Abem Lucas & Dylan Thornburg & Alan Rieger
ELEN 21L (59602)
5/16/23

Lab 6: Mini-Calculator with a small 4-bit Arithmetic Logic Unit (ALU)

Introduction:

For this lab, we designed and tested a 4-bit arithmetic logic unit (ALU) with adder/subtractor functions through the Verilog code that included a MUX hierarchy. Then we were able to use our ALU in a mini-calculator with one memory location on the FPGA board.

Procedures:

For Part 1, we designed and tested the ALU by compiling the Verilog code for myALU4. We used Y[0] as the simulator output which operated on the least significant bit only and the inputs used were A[0], B[0], P[2], and P[1]. After that, we checked if the arithmetic operation was correct by doing some of the test plans in our prelab. We demoed our operations once the Test plan worked.

For Part 2, we added a memory save to the mini-calculator by adding a 4-bit wide 2:1 MUX component and an 8dff component for the memory location. We followed the specific configurations/instructions for the setups for both components and connected the input to the MUX output. Finally, we continued with more testing as stated in the lab document (10-14).

Results:

Figure 1a: First Schematic for Part 1

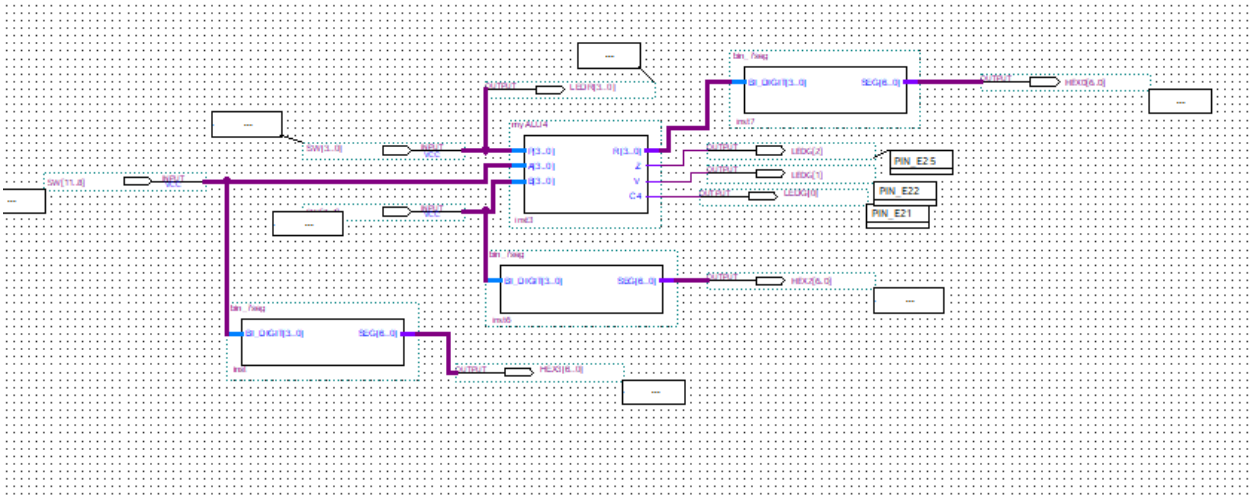


Figure 1b: Second Schematic for Part 2

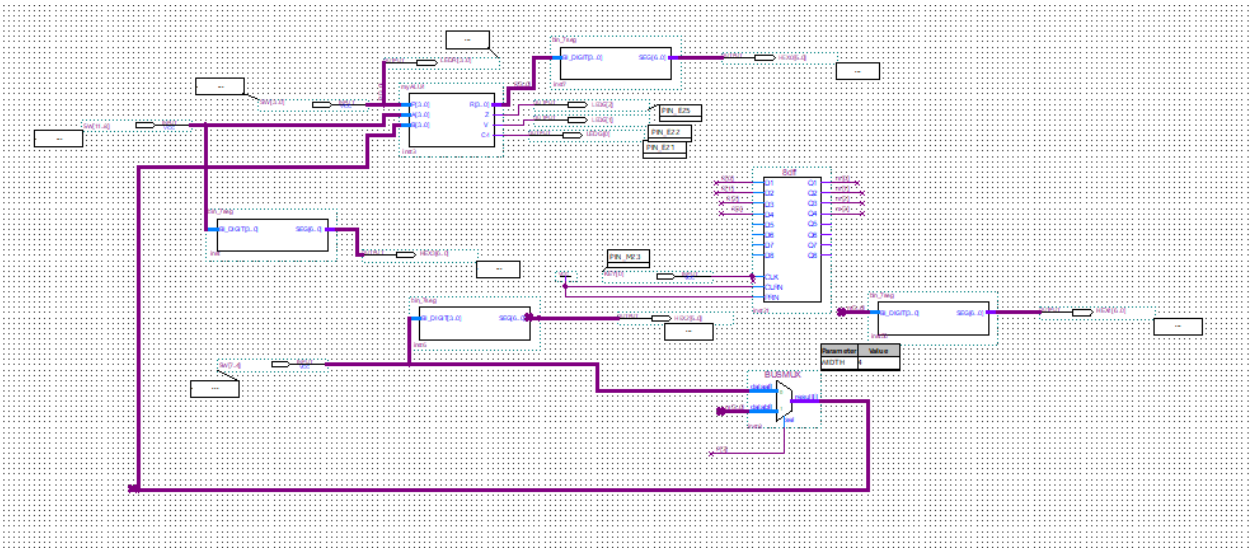


Figure 2: Verilog Code

```
module myfulladd(a,b,cin,sout,cout);
        input a,b,cin;
        output sout,cout;
        assign sout = a ^ b ^ cin;
        assign cout = (a & b) | (b & cin) | (a & cin);
endmodule
module myadder4(X,Y,c0,S,v,c4);
        input [3:0]X,Y;
        input c0;
        output[3:0]S;
```

```
        output v,c4;
        wire [3:1]C;
        myfulladd(X[0],Y[0],c0,S[0],C[1]);
        myfulladd(X[1],Y[1],C[1],S[1],C[2]);
        myfulladd(X[2],Y[2],C[2],S[2],C[3]);
        myfulladd(X[3],Y[3],C[3],S[3],c4);
        assign v = C[3] ^ c4;
endmodule
module myALU4(P,A,B,R,Z,V,C4);
        // This declaration section has been completed for you.
        // For simplicity, an actual C0 input is not needed,
        // as C0 is always the same as one of the P bits...
        input [3:0]P,A,B;
        output [3:0]R;
        output Z,V,C4;
        reg [3:0]Y; // Use reg instead of wire for always blocks.
        always @(*)
        // The case statement serves as a 4:1 MUX with P[2]
        // and P[1] select signals. Note that the notation
        // 2'b means a 2-bit binary value, so the first
        // case 2'b00 corresponds to when P[2]=0 and P[1]=0.
        // Complete the lines for the first and third cases.
        case(P[2:1])
                2'b00: Y = B;
                2'b01: Y = ~B;
                2'b10: Y = 4'b0000;
                2'b11: Y = 4'b1111;
        endcase
        // The below instance of myadder4 assumes the order
        // (X,Y,C0,S,V,C4), so you may have to change the
        // ordering to work with your myadder4 from lab 5.
        // Fill in the bit of P that should serve as C0.
        myadder4 U1(A,Y,P[0],R,V,C4);
        // Complete the assign statement to make Z=1 when R=0000.
        assign Z = (~P[0] & ~P[1] & ~P[2] & ~P[3]);
endmodule
module bin_7seg(BI_DIGIT,SEG);
  input [3:0] BI_DIGIT;
  output [6:0] SEG;
```

```
  reg [6:0] SEG;

  // seg = {g,f,e,d,c,b,a};
  // ---a----
  // |     |
  // f     b
  // |     |
  // ---g----
  // |     |
  // e     c
  // |     |
  // ---d----

  always @(BI_DIGIT)
    case (BI_DIGIT)
      4'h0: SEG = ~7'b0111111;
      4'h1: SEG = ~7'b0000110;
      4'h2: SEG = ~7'b1011011;
      4'h3: SEG = ~7'b1001111;
      4'h4: SEG = ~7'b1100110;
      4'h5: SEG = ~7'b1101101;
      4'h6: SEG = ~7'b1111101;
      4'h7: SEG = ~7'b0000111;
      4'h8: SEG = ~7'b1111111;
      4'h9: SEG = ~7'b1100111;
      4'ha: SEG = ~7'b1110111;
      4'hb: SEG = ~7'b1111100;
      4'hc: SEG = ~7'b1011000;
      4'hd: SEG = ~7'b1011110;
      4'he: SEG = ~7'b1111001;
      4'hf: SEG = ~7'b1110001;
    endcase
endmodule
```

Figure 3a: Successful Upload for Part 1

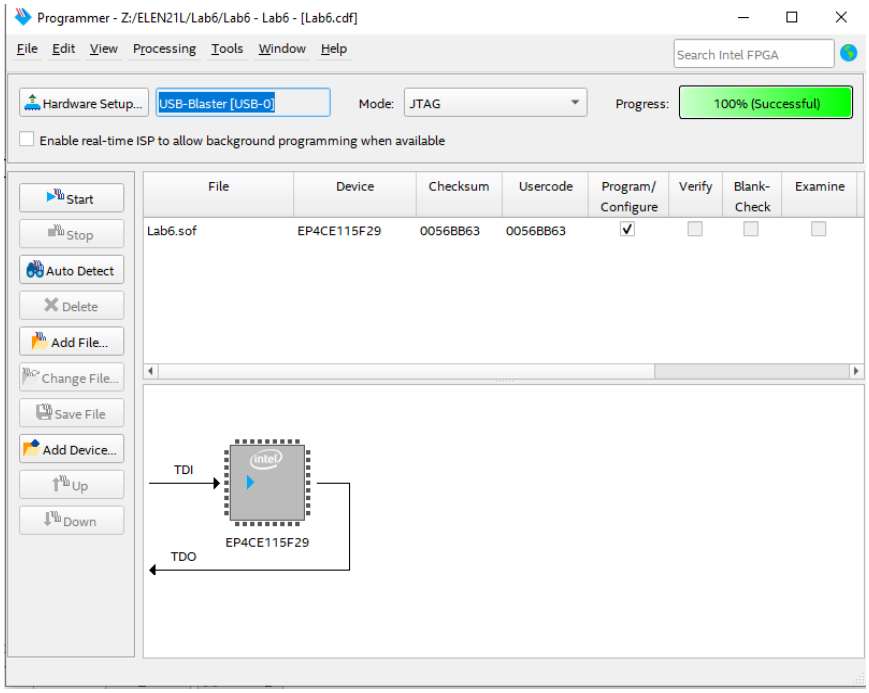


Figure 3b: Successful Upload for Part 2

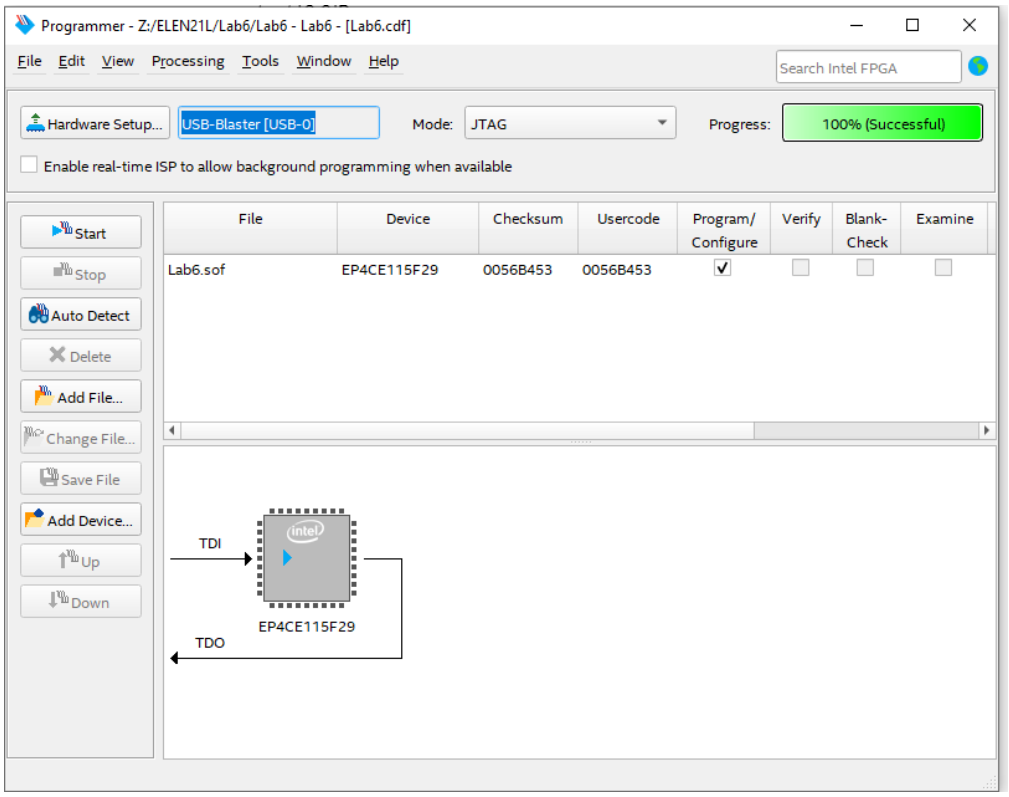


Figure 4: Test Plan

In regard to the Test Plan, we followed the table under the problem statement in the lab document for the demo that we did in Part 1. Ideally, we tested different combinations of A and B for different Op-Select Inputs in order to make sure the respective output R is correct for the correct operation. For example, Figure 5 includes an iteration of 0001 for the Op-Select Input for values A=1 and B=8, which resulted in a corresponding Output R which was, A + B + 1, or 1+8+1 = A.

Figure 5a: Results of Operation from Part 1

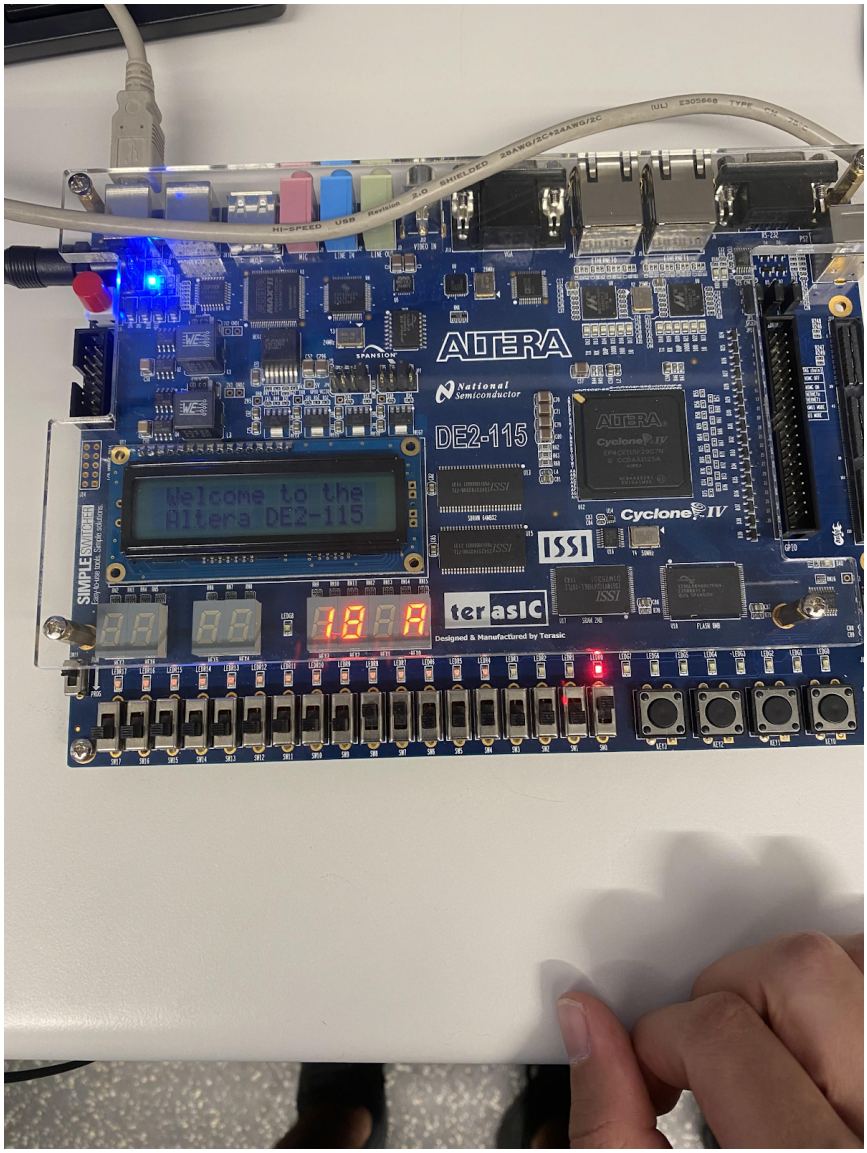
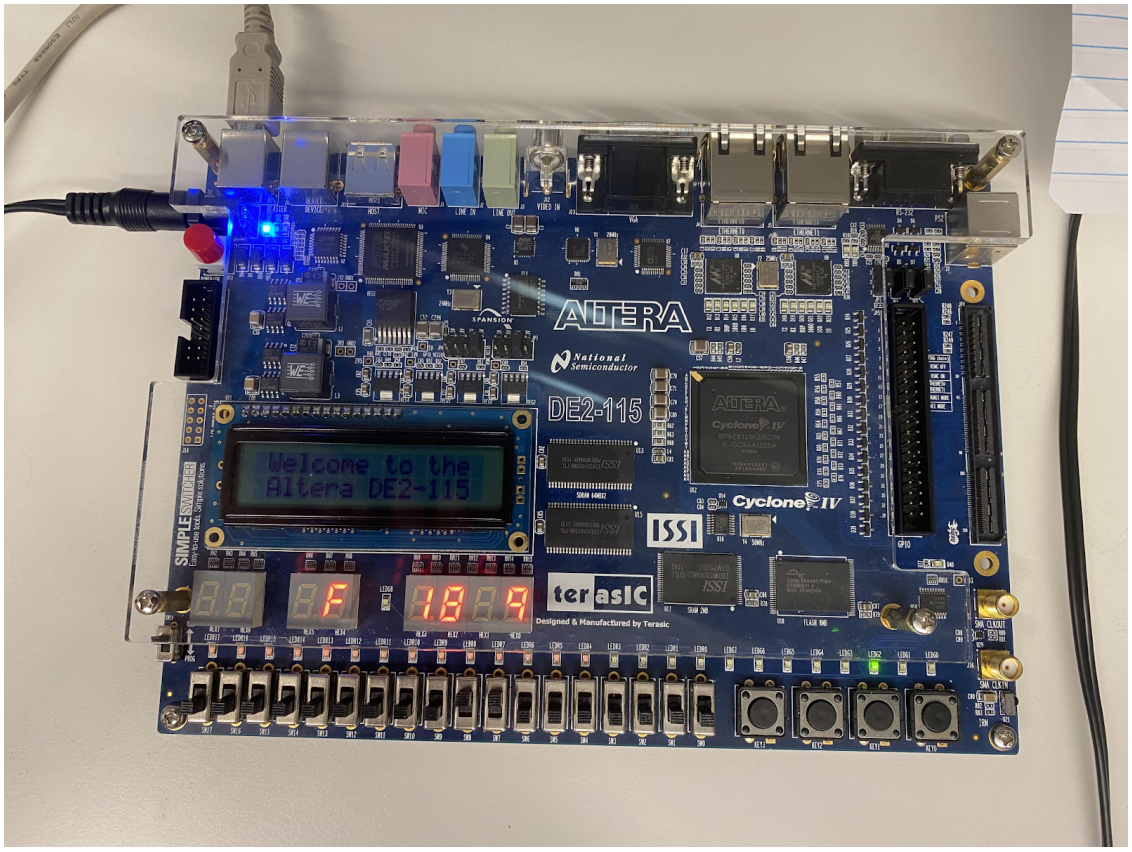


Figure 5b: Results of Operation from Part 2



Questions:

4. List three pairs of A, B inputs that will cause the Z output to be 1 when the operation is P = 0 0 0 0 and C0 is 0.

- A= 1000 ,B=1000
- A= 0111 ,B= 1001
- A= 0110 ,B= 1010

5. List three pairs of A, B inputs that will cause the Z output to be 1 when the operation is P = 0 0 1 1 and C0 is 0.

- A= 1000 ,B= 1000
- A= 0111 ,B= 0111
- A= 0110 ,B= 0110

7. If the B input values are zero, describe a sequence of operations using memory that you could use to get a mini-calculator output of –A.

I could put zero in memory, flip p[3], and flip p[1:0] to make it subtraction, and this will result in 0 - A which = -A

8. How would you connect two 4-bit ALUs to make an 8-bit ALU?

I would use the two four-bit ALUs and get their respective outputs, but I would only need the one 8-bit flip flop.

Conclusion:

Through this lab, we learned the fundamentals of how logic and arithmetic equations are actually performed through gates and how memory is utilized and stored through flip-flops. We understood before the lab that flip-flops could store data, but we hadn't seen it work in person before. Obviously, we also learned how to make a functional ALU with simple functions and how to create new functions as well.