ECE472 Lab 2

Ashtyn Moehlenhoff
Torben Rasmussen

Overview:

For this lab, we created a 16-bit ALU to handle 5 different opcodes for: 'AND', 'OR', 'add', 'subtract' and 'set on less than'. We were asked to use carry-look-ahead addition and set the zero/overflow flags correctly.

In order to implement this ALU using carry-look-ahead addition, we created an ALU Slice which takes 4-bit operands, a and b, a carry in bit, the 3-bit opcode, and 'Less'. This slice uses the CLA from lab 1 to add the two 4-bit operands. The slice also finds the result of and'ing and or'ing the four bit operands together. Now that we have all the possible results, we use a 4-to-1 MUX to get the correct output (based on the opcode).

In order to get the 'set on less than' operation to work, we had to create a specialized MSB ALU Slice that has an output, 'set', which is set if a < b (so if the result of a-b is negative, meaning the msb=1, then 'set'=1).

We combine these slices into a 16-bit ALU which is a combination of 3 ALU slices and 1 of the specialized MSB slices. The specialized MSB slice passes the 'set' bit into the first ALU slice, so that if a < b (and opcode==111), the output from the ALU will be 0x0001.

In order to get the zero flag working, we simply check if the output from the ALU is zero. If the output is zero, the flag is set. In order to get the overflow flag working correctly, we check if operands a and b are both positive but we get a negative output, the overflow flag is set. If both operands are negative but we get a positive output, the overflow is set. If one of the operands is positive and another is negative, there is no chance of overflow, so the overflow flag isn't set.

In order to check that our 16-bit ALU is working, we tested each operation with different inputs. You can see the results of our tests in the rest of this document.

Estimated time: ~6hrs

Most valuable: learning more verilog syntax, working with opcodes and mux's.

Least valuable: implementing CLA was repetitive and not very helpful.

Improvements: none

Test 'AND':

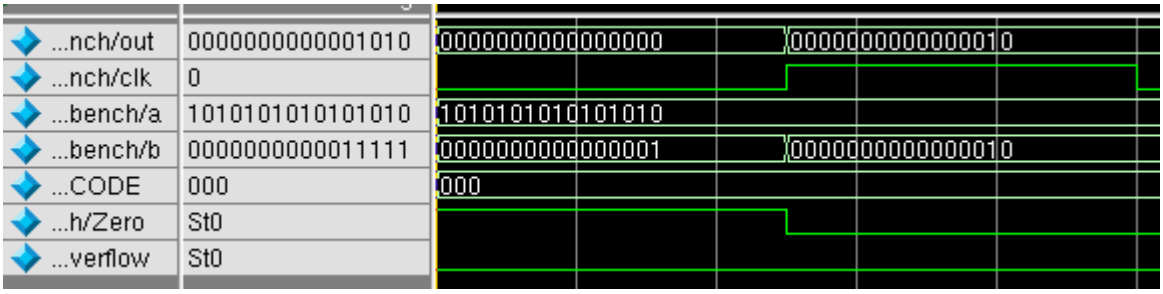
Case 1: a = 16'b0101010101010101; b = 16'b1010101010101010; expected = 16'b0000000000000000

You can see that the output is as we expected, which means this test case passes. Also, the zero flag has been set because the output is all zeros, which is correct!

Case 2: a = 16'b1010101010101010; b = 16'b1010101010101010; expected = 16'b1010101010101010

You can see that the output is as we expected, which means this test case passes. The output is no longer zeros, so the zero flag not being set is correct!

Case 3: a = 16'b1010101010101010; b = 16'b0000000000000001; (and b=b+1 on posedge)



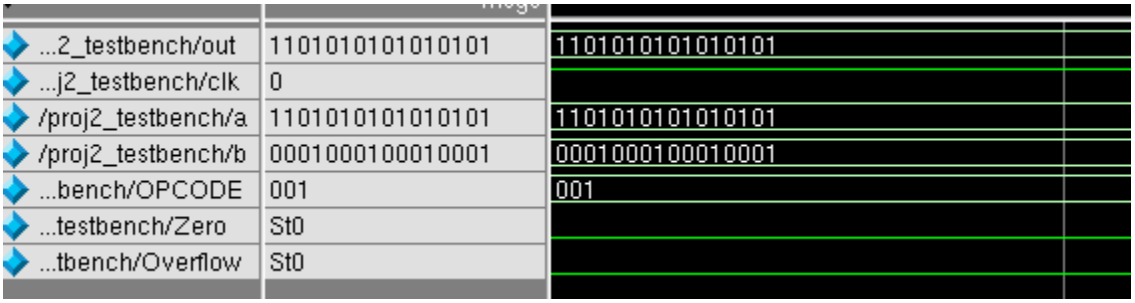
Now you can see that the results for some type of "random" input is correct, and the zero flag is still being set correctly. In all of these cases, the overflow flag is zero. This is because there is no overflow when you're doing an AND operation.

Test ‘OR’

Case 1: a = 16'b0000000000000000; b = 16'b0000000000000000; expected = 16’b0000000000000000

The output is as expected, and the zero flag is set because the output is all zeros.

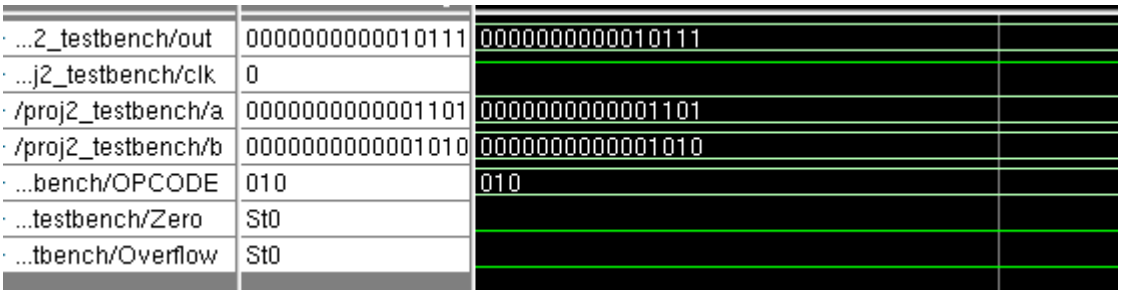
Case 2: a = 16'b1101010101010101; b = 16'b0001000100010001; expected = 16’b1101010101010101



For these more inputs, you can see that the output is as expected and the zero flag is no longer set.

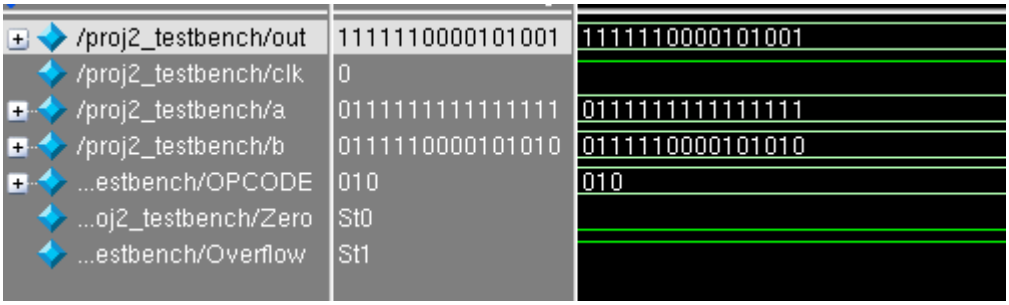
Test ‘add’:

Case 1: (test a simple carry) a = 16'b0000000000001101; b = 16'b0000000000001010; expected = 16’b0000000000010111



For these inputs the output is as expected. The zero and overflow flags aren’t set, which is correct.

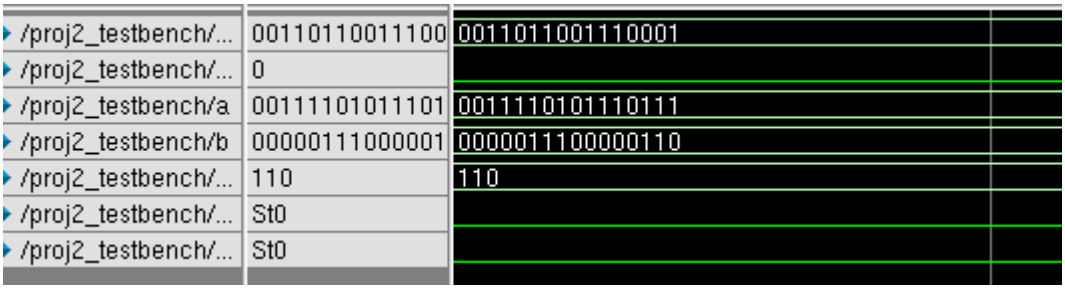
Case2: (test overflow) a = 16'b0111111111111111; b = 16'b0111110000101010;
expected = 16'b1111110000101001



For these inputs the overflow flag should be set because we're adding two positive numbers and the result is a negative number.

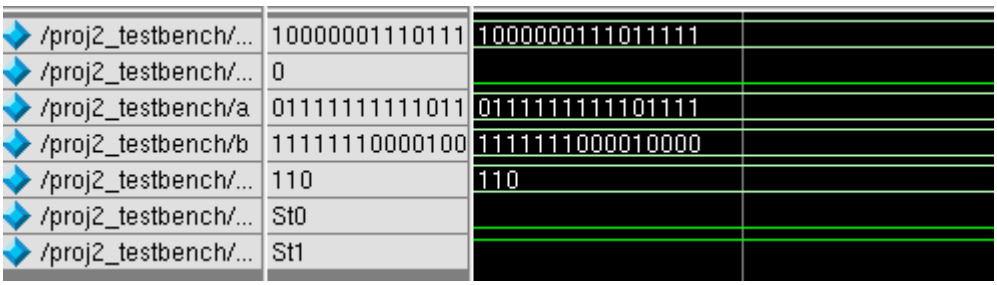
Test 'subtract':

Case 1: (with borrowing) a = 16'b0011110101110111; b = 16'b0000011100000110;
expected =16'b0011011001110001



The output is as expected, success!

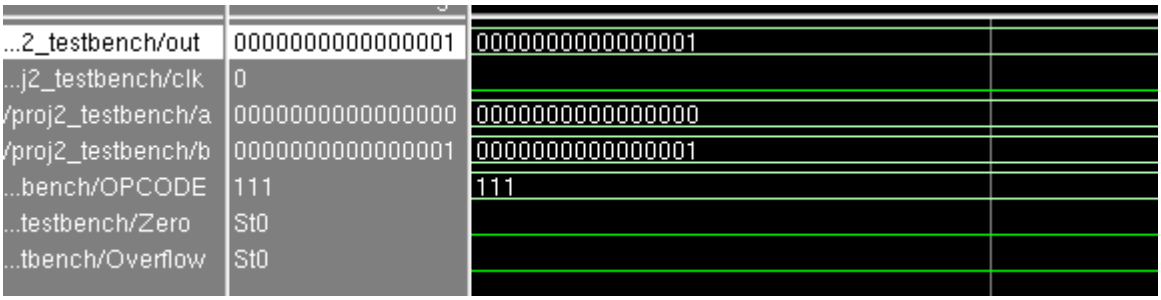
Case 2: (overflow) a = 16'b0111111111101111; b=16'b1111111000010000;
expected=16'b1000000111011111 (overflow set).



Here, because we are subtracting a negative number from a positive number, we should get a positive result. The fact that we get a negative result means we have overflow!

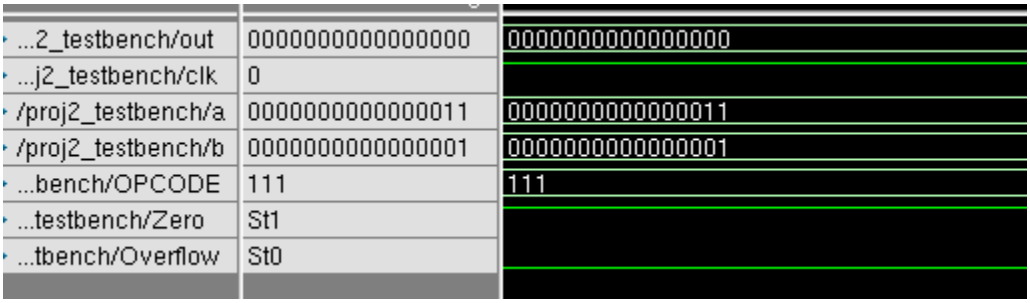
Test ‘set on less than’:

Case 1: (a<b) a = 16'b0000000000000000; b = 16'b0000000000000001;
expected = 16’b0000000000000001



Because a is less than b, the output should be 0x0001 and it is!

Case 2: (a>b) a = 16'b0000000000000011; b = 16'b0000000000000001;
expected = 16’b0000000000000000



Because a is greater than b, the output should be 0x0000 and it is! Also in this case the zero flag is set, which is correct.

```
module proj2_testbench;
  wire [15:0] out;
  reg clk;
  reg [15:0] a, b;
  reg [2:0] OPCODE;
  wire Zero, Overflow;

  ALU16_CLA DUT(a, b, OPCODE, out, Zero, Overflow);

  always
  #5 clk = ~clk;

  // Initialize signals
  initial begin
  clk = 1'b0; //initalize clk value to 0 at t=0

  a = 16'b0000000000000011;
  b = 16'b0000000000000001;
  OPCODE = 3'b111;
  end

  //always @(posedge clk)
 // a = a + 2;
  //always @(posedge clk)
  //b = b + 1;
endmodule


//ALU using Ripple/Fulladder Modules://
module ALU_slice(a, b, Carry_In, Carry_Out,OPCODE, out, Less);
  input a,b,Carry_In, Less;
  input [2:0] OPCODE;
  output out, Carry_Out;
  wire AND, OR, SUM, binvert;
  reg temp_b;

  assign binvert = OPCODE[2];
  always @*
  begin
  temp_b = b;
  if (binvert == 1'b1)
    temp_b = ~b;
  end
  assign AND = a & temp_b;
  assign OR = a | temp_b;
  fulladder f0(a, temp_b, Carry_In, SUM, Carry_Out);
  mux4 f1(AND, OR, SUM, Less, OPCODE[1:0], out);
endmodule

module ALU_slice_MSB(a, b, Carry_In, Overflow, OPCODE, out, set);
  input a,b,Carry_In;
  input [2:0] OPCODE;
  output Overflow, out, set;
  ALU_slice f0(a, b, Carry_In, Overflow, OPCODE, out, 1'b0);
  assign set = out;
endmodule

module ALU16(a, b, OPCODE, out, Zero, Overflow);
  input [15:0] a, b;
  input [2:0] OPCODE;
  output [15:0] out;
  output Zero, Overflow;
  wire [15:0] Carry;
  wire set;
  ALU_slice f00(a[0],  b[0],  OPCODE[2], Carry[0],  OPCODE, out[0], set); //pass
 in Binvert as the carry
  ALU_slice f01(a[1],  b[1],  Carry[0],  Carry[1],  OPCODE, out[1], 1'b0);
  ALU_slice f02(a[2],  b[2],  Carry[1],  Carry[2],  OPCODE, out[2], 1'b0);
  ALU_slice f03(a[3],  b[3],  Carry[2],  Carry[3],  OPCODE, out[3], 1'b0);
```



```
  ALU_slice f04(a[4],  b[4],  Carry[3],  Carry[4],  OPCODE, out[4], 1'b0);
  ALU_slice f05(a[5],  b[5],  Carry[4],  Carry[5],  OPCODE, out[5], 1'b0);
  ALU_slice f06(a[6],  b[6],  Carry[5],  Carry[6],  OPCODE, out[6], 1'b0);
  ALU_slice f07(a[7],  b[7],  Carry[6],  Carry[7],  OPCODE, out[7], 1'b0);
  ALU_slice f08(a[8],  b[8],  Carry[7],  Carry[8],  OPCODE, out[8], 1'b0);
  ALU_slice f09(a[9],  b[9],  Carry[8],  Carry[9],  OPCODE, out[9], 1'b0);
  ALU_slice f10(a[10], b[10], Carry[9],  Carry[10], OPCODE, out[10], 1'b0);
  ALU_slice f11(a[11], b[11], Carry[10], Carry[11], OPCODE, out[11], 1'b0);
  ALU_slice f12(a[12], b[12], Carry[11], Carry[12], OPCODE, out[12], 1'b0);
  ALU_slice f13(a[13], b[13], Carry[12], Carry[13], OPCODE, out[13], 1'b0);
  ALU_slice f14(a[14], b[14], Carry[13], Carry[14], OPCODE, out[14], 1'b0);
  ALU_slice_MSB f15(a[15], b[15], Carry[14], Carry[15], OPCODE, out[15], set);
  assign Zero = (out==16'h0000);
  assign Overflow = Carry[15];
endmodule


//ALU Carry Look Ahead Modules://
module ALU_slice_CLA(a, b, Carry_In, Carry_Out,OPCODE, out, Less);
  input [3:0] a,b,Less;
  input Carry_In;
  input [2:0] OPCODE;
  output [3:0] out;
  output Carry_Out;
  wire [3:0] AND, OR, SUM;
  wire binvert;
  reg [3:0] temp_b;

  assign binvert = OPCODE[2];
  always @*
  begin
  temp_b = b;
  if (binvert == 1'b1)
    temp_b = ~b;
  end
  assign AND = a & temp_b;
  assign OR = a | temp_b;
  ripple_adder_LA f0(a, temp_b, Carry_In, Carry_Out, SUM);

  mux4 f1(AND[0], OR[0], SUM[0], Less[0], OPCODE[1:0], out[0]);
  mux4 f2(AND[1], OR[1], SUM[1], Less[1], OPCODE[1:0], out[1]);
  mux4 f3(AND[2], OR[2], SUM[2], Less[2], OPCODE[1:0], out[2]);
  mux4 f4(AND[3], OR[3], SUM[3], Less[3], OPCODE[1:0], out[3]);
endmodule

module ALU_slice_CLA_MSB(a, b, Carry_In, Carry_Out, OPCODE, out, Less, set);
  output [3:0] set;
  input [3:0] a,b,Less;
  input Carry_In;
  input [2:0] OPCODE;
  output [3:0] out;
  output Carry_Out;
  wire [3:0] AND, OR, SUM;
  wire binvert;
  reg [3:0] temp_b;

  assign binvert = OPCODE[2];
  always @*
  begin
    temp_b = b;
  if (binvert == 1'b1)
    temp_b = ~b;
  end
  assign AND = a & temp_b;
  assign OR = a | temp_b;
  ripple_adder_LA f0(a, temp_b, Carry_In, Carry_Out, SUM);

  mux4 f1(AND[0], OR[0], SUM[0], Less[0], OPCODE[1:0], out[0]);
  mux4 f2(AND[1], OR[1], SUM[1], Less[1], OPCODE[1:0], out[1]);
```

```
  mux4 f3(AND[2], OR[2], SUM[2], Less[2], OPCODE[1:0], out[2]);
  mux4 f4(AND[3], OR[3], SUM[3], Less[3], OPCODE[1:0], out[3]);
  assign set[3:1] = 3'b000;
  assign set[0] = SUM[3];
endmodule

module ALU16_CLA(a, b, OPCODE, out, Zero, Overflow);
  input [15:0] a, b;
  input [2:0] OPCODE;
  output [15:0] out;
  output Zero, Overflow;
  wire [3:0] Carry;
  wire [3:0] set;
  reg overflow_temp = 1'b0;

  ALU_slice_CLA f0(a[3:0], b[3:0], OPCODE[2], Carry[0], OPCODE, out[3:0], set);
  ALU_slice_CLA f1(a[7:4], b[7:4], Carry[0],  Carry[1], OPCODE, out[7:4], 4'b000
0);
  ALU_slice_CLA f2(a[11:8], b[11:8], Carry[1], Carry[2], OPCODE, out[11:8], 4'b0
000);
  ALU_slice_CLA_MSB f3(a[15:12], b[15:12], Carry[2], Carry[3], OPCODE, out[15:12
], 4'b0000, set);

  assign Zero = (out==16'h0000);

  always @*
    if(OPCODE[2:1] == 2'b01)
      begin
        if(a[15]==1 && b[15]==1 && out[15]!=1)
          overflow_temp = 1'b1;
        else if(a[15]==0 && b[15]==0 && out[15]!=0)
          overflow_temp = 1'b1;
      end;

    always @*
    if(OPCODE[2:1] == 2'b11)
      begin
        if(a[15]==1 && b[15]==0 && out[15]!=1)
          overflow_temp = 1'b1;
        else if(a[15]==0 && b[15]==1 && out[15]!=0)
          overflow_temp = 1'b1;
        end;

  assign Overflow = overflow_temp;

endmodule

//4-to-1 MUX://
module mux4(a, b, c, d, s , out);
  input a,b,c,d;
  input [1:0] s;
  output out;
  reg out;

  always@(a or b or c or d or s)
  case (s)
    0:  out=a;
    1:  out=b;
    2:  out=c;
    3:  out=d;
  default: $display("Error in selection");
  endcase
endmodule

//CLA Adder Modules://
module ripple_adder_LA(a, b, cin, cout, sum);
input [3:0] a, b;
input cin;
output cout;
```



```
output [3:0] sum;
wire [4:0] c; // used for carry connections
wire [3:0] g;
wire [3:0] p;
wire P,G;
assign c[0]=cin;
fulladder_LA f0(a[0], b[0], c[0], sum[0], p[0], g[0]);
fulladder_LA f1(a[1], b[1], c[1], sum[1], p[1], g[1]);
fulladder_LA f2(a[2], b[2], c[2], sum[2], p[2], g[2]);
fulladder_LA f3(a[3], b[3], c[3], sum[3], p[3], g[3]);

lookahead f4(p[3],p[2],p[1],p[0],g[3],g[2],g[1],g[0],c[0],c[1],c[2],c[3],c[4],P,
 G);
assign cout = c[4];
endmodule

module fulladder_LA(a, b, cin, sum, p, g);
input a, b, cin;
output sum, p, g;
assign sum = a ^ b ^ cin;
assign p = a | b;
assign g = a & b;

endmodule

module lookahead(p3,p2,p1,p0,g3,g2,g1,g0,c0,c1,c2,c3,c4,P,G);
input p3,p2,p1,p0,g3,g2,g1,g0,c0;
output c1,c2,c3,c4,P,G;

assign c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) | (p3 & p2 & p
1 & p0 & c0);
assign c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c0);
assign c2 = g1 | (p1 & g0) | (p1 & p0 & c0);
assign c1 = g0 | (p0 & c0);
assign P = p3 & p2 & p1 & p0;
assign G = g3 & g2 & g1 & g0;

endmodule

//Ripple/Fulladder Modules://
module fulladder(a, b, cin, sum, cout);
  input a, b, cin;
  output sum, cout;
  assign sum = a ^ b ^ cin;
  assign cout = a & b | a & cin | b & cin;
endmodule

module ripple_adder(a, b, sum, cout);
  input [15:0] a, b;
  output [15:0] sum;
  output cout;
  wire [15:0] c; // used for carry connections
  assign c[0]=0;

  fulladder f0(a[0], b[0], c[0], sum[0], c[1]);
  fulladder f1(a[1], b[1], c[1], sum[1], c[2]);
  fulladder f2(a[2], b[2], c[2], sum[2], c[3]);
  fulladder f3(a[3], b[3], c[3], sum[3], c[4]);
  fulladder f4(a[4], b[4], c[4], sum[4], c[5]);
  fulladder f5(a[5], b[5], c[5], sum[5], c[6]);
  fulladder f6(a[6], b[6], c[6], sum[6], c[7]);
  fulladder f7(a[7], b[7], c[7], sum[7], c[8]);
  fulladder f8(a[8], b[8], c[8], sum[8], c[9]);
  fulladder f9(a[9], b[9], c[9], sum[9], c[10]);
  fulladder f10(a[10], b[10], c[10], sum[10], c[11]);
  fulladder f11(a[11], b[11], c[11], sum[11], c[12]);
  fulladder f12(a[12], b[12], c[12], sum[12], c[13]);
  fulladder f13(a[13], b[13], c[13], sum[13], c[14]);
  fulladder f14(a[14], b[14], c[14], sum[14], c[15]);
```

```
  fulladder f15(a[15], b[15], c[15], sum[15], cout);
endmodule
```