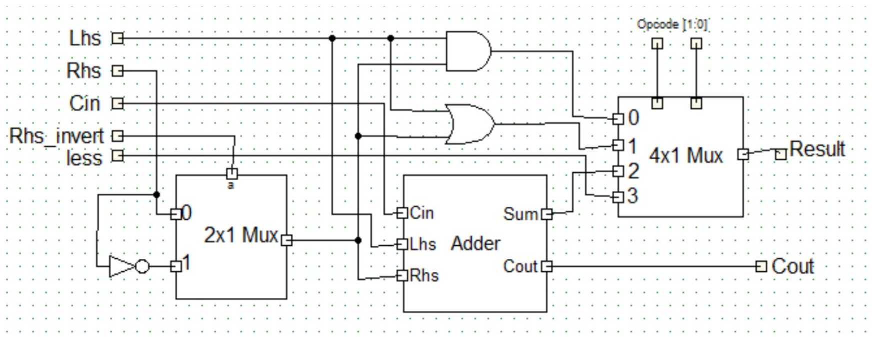
CS 354
Section 02
Fall 2019
Christopher Schmitt Assignment 2

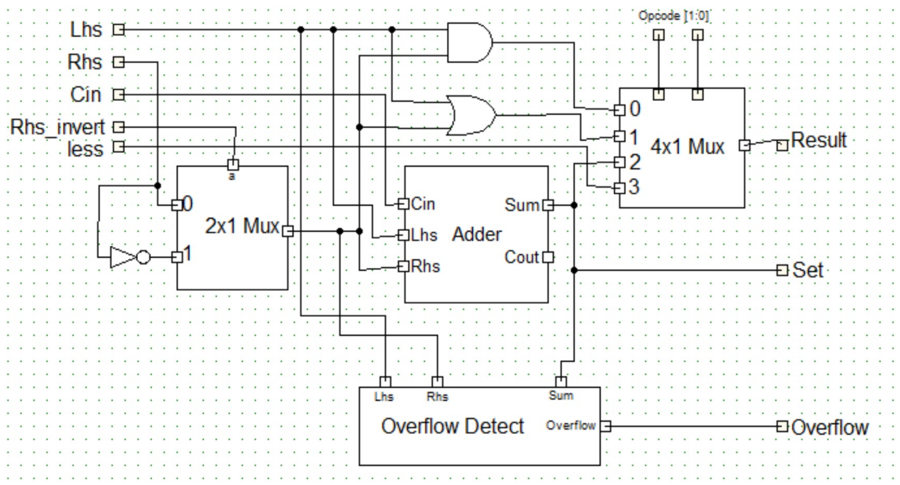
Problem 1

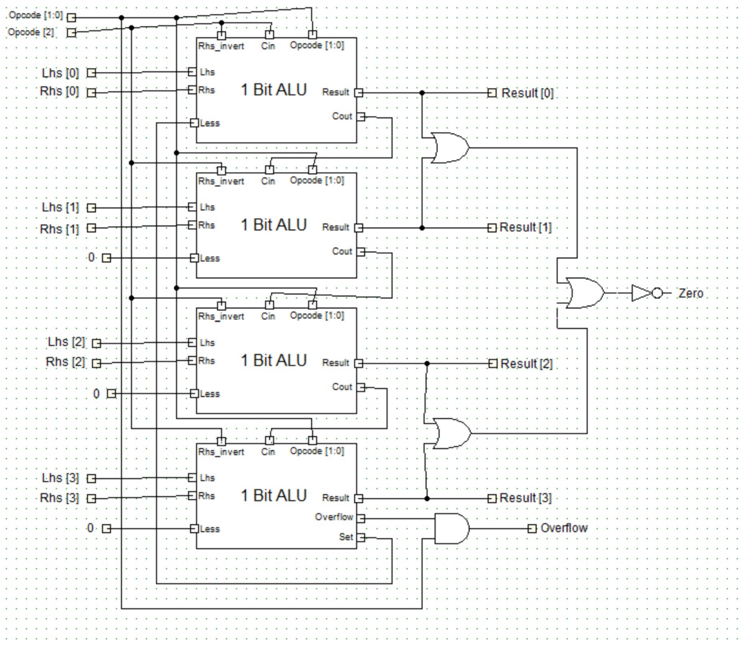
This simple ALU supports five functions: AND, OR, ADD, SUB, and SLT. This ALU can be simplified by using the fact that in a 2's complement scheme, the SUB operation can be achived by inverting the right hand, adding one, and then performing standard addition. The four-bit ALU is created by instantiating four one-bit ALUs. The design of the one bit ALU is very simple. It is created by instantiating an AND gate, an OR gate, an adder, and a buffer. All these operations are performed in parallel. The least two significant bits on the opcode line drives a 4x1 multiplexer which selects one of these computations and puts it on the result line. The effect of chaining four of these ALUs together is a four-bit ALU. The final ALU in the chain of four has some extra functions, namely, overflow detetion and driving the 'set' line. Overflow detection is achived by handling two unique cases (The sign of the operands do not match the sign of the output). In order to limit overflow detection to arithmetic operations (disabled for logical ops), the overflow line is ANDed with the second bit of the opcode (which is always 1 when the operation is arithmetic).

```
module HalfAdder(
  input wire lhs,
  input wire rhs,
  output wire sum,
  output wire cout
);

  // Compute sum
  xor(sum, lhs, rhs);

  // Comput cout
  and(cout, lhs, rhs);

endmodule // HalfAdder



module Adder(
  input wire cin,
  input wire lhs,
  input wire rhs,
  output wire sum,
  output wire cout
);

  // Instantiate two half-adders
  HalfAdder halfAdderA(cin, lhs, buffer, carry_left);
  HalfAdder halfAdderB(buffer, rhs, sum, carry_right);

  // Compute cout
  or(cout, carry_left, carry_right);

endmodule // Adder



module TwoOneMux(
  input wire select,
  input wire [1:0] data,
  output wire result
);

  // Invert select line
  not(select_, select);

  // Compute products
  and(lhs, data[0], select_);
  and(rhs, data[1], select);

  // Compute sum
  or(result, lhs, rhs);

endmodule // TwoOneMux



module FourOneMux(
  input wire [1:0] select,
  input wire [3:0] data,
  output wire result
);

  // Instantiate layer one
  TwoOneMux muxA(select[0], {data[0], data[1]}, AB);
  TwoOneMux muxB(select[0], {data[2], data[3]}, BC);

  // Instantiate layer two
  TwoOneMux muxC(select[1], {AB, BC}, result);

endmodule // FourOneMux
```

```
module OverflowDetector(
  input wire lhs_msb,
  input wire rhs_msb,
  input wire sum_msb,
  output wire overflow
);

  // Invert inputs
  not(lhs_prime, lhs_msb);
  not(rhs_prime, rhs_msb);
  not(sum_prime, sum_msb);

  // Condition one (positive operands, negative sum)
  and(buffer_one, lhs_prime, rhs_prime);
  and(condition_one, buffer_one, sum_msb);

  // Contition two (negative operands, positive sum)
  and(buffer_two, lhs_msb, rhs_msb);
  and(condition_two, buffer_two, sum_prime);

  // Compute overflow
  or(overflow, condition_one, condition_two);

endmodule // OverflowDetector



module OneBitALU(
  input wire rhs_invert,
  input wire [1:0] opcode,
  input wire cin,
  input wire lhs,
  input wire rhs,
  input wire less,
  output wire result,
  output wire cout
);

  // Handle rhs inversion
  not(rhs_, rhs);
  TwoOneMux inverter(rhs_invert, {rhs_, rhs}, rhs_inverted);

  // Instantiate output mux
  FourOneMux mux(opcode, {and_out, or_out, sum_out, less},
      result);
  // Instantiate operations
  and(and_out, lhs, rhs_inverted);
  or(or_out, lhs, rhs_inverted);
  Adder adder(cin, lhs, rhs_inverted, sum_out, cout);

endmodule // OneBitALU



module OneBitALU_Set(
  input wire rhs_invert,
  input wire [1:0] opcode,
  input wire cin,
  input wire lhs,
  input wire rhs,
  input wire less,
  output wire result,
  output wire overflow,
  output wire set
);

  // Handle rhs inversion
  not(rhs_, rhs);
  TwoOneMux inverter(rhs_invert, {rhs_, rhs}, rhs_inverted);

  // Instantiate output mux
  FourOneMux mux(opcode, {and_out, or_out, sum_out, less},
      result);

  // Instantiate operations
  and(and_out, lhs, rhs_inverted);
  or(or_out, lhs, rhs_inverted);
  Adder adder(cin, lhs, rhs_inverted, sum_out, cout); // NOTE:
      cout is unused in this block
  buf(set, sum_out);

  // Handle overflow detection
  OverflowDetector detector(lhs, rhs_inverted, sum_out,
      overflow);

endmodule // OneBitALU_Set



module ALU(
  input wire [2:0] opcode,
  input wire [3:0] lhs,
  input wire [3:0] rhs,
  output wire [3:0] result,
  output wire overflow,
  output wire zero
);

  // Instantiate four one-bit ALUs
  OneBitALU aluA(opcode[2], {opcode[1], opcode[0]}, opcode[2],
      lhs[0], rhs[0], set, result[0], coutA);
  OneBitALU aluB(opcode[2], {opcode[1], opcode[0]}, coutA,
      lhs[1], rhs[1], 1'b0, result[1], coutB);
  OneBitALU aluC(opcode[2], {opcode[1], opcode[0]}, coutB,
      lhs[2], rhs[2], 1'b0, result[2], coutC);
  OneBitALU_Set aluD(opcode[2], {opcode[1], opcode[0]}, coutC,
      lhs[3], rhs[3], 1'b0, result[3], overflow_buffer, set);

  // Compute zero
  or(left, result[0], result[1]);
  or(right, result[2], result[3]);
  or(zero_, left, right);
  not(zero, zero_);

  // Limit overflow to arithmetic operations
  and(overflow, overflow_buffer, opcode[1]);

endmodule // ALU



module Test;

  reg unsigned [2:0] opcode;
  reg signed [3:0] lhs;
  reg signed [3:0] rhs;
  wire signed [3:0] result;
  wire overflow;
  wire zero;

  ALU alu(opcode, lhs, rhs, result, overflow, zero);

  initial begin
    $display("operation lhs    rhs     result  overflow zero");
    $monitor("%d %b   %d %b  %d %b  %d %b   %d        %d", opcode,
        opcode, lhs, lhs, rhs, rhs, result, result, overflow,
        zero);
    opcode = 2; lhs = 2; rhs = 2;
    #20; opcode = 2; lhs = 4; rhs = 1;
    #20; opcode = 2; lhs = 6; rhs = 6;
    #20; opcode = 2; lhs = 1; rhs = 1;
    #20; opcode = 2; lhs = 7; rhs = -7;
    #20; opcode = 6; lhs = 5; rhs = 2;
    #20; opcode = 6; lhs = 5; rhs = 7;
    #20; opcode = 6; lhs = -6; rhs = -6;
    #20; opcode = 6; lhs = -7; rhs = 7;
    #20; opcode = 6; lhs = 2; rhs = -2;
    #20; opcode = 0; lhs = 4'b1111; rhs = 4'b0011;
    #20; opcode = 0; lhs = 4'b1111; rhs = 4'b1111;
    #20; opcode = 0; lhs = 4'b0011; rhs = 4'b1100;
    #20; opcode = 1; lhs = 4'b0000; rhs = 4'b0001;
    #20; opcode = 1; lhs = 4'b0000; rhs = 4'b0011;
    #20; opcode = 1; lhs = 4'b1111; rhs = 4'b1111;
    #20; opcode = 7; lhs = 3; rhs = 3;
    #20; opcode = 7; lhs = 3; rhs = 2;
    #20; opcode = 7; lhs = 2; rhs = 3;
  end

endmodule // Test
```







```
D:\School\CS-354>vvp a.out
operation lhs     rhs     result  overflow zero
2 010     2 0010  2 0010  4 0100  0        0
2 010     4 0100  1 0001  5 0101  0        0
2 010     6 0110  6 0110 -4 1100  1        0
2 010     1 0001  1 0001  2 0010  0        0
2 010     7 0111 -7 1001  0 0000  0        1
6 110     5 0101  2 0010  3 0011  0        0
6 110     5 0101  7 0111 -2 1110  0        0
```

```
6 110    -6 1010 -6 1010  0 0000  0         1
6 110    -7 1001  7 0111  2 0010  1         0
6 110     2 0010 -2 1110  4 0100  0         0
0 000    -1 1111  3 0011  3 0011  0         0
0 000    -1 1111 -1 1111 -1 1111  0         0
0 000     3 0011 -4 1100  0 0000  0         1
1 001     0 0000  1 0001  1 0001  0         0
1 001     0 0000  3 0011  3 0011  0         0
1 001    -1 1111 -1 1111 -1 1111  0         0
7 111     3 0011  3 0011  0 0000  0         1
7 111     3 0011  2 0010  0 0000  0         1
7 111     2 0010  3 0011  1 0001  0         0
```