Problem 1

This simple CPU supports six opertions: AND, OR, ADD, SUB, SLT, and LI. This CPU has four registers available to it: $0, $1, $2, and $3. Instructions are loaded into the instruction register. It should be noted that instructions can conform to two different formats, R-Type and L-Type, both of length nine. R-Type instructions follow the form: Opcode (3 bits), Lhs (2 bits), Rhs (2 bits), and destination (2 bits). L-Type instructions follow the form: Opcode (3 bits), Payload (4 bits), and destination (2 bits). An Li decoder is used to in conjunction with several multiplexers to distingush between L-Type and R-Type instructions. All of the actual operations are perfomed by the ALU, the rest of the CPU is simply moving and routing data around on the falling edge of each clock pulse. This data gets routed to the register file, which uses a register-select line to determine where the data on the write-data line should be routed.

```
module DLatch(
  input wire enable,
  input wire data,
  output wire Q
);

  not(data_, data);
  nand(x, data, enable);
  nand(y, data_, enable);
  nand(Q, x, Q1);
  nand(Q1, y, Q);

endmodule // DLatch



module DFlipFlop(
  input wire clk,
  input wire data,
  output wire Q
);

  not(clk_, clk);

  DLatch d0(clk, data, y);
  DLatch d1(clk_, y, Q);

endmodule // DFlipFlop
```

```
module InstructionReg(
  input wire clk,
  input wire [8:0] dataIn,
  output wire [8:0] dataOut
);

  DFlipFlop d0(clk, dataIn[0], dataOut[0]);
  DFlipFlop d1(clk, dataIn[1], dataOut[1]);
  DFlipFlop d2(clk, dataIn[2], dataOut[2]);
  DFlipFlop d3(clk, dataIn[3], dataOut[3]);
  DFlipFlop d4(clk, dataIn[4], dataOut[4]);
  DFlipFlop d5(clk, dataIn[5], dataOut[5]);
  DFlipFlop d6(clk, dataIn[6], dataOut[6]);
  DFlipFlop d7(clk, dataIn[7], dataOut[7]);
  DFlipFlop d8(clk, dataIn[8], dataOut[8]);

endmodule // InstructionReg
```

```
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
```

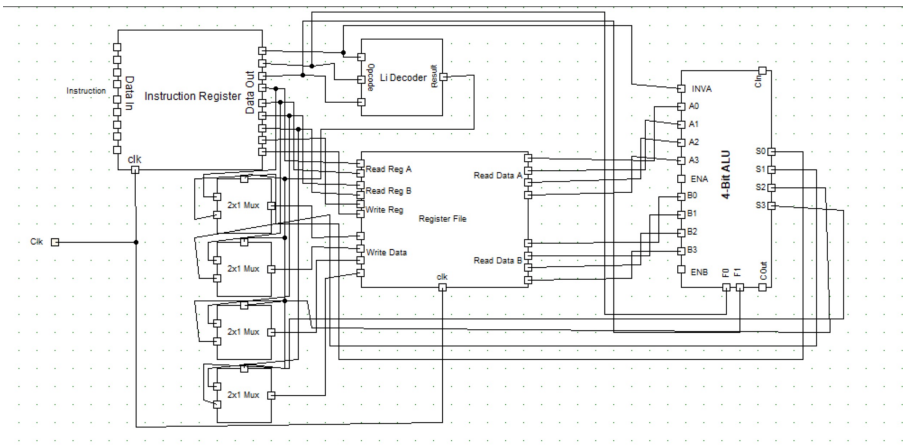
```
module ALU(
  input wire [2:0] opcode,
  input wire [3:0] lhs,
  input wire [3:0] rhs,
  output wire [3:0] result,
  output wire overflow,
  output wire zero
);

  // 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
```

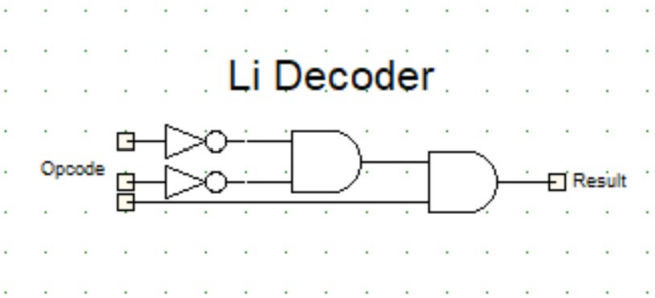
```
module RegFile(
  input wire clk,
  input wire [1:0] readRegA,
  input wire [1:0] readRegB,
  input wire [1:0] writeReg,
  input wire [3:0] writeData,
  output wire [3:0] readDataA,
  output wire [3:0] readDataB
);

  reg [3:0] memory [0:3];

  assign readDataA = memory[readRegA];
```

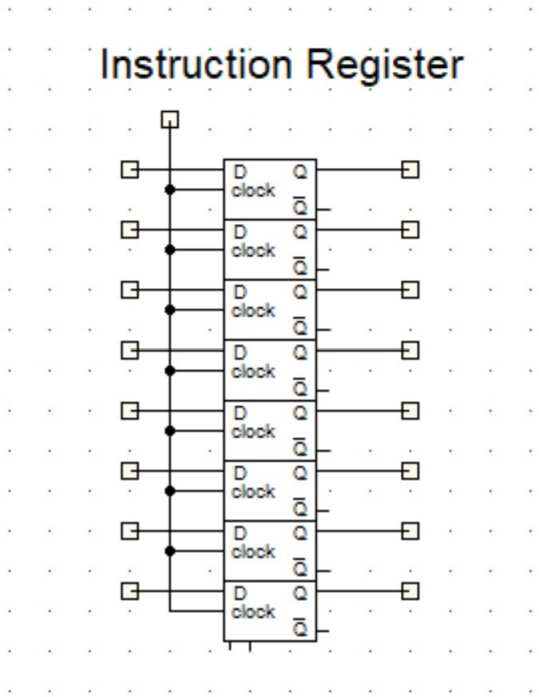
```
  assign readDataB = memory[readRegB];

  initial memory[0] = 0;

  always @(negedge clk) begin
    memory[writeReg] <= writeData;
  end

endmodule // RegFile



module LiDecoder(
  input wire [2:0] opcode,
  output wire result
);

  // Invert lower two bits
  not(op0_, opcode[0]);
  not(op1_, opcode[1]);

  // Compute product
  and(lowerBits, op0_, op1_);
  and(result, lowerBits, opcode[2]);

endmodule // LiDecoder



module CPU(
  input wire clk,
  input wire [8:0] instruction,
  output wire [3:0] writeData
);

  wire [8:0] IR;
  wire [3:0] lhs;
  wire [3:0] rhs;
  wire [3:0] aluResult;

  // Instatiate instruction register
  InstructionReg instructionReg(clk, instruction, IR);

  // Instantiate register file
  RegFile regFile(clk, IR[5:4], IR[3:2], IR[1:0], writeData,
      lhs, rhs);
```

```
  // Instantaite write-data mux
  TwoOneMux twoOneMux0(liResult, {IR[2], aluResult[0]},
      writeData[0]);
  TwoOneMux twoOneMux1(liResult, {IR[3], aluResult[1]},
      writeData[1]);
  TwoOneMux twoOneMux2(liResult, {IR[4], aluResult[2]},
      writeData[2]);
  TwoOneMux twoOneMux3(liResult, {IR[5], aluResult[3]},
      writeData[3]);

  // Instatiate the ALU
  ALU alu(IR[8:6], lhs, rhs, aluResult, overflow, zero);

  // Instantiate the li-decoder
  LiDecoder liDecoder(IR[8:6], liResult);

endmodule // CPU



module TestCPU;

  reg clk;
  reg [8:0] instruction;
  wire signed [3:0] writeData;

  CPU cpu(clk, instruction, writeData);

  initial begin
    $display("Instruction\tWriteData");
    $monitor("%b\t%b %d", instruction, writeData, writeData);
    #1; instruction = 9'b100111110; clk = 1; #1; clk = 0;     //
        Load 15 -> $2
    #1; instruction = 9'b100100011; clk = 1; #1; clk = 0;     //
        Load 8 -> $3
    #1; instruction = 9'b000101101; clk = 1; #1; clk = 0;     //
        $2 & $3 -> $1
    #1; instruction = 9'b110011011; clk = 1; #1; clk = 0;     //
        $1 - $2 -> $3
    #1; instruction = 9'b111110010; clk = 1; #1; clk = 0;     //
        SLT $2, $3, $0
    #1; instruction = 9'b010101101; clk = 1; #1; clk = 0;     //
        $2 + $3 -> $1
    #1; instruction = 9'b110000101; clk = 1; #1; clk = 0;     //
        $0 - $1 -> $1
```

```
    #1; instruction = 9'b001011011; clk = 1; #1; clk = 0;     //
        $1 | $2 -> $3
  end

endmodule // TestCPU
```








```
100111110    # Load 15 -> $2
100100011    # Load 8 -> $3
000101101    # $2 & $3 -> $1
110011011    # $1 - $2 -> $3
111110010    # SLT $2, $3, $0
010101101    # $2 + $3 -> $1
110000101    # $0 - $1 -> $1
001011011    # $1 | $2 -> $3
```

```
iverilog.exe ./CPU.v && vvp.exe a.out
Instruction   WriteData
xxxxxxxxx     xxxx  x
100111110     xxxx  x
100111110     1111 -1
100100011     1111 -1
100100011     1000 -8
000101101     1000 -8
000101101     1000 -8
110011011     1000 -8
```

```
110011011     1001 -7
111110010     1001 -7
111110010     0001  1
010101101     0001  1
010101101     1010 -6
110000101     1010 -6
110000101     0110  6
001011011     0110  6
001011011     0111  7
```