Assignment - 3

STATEMENT: Given that the sequence of instructions to be executed by the processor is guaranteed to be free from pipeline hazards, design a 4 – stage (Instruction Fetch; Decode and read operand; execute; write back) pipelined RISC processor that can execute following register to – register instructions with a throughput of one instruction per clock –cycle: ADD , Barrel shifter, XOR, NOR. The adder and barrel shifter, ALU designed in assignment-1 should be used here.

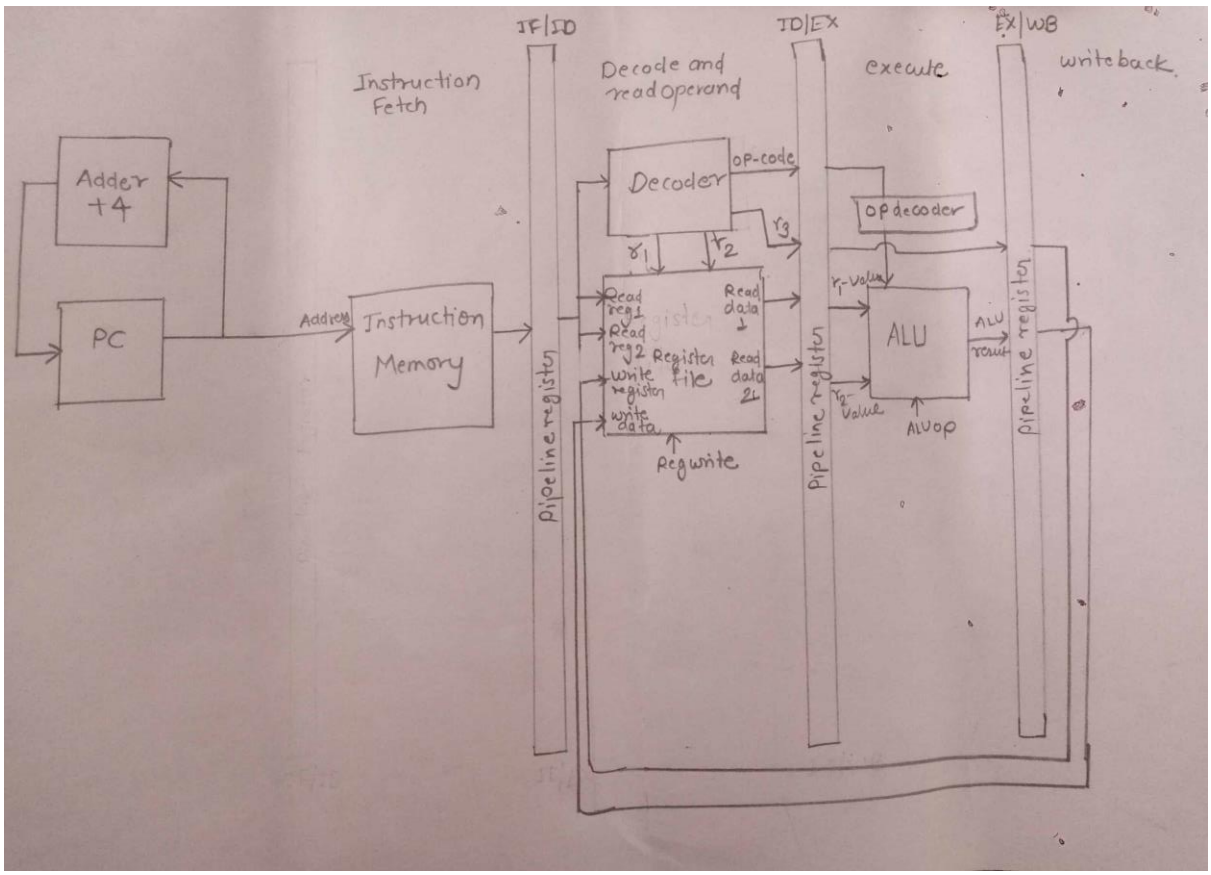
4-Stage Pipelined RISC Processor Design

Overview:

This processor is a 4-stage pipelined RISC processor executing the following register-to-register instructions: ADD, SHIFT (Barrel Shifter), XOR, and NOR. It supports a throughput of one instruction per clock cycle, assuming no hazards due to the specific ordering of instructions.

STEP 0: Draw the detailed architecture level diagram of the processor, naming and depicting the various architectural blocks (e.g. register file, instruction memory, ALU, pipeline registers, PC and combinational functional blocks etc).

Below is the architecture level diagram of the processor



STEP 1: Create Verilog behavioural models for each of the architectural blocks. (Register file, Instruction memory, ALU, Pipeline registers, PC etc)

Verilog Behavioral Models for Architectural Blocks

Below Verilog code for each block. Each module is implemented as per the specified design requirements.

1. Program Counter (PC)

The PC holds the current instruction address and increments by 4 for each instruction.

```
module PC(
    input clk,
    input reset,
    output reg [31:0] pc
);
    initial pc = 32'h0000; // Initialize PC to 0000 address

    always @(posedge clk or posedge reset) begin
        if (reset)
            pc <= 32'h0000;
        else
            pc <= pc + 4;  // Increment PC by 4
    end
endmodule
```

2. Instruction Memory

Instruction Memory with a read enable signal and 128-byte capacity, outputting a 32-bit instruction.

```
module InstructionMemory(
    input clk,
    input read_enable,
    input [31:0] address,
```

```
    output reg [31:0] instruction
);
    reg [7:0] memory [0:127]; // 128-byte memory, byte-addressable
    // Instruction initialization (program with 4 instructions as specified)
    initial begin
        memory[0]  = 8'h00; memory[1] = 8'h00; memory[2] = 8'h00; memory[3] = 8'h01;  // ADD instruction
        memory[4]  = 8'h00; memory[5] = 8'h00; memory[6] = 8'h00; memory[7] = 8'h02;  // SHIFT instruction
        memory[8]  = 8'h00; memory[9] = 8'h00; memory[10] = 8'h00; memory[11] = 8'h03; // XOR instruction
        memory[12] = 8'h00; memory[13] = 8'h00; memory[14] = 8'h00; memory[15] = 8'h04; // NOR instruction
    end
    always @(posedge clk) begin
        if (read_enable)
            instruction <= {memory[address], memory[address + 1], memory[address + 2], memory[address + 3]};
    end
endmodule
```

3. Control Unit

The Control Unit decodes the opcode and generates control signals for alu_op and write enables.

```
module ControlUnit(
    input [16:0] opcode,    // 17-bit opcode [31:15]
    output reg [1:0] alu_op,
    output reg reg_write
);

    always @(*) begin
```

```
        case (opcode)
            17'b00000000000000001: begin alu_op = 2'b00;
reg_write = 1; end // ADD
            17'b00000000000000010: begin alu_op = 2'b01;
reg_write = 1; end // SHIFT
            17'b00000000000000011: begin alu_op = 2'b10;
reg_write = 1; end // XOR
            17'b00000000000000100: begin alu_op = 2'b11;
reg_write = 1; end // NOR
            default: begin alu_op = 2'b00; reg_write = 0; end
// NOP or unrecognized opcode
        endcase
    end
endmodule
```

4. Register File

Contains 32 registers with two 32-bit read ports and one 32-bit write port.

```
module RegisterFile(
    input clk,
    input [4:0] read_addr_A, read_addr_B, write_addr,
    input [31:0] write_data,
    input reg_write,
    output reg [31:0] read_data_A, read_data_B
);
    reg [31:0] registers[0:31];
    // Register Initialization as per the given values
    initial begin
        registers[2]  = 32'd60;
        registers[5]  = 32'd40;
        registers[7]  = 32'hFFFF856D;
        registers[10] = 32'h1FFF756F;
        registers[1]  = 32'd40;
```

```
        registers[4]  = 32'd4;
        registers[8]  = 32'hEEEE3721;
        registers[11] = 32'hFFFF765E;
    end
    // Read Ports (combinational)
    always @(*) begin
        read_data_A = registers[read_addr_A];
        read_data_B = registers[read_addr_B];
    end
    // Write Port (synchronous, on falling edge)
    always @(negedge clk) begin
        if (reg_write)
            registers[write_addr] <= write_data;
    end
endmodule
```

5. ALU

The ALU performs the specified operations using a control signal from alu_op.

```
module ALU(
    input [1:0] alu_op,
    input [31:0] operand1,
    input [31:0] operand2,
    output reg [31:0] result
);
    always @(*) begin
        case (alu_op)
            2'b00: result = operand1 + operand2;       // ADD
            2'b01: result = operand1 << operand2[4:0]; //
Barrel Shift
            2'b10: result = operand1 ^ operand2;       // XOR
```

```
            2'b11: result = ~(operand1 | operand2);    // NOR
            default: result = 0;
        endcase
    end
endmodule
```

Pipeline Register Modules

Design the pipeline registers such that they are latched at the rising edge of the clock. Specify the size and format of all pipeline registers including the fields holding the decoded control signals as well as data.

1. **IF/ID Pipeline Register**: This register holds the fetched instruction and the current PC for the next stage.

```
module IF_ID(
    input clk,
    input reset,
    input [31:0] pc_in,
    input [31:0] instruction_in,
    output reg [31:0] pc_out,
    output reg [31:0] instruction_out
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc_out <= 32'b0;
            instruction_out <= 32'b0;
        end else begin
            pc_out <= pc_in;
            instruction_out <= instruction_in;
        end
    end
endmodule
```

2. **ID/EX Pipeline Register**: This register holds the control signals and operand data after decoding, ready for the execution stage.

```
module ID_EX(
  input clk,
  input reset,
  input [1:0] alu_op_in,
  input reg_write_in,
  input [31:0] pc_in,
  input [31:0] reg_data1,
  input [31:0] reg_data2,
  output reg [1:0] alu_op_out,
  output reg reg_write_out,
  output reg [31:0] pc_out,
  output reg [31:0] reg_data1_out,
  output reg [31:0] reg_data2_out
);
  always @(posedge clk or posedge reset) begin
      if (reset) begin
          alu_op_out <= 2'b0;
          reg_write_out <= 0;
          pc_out <= 32'b0;
          reg_data1_out <= 32'b0;
          reg_data2_out <= 32'b0;
      end else begin
          alu_op_out <= alu_op_in;
          reg_write_out <= reg_write_in;
          pc_out <= pc_in;
          reg_data1_out <= reg_data1;
          reg_data2_out <= reg_data2;
      end
```

```
    end
endmodule
```

3. **EX/WB Pipeline Register**: Holds the ALU result and control signals for the write-back stage.

```
module EX_WB(
    input clk,
    input reset,
    input [31:0] alu_result_in,
    input reg_write_in,
    output reg [31:0] alu_result_out,
    output reg reg_write_out
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            alu_result_out <= 32'b0;
            reg_write_out <= 0;
        end else begin
            alu_result_out <= alu_result_in;
            reg_write_out <= reg_write_in;
        end
    end
endmodule
```

STEP 2: Build the top-level **structural model** of the processor by instantiating and interconnecting the architectural blocks created in step 1.

Top-Level Module (PipelinedProcessor)

This module integrates all components, connecting pipeline registers between each stage.

```
module PipelinedProcessor(
    input clk,
    input reset
```

```
);
    // Program Counter
    wire [31:0] pc_in, pc_out;
    PC pc_inst(
        .clk(clk),
        .reset(reset),
        .pc(pc_out)
    );

    // Instruction Memory
    wire [31:0] instruction;
    InstructionMemory im_inst(
        .clk(clk),
        .read_enable(1'b1),
        .address(pc_out),
        .instruction(instruction)
    );

    // IF/ID Pipeline Register
    wire [31:0] if_id_pc_out, if_id_instruction_out;
    IF_ID if_id_reg(
        .clk(clk),
        .reset(reset),
        .pc_in(pc_out),
        .instruction_in(instruction),
        .pc_out(if_id_pc_out),
        .instruction_out(if_id_instruction_out)
    );
```

```
    // Control Unit
    wire [1:0] alu_op;
    wire reg_write;
    ControlUnit cu_inst(
        .opcode(if_id_instruction_out[31:15]),  // 17-bit
opcode
        .alu_op(alu_op),
        .reg_write(reg_write)
    );

    // Register File
    wire [4:0] read_addr_A, read_addr_B, write_addr;
    wire [31:0] read_data_A, read_data_B, write_data;
    RegisterFile rf_inst(
        .clk(clk),
        .read_addr_A(if_id_instruction_out[9:5]), // Source
register 1
        .read_addr_B(if_id_instruction_out[4:0]), // Source
register 2
        .write_addr(write_addr),
        .write_data(write_data),
        .reg_write(reg_write),
        .read_data_A(read_data_A),
        .read_data_B(read_data_B)
    );
    // ID/EX Pipeline Register
    wire [31:0] id_ex_pc_out, id_ex_reg_data1_out,
id_ex_reg_data2_out;
    wire [1:0] id_ex_alu_op;
    wire id_ex_reg_write;
    ID_EX id_ex_reg(
```

```
        .clk(clk),
        .reset(reset),
        .pc_in(if_id_pc_out),
        .reg_data1(read_data_A),
        .reg_data2(read_data_B),
        .alu_op_in(alu_op),
        .reg_write_in(reg_write),
        .pc_out(id_ex_pc_out),
        .reg_data1_out(id_ex_reg_data1_out),
        .reg_data2_out(id_ex_reg_data2_out),
        .alu_op_out(id_ex_alu_op),
        .reg_write_out(id_ex_reg_write)
    );

    // ALU
    wire [31:0] alu_result;
    ALU alu_inst(
        .alu_op(id_ex_alu_op),
        .operand1(id_ex_reg_data1_out),
        .operand2(id_ex_reg_data2_out),
        .result(alu_result)
    );

    // EX/WB Pipeline Register
    wire ex_wb_reg_write_out;
    EX_WB ex_wb_reg(
        .clk(clk),
        .reset(reset),
        .alu_result_in(alu_result),
```

```
        .reg_write_in(id_ex_reg_write),
        .alu_result_out(write_data),
        .reg_write_out(ex_wb_reg_write_out)
    );
    // Write Address Connection
    assign write_addr = if_id_instruction_out[14:10]; // 
Destination register address
    // PC Update Logic
    assign pc_in = pc_out + 4;  // Simple PC increment by 4 
for sequential execution
endmodule
```

Testbench:

```
`timescale 1ns / 1ps
module Pipeline_Testbench;
    // Clock and reset
    reg clk;
    reg reset;
    // PC signals
    reg [31:0] pc_in;
    wire [31:0] pc_out;
    // IF/ID signals
    reg [31:0] instruction_in;
    wire [31:0] pc_ifid_out;
    wire [31:0] instruction_ifid_out;
    // ID/EX signals
    reg [31:0] reg_data1, reg_data2;
    reg [4:0] rd;
    wire [31:0] pc_idex_out;
```

```
wire [31:0] reg_data1_out, reg_data2_out;
wire [4:0] rd_out;
// EX/WB signals
reg [31:0] alu_result_in;
wire [31:0] alu_result_out;
wire [4:0] rd_wb_out;
// Register file signals
reg [4:0] read_addr_A, read_addr_B, write_addr;
reg [31:0] write_data;
reg we;
wire [31:0] read_data_A, read_data_B;
// Module instantiations
PC pc_module (
    .clk(clk),
    .reset(reset),
    .pc_in(pc_in),
    .pc_out(pc_out)
);
IF_ID if_id_module (
    .clk(clk),
    .pc_in(pc_out),
    .instruction_in(instruction_in),
    .pc_out(pc_ifid_out),
    .instruction_out(instruction_ifid_out)
);
ID_EX id_ex_module (
    .clk(clk),
    .pc_in(pc_ifid_out),
    .reg_data1(read_data_A),
```

```
        .reg_data2(read_data_B),
        .rd(rd),
        .pc_out(pc_idex_out),
        .reg_data1_out(reg_data1_out),
        .reg_data2_out(reg_data2_out),
        .rd_out(rd_out)
    );
    EX_WB ex_wb_module (
        .clk(clk),
        .alu_result_in(alu_result_in),
        .rd_in(rd_out),
        .alu_result_out(alu_result_out),
        .rd_out(rd_wb_out)
    );
    RegisterFile regfile (
        .clk(clk),
        .read_addr_A(read_addr_A),
        .read_addr_B(read_addr_B),
        .write_addr(write_addr),
        .write_data(write_data),
        .we(we),
        .read_data_A(read_data_A),
        .read_data_B(read_data_B)
    );
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
```

```
    // Test sequence
    initial begin
        // Initialize inputs
        reset = 1;
        pc_in = 0;
        instruction_in = 32'h00000000;
        rd = 5'b00010;  // Set to register 2, for example
        alu_result_in = 32'h00000000;
        write_addr = 5'b00011;  // Set to register 3, for
example
        write_data = 32'h00000050;
        we = 0;
        // Apply reset
        #10 reset = 0;
        // Simulate some instructions and register operations
        #10;
        pc_in = 32'h00000004;  // PC increment
        instruction_in = 32'hABCD1234;  // Arbitrary
instruction
        read_addr_A = 5'b00010;  // Register 2
        read_addr_B = 5'b00101;  // Register 5
        #10;
        alu_result_in = 32'h0000003C;  // ALU result example
        we = 1;  // Enable write
        write_addr = 5'b00011;  // Register 3
        write_data = alu_result_in;
        #10 we = 0;  // Disable write
        // End simulation
        #100 $finish;
    end
```

```
    // Monitor outputs
    initial begin
        $monitor("Time = %0t | PC = %h | Instruction = %h |
Reg1 = %h | Reg2 = %h | ALU Out = %h",
                 $time, pc_out, instruction_ifid_out,
reg_data1_out, reg_data2_out, alu_result_out);
    end
     // Waveform dump
    initial begin
        $dumpfile("pipeline_waveform.vcd");  // Specify the
VCD file name
        $dumpvars(0, Pipeline_Testbench);    // Dump all
variables in the testbench
    end
endmodule
```

Note: The adder and barrel shifter, ALU designed in assignment-1 used here, so did not added that code here. modified pipelined processor code multiple times, to check that it works properly or not. So, there may be some mismatch the actual files simulated and added result.

To Specify the size and format of all pipeline registers including the fields holding the decoded control signals as well as data.

Control Signals and Their Sizes:

1. **ID/EX Control Signals:** These control signals are set during the instruction decode stage and propagate to the execute stage. They control various components such as the ALU, memory, and register writeback.

Common control signals for ID/EX:

- **ALUOp** (2 bits): Specifies the ALU operation (e.g., ADD, SUB, AND, OR, etc.).
- **ALUSrc** (1 bit): Determines whether the second operand to the ALU is from a register or an immediate value.
- **MemRead** (1 bit): Enables reading from data memory.
- **MemWrite** (1 bit): Enables writing to data memory.

- **RegWrite** (1 bit): Enables writing to a register in the register file.
- **MemToReg** (1 bit): Controls whether data to be written back comes from memory (for load instructions) or from the ALU (for other instructions).
- **Branch** (1 bit): Specifies if a branch instruction is being executed.
- **Jump** (1 bit): Specifies if a jump instruction is being executed.
- **RegDst** (1 bit): Determines if the destination register address is from the rt field or the rd field of the instruction.

Total control signal bits for ID/EX:
Assuming each control signal is 1 bit (except for ALUOp which is 2 bits), the total size of control signals is:

- ALUOp (2 bits)
- ALUSrc (1 bit)
- MemRead (1 bit)
- MemWrite (1 bit)
- RegWrite (1 bit)
- MemToReg (1 bit)
- Branch (1 bit)
- Jump (1 bit)
- RegDst (1 bit)

Total control signal bits = 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = **11 bits**

So, the control signal size in the ID/EX pipeline register is **11 bits**.

2. **EX/MEM Control Signals:** These control signals are set during the execute stage and propagate to the memory stage, where they control the memory access and register file writeback.

Common control signals for EX/MEM:

- **MemRead** (1 bit): Enables reading from data memory.
- **MemWrite** (1 bit): Enables writing to data memory.
- **RegWrite** (1 bit): Enables writing to a register in the register file.
- **MemToReg** (1 bit): Controls whether data to be written back comes from memory or the ALU.
- **Branch** (1 bit): Specifies if a branch instruction is being executed.

- **Jump** (1 bit): Specifies if a jump instruction is being executed.

Total control signal bits for EX/MEM:

- MemRead (1 bit)
- MemWrite (1 bit)
- RegWrite (1 bit)
- MemToReg (1 bit)
- Branch (1 bit)
- Jump (1 bit)

Total control signal bits = 1 + 1 + 1 + 1 + 1 + 1 = **6 bits**

So, the control signal size in the EX/MEM pipeline register is **6 bits**.

Command used:

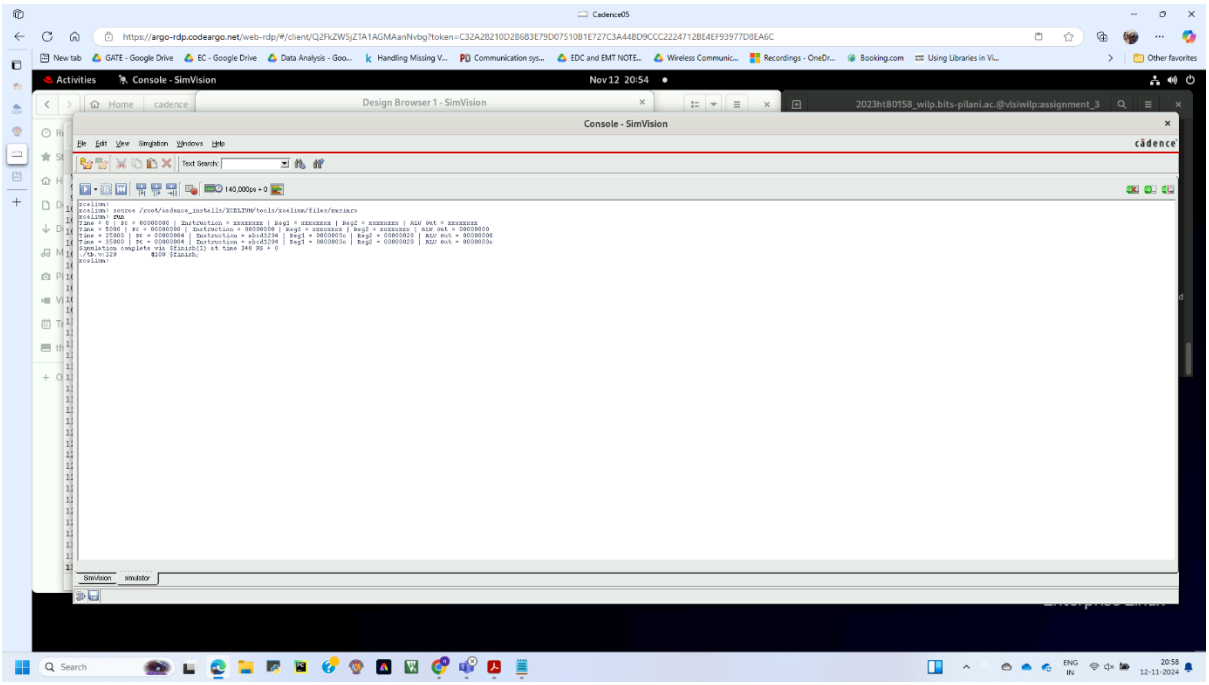
```
xrun adder.v bs.v alu.v id_ex_reg.v if_id_reg.v inst_mem.v
pc.v reg_file.v tb.v ex_wb_reg.v top_module.v and.v
nand_gate.sv 2_1_mux.sv 4_1_mux.sv or_gate.sv xor_gate.sv
and.v -gui -access +rwc
```

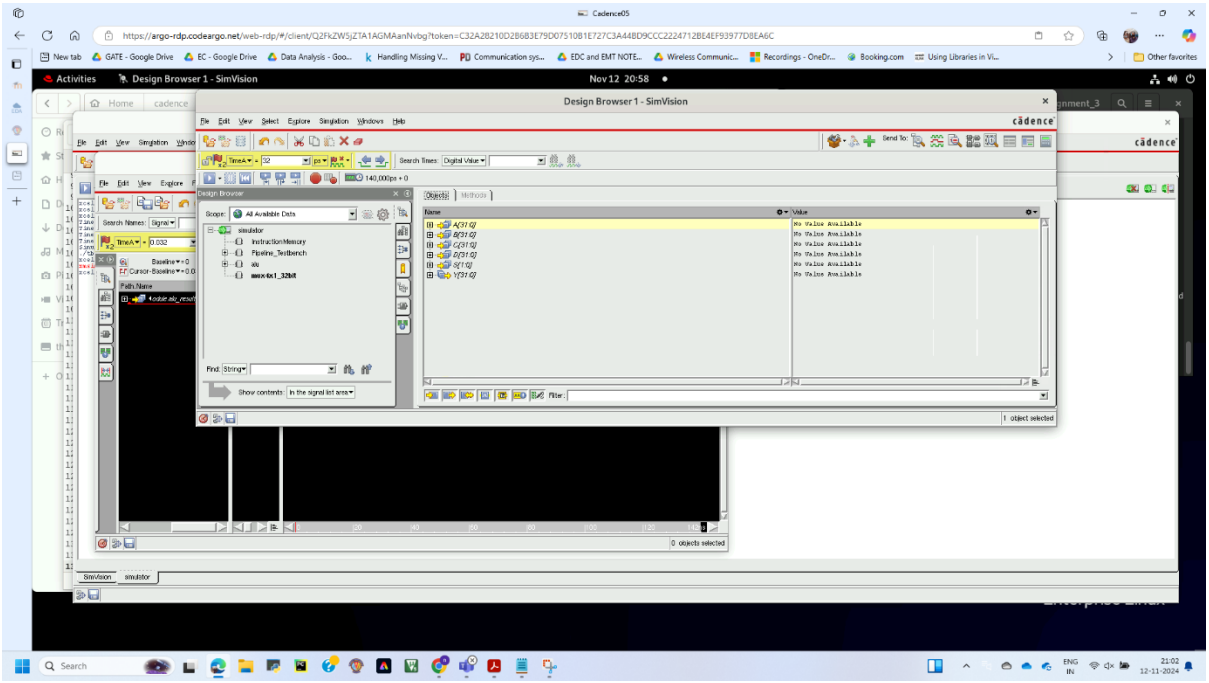
OUTPUT

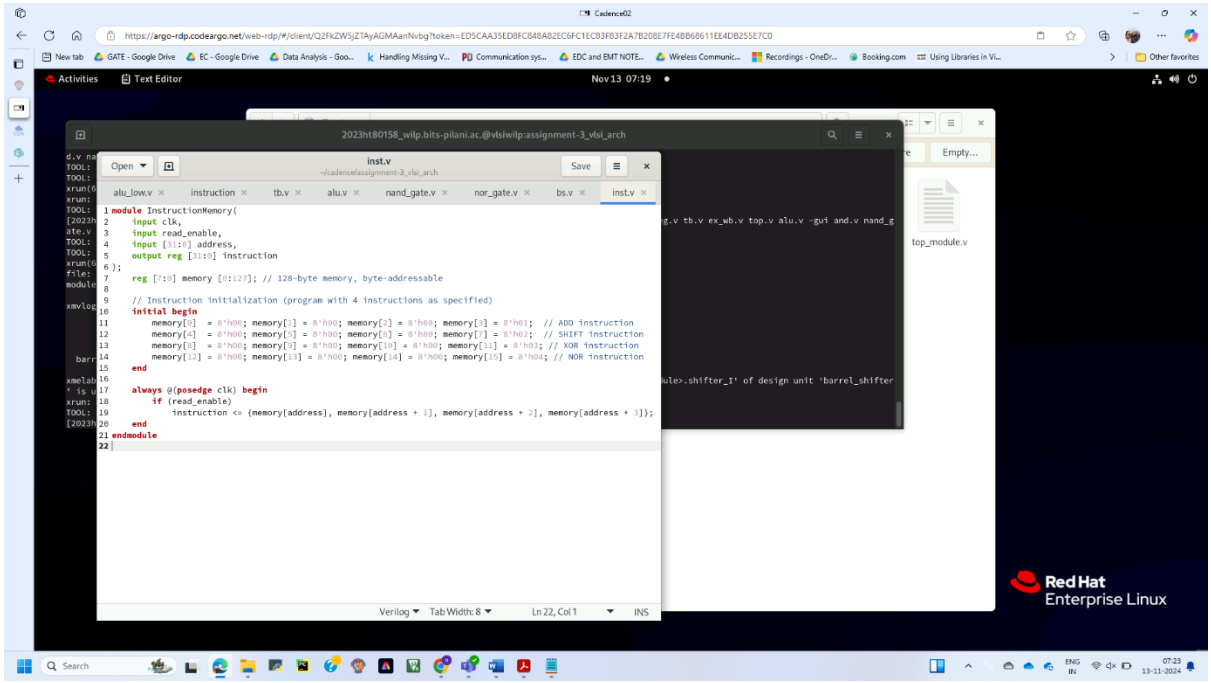
```
xcelium>
xcelium> source
/root/cadence_installs/XCELIUM/tools/xcelium/files/xmsimrc
xcelium> run
Time = 0 | PC = 00000000 | Instruction = xxxxxxxx | Reg1 =
xxxxxxxx | Reg2 = xxxxxxxx | ALU Out = xxxxxxxx
Time = 5000 | PC = 00000000 | Instruction = 00000000 | Reg1 =
xxxxxxxx | Reg2 = xxxxxxxx | ALU Out = 00000000
Time = 25000 | PC = 00000004 | Instruction = abcd1234 | Reg1 =
0000003c | Reg2 = 00000028 | ALU Out = 00000000
Time = 35000 | PC = 00000004 | Instruction = abcd1234 | Reg1 =
0000003c | Reg2 = 00000028 | ALU Out = 0000003c
Simulation complete via $finish(1) at time 140 NS + 0
./tb.v:120         #100 $finish;
xcelium> run
```

This assignment performed in remote lab:

Simulation result:






Cadence02
Activities
Text Editor
Nov 13 07:19
2023ht80158_wilp.bits-pilani.ac.@vlsiwilp:assignment-3_vlsi_arch
Open
inst.v
~/cadence/assignment-3_vlsi_arch
Save
alu_low.v
instruction
tb.v
alu.v
nand_gate.v
nor_gate.v
bs.v
inst.v
1 module InstructionMemory(
2 input clk,
3 input read_enable,
4 input [31:0] address,
5 output reg [31:0] instruction
6);
7 reg [7:0] memory [0:127]; // 128-byte memory, byte-addressable
8
9 // Instruction initialization (program with 4 instructions as specified)
10 initial begin
11 memory[0] = 8'h00; memory[1] = 8'h00; memory[2] = 8'h00; memory[3] = 8'h01; // ADD instruction
12 memory[4] = 8'h00; memory[5] = 8'h00; memory[6] = 8'h00; memory[7] = 8'h02; // SHIFT instruction
13 memory[8] = 8'h00; memory[9] = 8'h00; memory[10] = 8'h00; memory[11] = 8'h03; // XOR instruction
14 memory[12] = 8'h00; memory[13] = 8'h00; memory[14] = 8'h00; memory[15] = 8'h04; // NOR instruction
15 end
16
17 always @(posedge clk) begin
18 if (read_enable)
19 instruction <= {memory[address], memory[address + 1], memory[address + 2], memory[address + 3]};
20 end
21 endmodule
22
Verilog
Tab Width: 8
Ln 22, Col 1
INS
Empty...
top_module.v
Red Hat
Enterprise Linux
Search
07:23
13-11-2024