ELEC 374 – Phase 1 Report

Department of Electrical and Computer Engineering
Queen's University

Composed By:
Zeerak Asim (20237955)
Nicholas Seegobin (20246787)

Date of Submission:
13 March 2023

Table of Contents

General Register

register.v

```
/* Representation of a register in Verilog HDL. */
/* Declared 3, 1-bit signals; 1, 32-bit D-input; 1, 32-bit Q-output. */
/* Data type of each signal/input is a wire and output is a reg. */
module register(input wire clk, input wire clr, input wire enable, input wire [31:0] D,
                output reg [31:0] Q);

    /* While loop that iterates every positive clock edge. */
    always @(posedge clk)
    begin
        /* If clear signal is high, set Q output to 0. */
        if(clr)
            Q <= 0;

        /* If enable signal is high, set Q output to follow (or equal to) D input. */
        else if(enable)
            Q = D;
    end
endmodule // Register end.
```

Memory Data Register

md_register.v

```
/* Representation of a memory data register with 2 to 1 multiplexer in Verilog */
module md_register(input clk, input clr, input enable, input select, input [31:0] D1, input
[31:0] D2, output reg [31:0] Q);

    /* While loop that iterates every positive clock edge. */
    always @(posedge clk)
    begin

        /* If clear signal is high, set Q output to 0. */
        if(clr)
            Q <= 0;

        /* If enable signal is high, set Q output to follow (or equal to) D input. */
        else if(enable)
            Q <= select ? D1 : D2;
    end
endmodule // md_register end.
```

Arithmetic Logic Unit

alu.v

```
module alu (
        // ALU Inputs
        input wire [31:0] A, B,
        input wire [4:0] opcode,

        // ALU Output
        output reg [63:0] result);

   parameter and_op = 5'b00101, or_op = 5'b00110, addition_op = 5'b00011,
   subtraction_op = 5'b00100, multiplication_op = 5'b01111, division_op = 5'b10000,
   left_shift_op = 5'b01001, right_shift_op = 5'b00111, arith_shift_right_op = 5'b01000,
   left_rotate_op = 5'b01011, right_rotate_op = 5'b01010, negate_op = 5'b10001,
   not_op = 5'b10010;

   // And Operation
   wire [31:0] and_result;
   logical_and andInstance(.A(A), .B(B), .result(and_result));

   // Or Operation
   wire [31:0] or_result;
   logical_or orInstance(A, B, or_result);

   // Add Operation
   wire [31:0] sum_result;
   wire carryOut;
   adder adderInstance(A, B, sum_result, carryOut);

   // Subtractor Operation
   wire [31:0] difference_result;
   wire borrowOut;
   subtractor subtractorInstance(A, B, difference_result, borrowOut);

   // Multiplication Operation
   wire [63:0] product_result;
   multiplier multiplierInstance(A, B, product_result);

   // Division Operation
   wire [31:0] quotient_result, remainder_result;
   divider dividerInstance(A, B, quotient_result, remainder_result);

   // Left Shift Operation
   wire [31:0] leftShift_result;
   shift_left leftShiftInstance(A, B, leftShift_result);

   // Right Shift Operation
   wire [31:0] rightShift_result;
   shift_right rightShiftInstance(A, B, rightShift_result);

   // Arithmetic Right Shift Operation
   wire [31:0] arithShiftRight_result;
   arithmetic_shift_right arithShiftRightInstance(A, B, arithShiftRight_result);
```

```
// Left Rotate Operation
wire [31:0] leftRotate_result;
rotate_left leftRotateInstance(A, B, leftRotate_result);

// Right Rotate Operation
wire [31:0] rightRotate_result;
rotate_right rightRotateInstance(A, B, rightRotate_result);

// Negate Operation
wire [31:0] negate_result;
negate negateInstance(B, negate_result);

// Not Operation
wire [31:0] not_result;
logical_not notInstance(B, not_result);

 // Select operation
 always @(*)
   begin
    case(opcode)
       // And Operation
       and_op : begin
          result[31:0] <= and_result[31:0];
          result[63:32] <= 32'd0;
       end

       // Or Operation
       or_op  : begin
          result[31:0] <= or_result[31:0];
          result[63:32] <= 32'd0;
       end

       // Add Operation
       addition_op    : begin
          result[31:0] <= sum_result[31:0];
          result[63:32] <= 32'd0;
       end

       // Subtract Operation
       subtraction_op : begin
          result[31:0] <= difference_result[31:0];
          result[63:32] <= 32'd0;
       end

       // Multiply Operation
       multiplication_op      : begin
          result[63:0] <= product_result[63:0];
       end

       // Divide Operation
       division_op    : begin
          result[31:0] <= quotient_result[31:0];
          result[63:32] <= remainder_result[31:0];
       end
```

```
        // Left Shift Operation
        left_shift_op : begin
           result[31:0] <= leftShift_result[31:0];
           result[63:32] <= 32'd0;
        end

        // Right Shift Operation
        right_shift_op : begin
           result[31:0] <= rightShift_result[31:0];
           result[63:32] <= 32'd0;
        end

        // Arithmetic Right Shift Operation
        arith_shift_right_op   : begin
           result[31:0] <= arithShiftRight_result[31:0];
           result[63:32] <= 32'd0;
        end

        // Left Rotate Operation
        left_rotate_op : begin
           result[31:0] <= leftRotate_result[31:0];
           result[63:32] <= 32'd0;
        end

        // Right Rotate Operation
        right_rotate_op : begin
           result[31:0] <= rightRotate_result[31:0];
           result[63:32] <= 32'd0;
        end

        // Negate Operation
        negate_op : begin
           result[31:0] <= negate_result[31:0];
           result[63:32] <= 32'd0;
         end

        // Not Operation
        not_op : begin
           result[31:0] <= not_result[31:0];
           result[63:32] <= 32'd0;
        end

        // Default Case
        default : begin
           result[63:0] <= 64'd0;
        end
      endcase
    end
endmodule
```

AND Operation

logical_and.v

```v
/* Representation of a 32-bit AND operation in Verilog HDL. */
module logical_and (input wire [31:0] A, B, output wire [31:0] result);

        assign result = A & B;

endmodule // and end.
```

OR Operation

logical_or.v

```v
/* Representation of a 32-bit OR operation in Verilog HDL. */
module logical_or (input wire [31:0] A, B, output wire [31:0] result);

        assign result = A | B;

endmodule // logical_or end.
```

NOT Operation

logical_not.v

```v
/* Representation of a 32-bit NOT operation in Verilog HDL. */
/* Flips each bit of A and stores the result in 'result' */
module logical_not (input wire [31:0] A, output wire [31:0] result);

        assign result = ~A;

endmodule // logical_not end.
```

Negate Operation

negate.v

```v
/* Representation of a 32-bit Negate operation in Verilog HDL. */
/* Performs 'not' operation on 'A' and adds 1 which returns the two's complement of 'A' */
module negate (input wire [31:0] A, output wire [31:0] result);

        assign result = ~A + 1;

endmodule // negate end.
```

Addition Operation

adder.v

```v
/* Representation of a 32-bit Carry Look-Ahead Adder in Verilog HDL. */
module adder(
    // Adder Inputs
    input wire [31:0] A,
    input wire [31:0] B,

    // Adder Outputs
    output wire [31:0] sum,
    output wire carryOut);

// Generate (G) Stage
wire [31:0] G;
assign G = A & B;

// Propagate (P) Stage
wire [31:0] P;
assign P = A | B;

// Generate Propagate (GP) Stage
wire [31:0] Gp, Pp;
assign Gp = G & P;
assign Pp = G ^ P;

// Carry (C) Stage
wire [31:0] C;
wire [31:0] S;
assign C[0] = 0;
assign S[0] = A[0] ^ B[0] ^ C[0];

genvar i;
generate
  for (i = 1; i < 32; i = i + 1) begin : carry_lookahead_loop
    assign C[i] = Gp[i-1] | (Pp[i-1] & C[i-1]);
    assign S[i] = A[i] ^ B[i] ^ C[i];
  end
endgenerate

// Output Sum and Carry Out
assign sum = S;
assign carryOut = C[31];

endmodule // adder end.
```

Subtraction Operation

subtractor.v

```v
/* Representation of a Subtractor using the 32-bit Carry Look-Ahead Adder in Verilog HDL. */
module subtractor(
    // Subtractor Inputs
    input wire [31:0] A,
    input wire [31:0] B,

    // Subtractor Outputs
    output wire [31:0] difference,
    output wire borrowOut);

// Perform Two's Complement on 'B' and store it in 'negateB'
wire [31:0] negateB;
negate negateInstance(.A(B), .result(negateB));

// Carry Look-Ahead Adder Outputs
wire [31:0] sum;
wire carryOut;

// Use the Carry Look-Ahead Adder to compute A - B
adder adderInstance(.A(A), .B(negateB), .sum(sum), .carryOut(carryOut));

// Output 'difference' and 'borrowOut'
assign difference = sum;
assign borrowOut = ~carryOut;

endmodule // subtractor end.
```

Multiplication Operation

multiplier.v

```
/* Representation of a 32-bit Multiplier operation in Verilog HDL. */
module multiplier (
        // Multiplier Inputs
        input signed [31:0] A, B,

        // Multiplier Output
        output wire [63:0] multiplier_result);

        // Declare array of 16 elements, where each element is a 3-bit register
        // Used to store the control bits for each stage of the Booth's Algorithm
        reg [2:0] controlBits [15:0];

        // Declare an array of 16 elements, where each element is a 33-bit register
        // Used to store the partial products computed in each stage of the Booth's Algorithm
        reg [32:0] partialProducts [15:0];

        // Declare array of 16 elements, where each element is a 64-bit register
        // Used to store the signed version of the partial products computed in each stage of
        // the Booth's Algorithm
        reg [63:0] signedPartialProduct [15:0];

        // Stores the combined 'signedPartialProduct' and stores it in 'product'
        reg [63:0] product;

        // Stores the Two's Complement of the Multiplicand A
        wire [32:0] negateA;

        // Used as loop counters in the 'always' block
        integer i, k;

        // Negate Operation used to take the Two's Complement of A
        negate negateInstance(A, negateA);

        always @ (A or B or negateA)
        begin
          // Initializes control bits for least significant bits of multiplier
          controlBits[0] = {B[1],B[0],1'b0};

          // Generates the control bits needed to multiply two 32-bit numbers
          for(k = 1; k < 16; k = k + 1)
              controlBits[k] = {B[2*k+1], B[2*k], B[2*k-1]};
```

```
          // Generates appropriate partial product for each group of three bits based on
          // control signal
          for(k = 0; k < 16; k = k + 1)
           begin
            case(controlBits[k])
              // Generates a partial product that corresponds to a positive multiplicand
              // shifted left by one bit
              3'b001 , 3'b010 : partialProducts[k] = {A[31],A};

              // Generates a partial product that corresponds to a positive multiplicand
              // shifted left by two bits
              3'b011 : partialProducts[k] = {A,1'b0};

              // Generates a partial product that corresponds to a negative multiplicand
              // shifted left by one bit
              3'b100 : partialProducts[k] = {negateA[31:0],1'b0};

              // Generates a partial product that corresponds to a negative multiplicand (32-
              // bit negated multiplicand 'negateA')
              3'b101 , 3'b110 : partialProducts[k] = negateA;

              // Generates a partial product of zero
              default : partialProducts[k] = 0;
            endcase

            // Converts the 33-bit 'partialProducts[k]' signal into a signed 33-bit signal
            // stored in 'signedPartialProduct[k]'
            signedPartialProduct[k] = $signed(partialProducts[k]);

            // Performs a left shift operation on the 'signedPartialProduct[k]' signal by
            // multiplying it by 2^2 * k
            // Creates a series of shifted partial products so they are aligned for addition
            for(i = 0; i < k; i = i + 1)
                signedPartialProduct[k] = {signedPartialProduct[k], 2'b00};
         end

         // Initialize the 'product' variable with first bit of 'signedPartialProduct'
         product = signedPartialProduct[0];

         // Accumulates all the shifted partial products together
         for(k = 1; k < 16; k = k + 1)
             product = product + signedPartialProduct[k];
        end

        // Return the result
        assign multiplier_result = product;

endmodule // multiplier end.
```

Division Operation

divider.v

```v
/*
  This module performs a signed integer division operation.
  It takes a 32-bit signed integer dividend and divisor as input,
  and provides the quotient and remainder as output.
*/
module divider(
    input signed [31:0] dividend, // 32-bit signed dividend
    input signed [31:0] divisor,  // 32-bit signed divisor
    output wire [31:0] quotient,  // 32-bit signed quotient
    output wire [31:0] remainder  // 32-bit signed remainder
    );

  reg [32:0] A, M;  // Registers for storing the values of A and M
  wire [31:0] twosComp; // 2's complement of the divisor
  integer k,i; // Integer variables used in the loop
  reg signed [64:0] divide; // Register for storing the intermediate result of division

  // Calculate 2's complement of the divisor
  assign twosComp = ~divisor + 1;

  // Initialize A to zero
  initial A = 32'h000000000;

  // Combinational logic for division
  always @(*) begin
    if(divisor[31] == 1) begin
      // If the divisor is negative, use its 2's complement
      M = twosComp;
      k = 1;
    end
    else begin
      M = divisor;
      k = 0;
    end
    // Combine A and dividend to form a 64-bit dividend
    divide = {A, dividend};

    // Perform 32 iterations of the division algorithm
    for(i=0;i<32;i=i+1) begin
      // Left shift the dividend and quotient
      divide = divide << 1;

      if(divide[64] == 0) begin
        // Subtract M from the dividend if it's greater than or equal to M
        divide[64:32] = divide[64:32] - M;
        // Set the quotient bit to 1
        divide[0] = 1;
      end
```

```
      else if(divide[64] == 1) begin
        // Add M to the dividend if it's less than M
        divide[64:32] = divide[64:32] + M;
        // Set the quotient bit to 0
        divide[0] = 0;
      end
      // Check for overflow/underflow and adjust the dividend and quotient
      if(divide[64] == 0) divide[0] = 1;
      else if(divide[64] == 1) divide[0] = 0;
    end

    // Adjust the remainder and quotient if necessary
    // Add M to the dividend to obtain the correct remainder
    if(divide[64] == 1) divide[64:32] = divide[64:32] + M;

    // Negate the quotient if the divisor is negative
    if(k == 1) divide[64:32] = ~divide[64:32] + 1;
  end

  // Assign the quotient and remainder to the output ports
  assign quotient = divide[31:0];
  assign remainder = divide[63:32];

endmodule // divider end.
```

Left Shift Operation

shift_left.v

```
/* Representation of a 32-bit Left Bit Shift operation in Verilog HDL. */
/* 'A' is the data to be shifted, and 'B' specifies the number of bits to shift */
/* The double left-shift operator '<<' is a logical shift that fills in the vacated bits with
zeroes */
module shift_left(
    // Left Shift Inputs
    input wire [31:0] A, B,

    // Left Shift Output
    output wire [31:0] result);

    // Shifts 'A' Left by 'B' bits
    assign result = A << B;

endmodule // shift_left end.
```

Right Shift Operation

shift_right.v

```v
/* Representation of a 32-bit Right Bit Shift operation in Verilog HDL. */
/* 'A' is the data to be shifted, and 'B' specifies the number of bits to shift */
/* The double right-shift operator '>>' is a logical shift that fills in the vacated bits with zeroes */
module shift_right(
    // Right Shift Inputs
    input wire [31:0] A, B,

    // Right Shift Output
    output wire [31:0] result);

    // Shifts 'A' Right by 'B' bits
    assign result = A >> B;

endmodule // shift_right end.
```

Arithmetic Right Shift Operation

arithmetic_shift_right.v

```v
/* Representation of a 32-bit Arithmetic Shift Right operation in Verilog HDL. */
/* Shifts the bits of input signal 'A' to the right by the number of bits specified by input signal 'B' */
/* Operator used is >>> instead of >> which preserves the sign of the input signal */
module arithmetic_shift_right (
    // Arithemetic Shift Right Inputs
    input wire [31:0] A, B,

    // Arithemetic Shift Right Output
    output wire [31:0] result);

    // Shifted result assigned to output signal 'result'
    assign result = $signed(A) >>> B;

endmodule // arithmetic_shift_right end.
```

Left Rotate Operation

left_rotate

```
/* Representation of a 32-bit Left Bit Rotatation operation in Verilog HDL. */
/* 'A' is the data to be rotated, and 'B' specifies the number of bits to rotate */
module rotate_left(
    // Left Rotate Inputs
    input wire [31:0] A, B,

    // Left Rotate Output
    output wire [31:0] result);

    // Shift Instance Outputs
    wire [31:0] shiftOneResult, shiftTwoResult;

    // Shift Instances
    // Shifts left 'B' times
    shift_left leftShiftOne(A, B, shiftOneResult);

    // Shifts left (32 - B) times
    shift_right rightShiftTwo(A, (32 - B), shiftTwoResult);

    // OR Operation Instance
    // Performs the OR operation and produces the final result
    logical_or orInstance(shiftOneResult, shiftTwoResult, result);
endmodule // rotate_left end.
```

Right Rotate Operation

right_rotate

```
/* Representation of a 32-bit Right Bit Rotatation operation in Verilog HDL. */
/* 'A' is the data to be rotated, and 'B' specifies the number of bits to rotate */
module rotate_right (
    // Right Rotate Inputs
    input wire [31:0] A, B,

    // Right Rotate Output
    output wire [31:0] result);

    // Shift Instance Outputs
    wire [31:0] shiftOneResult, shiftTwoResult;

    // Shift Instances
    // Shifts right (32 - B) times
    shift_left leftShift(A, (32 - B), shiftOneResult);

    // Shifts right B times
    shift_right rightShift(A, B, shiftTwoResult);

    // OR Operation Instance
    // Performs the OR operation and produces the final result
    logical_or orInstance(shiftOneResult, shiftTwoResult, result);

endmodule // rotate_right end.
```

Bus System

Datapath

datapath.v

```
/* Representation of the datapath in Verilog HDL. */
/* Connected outputs of encoder to select signals in multiplexer. */
module datapath(
        // CPU signals
        input wire clk,
        input wire clr,

        // Register write/enable signals
        input wire r0_enable, r1_enable, r2_enable, r3_enable,
        input wire r4_enable, r5_enable, r6_enable, r7_enable,
        input wire r8_enable, r9_enable, r10_enable, r11_enable,
        input wire r12_enable, r13_enable, r14_enable, r15_enable,
        input wire PC_enable, PC_increment_enable, IR_enable,
        input wire Y_enable, Z_enable,
        input wire MAR_enable, MDR_enable,
        input wire HI_enable, LO_enable,

        // Memory Data Multiplexer Read/Select Signal
        input wire read,

        // Encoder Output Select Signals
        input wire r0_select, r1_select, r2_select, r3_select,
        input wire r4_select, r5_select, r6_select, r7_select,
        input wire r8_select, r9_select, r10_select, r11_select,
        input wire r12_select, r13_select, r14_select, r15_select,
        input wire PC_select,
        input wire HI_select, LO_select,
        input wire Z_HI_select, Z_LO_select,
        input wire MDR_select,
        input wire InPort_select,
        input wire c_select,
        output wire [4:0] encode_sel_signal,

        // ALU Opcode
        input wire [4:0] alu_instruction,

        // Input Data Signals
        input wire [31:0] MDataIN,

        // Output Data Signals
        output wire [31:0] bus_Data, // Data currently in the bus
        output wire [63:0] aluResult,

        output wire [31:0] R0_Data, R1_Data, R2_Data, R3_Data,
        output wire [31:0] R4_Data, R5_Data, R6_Data, R7_Data,
        output wire [31:0] R8_Data, R9_Data, R10_Data, R11_Data,
        output wire [31:0] R12_Data, R13_Data, R14_Data, R15_Data,
```

```
output wire [31:0] PC_Data, IR_Data,
output wire [31:0] Y_Data,
output wire [31:0] Z_HI_Data, Z_LO_Data,
output wire [31:0] MAR_Data, MDR_Data,
output wire [31:0] HI_Data, LO_Data,
output wire [31:0] InPort_Data,
output wire [31:0] C_sign_ext_Data);

// General purpose registers r0-r15
register r0 (.clk(clk), .clr(clr), .enable(r0_enable), .D(bus_Data), .Q(R0_Data));
register r1 (.clk(clk), .clr(clr), .enable(r1_enable), .D(bus_Data), .Q(R1_Data));
register r2 (.clk(clk), .clr(clr), .enable(r2_enable), .D(bus_Data), .Q(R2_Data));
register r3 (.clk(clk), .clr(clr), .enable(r3_enable), .D(bus_Data), .Q(R3_Data));
register r4 (.clk(clk), .clr(clr), .enable(r4_enable), .D(bus_Data), .Q(R4_Data));
register r5 (.clk(clk), .clr(clr), .enable(r5_enable), .D(bus_Data), .Q(R5_Data));
register r6 (.clk(clk), .clr(clr), .enable(r6_enable), .D(bus_Data), .Q(R6_Data));
register r7 (.clk(clk), .clr(clr), .enable(r7_enable), .D(bus_Data), .Q(R7_Data));
register r8 (.clk(clk), .clr(clr), .enable(r8_enable), .D(bus_Data), .Q(R8_Data));
register r9 (.clk(clk), .clr(clr), .enable(r9_enable), .D(bus_Data), .Q(R9_Data));
register r10 (.clk(clk), .clr(clr), .enable(r10_enable), .D(bus_Data), .Q(R10_Data));
register r11 (.clk(clk), .clr(clr), .enable(r11_enable), .D(bus_Data), .Q(R11_Data));
register r12 (.clk(clk), .clr(clr), .enable(r12_enable), .D(bus_Data), .Q(R12_Data));
register r13 (.clk(clk), .clr(clr), .enable(r13_enable), .D(bus_Data), .Q(R13_Data));
register r14 (.clk(clk), .clr(clr), .enable(r14_enable), .D(bus_Data), .Q(R14_Data));
register r15 (.clk(clk), .clr(clr), .enable(r15_enable), .D(bus_Data), .Q(R15_Data));

// C Output Registers
register HI (.clk(clk), .clr(clr), .enable(HI_enable), .D(bus_Data), .Q(HI_Data));
register LO (.clk(clk), .clr(clr), .enable(LO_enable), .D(bus_Data), .Q(LO_Data));
register Z_HI (.clk(clk), .clr(clr), .enable(Z_enable), .D(aluResult[63:32]),
               .Q(Z_HI_Data));
register Z_LO (.clk(clk), .clr(clr), .enable(Z_enable), .D(aluResult[31:0]),
               .Q(Z_LO_Data));
register Y (.clk(clk), .clr(clr), .enable(Y_enable), .D(bus_Data), .Q(Y_Data));

// PC and IR Registers
program_counter PC (.clk(clk), .clr(clr), .enable(PC_enable),
                    .incPC(PC_increment_enable),
                    .PC_Input(bus_Data),.PC_Output(PC_Data));

register IR (.clk(clk), .clr(clr), .enable(IR_enable), .D(bus_Data), .Q(IR_Data));

register MAR (.clk(clk), .clr(clr), .enable(MAR_enable), .D(bus_Data), .Q(MAR_Data));
md_register MDR (.clk(clk), .clr(clr), .enable(MDR_enable), .select(read),
                 .D1(MDataIN), .D2(bus_Data), .Q(MDR_Data));

// Encoder Instance
encoder encoder_instance(
.encodeIN_r0(r0_select), .encodeIN_r1(r1_select), .encodeIN_r2(r2_select),
.encodeIN_r3(r3_select), .encodeIN_r4(r4_select), .encodeIN_r5(r5_select),
.encodeIN_r6(r6_select), .encodeIN_r7(r7_select), .encodeIN_r8(r8_select),
.encodeIN_r9(r9_select), .encodeIN_r10(r10_select), .encodeIN_r11(r11_select),
.encodeIN_r12(r12_select), .encodeIN_r13(r13_select), .encodeIN_r14(r14_select),
.encodeIN_r15(r15_select), .encodeIN_HI(HI_select), .encodeIN_LO(LO_select),
.encodeIN_Z_HI(Z_HI_select), .encodeIN_Z_LO(Z_LO_select), .encodeIN_PC(PC_select),
.encodeIN_MDR(MDR_select), .encodeIN_InPort(InPort_select), .encodeIN_Cout(c_select),
.select_signals_OUT(encode_sel_signal));
```

```
        // Output from the register is the input to the MUX
        multiplexer multiplexer_instance(
        .select_signals_IN(encode_sel_signal), .muxIN_r0(R0_Data),
        .muxIN_r1(R1_Data), .muxIN_r2(R2_Data), .muxIN_r3(R3_Data), .muxIN_r4(R4_Data),
        .muxIN_r5(R5_Data), .muxIN_r6(R6_Data), .muxIN_r7(R7_Data), .muxIN_r8(R8_Data),
        .muxIN_r9(R9_Data), .muxIN_r10(R10_Data), .muxIN_r11(R11_Data), .muxIN_r12(R12_Data),
        .muxIN_r13(R13_Data), .muxIN_r14(R14_Data), .muxIN_r15(R15_Data), .muxIN_HI(HI_Data),
        .muxIN_LO(LO_Data), .muxIN_Z_HI(Z_HI_Data), .muxIN_Z_LO(Z_LO_Data),
        .muxIN_PC(PC_Data), .muxIN_MDR(MDR_Data), .muxIN_InPort(InPort_Data),
        .muxIN_C_sign_ext(C_sign_ext_Data), .muxOut(bus_Data));

       alu alu_instance(.A(Y_Data), .B(bus_Data), .opcode(alu_instruction),
                        .result(aluResult));

endmodule // datapath end.
```

32:1 Multiplexer

multiplexer.v

```
/* Representation of a multiplexer in Verilog HDL. */
/* Declared 32, 32-bit inputs, 5 select signals, and  1, 32-bit output. */
/* Data type of each input/select signal is a wire and output is a reg. */

module multiplexer(input [4:0] select_signals_IN, input [31:0] muxIN_r0,
input [31:0] muxIN_r1, input [31:0] muxIN_r2, input [31:0] muxIN_r3,
input [31:0] muxIN_r4, input [31:0] muxIN_r5, input [31:0] muxIN_r6,
input [31:0] muxIN_r7, input [31:0] muxIN_r8, input [31:0] muxIN_r9,
input [31:0] muxIN_r10, input [31:0] muxIN_r11, input [31:0] muxIN_r12,
input [31:0] muxIN_r13, input [31:0] muxIN_r14, input [31:0] muxIN_r15,
input [31:0] muxIN_HI, input [31:0] muxIN_LO, input [31:0] muxIN_Z_HI,
input [31:0] muxIN_Z_LO, input [31:0] muxIN_PC, input [31:0] muxIN_MDR,
input [31:0] muxIN_InPort, input [31:0] muxIN_C_sign_ext, output reg [31:0] muxOut);

    /* While loop to check for updates in the select signals, which updates the mux output. */
    always @(*) begin
            case (select_signals_IN)
                5'b00000: muxOut <= muxIN_r0[31:0];
                5'b00001: muxOut <= muxIN_r1[31:0];
                5'b00010: muxOut <= muxIN_r2[31:0];
                5'b00011: muxOut <= muxIN_r3[31:0];
                5'b00100: muxOut <= muxIN_r4[31:0];
                5'b00101: muxOut <= muxIN_r5[31:0];
                5'b00110: muxOut <= muxIN_r6[31:0];
                5'b00111: muxOut <= muxIN_r7[31:0];
                5'b01000: muxOut <= muxIN_r8[31:0];
                5'b01001: muxOut <= muxIN_r9[31:0];
                5'b01010: muxOut <= muxIN_r10[31:0];
                5'b01011: muxOut <= muxIN_r11[31:0];
                5'b01100: muxOut <= muxIN_r12[31:0];
                5'b01101: muxOut <= muxIN_r13[31:0];
                5'b01110: muxOut <= muxIN_r14[31:0];
                5'b01111: muxOut <= muxIN_r15[31:0];
                5'b10000: muxOut <= muxIN_HI[31:0];
                5'b10001: muxOut <= muxIN_LO[31:0];
                5'b10010: muxOut <= muxIN_Z_HI[31:0];
                5'b10011: muxOut <= muxIN_Z_LO[31:0];
                5'b10100: muxOut <= muxIN_PC[31:0];
                5'b10101: muxOut <= muxIN_MDR[31:0];
                5'b10110: muxOut <= muxIN_InPort[31:0];
                5'b10111: muxOut <= muxIN_C_sign_ext[31:0];
                default: muxOut <= 32'd0;
            endcase
        end
endmodule // Multiplexer end.
```

32:5 Encoder

encoder.v

```
/* Representation of an encoder in Verilog HDL. */
/* Declared 32, 1-bit inputs and 5 output select signals. */
/* Data type of each input is a wire. */
module encoder(input encodeIN_r0, input encodeIN_r1, input encodeIN_r2,
input encodeIN_r3, input encodeIN_r4, input encodeIN_r5, input encodeIN_r6,
input encodeIN_r7, input encodeIN_r8, input encodeIN_r9, input encodeIN_r10,
input encodeIN_r11, input encodeIN_r12, input encodeIN_r13, input encodeIN_r14,
input encodeIN_r15, input encodeIN_HI, input encodeIN_LO, input encodeIN_Z_HI,
input encodeIN_Z_LO, input encodeIN_PC, input encodeIN_MDR, input encodeIN_InPort,
input encodeIN_Cout, output reg [4:0] select_signals_OUT);

    /* While loop to update the select_signals_OUT output wire. */
    always @* begin
        if (encodeIN_Cout) select_signals_OUT <= 5'b10111;
        else if (encodeIN_InPort) select_signals_OUT <= 5'b10110;
        else if (encodeIN_MDR) select_signals_OUT <= 5'b10101;
        else if (encodeIN_PC) select_signals_OUT <= 5'b10100;
        else if (encodeIN_Z_LO) select_signals_OUT <= 5'b10011;
        else if (encodeIN_Z_HI) select_signals_OUT <= 5'b10010;
        else if (encodeIN_LO) select_signals_OUT <= 5'b10001;
        else if (encodeIN_HI) select_signals_OUT <= 5'b10000;
        else if (encodeIN_r15) select_signals_OUT <= 5'b01111;
        else if (encodeIN_r14) select_signals_OUT <= 5'b01110;
        else if (encodeIN_r13) select_signals_OUT <= 5'b01101;
        else if (encodeIN_r12) select_signals_OUT <= 5'b01100;
        else if (encodeIN_r11) select_signals_OUT <= 5'b01011;
        else if (encodeIN_r10) select_signals_OUT <= 5'b01010;
        else if (encodeIN_r9) select_signals_OUT <= 5'b01001;
        else if (encodeIN_r8) select_signals_OUT <= 5'b01000;
        else if (encodeIN_r7) select_signals_OUT <= 5'b00111;
        else if (encodeIN_r6) select_signals_OUT <= 5'b00110;
        else if (encodeIN_r5) select_signals_OUT <= 5'b00101;
        else if (encodeIN_r4) select_signals_OUT <= 5'b00100;
        else if (encodeIN_r3) select_signals_OUT <= 5'b00011;
        else if (encodeIN_r2) select_signals_OUT <= 5'b00010;
        else if (encodeIN_r1) select_signals_OUT <= 5'b00001;
        else if (encodeIN_r0) select_signals_OUT <= 5'b00000;
        else select_signals_OUT <= 5'b00000;  // optional, to avoid latch.
    end
endmodule // Encoder end.
```

PC Incrementor Unit

program_counter.v

```
/* Representation of a PC Incrementor in Verilog HDL. */
module program_counter(
    input wire clk, clr, enable, incPC,
    input wire [31:0] PC_Input,
    output wire [31:0] PC_Output
);

reg [31:0] Q;

initial Q <= 0;

always @ (posedge clk)
                begin
                        if (clr)
                                Q <= 0;
                        else if (enable)
                                Q <= PC_Input;
                        else if (incPC)
                                Q <= Q + 1;
                end

assign PC_Output = Q;

endmodule
```

Datapath Testbench Files

AND/OR Testbench

The **AND** operation and the **OR** operation have the same testbench. The main difference between the two is that the opcode in the AND operation is (32'h28918000 or 5'b00101) and the opcode in the OR operation is (32'h30918000 or 5'b00110).

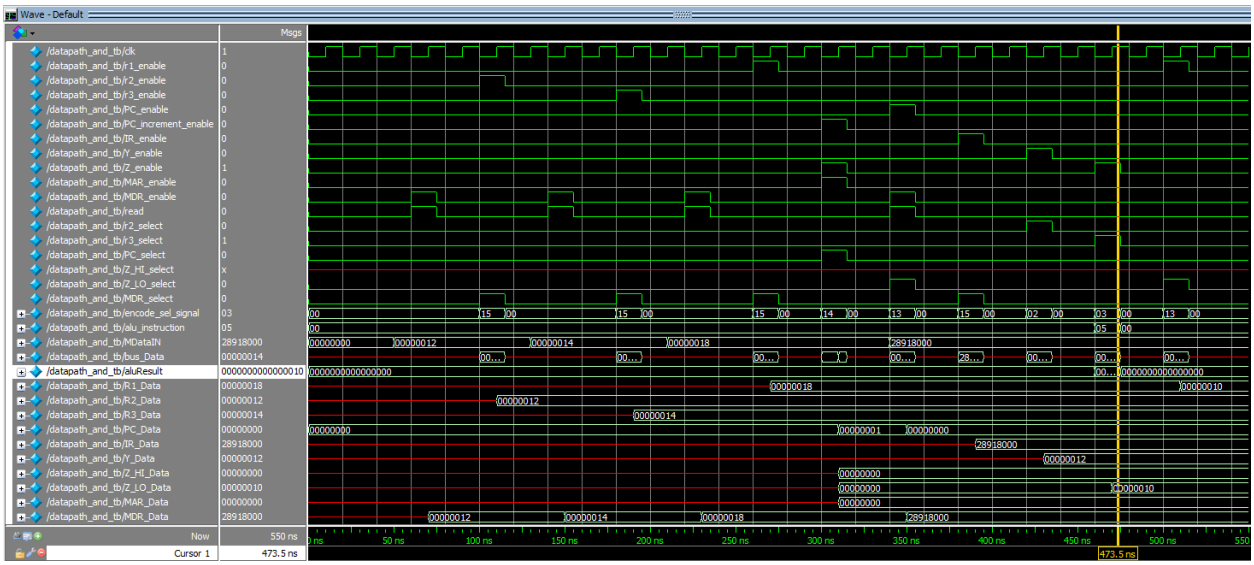
AND Operation Testbench Code

```
`timescale 1ns/10ps
module datapath_and_tb;
    // CPU signals
    reg clk;

    // Register write/enable signals
    reg r1_enable, r2_enable, r3_enable;
    reg PC_enable, PC_increment_enable, IR_enable;
    reg Y_enable, Z_enable;
    reg MAR_enable, MDR_enable;

    // Memory Data Multiplexer Read/Select Signal
    reg read;

    // Encoder Output Select Signals
    reg r2_select, r3_select;
    reg PC_select;
    reg Z_HI_select;
    reg Z_LO_select;
    reg MDR_select;

    wire [4:0] encode_sel_signal;

    // ALU Opcode
    reg [4:0] alu_instruction;

    // Input Data Signals
    reg [31:0] MDataIN;

    // Output Data Signals
    wire [31:0] bus_Data; // Data currently in the bus
    wire [63:0] aluResult;

    wire [31:0] R1_Data, R2_Data, R3_Data;

    wire [31:0] PC_Data, IR_Data;
    wire [31:0] Y_Data;
    wire [31:0] Z_HI_Data, Z_LO_Data;
    wire [31:0] MAR_Data, MDR_Data;

    // Time Signals and Load Registers
    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 datapathAND(   // CPU signals
    .clk(clk),

    // Register write/enable signals
    .r1_enable(r1_enable), .r2_enable(r2_enable), .r3_enable(r3_enable),
    .PC_enable(PC_enable), .PC_increment_enable(PC_increment_enable), .IR_enable(IR_enable),
    .Y_enable(Y_enable), .Z_enable(Z_enable),
    .MAR_enable(MAR_enable), .MDR_enable(MDR_enable),

    // Memory Data Multiplexer Read/Select Signal
    .read(read),
```

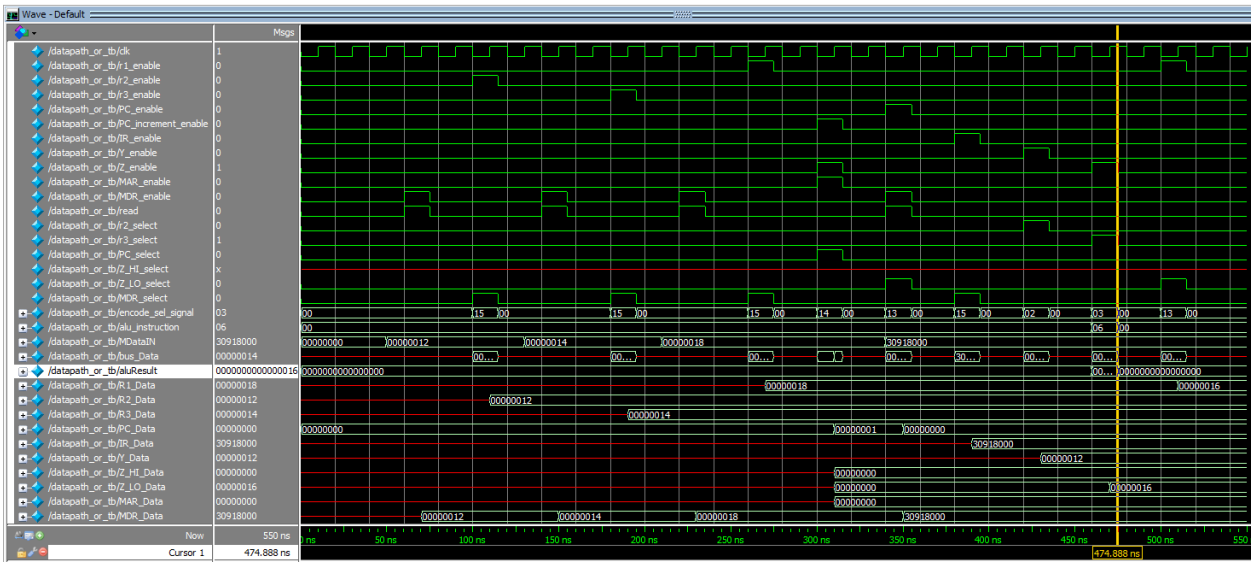
```
// Encoder Output Select Signals
.r2_select(r2_select), .r3_select(r3_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.encode_sel_signal(encode_sel_signal),

// ALU Opcode
.alu_instruction(alu_instruction),

// Input Data Signals
.MDataIN(MDataIN),

// Output Data Signals
.bus_Data(bus_Data), // Data currently in the bus
.aluResult(aluResult),

.R1_Data(R1_Data), .R2_Data(R2_Data), .R3_Data(R3_Data),

.PC_Data(PC_Data), .IR_Data(IR_Data),
.Y_Data(Y_Data),
.Z_HI_Data(Z_HI_Data), .Z_LO_Data(Z_LO_Data),
.MAR_Data(MAR_Data), .MDR_Data(MDR_Data));

// add test logic here
always #10 clk = ~clk;

initial begin
    clk = 0;
end

always @(posedge clk) // finite state machine; if clk 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 in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                PC_select <= 0; Z_LO_select <= 0; MDR_select <= 0; // initialize the signals
                r2_select <= 0; r3_select <= 0; MAR_enable <= 0; Z_enable <= 0;
                PC_enable <=0; MDR_enable <= 0; IR_enable <= 0; Y_enable <= 0;
                PC_increment_enable <= 0; read <= 0; alu_instruction <= 0;
                r1_enable <= 0; r2_enable <= 0; r3_enable <= 0; MDataIN <= 32'h00000000;
            end

            Reg_load1a: begin
                MDataIN <= 32'h00000012;
                read = 0; MDR_enable = 0; // the first zero is there for completeness
                #10 read <= 1; MDR_enable <= 1;
                #15 read <= 0; MDR_enable <= 0;
            end
```

```
                Reg_load1b: begin
                    #10 MDR_select <= 1; r2_enable <= 1;
                    #15 MDR_select <= 0; r2_enable <= 0; // initialize R2 with the value $12
                end

                Reg_load2a: begin
                    MDataIN <= 32'h00000014;
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load2b: begin
                    #10 MDR_select <= 1; r3_enable <= 1;
                    #15 MDR_select <= 0; r3_enable <= 0; // initialize R3 with the value $14
                end

                Reg_load3a: begin
                    MDataIN <= 32'h00000018;
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load3b: begin
                    #10 MDR_select <= 1; r1_enable <= 1;
                    #15 MDR_select <= 0; r1_enable <= 0; // initialize R1 with the value $18
                end

                T0: begin // see if you need to de-assert these signals
                    #10 PC_select <= 1; MAR_enable <= 1; PC_increment_enable <= 1; Z_enable <= 1;
                    #15 PC_select <= 0; MAR_enable <= 0; PC_increment_enable <= 0; Z_enable <= 0;
                end

                T1: begin
                    #10 Z_LO_select <= 1; PC_enable <= 1; read <= 1; MDR_enable <= 1; MDataIN <=
                        32'h28918000; // opcode for “AND R1, R2, R3"
                    #15 Z_LO_select <= 0; PC_enable <= 0; read <= 0; MDR_enable <= 0;
                end

                T2: begin
                    #10 MDR_select <= 1; IR_enable <= 1;
                    #15 MDR_select <= 0; IR_enable <= 0;
                end

                T3: begin
                    #10 r2_select <= 1; Y_enable <= 1;
                    #15 r2_select <= 0; Y_enable <= 0;
                end

                T4: begin
                    #10 r3_select <= 1; alu_instruction <= 5'b00101; Z_enable <= 1;
                    #15 r3_select <= 0; alu_instruction <= 0; Z_enable <= 0;
                end

                T5: begin
                    #10 Z_LO_select <= 1; r1_enable <= 1;
                    #15 Z_LO_select <= 0; r1_enable <= 0;
                end
            endcase
        end
endmodule
```

AND Operation RTL Simulation



OR Operation RTL Simulation



ADD/SUB Testbench

The **ADD** (addition) operation and **SUB** (subtraction) operation have the same testbench. The main difference between the two is that the opcode in the ADD operation is (32'h18228000 or 5'b00011) and the opcode in the SUB operation is (32'h20228000 or 5'b00100).

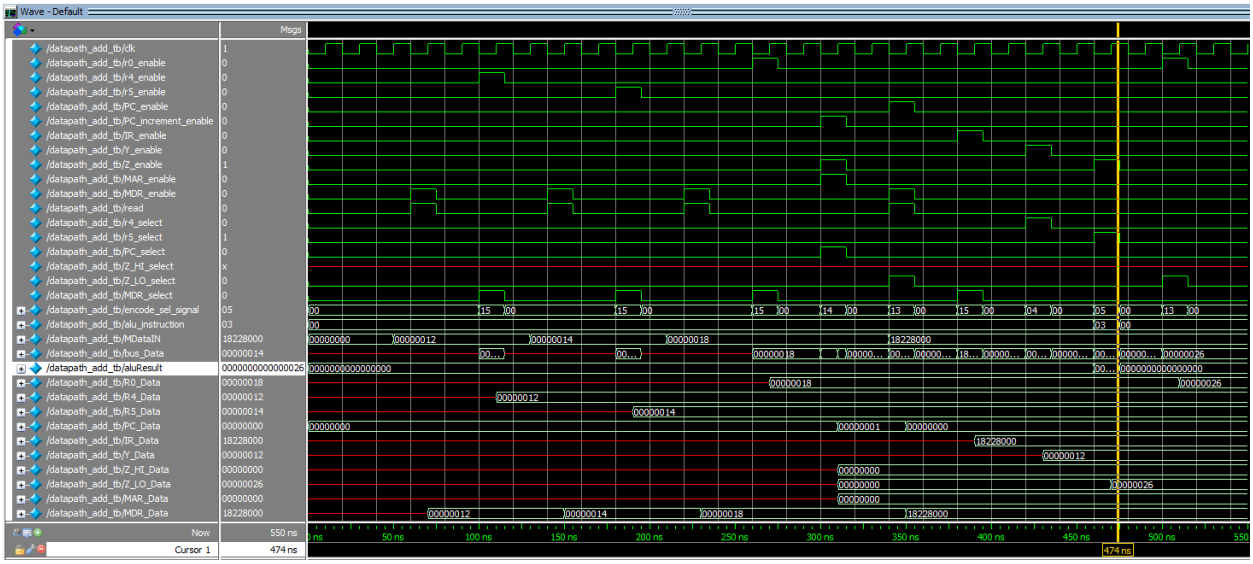
ADD Operation Testbench Code

```
`timescale 1ns/10ps
module datapath_add_tb;
    // CPU signals
    reg clk;

    // Register write/enable signals
    reg r0_enable, r4_enable, r5_enable;
    reg PC_enable, PC_increment_enable, IR_enable;
    reg Y_enable, Z_enable;
    reg MAR_enable, MDR_enable;

    // Memory Data Multiplexer Read/Select Signal
    reg read;

    // Encoder Output Select Signals
    reg r4_select, r5_select;
    reg PC_select;
    reg Z_HI_select;
    reg Z_LO_select;
    reg MDR_select;

    wire [4:0] encode_sel_signal;

    // ALU Opcode
    reg [4:0] alu_instruction;

    // Input Data Signals
    reg [31:0] MDataIN;

    // Output Data Signals
    wire [31:0] bus_Data; // Data currently in the bus
    wire [63:0] aluResult;

    wire [31:0] R0_Data, R4_Data, R5_Data;

    wire [31:0] PC_Data, IR_Data;
    wire [31:0] Y_Data;
    wire [31:0] Z_HI_Data, Z_LO_Data;
    wire [31:0] MAR_Data, MDR_Data;

    // Time Signals and Load Registers
    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 datapathAND(   // CPU signals
    .clk(clk),

    // Register write/enable signals
    .r0_enable(r0_enable), .r4_enable(r4_enable), .r5_enable(r5_enable),
    .PC_enable(PC_enable), .PC_increment_enable(PC_increment_enable), .IR_enable(IR_enable),
    .Y_enable(Y_enable), .Z_enable(Z_enable),
    .MAR_enable(MAR_enable), .MDR_enable(MDR_enable),

    // Memory Data Multiplexer Read/Select Signal
    .read(read),
```

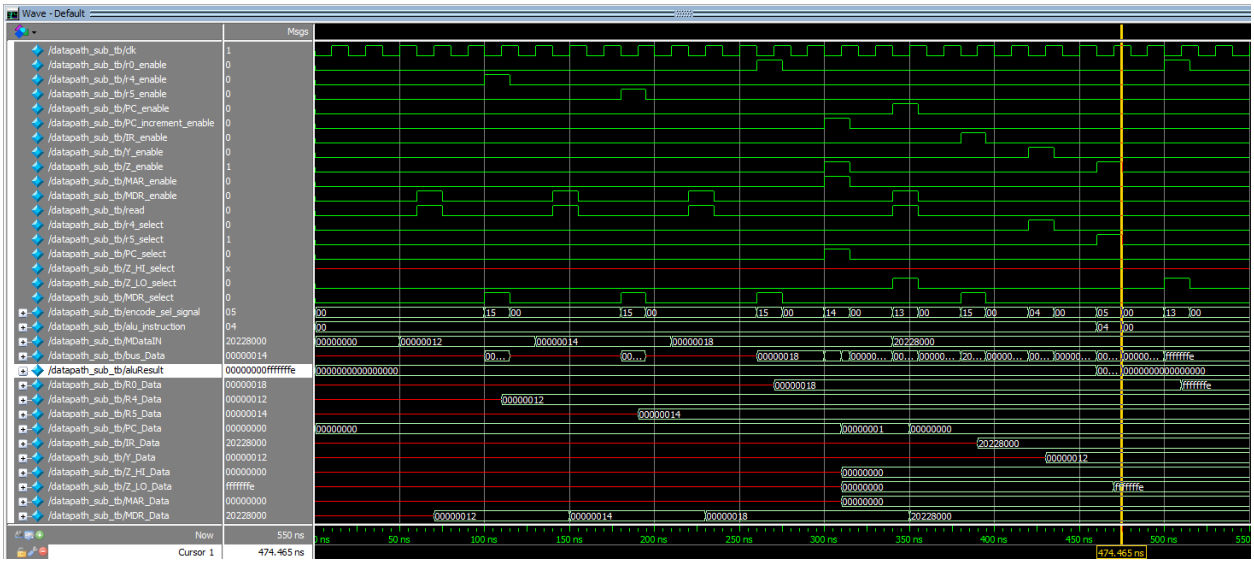
```
// Encoder Output Select Signals
.r4_select(r4_select), .r5_select(r5_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.encode_sel_signal(encode_sel_signal),

// ALU Opcode
.alu_instruction(alu_instruction),

// Input Data Signals
.MDataIN(MDataIN),

// Output Data Signals
.bus_Data(bus_Data), // Data currently in the bus
.aluResult(aluResult),

.R0_Data(R0_Data), .R4_Data(R4_Data), .R5_Data(R5_Data),

.PC_Data(PC_Data), .IR_Data(IR_Data),
.Y_Data(Y_Data),
.Z_HI_Data(Z_HI_Data), .Z_LO_Data(Z_LO_Data),
.MAR_Data(MAR_Data), .MDR_Data(MDR_Data));

// add test logic here
always #10 clk = ~clk;

initial begin
    clk = 0;
end

always @(posedge clk) // finite state machine; if clk 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 in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                PC_select <= 0; Z_LO_select <= 0; MDR_select <= 0; // initialize the signals
                r4_select <= 0; r5_select <= 0; MAR_enable <= 0; Z_enable <= 0;
                PC_enable <=0; MDR_enable <= 0; IR_enable <= 0; Y_enable <= 0;
                PC_increment_enable <= 0; read <= 0; alu_instruction <= 0;
                r0_enable <= 0; r4_enable <= 0; r5_enable <= 0; MDataIN <= 32'h00000000;
            end

            Reg_load1a: begin
                MDataIN <= 32'h00000012;
                read = 0; MDR_enable = 0; // the first zero is there for completeness
                #10 read <= 1; MDR_enable <= 1;
                #15 read <= 0; MDR_enable <= 0;
            end
```

```
                Reg_load1b: begin
                    #10 MDR_select <= 1; r4_enable <= 1;
                    #15 MDR_select <= 0; r4_enable <= 0; // initialize R2 with the value $12
                end

                Reg_load2a: begin
                    MDataIN <= 32'h00000014;
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load2b: begin
                    #10 MDR_select <= 1; r5_enable <= 1;
                    #15 MDR_select <= 0; r5_enable <= 0; // initialize R3 with the value $14
                end

                Reg_load3a: begin
                    MDataIN <= 32'h00000018;
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load3b: begin
                    #10 MDR_select <= 1; r0_enable <= 1;
                    #15 MDR_select <= 0; r0_enable <= 0; // initialize R1 with the value $18
                end

                T0: begin // see if you need to de-assert these signals
                    #10 PC_select <= 1; MAR_enable <= 1; PC_increment_enable <= 1; Z_enable <= 1;
                    #15 PC_select <= 0; MAR_enable <= 0; PC_increment_enable <= 0; Z_enable <= 0;
                end

                T1: begin
                    #10 Z_LO_select <= 1; PC_enable <= 1; read <= 1; MDR_enable <= 1; MDataIN <=
                        32'h18228000; // opcode for "add R0, R4, R5"
                    #15 Z_LO_select <= 0; PC_enable <= 0; read <= 0; MDR_enable <= 0;
                end

                T2: begin
                    #10 MDR_select <= 1; IR_enable <= 1;
                    #15 MDR_select <= 0; IR_enable <= 0;
                end

                T3: begin
                    #10 r4_select <= 1; Y_enable <= 1;
                    #15 r4_select <= 0; Y_enable <= 0;
                end

                T4: begin
                    #10 r5_select <= 1; alu_instruction <= 5'b00011; Z_enable <= 1;
                    #15 r5_select <= 0; alu_instruction <= 0; Z_enable <= 0;
                end

                T5: begin
                    #10 Z_LO_select <= 1; r0_enable <= 1;
                    #15 Z_LO_select <= 0; r0_enable <= 0;
                end
            endcase
        end
endmodule
```

ADD Operation RTL Simulation



SUB Operation RTL Simulation



MUL/DIV Testbench

The **MUL** (multiplication) operation and **DIV** (division) operation have the same testbench. The main difference between the two is that the opcode in the MUL operation is (32'h7B380000 or 5'b01111) and the opcode in the DIV operation is (32'h83380000 or 5'b10000).

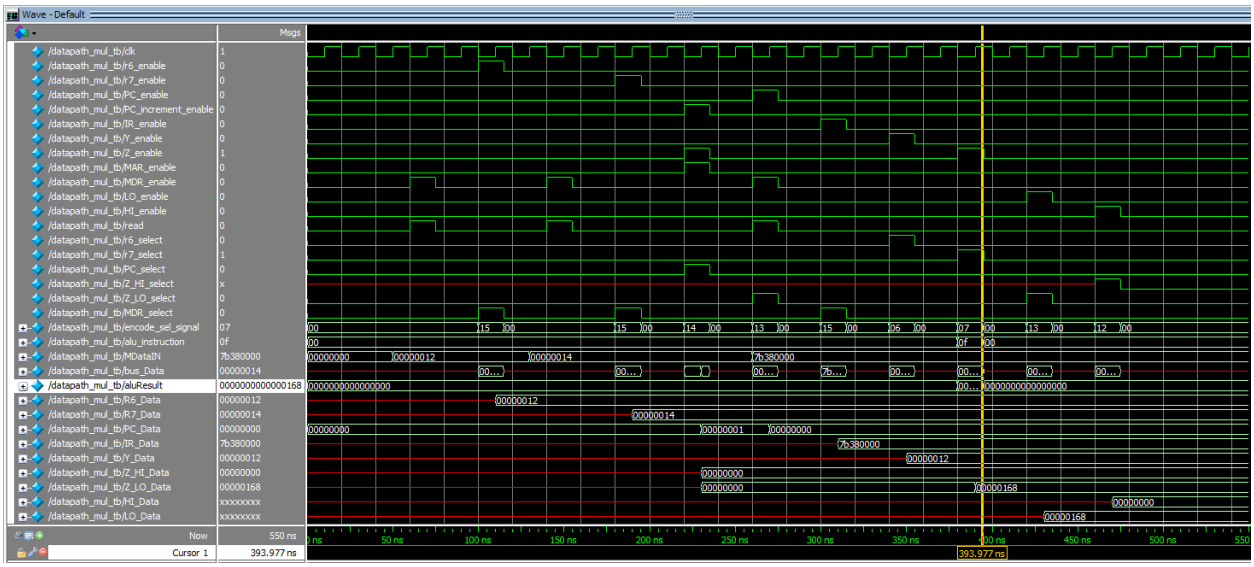
MUL Operation Testbench Code

```
`timescale 1ns/10ps
module datapath_mul_tb;
    // CPU signals
    reg clk;

    // Register write/enable signals
    reg r6_enable, r7_enable;
    reg PC_enable, PC_increment_enable, IR_enable;
    reg Y_enable, Z_enable;
    reg MAR_enable, MDR_enable;
    reg LO_enable, HI_enable;

    // Memory Data Multiplexer Read/Select Signal
    reg read;

    // Encoder Output Select Signals
    reg r6_select, r7_select;
    reg PC_select;
    reg Z_HI_select;
    reg Z_LO_select;
    reg MDR_select;

    wire [4:0] encode_sel_signal;

    // ALU Opcode
    reg [4:0] alu_instruction;

    // Input Data Signals
    reg [31:0] MDataIN;

    // Output Data Signals
    wire [31:0] bus_Data; // Data currently in the bus
    wire [63:0] aluResult;

    wire [31:0] R6_Data, R7_Data;

    wire [31:0] PC_Data, IR_Data;
    wire [31:0] Y_Data;
    wire [31:0] Z_HI_Data, Z_LO_Data;
    wire [31:0] HI_Data, LO_Data;
    wire [31:0] MAR_Data, MDR_Data;

    // Time Signals and Load Registers
    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] Present_state = Default;

    datapath datapathAND(   // CPU signals
    .clk(clk),

    // Register write/enable signals
    .r6_enable(r6_enable), .r7_enable(r7_enable),
    .PC_enable(PC_enable), .PC_increment_enable(PC_increment_enable), .IR_enable(IR_enable),
    .Y_enable(Y_enable), .Z_enable(Z_enable),
    .MAR_enable(MAR_enable), .MDR_enable(MDR_enable),
    .HI_enable(HI_enable), .LO_enable(LO_enable),

    // Memory Data Multiplexer Read/Select Signal
    .read(read),
```

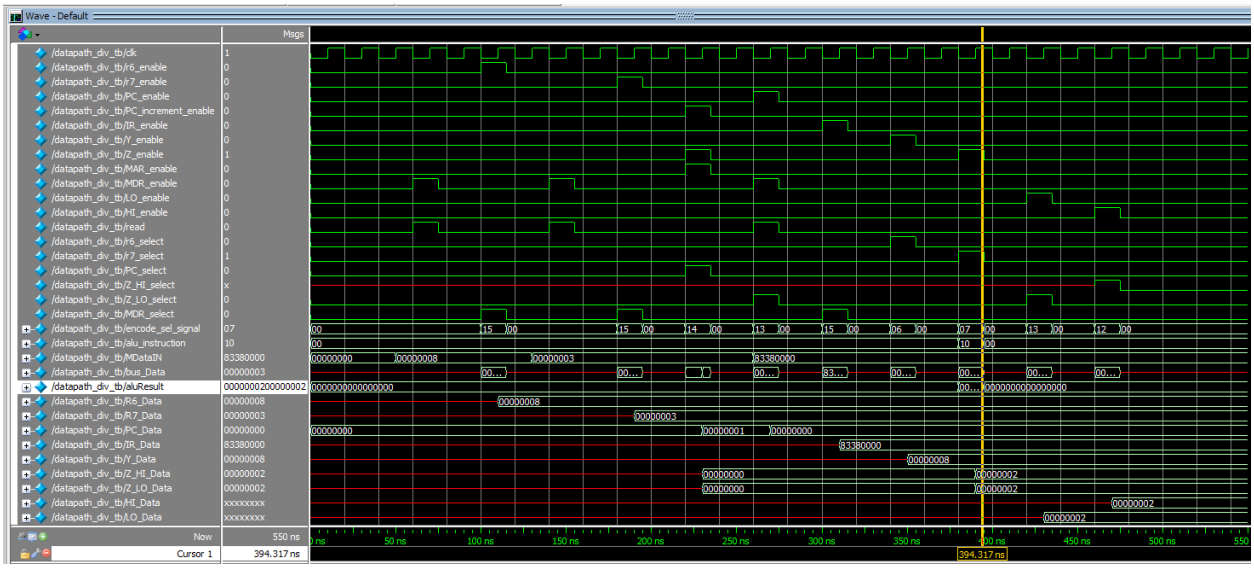
```
// Encoder Output Select Signals
.r6_select(r6_select), .r7_select(r7_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.encode_sel_signal(encode_sel_signal),

// ALU Opcode
.alu_instruction(alu_instruction),

// Input Data Signals
.MDataIN(MDataIN),

// Output Data Signals
.bus_Data(bus_Data), // Data currently in the bus
.aluResult(aluResult),

.R6_Data(R6_Data), .R7_Data(R7_Data),

.PC_Data(PC_Data), .IR_Data(IR_Data),
.Y_Data(Y_Data),
.Z_HI_Data(Z_HI_Data), .Z_LO_Data(Z_LO_Data),
.HI_Data(HI_Data), .LO_Data(LO_Data),
.MAR_Data(MAR_Data), .MDR_Data(MDR_Data));

// add test logic here
always #10 clk = ~clk;

initial begin
    clk = 0;
end

always @(posedge clk) // finite state machine; if clk 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 = 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;
            T5 : #40 Present_state = T6;
        endcase
    end

always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                PC_select <= 0; Z_LO_select <= 0; MDR_select <= 0; // initialize the signals
                r6_select <= 0; r7_select <= 0; MAR_enable <= 0; Z_enable <= 0;
                PC_enable <=0; MDR_enable <= 0; IR_enable <= 0; Y_enable <= 0;
                PC_increment_enable <= 0; HI_enable <= 0; LO_enable <= 0;
                read <= 0; alu_instruction <= 0; r6_enable <= 0; r7_enable <= 0;
                MDataIN <= 32'h00000000;
            end

            Reg_load1a: begin
                MDataIN <= 32'h00000012;
                read = 0; MDR_enable = 0; // the first zero is there for completeness
                #10 read <= 1; MDR_enable <= 1;
                #15 read <= 0; MDR_enable <= 0;
            end
```

```
                Reg_load1b: begin
                    #10 MDR_select <= 1; r6_enable <= 1;
                    #15 MDR_select <= 0; r6_enable <= 0; // initialize R2 with the value $12
                end

                Reg_load2a: begin
                    MDataIN <= 32'h00000014;
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load2b: begin
                    #10 MDR_select <= 1; r7_enable <= 1;
                    #15 MDR_select <= 0; r7_enable <= 0; // initialize R3 with the value $14
                end

                T0: begin // see if you need to de-assert these signals
                    #10 PC_select <= 1; MAR_enable <= 1; PC_increment_enable <= 1; Z_enable <= 1;
                    #15 PC_select <= 0; MAR_enable <= 0; PC_increment_enable <= 0; Z_enable <= 0;
                end

                T1: begin
                    #10 Z_LO_select <= 1; PC_enable <= 1; read <= 1; MDR_enable <= 1; MDataIN <=
                        32'h7B380000; // opcode for "mul R6, R7"
                    #15 Z_LO_select <= 0; PC_enable <= 0; read <= 0; MDR_enable <= 0;
                end

                T2: begin
                    #10 MDR_select <= 1; IR_enable <= 1;
                    #15 MDR_select <= 0; IR_enable <= 0;
                end

                T3: begin
                    #10 r6_select <= 1; Y_enable <= 1;
                    #15 r6_select <= 0; Y_enable <= 0;
                end

                T4: begin
                    #10 r7_select <= 1; alu_instruction <= 5'b01111; Z_enable <= 1;
                    #15 r7_select <= 0; alu_instruction <= 0; Z_enable <= 0;
                end

                T5: begin
                    #10 Z_LO_select <= 1; LO_enable <= 1;
                    #15 Z_LO_select <= 0; LO_enable <= 0;
                end

                T6: begin
                    #10 Z_HI_select <= 1; HI_enable <= 1;
                    #15 Z_HI_select <= 0; HI_enable <= 0;
                end
            endcase
        end
endmodule
```

MUL Operation RTL Simulation



DIV Operation RTL Simulation



SHL/SHR/SHRA Testbench

The **SHL** (shift left) operation, **SHR** (shift right) operation, and **SHRA** (shift right arithmetic) operation have the same testbench. The main difference between the three is that the opcode in the SHL operation is (32'h489A8000 or 5'b01001), the opcode in the SHR operation is (32'h389A8000 or 5'b00111) and the opcode in the SHRA operation is (32'h409A8000 or 5'b01000).

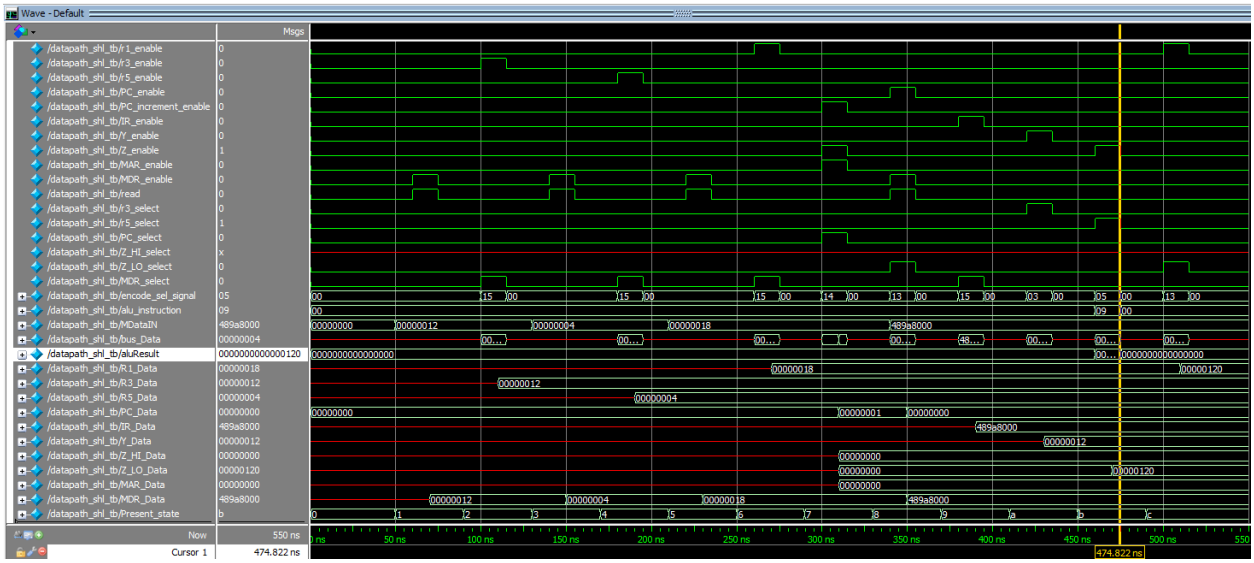
SHRA Operation Testbench Code

```
`timescale 1ns/10ps
module datapath_shra_tb;
    // CPU signals
    reg clk;

    // Register write/enable signals
    reg r1_enable, r3_enable, r5_enable;
    reg PC_enable, PC_increment_enable, IR_enable;
    reg Y_enable, Z_enable;
    reg MAR_enable, MDR_enable;

    // Memory Data Multiplexer Read/Select Signal
    reg read;

    // Encoder Output Select Signals
    reg r3_select, r5_select;
    reg PC_select;
    reg Z_HI_select;
    reg Z_LO_select;
    reg MDR_select;

    wire [4:0] encode_sel_signal;

    // ALU Opcode
    reg [4:0] alu_instruction;

    // Input Data Signals
    reg [31:0] MDataIN;

    // Output Data Signals
    wire [31:0] bus_Data; // Data currently in the bus
    wire [63:0] aluResult;

    wire [31:0] R1_Data, R3_Data, R5_Data;

    wire [31:0] PC_Data, IR_Data;
    wire [31:0] Y_Data;
    wire [31:0] Z_HI_Data, Z_LO_Data;
    wire [31:0] MAR_Data, MDR_Data;

    // Time Signals and Load Registers
    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 datapathAND(   // CPU signals
    .clk(clk),

    // Register write/enable signals
    .r1_enable(r1_enable), .r3_enable(r3_enable), .r5_enable(r5_enable),
    .PC_enable(PC_enable), .PC_increment_enable(PC_increment_enable), .IR_enable(IR_enable),
    .Y_enable(Y_enable), .Z_enable(Z_enable),
    .MAR_enable(MAR_enable), .MDR_enable(MDR_enable),

    // Memory Data Multiplexer Read/Select Signal
    .read(read),
```

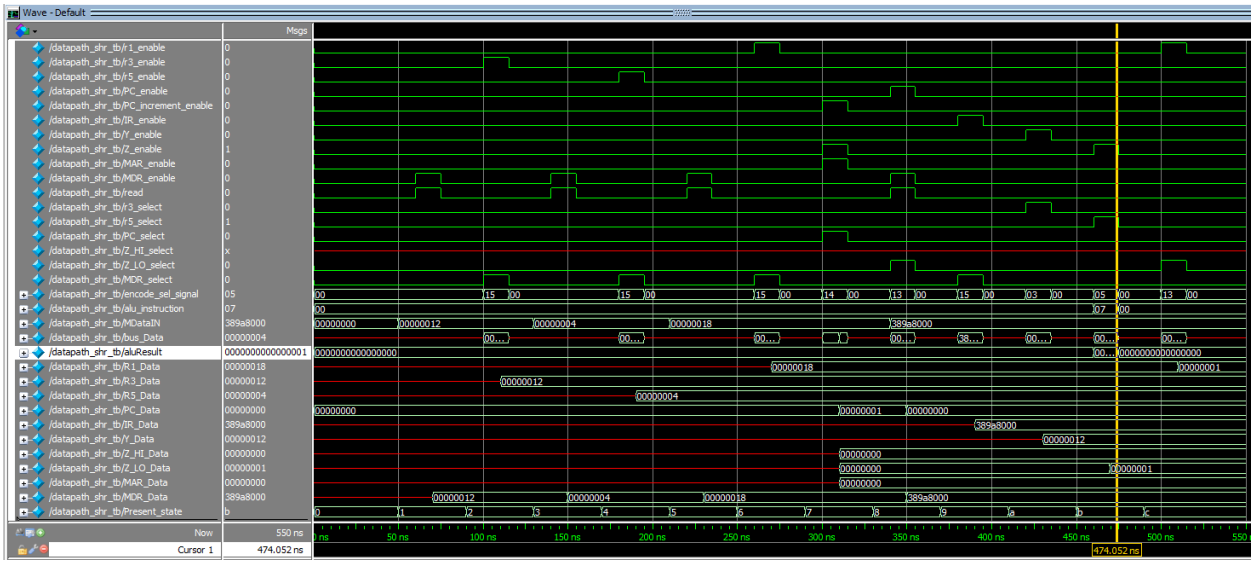
```
// Encoder Output Select Signals
.r3_select(r3_select), .r5_select(r5_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.encode_sel_signal(encode_sel_signal),

// ALU Opcode
.alu_instruction(alu_instruction),

// Input Data Signals
.MDataIN(MDataIN),

// Output Data Signals
.bus_Data(bus_Data), // Data currently in the bus
.aluResult(aluResult),

.R1_Data(R1_Data), .R3_Data(R3_Data), .R5_Data(R5_Data),

.PC_Data(PC_Data), .IR_Data(IR_Data),
.Y_Data(Y_Data),
.Z_HI_Data(Z_HI_Data), .Z_LO_Data(Z_LO_Data),
.MAR_Data(MAR_Data), .MDR_Data(MDR_Data));

// add test logic here
always #10 clk = ~clk;

initial begin
    clk = 0;
end

always @(posedge clk) // finite state machine; if clk 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 in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                PC_select <= 0; Z_LO_select <= 0; MDR_select <= 0; // initialize the signals
                r3_select <= 0; r5_select <= 0; MAR_enable <= 0; Z_enable <= 0;
                PC_enable <=0; MDR_enable <= 0; IR_enable <= 0; Y_enable <= 0;
                PC_increment_enable <= 0; read <= 0; alu_instruction <= 0;
                r1_enable <= 0; r3_enable <= 0; r5_enable <= 0; MDataIN <= 32'h00000000;
            end

            Reg_load1a: begin
                MDataIN <= 32'hFFFFFFFA;
                read = 0; MDR_enable = 0; // the first zero is there for completeness
                #10 read <= 1; MDR_enable <= 1;
                #15 read <= 0; MDR_enable <= 0;
            end
```

```
                Reg_load1a: begin
                    MDataIN <= 32'hFFFFFFFA;
                    read = 0; MDR_enable = 0; // the first zero is there for completeness
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load1b: begin
                    #10 MDR_select <= 1; r3_enable <= 1;
                    #15 MDR_select <= 0; r3_enable <= 0; // initialize R2 with the value $12
                end

                Reg_load2a: begin
                    MDataIN <= 32'h00000002;
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load2b: begin
                    #10 MDR_select <= 1; r5_enable <= 1;
                    #15 MDR_select <= 0; r5_enable <= 0; // initialize R3 with the value $14
                end

                Reg_load3a: begin
                    MDataIN <= 32'h00000018;
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load3b: begin
                    #10 MDR_select <= 1; r1_enable <= 1;
                    #15 MDR_select <= 0; r1_enable <= 0; // initialize R1 with the value $18
                end

                T0: begin // see if you need to de-assert these signals
                    #10 PC_select <= 1; MAR_enable <= 1; PC_increment_enable <= 1; Z_enable <= 1;
                    #15 PC_select <= 0; MAR_enable <= 0; PC_increment_enable <= 0; Z_enable <= 0;
                end

                T1: begin
                    #10 Z_LO_select <= 1; PC_enable <= 1; read <= 1; MDR_enable <= 1; MDataIN <=
                        32'h409A8000; // opcode for “SHRA R1, R3, R5"
                    #15 Z_LO_select <= 0; PC_enable <= 0; read <= 0; MDR_enable <= 0;
                end

                T2: begin
                    #10 MDR_select <= 1; IR_enable <= 1;
                    #15 MDR_select <= 0; IR_enable <= 0;
                end

                T3: begin
                    #10 r3_select <= 1; Y_enable <= 1;
                    #15 r3_select <= 0; Y_enable <= 0;
                end

                T4: begin
                    #10 r5_select <= 1; alu_instruction <= 5'b01000; Z_enable <= 1;
                    #15 r5_select <= 0; alu_instruction <= 0; Z_enable <= 0;
                end

                T5: begin
                    #10 Z_LO_select <= 1; r1_enable <= 1;
                    #15 Z_LO_select <= 0; r1_enable <= 0;
                end
            endcase
        end
endmodule
```

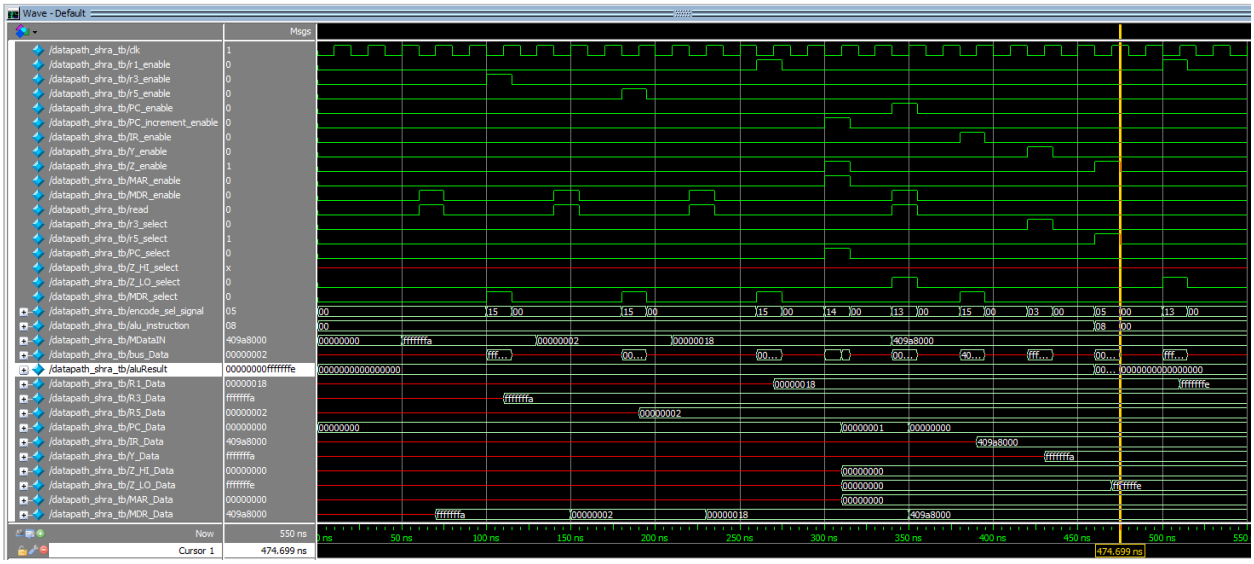
SHL Operation RTL Simulation



SHR Operation RTL Simulation



SHRA Operation RTL Simulation



ROL/ROR Testbench

The **ROL** (rotate left) and **ROR** (rotate right) have the same testbench. The main difference between the two is that the opcode in the ROL operation is (32'h589A8000 or 5'b01011) and the opcode in the ROR operation is (32'h509A8000 or 5'b01010).

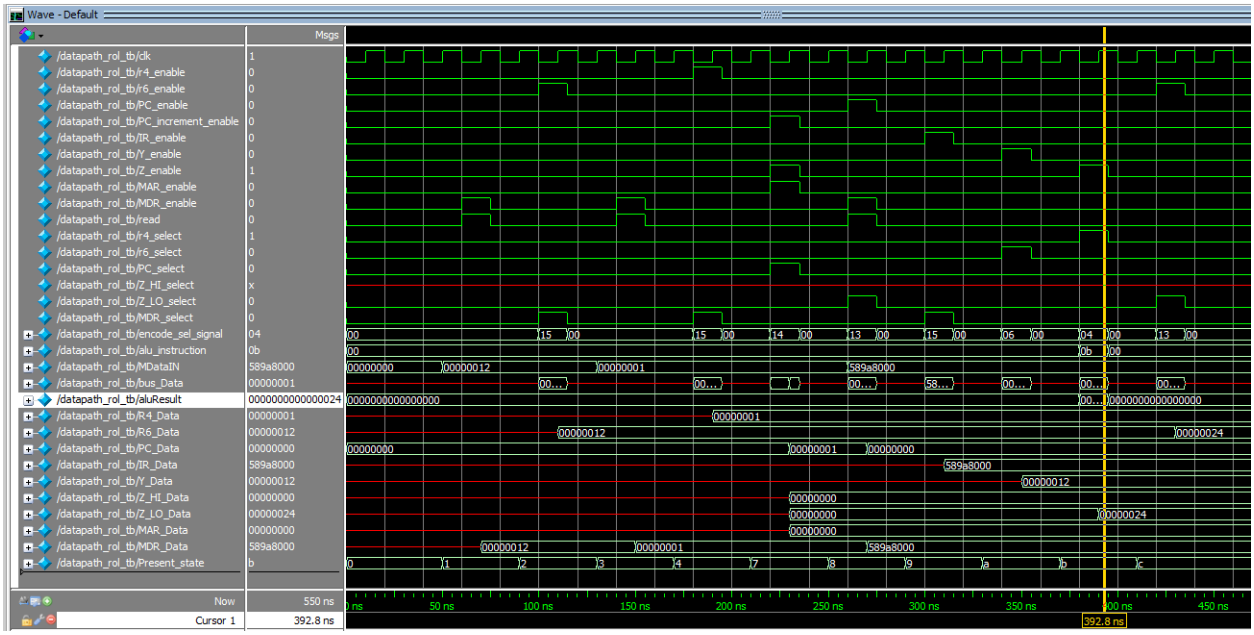
ROR Operation Testbench Code

```
`timescale 1ns/10ps
module datapath_ror_tb;
    // CPU signals
    reg clk;

    // Register write/enable signals
    reg r4_enable, r6_enable;
    reg PC_enable, PC_increment_enable, IR_enable;
    reg Y_enable, Z_enable;
    reg MAR_enable, MDR_enable;

    // Memory Data Multiplexer Read/Select Signal
    reg read;

    // Encoder Output Select Signals
    reg r4_select, r6_select;
    reg PC_select;
    reg Z_HI_select;
    reg Z_LO_select;
    reg MDR_select;

    wire [4:0] encode_sel_signal;

    // ALU Opcode
    reg [4:0] alu_instruction;

    // Input Data Signals
    reg [31:0] MDataIN;

    // Output Data Signals
    wire [31:0] bus_Data; // Data currently in the bus
    wire [63:0] aluResult;

    wire [31:0] R4_Data, R6_Data;

    wire [31:0] PC_Data, IR_Data;
    wire [31:0] Y_Data;
    wire [31:0] Z_HI_Data, Z_LO_Data;
    wire [31:0] MAR_Data, MDR_Data;

    // Time Signals and Load Registers
    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 datapathAND(   // CPU signals
    .clk(clk),

    // Register write/enable signals
    .r4_enable(r4_enable), .r6_enable(r6_enable),
    .PC_enable(PC_enable), .PC_increment_enable(PC_increment_enable), .IR_enable(IR_enable),
    .Y_enable(Y_enable), .Z_enable(Z_enable),
    .MAR_enable(MAR_enable), .MDR_enable(MDR_enable),

    // Memory Data Multiplexer Read/Select Signal
    .read(read),
```

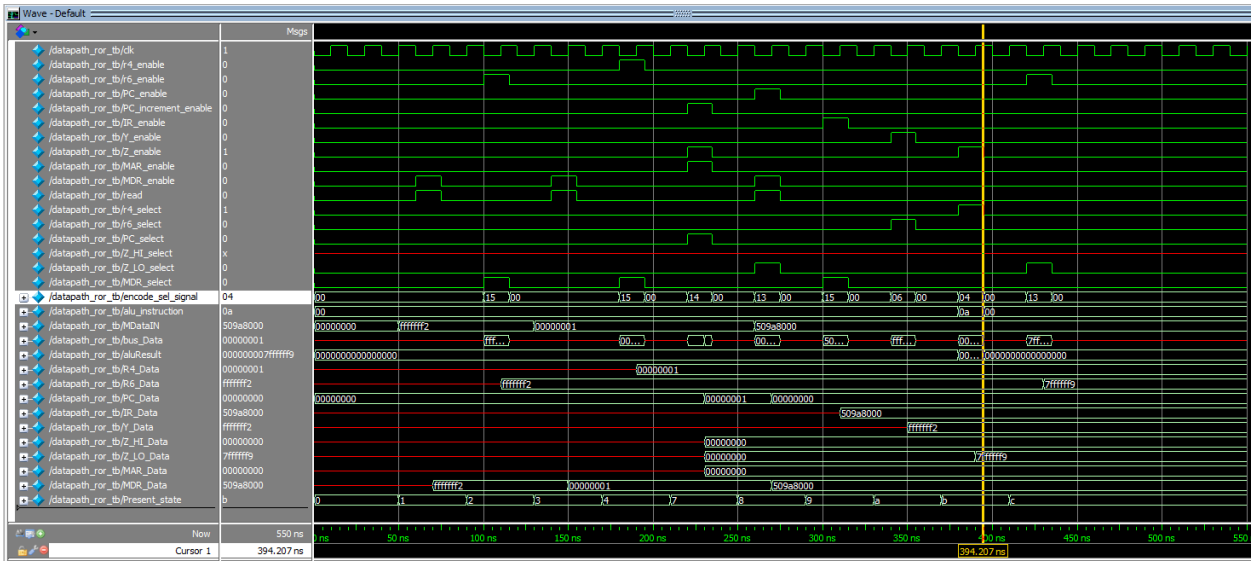
```
// Encoder Output Select Signals
.r4_select(r4_select), .r6_select(r6_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.encode_sel_signal(encode_sel_signal),

// ALU Opcode
.alu_instruction(alu_instruction),

// Input Data Signals
.MDataIN(MDataIN),

// Output Data Signals
.bus_Data(bus_Data), // Data currently in the bus
.aluResult(aluResult),

.R4_Data(R4_Data), .R6_Data(R6_Data),

.PC_Data(PC_Data), .IR_Data(IR_Data),
.Y_Data(Y_Data),
.Z_HI_Data(Z_HI_Data), .Z_LO_Data(Z_LO_Data),
.MAR_Data(MAR_Data), .MDR_Data(MDR_Data));

// add test logic here
always #10 clk = ~clk;

initial begin
    clk = 0;
end

always @(posedge clk) // finite state machine; if clk 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 = 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 in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                PC_select <= 0; Z_LO_select <= 0; MDR_select <= 0; // initialize the signals
                r4_select <= 0; r6_select <= 0; MAR_enable <= 0; Z_enable <= 0;
                PC_enable <=0; MDR_enable <= 0; IR_enable <= 0; Y_enable <= 0;
                PC_increment_enable <= 0; read <= 0; alu_instruction <= 0;
                r4_enable <= 0; r6_enable <= 0; MDataIN <= 32'h00000000;
            end

            Reg_load1a: begin
                MDataIN <= 32'h12345678;
                read = 0; MDR_enable = 0; // the first zero is there for completeness
                #10 read <= 1; MDR_enable <= 1;
                #15 read <= 0; MDR_enable <= 0;
            end
```

```
                Reg_load1b: begin
                    #10 MDR_select <= 1; r6_enable <= 1;
                    #15 MDR_select <= 0; r6_enable <= 0; // initialize R2 with the value $12
                end

                Reg_load2a: begin
                    MDataIN <= 32'h0000000A;
                    #10 read <= 1; MDR_enable <= 1;
                    #15 read <= 0; MDR_enable <= 0;
                end

                Reg_load2b: begin
                    #10 MDR_select <= 1; r4_enable <= 1;
                    #15 MDR_select <= 0; r4_enable <= 0; // initialize R3 with the value $14
                end

                T0: begin // see if you need to de-assert these signals
                    #10 PC_select <= 1; MAR_enable <= 1; PC_increment_enable <= 1; Z_enable <= 1;
                    #15 PC_select <= 0; MAR_enable <= 0; PC_increment_enable <= 0; Z_enable <= 0;
                end

                T1: begin
                    #10 Z_LO_select <= 1; PC_enable <= 1; read <= 1; MDR_enable <= 1; MDataIN <=
                        32'h509A8000; // opcode for “ROR R6, R6, R4"
                    #15 Z_LO_select <= 0; PC_enable <= 0; read <= 0; MDR_enable <= 0;
                end

                T2: begin
                    #10 MDR_select <= 1; IR_enable <= 1;
                    #15 MDR_select <= 0; IR_enable <= 0;
                end

                T3: begin
                    #10 r6_select <= 1; Y_enable <= 1;
                    #15 r6_select <= 0; Y_enable <= 0;
                end

                T4: begin
                    #10 r4_select <= 1; alu_instruction <= 5'b01010; Z_enable <= 1;
                    #15 r4_select <= 0; alu_instruction <= 0; Z_enable <= 0;
                end

                T5: begin
                    #10 Z_LO_select <= 1; r6_enable <= 1;
                    #15 Z_LO_select <= 0; r6_enable <= 0;
                end
            endcase
        end
endmodule
```

ROL Operation RTL Simulation



ROR Operation RTL Simulation



NOT/NEG Testbench

The **NOT** operation and **NEG** (negate) operation have the same testbench. The main difference between the two is that the opcode in the NOT operation is (32'h90980000 or 5'b10010) and the opcode in the NEG operation is (32'h88980000 or 5'b10001).

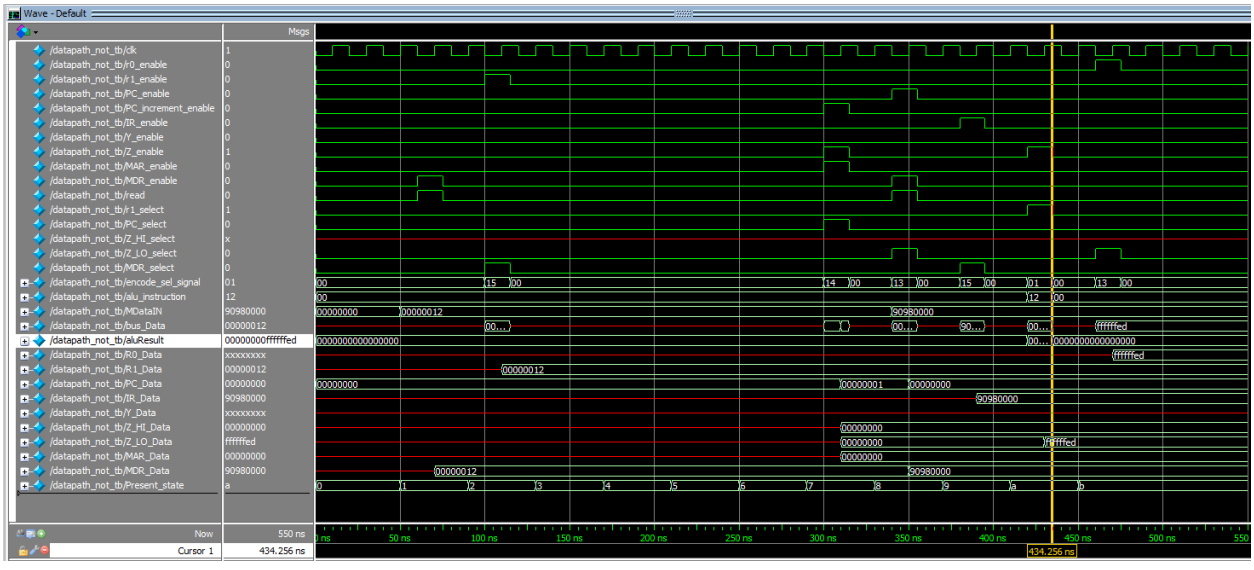
NOT Operation Testbench Code

```
`timescale 1ns/10ps
module datapath_not_tb;
    // CPU signals
    reg clk;

    // Register write/enable signals
    reg r0_enable, r1_enable;
    reg PC_enable, PC_increment_enable, IR_enable;
    reg Y_enable, Z_enable;
    reg MAR_enable, MDR_enable;

    // Memory Data Multiplexer Read/Select Signal
    reg read;

    // Encoder Output Select Signals
    reg r1_select;
    reg PC_select;
    reg Z_HI_select;
    reg Z_LO_select;
    reg MDR_select;

    wire [4:0] encode_sel_signal;

    // ALU Opcode
    reg [4:0] alu_instruction;

    // Input Data Signals
    reg [31:0] MDataIN;

    // Output Data Signals
    wire [31:0] bus_Data; // Data currently in the bus
    wire [63:0] aluResult;

    wire [31:0] R0_Data, R1_Data;

    wire [31:0] PC_Data, IR_Data;
    wire [31:0] Y_Data;
    wire [31:0] Z_HI_Data, Z_LO_Data;
    wire [31:0] MAR_Data, MDR_Data;

    // Time Signals and Load Registers
    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;

    reg [3:0] Present_state = Default;

    datapath datapathAND(   // CPU signals
    .clk(clk),

    // Register write/enable signals
    .r0_enable(r0_enable), .r1_enable(r1_enable),
    .PC_enable(PC_enable), .PC_increment_enable(PC_increment_enable), .IR_enable(IR_enable),
    .Y_enable(Y_enable), .Z_enable(Z_enable),
    .MAR_enable(MAR_enable), .MDR_enable(MDR_enable),

    // Memory Data Multiplexer Read/Select Signal
    .read(read),
```

```
// Encoder Output Select Signals
.r1_select(r1_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.encode_sel_signal(encode_sel_signal),

// ALU Opcode
.alu_instruction(alu_instruction),

// Input Data Signals
.MDataIN(MDataIN),

// Output Data Signals
.bus_Data(bus_Data), // Data currently in the bus
.aluResult(aluResult),

.R0_Data(R0_Data), .R1_Data(R1_Data),

.PC_Data(PC_Data), .IR_Data(IR_Data),
.Y_Data(Y_Data),
.Z_HI_Data(Z_HI_Data), .Z_LO_Data(Z_LO_Data),
.MAR_Data(MAR_Data), .MDR_Data(MDR_Data));

// add test logic here
always #10 clk = ~clk;

initial begin
    clk = 0;
end

always @(posedge clk) // finite state machine; if clk 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;
        endcase
    end

always @(Present_state) // do the required job in each state
    begin
        case (Present_state) // assert the required signals in each clk cycle
            Default: begin
                PC_select <= 0; Z_LO_select <= 0; MDR_select <= 0; // initialize the signals
                r1_select <= 0; MAR_enable <= 0; Z_enable <= 0;
                PC_enable <=0; MDR_enable <= 0; IR_enable <= 0; Y_enable <= 0;
                PC_increment_enable <= 0; read <= 0; alu_instruction <= 0;
                r0_enable <= 0; r1_enable <= 0; MDataIN <= 32'h00000000;
            end

            Reg_load1a: begin
                MDataIN <= 32'h00000012;
                read = 0; MDR_enable = 0; // the first zero is there for completeness
                #10 read <= 1; MDR_enable <= 1;
                #15 read <= 0; MDR_enable <= 0;
            end
```

```
                Reg_load1b: begin
                    #10 MDR_select <= 1; r1_enable <= 1;
                    #15 MDR_select <= 0; r1_enable <= 0; // initialize R2 with the value $12
                end

                T0: begin // see if you need to de-assert these signals
                    #10 PC_select <= 1; MAR_enable <= 1; PC_increment_enable <= 1; Z_enable <= 1;
                    #15 PC_select <= 0; MAR_enable <= 0; PC_increment_enable <= 0; Z_enable <= 0;
                end

                T1: begin
                    #10 Z_LO_select <= 1; PC_enable <= 1; read <= 1; MDR_enable <= 1; MDataIN <=
                        32'h90980000; // opcode for NOT R0, R1"
                    #15 Z_LO_select <= 0; PC_enable <= 0; read <= 0; MDR_enable <= 0;
                end

                T2: begin
                    #10 MDR_select <= 1; IR_enable <= 1;
                    #15 MDR_select <= 0; IR_enable <= 0;
                end

                T3: begin
                    #10 r1_select = 1; alu_instruction <= 5'b10010; Z_enable <= 1;
                    #15 r1_select = 0; alu_instruction <= 0; Z_enable <= 0;
                end

                T4: begin
                    #10 Z_LO_select <= 1; r0_enable <= 1;
                    #15 Z_LO_select <= 0; r0_enable <= 0;
                end
            endcase
        end
endmodule
```

NOT Operation RTL Simulation



NEG Operation RTL Simulation