Simulation Screenshots

Figure 1 displays the output produced using the datapath testbench that uses the datapath module to preform operations. The group struggled to display the register values and operation behaviour through this module. To display the correct functionality of the ALU operations, the ALU testbench, as seen in Figure 33, was used to produce simulated results shown in Figure 2 to Figure 13.

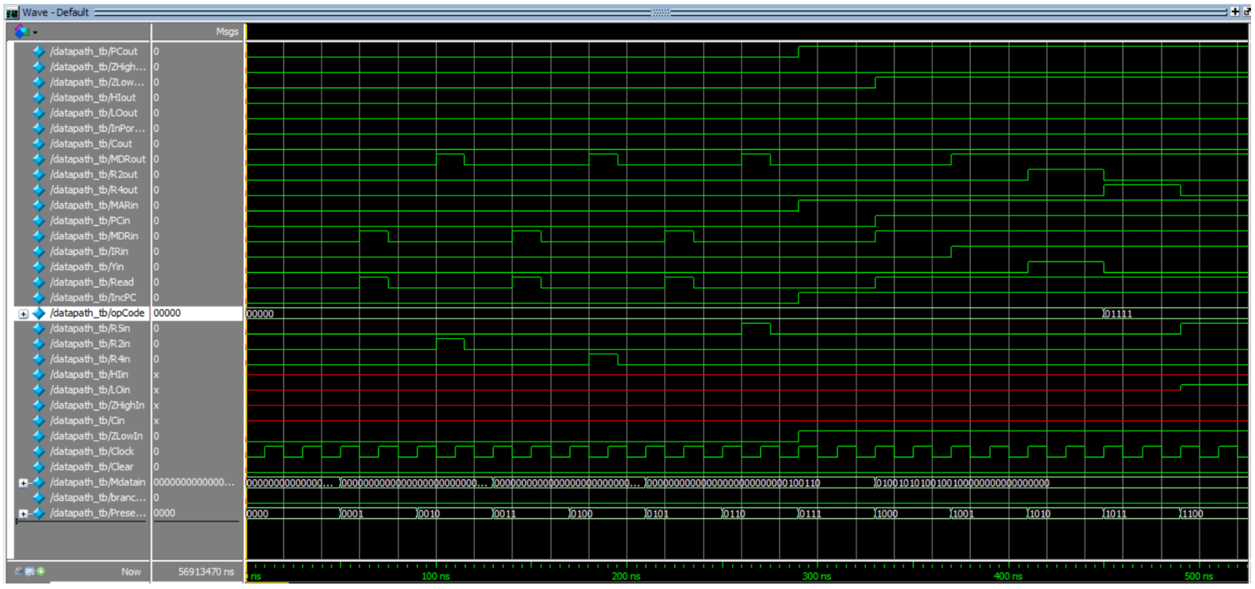
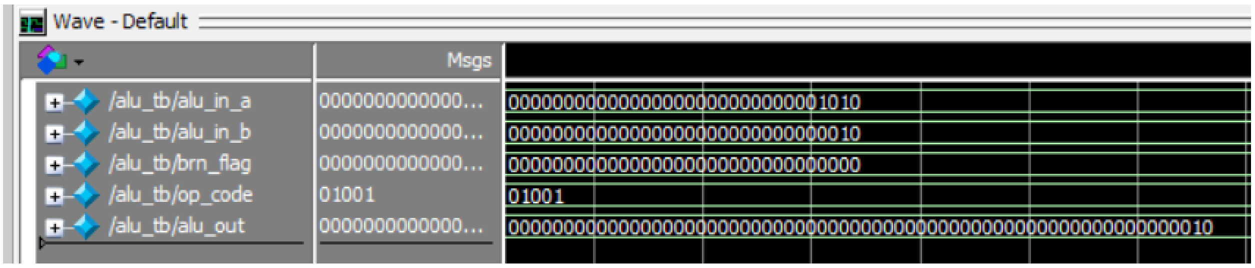


Figure 1: The datapath testbench output

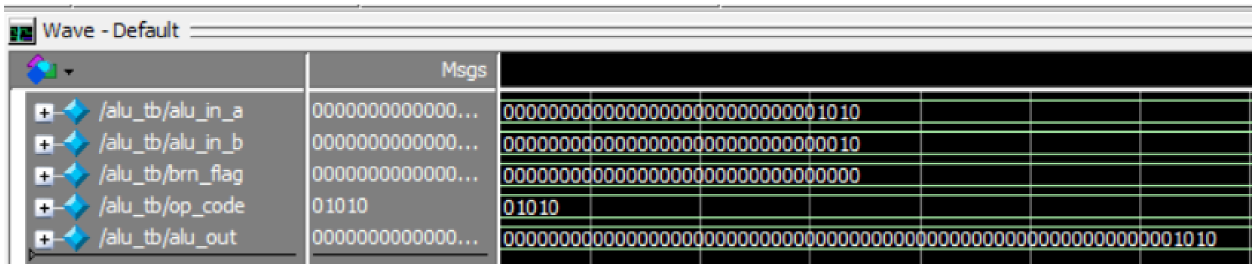
Logical AND



```
Wave - Default
                    Msgs
/alu_tb/alu_in_a    0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b    0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag    0000000000000...   00000000000000000000000000000000
/alu_tb/op_code     01001              01001
/alu_tb/alu_out     0000000000000...   0000000000000000000000000000000000000000000000000000000000000010
```

Figure 2: Signals for the logical and operation, a and b, with the corresponding output signal, 'out'

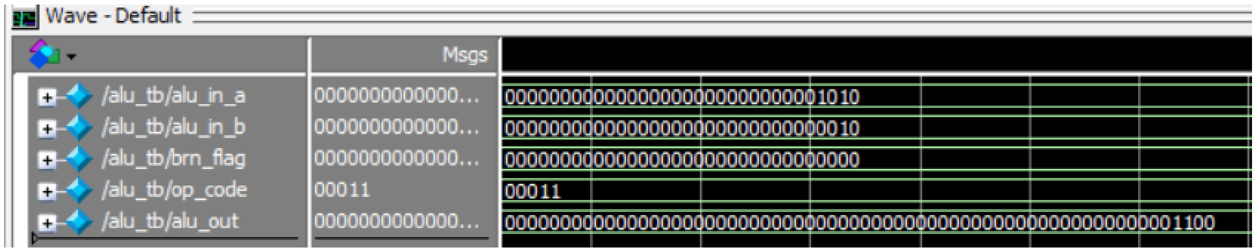
Logical OR



```
Wave - Default
                    Msgs
/alu_tb/alu_in_a    0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b    0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag    0000000000000...   00000000000000000000000000000000
/alu_tb/op_code     01010              01010
/alu_tb/alu_out     0000000000000...   0000000000000000000000000000000000000000000000000000000000001010
```

Figure 3: Signals for the logical or operation, a or b, with the corresponding output signal, 'out'

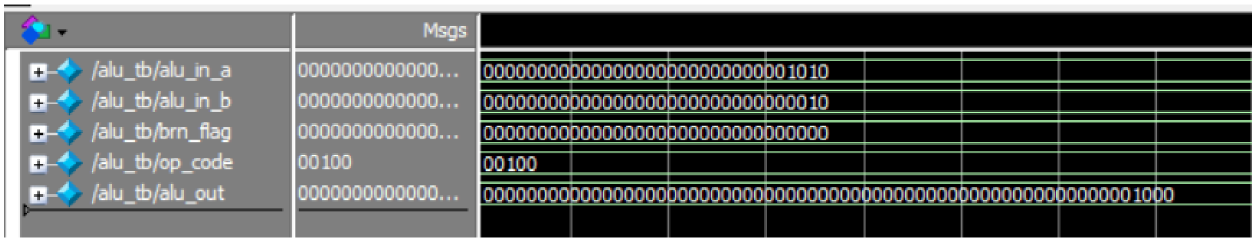
Add (ADD)



```
Wave - Default
                         Msgs
/alu_tb/alu_in_a   0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b   0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag   0000000000000...   00000000000000000000000000000000
/alu_tb/op_code    00011              00011
/alu_tb/alu_out    0000000000000...   0000000000000000000000000000000000000000000000000000000000001100
```

Figure 34:Ssignals for the addition operation, a + b, with the corresponding output signal, 'out'

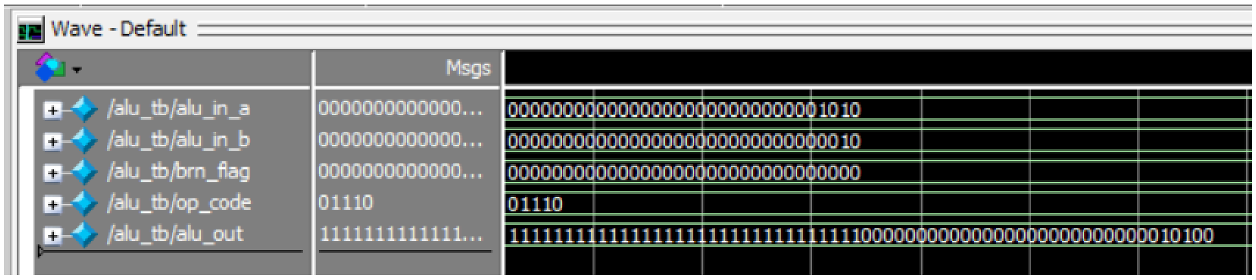
Subtract (SUB)



```
                         Msgs
/alu_tb/alu_in_a   0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b   0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag   0000000000000...   00000000000000000000000000000000
/alu_tb/op_code    00100              00100
/alu_tb/alu_out    0000000000000...   0000000000000000000000000000000000000000000000000000000000001000
```

Figure 5: Signals for the subtract operation, a - b with the corresponding output signal, 'out'

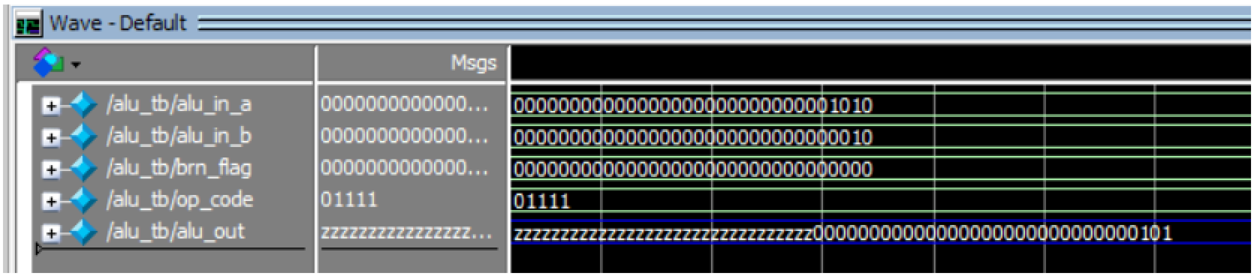
Multiply (MUL)



```
Wave - Default
                         Msgs
/alu_tb/alu_in_a   0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b   0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag   0000000000000...   00000000000000000000000000000000
/alu_tb/op_code    01110              01110
/alu_tb/alu_out    1111111111111...   1111111111111111111111111111111100000000000000000000000000010100
```

*Figure 6: Signals for the multiplication operation, a * b, with the corresponding output signal, 'out'*

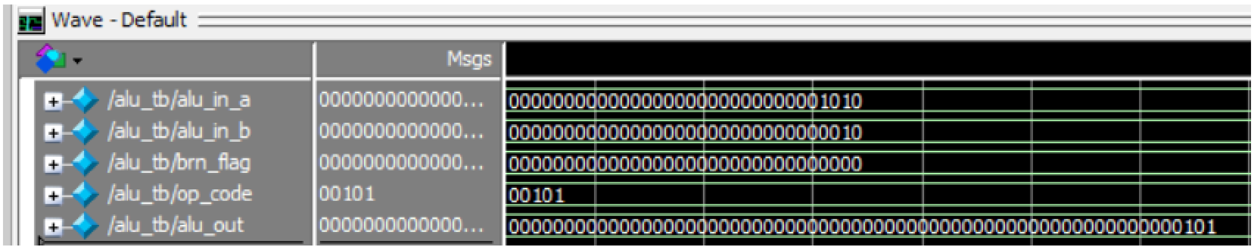
Division (DIV)



```
Wave - Default
                         Msgs
/alu_tb/alu_in_a   0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b   0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag   0000000000000...   00000000000000000000000000000000
/alu_tb/op_code    01111              01111
/alu_tb/alu_out    zzzzzzzzzzzzzzzz...   zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz00000000000000000000000000000101
```

Figure 7: Signals for the division operation, a / b with the corresponding output signal, 'out'

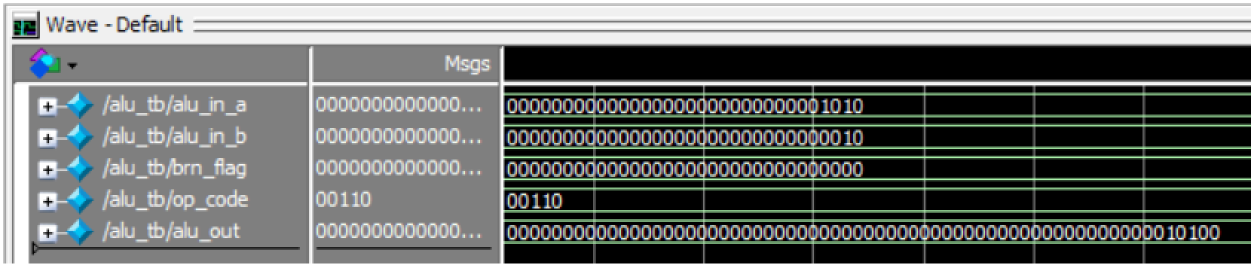
Shift Right (SHR)



```
Wave - Default
                     Msgs
/alu_tb/alu_in_a     0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b     0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag     0000000000000...   00000000000000000000000000000000
/alu_tb/op_code      00101              00101
/alu_tb/alu_out      0000000000000...   0000000000000000000000000000000000000000000000000000000000000101
```

Figure 8: Signals for the shift right operation, a shifted right with the corresponding output signal, 'out'

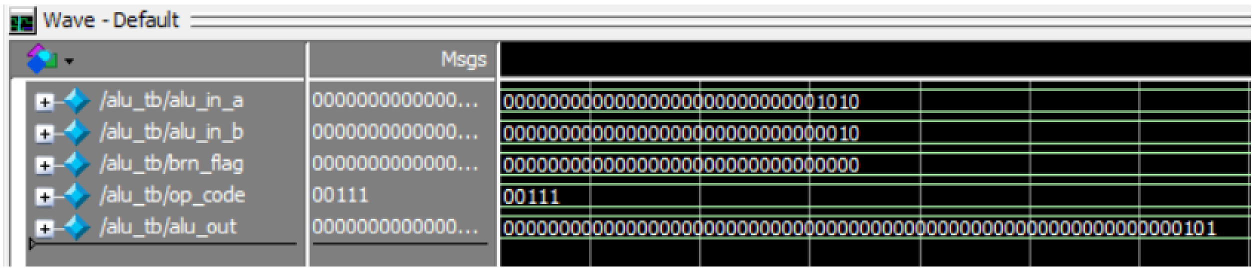
Shift Left (SHL)



```
Wave - Default
                     Msgs
/alu_tb/alu_in_a     0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b     0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag     0000000000000...   00000000000000000000000000000000
/alu_tb/op_code      00110              00110
/alu_tb/alu_out      0000000000000...   0000000000000000000000000000000000000000000000000000000000010100
```

Figure 9: Signals for the shift left operation, a shifted left with the corresponding output signal, 'out'

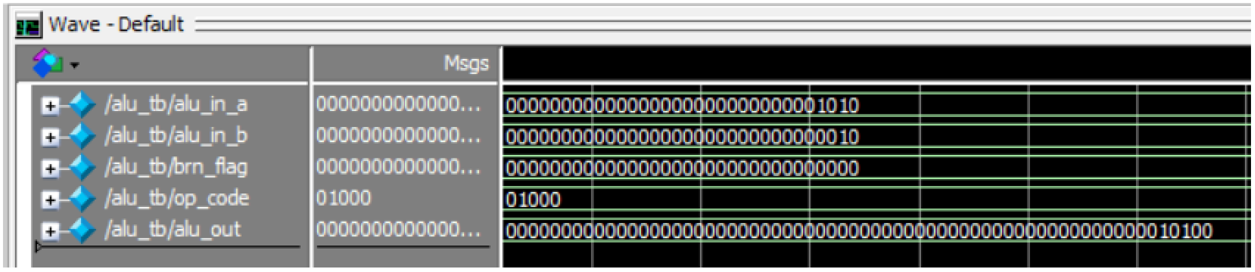
Rotate Right (ROR)



```
Wave - Default
                     Msgs
/alu_tb/alu_in_a     0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b     0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag     0000000000000...   00000000000000000000000000000000
/alu_tb/op_code      00111              00111
/alu_tb/alu_out      0000000000000...   0000000000000000000000000000000000000000000000000000000000000101
```

Figure 10: Signals for the rotate right operation, a rotated right with the corresponding output signal, 'out'

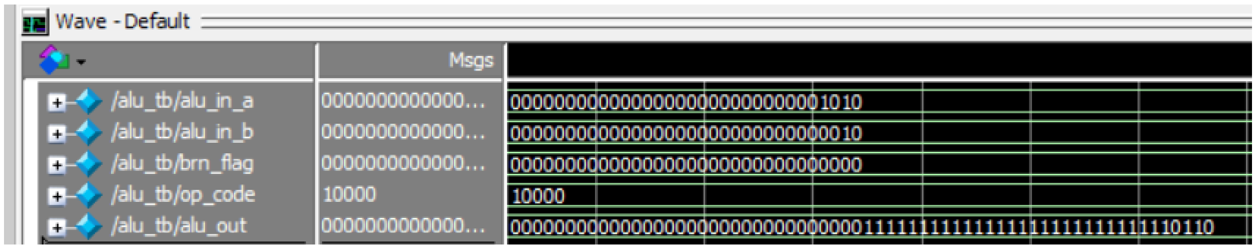
Rotate Left (ROL)



```
Wave - Default
                     Msgs
/alu_tb/alu_in_a     0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b     0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag     0000000000000...   00000000000000000000000000000000
/alu_tb/op_code      01000              01000
/alu_tb/alu_out      0000000000000...   0000000000000000000000000000000000000000000000000000000000010100
```

Figure 11: Signals for the rotate left operation, a rotating left with the corresponding output signal, 'out'

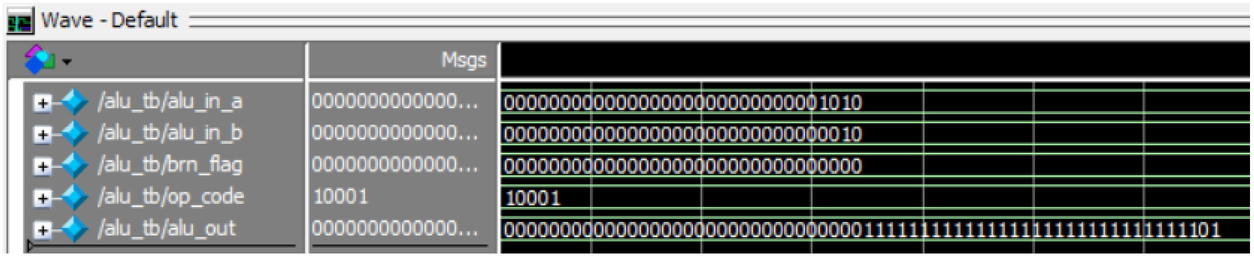
Negate (NEG)



```
Wave - Default
                     Msgs
/alu_tb/alu_in_a     0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b     0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag     0000000000000...   00000000000000000000000000000000
/alu_tb/op_code      10000              10000
/alu_tb/alu_out      0000000000000...   0000000000000000000000000000000011111111111111111111111111110110
```

Figure 12: Signals for the negate operation, negating a with the corresponding output signal, 'out'

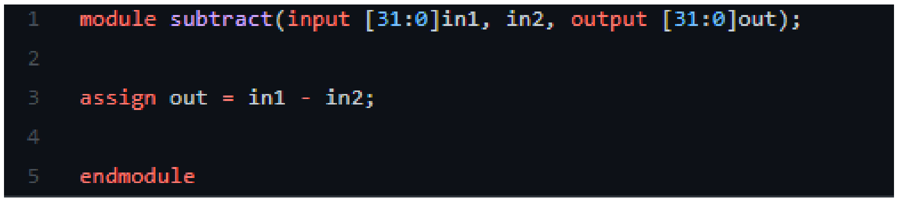
Logical NOT



```
Wave - Default
                     Msgs
/alu_tb/alu_in_a     0000000000000...   00000000000000000000000000001010
/alu_tb/alu_in_b     0000000000000...   00000000000000000000000000000010
/alu_tb/brn_flag     0000000000000...   00000000000000000000000000000000
/alu_tb/op_code      10001              10001
/alu_tb/alu_out      0000000000000...   0000000000000000000000000000000011111111111111111111111111110101
```

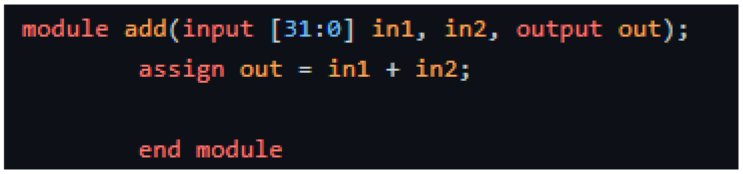
Figure 13: Signals for the logical not operation, not a, with the corresponding output signal, 'out'

HDL Code



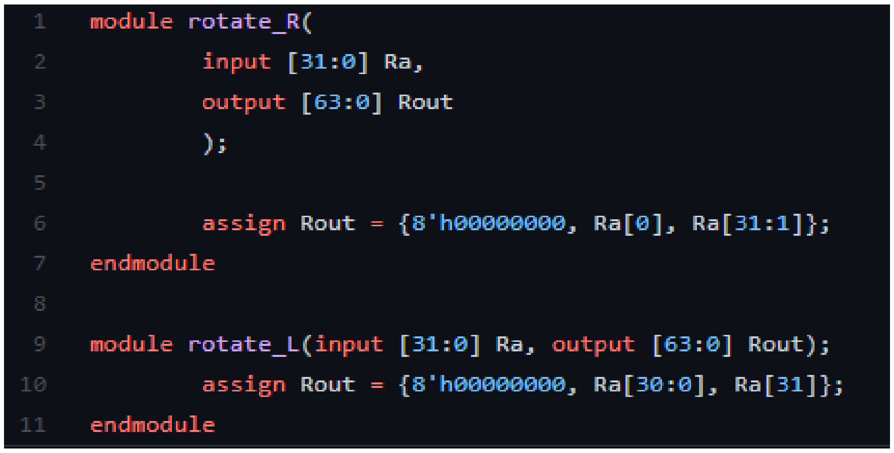
```verilog
module subtract(input [31:0]in1, in2, output [31:0]out);

assign out = in1 - in2;

endmodule
```

Figure 14: ALU subtract operation



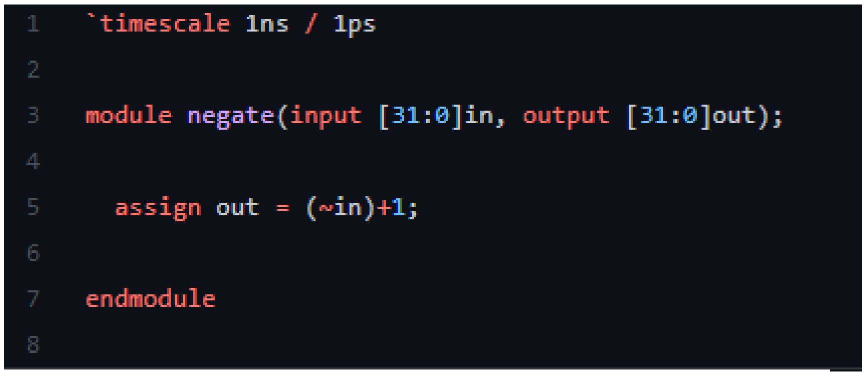
```verilog
module add(input [31:0] in1, in2, output out);
        assign out = in1 + in2;

        end module
```

Figure 15: ALU addition operation



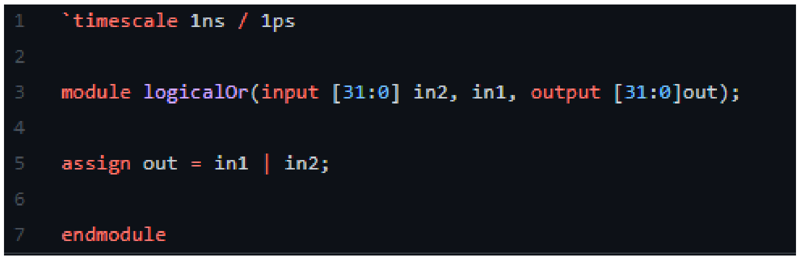
```verilog
module rotate_R(
        input [31:0] Ra,
        output [63:0] Rout
        );

        assign Rout = {8'h00000000, Ra[0], Ra[31:1]};
endmodule

module rotate_L(input [31:0] Ra, output [63:0] Rout);
        assign Rout = {8'h00000000, Ra[30:0], Ra[31]};
endmodule
```

Figure 16:ALU rotate left and right operations



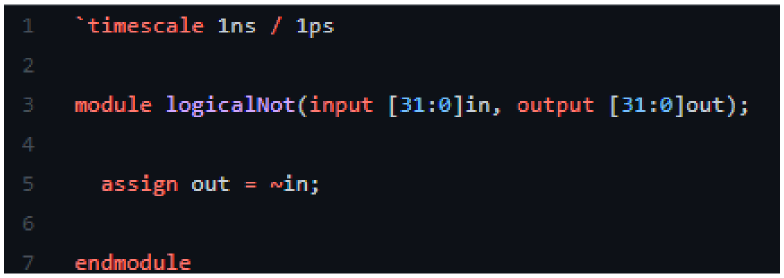
```verilog
`timescale 1ns / 1ps

module negate(input [31:0]in, output [31:0]out);

  assign out = (~in)+1;

endmodule

```

Figure 17: ALU negate operation



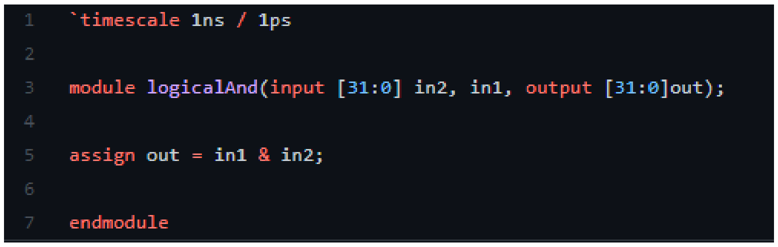
```verilog
`timescale 1ns / 1ps

module logicalOr(input [31:0] in2, in1, output [31:0]out);

assign out = in1 | in2;

endmodule
```

Figure 18: ALU logical or operation



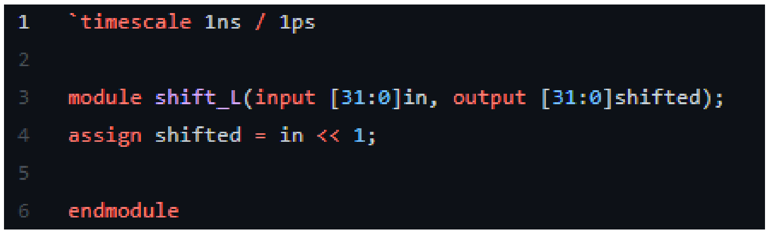
```verilog
`timescale 1ns / 1ps

module logicalNot(input [31:0]in, output [31:0]out);

  assign out = ~in;

endmodule
```

Figure 19: ALU logical not operation



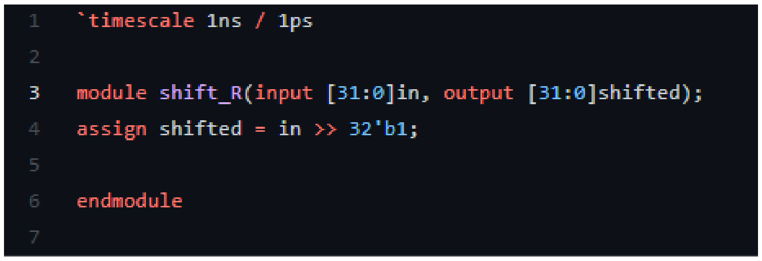
```
`timescale 1ns / 1ps

module logicalAnd(input [31:0] in2, in1, output [31:0]out);

assign out = in1 & in2;

endmodule
```

Figure 20: ALU logical and operation



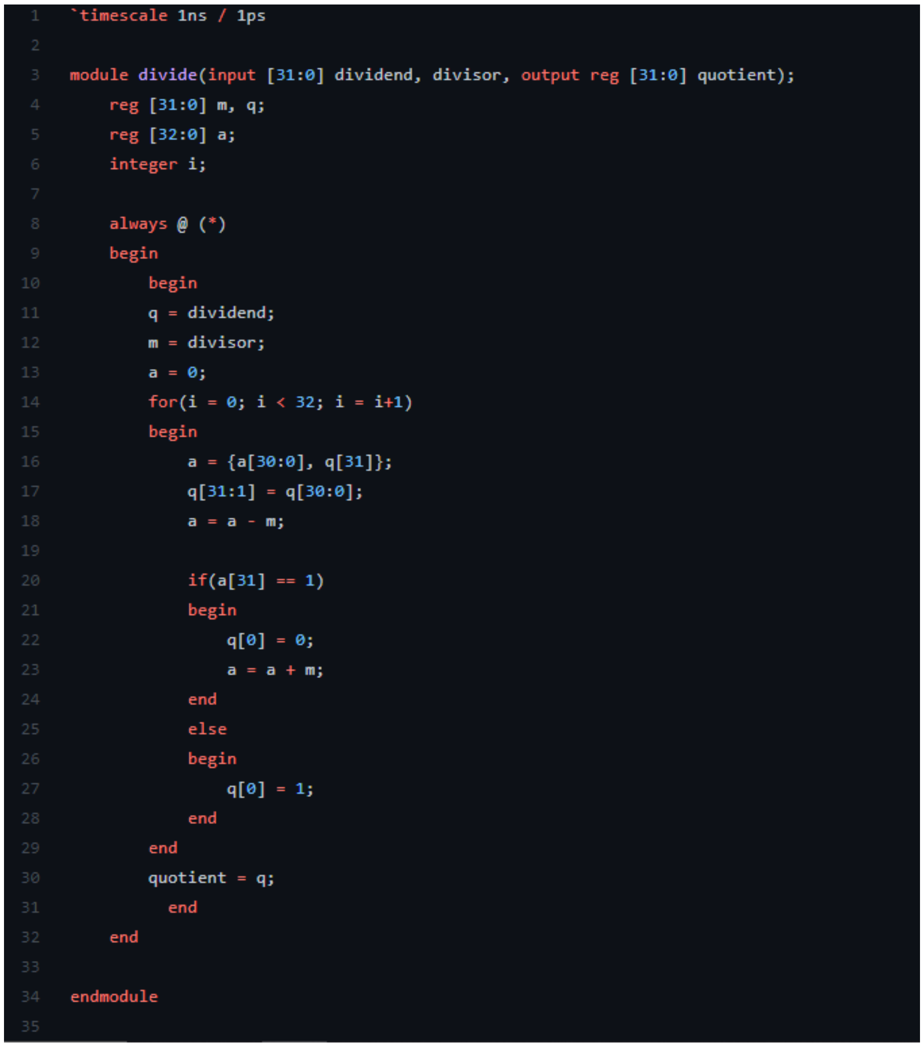
```
`timescale 1ns / 1ps

module shift_L(input [31:0]in, output [31:0]shifted);
assign shifted = in << 1;

endmodule
```

Figure 21: ALU shift left operation



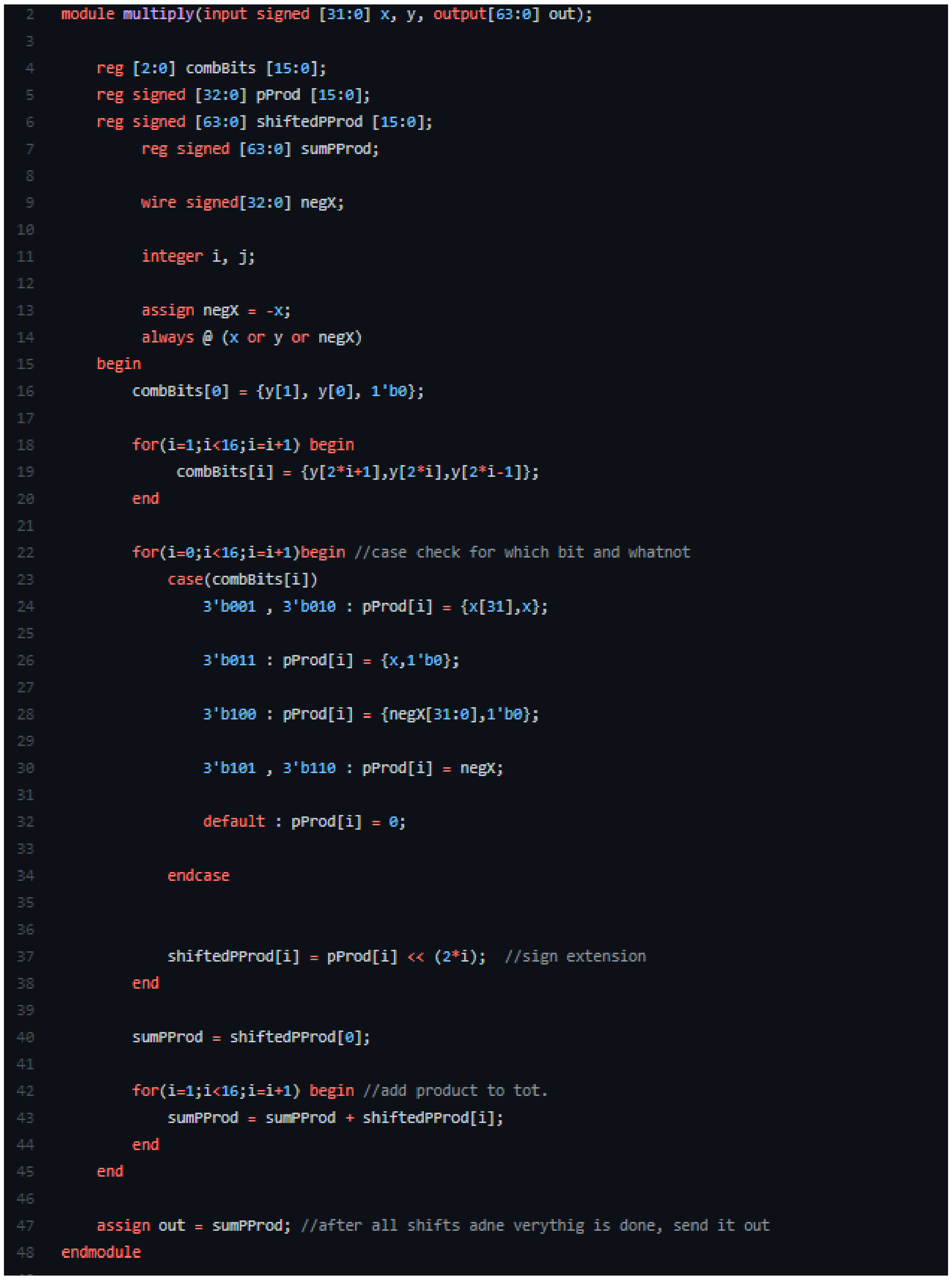
```
`timescale 1ns / 1ps

module shift_R(input [31:0]in, output [31:0]shifted);
assign shifted = in >> 32'b1;

endmodule

```

Figure 22: ALU shift right operation



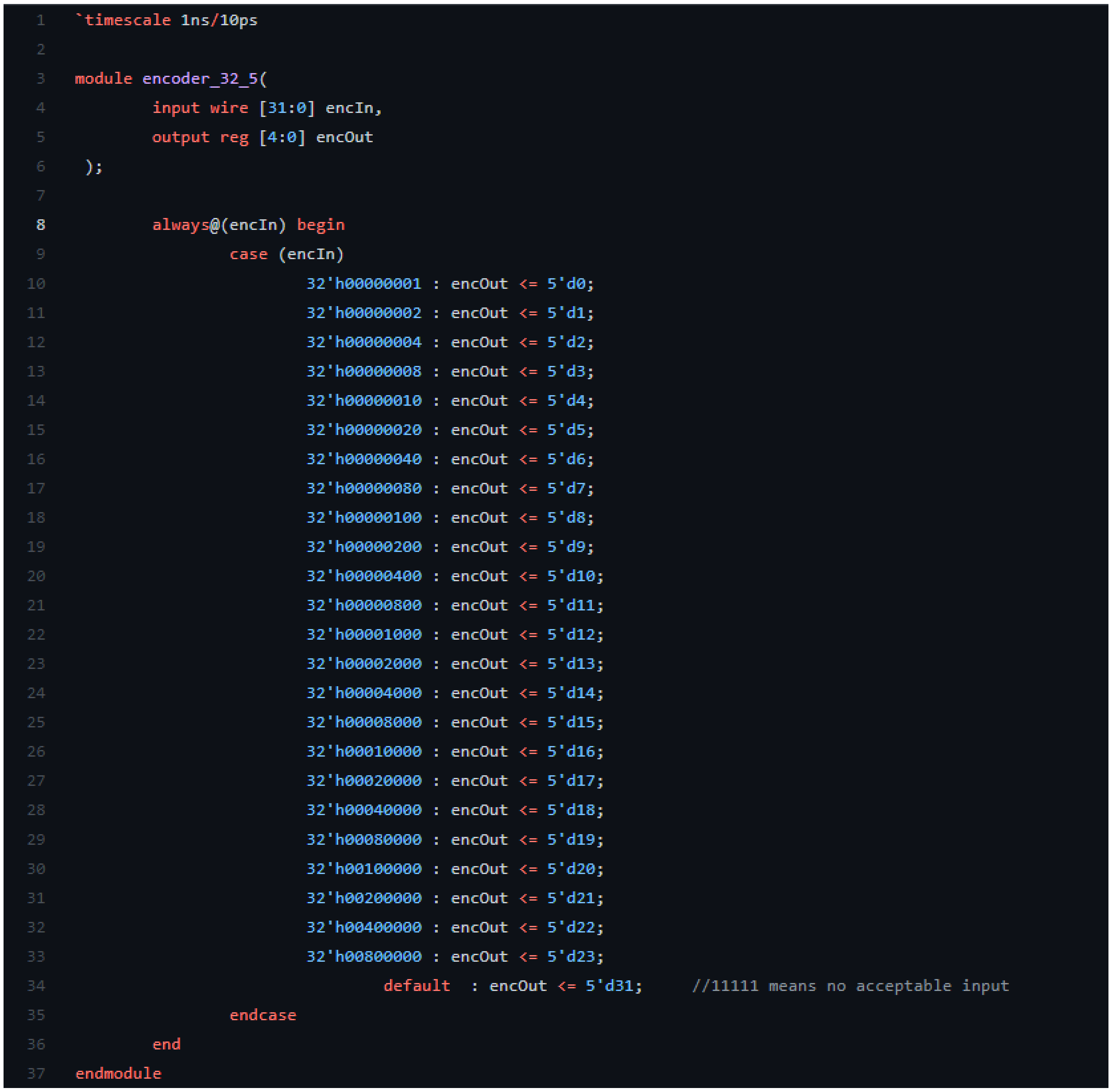
```verilog
`timescale 1ns / 1ps

module divide(input [31:0] dividend, divisor, output reg [31:0] quotient);
    reg [31:0] m, q;
    reg [32:0] a;
    integer i;

    always @ (*)
    begin
        begin
        q = dividend;
        m = divisor;
        a = 0;
        for(i = 0; i < 32; i = i+1)
        begin
            a = {a[30:0], q[31]};
            q[31:1] = q[30:0];
            a = a - m;

            if(a[31] == 1)
            begin
                q[0] = 0;
                a = a + m;
            end
            else
            begin
                q[0] = 1;
            end
        end
        quotient = q;
          end
    end

endmodule
```

Figure 23: Division algorithm



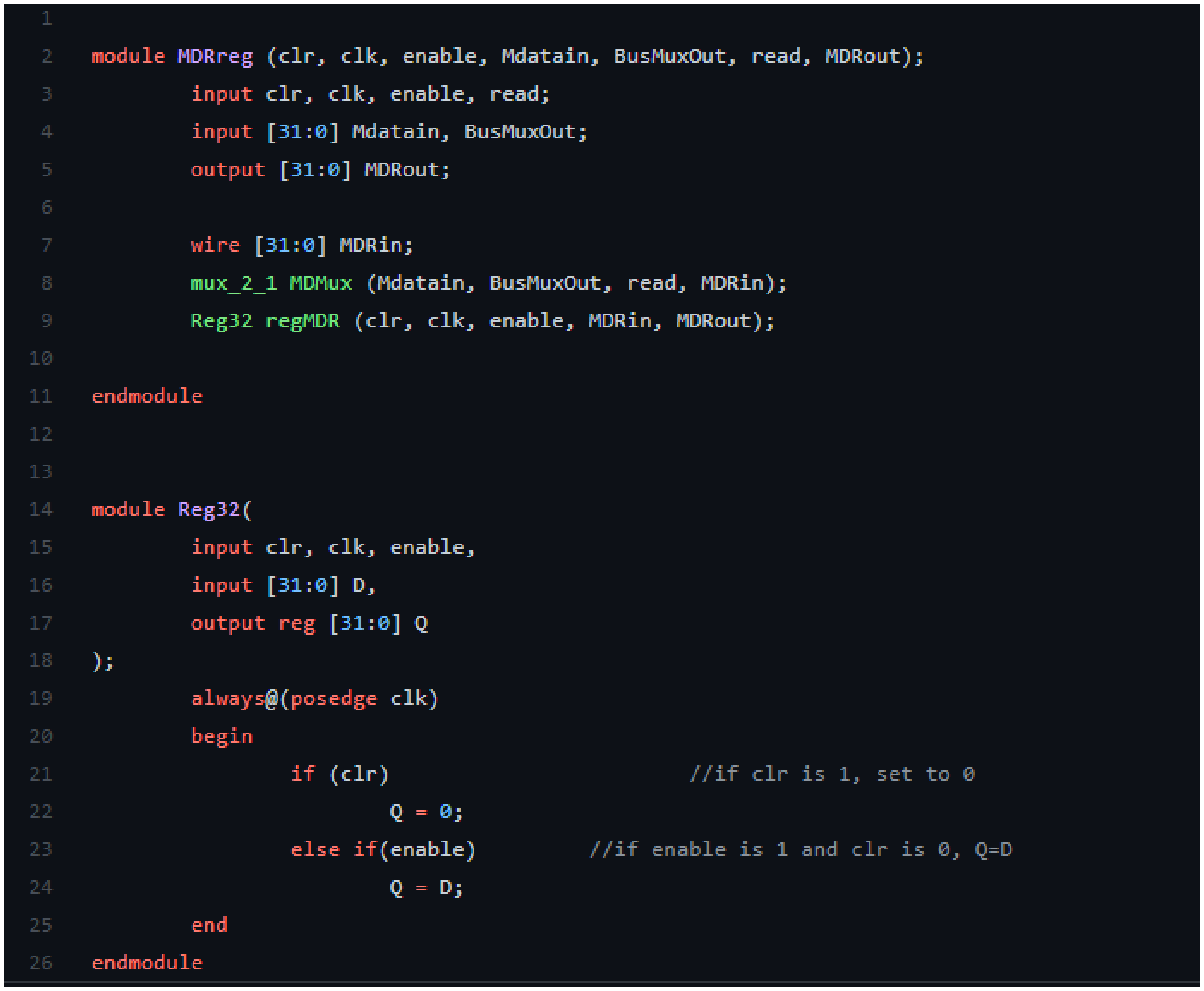
```verilog
module multiply(input signed [31:0] x, y, output[63:0] out);

    reg [2:0] combBits [15:0];
    reg signed [32:0] pProd [15:0];
    reg signed [63:0] shiftedPProd [15:0];
         reg signed [63:0] sumPProd;

         wire signed[32:0] negX;

         integer i, j;

         assign negX = -x;
         always @ (x or y or negX)
    begin
        combBits[0] = {y[1], y[0], 1'b0};

        for(i=1;i<16;i=i+1) begin
             combBits[i] = {y[2*i+1],y[2*i],y[2*i-1]};
        end

        for(i=0;i<16;i=i+1)begin //case check for which bit and whatnot
            case(combBits[i])
                3'b001 , 3'b010 : pProd[i] = {x[31],x};

                3'b011 : pProd[i] = {x,1'b0};

                3'b100 : pProd[i] = {negX[31:0],1'b0};

                3'b101 , 3'b110 : pProd[i] = negX;

                default : pProd[i] = 0;

            endcase


            shiftedPProd[i] = pProd[i] << (2*i);  //sign extension
        end

        sumPProd = shiftedPProd[0];

        for(i=1;i<16;i=i+1) begin //add product to tot.
            sumPProd = sumPProd + shiftedPProd[i];
        end
    end

    assign out = sumPProd; //after all shifts adne verythig is done, send it out
endmodule
```

Figure 24: Booth's Algorithm for multiplication



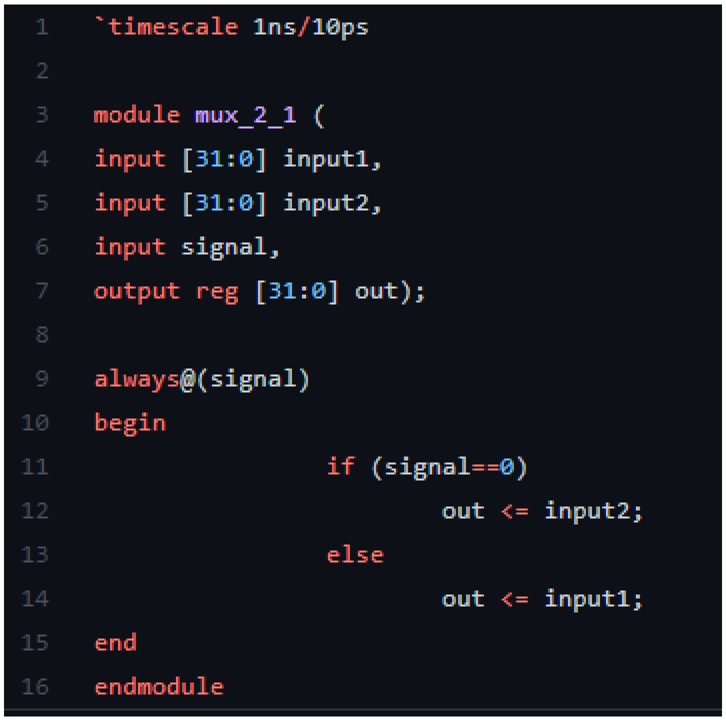
```verilog
`timescale 1ns/10ps

module encoder_32_5(
        input wire [31:0] encIn,
        output reg [4:0] encOut
 );

        always@(encIn) begin
                case (encIn)
                        32'h00000001 : encOut <= 5'd0;
                        32'h00000002 : encOut <= 5'd1;
                        32'h00000004 : encOut <= 5'd2;
                        32'h00000008 : encOut <= 5'd3;
                        32'h00000010 : encOut <= 5'd4;
                        32'h00000020 : encOut <= 5'd5;
                        32'h00000040 : encOut <= 5'd6;
                        32'h00000080 : encOut <= 5'd7;
                        32'h00000100 : encOut <= 5'd8;
                        32'h00000200 : encOut <= 5'd9;
                        32'h00000400 : encOut <= 5'd10;
                        32'h00000800 : encOut <= 5'd11;
                        32'h00001000 : encOut <= 5'd12;
                        32'h00002000 : encOut <= 5'd13;
                        32'h00004000 : encOut <= 5'd14;
                        32'h00008000 : encOut <= 5'd15;
                        32'h00010000 : encOut <= 5'd16;
                        32'h00020000 : encOut <= 5'd17;
                        32'h00040000 : encOut <= 5'd18;
                        32'h00080000 : encOut <= 5'd19;
                        32'h00100000 : encOut <= 5'd20;
                        32'h00200000 : encOut <= 5'd21;
                        32'h00400000 : encOut <= 5'd22;
                        32'h00800000 : encOut <= 5'd23;
                                default  : encOut <= 5'd31;     //11111 means no acceptable input
                endcase
        end
endmodule
```

Figure 25: 32:5 encoder



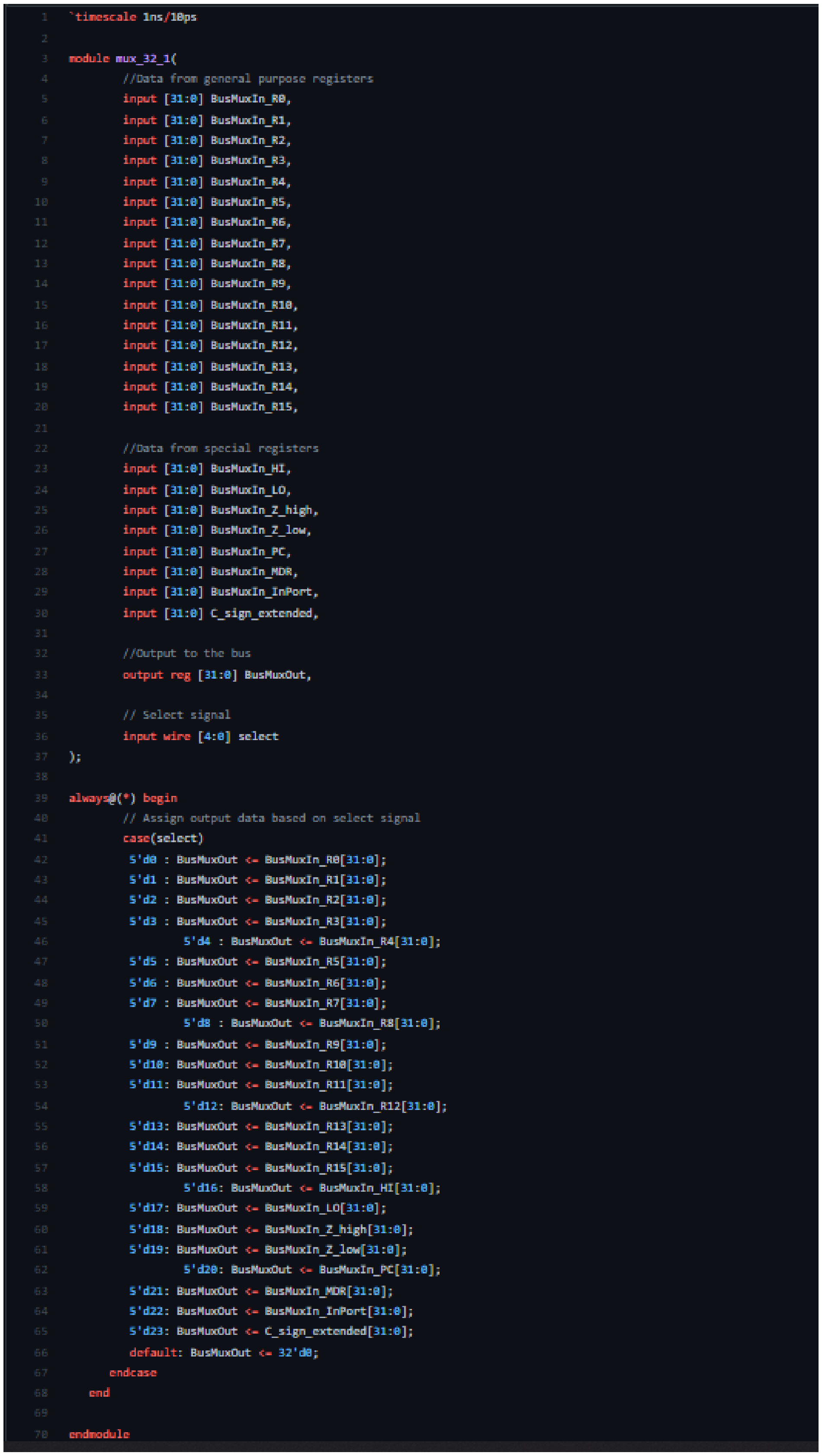
```verilog

module MDRreg (clr, clk, enable, Mdatain, BusMuxOut, read, MDRout);
        input clr, clk, enable, read;
        input [31:0] Mdatain, BusMuxOut;
        output [31:0] MDRout;

        wire [31:0] MDRin;
        mux_2_1 MDMux (Mdatain, BusMuxOut, read, MDRin);
        Reg32 regMDR (clr, clk, enable, MDRin, MDRout);

endmodule


module Reg32(
        input clr, clk, enable,
        input [31:0] D,
        output reg [31:0] Q
);
        always@(posedge clk)
        begin
                if (clr)                        //if clr is 1, set to 0
                        Q = 0;
                else if(enable)         //if enable is 1 and clr is 0, Q=D
                        Q = D;
        end
endmodule
```

Figure 26: MDR and 32-bit register



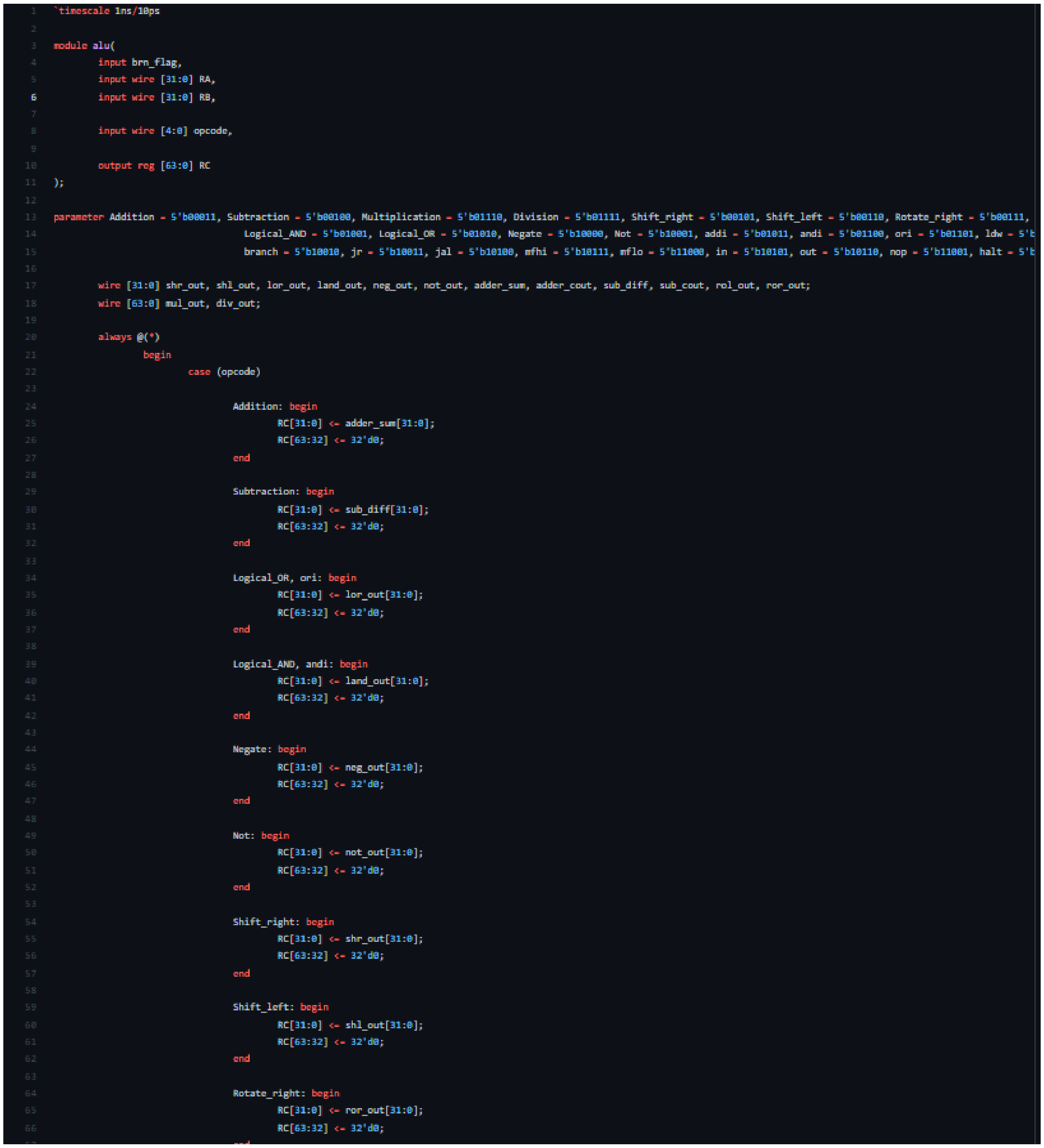
```verilog
`timescale 1ns/10ps

module mux_2_1 (
input [31:0] input1,
input [31:0] input2,
input signal,
output reg [31:0] out);

always@(signal)
begin
                if (signal==0)
                        out <= input2;
                else
                        out <= input1;
end
endmodule
```

Figure 27: 2:1 multiplexer



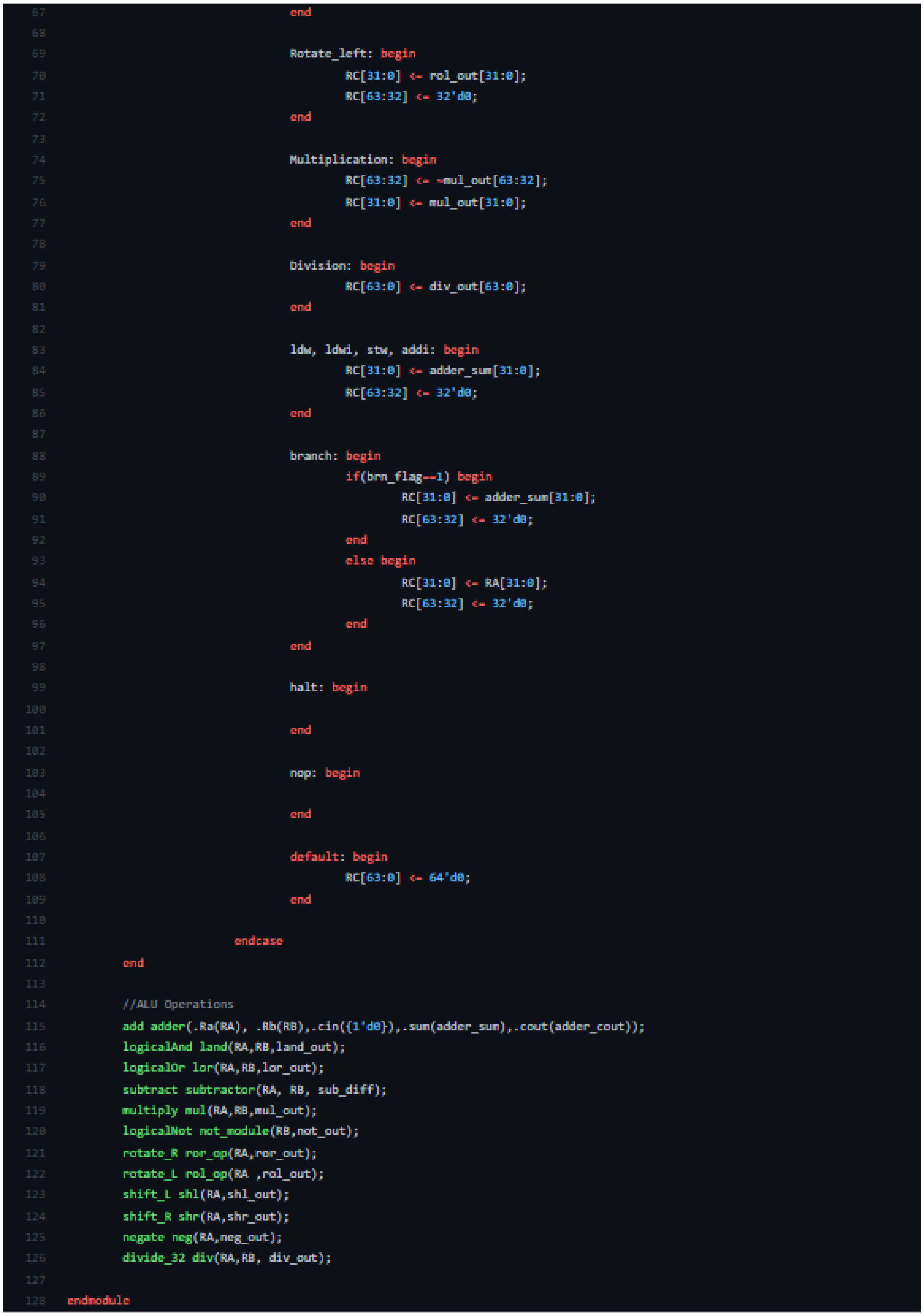
```verilog
`timescale 1ns/10ps

module mux_32_1(
        //Data from general purpose registers
        input [31:0] BusMuxIn_R0,
        input [31:0] BusMuxIn_R1,
        input [31:0] BusMuxIn_R2,
        input [31:0] BusMuxIn_R3,
        input [31:0] BusMuxIn_R4,
        input [31:0] BusMuxIn_R5,
        input [31:0] BusMuxIn_R6,
        input [31:0] BusMuxIn_R7,
        input [31:0] BusMuxIn_R8,
        input [31:0] BusMuxIn_R9,
        input [31:0] BusMuxIn_R10,
        input [31:0] BusMuxIn_R11,
        input [31:0] BusMuxIn_R12,
        input [31:0] BusMuxIn_R13,
        input [31:0] BusMuxIn_R14,
        input [31:0] BusMuxIn_R15,

        //Data from special registers
        input [31:0] BusMuxIn_HI,
        input [31:0] BusMuxIn_LO,
        input [31:0] BusMuxIn_Z_high,
        input [31:0] BusMuxIn_Z_low,
        input [31:0] BusMuxIn_PC,
        input [31:0] BusMuxIn_MDR,
        input [31:0] BusMuxIn_InPort,
        input [31:0] C_sign_extended,

        //Output to the bus
        output reg [31:0] BusMuxOut,

        // Select signal
        input wire [4:0] select
);

always@(*) begin
        // Assign output data based on select signal
        case(select)
         5'd0 : BusMuxOut <= BusMuxIn_R0[31:0];
         5'd1 : BusMuxOut <= BusMuxIn_R1[31:0];
         5'd2 : BusMuxOut <= BusMuxIn_R2[31:0];
         5'd3 : BusMuxOut <= BusMuxIn_R3[31:0];
                 5'd4 : BusMuxOut <= BusMuxIn_R4[31:0];
         5'd5 : BusMuxOut <= BusMuxIn_R5[31:0];
         5'd6 : BusMuxOut <= BusMuxIn_R6[31:0];
         5'd7 : BusMuxOut <= BusMuxIn_R7[31:0];
                 5'd8 : BusMuxOut <= BusMuxIn_R8[31:0];
         5'd9 : BusMuxOut <= BusMuxIn_R9[31:0];
         5'd10: BusMuxOut <= BusMuxIn_R10[31:0];
         5'd11: BusMuxOut <= BusMuxIn_R11[31:0];
                 5'd12: BusMuxOut <= BusMuxIn_R12[31:0];
         5'd13: BusMuxOut <= BusMuxIn_R13[31:0];
         5'd14: BusMuxOut <= BusMuxIn_R14[31:0];
         5'd15: BusMuxOut <= BusMuxIn_R15[31:0];
                 5'd16: BusMuxOut <= BusMuxIn_HI[31:0];
         5'd17: BusMuxOut <= BusMuxIn_LO[31:0];
         5'd18: BusMuxOut <= BusMuxIn_Z_high[31:0];
         5'd19: BusMuxOut <= BusMuxIn_Z_low[31:0];
                 5'd20: BusMuxOut <= BusMuxIn_PC[31:0];
         5'd21: BusMuxOut <= BusMuxIn_MDR[31:0];
         5'd22: BusMuxOut <= BusMuxIn_InPort[31:0];
         5'd23: BusMuxOut <= C_sign_extended[31:0];
         default: BusMuxOut <= 32'd0;
      endcase
   end

endmodule
```

Figure 28: 31:1 multiplexer



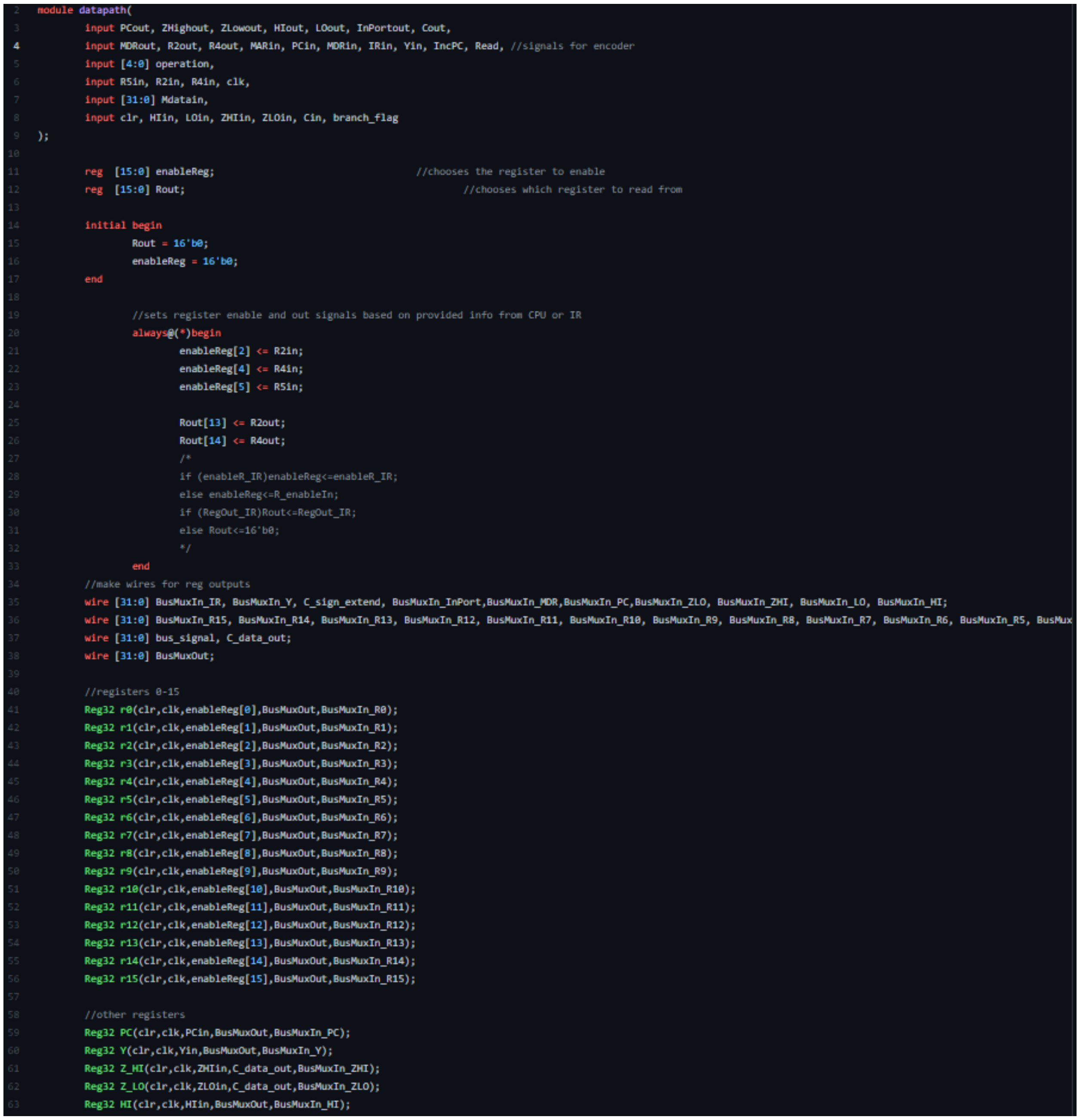
```verilog
`timescale 1ns/10ps

module alu(
        input brn_flag,
        input wire [31:0] RA,
        input wire [31:0] RB,

        input wire [4:0] opcode,

        output reg [63:0] RC
);

parameter Addition = 5'b00011, Subtraction = 5'b00100, Multiplication = 5'b01110, Division = 5'b01111, Shift_right = 5'b00101, Shift_left = 5'b00110, Rotate_right = 5'b00111,
                                  Logical_AND = 5'b01001, Logical_OR = 5'b01010, Negate = 5'b10000, Not = 5'b10001, addi = 5'b01011, andi = 5'b01100, ori = 5'b01101, ldw = 5'b
                                  branch = 5'b10010, jr = 5'b10011, jal = 5'b10100, mfhi = 5'b10111, mflo = 5'b11000, in = 5'b10101, out = 5'b10110, nop = 5'b11001, halt = 5'b

        wire [31:0] shr_out, shl_out, lor_out, land_out, neg_out, not_out, adder_sum, adder_cout, sub_diff, sub_cout, rol_out, ror_out;
        wire [63:0] mul_out, div_out;

        always @(*)
                begin
                        case (opcode)

                                Addition: begin
                                        RC[31:0] <= adder_sum[31:0];
                                        RC[63:32] <= 32'd0;
                                end

                                Subtraction: begin
                                        RC[31:0] <= sub_diff[31:0];
                                        RC[63:32] <= 32'd0;
                                end

                                Logical_OR, ori: begin
                                        RC[31:0] <= lor_out[31:0];
                                        RC[63:32] <= 32'd0;
                                end

                                Logical_AND, andi: begin
                                        RC[31:0] <= land_out[31:0];
                                        RC[63:32] <= 32'd0;
                                end

                                Negate: begin
                                        RC[31:0] <= neg_out[31:0];
                                        RC[63:32] <= 32'd0;
                                end

                                Not: begin
                                        RC[31:0] <= not_out[31:0];
                                        RC[63:32] <= 32'd0;
                                end

                                Shift_right: begin
                                        RC[31:0] <= shr_out[31:0];
                                        RC[63:32] <= 32'd0;
                                end

                                Shift_left: begin
                                        RC[31:0] <= shl_out[31:0];
                                        RC[63:32] <= 32'd0;
                                end

                                Rotate_right: begin
                                        RC[31:0] <= ror_out[31:0];
                                        RC[63:32] <= 32'd0;
                                end
```

Figure 29: Part 1/2 of the ALU code



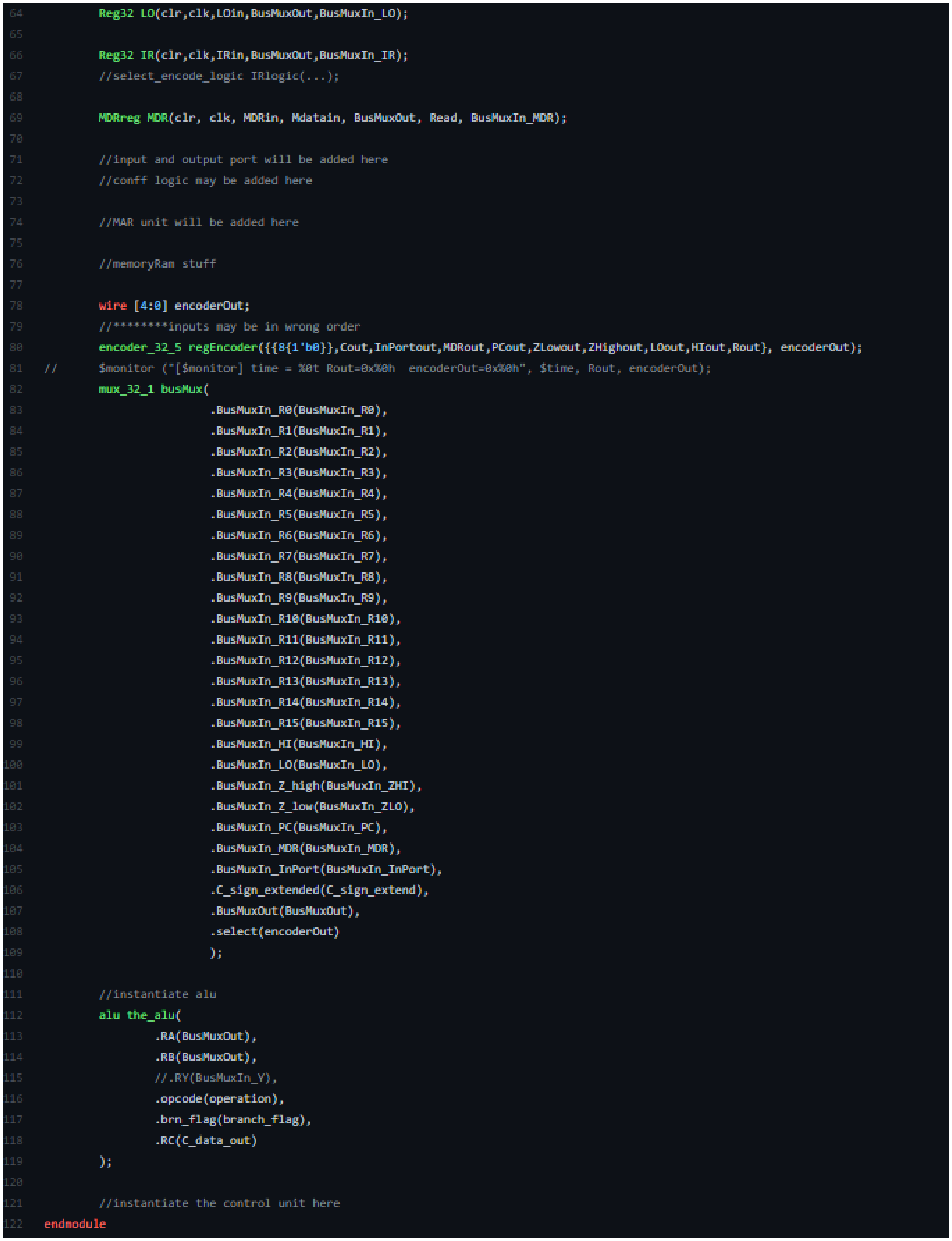
```
                end

                Rotate_left: begin
                    RC[31:0] <= rol_out[31:0];
                    RC[63:32] <= 32'd0;
                end

                Multiplication: begin
                    RC[63:32] <= ~mul_out[63:32];
                    RC[31:0] <= mul_out[31:0];
                end

                Division: begin
                    RC[63:0] <= div_out[63:0];
                end

                ldw, ldwi, stw, addi: begin
                    RC[31:0] <= adder_sum[31:0];
                    RC[63:32] <= 32'd0;
                end

                branch: begin
                    if(brn_flag==1) begin
                        RC[31:0] <= adder_sum[31:0];
                        RC[63:32] <= 32'd0;
                    end
                    else begin
                        RC[31:0] <= RA[31:0];
                        RC[63:32] <= 32'd0;
                    end
                end

                halt: begin

                end

                nop: begin

                end

                default: begin
                    RC[63:0] <= 64'd0;
                end

            endcase
    end

    //ALU Operations
    add adder(.Ra(RA), .Rb(RB),.cin({1'd0}),.sum(adder_sum),.cout(adder_cout));
    logicalAnd land(RA,RB,land_out);
    logicalOr lor(RA,RB,lor_out);
    subtract subtractor(RA, RB, sub_diff);
    multiply mul(RA,RB,mul_out);
    logicalNot not_module(RB,not_out);
    rotate_R ror_op(RA,ror_out);
    rotate_L rol_op(RA ,rol_out);
    shift_L shl(RA,shl_out);
    shift_R shr(RA,shr_out);
    negate neg(RA,neg_out);
    divide_32 div(RA,RB, div_out);

endmodule
```

Figure 30: Part 2/2 of the ALU code



```verilog
module datapath(
        input PCout, ZHighout, ZLowout, HIout, LOout, InPortout, Cout,
        input MDRout, R2out, R4out, MARin, PCin, MDRin, IRin, Yin, IncPC, Read, //signals for encoder
        input [4:0] operation,
        input R5in, R2in, R4in, clk,
        input [31:0] Mdatain,
        input clr, HIin, LOin, ZHIin, ZLOin, Cin, branch_flag
);

        reg  [15:0] enableReg;                                  //chooses the register to enable
        reg  [15:0] Rout;                                               //chooses which register to read from

        initial begin
                Rout = 16'b0;
                enableReg = 16'b0;
        end

                //sets register enable and out signals based on provided info from CPU or IR
                always@(*)begin
                        enableReg[2] <= R2in;
                        enableReg[4] <= R4in;
                        enableReg[5] <= R5in;

                        Rout[13] <= R2out;
                        Rout[14] <= R4out;
                        /*
                        if (enableR_IR)enableReg<=enableR_IR;
                        else enableReg<=R_enableIn;
                        if (RegOut_IR)Rout<=RegOut_IR;
                        else Rout<=16'b0;
                        */
                end
        //make wires for reg outputs
        wire [31:0] BusMuxIn_IR, BusMuxIn_Y, C_sign_extend, BusMuxIn_InPort,BusMuxIn_MDR,BusMuxIn_PC,BusMuxIn_ZLO, BusMuxIn_ZHI, BusMuxIn_LO, BusMuxIn_HI;
        wire [31:0] BusMuxIn_R15, BusMuxIn_R14, BusMuxIn_R13, BusMuxIn_R12, BusMuxIn_R11, BusMuxIn_R10, BusMuxIn_R9, BusMuxIn_R8, BusMuxIn_R7, BusMuxIn_R6, BusMuxIn_R5, BusMux
        wire [31:0] bus_signal, C_data_out;
        wire [31:0] BusMuxOut;

        //registers 0-15
        Reg32 r0(clr,clk,enableReg[0],BusMuxOut,BusMuxIn_R0);
        Reg32 r1(clr,clk,enableReg[1],BusMuxOut,BusMuxIn_R1);
        Reg32 r2(clr,clk,enableReg[2],BusMuxOut,BusMuxIn_R2);
        Reg32 r3(clr,clk,enableReg[3],BusMuxOut,BusMuxIn_R3);
        Reg32 r4(clr,clk,enableReg[4],BusMuxOut,BusMuxIn_R4);
        Reg32 r5(clr,clk,enableReg[5],BusMuxOut,BusMuxIn_R5);
        Reg32 r6(clr,clk,enableReg[6],BusMuxOut,BusMuxIn_R6);
        Reg32 r7(clr,clk,enableReg[7],BusMuxOut,BusMuxIn_R7);
        Reg32 r8(clr,clk,enableReg[8],BusMuxOut,BusMuxIn_R8);
        Reg32 r9(clr,clk,enableReg[9],BusMuxOut,BusMuxIn_R9);
        Reg32 r10(clr,clk,enableReg[10],BusMuxOut,BusMuxIn_R10);
        Reg32 r11(clr,clk,enableReg[11],BusMuxOut,BusMuxIn_R11);
        Reg32 r12(clr,clk,enableReg[12],BusMuxOut,BusMuxIn_R12);
        Reg32 r13(clr,clk,enableReg[13],BusMuxOut,BusMuxIn_R13);
        Reg32 r14(clr,clk,enableReg[14],BusMuxOut,BusMuxIn_R14);
        Reg32 r15(clr,clk,enableReg[15],BusMuxOut,BusMuxIn_R15);

        //other registers
        Reg32 PC(clr,clk,PCin,BusMuxOut,BusMuxIn_PC);
        Reg32 Y(clr,clk,Yin,BusMuxOut,BusMuxIn_Y);
        Reg32 Z_HI(clr,clk,ZHIin,C_data_out,BusMuxIn_ZHI);
        Reg32 Z_LO(clr,clk,ZLOin,C_data_out,BusMuxIn_ZLO);
        Reg32 HI(clr,clk,HIin,BusMuxOut,BusMuxIn_HI);
```

Figure 31: part 1/2 of datapath code

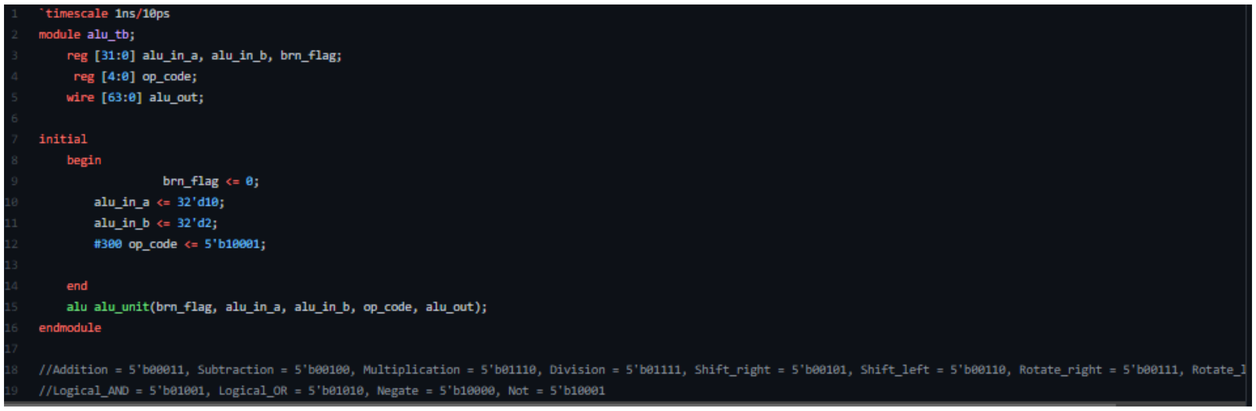


```verilog
        Reg32 LO(clr,clk,LOin,BusMuxOut,BusMuxIn_LO);

        Reg32 IR(clr,clk,IRin,BusMuxOut,BusMuxIn_IR);
        //select_encode_logic IRlogic(...);

        MDRreg MDR(clr, clk, MDRin, Mdatain, BusMuxOut, Read, BusMuxIn_MDR);

        //input and output port will be added here
        //conff logic may be added here

        //MAR unit will be added here

        //memoryRam stuff

        wire [4:0] encoderOut;
        //********inputs may be in wrong order
        encoder_32_5 regEncoder({{8{1'b0}},Cout,InPortout,MDRout,PCout,ZLowout,ZHighout,LOout,HIout,Rout}, encoderOut);
//      $monitor ("[$monitor] time = %0t Rout=0x%0h  encoderOut=0x%0h", $time, Rout, encoderOut);
        mux_32_1 busMux(
                        .BusMuxIn_R0(BusMuxIn_R0),
                        .BusMuxIn_R1(BusMuxIn_R1),
                        .BusMuxIn_R2(BusMuxIn_R2),
                        .BusMuxIn_R3(BusMuxIn_R3),
                        .BusMuxIn_R4(BusMuxIn_R4),
                        .BusMuxIn_R5(BusMuxIn_R5),
                        .BusMuxIn_R6(BusMuxIn_R6),
                        .BusMuxIn_R7(BusMuxIn_R7),
                        .BusMuxIn_R8(BusMuxIn_R8),
                        .BusMuxIn_R9(BusMuxIn_R9),
                        .BusMuxIn_R10(BusMuxIn_R10),
                        .BusMuxIn_R11(BusMuxIn_R11),
                        .BusMuxIn_R12(BusMuxIn_R12),
                        .BusMuxIn_R13(BusMuxIn_R13),
                        .BusMuxIn_R14(BusMuxIn_R14),
                        .BusMuxIn_R15(BusMuxIn_R15),
                        .BusMuxIn_HI(BusMuxIn_HI),
                        .BusMuxIn_LO(BusMuxIn_LO),
                        .BusMuxIn_Z_high(BusMuxIn_ZHI),
                        .BusMuxIn_Z_low(BusMuxIn_ZLO),
                        .BusMuxIn_PC(BusMuxIn_PC),
                        .BusMuxIn_MDR(BusMuxIn_MDR),
                        .BusMuxIn_InPort(BusMuxIn_InPort),
                        .C_sign_extended(C_sign_extend),
                        .BusMuxOut(BusMuxOut),
                        .select(encoderOut)
                        );

        //instantiate alu
        alu the_alu(
                .RA(BusMuxOut),
                .RB(BusMuxOut),
                //.RY(BusMuxIn_Y),
                .opcode(operation),
                .brn_flag(branch_flag),
                .RC(C_data_out)
        );

        //instantiate the control unit here
endmodule
```

Figure 32: part 2/2 of datapath code

Test Benches

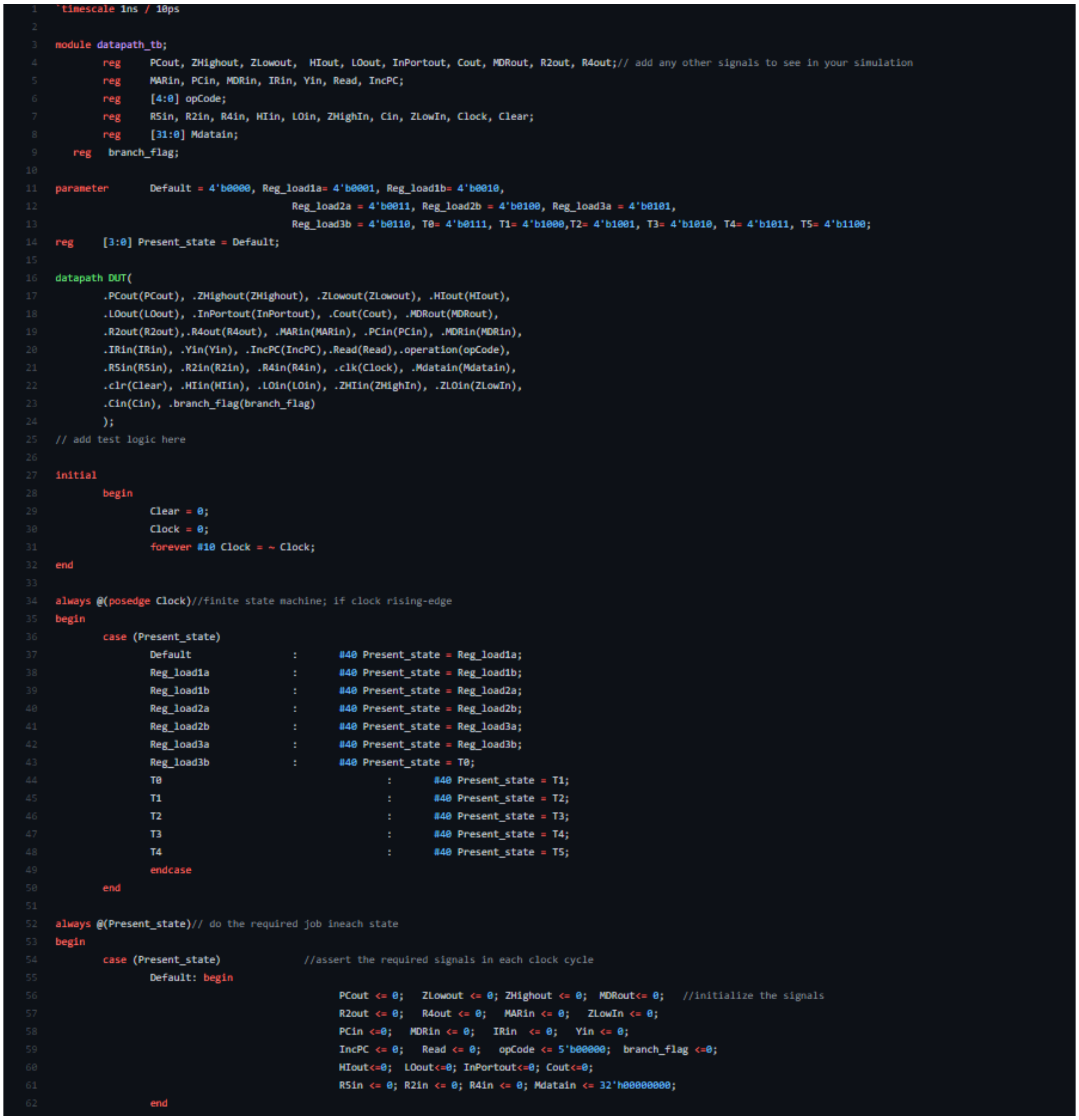
ALU Testbench (alu_tb)



```
`timescale 1ns/10ps
module alu_tb;
    reg [31:0] alu_in_a, alu_in_b, brn_flag;
     reg [4:0] op_code;
    wire [63:0] alu_out;

initial
    begin
                brn_flag <= 0;
        alu_in_a <= 32'd10;
        alu_in_b <= 32'd2;
        #300 op_code <= 5'b10001;

    end
    alu alu_unit(brn_flag, alu_in_a, alu_in_b, op_code, alu_out);
endmodule

//Addition = 5'b00011, Subtraction = 5'b00100, Multiplication = 5'b01110, Division = 5'b01111, Shift_right = 5'b00101, Shift_left = 5'b00110, Rotate_right = 5'b00111, Rotate_l
//Logical_AND = 5'b01001, Logical_OR = 5'b01010, Negate = 5'b10000, Not = 5'b10001
```

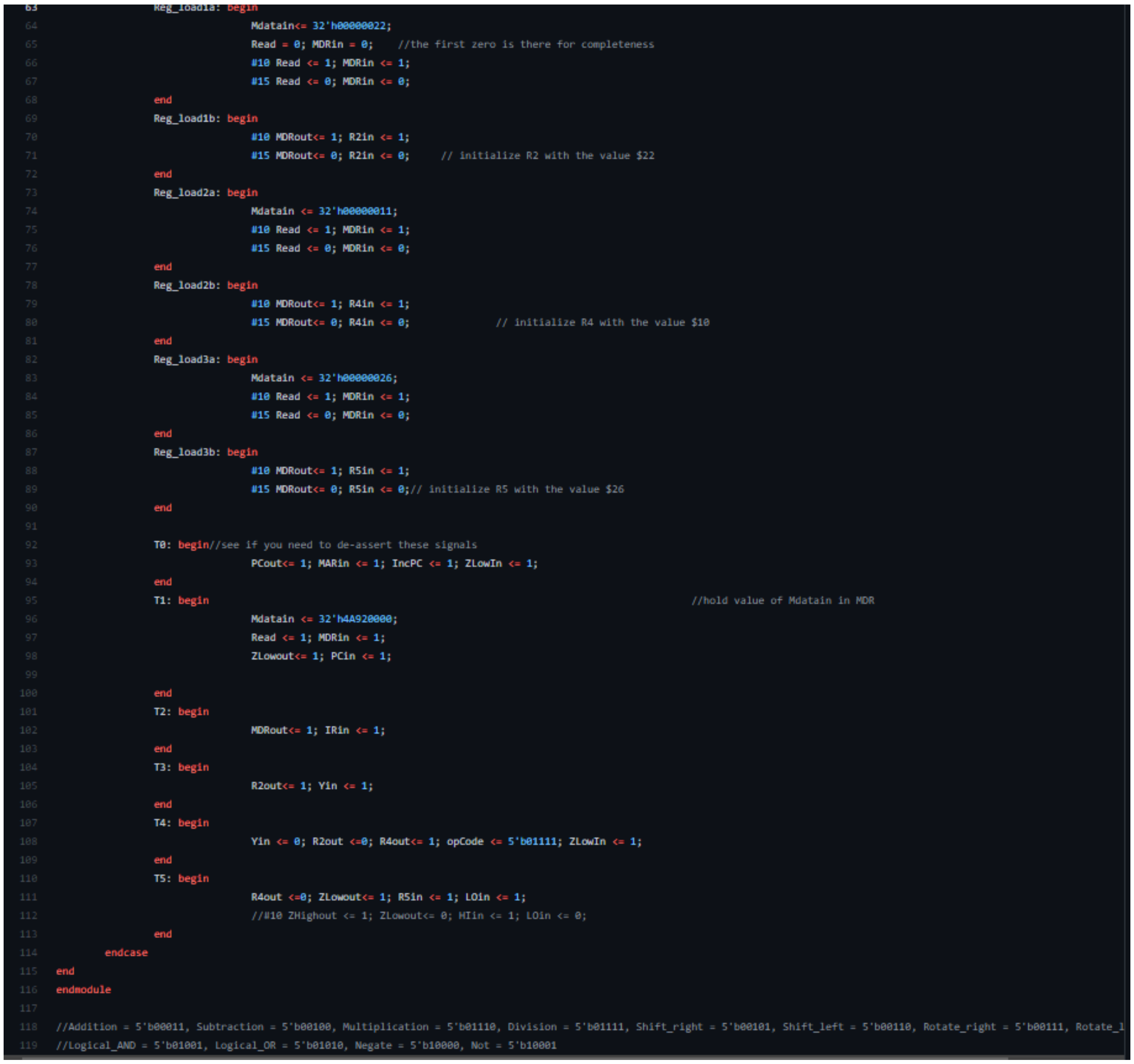
Figure 33: Testbench used to test the ALU

Datapath Testbench (datapath_tb)



```verilog
`timescale 1ns / 10ps

module datapath_tb;
        reg     PCout, ZHighout, ZLowout,  HIout, LOout, InPortout, Cout, MDRout, R2out, R4out;// add any other signals to see in your simulation
        reg     MARin, PCin, MDRin, IRin, Yin, Read, IncPC;
        reg     [4:0] opCode;
        reg     R5in, R2in, R4in, HIin, LOin, ZHighIn, Cin, ZLowIn, Clock, Clear;
        reg     [31:0] Mdatain;
   reg   branch_flag;

parameter       Default = 4'b0000, Reg_load1a= 4'b0001, Reg_load1b= 4'b0010,
                                Reg_load2a = 4'b0011, Reg_load2b = 4'b0100, Reg_load3a = 4'b0101,
                                Reg_load3b = 4'b0110, T0= 4'b0111, T1= 4'b1000,T2= 4'b1001, T3= 4'b1010, T4= 4'b1011, T5= 4'b1100;
reg     [3:0] Present_state = Default;

datapath DUT(
        .PCout(PCout), .ZHighout(ZHighout), .ZLowout(ZLowout), .HIout(HIout),
        .LOout(LOout), .InPortout(InPortout), .Cout(Cout), .MDRout(MDRout),
        .R2out(R2out),.R4out(R4out), .MARin(MARin), .PCin(PCin), .MDRin(MDRin),
        .IRin(IRin), .Yin(Yin), .IncPC(IncPC),.Read(Read),.operation(opCode),
        .R5in(R5in), .R2in(R2in), .R4in(R4in), .clk(Clock), .Mdatain(Mdatain),
        .clr(Clear), .HIin(HIin), .LOin(LOin), .ZHIin(ZHighIn), .ZLOin(ZLowIn),
        .Cin(Cin), .branch_flag(branch_flag)
        );
// add test logic here

initial
        begin
                Clear = 0;
                Clock = 0;
                forever #10 Clock = ~ Clock;
end

always @(posedge Clock)//finite state machine; if clock rising-edge
begin
        case (Present_state)
                Default                 :       #40 Present_state = Reg_load1a;
                Reg_load1a              :       #40 Present_state = Reg_load1b;
                Reg_load1b              :       #40 Present_state = Reg_load2a;
                Reg_load2a              :       #40 Present_state = Reg_load2b;
                Reg_load2b              :       #40 Present_state = Reg_load3a;
                Reg_load3a              :       #40 Present_state = Reg_load3b;
                Reg_load3b              :       #40 Present_state = T0;
                T0                                      :       #40 Present_state = T1;
                T1                                      :       #40 Present_state = T2;
                T2                                      :       #40 Present_state = T3;
                T3                                      :       #40 Present_state = T4;
                T4                                      :       #40 Present_state = T5;
                endcase
        end

always @(Present_state)// do the required job ineach state
begin
        case (Present_state)            //assert the required signals in each clock cycle
                Default: begin
                                PCout <= 0;   ZLowout <= 0; ZHighout <= 0;  MDRout<= 0;   //initialize the signals
                                R2out <= 0;   R4out <= 0;   MARin <= 0;   ZLowIn <= 0;
                                PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                                IncPC <= 0;   Read <= 0;   opCode <= 5'b00000;  branch_flag <=0;
                                HIout<=0;  LOout<=0; InPortout<=0; Cout<=0;
                                R5in <= 0; R2in <= 0; R4in <= 0; Mdatain <= 32'h00000000;

                end
```

Figure 33: part 1/2 of datapath testbench code



```
            Reg_load1a: begin
                        Mdatain<= 32'h00000022;
                        Read = 0; MDRin = 0;    //the first zero is there for completeness
                        #10 Read <= 1; MDRin <= 1;
                        #15 Read <= 0; MDRin <= 0;
            end
            Reg_load1b: begin
                        #10 MDRout<= 1; R2in <= 1;
                        #15 MDRout<= 0; R2in <= 0;     // initialize R2 with the value $22
            end
            Reg_load2a: begin
                        Mdatain <= 32'h00000011;
                        #10 Read <= 1; MDRin <= 1;
                        #15 Read <= 0; MDRin <= 0;
            end
            Reg_load2b: begin
                        #10 MDRout<= 1; R4in <= 1;
                        #15 MDRout<= 0; R4in <= 0;              // initialize R4 with the value $10
            end
            Reg_load3a: begin
                        Mdatain <= 32'h00000026;
                        #10 Read <= 1; MDRin <= 1;
                        #15 Read <= 0; MDRin <= 0;
            end
            Reg_load3b: begin
                        #10 MDRout<= 1; R5in <= 1;
                        #15 MDRout<= 0; R5in <= 0;// initialize R5 with the value $26
            end

            T0: begin//see if you need to de-assert these signals
                        PCout<= 1; MARin <= 1; IncPC <= 1; ZLowIn <= 1;
            end
            T1: begin                                                                       //hold value of Mdatain in MDR
                        Mdatain <= 32'h4A920000;
                        Read <= 1; MDRin <= 1;
                        ZLowout<= 1; PCin <= 1;

            end
            T2: begin
                        MDRout<= 1; IRin <= 1;
            end
            T3: begin
                        R2out<= 1; Yin <= 1;
            end
            T4: begin
                        Yin <= 0; R2out <=0; R4out<= 1; opCode <= 5'b01111; ZLowIn <= 1;
            end
            T5: begin
                        R4out <=0; ZLowout<= 1; R5in <= 1; LOin <= 1;
                        //#10 ZHighout <= 1; ZLowout<= 0; HIin <= 1; LOin <= 0;
            end
        endcase
end
endmodule

//Addition = 5'b00011, Subtraction = 5'b00100, Multiplication = 5'b01110, Division = 5'b01111, Shift_right = 5'b00101, Shift_left = 5'b00110, Rotate_right = 5'b00111, Rotate_l
//Logical_AND = 5'b01001, Logical_OR = 5'b01010, Negate = 5'b10000, Not = 5'b10001
```

Figure 34: part 2/2 of datapath testbench code