Kendall Won
Ryan Huang

Lab 6 Report

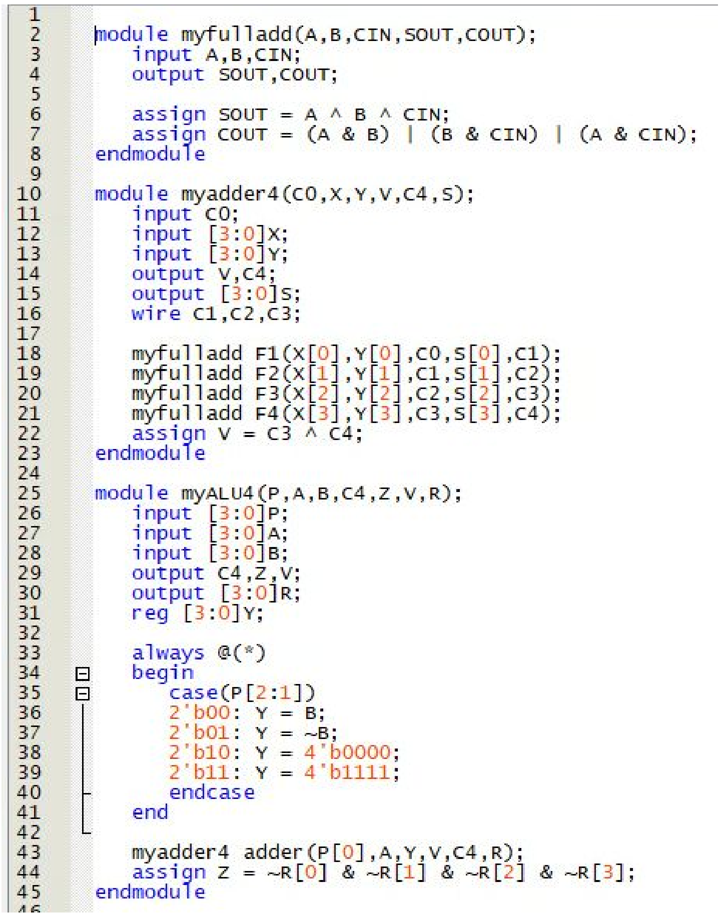
Introduction

In this lab, we created a mini-calculator using muxes, an ALU, 8dff and a busmax.The mini-calculator is able to take in two 4-bit numbers that add/subtract depending on the 4-bit op-select (P[3:0]) that is imputed to the ALU. It is also able to save the output of an operation and reuse it in future calculations.

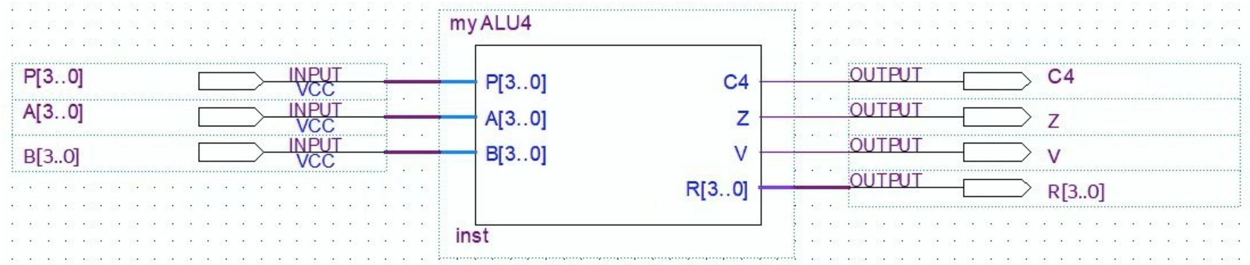
Procedure

We followed the lab instructions, tested out designs, and wrote this report.

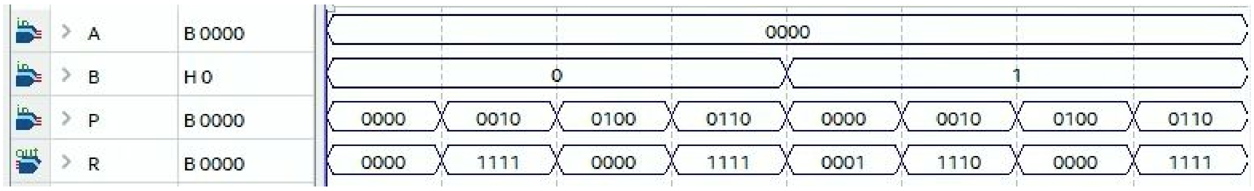
Part 1



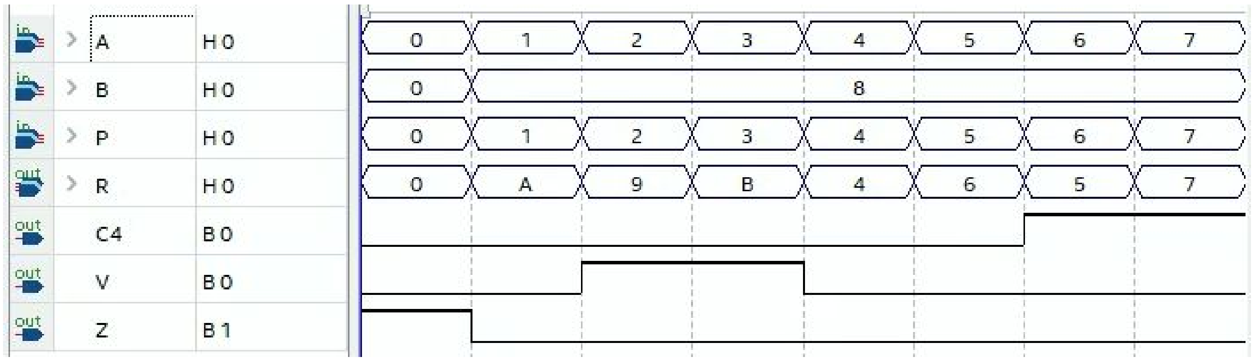
```

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(C0,X,Y,V,C4,S);
   input C0;
   input [3:0]X;
   input [3:0]Y;
   output V,C4;
   output [3:0]S;
   wire C1,C2,C3;

   myfulladd F1(X[0],Y[0],C0,S[0],C1);
   myfulladd F2(X[1],Y[1],C1,S[1],C2);
   myfulladd F3(X[2],Y[2],C2,S[2],C3);
   myfulladd F4(X[3],Y[3],C3,S[3],C4);
   assign V = C3 ^ C4;
endmodule

module myALU4(P,A,B,C4,Z,V,R);
   input [3:0]P;
   input [3:0]A;
   input [3:0]B;
   output C4,Z,V;
   output [3:0]R;
   reg [3:0]Y;

   always @(*)
   begin
      case(P[2:1])
      2'b00: Y = B;
      2'b01: Y = ~B;
      2'b10: Y = 4'b0000;
      2'b11: Y = 4'b1111;
      endcase
   end

   myadder4 adder(P[0],A,Y,V,C4,R);
   assign Z = ~R[0] & ~R[1] & ~R[2] & ~R[3];
endmodule
```



Because Y[0] is not directly exposed in our Verilog, we keep A[0] at 0 and look at R[0] instead. This is the same as testing Y[0], except that subtraction will yield a different result. Still, we checked and the results are correct.



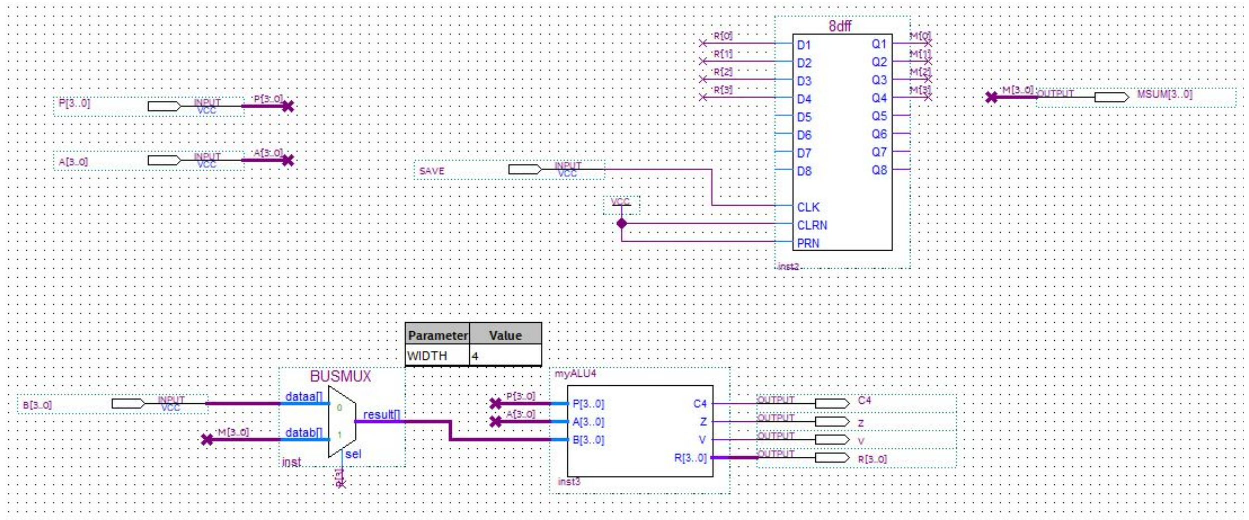
Here are the test results for all 8 functions:



Our test plan was to test all 8 operations and output flags with a variety of inputs. The results are correct, and we also see that Z, V, and C4 work, which is what we want.

Part 2

There is no new Verilog code for this section.



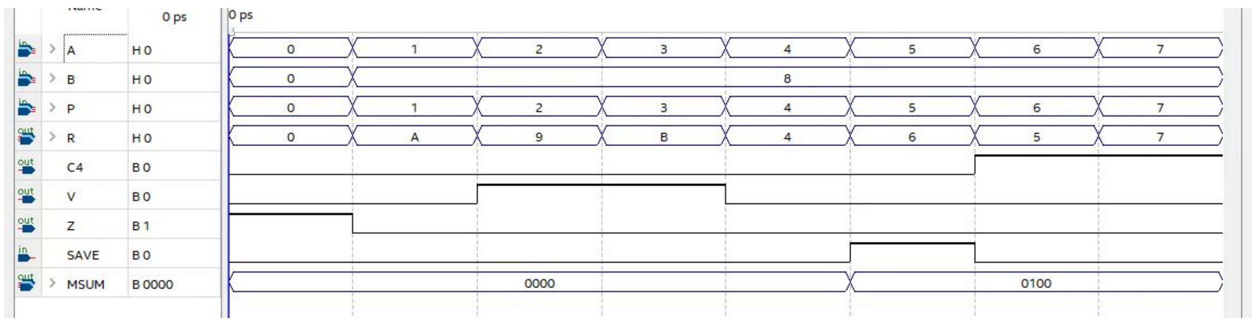
Testing

Test Plan

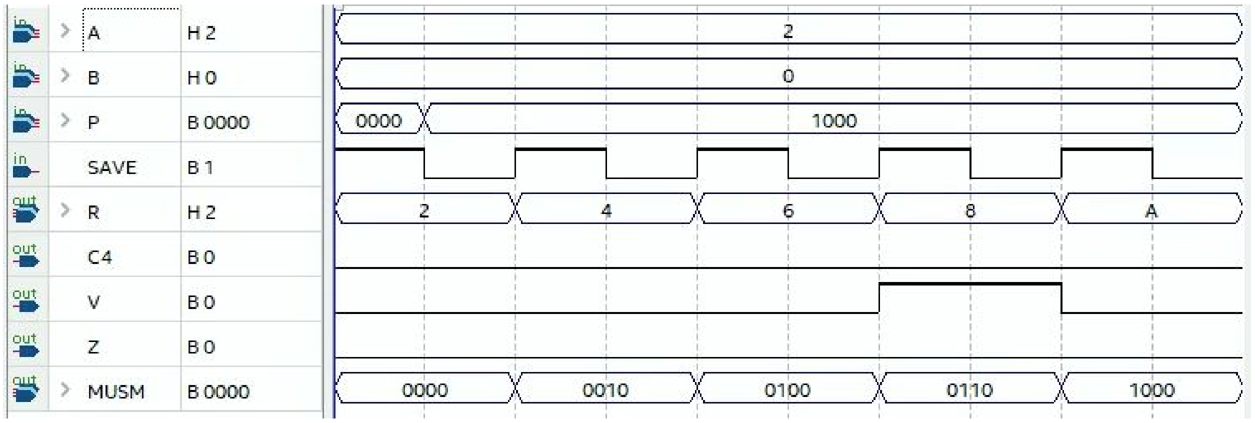
1. Initial set A to 0 and increment it by 1 each interval.
2. Set B to 0 for the first case, and then set it to 8 for the following intervals.
3. Initial set P to 0 and increment it by 1 each interval, in order to test all the ALU op-select inputs.
4. Set Save to 1 in the second interval and sixth interval to test the save is properly being done when only p3 = 1.

Results

1. C4 = 1: Intervals 6 & 7
2. V = 1: Intervals 2 & 3
3. Z = 1: Interval 0
4. MSUM != 0: Interval 5 -> Save = 1 and therefore MSUM = 0100 for intervals 5-7.



13. Here is a sequence of operations that will cause the ALU to count by twos:



The following output is 8, but V is set. This makes sense, because 8 is really -8. This is a two's complement overflow because we have 6 + 2 = -8.

14. If we save one more result, we get 8 (or -8) in M. This makes sense because that's the result of the previous addition.

Report

1. See above
2. See above
3. Negative results show as ordinary numbers in the simulation waveform output. They must be interpreted as two's complement signed numbers for the output to be correct.
4. (0, 0), (1, F), (2, E)
5. (0, 0), (1, 1), (2, 2)
6. Transfer A and store in M. Set A to 0 and subtract using M, which is the same as 0 - M = 0 - A = -A.
7. You would chain them together in the same way that you chain together two half-adders:
    a. ALU 1: adds together the lower four bits. P[3..0] is used. V is ignored.
    b. ALU 2: adds together the top four bits. ALU 1's C4 is connected to P0, P[3..1] are taken from P. C4 and V are output.
    c. Z must combine both Z's. The R's together are the 8-bit result.

Conclusion

Everything went pretty much according to plan, though we did run into some Verilog bugs (e.g. using "CO", which is the letter "O", instead of "C0", which is the number zero).

References

See files in Camino.