Project Overview:

Project Title: RISC-V-like Processor Design and Simulation

Components:

- **XILINX:** The project extensively utilizes Xilinx tools for Verilog-based coding and simulation phases.
- **LOGISIM:** Logisim serves as a supplementary tool for visualizing the RISC-V architecture and its components.
- **Instruction Fetch Unit (IFU):** Fetches instructions from memory based on the Program Counter.
- **Control Unit (CONTROL):** Decodes instructions and generates control signals.
- **Data path (DATAPATH):** Connects modules, performs ALU operations, and manages register operations.
- **Register File (REG_FILE):** Simulates registers and handles read/write operations.
- **Instruction Memory (INST_MEM):** Initializes and stores the initial set of instructions.
- **Processor (PROCESSOR):** Integrates all modules to emulate the RISC-V-like processor functionality.
- **Test bench (processor_sim4):** Drives the simulation environment and validates processor functionality.

Objective:

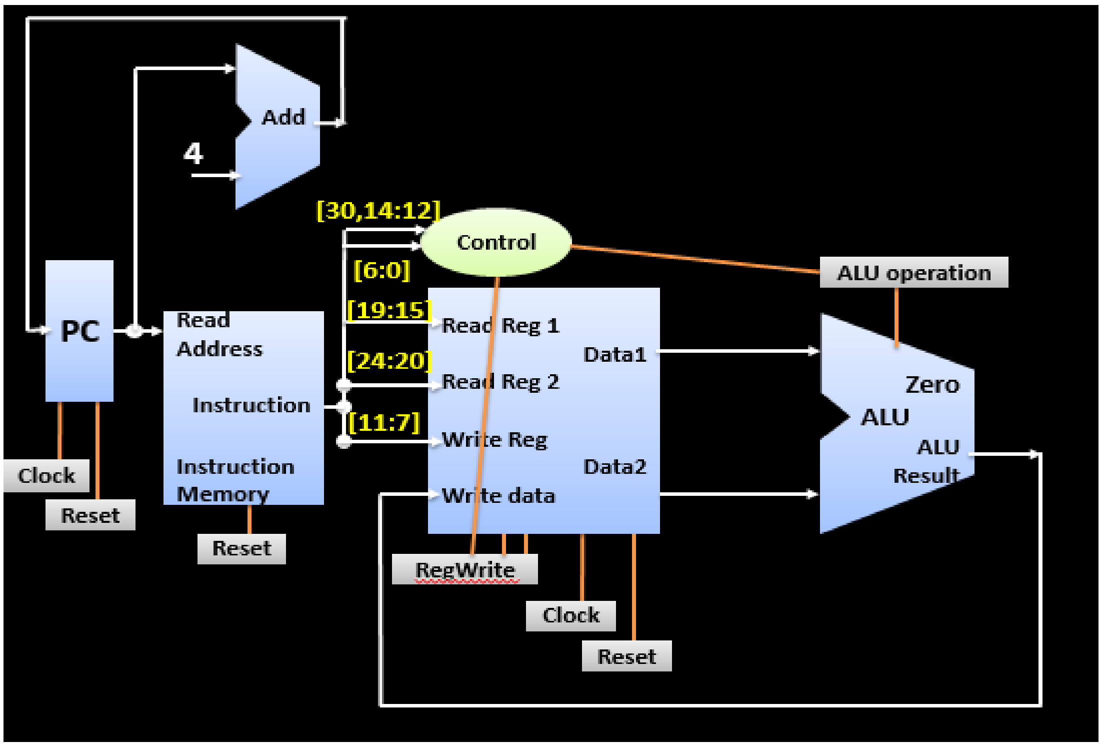
The primary objective of this project is twofold: firstly, to simulate and comprehend the fundamental principles of RISC-V architecture using Verilog HDL; and secondly, to design the logical circuitry within Logisim Tool. The project involves the development of a simplified processor design using Verilog HDL to emulate the RISC-V architecture's core principles. Additionally, it encompasses the design and visualization of logical circuitry through Logisim.

Introduction:

RISC-V, often pronounced as "risk-five," represents an open-source instruction set architecture (ISA) defining the language a processor comprehends and executes. It embodies modularity, simplicity, and adaptability, offering a blueprint for processor design. Its open nature encourages collaboration, while optional extensions cater to specialized functions. Focused on efficiency, RISC-V scales from small microcontrollers to robust servers, signifying a versatile and customizable approach to computing.

Project Implementation:

In this project we implement a 32-bit, RISC-V ISA based processor in Verilog using Xilinx. The sub-modules that are used and their interaction with each other are shown in the following picture.



Modules:

1) Instruction Memory:

```
/*
Instruction memory takes in two inputs: A 32-bit Program
counter and a 1-bit reset.
The memory is initialized when reset is 1.
When reset is set to 0, Based on the value of PC,
corresponding 32-bit Instruction code is output
*/
module INST_MEM(
    input [31:0] PC,
    input reset,
    output [31:0] Instruction_Code
);
    reg [7:0] Memory [31:0]; // Byte addressable memory with
32 locations

    // Under normal operation (reset = 0), we assign the
instr. code, based on PC
    assign Instruction_Code =
{Memory[PC+3],Memory[PC+2],Memory[PC+1],Memory[PC]};

    // Initializing memory when reset is one
    always @(reset)
    begin
        if(reset == 1)
        begin
```

```
            // Setting 32-bit instruction: add t1, s0,s1 =>
0x00940333
            Memory[3] = 8'h00;
            Memory[2] = 8'h94;
            Memory[1] = 8'h03;
            Memory[0] = 8'h33;
            // Setting 32-bit instruction: sub t2, s2, s3 =>
0x413903b3
            Memory[7] = 8'h41;
            Memory[6] = 8'h39;
            Memory[5] = 8'h03;
            Memory[4] = 8'hb3;
            // Setting 32-bit instruction: mul t0, s4, s5 =>
0x035a02b3
            Memory[11] = 8'h03;
            Memory[10] = 8'h5a;
            Memory[9] = 8'h02;
            Memory[8] = 8'hb3;
            // Setting 32-bit instruction: xor t3, s6, s7 =>
0x017b4e33
            Memory[15] = 8'h01;
            Memory[14] = 8'h7b;
            Memory[13] = 8'h4e;
            Memory[12] = 8'h33;
            // Setting 32-bit instruction: sll t4, s8, s9
            Memory[19] = 8'h01;
            Memory[18] = 8'h9c;
            Memory[17] = 8'h1e;
            Memory[16] = 8'hb3;
            // Setting 32-bit instruction: srl t5, s10, s11
            Memory[23] = 8'h01;
            Memory[22] = 8'hbd;
            Memory[21] = 8'h5f;
            Memory[20] = 8'h33;
            // Setting 32-bit instruction: and t6, a2, a3
            Memory[27] = 8'h00;
            Memory[26] = 8'hd6;
            Memory[25] = 8'h7f;
            Memory[24] = 8'hb3;
            // Setting 32-bit instruction: or a7, a4, a5
            Memory[31] = 8'h00;
            Memory[30] = 8'hf7;
            Memory[29] = 8'h68;
            Memory[28] = 8'hb3;
        end
    end

endmodule
```

Description: The INST_MEM module initializes the instruction memory upon reset, setting specific 32-bit instructions at designated memory addresses. It stores instructions that the processor fetches based on the Program Counter (PC). When reset is active, this module preloads the memory with predetermined instruction sequences, enabling subsequent instruction fetching during processor operation.

2) Register File:

```
/*
A register file can read two registers and write in to one
register.
The RISC V register file contains total of 32 registers each
of size 32-bit.
Hence 5-bits are used to specify the register numbers that
are to be read or written.
*/

/*
Register Read: Register file always outputs the contents of
the register corresponding to read register numbers
specified.
Reading a register is not dependent on any other signals.

Register Write: Register writes are controlled by a control
signal RegWrite.
Additionally the register file has a clock signal.
The write should happen if RegWrite signal is made 1 and if
there is positive edge of clock.
*/

module REG_FILE(
    input [4:0] read_reg_num1,
    input [4:0] read_reg_num2,
    input [4:0] write_reg,
    input [31:0] write_data,
    output [31:0] read_data1,
    output [31:0] read_data2,
    input regwrite,
    input clock,
    input reset
);

    reg [31:0] reg_memory [31:0]; // 32 memory locations each
32 bits wide
    integer i=0;

    //  When reset is triggered, we initialize the registers
with some values
    always @(posedge reset)
    begin
        // Bear with me for now, I tried using loops, but it
won't work
        // Just duct-taping this for now
         reg_memory[0] = 32'h0;
         reg_memory[1] = 32'h1;
         reg_memory[2] = 32'h2;
         reg_memory[3] = 32'h3;
         reg_memory[4] = 32'h4;
         reg_memory[5] = 32'h5;
         reg_memory[6] = 32'h6;
         reg_memory[7] = 32'h7;
         reg_memory[8] = 32'h8;
         reg_memory[9] = 32'h9;
         reg_memory[10] = 32'h10;
         reg_memory[11] = 32'h11;
         reg_memory[12] = 32'h12;
         reg_memory[13] = 32'h13;
         reg_memory[14] = 32'h14;
         reg_memory[15] = 32'h15;
         reg_memory[16] = 32'h16;
         reg_memory[17] = 32'h17;
         reg_memory[18] = 32'h18;
         reg_memory[19] = 32'h19;
         reg_memory[20] = 32'h20;
         reg_memory[21] = 32'h21;
         reg_memory[22] = 32'h22;
         reg_memory[23] = 32'h23;
         reg_memory[24] = 32'h24;
         reg_memory[25] = 32'h25;
             reg_memory[26] = 32'h26;
         reg_memory[27] = 32'h27;
         reg_memory[28] = 32'h28;
         reg_memory[29] = 32'h29;
         reg_memory[30] = 32'h30;
         reg_memory[31] = 32'h31;

    end

    // The register file will always output the vaules
corresponding to read register numbers
    // It is independent of any other signal
    assign read_data1 = reg_memory[read_reg_num1];
    assign read_data2 = reg_memory[read_reg_num2];

    // If clock edge is positive and regwrite is 1, we write
data to specified register
    always @(posedge clock)
    begin

        if (regwrite) begin
            reg_memory[write_reg] = write_data;
        end
    end

endmodule
```

Description: Within the REG_FILE module, a collection of 32 registers, each 32 bits wide, is managed. Upon reset, these registers initialize to predetermined values. This module facilitates reading and writing data during instruction execution, serving as a primary data storage component within the processor.

3) Instruction Fetch Unit:

```
`include "INST_MEM.v"

module IFU(
    input clock,reset,
    output [31:0] Instruction_Code,
      output  [31:0] PC
);
 reg [31:0] PC = 32'b0;  // 32-bit program counter is
initialized to zero

    // Initializing the instruction memory block
    INST_MEM instr_mem(PC,reset,Instruction_Code);

    always @(posedge clock, posedge reset)
    begin
        if(reset == 1)  //If reset is one, clear the program
counter
        PC <= 0;
        else
        PC <= PC+4;   // Increment program counter on
positive clock edge
    end

endmodule
```

Description: Responsible for fetching instructions from the instruction memory, the IFU module operates based on the PC. It fetches the next instruction in sequence, subsequently incrementing the PC to prepare for the subsequent instruction fetch during the processor's operation.

4) Control Unit:

```
module CONTROL(
    input [6:0] funct7,
    input [2:0] funct3,
    input [6:0] opcode,
    output reg [3:0] alu_control,
    output reg regwrite_control
);
    always @(funct3 or funct7 or opcode)
    begin
        if (opcode == 7'b0110011) begin // R-type
instructions

            regwrite_control = 1;
            case (funct3)
                0: begin
                    if(funct7 == 0)
                    alu_control = 4'b0010; // ADD
                    else if(funct7 == 32)
                    alu_control = 4'b0100; // SUB
                end
                6: alu_control = 4'b0001; // OR
                7: alu_control = 4'b0000; // AND
                1: alu_control = 4'b0011; // SLL
                5: alu_control = 4'b0101; // SRL
                    2: alu_control = 4'b0110; // MUL
                    4: alu_control = 4'b0111; // XOR
            endcase

      end

    end

endmodule
```

Description: The CONTROL module interprets fetched instructions, generating control signals vital for subsequent operations in the processor. By analyzing opcode and additional instruction bits, it produces control signals dictating ALU operations, register writes, and other essential functionalities within the processor.

5) **<u>Arithmetic Logic Unit:</u>**

```
/*
ALU module, which takes two operands of size 32-bits each and
a 4-bit ALU_control as input.
Operation is performed on the basis of ALU_control value and
output is 32-bit ALU_result.
If the ALU_result is zero, a ZERO FLAG is set.
*/

/*
ALU Control lines | Function
-----------------------------
        0000    Bitwise-AND
        0001    Bitwise-OR
        0010   Add (A+B)
        0100   Subtract (A-B)
        1000   Set on less than
        0011    Shift left logical
        0101    Shift right logical
        0110    Multiply
        0111    Bitwise-XOR
*/

module ALU (
    input [31:0] in1,in2,
    input[3:0] alu_control,
    output reg [31:0] alu_result,
    output reg zero_flag
);
    always @(*)
    begin
        // Operating based on control input
        case(alu_control)

        4'b0000: alu_result = in1&in2;
        4'b0001: alu_result = in1|in2;
        4'b0010: alu_result = in1+in2;
        4'b0100: alu_result = in1-in2;
        4'b1000: begin
            if(in1<in2)
            alu_result = 1;
            else
            alu_result = 0;
        end
        4'b0011: alu_result = in1<<in2;
        4'b0101: alu_result = in1>>in2;
        4'b0110: alu_result = in1*in2;
        4'b0111: alu_result = in1^in2;

        endcase

        // Setting Zero_flag if ALU_result is zero
        if (alu_result == 0)
            zero_flag = 1'b1;
        else
            zero_flag = 1'b0;

    end
endmodule
```

Description: Operating based on control signals generated by the CONTROL unit, the ALU module executes various arithmetic and logic operations. It performs tasks such as addition, subtraction, logical shifts, bitwise operations, and other specified operations, utilizing data sourced from the REG_FILE module.

6) Data Path Module:

```
`include "REG_FILE.v"
`include "ALU.v"

module DATAPATH(
    input [4:0]read_reg_num1,
    input [4:0]read_reg_num2,
    input [4:0]write_reg,
    input [3:0]alu_control,
    input regwrite,
    input clock,
    input reset,
    output zero_flag,
    output wire [31:0]read_data1,
    output wire [31:0]read_data2,
    output wire [31:0]write_data
);
    // Instantiating the register file
    REG_FILE reg_file_module(
    read_reg_num1,
    read_reg_num2,
    write_reg,
    write_data,
    read_data1,
    read_data2,
    regwrite,
    clock,
    reset
    );

    // Instanting ALU
    ALU alu_module(read_data1, read_data2, alu_control,
write_data, zero_flag);

endmodule
```

Description: The DATAPATH module serves as the overarching structure, coordinating data flow among different modules. It manages communication between the IFU, CONTROL, ALU, and REG_FILE, orchestrating the execution sequence of instructions and facilitating efficient data transfer among these core processor modules.

7) **<u>Processor:</u>**

```
`include "CONTROL.v"
`include "DATAPATH.v"
`include "IFU.v"


module PROCESSOR(
    input clock,
    input reset,
    output zero,
     // output  [31:0] instruction_code



   output wire [31:0] instruction_code,
   output wire [3:0] alu_control,
   output wire regwrite,
     output [31:0] PC,
     output [4:0] read_reg_num1,
     output [4:0] read_reg_num2,
     output [31:0] read_data1,
     output [31:0] read_data2,
     output [31:0] write_data,
     output [4:0] write_reg
);
assign read_reg_num1 = instruction_code[19:15];
assign read_reg_num2 = instruction_code[24:20];
assign write_reg = instruction_code[11:7];


    IFU IFU_module(clock, reset, instruction_code, PC);

    CONTROL control_module(instruction_code[31:25],
instruction_code[14:12],instruction_code[6:0], alu_control,
regwrite);

    DATAPATH datapath_module(instruction_code[19:15],
instruction_code[24:20], instruction_code[11:7], alu_control,
regwrite, clock, reset,
zero,read_data1,read_data2,write_data);

endmodule
```

Description: The PROCESSOR module acts as the overarching entity that integrates essential submodules to create a functioning processor design. It comprises interconnected units such as the Instruction Fetch Unit (IFU), Control Unit (CONTROL), and Datapath (DATAPATH). This module orchestrates the flow of information and control signals among these submodules, managing critical operations such as instruction fetching, decoding, execution, and data manipulation. It facilitates the seamless interaction and synchronization of individual units, playing a pivotal role in executing instructions and processing data within the simulated processor architecture.

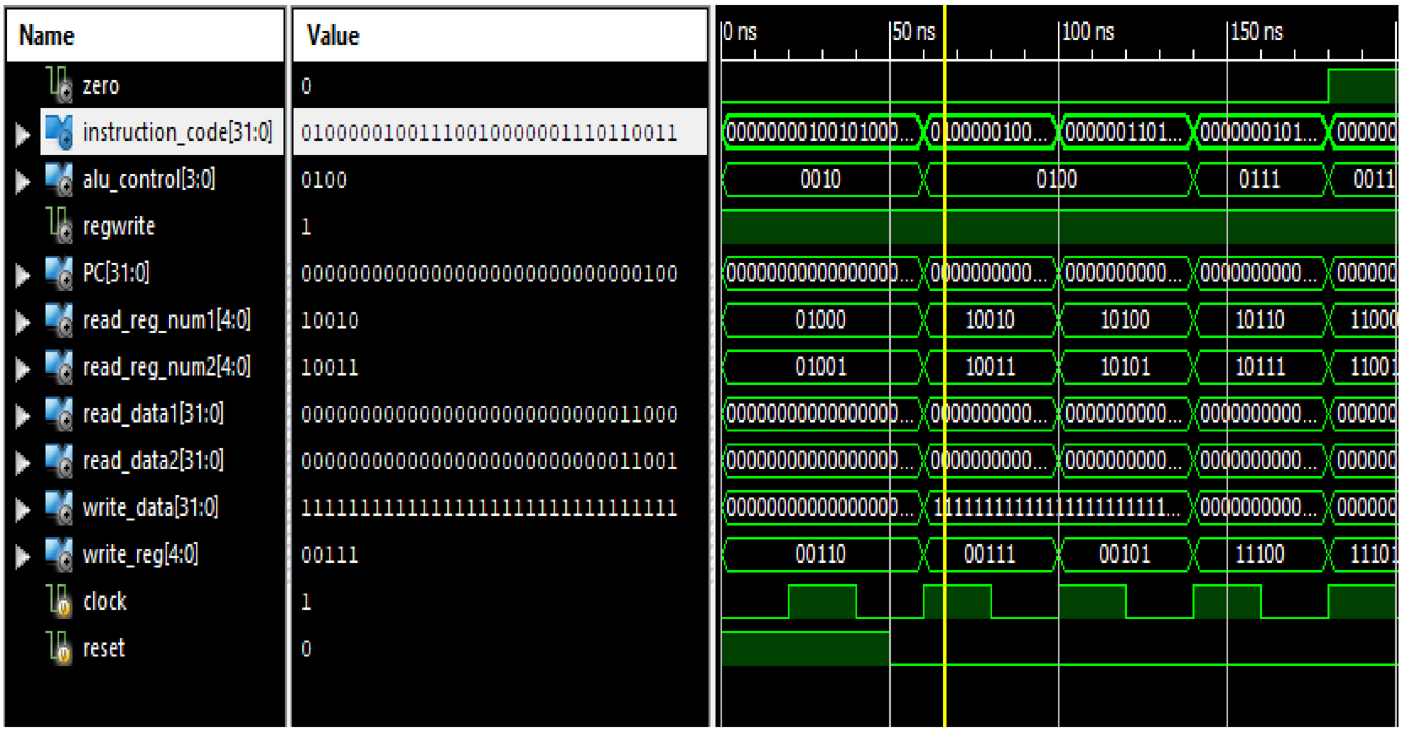
Simulation:

Code:

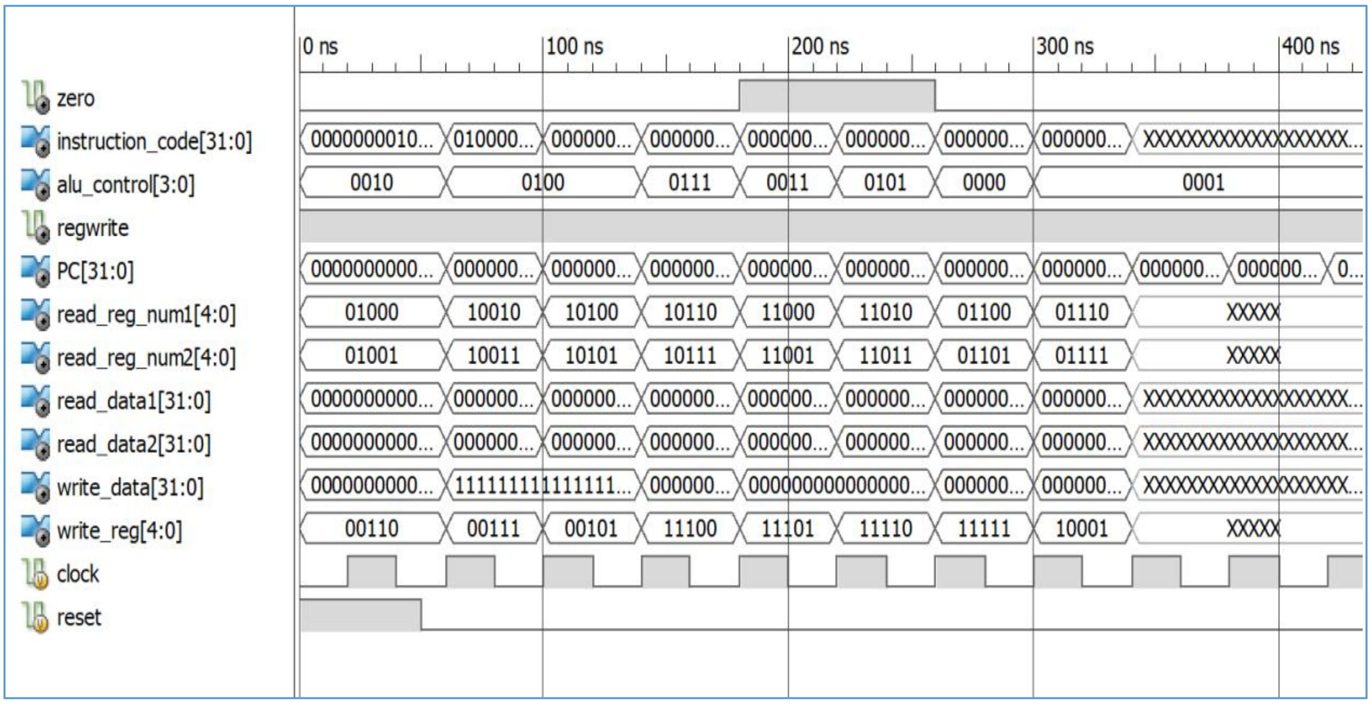
```
`timescale 1ns / 1ps
module processor_sim4;

	// Inputs
	reg clock;
	reg reset;

	// Outputs
	wire zero;
	wire [31:0] instruction_code;
	wire [3:0] alu_control;
	wire regwrite;
	wire [31:0] PC;
	wire [4:0] read_reg_num1;
	wire [4:0] read_reg_num2;
	wire [31:0] read_data1;
```

```
            wire [31:0] read_data2;
            wire [31:0] write_data;
            wire [4:0] write_reg;

            // Instantiate the Unit Under Test (UUT)
            PROCESSOR uut (
                        .clock(clock),
                        .reset(reset),
                        .zero(zero),
                        .instruction_code(instruction_code),
                        .alu_control(alu_control),
                        .regwrite(regwrite),
                        .PC(PC),
                        .read_reg_num1(read_reg_num1),
                        .read_reg_num2(read_reg_num2),
                        .read_data1(read_data1),
                        .read_data2(read_data2),
                        .write_data(write_data),
                        .write_reg(write_reg)
            );

            initial begin
                        reset = 1;
                        #50
                        reset = 0;
end
initial begin
      clock = 0;
                        forever #20 clock = ~clock;
end
      endmodule
```

Output:

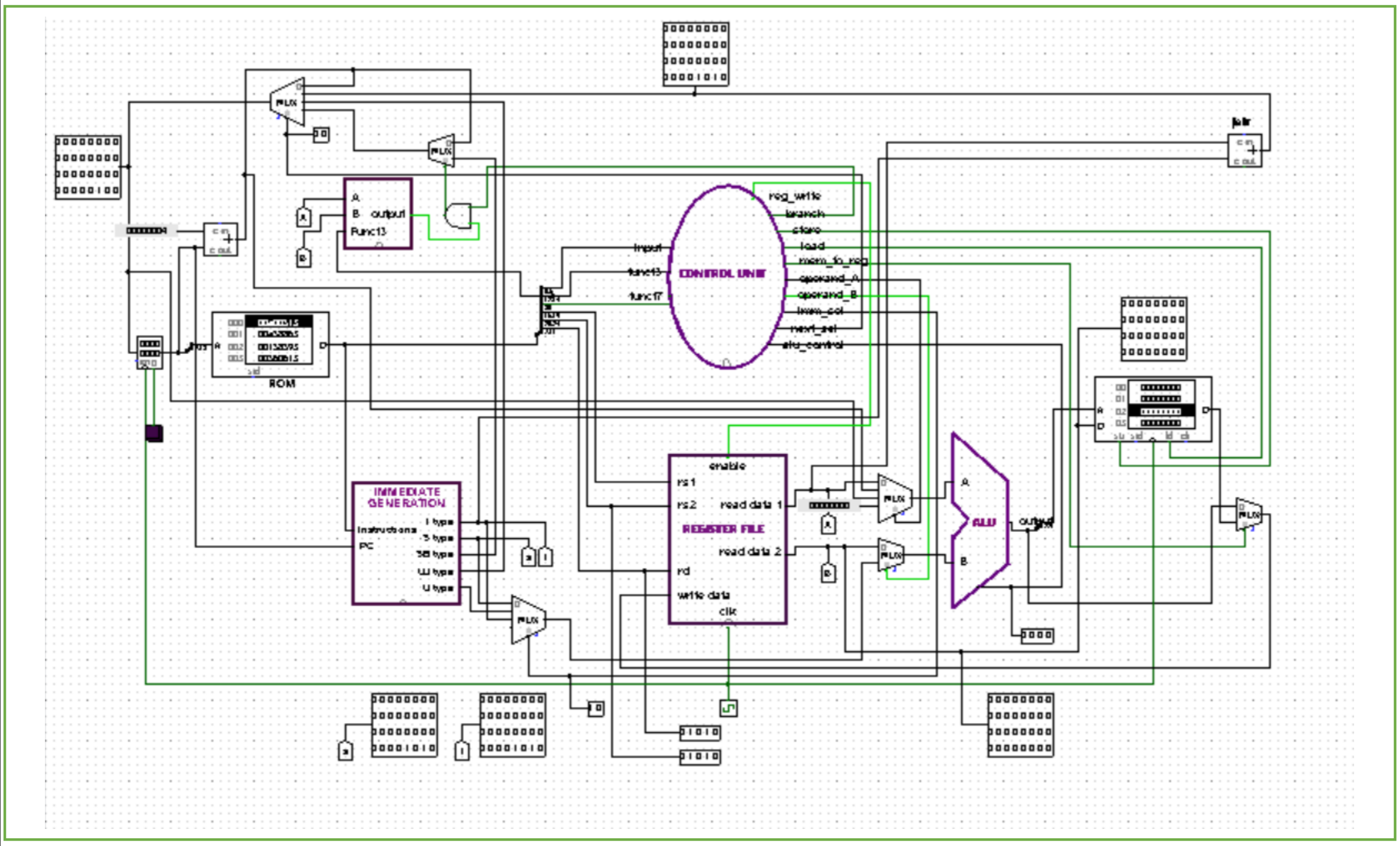


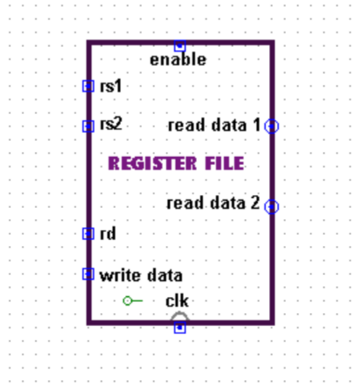
Name	Value
zero	0
instruction_code[31:0]	01000001001110010000001110110011
alu_control[3:0]	0100
regwrite	1
PC[31:0]	00000000000000000000000000000100
read_reg_num1[4:0]	10010
read_reg_num2[4:0]	10011
read_data1[31:0]	00000000000000000000000000011000
read_data2[31:0]	00000000000000000000000000011001
write_data[31:0]	11111111111111111111111111111111
write_reg[4:0]	00111
clock	1
reset	0

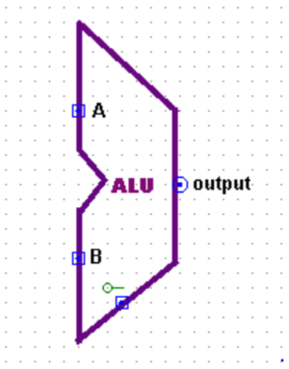


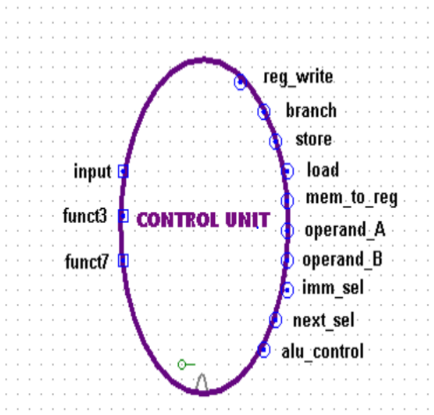
Description: This module, named processor_sim4, is a testbench module used to simulate and verify the behavior of the PROCESSOR module. It instantiates the PROCESSOR module (uut - Unit Under Test) and defines the inputs and outputs for simulation purposes. The processor_sim4 module establishes the simulation environment by initializing the clock and reset signals, emulating the clock behavior with a toggle every 20 time units, and initializing the reset signal after a delay of 50 time units to initiate the simulation of the PROCESSOR module's functionality. With the help of this testbench we can verify the different operations of our processor.

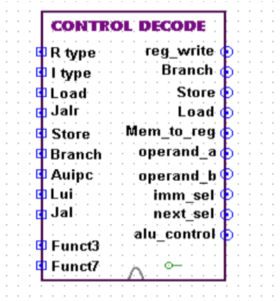
Logisim Circuit :











Register File:

The register file in RISC-V architecture is a set of registers that are used to store data temporarily during program execution. RISC-V typically has a large number of general-purpose registers, each with a specific purpose, and these registers are directly accessible by instructions. The register file plays a crucial role in the fast and efficient execution of instructions.

ALU:

The ALU is a digital circuit within the processor responsible for performing arithmetic and logical operations. In RISC-V, the ALU is designed to execute a set of basic operations such as addition, subtraction, AND, OR, and XOR. It plays a central role in executing arithmetic and logic instructions, contributing to the computational capabilities of the processor.

Control Unit:

The control unit in the RISC-V architecture is responsible for coordinating and controlling the activities of various components within the processor. It generates control signals that dictate the flow of data between registers, the ALU, and other parts of the processor. The control unit ensures that instructions are fetched, decoded, and executed in the correct sequence, maintaining the overall control flow of the program

Control Decode:

Control decode refers to the process of interpreting the control signals generated by the control unit. In RISC-V, control decode involves translating the opcode of an instruction into specific signals that control the various stages of instruction execution. This step is critical for coordinating the activities of the register file, ALU, and other components to ensure proper execution of the instruction.