VIETNAM NATIONAL UNIVERSITY HO CHI MINH
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL & ELECTRONICS ENGINEERING
DEPARTMENT OF ELECTRONICS

----------o0o----------

CAPSTONE PROJECT 1

DESIGN AND IMPLEMENTATION OF A RISC-V PIPELINED CPU ON FPGA

Instructor: PhD. Tran Hoang Linh

Students:

Nguyễn Chí Bảo – 2151175

Nguyễn Quốc Gia Bảo - 2151176

HO CHI MINH CITY, MAY 2024

Acknowledgment

We express our sincere gratitude to our project advisors, Dr. Tran Hoang Linh and Mr. Cao Xuan Hai, lecturers in the Electronics Department at Ho Chi Minh City University of Technology. Their invaluable guidance, insightful feedback, and unwavering support throughout the development of this capstone project were instrumental in its successful completion.

We would also like to extend our appreciation to the faculty members of the Electronics Department who have imparted their knowledge and expertise, laying a solid theoretical foundation that enabled us to undertake this project successfully.

Furthermore, we acknowledge the contributions of our project partner, whose collaboration, dedication, and teamwork were essential in overcoming challenges and achieving our objectives.

While we have strived to produce a comprehensive and well-researched project, we acknowledge that there may be room for improvement. We welcome any constructive feedback and guidance from our advisors and faculty, which will undoubtedly enhance our understanding and better prepare us for future professional endeavors.

Ho Chi Minh city, 24 May 2024 .

Students

ABSTRACT

This report presents the design and implementation of a 5-stage pipelined CPU based on the RISC-V RV32I instruction set architecture in Verilog HDL. The CPU incorporates hazard detection and handling mechanisms to ensure proper execution of instructions in the pipeline. The design is targeted for implementation on the Altera DE2 FPGA board, leveraging its on-board resources for I/O and display.

As a practical application, an assembly language program for a stopwatch is developed to showcase the CPU's functionality. The stopwatch program utilizes the DE2 board's switches to control the start, stop, and reset operations, while the elapsed time is displayed on the 7-segment LED display. The report details the CPU architecture, pipeline stages, hazard handling techniques, and the assembly language programming approach for the stopwatch application.

The successful implementation of the RISC-V CPU on the FPGA board demonstrates the design's feasibility and highlights the capabilities of modern hardware description languages and FPGA technology in realizing complex digital systems. The stopwatch application showcases the CPU's ability to execute assembly language programs and interact with external peripherals, making it a comprehensive project that combines various aspects of computer architecture, digital design, and embedded systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Overview

The objective of this project is to design and implement a 5-stage pipelined RISC-V CPU that supports the RV32I base instruction set architecture. The CPU will be designed using Verilog Hardware Description Language (HDL) and will be capable of handling various hazards that may arise during pipelined execution. Additionally, the project involves developing an assembly language program for a stopwatch application and interfacing the CPU with external peripherals such as 7-segment LED displays and switches on the DE2 FPGA board..

The stopwatch application will be written in assembly language, which provides low-level control over the CPU's operations and memory accesses. The assembly program will leverage the CPU's capabilities to implement the necessary functionality, such as measuring elapsed time, starting, stopping, and resetting the stopwatch. The stopwatch will be interfaced with external peripherals, including 7-segment LED displays for displaying the time and switches for controlling the stopwatch's operations.

The DE2 FPGA board from Terasic will serve as the target platform for implementing the designed CPU and the stopwatch application. FPGAs (Field-Programmable Gate Arrays) are reconfigurable hardware devices that allow the implementation of custom digital circuits, making them ideal for prototyping and testing complex systems like the RISC-V CPU.

Overall, this project aims to provide hands-on experience in designing a pipelined CPU architecture, implementing hazard handling mechanisms, developing assembly language programs, and interfacing with external peripherals using an FPGA board. The successful completion of this project will demonstrate proficiency in various aspects of computer architecture, digital logic design, and hardware-software co-design.

1.2 Mission

a. RISC-V CPU Design and Implementation

Requirements:

- Implement a 5-stage pipelined CPU that supports the RV32I base instruction set architecture
- Handle hazards such as data hazards, control hazards, and structural hazards

- Design the CPU using Verilog HDL

Expected Outcomes:

- A functional RISC-V CPU that can execute RV32I instructions correctly
- Proper handling of hazards to ensure correct program execution
- Approach and Implementation Ideas:
- Study the RISC-V ISA and RV32I instruction set thoroughly
- Design the datapath and control unit for each pipeline stage
- Implement hazard detection and handling mechanisms (e.g., data forwarding, stalling, flushing)
- Develop test benches for verification and debugging

b. Stopwatch Application Development

Requirements:

- Write an assembly language program for a stopwatch application
- Implement features like start, stop, reset, and time display

Expected Outcomes:

- A functional stopwatch application running on the RISC-V CPU
- Approach and Implementation Ideas:
- Learn the RISC-V assembly language and instruction encoding
- Design the stopwatch algorithm and implement it using assembly instructions
- Utilize the CPU's timer or counter registers for time measurement
- Develop subroutines for various stopwatch functionalities

c. Interfacing with External Peripherals

Requirements:

- Interface the RISC-V CPU with 7-segment LED displays for time display
- Connect switches on the DE2 FPGA board for stopwatch control

Expected Outcomes:

- Successful integration of the CPU with external peripherals
- Ability to control the stopwatch and display time on the 7-segment LEDs

Approach and Implementation Ideas:

- Study the DE2 FPGA board's documentation and peripheral interfaces
- Design a memory-mapped I/O scheme for interfacing with peripherals
- Implement driver routines in assembly language for controlling the peripherals
- Develop a communication protocol between the CPU and peripherals

d. FPGA Implementation

Requirements:

- Implement the designed RISC-V CPU and stopwatch application on the DE2 FPGA board

Expected Outcomes:

- Successful deployment and demonstration of the project on the FPGA board

Approach and Implementation Ideas:

- Learn the FPGA design flow and tools (e.g., Quartus Prime)
- Synthesize and map the Verilog HDL code to the FPGA's logic elements
- Configure the FPGA with the compiled design
- Test and debug the implementation on the FPGA board

Project Constraints:

- Limited time and resources for development and testing
- Complexity of designing a pipelined CPU and handling hazards
- Limitations of the DE2 FPGA board's resources (e.g., memory, I/O interfaces)

1.3 Division of work in the group

Research and Initial Planning:

Nguyễn Quốc Gia Bảo (2151176):

- Conducted research on the RISC-V architecture and RV32I base instruction set.
- Compiled resources and documentation on pipeline hazards and handling techniques.

Nguyễn Chí Bảo (2151175):

- Investigated Verilog HDL for CPU design.
- Gathered information on FPGA implementation, focusing on the DE2 board and its peripherals.

Pipeline CPU Design:

Nguyễn Quốc Gia Bảo (2151176):

- Developed the initial 5-stage pipeline architecture (IF, ID, EX, MEM, WB).
- Designed the instruction fetch and decode stages.

Nguyễn Chí Bảo (2151175):

- Designed the execute, memory access, and write back stages.
- Integrated pipeline hazard detection and handling logic.

Verilog HDL Coding:

Both Students:

- Collaboratively wrote the Verilog code for the datapath and control unit.
- Implemented hazard detection and handling mechanisms such as forwarding and stalling.

FPGA Implementation:

Nguyễn Quốc Gia Bảo (2151176):

- Configured the FPGA design flow, including synthesis, place, and route.
- Managed the memory-mapped I/O for peripheral interfacing.

Nguyễn Chí Bảo (2151175):

- Set up the DE2 FPGA board and connected the necessary peripherals (7-segment displays, switches).
- Conducted initial FPGA configuration and testing.

Assembly Programming:

Nguyễn Quốc Gia Bảo (2151176):

- Developed the RISC-V assembly code for the stopwatch algorithm.
- Implemented the peripheral interfacing code for controlling the 7-segment displays and switches.

Nguyễn Chí Bảo (2151175):

- Assisted in refining the assembly code and ensuring accurate timing and display updates.

- Debugged and tested the assembly program on the FPGA board.

Testing and Debugging:

Both Students:

- Performed extensive testing of the CPU design, including functional and timing simulations.
- Debugged issues collaboratively, refining both Verilog and assembly code as necessary.

Documentation and Report Writing:

Nguyễn Quốc Gia Bảo (2151176):

- Drafted sections on the RISC-V architecture, pipeline design, and hazard handling techniques.

Nguyễn Chí Bảo (2151175):

- Wrote the sections on FPGA implementation, assembly programming, and project conclusions.

Both Students:

- Reviewed and edited the entire report to ensure clarity, coherence, and completeness.

2. THEORY

2.1 RISC-V ISA and RV32I

2.1.1 Overview of RISC-V Architecture

a. Introduction to RISC-V:

RISC-V (Reduced Instruction Set Computing - Version 5) is an open-source instruction set architecture (ISA) designed for simplicity, flexibility, and scalability. It is not tied to any specific company, making it widely accessible for academic research, industry, and hobbyists. RISC-V's design principles emphasize modularity and extensibility, allowing the

ISA to support a wide range of applications, from small embedded systems to high-performance servers.

b. Key Features of RISC-V:

Open and Free: Unlike many proprietary ISAs, RISC-V is open-source, meaning anyone can use, modify, and implement it without paying licensing fees.

Modular Design: RISC-V is designed with a base integer instruction set, which can be extended with optional extensions (e.g., for floating-point operations, atomic operations, and vector processing).

Simplicity and Clean Slate Design: The RISC-V ISA is clean and simple, free from legacy features that complicate other ISAs. This simplicity makes it easier to implement and optimize.

Scalability: The RISC-V architecture supports various bit-widths (32, 64, and 128 bits), making it suitable for different performance and application requirements.

c. Comparison with Other ISAs:

RISC-V is often compared to other ISAs like ARM and x86. ARM is widely used in mobile and embedded devices, while x86 dominates the PC and server markets. RISC-V differentiates itself by being open-source and having a clean, extensible design that can adapt to a broad range of applications without the baggage of historical design decisions.

2.1.2 RV32I Base Integer Instruction Set

a. Overview:

The RV32I (RISC-V 32-bit Integer) instruction set is the foundational subset of the RISC-V architecture. It includes all the necessary instructions for basic integer computation, control flow, and memory access. The RV32I set is mandatory for all 32-bit RISC-V processors, ensuring a standard base for software development.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Table 1. RV32I Instruction Set

b. Instruction Categories:

Arithmetic and Logic Instructions: These instructions perform basic arithmetic (addition, subtraction, etc.) and logical operations (AND, OR, XOR).

Examples: ADD, SUB, AND, OR, XOR

Control Transfer Instructions: Instructions that alter the flow of execution, such as jumps and branches.

Examples: JAL (Jump and Link), BEQ (Branch if Equal), BNE (Branch if Not Equal)

Memory Access Instructions: Instructions for loading from and storing data to memory.

Examples: LW (Load Word), SW (Store Word)

Immediate Instructions: Instructions that perform operations using immediate (constant) values.

Examples: ADDI (Add Immediate), ANDI (AND Immediate)

2.2 5-Stage Pipeline Architecture

2.2.1 Introduction to Pipelining:

Pipelining improves CPU performance by overlapping the execution of multiple instructions, increasing throughput and efficient resource utilization.

2.2.2 5-Stage Pipeline:

Instruction Fetch (IF): Fetch the instruction from memory using the Program Counter (PC).

Instruction Decode (ID): Decode the instruction and read necessary registers.

Execute (EX): Perform arithmetic or logical operations in the ALU.

Memory Access (MEM): Access memory for load and store instructions.

Write Back (WB): Write the result back to the register file.

Data Flow Example:

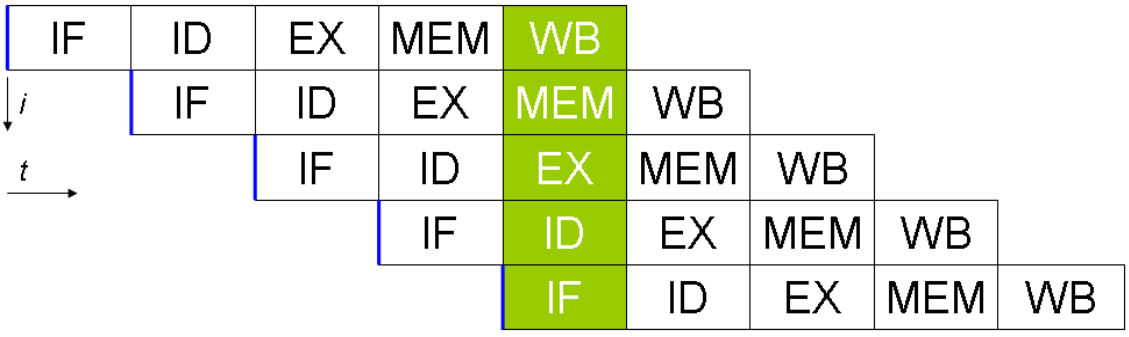


Figure 1. Example of data flow

2.2.3 Pipeline Hazards

Definition and Types:

Pipeline hazards are conditions that prevent the next instruction from executing in its designated clock cycle, classified into data, control, and structural hazards.

Data Hazards:

- Read-After-Write (RAW):

Occurs when an instruction depends on the result of a previous instruction.

Example: ADD x1, x2, x3 followed by SUB x4, x1, x5.

Mitigation: Forwarding or stalling.

- Control Hazards:

Branch Instructions:

Occur when the pipeline makes incorrect branch predictions.

Example: BEQ x1, x2, target.

Mitigation: Branch prediction and pipeline flushing.

- Structural Hazards:

Resource Contention: Occurs when multiple instructions compete for the same hardware resource.

Example: Single memory unit accessed by load and store instructions simultaneously.

Mitigation: Adding more resources or using hardware interlocks.

Summary:

Pipelining increases CPU performance by allowing simultaneous instruction processing across multiple stages. Handling pipeline hazards through techniques like forwarding, stalling, branch prediction, and resource allocation ensures efficient and correct instruction execution.

2.3 Hazard Handling Techniques

2.3.1 Data Hazard Handling: Forwarding, Stalling

a. Data Forwarding (Data Bypassing):

Explanation: Data forwarding resolves data hazards by rerouting data from pipeline stages where it is available (EX or MEM) directly to stages where it is needed, bypassing the register file.

How It Works: If an instruction in the EX stage requires a value produced by an earlier instruction still in the pipeline, the value is forwarded directly from the later pipeline stage (e.g., MEM or WB) to the EX stage.

Example: ADD x1, x2, x3 followed by SUB x4, x1, x5.

The result of ADD is forwarded to the EX stage of SUB as soon as it is available.

b. Stalling (Pipeline Bubbling):

Explanation: Stalling introduces pipeline bubbles (idle cycles) to delay instructions until the data hazard is resolved.

How It Works: If forwarding is not sufficient to resolve a hazard (e.g., when an instruction needs a value that is not yet computed), the pipeline control logic stalls the dependent instruction by inserting NOPs (no-operations) until the required data is available.

Example: LW x1, 0(x2) followed by ADD x3, x1, x4.

The pipeline inserts a stall cycle between LW and ADD to wait for the LW instruction to complete.

In this section, we briefly discussed hazard handling techniques in pipelined CPU design, focusing on data forwarding and stalling to mitigate data hazards. However, it's important to note that the hardware architecture design and implementation details will provide a more comprehensive understanding of how these techniques are integrated into the CPU pipeline. The upcoming sections will delve deeper into the hardware aspects, including the specific mechanisms employed to detect, manage, and resolve hazards within the pipeline stages. By exploring the intricacies of the hardware architecture, we can gain a clearer insight

into the practical implementation of hazard handling techniques and their impact on CPU performance and efficiency.

3. HARDWARE ARCHITECTURE DESIGN & IMPLEMENTATION

3.1 Main Design Concept

This project implements a RISC-V pipelined CPU on an FPGA, emphasizing performance optimization through hazard handling and peripheral interfacing. The 5-stage pipeline architecture, which includes register stages between each pipeline stage, ensures efficient instruction execution by maintaining data continuity. Hazard handling techniques prevent disruptions, while peripheral interfacing connects the CPU to external devices, enhancing functionality. This project showcases the effective design and implementation of a RISC-V CPU on FPGA, focusing on performance and functionality.

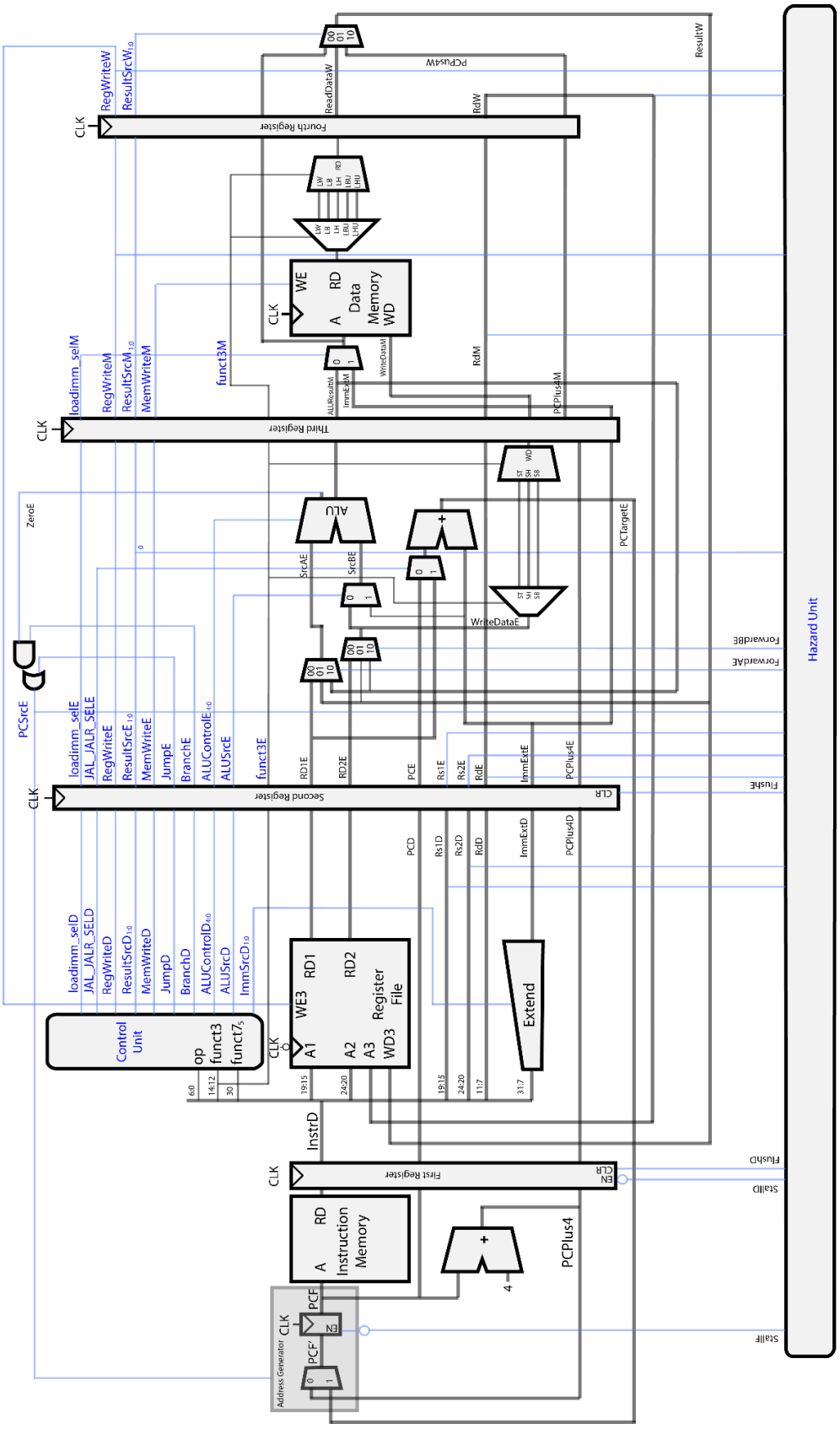


Figure 2. Block diagram of Risc Pipeline CPU

3.2 CPU Pipeline Design

3.2.1 Instruction Fetch (IF)

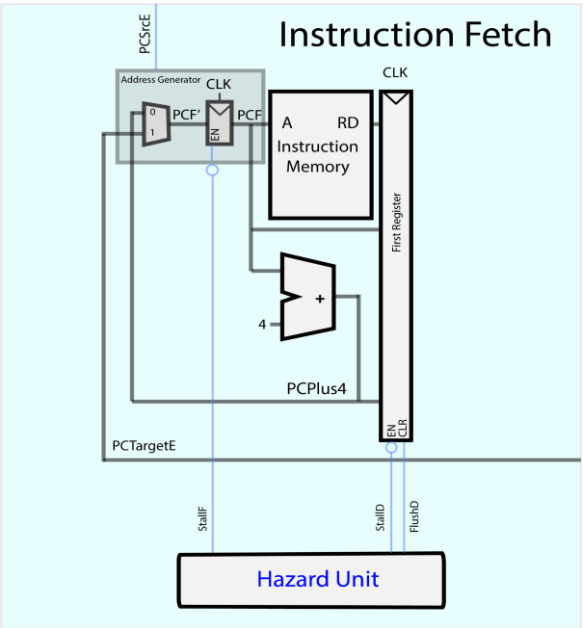


Figure 3. Block diagram of Fetch phase

Function: contains Instruction memory, where the program instructions to be executed are stored. Fetch Instruction fetches the corresponding instructions using the PC (Program Counter). The value of the PC can be either PC + 4, which is the next instruction, or a value returned from Execute Instruction. This control depends on two control signals: Br_sel and PC_enable.

Type	Name	Funtion
Input	CLK	Clock pulse operation of the CPU
	rst_ni	Reset PC and Instruction mem
	PCTargetE	Data from ALU in Execute Instruction
	PCPlus4	PC + 4
	PCSrcE	Select next address is PCPlus4 or PCTargetE
Output	PCF	PC of selected instruction
	instrD	Selected instruction

Table 2. Explanation of Input and Output of Fetch phase

3.2.2 Instruction Decode (ID)

Figure. Block diagram of Instruction Decode

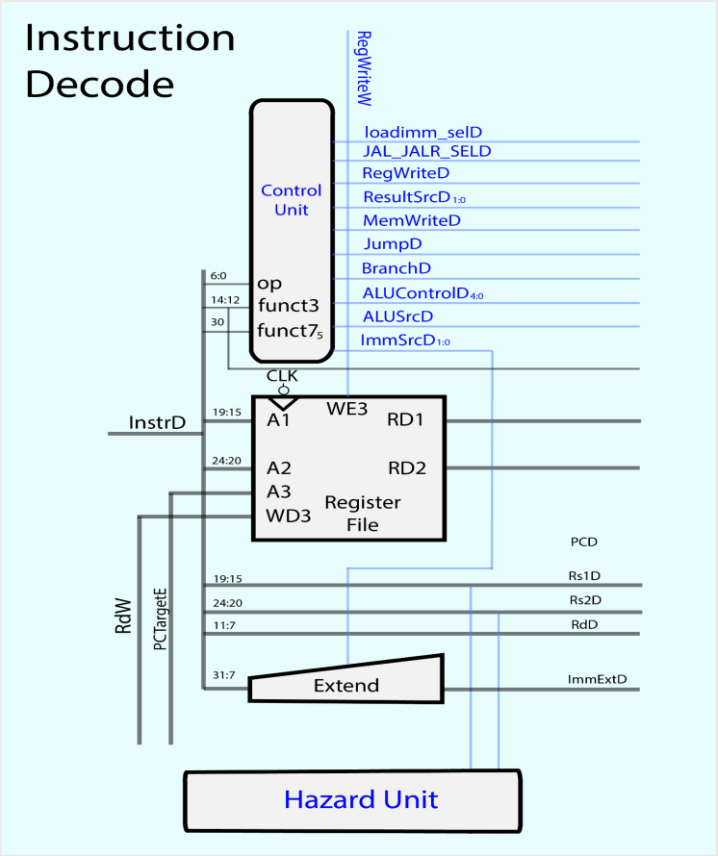


Figure 4. Block diagram of Decode phase

Function: receives the instruction code from the Fetch Phase and decodes it to select the data from the register required by the instruction, and provides the necessary control signals for executing the instruction in the subsequent phases, including Execute, Memory, and Write Back

Type	Name	Function
Input	InstrD	Executing Instruction
Output	MemWriteD	Controls whether data should be written to memory
	ALUSrcD	Selects whether the second ALU operand comes from a register or an immediate value
	RegWriteD	Controls whether the result should be written back to a register
	BranchD	Indicates a branch instruction
	JumpD	Indicates a jump instruction
	JAL_JALR_SELD	Select Jal or Jalr
	loadimm_selD	Select immediate for LUI instruction
	ResultSrcD	Selects the source of the result for register write-back
	ALUControlD	Specifies the ALU operation
	ImmSrcD	Selects the type of immediate value to use

Table 3. Explanation of Input and Output of Decode phase

Instruction	MemWriteD	ALUSrcD	RegWriteD	BranchD	JumpD	JAL_JALR_SELD	loadimm_selD	ResultSrcD	ALUControlD	ImmSrcD
lw	0	1	1	0	0	1	0	2'b01	5'b00000	3'b000
sw	1	1	0	0	0	x	0	2'bxx	5'b00000	3'b001
R-type	0	0	1	0	0	x	0	2'b00	Depend	xxx
B-type	0	0	0	1	0	x	0	2'bxx	5'b00001	3'b010
I-type	0	1	1	0	0	x	0	2'b00	Depend	3'b000
J-type	0	x	1	0	1	0	0	2'b10	Depend	3'b011
JALR	0	1	1	0	1	1	0	2'b10	Depend	3'b000
U-type	0	1	1	0	0	x	1	2'b00	5'b00000	3'b100
Default	0	x	0	0	0	x	x	2'b00	5'b00000	3'b000

Table 4. Control Unit decode

3.2.3 Execute (EX)

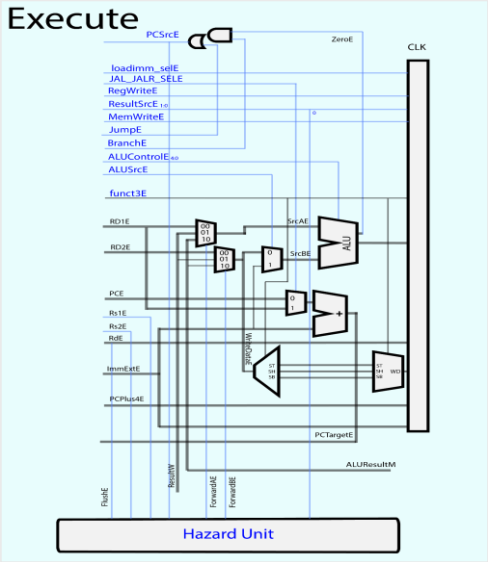


Figure 5. Block diagram of Execute

Function: execute the instructions based on the signal and register values from the previous phase, such as add, subtract, and, or, xor, not, etc.

In this phase, the input signals are the output signals from the Decode Phase as mentioned previously.

ALUControl	Intruction	R-type	I-type
00000	ADD	*rd = rs1 + rs2*	*rd = rs1 + imm*
00001	SUB	*rd = rs1 – rs2*	*rd = rs1 – imm*
00010	AND	*rd = rs1 & rs2*	*rd = rs1 & imm*
00011	OR	*rd = rs1 | rs2*	*rd = rs1 | imm*
00100	SLL	*rd = rs1 << rs2[4:0]*	*rd = rs1 << imm [4:0]*
00101	SLT	*rd = (signed rs1 <signed rs2) ? 32'd1 : 32'd0*	*rd = (signed rs1 <signed imm) ? 32'd1 : 32'd0*
00111	SRA	*rd = rs1 >>> rs2[4:0]*	*rd = rs1 >>> imm [4:0]*
01000	SLTU	*rd = (rs1 < rs2) ? 32'd1 : 32'd0*	*rd = (rs1 < imm) ? 32'd1 : 32'd0*
01010	XOR	*rd = rs1 ^ rs2*	*rd = rs1 ^ imm*
01110	SRL	*rd = rs1 >> rs2[4:0]*	*rd = rs1 >> imm [4:0]*

Table 5. Operations of RV32I ALU

Based on the information above, ALU design looks like:

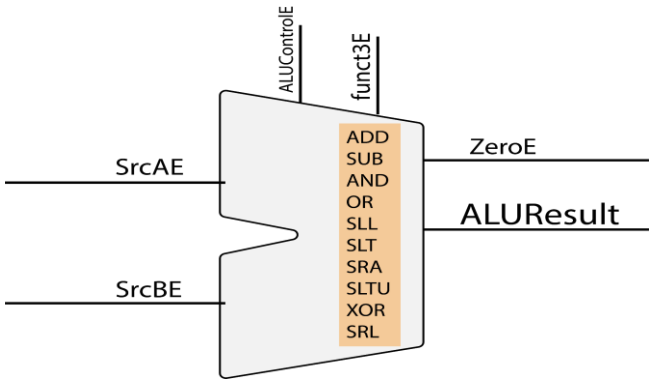


Figure 6. ALU Design

❖ **Branch comparison**

RISC- V processor has 6 conditional branch instructions (B-type), each of which take two source registers and a label indicating where to go. Different types of branches are distinguished by 3 bits of the funct3. Funct3 consists of bits 12 to 14 of the instruction. This is conditional branch instructions table below:

Funct3	Instruction	Description (B-type)
000	BEQ	*Branch when rs1 = rs2*
001	BNE	*Branch when rs1 ≠ rs2*
100	BLT	*Branch when signed rs1 < signed rs2*
101	BGE	*Branch when signed rs1 > signed rs2*
110	BLTU	*Branch when rs1 < rs2*
111	BGEU	*Branch when rs1 > rs2*

Table 6. **Branch Instructions**

Based on the information above, Branch Comparison design looks like:

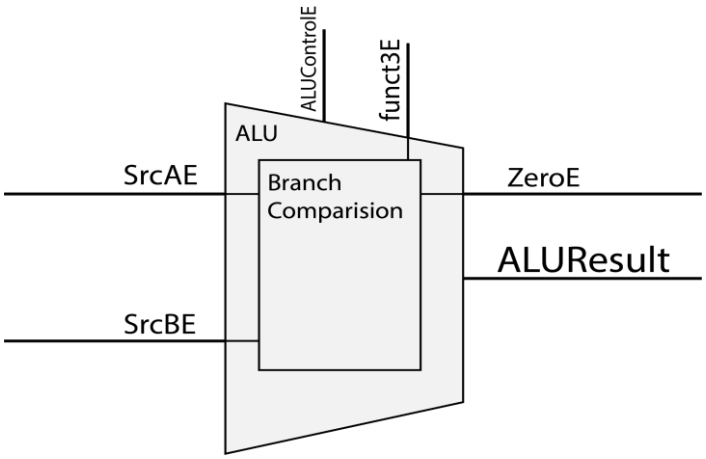


Figure 7. Block diagram of Comparison

3.2.4 Memory Access (MEM)

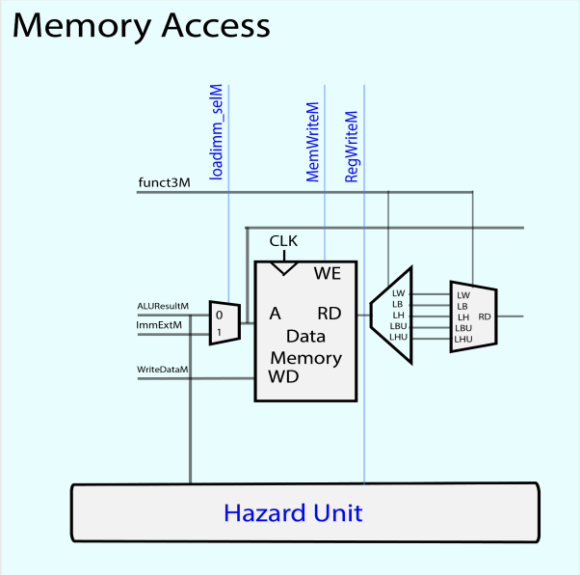


Figure 8. Block diagram of Execute

Function: performs Load/Store data from memory.

In this phase, the input signals are the output signals from the Execute Phase mentioned in the previous section.

Type	Name	Function
Input	WriteDataM	Data writing to address in Store instruction
	ALUResultM	Result of ALU in Execute Instruction
	clk	Clock pulse operation of the CPU
	MemWriteM	Controls whether data should be written to memory
	rst	Reset PC and Data mem
Output	ReadData	Data readed from memory

Table 7. Explanation of input and ouput of Memory Phase

❖ **Type of Load**

There are 6 types of load: LW, LBU, LB, LH, LHU. We created a mux based on the condition of function3 to distinguish each type of load. The data of the load instruction is the output of the Data Memory block. Each different type of load will select a different size of bits from the output of the Data Memory block. For example, the LH instruction loads 16 bits with sign, LB is 8 bits with sign, LHU is 16 bits, and LBU is 8 bits.

3.2.5 Write Back (WB)

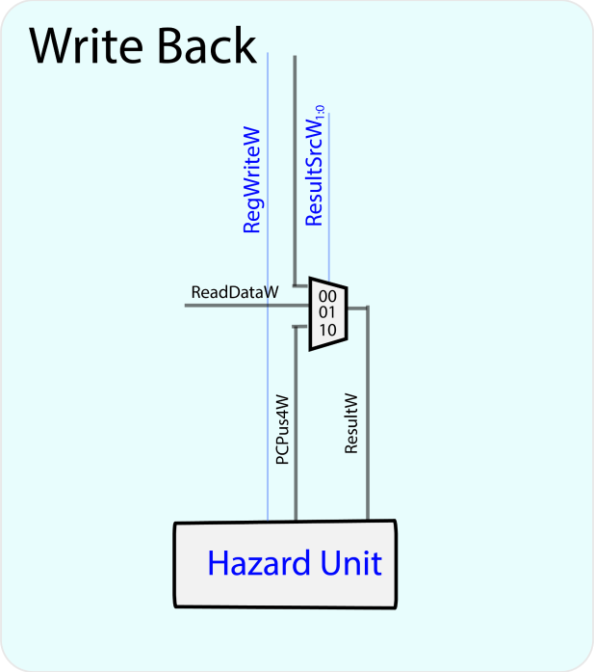


Figure 9. Block diagram of Execute

Function: Select the signal that needs to be written back to the register file. In this phase, the input signals are the output signals from the Memory Phase, as described in the previous section.

Type	Name	Function
Input	ResultSrcW	Select which data to writeback
Output	ResultW	Date back to register file

Table 8. Explanation of Input and Output of WriteBack phase

3.3 Hazard Handling Mechanisms

❖ **Analysis**

Pipeline dangers occur when the subsequent instruction cannot be carried out in the subsequent clock cycle. The three categories of these scenarios are control hazard, data hazard, and structural hazard.

Structural hazard: happens when a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute. In the present case, our design uses two memories, in the same clock cycle fetching data from multiple adjacent instructions will not be a problem. However, without two memories, our pipeline could have a structural hazard.

Data hazard: the condition in which an instruction that is ready but not yet available prevents it from executing within the correct clock cycle. When a command depends on a previous one that is still being processed, data hazards occur.

Control hazard: also known as branch hazard, occurs when an instruction does not execute in the correct pipeline clock cycle because the one that was fetched does not need to be fetched; in other words, the instruction addresses do not flow as predicted by the pipeline. This occurs when the pipeline processor carries out a branch instruction; if the incorrect next instruction is carried out, there may be a penalty branch.

❖ **Solutions:**

Structural hazard: this type of danger is resolved by applying the Harvard structure and two distinct memories.

Data hazard:

- **Forwarding:** the idea of forwarding is to provide data to the ALU's input for instructions that come after, even if the producing instruction hasn't reached WB yet to write the memory or registers.

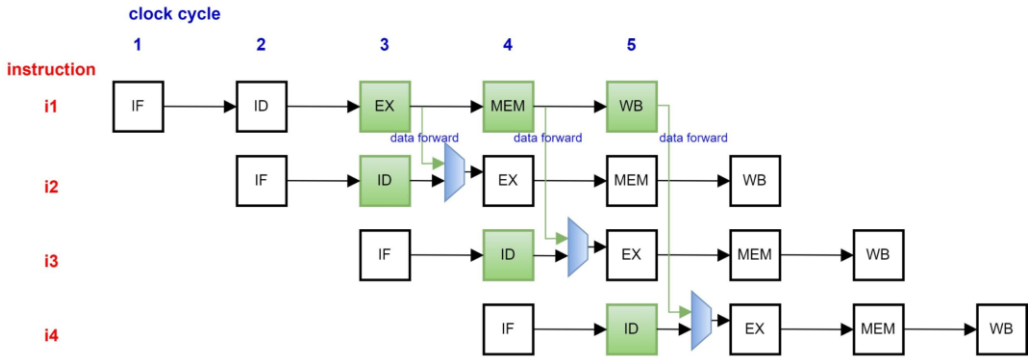


Figure 10. Data Forwarding to mitigate RAW Hazards in 5-stage Pipeline

- **Stalling:** the temporary halting of instruction flow across the pipeline. If certain dependencies prevent forwarding, the pipeline can be stopped or halted until the required data is available. This is delaying further instructions until the data is ready by inserting bubbles or "nop" (no operation) instructions.

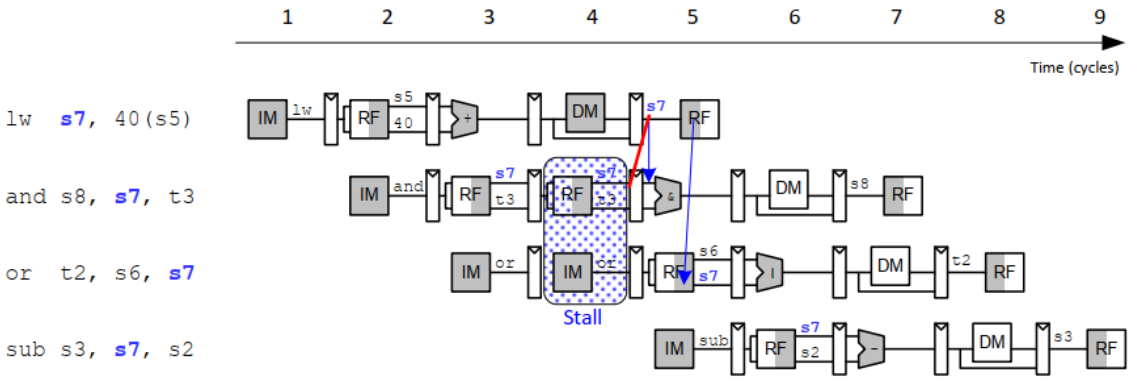


Figure 11. Example of stalling to solve lw Data Dependency

3.4 Peripheral Interfacing

3.4.1 Memory-Mapped I/O

Explanation of memory-mapped I/O.

Memory-mapped I/O involves assigning specific memory addresses to hardware peripherals so that the CPU can interact with these peripherals using standard memory read and write instructions. Instead of using dedicated I/O instructions, the peripherals are treated like memory locations.

Address mapping for peripherals.

In your module, several registers are defined to represent the addresses of different peripherals. These addresses are used to interact with the respective peripherals.

SW_ADDR (0x900): Address for the switches.

HEX0_ADDR (0x800): Address for the 7-segment LED display (HEX0).

HEX1_ADDR (0x810): Address for the 7-segment LED display (HEX1).

HEX2_ADDR (0x820): Address for the 7-segment LED display (HEX2).

HEX3_ADDR (0x830): Address for the 7-segment LED display (HEX3).

HEX4_ADDR (0x840): Address for the 7-segment LED display (HEX4).

HEX5_ADDR (0x850): Address for the 7-segment LED display (HEX5).

HEX6_ADDR (0x860): Address for the 7-segment LED display (HEX6).

HEX7_ADDR (0x870): Address for the 7-segment LED display (HEX7).

LEDG_ADDR (0x890): Address for the green LEDs.

LEDR_ADDR (0x8800): Address for the red LEDs.

LCD_DATA_ADDR (0x8A0): Address for the LCD data.

LCD_CTRL_ADDR (0x8B0): Address for the LCD control signals.

3.4.2 7-Segment LED Display Interface

The 7-segment LED display interface in our project uses memory-mapped I/O to control the display segments. Each digit of the 7-segment display is mapped to a specific address in the memory space, allowing the CPU to update the display by writing to these addresses.

Address Mapping:

Each digit of the 7-segment display is assigned a unique address:

HEX0_ADDR (0x800): Address for the 7-segment LED display (HEX0).

HEX1_ADDR (0x810): Address for the 7-segment LED display (HEX1).

HEX2_ADDR (0x820): Address for the 7-segment LED display (HEX2).

HEX3_ADDR (0x830): Address for the 7-segment LED display (HEX3).

HEX4_ADDR (0x840): Address for the 7-segment LED display (HEX4).

HEX5_ADDR (0x850): Address for the 7-segment LED display (HEX5).

HEX6_ADDR (0x860): Address for the 7-segment LED display (HEX6).

HEX7_ADDR (0x870): Address for the 7-segment LED display (HEX7).

Segment Decoding

The segment decoding logic is implemented in a separate block outside the main CPU block. This modular design approach simplifies integration and debugging.

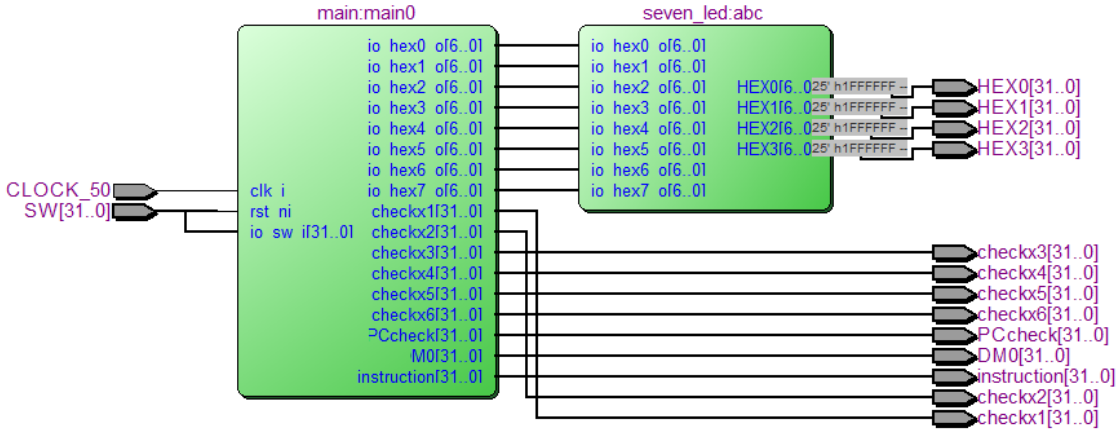


Figure 12. Block diagram of 7-segment led

The segment decoder translates binary values written to the control registers into the appropriate signals for the display segments. For instance, writing the value 0x40 to a display's address might light up the segments to display the digit "0".

3.4.3 Switches Interface

Switches Interface:

The switches interface in our project facilitates user input to control the RISC-V CPU. By integrating a single register to represent the state of the switches, the CPU can respond to user interactions such as starting, stopping, or resetting the stopwatch. This section outlines the design and implementation of the switches interface using memory-mapped I/O.

Address Mapping:

A single register is assigned for the switches interface:

SW_ADDR (32'h900): Address for the switch register.

Switch Input Handling:

The switch register holds the combined state of all switches. Writing a value to this register enables or disables the corresponding switches.

3. SIMULATION

In this project, we create several test cases to ensure that all of our instructions function correctly and produce the expected results. We generate the test cases by writing an assembly code program using the instructions of our CPU. Then, we use the Vernus website to convert the asm file to a hex file. Next, we modify the contents of the instruction.mem file. We create checkers representing the registers to verify the results of each instruction against the expected outcomes using waveforms in Quartus software. Below are some test cases:

- **Arithmetic test case:**

```
start:
# ================== Test for arithmetic ops ================
        lui x1, 0x80000             # x1                    = 0x80000000
        ori x1, x1, 0x010           # x1 = x1 | 0x010 = 0x80000010

        lui x2, 0x80000             # x2                    = 0x80000000
        ori     x2, x2, 0x001       # x2 = x2 | 0x001       = 0x80000001

        add x3, x2, x1          # x3                        = 0x00000011
        addi x3, x3, 0x0fe      # x3                        = 0x0000010f
        add x3, x3, x2          # x3                        = 0x80000110
```

```
        sub x3, x3, x2          # x3                    = 0x0000010f

# ================== Test for cmp ops ======================
        lui x1,0xffff0          # x1                    = 0xffff0000
        slt x2, x1, x0          # x2                    = 1           notice: signed
        sltu x2, x1, x0         # x2                    = 0           notice: unsigned
        lui x1, 0x00001                 # x1                      = 0x00001000
        slti x3, x1, -0x800     # x3                    = 0           notice: signed
        sltiu x3, x1, -0x800 # x3                       = 1           notice: signed extend
and unsigned comparation
```

Figure 13. Assembly code of Arithmetic

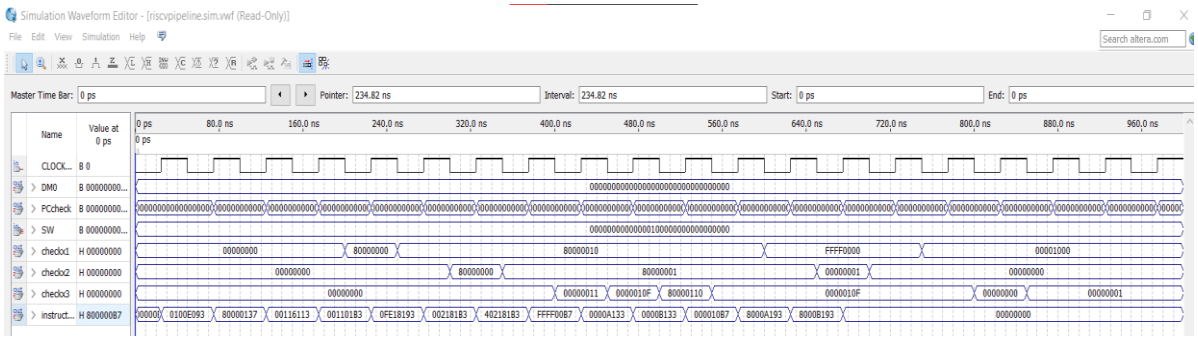


Figure 14. Arith's waveform simulation

❖ **Branches testcase**

```
_start:
        lui x3, 0x80000         # (1) x3 = 0x80000000
        ori x1, x0, 0x1         # (1) x1 = 0x00000001
        jal x0, s1              # jump to s1

1:
        ori x1, x0, 0x111
        ori x1, x0, 0x110

s1:
        ori x1, x0, 0x002       # (2) x1 = 0x00000002
        jal x5, s2              # jump to s2 and set x5 = 0x1c

        ori x1, x0, 0x110
        ori x1, x0, 0x111
        bne x1, x0, s3
        ori x1, x0, 0x110
        ori x1, x0, 0x111
```

```
s2:
        ori x1, x0, 0x003       # (3) x1 = 0x00000003
        or x2, x5, x0           # (3) x2 = 0x0000001c
        beq x3, x3, s3          # x3 == x3, jump to s3

        ori x1, x0, 0x111
        ori x1, x0, 0x110

s4:
        ori x1, x0, 0x005       # (5) x1 = 0x00000005
        bge x3, x1, s3          # (5) s3 < 0, not jump
        ori x1, x0, 0x006       # (6) x1 = 0x00000006
        bgeu x3, x1, s5         # (6) x3 > 0, jump to s5

bad:
        ori x1, x0, 0x111

s7:
        ori x1, x0, 0x010       # (10) x1 = 0x00000010
        bne x1, x3, s8          # (10) x1 != x3, jump to s8

s6:
        ori x1, x0, 0x008       # (8) x1 = 0x00000008
        blt x1, x0, bad         # (8) x1 > 0, not jump
        ori x1, x0, 0x009       # (9) x1 = 0x00000009
        bltu x1, x3, s7         # (9) x1 < x3 jump to s7

s5:
        ori x1, x0, 0x007       # (7) x1 = 0x00000007
        blt x3, x1, s6          # (7) x3 < 0, jump to s6


s3:
        ori x1, x0, 0x004       # (4) x1 = 0x00000004
        bge     x1, x0, s4      # if x1 >= x0 then s4




s8:
        ori x1, x0, 0x011       # (11) x1 = 0x00000011
        bne x1, x1, bad                 # (11) x1 = x1, not jump
        ori x1, x0, 0x012       # (12) x1 = 0x00000012

        ori x3, x0, 0x014       # (12) x3 = 0x00000014

_loop1:                         # for x1 = 0x12 to 0x14
        addi x1, x1, 0x1        # x1++
        blt x1, x3, _loop1
```

```
                                # x1 = 0x00000014
        srli x3, x1, 0x1        # x3 = x1/2 =0x0000000a

_loop2:                         # for x1 = 0x14 to -0x0a
        sub x1, x1, x3          # x1 -= x3
        bge x1, x0, _loop2
# x1 = 0xfffffff6
        nop
        nop
```

Figure 15. Assembly code of Branches

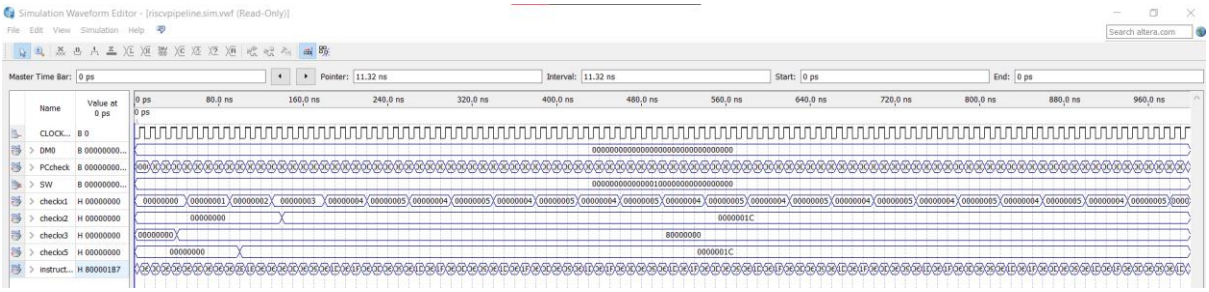


Figure 16. Branch's waveform simulation

❖ **Jal testcase**

```
_start:
# ================== Test for arithmatic ops ================
        lui x1, 0x80000                 # x1            = 0x80000000

        ori x1, x1, 0x010               # x1 = x1 | 0x010       = 0x80000010

        lui x2, 0x80000                 # x2            = 0x80000000
        ori     x2, x2, 0x001       # x2        = x2 | 0x001      = 0x80000001
_hello:
        add x3, x2, x1              # x3  = 0x00000011 (2) 0x80000110 (3)
0x0000020f
        addi x1, x3, 0x0fe          # x1        = 0x0000010f (2) 0x8000020e
        jal x4, _hello
        add x3, x3, x2              # x3        = 0x80000110

        sub x3, x3, x2              # x3        = 0x0000010f
```

Figure 17. Assembly code of Jal

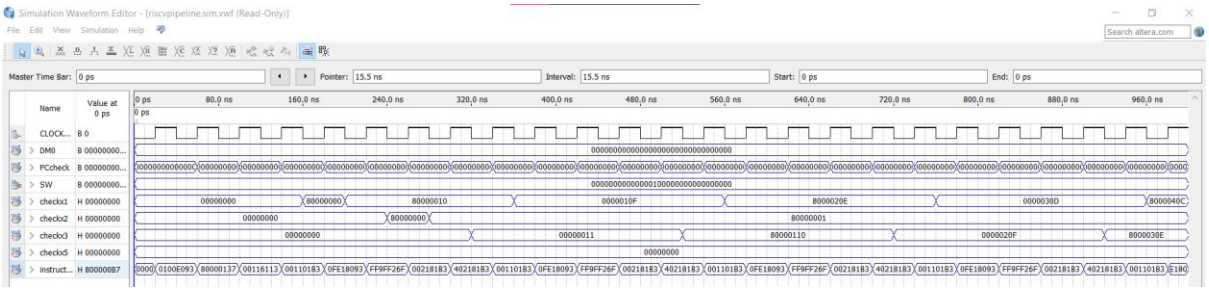


Figure 18. Jal's waveform simulation

❖ **Slt test case**

```
lui x1,0xffff0          # x1          = 0xffff0000
slt x2, x1, x0          # x2          = 1          notice: signed
sltu x2, x1, x0         # x2          = 0          notice: unsigned
lui x1, 0x00001         # x1          = 0x00001000
slti x3, x1, -0x800     # x3          = 0          notice: signed
sltiu x3, x1, -0x800    # x3          = 1          notice: signed
extend and unsigned comparation
```

Figure 19. Assembly code of Slt

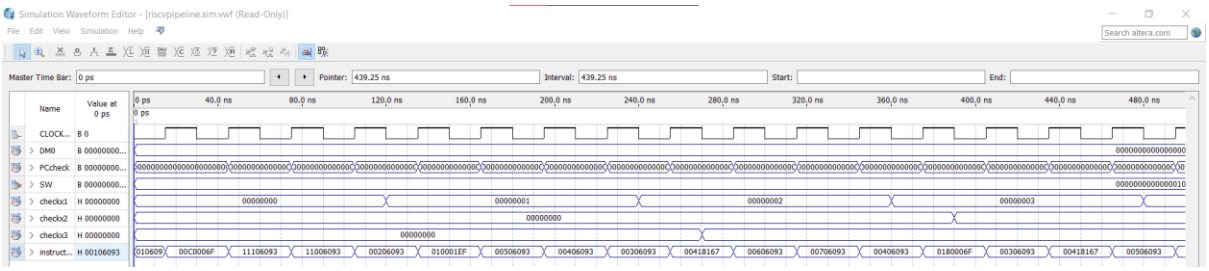
Figure 20. Slt's waveform simulation

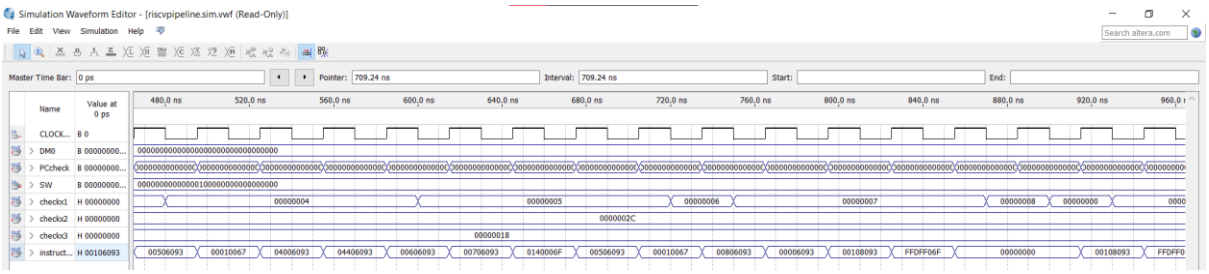
❖ **Jumps testcase**

```
_start:
ori x1, x0, 0x1                 # (1) x1 = 0x00000001
jal x0,  _j10                   # (1) jump to 0x10
ori x1, x0, 0x111
ori x1, x0, 0x110

_j10:
# addr = 0x10
ori x1, x0, 0x2                 # (2) x1 = 0x00000002
jal x3, _j24                    # jump to 0x24 and set x3 = 0x18

ori x1, x0, 0x005               # x1 = 0x00000005
ori x1, x0, 0x4                 # (4) x1 = 0x0000001c
```

```
jal x0,  _j38                                   # jump to 0x60

_j24:
# addr = 0x24
ori x1, x0, 0x3                                 # (3) x1 = 0x00000003
jalr x2, x3, 4                                  # (3) jump to 0x1c and set x2 = 2c

ori x1, x0, 0x6                                 # (6) x1 = 0x00000006
ori x1, x0, 0x7                                 # (7) x1 = 0x00000007
jal x0, _j48                                    # (6) jump to 0x80

_j38:
# addr =  0x38
ori x1, x0, 0x5                                 # (5) x1 = 0x00000005
jalr x0, x2, 0                                  # (5) jump to 0x2c

ori x1, x0, 0x040
ori x1, x0, 0x044

_j48:
# addr = 0x48
ori x1, x0, 0x8                                 # (8) x1 = 0x00000008
ori x1, x0, 0x0                                 # (8) x1 = 0x00000000

_loop:
        addi x1, x1, 0x1                        # (9+) ++x1
        jal x0,  _loop
```

Figure 21. Assembly code of Jump





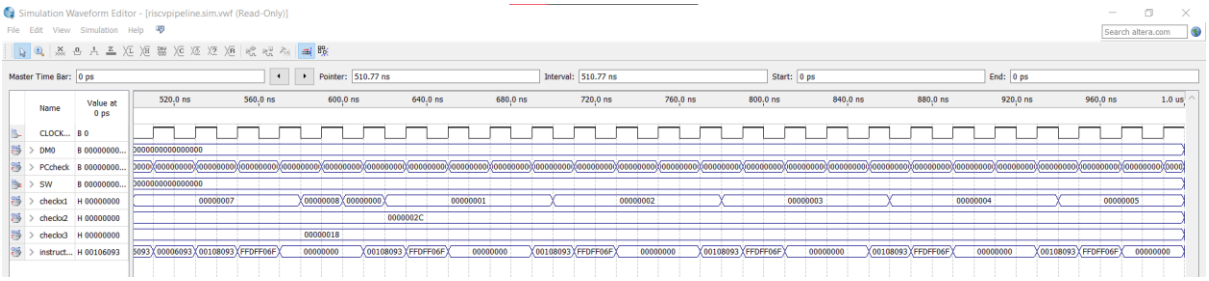


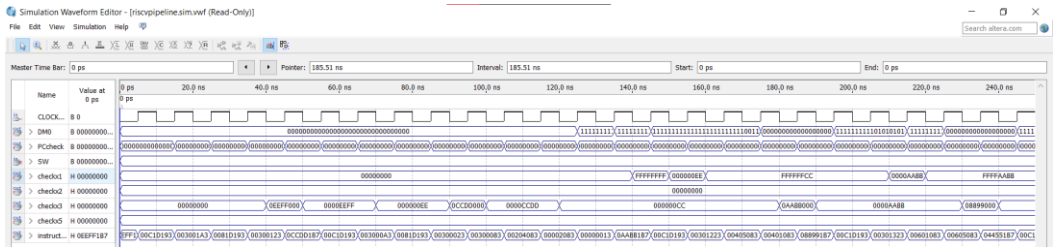
Figure 22. Jump's waveform simulation

❖ **Load_store testcase**

```
_start:
        lui     x3,0x0eeff
        srli x3, x3, 12
        sb      x3, 3(x0)           # [0x3] = 0xff
        srli  x3,x3,8
        sb      x3, 2(x0)           # [0x2] = 0xee
        lui x3,0xccdd
        srli x3, x3, 12
        sb      x3, 1(x0)           # [0x1] = 0xdd
        srli  x3,x3,8
        sb      x3, 0(x0)           # [0x0] = 0xcc
        lb      x1, 3(x0)           # x1 = 0xffffffff
        lbu  x1, 2(x0)              # x1 = 0x000000ee
        lw      x1, 0(x0)           # x1 = 0xffeeddcc
        nop

        lui x3,0xaabb
        srli x3, x3, 12
        sh      x3, 4(x0)           # [0x4] = 0xbb, [0x5] = 0xaa
        lhu  x1, 4(x0)              # x1 = 0x0000aabb
        lh      x1, 4(x0)           # x1 = 0xffffaabb

        lui x3,0x8899
        srli x3, x3, 12
        sh      x3, 6(x0)           # [0x6] = 0x99, [0x7] = 0x88
        lh      x1, 6(x0)           # x1 = 0xffff8899
        lhu  x1, 6(x0)              # x1 = 0x00008899

        lui x3,0x4455
        srli x3, x3, 12
        slli  x3,x3,0x10
        lui x2, 0x6677
        srli x2, x2, 12
        or x3, x2, x3               # x3 = 0x44556677
        sw      x3, 8(x0)           # [0x8] = 0x77, [0x9]= 0x66, [0xa]=
0x55, [0xb] = 0x44
        lw      x1, 8(x0)           # x1 = 0x44556677
```

```
_loop:
        jal x0, _loop
```

Figure 23. Assembly code of load_store



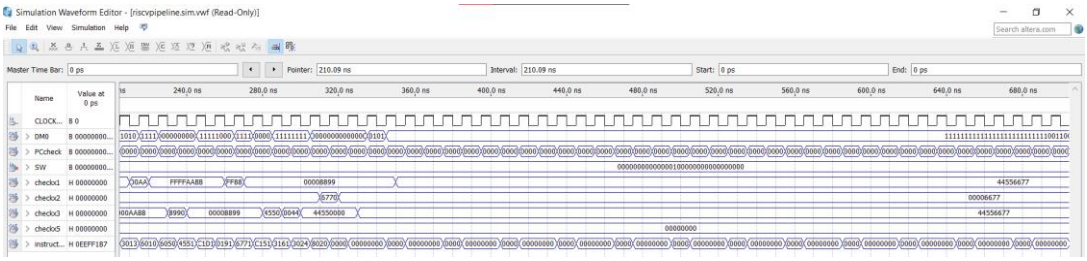


Figure 24. Load_store's waveform simulation

- **Load_stall testcase:**

```
_start:
   ori x1,x0,0x234    # x1 = 0x00000234
   sw  x1,0x0(x0)      # [0x0] = 0x00000234

   ori x2,x0,0x0234    # x2 = 0x00000234
   ori x1,x0,0x0       # x1 = 0x0
   lw  x1,0x0(x0)      # x1 = 0x00000234
   beq x1,x2,Label

   ori x1,x0,0x567

Label:
   ori x1,x0,0x7ab    # x1 = 0x00007ab

_loop:
   jal x0, _loop
   nop
```

Figure 25. Assembly code of load_stall

Figure 26. Load_stall's waveform simulation

❖ **Opi_and_op testcase**

```
_start:
        lui x1, 0x10100              # x1 = 0x10100000
        ori x1, x1, 0x101      # x1 = x1 | 0x101      = 0x10100101
        ori x2, x1, 0x010      # x2 = x1 | 0x010      = 0x10100111
        or x1, x1, x2          # x1 = x1 | x2         = 0x10100111
        andi x3, x1, 0x0ce     # x3 = x1 & 0x0ce      = 0x00000000
        and x1, x3, x1         # x1 = x3 & x1         = 0x00000000
        xori x4, x1, 0x7f0     # x4 = x1 ^ 0x7f0      = 0x000007f0
        xori x1, x4, 0x0f0     # x1 = x4 ^ 0x0f0      = 0x00000700
        xor x1, x4, x1         # x1 = x4 ^ x1         = 0x000000f0
```

Figure 27. Assembly code of Opi_and_op

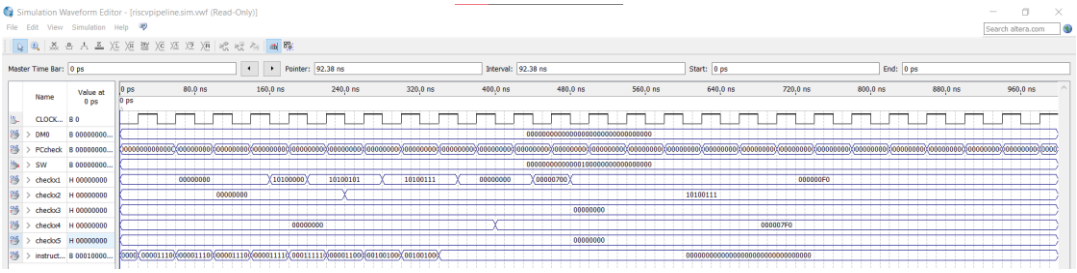


Figure 28. Opi_and_op's waveform simulation

❖ **Ori_forwarding testcase**

```
_start:
        ori x1, x0, 0x110 # x1 = 0x110
        ori x1, x1, 0x002 # x1= 0x112
        ori x1, x1, 0x440 # x1= 0x552
        ori x1, x1, 0x004 # x1 = 0x556
```

Figure 29. Assembly code of ori_forwarding

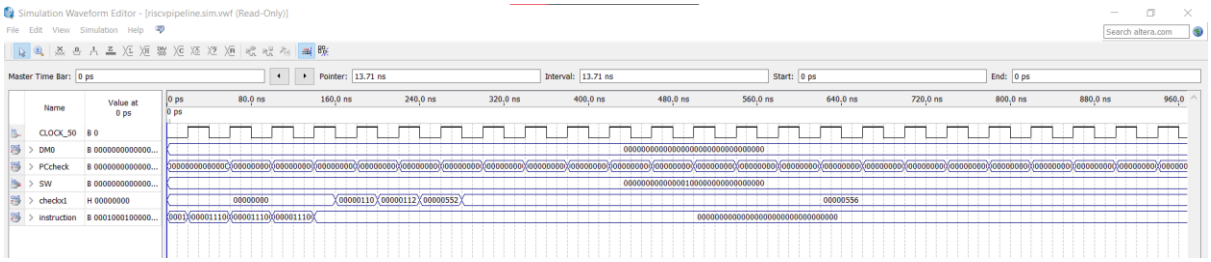


Figure 30. Ori_forwarding's waveform simulation

4. APPLICATION

We designed a stopwatch on the FPGA DE2 kit. We created an assembly program that performs the stopwatch function, using instructions from our CPU. In the assembly program, we count up HEX0, HEX2 from 0 to 9 and count up HEX1, HEX3 from 0 to 5. We use a delay subroutine between number changes. We use the state of Switch 0 as the Stop signal. If Switch0 is on, the program will enter a loop to wait until Switch 0 is off before returning to the main program to continue counting up. Below is the assembly program:

```
addi x10, x0, 0x80 // adrress of HEX0 is 0x800
slli x10, x10, 4

addi x11, x0, 0x81 // adrress of HEX1 is 0x810
slli x11, x11, 4

addi x12, x0, 0x82 // adrress of HEX2 is 0x820
slli x12, x12, 4

addi x13, x0, 0x83 // adrress of HEX3 is 0x830
slli x13, x13, 4

addi x14, x0, 0x90 // adrress of SW is 0x900
slli x14, x14, 4
addi x5, x0, 9
addi x15,x0,  6
addi x7, x0, 1

HEX0:
    lw x6, 0(x14)   // Update of Switch status
    and x8, x6, x7  // check switch_stop status
    beq x8, x7, STOP  // If switch_stop is on, jmp to STOP
CON:
    addi x1, x1, 1  // count up HEX0
    sw x1, 0(x10)  // display number on HEX0
    jal x30, DELAY_1s  // delay 1s
```

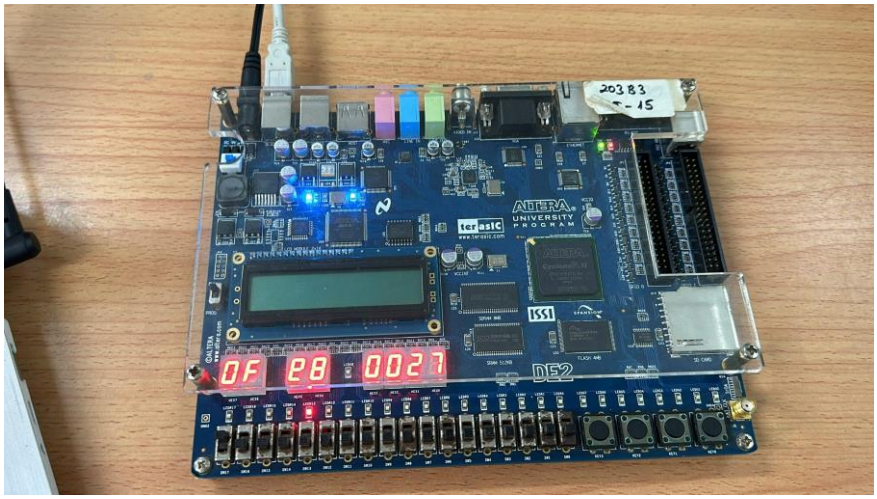
```
CON1:
    blt x1, x5 , HEX0   // if number of HEX0 is less than 10, jump to HEX
    addi x1, x0, 0
    sw x1, 0(x10)  // replace number of HEX0 by 0
HEX1:
    addi x2, x2, 1  // count up HEX1
    sw x2, 0(x11)  // display number on HEX1
    jal x30, DELAY_1s
    blt x2, x15, HEX0 // if number of HEX1 is less than 6, jump to HEX0
    addi x2, x0,  0
    sw x2, 0(x11)   // replace number of HEX1 by 0
HEX2:

    addi x3, x3, 1    // count up HEX2
    sw x3, 0(x12)   // display number on HEX2
    jal x30, DELAY_1s
    blt x3, x5, HEX0   // if number of HEX1 is less than 10, jump to HEX0
    addi x3, x0,  0
    sw x3, 0(x12)    // replace number of HEX1 by 0
HEX3:
    addi x4, x4, 1    // count up HEX3
    sw x4, 0(x13)    // display number on HEX3
    jal x30, DELAY_1s
    blt x4, x15, HEX0   // // if number of HEX1 is less than 6, jump to HEX0
    addi x1, x0, 0
    sw x1, 0(x10)    // replace number of HEX0 by 0
     jal x30, DELAY_1s
    addi x2, x0,  0
    sw x2, 0(x11)   // replace number of HEX1 by 0
     jal x30, DELAY_1s
    addi x3, x0,  0
    sw x3, 0(x12)    // replace number of HEX2 by 0
    jal x30, DELAY_1s
    addi x4, x0, 0
    sw x4, 0(x13)   // replace number of HEX3 by 0
    jal x30, DELAY_1s

STOP:
    lw x6, 0(x14)   // Update of Switch status
    and x8, x6, x7  // check switch_stop status
    beq x8, x7, STOP  // If switch_stop is ON, jump to STOP
    jal x30, CON   // If switch_stop is OFF, jump to main program

DELAY_1s:
   addi x20, x0, 0xF0
   slli x20, x20, 4
LOOP1:
   addi x21, x0, 0xFF
   slli x21, x21, 4
```

```
LOOP2:
    sub x21, x21, x7
    bne x21, x0, LOOP2
    sub x20, x20, x7
    bne x20, x0, LOOP1
    jalr x0, x30, 0
```

Figure 31. Assembly code of stop-watch

5. RESULT

The implementation of the stopwatch on the FPGA DE2 Kit was successful. A video demonstration showcasing the functionality of the stopwatch can be found in the link below:



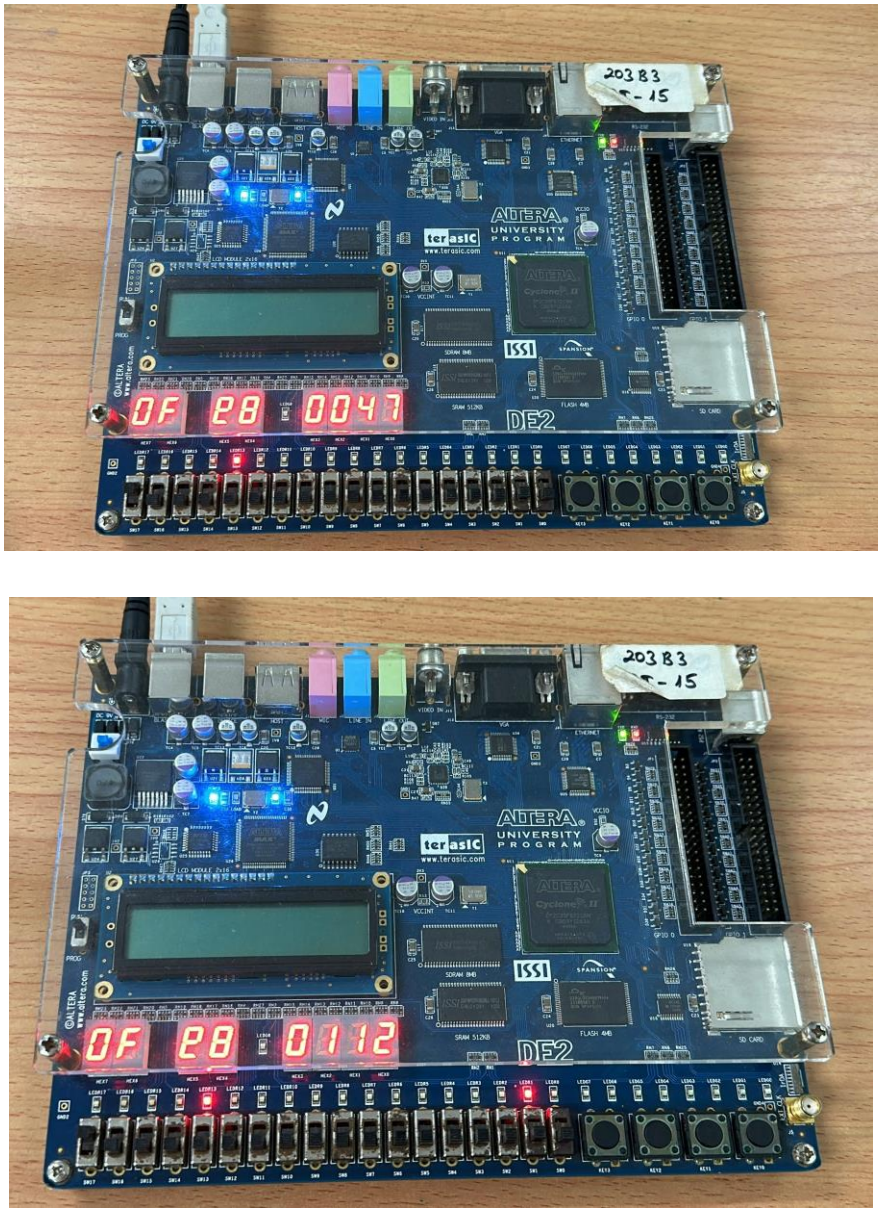


Figure 32. Simulation stop-watch on kit DE2

The stopwatch program effectively demonstrates the integration of the RISC-V pipelined CPU with peripheral interfacing, allowing user interaction through switches and displaying elapsed time on 7-segment LED displays. This practical application highlights the versatility and functionality of the CPU design.

Evaluation of Team Members

Both team members contributed equally to the success of the project. Nguyễn Chí Bảo (2151175) demonstrated strong research skills, particularly in understanding the RISC-V architecture and designing the CPU pipeline stages. Nguyễn Quốc Gia Bảo (2151176) excelled

in Verilog coding and FPGA implementation, ensuring the seamless integration of hardware components and peripheral interfacing.

Throughout the project, effective communication and collaboration were maintained, facilitating problem-solving and knowledge sharing. Both team members exhibited dedication and enthusiasm, resulting in a well-executed project.

6. CONCLUSION AND DEVELOPMENT DIRECTION

6.1 Conclusion

This project has successfully achieved its objectives of designing and implementing a RISC-V pipelined CPU on an FPGA, culminating in the development of a functional stopwatch application. Through the integration of a 5-stage pipeline architecture, advanced hazard handling techniques, and peripheral interfacing, the CPU demonstrated efficient instruction execution and enhanced functionality.

The project results, as showcased in the video demonstration, highlight the practical application of the CPU design in real-world scenarios. The stopwatch program effectively utilizes the CPU's capabilities, interacting with external devices and providing accurate timing functionality.

Throughout the project implementation, valuable experiences were gained, including in-depth research into RISC-V architecture, Verilog coding, FPGA implementation, and project management. The strengths of the project lie in the comprehensive understanding and execution of CPU design principles, effective hazard handling, and successful peripheral interfacing.

However, there were also challenges encountered, particularly in optimizing performance and addressing hardware limitations. Despite these challenges, the project outcomes align closely with the objectives set in Chapter 1, demonstrating the successful design and implementation of a RISC-V pipelined CPU on FPGA.

In conclusion, this project serves as a testament to the capabilities of modern FPGA-based systems and the potential for practical applications of CPU design principles. The experiences gained from this project will undoubtedly contribute to the ongoing pursuit of innovation and excellence in the field of digital system design.

6.2 Development

Explore performance optimization techniques such as out-of-order execution and superscalar architecture. Investigate enhancements to peripheral interfacing, including support for UART, SPI, and I2C. Extend the instruction set to include floating-point arithmetic and vector processing. Implement power-saving measures for improved energy efficiency. Incorporate hardware debugging features for easier testing. Explore multi-core architectures for concurrent task execution. Integrate encryption and secure boot mechanisms for enhanced security. Investigate hardware accelerators for AI algorithms. Develop compatibility with real-time operating systems for multitasking. Foster collaboration with the open-source community for collective improvement.

7. REFERENCE

[1] David Money Harris, Sarah L. Harris (2007). *Digital Design and Computer Architecture*

[2] CS Division, EECS Department, University of California, Berkeley (20119). *The RISV-V Intruction Set Manual*

[3] Chipmunk 8 July 2023, *Designing RISC-V CPU from scratch – Part 3: Dealing with Pipeline Hazards*. https://chipmunklogic.com/digital-logic-design/designing-pequeno-risc-v-cpu-from-scratch-part-3-dealing-with-pipeline-hazards/