Names:

Alistair Chambers

Surya Singh

Richard Piotrowitcz

Tasks completed:

RP: Flowcharts

SS: Set Description

AC: Coding, debugging, and machine language translation

The opcode we were currently using ADD, SUB, AND, OR, NOR, NAND, SLT, and ADDI. The ADD instruction adds any register values to each other and stores it in another. The SUB instruction subtracts any register values from one and stores it in another. The AND instruction compares the values in 2 registers and its destination register is 1 only when both operand bits are 1. The OR instruction compares the values in 2 registers as operands and puts that result in another register. The NOR instruction is the complement version of OR because while it compares the values in 2 registers as operands, the result for NOR would be the reverse from the result for OR. The NAND instruction compares the values in 2 registers and its destination register is The SLT (set on less than) instruction compares the values in 2 registers to each other (whether one is less than the other) and returns either 1 or 0 depending on what the values are. The ADDI instruction adds an immediate operand to a register and stores it in another.

Instruction	OpCode
ADD	0000
SUB	0001

AND	0010
OR	0011
NOR	0100
NAND	0101
SLT	0110
ADDI	0111

Extended the 4 bit ALU to 16 bits and rewrote the code at gate level. Then we had to create register files with four 16 bit registers with register 0 being read only and registers 1-3 being Read and Write to the register.

Instruction memory had to be translated from 32 bit MIPS instructions and code into 16 bits machine language.

ALUctrl and main control were combined into mainctl. This control has the opcode translated into RegDst, ALUSrc, RegWrite, and ALUctl. The 3 high bits were connected to the 2 bits 2x1 multiplexer, 16 bit 2x1 multiplexers and the RegFile. 7 bits (3 for RegDst, ALUSrc, and RegWrite and the remaining 4 are for the operations for each ALUOp which is then connected to the ALU control).

Code:

```
//half adder
module halfadder(S,C,x,y);
input x,y;
output S,C;
xor x1 (S,x,y);
and a1 (C,x,y);
```

```
endmodule


//full adder

module fulladder(S,C,x,y,z);

input x,y,z;

output S,C;

wire S1,C1,C2;

halfadder HA1 (S1,C1,x,y),

          HA2 (S,C2,S1,z);

   or C3(C,C2,C1);

endmodule


// 4x1 multiplexer

module mux4x1(i0,i1,i2,i3,select,y);


          input i0,i1,i2,i3;

          input [1:0] select;

          output y;

          wire S0,S1,w1,w2,w3,w4;


          not not1(S0,select[0]),

                    not2(S1,select[1]);


          and and1(w1,i0,S1,S0),

                    and2(w2,i1,S1,select[0]),

                    and3(w3,i2,select[1],S0),
                        and4(w4,i3,select[1],select[0]);


            or  or1(y,w1,w2,w3,w4);


endmodule


//2x1 multplexer
module mux2x1(A,B,select,OUT);
  input A,B,select;
  output OUT;


  not not1(i0, select);
  and and1(i1, A, i0);
  and and2(i2, B, select);
  or or1(OUT, i1, i2);
endmodule


module mux_2(result,a,b,c);
            input a,b,c;
            output result;
            wire w1,w2,w3;


            not (w3,c);
            and g1(w1,b,c),
                g2(w2,a,w3);
            or  g3(result,w1,w2);
```

```
endmodule


//16 bit ALU

module ALU(op,a,b,result,zero);

         input [15:0] a,b;

         input [3:0] op;

         output [15:0] result;

         output zero;

         wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15;


ALU1    alu0 (a[0],b[0],op[3],op[2],op[1:0],c16, op[2],c1,result[0]),

      alu1 (a[1],b[1],op[3],op[2],op[1:0],1'b0,c1,   c2,result[1]),

         alu2 (a[2],b[2],op[3],op[2],op[1:0],1'b0,c2,   c3,result[2]),

      alu3 (a[3],b[3],op[3],op[2],op[1:0],1'b0,c3,   c4,result[3]),

         alu4 (a[4],b[4],op[3],op[2],op[1:0],1'b0,c4,   c5,result[4]),

         alu5 (a[5],b[5],op[3],op[2],op[1:0],1'b0,c5,   c6,result[5]),

            alu6 (a[6],b[6],op[3],op[2],op[1:0],1'b0,c6,   c7,result[6]),

            alu7 (a[7],b[7],op[3],op[2],op[1:0],1'b0,c7,   c8,result[7]),

         alu8 (a[8],b[8],op[3],op[2],op[1:0],1'b0,c8,   c9,result[8]),

         alu9 (a[9],b[9],op[3],op[2],op[1:0],1'b0,c9,   c10,result[9]),

         alu10 (a[10],b[10],op[3],op[2],op[1:0],1'b0,c10,   c11,result[10]),

         alu11 (a[11],b[11],op[3],op[2],op[1:0],1'b0,c11,   c12,result[11]),

         alu12 (a[12],b[12],op[3],op[2],op[1:0],1'b0,c12,   c13,result[12]),

         alu13 (a[13],b[13],op[3],op[2],op[1:0],1'b0,c13,   c14,result[13]),

         alu14 (a[14],b[14],op[3],op[2],op[1:0],1'b0,c14,   c15,result[14]);
```

```
ALUmsb alu15 (a[15],b[15],op[3],op[2],op[1:0],1'b0,c15,   c16,result[15]);


or or1(or01, result[0],result[1]);

or or2(or23, result[2],result[3]);

nor nor1(zero,or01,or23);


endmodule


// 1bit Alu

module ALU1 (a,b,Ainvert,binvert,op,less,carryin,carryout,result);

  input a,b,carryin,binvert,Ainvert;

  input [1:0] op;

  input  less;

  output carryout,result;

  wire sum, a_and_b, a_or_b, b_inv,a_inv;


  not not1(a_inv, a);

  not not2(b_inv, b);


  mux2x1 x1(a,a_inv,Ainvert,a1),

         mux1(b,b_inv,binvert,b1);

  and and1(a_and_b, a1, b1);

  or or1(a_or_b, a1, b1);


  fulladder adder1(sum,carryout,a1,b1,carryin);

  mux4x1 mux2(a_and_b,a_or_b,sum,less,op[1:0],result);
```

```
endmodule


//1 bit alu significant bit

module ALUmsb (a,b,ainvert,binvert,op,less,carryin,carryout,result);

   input a,b,carryin,binvert,ainvert;

   input [1:0] op;

   input less;

   output carryout,result;

   wire sum, a_and_b, a_or_b, b_inv,a_inv;


   not not1(a_inv, a);

   not not2(b_inv, b);

   mux2x1 x1(a,a_inv,ainvert,a1),

            mux1(b,b_inv,binvert,b1);

   and and1(a_and_b, a1, b1);

   or or1(a_or_b, a1, b1);

   fulladder adder1(sum,carryout,a1,b1,carryin);

   mux4x1 mux2(a_and_b,a_or_b,sum,less,op[1:0],result);


endmodule


// register file

module reg_file(rr1,rr2,wr,wd,regwrite,rd1,rd2,clock);

            input [1:0] rr1,rr2,wr;

            input [15:0] wd;
```

```
        input regwrite,clock;

        output [15:0] rd1,rd2;

        wire [15:0] q1,q2,q3;


        register reg1(wd,c1,q1);

        register reg2(wd,c2,q2);

        register reg3(wd,c3,q3);


        mux4x1_16bit mux1 (16'b0,q1,q2,q3,rr1,rd1),

                     mux2 (16'b0,q1,q2,q3,rr2,rd2);


        decoder dec(wr[1],wr[0],w3,w2,w1,w0);


        and and0 (regwrite_and_clock,regwrite,clock),

            and1 (c1,regwrite_and_clock,w1),

            and2 (c2,regwrite_and_clock,w2),

            and3 (c3,regwrite_and_clock,w3);

endmodule



// 16 bit register

module register(WriteData,CLK, ReadData);

  input [15:0] WriteData;

  input CLK;

  output [15:0] ReadData;

  D_flip_flop r0(WriteData[0], CLK, ReadData[0]),
        r1(WriteData[1], CLK, ReadData[1]),
            r2(WriteData[2], CLK, ReadData[2]),
        r3(WriteData[3], CLK, ReadData[3]),
        r4(WriteData[4], CLK, ReadData[4]),
        r5(WriteData[5], CLK, ReadData[5]),
        r6(WriteData[6], CLK, ReadData[6]),
        r7(WriteData[7], CLK, ReadData[7]),
        r8(WriteData[8], CLK, ReadData[8]),
        r9(WriteData[9], CLK, ReadData[9]),
        r10(WriteData[10], CLK, ReadData[10]),
        r11(WriteData[11], CLK, ReadData[11]),
        r12(WriteData[12], CLK, ReadData[12]),
        r13(WriteData[13], CLK, ReadData[13]),
        r14(WriteData[14], CLK, ReadData[14]),
        r15(WriteData[15], CLK, ReadData[15]);
endmodule

//2 bit 2x1 mulitplexer
module mux2bit2x1(A,B,select,OUT);
        input [1:0] A,B;
input select;
        output [1:0] OUT;

mux2x1 m1(A[0], B[0], select, OUT[0]),
       m2(A[1], B[1], select, OUT[1]);
endmodule
```

```
// 16 bit 2x1 multiplexer

module mux16bit2x1(A, B, select, OUT);

            input [15:0] A,B;

    input select;

            output [15:0] OUT;



    mux2x1 mux1(A[0], B[0], select, OUT[0]),

           mux2(A[1], B[1], select, OUT[1]),

           mux3(A[2], B[2], select, OUT[2]),

           mux4(A[3], B[3], select, OUT[3]),

           mux5(A[4], B[4], select, OUT[4]),

           mux6(A[5], B[5], select, OUT[5]),

           mux7(A[6], B[6], select, OUT[6]),

           mux8(A[7], B[7], select, OUT[7]),

           mux9(A[8], B[8], select, OUT[8]),

           mux10(A[9], B[9], select, OUT[9]),

           mux11(A[10], B[10], select, OUT[10]),

           mux12(A[11], B[11], select, OUT[11]),

           mux13(A[12], B[12], select, OUT[12]),

           mux14(A[13], B[13], select, OUT[13]),

           mux15(A[14], B[14], select, OUT[14]),

           mux16(A[15], B[15], select, OUT[15]);

endmodule
```

```
// 16 bit 4x1 muliplexer

module mux4x1_16bit(i0,i1,i2,i3,select,y);

        input [15:0] i0,i1,i2,i3;

        input [1:0] select;

        output [15:0] y;


        mux4x1 mux1 (1'b0,i1[0], i2[0], i3[0], select[1:0],y[0]),

                   mux2 (1'b0,i1[1], i2[1], i3[1], select[1:0],y[1]),

                   mux3 (1'b0,i1[2], i2[2], i3[2], select[1:0],y[2]),

                   mux4 (1'b0,i1[3], i2[3], i3[3], select[1:0],y[3]),


                   mux5 (1'b0,i1[4], i2[4], i3[4], select[1:0],y[4]),

                   mux6 (1'b0,i1[5], i2[5], i3[5], select[1:0],y[5]),

                   mux7 (1'b0,i1[6], i2[6], i3[6], select[1:0],y[6]),

                   mux8 (1'b0,i1[7], i2[7], i3[7], select[1:0],y[7]),


                   mux9 (1'b0,i1[8], i2[8], i3[8], select[1:0],y[8]),

                   mux10(1'b0,i1[9], i2[9], i3[9], select[1:0],y[9]),

                   mux11(1'b0,i1[10],i2[10],i3[10],select[1:0],y[10]),

                   mux12(1'b0,i1[11],i2[11],i3[11],select[1:0],y[11]),


                   mux13(1'b0,i1[12],i2[12],i3[12],select[1:0],y[12]),

                   mux14(1'b0,i1[13],i2[13],i3[13],select[1:0],y[13]),

                   mux15(1'b0,i1[14],i2[14],i3[14],select[1:0],y[14]),

                   mux16(1'b0,i1[15],i2[15],i3[15],select[1:0],y[15]);

endmodule
```

```
// decoder
module decoder (S1,S0,D3,D2,D1,D0);
  input S0,S1;
  output D0,D1,D2,D3;

  not n1 (notS0,S0),
      n2 (notS1,S1);

  and a0 (D0,notS1,notS0),
      a1 (D1,notS1,   S0),
      a2 (D2,   S1,notS0),
      a3 (D3,   S1,   S0);
endmodule

// D-flip flop
module D_flip_flop(D,CLK,Q);

        input D,CLK;
        output Q;
        wire CLK1,Y;
        not not1 (CLK1,CLK);

        D_latch D1(D,CLK,Y),
                        D2(Y,CLK1,Q);
```

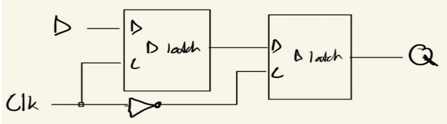
```
endmodule


// D-latch
module D_latch(D,C,Q);
        input D,C;
        output Q;
        wire x,y,D1,Q1;

        nand nand1(x,D,C),
                nand2(y,D1,C),
                nand3(Q,x,Q1),
                nand4(Q1,y,Q);

        not not1(D1,D);
endmodule


// 16 bit D Flip Flop
module D_16_Flip_flop(D,CLK,Q);
        input [15:0] D;
        input CLK;
        output [15:0] Q;

        D_flip_flop f0(D[0], CLK, Q[0]),
                            f1(D[1], CLK, Q[1]),
                            f2(D[2], CLK, Q[2]),
                            f3(D[3], CLK, Q[3]),
                            f4(D[4], CLK, Q[4]),
                            f5(D[5], CLK, Q[5]),
                            f6(D[6], CLK, Q[6]),
                            f7(D[7], CLK, Q[7]),
                            f8(D[8], CLK, Q[8]),
                            f9(D[9], CLK, Q[9]),
                            f10(D[10], CLK, Q[10]),
                            f11(D[11], CLK, Q[11]),
                            f12(D[12], CLK, Q[12]),
                            f13(D[13], CLK, Q[13]),
                            f14(D[14], CLK, Q[14]),
                            f15(D[15], CLK, Q[15]);
endmodule


// Alu/Main Control
module mainctrl(Op,Control);
  input [3:0] Op;
  output reg [6:0] Control;
        always @(Op) case (Op)
              4'b0000: Control <= 7'b101_0010;  //add
              4'b0001: Control <= 7'b101_0110;  //sub
              4'b0010: Control <= 7'b101_0000;  //and
              4'b0011: Control <= 7'b101_0001;  //or
              4'b0100: Control <= 7'b101_1100;  //nor
              4'b0101: Control <= 7'b011_1010;  //nand
              4'b0110: Control <= 7'b101_0111;  //slt
```

```
                    4'b0111: Control <= 7'b011_0010;  //addi
                    //4'b1000: Control <= 6'b10_1001;  //lw
                    //4'b1001: Control <= 6'b10_1100;  //sw
                    //4'b1010: Control <= 6'b10_1111;  //beq
                    //4'b1011: Control <= 6'b01_1010;  //bne
          endcase
endmodule


// 16 bit CPU
module CPU (clock,PC,ALUOut,IR);
  input clock;
  output [15:0] ALUOut,IR,PC;
  reg[15:0] PC;
  reg[15:0] IMemory[0:1023];
  wire [15:0] IR,NextPC,A,B,ALUOut,RD2,SignExtend;
  wire [3:0] ALUctl;
  //wire [2:0] ALUOp;
  wire [1:0] WR;
// Test Program
  initial begin
        //                              Assembly     | Result |      Binary IR
       //                          ---------------------------------------------------
       IMemory[0] = 16'b0111_00_01_00001111;   // addi $t1, $0,  15   ($t1=15)  0111 00 01
00001111
       IMemory[1] = 16'b0111_00_10_00000111;   // addi $t2, $0,  7    ($t2= 7)  0111 00 10
00000111
```

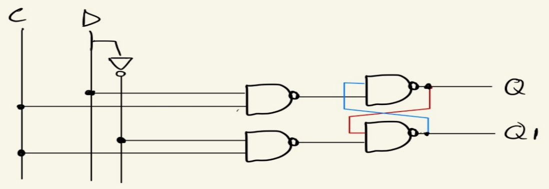
```
        IMemory[2] = 16'b0010_01_10_11_000000;  // and  $t3, $t1, $t2  ($t3= 7)  0010 01 10 11
xxxxxx
        IMemory[3] = 16'b0001_01_11_10_000000;  // sub  $t2, $t1, $t3  ($t2= 8)  0001 01 11 10
xxxxxx
        IMemory[4] = 16'b0011_10_11_10_000000;  // or   $t2, $t2, $t3  ($t2=15)  0011 10 11 10
xxxxxx
        IMemory[5] = 16'b0000_10_11_11_000000;  // add  $t3, $t2, $t3  ($t3=22)  0000 10 11 11
xxxxxx
        IMemory[6] = 16'b0100_10_11_01_000000;  // nor  $t1, $t2, $t3  ($t1=-32) 0100 10 11 01
xxxxxx
        IMemory[7] = 16'b0110_10_11_01_000000;  // slt  $t1, $t3, $t2  ($t1= 0)  0110 11 10 01
xxxxxx
        IMemory[8] = 16'b0110_11_10_01_000000;  // slt  $t1, $t2, $t3  ($t1= 1)  0110 10 11 01
xxxxxx
    end
  initial PC = 0;
  assign IR = IMemory[PC>>2];
  //assign WR = (RegDst) ? IR[7:6]: IR[9:8]; // RegDst Mux
  //assign B  = (ALUSrc) ? SignExtend: RD2; // ALUSrc Mux

  //assign wr
   mux2bit2x1 mx4(IR[9:8],IR[7:6],RegDst,WR);
  //assign B
   mux16bit2x1 mx2(RD2,SignExtend,ALUSrc,B);

  assign SignExtend = {{8{IR[7]}},IR[7:0]}; // sign extension unit
  reg_file rf (IR[11:10], IR[9:8], WR, ALUOut, RegWrite, A, RD2, clock);
  ALU fetch (4'b0010, PC, 16'b100, NextPC, Unused);
   ALU ex (ALUctl, A, B, ALUOut, Zero);

  //MainControl MainCtr (IR[15:12],{RegDst,ALUSrc,RegWrite,ALUOp});
  //ALUControl ALUCtrl(ALUOp, IR[5:0], ALUctl); // ALU control unit
          mainctrl ainCtr (IR[15:12],{RegDst,ALUSrc,RegWrite,ALUctl});
          //                            1     0      1      1110

  always @(negedge clock) begin
    PC <= NextPC;
  end
endmodule

//Test Module
module test ();
  reg clock;
  wire signed [15:0] WD,IR,PC;

  CPU test_cpu(clock,PC,WD,IR);
  always #1 clock = ~clock;
  initial begin
    $display ("Clock PC   IR                                  WD");
    $monitor ("%b     %2d   %b  %3d (%b)",clock,PC,IR,WD,WD);
    clock = 1;
    #16 $finish;
  end
endmodule


/* 32bit  Output
Clock PC   IR                               WD
1      0   00100000000010010000000000001111   15 (00000000000000000000000000001111)
0      4   00100000000010100000000000000111    7 (00000000000000000000000000000111)
1      4   00100000000010100000000000000111    7 (00000000000000000000000000000111)
0      8   00000001001010100101100000100100    7 (00000000000000000000000000000111)
1      8   00000001001010100101100000100100    7 (00000000000000000000000000000111)
0     12   00000001001010110101000000100010    8 (00000000000000000000000000001000)
1     12   00000001001010110101000000100010    8 (00000000000000000000000000001000)
0     16   00000001010010110101000000100101   15 (00000000000000000000000000001111)
1     16   00000001010010110101000000100101   15 (00000000000000000000000000001111)
0     20   00000001010010110101100000100000   22 (00000000000000000000000000010110)
1     20   00000001010010110101100000100000   22 (00000000000000000000000000010110)
0     24   00000001010010110100100000100111  -32 (11111111111111111111111111100000)
1     24   00000001010010110100100000100111  -32 (11111111111111111111111111100000)
0     28   00000001011010100100100000101010    0 (00000000000000000000000000000000)
1     28   00000001011010100100100000101010    0 (00000000000000000000000000000000)
0     32   00000001010010110100100000101010    1 (00000000000000000000000000000001)
1     32   00000001010010110100100000101010    1 (00000000000000000000000000000001)
*/


/*
Current output- 16 bits
```

```
Clock PC   IR                               WD

1      0   0111000100001111   15 (0000000000001111)

0      4   0111001000000111    7 (0000000000000111)

1      4   0111001000000111    7 (0000000000000111)

0      8   0010011011000000    7 (0000000000000111)

1      8   0010011011000000    7 (0000000000000111)

0     12   0001011110000000    8 (0000000000001000)

1     12   0001011110000000    8 (0000000000001000)

0     16   0011101110000000   15 (0000000000001111)

1     16   0011101110000000   15 (0000000000001111)

0     20   0000101111000000   22 (0000000000010110)

1     20   0000101111000000   22 (0000000000010110)

0     24   0100101101000000  -32 (1111111111100000)

1     24   0100101101000000  -32 (1111111111100000)

0     28   0110101101000000    0 (0000000000000000)

1     28   0110101101000000    0 (0000000000000000)

0     32   0110111001000000    1 (0000000000000001)

1     32   0110111001000000    1 (0000000000000001)

*/
```

D flip flop

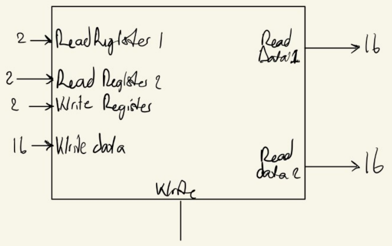


D Latch



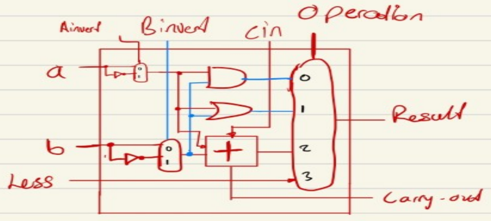
Register

Reg file - 16 bits

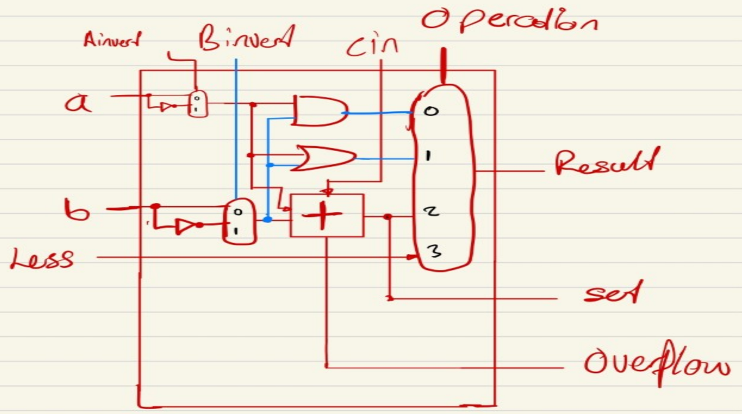


ALU - 1 bit

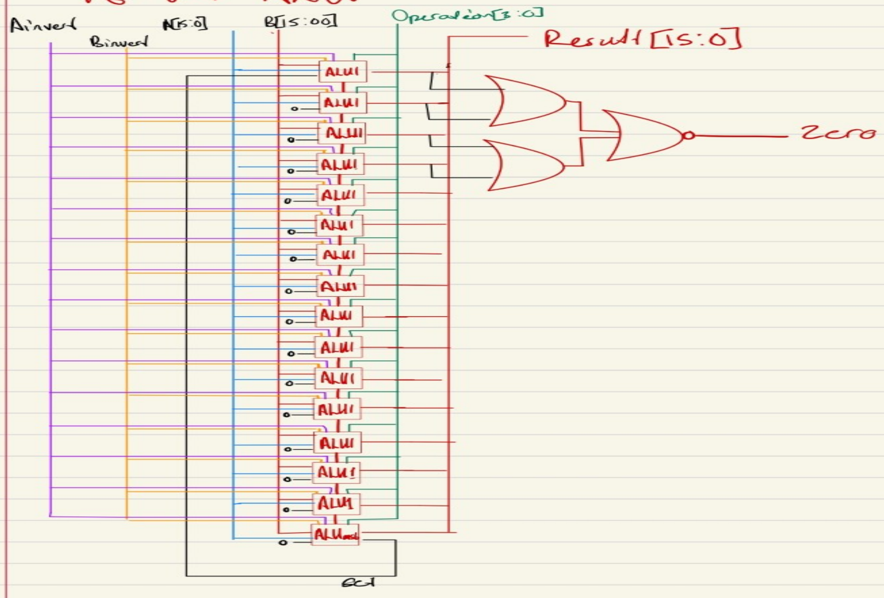
ALU1



ALUmsb



16 bit ALU


Ainvert
Binvert
A[15:0]
B[15:00]
Operation[3:0]
Result[15:0]
ALU1
ALUmsb
zero
set