374 Phase 1 Report

March 10, 2023

Group 4:
Liam Salass (20229595)
Abdellah Ghassel (20230384)

Contents

Verilog Code and Computer Structure 3
Registers 3
Bus 3
MDR Unit 7
ALU 8
Addition 13
Subtraction 14
Multiplication 14
Division 15
Shift Operations 18
Bitwise ALU Operations: 20
Testbenching Files and Simulation Outputs 22
Division Testbench 22
And Testbench 29
Or Testbench 29
Add Testbench 29
Sub Testbench 30
Multiplication Testbench 31
Shift Right Testbench 31
Shift Right Arithmetic Testbench 31
Shift Left Testbench 32
Rotate Right Testbench 32
Rotate Left Testbench 33
Negate Testbench 33
Not Testbench 34

Verilog Code and Computer Structure

Registers

32-bit registers with positive clock edge activation and high enable for writing. A high clear signal to reset Q (output value) to 0. The code for the register is as follows:

```
module reg32bit(clr, clk, enable, D, Q);
    input wire clr, clk, enable;
    input wire [31:0]D;
    output reg [31:0]Q;
    //At positive clock edge begin
    always @(posedge clk) begin
        if (clr) //If clr is high set to 0
            Q <= 0;
        else if (enable) //If enable is high, read value from bus to Q
            Q <= D;
    end
endmodule
```

Bus

The bus consisted of 3 main components, the bus module which connects all the registers, the 32-to-1 multiplexer for deciding which register can write values to the bus, and the 32 to 5 encoder which determines which 5-bit signal to pass to the multiplexer for selecting the register. The bus takes input signals to determine which register can write to the bus and has the output of the selected register as the BusMuxOut 32-bit data output. The bus code is as follows:

```
module bus  #(parameter wordSize = 32)(
    //encoder signals
    input R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, R8out, R9out,
R10out, R11out, R12out, R13out, R14out, R15out,
    input HIout, LOout, ZHighOut, ZLowOut, PCout, MDRout, InPortout, Cout,
    //multiplexer registers
    input [wordSize-1:0] BusMuxIn_R0, BusMuxIn_R1, BusMuxIn_R2, BusMuxIn_R3,
BusMuxIn_R4, BusMuxIn_R5, BusMuxIn_R6, BusMuxIn_R7, BusMuxIn_R8, BusMuxIn_R9,
BusMuxIn_R10, BusMuxIn_R11, BusMuxIn_R12, BusMuxIn_R13, BusMuxIn_R14,
BusMuxIn_R15,
    input [wordSize-1:0] BusMuxIn_HI, BusMuxIn_LO, BusMuxIn_Zhigh, BusMuxIn_Zlow,
BusMuxIn_PC, BusMuxIn_MDR, BusMuxIn_InPort,
    output [wordSize-1:0] BusMuxOut
);

wire [4:0] s;
```

```
wire [wordSize-1:0] c_sign_extended;

assign c_sign_extended = (Cout == 0) ? 32'd0 : 32'd1;

encoder32to5 myEncoder (
    .ein({8'b0, Cout, InPortout, MDRout, PCout, ZHighOut, ZLowOut, HIout, LOout,
        R15out, R14out, R13out, R12out, R11out, R10out, R9out, R8out, R7out,
R6out, R5out, R4out, R3out, R2out, R1out, R0out}),
    .eout(s)
);

mux32to1 BusMux(

    .data0(BusMuxIn_R0),
    .data1(BusMuxIn_R1),
    .data2(BusMuxIn_R2),
    .data3(BusMuxIn_R3),
    .data4(BusMuxIn_R4),
    .data5(BusMuxIn_R5),
    .data6(BusMuxIn_R6),
    .data7(BusMuxIn_R7),
    .data8(BusMuxIn_R8),
    .data9(BusMuxIn_R9),
    .data10(BusMuxIn_R10),
    .data11(BusMuxIn_R11),
    .data12(BusMuxIn_R12),
    .data13(BusMuxIn_R13),
    .data14(BusMuxIn_R14),
    .data15(BusMuxIn_R15),
    .data16(BusMuxIn_LO),
    .data17(BusMuxIn_HI),
    .data18(BusMuxIn_Zlow),
    .data19(BusMuxIn_Zhigh),
    .data20(BusMuxIn_PC),
    .data21(BusMuxIn_MDR),
    .data22(BusMuxIn_InPort),
    .data23(c_sign_extended),
    .data24(0),
    .data25(0),
    .data26(0),
    .data27(0),
    .data28(0),
    .data29(0),
    .data30(0),
    .data31(0),
```

```
    .s(s),
    .out(BusMuxOut)
    );
endmodule
```

The 32 to 1 multiplexer code is as follows:

```
module mux32to1(data0, data1, data2, data3, data4, data5, data6, data7, data8,
data9, data10, data11, data12, data13, data14, data15, data16, data17, data18,
data19, data20, data21, data22, data23, data24, data25, data26, data27, data28,
data29, data30, data31, out, s);
    input wire [31:0] data0, data1, data2, data3, data4, data5, data6, data7,
data8, data9, data10, data11, data12, data13, data14, data15, data16, data17,
data18, data19, data20, data21, data22, data23, data24, data25, data26, data27,
data28, data29, data30, data31;
    input wire [4:0] s;
    output reg [31:0] out;

    always @(*) begin
        if (s==5'd31) out <= data31;
        else if (s==5'd30) out <= data30;
        else if (s==5'd29) out <= data29;
        else if (s==5'd28) out <= data28;
        else if (s==5'd27) out <= data27;
        else if (s==5'd26) out <= data26;
        else if (s==5'd25) out <= data25;
        else if (s==5'd24) out <= data24;
        else if (s==5'd23) out <= data23;
        else if (s==5'd22) out <= data22;
        else if (s==5'd21) out <= data21;
        else if (s==5'd20) out <= data20;
        else if (s==5'd19) out <= data19;
        else if (s==5'd18) out <= data18;
        else if (s==5'd17) out <= data17;
        else if (s==5'd16) out <= data16;
        else if (s==5'd15) out <= data15;
        else if (s==5'd14) out <= data14;
        else if (s==5'd13) out <= data13;
        else if (s==5'd12) out <= data12;
        else if (s==5'd11) out <= data11;
        else if (s==5'd10) out <= data10;
        else if (s==5'd9) out <= data9;
        else if (s==5'd8) out <= data8;
        else if (s==5'd7) out <= data7;
        else if (s==5'd6) out <= data6;
```

```
        else if (s==5'd5) out <= data5;
        else if (s==5'd4) out <= data4;
        else if (s==5'd3) out <= data3;
        else if (s==5'd2) out <= data2;
        else if (s==5'd1) out <= data1;
        else if (s==5'd0) out <= data0;
    end
endmodule
```

The 32 to 5 encoder code is as follows:

```
module encoder32to5 (input wire [31:0]ein, output reg [4:0] eout);
    always @(ein) begin
        if(ein[31]==1) eout=5'b11111;
        else if (ein[30]==1) eout=5'b11110;
        else if (ein[29]==1) eout=5'b11101;
        else if (ein[28]==1) eout=5'b11100;
        else if (ein[27]==1) eout=5'b11011;
        else if (ein[26]==1) eout=5'b11010;
        else if (ein[25]==1) eout=5'b11001;
        else if (ein[24]==1) eout=5'b11000;
        else if (ein[23]==1) eout=5'b10111;
        else if (ein[22]==1) eout=5'b10110;
        else if (ein[21]==1) eout=5'b10101;
        else if (ein[20]==1) eout=5'b10100;
        else if (ein[19]==1) eout=5'b10011;
        else if (ein[18]==1) eout=5'b10010;
        else if (ein[17]==1) eout=5'b10001;
        else if (ein[16]==1) eout=5'b10000;
        else if (ein[15]==1) eout=5'b01111;
        else if (ein[14]==1) eout=5'b01110;
        else if (ein[13]==1) eout=5'b01101;
        else if (ein[12]==1) eout=5'b01100;
        else if (ein[11]==1) eout=5'b01011;
        else if (ein[10]==1) eout=5'b01010;
        else if (ein[9]==1) eout=5'b01001;
        else if (ein[8]==1) eout=5'b01000;
        else if (ein[7]==1) eout=5'b00111;
        else if (ein[6]==1) eout=5'b00110;
        else if (ein[5]==1) eout=5'b00101;
        else if (ein[4]==1) eout=5'b00100;
        else if (ein[3]==1) eout=5'b00011;
        else if (ein[2]==1) eout=5'b00010;
        else if (ein[1]==1) eout=5'b00001;
```

```
        else eout=5'bx;
    end
endmodule
```

MDR Unit

The MDR Unit code uses a 2-to-1 multiplexer (mux2to1) and a 32-bit register. The select signal for the multiplexer is the read value which determines if Mdatain is selected or the BusMuxOut. The code below is for the MDR unit:

```
module MD_reg32(
    input clr, clk, read, MDRin,
    input [31:0] BusMuxOut, Mdatain,
    output [31:0] Q
);

    wire [31:0] D;
     // instantiate a 2-to-1 mux with select signal as 'read'
     mux2to1 MDMux(
        .data_1(BusMuxOut),
        .data_2(Mdatain),
        .select(read),
        .out(D)
    );
     // instantiate a 32-bit register with enable signal as 'MDRin'
    reg32bit MDR(
        .clr(clr),
        .clk(clk),
        .enable(MDRin),
        .D(D), // connect the output of the mux to the input of the register
        .Q(Q) // connect the output of the register to the output of the module
    );

endmodule
```

Code for the 2-to-1 mux is shown below:

```
module mux2to1(data_1, data_2, out, select);
    input wire [31:0] data_1, data_2;
    input wire select;
    output reg [31:0]out;

    always@(*)begin
        if (select)
```

```
            out <= data_2;
        else
            out <= data_1;
    end
endmodule
```

ALU

Two different ALUs were created. Both ALUs were set up such that all calculations would occur asynchronously, then depending on the opcode given to it by the IR, it would select which data to place in the C register (64-bit register) that then would feed the Zlow and Zhigh registers too hold the lower and upper 32 bits of the 64-bit outputs. The Zhigh register holds the remainder value for division. The first ALU used multiple modules to execute arithmetic. The other used verilog's internal bitwise, and mathematical operations to calculate arithmetic. The first ALUs code is as follows:

```
`timescale 1ns/10ps

module alu (A, B, opcode, clk, clr, control, C);
    input wire [31:0] A, B, control;
    input wire clk, clr;
    input wire [4:0] opcode;
    output reg [63:0] C;


    wire[31:0] neg_reg, nor_reg, or_reg, not_reg, xor_reg, and_reg, shl_reg,
shr_reg, shra_reg, ror_reg, rol_reg, add_reg, sub_reg,
    add_cout_reg, sub_cout_reg;
    wire [63:0] mul_reg, div_reg;


    parameter add = 5'b00001, sub = 5'b00010, mul = 5'b00011, div = 5'b00100, shr
= 5'b00101, shl = 5'b00110, shra = 5'b00111,
                    ror = 5'b01000, rol = 5'b01001, log_and = 5'b01010, log_or =
5'b01011, log_neg = 5'b01100, log_xor = 5'b01101,
                    log_nor = 5'b01110, log_not = 5'b01111;

    cla32bit addop (.a(A), .b(B), .cin(1'd0), .s(add_reg),
.cout(add_cout_reg[0]));
    sub32 subop (.a(A), .b(B), .cin(1'd0), .s(sub_reg), .cout(sub_cout_reg));
    mul32 mulop (.multiplicand(A), .multiplier(B), .product(mul_reg));
    div32 divop (.dividend(A), .divisor(B), .out(div_reg));
    and32 andop (.a(A), .b(B), .c(and_reg));
    or32 orop (.a(A), .b(B), .c(or_reg));
    xor32 xorop (.a(A), .b(B), .c(xor_reg));
    neg32 negop (.in(B), .out(neg_reg));
    nor32 norop (.a(A), .b(B), .c(nor_reg));
```

```
    not32 notop (.in(B), .out(not_reg));
    shl32 shlop (B, A, shl_reg);
    shr32 shrop (B, A, shr_reg);
    shra32 shraop (B, A, shra_reg);
    ror32 rorop (B, A, ror_reg);
    rol32 rolop (B, A, rol_reg);


    always @(posedge clk) begin
        //if (opcode) begin -- don't need this since ur using cases
            case(opcode)
                add: begin
                    C [31:0] <= add_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                sub: begin
                    C [31:0] <= sub_reg [31:0];
                    C [63:32] <= 32'd0;
                end
                mul: begin
                    C [63:0] <= mul_reg [63:0];
                end
                div: begin
                    C [63:0] <= div_reg[63:0];
                end
                shr: begin
                    C [31:0] <= shr_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                shl: begin
                    C [31:0] <= shl_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                shra: begin
                    C [31:0] <= shra_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                ror: begin
                    C [31:0] <= ror_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                rol: begin
                    C [31:0] <= rol_reg[31:0];
                    C [63:32] <= 32'd0;
                end
```

```
                log_and: begin
                    C [31:0] <= and_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                log_or: begin
                    C [31:0] <= or_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                log_neg: begin
                    C [31:0] <= neg_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                log_xor: begin
                    C [31:0] <= xor_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                log_nor: begin
                    C [31:0] <= nor_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                log_not: begin
                    C [31:0] <= not_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                default: begin
                    C <= 64'd0;
                end
            endcase
        end
    //end
endmodule
```

The second ALUs code is as follows:

```
module alu_test #(parameter wordSize = 32)(
    input wire clk, clr,
    //32-bit input registers A, B,
    input wire [wordSize-1:0] A, B,
    //5-bit opcode
    input wire [4:0] opcode,
    //64-bit output register C
    output reg [(wordSize*2)-1:0] C
);
```

```
    parameter   add = 5'b00001, sub = 5'b00010, mul = 5'b00011, div = 5'b00100,
shr = 5'b00101, shl = 5'b00110, shra = 5'b00111,
                ror = 5'b01000, rol = 5'b01001, log_and = 5'b01010, log_or =
5'b01011, log_neg = 5'b01100, log_xor = 5'b01101,
                log_nor = 5'b01110, log_not = 5'b01111;
    wire tempAdd, tempSub;
    wire [wordSize-1:0] temp32_out, add32_out, sub32_out;
    wire [(wordSize*2)-1:0] temp64_out, mul64_out, div64_out;

    alu_32add myAdd (.a(A), .b(B), .cin(1'd0), .s(add32_out), .cout(tempAdd));

    wire [wordSize-1:0] neg_b;
    assign neg_b = ~B + 1;
    alu_32add mySub (.a(A), .b(neg_b), .cin(1'd0), .s(sub32_out),
.cout(tempSub));

    alu_32div myDiv (.dividend(A), .divisor(B), .out(div64_out));
    alu_32mul myMul (.multiplicand(A), .multiplier(B), .product(mul64_out));

    always @(posedge clk) begin

        case(opcode)
            add: begin
                C [31:0] <= add32_out[31:0];
                C [63:32] <= (add32_out[31] == 0) ? 32'd0 : 32'd1;
            end
            sub: begin
                // Calculate two's complement of b
                C [31:0] <= sub32_out[31:0];
                C [63:32] <= 32'd0;
            end
            mul: begin
                C [63:0] <= mul64_out [63:0];
            end
            div: begin
                C [63:0] <= div64_out[63:0];
            end
            shr: begin
                C [31:0] <= A >> B;
                C [63:32] <= 32'd0;
            end
            shl: begin
                C [31:0] <= A << B;
                C [63:32] <= 32'd0;
            end
```

```
            shra: begin
                C [31:0] <= A >>> B;
                C [63:32] <= 32'd0;
            end
            ror: begin
                C [31:0] <= (A >> B) | (A << ~B);
                C [63:32] <= 32'd0;
            end
            rol: begin
                C [31:0] <= (A << B) | (A >> ~B);
                C [63:32] <= 32'd0;
            end
            log_and: begin
                C [31:0] <= A & B;
                C [63:32] <= 32'd0;
            end
            log_or: begin
                C [31:0] <= A | B;
                C [63:32] <= 32'd0;
            end
            log_neg: begin
                C [31:0] <= -A;
                C [63:32] <= 32'd0;
            end
            log_xor: begin
                C [31:0] <= A ^ B;
                C [63:32] <= 32'd0;
            end
            log_nor: begin
                C [31:0] <= ~(A | B);
                C [63:32] <= 32'd0;
            end
            log_not: begin
                C [31:0] <= ~A;
                C [63:32] <= 32'd0;
            end
            default: begin
                C <= 64'd0;
            end
        endcase
    end
endmodule
```

Addition

The addition was done using a hierarchical carry-lookahead adder. The code for the CLA is as follows:

```
//Carry look ahead adder
module cla32bit (a,b,cin,s,cout);
    input wire [31:0] a,b;
    input wire cin;
    output wire [31:0] s;
    output wire cout;
    wire c1;

    cla16bit cla1 (.a(a[15:0]), .b(b[15:0]), .s(s[15:0]), .cin(cin), .cout(c1));
    cla16bit cla2 (.a(a[31:16]), .b(b[31:16]), .s(s[31:16]), .cin(c1),
.cout(cout));
endmodule

module cla16bit (a,b,cin,s,cout);
    input wire [15:0] a,b;
    input wire cin;
    output wire [15:0] s;
    output wire cout;
    wire c1, c2, c3;

    cla4bit cla1 (.a(a[3:0]), .b(b[3:0]), .s(s[3:0]), .cin(cin), .cout(c1));
    cla4bit cla2 (.a(a[7:4]), .b(b[7:4]), .s(s[7:4]), .cin(c1), .cout(c2));
    cla4bit cla3 (.a(a[11:8]), .b(b[11:8]), .s(s[11:8]), .cin(c2), .cout(c3));
    cla4bit cla4 (.a(a[15:12]), .b(b[15:12]), .s(s[15:12]), .cin(c3),
.cout(cout));
endmodule


module cla4bit (a,b,cin,s,cout);
    input wire [3:0] a,b;
    input wire cin;
    output wire [3:0] s;
    output wire cout;
    wire c1, c2, c3;

    cla cla1 (.a(a[0]), .b(b[0]), .s(s[0]), .cin(cin), .cout(c1));
    cla cla2 (.a(a[1]), .b(b[1]), .s(s[1]), .cin(c1), .cout(c2));
    cla cla3 (.a(a[2]), .b(b[2]), .s(s[2]), .cin(c2), .cout(c3));
    cla cla4 (.a(a[3]), .b(b[3]), .s(s[3]), .cin(c3), .cout(cout));
endmodule


module cla (a,b,s,cin,cout);
```

```
    input wire a,b,cin;
    output wire s,cout;

    wire xor1, and1, and2;

    assign xor1 = ((a)^(b));
    assign and1 = ((a)&(b));
    assign and2 = ((xor1)&(cin));
    assign s = ((xor1)^(cin));
    assign cout = ((and1)|(and2));
endmodule
```

Subtraction

Subtraction was done by using the negate module and the above add module. First, the value being subtracted is negated then, using the adder the newly generated negated value and the untouched value are added together. The code is as follows:

```
module sub32 (a, b, cin, s, cout);
    input wire signed [31:0] a, b;
    input wire cin;
    output wire signed [31:0] s;
    output wire cout;
    wire [31:0] temp;

    neg32 negate(.in(b), .out(temp));
    cla32bit add(.a(a), .b(temp), .cin(cin), .s(s), .cout(cout));
endmodule
```

Multiplication

Multiplication was using Booth's algorithm and bit-pair encoding. The output is a 64 bit register, in case the value is too large. Encoding was done by using a case statement to check the bottom 3 bits of the multiplier and applying the corresponding operation to the multiplicand. This was done 16 times and the value of the multiplicand was added on top of itself through each iteration.

```
module mul32 (
    input signed [31:0] multiplicand,
    input signed [31:0] multiplier,
    output reg signed [63:0] product
);

reg signed [31:0] multiplicand_reg;
reg signed [31:0] multiplier_reg;
reg signed [2:0] bitPattern;
reg signed [31:0] temp;
```

```
integer i;

always @(*) begin
    // Initialization
    multiplicand_reg <= (multiplicand <<< 1);
    temp <= 64'b0;

    for (i = 0; i < 16; i = i + 1) begin

        multiplier_reg <= multiplier;
        bitPattern = multiplier_reg[2:0];

        case (bitPattern)
            // +1 x M
            3'b001, 3'b010: temp <= temp + multiplicand_reg;
            // +2 x M
            3'b011: temp <= temp + (multiplicand_reg + multiplicand_reg);
            // -2 x M
            3'b100: temp <= temp - (multiplicand_reg + multiplicand_reg);
            // -1 x M
            3'b101, 3'b110: temp <= temp - multiplicand_reg;
        endcase

        multiplicand_reg <= multiplicand_reg <<< 2;
        multiplier_reg <= multiplier_reg >>> 2;
    end

    product <= temp;
end
endmodule
```

Division

Division was achieved using the non-restoring algorithm and the restoring algorithm. The non-restoring algorithm was used in the final design of the CPU. In the non-restoring algorithm, we first check if either dividend or divisor is negative and we flip their sign. Then we create a 64-bit register temp that's lower bits contain the value of the dividend. For 32 iterations (for the 32 bits in the quotient), we shift the temp register 1 to the left, and depending on whether the value in temp is negative or positive, we add the divisor to the top 32 bits or subtract it respectively. Then, if the value is still negative, we set the 0^{th} bit of the temp register to 0, otherwise, it's set to 1. The code for the non-restoring algorithm is as follows:

```
module alu_32div(
    input signed [31:0] dividend,
    input signed [31:0] divisor,
```

```
    output reg signed [63:0] out
);
    reg signed [63:0] temp;
    reg signed [31:0] D, O;
    integer i;

    always @(*) begin
        if (dividend[31])
            D = -dividend;
        else
            D = dividend;

        if (divisor[31])
            O = -divisor;
        else
            O = divisor;

        // Initialize temp with D and 32 zeros (A/Q reg)
        temp = {32'h0, D};
        // Perform division for 32 iterations
        for (i = 0; i < 32; i = i + 1) begin
            // Shift temp left by one bit
            temp = temp << 1;

            // If the most significant bit of temp is one,
            // add O to upper half of temp.
            if (temp[63] == 1'b1)
                temp[63:32] = temp[63:32]+ O;

            // Otherwise, subtract O from upper half of temp.
            else
                temp[63:32] = temp[63:32] - O;
            // If the most significant bit of temp is one,
            // set the least significant bit of quotient to zero.
            if (temp[63] == 1'b1)
                temp[0] = 1'b0;
            // Otherwise, set it to one.
            else
                temp[0] = 1'b1;
        end

        if ((dividend[31])||(divisor[31])) begin
            if (!((dividend[31])&&(divisor[31])))
                temp[31:0] = -temp[31:0];
        end
```

```
        // If the most significant bit of temp is one,
        // add 0 to remainder and set out accordingly.

        if (temp[63] == 1'b1) begin
            out = {temp[63:32] + 0, temp[31:0]};

        // Otherwise, set out accordingly without adding 0 to remainder.
        end else begin
            out = {temp[63:32],temp[31:0]};
        end
    end
endmodule
```

The restoring algorithm works similarly, but the value in the temp register isn't subtracted from if the value were to become negative post subtraction. The code is as follows:

```
module alu_32div(
    input signed [31:0] dividend,
    input signed [31:0] divisor,
    output reg signed [63:0] out
);
    reg signed [63:0] temp;
    integer i;

    always @(*) begin
        // Initialize temp with dividend and 32 zeros (A/Q reg)
        temp = {32'h0, dividend};

        // Perform division for 32 iterations
          for (i = 0; i < 32; i = i + 1) begin
            // Shift temp left by one bit
            temp = temp << 1;

            // If the most significant bit of temp is one,
            // add divisor to upper half of temp.
            if (temp[63] == 1'b1)
                temp[63:32] = temp[63:32] + divisor;

            // Otherwise, subtract divisor from upper half of temp.
             else
                temp[63:32] = temp[63:32] - divisor;


            // If the most significant bit of temp is one,
```

```
            // set the least significant bit of quotient to zero.
            if (temp[63] == 1'b1)
                temp[0] = 1'b0;
            // Otherwise, set it to one.
            else
                temp[0] = 1'b1;

        end

        // If the most significant bit of temp is one,
        // add divisor to remainder and set out accordingly.
        if (temp[63] == 1'b1) begin
            out = {temp[63:32] + divisor, temp[31:0]};

        // Otherwise, set out accordingly without adding divisor to remainder.
        end else begin
            out = {temp[63:32],temp[31:0]};
        end
    end
endmodule
```

Shift Operations

The code for the shift right operation is as follows:

```
module shr32 (in, num_shifts, out);
    input wire signed[31:0] in;
    input wire [31:0] num_shifts;
    output reg signed[31:0] out;

    always @(*) begin
        out[31:0] = in>>num_shifts;
    end
endmodule
```

The code for the shift right arithmetic is as follows:

```
module shra32 (in, num_shifts, out);
    input wire signed [31:0] in;
    input wire [31:0] num_shifts;
    output reg signed [31:0] out;

    always @(*) begin
        out[31:0] = in>>>num_shifts;
    end
endmodule
```

The shift left code is as follows:

```
module shl32 (in, num_shifts, out);
    input wire signed [31:0] in;
    input wire [31:0] num_shifts;
    output reg signed[31:0] out;

    always @(*) begin
        out[31:0] = in<<num_shifts;
    end
endmodule
```

The shift left arithmetic code is as follows:

```
module shla32 (in, num_shifts, out);
    input wire signed [31:0] in;
    input wire [31:0] num_shifts;
    output reg signed [31:0] out;

    always @(*) begin
        out[31:0] = in<<<num_shifts;
    end
endmodule
```

The rotate left and right functions are more complex than the normal shift operators. They are computed by shifting the input value to the right by num_shifts bits and then ORing it with the input value shifted to the left by negation of num_shifts bits. The result of this operation is assigned to the output out. The code for the rotate right is as follows:

```
module ror32 (in, num_shifts, out);
    input wire signed [31:0] in;
    input wire signed [31:0] num_shifts;
    output wire signed [31:0] out;

    assign out = (in >> num_shifts) | (in << ~num_shifts);

endmodule
```

Rotate left works similarly to rotate right, however instead the negation is shifted right and the normal input is shifted left. The code for the rotate left is as follows:

```
module rol32 (in, num_shifts, out);
    input wire signed[31:0] in;
    input wire signed[31:0] num_shifts;
    output wire signed[31:0] out;

    assign out = (in << num_shifts) | (in >> ~num_shifts);
endmodule
```

Bitwise ALU Operations:

All bitwise operations use the similar format of using the bitwise verilog operators on each bit by looping 32 times through all bit inputs.

The code for the AND operation is as follows:

```
module and32 (a, b, c);
    input wire [31:0] a, b;
    output reg [31:0] c;
    reg [31:0] temp;
    integer i;

    always @(*) begin
        for (i = 0; i < 31; i = i + 1) begin
            temp[i] = ((a[i])&(b[i]));
        end
        c = temp;
    end
endmodule
```

The code for the OR operation is as follows:

```
 module or32 (a,b,c);
    input wire [31:0]a,b;
    output wire [31:0]c;
    genvar i;

    generate
        for (i = 0; i < 31; i = i + 1) begin : loop
            assign c[i] = ((a[i])|(b[i]));
        end
    endgenerate
endmodule
```

The XOR code is as follows:

```
module xor32 (a,b,c);
    input wire [31:0]a,b;
    output reg [31:0]c;
    integer i;

    always @(*)begin
        for (i = 0; i < 31; i = i + 1) begin
            c[i] = ((a[i])^(b[i]));
        end
    end
endmodule
```

The NOT operation code is as follows:

```
module not32 (in, out);
    input wire signed[31:0] in;
    output wire signed[31:0] out;

    genvar i;
    generate
        for (i=0; i<32; i = i+1) begin : loop
            assign out[i] = !in[i];
        end
    endgenerate
endmodule
```

The NOR operation code is as follows:

```
module nor32 (a,b,c);
    input wire [31:0]a,b;
    output reg [31:0]c;
    integer i;
    always @(*)begin
        for (i = 0; i < 31; i = i + 1) begin
            c[i] = ((a[i])~^(b[i]));
        end
    end
endmodule
```

The NEG (negate) code works by first using the NOT operation code to NOT the input, then the Carry Lookahead Module to add 1 to the value. This is done to create the 2's compliment of the input values.

```
module neg32 (in, out);
    input wire signed[31:0] in;
```

```
    output wire signed[31:0] out;
    wire [31:0] temp;
    wire cout;

    not32 notop (.in(in),.out(temp));
    cla32bit add_op (.a(temp), .b(32'd1), .cin(1'd0), .s(out), .cout(cout));
endmodule
```

Testbenching Files and Simulation Outputs

Division Testbench

The division Testbench is the first Testbench file we wrote for testing our datapath and alu. The Testbench has 6 states for loading in values to the registers R6, R7, and R1. The other 7 states T0-6 are for moving the data on the bus so that the alu computes the division and the puts the values in the HI and LO registers. The LO register holds the quotient and the HI holds the remainder.

In the default state, all signals are set to low. In the Reg_load1a and Reg_load1b, the immediate values we will be using to input to our registers are placed on the bus and then in the registers. Subsequent reg loading states do similar things. T0 begins by placing the program counter on the bus, and incrementing the program counter. T1 places the new PC value in the program counter, and loads the instruction from memory to be placed on the bus. T2 loads in the instruction into the IR register from the bus. T3 places the value of R6 in the bus and then enables the Y register to take the value as an input. T4 places R7 on the bus and also gives the alu the corresponding opcode. T5 places the value in the Zlow register on the bus and loads it into the LO register. T6 places the Zhigh register contents on the bus and loads it into the HI register.

The code for the Testbench is shown below:

```
`timescale 1ns/1ns
//Testbench does the following operation:
// div LO/HI , R2, R3, (remainder is )
// R2 holds 16, R3 holds -2

module div_tb;
    reg clk = 0;
    reg clr = 0;
    reg R0in, R1in, R2in, R3in, R4in, R5in, R6in, R7in, R8in, R9in, R10in, R11in,
R12in, R13in, R14in, R15in;
    reg HIin, LOin, Zin, incPC, MARin, MDRin, Read, InPortin, Cin, Yin;
    reg R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, R8out, R9out,
R10out, R11out, R12out, R13out, R14out, R15out;
    reg HIout, LOout, ZLowOut, ZHighOut, MDRout, Cout, InPortOut, PCout, PCin;
    reg [4:0] opcode;
    reg [31:0] Mdatain;
```

```
    reg IRin; //just added
    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, T6 = 4'b1101;

    reg [3:0] cur_state = Default;

    datapath DUT(
        .clr(clr),
        .clk(clk),
        .R0in(R0in),
        .R1in(R1in),
        .R2in(R2in),
        .R3in(R3in),
        .R4in(R4in),
        .R5in(R5in),
        .R6in(R6in),
        .R7in(R7in),
        .R8in(R8in),
        .R9in(R9in),
        .R10in(R10in),
        .R11in(R11in),
        .R12in(R12in),
        .R13in(R13in),
        .R14in(R14in),
        .R15in(R15in),
        .PCin(PCin),
        .HIin(HIin),
        .LOin(LOin),
        .Zin(Zin),
        .incPC(incPC),
        .MARin(MARin),
        .MDRin(MDRin),
        .Read(Read),
        .InPortin(InPortin),
        .Cin(Cin),
        .Yin(Yin),
        .opcode(opcode),
        .Mdatain(Mdatain),

        .R0out(R0out),
        .R1out(R1out),
```

```
        .R2out(R2out),
        .R3out(R3out),
        .R4out(R4out),
        .R5out(R5out),
        .R6out(R6out),
        .R7out(R7out),
        .R8out(R8out),
        .R9out(R9out),
        .R10out(R10out),
        .R11out(R11out),
        .R12out(R12out),
        .R13out(R13out),
        .R14out(R14out),
        .R15out(R15out),
        .HIout(HIout),
        .LOout(LOout),
        .ZHighOut(ZHighOut),
        .ZLowOut(ZLowOut),
        .PCout(PCout),
        .MDRout(MDRout),
        .InPortOut(InPortOut),
        .Cout(Cout)
        );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    always @(posedge clk) begin
        case (cur_state)
            Default     : #40 cur_state = Reg_load1a;
            Reg_load1a  : #40 cur_state = Reg_load1b;
            Reg_load1b  : #40 cur_state = Reg_load2a;
            Reg_load2a  : #40 cur_state = Reg_load2b;
            Reg_load2b  : #40 cur_state = Reg_load3a;
            Reg_load3a  : #40 cur_state = Reg_load3b;
            Reg_load3b  : #40 cur_state = T0;
            T0          : #40 cur_state = T1;
            T1          : #40 cur_state = T2;
            T2          : #40 cur_state = T3;
            T3          : #40 cur_state = T4;
            T4          : #40 cur_state = T5;
            T5          : #40 cur_state = T6;
        endcase
```

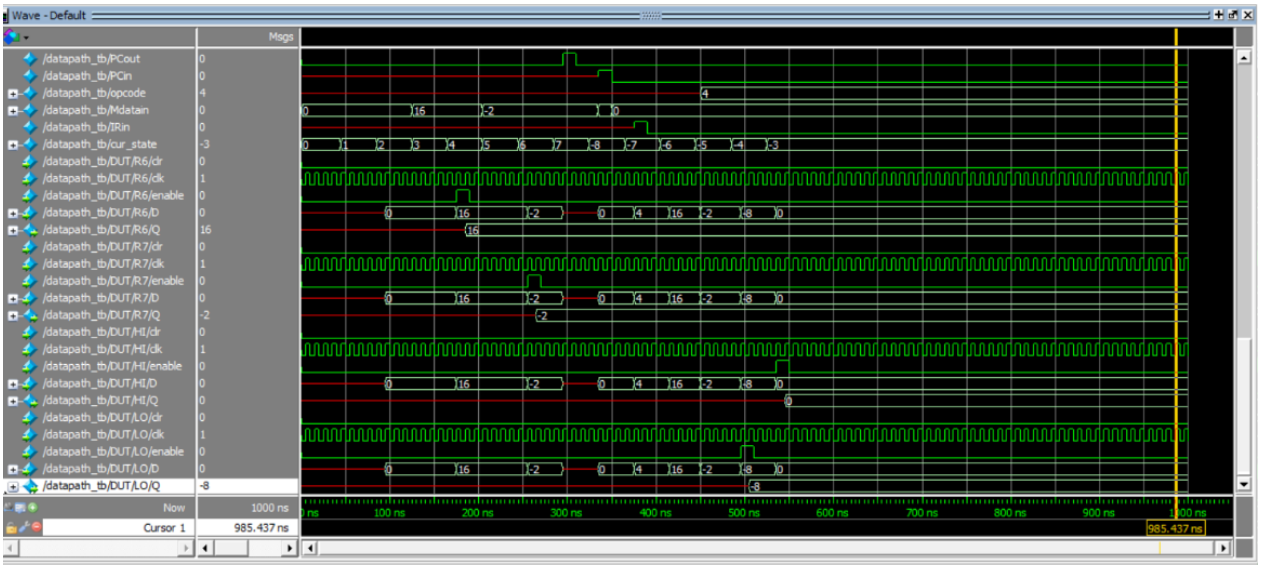
```
    end

    always @(cur_state) begin
        case (cur_state)
            Default: begin //Initialize everything to 0
                R1in <= 0;
                R2in <= 0;
                R3in <= 0;
                R4in <= 0;
                R5in <= 0;
                R6in <= 0;
                R7in <= 0;
                R8in <= 0;
                R9in <= 0;
                R10in <= 0;
                R11in <= 0;
                R12in <= 0;
                R13in <= 0;
                R14in <= 0;
                R15in <= 0;
                HIin <= 0;
                LOin <= 0;
                Mdatain <= 32'b0;
                R1out <= 0;
                R2out <= 0;
                R3out <= 0;
                R4out <= 0;
                R5out <= 0;
                R6out <= 0;
                R7out <= 0;
                R8out <= 0;
                R9out <= 0;
                R10out <= 0;
                R11out <= 0;
                R12out <= 0;
                R13out <= 0;
                R14out <= 0;
                R15out <= 0;
                HIout <= 0;
                LOout <= 0;
                PCout <= 0;
                ZHighOut <= 0;
                ZLowOut <= 0;
                MDRout <= 0;
                InPortOut <= 0;
```

```
            InPortin <= 0;
            Cout <= 0;
        end

        Reg_load1a: begin
            Mdatain <= 0;
            #10 Read <= 1; MDRin <= 1;
            #15 Read <= 0; MDRin <= 0;
        end

        Reg_load1b: begin
            #10 MDRout <= 1; R1in <= 1;
            #15 MDRout <= 0; R1in <= 0;
        end

        Reg_load2a: begin
            Mdatain <= 16;
            #10 Read <= 1; MDRin <= 1;
            #15 Read <= 0; MDRin <= 0;
        end

        Reg_load2b: begin
            #10 MDRout <= 1; R6in <= 1;
            #15 MDRout <= 0; R6in <= 0;
        end
        Reg_load3a: begin
            Mdatain <= -2;
            Read = 0; MDRin = 0;
            #10 Read <= 1; MDRin <= 1;
            #15 Read <= 0; MDRin <= 0;
        end

        Reg_load3b: begin
            #10 MDRout <= 1; R7in <= 1;
            #15 MDRout <= 0; R7in <= 0;
        end

        T0: begin
            #10
            incPC <= 1;
            MARin <= 1;
            PCout <= 1;
            Zin <= 1;
            #15
            incPC <= 0;
```

```
            MARin <= 0;
            PCout <= 0;
            Zin <= 0;
        end
        T1: begin
            #10
            ZLowOut <= 1;
            PCin <= 1;
            Read <= 1;
            MDRin <= 1;
            Mdatain <= 4; //Same as opcode

            #15
            ZLowOut <= 0;
            PCin <= 0;
            Read <= 0;
            MDRin <= 0;
            Mdatain <= 0;
        end
        T2: begin
            #10
            MDRout <= 1;
            IRin <= 1;
            #15
            MDRout <= 0;
            IRin <= 0;
        end
        T3: begin
            #10
            R6out <= 1;
            Yin <= 1;
            #15
            R6out <= 0;
            Yin <= 0;
        end
        T4: begin
            #5
            R7out <= 1;
                 opcode <= 5'b00100;
                 #5
                 R7out <= 0;
                 #5
                 Zin <= 1;
                 #5
            Zin <= 0;
```

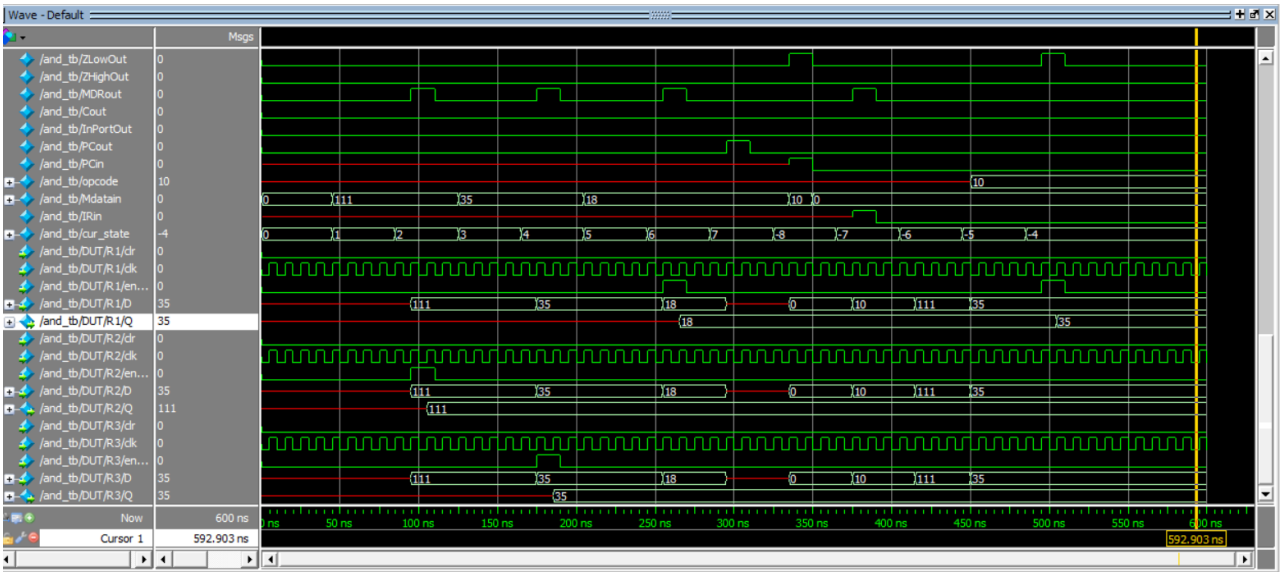
```
            end
            T5: begin
                #10
                ZLowOut <= 1;
                LOin <= 1;
                #15
                ZLowOut <= 0;
                LOin <= 0;
            end
            T6: begin
                #10
                ZHighOut <= 1;
                HIin <= 1;
                #15
                ZHighOut <= 0;
                HIin <= 0;
            end
        endcase
    end
endmodule
```

In the below simulation, the values 16 and -2 are placed in R6 and R7, computing div R6, R7. The value -8 goes into the LO register and 0 in the HI.



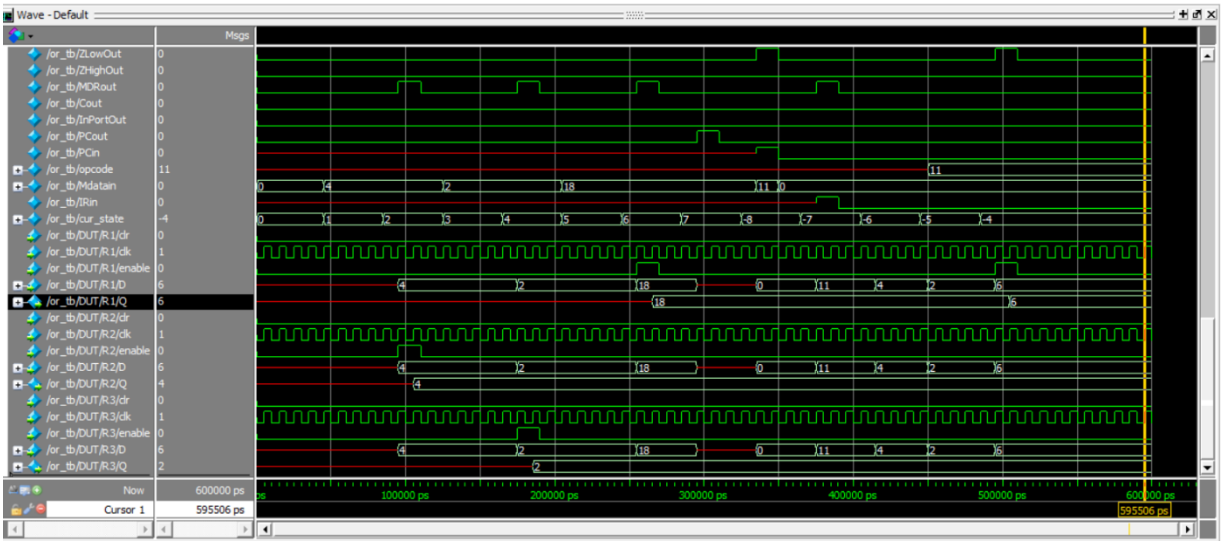
And Testbench

The and Testbench was changed to have one less cycle than the division operation since only one value is placed the R1 register. The Testbench is very similar, however different registers are used.



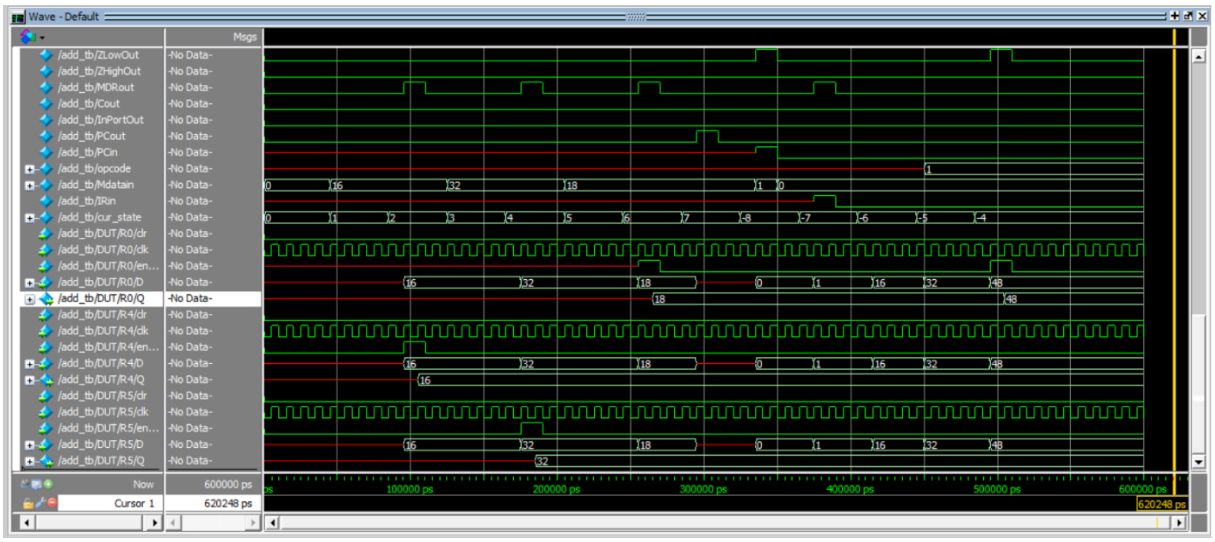
Or Testbench

Below is the Or Testbench's output: It used similar registers to the And Testbench.



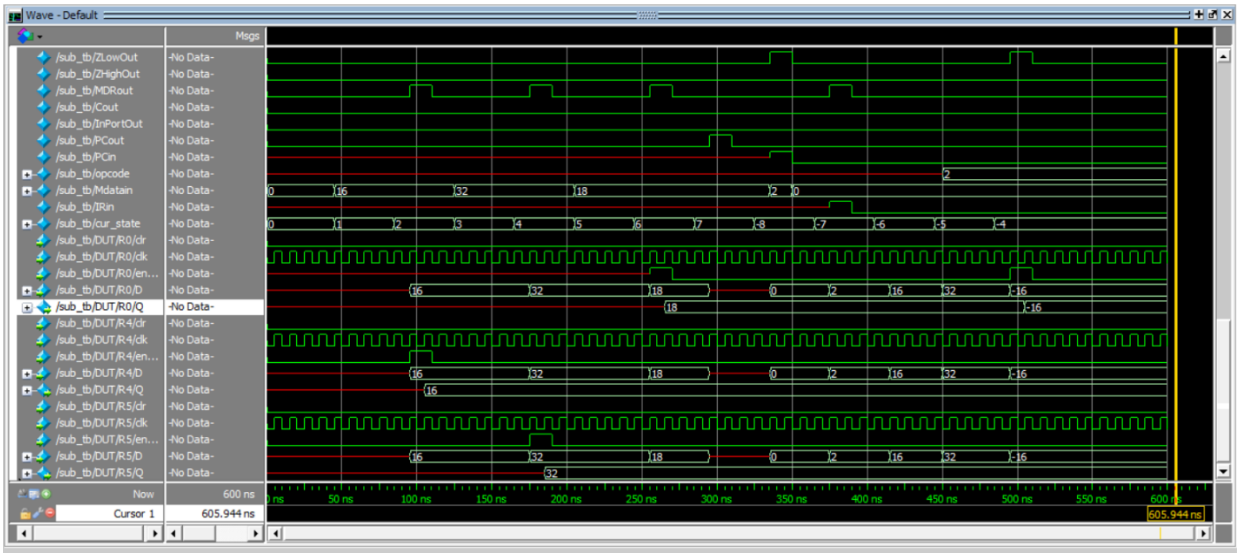
Add Testbench

The add Testbench is also similar to the And and Or Testbench's, however the sum is placed in the R0 register, and the R4 and R5 registers are used to give values for the operation to take place with. The values inputted were 32 and 16 and gave an output of 48 in register R0.



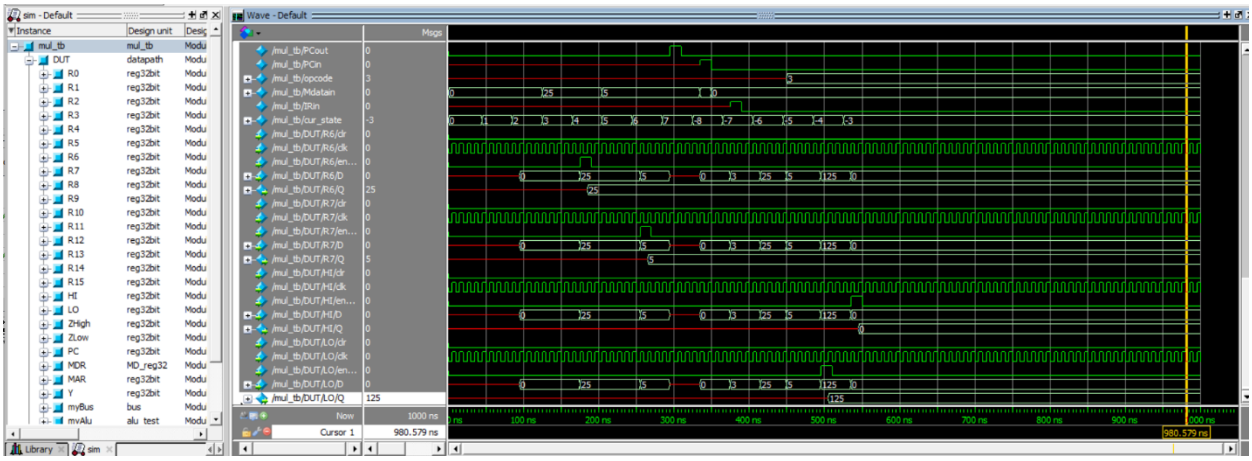
Sub Testbench

The sub Testbench was very similar to the Add Testbench and used the same numbers and registers. Only a different opcode. The output stored in the R0 register is -16, since 16 – 32 = -16.



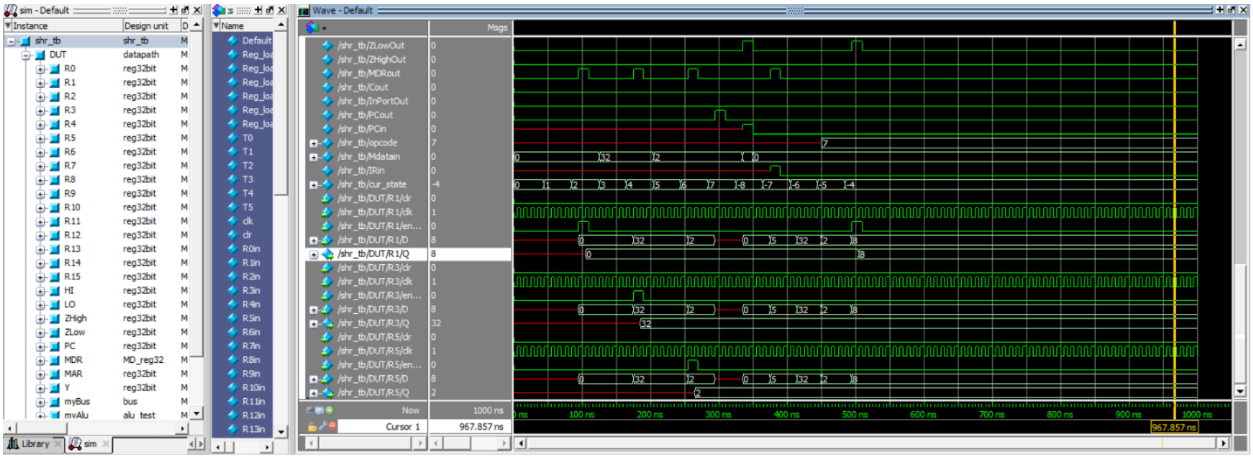
Multiplication Testbench

The multiplication Testbench is very similar to the division Testbench. The same registers are used and the values are stored in the HI and LO registers. The values in R6 and R7 are 25 and 5 respectively. The output is 125 in the LO register and 0 in the HI register.



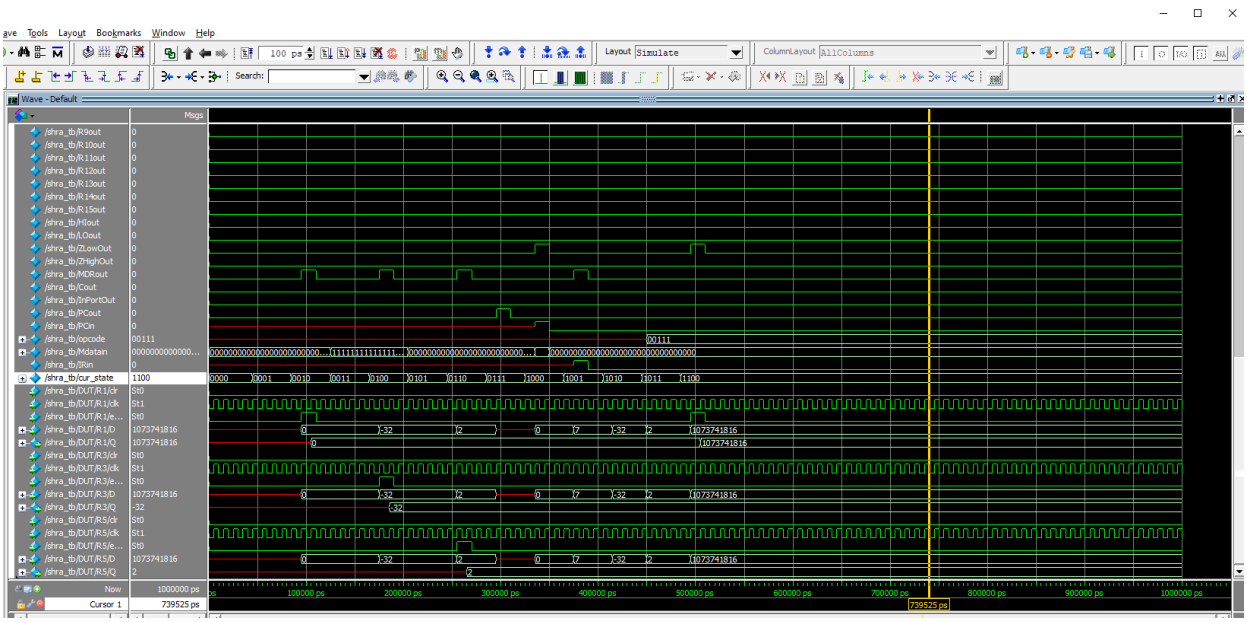
Shift Right Testbench

Below is the shift right simulation output. The operation being computed was SHR R1 R3 R5. Only cycles up to T5 were needed since we didn't have to deal with the HI and LO value outputs. The values loaded into R3 and R5 were 32 and 2 respectively. The output was 8 and was stored in R1.



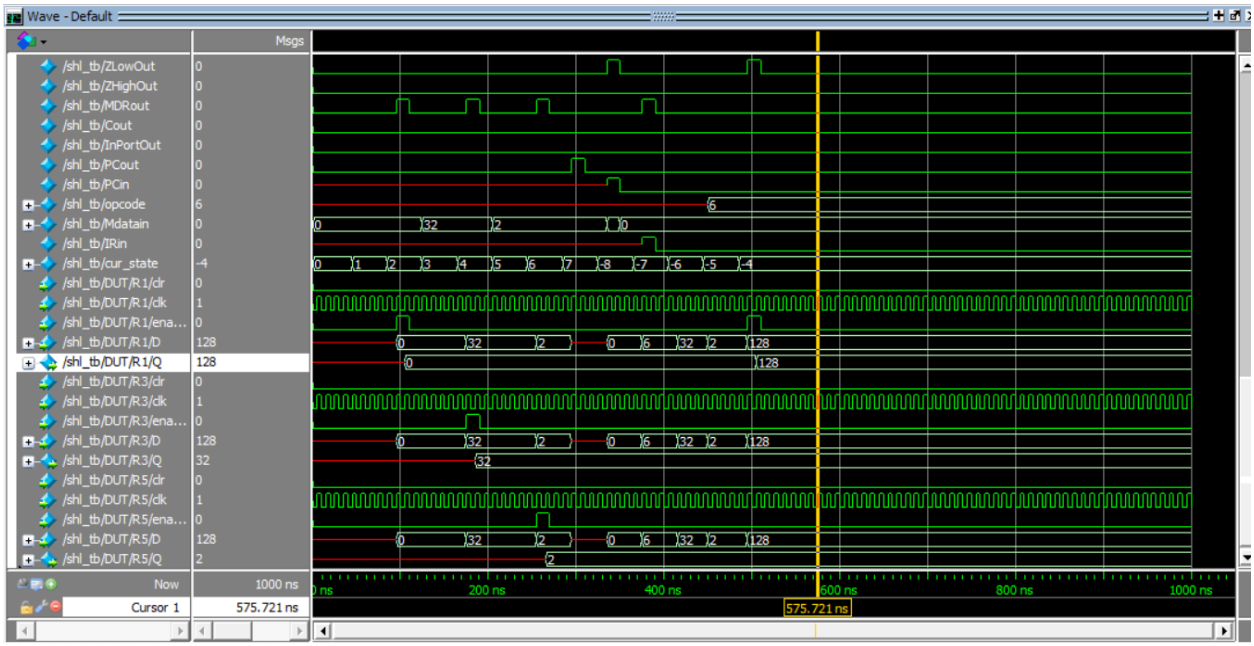
Shift Right Arithmetic Testbench

The shift right arithmetic Testbench code was very similar to the shift right code. The values placed in the R3 and R5 registers were -32 and 2 respectively. The output goes into the R1 register.



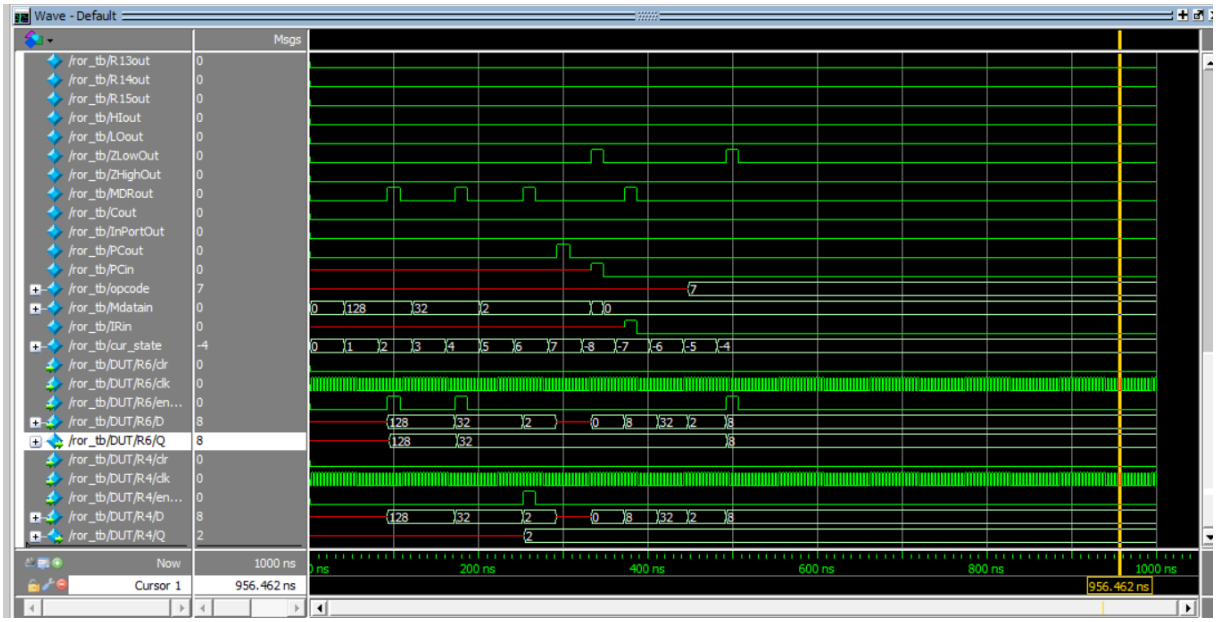
Shift Left Testbench

The Shift left Testbench is the same as the shift right Testbench, however the opcode is changed to compute a shift left operation. The output in register R1 is 128.



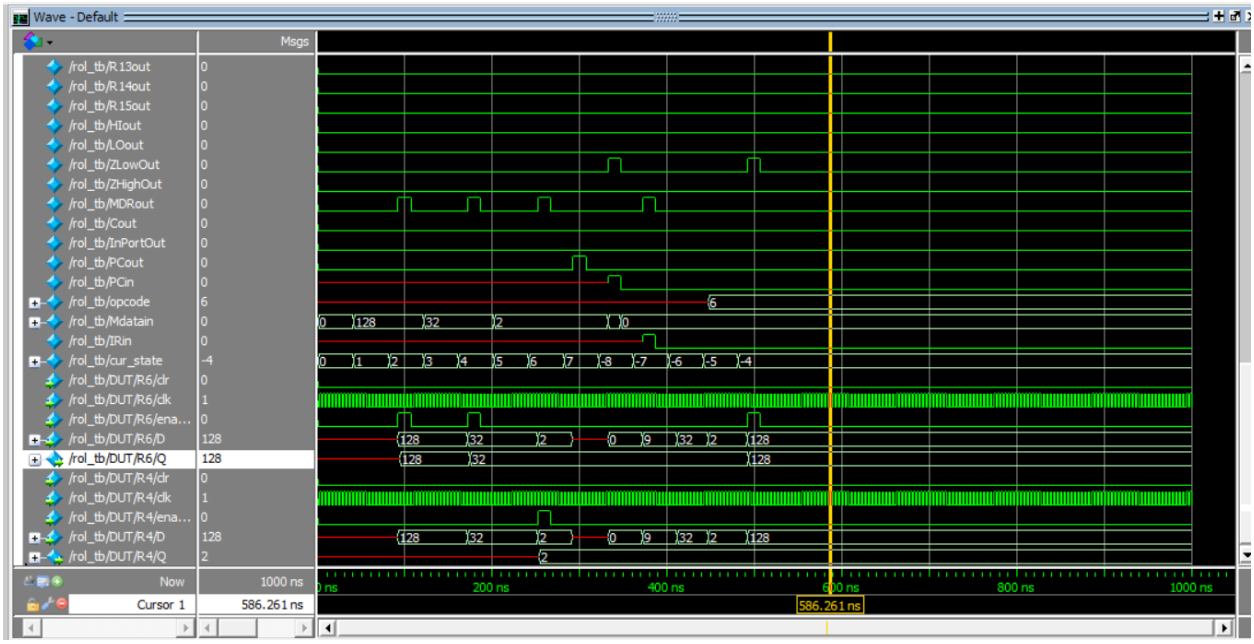
Rotate Right Testbench

The Rotate Right Testbench was very similar to the shift Testbenches in terms of what operations occur each cycle, however, the registers that were used were different. The rotate right Testbench computed ROR R6 R6 R4, and the initial values in R6 and R4 were 32 and 2. The output that was placed in R6 was 8.



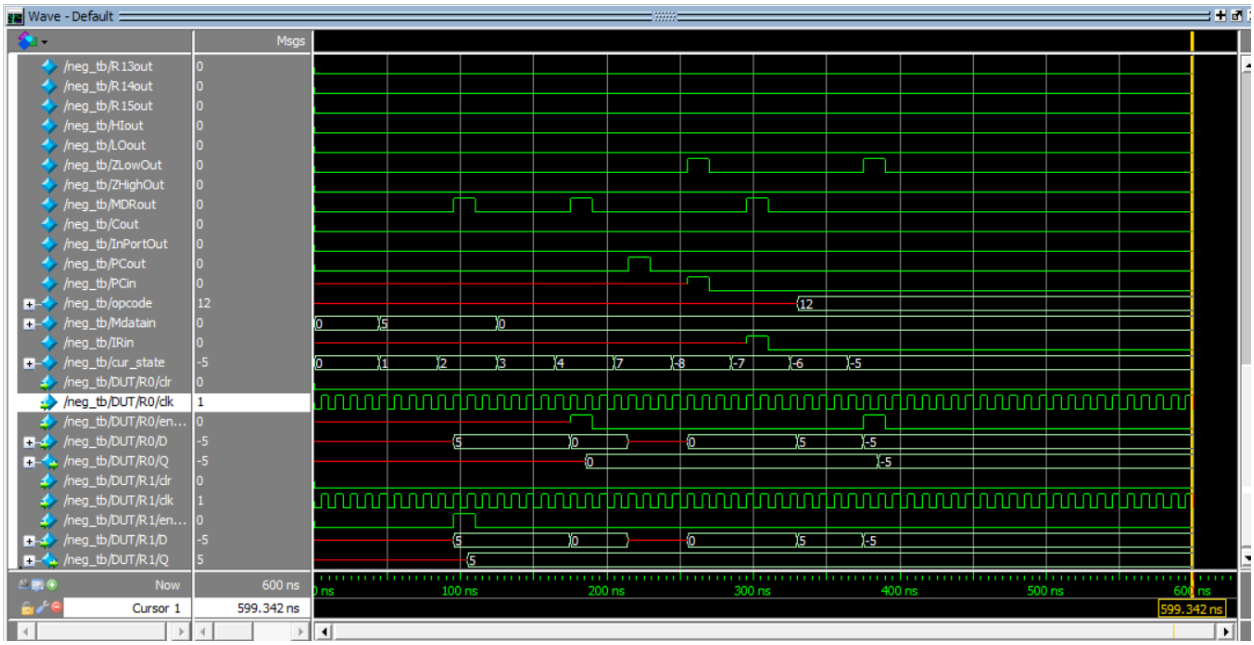
Rotate Left Testbench

The rotate left Testbench was the same as the rotate right Testbench, however, the opcode was the only thing that was changed. The output that was placed in R6 in the last cycle was 128.



Negate Testbench

The negate operation only has cycles T0-4 as it only takes input from a singular register R1 and negates the value then places it in R0. 5 was placed in R1 and R0 had an output of -5.



Not Testbench

The Not Testbench was the same as the negate Testbench, however, the opcode was changed for the not function. The same value of 5 was placed in the R1 register and gave an output of -6 in the R0 register.