ECEN 2350 – Digital Logic

Gabriel Anhalzer and Heinz Boehmer

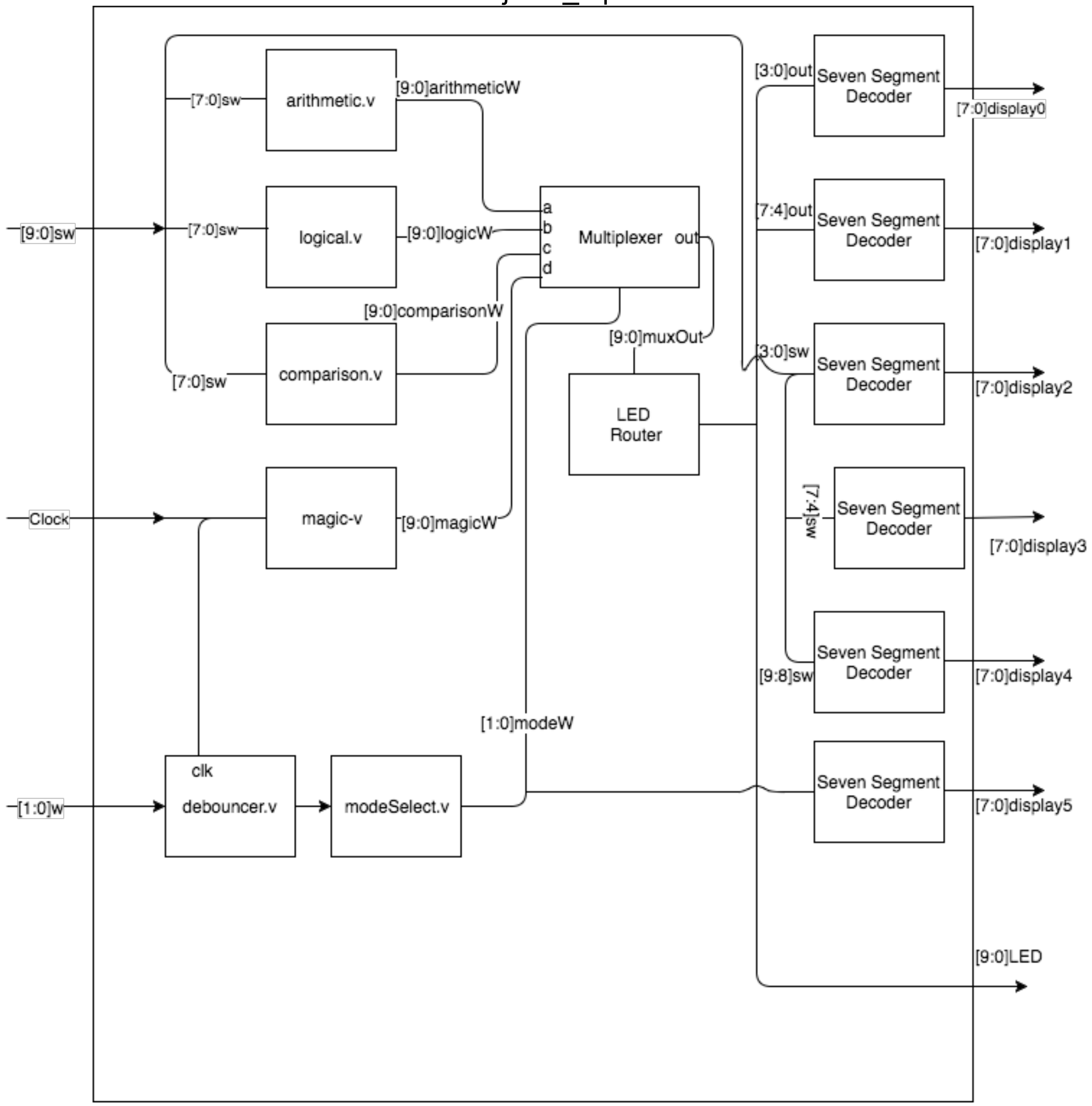
Due: Sunday March 5, 2017

Project 1

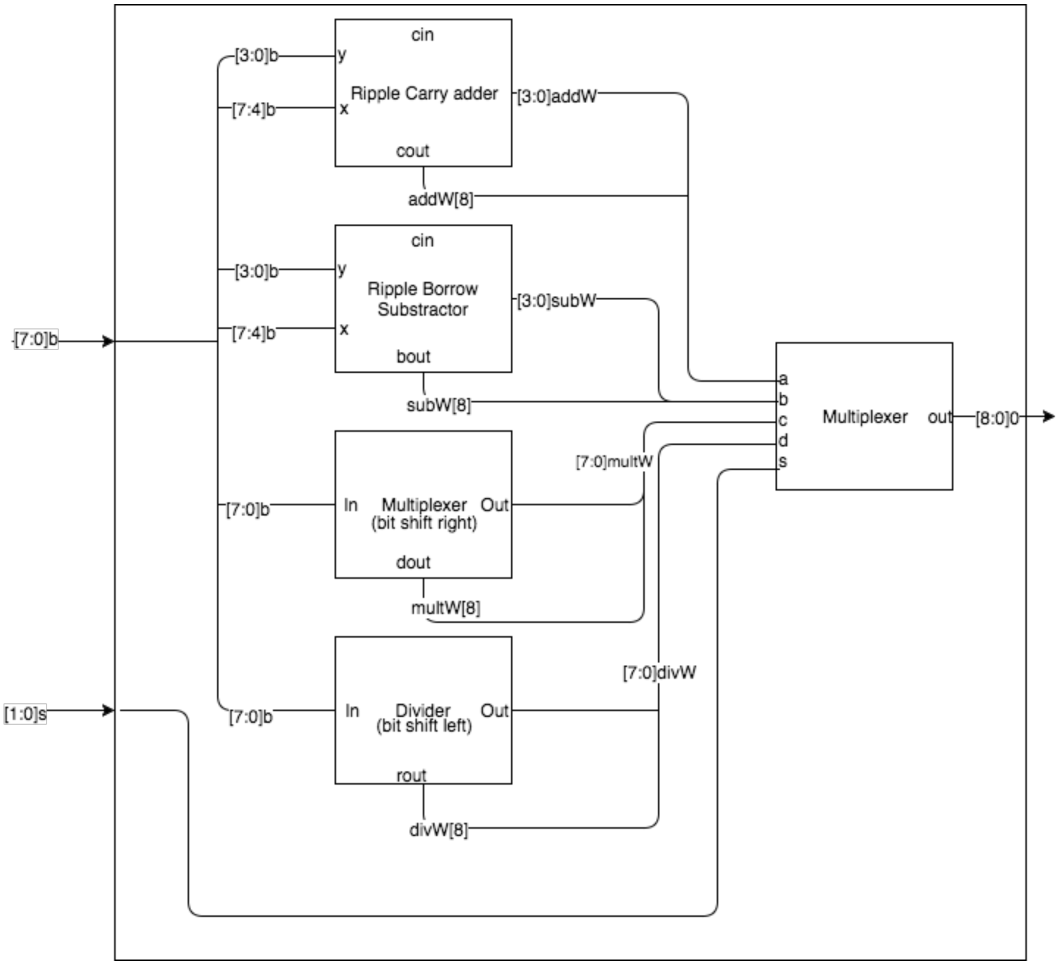
For our first project, we basically created an ALU (arithmetic logic unit). This is a hardware module designed specifically for performing arithmetic, bitwise, and logical operations on a set of inputs. To do this, we used a DE10-LITE board and Verilog. We used both switches and buttons as inputs and displayed out output on seven segment displays and LEDs.

Before starting to code, we made block diagrams for all the modules that we knew were going to be needed. Below are these diagrams.

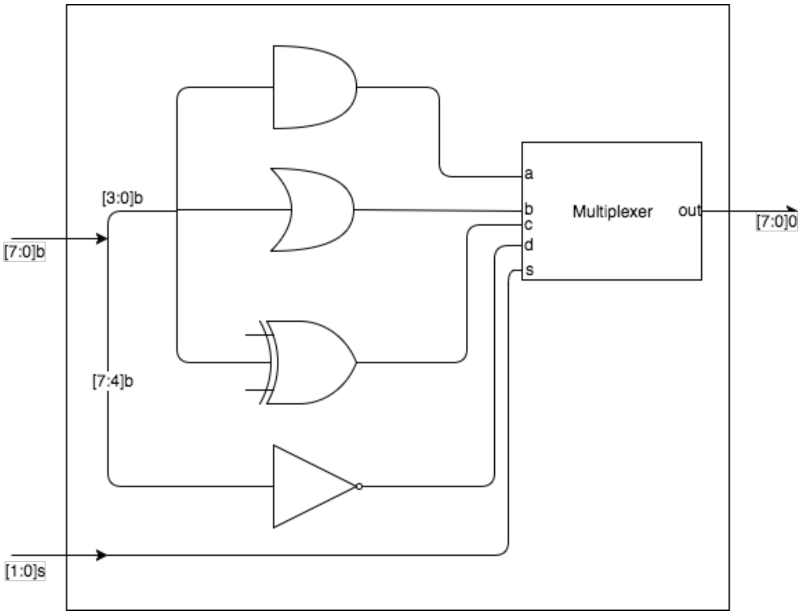
Project1_top.v



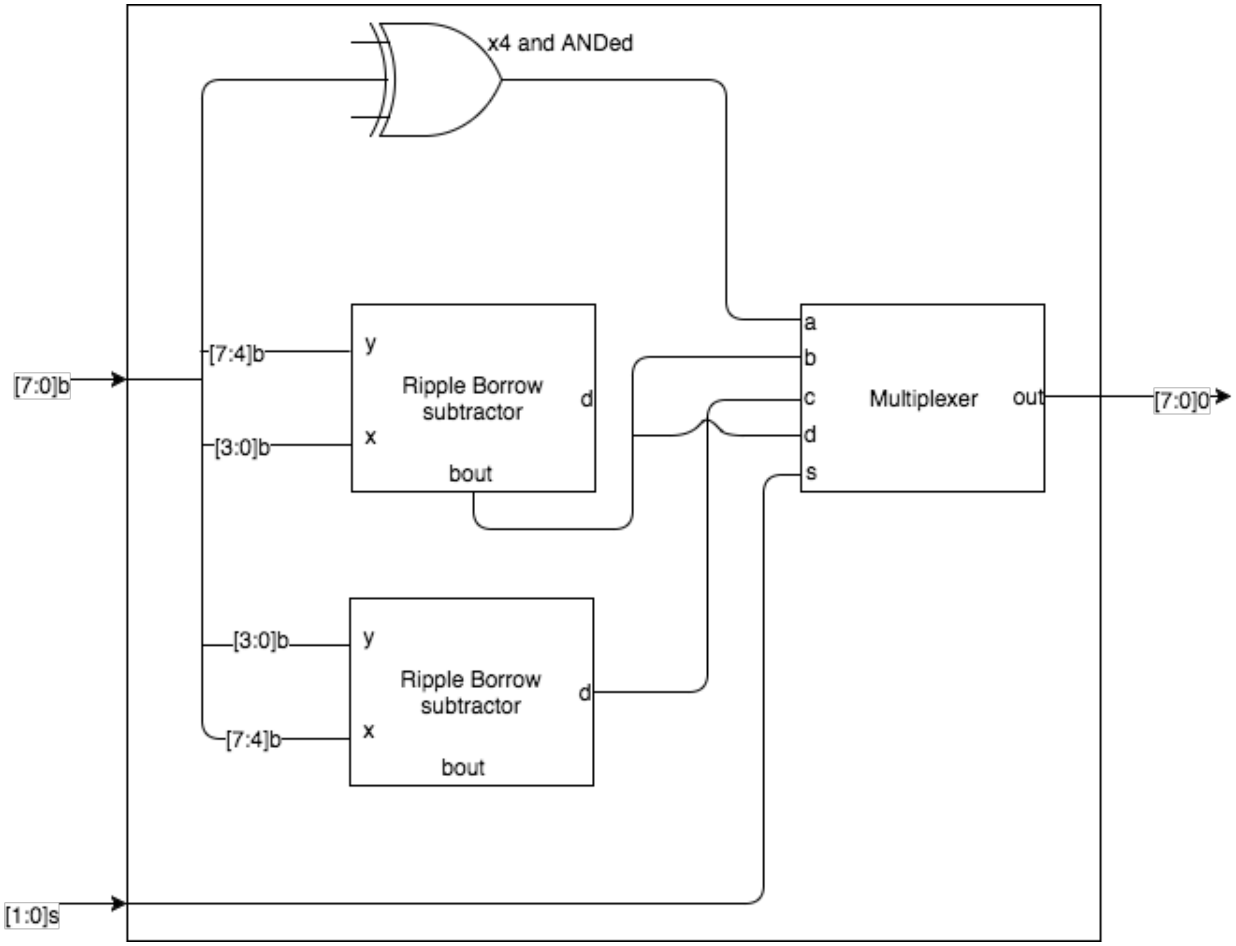
Arithmetic.v



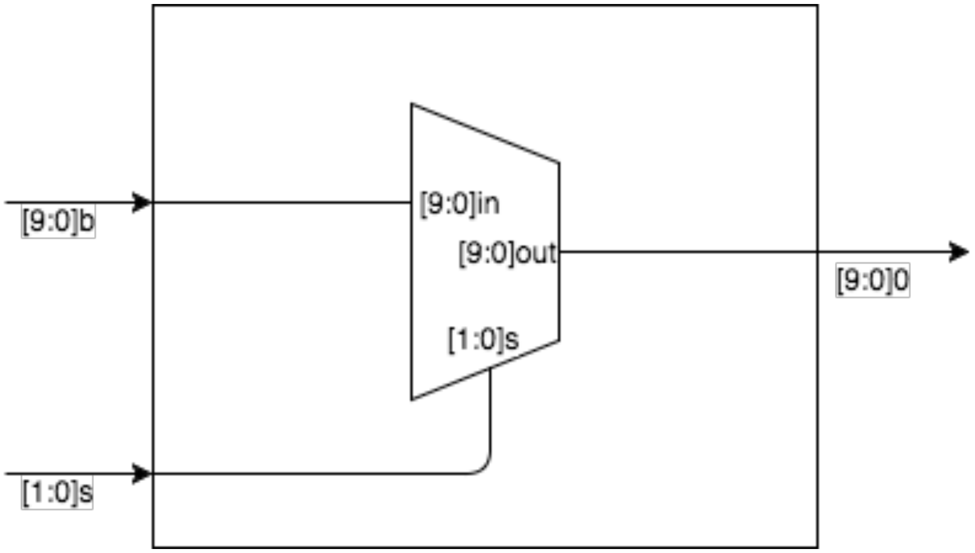
Logical.v



Comparison.v



Multiplexer.v



We decided to create a module for each of our four operations, so that their outputs could be switched by a multiplexer in our Project1_top.v file. The input for this module had to be provided by two buttons. We decided to use the left button to cycle between operations in one direction and the left to cycle in the opposite direction. This proved to be quite a challenge, as the buttons were not really debounced in software and we had to build a module to debounce them. This module takes in a clock signal as input and runs an always statement at each rising edge of the signal. This increments a reg and once this reaches 0hfffff, it evaluates the switch input and sets its output to the same value.

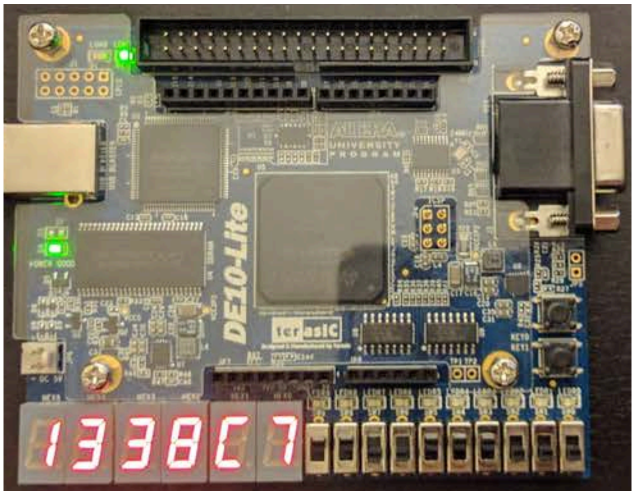
We also had to figure out a way to decide whether our output was going to the seven segment displays or to the LEDs (for the magic mode). We used a case statement for this that took the output from our mode selector module and set the output from our main multiplexer to the LEDs if magic mode had been selected and to the seven segment displays if it had not.

Taking both of these things into consideration, we designed an encoding scheme that allowed us to choose between the four different operations and their modes. This is what we came up with:

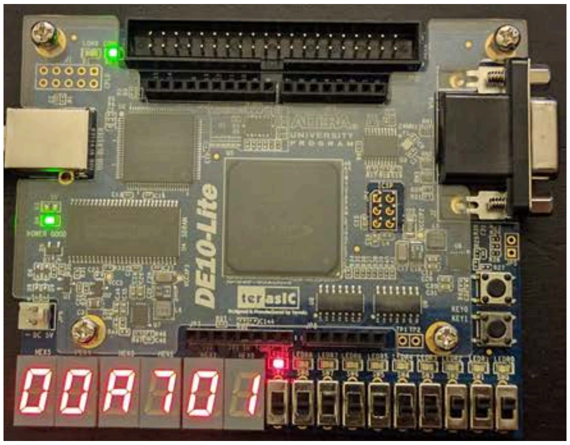
KEY0	KEY1	Operation
0	0	Arithmetic
0	1	Logical
1	0	Comparison
1	1	Magic

SW0	SW1	Mode (Depends on Operation)
0	0	Mode0
0	1	Mode1
1	0	Mode2
1	1	Mode3

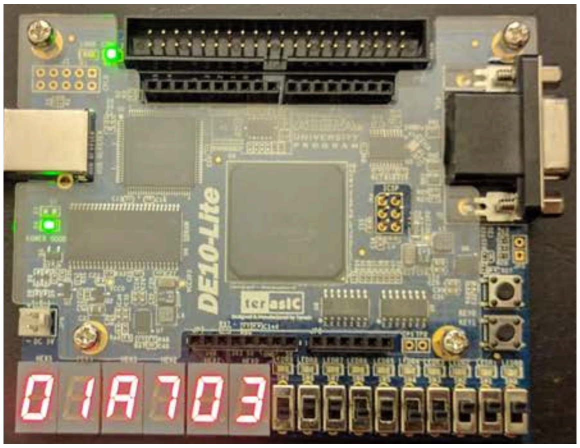
Additionally, we needed to decide on how to display which operation and mode we were in, as well as out inputs and outputs. We decided on using the two right-most seven segment displays for our outputs, the two middle ones for our inputs, the left-most one for the operation, and the second from the left for the mode. An example is shown below.



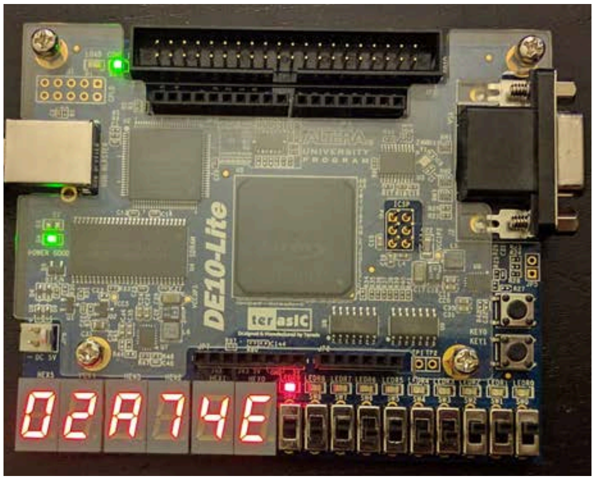
For our arithmetic module, we used ripple carry adders, ripple borrow subtractors, and bit-shifts. To start, we designed modules for both a full adder and a full subtractor. To add, we simply cascaded four full adders and used our switches as binary inputs. The carry out was outputted in combination with the sum and then these were displays on LED9 and the right-most seven segment display, respectively. A similar process was used for the difference. We cascaded four ripple borrow subtractors using the switches as inputs. Our difference and borrow out were also outputted in the same manner. Multiplication and division were rather straight forward, as they simply involved bit-shifts. We simply used several continuous assignments to route wires between the inputs and output for the shifts. At the end of the module, we included a multiplexer that used switches 8 and 9 as inputs to switch between the different outputs produced by each of the modes. Below are pictures of our module in its four different modes.



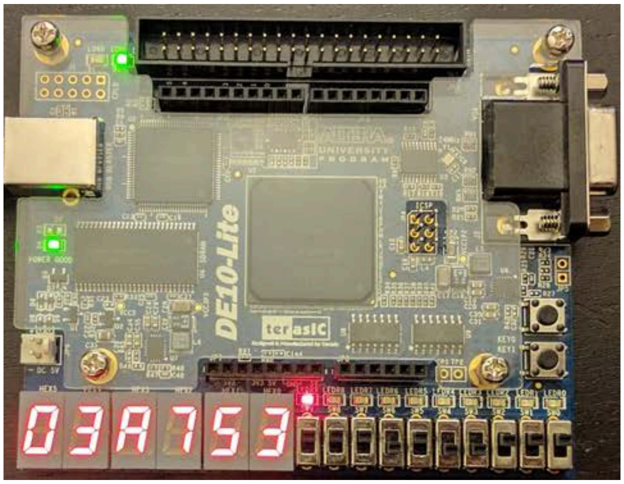
Addition



Subtraction

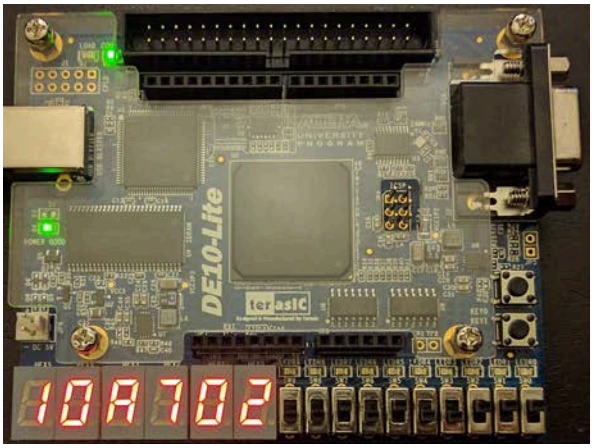


Multiplication

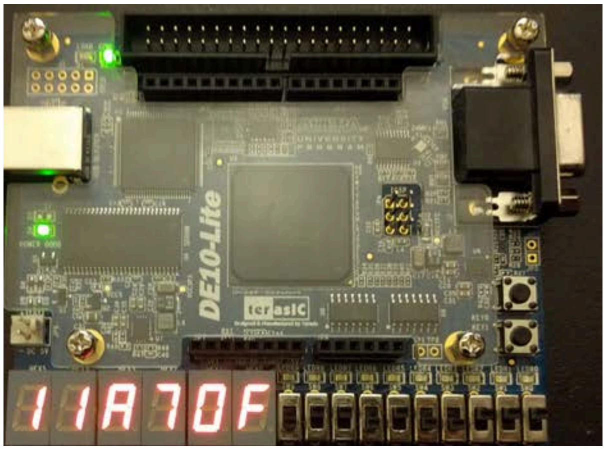


Division

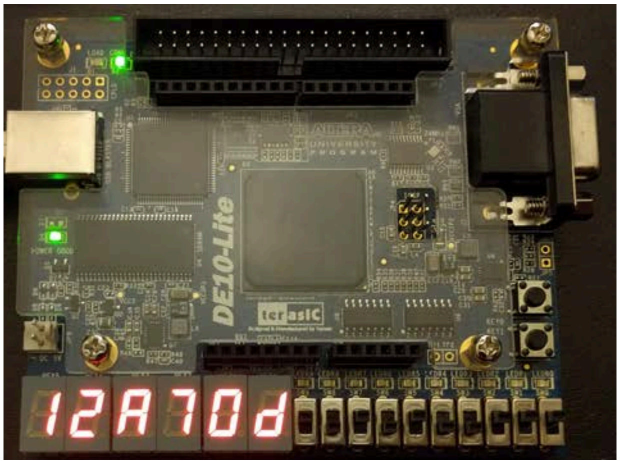
Our logical module was rather straight forward. We used continuous assignment to execute the logic. We simply assigned wire equals to the result of the logic operation on the inputs and ran these wires into a multiplexer in order to switch between modes. The following pictures show each of these four modes in action.



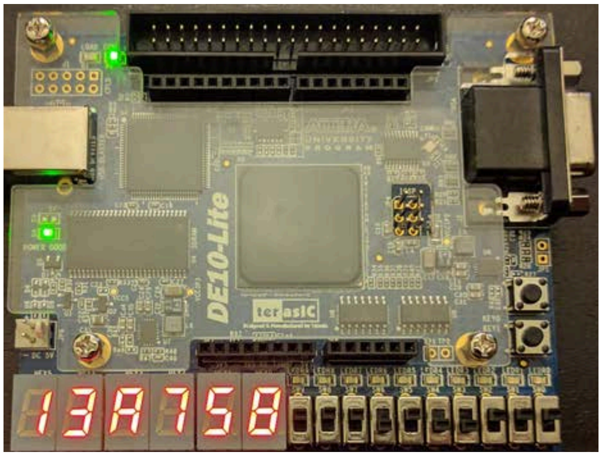
AND



OR



XOR

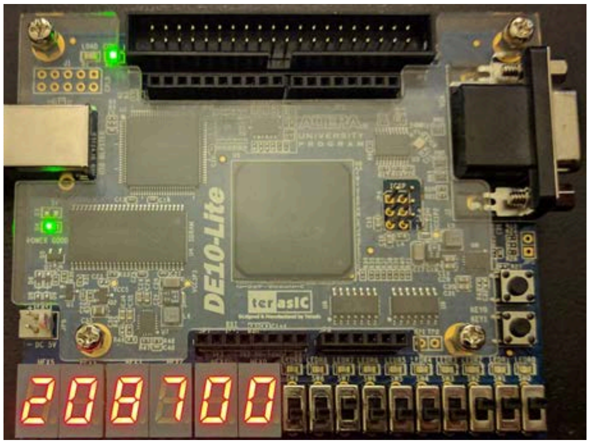


NOT

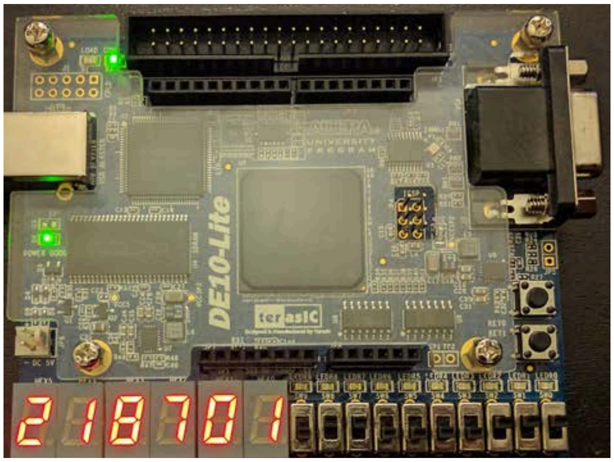
Although we could have used if statements combined with operations like <,>, and == for out comparison module, we decided against it, as we were not exactly sure how Verilog implemented these in hardware. Instead, we used our previously defined ripple carry subtractor to carry out or logic. Since this module produced a difference and a borrow out after performing x-y, we know that the carry will only be 1 if x>y. This allowed us to implement the greater than, less than, and max modes. However, for the max mode, we decided to display the number that was greater instead of displaying 1 is x was greater or 0 is y was greater. For the equal mode, we used a continuous assignment that ran each bit through an XNOR gate and then ran the output of these four gates into an AND gate, as shown below.

```
assign equalW=~(sw[7]^sw[3])&~(sw[6]^sw[2])&~(sw[5]^sw[1])&~(sw[4]^sw[0]);
```

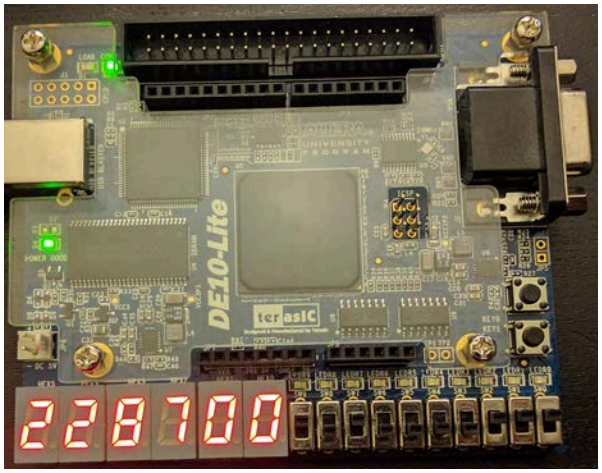
These for modes are shown in the images below.



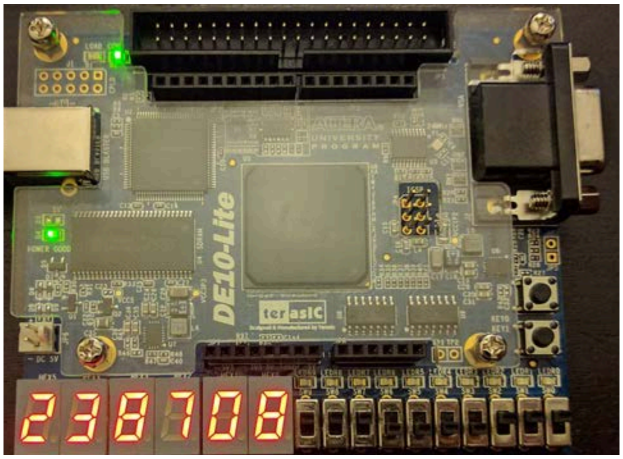
Equal



Greater

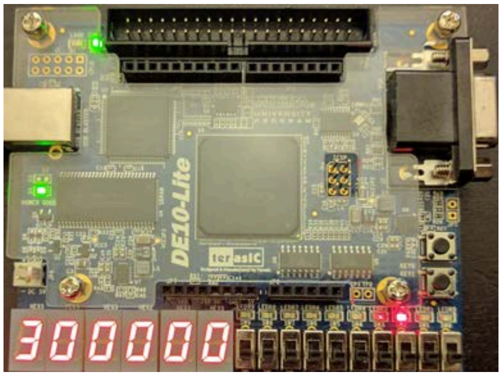


Less



Max

Since we had already figured out how to use the clock in order to debounce our switches, making the extra credit magic module was not as hard as it had seemed at first. We used a register to count up to 0xfffff on each rising clock edge and once this happened we incremented/decremented a counter. We then used a case statement to set our LEDs on and off. This gave the effect we were looking for and although it is not easy to convey in pictures, we have included one anyway. You can see that LED2 is completely lit and LED3 is not, indicating that had previously been lit and is in the process of being turned off.



This project definitely taught us a lot. It did not only teach us Verilog syntax, but also made us think in a way that was very different to the one we were used to. Digital signals are not assigned an order in which they should be evaluated, which is pretty hard to understand until you have spent hours trying to debug an issue related to this. This became quite apparent in the module that selected the operations. If statements don’t work the same way in Verilog as they do in c. Nonetheless, we managed to work out all of our bugs and create a working product. If we were to redo this project, we probably would have started with better block diagrams, as we found that our original ones lacked a lot of functionality. We forgot to include multiplexers and didn’t even think about the module for debouncing and the one for selecting operation. If we would have had the block diagrams we have now, we probably would have managed to fix our bugs a lot faster than we did.

Appendix

```
module
Project1_top(switch,button,clock,display0,display1,display2,display3,display4,display5,L
ED);

    input [9:0]switch;
    input [1:0]button;
    input clock;

    output reg [6:0]display0;
    output reg [6:0]display1;
    output reg [6:0]display2;
    output reg [6:0]display3;
    output reg [6:0]display4;
    output reg [6:0]display5;
    output reg [9:0]LED;

    wire [6:0]displayW0;
    wire [6:0]displayW1;
    wire [6:0]displayW2;
    wire [6:0]displayW3;
    wire [6:0]displayW4;
    wire [6:0]displayW5;
    wire [1:0]modeW;
    wire [9:0]muxOut;
    wire [9:0]arithmeticW;
    wire [9:0]logicW;
    wire [9:0]comparisonW;
    wire [9:0]magicW;

    modeSelector mode(~button,clock,modeW);
    multiplexer mux0(modeW,arithmeticW,logicW,comparisonW,magicW,muxOut);

    arithmetic a(switch,arithmeticW);
    logic l(switch,logicW);
    comparison s(switch, comparisonW);
    magic m(clock,magicW);

    decoder decoder0(muxOut[3:0],displayW0);
    decoder decoder1(muxOut[7:4],displayW1);
    decoder decoder2(switch[3:0],displayW2);
    decoder decoder3(switch[7:4],displayW3);
    decoder decoder4(switch[9:8],displayW4);
    decoder decoder5(modeW,displayW5);
    always@(modeW)
        begin

            case(modeW)

                2'b11:
                    begin
                        display0=7'b1000000;
                        display1=7'b1000000;
                        display2=7'b1000000;
                        display3=7'b1000000;
                        display4=7'b1000000;
                        display5=displayW5;
                        LED=muxOut;
                    end

                default:
                    begin
                        display0=displayW0;
                        display1=displayW1;
                        display2=displayW2;
                        display3=displayW3;
                        display4=displayW4;
                        display5=displayW5;
                        LED[8:0]=9'b000000000;
                        LED[9]=muxOut[8];
                    end

                endcase

        end

endmodule


module fullAdder(cinF,xF,yF,sF,coutF);

        input cinF,xF,yF;
        output reg sF;
        output reg coutF;

        always@(cinF,xF,yF)
            begin
                coutF=(xF&yF)|(yF&cinF)|(xF&cinF);
                sF=(xF)^(yF)^(cinF);
            end
```

```
endmodule


module rippleCarryAdder(cin,x,y,s,cout);

    parameter n=4;
    input cin;
    input [n-1:0]x;
    input [n-1:0]y;
    output [n-1:0]s;
    output cout;
    wire [n:0]cW;
    wire [n-1:0]sW;

    genvar i;

    generate
        for (i=0; i<n; i=i+1)
            begin:
                adderLoop
                fullAdder stage(cW[i],x[i],y[i],sW[i],cW[i+1]);
            end
    endgenerate

    assign cW[0]=cin;
    assign cout=cW[n];
    assign s=sW;

endmodule


module decoder(b,d);

    input [3:0]b;
    output reg [6:0]d;

    always@(b)
        begin
            case (b)
                4'b0000 : d=~7'b0111111;  //0
                4'b0001 : d=~7'b0000110;  //1
                4'b0010 : d=~7'b1011011;  //2
                4'b0011 : d=~7'b1001111;  //3
                4'b0100 : d=~7'b1100110;  //4
                4'b0101 : d=~7'b1101101;  //5
                4'b0110 : d=~7'b1111101;  //6
                4'b0111 : d=~7'b0000111;  //7
                4'b1000 : d=~7'b1111111;  //8
                4'b1001 : d=~7'b1101111;  //9
                4'b1010 : d=~7'b1110111;  //A
                4'b1011 : d=~7'b1111100;  //B
                4'b1100 : d=~7'b0111001;  //C
                4'b1101 : d=~7'b1011110;  //D
                4'b1110 : d=~7'b1111001;  //E
                4'b1111 : d=~7'b1110001;  //F
            endcase
        end

endmodule


module multiplexer(s,a,b,c,d,f);

    input [1:0]s;
    input [9:0]a;
    input [9:0]b;
    input [9:0]c;
    input [9:0]d;
    output reg [9:0]f;

    always@(s)
        begin
            case (s)
                2'b00 : f=a;    //0
                2'b01 : f=b;    //1
                2'b10 : f=c;    //2
                2'b11 : f=d;    //3
            endcase
        end

endmodule


module fullSubtractor(binF,xF,yF,dF,boutF);

        input binF,xF,yF;
        output reg dF;
        output reg boutF;

        always@(binF,xF,yF)
            begin
```

```
                                boutF=(~xF&binF)|(~xF&yF)|(yF&binF);
                                dF=(xF)^(yF)^(binF);
                        end

endmodule


module rippleBorrowSubtractor(bin,x,y,d,bout);

        parameter n=4;
        input bin;
        input [n-1:0]x;
        input [n-1:0]y;
        output [n-1:0]d;
        output bout;
        wire [n:0]bW;
        wire [n-1:0]dW;

        genvar i;

        generate
                for (i=0; i<n; i=i+1)
                        begin:
                                subtractorLoop
                                fullSubtractor stage(bW[i],x[i],y[i],dW[i],bW[i+1]);
                        end
        endgenerate

        assign bW[0]=bin;
        assign bout=bW[n];
        assign d=dW;

endmodule


module multiplier(x,p,cout);

        input [7:0]x;
        output [7:0]p, cout;

        assign p[0]=0;
        assign p[1]=x[0];
        assign p[2]=x[1];
        assign p[3]=x[2];
        assign p[4]=x[3];
        assign p[5]=x[4];
```

```
    assign p[6]=x[5];
    assign p[7]=x[6];
    assign cout=x[7];


endmodule


module modeSelector(sw,clk,select);

    input [1:0]sw;
    input clk;
    wire [1:0]swW;
    output reg [1:0]select;

    debounce d(clk,sw,swW);

    always@(posedge swW[0],posedge swW[1])
        begin

            if(swW[0])
                begin

                    select=select-1;

                    if(select<1)
                        select=4;

                end


            else
                begin

                    select=select+1;

                    if(select>4)
                        select=1;

                end

        end

endmodule
```

```
module divider(x,d,bout);

    input [7:0]x;
    output [7:0]d, bout;

    assign d[0]=x[1];
    assign d[1]=x[2];
    assign d[2]=x[3];
    assign d[3]=x[4];
    assign d[4]=x[5];
    assign d[5]=x[6];
    assign d[6]=x[7];
    assign d[7]=0;
    assign bout=x[0];


endmodule


module debounce(clk,bounced,debounced);

    input [1:0]bounced;
    input clk;
    reg [19:0]i;
    reg [1:0]debouncedW;
    output [1:0]debounced;

    always@(posedge clk)
        begin

            i=i+1;

            if(i==20'hfffff)
                debouncedW=bounced;

        end

    assign debounced=debouncedW;

endmodule


module arithmetic(sw,result);

    input [9:0]sw;
    output [8:0]result;
```

```
        wire [8:0]addW;
        wire [8:0]subW;
        wire [8:0]mulW;
        wire [8:0]divW;

        multiplexer muxArithmetic(sw[9:8],addW,subW,mulW,divW,result);

        rippleCarryAdder add(0,sw[7:4],sw[3:0],addW[3:0],addW[8]);
        rippleBorrowSubtractor sub(0,sw[7:4],sw[3:0],subW[3:0],subW[8]);
        multiplier mul(sw[7:0],mulW[7:0],mulW[8]);
        divider div(sw[7:0],divW[7:0],divW[8]);

endmodule


module logic(sw,result);

        input [9:0]sw;
        output [7:0]result;

        wire [8:0]andW;
        wire [8:0]orW;
        wire [8:0]xorW;
        wire [8:0]notW;

        multiplexer logic(sw[9:8],andW,orW,xorW,notW,result);

        assign andW=(sw[7:4])&(sw[3:0]);
        assign orW=(sw[7:4])|(sw[3:0]);
        assign xorW=(sw[7:4])^(sw[3:0]);
        assign notW=~sw[7:0];

endmodule


module comparison(sw,result);

        input [9:0]sw;
        output [7:0]result;

        wire equalW;
        wire greaterW;
        wire lessW;
        reg [3:0]maxW;
        wire [3:0]temp;
```

```
    multiplexer logic(sw[9:8],equalW,greaterW,lessW,maxW,result);

    assign equalW=~(sw[7]^sw[3])&~(sw[6]^sw[2])&~(sw[5]^sw[1])&~(sw[4]^sw[0]);
    rippleBorrowSubtractor subGreater(0,sw[3:0],sw[7:4],temp[3:0],greaterW);
    rippleBorrowSubtractor subLess(0,sw[7:4],sw[3:0],temp[3:0],lessW);
    always@(sw[7:0])
        begin

            if(greaterW)
                maxW=sw[7:4];

            else
                maxW=sw[3:0];
        end



endmodule


module magic(clk,LEDs);

    input clk;
    reg [19:0]i;
    reg [4:0]count;
    reg backwards;
    output reg [9:0]LEDs;

    always@(posedge clk)
        begin

            i=i+1;

            if(i==20'hfffff && backwards==0)
                    count=count+1;

            if(i==20'hfffff && backwards==1)
                    count=count-1;

            if(count==9)
                backwards=1;

            if(count==0)
                backwards=0;
                case(count)

                        0: LEDs=10'b0000000001;
                        1: LEDs=10'b0000000010;
                        2: LEDs=10'b0000000100;
                        3: LEDs=10'b0000001000;
                        4: LEDs=10'b0000010000;
                        5: LEDs=10'b0000100000;
                        6: LEDs=10'b0001000000;
                        7: LEDs=10'b0010000000;
                        8: LEDs=10'b0100000000;
                        9: LEDs=10'b1000000000;

                endcase


        end

endmodule
```