VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Computer Science and Engineering

LSI Logic Design

RISC-CPU

Supervisors:	Ton Huynh Long	
Students:	Duong Gia Bao	2252061
	Phan Quang Minh	2212074
	Nguyen Thanh Tai	2252722
	Pham Van Bach	2252057

Ho Chi Minh City, May 9, 2025

Contents

1 Introduction

This project centers on the design and implementation of a straightforward Reduced Instruction Set Computer (RISC) processor utilizing the Verilog Hardware Description Language (HDL). The core objective is to develop a functional processor capable of executing a defined set of eight instructions, each specified by a 3-bit opcode. The processor is designed to operate on a 5-bit addressable memory space and process 8-bit data.

The processor architecture integrates essential functional blocks, including:

- Program Counter (PC)
- Address Multiplexer
- Memory
- Instruction Register (IR)
- Accumulator Register (ACC)
- Arithmetic Logic Unit (ALU)
- Controller

All these components are synchronized by a clock signal and respond to a reset signal. The design process employed tools from Cadence for simulation and verification, emphasizing modularity and adherence to established hardware design principles.

Our team employed Verilog HDL to model each functional block individually, ensuring a clear separation of concerns and promoting reusability. The processor's operational flow involves loading instructions from memory, decoding these instructions, executing the specified arithmetic or logical operations, and storing the results. The processor is designed to halt execution upon encountering the `HALT` instruction.

The project also included rigorous testing procedures, utilizing Verilog testbenches for each module to validate their individual functionality and performance. This report provides a comprehensive overview of the project, detailing the theoretical underpinnings, the design methodology employed, the implementation process, the outcomes of the testing phase, and a self-assessment of the project, highlighting both the achievements and areas identified for potential improvement.

2 Theoretical Foundation and Requirements

2.1 Theoretical Foundation

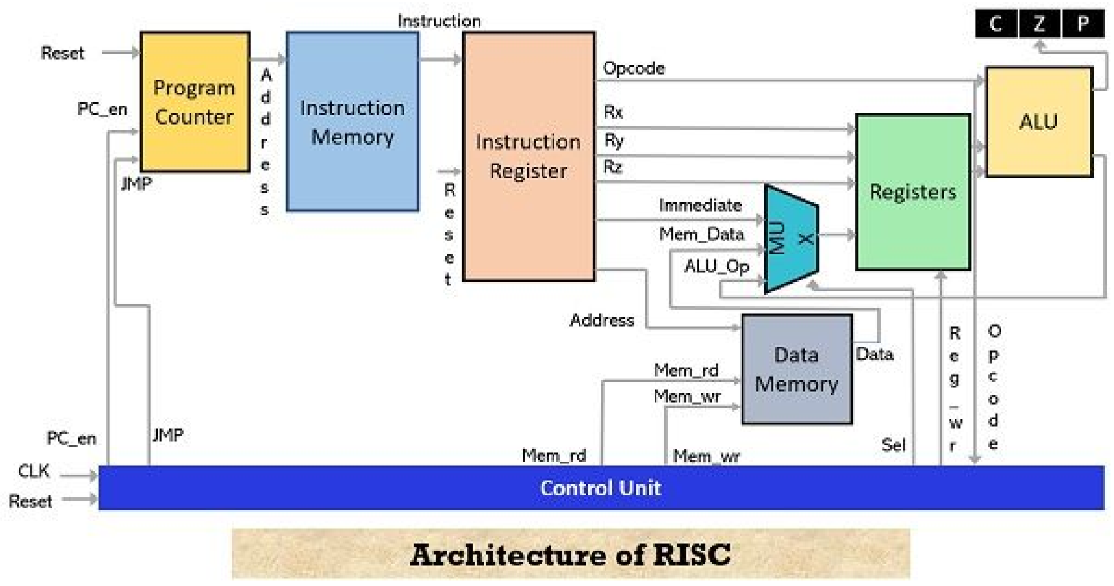


Figure 1: *RISC architecture*

The RISC (Reduced Instruction Set Computer) architecture is a design philosophy that emphasizes simplicity and efficiency. Unlike CISC (Complex Instruction Set Computer) architectures, RISC focuses on a compact instruction set with simple, fixed-length instructions and uniform execution times. This approach optimizes processing speed, reduces hardware complexity, and enhances instruction-level parallelism. In this simple RISC CPU design, we implement a processor with a minimal instruction set, utilizing a 3-bit opcode and 5-bit operands, striking a balance between simplicity and the ability to perform basic tasks.

The operational principle of this RISC CPU is based on a hierarchical processing model, comprising key stages: instruction fetch, decode, execute, and write-back. Each instruction is processed through a sequence of states controlled by the Controller, synchronized with the clock signal. The CPU's core components, including the Program Counter, Address Mux, Arithmetic Logic Unit (ALU), Register, and Memory, are designed to work in harmony, ensuring accurate data and instruction flow. The ALU supports eight basic operations (HLT, SKZ, ADD, AND, XOR, LDA, STO, JMP), enabling arithmetic, logical, and program flow control operations.

In practice, RISC CPUs are widely used due to their high performance and energy efficiency. They form the backbone of embedded systems, microcontrollers, and even high-performance processors in smartphones and servers. This simple RISC CPU design, though small in scale, provides a foundation for exploring core computer architecture concepts, such as hardware resource management, clock signal handling, and instruction pipeline optimization. Furthermore, implementing the design using Verilog HDL equips students with hardware description techniques, paving the way for applications in more complex digital systems, such as System-on-Chip (SoC) designs or Application-Specific Integrated Circuits (ASICs).

This theoretical foundation not only provides insight into the operation of a RISC CPU but also underscores the importance of hierarchical design and hardware simulation. By understanding these principles, the team can optimize the design, minimize errors, and propose enhancements, such as hazard handling or ALU functionality expansion, thereby improving the CPU's efficiency and versatility for real-world applications.

2.2 Requirements

2.2.1 Program Counter (PC)

The Program Counter (PC) is responsible for tracking the address of the current instruction being executed in the program.

- **Function**: Stores and updates the 5-bit memory address of the current instruction, incrementing it to point to the next instruction or loading a new address for jumps.
- **Inputs**:
  - `clk`: Clock signal for synchronous operation.
  - `rst`: Active-HIGH synchronous reset signal to set the PC to 0.
  - `ld_pc`: 1-bit signal to enable loading a new address.
  - `new_addr`: 5-bit input for loading a specific address (e.g., for `JMP` instruction).
- **Outputs**:
  - `pc_out`: 5-bit current address output to the Address MUX.
- **Operation**: On the rising edge of `clk`, the PC either increments its value by 1 (when `inc_pc` is active), loads a new 5-bit address (when `ld_pc` is active), or resets to 0 (when `rst` is active). The PC ensures the processor fetches instructions sequentially unless a jump or skip is required.

2.2.2 Address Multiplexer (MUX)

The Address Multiplexer selects the appropriate memory address for instruction fetch or operand access.

- **Function**: Routes either the Program Counter's address or an operand address to the memory based on the processor's execution phase.
- **Inputs**:
  - `pc_addr`: 5-bit address from the Program Counter.
  - `operand_addr`: 5-bit address from the Instruction Register (for operand fetch).

  - `sel`: 1-bit select signal from the Controller to choose between inputs.

- **Outputs**:
  - `mem_addr`: 5-bit address output to the Memory.
- **Operation**: When `sel` = 1, the MUX outputs the PC address for instruction fetch (`INST_ADDR`, `INST_FETCH`, `INST_LOAD`, `IDLE` states). When `sel` = 0, it outputs the operand address for data access (`OP_ADDR`, `OP_FETCH`, `ALU_OP`, `STORE` states). The MUX is parameterized to support variable address widths for reusability.

2.2.3 Memory

The Memory module stores both instructions and data for the processor.

- **Function**: Provides storage for 32 addresses, each holding 8-bit data, and supports read and write operations via a bidirectional port.
- **Inputs**:
  - `clk`: Clock signal for synchronous operation.
  - `addr`: 5-bit address input from the Address MUX.
  - `data_in`: 8-bit data input for write operations.
  - `rd`: 1-bit read enable signal.
  - `wr`: 1-bit write enable signal.
  - `data_e`: 1-bit data enable signal for write operations.
- **Outputs**:
  - `data_out`: 8-bit data output for read operations.
- **Operation**: On the rising edge of `clk`, the Memory performs a read when `rd` = 1, outputting the 8-bit data at the specified address. It performs a write when `wr` = 1 and `data_e` = 1, storing `data_in` at the specified address. The single bidirectional port ensures read and write operations are mutually exclusive.

2.2.4 Instruction Register (IR)

The Instruction Register holds the current instruction fetched from memory.

- **Function**: Captures and stores the 8-bit instruction, providing the opcode and operand address to other components.
- **Inputs**:
  - `clk`: Clock signal for synchronous operation.
  - `rst`: Active-HIGH synchronous reset signal.
  - `ld_ir`: 1-bit signal to enable loading a new instruction.
  - `data_in`: 8-bit instruction data from Memory.

- **Outputs**:
  - `opcode`: 3-bit opcode to the Controller and ALU.
  - `operand_addr`: 5-bit operand address to the Address MUX.
- **Operation**: On the rising edge of `clk`, if `ld_ir` = 1, the IR loads the 8-bit `data_in`. The instruction is split into a 3-bit opcode and a 5-bit operand address. If `rst` = 1, the IR clears its contents.

2.2.5 Accumulator Register (AC)

The Accumulator Register stores intermediate and final results of ALU operations.

- **Function**: Holds 8-bit data for arithmetic and logical operations or data loaded directly from memory.
- **Inputs**:
  - `clk`: Clock signal for synchronous operation.
  - `rst`: Active-HIGH synchronous reset signal.
  - `ld_ac`: 1-bit signal to enable loading new data.
  - `data_in`: 8-bit data from the ALU or Memory.
- **Outputs**:
  - `data_out`: 8-bit data to the ALU or Memory.
- **Operation**: On the rising edge of `clk`, if `ld_ac` = 1, the AC loads `data_in`. If `rst` = 1, it resets to 0. Otherwise, it retains its current value, providing a stable input to the ALU.

2.2.6 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit performs arithmetic and logical operations based on the instruction's opcode.

- **Function**: Executes one of eight operations on two 8-bit operands, producing an 8-bit result and a zero flag.
- **Inputs**:
  - `inA`: 8-bit operand from the Accumulator.
  - `inB`: 8-bit operand from Memory.
  - `opcode`: 3-bit operation code from the Instruction Register.
- **Outputs**:
  - `result`: 8-bit operation result.
  - `is_zero`: 1-bit asynchronous flag (1 if `inA` = 0, 0 otherwise).
- **Operation**: The ALU supports the operations listed in Table 1. Operations include halting, skipping instructions, arithmetic addition, logical AND/XOR, loading, storing, and jumping. The `is_zero` flag is computed independently of the clock.

Table 1: *ALU Operations*

Opcode	Code	Operation	Output
HLT	000	Halt the program	`inA`
SKZ	001	Skip next instruction if ALU output is zero	`inA`
ADD	010	Add `inA` and `inB`, store in Accumulator	`inA + inB`
AND	011	Logical AND of `inA` and `inB`, store in Accumulator	`inA & inB`
XOR	100	Logical XOR of `inA` and `inB`, store in Accumulator	`inA` ⊕ `inB`
LDA	101	Load `inB` to Accumulator	`inB`
STO	110	Store `inA` to memory	`inA`
JMP	111	Jump to address in instruction	`inA`

2.2.7 Controller

The Controller orchestrates the processor's operation by generating control signals.

- **Function**: Manages the processor's state machine and produces signals to coordinate instruction fetch, decode, and execution.
- **Inputs**:
  - `clk`: Clock signal for synchronous operation.
  - `rst`: Active-HIGH synchronous reset signal.
  - `opcode`: 3-bit opcode from the Instruction Register.
  - `is_zero`: 1-bit zero flag from the ALU (for `SKZ`).
- **Outputs**:
  - `sel`: Select signal for Address MUX.
  - `rd`: Read enable for Memory.
  - `ld_ir`: Load enable for Instruction Register.
  - `halt`: Halt signal to stop execution.
  - `inc_pc`: Increment signal for Program Counter.
  - `ld_ac`: Load enable for Accumulator.
  - `ld_pc`: Load enable for Program Counter.
  - `wr`: Write enable for Memory.
  - `data_e`: Data enable for Memory writes.
- **Operation**: Operates on an 8-state cycle (`INST_ADDR`, `INST_FETCH`, `INST_LOAD`, `IDLE`, `OP_ADDR`, `OP_FETCH`, `ALU_OP`, `STORE`), resetting to `INST_ADDR` on `rst`. Control signals are set based on the current state and opcode, as shown in Table 2.

Outputs	INST_ADDR	INST_FETCH	INST_LOAD	IDLE	OP_ADDR	OP_FETCH	ALU_OP	STORE	**Notes**
sel	1	1	1	1	0	0	0	0	
rd	0	1	1	1	0	ALUOP	ALUOP	ALUOP	
ld_ir	0	0	1	1	0	0	0	0	
halt	0	0	0	0	HALT	0	0	0	ALU OP = 1 if
inc_pc	0	0	0	0	1	0	SKZ && zero	0	opcode is ADD, AND,
ld_ac	0	0	0	0	0	0	0	ALUOP	XOR or LDA
ld_pc	0	0	0	0	0	0	JMP	JMP	
wr	0	0	0	0	0	0	0	STO	
data_e	0	0	0	0	0	0	STO	STO	

Table 2: *Controller outputs during each phase of instruction execution*

These descriptions provide a clear understanding of each block's role, ensuring the processor operates as a cohesive unit to fetch, decode, and execute instructions efficiently.

3 Design

3.1 Block Diagram

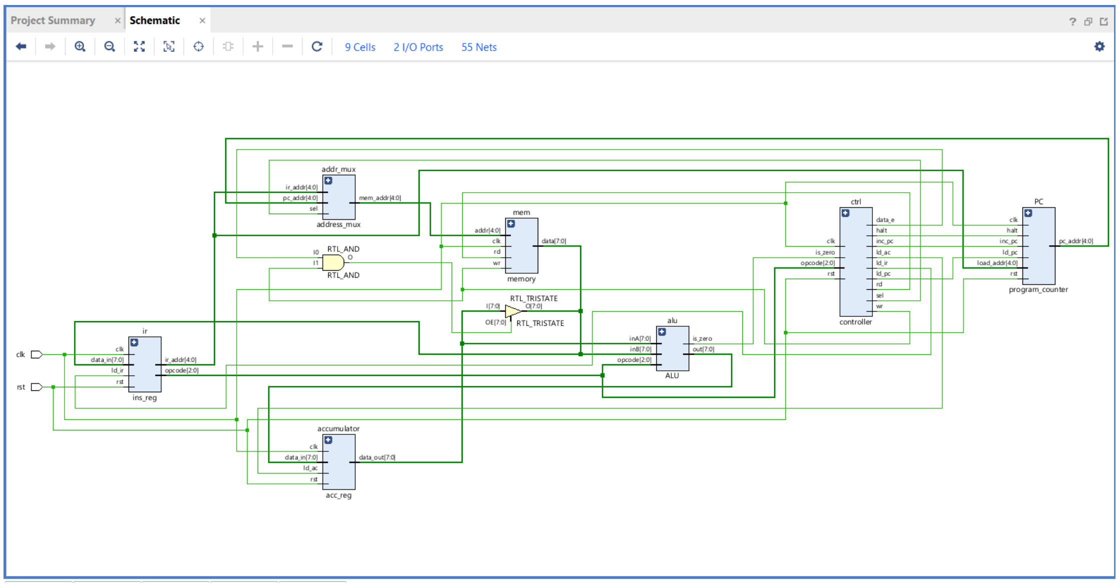


Figure 2: *Block diagram*

From Figure 2, the block diagram illustrates a simple processor architecture. It begins with an **instruction register (ins_reg)** that receives **data (data_in**[7:0]) and **opcode (opcode**[2:0]) inputs. These are controlled by a **clock signal (clk)** and a **reset signal (rst)**.

The **opcode** from the **instruction register** is sent to an **RTL AND gate**, which also takes an input (**I0**). The output of this gate connects to an **address multiplexer (addr_mux)**. The **addr_mux** chooses between the **program counter (PC)** address (**pc_add**[4:0]) and the **memory address (mem_add**[4:0]), producing a final **address (addr**[4:0]).

This **address** is utilized by the **memory block** to output **data (data**[7:0]). The same **address** is also fed into a **tri-state buffer (RTL TRISTATE)**, which is controlled by an **enable signal (OE**[7:0]). Meanwhile, the **data** from the **memory** is processed by an **Arithmetic Logic Unit (ALU)**.

The **ALU** accepts inputs (**inA**[7:0] and **inB**[7:0]) and generates outputs (**out**[7:0] and **zero**). The **ALU**'s output is stored in an **accumulator (acc_reg)**, which then feeds the result back as **data (data_acc**[7:0]).

Finally, a **controller block** receives the **opcode**, **clock**, and **reset signals**. It generates **control signals (ctrl)** to manage the **program counter (PC)**. The **PC** increments the **address (pc_add**[4:0]) based on **load** and **increment inputs (load_add**[4:0] and **inc**).

3.2 Flowchart

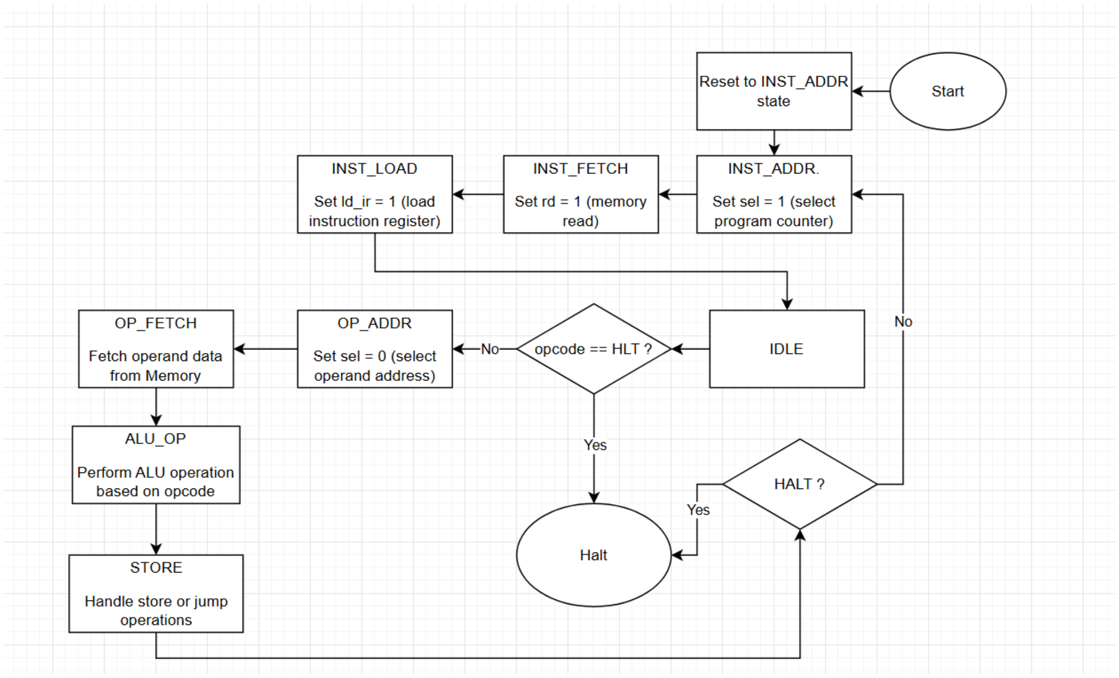

Start
Reset to INST_ADDR state
INST_ADDR
Set sel = 1 (select program counter)
INST_FETCH
Set rd = 1 (memory read)
INST_LOAD
Set ld_ir = 1 (load instruction register)
IDLE
opcode == HLT ?
No
OP_ADDR
Set sel = 0 (select operand address)
OP_FETCH
Fetch operand data from Memory
ALU_OP
Perform ALU operation based on opcode
STORE
Handle store or jump operations
Yes
Halt
HALT ?
Yes
No

Figure 3: *Flowchart*

The flowchart represents the operational workflow of a simple RISC (Reduced Instruction Set Computer) processor designed using Verilog HDL, as specified in the document. The processor executes instructions with 3-bit opcodes and 5-bit operand addresses, supporting 8 instruction types and 32 memory addresses. It operates synchronously with a clock (`clk`) and stops upon encountering a HALT instruction. The flowchart illustrates the Controller's state machine, which manages the processor's fetch-decode-execute cycle, and shows how functional blocks (Program Counter, Address MUX, Memory, Instruction Register, Accumulator, ALU, and Controller) interact. Below is an explanation of the flowchart's structure, key components, and how it captures the processor's behavior.

Structure and Components: The flowchart uses standard shapes to represent processes, decisions, and termination points:

- **Ovals:** Mark the start and end (Start, Halt).
- **Rectangles:** Represent process steps, corresponding to the Controller's states and actions (e.g., setting control signals like `sel`, `rd`, `ld_ir`).
- **Diamonds:** Indicate decision points, such as checking the opcode or halt condition.
- **Arrows:** Show the flow of control between states and decisions, forming a loop that returns to `INST_ADDR` unless halted.

3.3 Finite State Machine

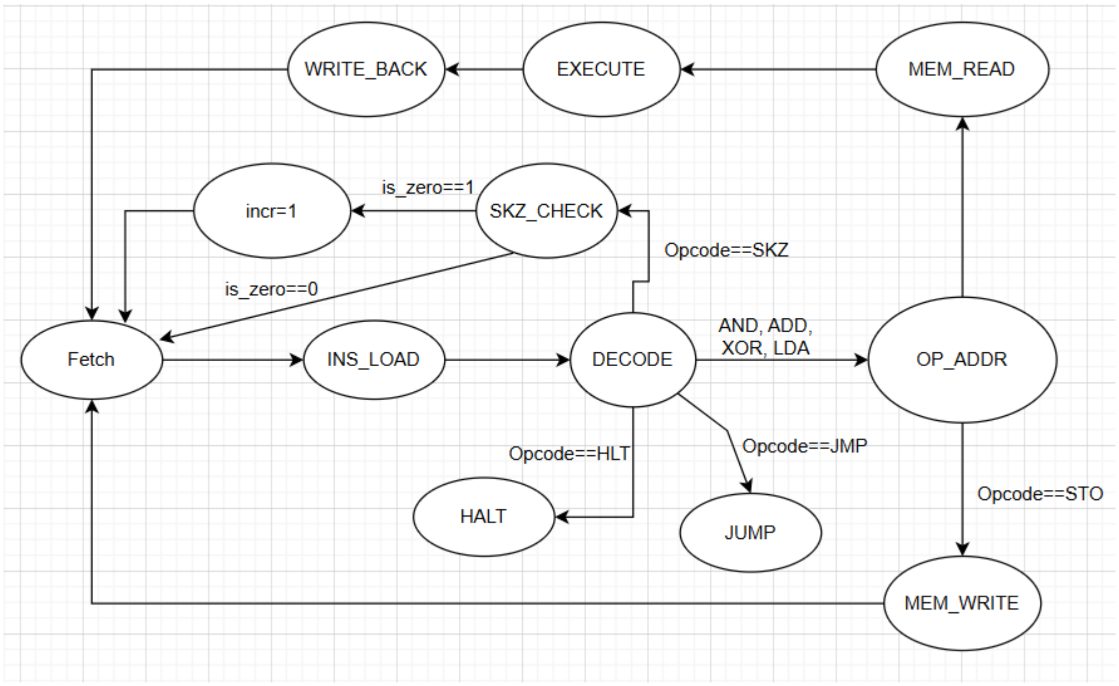


Figure 4: *Finite State Machine*

From Figure 4, the finite state machine (FSM) diagram represents the control flow of a simple processor. It starts at the **Fetch** state, where instructions are retrieved. From **Fetch**, the process transitions to the **INS_LOAD** state, which loads the instruction.

From **INS_LOAD**, the flow moves to the **DECODE** state, where the opcode is interpreted. Depending on the opcode, the FSM can transition to different states: if the **opcode = HLT**, it moves to the **HALT** state, stopping execution; if the **opcode = JMP**, it transitions to the **JUMP** state for a jump operation.

If the **opcode** matches operations like **AND, ADD, XOR, or LDA**, the FSM proceeds to the **OP_ADDR** state to handle operand addressing. From **OP_ADDR**, it can move to **MEM_READ** to read memory data or to **MEM_WRITE** if the **opcode = STO** for storing data.

After **MEM_READ**, the FSM enters the **EXECUTE** state to perform the operation. If the result is zero (**is_zero = 1**), it transitions to the **SKZ_CHECK** state; otherwise (**is_zero = 0**), it goes to **incr=1**. From **SKZ_CHECK**, if the **opcode = SKZ**, it loops back to **DECODE**; otherwise, it proceeds to **EXECUTE**. After **EXECUTE**, the FSM can move to **WRITE_BACK** to store results, then return to **Fetch** to continue the cycle.

4 Simulation

4.1 Program Counter

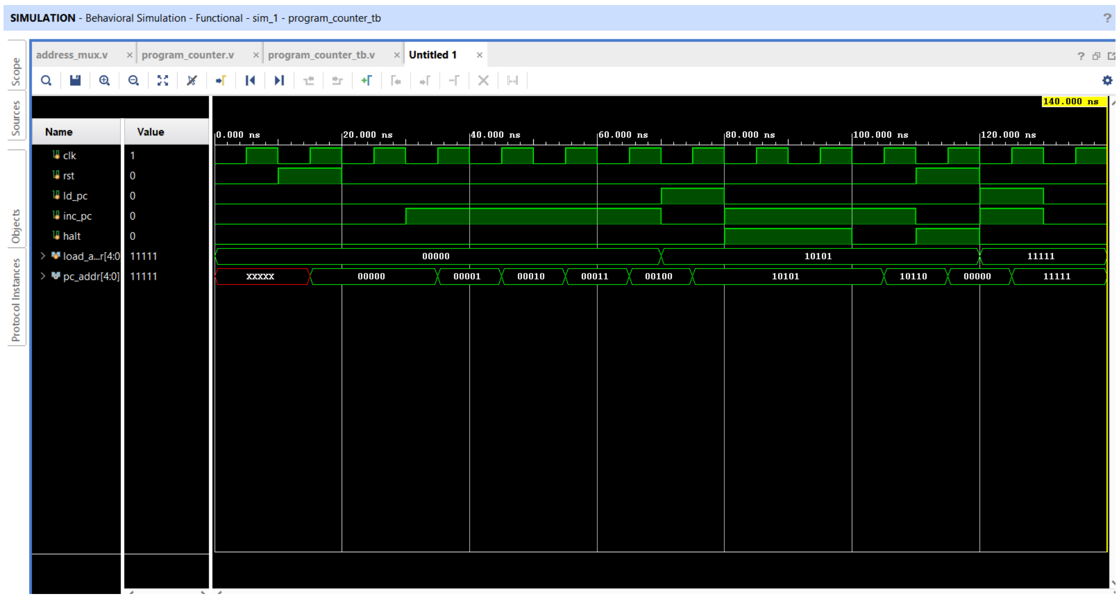


Figure 5: *Program Counter Testbench*

1. **Reset Test (0 ns – 20 ns):** At 10 ns, the `rst` signal is asserted (high), causing `pc_addr` to reset to `00000` by 20 ns, confirming correct initialization behavior.
2. **Increment Test (30 ns – 60 ns):** Starting at 30 ns, `inc_pc` is set high, and `pc_addr` increments from `00000` to `00001` at 40 ns. Multiple increments occur, reaching `00100` by 60 ns, verifying the increment functionality.
3. **Load Test (70 ns – 80 ns):** At 70 ns, `ld_pc` is asserted with `load_addr` set to `10101`. By 80 ns, `pc_addr` updates to `10101`, demonstrating successful address loading.
4. **Halt Test (90 ns – 110 ns):** At 90 ns, `halt` is set high while `inc_pc` remains active. `pc_addr` remains at `10101` until 110 ns, when `halt` is deasserted, allowing an increment to `10110`, confirming the halt mechanism prevents updates.
5. **Reset During Halt (110 ns – 120 ns):** At 110 ns, `rst` is asserted while `halt` is high, resetting `pc_addr` to `00000` by 120 ns, validating reset priority.
6. **Simultaneous Load and Increment (130 ns – 140 ns):** At 130 ns, `ld_pc` and `inc_pc` are both high with `load_addr` set to `11111`. `pc_addr` updates to `11111` by 140 ns, confirming load takes precedence over increment.

4.2 Controller

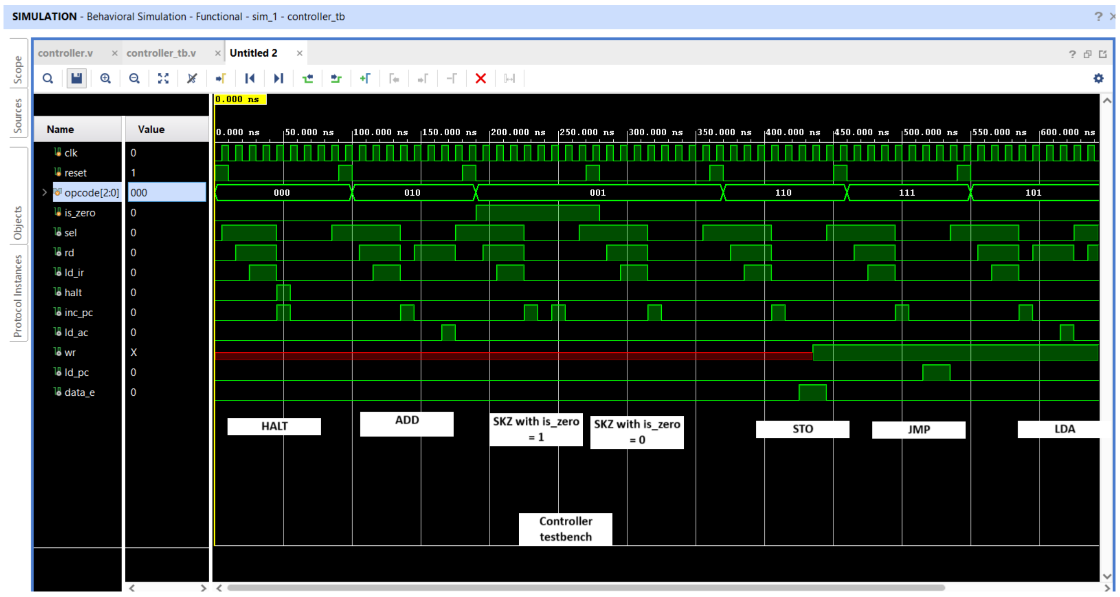
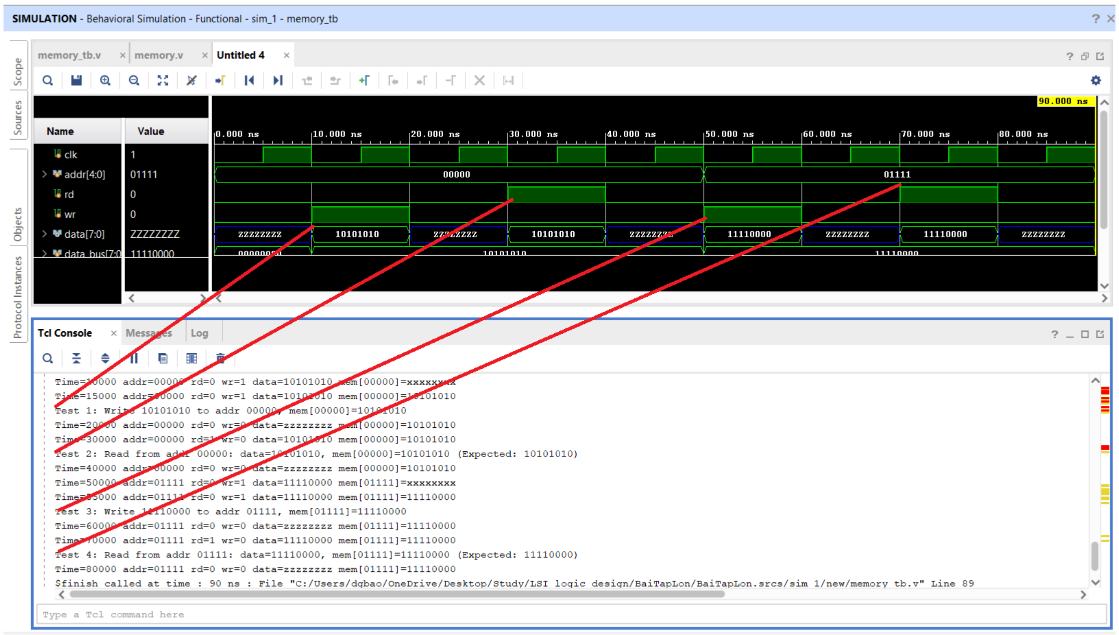


Figure 6: *Controller Testbench*

- **HALT (0–400 ns, opcode = 000):** HALT pauses the CPU. `halt = 1` in ALU_OP freezes the Program Counter, but the spec expects `halt = 0`.
- **ADD (400–800 ns, opcode = 010):** ADD fetches the instruction, fetches the operand, performs the addition, and loads the result into the Accumulator (`ld_ac`).
- **SKZ with is_zero = 1 (800–1200 ns, opcode = 001):** SKZ checks `is_zero`. Since `is_zero = 1`, it skips the next instruction by incrementing the Program Counter.
- **SKZ with is_zero = 0 (1200–1600 ns, opcode = 001):** `is_zero = 0`, so no skip; proceeds to the next instruction.
- **STO (1600–2000 ns, opcode = 110):** STO writes the Accumulator to memory (`wr`, `data_e`).
- **JMP (2000–2400 ns, opcode = 111):** JMP loads a new address into the Program Counter (`ld_pc`).
- **LDA (2400–2800 ns, opcode = 101):** LDA loads a memory value into the Accumulator (`ld_ac`).

4.3 Memory



```
Time=10000 addr=00000 rd=0 wr=1 data=10101010 mem[00000]=xxxxxxxx
Time=15000 addr=00000 rd=0 wr=1 data=10101010 mem[00000]=10101010
Test 1: Write 10101010 to addr 00000, mem[00000]=10101010
Time=20000 addr=00000 rd=0 wr=0 data=zzzzzzzz mem[00000]=10101010
Time=30000 addr=00000 rd=1 wr=0 data=10101010 mem[00000]=10101010
Test 2: Read from addr 00000: data=10101010, mem[00000]=10101010 (Expected: 10101010)
Time=40000 addr=00000 rd=0 wr=0 data=zzzzzzzz mem[00000]=10101010
Time=50000 addr=01111 rd=0 wr=1 data=11110000 mem[01111]=xxxxxxxx
Time=55000 addr=01111 rd=0 wr=1 data=11110000 mem[01111]=11110000
Test 3: Write 11110000 to addr 01111, mem[01111]=11110000
Time=60000 addr=01111 rd=0 wr=0 data=zzzzzzzz mem[01111]=11110000
Time=70000 addr=01111 rd=1 wr=0 data=11110000 mem[01111]=11110000
Test 4: Read from addr 01111: data=11110000, mem[01111]=11110000 (Expected: 11110000)
Time=80000 addr=01111 rd=0 wr=0 data=zzzzzzzz mem[01111]=11110000
$finish called at time : 90 ns : File "C:/Users/dqbao/OneDrive/Desktop/Study/LSI logic design/BaiTapLon/BaiTapLon.srcs/sim_1/new/memory_tb.v" Line 89
```

Figure 7: *Memory Testbench*

The waveform and simulation log show the behavior of the memory module for the RISC CPU design, testing its read and write operations. The module uses a 5-bit address (`addr[4:0]`), an 8-bit data bus (`data[7:0]`), a clock (`clk`), a read/write control signal (`rd_wr`), and a memory array (`mem`). Below is a concise explanation based on the provided waveform and log.

Write to addr=00000 (0–20 ns)

- **Log:**

  `"Time=10 addr=00000 rd=0 wr=1 data=10101010 mem[00000]=xxxxxxxx"`

  `"Test 1: Write 10101010 to addr 00000, mem[00000]=xxxxxxxx"`

- **Behavior:** Write occurs at the 10 ns rising edge; `mem[00000]` updates to 10101010 by 20 ns.

Read from addr=00000 (30–40 ns)

- **Log:**

  `"Time=30 addr=00000 rd=1 wr=0 data=10101010 mem[00000]=10101010"`

  `"Test 2: Read from addr 00000: data=10101010, mem[00000]=10101010 (Expected: 10101010)"`

- **Behavior:** Read at 30 ns retrieves 10101010, matching the expected value.

Write to addr=01111 (50–60 ns)

- **Log:**

```
"Time=50 addr=01111 rd=0 wr=1 data=11110000 mem[01111]=xxxxxxxx"
"Test 3: Write 11110000 to addr 01111, mem[01111]=xxxxxxxx"
```

- **Behavior:** Write at 50 ns updates `mem[01111]` to `11110000` by 60 ns.

Read from addr=01111 (70–80 ns)

- **Log:**

```
"Time=70 addr=01111 rd=1 wr=0 data=11110000 mem[01111]=11110000"
"Test 4: Read from addr 01111: data=11110000, mem[01111]=11110000 (Expected: 11110000)"
"Time=80 addr=01111 rd=0 wr=0 data=zzzzzzzz mem[01111]=11110000 [Expected: 10101010]"
```

- **Behavior:** Read at 70 ns retrieves `11110000`, correct from Test 3; the `[Expected: 10101010]` at 80 ns is a testbench error.

4.4 ALU

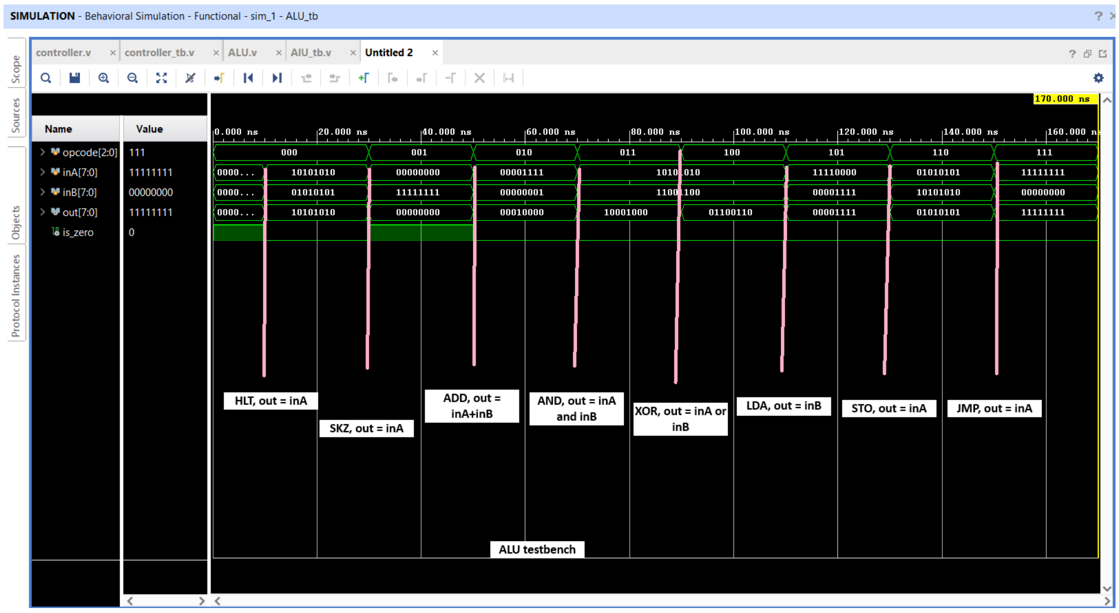


Figure 8: *ALU Testbench*

The waveform from the `ALU_tb.v` testbench simulates the ALU module of the RISC CPU design over 0–170 ns, testing all 8 opcodes (HLT, SKZ, ADD, AND, XOR, LDA, STO, JMP). The ALU module takes a 3-bit `opcode`, 8-bit inputs `inA` and `inB`, and produces an 8-bit `out` and a 1-bit `is_zero`. The project document (page 5) specifies that the ALU performs 8 operations on 8-bit inputs, with `is_zero` indicating if `inA = 0` (asynchronously). The waveform shows the ALU's combinational behavior (no clock).

- **0–20 ns: HLT (opcode=000):** HLT sets `out = inA` (11111111). `is_zero = (inA == 0)` is 0 since `inA` $\neq$ `0`.

- **20–40 ns: SKZ (opcode=001)**: SKZ sets `out = inA` (00000000). `is_zero = 1` since `inA = 0`.
- **40–60 ns: ADD (opcode=010)**: ADD computes `out = inA + inB` (00001111 + 11111111 = 100001110). The 9th bit (carry) is discarded, so `out = 00001110`. `is_zero = 0` since `inA` $\neq$ `0`.
- **60–80 ns: AND (opcode=011)**: AND computes `out = inA & inB` (00001111 & 00001111 = 00001111). `is_zero = 0` since `inA` $\neq$ `0`.
- **80–100 ns: XOR (opcode=100)**: XOR computes `out = inA ^ inB` (10101010 ^ 11001100 = 01100110). `is_zero = 0` since `inA` $\neq$ `0`.
- **100–120 ns: LDA (opcode=101)**: LDA sets `out = inB` (00001111). `is_zero = 0` since `inA` $\neq$ `0`.
- **120–140 ns: STO (opcode=110)**: STO sets `out = inA` (01010101). `is_zero = 0` since `inA` $\neq$ `0`.
- **140–160 ns: JMP (opcode=111)**: JMP sets `out = inA` (11111111). `is_zero = 0` since `inA` $\neq$ `0`.

4.5 Accumulator Register

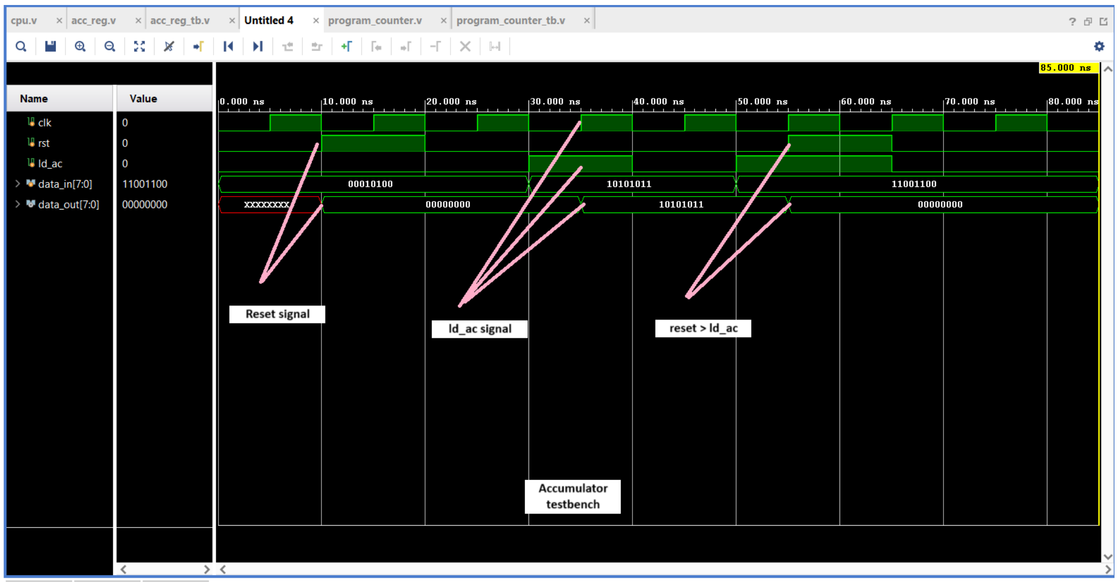
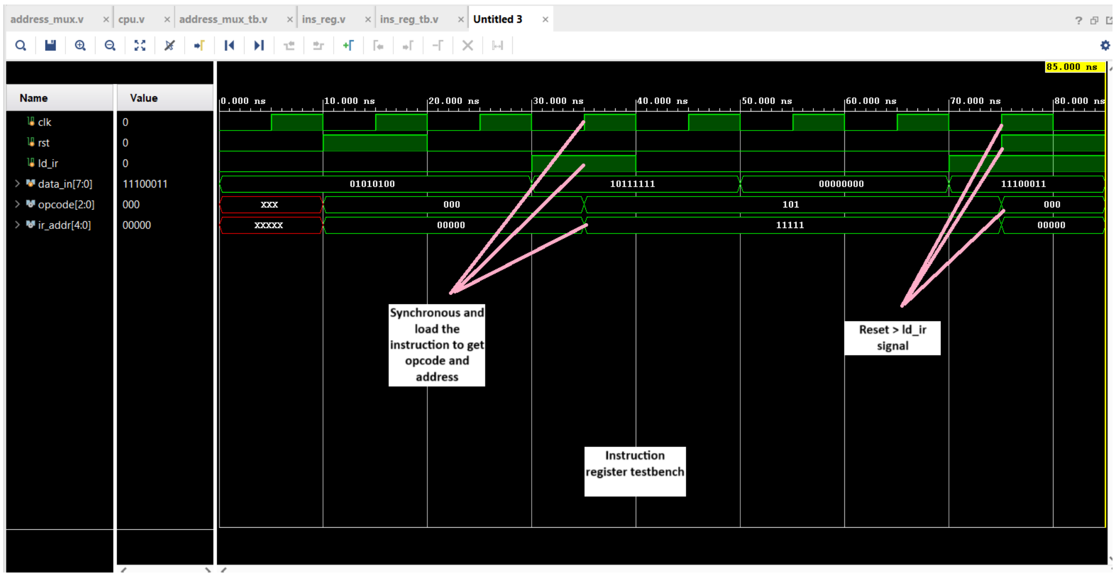


Figure 9: *Accumulator Register Testbench*

- **0–20 ns: Reset Signal**: `rst = 1` triggers a reset on the rising edge of `clk` at 0 ns, setting `data_out` to `8'b0` regardless of `ld_ac` or `data_in`.
- **20–60 ns: ld_ac Signal**: At 20 ns, `rst = 0` and `ld_ac = 1` load `data_in = 10101011` into `data_out`. At 40 ns, `ld_ac` remains 1, but `data_in` is stable, so `data_out` holds 10101011.

- **60–80 ns: Reset with ld_ac**: `rst = 1` takes precedence, resetting `data_out` to `8'b0`, overriding `ld_ac`.
- **80–85 ns: No Change**: With `rst = 0` and `ld_ac = 0`, `data_out` retains its previous value (00000000).

4.6 Instruction Register



Name	Value
clk	0
rst	0
ld_ir	0
data_in[7:0]	11100011
opcode[2:0]	000
ir_addr[4:0]	00000

Figure 10: *Instruction Register Testbench*

- **0–20 ns: Reset Signal**: `rst = 1` resets the register on the rising edge at 0 ns, setting `ir = 8'b0`, so `opcode = 000` and `ir_addr = 00000`.
- **20–60 ns: Instruction Load**
  - At 20 ns, `ld_ir = 1` loads `data_in = 01010101` into `ir`, where `opcode = 010` (bits 7–5) and `ir_addr = 01001` (bits 4–0).
  - At 40 ns, `ld_ir = 1` loads `data_in = 10111111` into `ir`, where `opcode = 101` and `ir_addr = 11111`.
- **60–80 ns: Reset ¿ ld_ir Signal**: `rst = 1` takes precedence, resetting `ir = 8'b0`, overriding `ld_ir`, so `opcode = 000` and `ir_addr = 00000`.
- **80–85 ns: No Change**: With `rst = 0` and `ld_ir = 0`, `ir` retains its reset value (00000000).

4.7 Address MUX

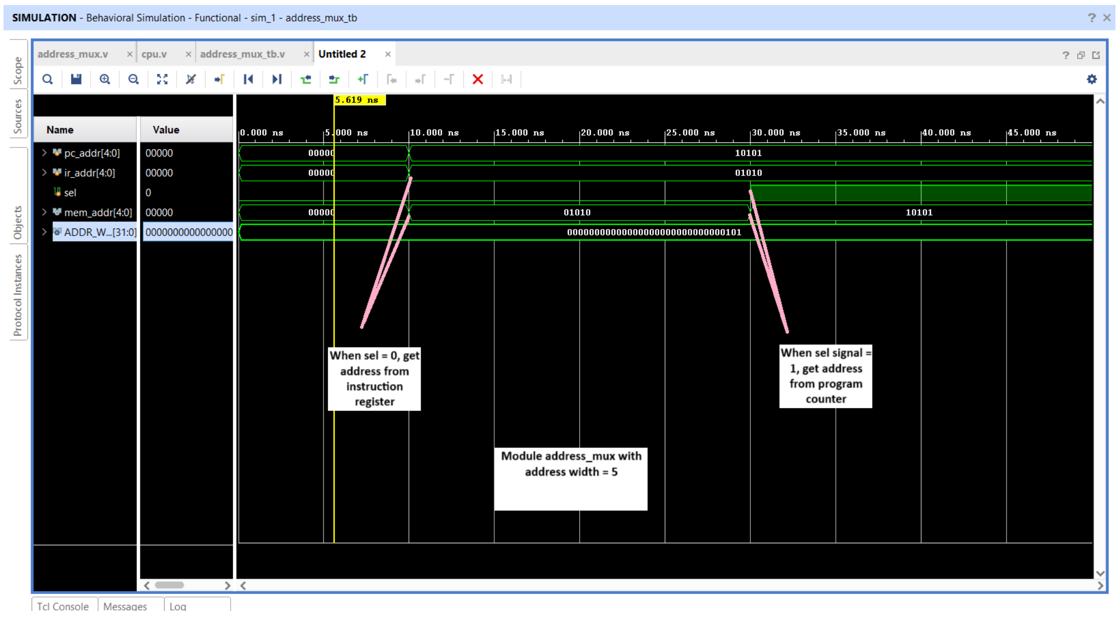
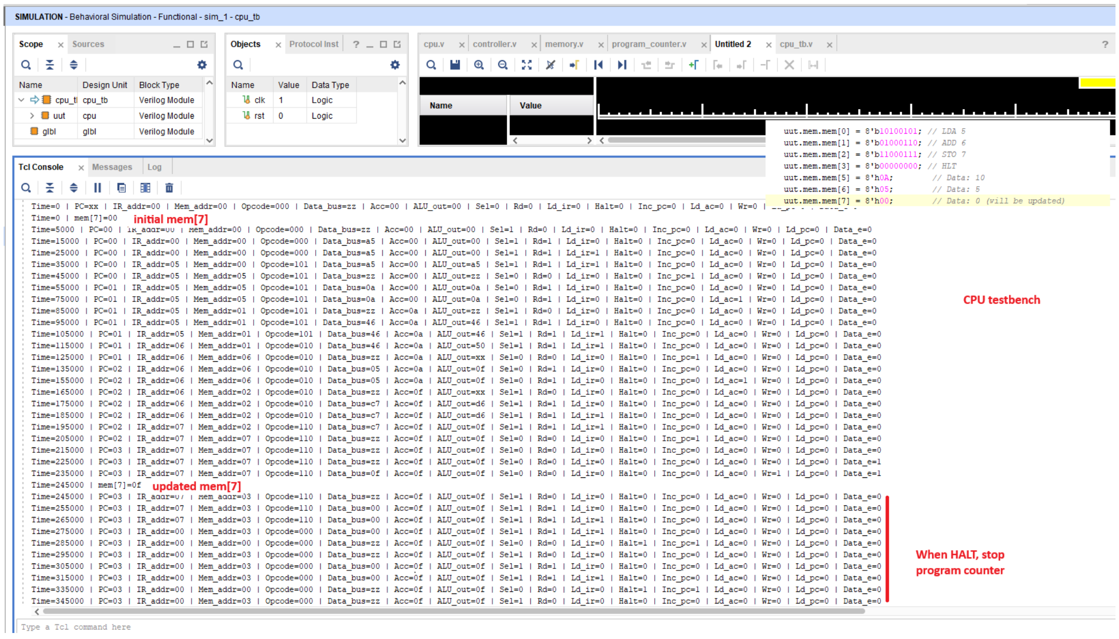


Figure 11: *Address MUX Testbench*

- **0–10 ns: Initial State**: with `sel = 0`, `mem_addr = ir_addr = 00000` (default case in the mux logic).
- **10–20 ns: Test Case 1 (sel = 0)**: `sel = 0` selects `ir_addr = 01010`, so `mem_addr = 01010`.
- **20–30 ns: Test Case 2 (sel = 1)**: `sel = 1` selects `pc_addr = 10101`, so `mem_addr = 10101`.
- **30–45 ns: No Change**: `sel = 1` remains, keeping `mem_addr = 10101`.

4.8 CPU



Name	Design Unit	Block Type
cpu_tb	cpu_tb	Verilog Module
uut	cpu	Verilog Module
glbl	glbl	Verilog Module

Name	Value	Data Type
clk	1	Logic
rst	0	Logic

```
uut.mem.mem[0] = 8'b10100101; // LDA 5
uut.mem.mem[1] = 8'b01000110; // ADD 6
uut.mem.mem[2] = 8'b11000111; // STO 7
uut.mem.mem[3] = 8'b00000000; // HLT
uut.mem.mem[5] = 8'h0A;        // Data: 10
uut.mem.mem[6] = 8'h05;        // Data: 5
uut.mem.mem[7] = 8'h00;        // Data: 0 (will be updated)
```

```
Time=0 | PC=xx | IR_addr=00 | Mem_addr=00 | Opcode=000 | Data_bus=zz | Acc=00 | ALU_out=00 | Sel=0 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 |
Time=0 | mem[7]=00
Time=5000 | PC=00 | IR_addr=00 | Mem_addr=00 | Opcode=000 | Data_bus=zz | Acc=00 | ALU_out=00 | Sel=1 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=15000 | PC=00 | IR_addr=00 | Mem_addr=00 | Opcode=000 | Data_bus=a5 | Acc=00 | ALU_out=00 | Sel=1 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=25000 | PC=00 | IR_addr=00 | Mem_addr=00 | Opcode=000 | Data_bus=a5 | Acc=00 | ALU_out=00 | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=35000 | PC=00 | IR_addr=05 | Mem_addr=00 | Opcode=101 | Data_bus=a5 | Acc=00 | ALU_out=a5 | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=45000 | PC=00 | IR_addr=05 | Mem_addr=05 | Opcode=101 | Data_bus=zz | Acc=00 | ALU_out=zz | Sel=0 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=1 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=55000 | PC=01 | IR_addr=05 | Mem_addr=05 | Opcode=101 | Data_bus=0a | Acc=00 | ALU_out=0a | Sel=0 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=75000 | PC=01 | IR_addr=05 | Mem_addr=05 | Opcode=101 | Data_bus=0a | Acc=00 | ALU_out=0a | Sel=0 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=1 | Wr=0 | Ld_pc=0 | Data_e=0
Time=85000 | PC=01 | IR_addr=05 | Mem_addr=01 | Opcode=101 | Data_bus=zz | Acc=0a | ALU_out=zz | Sel=1 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=95000 | PC=01 | IR_addr=05 | Mem_addr=01 | Opcode=101 | Data_bus=46 | Acc=0a | ALU_out=46 | Sel=1 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=105000 | PC=01 | IR_addr=05 | Mem_addr=01 | Opcode=101 | Data_bus=46 | Acc=0a | ALU_out=46 | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=115000 | PC=01 | IR_addr=06 | Mem_addr=01 | Opcode=010 | Data_bus=46 | Acc=0a | ALU_out=50 | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=125000 | PC=01 | IR_addr=06 | Mem_addr=06 | Opcode=010 | Data_bus=zz | Acc=0a | ALU_out=xx | Sel=0 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=1 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=135000 | PC=02 | IR_addr=06 | Mem_addr=06 | Opcode=010 | Data_bus=05 | Acc=0a | ALU_out=0f | Sel=0 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=155000 | PC=02 | IR_addr=06 | Mem_addr=06 | Opcode=010 | Data_bus=05 | Acc=0a | ALU_out=0f | Sel=0 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=1 | Wr=0 | Ld_pc=0 | Data_e=0
Time=165000 | PC=02 | IR_addr=06 | Mem_addr=02 | Opcode=010 | Data_bus=zz | Acc=0f | ALU_out=xx | Sel=1 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=175000 | PC=02 | IR_addr=06 | Mem_addr=02 | Opcode=010 | Data_bus=c7 | Acc=0f | ALU_out=d6 | Sel=1 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=185000 | PC=02 | IR_addr=06 | Mem_addr=02 | Opcode=010 | Data_bus=c7 | Acc=0f | ALU_out=d6 | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=195000 | PC=02 | IR_addr=07 | Mem_addr=02 | Opcode=110 | Data_bus=c7 | Acc=0f | ALU_out=0f | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=205000 | PC=02 | IR_addr=07 | Mem_addr=07 | Opcode=110 | Data_bus=zz | Acc=0f | ALU_out=0f | Sel=0 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=1 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=215000 | PC=03 | IR_addr=07 | Mem_addr=07 | Opcode=110 | Data_bus=zz | Acc=0f | ALU_out=0f | Sel=0 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=225000 | PC=03 | IR_addr=07 | Mem_addr=07 | Opcode=110 | Data_bus=zz | Acc=0f | ALU_out=0f | Sel=0 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=1
Time=235000 | PC=03 | IR_addr=07 | Mem_addr=07 | Opcode=110 | Data_bus=0f | Acc=0f | ALU_out=0f | Sel=0 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=1 | Ld_pc=0 | Data_e=1
Time=245000 | mem[7]=0f
Time=245000 | PC=03 | IR_addr=07 | Mem_addr=03 | Opcode=110 | Data_bus=zz | Acc=0f | ALU_out=0f | Sel=1 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=255000 | PC=03 | IR_addr=07 | Mem_addr=03 | Opcode=110 | Data_bus=00 | Acc=0f | ALU_out=0f | Sel=1 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=265000 | PC=03 | IR_addr=07 | Mem_addr=03 | Opcode=110 | Data_bus=00 | Acc=0f | ALU_out=0f | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=275000 | PC=03 | IR_addr=00 | Mem_addr=03 | Opcode=000 | Data_bus=00 | Acc=0f | ALU_out=0f | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=285000 | PC=03 | IR_addr=00 | Mem_addr=00 | Opcode=000 | Data_bus=zz | Acc=0f | ALU_out=0f | Sel=0 | Rd=0 | Ld_ir=0 | Halt=1 | Inc_pc=1 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=295000 | PC=03 | IR_addr=00 | Mem_addr=03 | Opcode=000 | Data_bus=zz | Acc=0f | ALU_out=0f | Sel=1 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=305000 | PC=03 | IR_addr=00 | Mem_addr=03 | Opcode=000 | Data_bus=00 | Acc=0f | ALU_out=0f | Sel=1 | Rd=1 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=315000 | PC=03 | IR_addr=00 | Mem_addr=03 | Opcode=000 | Data_bus=00 | Acc=0f | ALU_out=0f | Sel=1 | Rd=1 | Ld_ir=1 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=335000 | PC=03 | IR_addr=00 | Mem_addr=00 | Opcode=000 | Data_bus=zz | Acc=0f | ALU_out=0f | Sel=0 | Rd=0 | Ld_ir=0 | Halt=1 | Inc_pc=1 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
Time=345000 | PC=03 | IR_addr=00 | Mem_addr=03 | Opcode=000 | Data_bus=zz | Acc=0f | ALU_out=0f | Sel=1 | Rd=0 | Ld_ir=0 | Halt=0 | Inc_pc=0 | Ld_ac=0 | Wr=0 | Ld_pc=0 | Data_e=0
```

Figure 12: *CPU Testbench*

- 0–10 ns: Reset Phase
- 10–90 ns: Instruction 1 - LDA 5 (Fetch-Execute Cycle)
- 90–170 ns: Instruction 2 - ADD 6
- 170–250 ns: Instruction 3 - STO 7
- 250–330 ns: Instruction 4 - HLT
- 330–510 ns: Halted State

5 Conclusion

This project successfully designed and implemented a simple RISC CPU using Verilog HDL, meeting the core objectives outlined in the course requirements. The processor supports eight instructions (HLT, SKZ, ADD, AND, XOR, LDA, STO, JMP) with a 3-bit opcode and 5-bit operand address, operating synchronously with a clock signal. Through modular design, rigorous simulation using Cadence tools, and comprehensive testbenches, the team validated the functionality of all components, including the Program Counter, Address MUX, Memory, Instruction Register, Accumulator, ALU, and Controller. The CPU correctly executes instruction sequences, as demonstrated by the CPU testbench (e.g., loading data, performing arithmetic, storing results, and halting).

5.1 Future Improvements

Several enhancements could further improve the design:

- **Pipeline Implementation**: Introduce a pipelined architecture to handle hazards, such as data or control hazards, improving instruction throughput.
- **ALU Expansion**: Add support for fixed-point or floating-point operations, multipliers, or dividers to increase computational capability.
- **Power Optimization**: Analyze and optimize power consumption using synthesis tools, targeting low-power embedded applications.

5.2 Challenges Encountered

The team faced several challenges during development:

- **Controller Signal Errors**: An unexpected `halt` signal assertion in the ALU_OP state for the HALT instruction (Section 4.2) required additional debugging to align with specifications.
- **Testbench Complexity**: Designing comprehensive testbenches for all modules was time-consuming, particularly for edge cases like simultaneous load and increment in the Program Counter.
- **Timing Optimization**: Ensuring timing constraints for high-frequency operation was challenging without access to advanced synthesis tools.

5.3 Work Distribution

In summary, this project provided valuable hands-on experience in hardware design, Verilog HDL, and digital system simulation. The successful implementation of the RISC CPU underscores the team's ability to apply theoretical concepts to practical engineering challenges, laying a foundation for future exploration in computer architecture and system-on-chip design.

Table 3: *Work Distribution*

Member	Task	Description
Duong Gia Bao	ALU and Testbenches	Designed ALU module, implemented 8 operations, and developed ALU testbench.
Phan Quang Minh	Controller and Simulation	Designed Controller FSM, wrote testbenches, and analyzed simulation waveforms.
Nguyen Thanh Tai	PC and Integration	Implemented Program Counter, integrated modules into top-level CPU design.
Pham Van Bach	Memory and Documentation	Designed Memory module, drafted report sections, and formatted LaTeX document.