CSE 590
Project 1: Report

8-bit Processor (non-pipelined)

GitHub: https://github.com/ramemanatingideas/comp-arch-8-bit/tree/main

---

Team Details:

Sr.no.	Name	UB Person #
1	Ananthakrishnapuram Sampath Ram narayanan (anantha2)	50560715
2	Mayur Rajendra Patil (mayurraj)	50560550
3	MohammadAnas Mansuri (mmansuri)	50556331
4	Ritik Ranjan (ritikran)	50568057

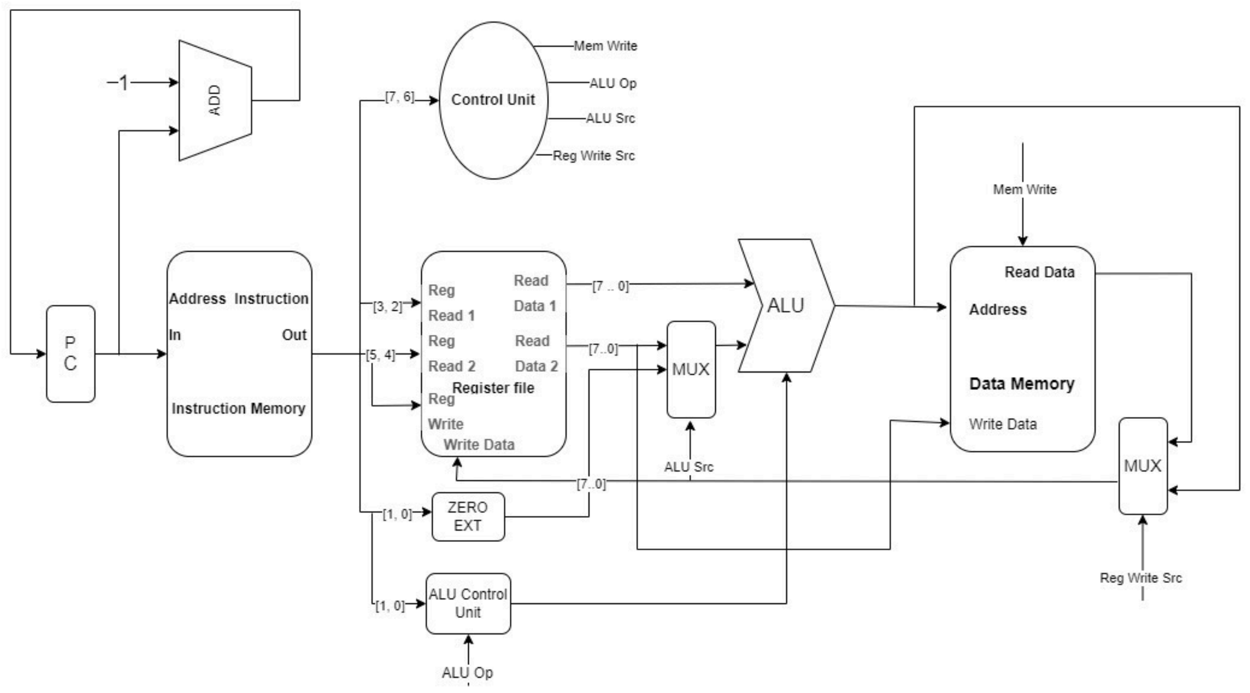
Note:
mayurraj and **mmansuri** own MacBooks, and they made significant contributions to the implementation and testing of major components required for the Datapath on edaplayground.com, as well as in report preparation. **anantha2** and **ritikran** contributed extensively to the software part, Datapath components integration and the simulation of the components on Vivado, and burning the Datapath on Xilinx Basys Board.

Index:

Schematic Of Datapath

&

Control Path



Datapath Schematic:

1. **Instruction Memory:**
    a. Connected to the Program Counter to fetch instructions based on the current address.
2. **Data Memory:**
    a. Interfaces with the ALU or Register File for data storage or retrieval.
    b. Outputs data to the ALU or receives data from the ALU for memory operations.
3. **Register File:**
    a. Provides source operands to the ALU and other functional units.
    b. Receives data from the ALU or Zero Extension for storage.
4. **ALU:**
    a. Receives data from the Register File or Zero Extension for arithmetic and logical operations.
    b. Outputs results to the Register File or other destinations.
5. **Zero Extension:**
    a. Extends shorter data from the Register File for compatibility with ALU operations.
6. **Multiplexer(s):**
    a. Select between different data sources (e.g., Register File, Data Memory) based on signal.

    b. Controls the flow of data to the ALU or other functional units.

7. **Program Counter:**
    a. Outputs the address of the next instruction to the Instruction Memory.
    b. Receives the incremented address for the next instruction fetch.

Control Path Schematic:

Control Unit(s):

1. Generates control signals based on the current instruction opcode.
2. Controls the operation of multiplexers to select appropriate data sources for ALU operations or other functional units.
3. Coordinates the timing of instruction execution by controlling the Program Counter and fetching instructions from the Instruction Memory.
4. Generates control signals to enable/disable specific functional units such as the ALU, Data Memory, or Register File based on the instruction being executed.

Explanation:

The datapath schematic illustrates the flow of data within the processor, showing how instructions are fetched from memory, operated on by the ALU, and stored back into memory or registers. Each component plays a vital role in executing instructions and managing data.

On the other hand, the control path schematic depicts the control signals generated by the control unit(s) based on the instruction opcode. These signals determine the operation of multiplexers, enabling or disabling specific functional units, and controlling the flow of execution within the processor.

This dual-path approach ensures efficient execution of instructions by coordinating data flow and control signals throughout the processor architecture.

Short Description Of Every Component

&

Simulation Results For Each Component

Data Path:

Component Description:

The Datapath module orchestrates the flow of data within a processor, facilitating the execution of instructions. It consists of various components such as the Program Counter, Instruction Memory, Control Unit, Register File, ALU, Data Memory, and Multiplexers, all working together to execute instructions efficiently.

Understanding the Verilog Implementation:

Click here to access the Verilog code of Data Path

Module Declaration:

```
module datapath(
    input clk,
    input reset,
    input enable,
    // Outputs from processor
    output wire [7:0] pc_out_addr,
    output wire [7:0] inst,
    output [7:0] result,
    output [7:0] readData1,
    output [7:0] readData2
);
```

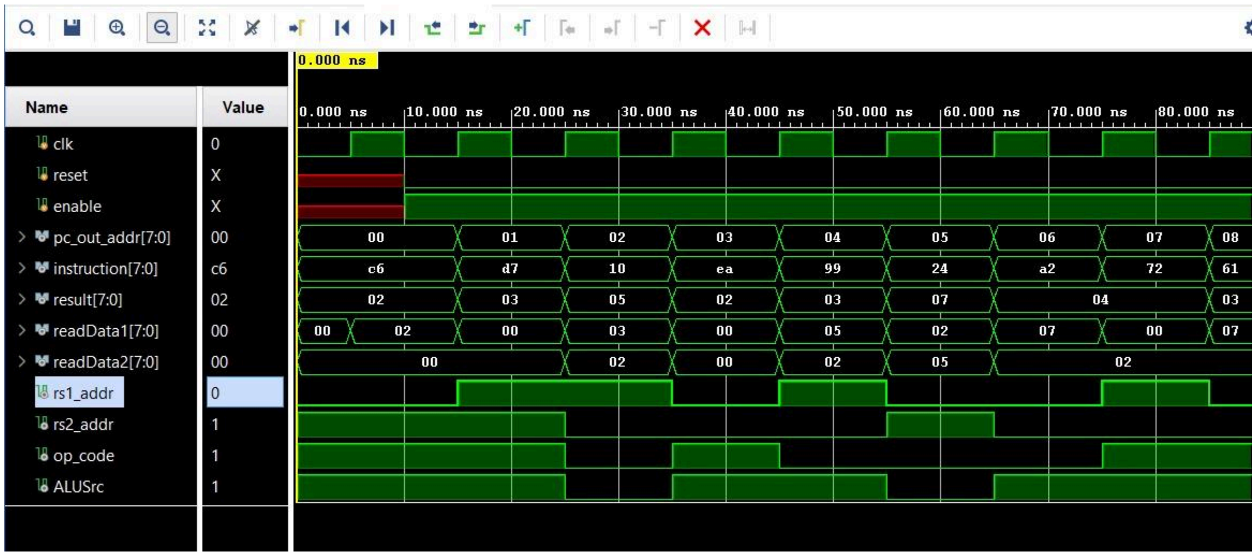
The `*datapath*` module is declared with inputs `*clk*`, `reset`, and `*enable*`, along with various outputs including program counter address (`*pc_out_addr*`), fetched instruction (`*inst*`), computation result (`*result*`), and data from two source registers (`*readData1*` and `*readData2*`).

Operation Logic:

Refer the Data Component

In summary, the Datapath module integrates various components to form a complete processing unit capable of executing instructions. It handles instruction fetching, decoding, register operations, arithmetic and logical computations, memory operations, and result write-back, orchestrating the flow of data according to the instruction being executed.

Simulation Result:



ALU (Arithmetic Logic Unit):

Component Description:

The ALU (Arithmetic Logic Unit) serves as the computational heart of the CPU, executing fundamental arithmetic and logical operations on binary data. It's akin to the brain of the processor, handling tasks like addition, subtraction, shifting, and logical operations.

Understanding the Verilog Implementation:

Click here to access the Verilog code of ALU

Module Declaration:

```
module alu (
    input [7:0] rs_data1,      // Input data from the first source register
    input [7:0] rs_data2,      // Input data from the second source register
    input [1:0] alu_op,        // ALU operation code
    input alu_src,             // Determine if it's immediate or Rtype
    output reg [7:0] result    // Output result of ALU operation
);
```

This module defines input ports for the two source registers (`*rs_data1*` and `*rs_data2*`), an operation code (`*alu_op*`), and a control signal (`*alu_src*`). The output port `*result*` holds the computed result of the ALU operation.

Parameter Definitions:

```
parameter ADD = 2'b00;     // Addition
parameter SUB = 2'b01;     // Subtraction
parameter SLL = 2'b10;     // Shift Left Logical
parameter AND = 2'b11;     // Bitwise AND
```

Here, parameters are defined for different ALU operation codes, providing a convenient and descriptive way to refer to these operations within the code.

Initial Block:

```
initial begin
  result = 8'b00000000;
end
```

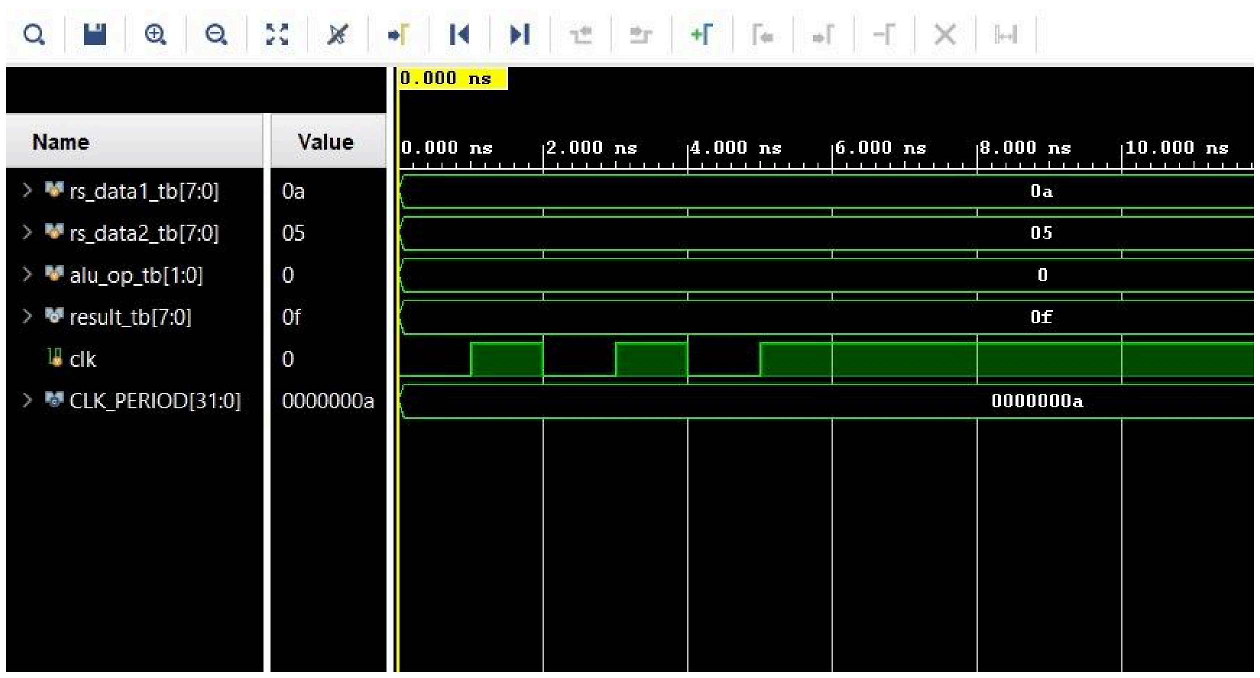
The `*initial*` block initializes the `result` to zero, ensuring a clean start for subsequent computations.

ALU Operation Logic:

```
always @(*) begin
    if(alu_src) begin
        result = rs_data1 + rs_data2;
    end else begin
        case (alu_op)
            ADD: result = rs_data1 + rs_data2;    // Perform addition
            SUB: result = rs_data1 - rs_data2;    // Perform subtraction
            SLL: result = rs_data1 << rs_data2;  // Perform left logical shift
            AND: result = rs_data1 & rs_data2;   // Perform bitwise AND
            default: result = 8'b0;              // Default value: no operation
        endcase
    end
end
```

This is where the magic happens! The `*always @(*)*` block represents continuous logic, meaning it updates whenever any of its inputs change. If `*alu_src*` is active, the ALU directly adds `*rs_data1*` and `*rs_data2*`. Otherwise, based on the `*alu_op*`, it performs the corresponding operation, such as addition, subtraction, left logical shift, or bitwise AND.

Simulation Result:



Name	Value
rs_data1_tb[7:0]	0a
rs_data2_tb[7:0]	05
alu_op_tb[1:0]	0
result_tb[7:0]	0f
clk	0
CLK_PERIOD[31:0]	0000000a

Control_Unit:

Component Description:

The Control Unit plays a pivotal role in directing various operations within the CPU by generating control signals based on the opcode received. These control signals regulate activities such as register writes, memory reads and writes, and ALU operations, among others.

Understanding the Verilog Implementation:

Click here to access the Verilog code of the Control Unit

Module Declaration:

```
module control_unit(
    input [1:0] opcode,          // Input opcode representing the operation code
    output reg RegWrite,         // Control signal for register write
    output reg MemWrite,         // Control signal for memory write
    output reg MemRead,          // Control signal for memory read
    output reg ALUSrc,           // Control signal for ALU source
    output reg RegWriteSrc       // Control signal for register write source
);
```

The `*control_unit*` module takes a 2-bit input `*opcode*` representing the operation code and generates control signals (`*RegWrite*`, `*MemWrite*`, `*MemRead*`, `*ALUSrc*`, `*RegWriteSrc*`) based on this opcode.

Parameter Definitions:

The control signals (`*RegWrite*`, `*MemWrite*`, `*MemRead*`, `*ALUSrc*`, `*RegWriteSrc*`) are output signals controlling various operations in a processor, each determined by the value of the opcode.

Initial Block:

```
always @* begin
    begin
        case (opcode)
          // Cases for different opcodes determining control signals
        endcase
    end
end
```

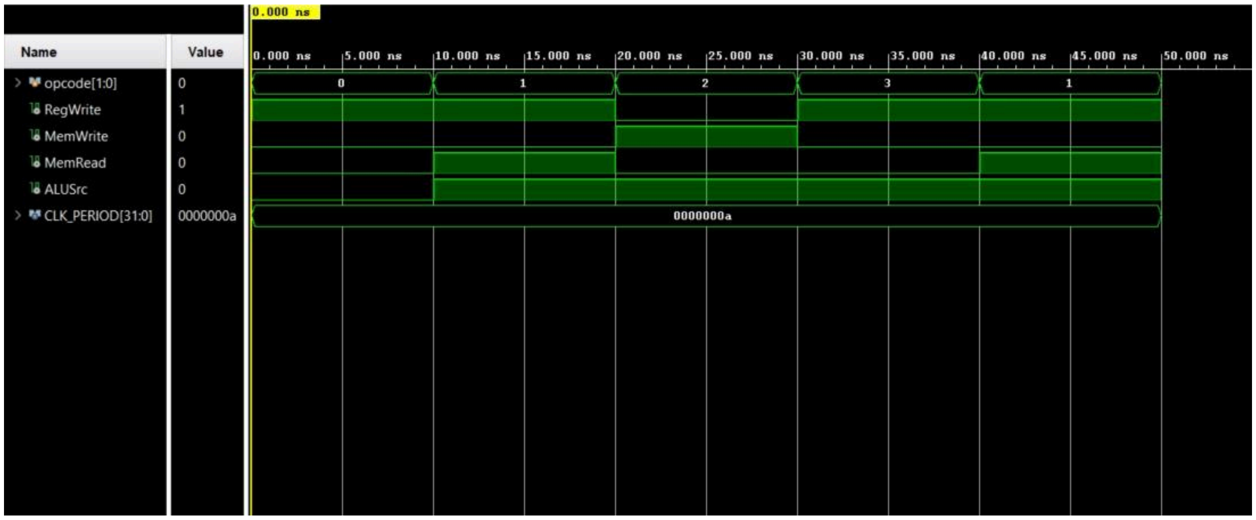
An `*always*` block is used to implement combinational logic, where the `@*` sensitivity list indicates that the block should trigger whenever any of its inputs change. Inside the block, a `*case*` statement is used to determine the control signals based on the `opcode`.

Operation Logic:

The `*case*` statement evaluates the `*opcode*` and selects the appropriate block of code based on its value. For each possible value of `*opcode*` (00, 01, 10, 11), specific control signals are assigned accordingly. The `*default*` case provides default control signals if none of the specified cases match.

```
case (opcode)
    2'b00: begin // Control signals for opcode 00
    end
    2'b01: begin // Control signals for opcode 01
    end
    2'b10: begin // Control signals for opcode 10
    end
    2'b11: begin // Control signals for opcode 11
    end
    default: begin // Default control signals
    end
endcase
```

Simulation Result:



DataMemory (RAM):

Component Description:

The Data Memory module, often referred to as RAM (Random Access Memory), stores and retrieves data synchronously. It serves as a temporary storage location within the CPU, facilitating read and write operations.

Understanding the Verilog Implementation:

Click here to access the Verilog code of the Data Memory

Module Declaration:

```
module data_memory(
    input clk,                  // Clock input
    input [7:0] data_in,        // Data input to be written to memory
    input wr,                   // Write control signal
    input rd,                   // Read control signal
    input [7:0] addr,           // Address input
    output reg [7:0] data_out  // Data output read from memory
);
```

Parameter Definitions:

The `*memory*` parameter represents a 256x8-bit RAM array, declared using the `*reg*` datatype. It stores data written by the `*data_in*` signal based on the address provided.

Initial Block:

The RAM array `*memory*` is initialized implicitly to all zeros.

Operation Logic:

Write Operation:

```
always @(negedge clk) begin
    if (wr) begin
        memory[addr] <= data_in; // Write data from data_in to the memory location specified by
addr
    end
end
```

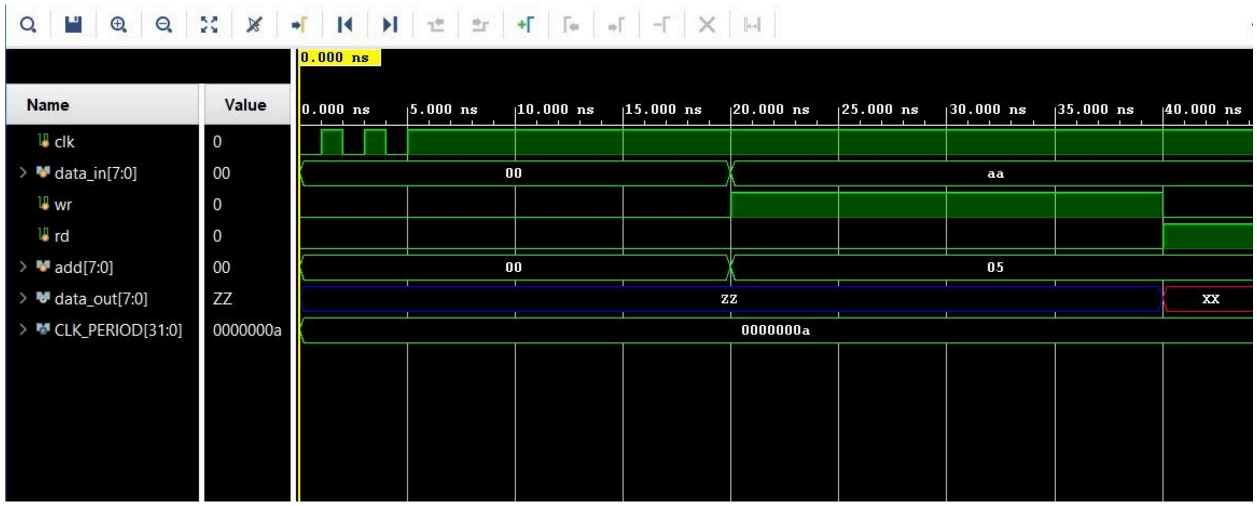
The `*always*` block triggers on the falling edge of the clock (`*negedge clk*`). It checks if the write control signal `*wr*` is high. If `*wr*` is high, it writes the data from `*data_in*` to the memory location specified by the address `*addr*`.

Read Operation:

```
always @(addr or rd) begin
   if (rd) begin
        data_out = memory[addr]; // Read data from the memory location specified by addr and
assign it to data_out
   end
   else begin
      data_out = 8'bzzzzzzzz; // If rd is low, output high-impedance ("don't care" state)
   end
end
```

This block triggers whenever `*addr*` or `*rd*` changes. If the read control signal `*rd*` is high, it reads the data from the memory location specified by the address `*addr*` and assigns it to `*data_out*`. If `*rd*` is low, `*data_out*` is set to high-impedance (`*8'bzzzzzzzz*`), indicating a "don't care" state.

Simulation Result:



Instruction-Memory:

Component Description:

The Instruction Memory module, often referred to as the Program Memory, is responsible for storing and retrieving instructions based on the program counter address. It plays a crucial role in the instruction fetch stage of the CPU pipeline.

Understanding the Verilog Implementation:

[Click here to access the Verilog code of the Instruction Memory]

Module Declaration:

```
module ins_mem(
    input [7:0] pc_addr,  // Program counter address input
    output [7:0] inst     // Instruction output
);
```

The `*ins_mem*` module is declared with an input `*pc_addr*`, representing the program counter address, and an output `*inst*`, representing the fetched instruction.

Internal Signals and Memory Declaration:

```
reg [7:0] inst;             // 8-bit register to store the current instruction
reg [7:0] ins_mem[255:0];   // 256x8-bit memory array to store instructions
```

Internal signals `*inst*` and `*ins_mem*` are defined. `*inst*` is an 8-bit register used to store the current instruction, while `*ins_mem*` is a 256x8-bit memory array used to store instructions.

Memory Initialization:

```
initial begin
        $readmemb("InsMem.mem", ins_mem); // Initialize ins_mem using contents from
"InsMem.mem"
end
```

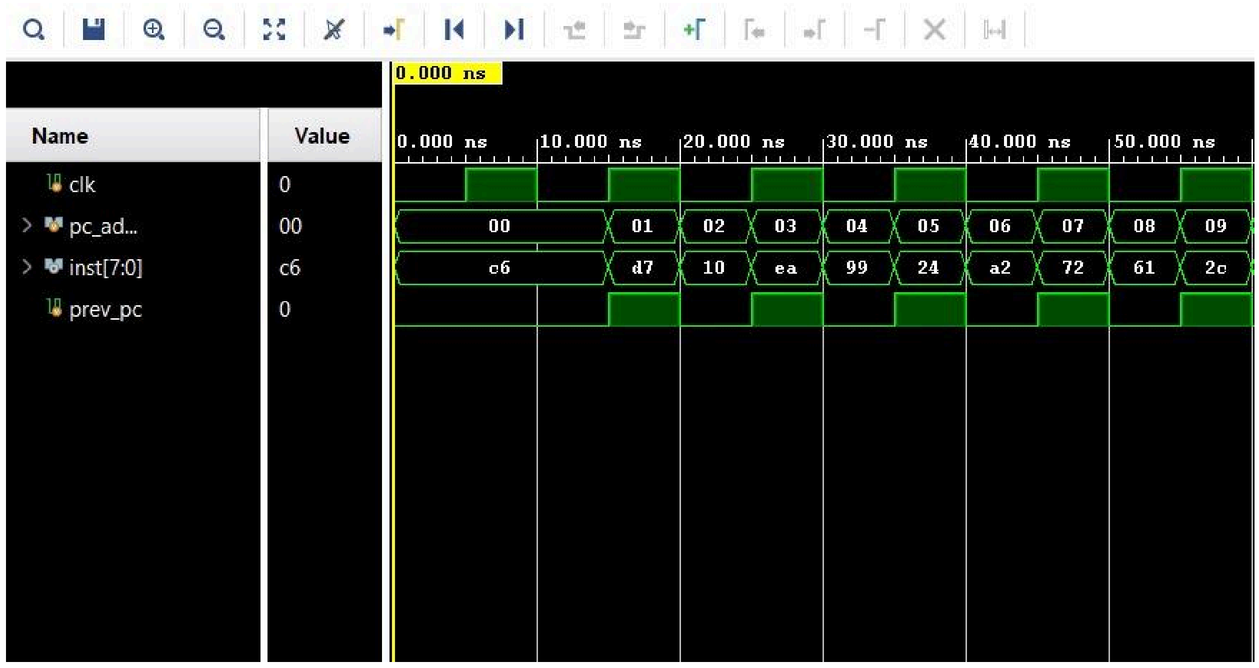
The `*initial*` block reads the contents of a memory initialization file ("InsMem.mem") and initializes the `*ins_mem*` memory array with the data.

Operation Logic:

```
always @(pc_addr) begin
    inst <= ins_mem[pc_addr]; // Fetch the instruction from ins_mem at the address specified by
pc_addr
end
```

This `*always*` block is sensitive to changes in `*pc_addr*`. Whenever `*pc_addr*` changes, it fetches the instruction from the memory `*ins_mem*` at the address specified by `*pc_addr*` and assigns it to the `*inst*` register.

Simulation Result:



Name	Value
clk	0
pc_ad...	00
inst[7:0]	c6
prev_pc	0

Program_counter:

Component Description:

The Program Counter module, often abbreviated as PC, maintains the address of the next instruction to be fetched in the instruction memory. It operates based on clock signals and control inputs such as reset and enable.

Understanding the Verilog Implementation:

Click here to access the Verilog code of the Program Counter

Module Declaration:

```
module program_counter(
    input wire clk,       // Clock input
    input wire reset,     // Reset signal
    input wire enable,    // Enable signal
    output [7:0] pc       // Program counter output
);
```

The `*program_counter*` module is declared with inputs `*clk*`, `*reset*`, and `*enable*`, and an output `*pc*`, representing the program counter value.

Program Counter Register and Initialization:

```
reg [7:0] pc_reg;  // 8-bit register to hold the program counter value

initial begin
    pc_reg <= 8'b00000000;  // Initialize pc_reg to 0
end
```

An 8-bit register `*pc_reg*` is declared to hold the program counter value. In the `*initial*` block, `*pc_reg*` is initialized to zero when the simulation starts.

Program Counter Update:

```
always @(posedge clk or posedge reset) begin
    if (reset) begin
        pc_reg <= 8'b00000000;  // Reset the program counter to 0
    end
    else if (enable) begin
```

```
        pc_reg <= pc_reg + 1;   // Increment the program counter by 1 on each clock cycle if
enable is high
    end
end
```

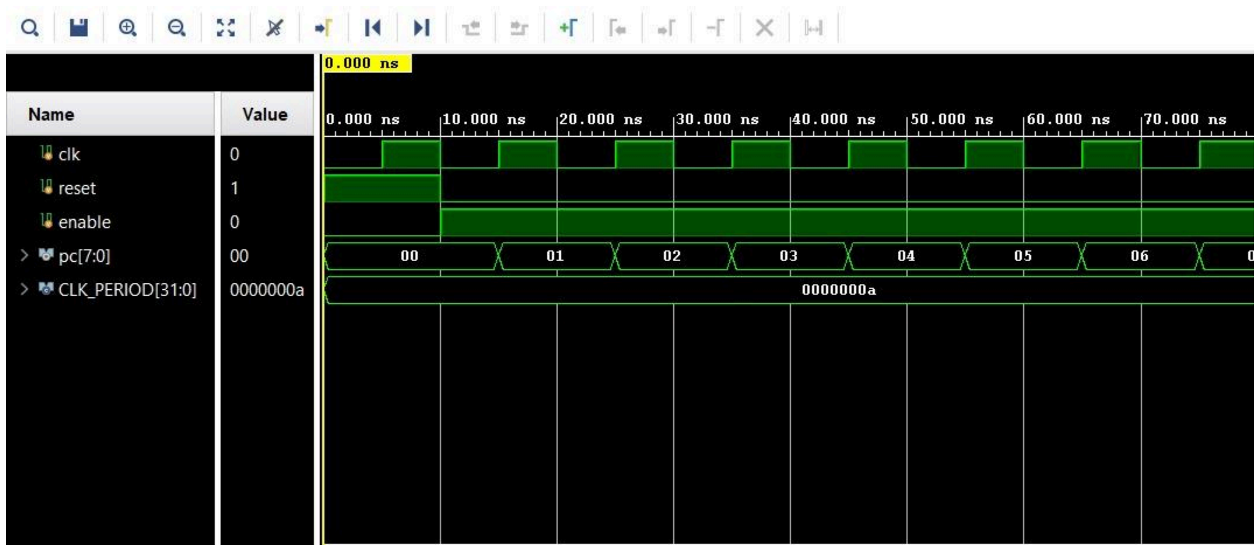
This `*always*` block triggers on the rising edge of the clock (`*posedge clk*`) or the rising edge of the reset signal (`*posedge reset*`). If `*reset*` is high, the program counter is reset to zero. If `*enable*` is high, the program counter (`*pc_reg*`) is incremented by 1 on each clock cycle.

Assigning Program Counter Output:

```
assign pc = pc_reg;  // Assign the value of pc_reg to the output pc
```

This `*assign*` statement continuously drives `*pc*` with the value of the program counter register `*pc_reg*`.

Simulation Result:



Register_file:

Component Description:

The Register File module serves as a collection of registers that store data for various operations within the CPU. It allows for reading and writing data to and from registers based on specified addresses.

Understanding the Verilog Implementation:

Click here to access the Verilog code of the Register File

Module Declaration:

```
module register_file (
    input wire clk,            // Clock signal
    input wire reset,          // Reset signal
    input [1:0] rs1_addr,      // Address for the first source register
    input [1:0] rs2_addr,      // Address for the second source register
    input [1:0] wr_addr,       // Address where data is to be written
    input [7:0] wr_data,       // Data to be written into the register file
    input reg_wr_en,           // Write enable signal
    output reg [7:0] rs1_data,// Data from the first source register
    output reg [7:0] rs2_data // Data from the second source register
);
```

The `*register_file*` module is declared with various inputs and outputs to handle register operations. Inputs include clock signal `*clk*`, reset signal `*reset*`, register addresses `*rs1_addr*`, `*rs2_addr*`, `*wr_addr*`, data to be written `*wr_data*`, and write enable signal `*reg_wr_en*`. Outputs include data from the first source register `*rs1_data*` and data from the second source register `*rs2_data*`.

File Declaration and Initialization:

```
reg [7:0] registers [0:3]; // Define an array of registers with 4 8-bit registers

initial begin
    // Initialize all registers to zero
    registers[0] = 8'b00000000;
    registers[1] = 8'b00000000;
    registers[2] = 8'b00000000;
    registers[3] = 8'b00000000;
end
```

An array `*registers*` of 4 8-bit registers is declared to represent the register file. In the `*initial*` block, all registers are initialized to zero.

Operation Logic:

Write Operation:

```
always @(posedge clk) begin
    if (reset) begin
        // Reset all registers to zero
        registers[0] <= 8'b00000000;
        registers[1] <= 8'b00000000;
        registers[2] <= 8'b00000000;
        registers[3] <= 8'b00000000;
    end
    if (reg_wr_en) begin
        // Write data into the register specified by wr_addr
        registers[wr_addr] <= wr_data;
    end
end
```

This `*always*` block triggers on the positive edge of the clock. If `*reset*` is high, all registers are reset to zero. If `*reg_wr_en*` is high, data specified by `*wr_data*` is written into the register specified by `*wr_addr*`.

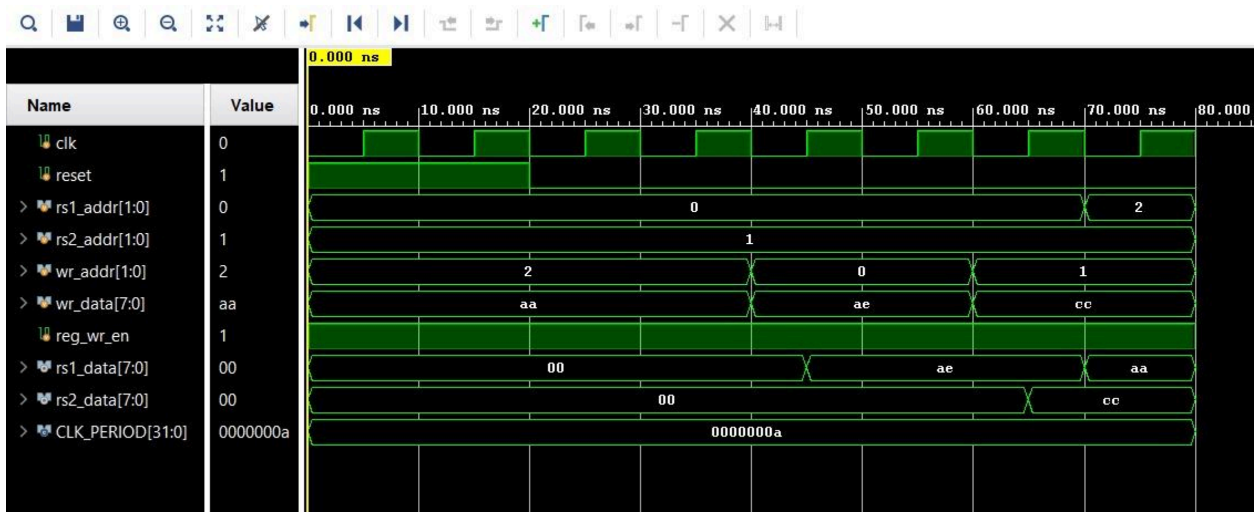
Read Operation:

```
always @* begin
    // Extract data from the register file based on the rs1_addr
    case (rs1_addr)
        // Cases for different register addresses
    endcase

    // Extract data from the register file based on the rs2_addr
    case (rs2_addr)
        // Cases for different register addresses
    endcase
end
```

This block triggers whenever any of the inputs (`*rs1_addr*` or `*rs2_addr*`) change. It extracts data from the register file based on the provided register addresses and assigns them to `*rs1_data*` and `*rs2_data*`, respectively. If the provided register address is invalid, a default value of 8'b0 is assigned.

Simulation Result:



Multiplexer:

Component Description:

The Multiplexer (mux2) module is a fundamental digital logic component that selects one of two input signals based on a control signal and forwards it to the output. It serves as a versatile switch within digital circuits, allowing for data routing based on a selection criterion.

Understanding the Verilog Implementation:

Click here to access the Verilog code of the Multiplexer

Module Declaration:

```
module mux2 #(parameter BIT_LEN=8)
        (input [BIT_LEN-1:0] input1,
         input [BIT_LEN-1:0] input2,
         input select,
         output [BIT_LEN-1:0] out);
```

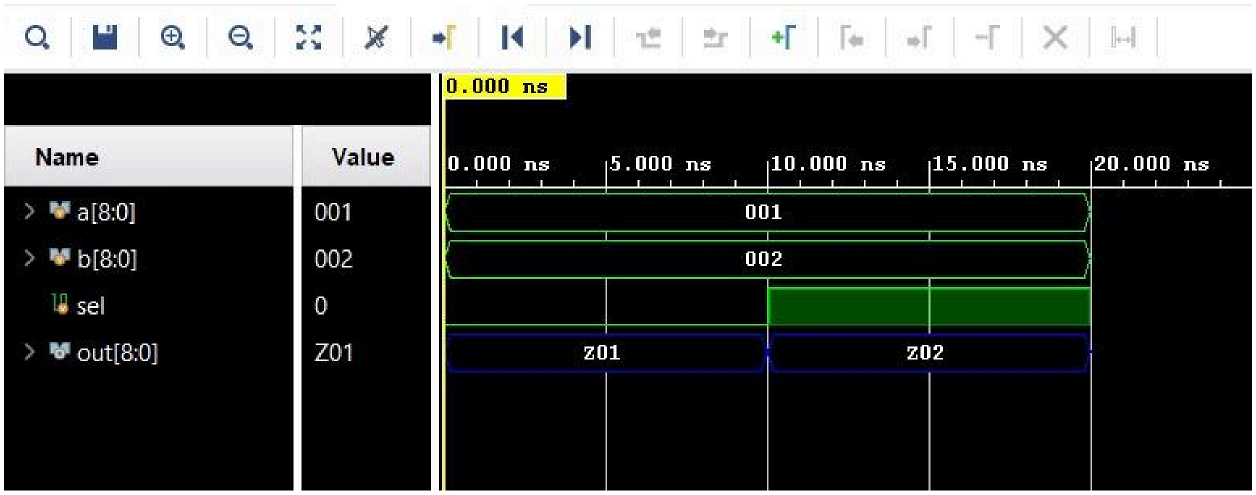
The `*mux2*` module is declared as a 2-to-1 multiplexer with a parameter `*BIT_LEN*` determining the bit width of the inputs and output. Inputs include `*input1*` and `*input2*`, both of width `*BIT_LEN*`, a single-bit `select` input, and an output `out` of width `*BIT_LEN*`.

Operation Logic:

```
assign out = select == 0 ? input1 : input2;
```

The `*assign*` statement is used to assign a value to the output `*out*`. It selects either `*input1*` or `*input2*` based on the value of the `*select*` signal. If `select` is 0, `*input1*` is selected; otherwise, `*input2*` is selected.

Simulation Result:



Zero Extension:

Component Description:

The Zero Extension module (zero_extension) is designed to extend a 2-bit input by padding zeros to make it 8 bits wide. It's a common operation in digital systems, often used to prepare data for arithmetic or logical operations where data width compatibility is required.

Understanding the Verilog Implementation:

Click here to access the Verilog code of the Zero Extension

Module Declaration:

```
module zero_extension(
    input [1:0] input_data,       // Input data to be zero-extended
    output reg [7:0] output_data  // Zero-extended output data
);
```

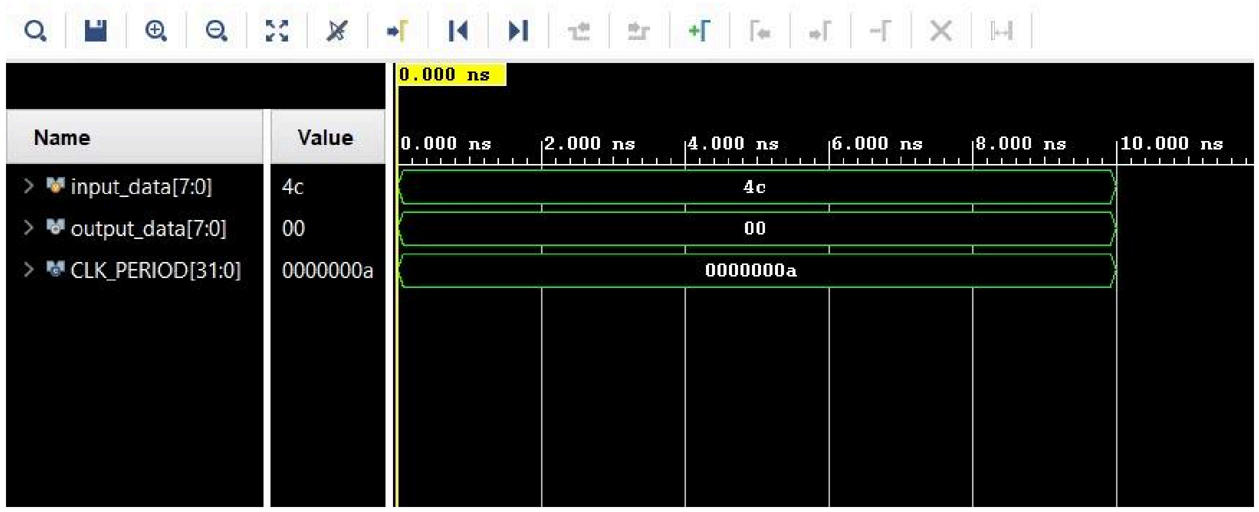
The `*zero_extension*` module is declared with an input `*input_data*` of width 2 bits and an output `*output_data*` of width 8 bits.

Operation Logic:

```
always @* begin
      output_data = {6'b0, input_data}; // Zero-extend input_data by concatenating 6 zero bits
with the input
end
```

This `*always*` block triggers whenever there is a change in `*input_data*`. Inside the block, the `*output_data*` is assigned the value of `*input_data*` with zero extension. `*{6'b0, input_data}*` concatenates 6 zero bits (`6'b0`) with the 2-bit `*input_data*`, resulting in an 8-bit output.

Simulation Result:



```
input_data[7:0]     4c          4c
output_data[7:0]    00          00
CLK_PERIOD[31:0]    0000000a    0000000a
```

Component Code in Verilog

ALU:

```
// Arithmetic Logic Unit (ALU) Module
module alu (
    input [7:0] rs_data1,     // Input data from the first source register
    input [7:0] rs_data2,     // Input data from the second source register
    input [1:0] alu_op,       // ALU operation code
    input alu_src,            // Determine if its immediate or Rtype
    output reg [7:0] result   // Output result of ALU operation
);
    initial begin
     result = 8'b00000000;
    end
    // Define parameters for ALU operation codes
    parameter ADD = 2'b00;     // Addition
    parameter SUB = 2'b01;     // Subtraction
    parameter SLL = 2'b10;     // Shift Left Logical
    parameter AND = 2'b11;     // Bitwise AND

    // Perform ALU operation based on alu_op
    always @(*) begin
        if(alu_src) begin
            result = rs_data1 + rs_data2;
        end else begin
            case (alu_op)
                ADD: result = rs_data1 + rs_data2;    // Perform addition
                SUB: result = rs_data1 - rs_data2;    // Perform subtraction
                SLL: result = rs_data1 << rs_data2;  // Perform left logical shift
                AND: result = rs_data1 & rs_data2;   // Perform bitwise AND
                default: result = 8'b0 ;// Default value is no op
            endcase
        end
    end

endmodule
```

Control Unit

```
`timescale 1ns / 1ps

 module control_unit(input [1:0] opcode,
                     output reg RegWrite,
                     output reg MemWrite,
                     output reg MemRead,
                     output reg ALUSrc,
                     output reg RegWriteSrc
                     );
    always @* begin
        begin
            case (opcode)
                2'b00: begin
                    RegWrite = 1;
                    MemWrite = 0;
                    MemRead = 0;
                    ALUSrc = 0;
                    RegWriteSrc = 0;
                end
                2'b01: begin
                    RegWrite = 1;
                    MemWrite = 0;
                    MemRead = 1;
                    ALUSrc = 1;
                    RegWriteSrc = 1;
                end
                2'b10: begin
                    RegWrite = 0;
                    MemWrite = 1;
                    MemRead = 0;
                    ALUSrc = 1;
                    RegWriteSrc = 0;
                end
                2'b11: begin
                    RegWrite = 1;
                    MemWrite = 0;
                    MemRead = 0;
                    ALUSrc = 1;
                    RegWriteSrc = 0;
                end

                default: begin
                    RegWrite = 1;
                    MemWrite = 0;
                    MemRead = 0;
                    ALUSrc = 0;
                    RegWriteSrc = 0;
                end
            endcase
        end
    end
endmodule
```

Data Memory (RAM)

```
`timescale 1ns / 1ps
module data_memory(
  input clk,                  // Clock input
  input [7:0] data_in,        // Data input to be written to memory
  input wr,                   // Write control signal
  input rd,                   // Read control signal
  input [7:0] addr,           // Address input
  output reg [7:0] data_out  // Data output read from memory
);

 reg [7:0] memory [0:255];  // Declaration of 256x8-bit RAM array

  always @(negedge clk) begin
//    $display("write control signal: wr: %b, Address: %b", wr, addr);
//    $display("data_in: %b", data_in);
    if (wr) begin
      memory[addr] <= data_in;
    end
  end

  always @(addr or rd) begin
//   $display("read control signal (only reading): rd: %b with addr %b", rd, addr);
    if (rd) begin
      data_out = memory[addr];
    end
    else begin
      data_out = 8'bzzzzzzzz;       // If rd is low, output high-impedance ("don't care" state)
    end
  end

endmodule
```

Instruction Memory

```
module ins_mem(inst,pc_addr);
    input [7:0] pc_addr;
    output [7:0] inst;
    reg [7:0] inst;
    reg [7:0] ins_mem[255:0];
    initial begin
          $readmemb("InsMem.mem",ins_mem);
    end
    always @(pc_addr)
        begin
            inst <= ins_mem[pc_addr];
        end
endmodule
```

Program Counter

```
module program_counter(
    input wire clk,
    input wire reset,
    input wire enable,
    output [7:0] pc
);

 // Register to hold the program counter value
reg [7:0] pc_reg;

initial begin
    pc_reg <= 8'b00000000;
end

always @(posedge clk  or posedge reset ) begin
    if(reset) begin
        pc_reg <= 8'b00000000;
    end
    else if (enable) begin
            // Increment the program counter
            pc_reg <= pc_reg + 1;
    end
end

assign pc = pc_reg;


endmodule
```

Register File

```
// Register File Module
module register_file (
    input wire clk,            // Clock signal
    input wire reset,          // Reset signal
    input [1:0] rs1_addr,      // Address for the first source register
    input [1:0] rs2_addr,      // Address for the second source register
    input [1:0] wr_addr,       // Address where data is to be written
    input [7:0] wr_data,       // Data to be written into the register file
    input reg_wr_en,           // Write enable signal
    output reg [7:0] rs1_data, // Data from the first source register
    output reg [7:0] rs2_data  // Data from the second source register
);

    reg [7:0] registers [0:3]; // Define an array of registers with 4 8-bit registers

    initial begin
            registers[0] = 8'b00000000;
            registers[1] = 8'b00000000;
            registers[2] = 8'b00000000;
            registers[3] = 8'b00000000;
    end

    // Write data into the register file when reg_wr_en is high
    always @(posedge clk) begin
        if(reset) begin
            registers[0] = 8'b00000000;
            registers[1] = 8'b00000000;
            registers[2] = 8'b00000000;
            registers[3] = 8'b00000000;
        end
        if (reg_wr_en) begin
            // Write data into the register specified by wr_addr
            registers[wr_addr] <= wr_data;
//            $display("wr_data %b", wr_data);
        end
    end

    // Assign data to the output ports based on the addresses provided
    always @* begin
//        $display("rs1_addr %b", rs1_addr);
//        $display("rs2_addr %b", rs2_addr);
//        $display("register[0] %b", registers[0]);
//        $display("register[1] %b", registers[1]);
//        $display("register[2] %b", registers[2]);
//        $display("register[3] %b", registers[3]);
        // Extract data from the register file based on the rs1_addr
        case (rs1_addr)
            2'b00: rs1_data = registers[0]; // Register $s0
            2'b01: rs1_data = registers[1]; // Register $s1
            2'b10: rs1_data = registers[2]; // Register $s2
            2'b11: rs1_data = registers[3]; // Register $s3
            default: rs1_data = 8'b0; // Default value
        endcase

        // Extract data from the register file based on the rs2_addr
        case (rs2_addr)
            2'b00: rs2_data = registers[0]; // Register $s0
            2'b01: rs2_data = registers[1]; // Register $s1
            2'b10: rs2_data = registers[2]; // Register $s2
            2'b11: rs2_data = registers[3]; // Register $s3
            default: rs2_data = 8'b0; // Default value
        endcase
    end

endmodule
```

Multiplexer

```
module mux2 #(parameter BIT_LEN=8)
        (     input [BIT_LEN-1:0] input1,
              input [BIT_LEN-1:0] input2,
              input  select,
              output [BIT_LEN-1:0] out );
    assign out = select == 0 ? input1 : input2;
endmodule
```

Zero Extension

```
`timescale 1ns / 1ps

module zero_extension(
    input [1:0] input_data,       // Input data to be zero-extended
    output reg [7:0] output_data  // Zero-extended output data
);
    // Zero extension operation
    always @* begin
//        output_data = {{6{input_data[1]}}, input_data};
        output_data = {6'b0, input_data};
    end
endmodule
```

Data Path

```
module datapath(
    input clk,
    input  reset,
    input  enable,
    // Outputs from processor
    output wire [7:0] pc_out_addr,
    output wire [7:0] inst,
    output [7:0] result,
    output [7:0] readData1,
    output [7:0] readData2
);
```

```
// Write back result from Mux (Mem to Reg)
wire [7:0] mux_wb_out;

program_counter PC(
   .clk(clk),
   .reset(reset),
   .enable(enable),
   .pc(pc_out_addr)
);

wire [7:0] inst, instruction;
wire [1:0] rs1_addr, rs2_addr;
// Instantiate Instruction Memory
ins_mem InstructionMemory (
    .pc_addr(pc_out_addr),
    .inst(instruction)
);

assign inst = instruction;
assign rs1_addr = instruction[5:4]; // rt/d reg
assign rs2_addr = instruction[3:2]; // rs reg


wire [1:0] op_code; // opcode
assign op_code = instruction[7:6];

wire RegWrite;              // Output RegWrite signal
wire MemWrite;              // Output MemWrite signal
wire MemRead;               // Output MemRead signal
wire ALUSrc;                // Output ALUSrc signal
wire RegWriteSrc;               // Output ALUSrc signal

control_unit CU(
    .opcode(op_code),
    .RegWrite(RegWrite),
    .MemWrite(MemWrite),
    .MemRead(MemRead),
    .ALUSrc(ALUSrc),
    .RegWriteSrc(RegWriteSrc)
);

wire [7:0] read_data1, read_data2;
```

```
// Instantiate Register File
register_file RF (
    .clk(clk),
    .reset(reset),
    .rs1_addr(rs1_addr),
    .rs2_addr(rs2_addr),
    .wr_addr(rs1_addr),
    .wr_data(mux_wb_out),
    .reg_wr_en(RegWrite),
    .rs1_data(readData1),
    .rs2_data(readData2)
);

assign read_data1 = readData1;
assign read_data2 = readData2;

wire[1:0] imm;
assign imm = instruction[1:0];
wire [7:0] imm_res, zero_ext_value;

zero_extension ZEX(
  .input_data(imm),
  .output_data(zero_ext_value)
);

assign imm_res = zero_ext_value;

wire[7:0] mux_alu_out, mux_out;
mux2 #(8) MUX_ALU(
    .input1(read_data1),
    .input2(imm_res), // Extended value
    .select(ALUSrc), // ALU Src to determine if Itype or Rtype
    .out(mux_out)
);

assign mux_alu_out = mux_out;

wire [1:0] alu_op;
assign alu_op = instruction[1:0];

wire [7:0] alu_result;

alu ALU (
```

```verilog
    .rs_data1(mux_out),
    .rs_data2(read_data2),
    .alu_op(alu_op),
    .alu_src(ALUSrc),
    .result(result)
);

assign alu_result = result;

wire [7:0] mem_data, data_out;
// Instantiate Data Memory
data_memory DM (
    .clk(clk),
    .data_in(read_data1),
    .wr(MemWrite),
    .rd(MemRead),
    .addr(alu_result),
    .data_out(data_out)
);

assign mem_data = data_out;

// Instantiate MUX_WB
mux2 #(8) MUX_WB (
    .input1(alu_result),
    .input2(mem_data),
    .select(RegWriteSrc),
    .out(mux_wb_out)
);

endmodule
```

Contributions:

UB ID	Description
anantha2	1. Implementation and Testing a. Data Path 2. Design a. Data Path of Processor b. Control Path Blue-print 3. Simulations of integration of individual components into the data path 4. Simulation setup 5. Hardware Integration

mayurraj	1. Implementation and Testing
	    a. Register File
	    b. ALU
	    c. Zero Extension
	    d. Multiplexer
	2. Report and Code Block Preparation

mmansuri	1. Implementation and Testing
	    a. Data Memory
	    b. Program Counter
	2. Report Preparation

ritikran	1. Implementation and Testing a. Instruction Memory b. Control Unit(s) 2. Simulation of All components 3. Constraint File for Hardware