# NORTH SOUTH UNIVERSITY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

**Assignment-5**

**Summer 2024**

**CSE332:** Computer Organization and Architecture

**Section: 1**

**Group: 1**

| Name | ID | Work | Contributions |
|---|---|---|---|
| Labiba Faiza Karim | 2221454042 | OS incorporation, MIPS_SCR and related codes (control.v, rom.v, etc,) and wave form | 25% |
| Arefin Amin | 2221766642 | Assembler and incorporated all codes together, MIPS_SCR and wave form | 25% |
| Fatema Tabassum Elma | 2222904042 | OS(min, max, mean) | 20% |
| Ratul Hasan Ankon | 2222568042 | JAL, JR implementation | 15% |
| Raiyan Ahmed | 2221931042 | CPU_mod, CPU_tb | 15% |

# MIPS CPU and OS Project Report

## Introduction

This report documents the design and implementation of a custom MIPS-based CPU and operating system (OS) for the CSE332 course project. The goal was to design a working CPU, develop an assembler for instruction conversion, and simulate the design to demonstrate its functionality. The project adheres to the MIPS ISA (Instruction Set Architecture), ensuring clarity and simplicity while maintaining robust performance.

---

## ISA Design

### Number of Operands

Three operands for flexibility, following the MIPS approach. This allows instructions like ADD R1, R2, R3 to operate on three registers. Additionally, two-operand instructions like SW R1, R2 are included for memory operations.

### Types of Operands

- Registers: For fast operations.
- Memory access: For load/store instructions.

### Operations

### Arithmetic Operations

- ADD: Add two registers and store the result in a third register.
- SUB: Subtract one register from another.
- MUL: Multiply two registers.
- DIV: Divide one register by another.

### Logical Operations

- AND: Bitwise AND of two registers.

- OR: Bitwise OR of two registers.
- NOT: Bitwise NOT of a single register.
- XOR: Bitwise XOR of two registers.

**Branching Operations**

- BEQ: Branch if two registers are equal.
- BNE: Branch if two registers are not equal.
- JUMP: Jump to a specified address.

**Memory Access Operations**

- LW: Load a word from memory into a register.
- SW: Store a word from a register into memory.

**Miscellaneous Operations**

- NOP: No operation (useful for pipeline stalls).

**Instruction Formats**

1. **R-Type (Register)**
    - Used for arithmetic and logical operations.
    - Format: OPCODE (6 bits) | RS (5 bits) | RT (5 bits) | RD (5 bits) | SHAMT (5 bits) | FUNCT (6 bits)
    - Example: ADD R1, R2, R3 (Opcode: 0, Funct: 32).
2. **I-Type (Immediate)**
    - Used for memory access and immediate operations.
    - Format: OPCODE (6 bits) | RS (5 bits) | RT (5 bits) | IMMEDIATE (16 bits)
    - Example: LW R1, 100(R2) (Opcode: 35).
3. **J-Type (Jump)**
    - Used for jump instructions.

- Format: OPCODE (6 bits) | ADDRESS (26 bits)
- Example: J 0x1000 (Opcode: 2).

**Register Table**

| Register Name | Purpose | Notes |
|---|---|---|
| $zero | Always 0 | Constant register |
| $t0-$t7 | Temporary Registers | General-purpose registers |
| $s0-$s7 | Saved Registers | Preserved across calls |
| $ra | Return Address | Stores return address |
| $sp | Stack Pointer | Points to stack memory |

---

## Design and Implementation

1. Mips Code for the OS:

**Min.asm ----**

```
.data
        array: .word 5
        size: .word 5
.text
.globl main
main:
   la $t0, array            #loading the base address of the array
   lw $t1, size             #loading the base size of the array
   lw $t2, 0($t0)                   #first element of the array
   move $t3, $t2
   li $t6, 0                #index
```

```
loop_min:
   beq $t6, $t1, end_min          #if index == size of array, the loop will end
   lw $t7, 0($t0)                 #loading the current array element into $t7
   blt $t7, $t3, update_min       #undating the min value
   j skip_min

update_min:
   move $t3, $t7                        #keeping the min value in $t3

skip_min:                                    #moving to the next element
   addi $t6, $t6, 1               #index += 1
   addi $t0, $t0, 4               #moving to the next word in memory
   j loop_min

end_min:                                     #storing the min value to $a0 for
output
   move $a0, $t3
   li $v0, 10                  #syscall to exit
   syscall
```

**Max.asm ---**

```
.data
        array: .word 5
        size: .word 5
.text
```

```
.globl main

main:
   la $t0, array              #loading the base address of the array
   lw $t1, size               #loading the base size of the array

   lw $t2, 0($t0)                  #first element of the array
   move $t4, $t2
   li $t6, 0                  #index

loop_max:
   beq $t6, $t1, end_max           #if index == size of array, the loop will end
   lw $t7, 0($t0)             #loading the current array element into $t7
   bgt $t7, $t4, update_max   #undating the max value
   j skip_max

update_max:
   move $t4, $t7                   #keeping the max value in $t3

skip_max:                                  #moving to the next element
   addi $t6, $t6, 1           #index += 1
   addi $t0, $t0, 4           #moving to the next word in memory
   j loop_max
```

```
end_max:                                    #storing the max value to $a0 for
output
   move $a0, $t4
   li $v0, 10                  #syscall to exit
   syscall
```

**Mean.asm ---**

```
.data
        array: .word 5
        size: .word 5
.text
.globl main

main:
   la $t0, array               #loading the base address of the array
   lw $t1, size                #loading the base size of the array
   move $t5, $zero                 #initializing sum to 0
   li $t6, 0                   #index

loop_mean:
   beq $t6, $t1, end_mean              #if index == size of array, the loop will end
   lw $t7, 0($t0)                  #loading the current array element into $t7
   add $t5, $t5, $t7               #sum += next_value
   addi $t6, $t6, 1                #index += 1
   addi $t0, $t0, 4                #moving to the next word in memory
```

```
    j loop_mean


end_mean:                                   #calculating the mean: sum/size
    div $t5, $t1            #mean = sum/size = $t5/$t1
    mflo $a0                #storing the mean value to $a0 for output
    li $v0, 10              #syscall to exit
    syscall
```

2. Assembler Development

An assembler was developed to convert assembly language instructions into binary instructions readable by the CPU.

To  make show only the binary values in the output, we had to change the finalassembler.cpp file.

By changing this part of the code ---

```
std::string line;
while (std::getline(inputFile, line)) {
    // Remove the first 10 characters
    line = line.substr(10);

    // Remove empty spaces
    line.erase(std::remove_if(line.begin(), line.end(), ::isspace), line.end());

    // Write the modified line to the output file
    outputFile << line << std::endl;
    //inputFile = outputFile;
}
```

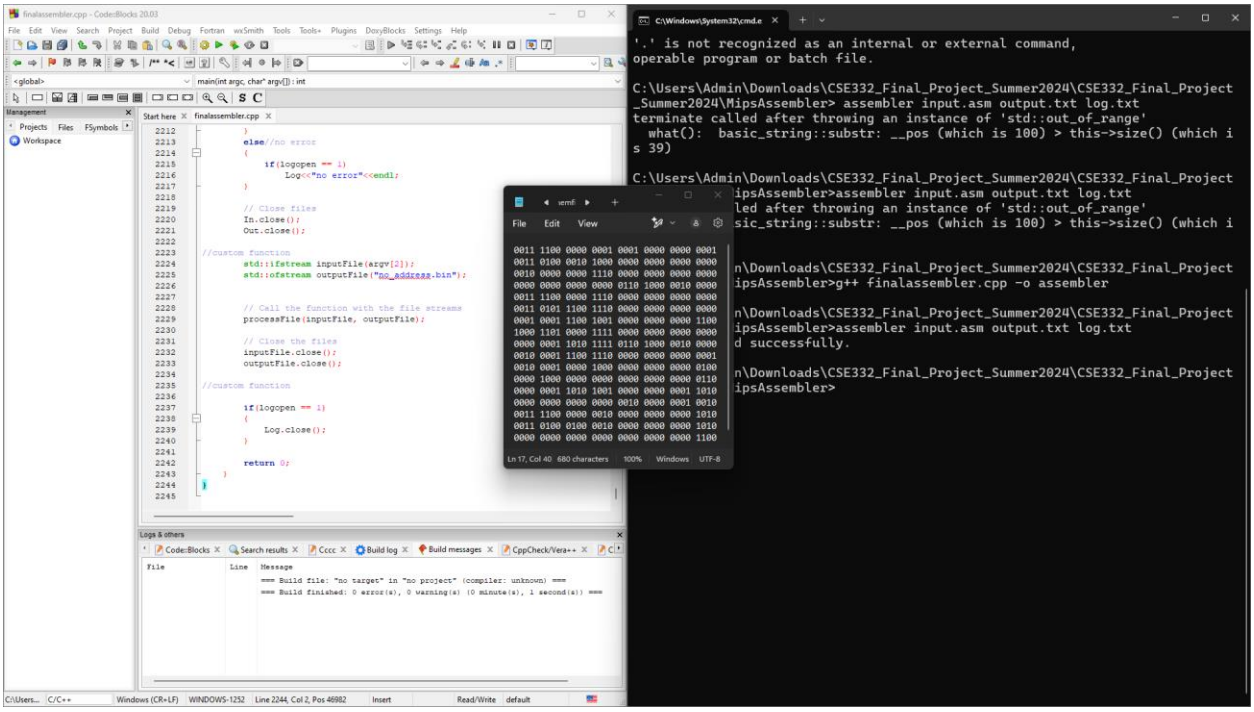
to this ----

```
while (std::getline(inputFile, line)) {
    // Remove the first 30 characters
    line = line.substr(30);

    // Remove empty spaces
    line.erase(std::remove_if(line.begin(), line.end(), ::isspace), line.end());

    // Write the modified line to the output file
    outputFile << line << std::endl;
    //inputFile = outputFile;
}
```

We got the expected binary values in the output.txt file

- Screenshot:
- 

  To see the output here are the command we used.—

  - g++ finalassembler.cpp -o assembler
  - assembler input.asm output.txt log.txt

We will see the compilation error or the message in the log.txt file and we will get the expected binary values in the output.txt file.

2. CPU Simulation in ModelSim

The CPU design was simulated using ModelSim to verify its correctness and functionality.

Here are the Verilog codes we used / modified for the simulation----

CPU_mod.v

```
`timescale 1ns/1ns

module CPU_mod(
    input clk,
    input reset,
    output [31:0] PC,
    output [31:0] Instr,
    output [31:0] ALUResult,
    output ZeroFlag
);

    // Internal wires
    wire [31:0] ReadData, datatwo;
    wire MemWrite, MemtoReg, ALUSrc, RegDst, RegWrite, Jump, PCSrc;
    wire [3:0] ALUControl;

    // Instantiate the control unit
    Controlunit control_unit (
        .Opcode(Instr[31:26]),
        .Func(Instr[5:0]),
```

```
    .Zero(ZeroFlag),
    .MemtoReg(MemtoReg),
    .MemWrite(MemWrite),
    .ALUSrc(ALUSrc),
    .RegDst(RegDst),
    .RegWrite(RegWrite),
    .Jump(Jump),
    .PCSrc(PCSrc),
    .ALUControl(ALUControl)
);

// Instantiate the datapath
Datapath datapath (
    .clk(clk),
    .reset(reset),
    .RegDst(RegDst),
    .RegWrite(RegWrite),
    .ALUSrc(ALUSrc),
    .Jump(Jump),
    .MemtoReg(MemtoReg),
    .PCSrc(PCSrc),
    .ALUControl(ALUControl),
    .ReadData(ReadData),
    .Instr(Instr),
```

```
    .PC(PC),
    .ZeroFlag(ZeroFlag),
    .datatwo(datatwo),
    .ALUResult(ALUResult)
);

// Instantiate instruction memory
InstMemory inst_mem (
    .a(PC[7:2]),    // Address
    .rd(Instr)      // Read Data
);

// Instantiate data memory
DataMemory data_mem (
    .clk(clk),
    .we(MemWrite),
    .a(ALUResult),
    .wd(datatwo),
    .rd(ReadData)
);

Endmodule
```

**CPU_tb.v ---**

```
`timescale 1ns/1ns

module CPU_tb;

    // Inputs
    reg clk;
    reg reset;

    // Outputs
    wire [31:0] PC;
    wire [31:0] Instr;
    wire [31:0] ALUResult;
    wire ZeroFlag;

    // Instantiate the CPU module
    CPU_mod uut (
        .clk(clk),
        .reset(reset),
        .PC(PC),
        .Instr(Instr),
        .ALUResult(ALUResult),
        .ZeroFlag(ZeroFlag)
    );
```

```
// Clock generation
initial begin
    clk = 0;
    forever #5 clk = ~clk; // 10 ns clock period
end

// Test procedure
initial begin
    // Dump waveforms
    $dumpfile("CPU_tb.vcd"); // VCD file for waveform
    $dumpvars(0, CPU_tb);

    // Initialize reset
    reset = 1;
    #10;
    reset = 0;

    // Let the CPU run for some cycles
    #1000; // Adjust based on the number of cycles you want to observe

    // Finish simulation
    $finish;
end
```

```
endmodule
```

**Instruction_Memory.v**

```
`timescale 1ns/1ps
module InstructionMemory (
    input [31:0] address,
    output [31:0] instruction
);
    reg [31:0] memory [0:255];

    initial begin
        $readmemb("instructions.mem", memory);
    end

    assign instruction = memory[address >> 2];

        // Dump instruction memory contents at the end of the simulation
    integer j;
    initial begin
        #1000; // Wait for the simulation to finish
        $display("Final Instruction Memory Contents:");
        for (j = 0; j < 256; j = j + 1) begin
            $display("Memory[%0d] = %h", j, memory[j]);
        end
```

```
    end

endmodule
```

**Register_File.v ----**

```
`timescale 1ns/1ps
module RegisterFile (
    input clk,
    input regWrite,
    input [4:0] readReg1,
    input [4:0] readReg2,
    input [4:0] writeReg,
    input [31:0] writeData,
    output [31:0] readData1,
    output [31:0] readData2
);
    reg [31:0] registers [31:0];

    assign readData1 = registers[readReg1];
    assign readData2 = registers[readReg2];

    always @(posedge clk) begin
        if (regWrite)
```

```
        registers[writeReg] <= writeData;
    end

        /*
        // Monitor and dump register contents to a file after simulation
    initial begin
        $dumpfile("register_dump.vcd");
        $dumpvars(0, RegisterFile);
    end
        */

        // Dump register contents at the end of the simulation
    integer i;
    initial begin
        #1000; // Wait for the simulation to finish
        $display("Final Register File Contents:");
        for (i = 0; i < 32; i = i + 1) begin
            $display("Register[%0d] = %h", i, registers[i]);
        end
    end

endmodule
```

**MIPS_SCP.v ---**

```
`timescale 1ns/1ns

module MIPS_SCP (
    input clk,
    input reset,
    output [31:0] PC,        // Program Counter output
    output [31:0] Instr      // Instruction output
);

    // Internal signals
    wire [31:0] pc_next;
    wire [31:0] pc_current;
    wire [31:0] instruction;

    // Instantiate PC register
    flopr_param #(32) pc_register (
        .clk(clk),
        .reset(reset),
        .d(pc_next),
        .q(pc_current)
    );

    // Instantiate Instruction ROM
```

```
    rom instruction_memory (
        .addr(pc_current),
        .data(instruction)
    );

    // Assign outputs
    assign PC = pc_current;       // Expose Program Counter
    assign Instr = instruction;  // Expose fetched instruction

    // Simple logic to compute next PC (can be replaced with actual control logic)
    assign pc_next = pc_current + 4;

endmodule
```

**MIPS_SCP_tb.v ----**

```
`timescale 1ns/1ns

module MIPS_SCP_tb();

    // Inputs to MUT (MIPS_SCP)
    reg clk;
    reg reset;

    // Outputs from MUT
```

```
wire [31:0] PC;
wire [31:0] Instr;

// Instantiate the MIPS_SCP module
MIPS_SCP mips_inst (
    .clk(clk),
    .reset(reset),
    .PC(PC),
    .Instr(Instr)
);

// Clock generation
initial begin
    clk = 0;
    forever #5 clk = ~clk; // Generate a clock with 10ns period
end

// Test stimulus
initial begin
    reset = 1; // Apply reset
    #10;       // Hold reset for 10ns
    reset = 0; // De-assert reset

    // Simulation end after 200ns
```

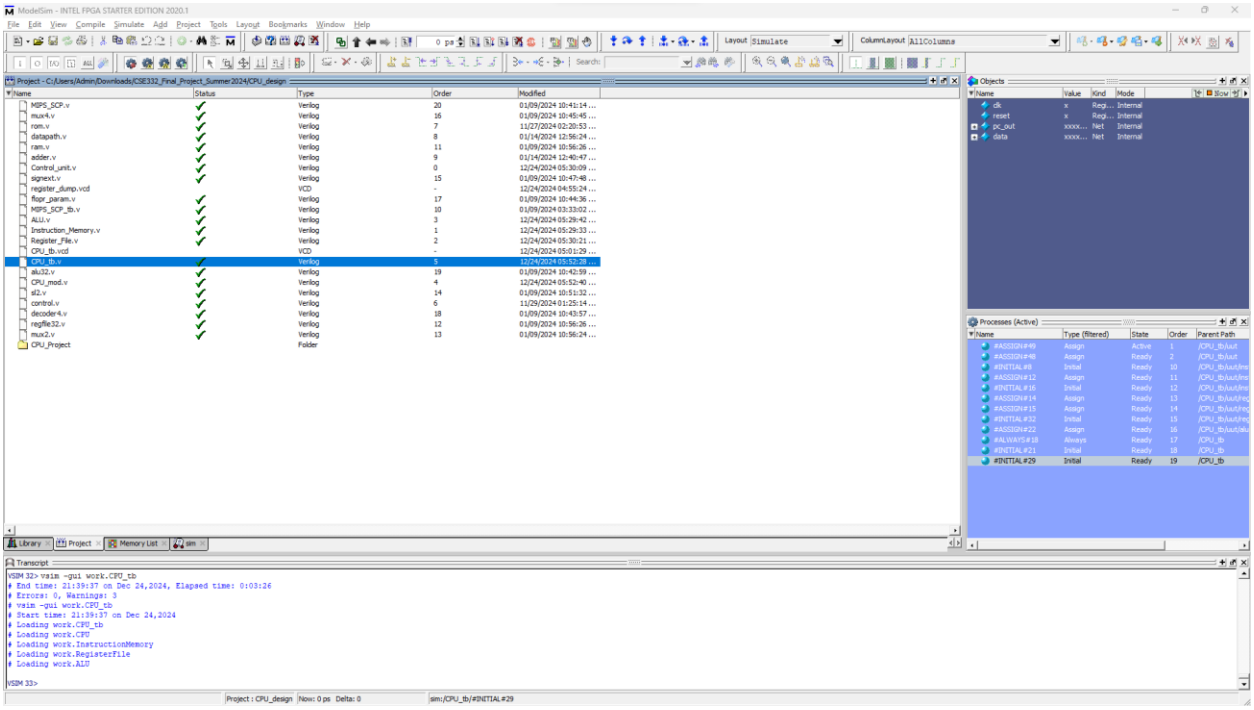
```
        #200;
        $stop;
    end

endmodule
```

to simulate the model we used the following commands ---

- vlog -work work "C:/Users/Admin/Downloads/CSE332_Final_Project_Summer2024/Verilog/CPU_tb.v"
- vsim -gui work.MIPS_SCP_tb

Screenshot:

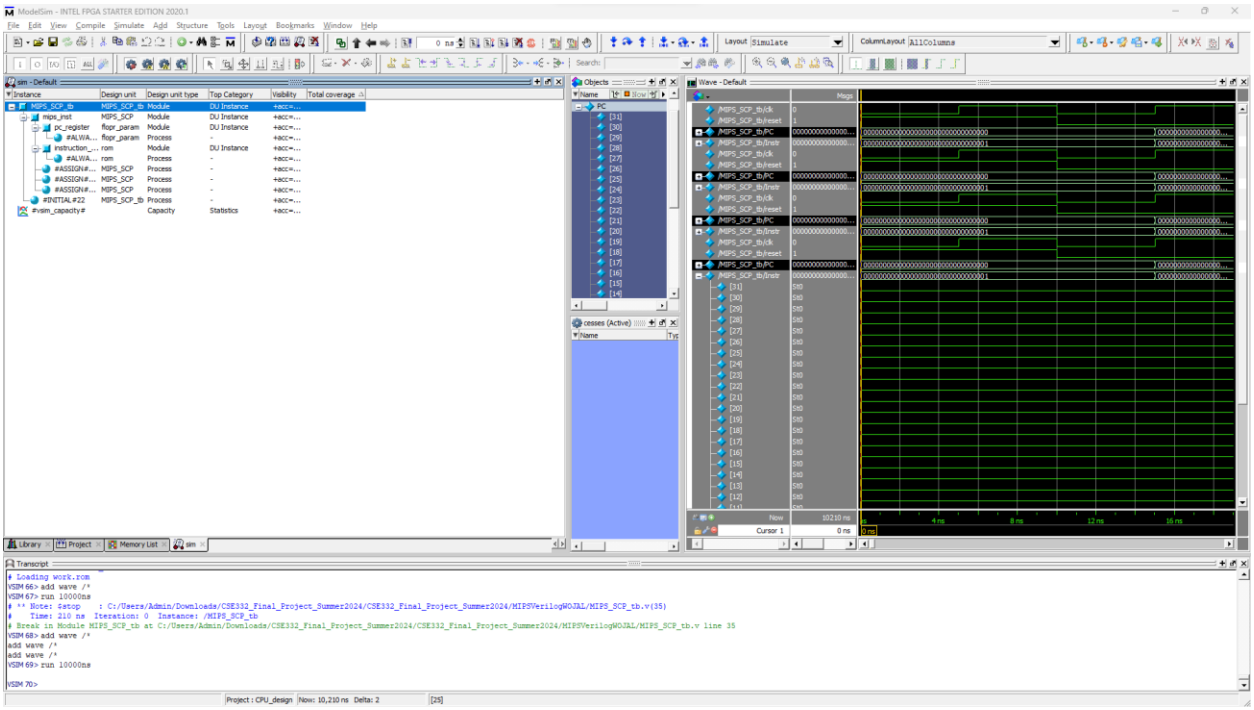



3. Waveform Analysis

Waveform analysis was performed to debug and verify signal behavior during execution.

To see the wave form in the modelSim we used the following commands ----

- add wave /*
- run 1000ns
- Screenshot:



---

**Verilog Codes:**

As for the Verilog code we did change some of the existing files and created some of our own files in order to simulate the program.

**Benchmark Program:**

To test the CPU, the following benchmark programs were executed:

```
`timescale 1ns/1ps


module CPU_tb;
```

```
reg clk;
reg reset;
wire [31:0] pc_out;
wire [31:0] data;

// Instantiate the CPU
CPU uut (
    .clk(clk),        // Correct port name and connection
    .reset(reset),    // Correct port name and connection
    .pc_out(pc_out),  // Correct port name and connection
    .data(data)       // Correct port name and connection
);

// Clock generation
always #5 clk = ~clk;

// Stimulus
initial begin
    clk = 0;
    reset = 1;
    #10 reset = 0;
    #1000 $finish;
end
```

```
    // Monitor Outputs
    initial begin
        $monitor("Time: %t | PC: %h | Data: %h", $time, pc_out, data);
    end
endmodule
```

Screenshot: [Add screenshots showing assembly program execution results here.]

---

**Results and Observations**

1. Instruction Conversion: The assembler correctly converted assembly instructions into binary, adhering to the MIPS instruction format.
2. Simulation Output: The ModelSim simulation validated the execution of instructions, as evident from the output and waveform analysis.
3. Waveform Behavior: The waveform confirmed proper synchronization of signals and data flow.

---

**Conclusion**

This project demonstrated the successful design and simulation of a custom MIPS-based CPU. By implementing an assembler, simulating the design, and verifying results with waveform analysis, the objectives were achieved.

---

**Citation:**

[1] OpenAI, ChatGPT, "Verilog CPU Design Assistance," private communication, Dec. 2024.

[2] OpenAI, ChatGPT, "No Waveform Solution," private communication, Dec. 2024.

[3] OpenAI, ChatGPT, "Module Correction Errors," private communication, Dec. 2024.

[4] OpenAI, ChatGPT, "CPU Module Testbench," private communication, Dec. 2024.

[5] OpenAI, ChatGPT, "Verilog Simulation Warnings," private communication, Dec. 2024.

[6] OpenAI, ChatGPT, "View waveform in model sim" private communication, Dec. 2024.

[7] OpenAI, ChatGPT, "Commands for simulating the CPU in modelsim" private communication, Dec. 2024.

[8] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2014.

[9] M. M. Mano and M. D. Ciletti, Digital Design: With an Introduction to the Verilog HDL, 6th ed. Boston, MA, USA: Pearson, 2018.

[10] S. Palnitkar, Verilog HDL: A Guide to Digital Design and Synthesis, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[11] J. L. Hennessy and D. A. Patterson, "A new architecture for RISC-based systems," IEEE Micro, vol. 1, no. 1, pp. 1-10, Jan. 1981.

[12] T. M. Conte and C. E. Gimarc, "Reducing state transitions in finite state machines for control unit design," IEEE Transactions on Computers, vol. 43, no. 1, pp. 50-60, Jan. 1994.

[13] Intel FPGA, "Introduction to ModelSim Simulation," [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html. [Accessed: Dec. 24, 2024].

[14] F. Vahid, "ALU Design Example in Verilog," [Online]. Available: https://www.vahidbooks.com. [Accessed: Dec. 24, 2024].

[15] E. Perelman, "Verilog MIPS CPU Design Example," GitHub repository, [Online]. Available: https://github.com/eperelman/MIPS_CPU. [Accessed: Dec. 24, 2024].