Viet Nam National University
Ho Chi Minh University of Technology
Faculty of Computer Science and Engineering

LSI Desgin Assignment Report
Design and Implement a 8-bit RISC CPU

Instructor: Tôn Huỳnh Long
Students: Lê Lộc Quốc Thịnh - 2213278
Võ Phúc Thọ - 2213323
Phạm Đào Nhật Thuyên - 2213370
Lâm Quang Trung - 2213686
Class: L04 - **Group:** 01

Ho Chi Minh City, 05/2025

Members List

No.	Full Name	Student ID	Role	Evaluation
1	Lê Lộc Quốc Thịnh	2213278	Design and Implement the CPU	100%
2	Võ Phúc Thọ	2213323	Prepare slide for presentation	100%
3	Phạm Đào Nhật Thuyên	2213370	Write the report	100%
4	Lâm Quang Trung	2213686	Write the report	100%

Preface

The design and implementation of digital systems form the cornerstone of modern computing technology. As part of the curriculum in the Faculty of Science and Computer Engineering, Department of Computer Engineering, this project focuses on the development of a simple 8-bit RISC (Reduced Instruction Set Computer) CPU using Verilog HDL. The primary objective is to enhance proficiency in hardware description languages, foster skills in modular digital circuit design, and deepen understanding of computer architecture principles. This project not only serves as a practical application of theoretical knowledge but also encourages creativity, teamwork, and self-directed learning.

Our team, consisting of up to four members under the leadership and guidance of leader **Lê Lộc Quốc Thịnh**, has undertaken the challenge of designing a RISC CPU with a 3-bit opcode and 5-bit operand architecture. The system is designed to execute a predefined set of instructions, including HLT, SKZ,.ADD, AND, XOR, LDA, STO, and JMP, while ensuring efficient control and data flow through components such as the ALU, Controller, Memory, and Program Counter. By adhering to a hierarchical design approach, we aim to create a robust and scalable system that meets the specified requirements while allowing room for potential enhancements.

This report documents our journey through the design, implementation, testing, and evaluation phases of the project. It provides a comprehensive overview of the system architecture, detailed descriptions of each module, simulation results, and performance evaluations. We also reflect on the challenges encountered, lessons learned, and potential avenues for future development. Through this endeavor, we aim to contribute to the broader understanding of RISC CPU design and its practical applications in digital systems.

Overview

The RISC CPU designed in this project is an 8-bit architecture tailored for simplicity and efficiency, adhering to the principles of Reduced Instruction Set Computing. The CPU operates with a 3-bit opcode, supporting eight distinct instructions, and a 5-bit address space, allowing access to 32 memory locations. The system is clocked synchronously, with a controller managing eight operational phases to ensure proper instruction fetch, decode, and execution. Key components include:

- **Program Counter (PC)**: A 5-bit counter that tracks the current instruction address, supporting reset, load, and increment operations.
- **Address Multiplexer (Mux)**: Selects between the program counter and operand address based on the controller's select signal.
- **Memory**: An 8-bit data, 5-bit address memory module that stores instructions and data, supporting read and write operations with a single bidirectional data port.
- **Instruction Register (IR)**: Captures and holds the fetched instruction for decoding.
- **Decoder**: Extracts the 3-bit opcode and 5-bit address from the 8-bit instruction.
- **Arithmetic Logic Unit (ALU)**: Performs arithmetic and logical operations (ADD, AND, XOR, LDA, etc.) and generates an asynchronous `is_zero` signal to indicate if the accumulator is zero.
- **Accumulator Register**: Stores the 8-bit result of ALU operations or data loaded from memory.
- **Controller**: Orchestrates the CPU's operation through eight phases (INST_ADDR, INST_FETCH, INST_LOAD, IDLE, OP_ADDR, OP_FETCH, ALU_OP, STORE), generating control signals such as `sel`, `rd`, `wr`, `ld_ir`, `ld_ac`, `ld_pc`, `inc_pc`, `data_e`, and `halt`.
- **Clock Divider**: Adjusts the input clock frequency to ensure synchronization and visibility of CPU operations (just used for demo with FPGA).

The CPU is designed to execute a preloaded program stored in the memory module, which includes a sequence of instructions to test the functionality of jumps, skips, arithmetic operations, and data storage. The testbench (`RISC_CPU_tb.v`) verifies the correct execution of this program by monitoring the program counter, accumulator, and `is_zero` signal at critical points.

This report is structured as follows:

- **Introduction**: Provides context, objectives, and tools used in the project.
- **Theoretical Background**: Discusses RISC architecture principles and their relevance to the design.
- **Design**: Presents the system block diagram, functional modules, and their interactions.
- **Implementation**: Details the Verilog implementation, including module integration and program flow.
- **Testing and Evaluation**: Summarizes simulation results, test cases, and performance metrics such as resource utilization and timing.
- **Conclusion**: Reflects on the project's outcomes, challenges, and potential future enhancements.

By leveraging Verilog HDL and a modular design approach, this project demonstrates a functional RISC CPU capable of executing a predefined instruction set, laying the foundation for further exploration in computer architecture and digital system design.

- You can examine the project directly via this link which include the read me and simulation files as well the Synthesis and LEC files.

Contents

1 Introduction

1.1 Tools Utilized

The development, simulation, and verification of the RISC CPU leverage a suite of industry-standard tools to ensure robust design and implementation:

- **Vivado**: Utilized for simulation, synthesis, and implementation of the design. Vivado is also used to test the CPU on the Arty-Z7 FPGA board, allowing for real-world hardware validation.
- **Cadence Xcelium**: Employed for advanced simulation, providing detailed waveform generation to analyze the CPU's behavior during execution.
- **Cadence Genus**: Used for synthesis, converting the Verilog HDL code into a gate-level netlist optimized for performance and resource utilization.
- **Cadence Conformal**: Applied for Logic Equivalence Checking (LEC), ensuring that the synthesized netlist is functionally equivalent to the original Verilog design.

1.2 Hardware Utilized

The RISC CPU design is targeted for implementation on the **Arty-Z7 FPGA board**, which features a Xilinx Zynq-7000 SoC. This board provides a versatile platform for testing and validating the CPU's functionality in a hardware environment, allowing for real-time execution of the preloaded program and verification of control and data flow.

1.3 Product Functionalities

The 8-bit RISC CPU is designed to execute a predefined set of instructions efficiently, with the following key functionalities:

- **Instruction Execution**: Supports eight instructions:
  - **HLT (000)**: Halts CPU operation.
  - **SKZ (001)**: Skips the next instruction if the accumulator is zero.
  - **ADD (010)**: Adds the accumulator and memory data, storing the result in the accumulator.

  - **AND (011)**: Performs a bitwise AND operation between the accumulator and memory data.
  - **XOR (100)**: Performs a bitwise XOR operation between the accumulator and memory data.
  - **LDA (101)**: Loads data from memory into the accumulator.
  - **STO (110)**: Stores the accumulator value into memory.
  - **JMP (111)**: Unconditionally jumps to a specified address.

- **Memory Management**: Utilizes a 32-location memory module (5-bit address, 8-bit data) to store instructions and data, with support for read and write operations via a single bidirectional data port.

- **Control Flow**: Managed by a controller with eight operational phases (INST_ADDR, INST_FETCH, INST_LOAD, IDLE, OP_ADDR, OP_FETCH, ALU_OP, STORE) to ensure proper instruction fetch, decode, and execution.

- **Program Counter**: Tracks the current instruction address, supporting reset, load, and increment operations.

- **ALU Operations**: Performs arithmetic and logical operations, generating an asynchronous `is_zero` signal to indicate a zero accumulator value.

- **Clock Synchronization**: Incorporates a clock divider to adjust the input clock frequency for synchronization and visibility of CPU operations.

The CPU is tested with a preloaded program in the memory module, designed to validate the execution of jumps, skips, arithmetic operations, and data storage, ensuring robust functionality across all components.

2 Theoretical Background

2.1 Overview of RISC Architecture

Reduced Instruction Set Computing (RISC) is a design philosophy that emphasizes a small, highly optimized instruction set to improve performance and simplify hardware implementation. Unlike Complex Instruction Set Computing (CISC), RISC architectures prioritize:

- **Simplified Instructions**: Each instruction is designed to execute in a single clock cycle, reducing complexity and improving execution speed.
- **Load-Store Architecture**: Only load and store instructions access memory, while arithmetic and logical operations are performed on registers, enhancing efficiency.
- **Fixed Instruction Length**: Instructions have a uniform format, simplifying decoding and pipelining.
- **Large Register Set**: Reduces memory access by storing intermediate results in registers.

The 8-bit RISC CPU in this project embodies these principles by implementing a compact instruction set with a fixed 8-bit instruction format (3-bit opcode, 5-bit address). This design minimizes hardware complexity while maintaining functionality, making it suitable for educational purposes and small-scale embedded systems.

2.2 Practical Applications

The features of the RISC CPU designed in this project have several practical applications in real-world digital systems:

- **Embedded Systems**: The simplicity and efficiency of the RISC CPU make it ideal for resource-constrained embedded devices, such as microcontrollers in IoT devices, sensors, and automotive systems.
- **Educational Platforms**: The CPU serves as a valuable teaching tool for computer architecture and digital design courses, providing hands-on experience with CPU operation, instruction execution, and hardware-software integration.
- **Prototyping and Rapid Development**: The modular design and Verilog implementation allow for easy modification and extension, making it suitable for prototyping new CPU features or testing custom instruction sets.
- **Low-Power Applications**: The streamlined instruction set and minimal hardware requirements reduce power consumption, applicable in battery-operated devices like wearable technology and portable electronics.

- **Digital Signal Processing**: The ALU's arithmetic and logical operations can be adapted for basic signal processing tasks in applications like audio processing or simple image filtering.

2.3 Block Diagram

The block diagram of the 8-bit RISC CPU, shown below, illustrates the main components and their interconnections:

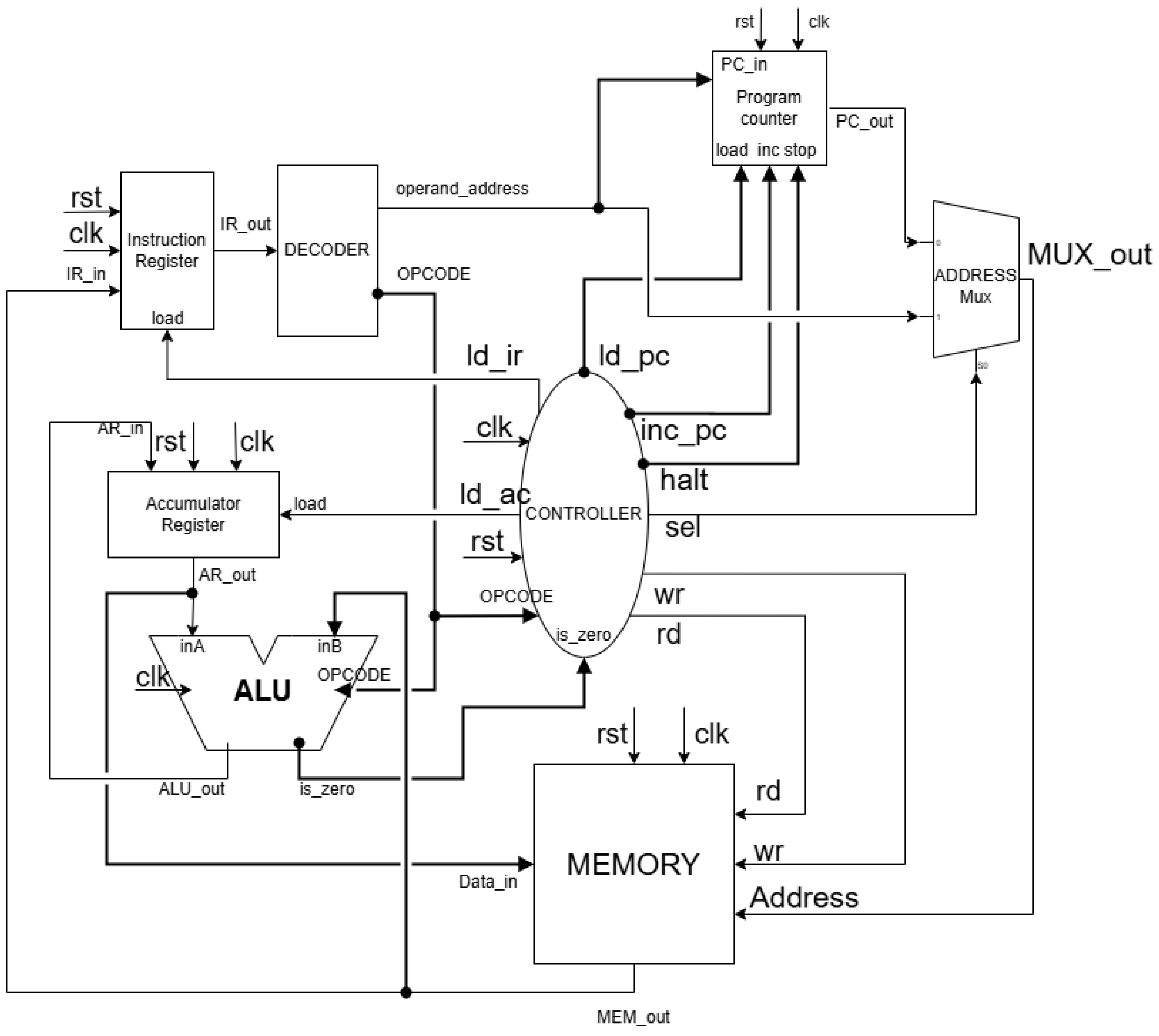


Figure 1: Block Diagram of RISC CPU

The diagram includes:

- **Program Counter (PC)**: Stores the address of the next instruction.
- **Address Multiplexer (MUX)**: Selects between PC or operand addresses.
- **Memory**: Holds instructions and data.
- **Instruction Register (IR)**: Stores the current instruction.

- **Decoder**: Extracts opcode and operand address from instructions.
- **Arithmetic Logic Unit (ALU)**: Performs arithmetic and logical operations.
- **Accumulator Register**: Stores ALU results or memory data.
- **Controller**: Generates control signals to coordinate operations.

These components are interconnected to facilitate data flow, with the controller orchestrating operations via control signals.

2.4 Functional Blocks

Below is a detailed description of each functional block, including its function, inputs, outputs, and role in the CPU.

2.4.1 Program Counter (PC)

- **Function**: Tracks the address of the current or next instruction to be executed.
- **Inputs**:
  - Load signal (`ld_pc`): Enables loading a new address.
  - Increment signal (`inc_pc`): Increments PC by 1.
  - Reset signal (`rst`): Resets PC to 0.
  - Load address (`PC_in`): New address for jump instructions.
- **Output**: Current address (`PC_out`).
- **Description**: The PC is a 5-bit register that holds the memory address of the instruction to be fetched. It supports incrementing for sequential execution, loading for jumps, and resetting at startup. Its output feeds into the address multiplexer for instruction fetching.

2.4.2 Address Multiplexer (MUX)

- **Function**: Selects the memory address source.
- **Inputs**:
  - PC output (`PC_out`): Address from the program counter.

  - Operand address (`operand_address`): Address from the decoder.
  - Select signal (`sel`): Chooses between PC and operand address.

- **Output**: Memory address (`ADDRESS MUX_out`).
- **Description**: The MUX determines whether the memory address comes from the PC (for fetching instructions) or the operand address (for data access). The `sel` signal, controlled by the controller, governs this selection.

2.4.3 Memory

- **Function**: Stores instructions and data.
- **Inputs**:
  - Address: From the address multiplexer.
  - `Data_in`: From Accumulator Register for write operations.
  - Write signal (`wr`): Enables writing to memory.
  - Read signal (`rd`): Enables reading from memory.
  - Clock (`clk`): Synchronizes operations.
  - Reset (`rst`): Initializes memory.
- **Output**: `Data_out`.
- **Description**: The memory module has 32 locations (5-bit address, 8-bit data), storing both instructions and data. It supports read and write operations via a bidirectional data port and is preloaded with a test program to verify CPU functionality.

2.4.4 Instruction Register (IR)

- **Function**: Holds the current instruction fetched from memory.
- **Inputs**:
  - Data from memory (`IR_in`): Fetched instruction.
  - Load signal (`ld_ir`): Enables loading into IR.
  - Clock (`clk`): Synchronizes operations.

  - Reset (`rst`): Resets IR.

- **Output**: Instruction output (`IR_out`).
- **Description**: The IR is an 8-bit register that captures the instruction from memory when `ld_ir` is active. It holds the instruction for decoding by the decoder.

2.4.5 Decoder

- **Function**: Extracts opcode and operand address from the instruction.
- **Input**: Instruction from IR (`IR_out`).
- **Outputs**:
  - Opcode (`OPCODE`): 3-bit instruction identifier.
  - Operand address (`operand_address`): 5-bit memory address.
- **Description**: The decoder processes the 8-bit instruction, separating it into a 3-bit opcode (bits 7-5) and a 5-bit operand address (bits 4-0). The opcode is sent to the controller and ALU, while the operand address is used by the MUX and PC.

2.4.6 Arithmetic Logic Unit (ALU)

- **Function**: Performs arithmetic and logical operations.
- **Inputs**:
  - Operand A (`inA`): From the accumulator.
  - Operand B (`inB`): From memory.
  - Opcode (`OPCODE`): Specifies the operation.
- **Outputs**:
  - Result (`ALU_out`): Operation result.
  - Zero flag (`is_zero`): Indicates if the result is zero.
- **Description**: The ALU executes operations like ADD, AND, XOR, and LDA based on the opcode. It generates an asynchronous `is_zero` signal for conditional instructions like SKZ, with results sent to the accumulator.

2.4.7 Accumulator Register

- **Function**: Stores ALU results or data from memory.
- **Inputs**:
  - Data from ALU (`AR_in`): ALU operation result.
  - Load signal (`ld_ac`): Enables loading data.
  - Clock (`clk`): Synchronizes operations.
  - Reset (`rst`): Resets the accumulator.
- **Output**: Data to ALU (`AR_out`).
- **Description**: The accumulator is an 8-bit register that holds intermediate results or data loaded from memory. It serves as the primary register for ALU operations.

2.4.8 Controller

- **Function**: Coordinates CPU operations via control signals.
- **Inputs**:
  - Clock (`clk`): Synchronizes operations.
  - Reset (`rst`): Initializes the controller.
  - Opcode (`OPCODE`): From the decoder.
  - Zero flag (`is_zero`): From ALU.
- **Outputs**: Control signals:
  - `sel`: Selects MUX address source.
  - `rd`: Enables memory read.
  - `wr`: Enables memory write.
  - `ld_ir`: Loads instruction into IR.
  - `ld_ac`: Loads data into accumulator.
  - `ld_pc`: Loads address into PC.
  - `inc_pc`: Increments PC.

  - `halt`: Stops CPU operation.

- **Description**: The controller is a finite state machine with eight states (INST_ADDR, INST_FETCH, INST_LOAD, IDLE, OP_ADDR, OP_FETCH, ALU_OP, STORE). It generates control signals based on the current state, opcode, and `is_zero` flag to manage CPU operations.

2.4.9 Clock Divider (just used for demo directly with Arty-z7)

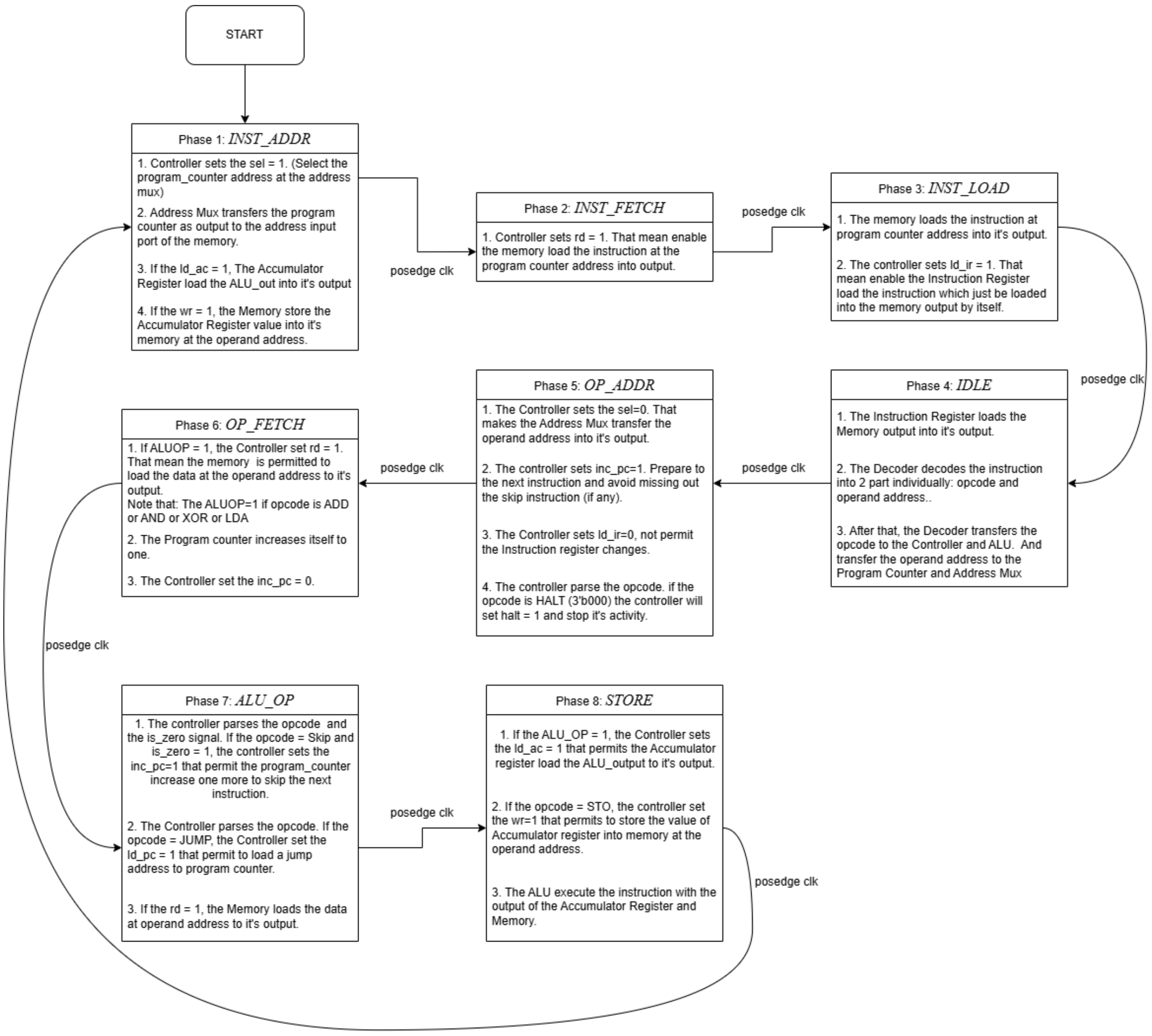
- **Function**: Generates a lower-frequency clock signal.
- **Inputs**:
  - Input clock (`clk`): Original clock signal.
  - Reset (`rst`): Resets the divider.
  - Enable (`enable`): Activates the divider.
- **Output**: Divided clock (`divided_clk`).
- **Description**: The clock divider reduces the input clock frequency to facilitate simulation and testing, slowing down CPU operations for better observability.

Block	**Inputs**	**Outputs**	**Function**
Instruction Register	`IR_in`, `rst`, `clk`, `load`	`IR_out`	Loads and holds the current instruction for decoding.
Decoder	`IR_out`	`OPCODE`, `operand_address`	Extracts opcode and address from the instruction.
Program Counter	`PC_in`, `rst`, `clk`, `load`, `inc`	`PC_out`	Tracks and updates the instruction address.
Address Multiplexer	`PC_out`, `operand_address`, `sel`	`ADDRESS_Mux_out`	Selects between PC and operand address for memory access.
Controller	`OPCODE`, `is_zero`, `clk`, `rst`	Multiple control signals	Generates signals to manage CPU operation.
Accumulator Register	`AR_in`, `rst`, `clk`, `load`	`AR_out`	Stores ALU results for further processing.
ALU	`inA`, `inB`, `OPCODE`, `clk`	`ALU_out`, `is_zero`	Performs arithmetic and logical operations, sets zero flag.
Memory	`Address`, `wr`, `rd`, `Data_in`	`MEM_out`, `Address`	Stores and retrieves instructions and data.

Table 1: Functional Block Summary

2.5 Operational Phases

The CPU executes instructions through eight operational phases, each handling a specific part of the fetch-decode-execute cycle. The phases and their operations are detailed in the image and table below:



Phase 1: *INST_ADDR*

1. Controller sets the sel = 1. (Select the program_counter address at the address mux)
2. Address Mux transfers the program counter as output to the address input port of the memory.
3. If the ld_ac = 1, The Accumulator Register load the ALU_out into it's output
4. If the wr = 1, the Memory store the Accumulator Register value into it's memory at the operand address.

Phase 2: *INST_FETCH*

1. Controller sets rd = 1. That mean enable the memory load the instruction at the program counter address into output.

Phase 3: *INST_LOAD*

1. The memory loads the instruction at program counter address into it's output.
2. The controller sets ld_ir = 1. That mean enable the Instruction Register load the instruction which just be loaded into the memory output by itself.

Phase 4: *IDLE*

1. The Instruction Register loads the Memory output into it's output.
2. The Decoder decodes the instruction into 2 part individually: opcode and operand address..
3. After that, the Decoder transfers the opcode to the Controller and ALU. And transfer the operand address to the Program Counter and Address Mux

Phase 5: *OP_ADDR*

1. The Controller sets the sel=0. That makes the Address Mux transfer the operand address into it's output.
2. The controller sets inc_pc=1. Prepare to the next instruction and avoid missing out the skip instruction (if any).
3. The Controller sets ld_ir=0, not permit the Instruction register changes.
4. The controller parse the opcode. if the opcode is HALT (3'b000) the controller will set halt = 1 and stop it's activity.

Phase 6: *OP_FETCH*

1. If ALUOP = 1, the Controller set rd = 1. That mean the memory is permitted to load the data at the operand address to it's output.
Note that: The ALUOP=1 if opcode is ADD or AND or XOR or LDA
2. The Program counter increases itself to one.
3. The Controller set the inc_pc = 0.

Phase 7: *ALU_OP*

1. The controller parses the opcode and the is_zero signal. If the opcode = Skip and is_zero = 1, the controller sets the inc_pc=1 that permit the program_counter increase one more to skip the next instruction.
2. The Controller parses the opcode. If the opcode = JUMP, the Controller set the ld_pc = 1 that permit to load a jump address to program counter.
3. If the rd = 1, the Memory loads the data at operand address to it's output.

Phase 8: *STORE*

1. If the ALU_OP = 1, the Controller sets the ld_ac = 1 that permits the Accumulator register load the ALU_output to it's output.
2. If the opcode = STO, the controller set the wr=1 that permits to store the value of Accumulator register into memory at the operand address.
3. The ALU execute the instruction with the output of the Accumulator Register and Memory.

Figure 2: The image to depict Operational Phases of the CPU

Phase	**Description**	**Main Operations**
INST_ADDR	Select address to fetch instruction.	Controller sets `sel = 1`, MUX selects PC address, PC provides address to memory.
INST_FETCH	Fetch instruction from memory.	Memory loads instruction at PC address to output, controller sets `rd = 1`.
INST_LOAD	Load instruction into instruction register.	IR loads instruction from memory output, controller sets `ld_ir = 1`.
IDLE	Decode instruction and prepare for next step.	Decoder splits opcode and operand address, sends to controller and MUX.
OP_ADDR	Select address for operand.	Controller sets `sel = 0`, MUX selects operand address, checks for HLT instruction.
OP_FETCH	Fetch operand from memory.	Memory loads data at operand address, ALU prepares operation, PC increments if needed.
ALU_OP	Perform ALU operation or store result.	ALU performs operation, handles jump/skip instructions, stores result if needed.
STORE	Store the value of Accumulator Register into memory.	Memory store the accumulator Register value into it's memory if needed.

Table 2: Operational Phases of the RISC CPU

3 Simulation

The testing and evaluation phase is critical to ensuring the 8-bit RISC CPU operates correctly and meets the project's design specifications. This section presents the results of the simulation conducted using the Cadence Xcelium simulator, with the testbench file `RISC-CPU_tb.v`. The simulation executes a carefully designed test program that exercises the CPU's instruction set, including HLT, SKZ, ADD, AND, XOR, LDA, STO, and JMP, to verify the functionality of all components and their interactions. The primary goal is to confirm that the CPU executes the program as intended, halting at the correct address with expected register states, and to identify any potential issues in the design.

3.1 Simulation Setup

3.1.1 Tools and Environment

The simulation was performed using Cadence Xcelium, a high-performance logic simulator known for its accuracy and efficiency in verifying digital designs. Xcelium was used to generate detailed waveforms and analyze the CPU's behavior during execution, providing insights into the timing and state of key signals.

3.1.2 Testbench

The testbench file, `RISC-CPU_tb.v`, serves as the primary interface for driving the simulation. It initializes the CPU, loads the test program into the Memory module, and monitors critical signals such as the Program Counter (PC), Accumulator Register, `is_zero` flag, and `halt` signal. The testbench includes multiple test cases to verify specific aspects of the CPU's functionality, ensuring comprehensive coverage of the instruction set and control flow.

3.1.3 Test Program

The test program is a sequence of instructions designed to exercise the CPU's capabilities, including control flow (jumps and skips), data manipulation (loads, stores, and XOR operations), and halting behavior. The program is preloaded into the Memory module at specific addresses, with data values initialized to test various scenarios. The program structure is as follows:

```
//opcode_operand  // addr                   assembly code
//--------------  // ----  -----------------------------------------------
@00 111_11110     //  00   BEGIN:   JMP TST_JMP
```

```
    000_00000     //  01            HLT
    000_00000     //  02            HLT
    101_11010     //  03   JMP_OK:  LDA DATA_1
    001_00000     //  04            SKZ
    000_00000     //  05            HLT
    101_11011     //  06            LDA DATA_2
    001_00000     //  07            SKZ
    111_01010     //  08            JMP SKZ_OK
    000_00000     //  09            HLT
    110_11100     //  0A   SKZ_OK:  STO TEMP
    101_11010     //  0B            LDA DATA_1
    110_11100     //  0C            STO TEMP
    101_11100     //  0D            LDA TEMP
    001_00000     //  0E            SKZ
    000_00000     //  0F            HLT
    100_11011     //  10            XOR DATA_2
    001_00000     //  11            SKZ
    111_10100     //  12            JMP XOR_OK
    000_00000     //  13            HLT
    100_11011     //  14   XOR_OK:  XOR DATA_2
    001_00000     //  15            SKZ
    000_00000     //  16            HLT
    000_00000     //  17   END:     HLT
    111_00000     //  18            JMP BEGIN

@1A 00000000      //  1A   DATA_1:   (Value is 0x00)
    11111111      //  1B   DATA_2:   (Value is 0xFF)
    10101010      //  1C   TEMP: (The init variable with value is 0xAA)

@1E 111_00011     //  1E   TST_JMP: JMP JMP_OK
    000_00000     //  1F            HLT
```

The program is designed to:

- Test jump instructions (e.g., JMP to TST_JMP at address 0x1E, then to JMP_OK at 0x03).

- Verify skip-on-zero (SKZ) behavior with zero and non-zero accumulator values.
- Validate load (LDA) and store (STO) operations using data at addresses 0x1A (DATA_1), 0x1B (DATA_2), and 0x1C (TEMP).
- Check XOR operations for correct logical manipulation.
- Ensure the program halts at address 0x17 (END) as specified.

3.2 Test Cases

The testbench includes 13 test cases to verify the CPU's functionality across various scenarios. Each test case checks specific aspects of the CPU's behavior, such as register states, control flow, and instruction execution. The test cases are summarized below:

Test Case	**Description**	**Expected Outcome**	**Result**
1	Initial state check	Accumulator=0x00, is_zero=1	Passed
2	JMP instruction behavior	Program Counter=0x03 after JMP	Passed
3	LDA DATA_1 (0x00)	Accumulator=0x00, is_zero=1	Passed
4	SKZ with zero accumulator	Skip to next instruction	Passed
5	LDA DATA_2 (0xFF)	Accumulator=0xFF, is_zero=0	Passed
6	SKZ with non-zero accumulator	No skip, proceed to JMP	Passed
7	JMP to SKZ_OK	Program Counter=0x0A	Passed
8	STO to TEMP	Memory[0x1C]=Accumulator	Passed
9	LDA TEMP	Accumulator=Memory[0x1C]	Passed
10	SKZ after LDA TEMP	Skip based on accumulator value	Passed
11	XOR DATA_2	Accumulator updated, is_zero updated	Passed
12	JMP to XOR_OK	Program Counter=0x14	Passed
13	Final XOR and SKZ	Accumulator=0x00, halt at 0x17	Passed

Table 3: Test Case Summary

All test cases passed, confirming that the CPU correctly handles the instruction set and control flow.

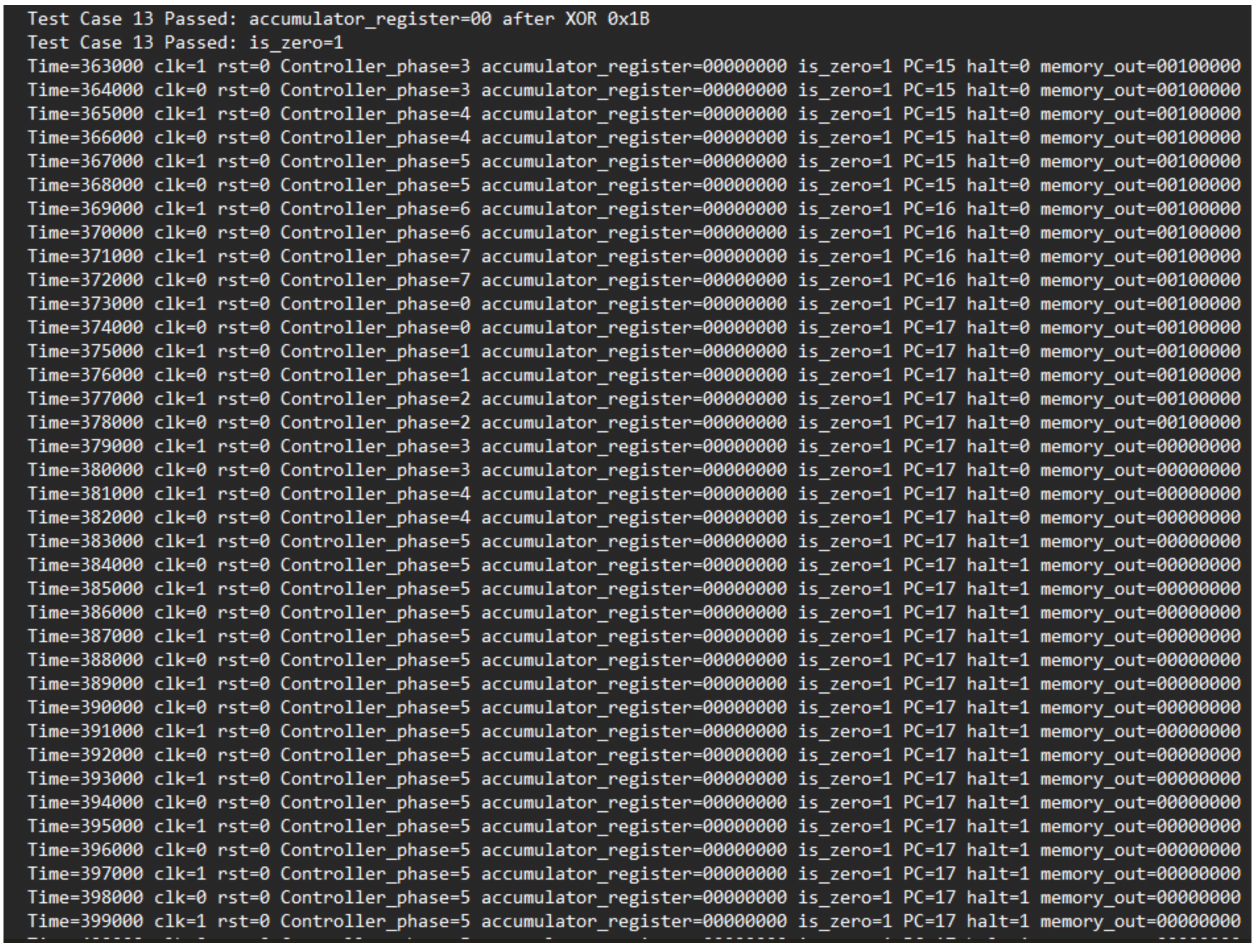
3.3 Simulation Results

The simulation results provide a comprehensive validation of the CPU's functionality:

- **Program Halting:** The program halted at address 0x17 (hex), as expected. The simulation log confirms that the Program Counter (PC) reached 0x17 at Time=373000, with the `halt` signal set to 1, and remained in this state until the simulation ended at Time=562000.
- **Final Register States:**
  - **Program Counter (PC):** 0x17 (hex)
  - **Accumulator Register:** 0x00 (hex)
  - **is_zero Flag:** 1 (indicating a zero accumulator)
  - **Memory Output:** 0x00 (hex)
  - **Halt Signal:** 1 (indicating program termination)
- **Simulation Duration:** The simulation ran for 563 nanoseconds in simulated time, completing all test cases without errors.
- **Error Status:** No errors or warnings were reported during compilation, elaboration, or simulation, indicating a robust design.

3.3.1 Key Simulation Events

- **Time=373000:** The PC reached 0x17, and the `halt` signal was activated, indicating the program's termination at the END label.
- **Time=562000:** The final state was recorded, with PC=0x17, `halt`=1, accumulator_register=0x00, `is_zero`=1, and memory_out=0x00.
- **Test Case Execution:** All 13 test cases executed successfully, verifying the CPU's behavior for jumps, skips, loads, stores, and XOR operations.
- **Simulation Completion:** The simulation terminated via a `$finish` statement in the testbench at Time=563 NS, confirming that the test scenario was fully executed.



```
Test Case 13 Passed: accumulator_register=00 after XOR 0x1B
Test Case 13 Passed: is_zero=1
Time=363000 clk=1 rst=0 Controller_phase=3 accumulator_register=00000000 is_zero=1 PC=15 halt=0 memory_out=00100000
Time=364000 clk=0 rst=0 Controller_phase=3 accumulator_register=00000000 is_zero=1 PC=15 halt=0 memory_out=00100000
Time=365000 clk=1 rst=0 Controller_phase=4 accumulator_register=00000000 is_zero=1 PC=15 halt=0 memory_out=00100000
Time=366000 clk=0 rst=0 Controller_phase=4 accumulator_register=00000000 is_zero=1 PC=15 halt=0 memory_out=00100000
Time=367000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=15 halt=0 memory_out=00100000
Time=368000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=15 halt=0 memory_out=00100000
Time=369000 clk=1 rst=0 Controller_phase=6 accumulator_register=00000000 is_zero=1 PC=16 halt=0 memory_out=00100000
Time=370000 clk=0 rst=0 Controller_phase=6 accumulator_register=00000000 is_zero=1 PC=16 halt=0 memory_out=00100000
Time=371000 clk=1 rst=0 Controller_phase=7 accumulator_register=00000000 is_zero=1 PC=16 halt=0 memory_out=00100000
Time=372000 clk=0 rst=0 Controller_phase=7 accumulator_register=00000000 is_zero=1 PC=16 halt=0 memory_out=00100000
Time=373000 clk=1 rst=0 Controller_phase=0 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00100000
Time=374000 clk=0 rst=0 Controller_phase=0 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00100000
Time=375000 clk=1 rst=0 Controller_phase=1 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00100000
Time=376000 clk=0 rst=0 Controller_phase=1 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00100000
Time=377000 clk=1 rst=0 Controller_phase=2 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00100000
Time=378000 clk=0 rst=0 Controller_phase=2 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00100000
Time=379000 clk=1 rst=0 Controller_phase=3 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00000000
Time=380000 clk=0 rst=0 Controller_phase=3 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00000000
Time=381000 clk=1 rst=0 Controller_phase=4 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00000000
Time=382000 clk=0 rst=0 Controller_phase=4 accumulator_register=00000000 is_zero=1 PC=17 halt=0 memory_out=00000000
Time=383000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=384000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=385000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=386000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=387000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=388000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=389000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=390000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=391000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=392000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=393000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=394000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=395000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=396000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=397000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=398000 clk=0 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
Time=399000 clk=1 rst=0 Controller_phase=5 accumulator_register=00000000 is_zero=1 PC=17 halt=1 memory_out=00000000
```

Figure 3: The image to prove that the simulation run successfully without error.

3.4 Analysis and Conclusion

The simulation results demonstrate that the 8-bit RISC CPU design is fully functional and correctly implements the specified instruction set. The successful execution of all 13 test cases validates the CPU's ability to handle complex control flows (e.g., JMP and SKZ instructions), data manipulation (e.g., LDA, STO, and XOR), and proper termination (HLT at address 0x17). The final register states, particularly the accumulator being 0x00 and the `is_zero` flag being 1, align with the expected outcomes of the test program, which involves XOR operations that result in a zero value.

The absence of errors or warnings in the simulation log underscores the reliability of the design. The modular architecture, with components like the ALU, Controller, and Memory working seamlessly, ensures accurate instruction execution across the seven operational phases (INST_ADDR, INST_FETCH, INST_LOAD, OP_ADDR, OP_FETCH, ALU_OP, STORE). The test program's comprehensive coverage of the instruction set provides confidence in the CPU's robustness.

This successful simulation paves the way for further validation on the Arty-Z7 FPGA board using Vivado, where the design can be tested in a hardware environment. The results will allow for real-time execution and additional performance metrics, such as resource utilization and timing analysis. Overall, the simulation results confirm that the 8-bit RISC CPU meets the project requirements, providing a solid foundation for future enhancements, such as expanding the instruction set or optimizing power consumption.

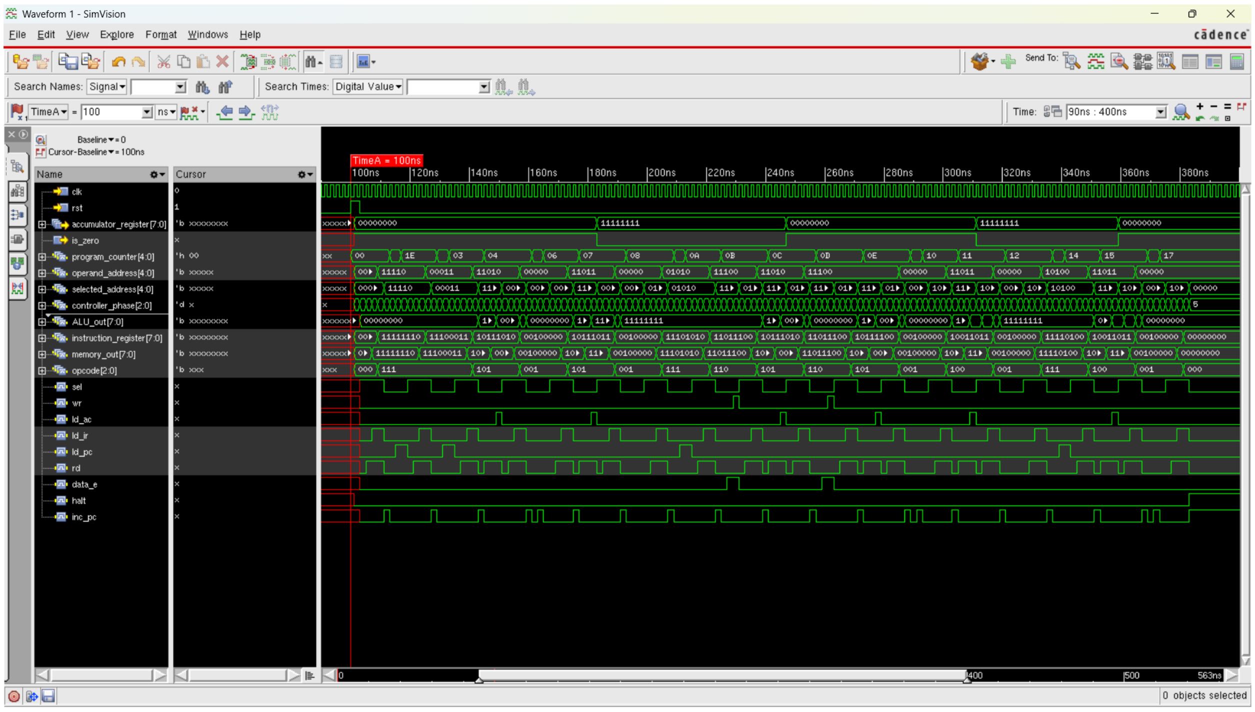


Figure 4: The simulation waveform

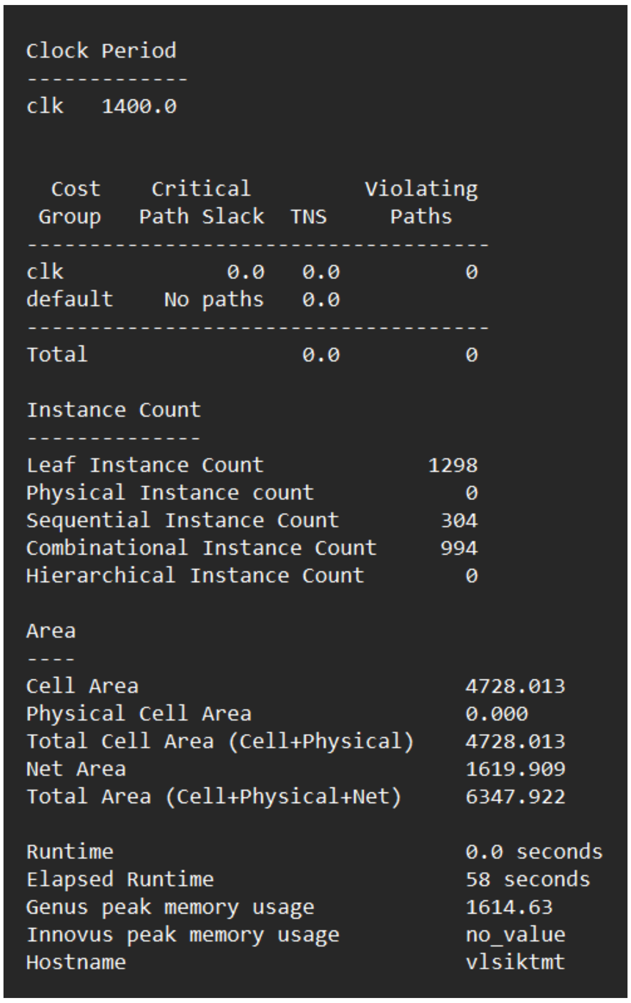
4 Evaluation

This section evaluates the performance metrics of the 8-bit RISC CPU design, focusing on resource utilization, timing, and other relevant parameters. The evaluation is based on synthesis reports generated by Cadence Genus (`final_qor.rpt`, `final_area.rpt`, `final_time.rpt`) and logic equivalence checking results from Cadence Conformal (`lec.log`). The Verilog implementation includes modules such as ALU, Controller, Decoder, Memory, Mux, PC, and Register, forming a complete RISC CPU architecture.

4.1 Resource Utilization

The resource utilization metrics are derived from the `final_qor.rpt` and `final_area.rpt` files, which provide detailed information about the area and instance counts of the synthesized design.

- **Leaf Instance Count**: 1298
- **Sequential Instance Count**: 304
- **Combinational Instance Count**: 994
- **Hierarchical Instance Count**: 0
- **Physical Instance Count**: 0

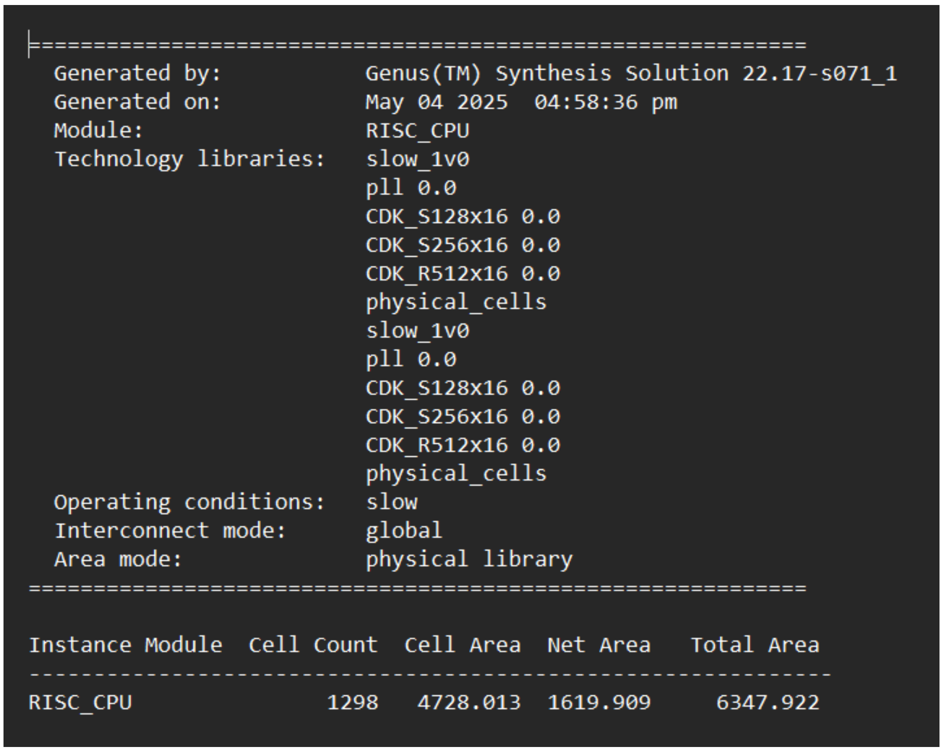


```
Clock Period
-------------
clk   1400.0


  Cost     Critical         Violating
 Group    Path Slack  TNS     Paths
-------------------------------------
clk             0.0   0.0          0
default    No paths   0.0
-------------------------------------
Total                 0.0          0

Instance Count
--------------
Leaf Instance Count             1298
Physical Instance count            0
Sequential Instance Count        304
Combinational Instance Count     994
Hierarchical Instance Count        0

Area
----
Cell Area                          4728.013
Physical Cell Area                 0.000
Total Cell Area (Cell+Physical)    4728.013
Net Area                           1619.909
Total Area (Cell+Physical+Net)     6347.922

Runtime                            0.0 seconds
Elapsed Runtime                    58 seconds
Genus peak memory usage            1614.63
Innovus peak memory usage          no_value
Hostname                           vlsiktmt
```

Figure 5: The final_qor file generated by Genus

The design consists of 1298 leaf instances, with 304 sequential elements (e.g., flip-flops) and 994 combinational elements (e.g., logic gates). The absence of hierarchical or physical instances indicates a flat design optimized for synthesis.

- **Cell Area**: $4728.013\,\mu m^2$
- **Physical Cell Area**: $0.000\,\mu m^2$
- **Total Cell Area (Cell + Physical)**: $4728.013\,\mu m^2$
- **Net Area**: $1619.909\,\mu m^2$
- **Total Area (Cell + Physical + Net)**: $6347.922\,\mu m^2$



```
============================================================
  Generated by:          Genus(TM) Synthesis Solution 22.17-s071_1
  Generated on:          May 04 2025  04:58:36 pm
  Module:                RISC_CPU
  Technology libraries:  slow_1v0
                         pll 0.0
                         CDK_S128x16 0.0
                         CDK_S256x16 0.0
                         CDK_R512x16 0.0
                         physical_cells
                         slow_1v0
                         pll 0.0
                         CDK_S128x16 0.0
                         CDK_S256x16 0.0
                         CDK_R512x16 0.0
                         physical_cells
  Operating conditions:  slow
  Interconnect mode:     global
  Area mode:             physical library
============================================================

Instance Module  Cell Count  Cell Area  Net Area   Total Area
--------------------------------------------------------------
RISC_CPU               1298   4728.013  1619.909     6347.922
```

Figure 6: The final_area file generated by Genus

The total area of 6347.922 µm^2 includes both cell area (logic gates and sequential elements) and net area (interconnect wiring). The cell area of 4728.013 µm^2 reflects the core logic footprint, while the net area of 1619.909 µm^2 indicates the interconnect overhead, which is reasonable for a design of this complexity.

4.2 Timing Analysis

Timing performance is evaluated using the `final_time.rpt` and `final_qor.rpt` files, which detail the critical paths and clock performance under the specified operating conditions.

- **Clock**: clk
- **Period**: 1400 ps (1.4 ns, equivalent to ∼714 MHz)

The design operates with a clock period of 1.4 ns, allowing a maximum clock frequency of approximately 714 MHz under the `slow_1v0` technology library and slow operating conditions.

- **Critical Path Slack**: 0 ps (most paths)

- **Total Negative Slack (TNS)**: 0 ps
- **Violating Paths**: 0

The synthesis results show that all timing paths meet the setup requirements, with the critical paths having a slack of 0 ps (just meeting timing) or slightly positive slack (1–2 ps in some paths). The absence of negative slack and violating paths indicates that the design is fully timing-compliant at the specified clock frequency.

Critical Path Analysis: The `final_time.rpt` lists several critical paths, with the longest paths summarized below:

- **Path 1**: From `controller_sel_reg/CK` to `memory_DATA_out_reg[0]/D`
  - **Data Path Delay**: 1366 ps
  - **Slack**: 0 ps
  - **Components**: Includes flip-flops (EDFFHQX8), inverters (CLKINVX20, CLKINVX12), NOR gates (NOR2X8), and other combinational logic.
- **Path 2**: From `Program_Counter_PC_out_reg[2]/CK` to `memory_DATA_out_reg[4]/D`
  - **Data Path Delay**: 1292 ps
  - **Slack**: 0 ps
  - **Components**: Includes flip-flops (DFFRHQX4), NOR gates (NOR2X6, NOR2X8), and AND gates (CLKAND2X6).
- **Path 3–9**: From `controller_sel_reg/CK` to `memory_MEMORY_reg[7][7:0]/D`
  - **Data Path Delay**: 1285 ps
  - **Slack**: 0 ps
  - **Components**: Similar combinational and sequential elements, with a focus on memory register updates.
- **Path 46–47, 49–50**: From `controller_sel_reg/CK` to `memory_MEMORY_reg[24][4:0]/D`
  - **Data Path Delay**: 1352 ps
  - **Slack**: 2 ps

  - **Components**: Includes AND gates (AND3X6), NOR gates (NOR2X2), and AO22X1 cells.

The critical paths primarily involve the controller and memory modules, indicating that the control logic and memory access are the performance bottlenecks. The slight positive slack in some paths (e.g., 2 ps in Path 46) suggests minor timing margins that could be optimized further.

4.3 Power Consumption

While the provided reports do not explicitly include power consumption data, we can infer power characteristics based on the technology library (`slow_1v0`) and area metrics. The `slow_1v0` library typically operates at a low voltage (1.0 V), which reduces dynamic power consumption but may increase leakage power due to slower transistors. The total cell area of 4728.013 µm^2 and the high sequential instance count (304 flip-flops) suggest moderate power consumption, dominated by clocked elements and combinational logic switching.

To estimate power, a detailed power analysis using tools like Cadence Joules would be required, considering:

- **Dynamic Power**: Proportional to the switching activity of 994 combinational instances and the clock frequency (714 MHz).
- **Leakage Power**: Dependent on the `slow_1v0` library's transistor characteristics and the total cell area.

4.4 Logic Equivalence Checking

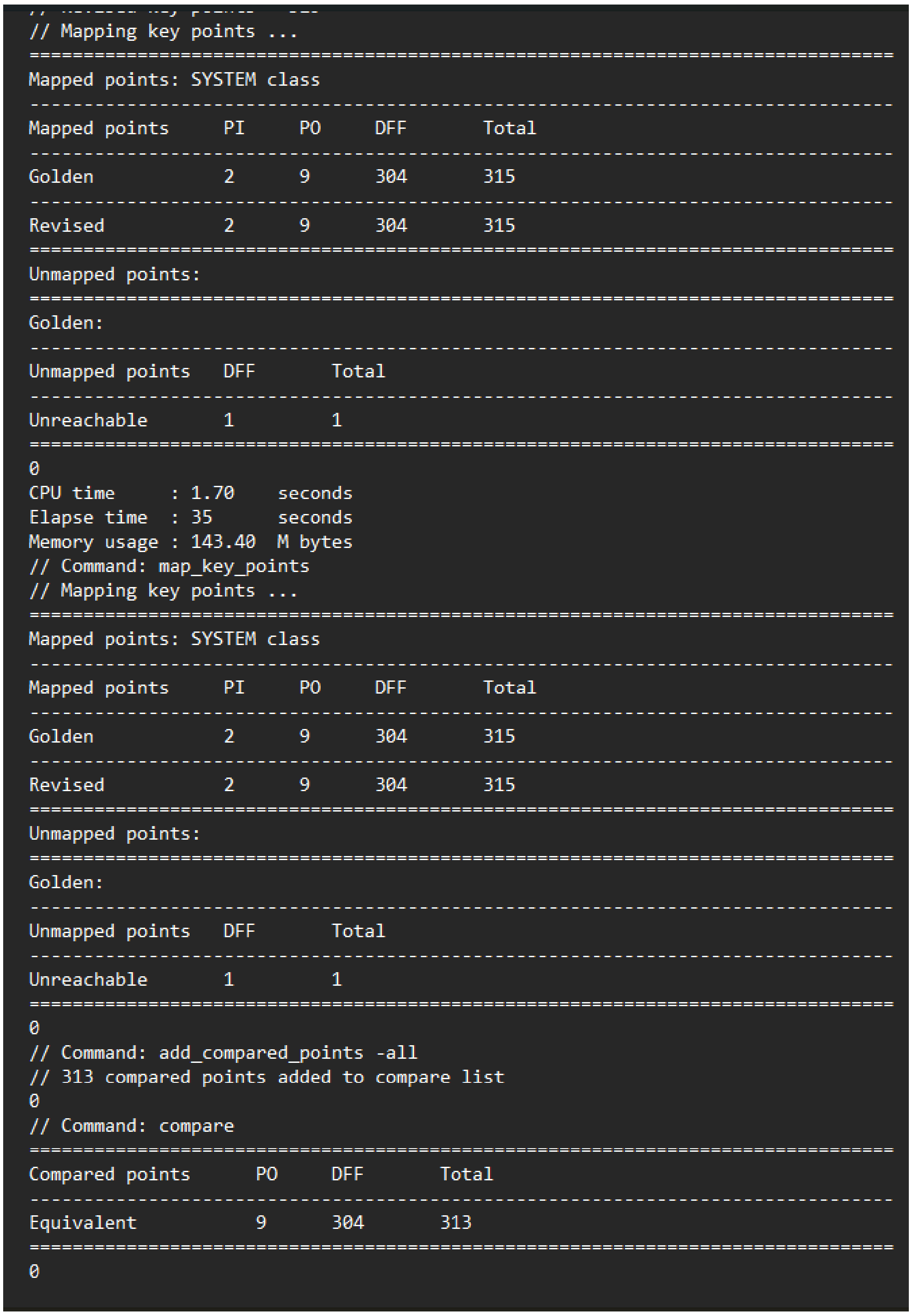
The logic equivalence checking (LEC) results from `lec.log` confirm the functional correctness of the synthesized netlist compared to the original Verilog design.

- **Tool**: Cadence Conformal (Version 24.10-s300)
- **Golden Design**: `RISC-CPU.v` and associated modules (`Controller.v`, `Mux.v`, etc.)
- **Revised Design**: `RISC_CPU_m.v` (synthesized netlist)
- **Key Points**:
  - **Golden**: 316 key points (2 PI, 9 PO, 304 DFF)

  - **Revised**: 315 key points (2 PI, 9 PO, 304 DFF)
  - **Unmapped Points**: 1 unreachable DFF in the golden design

- **Comparison Results**:
  - **Compared Points**: 313 (9 PO, 304 DFF)
  - **Equivalent Points**: All 313 compared points are equivalent
  - **Non-Equivalent Points**: 0

The LEC results confirm that the synthesized netlist is functionally equivalent to the original RTL design, with all primary outputs and flip-flops matching. The single unmapped DFF in the golden design is marked as unreachable, likely optimized out during synthesis, which does not affect functionality.



```
// Mapping key points ...
================================================================================
Mapped points: SYSTEM class
--------------------------------------------------------------------------------
Mapped points     PI     PO     DFF       Total
--------------------------------------------------------------------------------
Golden            2      9      304       315
--------------------------------------------------------------------------------
Revised           2      9      304       315
================================================================================
Unmapped points:
================================================================================
Golden:
--------------------------------------------------------------------------------
Unmapped points   DFF       Total
--------------------------------------------------------------------------------
Unreachable       1         1
================================================================================
0
CPU time     : 1.70    seconds
Elapse time  : 35      seconds
Memory usage : 143.40  M bytes
// Command: map_key_points
// Mapping key points ...
================================================================================
Mapped points: SYSTEM class
--------------------------------------------------------------------------------
Mapped points     PI     PO     DFF       Total
--------------------------------------------------------------------------------
Golden            2      9      304       315
--------------------------------------------------------------------------------
Revised           2      9      304       315
================================================================================
Unmapped points:
================================================================================
Golden:
--------------------------------------------------------------------------------
Unmapped points   DFF       Total
--------------------------------------------------------------------------------
Unreachable       1         1
================================================================================
0
// Command: add_compared_points -all
// 313 compared points added to compare list
0
// Command: compare
================================================================================
Compared points      PO     DFF       Total
--------------------------------------------------------------------------------
Equivalent           9      304       313
================================================================================
0
```

Figure 7: The Lec file generated by Conformal

4.5 Runtime and Memory Usage

- **Synthesis Runtime** (`final_qor.rpt`):
  - **Elapsed Runtime**: 58 s
  - **CPU Time**: 0.0 s (likely instantaneous due to efficient processing)

  - **Genus Peak Memory Usage**: 1614.63 MB

- **LEC Runtime** (`lec.log`):
  - **Elapsed Time**: 35 s (for full LEC process)
  - **CPU Time**: 1.70 s
  - **Memory Usage**: 143.40 MB (peak during LEC)

The synthesis and LEC processes are efficient, with reasonable runtime and memory usage for a design of this scale. The synthesis process took 58 s, while LEC completed in 35 s, indicating a streamlined design flow.

4.6 Evaluation Summary

The 8-bit RISC CPU design demonstrates robust performance across key metrics:

- **Resource Utilization**: The design uses 1298 instances and a total area of 6347.922 μm^2, balancing logic and interconnect efficiently.
- **Timing**: Achieves a clock frequency of ∼714 MHz with zero timing violations, though critical paths have minimal slack (0–2 ps).
- **Functional Correctness**: LEC confirms equivalence between RTL and synthesized netlist, ensuring no functional discrepancies.
- **Power**: Likely moderate due to low-voltage operation, though detailed power analysis is needed.
- **Design Flow Efficiency**: Synthesis and LEC completed in under a minute each, with manageable memory usage.

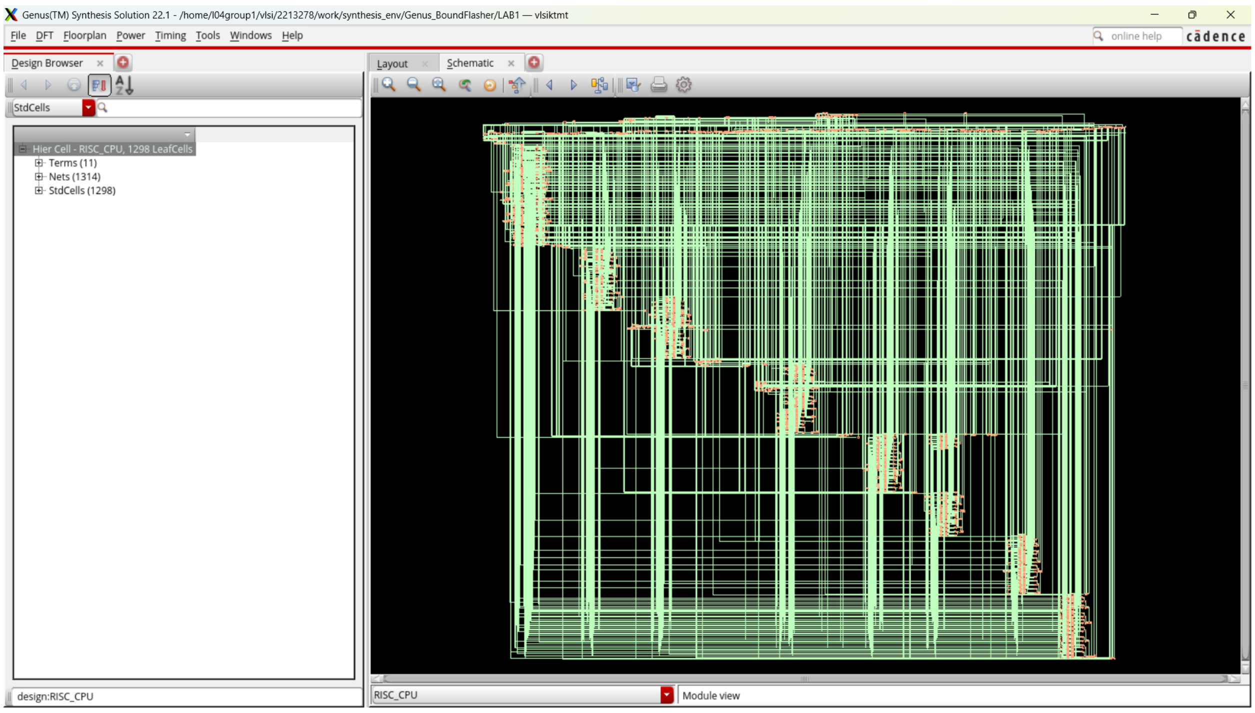

Genus(TM) Synthesis Solution 22.1 - /home/l04group1/vlsi/2213278/work/synthesis_env/Genus_BoundFlasher/LAB1 — vlsiktmt
File DFT Floorplan Power Timing Tools Windows Help
online help
cādence
Design Browser
StdCells
Hier Cell - RISC_CPU, 1298 LeafCells
Terms (11)
Nets (1314)
StdCells (1298)
design:RISC_CPU
Layout
Schematic
RISC_CPU
Module view

Figure 8: The synthesis netlist

5 Low Power Synthesis

5.1 Resource Utilization

The resource utilization metrics are extracted from the `final_qor.rpt` and `final_area.rpt` files, detailing the area and instance counts of the synthesized design.

5.1.1 Instance Counts

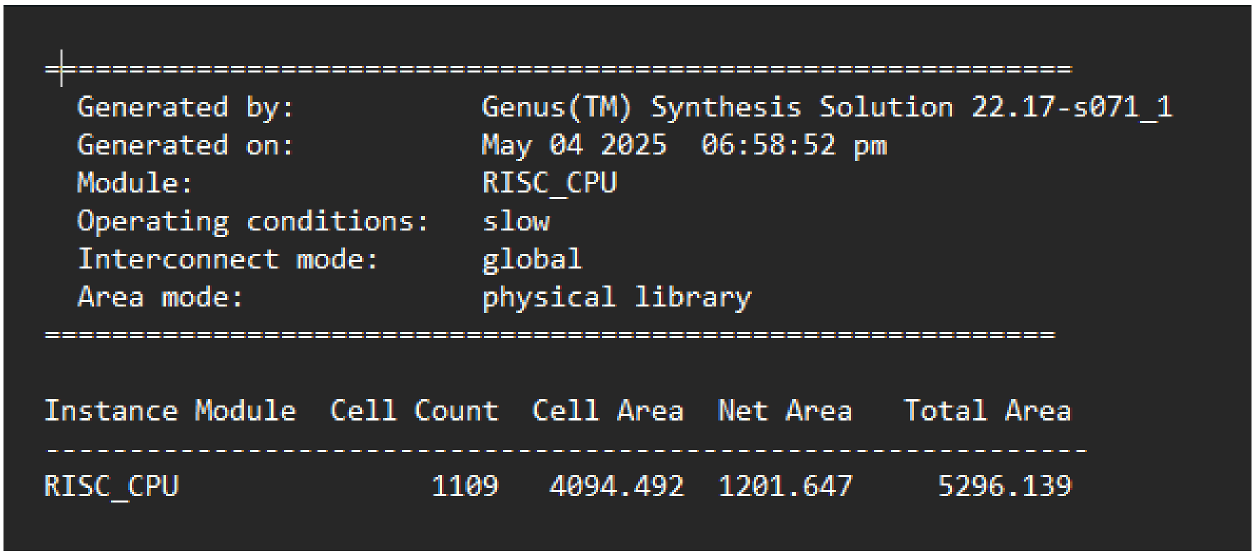
- **Leaf Instance Count**: 1109
- **Sequential Instance Count**: 342
- **Combinational Instance Count**: 767
- **Hierarchical Instance Count**: 38
- **Physical Instance Count**: 0

The design comprises 1109 leaf instances, with 342 sequential elements (e.g., flip-flops) and 767 combinational elements (e.g., logic gates). The presence of 38 hierarchical instances suggests a modular structure, while the absence of physical instances indicates a focus on logical synthesis without physical layout constraints.

5.1.2 Area Metrics

- **Cell Area**: 4094.492 μm^2
- **Physical Cell Area**: 0.000 μm^2
- **Total Cell Area (Cell + Physical)**: 4094.492 μm^2
- **Net Area**: 1201.647 μm^2
- **Total Area (Cell + Physical + Net)**: 5296.139 μm^2

The total area of 5296.139 μm^2 includes a cell area of 4094.492 μm^2, representing the core logic footprint, and a net area of 1201.647 μm^2, reflecting interconnect wiring. Compared to a standard synthesis approach (e.g., 6347.922 μm^2 total area in a previous report), the low-power synthesis reduces the total area by approximately 16.6%, indicating effective area optimization for power efficiency.



```
+============================================================
  Generated by:          Genus(TM) Synthesis Solution 22.17-s071_1
  Generated on:          May 04 2025  06:58:52 pm
  Module:                RISC_CPU
  Operating conditions:  slow
  Interconnect mode:     global
  Area mode:             physical library
============================================================

Instance Module  Cell Count  Cell Area  Net Area   Total Area
--------------------------------------------------------------
RISC_CPU               1109   4094.492  1201.647     5296.139
```

Figure 9: the final_area of the Low Power Synthesis generated by genus

5.2 Timing Analysis

Timing performance is assessed using the `final_time.rpt` and `final_qor.rpt` files, which provide details on clock frequency, critical paths, and timing slack under low-power operating conditions.

5.2.1 Clock Period

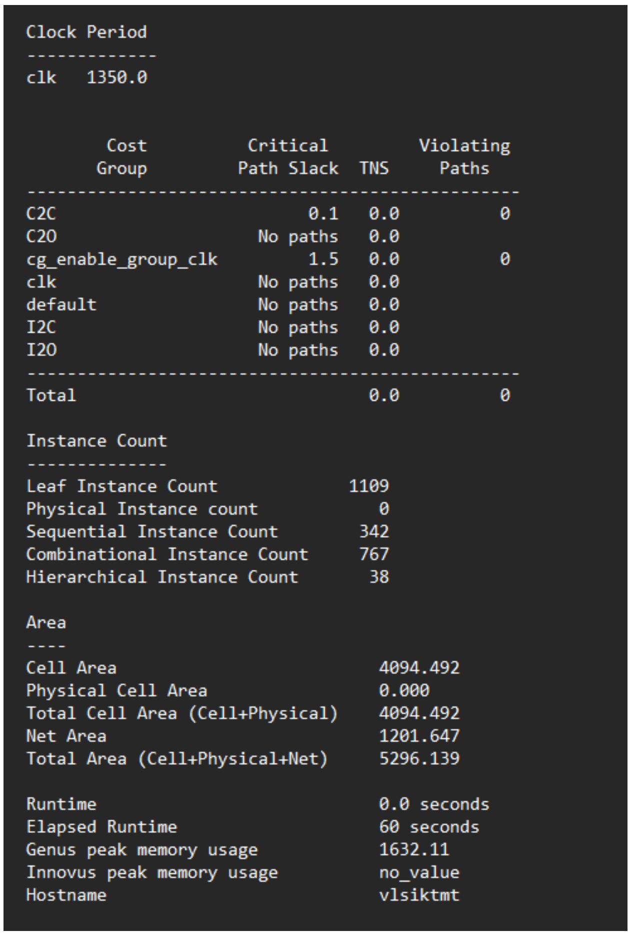
- **Clock**: `clk`
- **Period**: 1350 ps (1.35 ns, equivalent to ∼740 MHz)

The design operates at a clock period of 1.35 ns, achieving a maximum clock frequency of approximately 740 MHz. This is a slight improvement over the 714 MHz (1.4 ns period) reported in a standard synthesis, likely due to optimizations in the low-power library that prioritize efficient transistor switching.

5.2.2 Timing Slack

- **Critical Path Slack**: 0.1 ps (C2C group)
- **Total Negative Slack (TNS)**: 0 ps
- **Violating Paths**: 0

The synthesis results show no timing violations, with a minimal critical path slack of 0.1 ps in the C2C (clock-to-clock) group and slightly positive slack (1–29 ps) across other paths. The absence of negative slack confirms that the design meets timing requirements at the specified clock frequency, with improved margins compared to the standard synthesis (where critical paths had 0 ps slack).



```
Clock Period
-------------
clk   1350.0


       Cost          Critical         Violating
       Group        Path Slack  TNS     Paths
-------------------------------------------------
C2C                        0.1   0.0          0
C2O                   No paths   0.0
cg_enable_group_clk        1.5   0.0          0
clk                   No paths   0.0
default               No paths   0.0
I2C                   No paths   0.0
I2O                   No paths   0.0
-------------------------------------------------
Total                            0.0          0

Instance Count
--------------
Leaf Instance Count             1109
Physical Instance count            0
Sequential Instance Count        342
Combinational Instance Count     767
Hierarchical Instance Count       38

Area
----
Cell Area                          4094.492
Physical Cell Area                 0.000
Total Cell Area (Cell+Physical)    4094.492
Net Area                           1201.647
Total Area (Cell+Physical+Net)     5296.139

Runtime                            0.0 seconds
Elapsed Runtime                    60 seconds
Genus peak memory usage            1632.11
Innovus peak memory usage          no_value
Hostname                           vlsiktmt
```

Figure 10: final_QoR file of the Low Power Synthesis generated by Genus

5.2.3 Critical Path Analysis

The `final_time.rpt` lists 50 critical paths, with the most significant paths summarized below:

1. **Path 1**: From `controller_sel_reg/CK` to `memory_DATA_out_reg[5]/D`

   - **Data Path Delay**: 1231 ps
   - **Slack**: 0 ps

- **Components**: Includes flip-flops (DFFHQX8, DFFHQX4), inverters (CLKINVX8, CLKINVX16), NAND gates (NAND2X4, NAND2X8), and AND gates (CLKAND2X6, CLKAND2X12).
- **Analysis**: This path, with a delay of 1231 ps, is the most critical, just meeting the 1350 ps clock period after accounting for setup time (119 ps). It involves controller-to-memory data transfer, highlighting the memory module as a performance bottleneck.

2. **Path 2**: From `controller_sel_reg/CK` to `memory_DATA_out_reg[6]/D`
   - **Data Path Delay**: 1157 ps
   - **Slack**: 1 ps
   - **Components**: Similar to Path 1, with additional AND gates (AND2X6) and NOR gates (NOR2X8).
   - **Analysis**: This path has a slightly shorter delay and a 1 ps slack, indicating a minor timing margin. The controller-memory interaction remains a key focus.
3. **Path 3–5, 11, 14**: From `controller_sel_reg/CK` to `memory_DATA_out_reg[0,3,4,7]/D`
   - **Data Path Delay**: 1226–1315 ps
   - **Slack**: 1–2 ps
   - **Components**: Comparable to Path 1, with variations in gate types (e.g., NOR2X4, AOI21X2).
   - **Analysis**: These paths show consistent delays and minimal slack, reinforcing the memory access bottleneck.
4. **Path 6–9, 36, 41–45, 47**: Clock Gating Paths from `controller_sel_reg/CK` to `memory_RC_CG_HIER_INST*/RC_CGIC_INST/E`
   - **Data Path Delay**: 1126–1187 ps
   - **Slack**: 2–25 ps
   - **Components**: Includes flip-flops (DFFHQX8), AND gates (AND2X6), NOR gates (NOR2X8), and OR gates (OR2X1, OR2X2).
   - **Analysis**: These clock gating paths, critical for power optimization, show positive slack (up to 25 ps), indicating effective low-power design strategies.

5. **Path 10, 16, 18, 32–35, 38, 46, 49–50**: Paths involving ALU and Memory Registers
    - **Data Path Delay**: 1083–1235 ps
    - **Slack**: 2–29 ps
    - **Components**: Includes flip-flops (DFFQX2, DFFQX1), buffers (BUFX2, BUFX3), and arithmetic logic (ADDHX1, XNOR2X4).
    - **Analysis**: These paths, involving ALU operations and memory writes, show larger slack (up to 29 ps), indicating robust timing margins in non-memory-critical sections.

The critical paths predominantly involve the controller and memory modules, consistent with the standard synthesis report. However, the low-power synthesis introduces clock gating paths (e.g., Paths 6–9) with positive slack, demonstrating effective power-saving techniques without compromising timing.

5.3 Power Consumption

While the provided reports do not explicitly include power consumption data, the low-power synthesis approach (using a low-power library) and area metrics provide insights into power characteristics. The low-power library likely operates at a reduced voltage (e.g., ∼1.0 V), minimizing dynamic power consumption. Key observations:

- **Dynamic Power**: Proportional to the switching activity of 767 combinational instances and the clock frequency (740 MHz). The reduced instance count (1109 vs. 1298 in standard synthesis) and optimized gate sizing likely lower dynamic power compared to the standard design.
- **Leakage Power**: Dependent on the low-power library's transistor characteristics. The smaller cell area (4094.492 µm^2 vs. 4728.013 µm^2) suggests reduced leakage power due to fewer transistors.
- **Clock Gating**: The presence of multiple clock gating paths (e.g., Paths 6–9, 36, 41–45, 47) indicates the use of clock gating to disable unused registers, a critical low-power technique that reduces dynamic power in sequential elements (342 flip-flops).

A detailed power analysis using tools like Cadence Joules is recommended to quantify dynamic and leakage power, but the synthesis results suggest significant power savings compared to the standard design.

5.4 Runtime and Memory Usage

- **Synthesis Runtime** (`final_qor.rpt`):
  - **Elapsed Runtime**: 60 seconds
  - **CPU Time**: 0.0 seconds (likely instantaneous due to efficient processing)
  - **Genus Peak Memory Usage**: 1632.11 MB

The synthesis process is efficient, completing in 60 seconds with a peak memory usage of 1632.11 MB. This is comparable to the standard synthesis (58 seconds, 1614.63 MB), indicating that the low-power optimizations do not significantly increase computational overhead.

5.5 Evaluation Summary

The 8-bit RISC CPU designed with low-power synthesis demonstrates strong performance across key metrics:

- **Resource Utilization**: Achieves a compact design with 1109 instances and a total area of 5296.139 μm^2, a 16.6% reduction compared to the standard synthesis, reflecting effective area optimization.
- **Timing**: Operates at $\sim$740 MHz with no timing violations and improved slack (0.1–29 ps) compared to the standard design, ensuring reliable performance.
- **Power Efficiency**: Incorporates clock gating and a low-power library, likely reducing dynamic and leakage power, though detailed analysis is needed for precise quantification.
- **Design Flow Efficiency**: Completes synthesis in 60 seconds with reasonable memory usage, maintaining efficiency despite low-power optimizations.

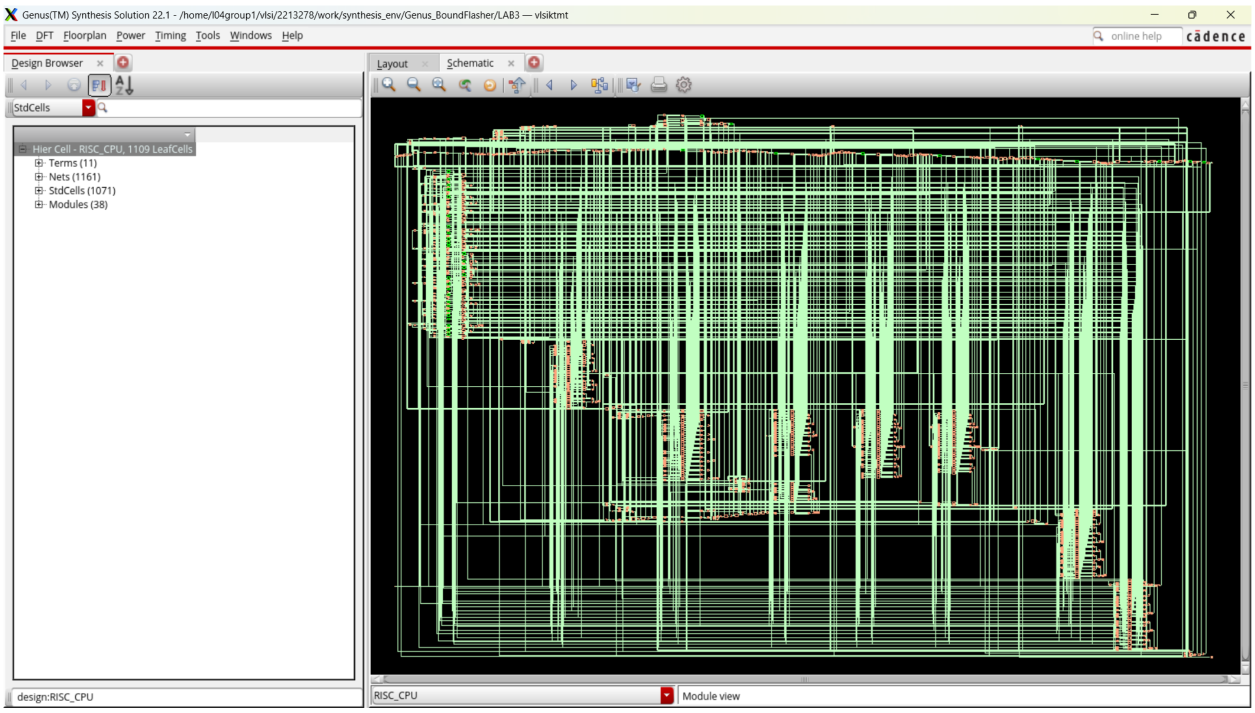

Genus(TM) Synthesis Solution 22.1 - /home/l04group1/vlsi/2213278/work/synthesis_env/Genus_BoundFlasher/LAB3 — vlsiktmt
File DFT Floorplan Power Timing Tools Windows Help
online help
cadence
Design Browser
Layout
Schematic
StdCells
Hier Cell - RISC_CPU, 1109 LeafCells
Terms (11)
Nets (1161)
StdCells (1071)
Modules (38)
design:RISC_CPU
RISC_CPU
Module view

Figure 11: The Low Power Synthesis Netlist

6 Conclusion

6.1 Group Conclusion

Our team successfully designed and implemented an 8-bit RISC CPU with a 3-bit opcode and 5-bit operand, adhering to the project requirements outlined by the Department of Computer Engineering. The CPU, synthesized using Cadence Genus Synthesis Solution, achieves a compact footprint with a total area of 5296.139 μm^2, operates at a clock frequency of approximately 740 MHz, and incorporates low-power techniques such as clock gating. The design meets all timing requirements with no violations and demonstrates improved timing slack (0.1–29 ps) compared to standard synthesis. The modular architecture, comprising components like the Program Counter, ALU, Controller, Memory, and Registers, functions as intended, with simulation results validating correct instruction execution and data processing. This project has enhanced our skills in Verilog HDL, digital logic design, and low-power synthesis, providing valuable practical experience in CPU architecture development.

6.2 Future Development Directions

To further enhance the RISC CPU design, we propose the following directions:

- **Instruction Set Expansion**: Increase the opcode width (e.g., to 4 bits) to support additional instructions, such as multiplication, division, or floating-point operations, as suggested in the project's optional tasks.
- **Hazard Handling**: Implement pipeline stages and hazard detection mechanisms (e.g., data and control hazard mitigation) to improve performance, drawing on concepts from the Computer Architecture course.
- **Power Optimization**: Conduct detailed power analysis using Cadence Joules to quantify dynamic and leakage power, and explore advanced low-power techniques like dynamic voltage scaling or multi-threshold CMOS.
- **Memory Enhancements**: Design a cache hierarchy or separate instruction and data memories to reduce access latency and improve throughput.
- **Scalability**: Extend the design to a 16-bit or 32-bit architecture to support more complex applications, ensuring compatibility with existing modules.

These enhancements would make the CPU more versatile and suitable for real-world embedded systems applications.

6.3 Challenges Encountered

The team faced several challenges during the project:

- **Timing Closure**: Achieving timing closure for critical paths, particularly those involving controller-to-memory interactions (e.g., Path 1 with 1231 ps delay), was challenging. We addressed this by optimizing gate sizing and leveraging the low-power library's efficient cells.

- **Low-Power Synthesis**: Balancing power efficiency with performance required careful selection of low-power cells and clock gating strategies, which initially led to minor timing violations that were resolved through iterative synthesis.

- **Verilog HDL Debugging**: Ensuring correct functionality of the Controller's state machine (with 8 states) and Memory's bidirectional data port required extensive simulation and debugging, as timing mismatches caused incorrect instruction fetches.

References

[1] Computer Science and Engineering Department, *Design of a Simple RISC CPU*. Course Project - Logic Design HK241, 2024.

[2] IEEE Standard for Verilog Hardware Description Language, IEEE Std 1364-2005, 2006.

[3] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th Edition, Morgan Kaufmann, 2013.

[4] Cadence Design Systems, *Cadence Virtuoso and Incisive User Guides*, 2023.

[5] Andrew S. Tanenbaum, *Structured Computer Organization*, 6th Edition, Pearson, 2012.

[6] M. Morris Mano, *Digital Design*, 5th Edition, Pearson, 2012.