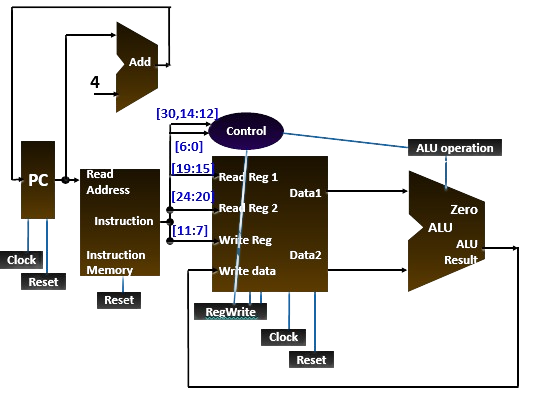
**Report on Implementation of RISC-V using Verilog HDL Project**

**1. Introduction**

The RISC-V architecture is an open-source instruction set architecture (ISA) designed for efficiency, simplicity, and scalability. It is used widely for various embedded and hardware applications. This project demonstrates the design and implementation of a RISC-V processor using Verilog Hardware Description Language (HDL). The project encompasses the design of a processor from scratch, focusing on modular design principles. The processor includes various functional blocks like the Control Unit, Arithmetic Logic Unit (ALU), Instruction Fetch Unit (IFU), Register File, and Instruction Memory.

**2. Working Principle**



The RISC-V processor fetches and executes instructions in a simple, streamlined process:

1. **Instruction Fetch**: The Instruction Fetch Unit (IFU) retrieves the next instruction from memory based on the program counter (PC).
2. **Instruction Decode**: The Control Unit decodes the fetched instruction, determining the operation to be performed by the ALU and whether data is to be written back to registers.
3. **Execute**: The ALU performs arithmetic or logical operations based on the control signals received from the Control Unit.
4. **Write Back**: The result of the ALU operation is written back to the Register File, depending on the control signals.

Each of these steps is carried out sequentially in every clock cycle, ensuring smooth execution of the program instructions.

### **Processor Design Levels:**

* **Level 1 (Higher-Level Module)**: PROCESSOR
* **Level 2 (Control and Data Flow)**: CONTROL UNIT, IFU, DATAPATH
* **Level 3 (Functional Modules)**: ALU, REGISTER FILE, INSTRUCTION MEMORY

## ****3. Processor Design****

The design of the RISC-V processor is modular, composed of three main levels:

### **Level 1: Higher-Level Module - PROCESSOR**

This is the top-level module that integrates all the control and data flow components. It serves as the backbone of the entire processor, ensuring communication between different functional blocks like the ALU, IFU, and Register File.

**Verilog Code**

`timescale 1ns / 1ps

`include "CONTROL.v"

`include "DATAPATH.v"

`include "IFU.v"

module PROCESSOR(

input clock,

input reset,

output zero

);

wire [31:0] instruction\_code;

wire [3:0] alu\_control;

wire regwrite;

IFU IFU\_module(clock, reset, instruction\_code);

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);

endmodule

**Verilog TB**

`timescale 1ns / 1ps

module tb\_Processor\_v;

// Inputs

reg clock;

reg reset;

// Outputs

wire zero;

// Instantiate the Unit Under Test (UUT)

PROCESSOR uut (

.clock(clock),

.reset(reset),

.zero(zero)

);

initial begin

reset = 1;

#50 reset = 0;

end

initial begin

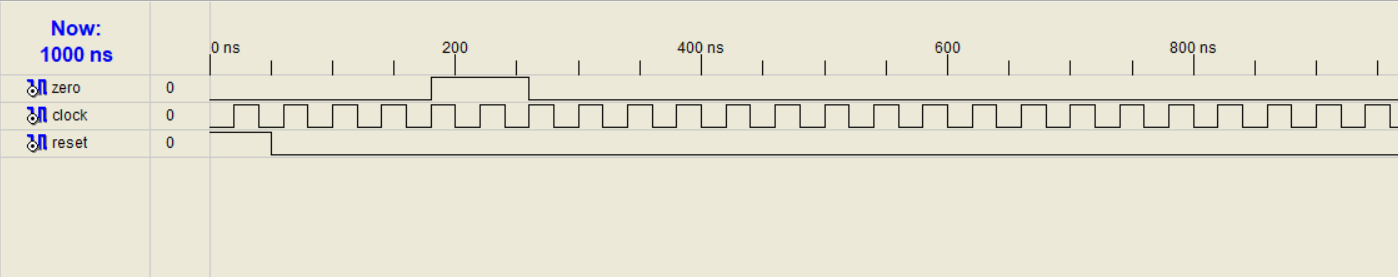
clock = 0;

forever #20 clock = ~clock;

end

endmodule

**Output**

****

### **Level 2: Control and Data Flow**

#### **Control Unit (CONTROL)**

The Control Unit is responsible for decoding the instructions and generating control signals. It takes the opcode, funct7, and funct3 fields from the instruction and controls the ALU and register write operations accordingly.

**Verilog code**

`timescale 1ns / 1ps

/\*

Control unit controls takes opcode, funct7, funct3 of the instruction code to determine

and control regwrite in IFU, alu control in ALU to execute proper instruction

\*/

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

#### **Instruction Fetch Unit (IFU)**

The IFU fetches the next instruction from memory. It contains the program counter (PC) and an adder to increment the PC by 4 after every cycle.

**Verilog code**

`timescale 1ns / 1ps

/\*

The instruction fetch unit has clock and reset pins as input and 32-bit instruction code as output.

Internally the block has Instruction Memory, Program Counter(P.C) and an adder to increment counter by 4,

on every positive clock edge.

\*/

module IFU(

input clock,reset,

output [31:0] Instruction\_Code

);

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

**Verilog tb**

`timescale 1ns / 1ps

module tb\_IFU\_v;

// Inputs

reg clock;

reg reset;

// Outputs

wire [31:0] Instruction\_Code;

// Instantiate the Unit Under Test (UUT)

IFU uut (

.clock(clock),

.reset(reset),

.Instruction\_Code(Instruction\_Code)

);

// Clock Generation: Toggle the clock every 20 time units

initial begin

clock = 0;

forever #20 clock = ~clock; // Clock toggles every 20 time units

end

// Reset Logic and Test Sequence

initial begin

// Initialize Inputs

reset = 0;

// Initial reset

#20 reset = 1; // Assert reset for 20 time units

#20 reset = 0; // Deassert reset

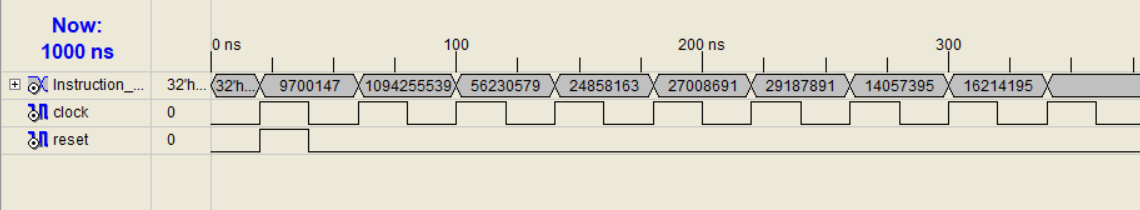
// Monitor changes in the instruction code and clock

$monitor("Time = %0t | Clock = %b | Reset = %b | Instruction\_Code = %h", $time, clock, reset, Instruction\_Code);

end

endmodule

**Output**

****Time = 40000 | Clock = 0 | Reset = 0 | Instruction\_Code = 00940333

Time = 60000 | Clock = 1 | Reset = 0 | Instruction\_Code = 413903b3

Time = 80000 | Clock = 0 | Reset = 0 | Instruction\_Code = 413903b3

Time = 100000 | Clock = 1 | Reset = 0 | Instruction\_Code = 035a02b3

Time = 120000 | Clock = 0 | Reset = 0 | Instruction\_Code = 035a02b3

Time = 140000 | Clock = 1 | Reset = 0 | Instruction\_Code = 017b4e33

Time = 160000 | Clock = 0 | Reset = 0 | Instruction\_Code = 017b4e33

Time = 180000 | Clock = 1 | Reset = 0 | Instruction\_Code = 019c1eb3

Time = 200000 | Clock = 0 | Reset = 0 | Instruction\_Code = 019c1eb3

Time = 220000 | Clock = 1 | Reset = 0 | Instruction\_Code = 01bd5f33

Time = 240000 | Clock = 0 | Reset = 0 | Instruction\_Code = 01bd5f33

Time = 260000 | Clock = 1 | Reset = 0 | Instruction\_Code = 00d67fb3

Time = 280000 | Clock = 0 | Reset = 0 | Instruction\_Code = 00d67fb3

Time = 300000 | Clock = 1 | Reset = 0 | Instruction\_Code = 00f768b3

Time = 320000 | Clock = 0 | Reset = 0 | Instruction\_Code = 00f768b3

Time = 340000 | Clock = 1 | Reset = 0 | Instruction\_Code = xxxxxxxx

#### **Datapath**

The Datapath module connects the ALU and the Register File, routing the necessary signals and data for correct instruction execution. It also includes zero flag generation.

**Verilog Code**

`timescale 1ns / 1ps

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

);

// Declaring internal wires that carry data

wire [31:0]read\_data1;

wire [31:0]read\_data2;

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

**Verilog Tb**

`timescale 1ns / 1ps

module tb\_DATAPATH\_v;

// Inputs

reg [4:0] read\_reg\_num1;

reg [4:0] read\_reg\_num2;

reg [4:0] write\_reg;

reg [3:0] alu\_control;

reg regwrite;

reg clock;

reg reset;

// Outputs

wire zero\_flag;

// Instantiate the Unit Under Test (UUT)

DATAPATH uut (

.read\_reg\_num1(read\_reg\_num1),

.read\_reg\_num2(read\_reg\_num2),

.write\_reg(write\_reg),

.alu\_control(alu\_control),

.regwrite(regwrite),

.clock(clock),

.reset(reset),

.zero\_flag(zero\_flag)

);

// Initialize Inputs

initial begin

read\_reg\_num1 = 0;

read\_reg\_num2 = 0;

write\_reg = 0;

alu\_control = 4'b0010; // Set initial ALU control

regwrite = 0;

clock = 0;

reset = 1;

// Global reset

#20 reset = 0;

end

// Apply stimulus for reading registers

initial begin

#25 read\_reg\_num1 = 0; read\_reg\_num2 = 0;

#20 read\_reg\_num1 = 0; read\_reg\_num2 = 1;

#20 read\_reg\_num1 = 0; read\_reg\_num2 = 1;

#20 read\_reg\_num1 = 1; read\_reg\_num2 = 2;

end

// Apply stimulus for writing data

initial begin

#10 regwrite = 1; // Enable writing after 10 time units

end

// Clock signal generation

initial begin

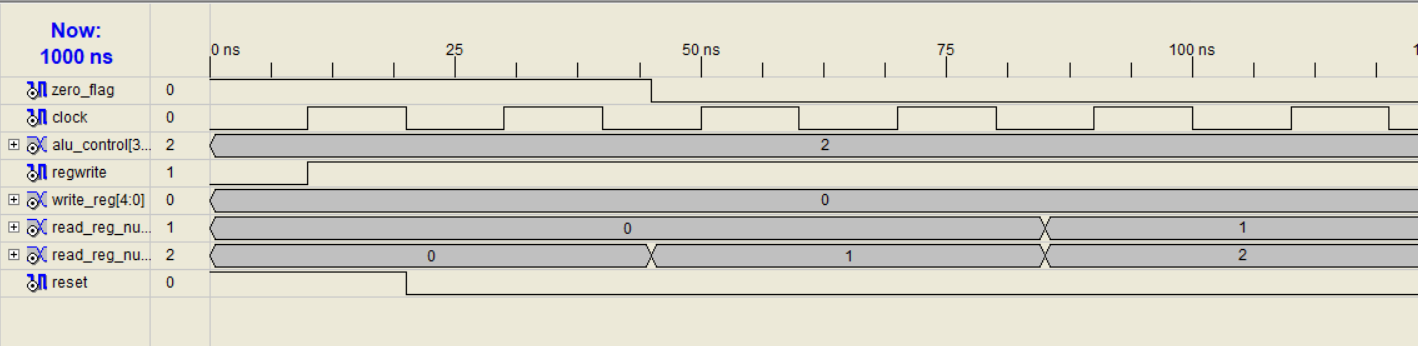
clock = 0;

forever #10 clock = ~clock;

end

endmodule

**Output**



### **Level 3: Functional Modules**

#### **Arithmetic Logic Unit (ALU)**

The ALU performs arithmetic and logical operations based on the control signals from the Control Unit. It takes two operands, applies the selected operation, and generates a result along with a zero flag if the result is zero.

**Verilog Code**

`timescale 1ns / 1ps

/\*

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

**Verilog tb**

`timescale 1ns / 1ps

module tb\_ALU\_v;

// Inputs

reg [31:0] in1;

reg [31:0] in2;

reg [3:0] alu\_control;

// Outputs

wire [31:0] alu\_result;

wire zero\_flag;

// Instantiate the Unit Under Test (UUT)

ALU uut (

.in1(in1),

.in2(in2),

.alu\_control(alu\_control),

.alu\_result(alu\_result),

.zero\_flag(zero\_flag)

);

// Initialize inputs and apply stimulus

initial begin

// Initialize Inputs

in1 = 32'd0;

in2 = 32'd0;

alu\_control = 4'b0000;

// Wait for 100 ns for global reset

#100;

// Add stimulus

in1 = 32'd23; in2 = 32'd42; alu\_control = 4'b0000; // AND operation

#20 in1 = 32'd23; in2 = 32'd42; alu\_control = 4'b0001; // OR operation

#20 in1 = 32'd23; in2 = 32'd42; alu\_control = 4'b0010; // ADD operation

#20 in1 = 32'd23; in2 = 32'd42; alu\_control = 4'b0100; // XOR operation

#20 in1 = 32'd23; in2 = 32'd42; alu\_control = 4'b1000; // SUB operation

#20 in1 = 32'd42; in2 = 32'd23; alu\_control = 4'b1000; // SUB operation (reverse)

#20 in1 = 32'd42; in2 = 32'd23; alu\_control = 4'b0100; // XOR operation (reverse)

end

// Monitor changes in values

initial begin

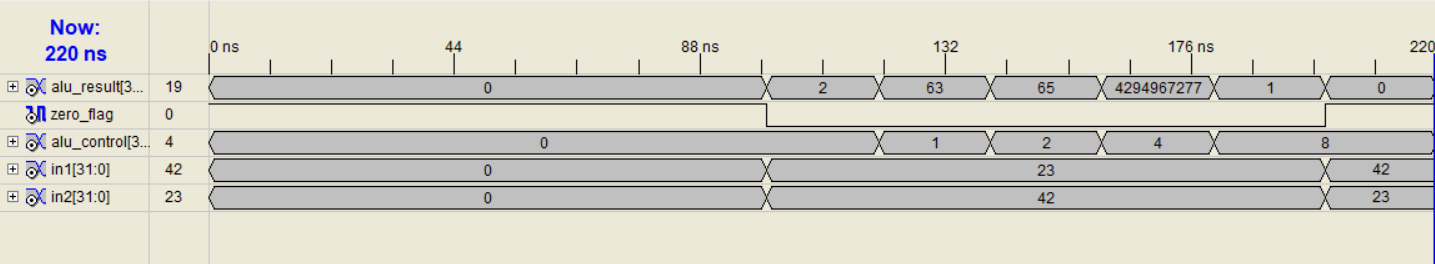
$monitor("Time: %0t | in1 = %d | in2 = %d | ALU\_Control = %b | ALU\_Result = %d | Zero\_Flag = %b",

$time, in1, in2, alu\_control, alu\_result, zero\_flag);

end

endmodule

**Output**

Time: 0 | in1 = 0 | in2 = 0 | ALU\_Control = 0000 | ALU\_Result = 0 | Zero\_Flag = 1

Time: 100000 | in1 = 23 | in2 = 42 | ALU\_Control = 0000 | ALU\_Result = 2 | Zero\_Flag = 0

Time: 120000 | in1 = 23 | in2 = 42 | ALU\_Control = 0001 | ALU\_Result = 63 | Zero\_Flag = 0

Time: 140000 | in1 = 23 | in2 = 42 | ALU\_Control = 0010 | ALU\_Result = 65 | Zero\_Flag = 0

Time: 160000 | in1 = 23 | in2 = 42 | ALU\_Control = 0100 | ALU\_Result = 4294967277 | Zero\_Flag = 0

Time: 180000 | in1 = 23 | in2 = 42 | ALU\_Control = 1000 | ALU\_Result = 1 | Zero\_Flag = 0

Time: 200000 | in1 = 42 | in2 = 23 | ALU\_Control = 1000 | ALU\_Result = 0 | Zero\_Flag = 1

Time: 220000 | in1 = 42 | in2 = 23 | ALU\_Control = 0100 | ALU\_Result = 19 | Zero\_Flag = 0

#### **Register File (REG\_FILE)**

The Register File holds the general-purpose registers of the processor. It allows two registers to be read and one to be written in each clock cycle, based on the control signals.

**Verilog Code**

`timescale 1ns / 1ps

/\*

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

**Verilog TB**

`timescale 1ns / 1ps

module tb\_reg\_file\_v;

// Inputs

reg [4:0] read\_reg\_num1;

reg [4:0] read\_reg\_num2;

reg [4:0] write\_reg;

reg [31:0] write\_data;

reg regwrite;

reg clock;

reg reset;

// Outputs

wire [31:0] read\_data1;

wire [31:0] read\_data2;

// Instantiate the Unit Under Test (UUT)

REG\_FILE uut (

.read\_reg\_num1(read\_reg\_num1),

.read\_reg\_num2(read\_reg\_num2),

.write\_reg(write\_reg),

.write\_data(write\_data),

.read\_data1(read\_data1),

.read\_data2(read\_data2),

.regwrite(regwrite),

.clock(clock),

.reset(reset)

);

// Clock generation (toggling every 10 time units)

initial begin

clock = 0;

forever #10 clock = ~clock;

end

// Initialize inputs and apply stimulus

initial begin

// Initialize inputs

reset = 1; // Reset active

regwrite = 0; // No write operations initially

read\_reg\_num1 = 0;

read\_reg\_num2 = 0;

write\_reg = 0;

write\_data = 0;

#10 reset = 0; // Deassert reset after 10 time units

// Start writing to registers

#10 regwrite = 1; write\_data = 32'd20; write\_reg = 5'd0; // Write 20 to register 0

#10 regwrite = 1; write\_data = 32'd30; write\_reg = 5'd1; // Write 30 to register 1

#10 regwrite = 1; write\_data = 32'd40; write\_reg = 5'd2; // Write 40 to register 2

#10 regwrite = 1; write\_data = 32'd50; write\_reg = 5'd3; // Write 50 to register 3

// Disable writing

#10 regwrite = 0;

// Start reading from registers

#10 read\_reg\_num1 = 5'd0; read\_reg\_num2 = 5'd1; // Read register 0 and 1

#10 read\_reg\_num1 = 5'd2; read\_reg\_num2 = 5'd3; // Read register 2 and 3

#10 read\_reg\_num1 = 5'd1; read\_reg\_num2 = 5'd2; // Read register 1 and 2

#10 read\_reg\_num1 = 5'd0; read\_reg\_num2 = 5'd3; // Read register 0 and 3

// Continue running indefinitely (no $finish)

end

// Monitor output for debug

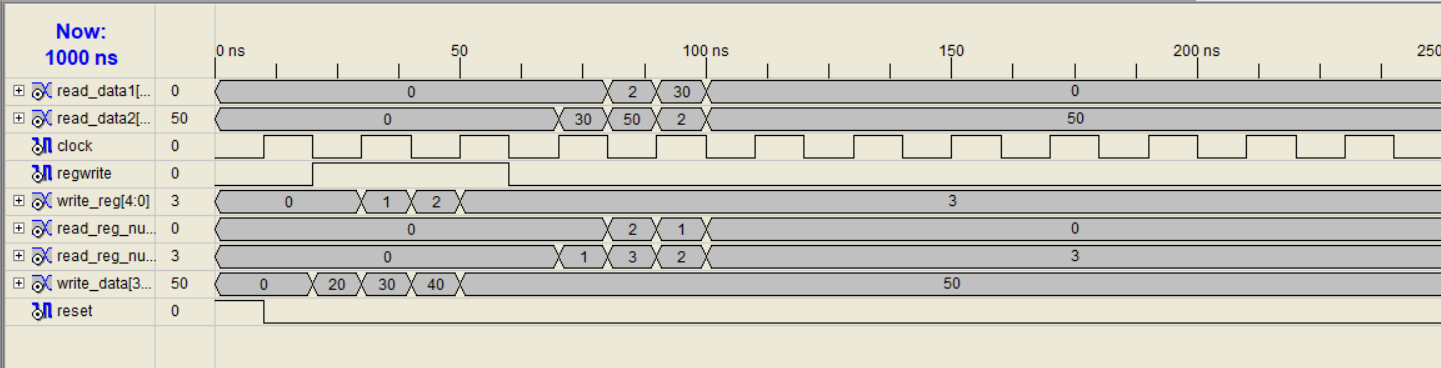
initial begin

$monitor("Time: %0t | read\_data1 = %h | read\_data2 = %h", $time, read\_data1, read\_data2);

end

endmodule

**Output**



Time: 0 | read\_data1 = 00000000 | read\_data2 = 00000000

Time: 70000 | read\_data1 = 00000000 | read\_data2 = 0000001e

Time: 80000 | read\_data1 = 00000002 | read\_data2 = 00000032

Time: 90000 | read\_data1 = 0000001e | read\_data2 = 00000002

Time: 100000 | read\_data1 = 00000000 | read\_data2 = 00000032

#### **Instruction Memory (INST\_MEM)**

The Instruction Memory stores the program instructions. It outputs the instruction corresponding to the current program counter value.

**Verilog Code**

`timescale 1ns / 1ps

/\*

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

**Verilog Tb**

`timescale 1ns / 1ps

module tb\_inst\_mem\_v;

// Inputs

reg [31:0] PC;

reg reset;

// Outputs

wire [31:0] Instruction\_Code;

// Instantiate the Unit Under Test (UUT)

INST\_MEM uut (

.PC(PC),

.reset(reset),

.Instruction\_Code(Instruction\_Code)

);

// Initialize Inputs and set up stimulus

initial begin

// Set initial values

PC = 32'd0;

reset = 1'b0;

// Apply stimulus

#20 reset = 1'b1; // Assert reset after 20 time units

#20 PC = 32'd0; // First instruction

#20 PC = 32'd4; // Second instruction

#20 PC = 32'd8; // Third instruction

#20 PC = 32'd12; // Fourth instruction

end

// Monitor the output

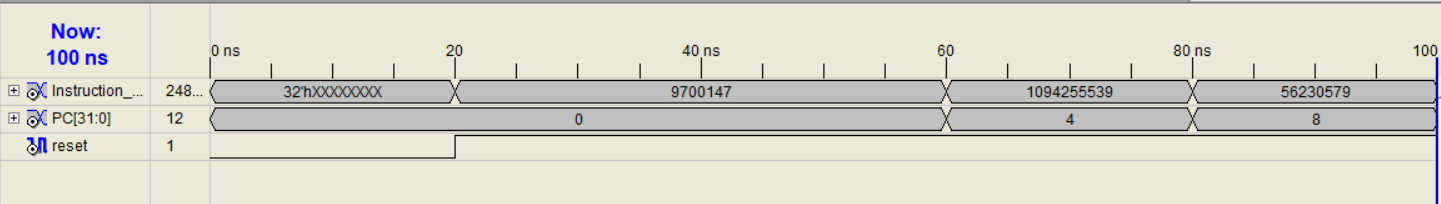
initial begin

$monitor($time, " PC = %d, Instruction Code = %h", PC, Instruction\_Code);

end

endmodule

**Output**



0 PC = 0, Instruction Code = xxxxxxxx

20 PC = 0, Instruction Code = 00940333

60 PC = 4, Instruction Code = 413903b3

80 PC = 8, Instruction Code = 035a02b3

100 PC = 12, Instruction Code = 017b4e33

## ****4. Conclusion****

The RISC-V processor was successfully implemented using Verilog HDL, and each module was tested thoroughly to ensure proper functionality. Future work could include pipelining to improve performance and extending the design to support additional instruction sets.