ELEC 374 – Final Report

Department of Electrical and Computer Engineering
Queen's University

Group 16

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

Date of Submission:
3 April 2023

"We do hereby verify that this written lab report is our own work and contains our own original ideas, concepts, and designs. No portion of this report has been copied in whole or in part from another source, with the possible exception of properly referenced material".

Abstract

This project involved the development of a Reduced Instruction Set Computing (RISC) style processor, featuring general registers, a bidirectional bus, an Arithmetic Logic Unit (ALU), a memory subsystem, and a control unit. The design was implemented onto a Cyclone board, providing a practical demonstration of its functionality.

The processor was designed using Verilog, a hardware description language that allows for the modeling digital circuits. Verilog was used to create a series of modules, each responsible for a specific aspect of the processor's functionality. Once each module was complete, they were integrated together using Verilog's module instantiation feature. This involved connecting the inputs and outputs of each module together to create a complete processor design.

Once the design for each phase was complete, it was simulated using ModelSim Altera to ensure it was functional. The design was then synthesized using Quartus II, a software tool that converts the design into a format that can be programmed onto the Cyclone board.

The specifications of a RISC processor are typically driven by the need to achieve high performance with low power consumption. To achieve this, the processor is designed to execute a small set of instructions quickly and efficiently, with a focus on executing the most common operations required by software. The key design specifications of the processor include:

- Implementation of a 32-bit carry look-ahead adder used for the addition and subtraction
- A 32-bit multiplier with bit-pair recoding based on Booth's algorithm
- A 32-bit signed integer division unit using the restoring division algorithm
- Dedicated program counter incrementor
- Input and output capabilities
- Custom synchronous RAM module with read/write capabilities
- Select and encode logic used to decode the instruction from RAM
- CON FF logic used to implement branching features
- Control unit used to assert enable and select signals based on the operation in the opcode

All the features listed above are functional and can be seen in the functional simulation screenshots located in the Appendices.

Due to a lack of time, the completed processor was not successfully implemented onto the Cyclone board. However, most features of the processor are fully functional and with some more time, it could be implemented onto a chip. The team plans to continue development to complete the processor and add more bonus features.

Some recommendations for future improvements that are being investigated are optimizations to the code organization, processor efficiency, and code documentation.

Table of Contents

Project Specification

The purpose of this project was to design and program a Reduced Instruction Set Computing (RISC) style processor using Verilog and implement it onto a Cyclone board.

The project was split up into four phases. The purpose of the first phase was to develop an initial outline of the processor by implementing the datapath which can be seen in Figure 3. This phase included the creation of:

- Sixteen 32-bit general purpose registers (R0 – R15)
- Seven 32-bit special registers: PC, IR, Y, Z_HI, Z_LO, HI, and LO
- A module dedicated to incrementing the program counter
- A single bidirectional bus implemented using a 32-to-5 encoder and a 32-to-1 multiplexer
- A 32-bit memory data register that contains a 2-to-1 multiplexer controlled with a read signal
- An arithmetic logic unit (ALU) with the following operations: addition, subtraction, multiplication, division, bit-shift, bit-rotate, logical AND, logical OR, logical NOT, negate

In phase two, the main objective was to further develop the memory subsystem. This phase also introduced new ALU operations, instruction decoding, branching logic, I/O capabilities, and modifications to R0 and R15.

In phase three, the control unit was created. The purpose of this unit was to receive inputs from various signals and generate outputs in order to execute instructions. This phase also included a list of instructions to be stored in the RAM for execution.

In the final phase, the goal was for the processor to be implemented onto the cyclone III, cyclone V or any larger chip that could implement the final design. Unfortunately, the team did not make it to this stage but with the constant code optimization, the team was aiming to implement the processor on the cyclone III chip.

Project Design and Implementation

The purpose of the section is to provide in-depth detail on each aspect of the simple RISC computer project.

Register Implementation

The register serves as a temporary storage location for values. The block diagram of a register can be seen in Figure 1. The implementation of the register takes a 32-bit D-input and sends out a 32-bit Q-output whenever the R1in (enable) signal is high. In our design, we named the input bus_Data to clearly depict that it comes from the bus and the output is named R1_Data.

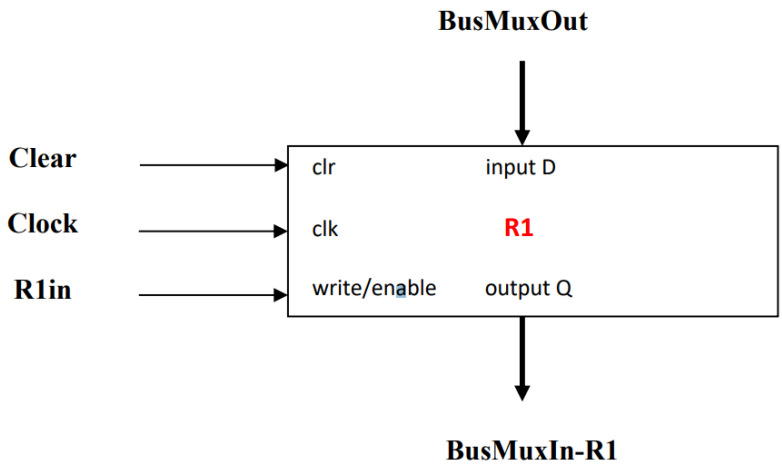


Figure 1: A block diagram of a register that specifies its inputs and outputs.

Bus Design

The processor uses a 32-to-5 encoder and a 32-to-1 multiplexer to create a bidirectional bus capable of carrying a 32-bit signal as seen in Figure 2. This design is used to select a specific register to display on the bus. The single bus design ensures one register is read at a time. The bus works by sending select signals into the encoder which generates 5 output signals. The output of the encoder is then sent to the input of the multiplexer select signals within the datapath. By doing this, each input to the multiplexer is mapped to a 5-bit pattern that is used to select which signal is put on the bus.

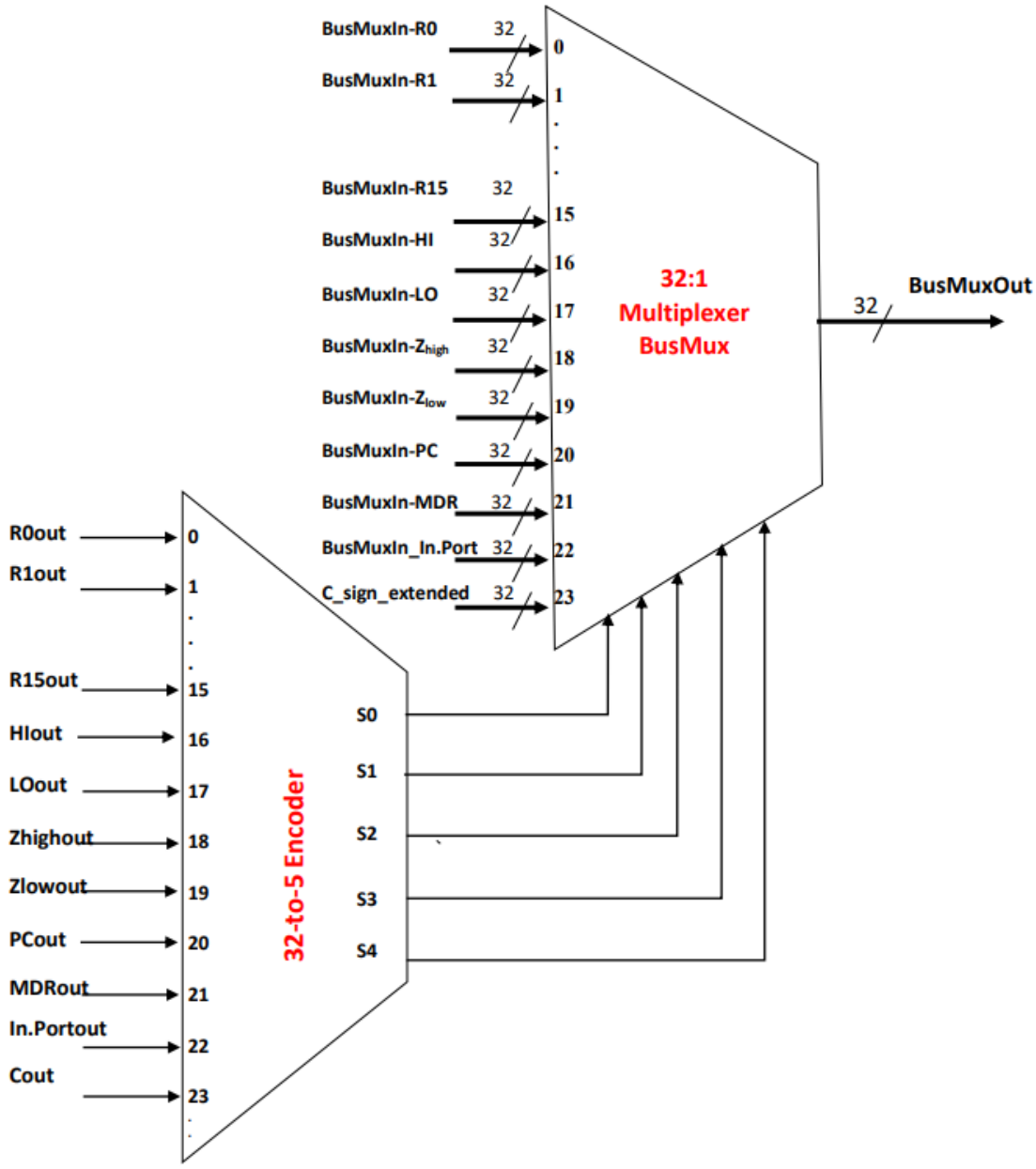


Figure 2: A schematic that shows how the bus chooses a value to display.

Datapath

The schematic of the datapath can be seen below in Figure 3. The datapath was created to connect all the logic units of the processor together.

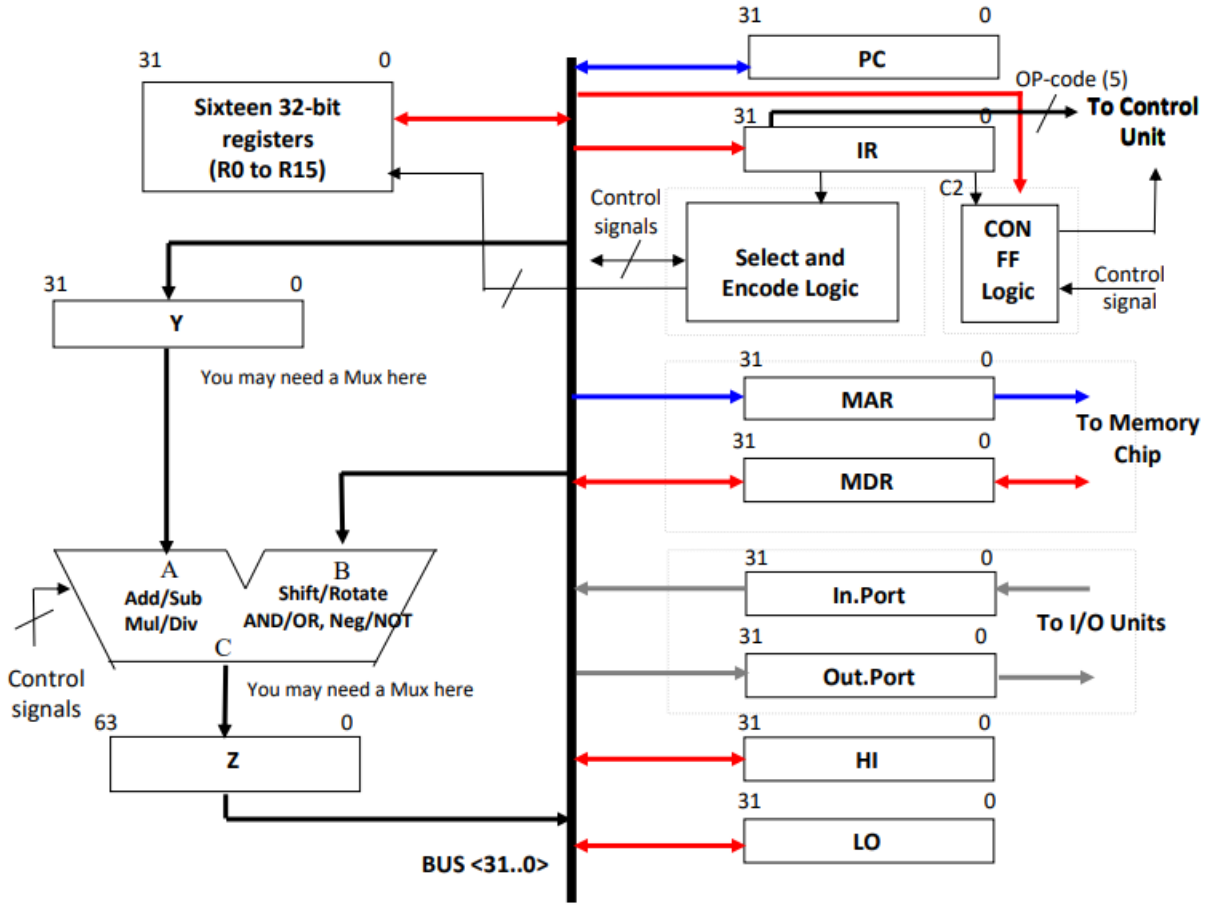


Figure 3: A schematic that shows the interaction of the processor components in phase one, also called the datapath.

The datapath contains a 32-bit architecture bidirectional bus that was implemented using a 32-to-5 encoder and a 32-to-1 multiplexer. Sixteen general purpose registers, special registers, an ALU, a memory subsystem, I/O ports, select and encode logic, and CON FF logic.

Memory Data Register

The Memory Data Register (MDR) is a register that transfers data to the bus from memory (read), and to memory from the bus (write). The MDR was implemented by using an instance of the normal register and adding some logic to represent the 2-to-1 multiplexer. The purpose of this multiplexer is to choose between two data input sources: the bus, and the RAM. An image of the MDR block diagram can be seen in Figure 4.

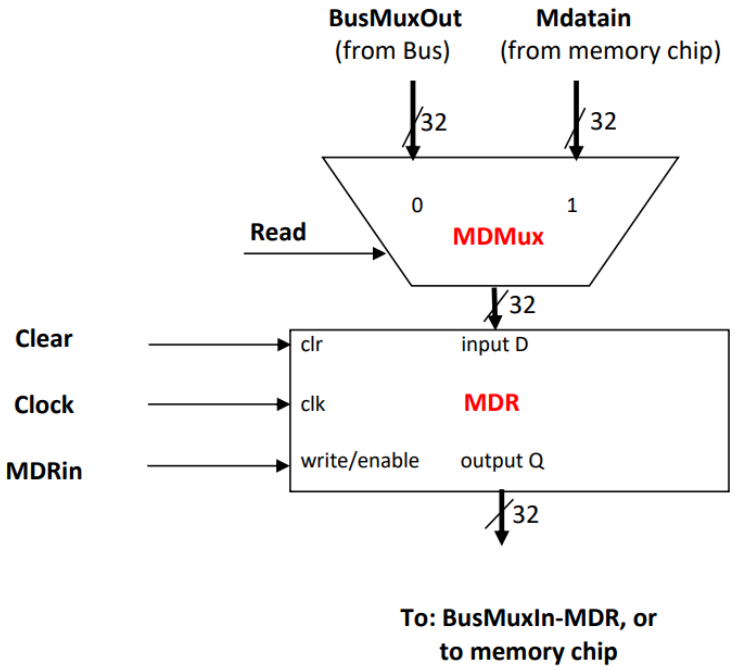


Figure 4: Block Diagram of the memory data register (MDR).

Arithmetic Logic Unit (ALU)

The ALU contains operations for: load, load immediate, store, add, sub, AND, OR, right bit shift, arithmetic right bit shift, left bit shift, right bit rotate, left bit rotate, add immediate, AND immediate, OR immediate, multiplication, division, negate, NOT, branch, jr, jal, in, out, mfhi, mflo.

The addition unit was implemented using a carry look-ahead adder. This logic circuit was also used in the subtraction unit but the second operand needed to be negated in order to use the CLA for subtraction.

The multiplication unit was implemented using booth's algorithm with bit-pair recoding. The output of the multiplication unit is 64-bits.

The division unit was implemented using a restoring division algorithm. The upper 32-bits of the result represents the remainder, and the lower 32-bits represents the quotient.

Memory Subsystem

The memory subsystem stores and retrieves instructions and data for the simple RISC computer. It consists of the Random Access Memory (RAM), memory data register (MDR), and the memory address register (MAR). This memory subsystem is used when the CPU needs to read instructions or specific data. It makes a read or write request depending on the action it needs to perform.

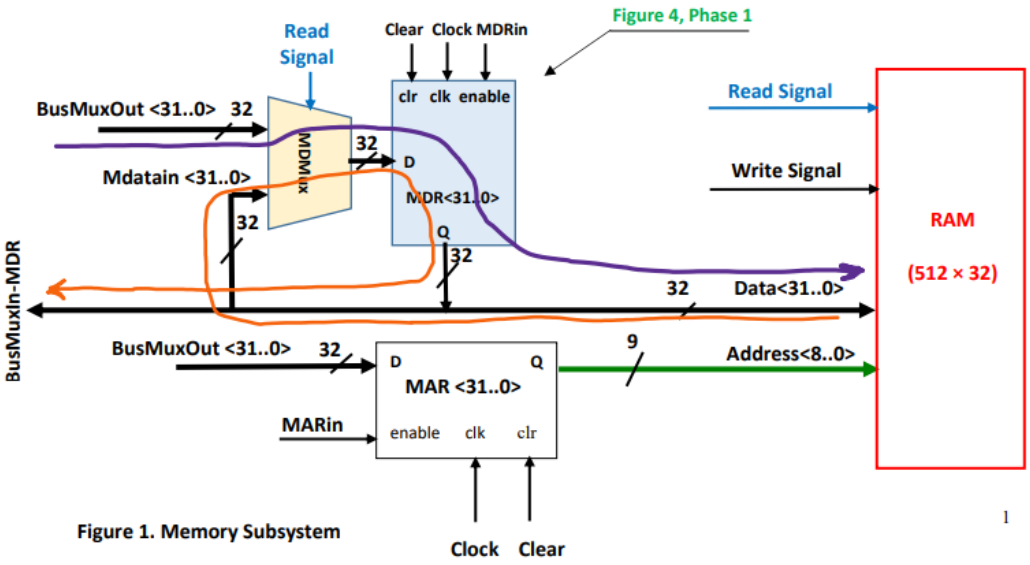


Figure 5: A figure that depicts the memory subsystem which houses the RAM, MDR, and MAR.

Select & Encode Logic

The select and encode logic is used to assert enable and select signals for the sixteen general purpose registers. It can do this by decoding the information in the instruction register (IR).

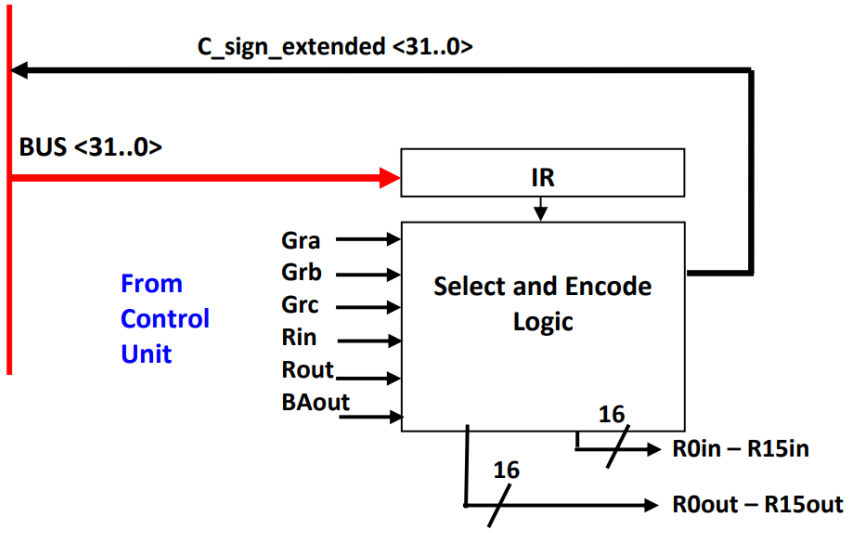


Figure 6: A block diagram of the select and encode logic unit.

As shown in Figure 6, the “Select and Encode” logic accepts the Gra, Grb, Grc, Rin, Rout, and BAout signals as external inputs.

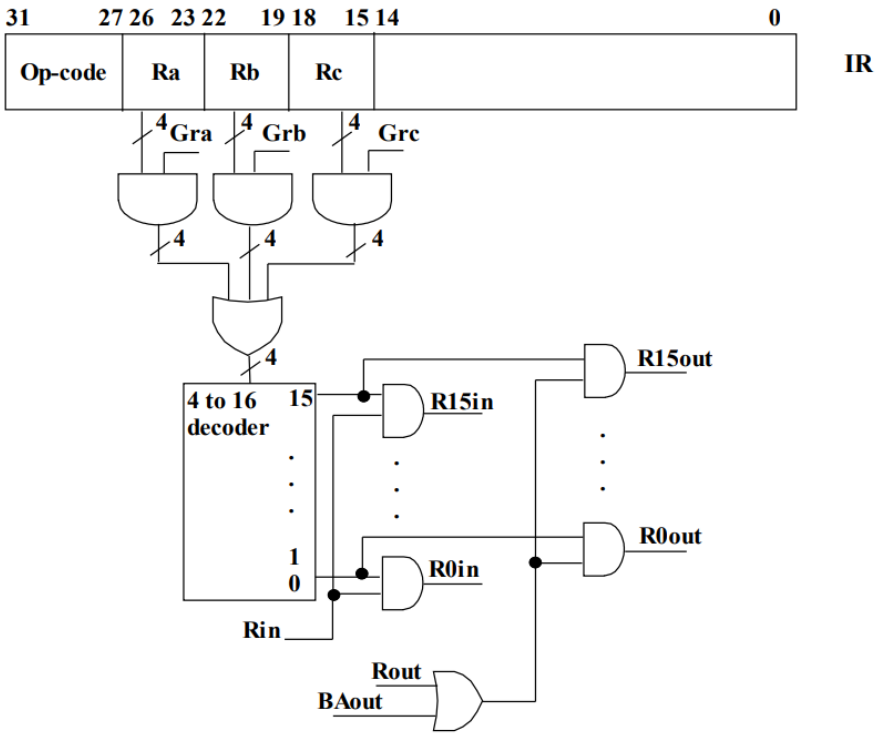


Figure 7: Digital logic circuit for the select and encode logic.

Revised Register R0

The revision of the R0 register makes it so that the register always holds a value of 0. It is designed not to be overwritten and to simplify the design to ensure no repeated checks of whether R0 holds the value of 0 must take place. A schematic diagram can be seen below in Figure 8.

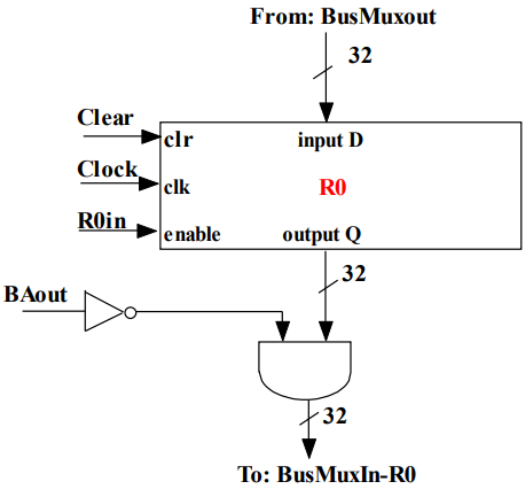


Figure 8: A block diagram of the revised R0 register.

CON FF Logic

The CON FF logic circuit can be seen in Figure 9. The purpose of this circuit is to determine whether the conditions for a branch instruction have been met. In order to implement this, bit 19 and 20 from the

instruction register were used to determine the type of branch instruction. This was then sent through a 2-to-4 decoder to map each branch outcome to a 4-bit pattern. The flip flop in the CON FF logic outputs a one bit signal which indicates that the branch conditions have been met and a branch should be executed.

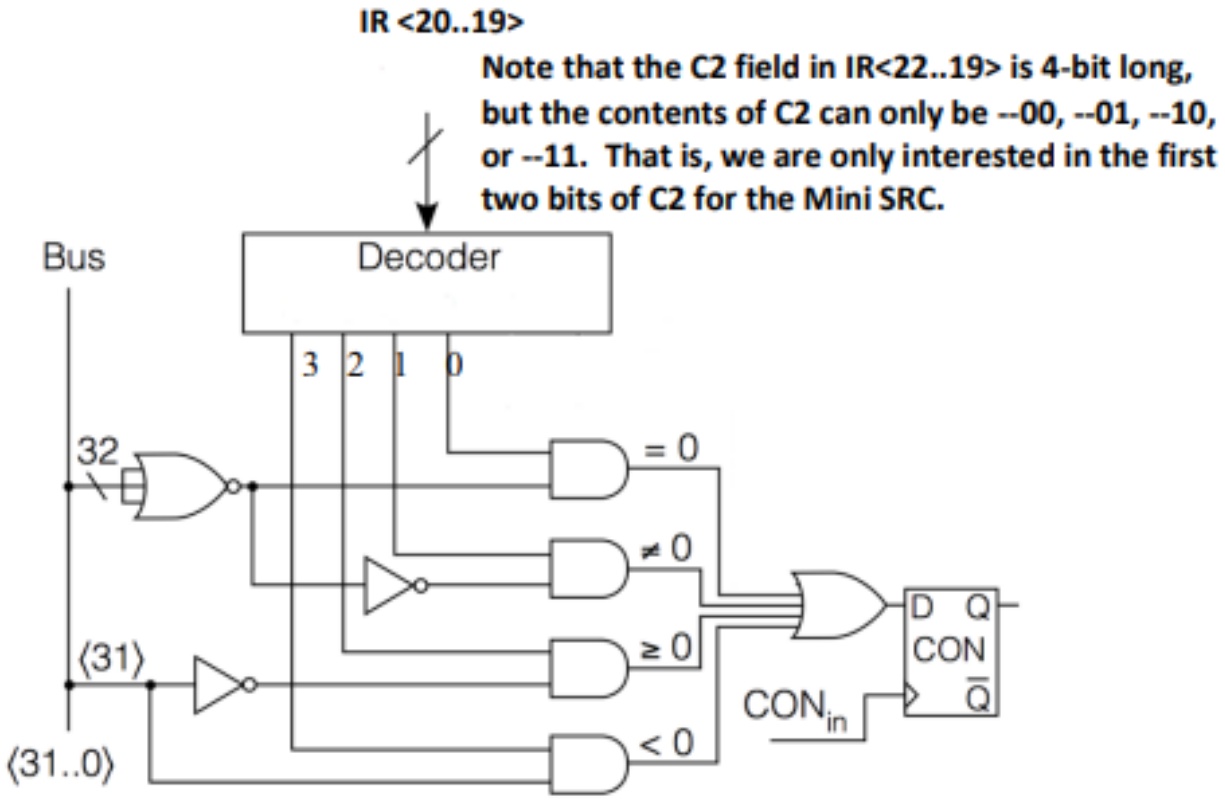
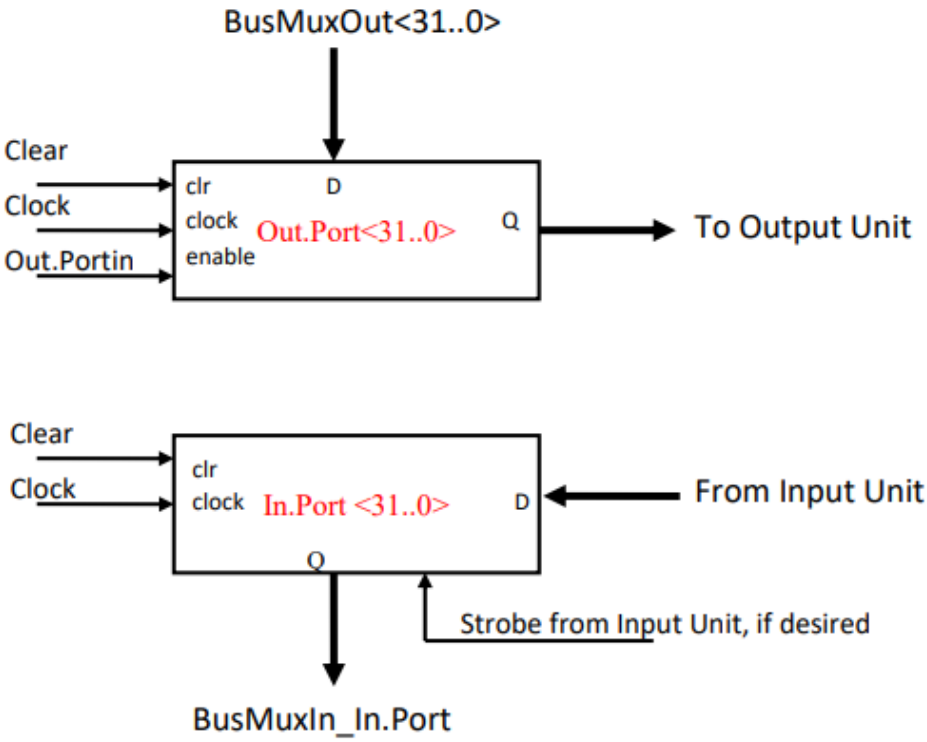


Figure 9: An image of the logic circuit for the CON FF branch logic.

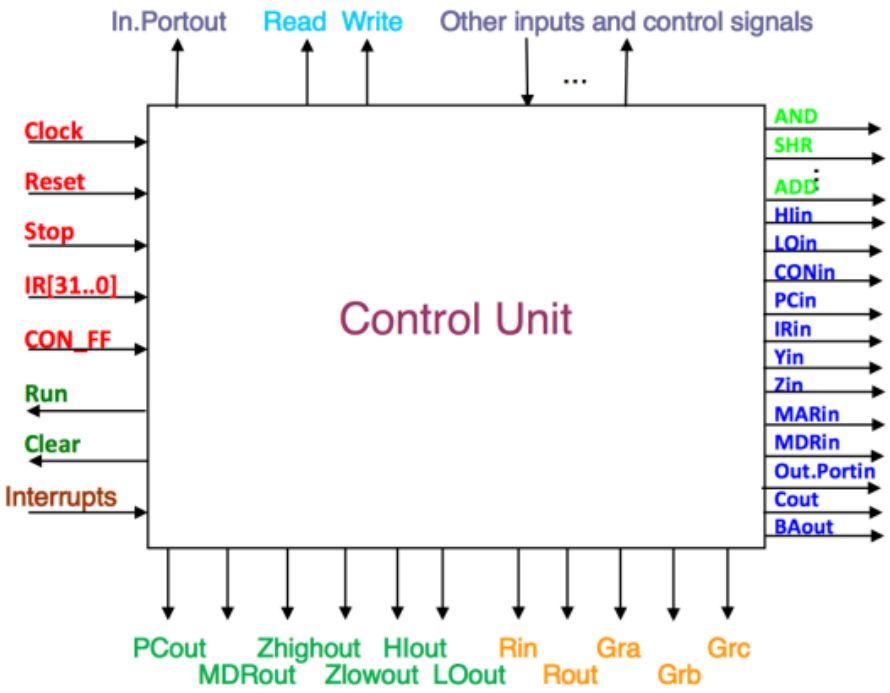
Input and Output Ports

The input and output ports are a way for the processor to communicate with the outside world. The input port can take inputs from outside and put it on the bus. The output port can take data from the bus and send it out to a device. These were implemented by using registers but the only difference is that the inputs and outputs of the registers are different.



Control Unit

The control unit is one of the most crucial parts of the processor. It generates the required control signals for an operation, fetches an instruction from memory, and decodes the provided instructions to decide what logical operation must be done in sequence to execute the instruction correctly. It can be considered the communicator between the other components, CON Flip Flop, the memory subsystem, and ALU. The control unit is shown below.



Carry Lookahead Adder

The carry lookahead adder is a 32-bit adder circuit implementation designed to increase the speed and optimize the addition operation in the ALU. It reduces the delay in calculating the carry bit and keeps track by simultaneously implementing carry signals for each stage. This can be seen in the adder.v module in Appendices.

Program Counter Implementation

The program counter was implemented into its module, which does not require using the ALU for incrementation. The program counter also includes logic to increment the PC contents by one when incPC is enabled, and then it stops incrementing until incPC is high again. This can be seen in the program_counter.v module.

Custom Memory

The team decided not to use the Quartus built-in RAM template and designed a custom synchronous RAM that operates in sync with the processor's clock. The RAM has a built-in clock to control the ability to read and write certain data operations. This can be seen in the Appendix as ram.v.

Evaluation

During the first phase, the team implemented a working Datapath, ALU, that successfully executed the logical operations it was provided. The team was beginning to learn and get more exposed to the logic behind the Datapath and ALU.

During the second/third phase, the team finished the code for the RAM, memory subsystem, CON-FF, inport, outport, and control unit, but since there were no specified instructions, it was unclear how the

team would implement a testbench for phase 3. Thus, the results and simulations of phase 2 were used to show that most required features are functioning. It was demonstrated how:

- IR holds the instruction from the .MIF file
- PC is a dedicated module, and it will only increment once every time the increment PC signal is enabled
- PC works with all branch instructions
- All registers can hold data from the Bus, and the select and encode logic works
- R15_enable is also modified to enable from the select encode logic and another manual enable signal
- Instruction fetch (shown in control unit) and decode work

The decoder allowed the select and encode logic to determine which register needs to be enabled or selected, and then the r_enable, r_select, and ba_select signals allow the select and encode logic to output the proper enable and select signals. The C_sign_extended_Data variable holds the first 19 bits, usually the immediate value. For some instructions, bits 15-19 can be Rc. This can be all seen in the Appendix.

Unfortunately, during the fourth phase, the team could not successfully implement and simulate on the DE0 board. This meant not knowing how much of the chip area was consumed, how fast the chip could perform, the average cycle per instruction, and whether the chip could recognize any input and display output.

Discussion

Due to a lack of time, the completed processor was not successfully implemented onto the Cyclone board. As the team went further into the semester, this resulted in this incompletion due to a lack of time and a heavier course load. However, most features of the processor are fully functional. If more time were provided, it could be implemented onto a chip. The team plans to continue development to complete the processor and add more bonus features.

Conclusion & Future Steps

The design of the Datapath, memory subsystem, and control unit was completed successfully. As mentioned previously, a lack of time and a heavy courseload resulted in an incomplete processor. If the Phase 3 testbench and Phase 4 Cyclone chip implementation were to be accomplished in the near future, this would have changed the outcome of this project. Some recommendations for future improvements being investigated are optimizations to the code organization, processor efficiency, code documentation, and time management.

Appendices

ALU

AND Operation

logical_and.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

```
/* 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

```
/* 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

```
/* 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.
```

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).

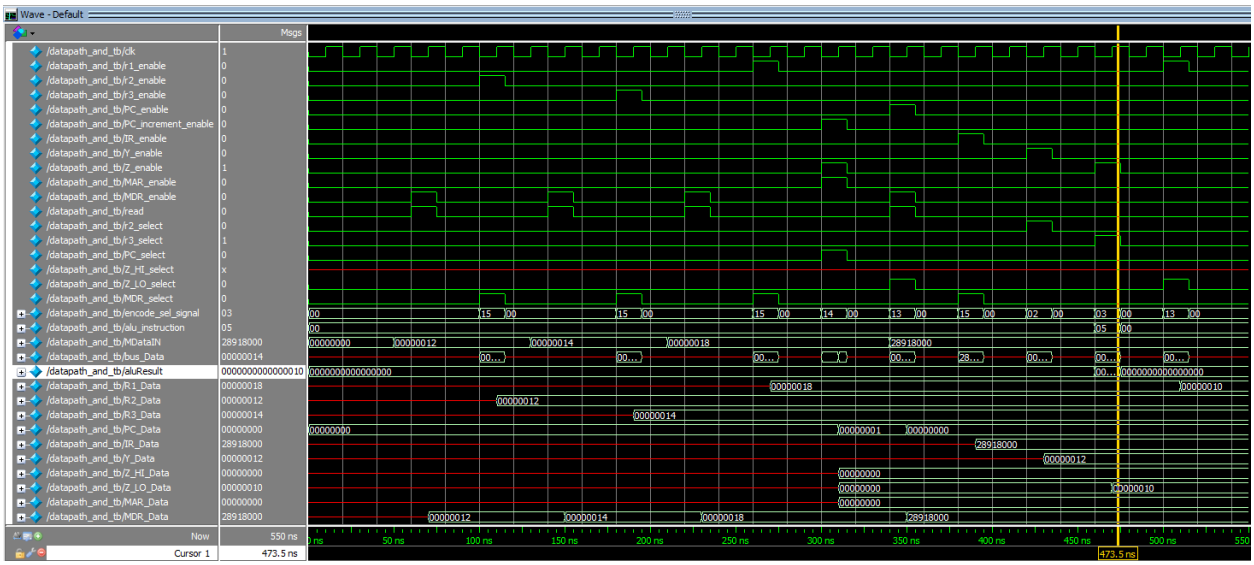
AND Operation Testbench Code

```
`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),
```

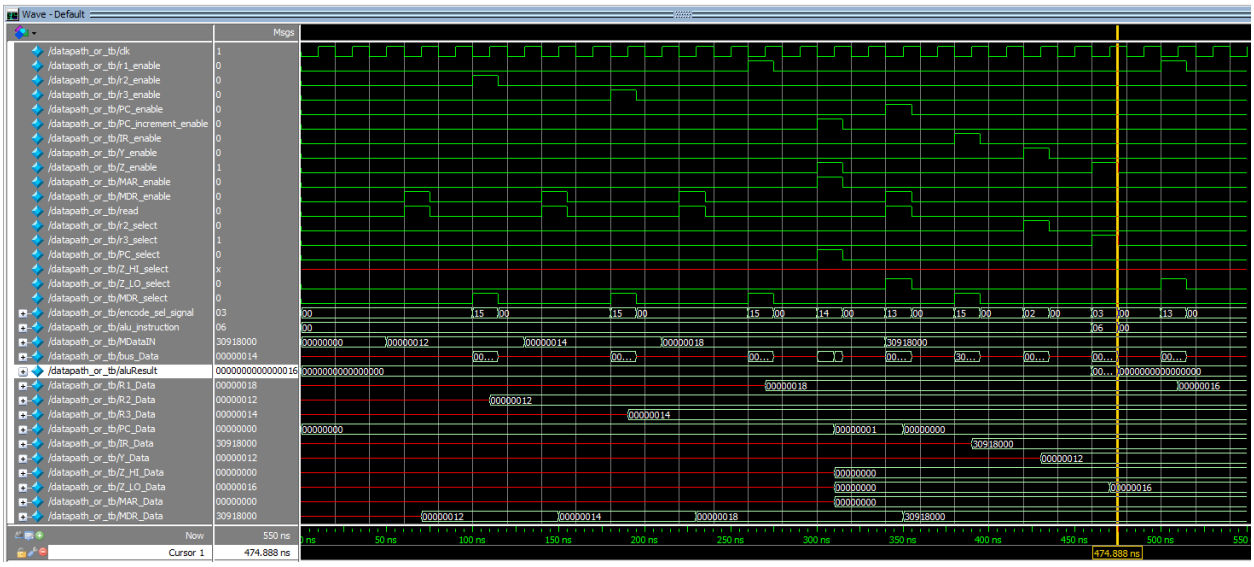
```
// Encoder Output Select Signals
.r2_select(r2_select), .r3_select(r3_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.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).

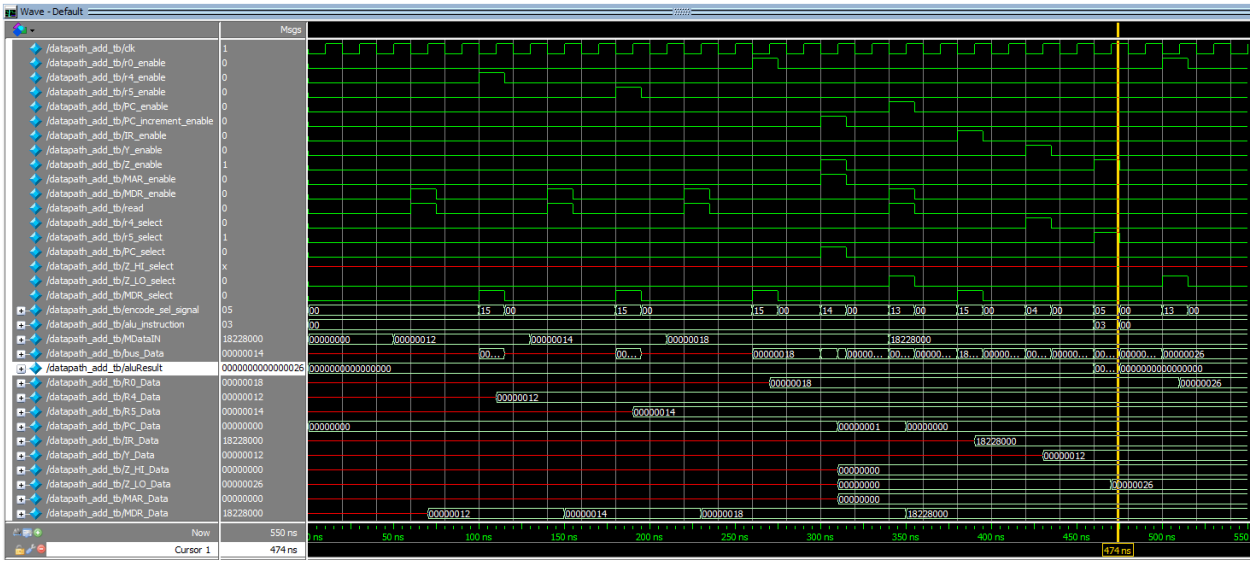
ADD Operation Testbench Code

```
`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),
```

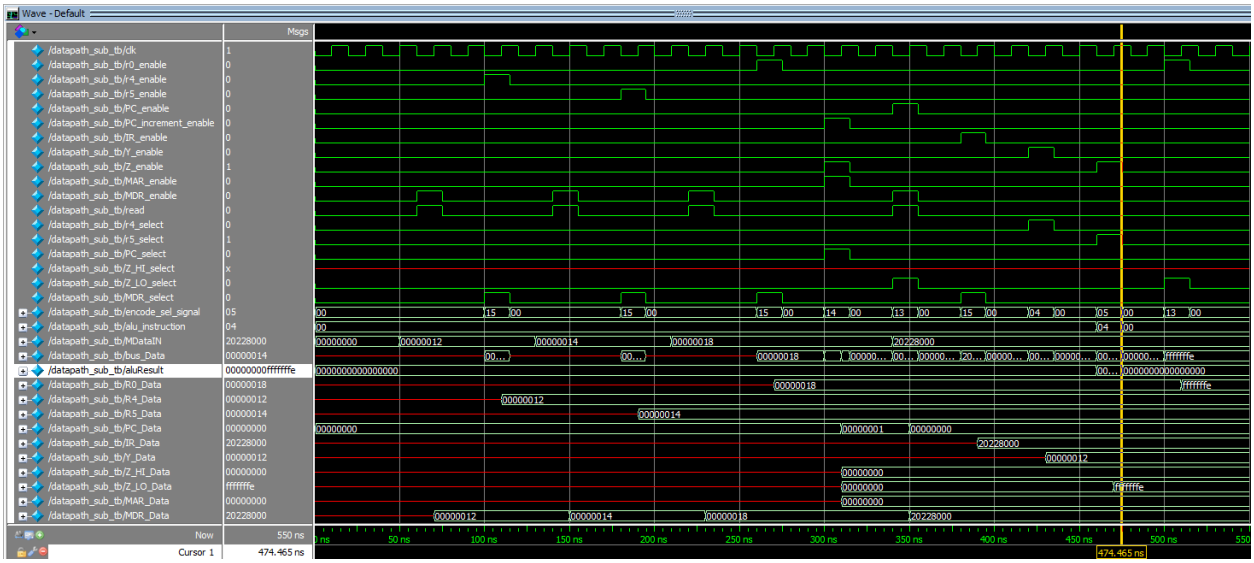
```
// Encoder Output Select Signals
.r4_select(r4_select), .r5_select(r5_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.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).

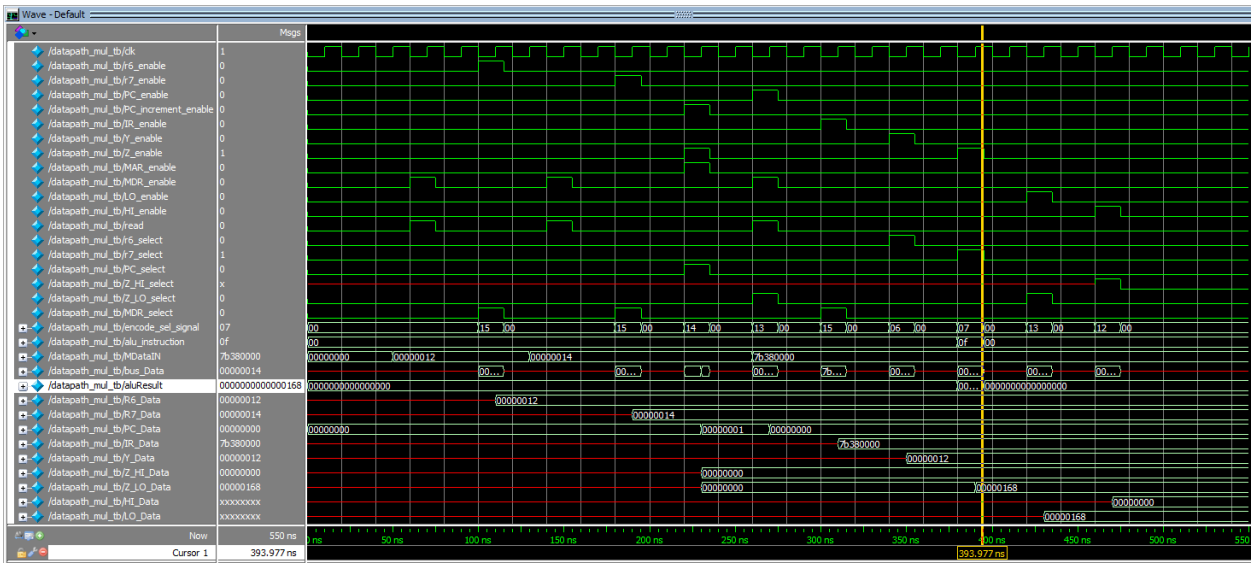
MUL Operation Testbench Code

```
`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),
```

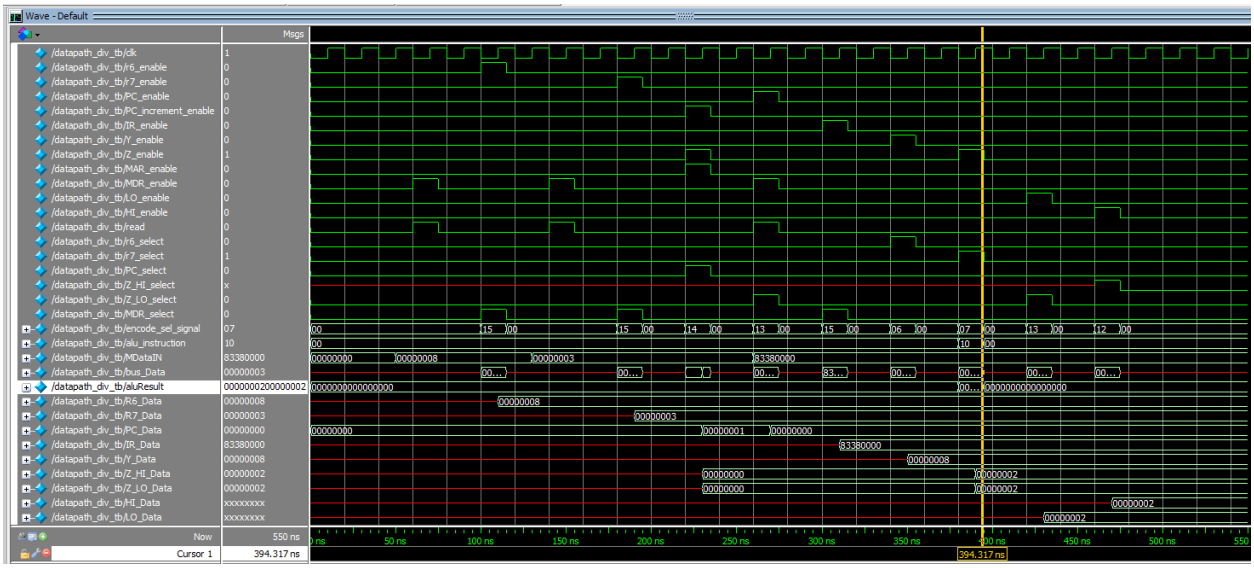
```
// Encoder Output Select Signals
.r6_select(r6_select), .r7_select(r7_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.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).

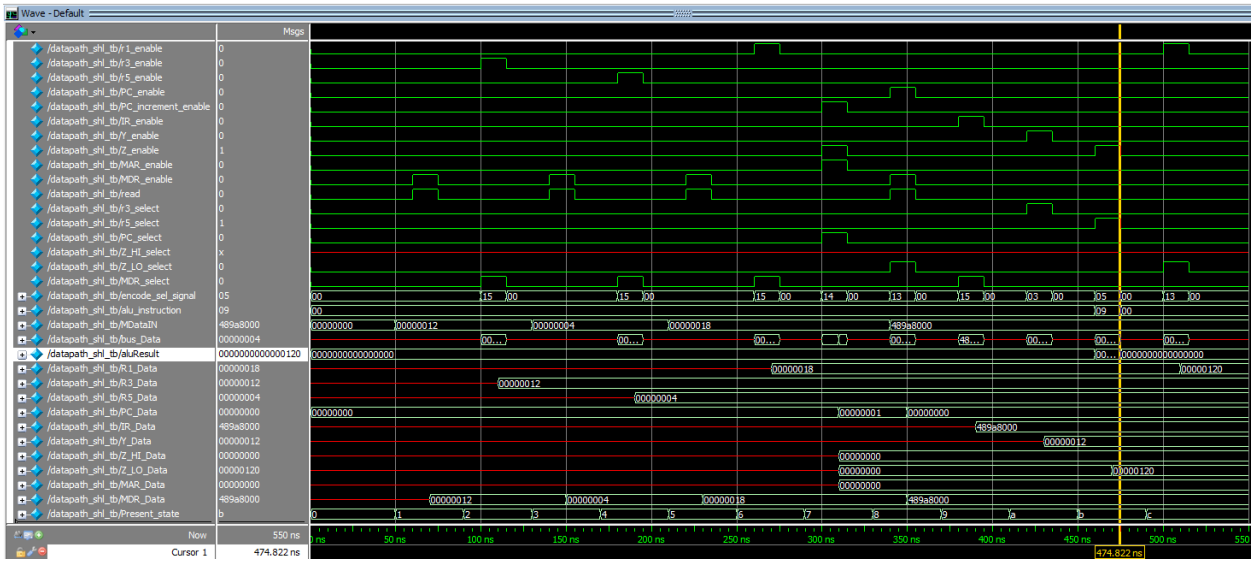
SHRA Operation Testbench Code

```
`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),
```

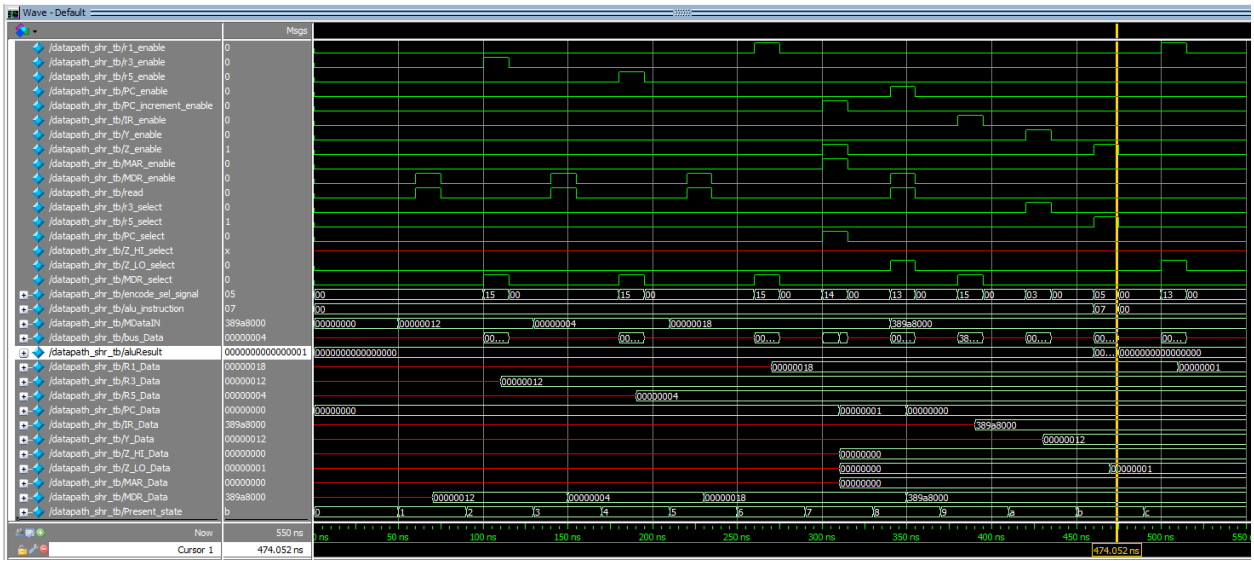
```
// Encoder Output Select Signals
.r3_select(r3_select), .r5_select(r5_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.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
```

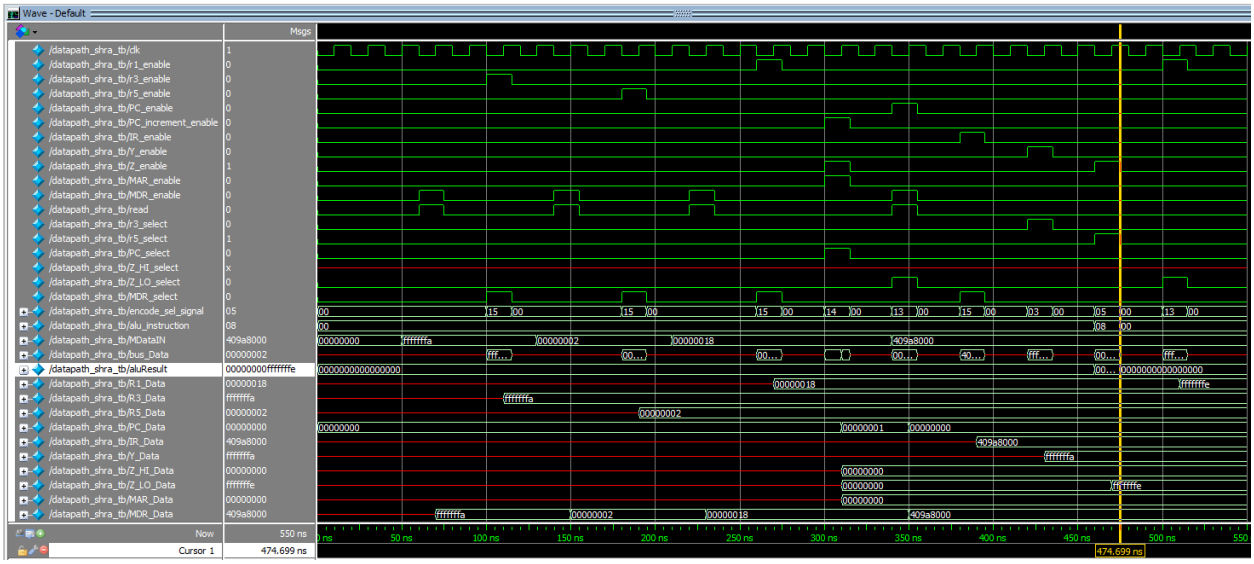
SHL Operation RTL Simulation



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).

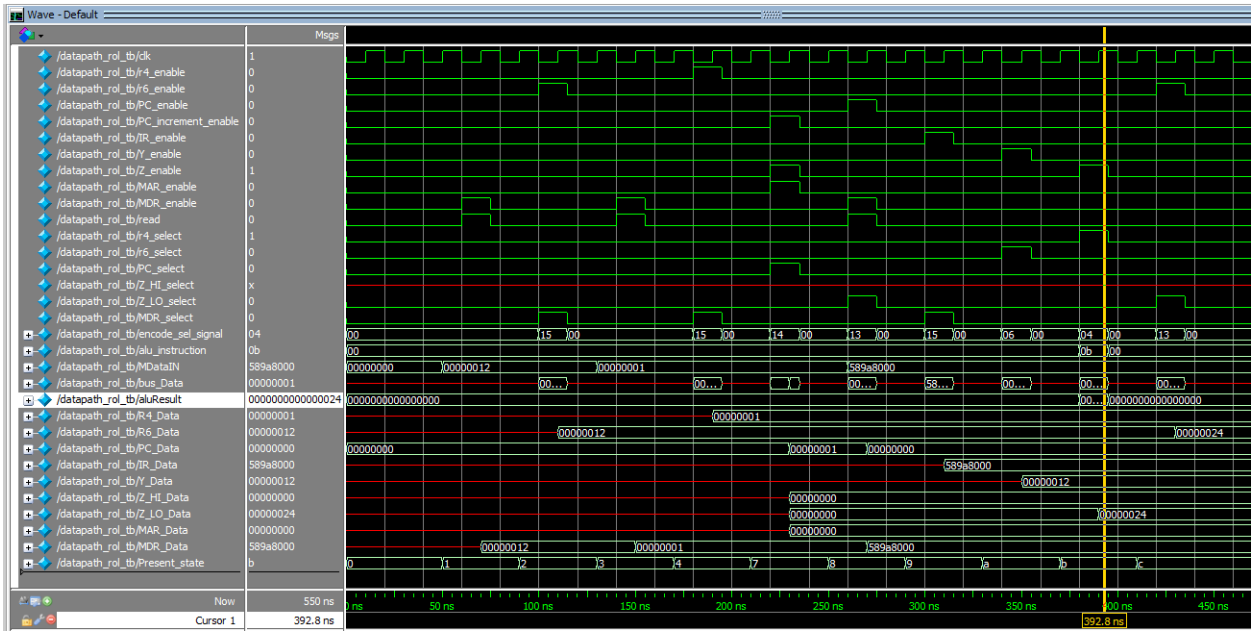
ROR Operation Testbench Code

```
`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),
```

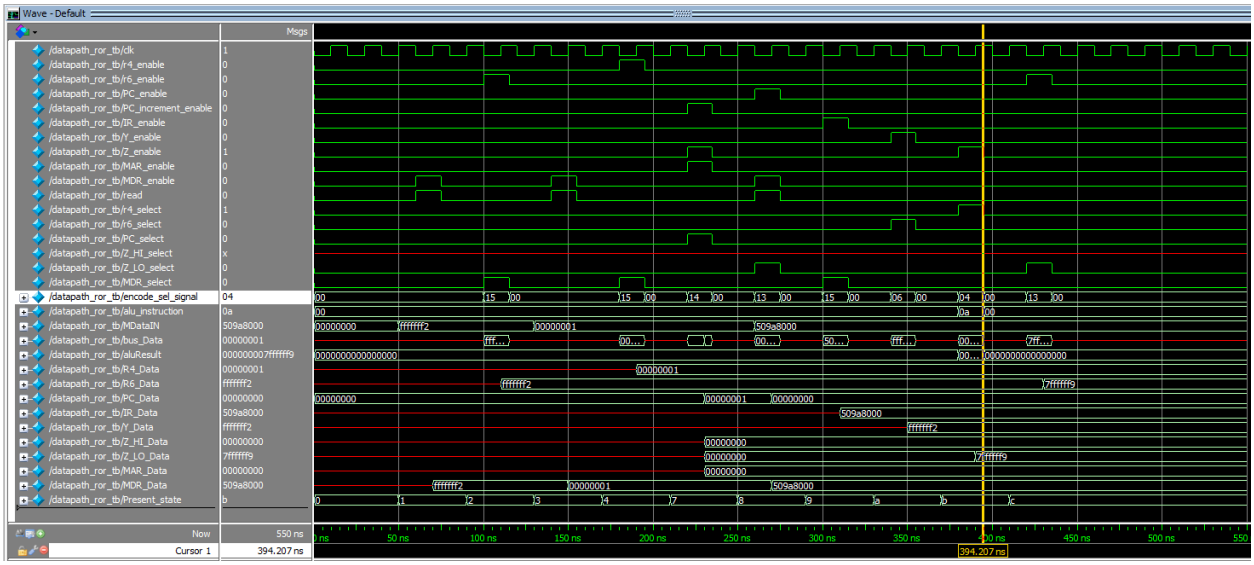
```
// Encoder Output Select Signals
.r4_select(r4_select), .r6_select(r6_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.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).

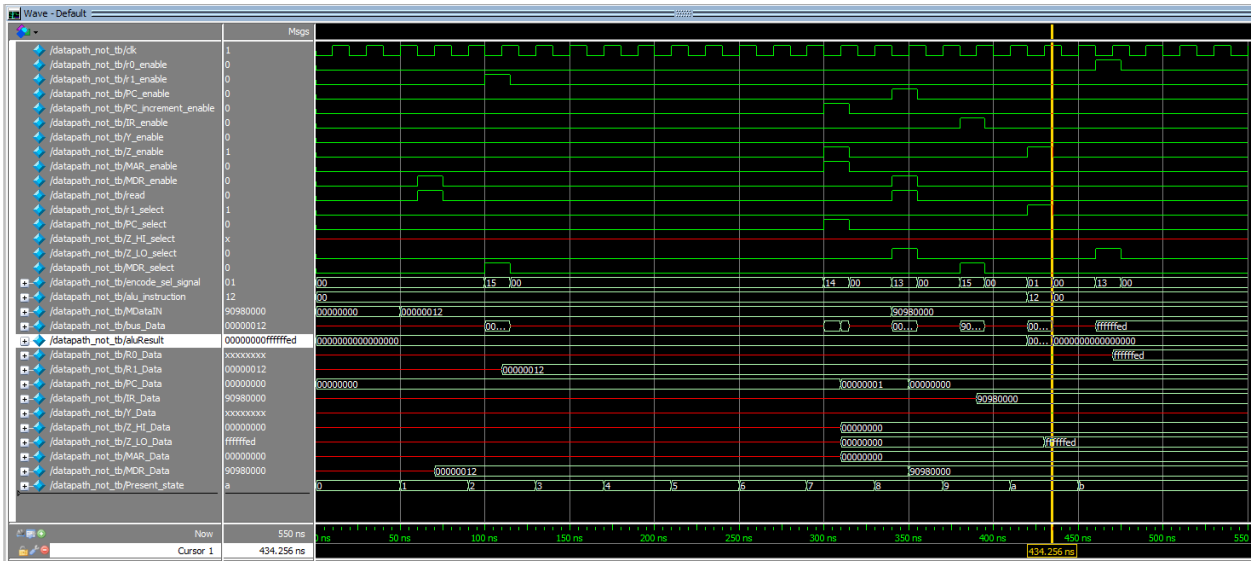
NOT Operation Testbench Code

```
`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),
```

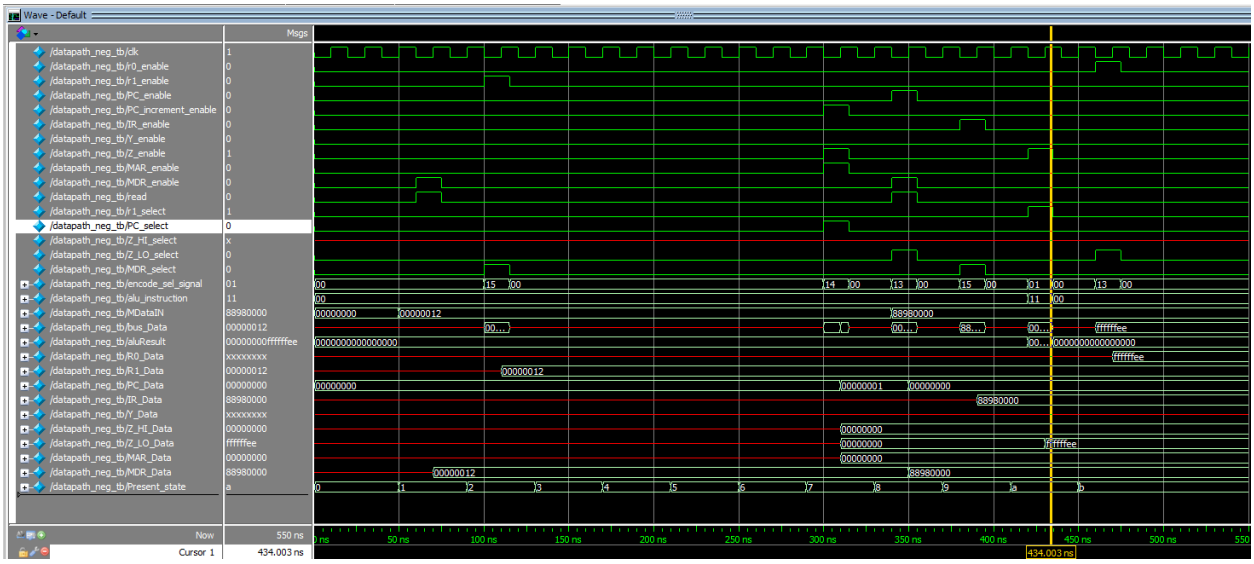
```
// Encoder Output Select Signals
.r1_select(r1_select),
.PC_select(PC_select),
.Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select),
.MDR_select(MDR_select),

.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



Registers

General Register

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

hdl > registers > md_register.v

```v
/* Representation of a memory data register with 2 to 1 multiplexer in Verilog */
module md_register(input clk, input clr, input enable, input read, input [31:0] MDataIN, input [31:0] bus_Data, 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 <= read ? MDataIN : bus_Data;
    end
endmodule // md_register end.
```

Revised Register 0

hdl > registers > R0_revised.v

```v
/* Representation of a register in Verilog HDL. */
/* Declared 3, 1-bit signals, 1, 32-bit D-input, and 1, 32-bit Q-output. */
/* Data type of each signal/input is a wire and output is a reg. */
module R0_revised(input clk, input clr, input enable, input ba_select, input [31:0] bus_Data, output wire [31:0] R0_Data);

    wire [31:0] registerOutput;

    register R0(clk, clr, enable, bus_Data, registerOutput);
    assign R0_Data = registerOutput & {32{!ba_select}};

endmodule // revised_register_R0 end.
```

Program Counter

hdl > registers > program_counter.v

```
1   module program_counter(
2       input wire clk, clr, enable, incPC,
3       input wire [31:0] PC_Input,
4       output wire [31:0] PC_Output
5   );
6
7   reg [31:0] Q;
8   reg incFlag; // flag to indicate that PC should be incremented on next clock edge
9
10  initial begin
11      Q <= 0;
12      incFlag = 1;
13  end
14
15  always @ (posedge clk) begin
16      if (clr) begin
17          Q <= 0;
18      end
19      else if (enable) begin
20          Q <= PC_Input;
21      end
22      else begin
23          if (incPC == 1 && incFlag == 1) begin
24              Q <= Q + 1;
25              incFlag <= 0;
26          end
27          else if (incPC == 0) begin
28              incFlag <= 1;
29          end
30      end
31  end
32
33  assign PC_Output = Q;
34
35  endmodule
36
```

Arithmetic Logic Unit

hdl > alu > alu.v

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

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

    parameter load = 5'b00000, loadImm = 5'b00001, store = 5'b00010, add = 5'b00011, sub = 5'b00100,
    AND = 5'b00101, OR = 5'b00110, right_shift = 5'b00111, arith_shift_right = 5'b01000, left_shift = 5'b01001,
    right_rotate = 5'b01010, left_rotate = 5'b01011, addImm = 5'b01100, AND_Imm = 5'b01101, OR_Imm = 5'b01110,
    mul = 5'b01111, div = 5'b10000, negate = 5'b10001, NOT = 5'b10010, branch = 5'b10011, jr = 5'b10100, jal = 5'b10101,
    in = 5'b10110, out = 5'b10111, mfhi = 5'b11000, mflo = 5'b11001;

    // 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);

```

```
49      // Arithmetic Right Shift Operation
50      wire [31:0] arithShiftRight_result;
51      arithmetic_shift_right arithShiftRightInstance(A, B, arithShiftRight_result);
52
53      // Left Rotate Operation
54      wire [31:0] leftRotate_result;
55      rotate_left leftRotateInstance(A, B, leftRotate_result);
56
57      // Right Rotate Operation
58      wire [31:0] rightRotate_result;
59      rotate_right rightRotateInstance(A, B, rightRotate_result);
60
61      // Negate Operation
62      wire [31:0] negate_result;
63      negate negateInstance(B, negate_result);
64
65      // Not Operation
66      wire [31:0] not_result;
67      logical_not notInstance(B, not_result);
68
```

```
69      // Select operation
70      always @(*) begin
71          case(opcode)
72              // Add Operation
73              add : begin
74                  result[31:0] <= sum_result[31:0];
75                  result[63:32] <= 32'd0;
76              end
77
78              // Subtract Operation
79              sub : begin
80                  result[31:0] <= difference_result[31:0];
81                  result[63:32] <= 32'd0;
82              end
83
84              // And Operation
85              AND, AND_Imm : begin
86                  result[31:0] <= and_result[31:0];
87                  result[63:32] <= 32'd0;
88              end
89
90              // Or Operation
91              OR, OR_Imm : begin
92                  result[31:0] <= or_result[31:0];
93                  result[63:32] <= 32'd0;
94              end
95
96              // Right Shift Operation
97              right_shift : begin
98                  result[31:0] <= rightShift_result[31:0];
99                  result[63:32] <= 32'd0;
100             end
101
102             // Arithmetic Right Shift Operation
103             arith_shift_right : begin
104                 result[31:0] <= arithShiftRight_result[31:0];
105                 result[63:32] <= 32'd0;
106             end
107
108             // Left Shift Operation
109             left_shift : begin
110                 result[31:0] <= leftShift_result[31:0];
111                 result[63:32] <= 32'd0;
112             end
113
```

```
114             // Right Rotate Operation
115             right_rotate : begin
116                 result[31:0] <= rightRotate_result[31:0];
117                 result[63:32] <= 32'd0;
118             end
119
120             // Left Rotate Operation
121             left_rotate : begin
122                 result[31:0] <= leftRotate_result[31:0];
123                 result[63:32] <= 32'd0;
124             end
125
126             // Multiply Operation
127             mul : begin
128                 result[63:0] <= product_result[63:0];
129             end
130
131             // Divide Operation
132             div : begin
133                 result[31:0] <= quotient_result[31:0];
134                 result[63:32] <= remainder_result[31:0];
135             end
136
137             // Negate Operation
138             negate : begin
139                 result[31:0] <= negate_result[31:0];
140                 result[63:32] <= 32'd0;
141             end
142
143             // Not Operation
144             NOT : begin
145                 result[31:0] <= not_result[31:0];
146                 result[63:32] <= 32'd0;
147             end
148
149             // Default Case
150             default : begin
151                 result[63:0] <= 64'd0;
152             end
153         endcase
154     end
155 endmodule
156
```

Bus System

Datapath

hdl > bus > datapath.v

```v
/* Representation of the datapath in Verilog HDL. */
module datapath(
    // CPU signals
    input wire clk, clr,
    input wire [31:0] input_Data,
    output wire [31:0] outport_Data
    );

    /* Enable Signals */
    // General Register Enable Signals
    wire R15_enable;
    wire manual_R15_enable;
    wire [15:0] register_enable;

    // Program Counter Enable Signals
    wire PC_enable, PC_increment_enable;

    // Instruction Register Enable Signal
    wire IR_enable;

    // CON Flip Flop Enable Signal
    wire con_enable;

    // ALU 'Y' Register Enable Signal
    wire Y_enable;

    // Z_HI and Z_LO Enable Signal
    wire Z_enable;

    // HI and LO Register Enable Signals
    wire HI_enable, LO_enable;

    // Memory Register Enable Signals
    wire MAR_enable, MDR_enable, read, write;

    // Outport Register Enable Signal
    wire outport_enable;

    /* Select Signals */
    // General Register Select Signals
    wire [15:0] register_select;

    // 32-to-5 Encoder Output
    wire [4:0] bus_select;

    // PC Register Select Signal
    wire PC_select;

```

```
49      // HI/LO Register Select Signal
50      wire HI_select, LO_select;
51
52      // Z_HI/Z_LO Register Select Signal
53      wire Z_HI_select, Z_LO_select;
54
55      // MDR Select Signal
56      wire MDR_select;
57
58      // Inport Select Signal
59      wire inport_select;
60
61      // C_Sign_Extended Select Signal
62      wire c_select;
63
64      /* Data Signals */
65      wire [31:0] bus_Data; // Data currently in the bus
66
67      wire [63:0] aluResult;
68      wire [4:0] alu_instruction; // ALU Opcode
69
70      // General Register Contents
71      wire [31:0] R0_Data, R1_Data, R2_Data, R3_Data, R4_Data, R5_Data, R6_Data, R7_Data,
72      R8_Data, R9_Data, R10_Data, R11_Data, R12_Data, R13_Data, R14_Data, R15_Data;
73
74      // Instruction Register and Program Counter Contents
75      wire [31:0] PC_Data, IR_Data;
76
77      // ALU Input Register 'Y' Contents
78      wire [31:0] Y_Data;
79
80      // ALU Output Register Contents
81      wire [31:0] Z_HI_Data, Z_LO_Data;
82      wire [31:0] HI_Data, LO_Data;
83
84      // RAM Memory Address Register (MAR) Contents
85      wire [8:0] MAR_Data;
86
87      // RAM Memory Data Register (MDR) Contents
88      wire [31:0] MDR_Data, MDataIN;
89
90      // C Sign Extended Data
91      wire [31:0] C_sign_ext_Data;
92
93      // Port Register Contents
94      wire [31:0] inport_Data;
95
```

hdl > bus > datapath.v

```
    /* Select and Encode Signals */
    wire Gra, Grb, Grc, r_enable, r_select, ba_select;

    /* CON FF Output */
    wire con_output;

    /* Bus Components */
    // 32-to-5 Encoder
    encoder encoder_instance(.register_select(register_select), .HI_select(HI_select), .LO_select(LO_select),
    .Z_HI_select(Z_HI_select), .Z_LO_select(Z_LO_select), .PC_select(PC_select), .MDR_select(MDR_select),
    .inport_select(inport_select), .c_select(c_select), .selectSignal(bus_select));

    // 32-to-1 Multiplexer
    multiplexer multiplexer_instance(.selectSignal(bus_select), .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));

    // General purpose registers r0 -> r15
    R0_revised r0 (.clk(clk), .clr(clr), .enable(register_enable[0]), .ba_select(ba_select), .bus_Data(bus_Data), .R0_Data(R0_Data));
    register r1 (.clk(clk), .clr(clr), .enable(register_enable[1]), .D(bus_Data), .Q(R1_Data));
    register r2 (.clk(clk), .clr(clr), .enable(register_enable[2]), .D(bus_Data), .Q(R2_Data));
    register r3 (.clk(clk), .clr(clr), .enable(register_enable[3]), .D(bus_Data), .Q(R3_Data));
    register r4 (.clk(clk), .clr(clr), .enable(register_enable[4]), .D(bus_Data), .Q(R4_Data));
    register r5 (.clk(clk), .clr(clr), .enable(register_enable[5]), .D(bus_Data), .Q(R5_Data));
    register r6 (.clk(clk), .clr(clr), .enable(register_enable[6]), .D(bus_Data), .Q(R6_Data));
    register r7 (.clk(clk), .clr(clr), .enable(register_enable[7]), .D(bus_Data), .Q(R7_Data));
    register r8 (.clk(clk), .clr(clr), .enable(register_enable[8]), .D(bus_Data), .Q(R8_Data));
    register r9 (.clk(clk), .clr(clr), .enable(register_enable[9]), .D(bus_Data), .Q(R9_Data));
    register r10 (.clk(clk), .clr(clr), .enable(register_enable[10]), .D(bus_Data), .Q(R10_Data));
    register r11 (.clk(clk), .clr(clr), .enable(register_enable[11]), .D(bus_Data), .Q(R11_Data));
    register r12 (.clk(clk), .clr(clr), .enable(register_enable[12]), .D(bus_Data), .Q(R12_Data));
    register r13 (.clk(clk), .clr(clr), .enable(register_enable[13]), .D(bus_Data), .Q(R13_Data));
    register r14 (.clk(clk), .clr(clr), .enable(register_enable[14]), .D(bus_Data), .Q(R14_Data));

    assign R15_enable = manual_R15_enable | register_enable[15];
    register r15 (.clk(clk), .clr(clr), .enable(R15_enable), .D(bus_Data), .Q(R15_Data));

    // ALU 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));

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

    // 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));

    // Memory Registers
    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), .read(read), .MDataIN(MDataIN), .bus_Data(bus_Data), .Q(MDR_Data));

    // Ram Instance
    ram ramInstance(.debug_port_01(debug_port_01), .debug_port_02(debug_port_02), .clk(clk), .read(read), .write(write), .data_in(MDR_Data), .address_in(MAR_Data), .data_out(MDataIN));

    select_encode_logic selInstance(.instruction(IR_Data), .Gra(Gra), .Grb(Grb), .Grc(Grc), .r_enable(r_enable), .r_select(r_select),
    .ba_select(ba_select), .C_sign_ext_Data(C_sign_ext_Data), .register_enable(register_enable), .register_select(register_select));

    con_ff conInstance(.bus_Data(bus_Data), .instruction(IR_Data), .con_enable(con_enable), .con_output(con_output));

    inport inportInstance(.clk(clk), .clr(clr), .input_Data(input_Data), .inport_Data(inport_Data));
    outport outportInstance(.clk(clk), .clr(clr), .enable(outport_enable), .bus_Data(bus_Data), .outport_Data(outport_Data));

    control_unit cuInstance(
        clk, reset, con_output, IR_Data, alu_instruction, read, write, Gra, Grb, Grc, r_enable, r_select, ba_select,
        PC_enable, PC_increment_enable, IR_enable, con_enable, Y_enable, Z_enable, HI_enable, LO_enable,
        MAR_enable, MDR_enable, outport_enable, manual_R15_enable,
        MDR_select, Z_HI_select, Z_LO_select, HI_select, LO_select, PC_select, inport_select, c_select);

endmodule // Datapath end.
```

Select Encode Logic

hdl > bus > select_encode_logic > select_encode_logic.v

```
module select_encode_logic(
    // Instruction Register Data
    input [31:0] instruction,

    // Select and Encode Logic Control Signals
    input Gra, Grb, Grc, r_enable, r_select, ba_select,

    // Array of Enable and Select Output Signals
    output [15:0] register_enable, register_select,

    // Sign Extension of Constant C
    output [31:0] C_sign_ext_Data);

    // Instruction Register Content Variables
    reg [3:0] Ra, Rb, Rc;

    // 4 to 16 Decoder Input/Output Variable
    reg [3:0] decoder_input;
    reg [15:0] decoder_out;

    always @ (instruction, Gra, Grb, Grc) begin
        // Assigning Values to Instruction Register Content Variables
        Ra = instruction [26:23];        // Bit 23 -> 26 of the instruction register is register a
        Rb = instruction [22:19];        // Bit 19 -> 22 of the instruction register is register b
        Rc = instruction [19:15];        // Bit 15 -> 19 of the instruction register is register c

        if (Gra) decoder_input = Ra;
        else if (Grb) decoder_input = Rb;
        else if (Grc) decoder_input = Rc;

        case (decoder_input)
            4'd0: decoder_out = 16'd1;
            4'd1: decoder_out = 16'd2;
            4'd2: decoder_out = 16'd4;
            4'd3: decoder_out = 16'd8;
            4'd4: decoder_out = 16'd16;
            4'd5: decoder_out = 16'd32;
            4'd6: decoder_out = 16'd64;
            4'd7: decoder_out = 16'd128;
            4'd8: decoder_out = 16'd256;
            4'd9: decoder_out = 16'd512;
            4'd10: decoder_out = 16'd1024;
            4'd11: decoder_out = 16'd2048;
            4'd12: decoder_out = 16'd4096;
            4'd13: decoder_out = 16'd8192;
            4'd14: decoder_out = 16'd16384;
            4'd15: decoder_out = 16'd32768;
        endcase
    end

    // Assigning Values to the Outputs
    assign register_enable = {16{r_enable}} & decoder_out;
    assign register_select = ({16{ba_select}} | {16{r_select}}) & decoder_out;
    assign C_sign_ext_Data = {{13{instruction[18]}}, instruction[18:0]};

endmodule
```

Encoder

hdl > bus > encoder.v

```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 [15:0] register_select, input HI_select, input LO_select, input Z_HI_select, input Z_LO_select,
input PC_select, input MDR_select, input inport_select, input c_select, output reg [4:0] selectSignal);

    /* While loop to update the selectSignal output wire. */
    always @* begin
        if (register_select[0]) selectSignal <= 5'b00000;
        else if (register_select[1]) selectSignal <= 5'b00001;
        else if (register_select[2]) selectSignal <= 5'b00010;
        else if (register_select[3]) selectSignal <= 5'b00011;
        else if (register_select[4]) selectSignal <= 5'b00100;
        else if (register_select[5]) selectSignal <= 5'b00101;
        else if (register_select[6]) selectSignal <= 5'b00110;
        else if (register_select[7]) selectSignal <= 5'b00111;
        else if (register_select[8]) selectSignal <= 5'b01000;
        else if (register_select[9]) selectSignal <= 5'b01001;
        else if (register_select[10]) selectSignal <= 5'b01010;
        else if (register_select[11]) selectSignal <= 5'b01011;
        else if (register_select[12]) selectSignal <= 5'b01100;
        else if (register_select[13]) selectSignal <= 5'b01101;
        else if (register_select[14]) selectSignal <= 5'b01110;
        else if (register_select[15]) selectSignal <= 5'b01111;
        else if (HI_select) selectSignal <= 5'b10000;
        else if (LO_select) selectSignal <= 5'b10001;
        else if (Z_HI_select) selectSignal <= 5'b10010;
        else if (Z_LO_select) selectSignal <= 5'b10011;
        else if (PC_select) selectSignal <= 5'b10100;
        else if (MDR_select) selectSignal <= 5'b10101;
        else if (inport_select) selectSignal <= 5'b10110;
        else if (c_select) selectSignal <= 5'b10111;
        else selectSignal <= 5'b00000;  // optional, to avoid latch.
    end
endmodule // Encoder end.
```

Multiplexer

hdl > bus > 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] selectSignal, 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 (selectSignal)
                5'b00000: muxOut <= muxIN_r0;
                5'b00001: muxOut <= muxIN_r1;
                5'b00010: muxOut <= muxIN_r2;
                5'b00011: muxOut <= muxIN_r3;
                5'b00100: muxOut <= muxIN_r4;
                5'b00101: muxOut <= muxIN_r5;
                5'b00110: muxOut <= muxIN_r6;
                5'b00111: muxOut <= muxIN_r7;
                5'b01000: muxOut <= muxIN_r8;
                5'b01001: muxOut <= muxIN_r9;
                5'b01010: muxOut <= muxIN_r10;
                5'b01011: muxOut <= muxIN_r11;
                5'b01100: muxOut <= muxIN_r12;
                5'b01101: muxOut <= muxIN_r13;
                5'b01110: muxOut <= muxIN_r14;
                5'b01111: muxOut <= muxIN_r15;
                5'b10000: muxOut <= muxIN_HI;
                5'b10001: muxOut <= muxIN_LO;
                5'b10010: muxOut <= muxIN_Z_HI;
                5'b10011: muxOut <= muxIN_Z_LO;
                5'b10100: muxOut <= muxIN_PC;
                5'b10101: muxOut <= muxIN_MDR;
                5'b10110: muxOut <= muxIN_inport;
                5'b10111: muxOut <= muxIN_C_sign_ext;
                default: muxOut <= 32'd0;
            endcase
        end
endmodule // Multiplexer end.

```

Con Flip Flop

hdl > con_ff > con_ff.v

```
/* This module determines whether the correct condition has been met to cause branching to take place in a conditional branch instruction. */
module con_ff(
    // Bus Input
    input [31:0] bus_Data,

    // Instruction Register Input
    input [31:0] instruction,

    // Enable Signal for the CON Flip Flop
    input con_enable,

    // Q Output Signal from the CON Flip Flop
    output con_output);

    reg [3:0] decoder_out;
    reg FF_Output;
    assign con_output = FF_Output;

    always @ (instruction[20:19]) begin
        case (instruction[20:19])
            2'b00: begin
                decoder_out = 4'b0001;
            end

            2'b01: begin
                decoder_out = 4'b0010;
            end

            2'b10: begin
                decoder_out = 4'b0100;
            end

            2'b11: begin
                decoder_out = 4'b1000;
            end
        endcase
    end

always @ (bus_Data && con_enable) begin
    if (bus_Data == 0 && decoder_out[0]) begin
        FF_Output = 1;
    end else if (bus_Data != 0 && decoder_out[1]) begin
        FF_Output = 1;
    end else if (bus_Data >= 0 && decoder_out[2]) begin
        FF_Output = 1;
    end else if (bus_Data[31] && decoder_out[3]) begin
        FF_Output = 1;
    end else begin
        FF_Output = 0;
    end
end

endmodule
```

Control Unit

hdl > control_unit > control_unit.v

```
1   `timescale 1ns/10ps
2   module control_unit(
3       // Control Unit Inputs
4       input wire clk, reset, con_output,
5       input wire [31:0] IR_Data,
6
7       // ALU Opcode
8       output reg [4:0] alu_instruction,
9
10      // RAM Read/Write Signals
11      output reg read, write,
12
13      // Select Encode Logic Signals
14      output reg Gra, Grb, Grc, r_enable, r_select, ba_select,
15
16      // Control Unit Enable Signal Outputs
17      output reg PC_enable, PC_increment_enable, IR_enable, con_enable, Y_enable, Z_enable, HI_enable, LO_enable,
18      MAR_enable, MDR_enable, outport_enable, manual_R15_enable,
19
20      // Control Unit Select Signal Outputs
21      output reg MDR_select, Z_HI_select, Z_LO_select, HI_select, LO_select, PC_select, inport_select, c_select);
22
23  parameter
24  // Initial State
25  reset_state = 8'b00000000,
26
27  // Instruction Fetch
28  fetch0 = 8'b00000001, fetch1 = 8'b00000010, fetch2= 8'b00000011,
29
30  // Add Instruction
31  add3 = 8'b00000100, add4= 8'b00000101, add5= 8'b00000110,
32
33  // Sub Instruction
34  sub3 = 8'b00000111, sub4 = 8'b00001000, sub5 = 8'b00001001,
35
36  // Multiply Instruction
37  mul3 = 8'b00001010, mul4 = 8'b00001011, mul5 = 8'b00001100, mul6 = 8'b00001101,
38
39  // Divide Instruction
40  div3 = 8'b00001110, div4 = 8'b00001111, div5 = 8'b00010000, div6 = 8'b00010001,
41
42  // Or Instruction
43  or3 = 8'b00010010, or4 = 8'b00010011, or5 = 8'b00010100, and3 = 8'b00010101,
44
45  // And Instruction
46  and4 = 8'b00010110, and5 = 8'b00010111,
47
```

hdl > control_unit > control_unit.v

```
47
48    // Shift Left Instruction
49    shl3 = 8'b00011000, shl4 = 8'b00011001, shl5 = 8'b00011010,
50
51    // Shift Right Instruction
52    shr3 = 8'b00011011, shr4 = 8'b00011100, shr5 = 8'b00011101,
53
54    // Rotate Left Instruction
55    rol3 = 8'b00011110, rol4 = 8'b00011111, rol5 = 8'b00100000,
56
57    // Rotate Right Instruction
58    ror3 = 8'b00100001, ror4 = 8'b00100010, ror5 = 8'b00100011,
59
60    // Negate Instruction
61    neg3 = 8'b00100100, neg4 = 8'b00100101, neg5 = 8'b00100110,
62
63    // Not Instruction
64    not3 = 8'b00100111, not4 = 8'b00101000, not5 = 8'b00101001,
65
66    // Load Instruction
67    ld3 = 8'b00101010, ld4 = 8'b00101011, ld5 = 8'b00101100, ld6 = 8'b00101101, ld7 = 8'b00101110,
68
69    // Load Immediate Instruction
70    ldi3 = 8'b00101111, ldi4 = 8'b00110000, ldi5 = 8'b00110001,
71
72    // Store Instruction
73    st3 = 8'b00110010, st4 = 8'b00110011, st5 = 8'b00110100, st6 = 8'b00110101,
74
75    // Add Immediate Instruction
76    addi3 = 8'b00110111, addi4 = 8'b00111000, addi5 = 8'b00111001,
77
78    // And Immediate Instruction
79    andi3 = 8'b00111010, andi4 = 8'b00111011, andi5 = 8'b00111100,
80
81    // Or Immediate Instruction
82    ori3 = 8'b00111101, ori4 = 8'b00111110, ori5 = 8'b00111111,
83
84    // Branch Instruction
85    br3 = 8'b01000000, br4 = 8'b01000001, br5 = 8'b01000010, br6 = 8'b01000011,
86
87    // Jump Instructions
88    jr3 = 8'b01000100,
89
90    jal3 = 8'b01000101, jal4 = 8'b01000110,
91
92    // Move from LO/HI Instruction
93    mfhi3 = 8'b01000111, mflo3 = 8'b01001000,
94
```

hdl > control_unit > control_unit.v

```
95   // In/Out Port Instruction
96   in3 = 8'b01001001, out3 = 8'b01001010,
97
98   // No Instruction
99   nop3 = 8'b01001011, halt3 = 8'b01001100,
100
101  // Shift Right Arithmetic Instruction
102  shra3 = 8'b01001101, shra4 = 8'b01001110, shra5 = 8'b01001111;
103
104  parameter ld = 5'b00000, ldi = 5'b00001, st = 5'b00010,
105  add = 5'b00011, sub = 5'b00100, AND = 5'b00101, OR = 5'b00110,
106  shr = 5'b00111, shra = 5'b01000, shl = 5'b01001, ror = 5'b01010, rol = 5'b01011,
107  mul = 5'b01111, div = 5'b10000, neg = 5'b10001, NOT = 5'b10010,
108  addi = 5'b01100, andi = 5'b01101, ori = 5'b01110,
109  br = 5'b10011, jr = 5'b10100, jal = 5'b10101,
110  in = 5'b10110, out = 5'b10111, mfhi = 5'b11000, mflo = 5'b11001,
111  nop = 5'b11010, halt = 5'b11011;
112
113  reg [7:0] present_state = reset_state;      // adjust the bit pattern based on the number of states
114
115  always @(posedge clk, posedge reset)  //con_ff finite state machine; if clk or reset rising-edge
116  begin
117     if (reset == 1'b1) present_state = reset_state;
118     else case (present_state)
119          reset_state : present_state = fetch0;
120          fetch0 : present_state = fetch1;
121          fetch1 : present_state = fetch2;
122          fetch2 : begin
123          case (IR_Data[31:27]) // inst. decoding based on the opcode to set the next state
124              5'b00011 : present_state = add3;
125              5'b00100 : present_state = sub3;
126              5'b01111 : present_state = mul3;
127              5'b10000 : present_state = div3;
128              5'b00111 : present_state = shr3;
129                  5'b01000 : present_state = shra3;
130              5'b01001 : present_state = shl3;
131              5'b01010 : present_state = ror3;
132              5'b01011 : present_state = rol3;
133              5'b00101 : present_state = and3;
134              5'b00110 : present_state = or3;
135              5'b10001 : present_state = neg3;
136              5'b10010 : present_state = not3;
137              5'b00000 : present_state = ld3;
138              5'b00001 : present_state = ldi3;
139              5'b00010 : present_state = st3;
140              5'b01100 : present_state = addi3;
141              5'b01100 : present_state = andi3;
142              5'b01110 : present_state = ori3;
143              5'b10011 : present_state = br3;
```

hdl > control_unit > control_unit.v

```
144                 5'b10100 : present_state = jr3;
145                 5'b10101 : present_state = jal3;
146                 5'b11000 : present_state = mfhi3;
147                 5'b11001 : present_state = mflo3;
148                 5'b10110 : present_state = in3;
149                 5'b10111 : present_state = out3;
150                 5'b11010 : present_state = nop3;
151                 5'b11011 : present_state = halt3;
152             endcase
153             end
154
155             add3 : present_state = add4;
156             add4 : present_state = add5;
157             add5 : present_state = fetch0;
158
159             addi3 : present_state = addi4;
160             addi4 : present_state = addi5;
161             addi5 : present_state = fetch0;
162
163             sub3 : present_state = sub4;
164             sub4 : present_state = sub5;
165             sub5 : present_state = fetch0;
166
167             mul3 : present_state = mul4;
168             mul4 : present_state = mul5;
169             mul5 : present_state = mul6;
170             mul6 : present_state = fetch0;
171
172             div3 : present_state = div4;
173             div4 : present_state = div5;
174             div5 : present_state = div6;
175             div6 : present_state = fetch0;
176
177             or3 : present_state = or4;
178             or4 : present_state = or5;
179             or5 : present_state = fetch0;
180
181             and3 : present_state = and4;
182             and4 : present_state = and5;
183             and5 : present_state = fetch0;
184
185             shl3 : present_state = shl4;
186             shl4 : present_state = shl5;
187             shl5 : present_state = fetch0;
188
189             shr3 : present_state = shr4;
190             shr4 : present_state = shr5;
191             shr5 : present_state = fetch0;
192
```

hdl > control_unit > control_unit.v

```
192
193             shra3 : present_state = shra4;
194         shra4 : present_state = shra5;
195         shra5 : present_state = fetch0;
196
197         rol3 : present_state = rol4;
198         rol4 : present_state = rol5;
199         rol5 : present_state = fetch0;
200
201         ror3 : present_state = ror4;
202         ror4 : present_state = ror5;
203         ror5 : present_state = fetch0;
204
205         neg3 : present_state = neg4;
206         neg4 : present_state = fetch0;
207
208         not3 : present_state = not4;
209         not4 : present_state = fetch0;
210
211         ld3 : present_state = ld4;
212         ld4 : present_state = ld5;
213         ld5 : present_state = ld6;
214         ld6 : present_state = ld7;
215         ld7 : present_state = fetch0;
216
217         ldi3 : present_state = ldi4;
218         ldi4 : present_state = ldi5;
219         ldi5 : present_state = fetch0;
220
221         st3 : present_state = st4;
222         st4 : present_state = st5;
223         st5 : present_state = st6;
224         st6 : present_state = fetch0;
225
226         andi3 : present_state = andi4;
227         andi4 : present_state = andi5;
228         andi5 : present_state = fetch0;
229
230         ori3 : present_state = ori4;
231         ori4 : present_state = ori5;
232         ori5 : present_state = fetch0;
233
234         jal3 : present_state = jal4;
235         jal4 : present_state = fetch0;
236
237         jr3 : present_state = fetch0;
238
```

```
238
239             br3 : present_state = br4;
240             br4 : present_state = br5;
241             br5 : present_state = br6;
242             br6 : present_state = fetch0;
243
244             out3 : present_state = fetch0;
245
246             in3 : present_state = fetch0;
247
248             mflo3 : present_state = fetch0;
249
250             mfhi3 : present_state = fetch0;
251
252             nop3 : present_state = fetch0;
253         endcase
254     end
255
256     always @(present_state) // do the job for each state
257     begin
258         case (present_state) // assert the required signals in each state
259         reset_state: begin
260         // RAM Read/Write Signals
261         read <= 0; write <= 0;
262
263         // Select Encode Logic Signals
264         Gra <= 0; Grb <= 0; Grc <= 0; r_enable <= 0; r_select <= 0; ba_select <= 0;
265
266         // Control Unit Enable Signal Outputs
267         PC_enable <= 0; PC_increment_enable <= 0; IR_enable <= 0; con_enable <= 0; Y_enable <= 0;
268         Z_enable <= 0; HI_enable <= 0; LO_enable <= 0; MAR_enable <= 0; MDR_enable <= 0; outport_enable <= 0;
269         manual_R15_enable <= 0;
270
271         // Control Unit Select Signal Outputs
272         MDR_select <= 0; Z_HI_select <= 0; Z_LO_select <= 0; HI_select <= 0; LO_select <= 0; PC_select <= 0;
273         inport_select <= 0; c_select <= 0;
274
275         // ALU Instruction
276         alu_instruction <= 0;
277
278         end
279
280         fetch0: begin
281             #10 PC_select <= 1; MAR_enable <= 1;
282             #75 PC_select <= 0; MAR_enable <= 0;
283         end
284
```

hdl > control_unit > control_unit.v

```
285     fetch1: begin
286         #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
287         #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
288     end
289
290     fetch2: begin
291         #10 MDR_select <= 1; IR_enable <= 1;
292         #75 MDR_select <= 0; IR_enable <= 0;
293     end
294
295     // Addition Operation
296     add3: begin
297         #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
298         #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
299     end
300
301     add4: begin
302         #10 Grc <= 1; r_select <= 1; alu_instruction <= add; Z_enable <= 1;
303         #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
304     end
305
306     add5: begin
307         #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
308         #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
309     end
310
311     // Subtraction Operation
312     sub3: begin
313         #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
314         #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
315     end
316
317     sub4: begin
318         #10 Grc <= 1; r_select <= 1; alu_instruction <= sub; Z_enable <= 1;
319         #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
320     end
321
322     sub5: begin
323         #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
324         #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
325     end
326
327     // Or Operation
328     or3: begin
329         #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
330         #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
331     end
332
```

hdl > control_unit > control_unit.v

```
333         or4: begin
334             #10 Grc <= 1; r_select <= 1; alu_instruction <= OR; Z_enable <= 1;
335             #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
336         end
337 
338         or5: begin
339             #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
340             #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
341         end
342 
343         // And Operation
344         and3: begin
345             #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
346             #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
347         end
348 
349         and4: begin
350             #10 Grc <= 1; r_select <= 1; alu_instruction <= AND; Z_enable <= 1;
351             #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
352         end
353 
354         and5: begin
355             #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
356             #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
357         end
358 
359         // SHR Operation
360         shr3: begin
361             #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
362             #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
363         end
364 
365         shr4: begin
366             #10 Grc <= 1; r_select <= 1; alu_instruction <= shr; Z_enable <= 1;
367             #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
368         end
369 
370         shr5: begin
371             #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
372             #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
373         end
374 
375         // SHL Operation
376         shl3: begin
377             #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
378             #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
379         end
380 
```

```
    shl4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= shl; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    shl5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // shra Operation
    shra3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    shra4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= shra; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    shra5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // ror Operation
    ror3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end

    ror4: begin
        #10 Grc <= 1; r_select <= 1; alu_instruction <= ror; Z_enable <= 1;
        #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
    end

    ror5: begin
        #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
        #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
    end

    // rol Operation
    rol3: begin
        #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
        #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
    end
```

hdl > control_unit > control_unit.v

```
428
429     rol4: begin
430         #10 Grc <= 1; r_select <= 1; alu_instruction <= rol; Z_enable <= 1;
431         #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
432     end
433
434     rol5: begin
435         #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
436         #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
437     end
438
439    // Multiplication Operation
440     mul3: begin
441         #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
442         #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
443     end
444
445     mul4: begin
446         #10 Grc <= 1; r_select <= 1; alu_instruction <= mul; Z_enable <= 1;
447         #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
448     end
449
450     mul5: begin
451         #10 Z_LO_select <= 1; LO_enable <= 1;
452         #75 Z_LO_select <= 0; LO_enable <= 0;
453     end
454
455     mul6: begin
456         #10 Z_HI_select <= 1; HI_enable <= 1;
457         #75 Z_HI_select <= 0; HI_enable <= 0;
458     end
459
460     // Division Operation
461     div3: begin
462         #10 Grb <= 1; r_select <= 1; Y_enable <= 1;
463         #75 Grb <= 0; r_select <= 0; Y_enable <= 0;
464     end
465
466     div4: begin
467         #10 Grc <= 1; r_select <= 1; alu_instruction <= div; Z_enable <= 1;
468         #75 Grc <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
469     end
470
471     div5: begin
472         #10 Z_LO_select <= 1; LO_enable <= 1;
473         #75 Z_LO_select <= 0; LO_enable <= 0;
474     end
475
```

hdl > control_unit > control_unit.v

```
476     div6: begin
477         #10 Z_HI_select <= 1; HI_enable <= 1;
478         #75 Z_HI_select <= 0; HI_enable <= 0;
479     end
480 
481     // Not Operation
482     not3: begin
483         #10 Grb <= 1; r_select <= 1; alu_instruction <= NOT; Y_enable <= 1;
484         #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Y_enable <= 0;
485     end
486 
487     not4: begin
488         #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
489         #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
490     end
491 
492     // Negate Operation
493     neg3: begin
494         #10 Grb <= 1; r_select <= 1; alu_instruction <= neg; Y_enable <= 1;
495         #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Y_enable <= 0;
496     end
497 
498     neg4: begin
499         #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
500         #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
501     end
502 
503     // And Immediate Operation
504     andi3: begin
505         #10 c_select <= 1; Y_enable <= 1;
506         #75 c_select <= 0; Y_enable <= 0;
507     end
508 
509     andi4: begin
510         #10 Grb <= 1; r_select <= 1; alu_instruction <= AND; Z_enable <= 1;
511         #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
512     end
513 
514     andi5: begin
515         #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
516         #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
517     end
518 
519     // Add Immediate Operation
520     addi3: begin
521         #10 c_select <= 1; Y_enable <= 1;
522         #75 c_select <= 0; Y_enable <= 0;
523     end
```

```
525         addi4: begin
526             #10 Grb <= 1; r_select <= 1; alu_instruction <= add; Z_enable <= 1;
527             #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
528         end
529 
530         addi5: begin
531             #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
532             #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
533         end
534 
535         // Or Immediate Operation
536         ori3: begin
537             #10 c_select <= 1; Y_enable <= 1;
538             #75 c_select <= 0; Y_enable <= 0;
539         end
540 
541         ori4: begin
542             #10 Grb <= 1; r_select <= 1; alu_instruction <= OR; Z_enable <= 1;
543             #75 Grb <= 0; r_select <= 0; alu_instruction <= 0; Z_enable <= 0;
544         end
545 
546         ori5: begin
547             #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
548             #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
549         end
550 
551         // Load Operation
552         ld3: begin
553             #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
554             #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
555         end
556 
557         ld4: begin
558             #10 c_select <= 1; alu_instruction <= add; Z_enable <= 1;
559             #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
560         end
561 
562         ld5: begin
563             #10 Z_LO_select <= 1; MAR_enable <= 1;
564             #75 Z_LO_select <= 0; MAR_enable <= 0;
565         end
566 
567         ld6: begin
568             #10 read <= 1; MDR_enable <= 1;
569             #75 read <= 0; MDR_enable <= 0;
570         end
571 
```

```
572     ld7: begin
573         #10 MDR_select <= 1; Gra <= 1; r_enable <= 1;
574         #75 MDR_select <= 0; Gra <= 0; r_enable <= 0;
575     end
576 
577     // Load Immediate Operation
578     ldi3: begin
579         #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
580         #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
581     end
582 
583     ldi4: begin
584         #10 c_select <= 1; alu_instruction <= add; Z_enable <= 1;
585         #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
586     end
587 
588     ldi5: begin
589         #10 Z_LO_select <= 1; Gra <= 1; r_enable <= 1;
590         #75 Z_LO_select <= 0; Gra <= 0; r_enable <= 0;
591     end
592 
593     //Store Operation
594     st3: begin
595         #10 Grb <= 1; ba_select <= 1; Y_enable <= 1;
596         #75 Grb <= 0; ba_select <= 0; Y_enable <= 0;
597     end
598 
599     st4: begin
600         #10 c_select <= 1; alu_instruction <= add; Z_enable <= 1;
601         #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
602     end
603 
604     st5: begin
605         #10 Z_LO_select <= 1; MAR_enable <= 1;
606         #75 Z_LO_select <= 0; MAR_enable <= 0;
607     end
608 
609     st6: begin
610         #10 write <= 1; MDR_enable <= 1; Gra <= 1; r_select <= 1;
611         #75 write <= 0; MDR_enable <= 0; Gra <= 0; r_select <= 0;
612     end
613 
614     // Jump Register Operation
615     jr3: begin
616         #10 Gra <= 1; r_select <= 1; PC_enable <= 1;
617         #75 Gra <= 0; r_select <= 0; PC_enable <= 0;
618     end
619 
```

```
620     //Jump and Link Operation
621     jal3: begin
622         #10 manual_R15_enable <= 1; PC_select <= 1;
623         #75 manual_R15_enable <= 0; PC_select <= 0;
624     end
625 
626     jal4: begin
627         #10 Gra <= 1; r_select <= 1; PC_enable <= 1;
628         #75 Gra <= 0; r_select <= 0; PC_enable <= 0;
629     end
630 
631     // Move from Hi Register Operation
632     mfhi3: begin
633         #10 Gra <= 1; r_enable <= 1; HI_select <= 1;
634         #75 Gra <= 0; r_enable <= 0; HI_select <= 0;
635     end
636 
637     // Move from Lo Register Operation
638     mflo3: begin
639         #10 Gra <= 1; r_enable <= 1; LO_select <= 1;
640         #75 Gra <= 0; r_enable <= 0; LO_select <= 0;
641     end
642 
643     // Inputting Operation
644     in3: begin
645         #10 Gra <= 1; r_enable <= 1; inport_select <= 1;
646         #75 Gra <= 0; r_enable <= 0; inport_select <= 0;
647     end
648 
649     // Outputting Operation
650     out3: begin
651         #10 Gra <= 1; r_select <= 1; outport_enable <= 1;
652         #75 Gra <= 0; r_select <= 0; outport_enable <= 0;
653     end
654 
655     // Branch Operation
656     br3: begin
657         #10 Gra <= 1; r_select <= 1; con_enable <= 1;
658         #75 Gra <= 0; r_select <= 0; con_enable <= 0;
659     end
660 
661     br4: begin
662         #10 PC_select <= 1; Y_enable <= 1;
663         #75 PC_select <= 0; Y_enable <= 0;
664     end
665 
```

```
666         br5: begin
667             #10 c_select <= 1; alu_instruction <= add; Z_enable <= 1;
668             #75 c_select <= 0; alu_instruction <= 0; Z_enable <= 0;
669         end
670
671         br6: begin
672             #10 Z_LO_select <= 1; PC_enable <= con_output;
673             #75 Z_LO_select <= 0; PC_enable <= 0;
674         end
675
676         // No Operation
677         nop3: begin
678             MDR_select <= 0; IR_enable <= 0;
679         end
680
681         // Halt Operation (Stops instruction execution)
682         halt3: begin
683             MDR_select <= 0; IR_enable <= 0;
684         end
685
686         //Any other input would be default to do nothing
687         default: begin
688         end
689         endcase
690     end
691     endmodule
692
```

Memory Subsystem

Ram

hdl > memory_subsystem > ram.v

```v
module ram(
    input clk,          // RAM enable control signal (active high)
    input read,                 // Read control signal (active high)
    input write,                // Write control signal (active high)
    input [31:0] data_in,       // Input data
    input [8:0] address_in,     // Address input
    output wire [31:0] data_out, // Output data

    output [31:0] debug_port_01,
    output [31:0] debug_port_02
);

reg [31:0] mem [511:0]; // Memory array
reg [31:0] tempData;    // Temporary data storage

initial begin
    $readmemh("ram.mif", mem);
end

always @(posedge clk) begin
    if (write) begin
        mem[address_in] <= data_in; // Write data to memory location
    end
    if (read) begin
        tempData <= mem[address_in]; // Read data from memory location
    end
end

assign data_out = tempData;

assign debug_port_01 = mem [144];
assign debug_port_02 = mem [247];

endmodule

```

Ram.mif File

```
1   08800002
2   08080000
3   01000068
4   0917FFFC
5   00900001
6   09800069
7   99980004
8   09980002
9   039FFFFD
10  D0000000
11  9B900002
12  09000005
13  09880002
14  19918000
15  63B80002
16  8BB80000
17  93B80000
18  6BB8000F
19  50880000
20  7388001C
21  43B80000
22  39180000
23  11000052
24  59100000
25  31180000
26  28908000
27  11880060
28  21918000
29  48900000
30  0A000006
31  0A800032
32  7AA00000
33  C3800000
34  CB000000
35  82A00000
36  0C27FFFF
37  0CAFFFED
38  0D300000
39  0DB80000
40  AD000000
41  D8000000
42  ffffffff
43  ffffffff
44  ffffffff
45  ffffffff
46  ffffffff
47  ffffffff
48  ffffffff
49  ffffffff
50  ffffffff
51  ffffffff
52  ffffffff
53  ffffffff
54  ffffffff
55  ffffffff
56  ffffffff
57  ffffffff
58  ffffffff
59  ffffffff
60  ffffffff
61  ffffffff
62  ffffffff
63  ffffffff
64  ffffffff
65  ffffffff
66  ffffffff
67  ffffffff
68  ffffffff
69  ffffffff
70  ffffffff
71  ffffffff
72  ffffffff
73  ffffffff
74  ffffffff
75  ffffffff
76  ffffffff
77  ffffffff
78  ffffffff
79  ffffffff
80  ffffffff
81  ffffffff
82  ffffffff
83  00000026
84  ffffffff
85  ffffffff
86  ffffffff
87  ffffffff
88  ffffffff
89  ffffffff
90  ffffffff
91  ffffffff
92  ffffffff
93  ffffffff
94  ffffffff
95  ffffffff
96  ffffffff
97  ffffffff
98  ffffffff
99  ffffffff
100 ffffffff
101 ffffffff
102 ffffffff
103 ffffffff
104 ffffffff
105 00000055
106 ffffffff
107 ffffffff
108 ffffffff
109 ffffffff
110 ffffffff
111 ffffffff
112 ffffffff
113 ffffffff
114 ffffffff
115 ffffffff
116 ffffffff
117 ffffffff
118 ffffffff
119 ffffffff
120 ffffffff
121 ffffffff
122 ffffffff
123 ffffffff
124 ffffffff
125 ffffffff
126 ffffffff
127 ffffffff
128 ffffffff
129 ffffffff
130 ffffffff
131 ffffffff
132 ffffffff
133 ffffffff
134 ffffffff
135 ffffffff
136 ffffffff
137 ffffffff
138 ffffffff
139 ffffffff
140 ffffffff
141 ffffffff
142 ffffffff
143 ffffffff
144 ffffffff
145 ffffffff
146 ffffffff
147 ffffffff
148 ffffffff
149 ffffffff
150 ffffffff
151 ffffffff
152 ffffffff
153 ffffffff
154 ffffffff
155 ffffffff
156 ffffffff
157 ffffffff
158 ffffffff
159 ffffffff
160 ffffffff
161 ffffffff
162 ffffffff
163 ffffffff
164 ffffffff
165 ffffffff
166 ffffffff
167 ffffffff
168 ffffffff
169 ffffffff
170 ffffffff
171 ffffffff
172 ffffffff
173 ffffffff
174 ffffffff
175 ffffffff
176 ffffffff
177 ffffffff
178 ffffffff
179 ffffffff
180 ffffffff
181 ffffffff
182 ffffffff
183 ffffffff
184 ffffffff
185 ffffffff
186 ffffffff
187 ffffffff
188 ffffffff
189 ffffffff
190 ffffffff
191 ffffffff
192 ffffffff
193 ffffffff
194 ffffffff
195 ffffffff
196 ffffffff
197 ffffffff
198 ffffffff
199 ffffffff
200 ffffffff
201 ffffffff
202 ffffffff
203 ffffffff
204 ffffffff
205 ffffffff
206 ffffffff
207 ffffffff
208 ffffffff
209 ffffffff
210 ffffffff
211 ffffffff
212 ffffffff
213 ffffffff
214 ffffffff
215 ffffffff
216 ffffffff
217 ffffffff
```

```
218 ffffffff
219 ffffffff
220 ffffffff
221 ffffffff
222 ffffffff
223 ffffffff
224 ffffffff
225 ffffffff
226 ffffffff
227 ffffffff
228 ffffffff
229 ffffffff
230 ffffffff
231 ffffffff
232 ffffffff
233 ffffffff
234 ffffffff
235 ffffffff
236 ffffffff
237 ffffffff
238 ffffffff
239 ffffffff
240 ffffffff
241 ffffffff
242 ffffffff
243 ffffffff
244 ffffffff
245 ffffffff
246 ffffffff
247 ffffffff
248 ffffffff
249 ffffffff
250 ffffffff
251 ffffffff
252 ffffffff
253 ffffffff
254 ffffffff
255 ffffffff
256 ffffffff
257 ffffffff
258 ffffffff
259 ffffffff
260 ffffffff
261 ffffffff
262 ffffffff
263 ffffffff
264 ffffffff
265 ffffffff
266 ffffffff
267 ffffffff
268 ffffffff
269 ffffffff
270 ffffffff
271 ffffffff
272 ffffffff
273 ffffffff
274 ffffffff
275 ffffffff
276 ffffffff
277 ffffffff
278 ffffffff
279 ffffffff
280 ffffffff
281 ffffffff
282 ffffffff
283 ffffffff
284 ffffffff
285 ffffffff
286 ffffffff
287 ffffffff
288 ffffffff
289 ffffffff
290 ffffffff
291 ffffffff
292 ffffffff
293 ffffffff
294 ffffffff
295 ffffffff
296 ffffffff
297 ffffffff
298 ffffffff
299 ffffffff
300 ffffffff
301 1EC50000
302 264D8000
303 26EE0000
304 A7800000
305 ffffffff
306 ffffffff
307 ffffffff
308 ffffffff
309 ffffffff
310 ffffffff
311 ffffffff
312 ffffffff
313 ffffffff
314 ffffffff
315 ffffffff
316 ffffffff
317 ffffffff
318 ffffffff
319 ffffffff
320 ffffffff
321 ffffffff
322 ffffffff
323 ffffffff
324 ffffffff
325 ffffffff
326 ffffffff
327 ffffffff
328 ffffffff
329 ffffffff
330 ffffffff
331 ffffffff
332 ffffffff
333 ffffffff
334 ffffffff
335 ffffffff
336 ffffffff
337 ffffffff
338 ffffffff
339 ffffffff
340 ffffffff
341 ffffffff
342 ffffffff
343 ffffffff
344 ffffffff
345 ffffffff
346 ffffffff
347 ffffffff
348 ffffffff
349 ffffffff
350 ffffffff
351 ffffffff
352 ffffffff
353 ffffffff
354 ffffffff
355 ffffffff
356 ffffffff
357 ffffffff
358 ffffffff
359 ffffffff
360 ffffffff
361 ffffffff
362 ffffffff
363 ffffffff
364 ffffffff
365 ffffffff
366 ffffffff
367 ffffffff
368 ffffffff
369 ffffffff
370 ffffffff
371 ffffffff
372 ffffffff
373 ffffffff
374 ffffffff
375 ffffffff
376 ffffffff
377 ffffffff
378 ffffffff
379 ffffffff
380 ffffffff
381 ffffffff
382 ffffffff
383 ffffffff
384 ffffffff
385 ffffffff
386 ffffffff
387 ffffffff
388 ffffffff
389 ffffffff
390 ffffffff
391 ffffffff
392 ffffffff
393 ffffffff
394 ffffffff
395 ffffffff
396 ffffffff
397 ffffffff
398 ffffffff
399 ffffffff
400 ffffffff
401 ffffffff
402 ffffffff
403 ffffffff
404 ffffffff
405 ffffffff
406 ffffffff
407 ffffffff
408 ffffffff
409 ffffffff
410 ffffffff
411 ffffffff
412 ffffffff
413 ffffffff
414 ffffffff
415 ffffffff
416 ffffffff
417 ffffffff
418 ffffffff
419 ffffffff
420 ffffffff
421 ffffffff
422 ffffffff
423 ffffffff
424 ffffffff
425 ffffffff
426 ffffffff
427 ffffffff
428 ffffffff
429 ffffffff
430 ffffffff
431 ffffffff
432 ffffffff
433 ffffffff
434 ffffffff
435 ffffffff
436 ffffffff
437 ffffffff
438 ffffffff
439 ffffffff
440 ffffffff
441 ffffffff
442 ffffffff
443 ffffffff
444 ffffffff
445 ffffffff
446 ffffffff
447 ffffffff
448 ffffffff
449 ffffffff
```

```
450  ffffffff
451  ffffffff
452  ffffffff
453  ffffffff
454  ffffffff
455  ffffffff
456  ffffffff
457  ffffffff
458  ffffffff
459  ffffffff
460  ffffffff
461  ffffffff
462  ffffffff
463  ffffffff
464  ffffffff
465  ffffffff
466  ffffffff
467  ffffffff
468  ffffffff
469  ffffffff
470  ffffffff
471  ffffffff
472  ffffffff
473  ffffffff
474  ffffffff
475  ffffffff
476  ffffffff
477  ffffffff
478  ffffffff
479  ffffffff
480  ffffffff
481  ffffffff
482  ffffffff
483  ffffffff
484  ffffffff
485  ffffffff
486  ffffffff
487  ffffffff
488  ffffffff
489  ffffffff
490  ffffffff
491  ffffffff
492  ffffffff
493  ffffffff
494  ffffffff
495  ffffffff
496  ffffffff
497  ffffffff
498  ffffffff
499  ffffffff
500  ffffffff
501  ffffffff
502  ffffffff
503  ffffffff
504  ffffffff
505  ffffffff
506  ffffffff
507  ffffffff
508  ffffffff
509  ffffffff
510  ffffffff
511  ffffffff
512  ffffffff
```

Ports

Inport

hdl > ports > inport.v

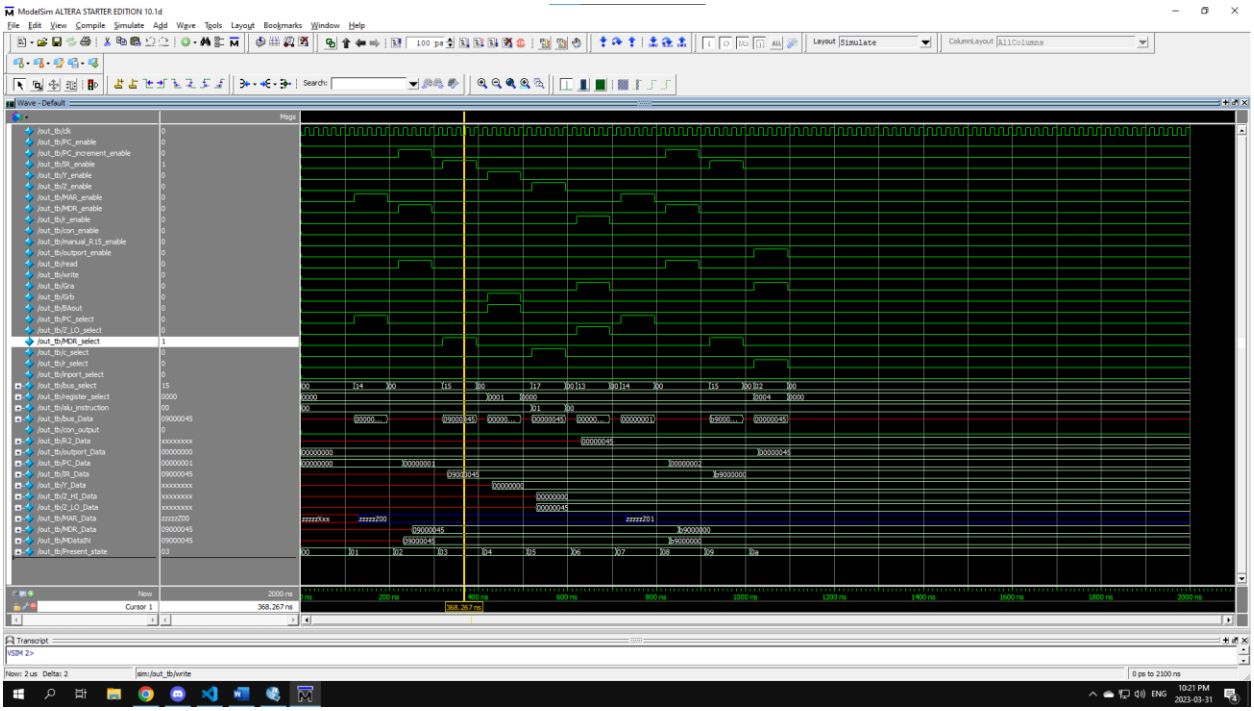
```
1  module inport(
2      input clk, clr,
3      input wire [31:0] input_Data,
4      output wire [31:0] inport_Data);
5
6  reg [31:0] tempData;
7  initial tempData = 32'h0;
8
9  always @(posedge clk)
10 begin
11     if (clr) tempData <= {32{1'b0}};
12     else tempData <= input_Data;
13 end
14
15 assign inport_Data = tempData[31:0];
16
17 endmodule
18
```

Outport

hdl > ports > outport.v

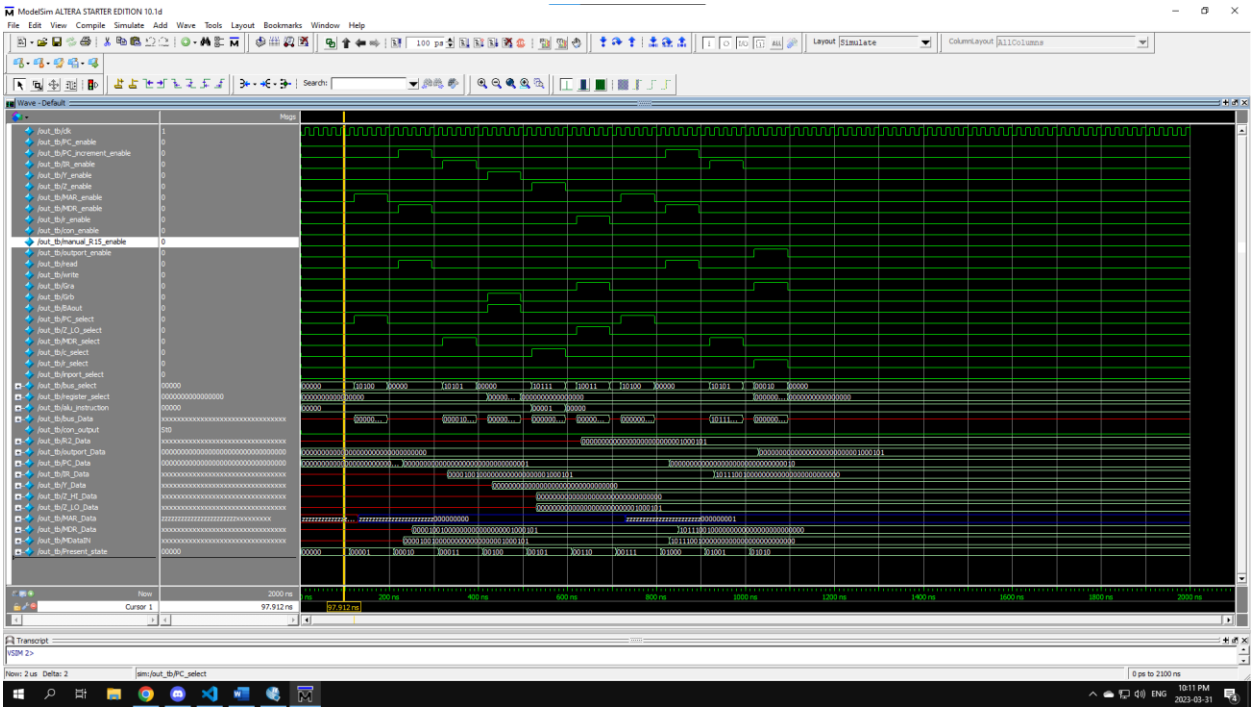
```v
module outport(
    input clk, clr, enable,
    input wire [31:0] bus_Data,
    output wire [31:0] outport_Data);

reg [31:0] tempData;
initial tempData = 32'h0;

always @(posedge clk)
begin
    if (clr) tempData <= {32{1'b0}};
    else if (enable) tempData <= bus_Data;
end

assign outport_Data = tempData[31:0];

endmodule
```

IR Storing Data from Ram
Example of IR Holding the Instruction from Memory:

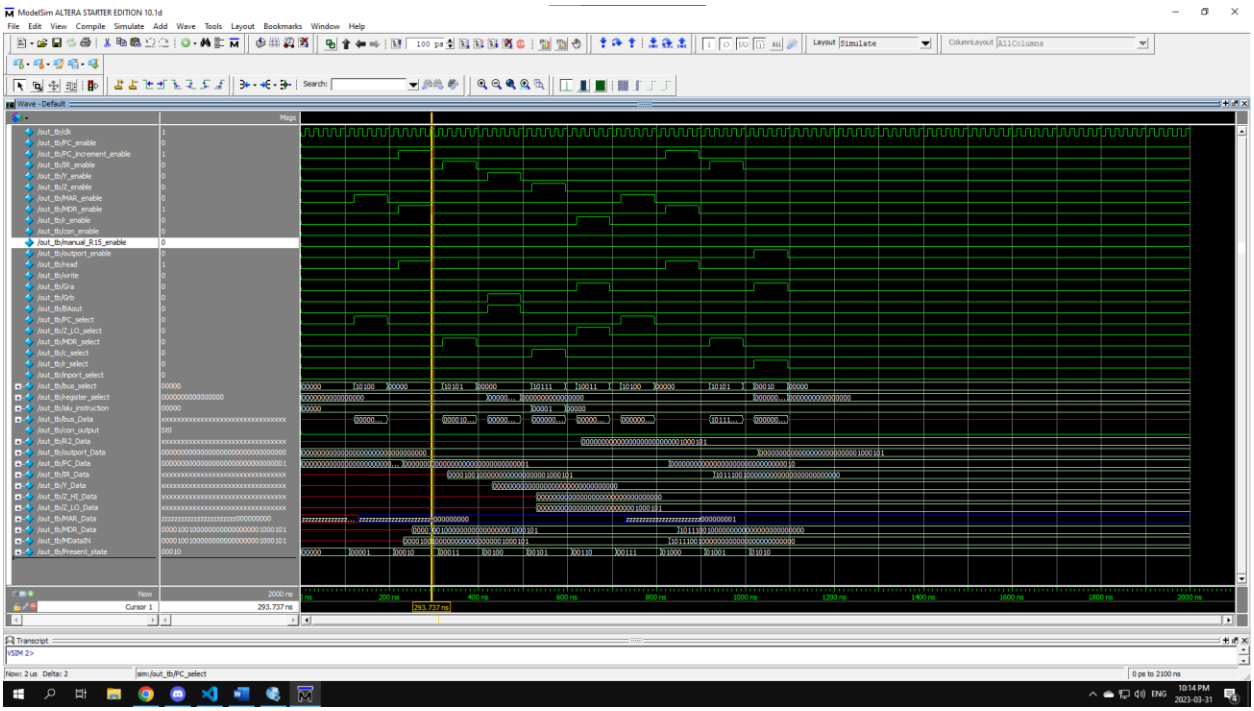


Dedicated PC Incrementor

Example Testbench at the Start:

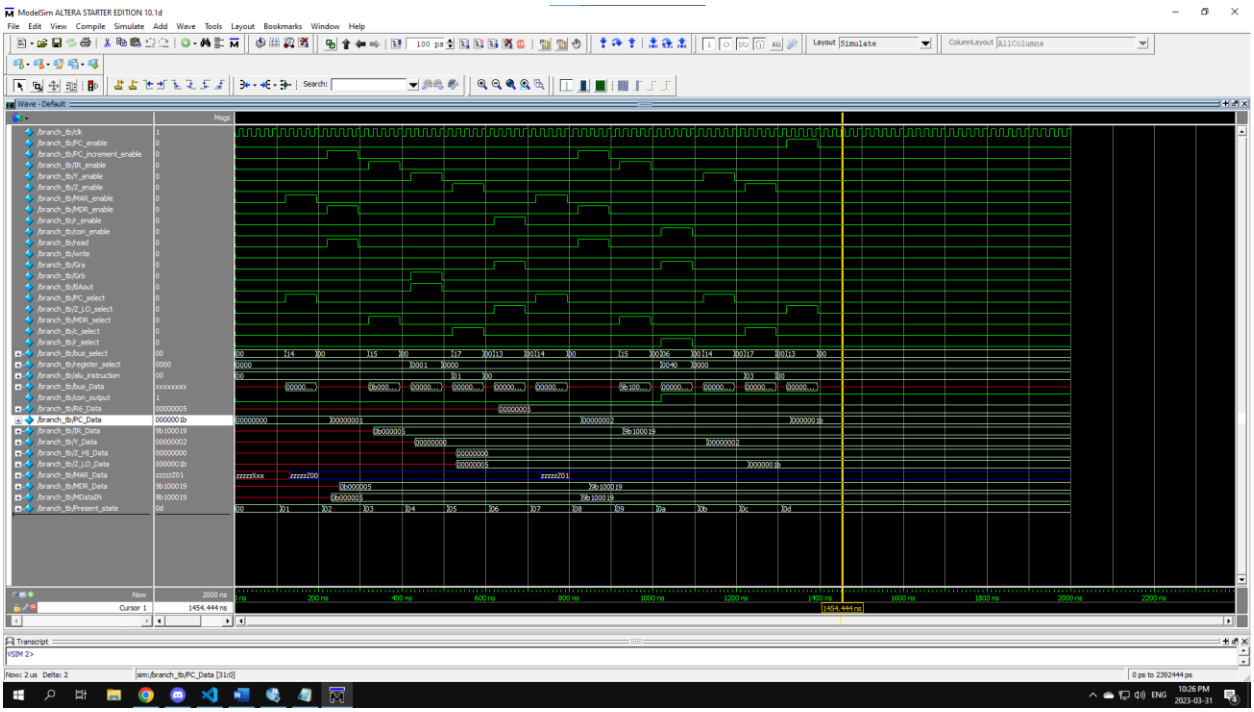


Example Testbench Showing that PC Incremented only 1 time:

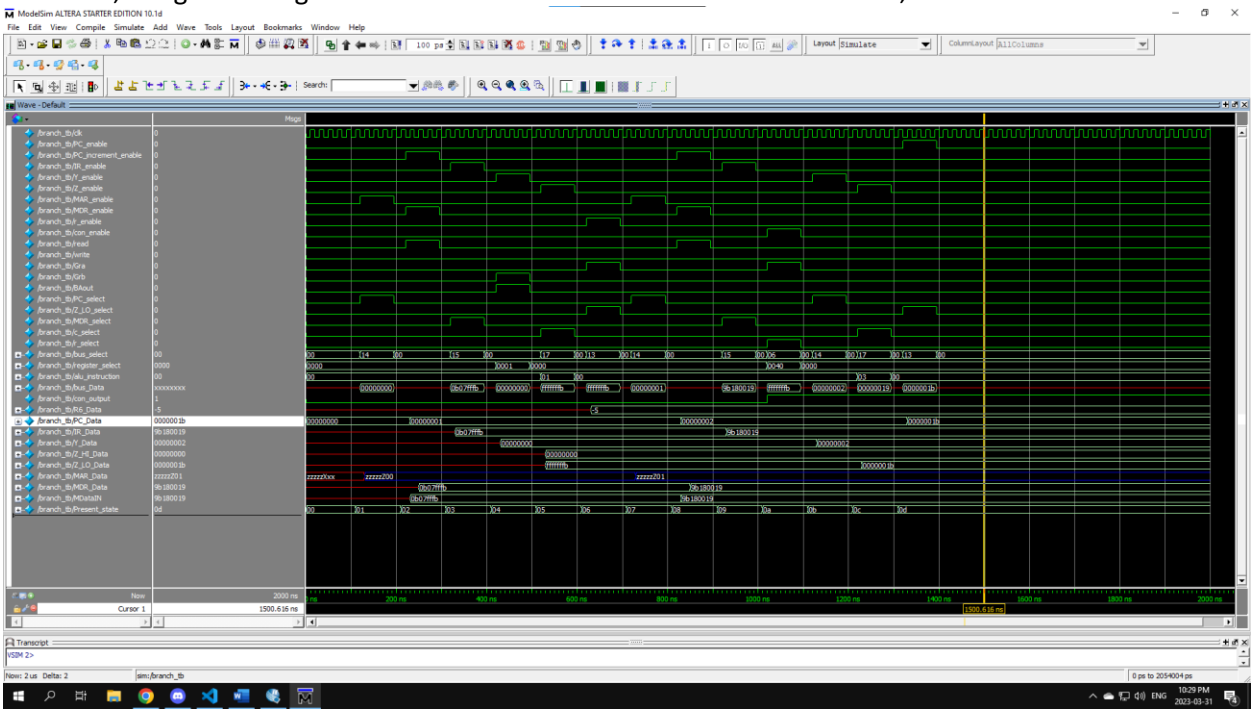


PC Changing for BRMI and BRPL

For BRPL, the general register 6 holds the value 5 and after the instruction, PC holds the value 0x1b.



For BRMI, the general register 6 holds the value -5 and after the instruction, PC holds the value 0x1b.



Instruction Fetch and Decode

```
fetch0: begin
    #10 PC_select <= 1; MAR_enable <= 1;
    #75 PC_select <= 0; MAR_enable <= 0;
end

fetch1: begin
    #10 PC_increment_enable <= 1; read <= 1; MDR_enable <= 1;
    #75 PC_increment_enable <= 0; read <= 0; MDR_enable <= 0;
end

fetch2: begin
    #10 MDR_select <= 1; IR_enable <= 1;
    #75 MDR_select <= 0; IR_enable <= 0;
end
```

```
always @ (instruction, Gra, Grb, Grc) begin
    // Assigning Values to Instruction Register Content Variables
    Ra = instruction [26:23];        // Bit 23 -> 26 of the instruction register is register a
    Rb = instruction [22:19];        // Bit 19 -> 22 of the instruction register is register b
    Rc = instruction [19:15];        // Bit 15 -> 19 of the instruction register is register c

    if (Gra) decoder_input = Ra;
    else if (Grb) decoder_input = Rb;
    else if (Grc) decoder_input = Rc;
```

```
    case (decoder_input)
        4'd0: decoder_out = 16'd1;
        4'd1: decoder_out = 16'd2;
        4'd2: decoder_out = 16'd4;
        4'd3: decoder_out = 16'd8;
        4'd4: decoder_out = 16'd16;
        4'd5: decoder_out = 16'd32;
        4'd6: decoder_out = 16'd64;
        4'd7: decoder_out = 16'd128;
        4'd8: decoder_out = 16'd256;
        4'd9: decoder_out = 16'd512;
        4'd10: decoder_out = 16'd1024;
        4'd11: decoder_out = 16'd2048;
        4'd12: decoder_out = 16'd4096;
        4'd13: decoder_out = 16'd8192;
        4'd14: decoder_out = 16'd16384;
        4'd15: decoder_out = 16'd32768;
    endcase
end

// Assigning Values to the Outputs
assign register_enable = {16{r_enable}} & decoder_out;
assign register_select = ({16{ba_select}} | {16{r_select}}) & decoder_out;
assign C_sign_ext_Data = {{13{instruction[18]}}, instruction[18:0]};
```