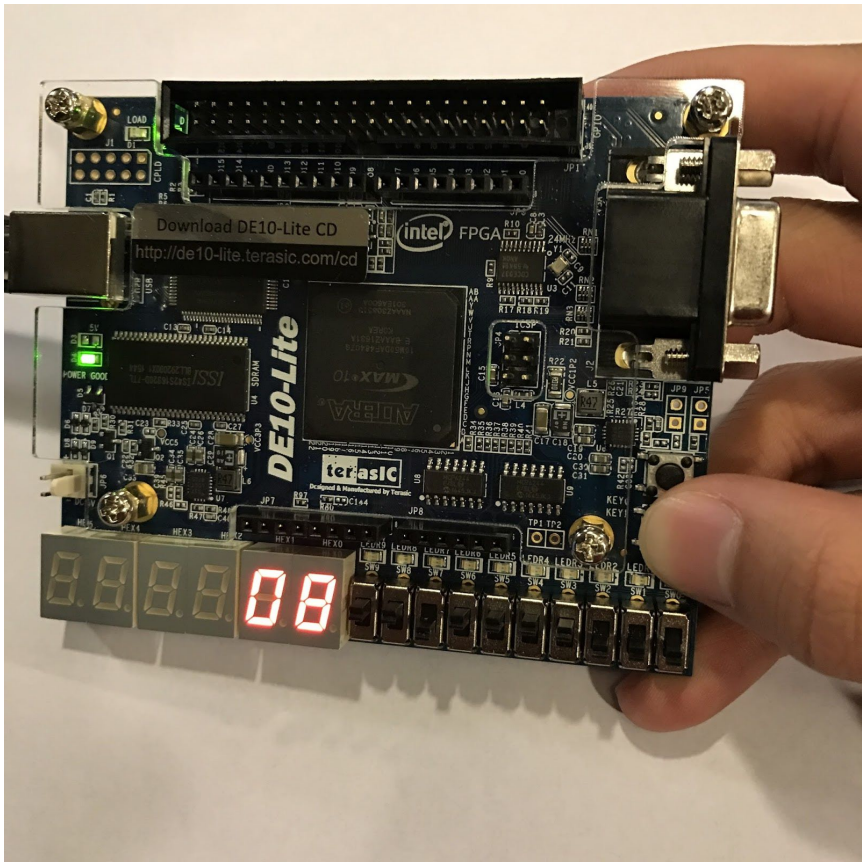

Figure 17: Test output for comparison max

Seven Segment Display

The segment display function is quite straightforward. It is just one large case statement in which we assign the hexadecimal seven segment representation to the four bit binary number.

Project1 (Top File)

The Top file takes in input of the switches, buttons, and outputs to the seven segment display and LEDs. Each of the three big modules, Arithmetic, Logical, and Comparison output an eight-bit vector, so, the top file instantiates each of these to create an eight-bit wire.These use the switches, where in the place of X, Sw[3:0] is used, and for Y, Sw[7:4] is used. These are passed into the eight-bit multiplexer where the fourth input is zeroes (instead of Magic). The selector is Key[1:0] from the buttons. The eight bit output of this multiplexer is then split into [7:4] and [3:0] which are separately passed into the segment display module to output to the two 7 segment displays. The carry out LED only applies to the arithmetic module, so, to handle this a multiplexer is used. The carry out output of the arithmetic module is passed into a 4-1 one-bit multiplexer using the buttons as a selector again. The carry out corresponds to the selector combination that is used to select the arithmetic module in the eight-bit multiplexer. All other cases output 0 from the multiplexer. Finally, LED[9] is assigned the the value of this multiplexer, so that the LED lights up only while in Arithmetic mode.

Conclusion

The hardest part of this project was starting. Learning the syntax of verilog was extremely difficult to get, especially since a lot of it felt counterintuitive in comparison to programming. Learning the difference between a wire and a register and where to use them also proved to be difficult. However, once we moved past these issues, the project all fell together. We started with logical, which was fairly simple, but learning the syntax was difficult. Making the comparison module was the most difficult, as it was hard to work with registers. However, after that, doing the arithmetic module took about 10 minutes, given the logical equations we derived in class. While arithmetic was likely not easier than the comparison module, it felt easier as we picked up on the syntax of verilog as well as how to think about these from a digital logic approach rather than a programming approach.

Generally, the reason why things didn't work was due to syntax error and not knowing how to use the interface. We had a lot of issues successfully coding procedural assignments because they didn't take assignments and we had to create registers to store the information continuously. A lot of our errors were resolved by trouble-shooting and replacing and removing different keywords and punctuation.

For next time, we should definitely start our project earlier than we did for Project 1. However, now that we are more familiar with the way Quartus works and the syntax used for VHDL, I think future projects may progress more smoothly. Also, with starting earlier, we would have more time to attempt the extra credit and learn more Verilog programming.

Appendix

Code Screenshots

Project1.v

```
module Project1(HEXA, HEXB, LED, SW, BUT);
   input [9:0]SW;
   input [1:0]BUT;
   output reg [9:0]LED;
   output [6:0]HEXA;
   output [6:0]HEXB;

   //Wires for comparison
   wire equal;
   wire greater;
   wire less;

   //Module outputs
   wire [7:0]logical_out;
   wire [7:0]arithmetic_out;
   wire carry;
   wire [7:0]compare_out;
   wire [7:0]segwire;

   //Module instantiation
   logical marinara(SW[3:0],SW[7:4],SW[9:8],logical_out);
   Compare2 alfredo(SW[3:0],SW[7:4],SW[9:8],compare_out, equal, greater, less);
   arithmetic pesto(SW[3:0],SW[7:4],SW[9:8],arithmetic_out, carry);

   //Multiplexer
   eightbitmux supersaucy (arithmetic_out, logical_out, compare_out, 8'b00000000, BUT, segwire);

   //7-seg displays instantiation
   segdisplay u1(HEXA,segwire[3:0]);
   segdisplay u2(HEXB,segwire[7:4]);

   //Carry LED instantiation
   always @ (carry)
      begin
      LED[9] = carry;
      end

endmodule
```

segdisplay.v

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

//7-segment assignments
always @*
   begin
      case(SW)
         4'b0000: HEXB = 7'b1000000;// Hexadecimal 0
         4'b0001: HEXB = 7'b1111001;// Hexadecimal 1
         4'b0010: HEXB = 7'b0100100;// Hexadecimal 2
         4'b0011: HEXB = 7'b0110000;// Hexadecimal 3
         4'b0100: HEXB = 7'b0011001;// Hexadecimal 4
         4'b0101: HEXB = 7'b0010010;// Hexadecimal 5
         4'b0110: HEXB = 7'b0000010;// Hexadecimal 6
         4'b0111: HEXB = 7'b1111000;// Hexadecimal 7
         4'b1000: HEXB = 7'b0000000;// Hexadecimal 8
         4'b1001: HEXB = 7'b0011000;// Hexadecimal 9
         4'b1010: HEXB = 7'b0001000;// Hexadecimal A
         4'b1011: HEXB = 7'b0000011;// Hexadecimal B
         4'b1100: HEXB = 7'b1000110;// Hexadecimal C
         4'b1101: HEXB = 7'b0100001;// Hexadecimal D
         4'b1110: HEXB = 7'b0000110;// Hexadecimal E
         4'b1111: HEXB = 7'b0001110;// Hexadecimal F
      endcase
   end
endmodule
```

onebitmux.v

```
module onebitmux(A, B, C, D, op, out);
   input A;
   input B;
   input C;
   input D;
   input [1:0]op;
   output out;

   //Multiplexer
   assign out = (~op[0] & ~op[1] & A ) |
                (~op[0] & op[1] & B) |
                (op[0] & ~op[1] & C) |
                (op[0] & op[1] & D);
endmodule
```

eightbitmux.v

```
module eightbitmux (A, B, C, D, sel, out);
   input [7:0]A;
   input [7:0]B;
   input [7:0]C;
   input [7:0]D;
   input [1:0]sel;
   output[7:0]out;

   //Duplicate multiplexer for all 8 bits
   genvar i;
   generate
      for(i=0; i<8; i = i + 1)
      begin: NMuxies
            onebitmux stage(A[i], B[i], C[i], D[i], sel[1:0], out[i]);
      end
   endgenerate

endmodule
```

arithmetic.v

```
module arithmetic (x, y, op, z, carry);
   input [3:0]x;
   input [3:0]y;
   input [1:0]op;
   output [7:0]z;
   output carry;

   wire [7:0]aadd;
   wire [7:0]asubtract;
   wire [7:0]amultiply;
   wire [7:0]adivide;

   //Wires holding carries for arithmetic operations
   wire [4:0]coutadd;
   wire [4:0]bsub;
   wire coutmult;
   wire remdiv;

   //Add
   assign coutadd[0] = 1'b0;
   genvar i;
   generate
      for (i=0; i<4; i = i + 1)
         begin : tortellini
            adder stage(x[i], y[i], coutadd[i], aadd[i], coutadd[i+1]);
            end
         endgenerate

   //Subtract
   assign bsub[0] = 1'b0;
   genvar j;
   generate
      for (j=0; j<4; j=j+1)
         begin : ravioli
            subtractor stage(x[j], y[j], bsub[j], asubtract[j], bsub[j+1]);
            end
         endgenerate

   //Multiply
   assign amultiply[7:0] = {y,x} << 1;
   assign coutmult = y[3];

   //Divide
   assign adivide[7:0] = {y,x} >> 1;
   assign remdiv = x[0];

   //Multiplexer for arithmetic operations and carries
   eightbitmux remi(aadd, asubtract, amultiply, adivide, op, z);
   onebitmux collete(coutadd[4], bsub[4], coutmult, remdiv, op, carry);

endmodule
```

adder.v

```
module adder( x, y, cin, z, cout);
   input x;
   input y;
   input cin;
   output cout;
   output z;

   //Ripple carry adder logical equations
   assign z = x ^ y ^ cin;
   assign cout = (x & y) | (x & cin) | (y & cin);
endmodule
```

subtractor.v

```
module subtractor ( x, y, bin, z, bout);
   input x;
   input y;
   input bin;
   output z;
   output bout;

   //Ripple borrow subtractor logical equations
   assign z = x ^ y ^ bin;
   assign bout = (~x & bin) | (~x & y) | (y & bin);
endmodule
```

logical.v

```
module logical(x, y, op, z);
   input [3:0]x;
   input[3:0]y;
   input [1:0]op;
   output [7:0]z;

   wire [7:0]land;
   wire [7:0]lor;
   wire [7:0]lxor;
   wire [7:0]lnot;

   //Concatenate 4 bits of zeros to operation output
   assign land [7:4] = 4'b0000;
   assign lor [7:4] = 4'b0000;
   assign lxor [7:4] = 4'b0000;

   //Logical operations
   assign land[3:0] = x[3:0] & y[3:0];
   assign lor[3:0] = x[3:0] | y[3:0];
   assign lxor[3:0] = x[3:0] ^ y[3:0];
   assign lnot = ~{y,x};

   //Multiplexer
   eightbitmux answer(land, lor,  lxor, lnot, op, z);
endmodule

```

Compare2.v

```
module Compare2 (x, y, op, z, equal, greater, less);
   input [3:0]x;
   input [3:0]y;
   input [1:0]op;
   output [7:0]z;

   //Procedural assignment requires a register
   output reg equal;
   output reg greater;
   output reg less;

   wire[7:0]cmax;
   wire [7:0]cequal;
   wire[7:0]cgreater;
   wire[7:0]cless;

   //Comparison module operations
   always @ (x, y)
   begin : linda
      if (x<y) begin
         equal = 0;
         greater = 0;
         less = 1;
      end
      else if (x>y) begin
         equal = 0;
         greater = 1;
         less = 0;
      end
      else begin
         equal = 1;
         greater = 0;
         less = 0;
      end
   end

   //Concatenate zeros on output to form an 8-bit number
   assign cequal[7:1] = 7'b0000000;
   assign cequal[0] = equal;

   assign cgreater[7:1] = 7'b0000000;
   assign cgreater[0] = greater;

   assign cless[7:1] = 7'b0000000;
   assign cless[0] = less;

   assign cmax[7:4] = 4'b0000;

   //Max output
   genvar i;
   generate
      for (i = 0 ; i<4 ; i = i+1)
      begin : pizza
         maxmux pizzasauce (x[i], y[i], greater, cmax[i]);
      end
   endgenerate

   //Multiplexer
   eightbitmux muxed(cequal, cgreater, cless, cmax, op, z);
endmodule
```

maxmux.v

```
module maxmux(x,y,sel,z);
   input x;
   input y;
   input sel;
   output z;

   //Multiplexer used to find max number
   assign z = (x & sel) | (y & ~sel);
endmodule
```