DESIGN AND SIMULATION OF AN 8-BIT RISC PROCESSOR USING VERILOG HDL

By

V JAISHREE 24BEC1407

JANANI S K 24BEC1540

V G MANASA 24BEC1419

SHRUTHI NARAYANAN 24BEC1458

Under the guidance of

Dr. Sakthivel S M

SCHOOL OF SENSE

VELLORE INSTITUTE OF TECHNOLOGY, CHENNAI-600127

TABLE OF CONTENTS

INTRODUCTION

The design and implementation of a Reduced Instruction Set Computer (RISC) processor form one of the fundamental applications of digital system design and computer architecture. This project focuses on developing an 8-bit single-cycle RISC processor using Verilog HDL, running simulations in ModelSim and implementing it on the Altera DE10-Lite FPGA development board. The processor is capable of executing a limited but efficient instruction set including arithmetic, logical, and control operations. The main goal is to understand the basic structure of a CPU, including its datapath, control logic, memory interfacing, and I/O display. The design also integrates peripherals such as LEDs and 7-segment displays, counter outputs on the LEDs, Keys and switches on the board.

THEORY

RISC (Reduced Instruction Set Computer) architecture is a design philosophy that simplifies the instruction set of the processor to enable faster execution and efficient pipelining. In contrast to CISC (Complex Instruction Set Computer), RISC processors use simple, fixed-length instructions that can be executed in a single clock cycle. This simplicity allows for reduced hardware complexity, lower power consumption, and predictable timing behavior.

In the 8-bit RISC processor designed here, instructions are 8 bits wide, divided into a 4-bit opcode and a 4-bit operand field. The processor executes one instruction per clock cycle, achieving single-cycle operation. Basic operations such as load, store, add, subtract, and logical operations are supported, along with control instructions like jump and halt.

METHODOLOGY

The methodology describes the step-by-step process followed for designing, implementing, simulating, and testing an 8-bit RISC (Reduced Instruction Set Computer) processor. The main objective was to design a simple yet functional processor capable of performing basic arithmetic and logical operations, using Verilog HDL and verified through simulation and hardware implementation on FPGA. The overall design process was divided into several systematic stages to ensure clarity, correctness, and efficient development. Each stage built upon the previous one to move from conceptual understanding to physical realization of the processor. The stages are as follows:

1. Requirement Analysis

The first step in the methodology was to clearly define the requirements and specifications of the processor. The goal was to design an 8-bit RISC processor that could handle basic arithmetic and logical operations using minimal hardware modules. The processor was required to handle 8-bit wide data and support a small set of instructions such as addition, subtraction, logical AND, and logical OR.To meet these requirements, the design needed to include several key hardware modules:

- Arithmetic Logic Unit (ALU) – for performing arithmetic and logical operations.
- Register File – for temporary storage of data and operands.
- Program Counter (PC) – to hold the address of the next instruction to be executed.
- Instruction Memory – to store the instruction set.
- Control Unit – to decode instructions and generate control signals.
- Data Memory – to store intermediate and final results.

The processor was planned to be implemented using Verilog HDL and simulated using appropriate EDA tools like ModelSim to validate its performance before moving to hardware.

2. Instruction Set Design

The instruction set forms the backbone of the processor's functionality. For this 8-bit RISC processor, the instruction set was kept simple to make implementation easier and efficient. A minimal instruction set architecture (ISA) was designed that covered all basic operations required for testing and demonstration.

Each instruction was assigned a unique opcode, which the control unit would interpret to perform specific actions. The instruction set included the following operations:

- Addition (ADD) – for performing arithmetic addition of registers.
- Subtraction (SUB) – for arithmetic subtraction.
- Logical AND and OR – for bitwise logical operations
- Data Transfer Instructions – to move data between registers or between memory and registers.

This simplicity ensured that the control unit and decoder design remained manageable while still demonstrating the essential functions of a working RISC processor.

3. Architecture Design

Once the instruction set was finalized, the next step was to design the architecture of the processor. The architecture consists of several interconnected hardware modules, each responsible for a specific task. The main components of the 8-bit RISC processor architecture include:

- Program Counter (PC): Stores the address of the next instruction to be executed and increments after each instruction cycle.
- Instruction Register (IR): Holds the current instruction fetched from memory and provides it to the decoder.
- Arithmetic Logic Unit (ALU): Performs arithmetic (ADD, SUB) and logical (AND, OR) operations as specified by the instruction.
- General Purpose Registers: Temporarily store operands and intermediate results during program execution.

- Control Unit: Decodes the opcode from the instruction register and generates necessary control signals to coordinate data flow among the components.
- Memory Interface: Manages the transfer of data and instructions between the processor and the memory.

The entire architecture was designed to operate synchronously using a clock signal, ensuring that all components work in coordination.

4. Verilog Implementation

After completing the architectural design, the processor was implemented using Verilog Hardware Description Language (HDL). Each module of the processor—such as the ALU, Program Counter, Registers, Control Unit, and Memory Interface—was coded as a separate Verilog file to maintain modularity and clarity.

Once individual modules were created, they were interconnected to form the top-level processor design. The testbenches were developed to verify each module's functionality and to ensure that the integrated system operated correctly. Through simulation, different instruction sequences were applied to check if the processor executed them accurately. This modular implementation approach made it easier to identify and fix issues during testing and allowed reuse of components in future designs.

5. Simulation and Functional Verification

Functional verification was a crucial step to ensure that all modules performed as intended. Simulation was carried out using ModelSim software. The verification process included:

- Simulating all Verilog modules to validate their correctness.
- Creating comprehensive testbenches to cover all possible instructions, data cases, and boundary conditions.
- Performing waveform analysis to verify correct timing, data flow, and control signals.
- Ensuring that arithmetic, logical, and branch instructions produced the expected outputs.

By examining simulation waveforms, the timing between control signals, data buses, and outputs was confirmed to be consistent with the design expectations. This phase helped detect logical or timing errors early before proceeding to hardware implementation.

6. Debugging and Optimization

After simulation, any mismatches or unexpected outputs were carefully analyzed and corrected. The debugging process involved identifying errors in logic, timing, or signal flow using waveform analysis and signal tracing tools within ModelSim.

Optimization was then carried out to improve performance and efficiency:

- Redundant logic in RTL code was removed.
- Critical paths were minimized to enhance the overall operating speed.
- Code modularity was refined to allow easier upgrades and future scalability.

These improvements ensured that the final processor design was compact, efficient, and reliable.

7. Hardware Implementation

Once the simulation results were satisfactory, the verified RTL design was synthesized for hardware implementation using FPGA DE 10 Lite Altera Board.

The key steps in this phase included:

- Synthesizing the Verilog design into FPGA-compatible logic.
- Assigning FPGA pin configurations according to board specifications.
- Generating the bitstream file and programming it into the FPGA.
- Integrating clock, reset, and I/O interfaces for smooth processor operation.

The processor was now functional on the FPGA board, capable of executing instructions and interacting with input/output components.

8. Final Testing

The last stage involved testing the processor on actual hardware to confirm correct real-world performance. Predefined instruction sequences were executed on the FPGA board to verify proper operation. The output was monitored through LEDs and seven-segment displays

Tests were conducted under different clock frequencies and load conditions to validate processor stability and consistency. Performance parameters such as speed, accuracy, and reliability were

evaluated. The processor successfully executed basic arithmetic and logical instructions, demonstrating its ability to handle real-time operations.

Final testing also ensured that the design remained stable during continuous operation, confirming the robustness and correctness of the developed 8-bit RISC processor.

BLOCK DIAGRAM

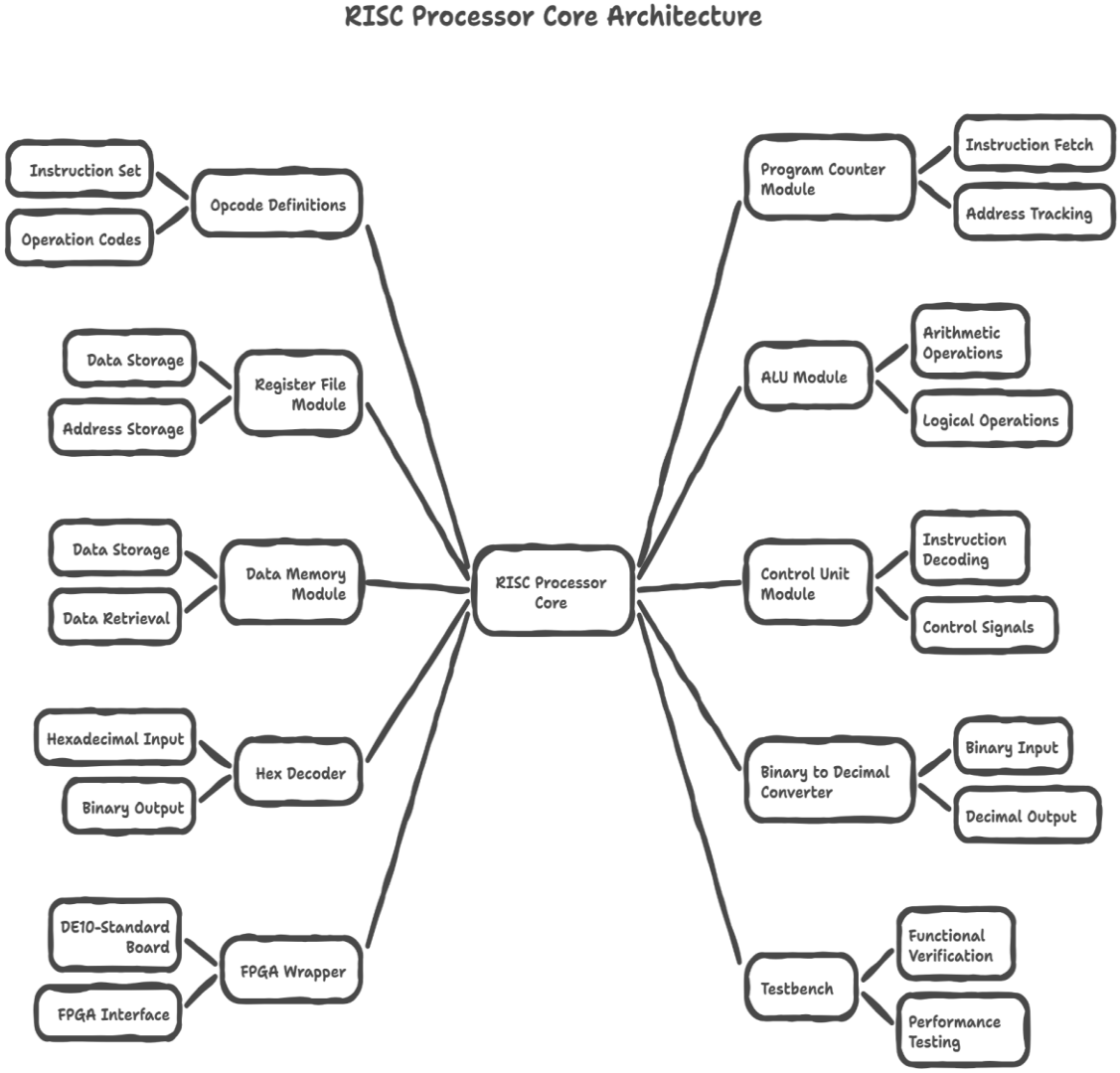
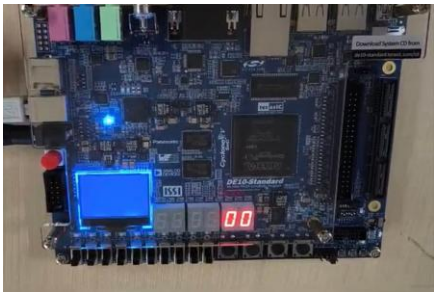
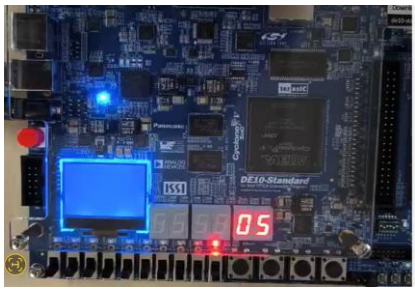
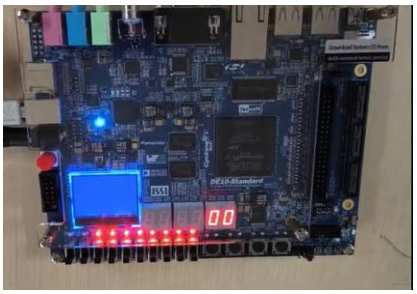


PHOTO GALLERY

PROGRAM COUNTER:







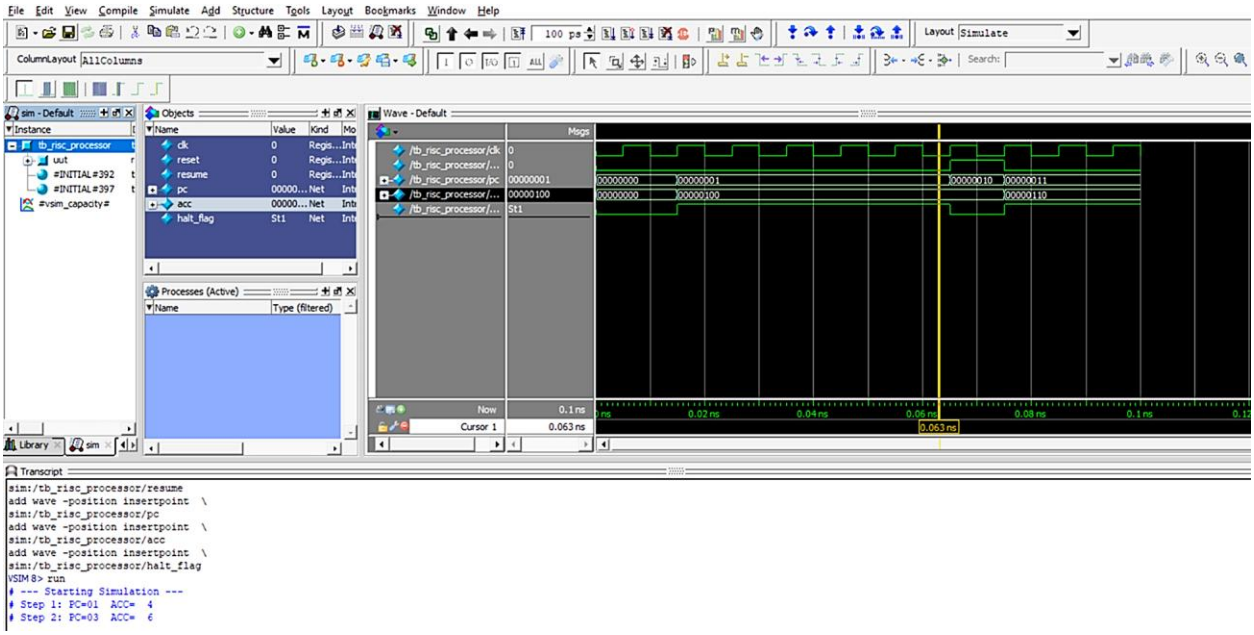
VERILOG CODE:

```
// ========================================
// 8-BIT RISC PROCESSOR - SINGLE-CYCLE DESIGN
// Target: Intel Cyclone V (5CSXFC6D6F31C6N)
// Top module name: project
// ========================================

// Opcode Definitions (shared across modules)
`define LDI   4'b0000 // Load Immediate (R0 <- Imm)
`define STORE 4'b0001 // Store R0 to Memory (Mem[Imm] <- R0)
`define ADD   4'b0010 // R0 <- R0 + R[Operand]
`define SUB   4'b0011 // R0 <- R0 - R[Operand]
`define AND   4'b0100
`define OR    4'b0101
`define XOR   4'b0110
`define JMP   4'b0111
`define JZ    4'b1000
`define HALT  4'b1111

// ========================================
// 1. PROGRAM COUNTER MODULE
// ========================================
module program_counter(
    input clk,
    input reset,
    input load_pc,
    input pc_enable,
    input [7:0] next_pc_address,
    output reg [7:0] pc_out
);
    always @(posedge clk or posedge reset) begin
        if (reset)
            pc_out <= 8'h00;
        else if (load_pc)
            pc_out <= next_pc_address;
        else if (pc_enable)
            pc_out <= pc_out + 1;
    end
endmodule


// ========================================
// 2. INSTRUCTION MEMORY MODULE
// ========================================
module instruction_memory(
    input  wire [7:0] address,
    output reg  [7:0] instruction
);
    reg [7:0] mem [0:255];
    integer i;
    initial begin
        // Program: Display 4, then 6, then 4+6=10
        // Step 1: Load 4 into R0, HALT to display
        mem[8'h00] = {`LDI, 4'h4};    // R0 = 4
        mem[8'h01] = {`HALT, 4'h0};   // Halt to display 4

        // Step 2: Load 6 into R0, HALT to display
        mem[8'h02] = {`LDI, 4'h6};    // R0 = 6
        mem[8'h03] = {`HALT, 4'h0};   // Halt to display 6

        // Step 3: Load 4, ADD 6 from R1, HALT to display result
        mem[8'h04] = {`LDI, 4'h4};    // R0 = 4
        mem[8'h05] = {`ADD, 4'h1};    // R0 = R0 + R1 (R1=6, so R0=10)
        mem[8'h06] = {`HALT, 4'h0};   // Halt to display 10

        for (i = 7; i < 256; i = i + 1)
            mem[i] = 8'h00;
    end

    always @(*) begin
        instruction = mem[address];
    end
endmodule

// ========================================
// 3. REGISTER FILE MODULE
// ========================================
module register_file(
    input wire clk,
    input wire reset,
    input wire reg_write,
    input wire [2:0] read_addr1,
    input wire [2:0] read_addr2,
    input wire [2:0] write_addr,
    input wire [7:0] write_data,
    output reg [7:0] read_data1,
    output reg [7:0] read_data2
);
    reg [7:0] registers [0:7];

    integer i;

    initial begin
        for (i = 0; i < 8; i = i + 1)
            registers[i] = 8'h00;
        registers[3'h1] = 8'h06; // R1 = 6 (for addition)
        registers[3'h2] = 8'h04; // R2 = 4 (optional)
    end

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (i = 0; i < 8; i = i + 1)
                registers[i] <= 8'h00;
            registers[3'h1] <= 8'h06; // R1 = 6
            registers[3'h2] <= 8'h04; // R2 = 4
        end else if (reg_write && write_addr == 3'b000)
            registers[write_addr] <= write_data;
    end

    always @(*) begin
        read_data1 = registers[read_addr1];
        read_data2 = registers[read_addr2];
    end
endmodule

// ========================================
// 4. ALU MODULE
// ========================================
module alu(
    input wire [7:0] operand_a,
    input wire [7:0] operand_b,
    input wire [3:0] alu_control,
    output reg [7:0] alu_result,
    output reg zero_flag
);
    always @(*) begin
        case (alu_control)
            `ADD: alu_result = operand_a + operand_b;
            `SUB: alu_result = operand_a - operand_b;
            `AND: alu_result = operand_a & operand_b;
            `OR:  alu_result = operand_a | operand_b;
            `XOR: alu_result = operand_a ^ operand_b;
            default: alu_result = operand_a;
        endcase
```

```
    end
endmodule

// ========================================
// 7. MAIN RISC PROCESSOR CORE
// ========================================
module risc_processor_8bit(
    input wire clk,
    input wire reset,
    output wire [7:0] pc_out,
    output wire [7:0] acc_out,
    output wire halt_flag
);
    wire [7:0] pc_current, instruction, reg_data1, reg_data2, alu_result, mem_data,
reg_write_data;
    wire [3:0] opcode = instruction[7:4];
    wire [3:0] operand = instruction[3:0];
    wire zero_flag, pc_write_cu, reg_write, mem_write, mem_read, halt;
    wire [1:0] reg_write_src;
    wire [3:0] alu_control;

    wire load_pc_jump = (opcode == `JMP) || (opcode == `JZ && zero_flag);
    wire [7:0] jump_address = {4'h0, operand};
    wire pc_clk = clk;  // Always run PC with system clock


    assign reg_write_data = (reg_write_src == 2'b10) ? {4'h0, operand} :
                            (reg_write_src == 2'b01) ? mem_data : alu_result;

    assign pc_out = pc_current;
    assign acc_out = RF.registers[0];
    assign halt_flag = halt;

    program_counter PC (.clk(pc_clk),
    .reset(reset),
    .load_pc(load_pc_jump),
    .pc_enable(pc_write_cu),    // <--- connect control signal
    .next_pc_address(jump_address),
    .pc_out(pc_current)
);
    instruction_memory IMEM (.address(pc_current), .instruction(instruction));
    register_file RF (.clk(clk), .reset(reset), .reg_write(reg_write),
                      .read_addr1(3'b000), .read_addr2(operand[2:0]),
                      .write_addr(3'b000), .write_data(reg_write_data),
                      .read_data1(reg_data1), .read_data2(reg_data2));
    alu ALU (.operand_a(reg_data1), .operand_b(reg_data2),
             .alu_control(alu_control), .alu_result(alu_result), .zero_flag(zero_flag));
    data_memory DMEM (.clk(clk), .mem_write(mem_write), .mem_read(mem_read),
                      .address({4'h0, operand}), .write_data(reg_data1), .read_data(mem_data));
    control_unit CU (.opcode(opcode), .zero_flag(zero_flag), .pc_write_cu(pc_write_cu),
                     .reg_write(reg_write), .mem_write(mem_write), .mem_read(mem_read),
                     .reg_write_src(reg_write_src), .alu_control(alu_control), .halt(halt));
endmodule

// ========================================
// 8. HEX DECODER
// ========================================
module hex7seg(
    input  [3:0] nibble,
    output reg [6:0] seg
);
    always @(*) begin
        case (nibble)
            4'h0: seg = 7'b1000000;
            4'h1: seg = 7'b1111001;
            4'h2: seg = 7'b0100100;
            4'h3: seg = 7'b0110000;
            4'h4: seg = 7'b0011001;
            4'h5: seg = 7'b0010010;
            4'h6: seg = 7'b0000010;
            4'h7: seg = 7'b1111000;
            4'h8: seg = 7'b0000000;
            4'h9: seg = 7'b0010000;
            4'hA: seg = 7'b0001000;
            4'hB: seg = 7'b0000011;
            4'hC: seg = 7'b1000110;
            4'hD: seg = 7'b0100001;
            4'hE: seg = 7'b0000110;
            4'hF: seg = 7'b0001110;
        endcase
    end
endmodule
// ========================================
// 11. BINARY TO DECIMAL CONVERTER (0–99)
// ========================================
module bin_to_dec (
    input  [7:0] binary,
    output reg [3:0] tens,
    output reg [3:0] ones
);
    integer value;
    always @(*) begin
        value = binary;
        tens = (value / 10) % 10;
        ones = value % 10;
    end
endmodule


// ========================================
// 9. FPGA WRAPPER FOR DE10-STANDARD BOARD
// ========================================
module project (
    input  wire        CLOCK_50,  // 50 MHz system clock
    input  wire [3:0]  KEY,       // Push-buttons (KEY0 active-low reset)
    input  wire [9:0]  SW,        // Slide switches
    output wire [9:0]  LED,       // LEDs
    output wire [6:0]  HEX0,
    output wire [6:0]  HEX1
);
    wire clk = CLOCK_50;
        // Slow clock divider for visual debug (~3 Hz)
        reg [23:0] slow;
        always @(posedge CLOCK_50)
    slow <= slow + 1;
        wire slow_clk = slow[23];
    wire reset = ~KEY[0];

    wire [7:0] pc_out, acc_out;
    wire halt_flag;

    risc_processor_8bit CPU (
        .clk(slow_clk),
        .reset(reset),
        .pc_out(pc_out),
        .acc_out(acc_out),
        .halt_flag(halt_flag)
    );

    assign LED[7:0] = pc_out;
    assign LED[9:8] = {halt_flag, reset};
wire [3:0] tens_digit, ones_digit;

bin_to_dec DEC (.binary(acc_out), .tens(tens_digit), .ones(ones_digit));

hex7seg h0 (.nibble(ones_digit), .seg(HEX0)); // Ones place
hex7seg h1 (.nibble(tens_digit), .seg(HEX1)); // Tens place

endmodule

// ========================================
// 10. TESTBENCH (for simulation only)
// ========================================
module tb_risc_processor;
    reg clk, reset;
    wire [7:0] pc, acc;
    wire halt_flag;
    risc_processor_8bit uut (.clk(clk), .reset(reset), .pc_out(pc), .acc_out(acc),
.halt_flag(halt_flag));
    initial begin clk = 0; forever #5 clk = ~clk; end
    initial begin
        $display("--- Starting Simulation ---");
        reset = 1; #15; reset = 0;
        wait(halt_flag == 1); #50;
        $display("PC=%h  ACC=%h", pc, acc);
        $stop;
    end
endmodule
```

ADDITION OPERATION:

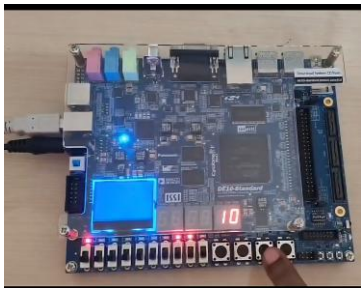
MODELSIM SIMULATION OUTPUT:



FPGA:

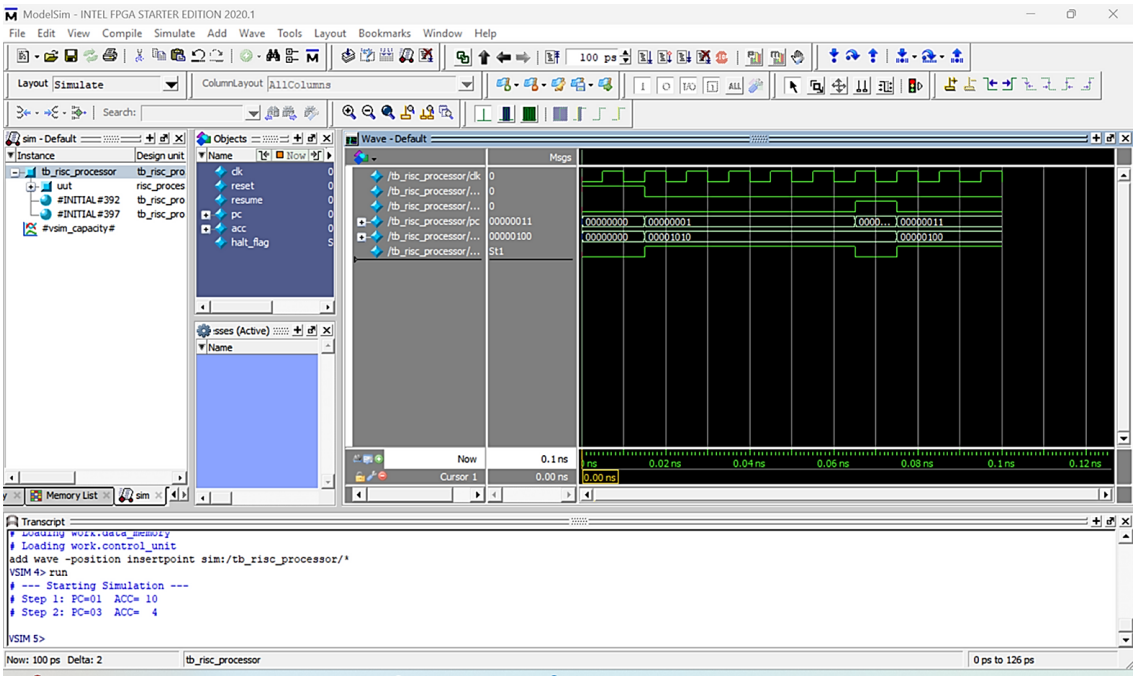
Input:

Output:



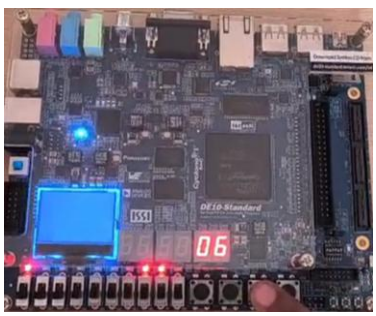
SUBTRACTION OPERATION:

MODELSIM OUTPUT:



FPGA:

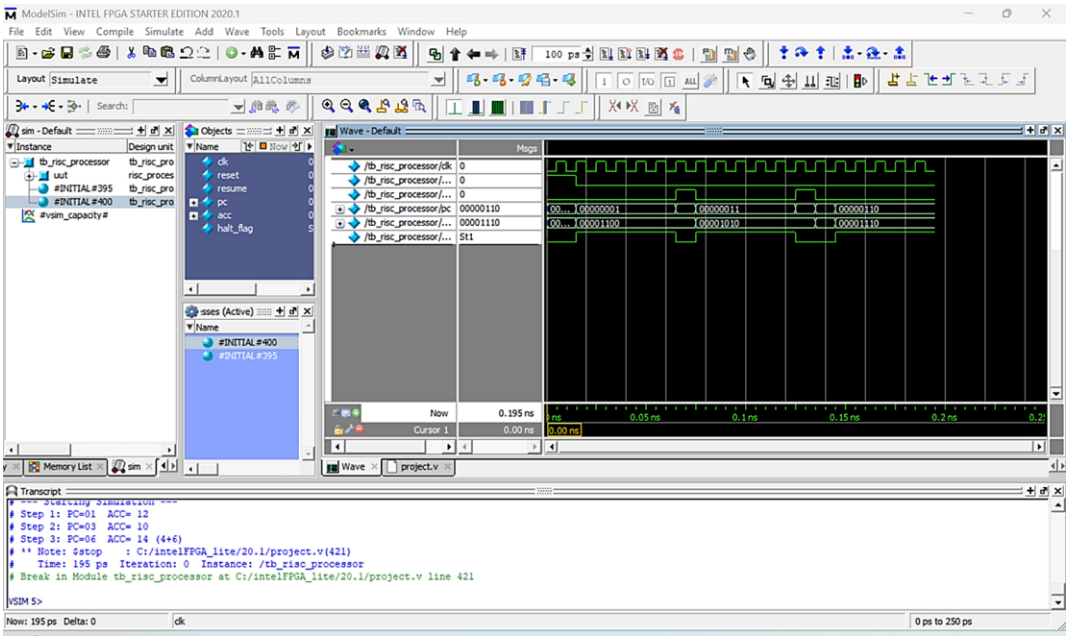
Input:



Output:

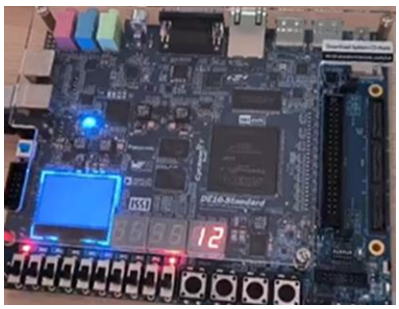
OR OPERATION:

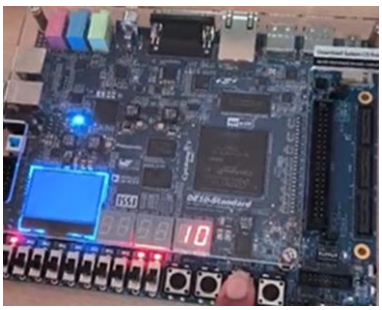
MODELSIM OUTPUT:

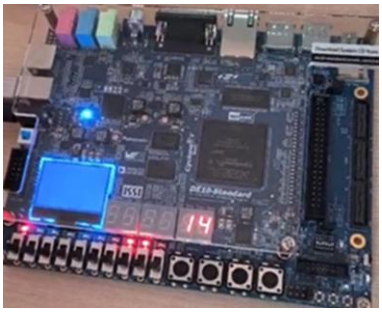


FPGA:

Input: Output:

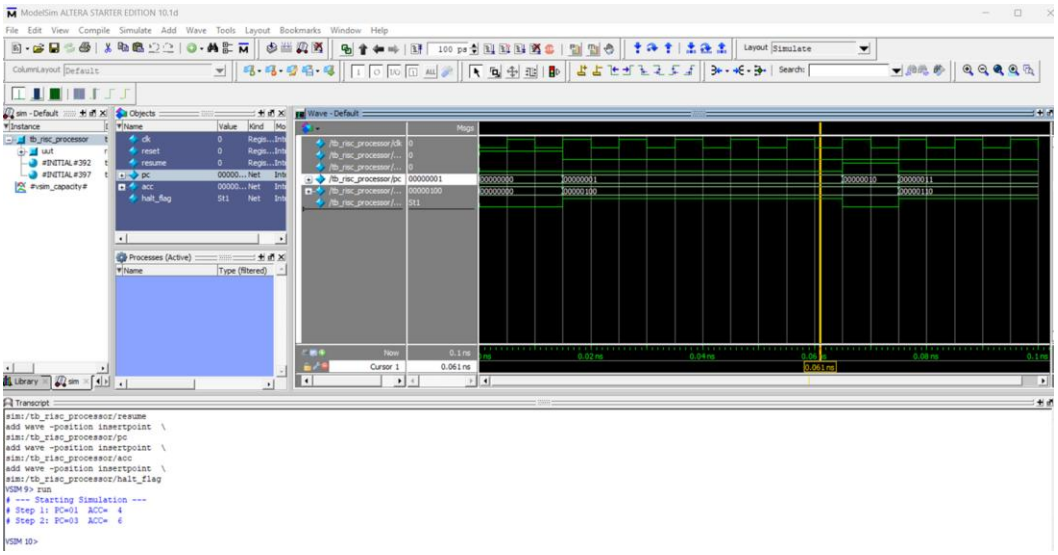






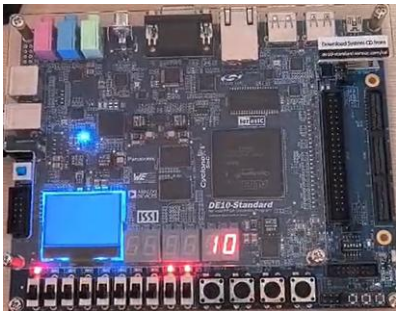
AND OPERATION:

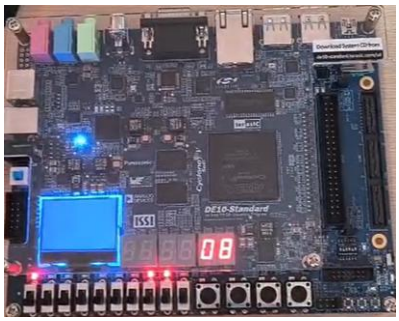
MODELSIM OUTPUT:



FPGA:

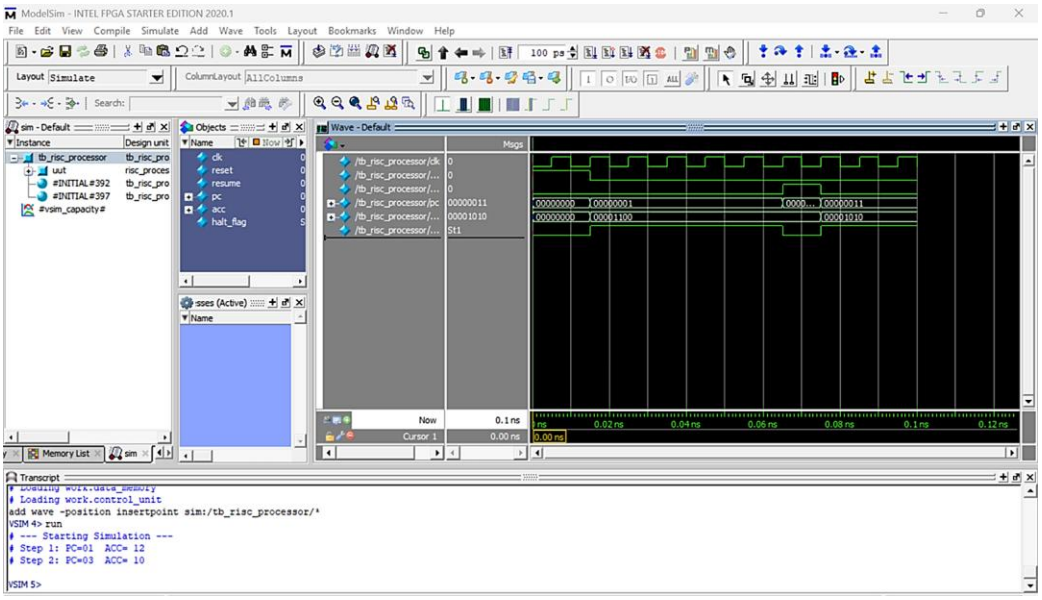
Input: Output:





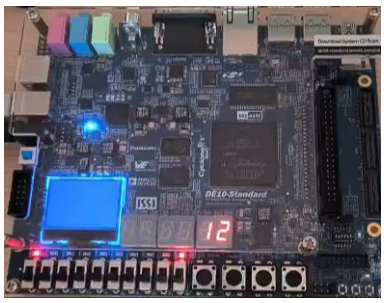
XOR OPERATION:

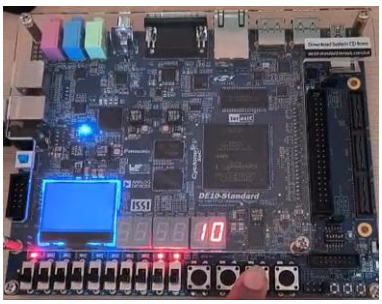
MODELSIM OUTPUT:

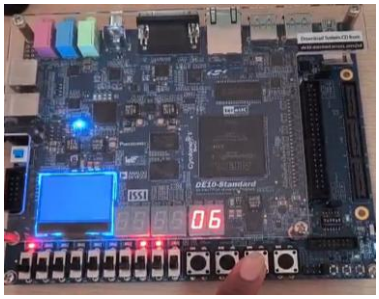


FPGA:

Input: Output:







RESULTS AND DISCUSSION

The 8-bit RISC processor was successfully simulated in ModelSim and implemented on the Altera DE10-Lite FPGA. Simulation results confirmed correct operation of all modules, including the Program Counter, Instruction Memory, ALU, Register File, and Control Unit.

During testing, the processor executed a sample program consisting of Load Immediate, Subtract, Store, and Halt instructions. The accumulator correctly stored the result 4 after performing the operation 10 – 6, which was verified through the 7-segment display output (04) and memory inspection. The HALT flag and LED indicators verified program completion and proper data flow.

The single-cycle design provided predictable instruction timing and validated the core RISC principles of simplicity and efficiency. The FPGA implementation further demonstrated stable real-time operation, with no timing or logic conflicts.

However, the design is limited to a single accumulator and non-pipelined execution. Future enhancements could include expanding the register file, introducing pipelining, and adding peripheral or interrupt support to improve performance and flexibility.

FUTURE SCOPE

The 8-bit RISC processor designed using Verilog can be further enhanced in several ways. The architecture may be upgraded to 16-bit or 32-bit for improved performance and data handling. Pipelining techniques can be implemented to increase instruction throughput. The design can also be realized on an FPGA board for real-time verification.

Future developments may include the addition of advanced instructions, memory and I/O interfaces, and interrupt handling mechanisms. Power optimization techniques and the development of a custom assembler or compiler can make the processor more efficient and user-friendly for embedded applications.

CONCLUSION

Through this systematic methodology, the 8-bit RISC processor was successfully designed, implemented, verified, and tested. Beginning from requirement analysis to final hardware validation, each stage played a crucial role in ensuring the accuracy and functionality of the system. The final processor met all design goals — simplicity, efficiency, modularity, and correctness — demonstrating a clear understanding of fundamental processor design principles using Verilog HDL and FPGA implementation.

REFERENCES

1. E. Ayeh, K. Agbedanu, Y. Morita, O. Adamo and P. Guturu, "FPGA Implementation of an 8-bit Simple Processor," 2008 IEEE Region 5 Conference, Kansas City, MO, USA, 2008,
2. Hernandez Zavala, A., Camacho Nieto, O., Huerta Ruelas, J. A., & Carvallo Domínguez, A. R. (2015).
3. Jeemon, J. (2018). 8-Bit RISC Processor Design using Verilog HDL on FPGA.