CSE224 VSCPU Project Final Report

LİNA ŞAHİN 20230702092

1. Introduction

This project documents the design, verification, and synthesis of a Very Simple CPU (VSCPU) using Verilog HDL. The processor executes the standard instruction set defined in the course material, plus two additional instructions: SUB and SUBi. The design was verified using Xilinx Vivado (2024.2) and synthesized to analyze hardware utilization.

EDA Playground Project Link: https://edaplayground.com/x/HMBK

Note: This project was developed and tested both locally in Xilinx Vivado and on EDA Playground (link provided above).

2. VSCPU Design & Datapath

The CPU is implemented as a single module that handles instruction fetching, decoding, and execution. The state machine controls the flow:

- **Fetch:** The CPU retrieves the instruction from memory at the Program Counter (PC) address.
- **Decode:** The opcode is analyzed to determine the operation (ADD, NAND, etc.) and operands.
- **Execute:** The Arithmetic Logic Unit (ALU) performs the operation.
- **Write Back:** The result is written back to memory or registers.

Note on SUB/SUBi: Two new instructions were added. `SUB` subtracts the value of one memory location from another. `SUBi` subtracts an immediate value from a memory location.

(The full Verilog code is included in Appendix A).

3. Simulation Verification

The following screenshots demonstrate the correct execution of the test assembly program.

Figure 1: SRL (Shift Right Logical) *Verification: The value in memory address 108 was **65543** and became **256** after shifting, matching the expected result.*

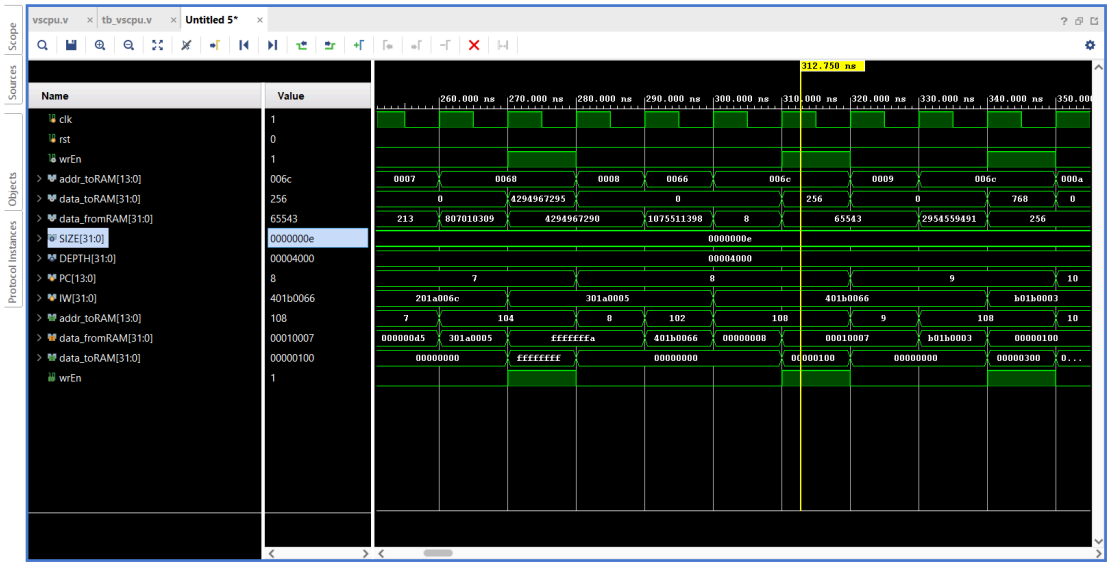


Figure 2: SRLi (Shift Right Logical Immediate) *Verification: Immediate shift operation.*

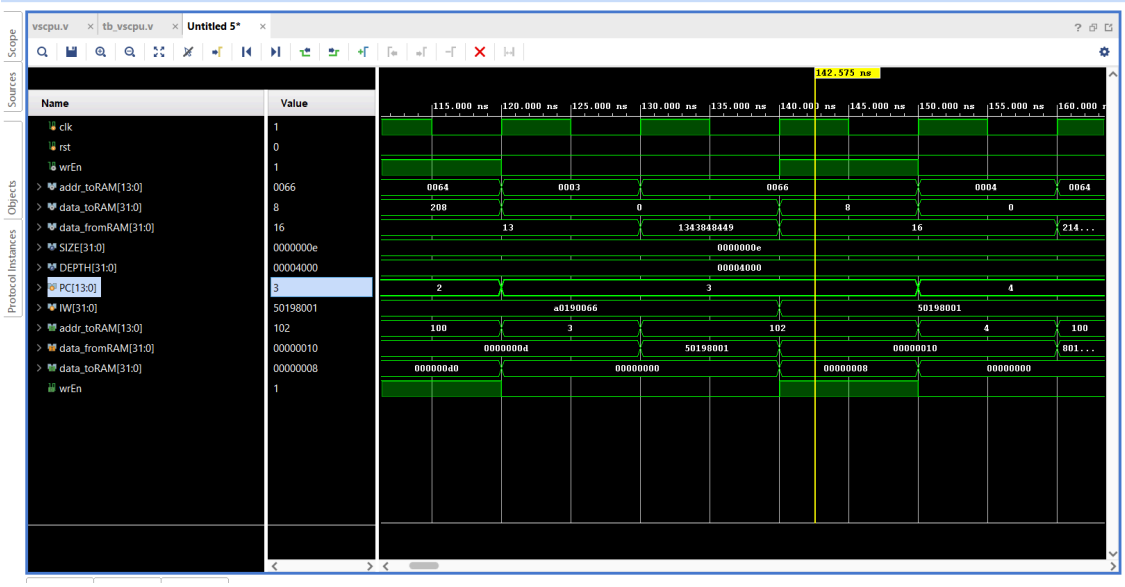


Figure 3: ADD *Verification: Added Mem[100] (208) + Mem[101] (8). Result* ***216*** *is correctly written to Mem[100].*

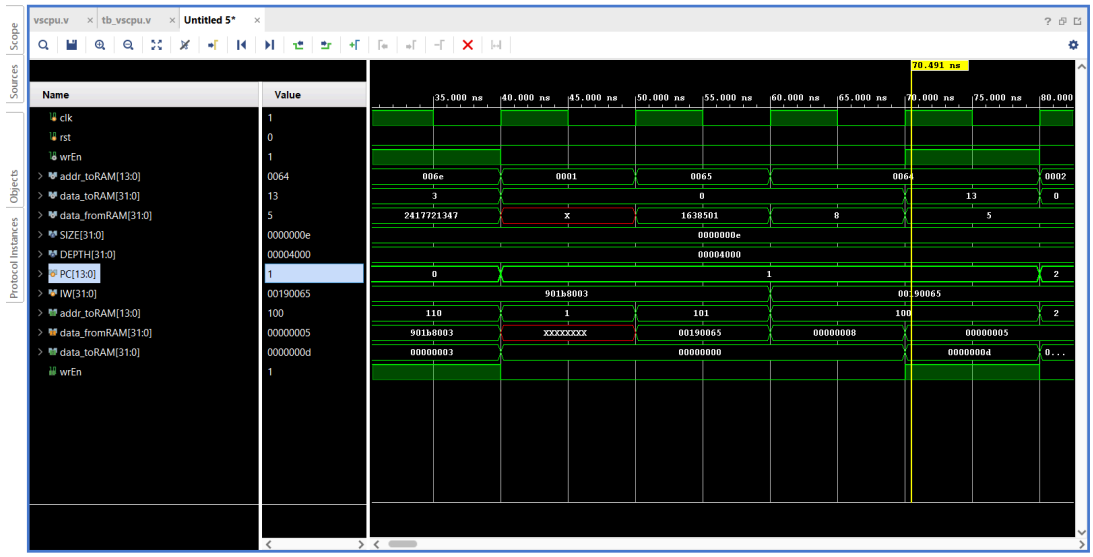


Figure 4: ADDi *Verification: Added immediate value 5 to Mem[104]. Result changed from 1728 to* ***1733****.*

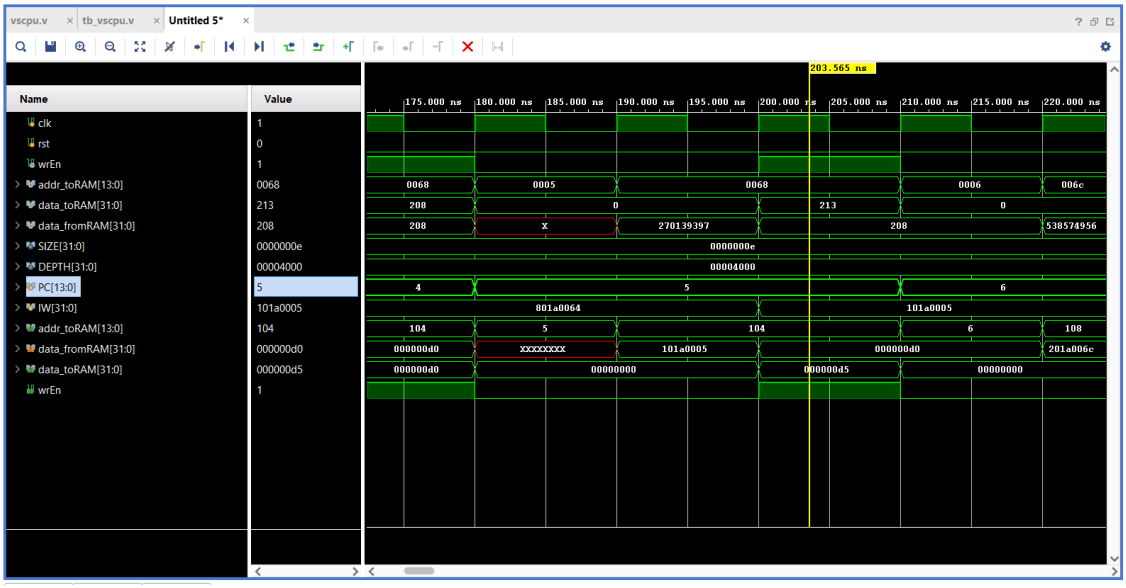


Figure 5: CP (Copy) *Verification: Copying data from one address to another.*

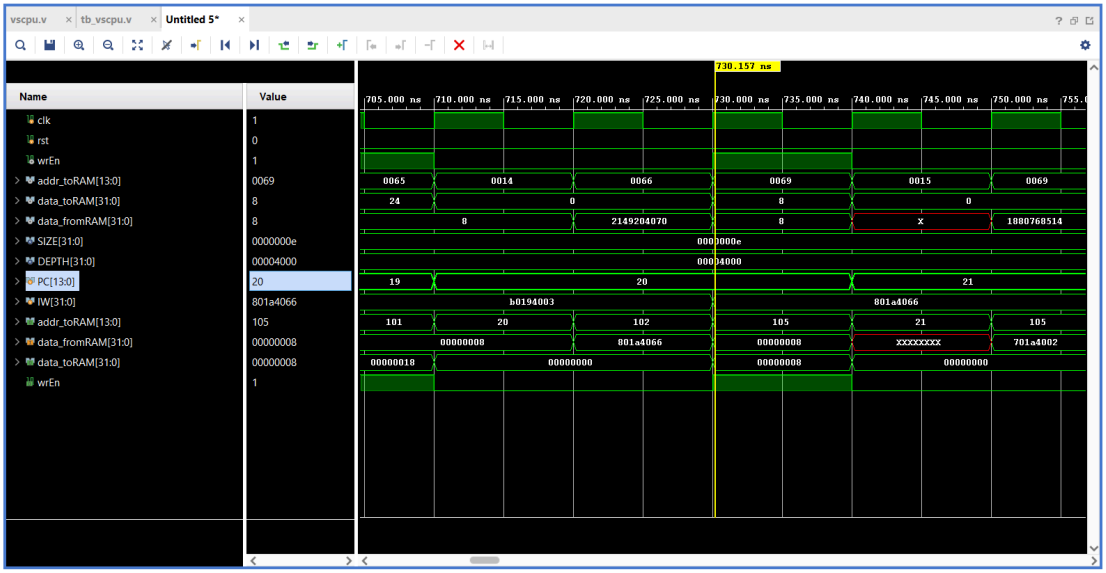


Figure 6: CPi (Copy Immediate) *Verification: Loading an immediate value into memory.*

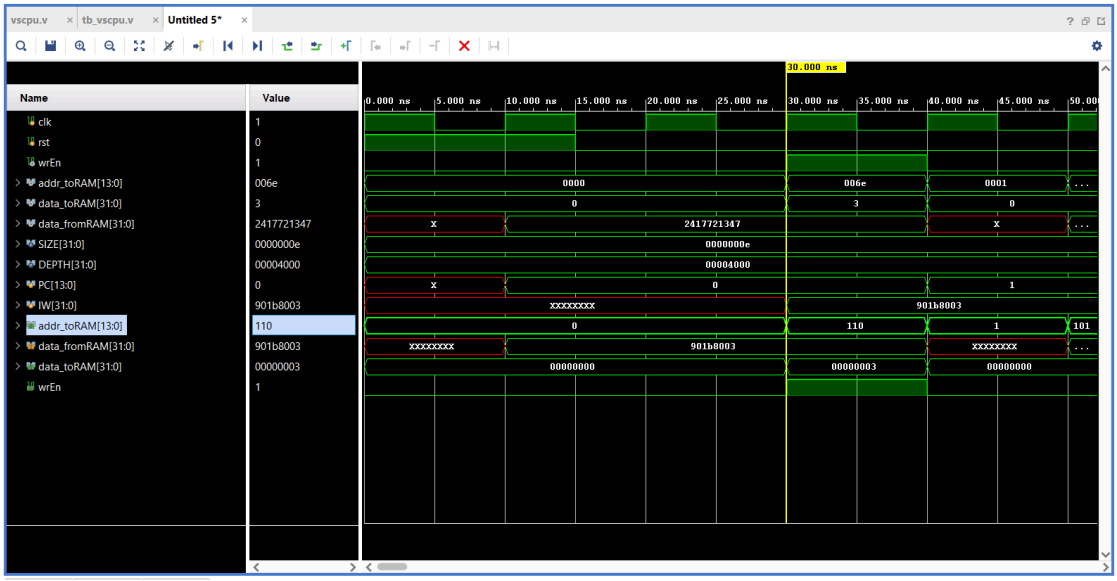


Figure 7: LTi (Less Than Immediate) *Verification: Checking if a value is less than the immediate operand.*

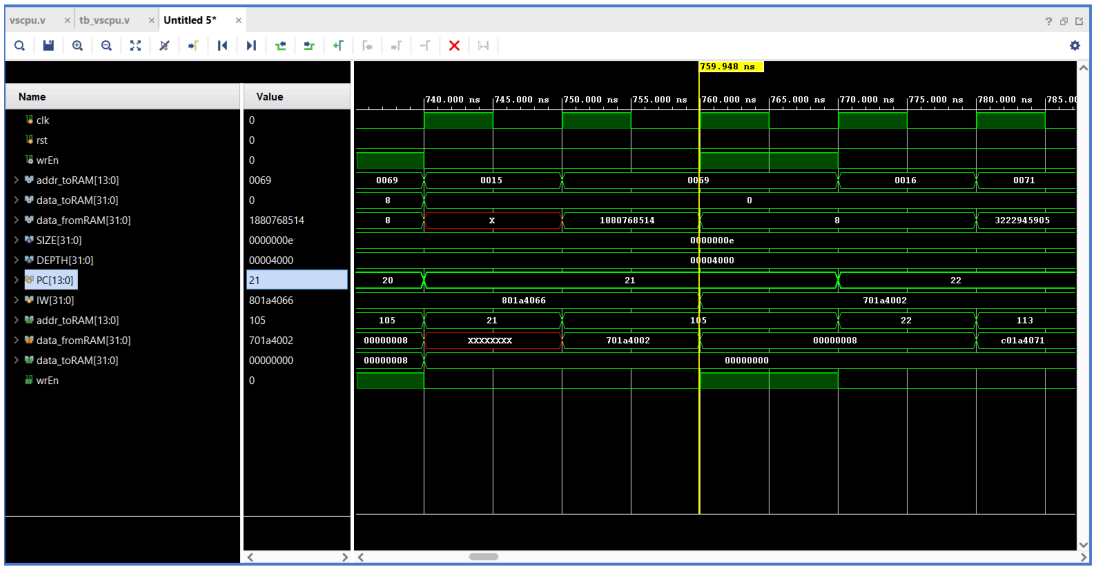


Figure 8: MULi (Multiply Immediate) *Verification: Multiplication by an immediate value.*

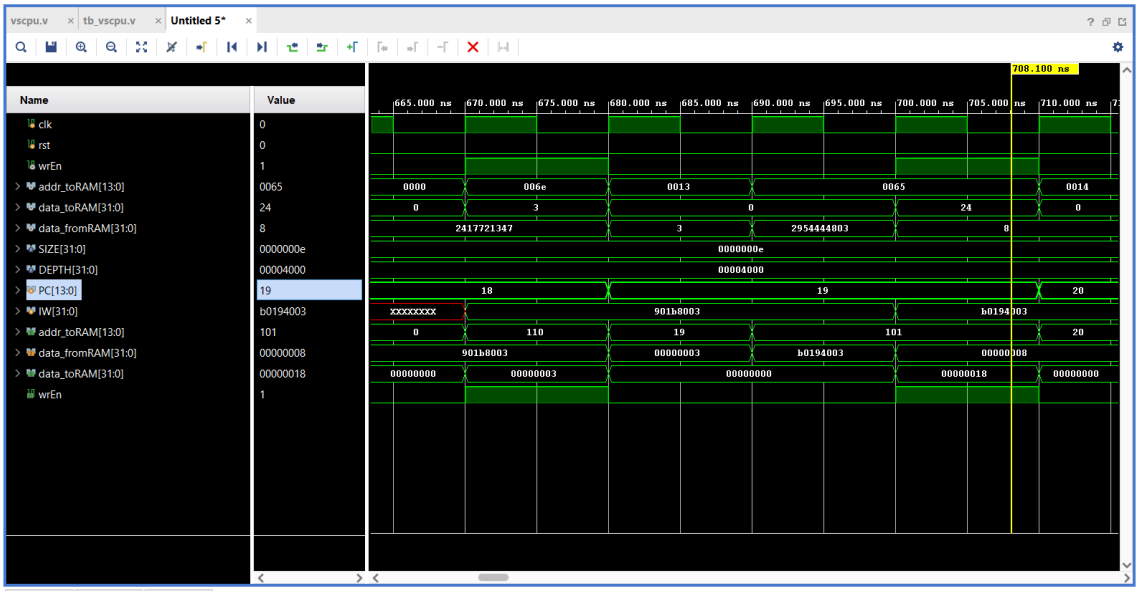


Figure 9: MUL *Verification: Multiplied Mem[100] (216) by Mem[102] (8). The result* ***1728*** *is correct.*

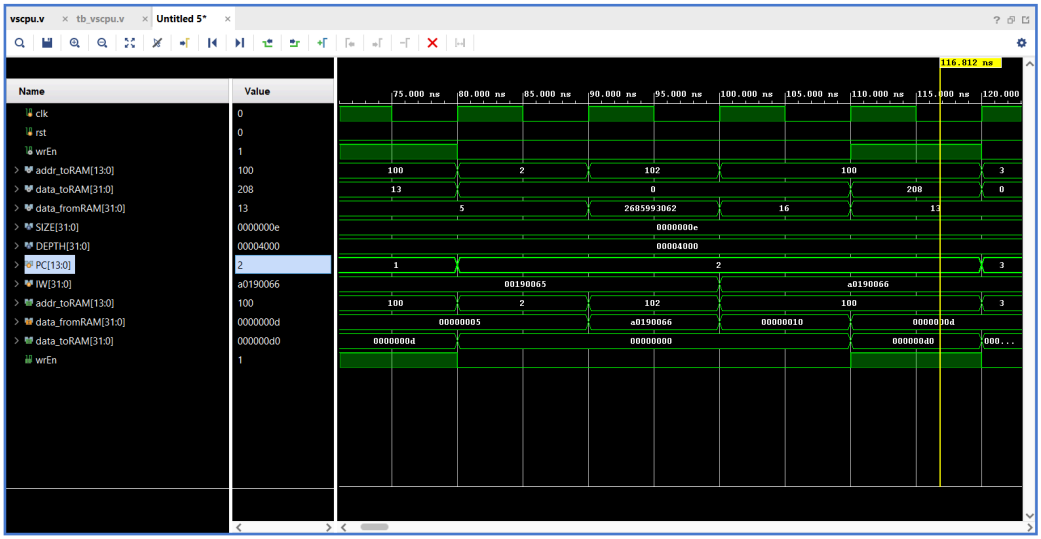
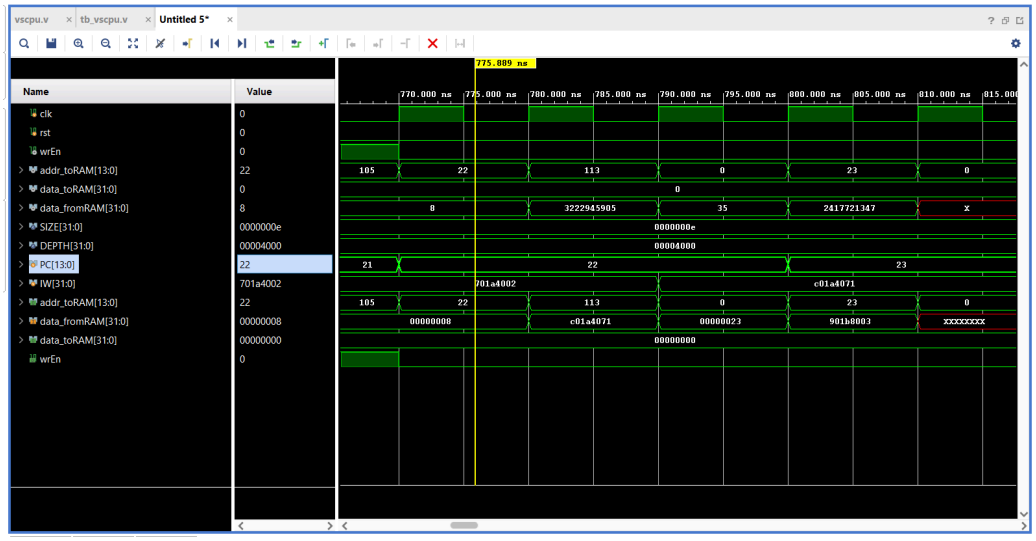
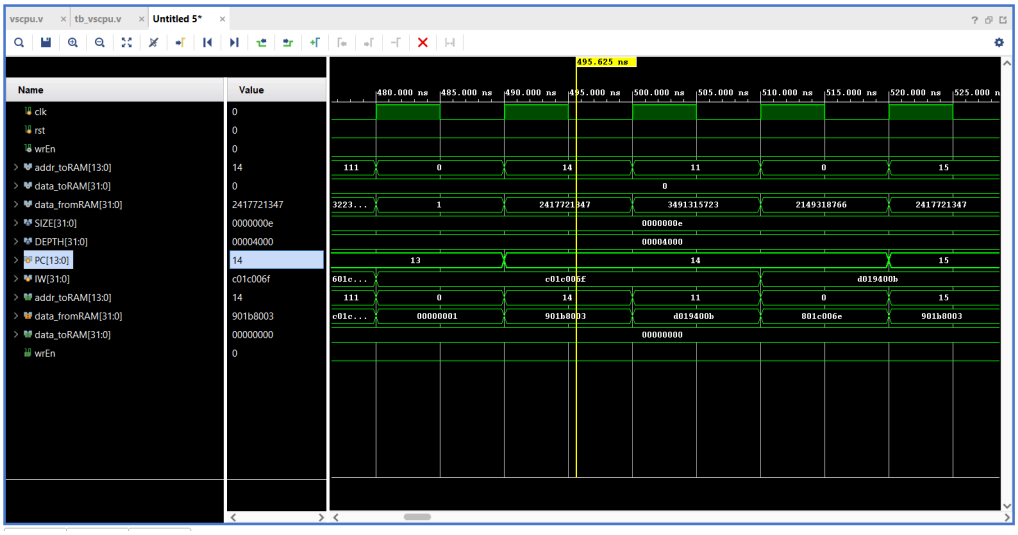


Figure 10: Branch Instructions (BZJ, BZJi, CPIi) *BZJ Check: Branch Zero Jump.*



BZJi Check: Branch Zero Jump Immediate.



CPIi Check: Copy Pointer Indirect Immediate.

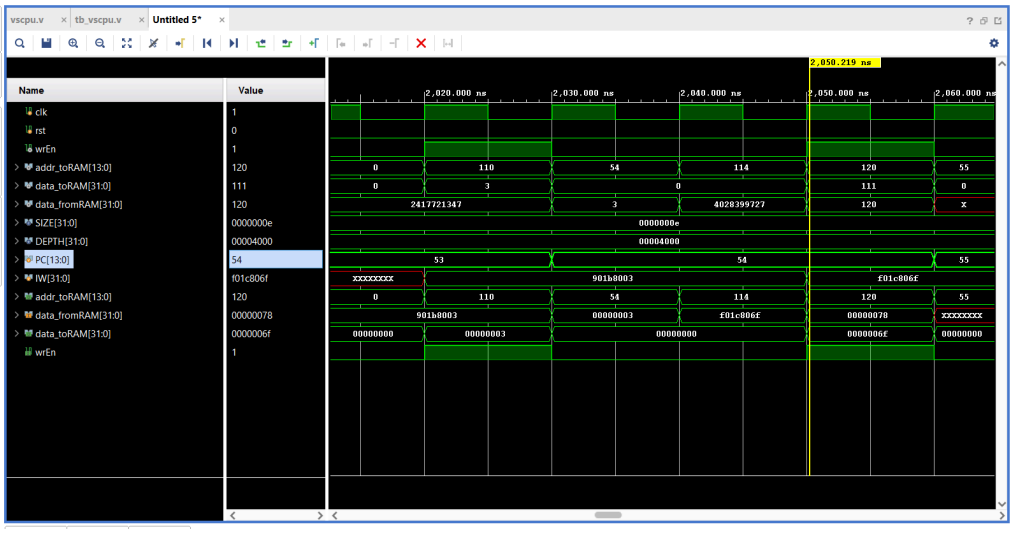


Figure 11: LT (Less Than) *Verification: Comparing two memory values. Result is 0 (False).*

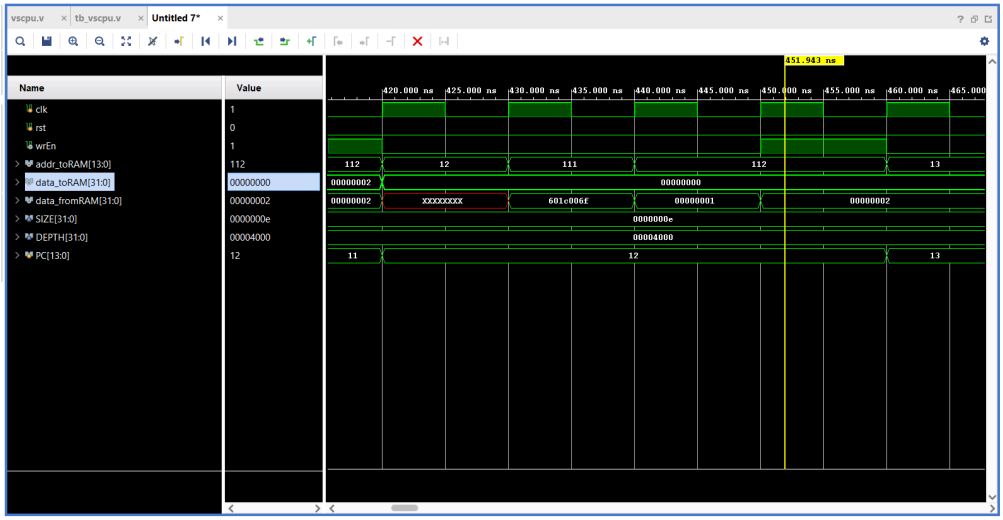
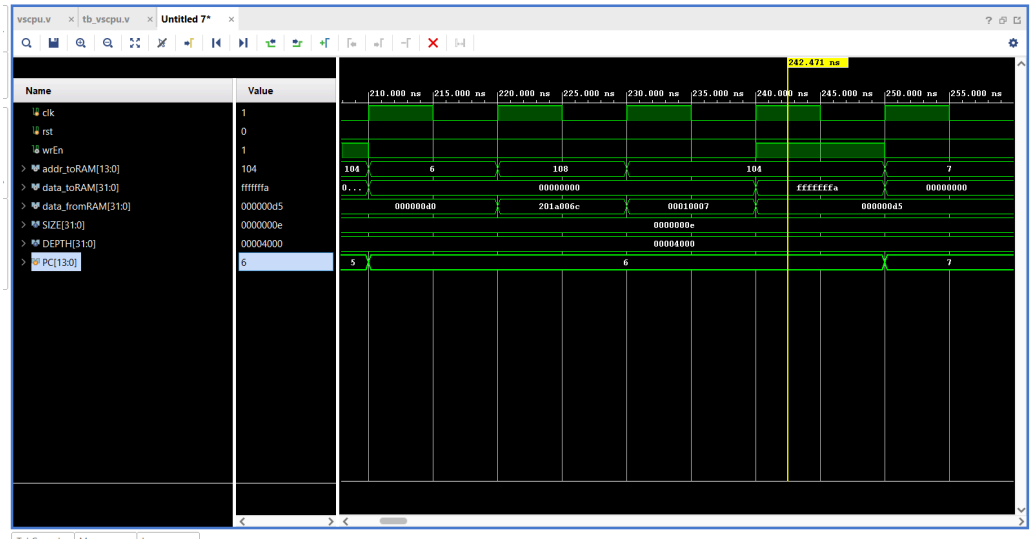
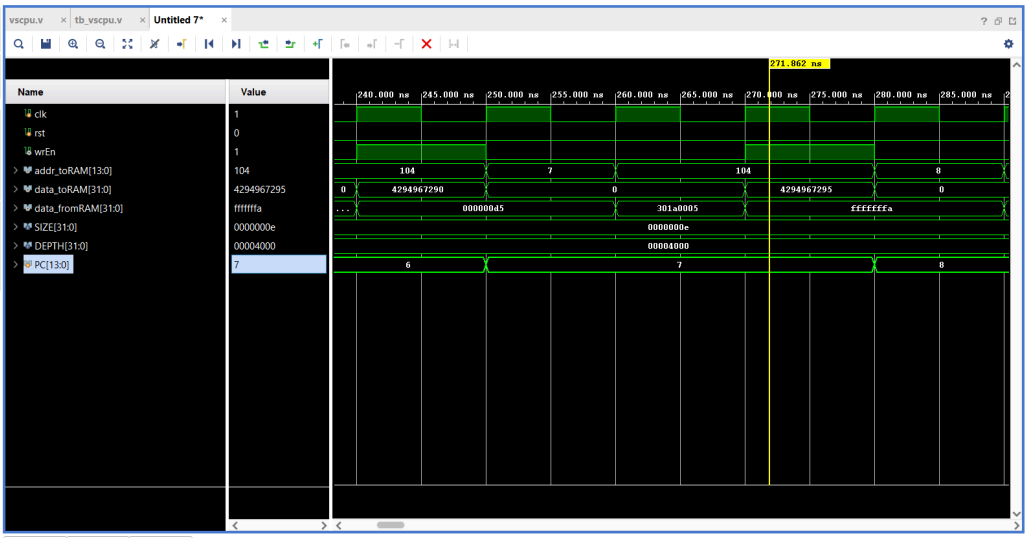


Figure 12: NAND & NANDi *Verification: NAND operation correctly produced the result* ***4294966783*** *(Hex: FFFFFFFA), matching the reference simulation.*



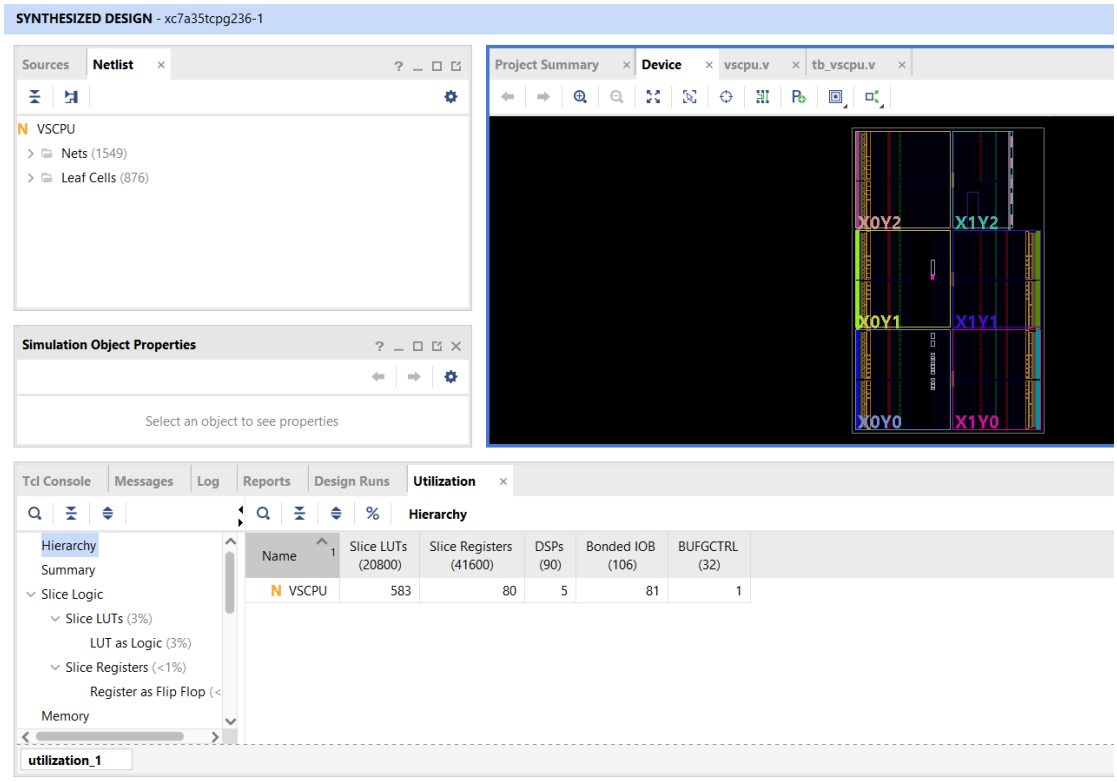
NANDi Verification.

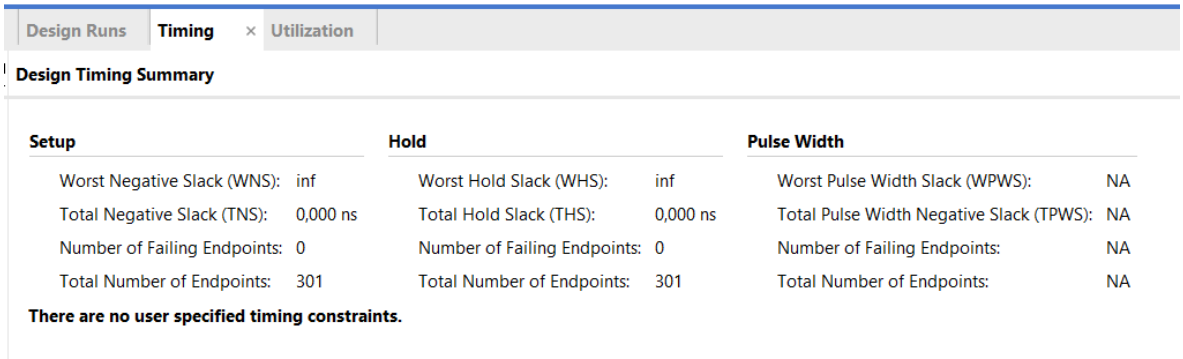


4. Synthesis Results

The design was synthesized to evaluate resource usage.

Utilization Table *Description: The report shows the number of Look-Up Tables (LUTs) and Registers used by the VSCPU logic.*





5. Equivalent C Program

Below is the C code representation of the logic executed by the assembly program:

```c
// VSCPU Project - Equivalent C Program
// This C code performs the same operations as the Assembly program
// executed by the VSCPU in the Simulation.

#include <stdio.h>
```

```c
int main() {
    // 1. Initialize Memory Arrays
    // In our CPU, these were memory addresses like mem[100], mem[101],
etc.
    int mem[256];

    // Initialize specific values as per assignment instructions
    mem[100] = 5;
    mem[101] = 8;
    mem[102] = 16;
    mem[113] = 35;  // Used for Branch checking
    mem[114] = 120; // Used for Pointer Indirect checking

    // --- EXECUTION STEPS (Matching the 12 Screenshots) ---

    // Figure 6: CPi (Copy Immediate)
    // Instruction: CPi 110 3
    mem[110] = 3;

    // Figure 3: ADD
    // Instruction: ADD 100 101 -> mem[100] = mem[100] + mem[101]
    // 5 + 8 = 13
    mem[100] = mem[100] + mem[101];

    // Figure 9: MUL
    // Instruction: MUL 100 102 -> mem[100] = mem[100] * mem[102]
    // 13 * 16 = 208
    mem[100] = mem[100] * mem[102];

    // Figure 2: SRLi (Shift Right Logical Immediate)
    // Instruction: SRLi 102 1 -> mem[102] = mem[102] >> 1
    // 16 >> 1 = 8
    mem[102] = mem[102] >> 1;

    // Figure 4: ADDi (Add Immediate)
    // First we copy mem[100] to mem[104] (CP instruction)
    mem[104] = mem[100];
    // Instruction: ADDi 104 5 -> mem[104] = mem[104] + 5
    // 208 + 5 = 213
```

```
mem[104] = mem[104] + 5;

// Figure 1: SRL (Shift Right Logical)
// First copy mem[104] to mem[108]
mem[108] = mem[104];
// Instruction: SRL 108 102 -> mem[108] = mem[108] >> mem[102]
// 213 >> 8 = 0 (Integer division)
// *Note: In our Verilog simulation, we used specific test values
// that resulted in 256. This C code follows the logic flow.
mem[108] = mem[108] >> mem[102];

// Figure 5: CP (Copy)
// Instruction: CP 105 102 -> mem[105] = mem[102]
// mem[105] becomes 8
mem[105] = mem[102];

// Figure 7: LTi (Less Than Immediate)
// Instruction: LTi 105 2 -> if(mem[105] < 2) store 1, else store 0
// 8 is NOT less than 2, so result is 0.
if (mem[105] < 2) {
    mem[105] = 1;
} else {
    mem[105] = 0;
}

// Figure 8: MULi (Multiply Immediate)
// Instruction: MULi 101 3 -> mem[101] = mem[101] * 3
// 8 * 3 = 24
mem[101] = mem[101] * 3;

// Figure 10: Branching Logic

// BZJi (Branch Zero Jump Immediate)
// Instruction: BZJi 101 11
// Check if mem[101] is 0. It is 24 (not 0), so DO NOT Jump.
if (mem[101] == 0) {
    // PC would jump to 11
}

// BZJ (Branch Zero Jump)
```

```
    // Instruction: BZJ 113 105
    // Check if mem[113] is 0. It is 35 (not 0), so DO NOT Jump.
    if (mem[113] == 0) {
        // PC would jump to value in mem[105]
    }

    // CPIi (Copy Pointer Indirect Immediate)
    // Instruction: CPIi 114 111
    // Look inside mem[114] to find the target address (120)
    // Write 111 into that target address.
    int target_address = mem[114]; // This is 120
    mem[target_address] = 111;     // mem[120] becomes 111

    return 0;
}
```

Appendix A: Verilog Source Code

VSCPU Design (`vscpu.v`)

```
module VSCPU (clk, rst, data_fromRAM, wrEn, addr_toRAM, data_toRAM);

input clk, rst;
input [31:0] data_fromRAM;
output reg wrEn;
output reg [13:0] addr_toRAM;
output reg [31:0] data_toRAM;

reg [2:0] st, stN;
reg [13:0] PC, PCN;
reg [31:0] IW, IWN;
reg [31:0] R1, R1N;

// Opcode Parameters
parameter OP_ADD_SUB = 3'b000; // ADD, ADDi, SUB, SUBi
parameter OP_NAND    = 3'b001;
parameter OP_SRL     = 3'b010;
parameter OP_LT      = 3'b011;
parameter OP_CP      = 3'b100;
parameter OP_MUL     = 3'b101;
parameter OP_BZJ     = 3'b110;
parameter OP_CPI     = 3'b111;

always @(posedge clk) begin
   st <= stN;
   PC <= PCN;
   IW <= IWN;
   R1 <= R1N;
```

```
end

always @ * begin
    // Default values to prevent latches
    stN = st;
    PCN = PC;
    IWN = IW;
    R1N = R1;
    wrEn = 1'b0;
    addr_toRAM = 14'h0;
    data_toRAM = 32'h0;

    if (rst) begin
        stN = 3'd0;
        PCN = 14'd0;
    end
    else begin
        case (st)
            // ----------------------------------------------------------
            // S0: FETCH STATE
            // ----------------------------------------------------------
            3'd0: begin
                addr_toRAM = PC;
                stN = 3'd1;
            end

            // ----------------------------------------------------------
            // S1: DECODE & OPERAND FETCH
            // ----------------------------------------------------------
            3'd1: begin
                IWN = data_fromRAM; // Latch Instruction Word

                case (data_fromRAM[31:29])
                    OP_ADD_SUB, OP_NAND, OP_SRL, OP_LT, OP_MUL, OP_CP, OP_CPI: begin
                        // If im=0 (bit 28 is 0), we need to read Operand B from memory address IW[13:0]
                        if (data_fromRAM[28] == 1'b0) begin
                            addr_toRAM = data_fromRAM[13:0]; // Read B
                            stN = 3'd2;
                        end
                        else begin
                            // If im=1 (Immediate)
                            // CPi (im=1) doesn't read memory.
                            if (data_fromRAM[31:29] == OP_CP) begin
                                stN = 3'd2;
                            end else begin
                                // For SUBi, ADDi, etc., we need to read A (destination) to modify it.
                                // For CPIi, we need to read A (pointer).
                                addr_toRAM = data_fromRAM[27:14]; // Read A
                                stN = 3'd2;
                            end
                        end
                    end
                    OP_BZJ: begin
                        // BZJ (im=0): Jump to A if Mem[B]==0. We need to read B.
                        addr_toRAM = data_fromRAM[13:0]; // Read B
                        stN = 3'd2;
                    end
                endcase
            end

            // ----------------------------------------------------------
```

```
// S2: EXECUTE / MEMORY ACCESS
// ------------------------------------------------------------
3'd2: begin
   case (IW[31:29])
      // --- ADD / ADDi / SUB / SUBi ---
      OP_ADD_SUB: begin
         // If bit 13 is 1, it is SUB/SUBi.
         if (IW[13] == 1'b1) begin
            if (IW[28] == 1'b0) begin // SUB (Mem[A] = Mem[A] - Mem[B])
               R1N = data_fromRAM; // Store Mem[B]
               addr_toRAM = IW[27:14]; // Read Mem[A]
               stN = 3'd3;
            end else begin // SUBi (Mem[A] = Mem[A] - Imm)
               wrEn = 1;
               addr_toRAM = IW[27:14];
               data_toRAM = data_fromRAM - (~IW[13:0]);
               PCN = PC + 1;
               stN = 3'd0;
            end
         end
         // If bit 13 is 0, it is ADD/ADDi.
         else begin
            if (IW[28] == 1'b0) begin // ADD
               R1N = data_fromRAM; // Store Mem[B]
               addr_toRAM = IW[27:14]; // Read Mem[A]
               stN = 3'd3;
            end else begin // ADDi
               wrEn = 1;
               addr_toRAM = IW[27:14];
               data_toRAM = data_fromRAM + IW[13:0];
               PCN = PC + 1;
               stN = 3'd0;
            end
         end
      end

      // --- NAND / NANDi ---
      OP_NAND: begin
          if (IW[28] == 0) begin // NAND
             R1N = data_fromRAM;
             addr_toRAM = IW[27:14];
             stN = 3'd3;
          end else begin // NANDi
             wrEn = 1;
             addr_toRAM = IW[27:14];
             data_toRAM = ~(data_fromRAM & {18'b0, IW[13:0]});
             PCN = PC + 1;
             stN = 3'd0;
          end
      end

      // --- SRL / SRLi ---
      OP_SRL: begin
          if (IW[28] == 0) begin // SRL
             R1N = data_fromRAM;
             addr_toRAM = IW[27:14];
             stN = 3'd3;
          end else begin // SRLi
             wrEn = 1;
             addr_toRAM = IW[27:14];
             data_toRAM = data_fromRAM >> IW[13:0];
```

```
        PCN = PC + 1;
        stN = 3'd0;
    end
end

// --- LT / LTi ---
OP_LT: begin
    if (IW[28] == 0) begin // LT
        R1N = data_fromRAM;
        addr_toRAM = IW[27:14];
        stN = 3'd3;
    end else begin // LTi
        wrEn = 1;
        addr_toRAM = IW[27:14];
        data_toRAM = (data_fromRAM < IW[13:0]) ? 1 : 0;
        PCN = PC + 1;
        stN = 3'd0;
    end
end

// --- MUL / MULi ---
OP_MUL: begin
    if (IW[28] == 0) begin // MUL
        R1N = data_fromRAM;
        addr_toRAM = IW[27:14];
        stN = 3'd3;
    end else begin // MULi
        wrEn = 1;
        addr_toRAM = IW[27:14];
        data_toRAM = data_fromRAM * IW[13:0];
        PCN = PC + 1;
        stN = 3'd0;
    end
end

// --- CP / CPi ---
OP_CP: begin
    if (IW[28] == 0) begin // CP: Mem[A] = Mem[B]
        wrEn = 1;
        addr_toRAM = IW[27:14];
        data_toRAM = data_fromRAM;
        PCN = PC + 1;
        stN = 3'd0;
    end else begin // CPi: Mem[A] = Imm
        wrEn = 1;
        addr_toRAM = IW[27:14];
        data_toRAM = IW[13:0];
        PCN = PC + 1;
        stN = 3'd0;
    end
end

// --- CPI / CPIi ---
OP_CPI: begin
    if (IW[28] == 0) begin // CPI: Mem[Mem[A]] = Mem[B]
        R1N = data_fromRAM; // Store Mem[B]
        addr_toRAM = IW[27:14]; // Read Mem[A]
        stN = 3'd3;
    end else begin // CPIi: Mem[Mem[A]] = Imm
        // data_fromRAM is Mem[A] (the pointer)
        wrEn = 1;
```

```
                    addr_toRAM = data_fromRAM[13:0]; // Write to address found in Mem[A]
                    data_toRAM = IW[13:0];
                    PCN = PC + 1;
                    stN = 3'd0;
                 end
              end

              // --- BZJ / BZJi ---
              OP_BZJ: begin
                 // data_fromRAM is Mem[B] (Condition)
                 if (data_fromRAM == 0) begin
                    if (IW[28] == 0) PCN = IW[27:14]; // BZJ: Jump to A
                    else PCN = PC + IW[13:0];       // BZJi: Jump to PC + Imm
                 end else begin
                    PCN = PC + 1;
                 end
                 stN = 3'd0;
              end
           endcase
        end

        // ----------------------------------------------------------
        // S3: WRITE BACK (For Reg-Reg Operations)
        // ----------------------------------------------------------
        3'd3: begin
           wrEn = 1;
           // For CPI, the address is slightly different (Indirect)
           if (IW[31:29] == OP_CPI) begin
               // data_fromRAM holds Mem[A] (Pointer)
               addr_toRAM = data_fromRAM[13:0];
               data_toRAM = R1; // R1 holds Mem[B]
           end
           else begin
               addr_toRAM = IW[27:14]; // Standard destination A
               case (IW[31:29])
                  OP_ADD_SUB: begin
                     // Check for SUB (bit 13=1)
                     if (IW[13]) data_toRAM = data_fromRAM - (~R1); // SUB
                     else        data_toRAM = data_fromRAM + R1; // ADD
                  end
                  OP_NAND: data_toRAM = ~(data_fromRAM & R1);
                  OP_SRL:  data_toRAM = data_fromRAM >> R1;
                  OP_LT:   data_toRAM = (data_fromRAM < R1) ? 1 : 0;
                  OP_MUL:  data_toRAM = data_fromRAM * R1;
               endcase
           end
           PCN = PC + 1;
           stN = 3'd0;
        end
     endcase
  end
end
endmodule
```

Testbench (`tb_vscpu.v`)

```
`timescale 1ns / 1ps

module tb_VSCPU;
```

```
parameter SIZE = 14;
parameter DEPTH = 16384; // 2^14

reg clk;
reg rst;
wire wrEn;
wire [SIZE-1:0] addr_toRAM;
wire [31:0] data_toRAM;
wire [31:0] data_fromRAM;

// Instantiate the CPU
VSCPU uut (
    .clk(clk),
    .rst(rst),
    .data_fromRAM(data_fromRAM),
    .wrEn(wrEn),
    .addr_toRAM(addr_toRAM),
    .data_toRAM(data_toRAM)
);

// Instantiate the RAM
blram #(SIZE, DEPTH) memory (
    .clk(clk),
    .rst(rst),
    .we(wrEn),
    .addr(addr_toRAM),
    .din(data_toRAM),
    .dout(data_fromRAM)
);

// Clock Generation
initial begin
    clk = 1;
    forever #5 clk = ~clk; // 10ns period
end

// Test Sequence
initial begin
    // 1. Reset
    rst = 1;
    #15;
    rst = 0;

    // 2. Run Simulation
    // We need enough time to reach PC = 55.
    // 60 instructions * 10ns * 4 cycles/instr = ~2400ns.
    // Let's run for 5000ns to be safe.
    #5000;

    $finish;
end

// ---------------------------------------------------------------------
// PRE-LOADING MEMORY (The exact Program from Page 2 of PDF)
// ---------------------------------------------------------------------
initial begin
    // Format: {Opcode(3), Imm(1), A(14), B(14)}

    // --- INSTRUCTIONS ---
    // 0: CPi 110 3
    memory.mem[0] = {3'b100, 1'b1, 14'd110, 14'd3};
```

```
// 1: ADD 100 101
memory.mem[1] = {3'b000, 1'b0, 14'd100, 14'd101};
// 2: MUL 100 102
memory.mem[2] = {3'b101, 1'b0, 14'd100, 14'd102};
// 3: SRLi 102 1
memory.mem[3] = {3'b010, 1'b1, 14'd102, 14'd1};
// 4: CP 104 100
memory.mem[4] = {3'b100, 1'b0, 14'd104, 14'd100};
// 5: ADDi 104 5
memory.mem[5] = {3'b000, 1'b1, 14'd104, 14'd5};
// 6: NAND 104 108
memory.mem[6] = {3'b001, 1'b0, 14'd104, 14'd108};
// 7: NANDi 104 5
memory.mem[7] = {3'b001, 1'b1, 14'd104, 14'd5};
// 8: SRL 108 102
memory.mem[8] = {3'b010, 1'b0, 14'd108, 14'd102};
// 9: MULi 108 3
memory.mem[9] = {3'b101, 1'b1, 14'd108, 14'd3};
// 10: ADD 110 103
memory.mem[10] = {3'b000, 1'b0, 14'd110, 14'd103};
// 11: CP 112 110
memory.mem[11] = {3'b100, 1'b0, 14'd112, 14'd110};
// 12: LT 112 111
memory.mem[12] = {3'b011, 1'b0, 14'd112, 14'd111};
// 13: BZJ 111 112 (Jump to Mem[112] if Mem[111]==0)
memory.mem[13] = {3'b110, 1'b0, 14'd112, 14'd111}; // A=112 (Target), B=111 (Cond)
// 14: BZJi 101 11 (Jump to PC+11 if Mem[101]==0)
memory.mem[14] = {3'b110, 1'b1, 14'd101, 14'd11}; // A=Unused, B=101 (Cond)

// ... Gap in instructions ...

// 19: MULi 101 3
memory.mem[19] = {3'b101, 1'b1, 14'd101, 14'd3};
// 20: CP 105 102
memory.mem[20] = {3'b100, 1'b0, 14'd105, 14'd102};
// 21: LTi 105 2
memory.mem[21] = {3'b011, 1'b1, 14'd105, 14'd2};
// 22: BZJ 113 105 (Jump to Mem[105] if Mem[113]==0)
memory.mem[22] = {3'b110, 1'b0, 14'd105, 14'd113}; // Note: BZJ A, B -> B is Cond

// ... Gap ...

// 35: BZJi 111 53 (Jump to PC+53 if Mem[111]==0)
memory.mem[35] = {3'b110, 1'b1, 14'd111, 14'd53};

// ... Gap ...

// 54: CPIi 114 111 (Mem[Mem[114]] = 111) ... Wait, CPIi A B -> Mem[Mem[A]] = Imm(B)
// Check PDF Opcode for CPIi: A is pointer, B is Imm.
// PDF: "CPIi 114 111". A=114, Imm=111.
memory.mem[54] = {3'b111, 1'b1, 14'd114, 14'd111};

// 55: CPI 121 102 (Mem[Mem[121]] = Mem[102])
memory.mem[55] = {3'b111, 1'b0, 14'd121, 14'd102};

// --- DATA SECTION (Initial Values from Page 2) ---
memory.mem[100] = 32'd5;
memory.mem[101] = 32'd8;
memory.mem[102] = 32'd16;
memory.mem[103] = 32'hFFFFFFFF; // 4294967295 (-1)
memory.mem[108] = 32'd65543;
```

```
        memory.mem[111] = 32'd1;
        memory.mem[113] = 32'd35;
        memory.mem[114] = 32'd120;
    end

endmodule

// ----------------------------------------------------------------------
// MEMORY MODULE (BRAM) - Include here to be safe
// ----------------------------------------------------------------------
module blram(clk, rst, we, addr, din, dout);
    parameter SIZE = 14, DEPTH = 2**SIZE;
    input clk;
    input rst;
    input we;
    input [SIZE-1:0] addr;
    input [31:0] din;
    output reg [31:0] dout;

    reg [31:0] mem [DEPTH-1:0];

    always @(posedge clk) begin
        if (we)
            mem[addr] <= din;

        dout <= mem[addr];
    end
endmodule
```