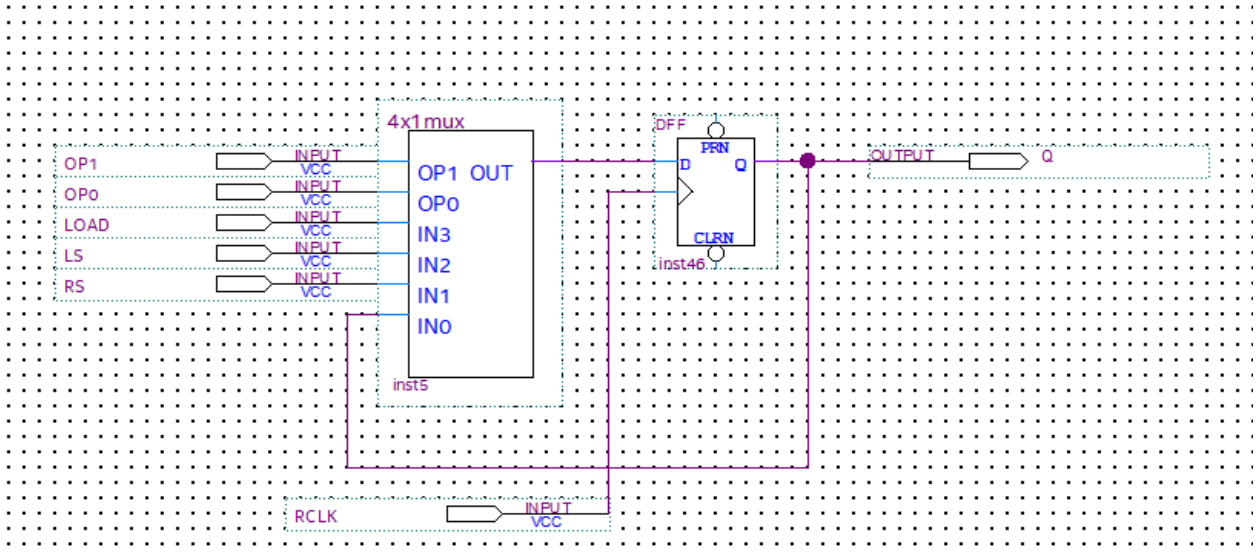
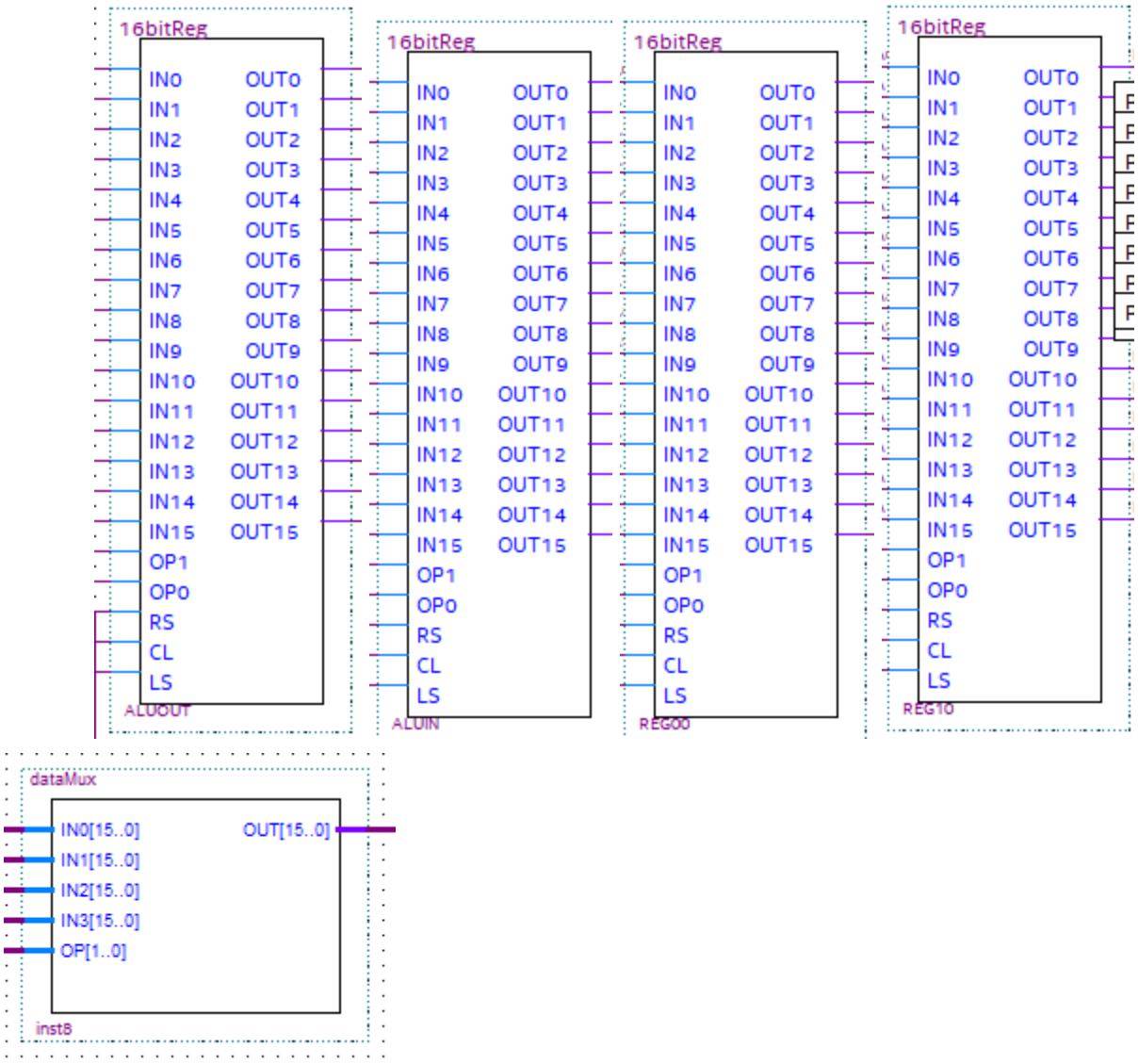
CprE281 Final Project

Calculator: Braydon Clay & Joey Banowetz

Our original idea was to make a calculator that takes in two 8-bit numbers stored in 2's compliment, and can perform one of the 5 operations: add, subtract, multiply, divide, and exponentiate. We used the 7 segment decoders to show the numbers in hexadecimal, but showing a negative sign if the user implements a negative number. To add another cool feature to "stand out", we came up with the idea to be able to store a number into an 8 bit register, and to "pull" that number to be reused later (very similar to how modern calculators can do "Ans + x"). We started out first by building a universal 16 bit register, which uses inputs given as OP0 and OP1 to determine the action to be taken. By then linking 16 of these together, we had our 16 bit register.



We used 4 of these registers total. 2 of these registers were used for storage, and the other 2 were used for the input and output of the entire ALU. We then took all the output of these registers, and fed them into one multiplexer, being our dataMux, with the output from dataMux being the DATA bus line that gave the input throughout the ALU.



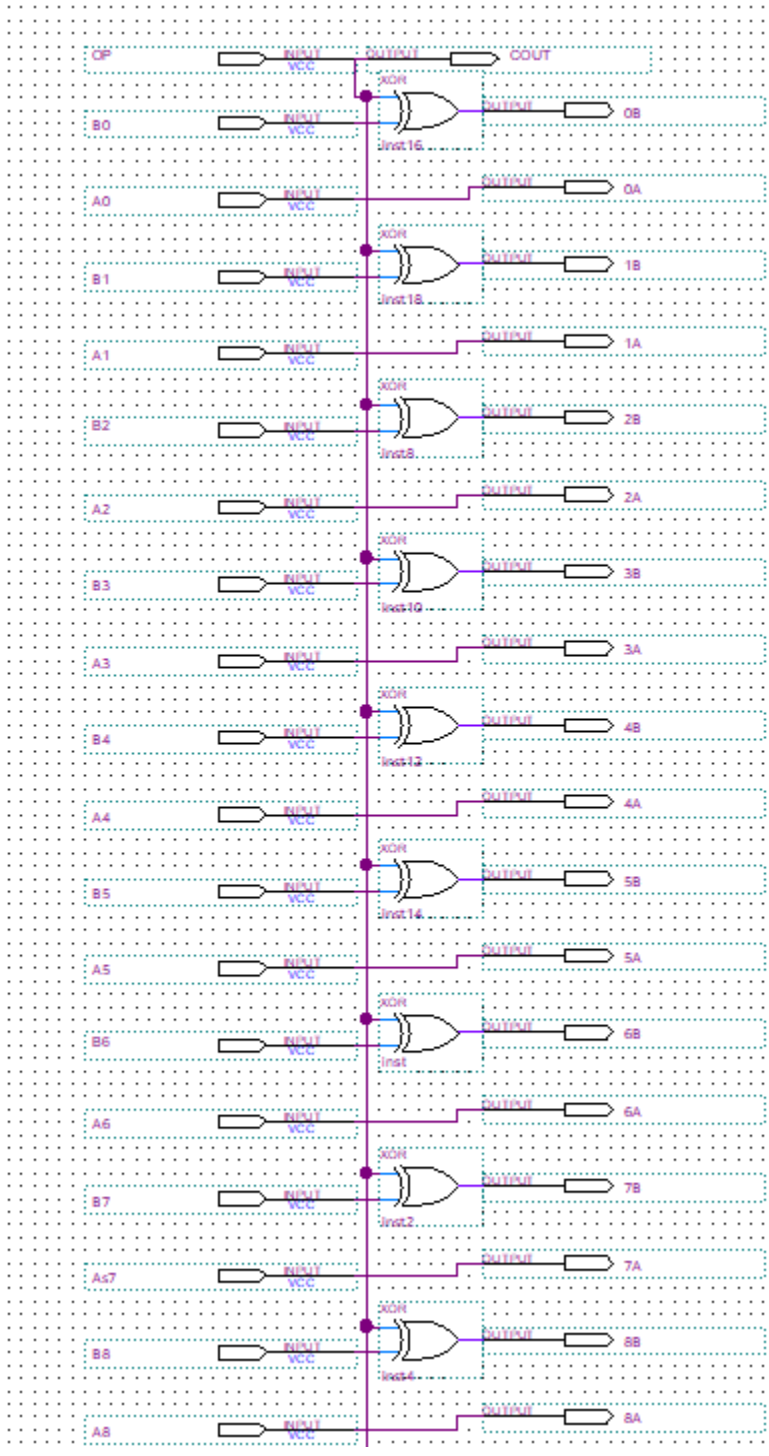
module dataMux (input [15:0]IN0, input [15:0]IN1, input [15:0]IN2, input [15:0]IN3, output [15:0]OUT, input [1:0]OP);

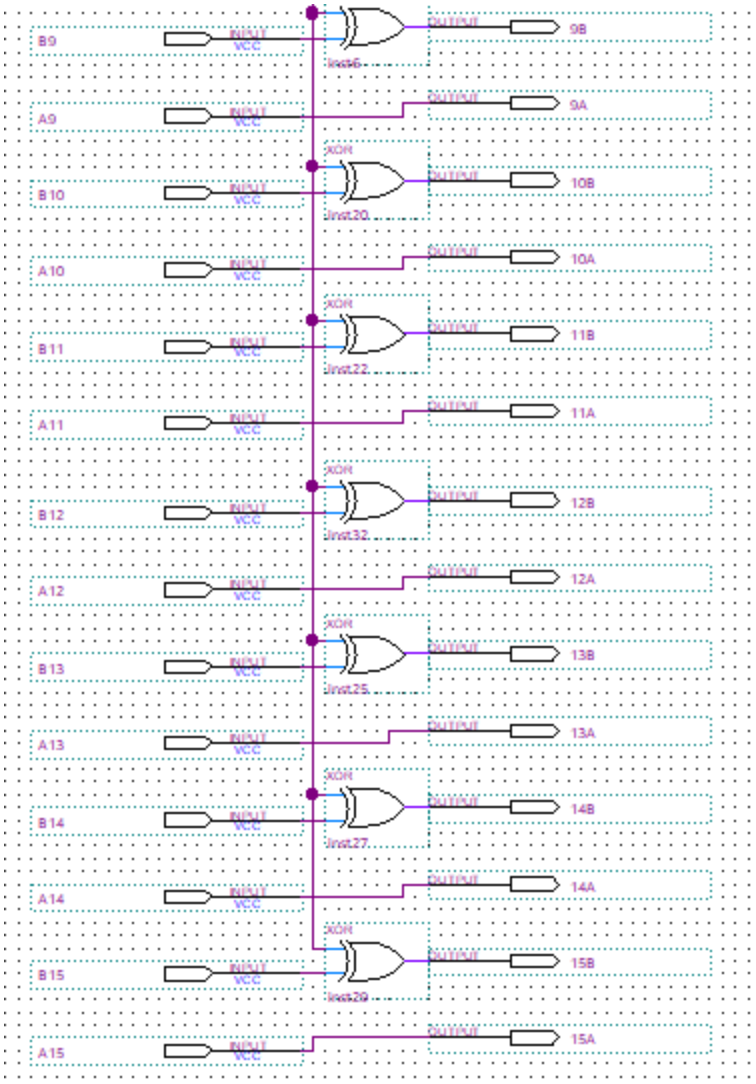
assign OUT = OP[1] ? (OP[0] ? IN3 : IN2) : (OP[0] ? IN1 : IN0);
endmodule

Unfortunately after some testing, we realized that it was not possible to implement multiplication, division, and exponential functions, due to the limitations of the board speed. We realized that for all those operations, it would likely take multiple minutes to complete the operation, and it would not be feasible with this limitation.

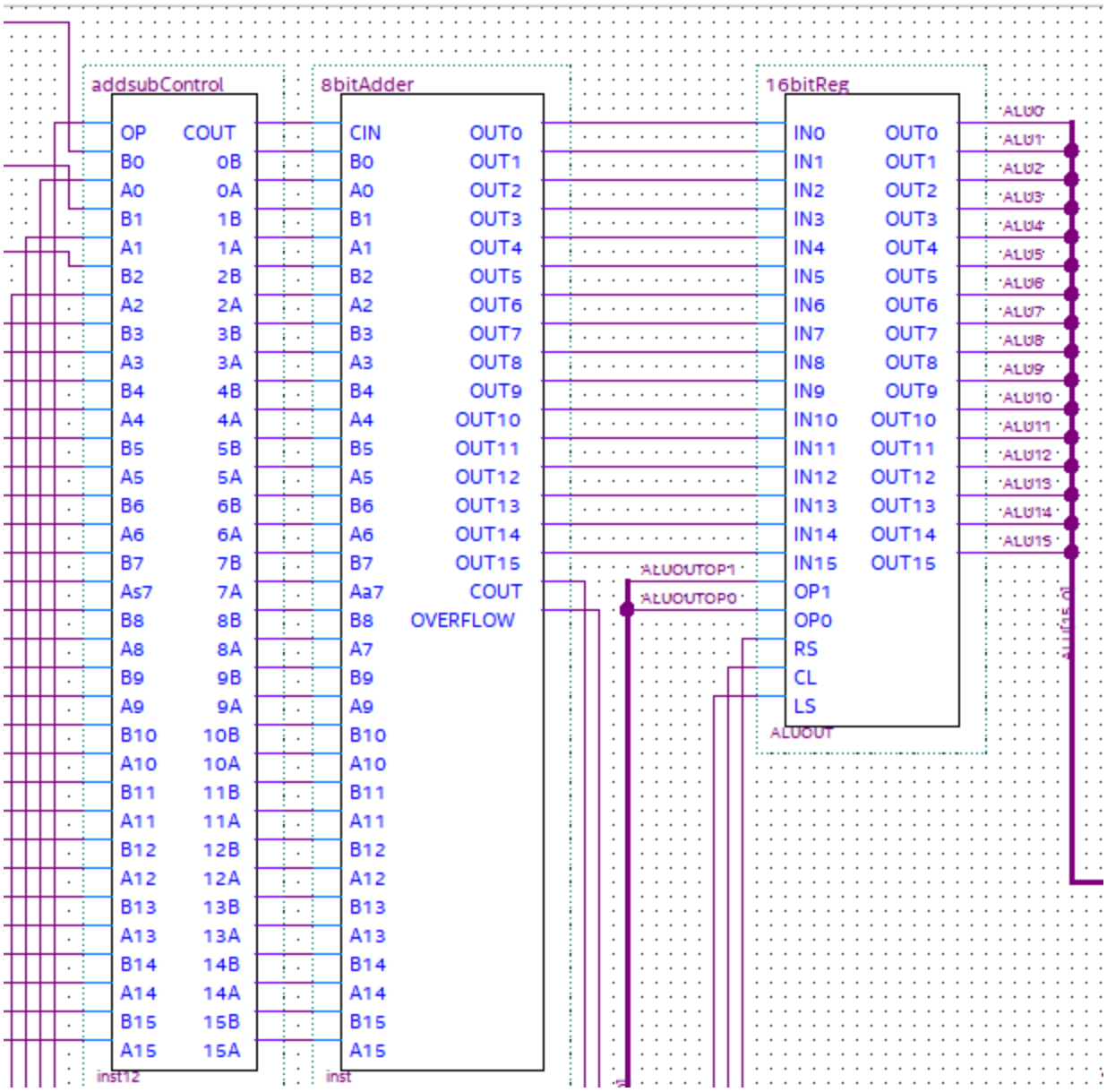
For our addition and subtraction, we took the data provided from the dataMux, and ran it into a large operation controller. This is because, if the input OP is 1, it will then use XOR gates to generate the two's complement of the negative number, to then be added. It goes up to 16 bits,

as we would also use this for multiplication had we implemented it.

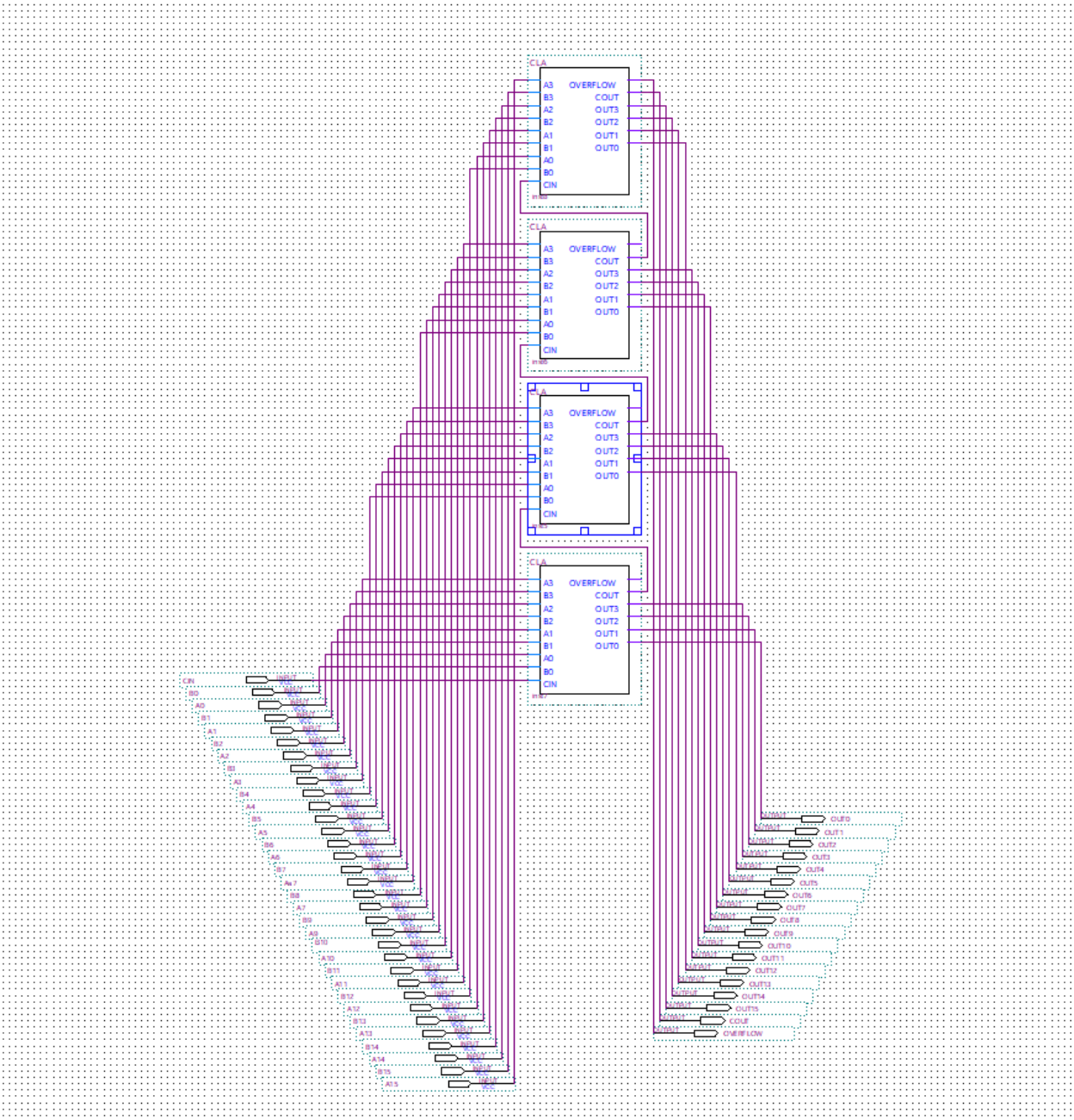




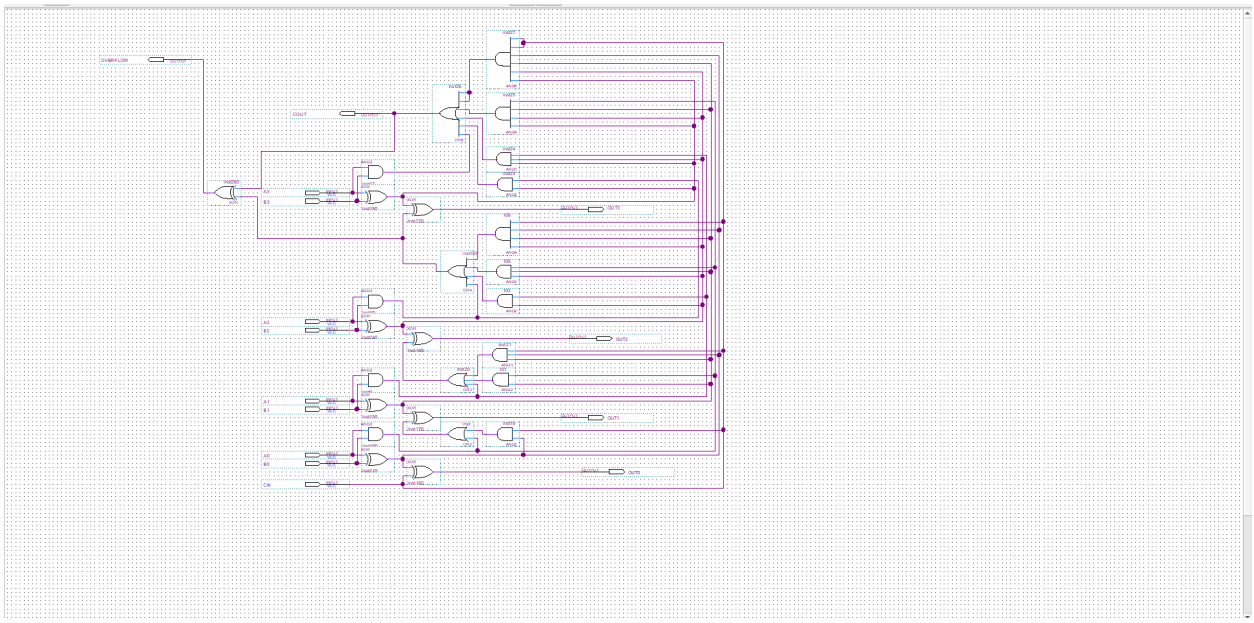
The outputs from the control converter then ran directly into a large adder, which then carried its outputs to “ALUOUT”, with the COUT and OVERFLOW running to the FSM.



General block diagram of the ALU

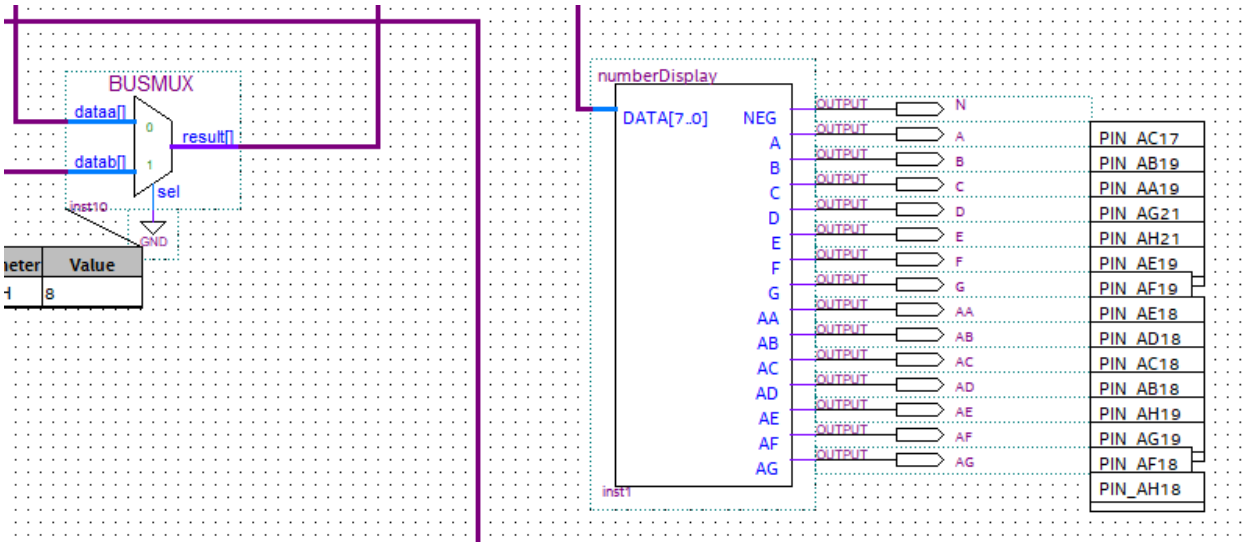


The large 16 bit bit adder

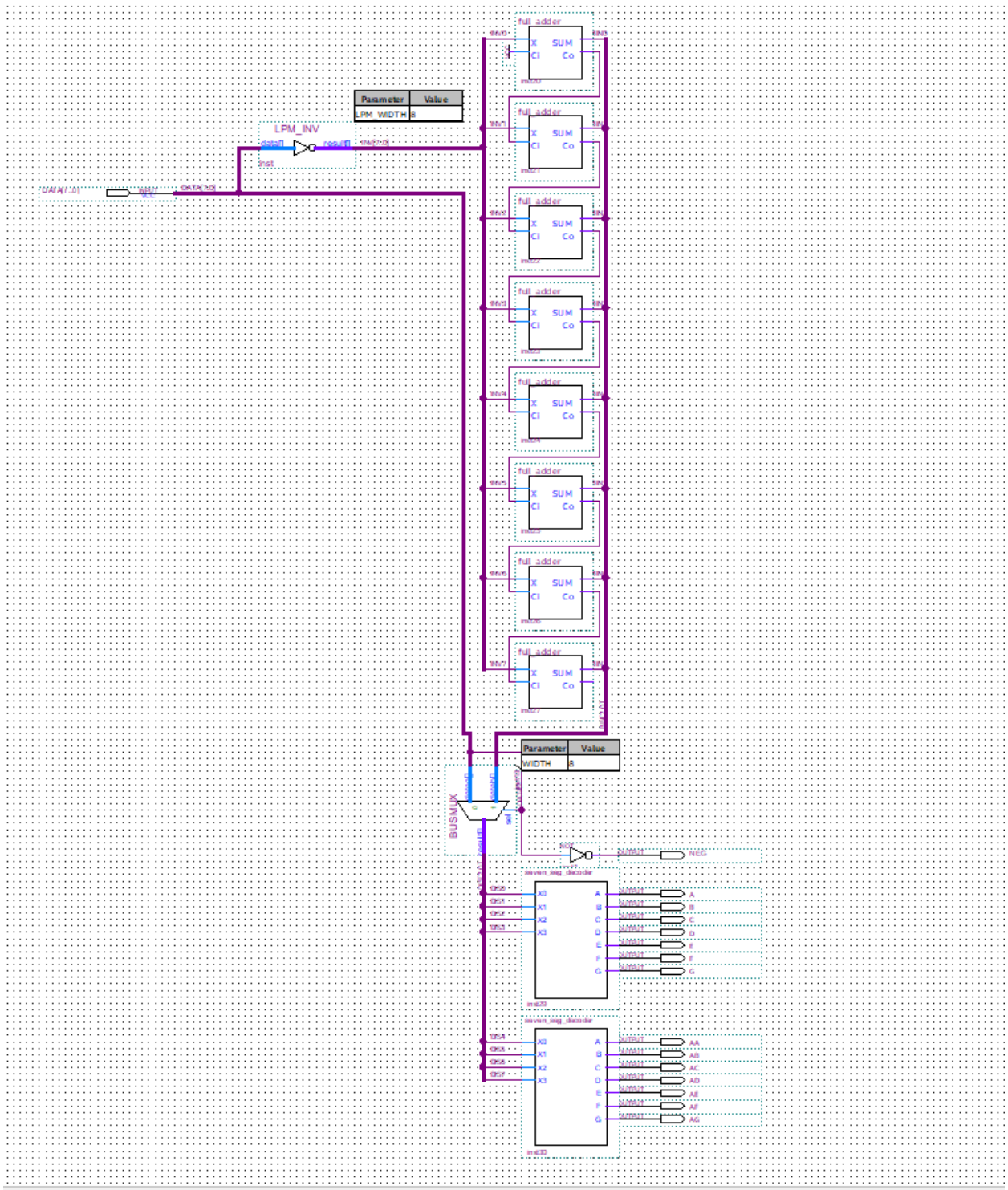


The CLA block diagram used in the adder.

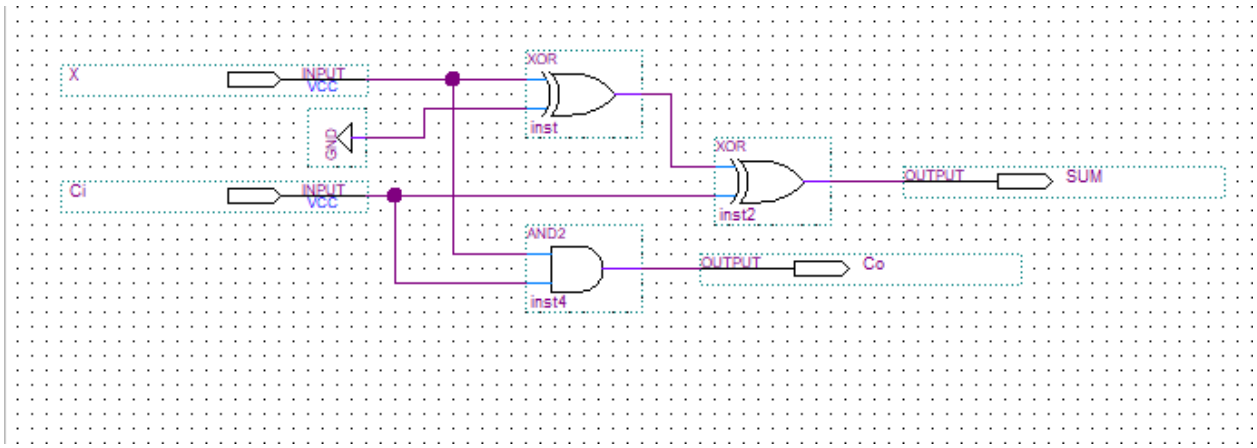
We then had a BUSMUX with the inputs from the operation that had just finished, as well as the output from multiply (which was promptly never used, so the selector is grounded to always be 0). We then ran this then to a 7 segment decoder, which was then used to to show the output with 2 hexadecimal numbers on the board. We also included one segment in a third hexadecmal output for the negative sign, if the number returned is negative.



General format in the ALU

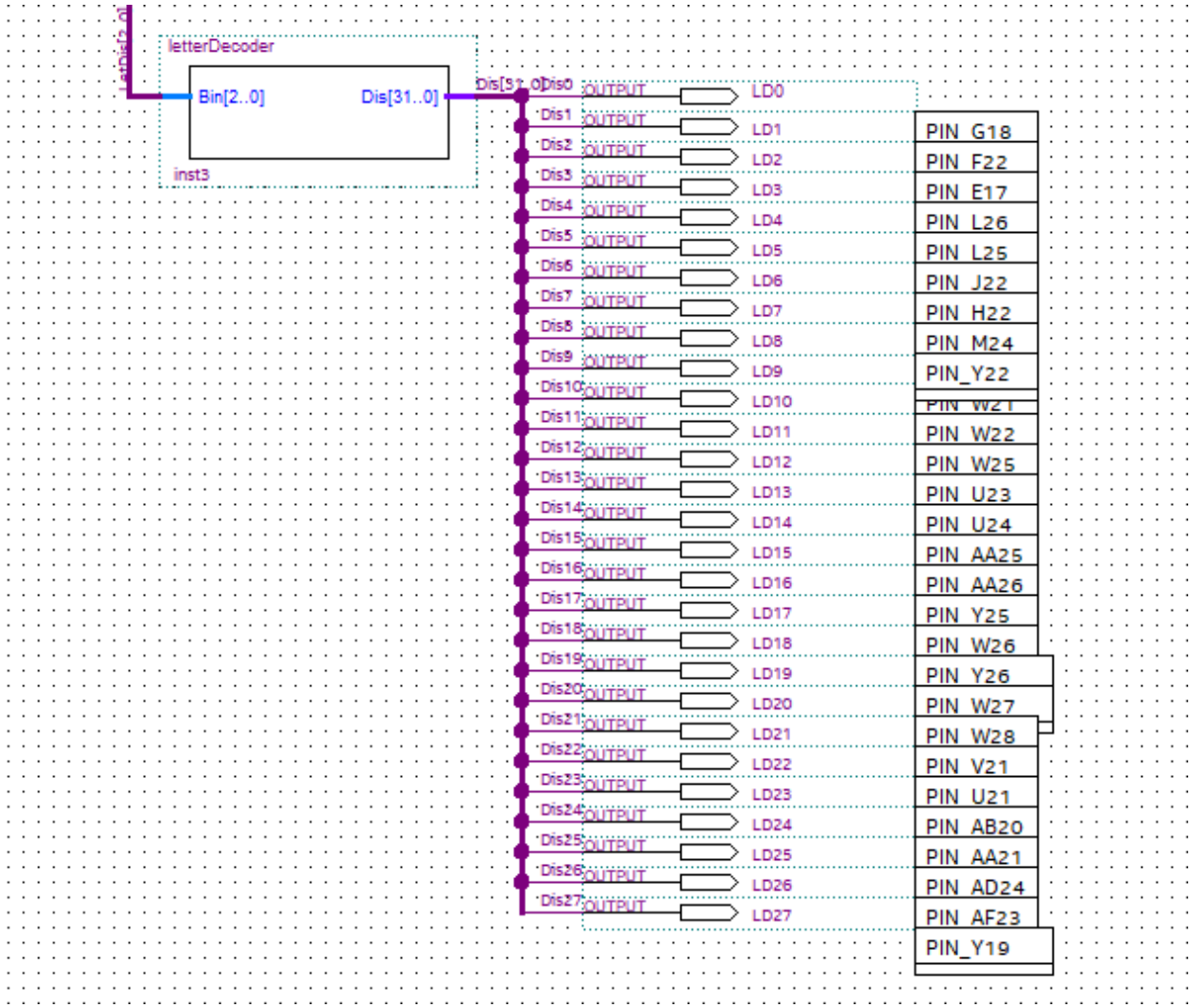


Inside “numberDisplay”



Full adder which was used.

We also included a separate decoder, dependent on which state you were currently in. If you were in the “addition” state, it would show “Add” on the board using the 7 segment. If you were in the “subtraction” state, it would show “Sub” on the board using the 7 segment. As well, we created one for “Mul” if you were in the multiplication state, and “Div” if you were in the division state, though they were never used.



General format in the ALU

```
module letterDecoder(input [2:0]Bin, output reg [31:0]Dis);
always @(Bin)
        begin
        if(Bin == 3'b000) Dis = 32'b11111111111111111111111111111111;
        else if(Bin == 3'b001) Dis = 32'b00100100001000110000110000110;
        else if(Bin == 3'b010) Dis = 32'b11111110001000010000010100001;
        else if(Bin == 3'b011) Dis = 32'b11111110010010100000010000011;
        else if(Bin == 3'b100) Dis = 32'b00010110111011110000011000111;
        else if(Bin == 3'b101) Dis = 32'b11111110100001110111111100011;
        else Dis = 32'b00000000000000000000000000000000;
        end
Endmodule
```

There is a partial implementation of Multiply, which is given below.

```
module multiply (Clock,
Resetn,
LA,
LB,
s,
DataA,
DataB,
P,
Done);
input Clock, Resetn, LA, LB, s;
input [7:0] DataA, DataB;
output [15:0] P;
output reg Done;
wire z;
reg [15:0] DataP;
wire [15:0] A, Sum;
reg [1:0] y, Y;
wire [7:0] B;
reg EA, EB, EP, Psel;
integer k;
parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;
always @(s, y, z)
begin: State_table
        case (y)
        S1: if(s==0)Y=S1;
                else Y = S2;
        S2: if(z==0)Y=S2;
                else Y = S3;
        S3: if(s==1)Y=S3;
                else Y = S1;
        default: Y = 2'bxx;
        endcase
end
always @(posedge Clock, negedge Resetn)
        begin: State_flipflops
                if (Resetn == 0) y <= S1;
                else y <= Y;
        end

        always @(s, y, B[0])
                begin: FSM_outputs
        EA = 0; EB = 0; EP = 0; Done = 0; Psel = 0;
        case (y)
                S1: EP=1;
```

```
                S2: begin
                        EA = 1;
                        EB = 1;
                        Psel = 1;
                        if (B[0]) EP = 1;
                        else EP = 0;
                end
        S3: Done=1;
        endcase
end
//datapath circuit
        shiftrne ShiftB (DataB, LB, EB, 1'b0, Clock, B);
        defparam ShiftB.n = 8;
        shiftlne ShiftA ({{8{1'b0}}, DataA}, LA, EA, 1'b0, Clock, A);
        defparam ShiftA.n = 16;
        assign z = (B == 0);
        assign Sum = A + P;
always @(Psel, Sum)
        for (k = 0; k < 16; k = k+1)
        DataP[k] = Psel ? Sum[k] : 1'b0;
        regne RegP (DataP, Clock, Resetn, EP, P); defparam RegP.n = 16;
endmodule
```

Finally, we have the FSM. The FSM started by default, at state A. By then pressing the Key0, you could alternate between user input (state B) and “pull number from register 1” (state C). Once you have found your desired option, you press Key1 to confirm and go to the next part of the process. It will then take you to where you choose your desired operation, being add,sub, and mul, each of these being its own state. In this state, you would also flip switches 7-0 to choose the number you would like as your second number in the operation. By then clicking Key1 again, it will perform the given operation with the two numbers given, and move to the state for output, and it will stay at that output. To use the calculator again, you flip switch 17 off and then back on, to reset the FSM to state A. Below is the code for the FSM.

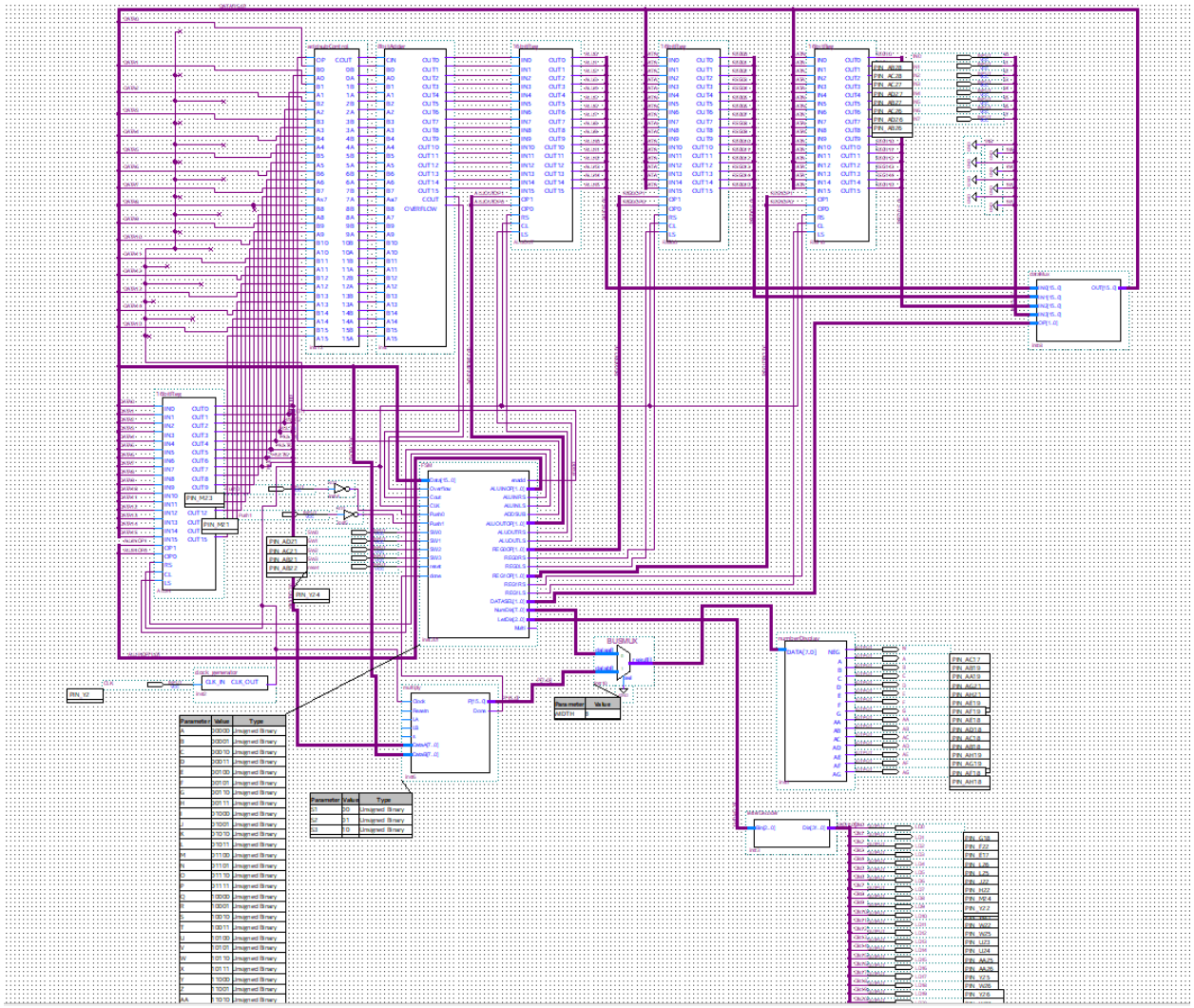
```
module FSM(
input [15:0]Data,
input Overflow,
input Cout,
input CLK,
input Push0,
input Push1,
input SW0,
input SW1,
```

```
input SW2,
input SW3,
input reset,
input done,
output reg enadd,
output reg [1:0]ALUINOP,
output reg ALUINRS,
output reg ALUINLS,
output reg ADDSUB,
output reg [1:0]ALUOUTOP,
output reg ALUOUTRS,
output reg ALUOUTLS,
output reg [1:0]REG0OP,
output reg REG0RS,
output reg REG0LS,
output reg [1:0]REG1OP,
output reg REG1RS,
output reg REG1LS,
output reg [1:0]DATASEL,
output reg [7:0]NumDis,
output reg [2:0]LetDis);

reg [4:0] x,y;
parameter [4:0] A = 5'b00000, B = 5'b00001, C = 5'b00010, D = 5'b00011, E = 5'b00100, F =
5'b00101, G = 5'b00110, H = 5'b00111, I = 5'b01000, J = 5'b01001, K = 5'b01010, L = 5'b01011,
M = 5'b01100, N = 5'b01101, O = 5'b01110, P = 5'b01111, Q = 5'b10000, R = 5'b10001, S =
5'b10010, T = 5'b10011, U = 5'b10100, V = 5'b10101, W = 5'b10110, X = 5'b10111, Y =
5'b11000, Z = 5'b11001, AA = 5'b11010, AB = 5'b11011, AC = 5'b11100, AD = 5'b11101, AE =
5'b11110, AF = 5'b11111;

always @(Push0, Push1, SW0, SW1, SW2, SW3)

        begin: StateTable
                case (x)
                        A: if(Push0) y = B;
                                else if(Push1) y = C;
                                else y = A;
                        B: if(Push0) y = A;
                                else if(Push1) y = C;
                                else y = B;
                        C: if(Push0) y = D;
                                else if(Push1) y = G;//add;
                                else y = C;
                        D: if(Push0) y = E;
```

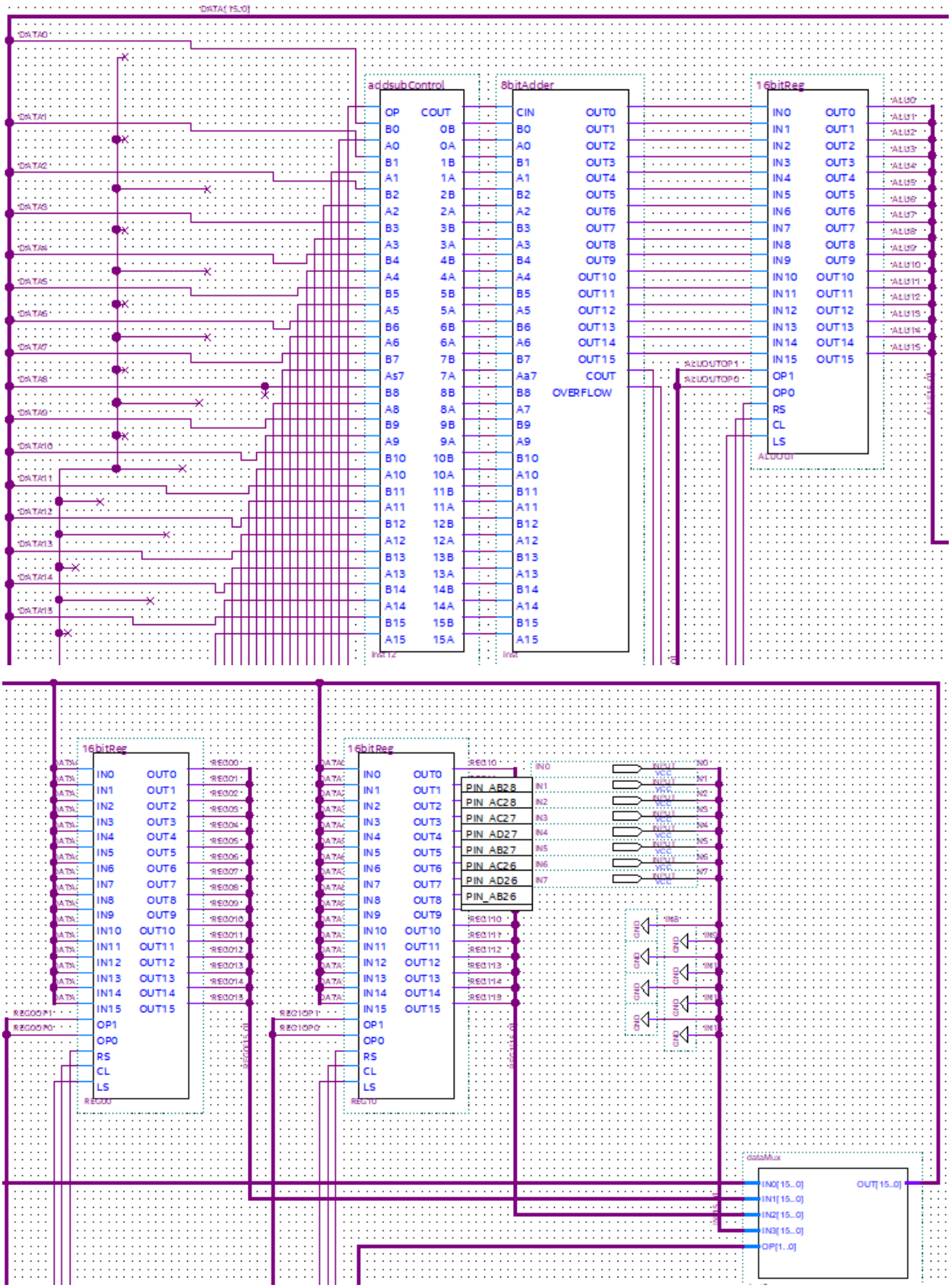
```
                        else if(Push1) y = I;//sub;
                        else y = D;
                    E: if(Push0) y = C;
                        else if(Push1) y = K;//mult;
                        else y = E;
                    G: if(Push0) y = H;
                        else if(Push1) y = T;//o;
                        else y = G;
                    H: if(Push0) y = G;
                        else if(Push1) y = T;//o;
                        else y = H;
                    I: if(Push0) y = J;
                        else if(Push1) y = T;//o;
                        else y = I;
                    J: if(Push0) y = I;
                        else if(Push1) y = T;//o;
                        else y = J;
                    K: if(Push0) y = L;
                        else if(Push1) y = A;//o;
                        else y = K;
                    L: if(Push0) y = K;
                        else if(Push1) y = A;//o;
                        else y = L;
                    T: y = T;
                    default: y = 5'b00000;
                endcase
        end
always @(negedge reset, posedge CLK)
        begin: State_FlipFlops
                if(reset == 0) x <= A;
                else x <= y;
        end

always @(posedge CLK)
        begin: Output
        LetDis = 3'b000;ALUINOP = 2'b00; ALUOUTOP = 2'b00; REG0OP = 2'b00; REG1OP =
2'b00;DATASEL = 2'b00;enadd = 1'b1;ADDSUB = 1'b0;
                case (x)
                    A: begin
                        DATASEL <= 2'b11;
                        ALUINOP <= 2'b11;
                        NumDis <= Data[7:0];
                        end
                    B: begin
```

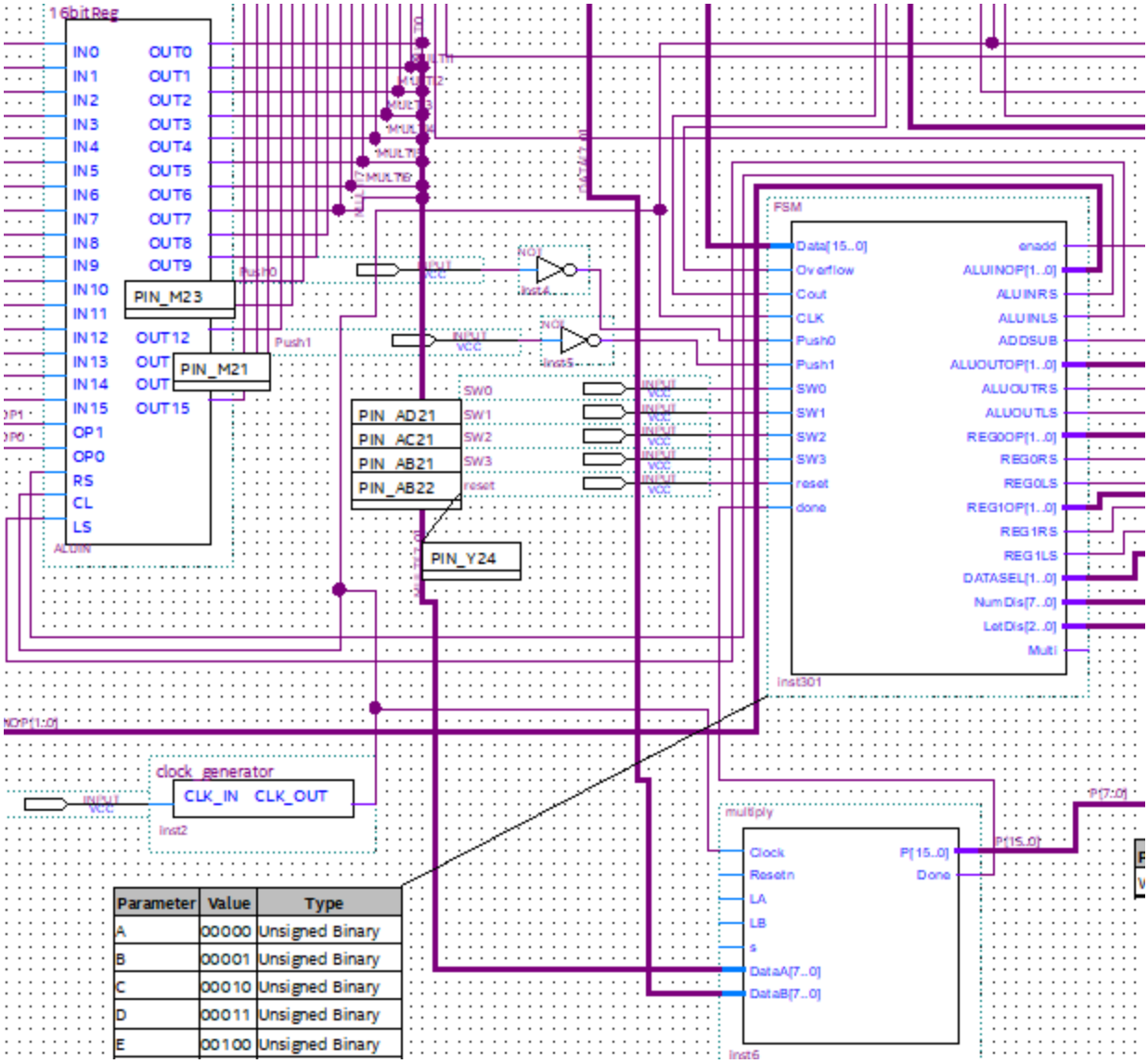
```
        DATASEL <= 2'b01;
        ALUINOP <= 2'b11;
        NumDis <= 8'b00000001;
        LetDis <= 3'b001;
        end
C: begin
        LetDis <= 3'b010;
        NumDis <= 8'b00000000;
        end
D: begin
        LetDis <= 3'b011;
        NumDis <= 8'b00000000;
        end
E: begin
        LetDis <= 3'b100;
        NumDis <= 8'b00000000;
        end
F:      begin
        LetDis <= 3'b101;
        NumDis <= 8'b00000000;
        end
G: begin
        DATASEL <= 2'b11;
        NumDis <= Data[7:0];
        ADDSUB <= 1'b0;
        ALUOUTOP <= 2'b11;
        end
H: begin
        DATASEL <= 2'b01;
        NumDis <= 8'b00000001;
        LetDis <= 3'b001;
        ADDSUB <= 1'b0;
        ALUOUTOP <= 2'b11;
        end
I: begin
        DATASEL <= 2'b11;
        NumDis <= Data[7:0];
        ADDSUB <= 1'b1;
        ALUOUTOP <= 2'b11;
        end
J: begin
        DATASEL <= 2'b01;
        NumDis <= 8'b00000001;
        LetDis <= 3'b001;
```

```
                        ADDSUB <= 1'b1;
                        ALUOUTOP <= 2'b11;
                        end
                K: begin
                        DATASEL <= 2'b11;
                        NumDis <= Data[7:0];
                        end
                L: begin
                        DATASEL <= 2'b01;
                        NumDis <= Data[7:0];
                        end
                M:
                T: begin
                        DATASEL <= 2'b00;
                        NumDis <= Data[7:0];
                        LetDis <= 3'b000;
                        end

        endcase
    end

endmodule
```

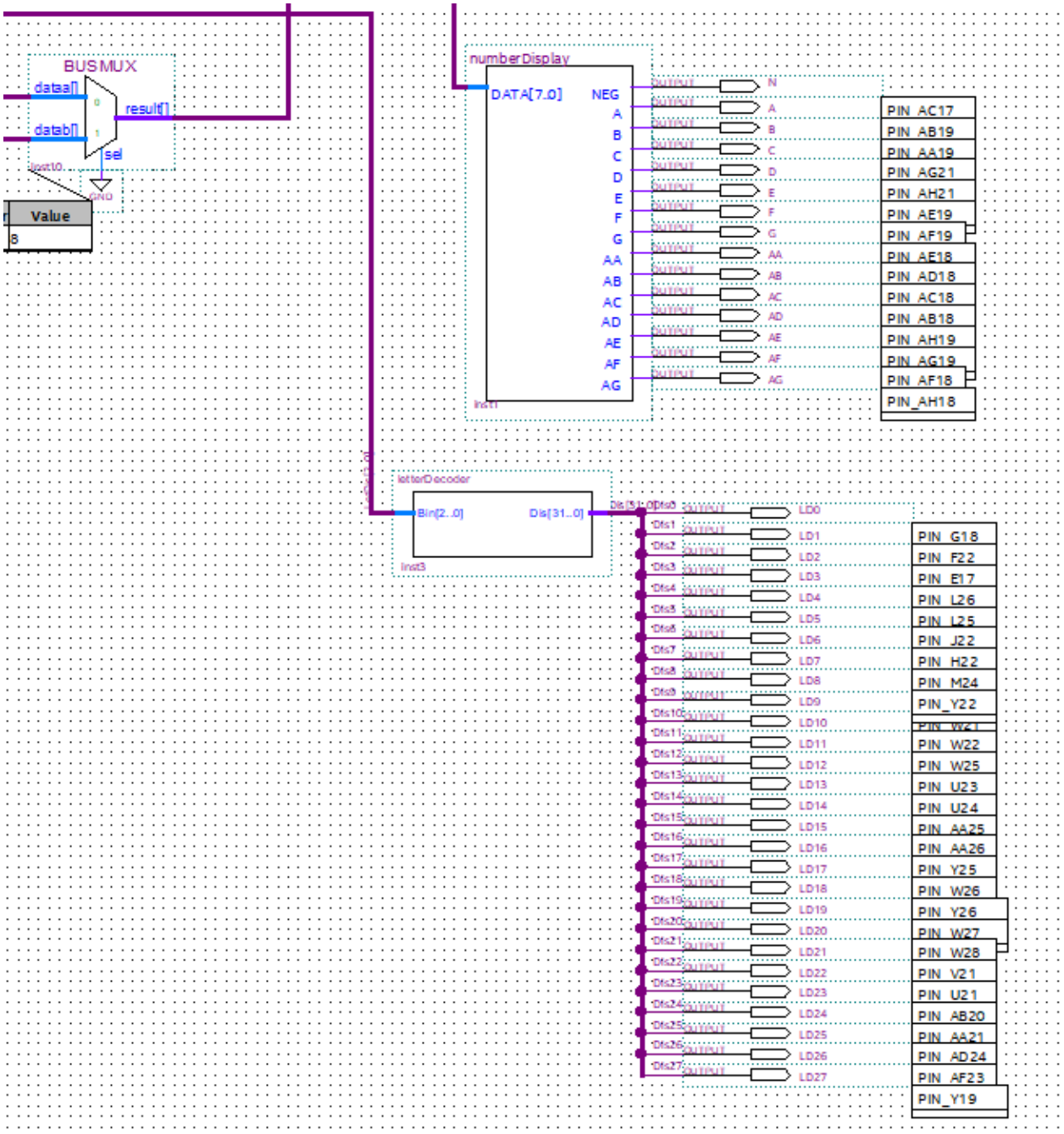


This is the diagram of the entire ALU. Below are more zoomed in pictures.


addsubControl
8bitAdder
16bitReg
OP
COUT
CIN
OVERFLOW
IN0
OUT0
OP1
OP0
RS
CL
LS
PIN_AB28
PIN_AC28
PIN_AC27
PIN_AD27
PIN_AB27
PIN_AC26
PIN_AD26
PIN_AB26
dataMux
IN0[15..0]
IN1[15..0]
IN2[15..0]
IN3[15..0]
OUT[15..0]



Parameter	Value	Type
A	00000	Unsigned Binary
B	00001	Unsigned Binary
C	00010	Unsigned Binary
D	00011	Unsigned Binary
E	00100	Unsigned Binary


BUSMUX
dataa[]
datab[]
result[]
sel
inst10
GND
Value
8
numberDisplay
DATA[7..0]
NEG
A
B
C
D
E
F
G
AA
AB
AC
AD
AE
AF
AG
inst11
PIN_AC17
PIN_AB19
PIN_AA19
PIN_AG21
PIN_AH21
PIN_AE19
PIN_AF19
PIN_AE18
PIN_AD18
PIN_AC18
PIN_AB18
PIN_AH19
PIN_AG19
PIN_AF18
PIN_AH18
letterDecoder
Bin[2..0]
Dis[31..0]
inst3
LD0
LD1
LD2
LD3
LD4
LD5
LD6
LD7
LD8
LD9
LD10
LD11
LD12
LD13
LD14
LD15
LD16
LD17
LD18
LD19
LD20
LD21
LD22
LD23
LD24
LD25
LD26
LD27
PIN_G18
PIN_F22
PIN_E17
PIN_L26
PIN_L25
PIN_J22
PIN_H22
PIN_M24
PIN_Y22
PIN_W21
PIN_W22
PIN_W25
PIN_U23
PIN_U24
PIN_AA25
PIN_AA26
PIN_Y25
PIN_W26
PIN_Y26
PIN_W27
PIN_W28
PIN_V21
PIN_U21
PIN_AB20
PIN_AA21
PIN_AD24
PIN_AF23
PIN_Y19