```
/*
   ALU instantiates all computational units and selects the desired result based on a respective control signal.
*/

/*
   DESIGN DECISION:
   We chose to *vectorize* the control signals for design clarity.
   Therefore, signals like 'add', 'sub', etc. are grouped as ALUopp, which is one-hot encoded.
   We use the following const. definitions to refer to each control signal under an indexable framework.
*/

`define ADD 0
`define SUB 1
`define NEG 2
`define MUL 3
`define DIV 4
`define AND 5
`define OR  6
`define ROR 7
`define ROL 8
`define SLL 9
`define SRA 10
`define SRL 11
`define NOT 12
`define INC 13


module ALU (
   input [31:0] x, y,
   input [15:0] ALUopp,
   input clk, // for division algorithm
   output reg [63:0] Z
   );

   //------------- ADD/SUB/NEG/INC -------------//

   wire [31:0] adder_operand1, adder_operand2, input_to_XOR;
   assign input_to_XOR = (ALUopp[`NEG]) ? x : y;

   assign adder_operand1 = (ALUopp[`NEG]) ? 32'b0 :
                           (ALUopp[`INC]) ? 32'b1 : x;

   assign adder_operand2 = input_to_XOR ^ {32{ALUopp[`SUB] | ALUopp[`NEG]}};

   wire [31:0] adder_result;

   // 32-bit CLA instance that covers all four instructions through careful selection of operands.
   adder_32b add (.x(adder_operand1), .y(adder_operand2), .cin(ALUopp[`SUB] | ALUopp[`NEG]), .s(adder_result), .cout());

   //------------- MUL -------------//

   wire [63:0] mult_result;
   multiplier_32b mul (.M(x), .Q(y), .result(mult_result));

   //------------- DIV -------------//

   //wire [63:0] div_result;
   //DIV divider(.Q(x), .M(y), .clk(clk), .resetn(ALUopp[`DIV]), .quotient(div_result[31:0]), .remainder(div_result[63:32]));

   //------------- Shift and Rotate -------------//

   // BARREL SHIFT/ROTATE Design.
   reg [31:0] shift_result;
   always @(*) begin
      shift_result = x;
      if (ALUopp[`SLL]) begin // Barrel shift left
         if (y[4]) shift_result = {shift_result[15:0], 16'b0};
         if (y[3]) shift_result = {shift_result[23:0], 8'b0};
         if (y[2]) shift_result = {shift_result[27:0], 4'b0};
         if (y[1]) shift_result = {shift_result[29:0], 2'b0};
```

```
            if (y[0]) shift_result = {shift_result[30:0], 1'b0};
         end
         else
         if (ALUopp[`SRL]) begin // Barrel shift right
            if (y[4]) shift_result = {16'b0, shift_result[31:16]};
            if (y[3]) shift_result = {8'b0, shift_result[31:8]};
            if (y[2]) shift_result = {4'b0, shift_result[31:4]};
            if (y[1]) shift_result = {2'b0, shift_result[31:2]};
            if (y[0]) shift_result = {1'b0, shift_result[31:1]};
         end
         else
         if (ALUopp[`SRA]) begin // Barrel shift right arithmetic (>>>)
            if (y[4]) shift_result = {{16{shift_result[31]}}, shift_result[31:16]};
            if (y[3]) shift_result = {{8{shift_result[31]}}, shift_result[31:8]};
            if (y[2]) shift_result = {{4{shift_result[31]}}, shift_result[31:4]};
            if (y[1]) shift_result = {{2{shift_result[31]}}, shift_result[31:2]};
            if (y[0]) shift_result = {shift_result[31], shift_result[31:1]};
         end
         else
         if (ALUopp[`ROR]) begin // Barrel rotate right
            if (y[4]) shift_result = {shift_result[15:0], shift_result[31:16]};
            if (y[3]) shift_result = {shift_result[7:0], shift_result[31:8]};
            if (y[2]) shift_result = {shift_result[3:0], shift_result[31:4]};
            if (y[1]) shift_result = {shift_result[1:0], shift_result[31:2]};
            if (y[0]) shift_result = {shift_result[0], shift_result[31:1]};
         end
         else
         if (ALUopp[`ROL]) begin // Barrel rotate left
            if (y[4]) shift_result = {shift_result[15:0], shift_result[31:16]};
            if (y[3]) shift_result = {shift_result[23:0], shift_result[31:24]};
            if (y[2]) shift_result = {shift_result[27:0], shift_result[31:28]};
            if (y[1]) shift_result = {shift_result[29:0], shift_result[31:30]};
            if (y[0]) shift_result = {shift_result[30:0], shift_result[31]};
         end

      end

      // Choose ALU Operation of interest, based on control signal ALUopp.

      always @(*) begin
         if (ALUopp[`ADD] | ALUopp[`SUB] | ALUopp[`NEG] | ALUopp[`INC])
            Z = adder_result;
         else if (ALUopp[`MUL])
            Z = mult_result;
         else if (ALUopp[`DIV])
            Z = {x % y, x / y};
         else if (ALUopp[`AND])
            Z = x & y;
         else if (ALUopp[`OR])
            Z = x | y;
         else if (ALUopp[`SLL] | ALUopp[`SRA] | ALUopp[`SRL] | ALUopp[`ROR] | ALUopp[`ROL])
            Z = shift_result;
         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 signed [31:0] a, input signed [31:0] b, input signed [31:0] expected_result);
        begin
            ALUopp = op;
            x = a;
            y = b;
            #10; // Wait for operation to complete

            if ($signed(Z) !== expected_result) begin
                $display("Test Failed: op=%d, a=%d, b=%d -> Expected result=%d but got result=%d",
                        op, a, b, expected_result, Z);
            end
        end
    endtask

    // Test procedure
    initial begin
        test_op(1 << `ADD,  32'sd10,  32'sd5,  32'sd15);   // ADD: 10 + 5 = 15
        test_op(1 << `SUB,  32'sd15,  32'sd5,  32'sd10);   // SUB: 15 - 5 = 10
        test_op(1 << `NEG,  32'sd7,   32'sd0, -32'sd7);    // NEG: -7
        test_op(1 << `MUL,  32'sd4,   32'sd3,  32'sd12);   // MUL: 4 * 3 = 12
        test_op(1 << `DIV,  32'sd20,  32'sd5,  32'sd4);    // DIV: 20 / 5 = 4

        #340; // Wait before bitwise and shift operations

        test_op(1 << `AND,  32'hF0F0F0F0, 32'h0F0F0F0F, 32'h00000000); // AND
        test_op(1 << `OR,   32'hF0F0F0F0, 32'h0F0F0F0F, 32'hFFFFFFFF); // OR
        test_op(1 << `ROR,  32'h80000001, 32'd1,        32'hC0000000); // Rotate Right (example)
        test_op(1 << `ROL,  32'h40000000, 32'd1,        32'h80000000); // Rotate Left
        test_op(1 << `SLL,  32'h00000001, 32'd2,        32'h00000004); // Shift Left Logical
        test_op(1 << `SRA,  32'h80000000, 32'd2,        32'hE0000000); // Shift Right Arithmetic
        test_op(1 << `SRL,  32'h80000000, 32'd2,        32'h20000000); // Shift Right Logical
        test_op(1 << `NOT,  32'hAAAAAAAA, 32'd0,        32'h55555555); // NOT

        $display("Testbench completed successfully");

        $stop;
    end

endmodule
```