```
1   /*
2      ALU instantiates all computational units and selects the desired result based on a
    respective control signal.
3   */
4
5   /*
6      DESIGN DECISION:
7      We chose to *vectorize* the control signals for design clarity.
8      Therefore, signals like 'add', 'sub', etc. are grouped as ALUopp, which is one-hot
    encoded.
9      We use the following const. definitions to refer to each control signal under an
    indexable framework.
10  */
11
12  `define ADD 0
13  `define SUB 1
14  `define NEG 2
15  `define MUL 3
16  `define DIV 4
17  `define AND 5
18  `define OR  6
19  `define ROR 7
20  `define ROL 8
21  `define SLL 9
22  `define SRA 10
23  `define SRL 11
24  `define NOT 12
25  `define INC 13
26
27
28  module ALU (
29     input [31:0] x, y,
30     input [15:0] ALUopp,
31     input clk, // for division algorithm
32     output reg [63:0] Z
33     );
34
35     //------------- ADD/SUB/NEG/INC -------------//
36
37     wire [31:0] adder_operand1, adder_operand2, input_to_XOR;
38     assign input_to_XOR = (ALUopp[`NEG]) ? x : y;
39
40     assign adder_operand1 = (ALUopp[`NEG]) ? 32'b0 :
41                             (ALUopp[`INC]) ? 32'b1 : x;
42
43     assign adder_operand2 = input_to_XOR ^ {32{ALUopp[`SUB] | ALUopp[`NEG]}};
44
45     wire [31:0] adder_result;
46
47     // 32-bit CLA instance that covers all four instructions through careful selection of
    operands.
48     adder_32b add (.x(adder_operand1), .y(adder_operand2), .cin(ALUopp[`SUB] | ALUopp[`NEG]),
     .s(adder_result), .cout());
49
50     //------------- MUL -------------//
51
52     wire [63:0] mult_result;
53     multiplier_32b mul (.M(x), .Q(y), .result(mult_result));
54
55     //------------- DIV -------------//
56
57     wire [63:0] div_result;
58     DIV divider(.Q(x), .M(y), .clk(clk), .resetn(ALUopp[`DIV]), .quotient(div_result[31:0]),
    .remainder(div_result[63:32]));
59
60     // Choose ALU Operation of interest
61
62     always @(*) begin
63        if (ALUopp[`ADD] | ALUopp[`SUB] | ALUopp[`NEG] | ALUopp[`INC])
64           Z = adder_result;
65        else if (ALUopp[`MUL])
66           Z = mult_result;
67        else if (ALUopp[`DIV])
68           Z = div_result; // TO BE UPDATED
```

```
        else if (ALUopp[`AND])
            Z = x & y;
        else if (ALUopp[`OR])
            Z = x | y;
        else if (ALUopp[`ROR])
            Z = {x[0], x[31:1]};
        else if (ALUopp[`ROL])
            Z = {x[30:0], x[31]};
        else if (ALUopp[`SLL])
            Z = {x[30:0], 1'b0};
        else if (ALUopp[`SRA])
            Z = {x[31], x[31:1]};
        else if (ALUopp[`SRL])
            Z = {1'b0, x[31:1]};
        else if (ALUopp[`NOT])
            Z = ~x;
        else
            Z = 64'b0;
    end
endmodule


`timescale 1ns / 1ps

module ALU_tb;

    reg [31:0] x, y;
    reg [15:0] ALUopp;
    reg clk;

    wire [63:0] Z;

    ALU dut (x, y, ALUopp, clk, Z);

    initial begin
      clk <= 0;
      forever #(5) clk <= ~clk;
    end

    // Task to test a single operation
    task test_op(input [15:0] op, input [31:0] a, input [31:0] b);
        begin
            ALUopp = op;
            x = a;
            y = b;
            #10; // Wait for operation to complete
        end
    endtask

    // Test procedure
    initial begin

        test_op(1 << `ADD, 32'd10, 32'd5);   // ADD: 10 + 5 = 15
        test_op(1 << `SUB, 32'd15, 32'd5);   // SUB: 15 - 5 = 10
        test_op(1 << `NEG, 32'd7, 32'd0);    // NEG: -7
        test_op(1 << `MUL, 32'd4, 32'd3);    // MUL: 4 * 3 = 12
        test_op(1 << `DIV, 32'd20, 32'd5);   // DIV: 20 / 5 = 4
        #340;
        test_op(1 << `AND, 32'hF0F0F0F0, 32'h0F0F0F0F); // AND
        test_op(1 << `OR,  32'hF0F0F0F0, 32'h0F0F0F0F); // OR
        test_op(1 << `ROR, 32'h80000001, 0); // Rotate Right
        test_op(1 << `ROL, 32'h40000000, 0); // Rotate Left
        test_op(1 << `SLL, 32'h00000001, 2); // Shift Left Logical
        test_op(1 << `SRA, 32'h80000000, 2); // Shift Right Arithmetic
        test_op(1 << `SRL, 32'h80000000, 2); // Shift Right Logical
        test_op(1 << `NOT, 32'hAAAAAAAA, 0); // NOT
        $stop;
    end
endmodule

```