# 32-Bit Single Cycle RISC-V Processor Design

Group Project

A report submitted to the

Department of Electrical and Information Engineering
Faculty of Engineering
University of Ruhuna
Sri Lanka

On 18th of February 2024

In completing the group project for the module
EE5260 - Hardware Description Language

By

EG/2020/4368 : Gurunayake G. M. R. S. B
EG/2020/4087 : Naveendi H. T
EG/2020/4096 : Nethsarani D. W. D

# Abstract

This paper presents a processor which is 32-bit single-cycle RISC-V architecture under factors of design and implementation. RISC-V is an open-source instruction set architecture (ISA) that has been growing since its introduction in 2015 in applications like embedded systems and microcontrollers. The simplicity, modularity, and scalability of this architecture cause the rapid spread of this method in the field.

A RISC-V 32I single-cycle CPU is a processor that stands for the Reduced Instruction Set Computing (RISC) architecture, utilizing a 32-bit instruction set and executing each instruction within a single clock cycle, resulting in swift and efficient instruction processing. The "32I" in its name signifies the 32-bit instruction width, striking a balance between code compactness and processing speed. First, this project designs and then implements methods for the RISC-V instruction set without applying pipelining techniques. Then the design is combined onto a field programmable gate array after the implementation is set to obtain the final results whether they are as expected or not.

# Contents

Abstract ..... i
Contents ..... ii
Introduction ..... 1
Methodology ..... 2
Architecture and Design ..... 3
Implementation ..... 1
1. Design Steps ..... 1
1.1. Execution of R-type instructions ..... 1
1.2. Execution of I-type instructions ..... 2
2. Verilog Implementation ..... 4
3. Key challenges ..... 13
Testing and Verification ..... 14
1. Testing for R-type instructions ..... 14
2. Testing for I-type instructions ..... 16
Results and Discussion ..... 18
3. Overall performance and functionality of the 32-bit RISC-V processor ..... 18
4. Limitations or areas for improvement. ..... 19
Conclusion ..... 21
Future Work ..... 22
References ..... 23

# Introduction

This project involves designing a single-cycle 32-bit RISC-V processor without pipelining. In this project, the processor will be implemented in Verilog HDL and will focus on executing R-type and I-type instructions. The design will prioritize simplicity and completeness, aiming to create a processor that is both easy to understand and capable of performing a variety of tasks efficiently.

The primary objectives of this project include gaining familiarity with Verilog HDL, understanding the principles of RISC architecture and instruction set of a RISC-V processor, designing a single-cycle 32-bit RISC processor capable of executing R-type and I-type instructions, implementing the processor in Verilog HDL and verifying the functionality of the processor using simulation. The design and implementation of the 32-bit RISC-V processor will focus on R-type and I-type instructions, with an emphasis on single-cycle execution that does not include pipelining. R-type instructions are used for arithmetic and logical operations, while I-type instructions are used for data transfer and immediate operations.

A 32-bit RISC (Reduced Instruction Set Computing) processor is a type of microprocessor that uses a simplified instruction set to perform operations. Compared to Complex Instruction Set Computing (CISC) processors, which have a larger and more complex set of instructions, RISC processors aim to execute instructions quickly by using a smaller, more streamlined set of instructions. The significance of a 32-bit RISC processor lies in its ability to balance performance and complexity. By using a reduced instruction set, RISC processors can execute instructions more quickly and efficiently, making them well-suited for applications where speed and power efficiency are crucial. Additionally, the simplicity of RISC instruction sets makes them easier to design, implement, and optimize. This simplicity also makes RISC processors more reliable and easier to debug, which is important for applications where system stability is critical. Overall, 32-bit RISC processors are widely used in a variety of applications, including embedded systems, IoT devices, consumer electronics, and more, due to their performance, efficiency, and ease of use.

# Methodology

The methodology followed for designing and implementing the 32-bit RISC-V processor involved several key steps. Initially, a thorough study was conducted on general RISC-V processor architecture and its components. Subsequently, a decision was made to focus specifically on implementing R-type and I-type instructions, narrowing down the scope to include only the essential components for these instruction types. Following this, the specific modules required for the implementation were identified, including the program counter, program counter increment (PCPlus4), instruction memory, control unit, register file, immediate generator, ALU, ALU control, Mux for ALU (MUX), and data memory. These modules were selected based on their necessity for executing R-type and I-type instructions efficiently. Next, the instruction format was defined, with all instructions containing 32 bits and conforming to the selected R-type and I-type instruction formats of the RISC-V processor. This step ensured consistency and compatibility with the chosen instruction set. Subsequently, the data path for executing these instructions was defined, outlining the flow of data and control signals between the various modules to achieve the desired functionality. This step was crucial for ensuring that the processor could correctly execute the selected instructions. Finally, the Verilog implementation was carried out using MODELSIM as the simulation tool. Verilog was used as the programming language for describing the behavior of the processor's modules and their interactions.

In this project, the tools and the programming languages used were MODELSIM and Verilog respectively. MODELSIM is a hardware simulation and debugging tool used in digital design to simulate HDL (Hardware Description Language) designs. It is commonly used for simulating and verifying Verilog, VHDL, and System Verilog designs. MODELSIM provides a graphical interface for designing test benches, running simulations, and analyzing waveform results. It allows designers to simulate the behavior of their digital designs before implementing them in hardware, helping to identify and fix design issues early in the development process. Verilog is a hardware description language used to model electronic systems. It is widely used in the design and verification of digital circuits and systems. Verilog allows designers to describe the behavior and structure of digital circuits, making it easier to design complex systems. Verilog is used in various stages of the design process, from initial concept and simulation to synthesis and implementation on FPGA (Field-Programmable Gate Array) or ASIC (Application-Specific Integrated Circuit) devices.

# Architecture and Design

The diagram below shows the data path and control signal path of our RISC V processor.

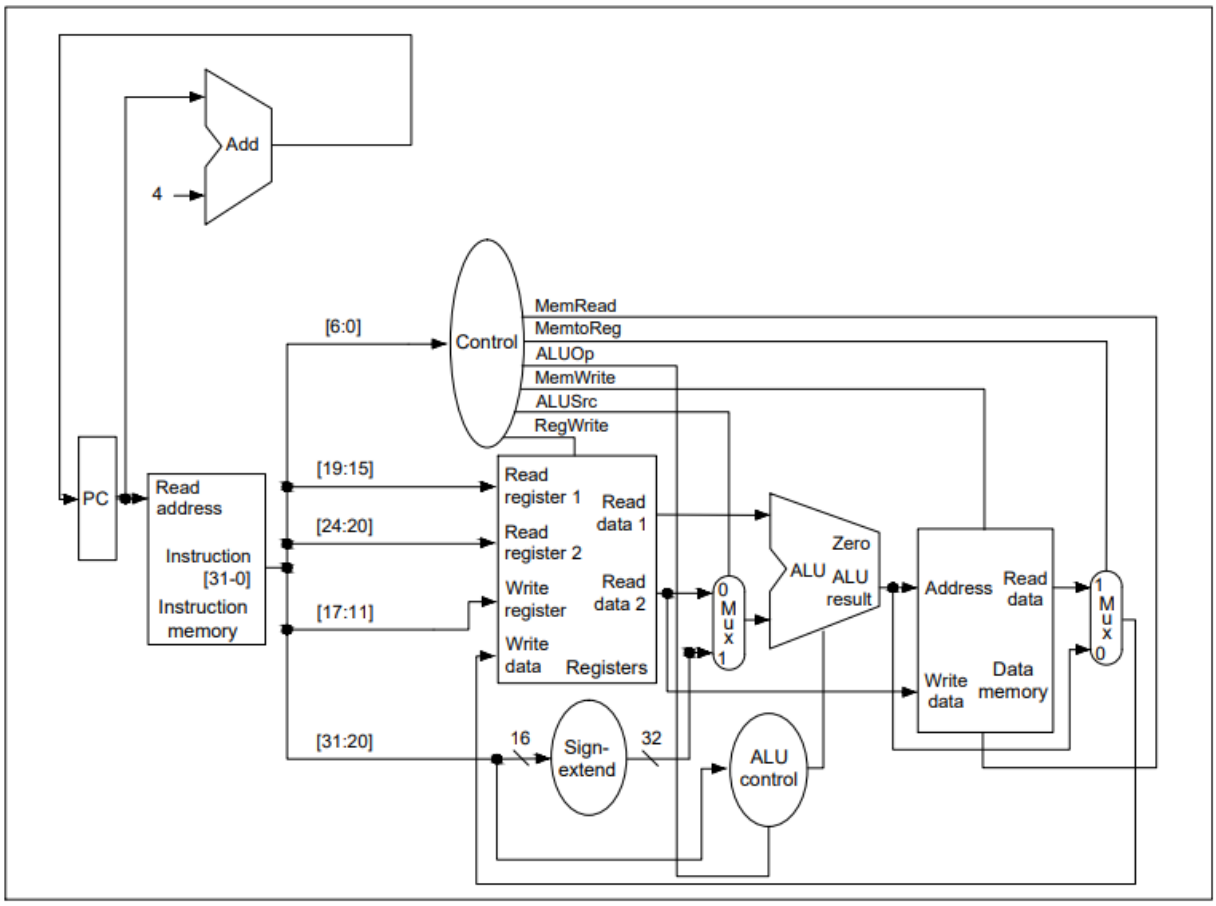



Figure 1: Datapath and control signal path

According to the diagram we could consider some main components in our designed RISC V processor.

1) Program Counter

The program counter (PC) is a register that stores the memory address of the currently executing instruction. It automatically advances after each instruction execution, ensuring that the next instruction is fetched from the subsequent memory location.

2) Instruction Memory

The instruction memory in our processor is responsible for storing the program instructions to be executed, separate from the data memory where processing data is stored. The processor fetches instructions from the instruction memory, and in the control unit, these instructions are decoded, and control signals are generated to execute them. Instructions are represented in a 32-bit fixed-length binary format, simplifying the decoding and execution process. Typically, instruction memory is implemented using read-only memory (ROM) or flash memory, ensuring that it remains unalterable during execution. In our design, the instruction memory has a size of $32 \times 64$, providing the capacity to store a substantial number of instructions for program execution.

The following figure shows the type of instructions that our processor is capable of executing.

| Type | Instruction | Name | Description |
|---|---|---|---|
| R- type | ADD | ADD | rd = rs1 + rs2 |
| | SUB | SUB | rd = rs1 - rs2 |
| | XOR | XOR | rd = rs1 ^ rs2 |
| | OR | OR | rd = rs1 \| rs2 |
| | AND | AND | rd = rs1 & rs2 |
| | SLL | Shift Left Logical | rd = rs1 << rs2 |
| | SRL | Shift Right Logical | rd = rs1 >> rs2 |
| I-type | ADDI | ADD Immediate | rd = rs1 + imm |
| | SUB | rd = rs1 - rs2 | rd = rs1 - imm |
| | XORI | XOR Immediate | rd = rs1 ^ imm |
| | ORI | OR Immediate | rd = rs1 \| imm |
| | ANDI | AND Immediate | rd = rs1 & imm |
| | SLLI | Shift Left Logical Imm | rd = rs1 << imm[0:4] |
| | SRLI | Shift Right Logical Imm | rd = rs1 >> imm[0:4] |

3) Control Unit

In a RISC-V 32I processor, the control unit plays a pivotal role by decoding instructions retrieved from the instruction memory and generating control signals to the processor's internal operations. It's responsible and it manages different types of instruction categories, including those related to registers, and immediate value operations.

Instructions in the RISC-V 32I architecture adhere to a fixed-length and binary encoding format. The control unit diligently deciphers each instruction and transmits specific control signals that govern the flow of data and instructions within the processor. These control signals act as directives, harmonizing the activities of various components within the processor, such as the Arithmetic Logic Unit (ALU), the register file, and the memory unit.

The control unit generates these control signals solely based on the instruction's opcode. Here,

- ALUOp 3-bit signal allows for the subsequent ‘ALU controller’ module to determine the ALU operation.
- MemtoReg signal is used to control the multiplexer that selects between the data from the (cache) memory and the ALU result for storing in the register during a register write.
- MemWrite signal indicates whether the processor should write data to memory.
- MemRead signal specifies whether the processor should read data from memory into a register during an instruction execution cycle.
- ALUSrc signal determines the source of the data (2nd register or immediate value) that should be given to the ALU B-port
- RegWrite signal enables writing to the register file.

4) Sign Extend

The sign extender ensures the correct handling of immediate values in instructions. It extends the sign bit of a shorter immediate value to match the required length, ensuring consistent and accurate data representation. By using the sign bit, the sign extender enables to correctly interpret immediate values in both positive and negative numbers.

5) ALU

The ALU in a RISC-V 32I processor is a versatile unit responsible for a wide array of operations, encompassing addition, subtraction, multiplication, division, bit shifting, and various bitwise logical operations such as AND, OR, and XOR. The ALU's actions are determined by the ALU Controller, which obtains its instructions from the control unit and the instruction decoder. This controller instructs the ALU on which operation to perform. The ALU then takes its input data from the register file and conducts the specified operation. The resulting output is then written back into the register file or stored in memory, depending on the specific task at hand

6) ALU Controller

In our processor, the ALU controller generates a 4-bit ALU control signal that dictates the operation to be performed by the Arithmetic Logic Unit (ALU). The ALU control signal is determined based on the opcode or specific instruction being processed, directing the ALU to perform operations such as addition, subtraction, bitwise logical operations, and more depending on the immediate instruction requirements.

7) Data Memory

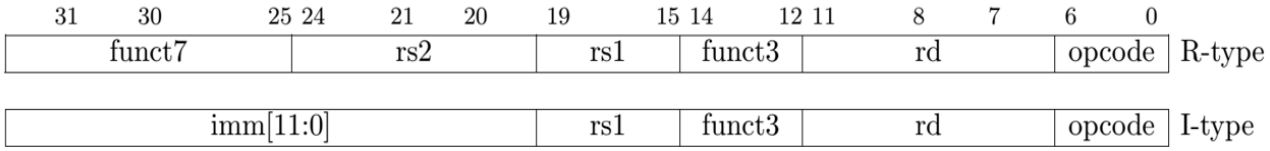
In our processor, the data memory serves as the repository for variables and data structures employed by the program. Data is fetched from the data memory when the program necessitates operations, and the results are subsequently stored back in the data memory upon completion. Typically, the data memory is structured as an extensive array of memory cells. In our design, the data memory is configured with a size of $32 \times 64$ bits and operates on a byte-addressable basis, ensuring ample storage capacity and granularity for handling data manipulation tasks

# Implementation

## 1. Design Steps

- As per the processor architecture we have selected, our first task was to identify the different components of the processor that we need to implement as modules in our Verilog file.
- In the scope of this project, our target was to design a processor that can execute R and I type instructions in the RISC V architecture. For those specifications, given below are the modules or the components we have identified.
    1) Program counter
    2) Program counter increment (PCPlus4)
    3) Instruction memory
    4) Control unit
    5) Register file
    6) Immediate generator
    7) ALU
    8) ALU control
    9) Mux for ALU (MUX)
    10) Data memory
    11) Mux for Data memory (MUX2)

- Our next step was to define the instruction format. Here, all of the instructions contained 32 bits and the selected instructions were R-type and I-type instructions. Given below are the formats of those two types of instructions.




- Our next step is to define the Datapath of the execution of these instructions. For that, let us consider how each type of instruction is executed step by step.

## 1.1. Execution of R-type instructions

- This execution consists of the following steps.
    1) Fetch the instruction address from the program counter to the instruction memory.
    2) Decode the instruction
    3) Pass rs1, rs2, rd to the inputs Read_register_1, Read_register_2 and Write_register in the register file respectively.
    4) Retrieve data from the registers rs1, rs2 to the Read_data_1 and Read_data_2 respectively.
    5) Pass the opcode to the control unit
    6) In the control unit, assign 1 to RegWrite and '111' to the ALUop outputs while the other outputs are assigned the value 0.
    7) Pass the ALUSrc to the mux connected with the ALU. Here, this value acts as the selector of the MUX between the Read_data_2 and the immediate value. For R-type instructions, the value is one and the Read_data_2 is selected from the MUX.
    8) Pass the ALUOp, func3, and func7 to the ALU control. Here, based on this value, the corresponding operation to be performed in the ALU is determined.

9) Pass the Read_data_1 directly to the ALU and the Read_data_2 via the MUX.
10) Perform the relevant operation with the two inputs based on the value given by the ALU control and pass the output to the Address input in the Data Memory and also to the MUX connected to it.
11) In the MUX connected to the data memory, the MemToReg output from the control unit acts as the selector. For R-type instructions, this value is 0 and the direct output from the ALU is selected from the MUX and given as the output and pass it to the Write_data input of the Register file.
12) Increment the program counter by 4.

- Given below is the complete data path of R-type instruction execution.

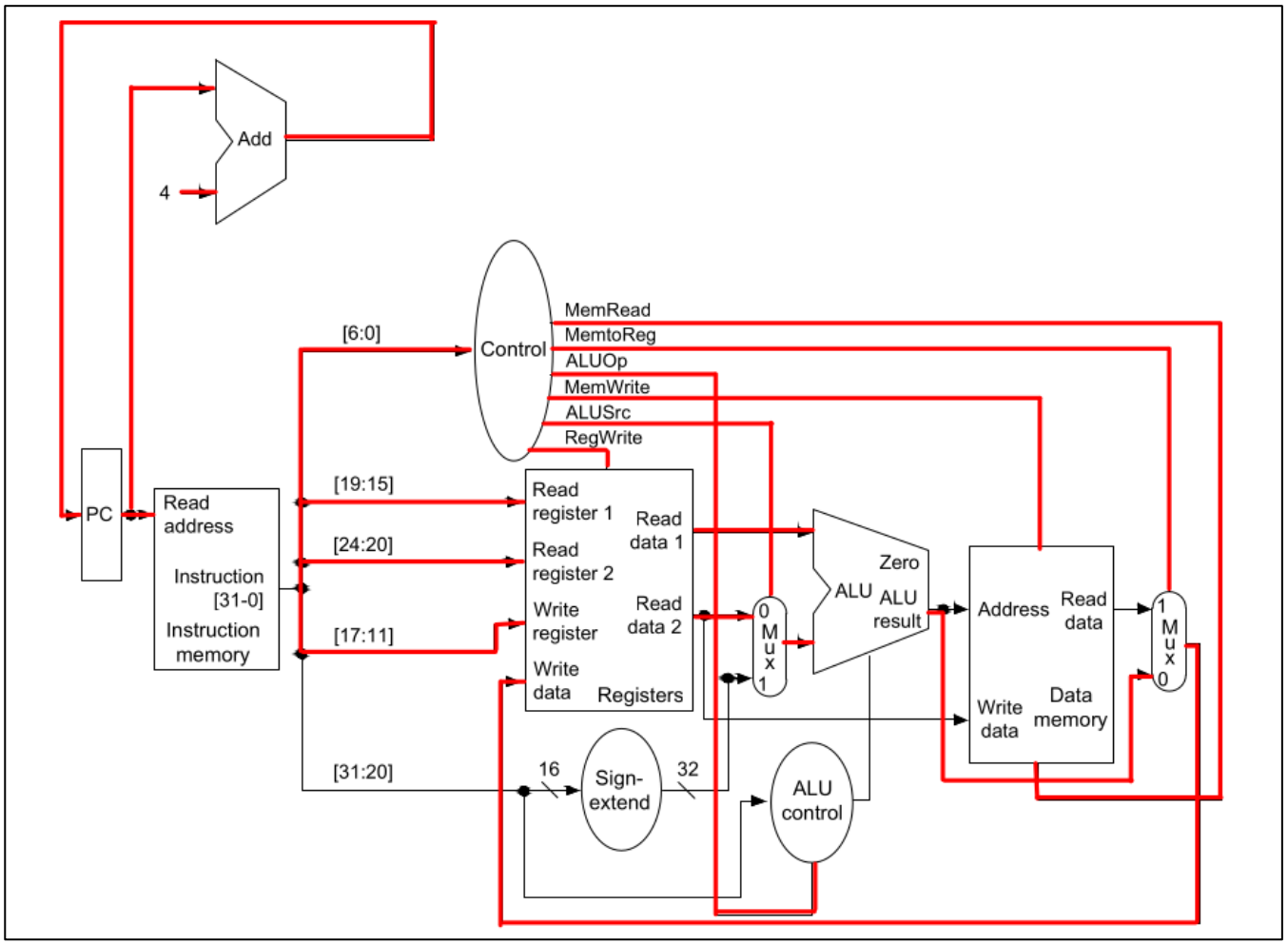



Figure 3: Datapath of a R-type instruction

## 1.2. Execution of I-type instructions

- This execution consists of the following steps.
  1) Fetch the instruction address from the program counter to the instruction memory.
  2) Decode the instruction
  3) Pass rs1 and rd to the inputs Read_register_2 and Write_register in the register file respectively.
  4) Retrieve data from the register rs1 to the Read_data_1.

5) Pass the opcode to the control unit
6) In the control unit, assign 1 to RegWrite and '000' to the ALUop outputs while the other outputs are assigned the value 0.
7) Pass the immediate value consist of 12 bits into the immediate generator and sign extend that value into 32 bits and pass the output to the mux connected to the ALU.
8) Pass the ALUSrc to the mux connected with the ALU. Here, this value acts as the selector of the MUX between the Read_data_2 and the immediate value. For I-type instructions, the value is 0 and the output from the immediate generator is selected from the MUX.
9) Pass the ALUOp to the ALU control. Here, based on this value, the corresponding operation to be performed in the ALU is determined.
10) Pass the Read_data_1 directly and the sign-extended immediate generator output via the MUX to the ALU.
11) Perform the relevant operation with the two inputs based on the value given by the ALU control and pass the output to the Address input in the Data Memory and also to the MUX connected to it.
12) In the MUX connected to the data memory, the MemToReg output from the control unit acts as the selector. For I-type instructions also, this value is 0 and the direct output from the ALU is selected from the MUX and given as the output and pass it to the Write_data input of the Register file.
13) Increment the program counter by 4.

- Given below is the complete data path of R-type instruction execution.

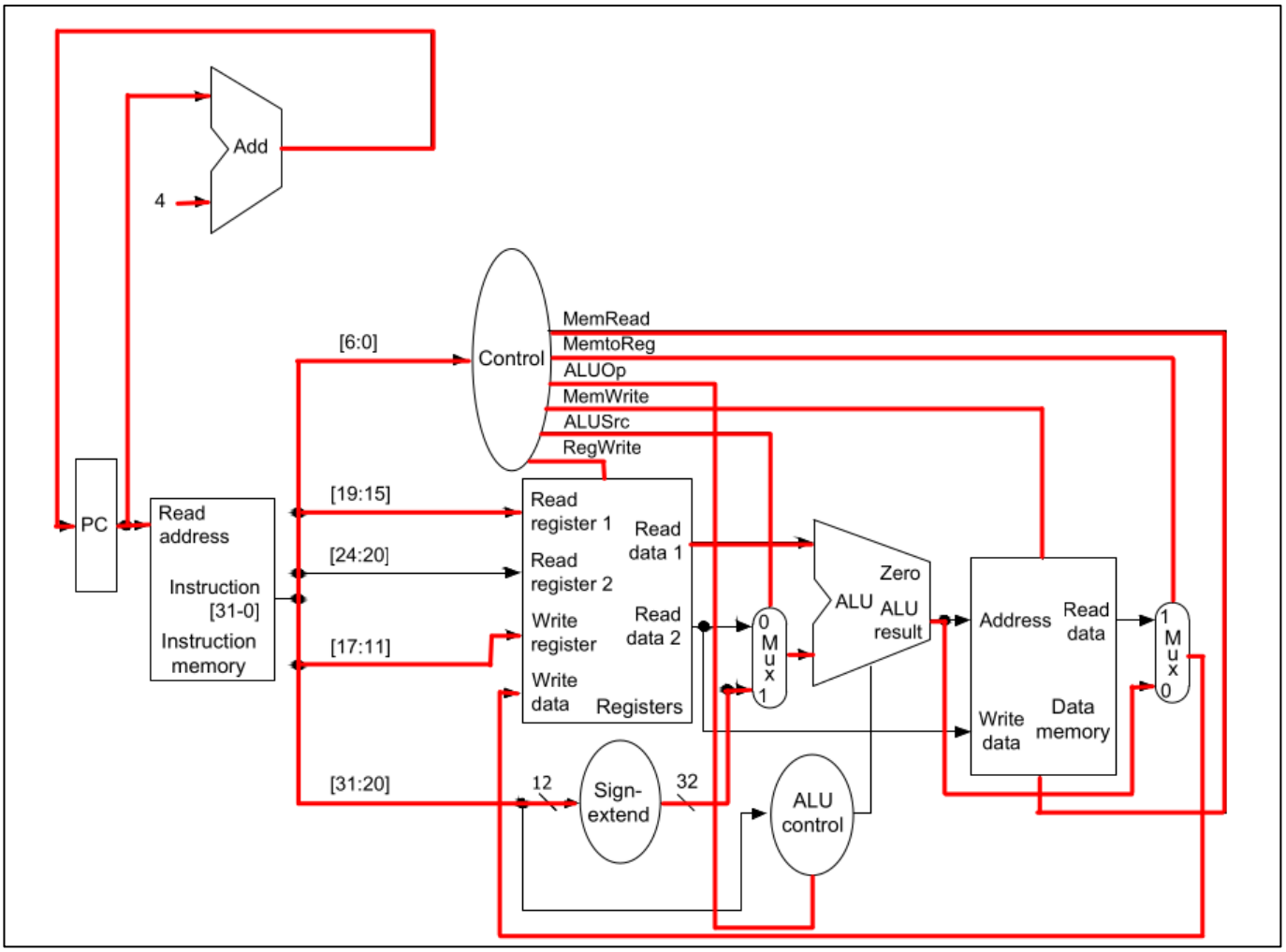



Figure 4: Datapath of a R-type instruction

## 2. Verilog Implementation

- After defining the Datapath of the instructions, our next task was to implement the processor modules using Verilog.

**Module: ProgramCounter**

```
// Program Counter

module ProgramCounter(
    input clk, reset,
    input [31:0] PC_in,
    output reg [31:0] PC_out
);

always @(posedge clk or posedge reset) begin
    if (reset == 1'b1) begin
        PC_out <= 0;
    end else begin
        PC_out <= PC_in;
    end
end

endmodule
```

**Module: Instruction_Memory**

```
// Instruction Memory

module Instruction_Memory(
    input clk, reset,
    input [31:0] read_address,
    output reg [31:0] instruction
);

reg [31:0] memory [63:0];
integer i;
assign instruction = memory[read_address];

always @(posedge clk)
begin
 if(reset) begin
  for(i=0; i<64; i=i+1) begin
   memory[i] <= 0;
  end
 end else if(~reset)
  //R - type instructions

    memory[6'd4] = 32'b0000000_01000_00111_110_01001_0110011; //SLL
    memory[6'd8] = 32'b0000001_00101_00100_110_00110_0110011; //SUB
    memory[6'd12] = 32'b0000010_00010_00001_110_00011_0110011; //ADD
    memory[6'd16] = 32'b0000101_10001_10000_110_10010_0110011; //XOR
    memory[6'd20] = 32'b0000110_10100_10011_110_10101_0110011; //SRL
    memory[6'd24] = 32'b0001000_11010_11001_110_11011_0110011; //OR
```

```
    memory[6'd28] = 32'b0001001_11101_11100_110_11110_0110011; //AND

    //I - type instructions
    memory[6'd32] = 32'b000000011001_11001_000_01001_0111111; //SLLI
    memory[6'd36] = 32'b000001000100_11101_001_11000_0111111; //SUBI
    memory[6'd40] = 32'b000000010000_00110_010_00011_0111111; //ADDI
    memory[6'd44] = 32'b000000110100_00100_011_10010_0111111; //XORI
    memory[6'd48] = 32'b000000111100_11101_100_10101_0111111; //SRLI
    memory[6'd52] = 32'b000001001110_01110_101_11011_0111111; //ORI
    memory[6'd56] = 32'b000001010111_00111_110_11110_0111111; //ANDI

end

endmodule
```

**Module: Register_File**

```
// Register File

module Register_File(
    input clk, reset, RegWrite,
    input [4:0] Rs1, Rs2, Rd,
    input [31:0] Write_data,
    output reg [31:0] read_data1, read_data2
);

reg [31:0] registers [31:0];

initial begin
 registers[0] = 32'd0;
 registers[1] = 32'd1;
 registers[2] = 32'd6;
 registers[3] = 32'd34;
 registers[4] = 32'd5;
 registers[5] = 32'd8;
 registers[6] = 32'd2;
 registers[7] = 32'd67;
 registers[8] = 32'd56;
 registers[9] = 32'd45;
 registers[10] = 32'd50;
 registers[11] = 32'd41;
 registers[12] = 32'd24;
 registers[13] = 32'd23;
 registers[14] = 32'd24;
 registers[15] = 32'd35;
 registers[16] = 32'd46;
 registers[17] = 32'd57;
 registers[18] = 32'd68;
 registers[19] = 32'd29;
 registers[20] = 32'd30;
 registers[21] = 32'd41;
 registers[22] = 32'd52;
 registers[23] = 32'd53;
 registers[24] = 32'd44;
 registers[25] = 32'd75;
```

```
 registers[26] = 32'd56;
 registers[27] = 32'd57;
 registers[28] = 32'd48;
 registers[29] = 32'd39;
 registers[30] = 32'd80;
 registers[31] = 32'd91;
end

integer i;
always @(posedge clk) begin
 if(reset) begin
  for(i=0; i<32; i=i+1) begin
   registers[i] <= 0;
  end
 end else if(~reset) begin
  if(RegWrite) begin
   registers[Rd] <= Write_data;
  end
 end
end

assign read_data1 = registers[Rs1];
assign read_data2 = registers[Rs2];

endmodule
```

**Module: Control_Unit**

```
// Control Unit

module Control_Unit(
 input reset,
 input [6:0] opcode,
 output reg RegWrite, MemtoReg, MemRead, MemWrite, Branch, ALUSrc,
 output reg [2:0] ALUOp
);

always @(*) begin
 if(reset) begin
  {ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} =
7'b0000000;
 end
 else begin
  case(opcode)
   7'b0110011: begin // R-type
    ALUSrc <= 0;
    MemtoReg <= 0;
    RegWrite <= 1;
    MemRead <= 0;
    MemWrite <= 0;
    Branch <= 0;
    ALUOp <= 3'b111;
   end
   7'b0111111: begin // I-type
    ALUSrc <= 1;
```

```
    MemtoReg <= 0;
    RegWrite <= 1;
    MemRead <= 1;
    MemWrite <= 0;
    Branch <= 0;
    ALUOp <= 3'b000;
   end
   default: begin // R-type
    ALUSrc <= 0;
    MemtoReg <= 0;
    RegWrite <= 1;
    MemRead <= 0;
    MemWrite <= 0;
    Branch <= 0;
    ALUOp <= 3'b111;
   end
  endcase
 end

end

endmodule
```

**Module: Immediate_Generator**

```
module Immediate_Generator(
    input [11:0] instruction,
    output reg [31:0] data_out
);

always @(*) begin
    // Sign extension
    if (instruction[11] == 1'b1)
        data_out = { {20{instruction[11]}}, instruction };
    else
        data_out = {20'b0, instruction};
end

endmodule
```

**Module: Mux**

```
module Mux (
 input Sel,
 input [31:0] A1, B1,
 output [31:0] Mux_out
);

assign Mux_out = (Sel==1'b0) ? A1: B1;

endmodule
```

**Module: Mux2**

```
module Mux2 (
 input Sel,
input [31:0] A2, B2,
output [31:0] Mux2_out
);

assign Mux2_out = (Sel==1'b0) ? A2: B2;
endmodule
```

**Module: ALU_Control**

```
// ALU control

module ALU_Control (
 input [2:0] ALUOp,
 input [6:0] func7,
 input [2:0] func3,
 output reg [3:0] ALUControl_out
);

always @(*) begin

 if(ALUOp == 3'b111) begin
  case ({ALUOp, func7, func3})
   13'b111_0000000_110: ALUControl_out = 4'b0001; // SLL (Shift Left
Logical) 1
   13'b111_0000001_110: ALUControl_out = 4'b0010; // SUB (Subtraction) 2
   13'b111_0000010_110: ALUControl_out = 4'b0011; // ADD (Addition) 3
   13'b111_0000101_110: ALUControl_out = 4'b0110; // XOR (Bitwise XOR) 6
   13'b111_0000110_110: ALUControl_out = 4'b0111; // SRL (Shift Right
Logical) 7
   13'b111_0001000_110: ALUControl_out = 4'b1001; // OR (Bitwise OR) 9
   13'b111_0001001_110: ALUControl_out = 4'b1010; // AND (Bitwise AND) 10
   default : ALUControl_out = 4'bxxxx; // Handle invalid inputs
  endcase
 end else begin
  case ({ALUOp, func3})
   6'b000_000: ALUControl_out = 4'b0001; // SLLI (Shift Left Logical
Immediate) 4
   6'b000_001: ALUControl_out = 4'b0010; // SUBI (Subtraction Immediate) 5
   6'b000_010: ALUControl_out = 4'b0011; // ADDI (Addition Immediate) 8
   6'b000_011: ALUControl_out = 4'b0110; // XORI (Bitwise XOR Immediate)
11
   6'b000_100: ALUControl_out = 4'b0111; // SRLI (Shift Right Logical
Immediate) 3
   6'b000_101: ALUControl_out = 4'b1001; // ORI (Bitwise OR Immediate) 13
   6'b000_110: ALUControl_out = 4'b1010; // ANDI (Bitwise AND Immediate)
14
   default : ALUControl_out = 4'bxxxx; // Handle invalid inputs
  endcase
 end
end
endmodule
```

**Module: ALU**

```
// ALU

module ALU(
    input [31:0] A, B,
    input [3:0] ALU_control_in,
    output reg [31:0] ALU_result,
    output reg zero
);

always @(ALU_control_in or A or B)
begin
 case(ALU_control_in)
  4'b0011: begin // add
   zero <= 0;
   ALU_result <= A + B;
  end
  4'b0010: begin // sub
   zero <= 0;
   ALU_result <= A - B;
  end
  4'b0001: begin // SLL - bitwise left shift
   zero <= 0;
   ALU_result <= A << B;
  end
  4'b0110: begin //XOR
   zero <= 0;
   ALU_result <= A ^ B;
  end
  4'b0111: begin //SRL - Shift Right Logic
   zero <= 0;
   ALU_result <= A >> B;
  end
  4'b1001: begin //OR - bitwise OR
   zero <= 0;
   ALU_result <= A | B;
  end
  4'b1010: begin //AND - bitwise AND
   zero <= 0;
   ALU_result <= (A & B);
  end
  default: begin // default to add
   zero <= 0;
   ALU_result <= A;
  end
 endcase
end

endmodule
```

**Module: Data_Memory**

```
// Data Memory

module Data_Memory(
 input clk, reset, MemWrite, MemRead,
 input [31:0] address, Writedata,
 output reg [31:0] Data_out
);

  reg [31:0] DataMemory [63:0];
  integer k;

  assign Data_out = (MemRead) ? DataMemory[address] : 32'b0;

  always @(posedge clk or posedge reset) begin
    if (reset == 1'b1) begin
      for (k = 0; k < 64; k = k + 1) begin
        DataMemory[k] = 32'b0;
      end
    end
    else if (MemWrite) begin
      DataMemory[address] = Writedata;
    end
  end

endmodule
```

**Module: PCPlus4**

```
// Program counter adder

module PCplus4(
    input [31:0] fromPC,
    output [31:0] nextToPC
);

assign nextToPC = fromPC + 32'h00000004;

endmodule
```

- Note that here,
  - A 32-bit ROM is created inside the Register file added each location is initialized with a value.
  - A Instruction memory consists of a memory of 32 bit sized 64 instruction locations.
  - Instructions are hard coded inside the instruction memory in order to ease the testing of the instructions.
  - Program counter is incremented by 4 instead of by 1.
  - The ALU operations are determined by the ALUOp, func3 and the func7, not solely by the opcode.

- After implementing each module, we have implemented the processor by combining all of these modules as per the design requirement.

**Module: RISCV**

```
// Top Module

module RISCV(
 input CLK, RESET
);

wire [31:0] PC_OUT_TOP,
            PC_IN_TOP,
            IMEM_OUT_TOP,
            RD1_TOP,
            ALU_MUX_OUT_TOP,
            RD2_TOP,
            ALU_OUT_TOP,
            DMEM_OUT_TOP,
            IMG_OUT_TOP,
            WRITEBACK_TOP;

wire REGWRITE_TOP,
     ALUSrc_top,
     MEMWRITE_TOP,
     MEMREAD_TOP,
     MEMTOREG_TOP;

wire [3:0] ALU_CTRL_OUT_TOP;

wire [2:0] ALUOP_TOP;

PCplus4 pc4(
 .fromPC(PC_OUT_TOP),
 .nextToPC(PC_IN_TOP)
);

ProgramCounter pc(
 .clk(CLK),
 .reset(RESET),
 .PC_in(PC_IN_TOP),
 .PC_out(PC_OUT_TOP)
);

Instruction_Memory imem(
 .clk(CLK),
 .reset(RESET),
 .read_address(PC_OUT_TOP),
 .instruction(IMEM_OUT_TOP)
);

Register_File reg_file(
```

```
 .clk(CLK),
 .reset(RESET),
 .RegWrite(REGWRITE_TOP),
 .Rs1(IMEM_OUT_TOP[19:15]),
 .Rs2(IMEM_OUT_TOP[24:20]),
 .Rd(IMEM_OUT_TOP[11:7]),
 .Write_data(WRITEBACK_TOP),
 .read_data1(RD1_TOP),
 .read_data2(RD2_TOP)
);

ALU alu(
 .A(RD1_TOP),
 .B(ALU_MUX_OUT_TOP),
 .ALU_control_in(ALU_CTRL_OUT_TOP),
 .ALU_result(ALU_OUT_TOP),
 .zero()
);

Mux mux1(
 .Sel(ALUSrc_top),
 .A1(RD2_TOP),
 .B1(IMG_OUT_TOP),
 .Mux_out(ALU_MUX_OUT_TOP)
);

ALU_Control alu_control(
 .ALUOp(ALUOP_TOP),
 .func7(IMEM_OUT_TOP[31:25]),
 .func3(IMEM_OUT_TOP[14:12]),
 .ALUControl_out(ALU_CTRL_OUT_TOP)
);

Data_Memory dmem(
 .clk(CLK),
 .reset(RESET),
 .MemWrite(MEMWRITE_TOP),
 .MemRead(MEMREAD_TOP),
 .address(ALU_OUT_TOP),
 .Writedata(ALU_MUX_OUT_TOP),
 .Data_out(DMEM_OUT_TOP)
);

Mux2 mux2(
 .Sel(MEMTOREG_TOP),
 .A2(ALU_OUT_TOP),
 .B2(DMEM_OUT_TOP),
 .Mux2_out(WRITEBACK_TOP)
);

Control_Unit control_unit(
 .reset(RESET),
 .opcode(IMEM_OUT_TOP[6:0]),
 .RegWrite(REGWRITE_TOP),
 .MemtoReg(MEMTOREG_TOP),
```

```
 .MemRead(MEMREAD_TOP),
 .MemWrite(MEMWRITE_TOP),
 .Branch(),
 .ALUSrc(ALUSrc_top),
 .ALUOp(ALUOP_TOP)
);

Immediate_Generator imdgen(
 .instruction(IMEM_OUT_TOP[31:20]),
 .data_out(IMG_OUT_TOP)
);

endmodule
```

## 3. Key challenges

- Our main challenge when implementing this processor is to define the instruction set. That is, we needed a way to differentiate the R-Type and I-Type instructions and perform ALU operations on them.
- For that, we have basically used the opcode component of the instruction. That is, we have given unique opcodes '0110011' and '0111111' to R-type and I-type instructions respectively so that we can differentiate those to at the first place.
- After that, we have given a shortened opcode of 3-bits to each of these instructions namely '111' and '000' to R-type and I-type instructions respectively as the ALUOp.
- Then, in order to differentiate the type of the ALU operation,
    - For R-type instructions, we have created a special ALUControl value using the func3, func7 and the ALUOp values.
    - For I-type instructions, we have created a special ALUControl value using the func3 and the ALUOp values
- Based on the ALUControl value, corresponding ALU operation is done on the operands.

# Testing and Verification

- To test the processor activity, we have created a test bench that provides the clock input to the processor as follows.

**Module: RISCV_TB**

```
// Testbench

module Top_tb;

reg clk, reset;

RISCV top(.CLK(clk), .RESET(reset));

initial begin
 clk = 0;
end

always #50 clk = ~clk;

initial begin
 reset = 1'b1;
 #50 reset = 1'b0;
 #400;
end

endmodule
```

- As we have included the instructions in the instruction memory itself and the initial values of the ROM in the register file, it is easy to do the testing with only the clock input. Otherwise, we need to provide the instructions explicitly to the instruction memory.
- As this processor facilitates the R-type and I-type instruction execution, we need to verify the results obtained by the processor and the expected result.

## 1. Testing for R-type instructions

- To test the R-type instructions, we have selected the following instruction which performs the addition operation.

```
memory[6'd12] = 32'b0000010_00010_00001_110_00011_0110011; //ADD
```

- Note that,
  - This instruction should be executed when the program counter output is 12.
  - We can break down the instructions as follows.

| 0000010 | 00010 | 00001 | 110 | 00011 | 0110011 |
|---|---|---|---|---|---|
| Func7 | rs2 | rs1 | Func3 | rd | opcode |
| 0000010 | 2 | 1 | 110 | 3 | 0110011 |

- The expected process should be to add the contents in locations 2 and 1 and put the result in location 3 of the ROM.
- Initial values are,

| | 00010 | 00001 | 00011 |
|---|---|---|---|
| | rs2 | rs1 | rd |
| | 2 | 1 | 3 |
| Decimal value in this location | 6 | 1 | 34 |

- Final values should be,

| | 00010 | 00001 | 00011 |
|---|---|---|---|
| | rs2 | rs1 | rd |
| | 2 | 1 | 3 |
| Decimal value in this location | 6 | 1 | 7 |

- Now consider the actual output of the processor.

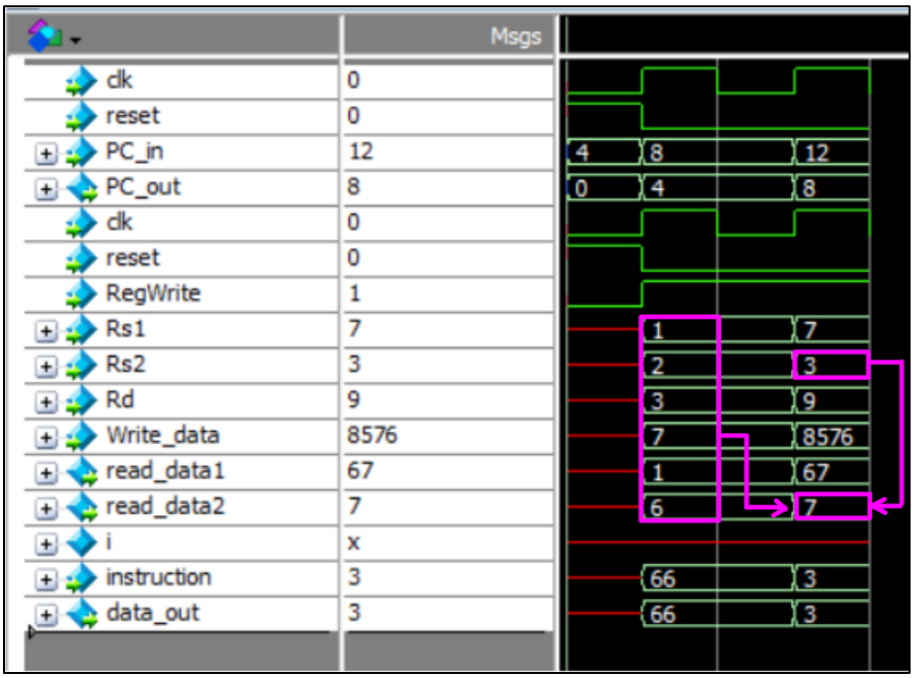



Figure 5: Processor output of the R-type Add instruction

- We can see that the observed output and the expected output are the same. So that the processor performs as expected for the R-type instructions.

## 2. Testing for I-type instructions

- To test the I-type instructions, we have selected the following instruction which performs the addition operation.

```
memory[6'd40] = 32'b000000010000_01110_010_00011_0111111; //ADDI
```

- Note that,
  - This instruction should be executed when the program counter output is 40.
  - We can break down the instructions as follows.

| 000000010000 | 01110 | 010 | 00011 | 0110011 |
|---|---|---|---|---|
| Im | rs1 | Func3 | rd | opcode |
| 16 | 14 | 110 | 3 | 0110011 |

  - Expected process should be add the content in the location 14 and the value 16 and put the result in the location 3 of the ROM.

  - Initial values are,

| | 01110 | 00011 |
|---|---|---|
| | rs1 | rd |
| | 14 | 3 |
| Decimal value in this location | 24 | 7 |

  - Final values should be,

| | 00001 | 00011 |
|---|---|---|
| | rs1 | rd |
| | 1 | 3 |
| Decimal value in this location | 24 | 40 |

- Now consider the actual output of the processor.

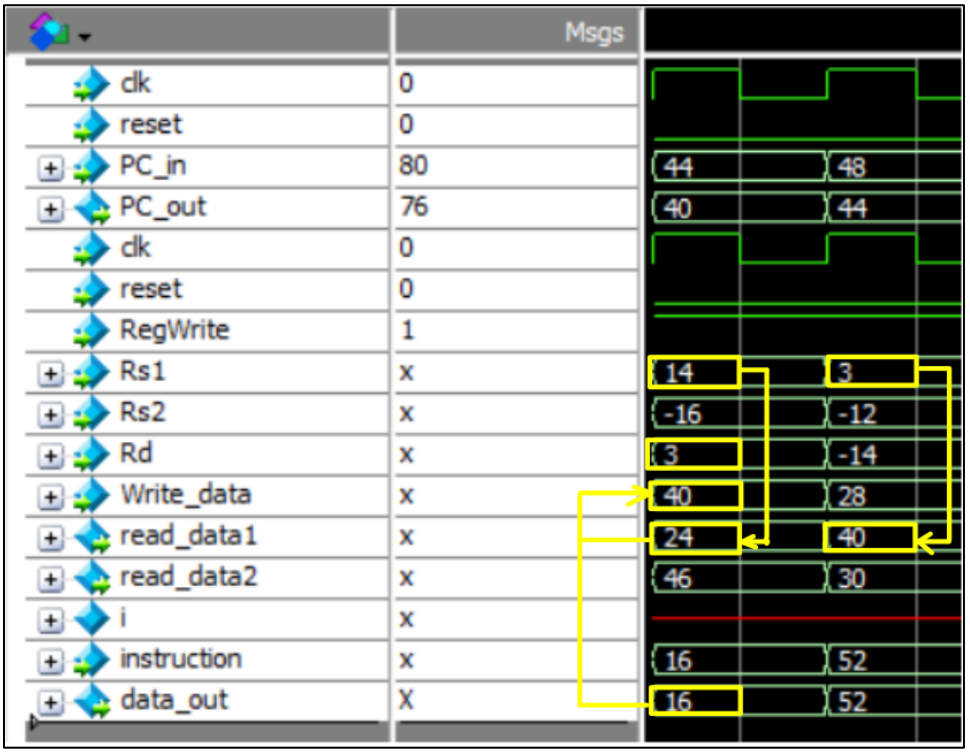



Figure 6: Processor output of the I-type Add instruction

- We can see that the observed output and the expected output are the same. So that the processor performs as expected for the I-type instructions.

# Results and Discussion

## 3. Overall performance and functionality of the 32-bit RISC-V processor

- As we have seen in the testing section, this processor functions well as expected for the ADD instruction for both R-type and I-type.
- Apart from that, we have included six more instructions namely
    - SLL (Shift Left Logical):
        - This operation shifts the bits of a binary number to the left by a certain number of positions.
        - The vacant bits introduced by shifting are filled with zeros.
        - In this processor, the value in rs1 is left shifted by bits equal to the value in rs2.
        - Eg)
            - When 67 and 7 are the values in rs1 and rs2 respectively. The result should be 8576
    - SUB (Subtraction):
        - In this processor, the value in rs1 is subtracted by the value in rs2.
        - Eg)
            - When 5 and 8 are the values in rs1 and rs2 respectively. The result should be -3
    - XOR (Exclusive OR):
        - This operation returns true (1) if and only if exactly one of the operands is true (1).
        - If both operands are the same, it returns false (0).
        - Eg)
            - When 46 and 57 are the values in rs1 and rs2 respectively. The result should be 23
    - SRL (Shift Right Logical):
        - This operation shifts the bits of a binary number to the right by a certain number of positions.
        - The vacant bits introduced by shifting are typically filled with zeros.
        - Eg)
            - When 29 and 30 are the values in rs1 and rs2 respectively. The result should be 0
    - OR:
        - This is a logical operation that performs a bitwise OR operation on each pair of corresponding bits.
        - The result is 1 if at least one of the corresponding bits is 1.
        - Eg)
            - When 75 and 56 are the values in rs1 and rs2 respectively. The result should be 123
    - AND:
        - This is a logical operation that performs a bitwise AND operation on each pair of corresponding bits.
        - The result is 1 only if both of the corresponding bits are 1.
        - Eg)
            - When 48 and 39 are the values in rs1 and rs2 respectively. The result should be 32

- We have tested and verified all these operations for both R-type and I-type as follows. Note that all the operations are in the order mentioned above.

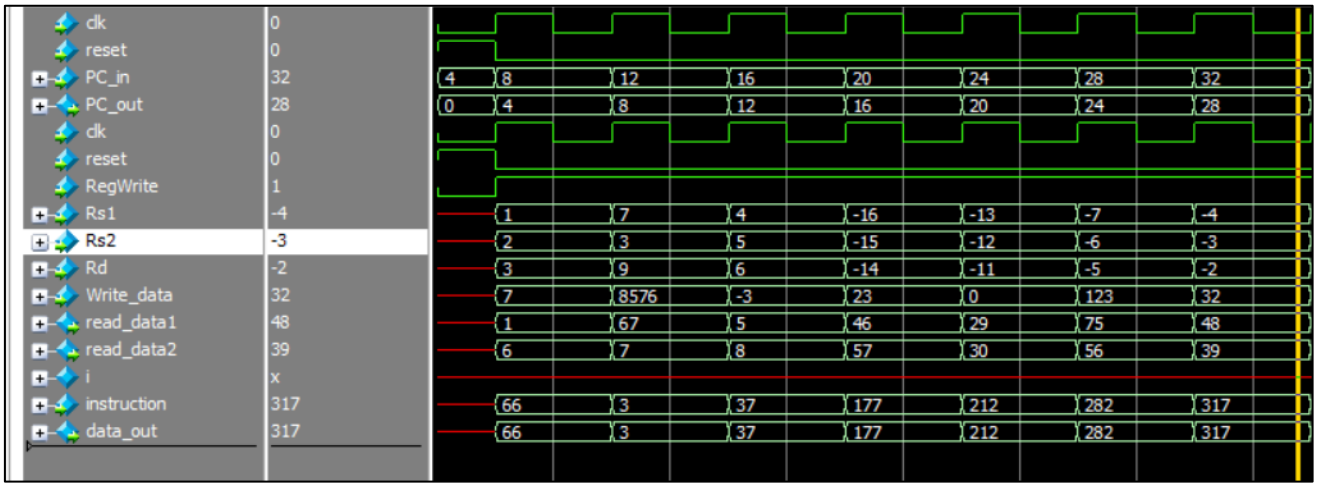



Figure 7: Verification for R-type instructions

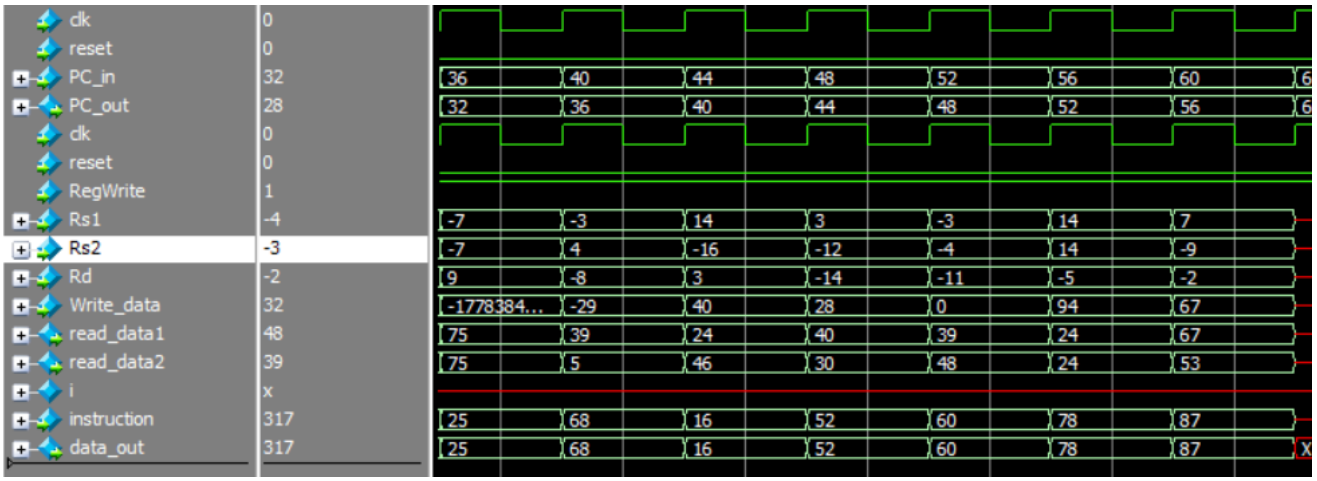



Figure 8: Verification for I-type instructions

## 4. Limitations or areas for improvement.

- The main limitation of this processor is, it is limited to perform only the arithmetic operations on R-type and I-type instructions only. But, when considering a typical RISC-V processor, there are different types of instructions that perform many complex operations. Some of them are,
    - J-Type Instructions:
        - These are jump instructions used for unconditional branching. They typically involve specifying a target address directly within the instruction rather than through a register or immediate value.
        - Examples of J-Type instructions include:
            - Jump (J)
            - Jump and Link (JAL)

- B-Type Instructions:
  - These are branch instructions used for conditional branching. They typically involve specifying a relative offset from the current program counter (PC) to determine the target address.
  - Examples of B-Type instructions include:
    - Branch if Less Than Zero (BLTZ)
    - Branch if Greater Than Zero (BGTZ)
    - Branch if Equal (BEQ)
    - Branch if Not Equal (BNE)

- S-Type Instructions (Store Type):
  - S-Type instructions are used for storing data from a register into memory.
  - They are similar to I-Type instructions, but instead of loading data from memory, they store data into memory.
  - The instruction format typically includes fields for the operation code (opcode), source register (rs1), destination register (rs2), and immediate value (imm) which specifies the memory offset.
  - Example instruction:
    - Store Word (SW)

- U-Type Instructions (Upper Immediate Type):
  - U-Type instructions are used for setting the upper bits of a register with an immediate value.
  - They are used for loading immediate values larger than the 12-bit immediate field available in I-Type instructions.
  - The instruction format typically includes fields for the operation code (opcode) and a larger immediate value (imm) which is then shifted to occupy the upper bits of a register.
  - Example instructions:
    - Load Upper Immediate (LUI)
    - Add Upper Immediate to PC (AUIPC)

# Conclusion

In this project, we have designed a RISC-V processor capable of executing R-Type and I-Type instructions using Verilog. Our main objective in this project was to familiarize ourselves with the Verilog HDL as well as get an understanding of how a RISC-V processor operates. However, considering the processor, given below are some points highlighting its significance and potential applications.

- Simplicity and Efficiency
    - RISC processors are designed with a simple and efficient instruction set architecture, making them easier to implement and optimize.
    - By focusing on R-Type and I-Type instructions, the processor design can maintain simplicity while still offering essential functionalities for arithmetic, logical operations, data transfer, and immediate value processing.
- General-purpose Computing:
    - This processor can serve as a general-purpose computing platform suitable for a wide range of applications, including embedded systems, microcontrollers, IoT devices, and low-power computing devices.
    - It can execute standard computational tasks efficiently, such as integer arithmetic, logical operations, data manipulation, and control flow operations.
- Education and Research:
    - A RISC-V processor capable of executing R-Type and I-Type instructions can be an excellent educational tool for teaching computer architecture, digital design, and Verilog HDL.
    - It can also serve as a platform for research and experimentation in processor design, optimization techniques, and computer architecture innovations.
- Customization and Specialized Applications:
    - The processor design can be customized and extended to meet specific application requirements or target specialized domains.

Overall, a 32-bit single-cycle RISC-V processor capable of executing R-Type and I-Type instructions holds significance in offering simplicity, efficiency, versatility, and openness for a wide range of computing applications and environments. Its potential applications span from general-purpose computing to specialized embedded systems, education, and research.

# Future Work

As we have discussed before, this processor is limited to the R-type and I-type instruction execution. But, because of the special architecture of these RISC processors, this can be improved such that it can perform various types of operations and other instructions types.

Expanding the capabilities of the RISC-V processor beyond R-Type and I-Type instructions opens up opportunities for enhancing its functionality and versatility. Here are some potential avenues for future work to improve the processor's capabilities:

1) Support for Additional Instruction Types:
    - Extend the instruction set architecture to support additional instruction types such as S-Type (Store Type), U-Type (Upper Immediate Type), J-Type (Jump Type), and B-Type (Branch Type) instructions.
    - Incorporate instructions for floating-point arithmetic (F-Type) to handle computations involving floating-point numbers with improved precision.
    - Introduce instructions for vector processing (V-Type) to enable parallel processing of data elements, which is beneficial for tasks such as multimedia processing, scientific computing, and machine learning.
2) Advanced Branch Prediction and Speculation:
    - Develop advanced branch prediction techniques to mitigate the impact of branch instructions on pipeline stalls, improving instruction throughput and overall performance.
    - Explore speculative execution strategies to execute instructions ahead of branch decisions, effectively hiding latency and maximizing processor utilization.

By pursuing these avenues for future work, this processor can evolve into a highly capable and adaptable computing platform, suitable for a diverse range of applications spanning from embedded systems and IoT devices to high-performance computing and data centers.

# References

[1] M. Schmalz, "Organization of Computer Systems," University of Florida, [Online]. Available: https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html. [Accessed 12 02 2024].

[2] "DEVOPEDIA," 2018. [Online]. Available: https://devopedia.org/risc-v-instruction-sets.