Indian Institute of Information Technology
Design and Manufacturing, Kurnool

Single Cycle MIPS: RTL to GDSII

VLSI System Design (EC-307)

Faculty: **Dr. P. Ranga Babu**

Submitted by:

V. Harshitha (123EC0019)

V. Sathwika (123EC0033)

Department of Electronics and Communication Engineering

Abstract

The development of Integrated Circuits (ICs) is fundamentally a hierarchical process of information-theoretic transformation, moving from a behavioral description (RTL) to a precise geometric representation (GDSII). The initial RTL stage formalizes the circuit's operation using synchronous state machines and combinatorial Boolean algebra, defining system behavior without reference to specific transistors. The process transitions through **Logic Synthesis**, where the abstract RTL model is formally mapped onto a technology-dependent library of standard cells, solving a constrained optimization problem to minimize area and power under critical timing (clock frequency) criteria.The subsequent **Physical Design** phase is a complex topological optimization problem, involving Floorplanning and *Placement* to determine the optimal coordinates for millions of cells, followed by *Clock Tree Synthesis (CTS)* to minimize temporal skew, and finally Routing to establish inter-cell connectivity based on graph theory and lithographic design rules (DRC). The culmination is **Physical Verification**, which employs parasitic extraction models to correlate the final physical layout with the initial logical netlist (LVS).The **GDSII file** represents the ultimate physical commitment, encoding the verified layout as a set of mask geometries necessary for subsequent fabrication through photolithography. This entire flow constitutes a rigorous, constraint-driven reduction of functional requirements into manufacturable, micro-scale physical reality.

Introduction

Modern computer systems are built upon the foundation of processor architectures that efficiently execute instructions. One of the most influential and educationally significant architectures is the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture, which follows the RISC (Reduced Instruction Set Computer) design philosophy. RISC architectures simplify the instruction set, allowing faster instruction execution and easier hardware implementation compared to CISC architectures. The MIPS processor exemplifies this simplicity and regularity, making it a widely used model for teaching computer architecture and for research in processor design.A Single-Cycle MIPS Processor represents the most fundamental implementation of this architecture. In this design, each instruction regardless of its type or complexity completes all its stages (fetch, decode, execute, memory access, and write-back) within a single clock cycle. Although this limits its practical performance due to timing constraints, it provides a clear and complete understanding of the internal working of a CPU and forms the conceptual basis for more advanced processors like multi-cycle and pipelined designs.

Theory and Working

MIPS Instruction Set Architecture

The **MIPS instruction set architecture (ISA)** is a load/store architecture that uses register-based operations and a fixed instruction length of 32 bits. All computations are performed on registers, while memory is accessed only through explicit `load (lw)` and `store (sw)` instructions.

The instruction format is divided into three types:

- **R-type (Register type):** Used for arithmetic and logical operations involving only registers.
- **I-type (Immediate type):** Used for operations involving constants or memory accesses.
- **J-type (Jump type):** Used for control flow changes such as jumps.

Each instruction in the MIPS ISA specifies the operation code (*opcode*) and other fields such as source and destination registers, shift amounts, function codes, and immediate values. These fields are decoded by the control unit to determine how data flows through the processor.

Datapath

The datapath is the central structure through which data flows and is processed in the CPU. In a **single-cycle MIPS processor**, all components are interconnected so that an instruction passes through every required stage in one complete clock cycle. This design ensures that the control unit, datapath elements, and memory operate synchronously to complete one instruction before the next one begins.

Functional Stages of the Datapath

The five major functional stages of the datapath are as follows:

1. **Instruction Fetch (IF):** The Program Counter (PC) supplies the address of the instruction to the Instruction Memory, which retrieves the corresponding 32-bit instruction. After fetching, the PC is updated (typically `PC + 4`) for sequential instructions.

2. **Instruction Decode (ID):** The instruction is decoded into its constituent fields (opcode, function, registers, immediate). The Control Unit interprets the opcode and sets control signals accordingly. The Register File reads the source registers based on instruction fields.

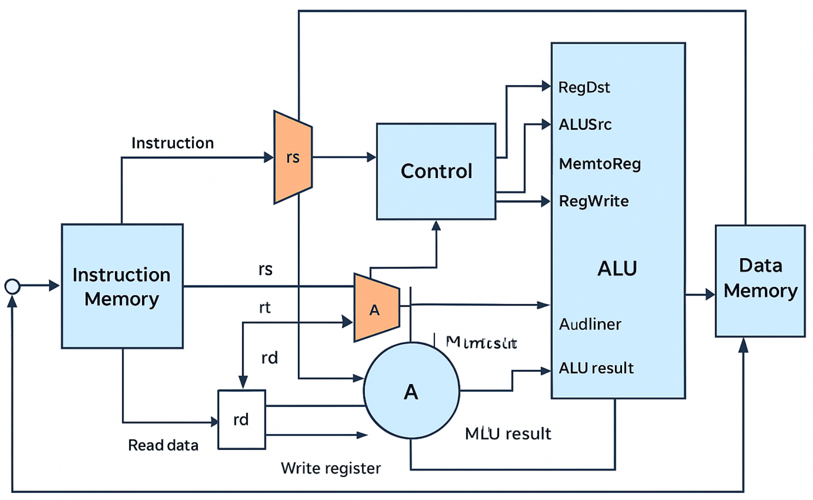
3. **Execution (EX):** The Arithmetic Logic Unit (ALU) performs the required computation — such as addition, subtraction, comparison, or logical operations. For memory-related instructions, the ALU computes the effective memory address.

4. **Memory Access (MEM):** For `load (lw)` instructions, the Data Memory is read at the address computed by the ALU. For `store (sw)` instructions, data from the Register File is written to memory.

5. **Write-Back (WB):** The result (either from the ALU or Data Memory) is written back into the destination register, completing the instruction cycle.

All these stages occur simultaneously within one clock cycle, controlled by precise timing and signal coordination.

Datapath Flow



Datapath Components

The major hardware components of the single-cycle MIPS processor include:

- **Program Counter (PC):** The Program Counter (PC) is a 32-bit register that holds the address of the instruction that is currently being executed. In every clock cycle, the processor fetches the instruction located at this address from the Instruction Memory.After fetching the instruction, the PC is updated so it points to the next instruction. Normally, the next instruction address is: $PC_{\text{next}} = PC + 4$ However, for **branch** or **jump** instructions, the Program Counter (PC) is updated differently:

For branch instructions (such as `beq` or `bne`), if the branch condition is true:

$$PC_{\text{next}} = PC + 4 + (\text{offset} \times 4)$$

Otherwise, it is simply:

$$PC_{\text{next}} = PC + 4$$

For jump instructions:

$$PC_{\text{next}} = \text{target\_address}$$

generally this how we represent it but we did't include it in our project for simplicity.

- **Instruction Memory:** A read-only memory block that stores program instructions. Each instruction is 32 bits wide.

Field	Bit Range	Meaning
opcode	[31–26]	Specifies operation type
rs	[25–21]	Source register 1
rt	[20–16]	Source register 2 or destination (I-type)
rd	[15–11]	Destination register (R-type)
shamt	[10–6]	Shift amount
funct	[5–0]	Specifies exact ALU operation
immediate / address	varies	Used for memory or branch instructions

Table 1: MIPS Instruction Format Fields

- **Register File:**

  The register file is a fundamental component of a CPU's datapath. It acts as a small, fast memory unit located very close to the processor's arithmetic and logic units (ALUs), used for storing operands and intermediate results during instruction execution.
  It's Structure is
  The register file in this design contains 32 general-purpose registers, each 32 bits wide.

  These registers are typically labeled `R0--R31`, and they can hold integer values, addresses, or other data needed by the processor.

  Each register can be accessed using its unique index (e.g., the instruction specifies which registers to read or write).

  Ports and Parallel Access

  The register file is designed with multiple ports to allow simultaneous data access, improving instruction throughput and reducing execution time.
  Two Read Ports: These allow the processor to read two operands at the same time. For example, an instruction like `ADD R3, R1, R2` needs to read the values of `R1` and `R2` simultaneously before computing the result. The two read ports make this possible in a single clock cycle.

  One Write Port: This allows the result of an operation to be written back into one register each cycle. Continuing the same example, after the ALU computes `R1 + R2`, the result is written to `R3` through the write port.

Table 2: Common R-type Instructions and their funct codes

Instruction	funct (binary)	funct (hex)	Operation Description
add	`100000`	`0x20`	Adds two registers (signed addition)
sub	`100010`	`0x22`	Subtracts one register from another (signed subtraction)
and	`100100`	`0x24`	Bitwise AND of two registers

- **ALU (Arithmetic Logic Unit):** Executes arithmetic (addition, subtraction) and logical (AND, OR, XOR, shift) operations. The specific operation is determined by the ALU control input.In this code for simplicity we took only 3 operations and,sub,sum. we didn't use any other instructions apart frpm R-type
- **Data Memory:** Used for read and write operations during `load (lw)` and `store (sw)` instructions.

Operation	Purpose	Description
Load Word (LW)	Read data	The processor loads a 32-bit word from memory into a register.
Store Word (SW)	Write data	The processor stores a 32-bit word from a register into memory.

Table 3: Memory Operations in MIPS

- **Control Unit:** Generates all necessary control signals to coordinate the operation of datapath elements based on the instruction opcode and function code.

Control Unit Signals Description

The Control Unit generates several control signals to coordinate datapath operations:

- **RegDst:** Determines which register will be written to in the Register File. When `RegDst = 0`, the destination register is `rt` (for I-type instructions like `lw`); when `RegDst = 1`, it is `rd` (for R-type instructions such as `add`).
- **ALUSrc:** Selects the second input to the ALU. If `ALUSrc = 0`, the second operand comes from the register file (`rt`). If `ALUSrc = 1`, the ALU uses the sign-extended immediate value (for instructions like `lw`, `sw`, or `addi`).
- **MemtoReg:** Controls the data written back to the register file. When `MemtoReg = 0`, the ALU result is written; when `MemtoReg = 1`, data read from memory is written (used in `lw`).
- **RegWrite:** Enables writing to the Register File. It is asserted (1) for instructions that modify registers (`add`, `sub`, `lw`, `addi`), and deasserted (0) for others like `sw` or `beq`.
- **MemRead:** Enables reading from Data Memory. It is active only for `lw` instructions.
- **MemWrite:** Enables writing to Data Memory. It is active for `sw` instructions, causing data from the register to be stored in memory.

- **Branch:** Activates branch comparison logic for instructions like `beq`. When the zero flag from the ALU is set and `Branch = 1`, the PC is updated to the branch target address.
- **ALUControl[1:0]:** Determines the specific ALU operation based on the instruction type (e.g., `00 → ADD`, `01 → SUB`, `10 → AND`, etc.) using both the opcode and function field.

Instruction Control Signal Behavior

Table 4: Control signal behavior for different MIPS instruction types (Single-Cycle Datapath).

Instruction Type	**Opcode**	**Control Behavior**
R-type (ADD, SUB, AND)	`000000`	RegDst = 1, ALUSrc = 0, RegWrite = 1, and ALUControl is determined by the function field (`funct` bits).
LW (Load Word)	`100011`	Uses ALU to compute memory address. Control signals: RegDst = 0, ALUSrc = 1, MemtoReg = 1, RegWrite = 1, and MemRead = 1.
SW (Store Word)	`101011`	Uses ALU to compute memory address. Control signals: ALUSrc = 1, MemWrite = 1. (Note: RegWrite and MemtoReg are 'don't care' or 0).
BEQ (Branch Equal)	`000100`	Performs subtraction in ALU for comparison. Control signals: Branch = 1, ALUOp = 01 (for Subtract), and ALUControl is set accordingly.
ADDI (Add Immediate)	`001000`	Executes addition with immediate value. Control signals: RegDst = 0, ALUSrc = 1, RegWrite = 1, ALUOp = 00 (for Add), and ALUControl is set accordingly.
Default / NOP	—	All control signals are typically set to 0 (no operation, no writes, no memory access).

Design Implementation in Verilog

The MIPS processor design was implemented using Verilog HDL on the Vivado simulation environment. The implementation follows a modular approach, dividing the architecture into functional blocks such as the ALU, Control Unit, Register File, Instruction Memory, and Data Memory. Each module was designed, verified, and then integrated to form the complete datapath and control flow system. The ALU performs arithmetic and logical operations, while the Control Unit generates appropriate control signals based on the opcode and function code. Instruction and data memories are modeled as register arrays for simulation of load and store operations. All registers and signals are initialized to known values to ensure predictable system behavior. The processor executes basic R-type, I-type, and branch instructions, verifying correct data flow and signal timing. Design verification was carried out using waveform analysis and testbench-based validation in Vivado. Proper use of non-blocking assignments and synchronous design principles ensures reliable operation. The overall implementation demonstrates a functional single-cycle MIPS processor architecture in Verilog.

Verilog Source Code

Listing 1: source code for Single-Cycle MIPS CPU

```verilog
// alu.v
`timescale 1ns/1ps

module alu (
    input  [31:0] a,
    input  [31:0] b,
    input  [1:0]  alu_control, // 00 ADD, 01 AND, 11 SUB
    output reg [31:0] result,
    output zero
);
    always @(*) begin
        case (alu_control)
            2'b01: result = a & b;     // AND
            2'b11: result = a - b;     // SUB
            default: result = a + b;   // ADD
        endcase
    end
    assign zero = (result == 32'b0);
endmodule

//register_file.v
`timescale 1ns/1ps
module register_file (
    input clk,
    input reg_write,
    input [4:0] rs, rt, rd,
    input [31:0] write_data,
    output reg [31:0] rs_data, rt_data
);
    reg [31:0] registers [0:31];
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1)
            registers[i] = 32'h0;
        registers[1] = 32'h0000000A;
        registers[2] = 32'h00000003;
        registers[5] = 32'h0000000F;
    end
    always @(*) begin
        rs_data = registers[rs];
        rt_data = registers[rt];
    end
    always @(posedge clk) begin
        if (reg_write && rd != 0)
            registers[rd] <= write_data;
    end
endmodule

//instruction_memory.v
`timescale 1ns/1ps
module instruction_memory (
    input  [31:0] addr,
    output [31:0] instr_out
);
    reg [31:0] instr_mem [0:255];
    integer i;
    initial begin
        instr_mem[0] = 32'h20010005;
        instr_mem[1] = 32'h20020007;
        instr_mem[2] = 32'h00221820;
        instr_mem[3] = 32'h00622022;
        instr_mem[4] = 32'h00822824;
        instr_mem[5] = 32'hAC030000;
        instr_mem[6] = 32'h8C060000;
        instr_mem[7] = 32'h10C30001;
        instr_mem[8] = 32'h20070099;
        instr_mem[9] = 32'h200800AA;
        for (i = 10; i < 256; i = i + 1)
            instr_mem[i] = 32'h00000000;
    end
    assign instr_out = instr_mem[addr[9:2]];
endmodule

//data_memory.v
`timescale 1ns/1ps
module data_memory (
    input clk,
    input mem_write,
    input mem_read,
    input [31:0] addr,
    input [31:0] write_data,
    output reg [31:0] read_data
);
    reg [31:0] mem [0:255];
    integer i;
    initial begin
        for (i = 0; i < 256; i = i + 1)
            mem[i] = 32'hDEADBEEF + i;
        read_data = 32'b0;
    end
    always @(posedge clk) begin
        if (mem_write)
            mem[addr[9:2]] <= write_data;
        if (mem_read)
            read_data <= mem[addr[9:2]];
    end
endmodule

//control_unit.v
`timescale 1ns/1ps
module control_unit (
    input  [5:0] opcode,
    input  [5:0] funct,
    output reg   RegDst,
    output reg   ALUSrc,
    output reg   MemtoReg,
    output reg   RegWrite,
    output reg   MemRead,
    output reg   MemWrite,
    output reg   Branch,
    output reg [1:0] ALUControl
);
    always @(*) begin
        RegDst=0;ALUSrc=0;MemtoReg=0;RegWrite=0;MemRead=0;MemWrite
           =0;Branch=0;
        ALUControl=2'b00;
        case (opcode)
            6'b000000: begin
                RegDst=1;RegWrite=1;ALUSrc=0;
                case (funct)
                    6'b100000: ALUControl=2'b00;
                    6'b100010: ALUControl=2'b11;
                    6'b100100: ALUControl=2'b01;
                endcase
            end
            6'b100011: begin
                RegDst=0;ALUSrc=1;MemtoReg=1;RegWrite=1;MemRead=1;
                   ALUControl=2'b00;
            end
            6'b101011: begin
                ALUSrc=1;MemWrite=1;ALUControl=2'b00;
            end
            6'b000100: begin
                Branch=1;ALUControl=2'b11;
            end
            6'b001000: begin
                RegDst=0;ALUSrc=1;RegWrite=1;ALUControl=2'b00;
            end
        endcase
    end
```

```
endmodule

// program_counter.v
`timescale 1ns/1ps
module program_counter (
    input clk, input reset,
    input [31:0] pc_in,
    output reg [31:0] pc_out
);
    always @(posedge clk or posedge reset)
        if (reset) pc_out <= 0;
        else pc_out <= pc_in;
endmodule

//mips_cpu.v (top-level)
`timescale 1ns/1ps
module mips_cpu (
    input clk, input reset,
    output [31:0] pc, instr, alu_result, mem_read_data
);
    wire [31:0] pc_next, pc_plus4;
    wire [5:0] opcode = instr[31:26];
    wire [4:0] rs = instr[25:21], rt = instr[20:16], rd = instr
       [15:11];
    wire [15:0] imm = instr[15:0];
    wire [5:0] funct = instr[5:0];
    wire [31:0] rs_data, rt_data, alu_b;
    wire alu_zero;
    wire [1:0] alu_control;
    wire RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite,
       Branch;
    wire [31:0] sign_ext_imm = {{16{imm[15]}}, imm};
    wire [31:0] imm_shifted = {sign_ext_imm[29:0], 2'b00};
    wire [31:0] branch_target = pc_plus4 + imm_shifted;
    wire [4:0] write_reg = (RegDst) ? rd : rt;
    wire [31:0] write_back_data = (MemtoReg) ? mem_read_data :
       alu_result;

    program_counter pc_reg (.clk(clk), .reset(reset), .pc_in(pc_next
       ), .pc_out(pc));
    instruction_memory imem (.addr(pc), .instr_out(instr));
    assign pc_plus4 = pc + 4;
    control_unit ctrl (.opcode(opcode), .funct(funct), .RegDst(
       RegDst),
                       .ALUSrc(ALUSrc), .MemtoReg(MemtoReg), .
                          RegWrite(RegWrite),
                       .MemRead(MemRead), .MemWrite(MemWrite), .
                          Branch(Branch),
                       .ALUControl(alu_control));
    register_file regs (.clk(clk), .reg_write(RegWrite), .rs(rs), .
       rt(rt),
                        .rd(write_reg), .write_data(write_back_data)
                            ,
                        .rs_data(rs_data), .rt_data(rt_data));
    assign alu_b = (ALUSrc) ? sign_ext_imm : rt_data;
    alu the_alu (.a(rs_data), .b(alu_b), .alu_control(alu_control),
                 .result(alu_result), .zero(alu_zero));
    data_memory dmem (.clk(clk), .mem_write(MemWrite), .mem_read(
       MemRead),
                      .addr(alu_result), .write_data(rt_data),
                      .read_data(mem_read_data));
    wire take_branch = Branch & alu_zero;
    assign pc_next = take_branch ? branch_target : pc_plus4;
endmodule
```

Verilog Testbench

Listing 2: Testbench for Single-Cycle MIPS CPU

```
‘timescale 1ns/1ps

module mips_cpu_tb;

    // Testbench signals
    reg clk;
    reg reset;
    wire [31:0] pc, instr, alu_result, mem_read_data;

    // Instantiate the DUT
    mips_cpu uut (
        .clk(clk),
        .reset(reset),
        .pc(pc),
        .instr(instr),
        .alu_result(alu_result),
        .mem_read_data(mem_read_data)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ˜clk;
    end

    // Simulation control
    initial begin
        $dumpfile("mips_cpu_tb.vcd");
        $dumpvars(0, mips_cpu_tb);
        reset = 1; #15; reset = 0;
        $display(">> CPU running...");
        #600;
```

```verilog
        $display("Simulation finished.");
        show_registers;
        show_data_memory;
        $finish;
    end

    // Task: display registers
    task show_registers;
        integer i;
        begin
            for (i=0; i<10; i=i+1)
                $display("R[%0d] = %h", i, uut.regs.registers[i]);
        end
    endtask

    // Task: display memory
    task show_data_memory;
        integer j;
        begin
            for (j=0; j<10; j=j+1)
                $display("MEM[%0d] = %h", j, uut.dmem.mem[j]);
        end
    endtask

endmodule
```

Simulation Results

The simulation waveform shows the sequential execution of MIPS instructions stored in the instruction memory. At every positive clock edge (clk), the Program Counter (PC) increments by 4, fetching the next instruction from memory.

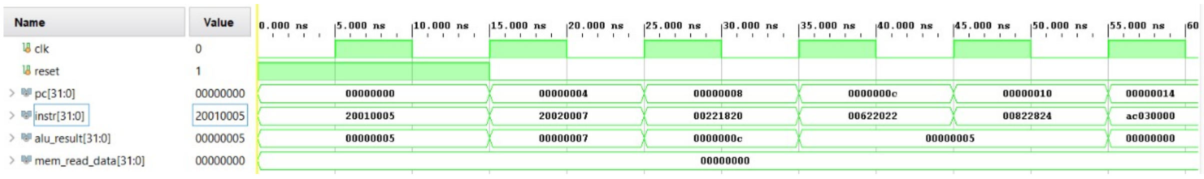


Figure 1: wave form

Simulation Output Explanation

Here's your LaTeX version **without any extra gaps** (compact format):
“‘latex

Simulation Output Explanation

- At 0 ns – Reset Active: The reset signal is 1, so the PC is initialized to 00000000 and all outputs are cleared.

- At 5 ns – First Instruction (20010005): Instruction `addi $1, $0, 5` is fetched. The ALU performs the addition operation, storing value 5 in register `$1`.
- At 15 ns – Second Instruction (20020007): Instruction `addi $2, $0, 7` loads immediate value 7 into register `$2`.
- At 25 ns – Third Instruction (00221820): Instruction `add $3, $1, $2` performs register addition → R3 = 5 + 7 = 12 (0x0000000C). This is reflected in the ALU result `0000000C`.
- At 35 ns – Fourth Instruction (00622022): Instruction `sub $4, $3, $2` performs subtraction → R4 = 12 - 7 = 5 (0x00000005).
- At 45 ns – Fifth Instruction (00822824): Instruction `and $5, $4, $2` performs bitwise AND → R5 = 5 & 7 = 5.
- At 55 ns – Sixth Instruction (ac030000): Instruction `sw $3, 0($0)` stores the value of `$3` (0x0000000C) into data memory location 0.

Schematic view in Vivado

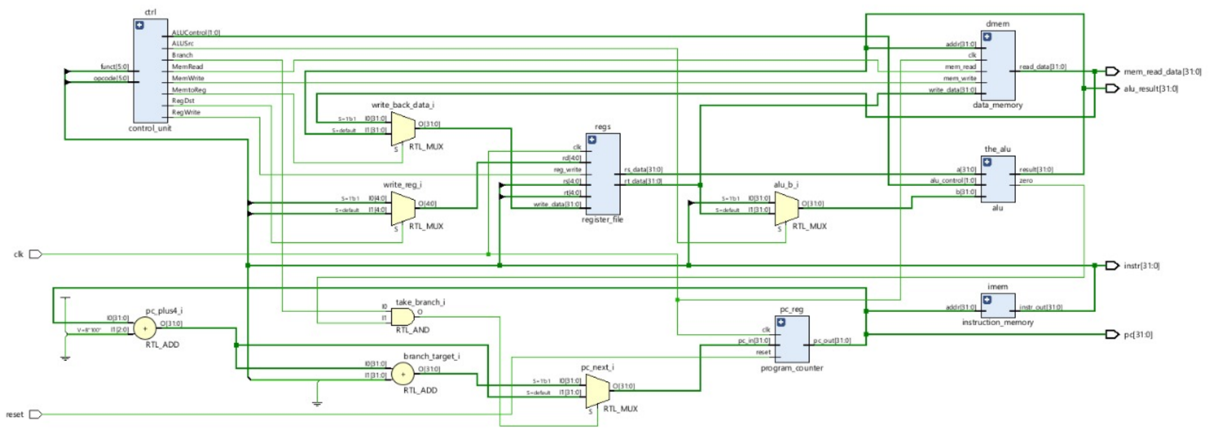


Figure 2: Schematic

The Schematic represents a single-cycle MIPS processor in which each instruction is executed within one clock cycle. It consists of essential components such as the control unit, ALU, register file, program counter, instruction memory, and data memory. Multiplexers (MUXes) are used to control data flow by selecting appropriate paths for register destinations, ALU inputs, and memory operations. The control unit generates the necessary control signals that determine the operation type and data movement within the processor. The program counter updates either sequentially or through a branch target address, ensuring the correct flow of instruction execution.

Advantages and Disadvantages of single cycle mips

Advantages

- Simple Design: The control and datapath are straightforward, making it easier to understand and implement.

- Predictable Timing: Each instruction completes in one clock cycle, simplifying timing analysis.
- No Pipeline Hazards: Since only one instruction is executed at a time, there are no data or control hazards.
- Easy Debugging: The one-cycle-per-instruction behavior makes debugging and simulation simpler.
- Good for Small Programs: Suitable for educational use and simple applications where performance is not critical.

Disadvantages

- Long Clock Cycle: The clock period must be long enough to accommodate the slowest instruction, reducing overall speed.
- Inefficient Hardware Usage: Functional units (like ALU and memory) remain idle for part of the cycle.
- Poor Performance Scaling: Not suitable for complex or high-speed processors where efficiency matters.
- High Power Consumption: Long cycle time and unused hardware during parts of execution waste energy.
- Not Practical for Modern CPUs: Real-world processors use pipelining or multi-cycle designs for better performance.

Cadence Implementation and Results- 90 nm

The single-cycle MIPS processor was synthesized and physically implemented using the Cadence Genus Synthesis Solution and Cadence Innovus Implementation System for a 90nm CMOS technology node. The design flow began with RTL coding and functional verification using simulation tools to ensure logical correctness. The verified RTL was then synthesized in Cadence Genus to convert the high-level Verilog description into a gate-level netlist optimized for area, power, and timing constraints specific to the 90nm process. Following synthesis, the design was imported into Cadence Innovus for physical implementation, which involved floorplanning, placement, clock tree synthesis (CTS), routing, and post-layout optimization. The physical design flow ensured that the layout met all design rule checks (DRC), layout versus schematic (LVS) verification, and timing closure requirements. The final implementation results included reports on area utilization, power consumption, and critical path delay, confirming that the single-cycle MIPS processor met the performance targets within the constraints of the 90nm CMOS technology.

Synthesis in Cadence Genus

Synthesis in Cadence for the single-cycle MIPS processor is performed using the Cadence Genus Synthesis Solution. The RTL Verilog code and 90nm standard cell library are imported into the tool. Design constraints such as clock period and input/output delays are defined using an SDC file. Genus elaborates the RTL, checks hierarchy, and prepares it for optimization. Logic optimization is performed to minimize area, delay, and power consumption. The design is then mapped to standard cells from the 90nm library to create a gate-level netlist. Timing, area, and power reports are generated to verify synthesis quality. Finally, the synthesized netlist is exported to Cadence Innovus for physical implementation.

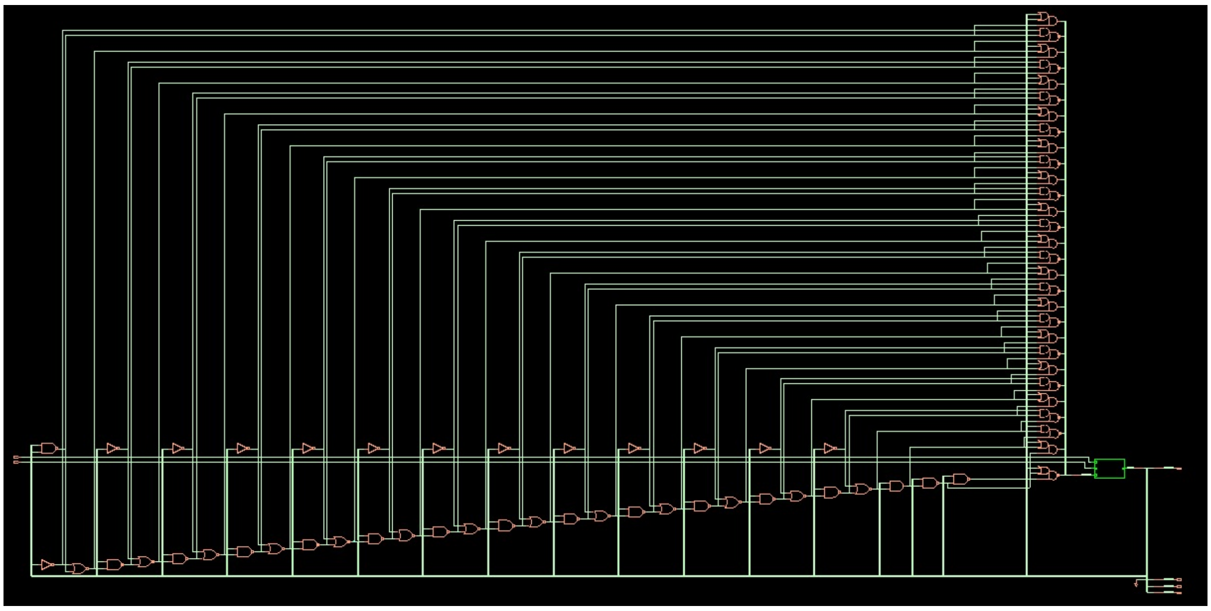


Figure 3: Gate-level schematic of the Single Cycle MIPS synthesized in Cadence Genus using the 90nm library

Table 5: Synthesis and Timing Analysis Summary of Single-Cycle MIPS Processor

Parameter	**Value**	**Unit / Notes**
Area and Cell Count		
Total Cell Count	91,011	–
Total Area	900,711	μm^2
Power Analysis		
Leakage Power	4.20×10^{-6}	W
Internal Power	5.97×10^{-5}	W
Switching Power	5.29×10^{-6}	W
Dynamic Power (Internal + Switching)	6.50×10^{-5}	W
Total Power	6.52×10^{-5}	W
Timing Analysis		
Worst Slack	+4,854	ps
Critical Path Delay	5,023	ps (5.02 ns)
Required Time	9,877	ps (9.88 ns)
Clock Period	10,000	ps (1.00×10^{1} ns)
Clock Frequency	100	MHz

The synthesis and timing analysis demonstrate that the single-cycle MIPS processor was efficiently implemented in 90nm CMOS technology. The synthesis process optimized the design for area, power, and timing while maintaining logical correctness. Timing analysis confirmed that all paths meet the required constraints, ensuring stable and reliable operation. The design exhibits efficient power utilization and balanced performance across all functional blocks. Overall, the implementation achieved successful timing closure, indicating a well-optimized and high-quality design.

Physical Design in Cadence Innovu

The gate-level netlist and constraint files were imported into Cadence Innovus for floor planning, placement, clock tree synthesis, routing, and physical verification. The Innovus automation flow was executed using the TCL script run innovus.tcl, supported by init design.tcl, place route.tcl, and post-layout timing constraints from constraints_input.sdc

Post-Layout Implementation

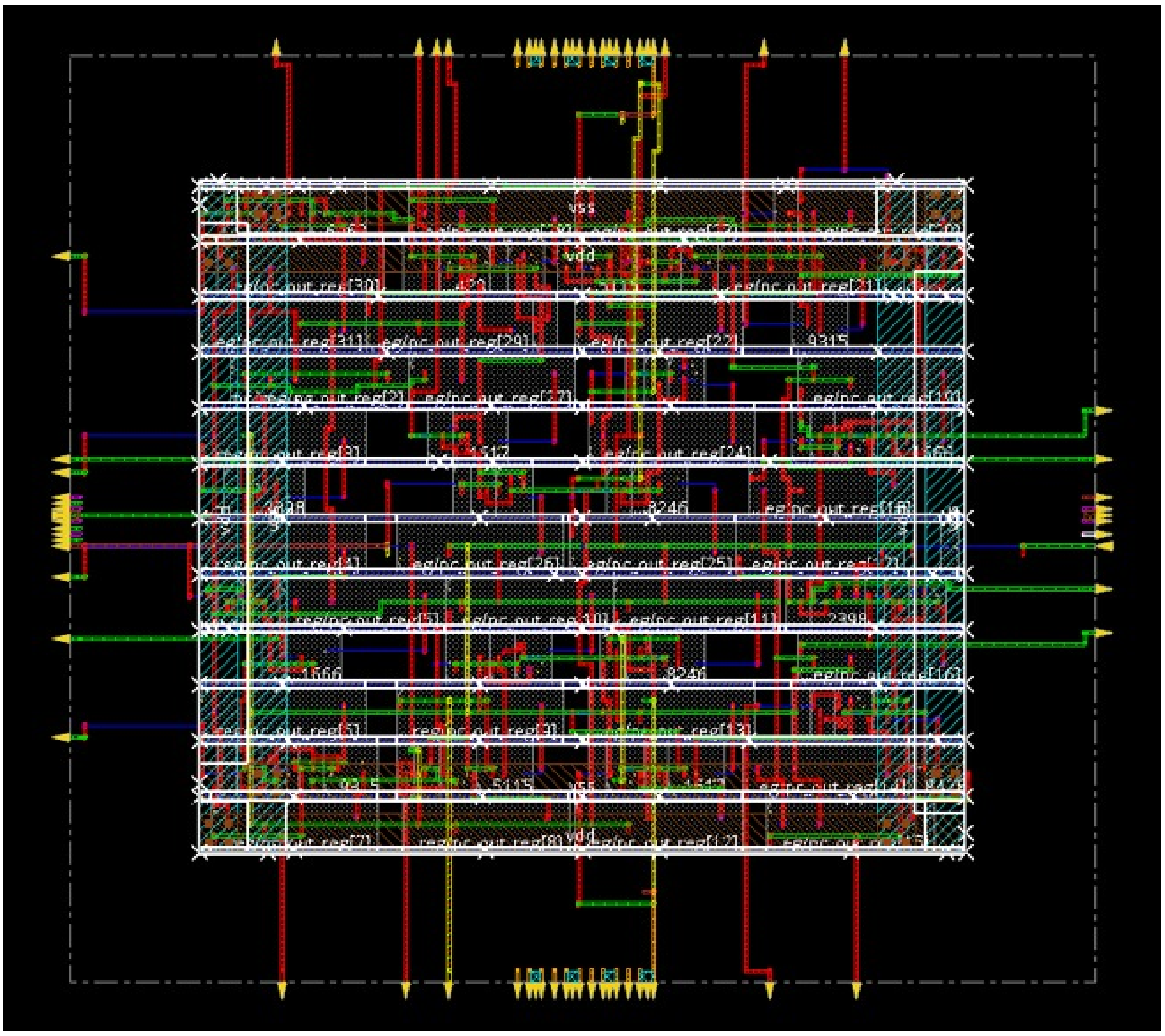


Figure 4: Layout

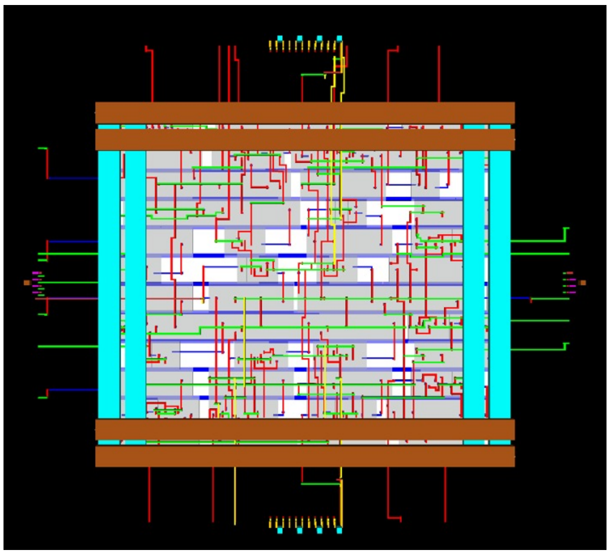


Figure 5: 3D view

The 90nm layout of the single-cycle MIPS processor shows a well-organized arrangement of standard cells with optimized interconnections. Routing is efficiently distributed across multiple metal layers, which helps reduce congestion and maintain proper clock balance. The physical design ensures effective area usage, reliable signal paths, and adherence to all layout design rules. A three-dimensional layout view illustrates the structured use of metal layers and uniform routing distribution. When compared with the pre-layout stage, the final layout displays improved floorplan density and smoother routing paths.

Physical Verification

During the physical verification of the 180nm design, both design rule checking (DRC) and layout versus schematic (LVS) analyses were performed. The DRC report confirmed that the layout is clean and adheres to all foundry design rules. Similarly, the LVS verification showed complete consistency between the layout and the schematic netlist. It is advisable to perform final DRC, LVS, and antenna checks before generating the GDSII file for fabrication. These verification steps ensure the physical integrity and manufacturability of the final chip layout.

Overview

The ASIC design flow moves through nine distinct stages, beginning with **RTL Design**, where the circuit's behavior is defined using `Verilog/VHDL` code based on a specification. This code is then processed in **Logic Synthesis**, where it's mapped to a `Gate-Level Netlist` using a standard cell library and constraints. Next, **Design for Test (DFT)** inserts scan structures into the `Gate-Level Netlist` to create a `Scan Netlist`, enabling post-fabrication fault testing. **Floorplanning** follows, defining the chip area and pin locations to establish the `Floorplan`. In the **Placement** stage, standard cells are optimized within the `Floorplan` to create a `Placed Netlist`, minimizing delay and congestion. **Clock Tree Synthesis (CTS)** builds a balanced clock network on the `Placed Netlist`, resulting in the `CTS Netlist` for uniform clock distribution. **Routing** then connects the cells using metal layers to form the `Routed Netlist`, adhering to design rule checks (DRC). The **Physical Verification** stage runs DRC, Layout Versus Schematic (LVS), and extraction on the `Routed Netlist`, yielding a `Verified Layout` that ensures physical and functional correctness. Finally, **GDSII Generation** streams out the final

`Verified Layout` as a `GDSII File`, which is the foundry-ready format for fabrication.

Conclusion

In conclusion, the single-cycle MIPS processor was successfully designed, synthesized, and implemented using the standard ASIC design flow. The Verilog RTL description was optimized through synthesis and verified through post-layout analysis. Timing closure was achieved with positive slack, ensuring reliable high-speed operation. Physical design steps such as placement, routing, and clock tree synthesis were efficiently completed. DRC and LVS verification confirmed the layout's compliance and correctness. Overall, the design demonstrates a complete and validated ASIC implementation ready for fabrication.

Reference

The project we have taken as a reference is **MIPS**