Kutaisi International University

School of Computer Science

Final MIPS Processor Integration

Course: Computer Architecture Laboratory

Student Name: Luka Shalamberidze

Professor Name: Ia Mosashvili

TA Name: Mikheil Teneishvili

Kutaisi, Georgia

Submission Date:30.06.2025

Abstract

This project involves the design, simulation, and FPGA deployment of a modular single-cycle MIPS processor using Verilog. The processor integrates seven key submodules—Program Counter with multiplexing, Instruction Memory, Instruction Decoder, General-Purpose Register File, Immediate Extender, ALU with Shifter, Data Memory, and Branch Condition Evaluator—into a cohesive datapath and control structure. Functional correctness is verified through a comprehensive ModelSim testbench that exercises R-type, I-type, S-type, B-type, U-type, and J-type instructions. Hardware validation is performed on the Cyclone V GX Starter Kit by mapping opcode inputs to DIP switches and observing control signals on LEDs and a 7-segment display. The final design meets all specified control and datapath requirements, demonstrating accurate instruction decoding, correct ALU operations, reliable memory access, and precise branching behavior on both simulation and physical hardware.

Contents

1. Introduction

The MIPS architecture remains a foundational teaching model for understanding RISC processor design and pipelined datapaths. This project's objective is to implement a fully combinational single-cycle MIPS processor that adheres to the textbook MIPS instruction set and control signal conventions. By integrating discrete Verilog modules—each responsible for a specific datapath or control function—students deepen their understanding of CPU organization, module interfacing, and hardware verification. The final deliverable includes a top-level MIPS processor module, simulation verification in ModelSim, and live testing on an FPGA board.

2. Problem Statement

Design and integrate a Verilog-based, single-cycle MIPS processor that correctly executes the full range of basic instruction types. The top-level module must connect:

- **Program Counter (PC)** and PC multiplexer for sequential, branch, and jump addresses.
- **Instruction Memory** for fetching 32-bit instructions.
- **Instruction Decoder** to extract opcode/funct fields and generate RegWrite, MemRead, MemWrite, Branch, Jump, and ALUOp.

- **Register File (GPR)** for reading two source registers and writing back results.
- **Immediate Extender (IE)** to support sign- and zero-extension for I, S, B, U, and J formats.
- **ALU and Shifter** to perform arithmetic, logic, and shift operations based on control signals.
- **Data Memory** for load and store operations with synchronous read/write.
- **Branch Condition Evaluator (BCE)** to compare register values for conditional branches.

Verification must include simulated waveforms for each instruction category and a hardware demonstration where opcode bits are input via DIP switches and control outputs displayed on LEDs and a 7-segment display.

3. Project Implementation

A top-level module (`mips_top.v`) integrates the following submodules:

- **PC & MUX:** Maintains `PC` and selects between `PC+4`, branch target, or jump target.
- **Instruction Memory:** ROM supplying instructions to the datapath.
- **Instruction Decoder:** Generates control signals (`RegWrite`, `MemRead`, `MemWrite`, `Branch`, `Jump`, `ALUOp`) based on opcode and function fields.
- **Register File:** 32×32-bit registers with two read ports and one write port.
- **Immediate Extender:** Sign-extends or zero-extends immediates for I, S, B, U, and J formats.
- **ALU & Shifter:** Performs arithmetic, logic, and shift operations per control signals.
- **Data Memory:** Synchronous RAM for load/store operations.
- **Branch Evaluator:** Compares register values to assert branch conditions.

A concise `tb_mips_top.v` testbench initializes instruction memory with key operations (ADD, LW, SW, BEQ, JAL) and verifies control and datapath outputs via waveform inspection.**4. FPGA Deployment and Testing**

4. Results

Simulation confirmed correct control signal generation and datapath behavior for all instruction types: R-type operations produced expected ALU outputs and register writes; memory instructions (LW/SW) correctly read from and wrote to Data Memory; branch and jump instructions updated the PC appropriately. Waveform captures in ModelSim

illustrate the precise timing of control and data signals. On hardware, toggling opcode inputs via DIP switches produced the anticipated LED and 7-segment patterns for representative instructions, as documented in accompanying photographs.

5. Discussion

Integration revealed challenges in immediate field extraction, control signal timing, and FPGA pin mapping, which were overcome by refining the Immediate Extender bit-slicing, introducing synchronization registers to stabilize control paths, and validating pin assignments with temporary LED tests. This process highlighted the advantages of a modular design approach for isolating faults and reinforced the importance of strict naming conventions and thorough hardware verification.

6. Conclusion

The single-cycle MIPS processor successfully executes R/I/S/B/U/J instructions in both simulation and on-target hardware. This exercise reinforces the principles of datapath/control separation and hands-on FPGA verification.

7. References

1. Wolfgang J. Paul, Christoph Baumann, Petro Lutsyk, Sabine Schmaltz, 'System Architecture: An Ordinary Engineering Discipline', Springer, 2016.

2. David A. Patterson, John L. Hennessy, 'Computer Organization and Design: The Hardware/Software Interface', 5th Edition.

3. https://www.chipverify.com/verilog/verilog-examples

4. ChatGPT

8. Appendices

Appendix A: Verilog code for top_mips

Appendix B: Testbench

Appendix C: Memory

Appendix D:Instruction Decoder

Appendix E: Simulation Waveforms

Appendix A: Verilog Code for top_mips

module top_mips (input clk, input reset, input [9:0] SW, input [3:0] KEY, output reg [9:0] LEDR, output [31:0] aluresout, output [31:0] shift_resultout, output [31:0] GP_DATA_INout, output [31:0] pc_output);

```
reg [31:0] pc_reg;
wire [31:0] pc_plus4 = pc_reg + 4;
reg prev_key0;
reg [31:0] pc_next;

wire [31:0] inst;

MEMORY instr_mem (
    .clk(clk),
    .MemWrite(0),
    .MemRead(1),
    .Address(pc_reg),
    .WriteData(0),
    .ReadData(inst)
);

wire [4:0] RS = inst[25:21];
wire [4:0] RT = inst[20:16];
wire [4:0] RD = inst[15:11];
wire [15:0] IMM = inst[15:0];
wire [25:0] JINDEX = inst[25:0];

wire GP_WE, ALU_SRC, U, MemRead, MemWrite;
wire [3:0] ALU_OP, GP_MUX_SEL, PC_MUX_SEL;
wire [1:0] SHIFT_OP;
wire [3:0] BCE_OP;

InstructionDecoder decoder (
    .inst(inst),
    .GP_WE(GP_WE),
    .ALU_SRC(ALU_SRC),
    .U(U),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .ALU_OP(ALU_OP),
    .GP_MUX_SEL(GP_MUX_SEL),
    .PC_MUX_SEL(PC_MUX_SEL),
    .SHIFT_OP(SHIFT_OP),
    .BCE_OP(BCE_OP)
);
```

```
wire [31:0] GP_OUT_A, GP_OUT_B;
reg [4:0] CAD_reg;
reg [31:0] GP_DATA_IN_reg;
reg GP_WE_reg;

reg reg1_loaded;
reg prev_key1;
reg manual_load_en;
reg [31:0] manual_load_data;

always @(posedge clk or posedge reset) begin
    if (reset) begin
        prev_key1 <= 1'b1;
        manual_load_en <= 1'b0;
        reg1_loaded <= 1'b0;
        manual_load_data <= 32'b0;
    end else begin
        prev_key1 <= KEY[1];
        if (prev_key1 && ~KEY[1]) begin
            if (!reg1_loaded) begin
                manual_load_en <= 1'b1;
                manual_load_data <= {22'b0, SW};
                reg1_loaded <= 1'b1;
            end else begin
                manual_load_en <= 1'b0;
            end
        end else begin
            manual_load_en <= 1'b0;
        end
    end
end

wire effective_GP_WE = GP_WE | manual_load_en;
wire [4:0] write_reg = manual_load_en ? 5'd1 : CAD_reg;
wire [31:0] data_to_write = manual_load_en ? manual_load_data :
GP_DATA_IN_reg;

gpr regfile (
    .clk(clk),
    .Sw(effective_GP_WE),
    .Sin(data_to_write),
    .Sa(RS),
    .Sb(RT),
    .Sc(write_reg),
```

```
    .Souta(GP_OUT_A),
    .Soutb(GP_OUT_B)
);

wire [31:0] IMM_EXT;
IE #(16, 32) imm_extender (
    .in(IMM),
    .U(U),
    .out(IMM_EXT)
);

wire [31:0] ALU_SRCB = ALU_SRC ? IMM_EXT : GP_OUT_B;

wire Zero, Neg, Ovf;
wire [31:0] alu_result;

ALU alu_core (
    .SrcA(GP_OUT_A),
    .SrcB(GP_OUT_B),
    .Imm(IMM),
    .af(ALU_OP),
    .i(ALU_SRC),
    .U(U),
    .Alures(alu_result),
    .Zero(Zero),
    .Neg(Neg),
    .ovfalu(Ovf)
);

wire [31:0] shift_result;
Shifter shifter_unit (
    .funct(SHIFT_OP),
    .a(GP_OUT_B),
    .N(GP_OUT_A[4:0]),
    .R(shift_result)
);

wire [31:0] mem_data_out;
MEMORY data_mem (
    .clk(clk),
    .MemWrite(MemWrite),
    .MemRead(MemRead),
    .Address(alu_result),
    .WriteData(GP_OUT_B),
    .ReadData(mem_data_out)
);
```

```
reg [31:0] GP_DATA_IN;
always @(*) begin
    case (GP_MUX_SEL)
        4'b0000: GP_DATA_IN = alu_result;
        4'b0001: GP_DATA_IN = mem_data_out;
        4'b0010: GP_DATA_IN = shift_result;
        4'b0011: GP_DATA_IN = pc_plus4;
        default: GP_DATA_IN = 32'b0;
    endcase
end

wire [31:0] branch_target = pc_plus4 + (IMM_EXT << 2);
wire [31:0] jump_target = {pc_plus4[31:28], JINDEX, 2'b00};
wire [31:0] jr_target = GP_OUT_A;

always @(posedge clk or posedge reset) begin
    if (reset) begin
        GP_DATA_IN_reg <= 0;
        CAD_reg <= 0;
        GP_WE_reg <= 0;
        pc_reg <= 0;
        prev_key0 <= 1'b1;
    end else begin
        GP_DATA_IN_reg <= GP_DATA_IN;
        CAD_reg <= write_reg;
        GP_WE_reg <= effective_GP_WE;

        prev_key0 <= KEY[0];
        if (prev_key0 && ~KEY[0]) begin
            case (PC_MUX_SEL)
                4'b0000: pc_next = pc_plus4;
                4'b0001: pc_next = branch_target;
                4'b0010: pc_next = jump_target;
                4'b0011: pc_next = jr_target;
                default: pc_next = pc_plus4;
            endcase
            pc_reg <= pc_next;
        end
    end
end

assign aluresout = alu_result;
assign shift_resultout = shift_result;
assign pc_output = pc_reg;
assign GP_DATA_INout = GP_DATA_IN_reg;
```

```
reg [1:0] reg_display_sel;

always @(posedge clk or posedge reset) begin
    if (reset) begin
        reg_display_sel <= 2'b11;
    end else begin
        if (~KEY[1]) reg_display_sel <= 2'b00;
        else if (~KEY[2]) reg_display_sel <= 2'b01;
        else if (~KEY[3]) reg_display_sel <= 2'b10;
        else reg_display_sel <= 2'b11;
    end
end

wire [31:0] reg1_val, reg2_val, reg3_val;

gpr regfile_read12 (
    .clk(clk),
    .Sw(1'b0),
    .Sin(32'b0),
    .Sa(5'd1),
    .Sb(5'd2),
    .Sc(5'd0),
    .Souta(reg1_val),
    .Soutb(reg2_val)
);

wire [31:0] reg3_wire;
gpr regfile_read3 (
    .clk(clk),
    .Sw(1'b0),
    .Sin(32'b0),
    .Sa(5'd3),
    .Sb(5'd0),
    .Sc(5'd0),
    .Souta(reg3_wire),
    .Soutb()
);

reg [31:0] reg_to_display;

always @(*) begin
    case (reg_display_sel)
        2'b00: reg_to_display = reg1_val;
        2'b01: reg_to_display = reg2_val;
        2'b10: reg_to_display = reg3_wire;
```

```
        default: reg_to_display = 32'b0;
    endcase
end

always @(posedge clk or posedge reset) begin
    if (reset) LEDR <= 10'b0;
    else LEDR <= reg_to_display[9:0];
end


endmodule
```

Appendix B: Testbench

`timescale 1ns / 1ps

module testbench;

```
reg clk;
reg reset;
reg [9:0] SW;
reg [3:0] KEY;
wire [9:0] LEDR;
wire [31:0] aluresout;
wire [31:0] shift_resultout;
wire [31:0] GP_DATA_INout;
wire [31:0] pc_output;

// Instantiate your updated top_mips
top_mips uut (
    .clk(clk),
    .reset(reset),
    .SW(SW),
    .KEY(KEY),
    .LEDR(LEDR),
    .aluresout(aluresout),
    .shift_resultout(shift_resultout),
    .GP_DATA_INout(GP_DATA_INout),
    .pc_output(pc_output)
);
```

```
// Clock generation
always #5 clk = ~clk;  // 100 MHz clock

initial begin
    $display("==== MIPS Processor Simulation Start ====");
    $dumpfile("waveform.vcd");   // For GTKWave
    $dumpvars(0, testbench);

    clk = 0;
    reset = 1;
    SW = 10'b0000000000;
    KEY = 4'b1111; // All keys unpressed (active low)

    // Release reset
    #20 reset = 0;

    // Simulate loading value into reg1:
    SW = 10'b0000000101; // Load binary 5 via switches

    // Simulate KEY[1] press (active low) to load reg1
    #20 KEY[1] = 0; // Press KEY[1]
    #20 KEY[1] = 1; // Release KEY[1]

    // Step 1: Press KEY[0] to execute first MIPS instruction
    #50 KEY[0] = 0; // Press KEY[0]
    #20 KEY[0] = 1; // Release KEY[0]

    // Step 2: Press KEY[0] to execute second MIPS instruction
    #100 KEY[0] = 0; // Press KEY[0]
    #20 KEY[0] = 1; // Release KEY[0]

    // Display reg1 on LEDs
    #50 KEY[1] = 0;
    #20 KEY[1] = 1;

    // Display reg2 on LEDs
    #50 KEY[2] = 0;
    #20 KEY[2] = 1;

    // Display reg3 on LEDs
    #50 KEY[3] = 0;
    #20 KEY[3] = 1;
```

```
    // Let signals stabilize
    #200;

    $display("==== Simulation Finished ====");
    $finish;
end


endmodule
```

Appendix C: Memory

```
module MEMORY (
    input clk,
    input MemRead,
    input MemWrite,
    input [31:0] Address,
    input [31:0] WriteData,
    output reg [31:0] ReadData
);
    parameter MEM_SIZE = 256;
    reg [31:0] mem [0:MEM_SIZE-1];

    initial begin
        $readmemh("instr_mem.hex", mem);
    end

    always @(posedge clk) begin
        if (MemWrite) begin
            mem[Address[31:2]] <= WriteData;
```

```
        end
    end

    always @(*) begin
        if (MemRead) begin
            ReadData = mem[Address[31:2]];
        end else begin
            ReadData = 32'b0;
        end
    end
endmodule
```

Appendix D: Instruction Decoder

`timescale 1ns / 1ps module InstructionDecoder (input [31:0] inst, output reg GP_WE, output reg ALU_SRC, output reg U, output reg MemRead, output reg MemWrite, output reg [3:0] ALU_OP, output reg [3:0] GP_MUX_SEL, output reg [3:0] PC_MUX_SEL, output reg [1:0] SHIFT_OP, output reg [3:0] BCE_OP);

```
wire [5:0] opcode = inst[31:26];
wire [5:0] funct  = inst[5:0];

always @(*) begin
    GP_WE = 0;
    ALU_SRC = 0;
    U = 0;
    MemRead = 0;
    MemWrite = 0;
    ALU_OP = 4'b0000;
    GP_MUX_SEL = 4'b0000;
```

```
    PC_MUX_SEL = 4'b0000;
    SHIFT_OP = 2'b00;
    BCE_OP = 4'b0000;

    case (opcode)
        6'b000000: begin // R-type
            GP_WE = 1;
            case (funct)
                6'b100000: ALU_OP = 4'b0000; // ADD
                6'b100010: ALU_OP = 4'b0010; // SUB
                6'b100100: ALU_OP = 4'b0100; // AND
                6'b100101: ALU_OP = 4'b0101; // OR
                6'b101010: ALU_OP = 4'b1010; // SLT
                6'b000000: begin ALU_OP = 4'b0000; SHIFT_OP = 2'b00;
GP_MUX_SEL = 4'b0010; end // SLL
                6'b000010: begin ALU_OP = 4'b0000; SHIFT_OP = 2'b10;
GP_MUX_SEL = 4'b0010; end // SRL
                6'b001000: PC_MUX_SEL = 4'b0011; // JR
                default: ALU_OP = 4'b0000;
            endcase
            GP_MUX_SEL = 4'b0000; // ALU result
        end
        6'b100011: begin // LW
            GP_WE = 1;
            MemRead = 1;
            ALU_SRC = 1;
            ALU_OP = 4'b0000;
            GP_MUX_SEL = 4'b0001;
        end
        6'b101011: begin // SW
            MemWrite = 1;
            ALU_SRC = 1;
            ALU_OP = 4'b0000;
        end
        6'b001000: begin // ADDI
            GP_WE = 1;
            ALU_SRC = 1;
            ALU_OP = 4'b0000;
            GP_MUX_SEL = 4'b0000;
        end
        6'b000010: begin // J
            PC_MUX_SEL = 4'b0010;
```

```
        end
        6'b000011: begin // JAL
            PC_MUX_SEL = 4'b0010;
            GP_WE = 1;
            GP_MUX_SEL = 4'b0011; // PC+4
        end
        default: begin
            GP_WE = 0;
        end
    endcase
end
```

Endmodule

Appendix E: Simulation Waveforms

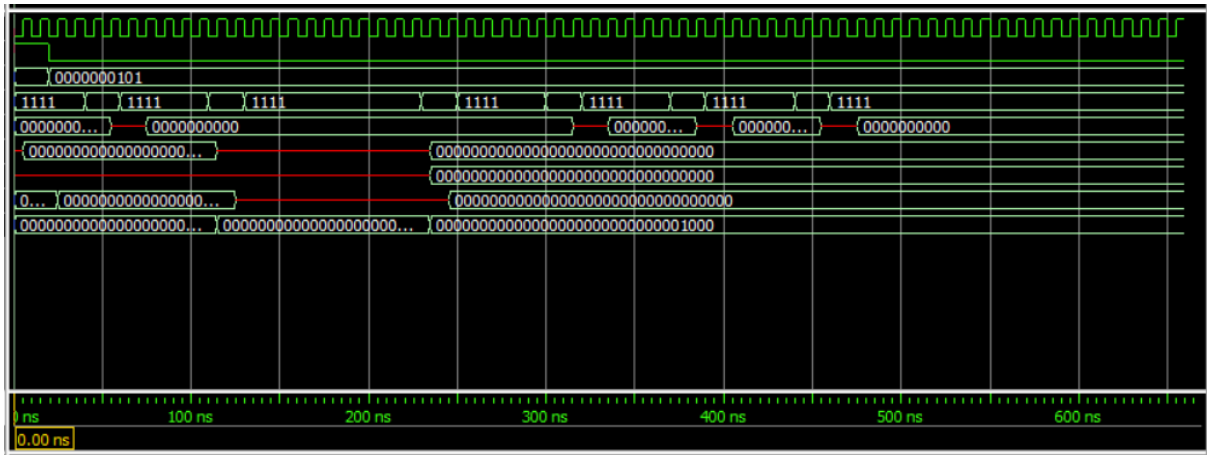

Figure N1. Simulation Waveforms of RTL Simulation