

```

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), .resets(ALUopp[`DIV]), .quotient(div_result[31:0]),
   .remainder(div_result[63:32]));
59
60 //----- Shift and Rotate -----//
61
62 // BARREL SHIFT/ROTATE Design.
63 reg [31:0] shift_result;
64 always @(*) begin
65     shift_result = x;
66     if (ALUopp[`SLL]) begin
67         if (y[4]) shift_result = shift_result << 16;
68         if (y[3]) shift_result = shift_result << 8;
69         if (y[2]) shift_result = shift_result << 4;

```

```

70         if (y[1]) shift_result = shift_result << 2;
71         if (y[0]) shift_result = shift_result << 1;
72     end
73     else
74         if (ALUopp[`SRL]) begin
75             if (y[4]) shift_result = shift_result >> 16;
76             if (y[3]) shift_result = shift_result >> 8;
77             if (y[2]) shift_result = shift_result >> 4;
78             if (y[1]) shift_result = shift_result >> 2;
79             if (y[0]) shift_result = shift_result >> 1;
80         end
81     else
82         if (ALUopp[`SRA]) begin
83             if (y[4]) shift_result = shift_result >>> 16;
84             if (y[3]) shift_result = shift_result >>> 8;
85             if (y[2]) shift_result = shift_result >>> 4;
86             if (y[1]) shift_result = shift_result >>> 2;
87             if (y[0]) shift_result = shift_result >>> 1;
88         end
89     else
90         if (ALUopp[`ROR]) begin
91             if (y[4]) shift_result = {shift_result[15:0], shift_result[31:16]};
92             if (y[3]) shift_result = {shift_result[7:0], shift_result[31:8]};
93             if (y[2]) shift_result = {shift_result[3:0], shift_result[31:4]};
94             if (y[1]) shift_result = {shift_result[1:0], shift_result[31:2]};
95             if (y[0]) shift_result = {shift_result[0], shift_result[31:1]};
96         end
97     else
98         if (ALUopp[`ROL]) begin
99             if (y[4]) shift_result = {shift_result[15:0], shift_result[31:16]};
100            if (y[3]) shift_result = {shift_result[23:0], shift_result[31:24]};
101            if (y[2]) shift_result = {shift_result[27:0], shift_result[31:28]};
102            if (y[1]) shift_result = {shift_result[29:0], shift_result[31:30]};
103            if (y[0]) shift_result = {shift_result[30:0], shift_result[31]};
104        end
105    end
106 end
107
108 // Choose ALU Operation of interest
109
110 always @(*) begin
111     if (ALUopp[`ADD] | ALUopp[`SUB] | ALUopp[`NEG] | ALUopp[`INC])
112         Z = adder_result;
113     else if (ALUopp[`MUL])
114         Z = mult_result;
115     else if (ALUopp[`DIV])
116         Z = div_result;
117     else if (ALUopp[`AND])
118         Z = x & y;
119     else if (ALUopp[`OR])
120         Z = x | y;
121     else if (ALUopp[`SLL] | ALUopp[`SLL] | ALUopp[`SLL] | ALUopp[`ROR] | ALUopp[`ROL])
122         Z = shift_result;
123     else if (ALUopp[`NOT])
124         Z = ~x;
125     else
126         Z = 64'b0;
127 end
128 endmodule
129
130
131 `timescale 1ns / 1ps
132
133 module ALU_tb;
134
135     reg [31:0] x, y;
136     reg [15:0] ALUopp;
137     reg clk;
138
139     wire [63:0] Z;
140
141     ALU dut (x, y, ALUopp, clk, Z);
142
143     initial begin
144         clk <= 0;

```

```
145     forever #(5) clk <= ~clk;
146 end
147
148 // Task to test a single operation
149 task test_op(input [15:0] op, input [31:0] a, input [31:0] b);
150     begin
151         ALUopp = op;
152         x = a;
153         y = b;
154         #10; // wait for operation to complete
155     end
156 endtask
157
158 // Test procedure
159 initial begin
160
161     test_op(1 << `ADD, 32'd10, 32'd5); // ADD: 10 + 5 = 15
162     test_op(1 << `SUB, 32'd15, 32'd5); // SUB: 15 - 5 = 10
163     test_op(1 << `NEG, 32'd7, 32'd0); // NEG: -7
164     test_op(1 << `MUL, 32'd4, 32'd3); // MUL: 4 * 3 = 12
165     test_op(1 << `DIV, 32'd20, 32'd5); // DIV: 20 / 5 = 4
166     #340;
167     test_op(1 << `AND, 32'hF0F0F0F0, 32'h0F0F0F0F); // AND
168     test_op(1 << `OR, 32'hF0F0F0F0, 32'h0F0F0F0F); // OR
169     test_op(1 << `ROR, 32'h80000001, 0); // Rotate Right
170     test_op(1 << `ROL, 32'h40000000, 0); // Rotate Left
171     test_op(1 << `SLL, 32'h00000001, 2); // Shift Left Logical
172     test_op(1 << `SRA, 32'h80000000, 2); // Shift Right Arithmetic
173     test_op(1 << `SRL, 32'h80000000, 2); // Shift Right Logical
174     test_op(1 << `NOT, 32'hAAAAAAAA, 0); // NOT
175     $stop;
176 end
177 endmodule
178
```