PROJECT REPORT
32-Bit 5-Stage Pipelined MIPS Processor
(Phase 1: RTL Architecture)

Author: Mina S. ElHanash
Junior, Electronic and Communication Engineering
Institution: Ain Shams University, Faculty of Engineering
Date: February 2026
Repository: https://github.com/MinaElHanash/MIPS

Table of Contents

ABSTRACT
This report details the Phase 1 design, synthesis, and implementation of a 32-bit, 5-stage pipelined MIPS processor. The core was developed using Verilog HDL with a focus on structural microarchitecture, incorporating full hazard detection and data forwarding to resolve Read-After-Write (RAW) and control hazards. The design was synthesized using Xilinx Vivado targeting a Spartan-7 FPGA (xc7s50csga324-1). Post-implementation timing analysis confirms the processor successfully meets a 100 MHz clock constraint, achieving a Maximum Clock Frequency (Fmax) of 105.7 MHz. This completed RTL architecture establishes the foundation for Phase 2, which will focus on functional verification using SystemVerilog and the Universal Verification Methodology (UVM).

1.0 INTRODUCTION

1.1 Background on MIPS Architecture

The MIPS (Microprocessor without Interlocked Pipelined Stages) architecture is a cornerstone of modern digital design, widely recognized as one of the most elegant implementations of Reduced Instruction Set Computer (RISC) principles. Developed in the early 1980s, the MIPS ISA was designed with a strict load/store architecture, meaning all arithmetic and logical operations are performed exclusively on data held within the processor's internal registers, while memory is accessed solely through dedicated load and store instructions. This deliberate simplicity, combined with a fixed 32-bit instruction length, allows the hardware to decode and execute instructions with high efficiency.

To maximize instruction throughput, the MIPS architecture inherently supports pipelining. By dividing instruction execution into five distinct, sequential stages-Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB)-the processor can process multiple instructions concurrently. While pipelining dramatically increases the theoretical execution rate by ensuring the CPU completes one instruction per clock cycle under ideal conditions, it introduces significant microarchitectural challenges known as hazards.

Hazards occur when the pipeline must be stalled to prevent incorrect execution. **Structural hazards** arise from hardware resource conflicts, **data hazards** (such as Read-After-Write) occur when an instruction depends on the result of a preceding, uncompleted instruction, and **control hazards** are caused by branch instructions altering the flow of the program counter. A sophisticated MIPS implementation must move beyond basic interlocked stalling and incorporate dynamic hardware solutions, such as data forwarding and branch prediction, to maintain pipeline efficiency.

1.2 Project Objectives

The primary objective of this project is to architect, synthesize, and validate a fully functional 32-bit, 5-stage pipelined MIPS processor at the Register Transfer Level (RTL). Rather than relying on high-level behavioral modeling, this project focuses on structural, synthesizable microarchitecture, serving as a comprehensive exercise in digital logic design and physical hardware implementation.

The specific objectives of Phase 1 include:

- **Gate-Level RTL Implementation:** To develop a complete Verilog HDL codebase that accurately models the MIPS data path and control logic, strictly adhering to synthesizable coding guidelines.
- **Dynamic Hazard Resolution:** To implement an active Forwarding Unit capable of bypassing the register file to resolve Read-After-Write (RAW) data hazards dynamically. Additionally, to construct a Hazard Detection Unit that injects hardware bubbles (stalls) only when data forwarding is mathematically insufficient, such as during Load-Use hazards.
- **Physical Synthesis and Benchmarking:** To move the design beyond idealized software simulation by targeting a modern Xilinx Spartan-7 FPGA (xc7s50csga324-1). This includes analyzing the physical mapping of logic gates to extract real-world hardware utilization metrics (Slice LUTs and Registers) and performing static timing analysis to verify the processor achieves a minimum Maximum Clock Frequency (Fmax) of 100 MHz.
- **Establishing a Verification Baseline:** To create a mathematically sound and physically proven RTL core (Phase 1) that will serve as the Design Under Test (DUT) for Phase 2. The ultimate objective of this multi-phase project is to subject this synthesizable core to rigorous functional verification using SystemVerilog and the Universal Verification Methodology (UVM) to achieve complete code and functional coverage.

2.0 PROJECT SPECIFICATIONS

Course Project (Major Task)

***Fall* 2025**

Aim

This project aims to design, simulate, and synthesize a modified single-cycle and pipelined RISC-style processor in Verilog on an FPGA that supports a subset of the RISC instruction set. The project includes the architecture form, where the students will focus on developing a tailored architecture to perform a specific function.

The *processor* should support:

- the arithmetic and logic operations: **add, sub, and, or, andi, addi, slt**
- Memory-reference: **lw, sw**
- jumping and branching: **j, beq**
- In addition to three new instructions: **jmn**, **swi**, **PMC**

where the format and description of each instruction is as follows:

	6-Bits	5-Bits	5-Bits	5-Bits	5-Bits	6-Bits
I-type	op	rs	rt	immediate		
J-type	op	---	address			
R-type	op	rs	rt	rd	---	func

pmc (rt), imm(rs)	# Perform two operations: ➢ **PC = Memory[R[rt]]**; set the value of **PC** to the loaded data from memory location **Memory[R[rt]]** ➢ **Memory[R[rs] + imm] = PC + 4**; store the new value of **PC** to a memory location **Memory[R[rs] + imm]**
jmn imm(rs)	**#Indirect Jump**. **PC = Memory[imm + R[rs]],** I-format instruction will cause the processor to jump to the address stored in the word at memory location **imm + R[rs].**
swi rt, imm(rs)	#I-format instruction will perform two operations: ➢ it stores the contents of **R[rt]** at the memory address **(R[rs] + imm)** then, increments **R[rs]** by the immadiate **imm.**

Phase 1 — ALU & Register File

Design and implement the ALU and register file:

- ALU ops: add, sub, and, or, andi, addi, slt, maybe logical immediates.
- Register file: 32×32 registers, two read ports, one write port, synchronous write, async read.

Deliverables

- RTL code (fully commented)
- Testbench and waveform screenshots

Phase 2 — Single-Cycle Processor

Integrate ALU and register file into a single-cycle CPU (fetch → decode → execute → mem → writeback) that executes the proposed instruction subset.

Specs

- Minimal ISA mentioned above.
- Memories: instruction memory (initial content from assembled test programs), data RAM (read/write).

Verification

- Use assembly test programs saved in Imem or load from a hex file.
- Check expected memory and register states after program execution.
- Create self-checking testbenches that compare expected registers/memory.

Synthesis

- Run the CAD tool to produce a flattened gate netlist.

Deliverables

- Single-cycle RTL + testbenches, simulation traces, synthesized netlist.
- Synthesis and timing reports describing datapath and control signals.

Phase 3 — Pipelined Processor

Convert a single-cycle CPU into a 5-stage pipelined CPU: IF, ID, EX, MEM, WB.

Architecture

- Add pipeline registers between stages and Split control and datapath accordingly
- Manage instruction fetch PC update, branch resolution (choose branch in EX or ID — pick one and document)

Verification & Simulation

- Use longer assembly test programs that reveal hazards (e.g., dependent instruction sequences)
- Create a self-checking harness that runs for *N* cycles and compares the final register/memory content.

Synthesis

- Synthesize pipeline; check timing reports using FPGA flow

Deliverables

- RTL with pipeline registers, test programs that show pipeline behavior, simulation waveforms
- Proof of correct output despite pipeline (before hazard handling)
- Synthesis and timing reports describing datapath and control signals.

Phase 4 — Hazard Detection & Forwarding (Stalling & Bypass)

Implement the hazard detection unit (HDU) and forwarding (bypass) unit to eliminate or reduce stalls caused by data hazards; implement control for load-use stall.

Verification

- Write specific test programs with sequences requiring forwarding and load-use:
  - e.g., ADD R1, R2, R3 followed by SUB R4, R1, R5 (forwarding from EX)
  - LW R1, 0(R2) followed by ADD R3, R1, R4 (load-use: one cycle stall required)
- Self-checking testbenches verify register contents at known cycle counts
- Provide waveforms showing forwarding mux selection and inserted bubble
- **Corner cases**
  - Multi-cycle memory/IO (if implemented)
  - Branch + load interactions
  - Structural hazards if you reuse memory ports — avoid by design or document

Deliverables

- Hazard and forwarding RTL, testbenches with annotated waveforms, final reports describing the algorithm, synthesis and verification

3.0 DESIGN METHODOLOGY

The processor was developed utilizing a top-down, modular design methodology, structured across iterative phases to isolate and verify complexity at each step.

3.1 Phase 1 of the Design: Single-Cycle Architecture Validation

The initial phase of the design focused on developing a fundamental single-cycle MIPS microarchitecture. This approach established the baseline data path and validated the core control logic, ensuring that all operations executed correctly within a single clock period.

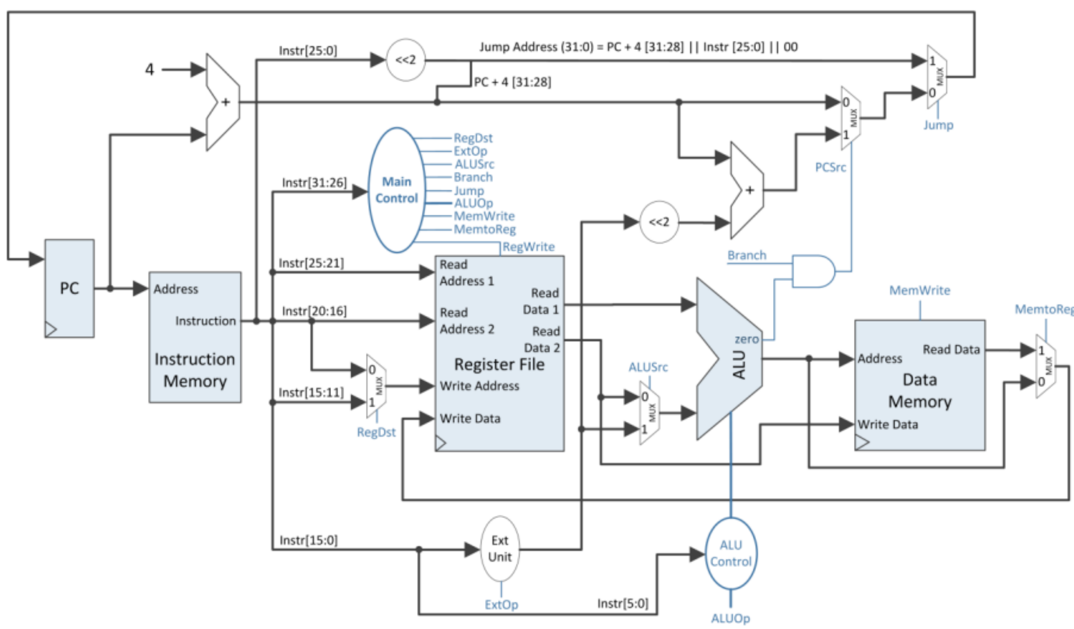

Figure 1: MIPS 32 Single-Cycle Data-Path [1].

During this stage, the core was modified to accommodate the project's specific custom instructions.

3.1.1 Program counter (pc) next-state logic expansion

The instruction fetch mechanism was modified by expanding the PC input routing into a three-stage multiplexer network to support advanced control flow. The first stage resolves conditional branches by evaluating a logical AND between the control unit's branch signal and the ALU's zero flag. The second stage resolves standard unconditional jump targets. The critical modification occurs in the third stage, which introduces a custom data path allowing the Program Counter to be loaded directly from the Data Memory output. This structural addition enables specialized instructions, such as indirect jumps or returning directly from a memory-stored address.

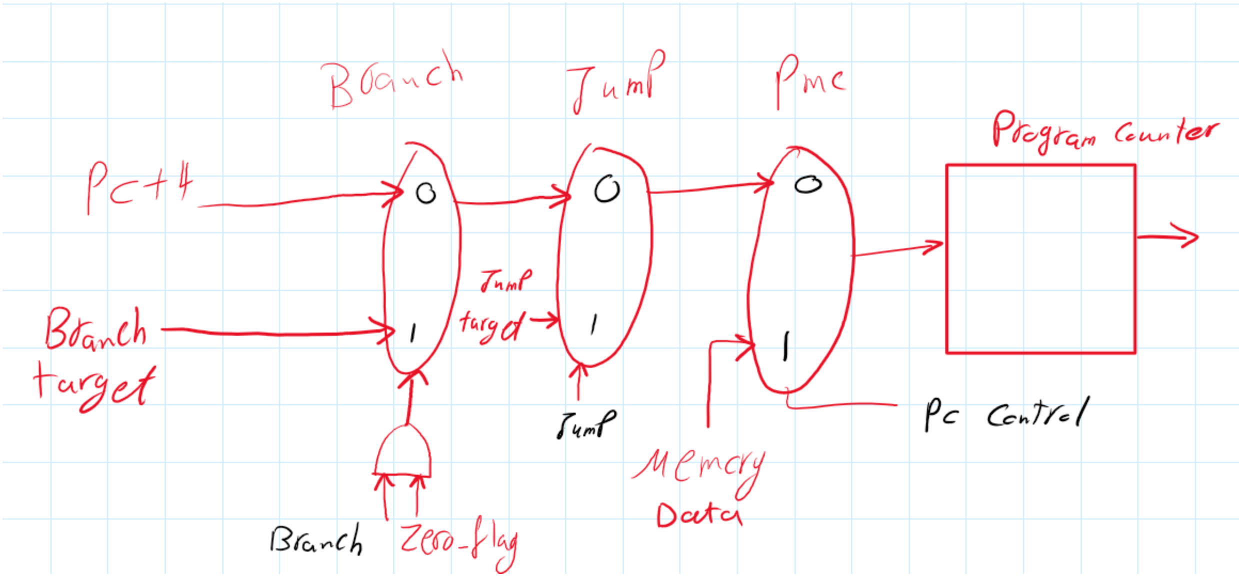

Figure 2: Added pmc MUX.

3.1.2 Decoupled data memory addressing

The standard memory interface was restructured to decouple the read and write address computations. In a traditional MIPS architecture, the ALU result strictly dictates the memory address. To support custom instructions, dedicated multiplexers were introduced at the address ports. The read address can now be dynamically sourced from either the ALU result or directly from the second register operand (Read Data 2). Concurrently, the write address can be sourced from the ALU result or the PC+4 path. This allows the architecture to execute custom operations, such as pushing the current program counter directly into memory or performing register-direct memory accesses without routing through the ALU.

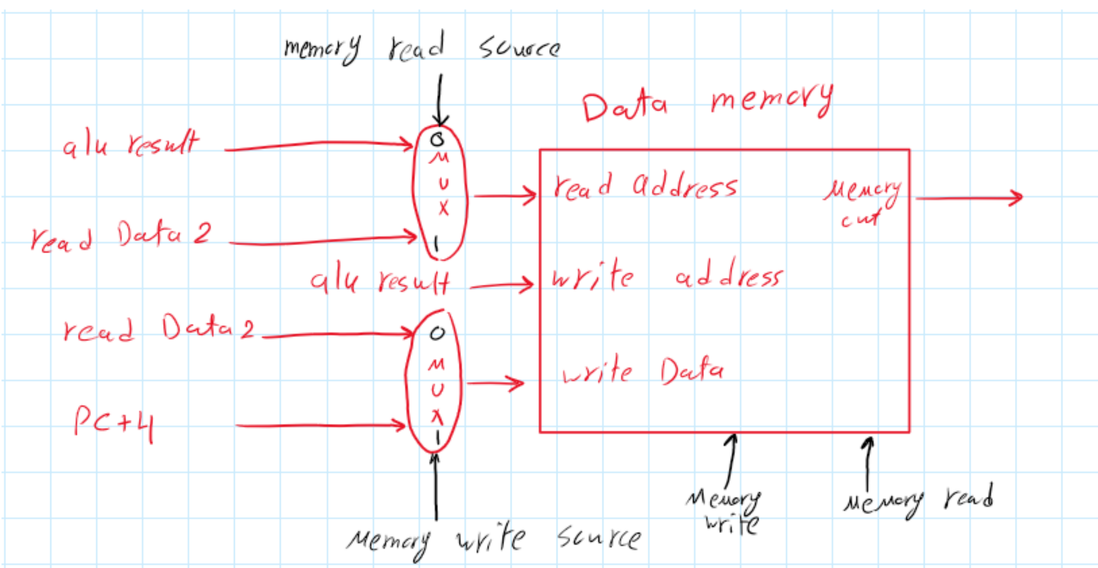

Figure 3: Dual port data memory with 2 MUXs to select the read address and the write data.

3.1.3 Extended register file destination routing

The write-back stage logic was modified by upgrading the standard 2-to-1 write destination multiplexer into a 3-to-1 multiplexer. While traditional MIPS limits the destination register to either the rt field (instruction[20:16]) for I-type or the rd field (instruction[15:11]) for R-type instructions, this architectural change introduces a third selection path. It allows the rs field (instruction[25:21]) to act as the write destination. This modification specifically accommodates custom instructions designed to compute a result and directly overwrite the first source register.

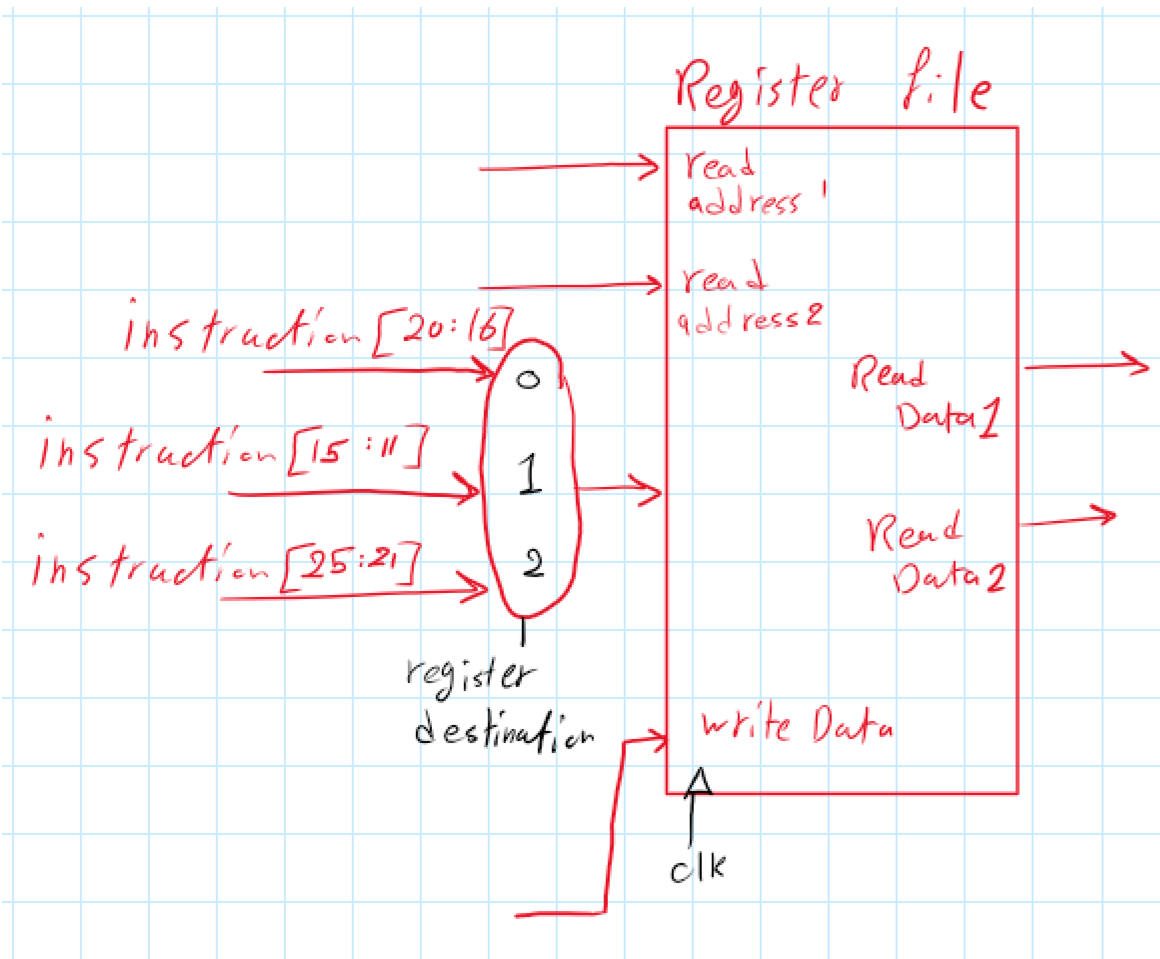

Figure 4: Expanded write address MUX.

3.2 Phase 2: Pipelined Architecture and Hazard Management

Following the validation of the single-cycle core, the architecture was partitioned and upgraded into a high-performance 5-stage pipelined design. To maintain data integrity across concurrent instruction executions, the following structural components were integrated:

- **Data Path:** Registers separate each of the five pipeline stages (IF/ID, ID/EX, EX/MEM, MEM/WB) to hold intermediate calculations and control signals.
- **Control Unit:** A hardwired main control unit decodes the 6-bit opcode in the ID stage and propagates the control signals through the pipeline registers to the EX, MEM, and WB stages.
- **Forwarding Unit:** To prevent stalling during RAW hazards, the forwarding unit monitors the destination registers of instructions in the EX/MEM and MEM/WB stages. If a dependent instruction enters the EX stage, the multiplexers bypass the Register File and forward the computed data directly to the ALU inputs.
- **Hazard Detection Unit:** If a Load-Use hazard occurs (e.g., an LW instruction followed immediately by an instruction that needs the loaded data), forwarding alone is insufficient. The Hazard Detection unit freezes the PC and IF/ID registers and flushes the ID/EX register (inserting a hardware bubble) for one clock cycle.

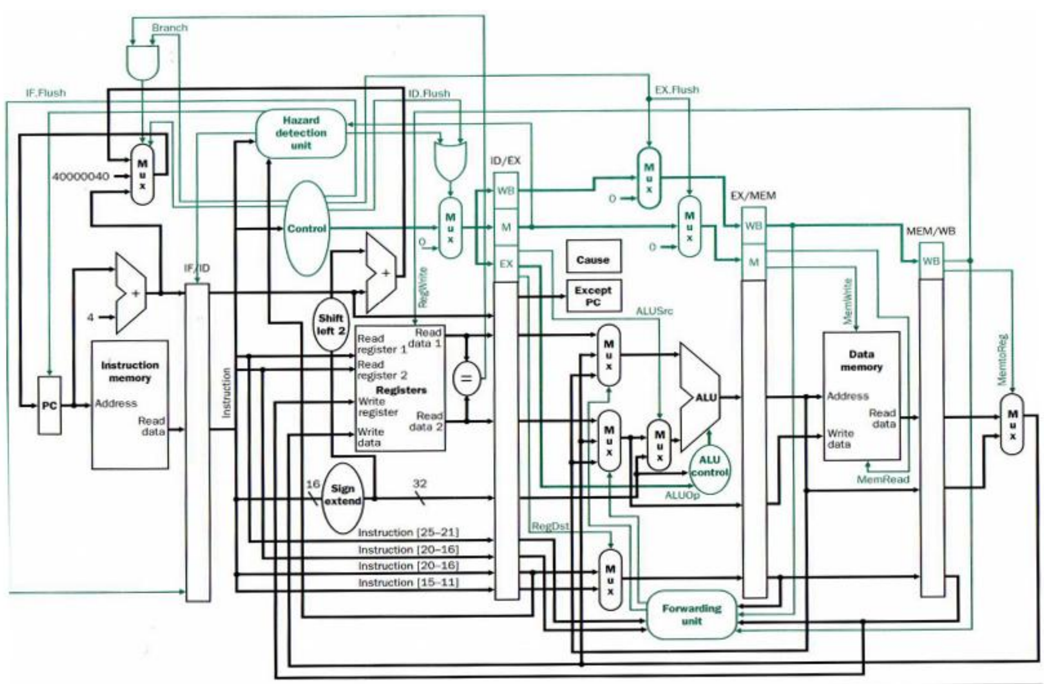

Figure 5: MIPS Pipeline CPU Architecture [2].

4.0 VERILOG CODES AND MODULE HIERARCHY

The complete, synthesizable Verilog source code, including all testbenches and constraints, is openly hosted and maintained on GitHub at:
https://github.com/MinaElHanash/MIPS

4.1 Program Counter

program_counter.v

```verilog
module program_counter (
    input [31:0] next_pc,
    input rst, clk, overflow_flag, write_enable,
    output reg [31:0] current_pc
);

    always @(posedge clk or posedge rst) // asynchronous reset, PC will reset regardless of th clk
        begin
            if (rst)
                begin
                    current_pc <= 32'd0;
                end
            else if (overflow_flag)
                begin
                    current_pc <= current_pc;
                end
            else if (write_enable) begin
                current_pc <= next_pc;
            end
            else
                begin
                    current_pc <= current_pc;
                end
        end

endmodule
```

4.2 MUX 2x1 32-Bits

mux_2to1_32bits.v

```
module mux_2to1_32bits (
    input [31:0] input_1, input_2,
    input select,
    output [31:0] mux_out
);

    assign mux_out = select?input_2:input_1;

endmodule
```

4.3 Instruction Memory

instruction_memory.v

```verilog
module instruction_memory (
    input [31:0] pc,
    output [31:0] instruction
);
    wire [31:0] pc_by_4; // as address 1 in instruction memory is equal to "pc = 4"
    // in other words, MIPS is byte aligned and each word is 4 bytes

    reg [31:0] instruction_memory_registers [0:255];

    assign pc_by_4 = pc >> 2;
    assign instruction = instruction_memory_registers [pc_by_4];

    initial // load "instructions.hex" into the instruction memory
        begin
            $readmemh("instructions.hex", instruction_memory_registers );
        end

endmodule
```

4.4 Adder

adder.v

```
module adder (
    input [31:0] input_1, input_2,
    output [31:0] adder_output
);

    assign adder_output = input_1 + input_2;

endmodule
```

4.5 IF ID Stage Register

if_id_stage.v

```
module if_id_stage (
    input clk, rst,
    input write_enable, flush,
    input [31:0] instruction_in, pc_in, pc_plus_4_in,
    output reg [31:0] instruction_out, pc_out, pc_plus_4_out
);

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            instruction_out <= 32'd0;
            pc_plus_4_out <= 32'd0;
            pc_out <= 32'd0;
        end
        else if (flush) begin
            instruction_out <= 32'd0;
            pc_plus_4_out <= 32'd0;
            pc_out <= 32'd0;
        end
        else if (write_enable) begin
            instruction_out <= instruction_in;
            pc_plus_4_out <= pc_plus_4_in;
            pc_out <= pc_in;
        end
        else begin
            instruction_out <= instruction_out;
            pc_plus_4_out <= pc_plus_4_out;
            pc_out <= pc_out;
        end
    end

endmodule
```

4.6 Register File

register_file.v

```
module register_file (
    input [4:0] read_address_1, read_address_2, write_address,
    input [31:0] write_data,
    input reg_write, clk, rst,
    output [31:0] reg_out_1, reg_out_2
);
    reg [31:0] registers [0:31]; // 32 register each with width of 32 bit
    integer i; //  used in the for loop for clearing the content ot the registers


    always @(posedge clk)
        begin
            if (rst) // reset the content of all register to zero
                begin
                    for (i = 1 ; i<32 ; i = i+1 ) begin
                        registers[i] <= 32'd0;
                    end
                end
            else if (reg_write && (write_address!= 5'd0)) // write if it is not the 0th register
                begin
                    registers[write_address] <= write_data;
                end
        end

    // hard wire the 0th register to zero
    assign reg_out_1 = (read_address_1 == 5'd0) ? 32'd0 : registers[read_address_1];
    assign reg_out_2 = (read_address_2 == 5'd0) ? 32'd0 : registers[read_address_2];

endmodule
```

4.7 Sign Extender

sign_extender.v

```
module sign_extender (
    input [15:0] in,
    output [31:0] out
);
    assign out = {{16{in[15]}}, in};
endmodule
```

4.8 Control Unit

control_unit.v

```
module control_unit (
    input [5:0] op_code,
    output reg [1:0] register_destination, alu_op,
    output reg jump, branch, memory_read, memory_write, memory_to_register, alu_source,
    output reg reg_write, pc_control, memory_write_source, memory_read_source
);
    always @(*) begin

        // initialize everything to zero
        register_destination = 2'b00;
        alu_op = 2'b00;
        jump = 1'b0;
        branch = 1'b0;
        memory_read = 1'b0;
        memory_write = 1'b0;
        memory_to_register = 1'b0;
        alu_source = 1'b0;
        reg_write = 1'b0;
        pc_control = 1'b0;
        memory_write_source = 1'b0;
        memory_read_source = 1'b0;

        case (op_code)

            6'b000000: begin // all R-type instructions
                alu_op = 2'b10;
                reg_write = 1'b1;
                register_destination = 2'b01;
            end

            6'b100011: begin // lw instruction
                alu_source = 1'b1;
                memory_read = 1'b1;
                memory_to_register = 1'b1;
                reg_write = 1'b1;
            end

            6'b101011: begin // sw instruction
                memory_write = 1'b1;
                alu_source = 1'b1;
            end

            6'b000100: begin // branch instruction
                branch = 1'b1;
                alu_op = 2'b01;
            end
```

4.8 Control Unit (Cont.)

```
            6'b001000: begin // addi instruction
                alu_source = 1'b1;
                reg_write = 1'b1;
                alu_op = 2'b00; // force add, it is 00 by default but i add it anyways to show that we treat addi as a normal add instruction
            end

            6'b001100: begin // andi instruction
                alu_source = 1'b1;
                alu_op = 2'b11;
                reg_write = 1'b1;
            end

            6'b000010: begin // jump instruction
                jump = 1'b1;
            end

            // custom instructions

            6'b110000: begin // jump mem indirect instruction
                alu_source = 1'b1;
                memory_read = 1'b1;
                alu_op = 2'b00; // force add
                pc_control = 1'b1;
            end

            6'b110001: begin // store and increment instruction
                alu_source = 1'b1;
                register_destination = 2'b10;
                reg_write = 1'b1;
                memory_write = 1'b1;
            end

            6'b110010: begin // program mem copy instruction
                alu_source = 1'b1;
                pc_control = 1'b1;
                memory_read = 1'b1;
                memory_write = 1'b1;
                memory_write_source = 1'b1;
                memory_read_source = 1'b1;
            end
        endcase
    end
endmodule
```

4.9 Hazard Detection Unit

hazard_detection_unit.v

```
module hazard_detection_unit (
    input [4:0] id_rs, id_rt, ex_rt,
    input mem_read,
    output reg hazard_flag, if_id_write_enable, pc_write_enable
);
    always @(*) begin
        hazard_flag = 1'b0;
        if_id_write_enable = 1'b1;
        pc_write_enable = 1'b1;

        if (mem_read && ( (id_rs == ex_rt) || (id_rt == ex_rt) ) ) begin
            hazard_flag = 1'b1;
            if_id_write_enable = 1'b0;
            pc_write_enable = 1'b0;
        end

    end
endmodule
```

4.10 ID EX Stage Register

id_ex_stage.v

```
module id_ex_stage (

    //*********************************************
    //inputs
    //*********************************************

    // system signals
    input clk, rst,
    input flush,

    // data signals
    input [31:0] pc_in, pc_plus_4_in, reg_file_out_1_in, reg_file_out_2_in, sign_extended_in,
    input [4:0] reg_rs_address_in, reg_rt_address_in, reg_rd_address_in,
    input [5:0] funct_in,

    // control signals
    input [1:0] register_destination_in, alu_op_in,
    input branch_in, memory_read_in, memory_write_in, memory_to_register_in,
    input alu_source_in, reg_write_in, pc_control_in, memory_write_source_in, memory_read_source_in,

    //*********************************************
    // outputs
    //*********************************************

    // data signals
    output reg [31:0] pc_out, pc_plus_4_out, reg_file_out_1_out, reg_file_out_2_out, sign_extended_out,
    output reg [4:0] reg_rs_address_out, reg_rt_address_out, reg_rd_address_out,
    output reg [5:0] funct_out,

    // control signals
    output reg [1:0] register_destination_out, alu_op_out,
    output reg branch_out, memory_read_out, memory_write_out, memory_to_register_out,
    output reg alu_source_out, reg_write_out, pc_control_out, memory_write_source_out, memory_read_source_out
);
```

4.10 ID EX Stage Register (Cont.)

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
                pc_out <= 32'd0;
                pc_plus_4_out <= 32'd0;
                reg_file_out_1_out <= 32'd0;
                reg_file_out_2_out <= 32'd0;
                sign_extended_out <= 32'd0;

                reg_rs_address_out <= 5'd0;
                reg_rt_address_out <= 5'd0;
                reg_rd_address_out <= 5'd0;

                funct_out <= 6'd0;

                register_destination_out <= 2'd0;
                alu_op_out <= 2'd0;

                branch_out <= 1'd0;
                memory_read_out <= 1'd0;
                memory_write_out <= 1'd0;
                memory_to_register_out <= 1'd0;
                alu_source_out <= 1'd0;
                reg_write_out <= 1'd0;
                pc_control_out <= 1'd0;
                memory_write_source_out <= 1'd0;
                memory_read_source_out <= 1'd0;
        end
```

4.10 ID EX Stage Register (Cont.)

```
    else if (flush) begin
            pc_out <= 32'd0;
            pc_plus_4_out <= 32'd0;
            reg_file_out_1_out <= 32'd0;
            reg_file_out_2_out <= 32'd0;
            sign_extended_out <= 32'd0;

            reg_rs_address_out <= 5'd0;
            reg_rt_address_out <= 5'd0;
            reg_rd_address_out <= 5'd0;

            funct_out <= 6'd0;

            register_destination_out <= 2'd0;
            alu_op_out <= 2'd0;

            branch_out <= 1'd0;
            memory_read_out <= 1'd0;
            memory_write_out <= 1'd0;
            memory_to_register_out <= 1'd0;
            alu_source_out <= 1'd0;
            reg_write_out <= 1'd0;
            pc_control_out <= 1'd0;
            memory_write_source_out <= 1'd0;
            memory_read_source_out <= 1'd0;
    end
```

4.10 ID EX Stage Register (Cont.)

```
            else begin
                pc_out <= pc_in;
                pc_plus_4_out <= pc_plus_4_in;
                reg_file_out_1_out <= reg_file_out_1_in;
                reg_file_out_2_out <= reg_file_out_2_in;
                sign_extended_out <= sign_extended_in;

                reg_rs_address_out <= reg_rs_address_in;
                reg_rt_address_out <= reg_rt_address_in;
                reg_rd_address_out <= reg_rd_address_in;

                funct_out <= funct_in;

                register_destination_out <= register_destination_in;
                alu_op_out <= alu_op_in;

                branch_out <= branch_in;
                memory_read_out <= memory_read_in;
                memory_write_out <= memory_write_in;
                memory_to_register_out <= memory_to_register_in;
                alu_source_out <= alu_source_in;
                reg_write_out <= reg_write_in;
                pc_control_out <= pc_control_in;
                memory_write_source_out <= memory_write_source_in;
                memory_read_source_out <= memory_read_source_in;
            end

        end

endmodule
```

4.11 ALU

alu.v

```verilog
module alu (
    input [31:0] input_1, input_2,
    input [2:0] alu_control,
    output reg [31:0] alu_result,
    output zero_flag,
    output reg overflow_flag
);
    always @(*)
        begin
            overflow_flag = 1'b0; // default to zero and change to 1 if an overflow is detected
            case (alu_control)
                3'b000: begin
                            alu_result = input_1+input_2; // add

                            if ( (input_1[31] == input_2[31]) & (alu_result[31] != input_1[31]) )
                            // if the inputs are +ve and the result is negative, then overflow, or visversa )
                                begin
                                    overflow_flag = 1'b1;
                                end
                        end
                3'b001: begin
                            alu_result = input_1-input_2; // sub

                            if ( (input_1[31] != input_2[31]) & (alu_result[31] == input_2[31]) )
                            // if +ve - -ve equal to -ve, then overflow
                            // if -ve - +ve equal to +ve, then overflow
                                begin
                                    overflow_flag = 1'b1;
                                end
                        end
                3'b010: alu_result = input_1&input_2; // and
                3'b011: alu_result = input_1|input_2; // or
                3'b100: alu_result = (input_1 < input_2)?32'd1:32'd0; // slt
                default: alu_result = input_1+input_2; // default case is add to avoid latches
            endcase
        end

    assign zero_flag = (alu_result == 32'd0) ? 1'b1 : 1'b0; // zero flag

endmodule
```

4.12 ALU Control Unit

alu_control_unit.v

```
module alu_control_unit (
    input [1:0] alu_op,
    input [5:0] funct,
    output reg [2:0] alu_control_out
);

    always @(*)
        begin
            case (alu_op)
                2'b00: alu_control_out = 3'b000; // add, force add for lw and sw instructions
                2'b01: alu_control_out = 3'b001; // sub, force sub for branch instruction
                2'b10:
                    begin
                        case (funct)
                            6'b100000: alu_control_out = 3'b000; // add
                            6'b100010: alu_control_out = 3'b001; // sub
                            6'b100100: alu_control_out = 3'b010; // and
                            6'b100101: alu_control_out = 3'b011; // or
                            6'b101010: alu_control_out = 3'b100; // slt
                            default: alu_control_out = 3'b000; // add
                        endcase
                    end
                2'b11: alu_control_out = 3'b010; //and, for andi instruction
                default: alu_control_out = 3'b000; // add
            endcase
        end

endmodule
```

4.13 MUX 4x1 32-Bits

mux_4to1_32bits.v

```
module mux_4to1_32bits (
    input [31:0] input_1, input_2, input_3, input_4,
    input [1:0] select,
    output reg [31:0] mux_out
);

    always @(*) begin
        case (select)
            2'b00: mux_out = input_1; // default alu input
            2'b01: mux_out = input_2; // destination register of ex_mem stage
            2'b10: mux_out = input_3; // destination register of mem_wb stage
            2'b11: mux_out = input_4; // 32'd0 for if the destination register was $0
            default: mux_out = input_1; // default alu input
        endcase
    end
endmodule
```

4.14 MUX 4x1 5-Bits

mux_4to1_5bits.v

```
module mux_4to1_5bits (
    input [4:0] input_1, input_2, input_3, input_4,
    input [1:0] select,
    output reg [4:0] mux_out
);

    always @(*) begin
        case (select)
            2'b00: mux_out = input_1; // for I-type
            2'b01: mux_out = input_2; // for R-type
            2'b10: mux_out = input_3; // for swi (custom instruction)
            2'b11: mux_out = input_4;
            default: mux_out = input_2; //  default will be like normal I-type
        endcase
    end
endmodule
```

4.15 Forwarding Unit

forwarding_unit.v

```
module forwarding_unit (
    input [4:0] destination_register_of_1st_previous_instruction,
    input [4:0] destination_register_of_2nd_previous_instruction,
    input [4:0] source_register_1, source_register_2,
    input reg_write_1st_instruction, reg_write_2nd_instruction,

    output reg [1:0] alu_input_1, alu_input_2
);

    always @(*) begin
        if (source_register_1 == 5'd0) begin
            alu_input_1 = 2'b11; // if source is $0 register then pass 32'd0
        end
        else if ((source_register_1 == destination_register_of_1st_previous_instruction) && reg_write_1st_instruction) begin
            alu_input_1 = 2'b01; // pass the value from ex_mem registe
        end
        else if ((source_register_1 == destination_register_of_2nd_previous_instruction) && reg_write_2nd_instruction) begin
            alu_input_1 = 2'b10; // pass the value from mem_wb registe
        end
        else alu_input_1 = 2'b00; // pass the degault value form the single cycle (read data 1)
    end

    always @(*) begin
        if (source_register_2 == 5'd0) begin
            alu_input_2 = 2'b11; // if source is $0 register then pass 32'd0
        end
        else if ((source_register_2 == destination_register_of_1st_previous_instruction) && reg_write_1st_instruction) begin
            alu_input_2 = 2'b01; // pass the value from ex_mem registe
        end
        else if ((source_register_2 == destination_register_of_2nd_previous_instruction) && reg_write_2nd_instruction) begin
            alu_input_2 = 2'b10; // pass the value from mem_wb registe
        end
        else alu_input_2 = 2'b00; // pass the degault value form the single cycle (alu_source mux output)
    end

endmodule
```

4.16 EX MEM Stage Register

id_ex_stage.v

```
module id_ex_stage (

    //*********************************************
    //inputs
    //*********************************************

    // system signals
    input clk, rst,
    input flush,

    // data signals
    input [31:0] pc_in, pc_plus_4_in, reg_file_out_1_in, reg_file_out_2_in, sign_extended_in,
    input [4:0] reg_rs_address_in, reg_rt_address_in, reg_rd_address_in,
    input [5:0] funct_in,

    // control signals
    input [1:0] register_destination_in, alu_op_in,
    input branch_in, memory_read_in, memory_write_in, memory_to_register_in,
    input alu_source_in, reg_write_in, pc_control_in, memory_write_source_in, memory_read_source_in,

    //*********************************************
    // outputs
    //*********************************************

    // data signals
    output reg [31:0] pc_out, pc_plus_4_out, reg_file_out_1_out, reg_file_out_2_out, sign_extended_out,
    output reg [4:0] reg_rs_address_out, reg_rt_address_out, reg_rd_address_out,
    output reg [5:0] funct_out,

    // control signals
    output reg [1:0] register_destination_out, alu_op_out,
    output reg branch_out, memory_read_out, memory_write_out, memory_to_register_out,
    output reg alu_source_out, reg_write_out, pc_control_out, memory_write_source_out, memory_read_source_out
);
```

4.16 EX MEM Stage Register (Cont.)

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        pc_out <= 32'd0;
        pc_plus_4_out <= 32'd0;
        reg_file_out_1_out <= 32'd0;
        reg_file_out_2_out <= 32'd0;
        sign_extended_out <= 32'd0;

        reg_rs_address_out <= 5'd0;
        reg_rt_address_out <= 5'd0;
        reg_rd_address_out <= 5'd0;

        funct_out <= 6'd0;

        register_destination_out <= 2'd0;
        alu_op_out <= 2'd0;

        branch_out <= 1'd0;
        memory_read_out <= 1'd0;
        memory_write_out <= 1'd0;
        memory_to_register_out <= 1'd0;
        alu_source_out <= 1'd0;
        reg_write_out <= 1'd0;
        pc_control_out <= 1'd0;
        memory_write_source_out <= 1'd0;
        memory_read_source_out <= 1'd0;
    end
```

4.16 EX MEM Stage Register (Cont.)

```
    else if (flush) begin
        pc_out <= 32'd0;
        pc_plus_4_out <= 32'd0;
        reg_file_out_1_out <= 32'd0;
        reg_file_out_2_out <= 32'd0;
        sign_extended_out <= 32'd0;

        reg_rs_address_out <= 5'd0;
        reg_rt_address_out <= 5'd0;
        reg_rd_address_out <= 5'd0;

        funct_out <= 6'd0;

        register_destination_out <= 2'd0;
        alu_op_out <= 2'd0;

        branch_out <= 1'd0;
        memory_read_out <= 1'd0;
        memory_write_out <= 1'd0;
        memory_to_register_out <= 1'd0;
        alu_source_out <= 1'd0;
        reg_write_out <= 1'd0;
        pc_control_out <= 1'd0;
        memory_write_source_out <= 1'd0;
        memory_read_source_out <= 1'd0;
    end
```

4.16 EX MEM Stage Register (Cont.)

```
        else begin
            pc_out <= pc_in;
            pc_plus_4_out <= pc_plus_4_in;
            reg_file_out_1_out <= reg_file_out_1_in;
            reg_file_out_2_out <= reg_file_out_2_in;
            sign_extended_out <= sign_extended_in;

            reg_rs_address_out <= reg_rs_address_in;
            reg_rt_address_out <= reg_rt_address_in;
            reg_rd_address_out <= reg_rd_address_in;

            funct_out <= funct_in;

            register_destination_out <= register_destination_in;
            alu_op_out <= alu_op_in;

            branch_out <= branch_in;
            memory_read_out <= memory_read_in;
            memory_write_out <= memory_write_in;
            memory_to_register_out <= memory_to_register_in;
            alu_source_out <= alu_source_in;
            reg_write_out <= reg_write_in;
            pc_control_out <= pc_control_in;
            memory_write_source_out <= memory_write_source_in;
            memory_read_source_out <= memory_read_source_in;
        end

    end

endmodule
```

4.17 Data Memory

data_memory.v

```verilog
module data_memory (
    input memory_read, memory_write, clk,
    input [31:0] read_address, write_address, write_data,
    output reg [31:0] output_data
);

    wire [31:0] actual_read_address, actual_write_address;

    // as each word (register) holds 4 bytes
    // in other words, MIPS is byte aligned and each word is 4 bytes, so address if the alu result is 8, then this is the 2nd regiter and not the 8th
    assign actual_read_address = read_address >> 2;
    assign actual_write_address = write_address >> 2;

    reg [31:0] data_memory_registers [0:1023];

    initial // load "data.hex" into the data memory
        begin
            $readmemh("data.hex", data_memory_registers );
        end

    always @(*) begin

        output_data = 32'd0; // default value to avoid latches

        if (memory_read) begin
            if (actual_read_address > 32'd1023) begin
                output_data = data_memory_registers[1023];
            end
            else begin
                output_data = data_memory_registers[actual_read_address];
            end
        end
    end

    always @(posedge clk) begin // wirte is synchronous to avoid potential errors, as we must make sure that the write address is ready and stable before actually write in it
        if (memory_write) begin
            if (actual_write_address > 32'd1023) begin
                data_memory_registers[1023] <= write_data;
            end
            else begin
                data_memory_registers[actual_write_address] <= write_data;
            end
        end
    end

endmodule
```

4.18 MEM WB Stage Register

mem_wb_stage.v

```
module mem_wb_stage (

    //*********************************************
    //inputs
    //*********************************************

    input clk, rst,

    input [31:0] pc_in,

    // data signals
    input [31:0] memory_data_in, alu_result_in,
    input [4:0] register_destination_in,

    // control signals
    input memory_to_register_in, reg_write_in,
    input overflow_flag_in,

    //*********************************************
    //outputs
    //*********************************************

    output reg [31:0] pc_out,

    // data signals
    output reg [31:0] memory_data_out, alu_result_out,
    output reg [4:0] register_destination_out,

    // control signals
    output reg memory_to_register_out, reg_write_out,
    output reg overflow_flag_out
);
```

4.18 MEM WB Stage Register (Cont.)

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
                pc_out <= 32'd0;

                memory_data_out <= 32'd0;
                alu_result_out <= 32'd0;

                register_destination_out <= 5'd0;

                memory_to_register_out <= 1'd0;
                reg_write_out <= 1'd0;

                overflow_flag_out <= 1'd0;
        end
        else begin
                pc_out <= pc_in;

                memory_data_out <= memory_data_in;
                alu_result_out <= alu_result_in;

                register_destination_out <= register_destination_in;

                memory_to_register_out <= memory_to_register_in;
                reg_write_out <= reg_write_in;

                overflow_flag_out <= overflow_flag_in;
        end
    end

endmodule
```

4.19 Top Module

top_module.v

```verilog
module top_module (
    input clk, rst
);

    //************************************************************
    //wires declaration
    //************************************************************

    //************************************************************
    //if stage
    //************************************************************

    wire [31:0] if_current_pc, if_instruction, if_current_pc_plus_4;

    wire [31:0] if_branch_mux_output, if_jump_mux_output, if_pmc_mux_output;

    //************************************************************
    //id stage
    //************************************************************

    wire [31:0] id_instruction, id_pc, id_pc_plus_4;

    // control signals
    wire [1:0] id_register_destination, id_alu_op;
    wire id_jump, id_branch, id_memory_read, id_memory_write, id_memory_to_register, id_alu_source;
    wire id_reg_write, id_pc_control, id_memory_write_source, id_memory_read_source;

    // register file signals
    wire [31:0] id_register_out_1, id_register_out_2;

    // sign extender signal
    wire [31:0] id_sign_extended;

    // hazard detection signals
    wire id_hazard_flage, id_pc_write_enable, if_id_write_enable;

    // jump signals
    wire [31:0] id_jump_address;
    wire [27:0] id_jump_imm_shifted_by_2;
```

4.19 Top Module (Cont.)

```
//************************************************************
//ex stage
//************************************************************

wire [31:0] ex_pc;

// MUXs inputs
wire [31:0] ex_mux_1_input_1, ex_mux_1_input_2, ex_mux_1_input_3, ex_mux_1_input_4;
wire [31:0] ex_mux_2_input_1, ex_mux_2_input_2, ex_mux_2_input_3, ex_mux_2_input_4;
wire [31:0] ex_alu_source_mux_input_1, ex_alu_source_mux_input_2;

// MUXs outputs
wire [31:0] ex_mux_1_output, ex_mux_2_output, ex_alu_source_mux_output;

// MUXs control signals
wire ex_alu_source;
wire [1:0] ex_alu_input_1_select, ex_alu_input_2_select;

// alu control unit signals
wire [1:0] ex_alu_op;
wire [2:0] ex_alu_control;
wire [5:0] ex_funct;

// register adresses
wire [4:0] ex_rt_address, ex_rd_address, ex_rs_address;
wire [4:0] ex_destination_register;

// alu signals
wire [31:0] ex_alu_result;
wire ex_zero_flag, ex_overflow_flag;

// branch signals
wire [31:0] ex_pc_plus_4, ex_imm, ex_imm_shifted_by_2;
wire [31:0] ex_branch_address;

// control signals
wire [1:0] ex_register_destination;
wire ex_branch, ex_memory_read, ex_memory_write, ex_memory_to_register;
wire ex_reg_write, ex_pc_control, ex_memory_write_source, ex_memory_read_source;
```

4.19 Top Module (Cont.)

```
//*************************************************************
//mem stage
//*************************************************************

wire [31:0] mem_pc;

// memory MUXs signals
wire [31:0] mem_alu_result, mem_register_file_output_2, mem_pc_plus_4;

// control signals
wire mem_branch_control;

// memory singals
wire [31:0] mem_memory_read_address, mem_memory_write_data;
wire [31:0] mem_memory_in_data, mem_memory_out_data;

// branch address
wire [31:0] mem_branch_address;

// control signals
wire mem_branch, mem_memory_read, mem_memory_write, mem_memory_to_register;
wire mem_reg_write, mem_pc_control, mem_memory_write_source, mem_memory_read_source;

wire [4:0] mem_destination_register;
wire mem_zero_flag, mem_overflow_flag;

//*************************************************************
//wb stage
//*************************************************************

wire [31:0] wb_pc;

// memory to register mux signals
wire [31:0] wb_memory_data, wb_alu_result, wb_register_file_data_wrtie;

// control signals
wire wb_memory_to_register;
wire wb_reg_write;
wire [4:0] wb_register_destination;

wire wb_overflow_flag;
```

4.19 Top Module (Cont.)

```
//**********************************************************
//some extra assigns
//**********************************************************

// jump address
assign id_jump_imm_shifted_by_2 = {id_instruction[25:0], 2'b00};
assign id_jump_address = {id_pc_plus_4[31:28], id_jump_imm_shifted_by_2};

// branch address
assign ex_imm_shifted_by_2 = {ex_alu_source_mux_input_2[29:0], 2'b00};

// 4th input for alu
assign ex_mux_1_input_4 = 32'd0;
assign ex_mux_2_input_4 = 32'd0;

// branch control
assign mem_branch_control = mem_branch & mem_zero_flag;

// flush signals
assign mem_stage_flush_trigger = mem_branch_control | mem_pc_control;
assign if_id_flush = id_jump | mem_stage_flush_trigger;
assign id_ex_flush = mem_stage_flush_trigger | id_hazard_flage;
```

4.19 Top Module (Cont.)

```
mux_2to1_32bits branch_mux (
    .input_1(if_current_pc_plus_4), .input_2(mem_branch_address),
    .select(mem_branch_control),
    .mux_out(if_branch_mux_output)
);

mux_2to1_32bits jump_mux (
    .input_1(if_branch_mux_output), .input_2(id_jump_address),
    .select(id_jump),
    .mux_out(if_jump_mux_output)
);

mux_2to1_32bits pmc_mux (
    .input_1(if_jump_mux_output), .input_2(mem_memory_out_data),
    .select(mem_pc_control),
    .mux_out(if_pmc_mux_output)
);

program_counter program_counter_module (
    .next_pc(if_pmc_mux_output),
    .clk(clk), .rst(rst), .overflow_flag(wb_overflow_flag), .write_enable(id_pc_write_enable),
    .current_pc(if_current_pc)
);

instruction_memory instruction_memory_module (
    .pc(if_current_pc),
    .instruction(if_instruction)
);

adder pc_adder_module (
    .input_1(32'd4), .input_2(if_current_pc),
    .adder_output(if_current_pc_plus_4)
);

if_id_stage if_id_stage_register (
    .clk(clk), .rst(rst),
    .write_enable(if_id_write_enable), .flush(if_id_flush),
    .instruction_in(if_instruction), .pc_in(if_current_pc), .pc_plus_4_in(if_current_pc_plus_4),
    .instruction_out(id_instruction), .pc_out(id_pc), .pc_plus_4_out(id_pc_plus_4)
);
```

4.19 Top Module (Cont.)

```
register_file register_file_module (
    .read_address_1(id_instruction[25:21]), .read_address_2(id_instruction[20:16]),
    .write_address(wb_register_destination), .write_data(wb_register_file_data_wrtie),
    .reg_write(wb_reg_write), .clk(clk), .rst(rst),
    .reg_out_1(id_register_out_1), .reg_out_2(id_register_out_2)
);

sign_extender sign_extender_module (
    .in(id_instruction[15:0]), .out(id_sign_extended)
);

control_unit control_unit_module (
    .op_code(id_instruction[31:26]),
    .register_destination(id_register_destination), .alu_op(id_alu_op),
    .jump(id_jump), .branch(id_branch), .memory_read(id_memory_read),
    .memory_write(id_memory_write), .memory_to_register(id_memory_to_register),
    .alu_source(id_alu_source), .reg_write(id_reg_write), .pc_control(id_pc_control),
    .memory_write_source(id_memory_write_source), .memory_read_source(id_memory_read_source)
);

hazard_detection_unit hazard_detection_unit_module (
    .id_rs(id_instruction[25:21]), .id_rt(id_instruction[20:16]), .ex_rt(ex_rt_address),
    .mem_read(ex_memory_read),
    .hazard_flag(id_hazard_flage), .if_id_write_enable(if_id_write_enable), .pc_write_enable(id_pc_write_enable)
);
```

4.19 Top Module (Cont.)

```
id_ex_stage id_ex_stage_register (
    //*********************************************
    //inputs
    //*********************************************

    // system signals
    .clk(clk), .rst(rst),
    .flush(id_ex_flush),

    // data signals
    .pc_in(id_pc), .pc_plus_4_in(id_pc_plus_4), .reg_file_out_1_in(id_register_out_1),
    .reg_file_out_2_in(id_register_out_2), .sign_extended_in(id_sign_extended),
    .reg_rs_address_in(id_instruction[25:21]), .reg_rt_address_in(id_instruction[20:16]),
    .reg_rd_address_in(id_instruction[15:11]), .funct_in(id_instruction[5:0]),

    // control signals
    .register_destination_in(id_register_destination), .alu_op_in(id_alu_op),
    .branch_in(id_branch), .memory_read_in(id_memory_read),
    .memory_write_in(id_memory_write), .memory_to_register_in(id_memory_to_register),
    .alu_source_in(id_alu_source), .reg_write_in(id_reg_write), .pc_control_in(id_pc_control),
    .memory_write_source_in(id_memory_write_source), .memory_read_source_in(id_memory_read_source),

    //*********************************************
    // outputs
    //*********************************************

    // data signals
    .pc_out(ex_pc), .pc_plus_4_out(ex_pc_plus_4), .reg_file_out_1_out(ex_mux_1_input_1),
    .reg_file_out_2_out(ex_alu_source_mux_input_1), .sign_extended_out(ex_alu_source_mux_input_2),
    .reg_rs_address_out(ex_rs_address), .reg_rt_address_out(ex_rt_address),
    .reg_rd_address_out(ex_rd_address), .funct_out(ex_funct),

    // control signals
    .register_destination_out(ex_register_destination), .alu_op_out(ex_alu_op),
    .branch_out(ex_branch), .memory_read_out(ex_memory_read),
    .memory_write_out(ex_memory_write), .memory_to_register_out(ex_memory_to_register),
    .alu_source_out(ex_alu_source), .reg_write_out(ex_reg_write), .pc_control_out(ex_pc_control),
    .memory_write_source_out(ex_memory_write_source), .memory_read_source_out(ex_memory_read_source)
);
```

4.19 Top Module (Cont.)

```
adder branch_adder_module (
    .input_1(ex_pc_plus_4), .input_2(ex_imm_shifted_by_2),
    .adder_output(ex_branch_address)
);

mux_4to1_32bits alu_input_1_mux (
    .input_1(ex_mux_1_input_1), .input_2(ex_mux_1_input_2),
    .input_3(ex_mux_1_input_3), .input_4(ex_mux_1_input_4),
    .select(ex_alu_input_1_select),
    .mux_out(ex_mux_1_output)
);

mux_2to1_32bits alu_source_mux (
    .input_1(ex_alu_source_mux_input_1), .input_2(ex_alu_source_mux_input_2),
    .select(ex_alu_source),
    .mux_out(ex_mux_2_input_1)
);

mux_4to1_32bits alu_input_2_mux (
    .input_1(ex_mux_2_input_1), .input_2(ex_mux_2_input_2),
    .input_3(ex_mux_2_input_3), .input_4(ex_mux_2_input_4),
    .select(ex_alu_input_2_select),
    .mux_out(ex_mux_2_output)
);

alu alu_module (
    .input_1(ex_mux_1_output), .input_2(ex_mux_2_output),
    .alu_control(ex_alu_control),
    .alu_result(ex_alu_result),
    .zero_flag(ex_zero_flag),
    .overflow_flag(ex_overflow_flag)
);

alu_control_unit alu_control_unit_module (
    .alu_op(ex_alu_op),
    .funct(ex_funct),
    .alu_control_out(ex_alu_control)
);

mux_4to1_5bits destination_register_mux (
    .input_1(ex_rt_address), .input_2(ex_rd_address),
    .input_3(ex_rs_address), .input_4(5'd0),
    .select(ex_register_destination),
    .mux_out(ex_destination_register)
);
```

4.19 Top Module (Cont.)

```
    forwarding_unit forwarding_unit_module (
        .destination_register_of_1st_previous_instruction(mem_destination_register),
        .destination_register_of_2nd_previous_instruction(wb_register_destination),
        .source_register_1(ex_rs_address), .source_register_2(ex_rt_address),
        .reg_write_1st_instruction(mem_reg_write), .reg_write_2nd_instruction(wb_reg_write),

        .alu_input_1(ex_alu_input_1_select), .alu_input_2(ex_alu_input_2_select)
    );

    ex_mem_stage ex_mem_stage_register (
        //*******************************************
        //inputs
        //*******************************************

        .clk(clk), .rst(rst), .flush(mem_stage_flush_trigger),
        .branch_target_in(ex_branch_address), .pc_in(ex_pc), .pc_plus_4_in(ex_pc_plus_4),
        .alu_result_in(ex_alu_result), .reg_file_out_2_in(ex_alu_source_mux_input_1),
        .register_destination_in(ex_register_destination),
        .zero_flag_in(ex_zero_flag), .overflow_flag_in(ex_overflow_flag),

        // control signals
        .branch_in(ex_branch), .memory_read_in(ex_memory_read),
        .memory_write_in(ex_memory_write), .memory_to_register_in(ex_memory_to_register),
        .reg_write_in(ex_reg_write), .pc_control_in(ex_pc_control),
        .memory_write_source_in(ex_memory_write_source), .memory_read_source_in(ex_memory_read_source),

        //*******************************************
        // outputs
        //*******************************************

        .branch_target_out(mem_branch_address), .pc_out(mem_pc), .pc_plus_4_out(mem_pc_plus_4),
        .alu_result_out(mem_alu_result), .reg_file_out_2_out(mem_register_file_output_2),
        .register_destination_out(mem_destination_register),
        .zero_flag_out(mem_zero_flag), .overflow_flag_out(mem_overflow_flag),

        // control signals
        .branch_out(mem_branch), .memory_read_out(mem_memory_read),
        .memory_write_out(mem_memory_write), .memory_to_register_out(mem_memory_to_register),
        .reg_write_out(mem_reg_write), .pc_control_out(mem_pc_control),
        .memory_write_source_out(mem_memory_write_source), .memory_read_source_out(mem_memory_read_source)
    );
```

4.19 Top Module (Cont.)

```
mux_2to1_32bits memory_read_source_mux (
    .input_1(mem_alu_result), .input_2(mem_register_file_output_2),
    .select(mem_memory_read_source),
    .mux_out(mem_memory_read_address)
);

mux_2to1_32bits memory_data_write_source_mux (
    .input_1(mem_register_file_output_2), .input_2(mem_pc_plus_4),
    .select(mem_memory_write_source),
    .mux_out(mem_memory_write_data)
);

data_memory data_memory_module (
    .memory_read(mem_memory_read), .memory_write(mem_memory_write), .clk(clk),
    .read_address(mem_memory_read_address), .write_address(mem_alu_result),
    .write_data(mem_memory_write_data),
    .output_data(mem_memory_out_data)
);
```

4.19 Top Module (Cont.)

```
    mem_wb_stage mem_wb_stage_register (
        //*********************************************
        //inputs
        //*********************************************

        .clk(clk), .rst(rst),

        .pc_in(mem_pc),

        // data signals
        .memory_data_in(mem_memory_out_data), .alu_result_in(mem_alu_result),
        .register_destination_in(mem_destination_register),

        // control signals
        .memory_to_register_in(mem_memory_to_register), .reg_write_in(mem_reg_write),
        .overflow_flag_in(mem_overflow_flag),

        //*********************************************
        //outputs
        //*********************************************

        .pc_out(wb_pc),

        // data signals
        .memory_data_out(wb_memory_data), .alu_result_out(wb_alu_result),
        .register_destination_out(wb_register_destination),

        // control signals
        .memory_to_register_out(wb_memory_to_register), .reg_write_out(wb_reg_write),
        .overflow_flag_out(wb_overflow_flag)
    );

    mux_2to1_32bits data_to_register_mux (
        .input_1(wb_alu_result), .input_2(wb_memory_data),
        .select(wb_memory_to_register),
        .mux_out(wb_register_file_data_wrtie)
    );
```

4.19 Top Module (Cont.)

```
    // overflow monitoring block
    always @(posedge clk) begin
        if (wb_overflow_flag) begin
            $display("--------------------------------------------------------------");
            $display("CRITICAL ERROR: Arithmetic Overflow Detected!");
            $display("Processor Halted at Faulting PC = %h", wb_pc);
            $display("--------------------------------------------------------------");
            $stop;
        end
    end

endmodule
```

5.0 SYNTHESIS AND IMPLEMENTATION

The RTL design was synthesized and implemented using Xilinx Vivado 2023. The target hardware was the **Xilinx Spartan-7 FPGA (xc7s50csga324-1)**. To accurately benchmark the total gate-level logic utilized by the architecture, a compiler directive (* dont_touch = "true" *) was applied to all submodules. This forced the synthesis engine to map the Data Memory directly to logic slices (Flip-Flops) rather than dedicated Block RAM (BRAM).

Implementation Results:

- **Slice LUTs (Look-Up Tables):** 11,333
- **Slice Registers (Flip-Flops):** 34,366
- **Worst Negative Slack (WNS):** +0.540 ns
- **Target Clock Constraint:** 10.000 ns (100 MHz)
- **Calculated Maximum Frequency (F_{max}):** 105.7 MHz

The timing report confirms that all user-specified timing constraints were met with zero setup or hold violations.

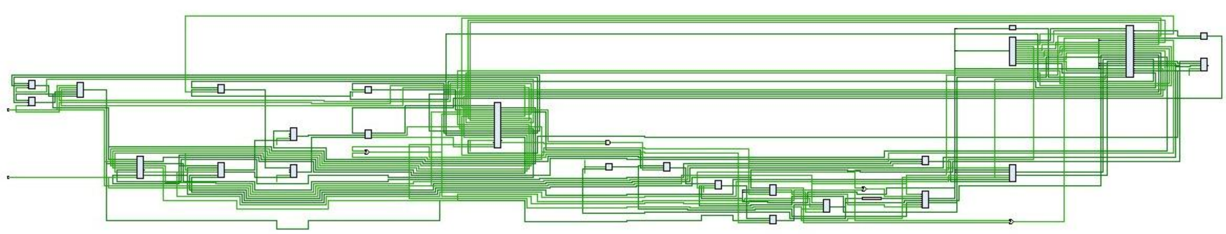

Figure 6: Elaboration schematic (post-elaboration netlist view).

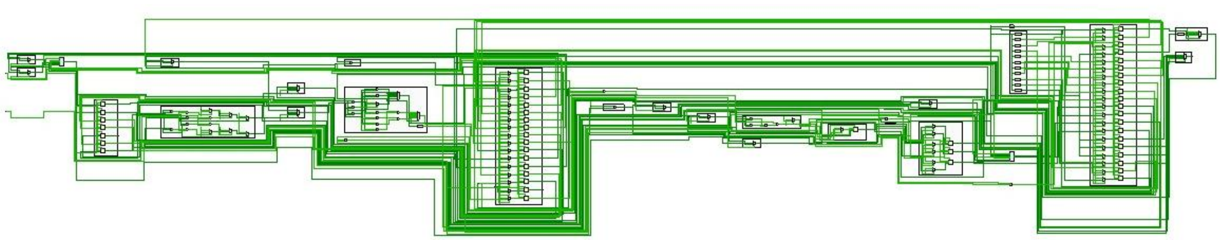

Figure 7: Elaboration schematic (Expanded View).

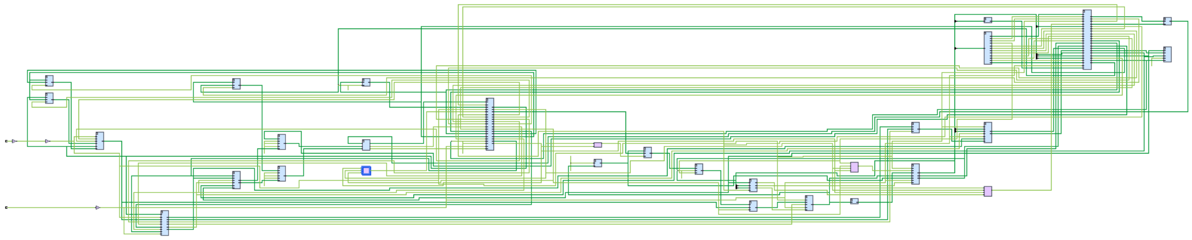

Figure 8: Synthesis schematic.

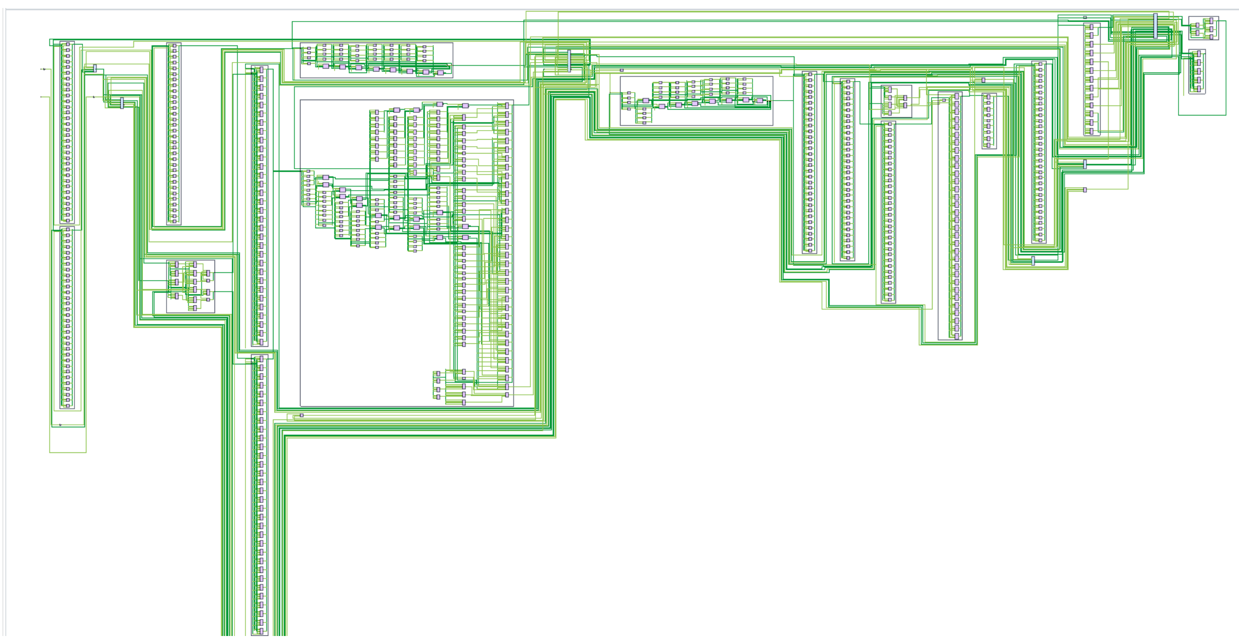

Figure 9: Synthsis schematic (Expanded).

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Bonded IOB	BUFGCTRL
N top_module	11333	34366	4608	2176	2	1
alu_module (alu)	153	0	0	0	0	0
data_memory_module (data_memory)	10046	32768	4352	2176	0	0
ex_mem_stage_register (ex_mem_stage)	90	180	0	0	0	0
id_ex_stage_register (id_ex_stage)	97	194	0	0	0	0
if_id_stage_register (if_id_stage)	49	96	0	0	0	0
mem_wb_stage_register (mem_wb_stage)	0	104	0	0	0	0
register_file_module (register_file)	607	992	256	0	0	0

Figure 10: Synthesis resource utilization (LUTs, FFs, BRAM, DSP slices).

Design Timing Summary

Setup

Worst Negative Slack (WNS)	0.540 ns
Total Negative Slack (TNS)	0.000 ns
Number of Failing Endpoints	0
Total Number of Endpoints	68227

Hold

Worst Hold Slack (WHS)	0.154 ns
Total Hold Slack (THS)	0.000 ns
Number of Failing Endpoints	0
Total Number of Endpoints	68227

Pulse Width

Worst Pulse Width Slack (WPWS)	4.500 ns
Total Pulse Width Negative Slack (TPWS)	0.000 ns
Number of Failing Endpoints	0
Total Number of Endpoints	34367

All user specified timing constraints are met.

Figure 11: Synthesis timing report excerpt (critical path and Fmax)

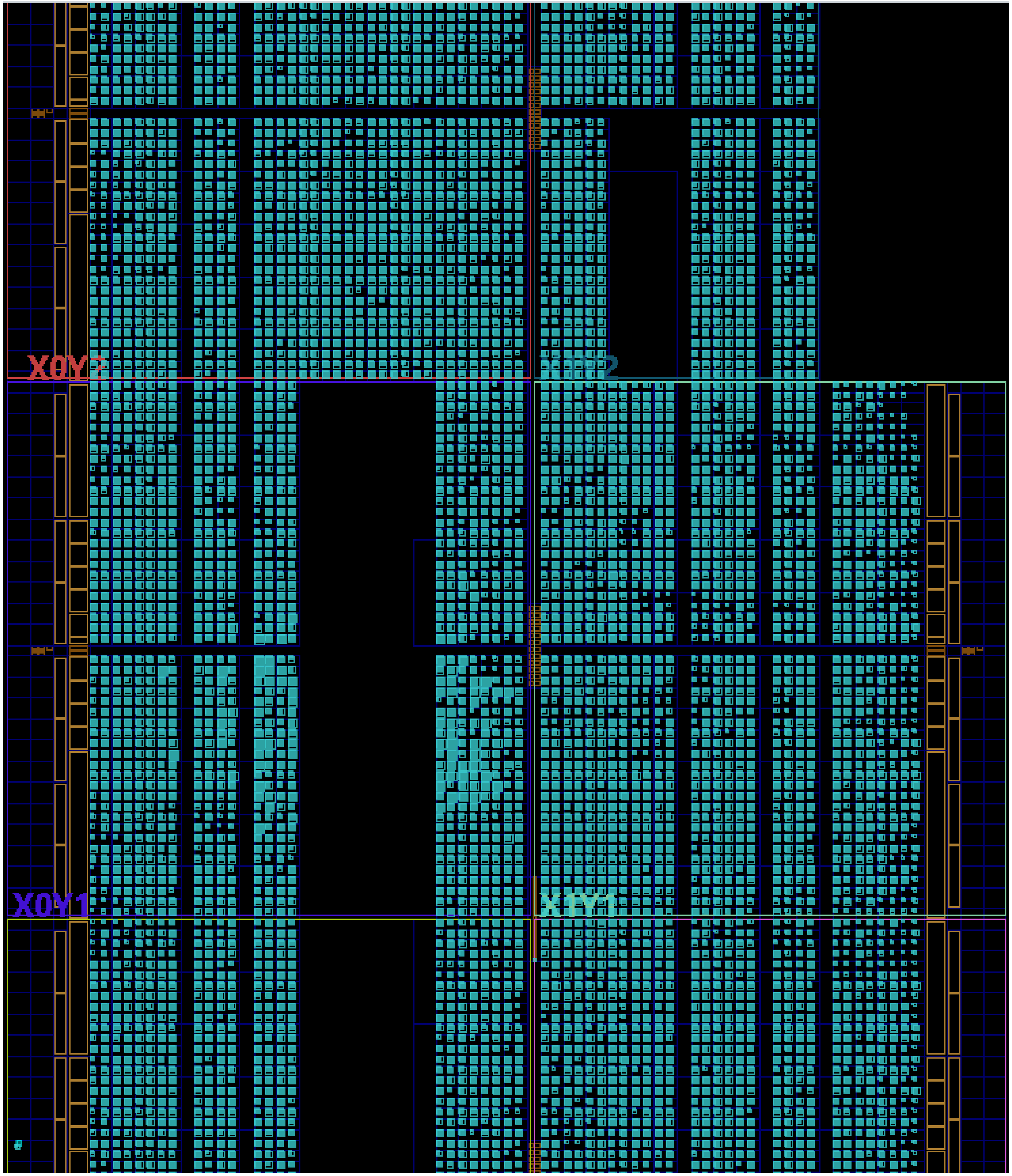

Figure 12: Device view on the target FPGA.

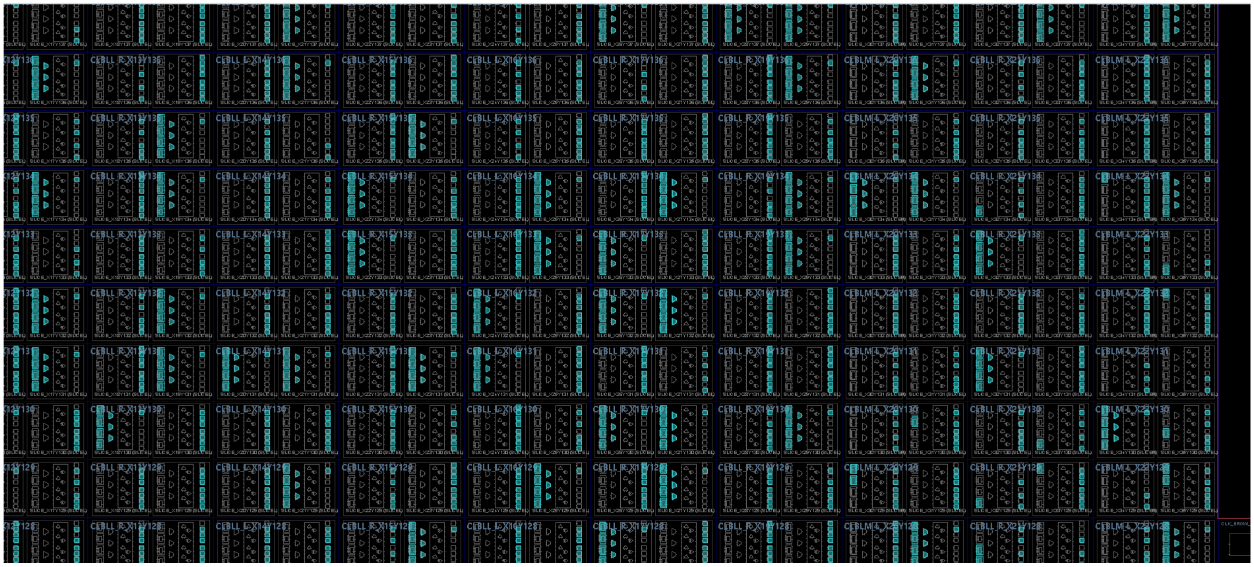

Figure 13: Zoomed in Device View.

6.0 CONCLUSION

Phase 1 of the MIPS processor project was highly successful. The RTL architecture correctly implements a 5-stage pipeline with robust hardware-level hazard resolution. Synthesizing the design proved that the Verilog code is not just simulation-ready, but physically realizable, achieving a stable clock speed of over 100 MHz on a modern Spartan-7 architecture. With the RTL hardware fully implemented and benchmarked, the project is now prepared for Phase 2: rigorous functional verification utilizing a SystemVerilog UVM testbench.

7.0 References

[1] Technical University of Cluj-Napoca, "Lab 10," [Online]. Available: https://mihai.utcluj.ro/wp-content/uploads/ca/labs/Lab10.pdf. (Accessed: Feb. 22, 2026).

[2] Stack Overflow, "MIPS pipeline CPU architecture," [Online]. Available: https://stackoverflow.com/questions/44331059/mips-pipeline-cpu-architecture. (Accessed: Feb. 22, 2026).

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2013.

[4] Xilinx, Inc., *Vivado Design Suite User Guide: Synthesis*, UG901, San Jose, CA, USA.

[5] Xilinx, Inc., *Vivado Design Suite User Guide: Using Constraints*, UG903, San Jose, CA, USA.