Internship for SEMICONDUCTOR TECHNOLOGIES VIETNAM

Design and Implementation of 32-bit RISC Processor

Instructor: Trung Nguyen-Duy
Student: Manh Tran-Duc - 2114026

ABSTRACT

This project focuses on the design and simulation of a MIPS (Microprocessor without Interlocked Pipeline Stages) processor using the Verilog hardware description language. MIPS processors are a family of RISC (Reduced Instruction Set Computer) architectures known for their simplicity and efficiency, making them a popular choice in both academic and commercial settings.

The MIPS processor is implemented using a single-cycle architecture to simplify the design and execution flow. The key modules implemented include the Arithmetic Logic Unit (ALU), ALU Control, Control unit, Instruction Memory (IMEM), Data Memory (DMEM), Register File (REG), and multiplexers for Program Counter (PC) and result selection. The functionality of these modules was simulated and verified using the ModelSim software.

Finally, the individual modules were integrated into a main risc_single module, and a simple testbench was written to verify the overall functionality of the MIPS-processor.

This project aims to provide a comprehensive understanding of MIPS processor design and the Verilog implementation, which can be valuable for both academic and practical applications in the field of computer architecture and digital design.

Contents

List of Figures

List of Tables

1 Overview

1.1 Introduction

The MIPS (Microprocessor without Interlocked Pipeline Stages) processor is a widely studied and implemented RISC (Reduced Instruction Set Computing) architecture known for its simplicity, efficiency, and scalability. Developed in the 1980s, MIPS has played a significant role in the evolution of computer architecture, offering a foundation for both academic instruction and commercial applications. This report explores the design principles, architectural features, and practical applications of the MIPS processor, highlighting its relevance in modern computing environments.

1.2 Application

32-bit MIPS processors are widely used in various applications, particularly in embedded systems. Some common applications include:

- **Networking Equipment:** : MIPS processors are commonly used in routers and switches due to their efficiency and reliability.
- **Consumer Electronics**: Devices like set-top boxes, digital TVs, and DVD players often incorporate MIPS processors.
- **Automotive Systems**: They are used in automotive control systems for their robustness and low power consumption.
- **Industrial Control**: Mips processor are found in industrial automation and control systems, where stability and efficiency are crucial.

2 MIPS Instruction

MIPS instructions are categorized into three main types: R-type (Register), I-type (Immediate), and J-type (Jump) instructions. Each type serves a specific purpose within the architecture, enabling efficient data processing, memory access, and program control. The simplicity and efficiency of these instruction types make MIPS a popular choice for educational purposes and a solid foundation for understanding computer architecture.

Figure 1: Mips Instruction Format

32 bits Mips include:

- **Op (opcode):** Instruction code, used to determine the execution instruction (for R-type, Op = 0).
- **Rs, Rt, Rd (register):** Identifies the registers (5-bit). For example, Rs = 4 means Rs is using register a0 or register 4.
- **Shamt (shift amount):** Specifies the number of bits to shift in shift instructions.
- **Immediate:** Represents a direct number, address, or offset.
- **Funct:** Specifies the exact operation to perform (e.g., add, subtract) in R-type instructions when Op = 0.
- **Op (opcode):** Instruction code, used to determine the execution instruction (for R-type, Op = 0).
- **Rs, Rt, Rd (register):** Identifies the registers (5-bit). For example, Rs = 4 means Rs is using register a0 or register 4.
- **Shamt (shift amount):** Specifies the number of bits to shift in shift instructions.
- **Immediate:** Represents a direct number, address, or offset.
- **Funct:** Specifies the exact operation to perform (e.g., add, subtract) in R-type instructions when Op = 0.

The table below shows the instructions that my processor can execute:

NAME	FORMAT	OPERATION	OPCODE / FUNCT
add	R	R[rd] = R[rs] + R[rt]	0_hex / 20_hex
and	R	R[rt] = R[rs]&R[rt]	0_hex / 24_hex
or	R	R[rd] = R[rs]|R[rt]	0_hex / 25_hex
sub	R	R[rd] = R[rs] − R[rt]	0_hex / 22_hex
slt	R	R[rd] = (R[rs] < R[rt])?1 : 0	0_hex / 2A_hex
addi	I	R[rt] = R[rs] + SignExtImm	8_hex
andi	I	R[rt] = R[rs]&SignExtImm	c_hex
slti	I	R[rt] = (R[rs] < SignExtImm)?1 : 0	a_hex
lw	I	R[rt] = Mem[R[rs] + SignExtImm]	23_hex
sw	I	Mem[R[rs] + SignExtImm] = R[rt]	2b_hex
beq	I	if(R[rs] == R[rt])PC = PC + 4 + BranchAddr	4_hex
j	J	PC = JumpAddr	2_hex

Table 1: MIPS Instructions

3 Functional implementation

- The single-cycle microarchitecture executes an entire instruction in one cycle. It is easy to explain and has a simple control unit. Because it completes the operation in one cycle, it does not require any nonarchitectural state.
- **Advantages:** One instruction is executed in one cycle. The processor is simple, efficient, and easy to implement.
- **Disadvantages:** One cycle takes a lot of time, and all instructions, whether fast or slow, are executed in one cycle, resulting in low efficiency
- The functional implementation of the MIPS processor can be described as follows:

  1. The controller module decodes the instruction fetched from the external memory and generates the necessary control signals.
  2. The datapath module uses these control signals to perform the required operations, such as reading/writing registers, executing ALU operations, and accessing memory.
  3. The results of the datapath operations are then used to update the processor state (e.g., writing back to the register file, updating the PC) and/or interact with the external memory.

  The modular design of the MIPS processor allows for easier implementation, testing, and optimization. The controller and datapath modules can be developed and verified independently, and the external memory interface can be adapted to different memory architectures as needed.

4 Interface

The instruction and data memories are separated from the main processor and connected by address and data busses. This is more realistic, because most real processors have external memory. It also illustrates how the processor can communicate with the outside world.

The processor is composed of a datapath and a controller. The controller, in turn, is composed of the Control and the ALUControl. Figure 2 shows a block diagram of the single-cycle MIPS processor interfaced to external memories.

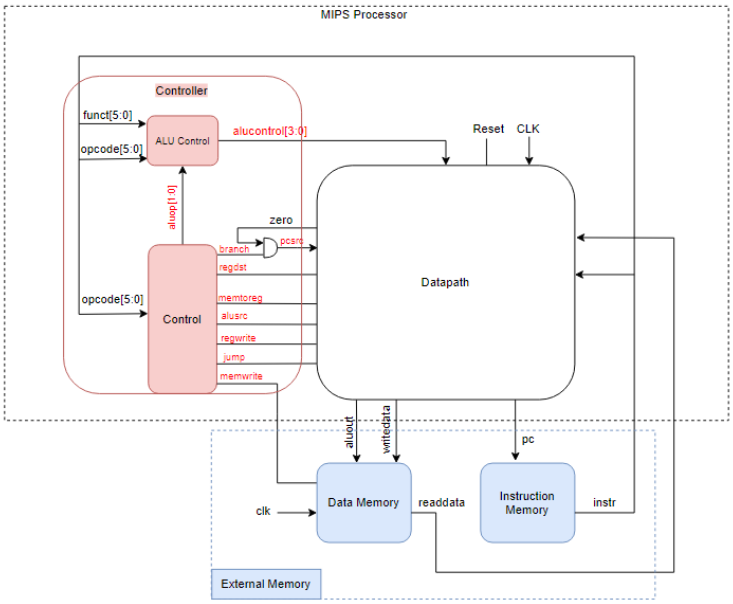

Figure 2: MIPS single-cycle processor interfaced to external memory

4.1 Controller

4.1.1 Control

The Control unit is the main component of the controller that generates the control signals for the entire datapath based on the instruction being executed. Its key functions include:

- Decoding the instruction opcode to determine the type of instruction (e.g., R-type, I-type, branch, jump).
- Generating the control signals for the datapath components, such as:
  - Register file read/write signals
  - Memory read/write signals
  - ALU operation selection
  - PC update logic
- Handling control flow instructions (branches and jumps) by updating the Program Counter (PC) accordingly.

The Control unit takes the instruction opcode as input and generates a set of control signals that are used to control the operation of the datapath components throughout the instruction execution process.

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	11	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	0	1	0	1	0	00	0
beq	000100	0	0	0	1	0	0	01	0
addi	001000	1	0	1	0	0	0	10	0
andi	001100	1	0	1	0	0	0	10	0
ori	001101	1	0	1	0	0	0	10	0
slti	001010	1	0	1	0	0	0	10	0
j	000010	0	0	0	0	0	0	0	1

Table 2: Control Unit truth table

4.1.2 ALUControl

The ALU Control is responsible for generating the appropriate control signals for the Arithmetic Logic Unit (ALU) based on the instruction being executed. Its main functions are:

- Decoding the ALU operation field from the instruction opcode.
- Generating the ALU control signals that specify the operation to be performed by the ALU (e.g., addition, subtraction, bitwise AND, bitwise OR).
- Handling special cases, such as determining the correct ALU operation for branch instructions.

The ALU Control unit takes the instruction opcode and the function field (for R-type instructions) as inputs, and outputs the ALU control signals that are used to configure the ALU's operation.

ALUOp meaning:

ALUOp	Meaning
00	add
01	subtract
10	Look at opcode bit
11	Look at funct fields

Instruction	Opcode	ALUOp	Operation	Funct	ALU Function	ALU Control
lw	100011	00	load word	XXXXXX	add	0010
sw	101011	00	store word	XXXXXX	add	0010
beq	000100	01	branch equal	XXXXXX	subtract	0110
addi	001000	10	add immediate	XXXXXX	add	0010
andi	001100	10	AND immediate	XXXXXX	AND	0000
ori	001101	10	OR immediate	XXXXXX	OR	0001
slti	001010	10	set-on-less-than immediate	XXXXXX	subtract	0111
R-type	000000	11	add	100000	add	0010
			subtract	100010	subtract	0110
			AND	100100	AND	0000
			OR	100101	OR	0001
			set-on-less-than	101010	set-on-less-than	0111

Table 3: ALUControl truth table:

4.1.3 Controller

The controller (composed of the ALU Control and the Control unit) operates with the datapath to ensure the correct execution of instructions. The typical sequence of operations is as follows:

- The instruction opcode is fetched from the Instruction Memory and passed to the controller.
- The Control unit decodes the opcode, generates the necessary control signals for the datapath, and coordinates the execution of the instruction.
- The ALU Control decodes the opcode and funct and generates the appropriate ALU control signals.
- The control signals from the controller are used to configure the datapath components (e.g., register file, ALU, memory) to perform the required operations.
- The execution of the instruction is completed, and the controller prepares for the next instruction.

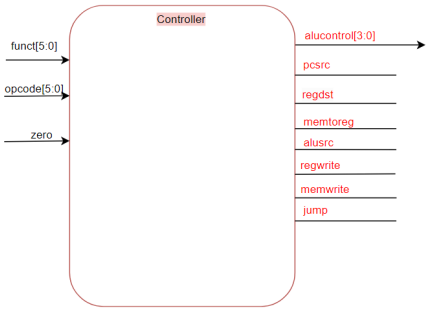

Figure 3: Controller Interface

4.2 Datapath

The datapath of the processor consists of four main components: the Program Counter (PC), the Arithmetic Logic Unit (ALU), the Register File, and the overall Datapath.

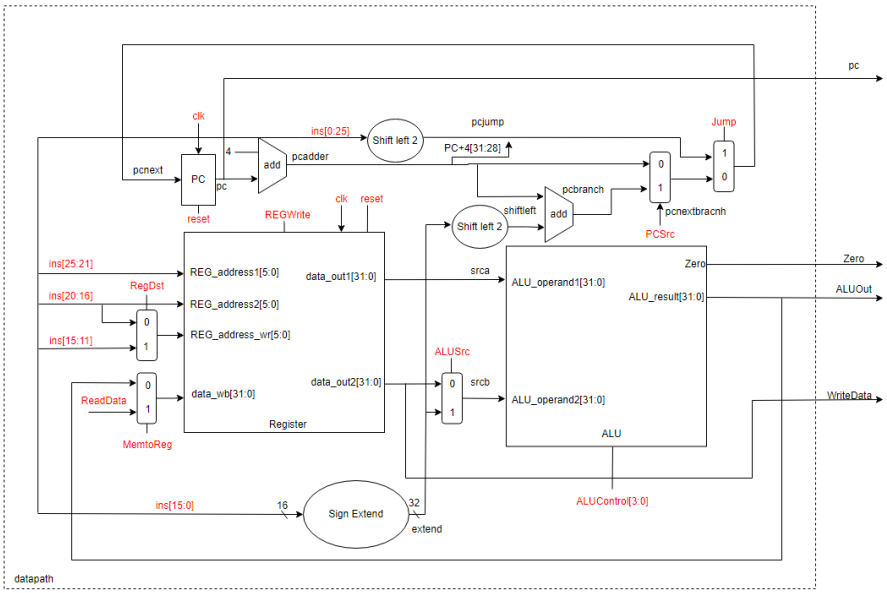

Figure 4: Datapath Block Diagram

4.2.1 PC module desgin

The Program Counter (PC) is a 32-bit register that holds the address of the current instruction being executed. The PC is updated during the instruction fetch stage based on the control signals from the controller. The PC can be updated in the following ways:

- Increment by 4 to fetch the next sequential instruction.
- Load a new value from the datapath for branch and jump instructions.
- Load a new value from the reset signal during processor initialization.

The PC provides the address to the Instruction Memory, which fetches the next instruction to be executed.

4.2.2 ALU module design

A simple ALU contains arithmetic and logical operations include:

- **Memory reference**: lw, sw
- **Arithmetic/logical**: add, sub, and, or, slt
- **Control transfer**: beq, j

The ALU takes two 32-bit operands and an operation control signal as inputs, and produces a 32-bit result. It also generates flags, such as zero, carry, and overflow, which are used by the controller for conditional branch instructions.

4.2.3 Register module design

- The Register File is a collection of 32 general-purpose 32-bit registers. It provides two read ports and one write port to support the operand fetch and write-back stages of the instruction execution.
- The Register File is implemented using a 2-to-1 multiplexer for each read port, allowing the selection of the appropriate register based on the control signals. The write port is controlled by the control signals to update the destination register with the result of the instruction.

4.2.4 Datapath

- The Datapath is the overall structural component that connects the PC, ALU, and Register File, as well as other components, to facilitate the execution of instructions. The Datapath provides the necessary data paths, control signals, and communication channels to enable the processor to fetch, decode, execute, and write back the results of instructions.

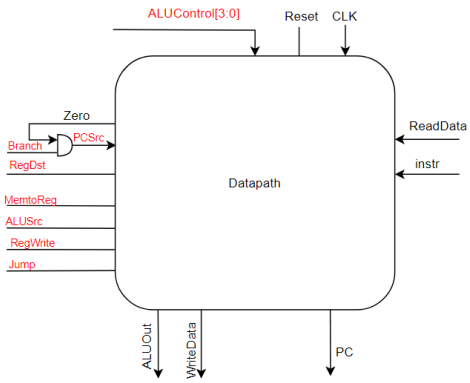

Figure 5: Datapath Interface

4.3 External Memory

4.3.1 Module Data Memory

- Data memory stores and retrieves data actively processed by the ALU. It has three inputs: **memwrite**, **data_address**, and **write_data**. The memory address (**data_address**) comes from the ALU's output, while the data to be stored (**write_data**) is provided by the register file. **memwrite** controls whether data is written to memory.

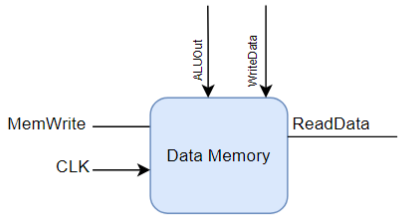

Figure 6: Data Memory module design

4.3.2 Module Instruction Memory

- In computer architecture, instruction memory is a component of the computer system that stores the instructions of a program. In this module, a 32-bit instruction set is generated and stored in a RAM array. The instruction to be fetched is based on the input of the Program Counter (PC).

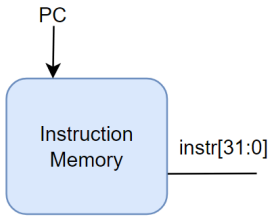

Figure 7: Instruction Memory module design

5 Synthesis

5.1 Controller module

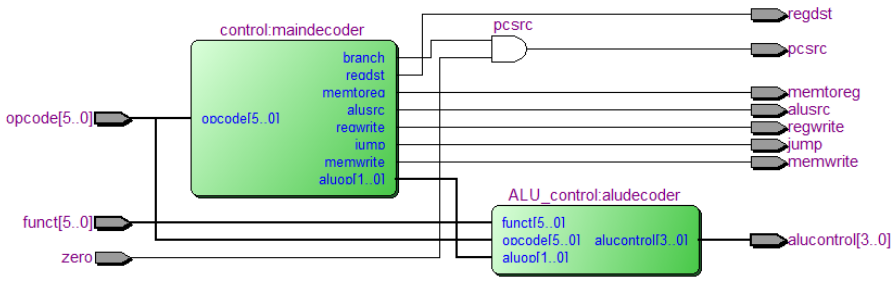

Figure 8: Controller module

5.2 Datapath module

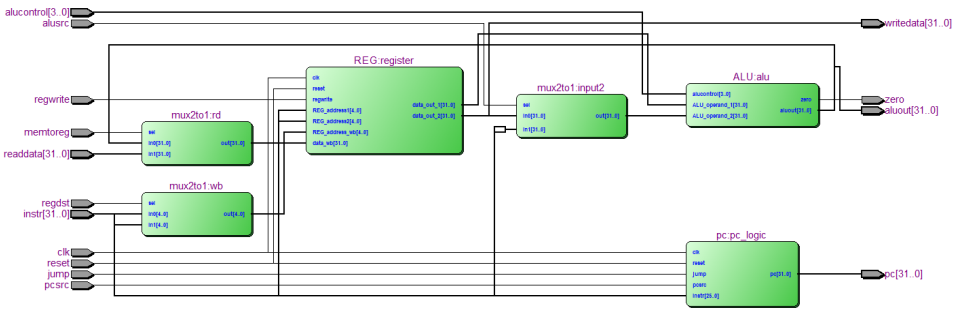

Figure 9: Datapath module

5.3 Mips module

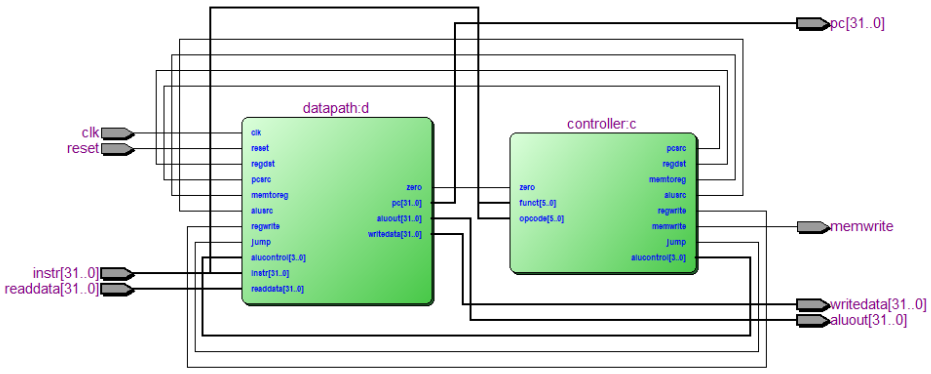

Figure 10: Mips module

5.4 Top module

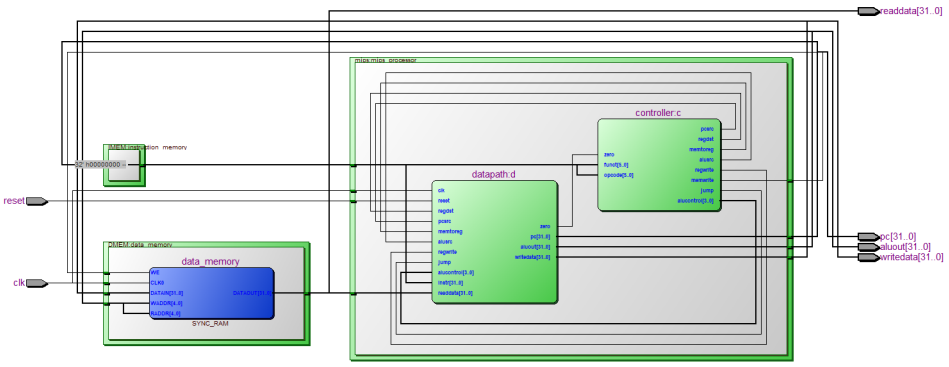

Figure 11: Top module

6 Simulation and analysis

The testbench **top_tb** is designed to verify the execution of arithmetic, logical, branch, memory, and jump instructions, as well as the reset functionality in the MIPS processor implemented in the top module. It ensures that the processor correctly performs arithmetic and logical operations, handles memory access operations (including loading from and storing to memory), accurately evaluates branch conditions, and updates the program counter (PC) as required for both branch and jump instructions. Additionally, it checks that all registers are cleared and the PC is reset to 0 when the reset signal is active.

Test case 1: Check the reset signal in the system and the system with arthematic and logic include: R-type and Immediate instruction:

Assembly	Description	Address
add R3, R2, R4	R3 = R2 + 4 = 0	0x00
addi R2, R1, 2	R2 = R1 + 2 = 2	0x04
ori R3, R1, 4	R3 = R1 | 4 = 4	0x08
slti R4, R1, 3	R4 = (R1 < 15) ? 1 : 0 = 1	0x0c
add R3, R4, R2	R3 = R4 + R2 = 3	0x10
sub R1, R3, R4	R1 = R3 - R4 = 2	0x14
or R3, R1, R2	R3 = R1 or R2 = 2	0x18
and R3, R1, R2	R3 = R1 & R2 = 2	0x1c
slt R3, R1, R2	R3 = (R1 < R2) ? 1 : 0 = 0	0x20

Table 4: Instruction in test case 1

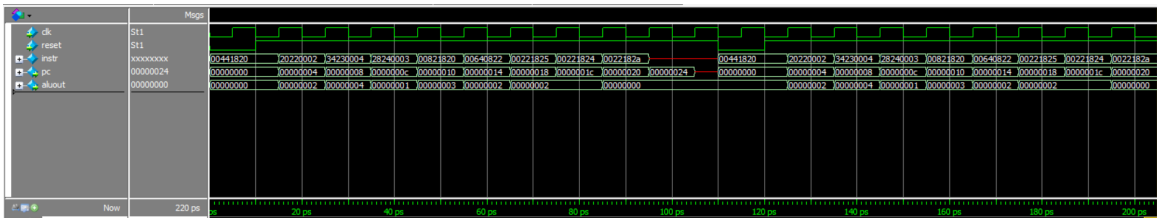

Figure 12: Waveform for arithmetic/logical and reset state

Simulation Analysis:

- **0-10 ns:** During this period, the reset signal is active, which sets the PC to 0 and clears all the registers. Consequently, **aluout** remains 0.
- **15-45 ns:** The Immediate (I-type) instructions are executed to initialize data into the registers: `R2 = 2`, `R3 = 4`, `R4 = 1`.
- **45-95 ns:** R-type instructions are executed to verify that the data written to the registers by the I-type instructions is correct. These instructions ensure that the values are written to the correct addresses and that the functions are executed accurately. The **aluout** and **pc** values are compared with the expected values shown in Table 4)
- **110-120 ns:** The reset signal is activated again to check whether the PC is reset to 0. The instruction at address 0x00, which is an R-type instruction, is executed to confirm that the values in the registers are reset to 0.
- **130 ns - end:** Repeat the previous instruction

Test case 2: Check the system with memory instruction(lw,sw) and branch case and jump case:

Assembly	Description	Address
addi R2, R0, 5	initialize R2 = 5	0x00
addi R4, R0, 10	initialize R4 = 10	0x04
addi R6, R0, 15	initialize R6 = 15	0x08
sw R2, 4(R1)	MEM[4 + R1] = R2 = 5	0x0c
lw R3, 4(R1)	R3 = MEM[4 + R1] = 5	0x10
sw R4, 8(R3)	MEM[8 + R3] = R4 = 10	0x14
lw R5, 8(R3)	R5 = MEM[8 + R3] = 10	0x18
sw R6, 12(R5)	MEM[12 + R5] = R6	0x1c
lw R7, 12(R5)	R7 = MEM[12 + R5] = 15	0x20
beq R2, R4, offset 2	Not Equal so PC = 0x28	0x24
beq R2, R3, offset 2	Equal so PC = PC + 4 + BranchAddr = 0x34	0x28
addi R3, $0, 12	Shouldn't happen	0x2c
addi R7, R3, -9	Shouldn't happen	0x30
j address 1	PC = {PC+4[31:28], address, 2'b0} = 0x04	0x34

Table 5: Instruction in test case 2

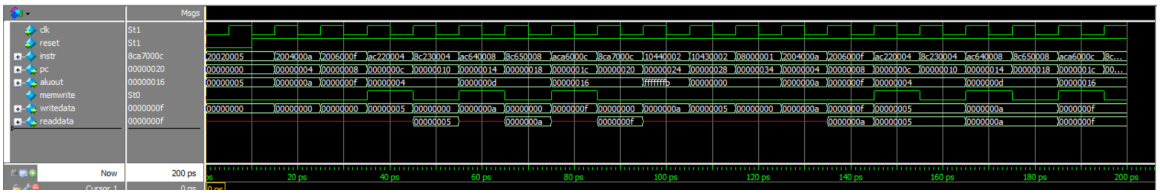

Figure 13: Waveform for Memory Instruction and branch/jump instruction

Simulation Analysis:

- **0-25 ns:** The simulation begins by using I-type instructions to initialize data in the registers: `R2 = 5`, `R4 = 10`, `R6 = 15`.
- **35-55 ns:** Memory instructions are tested for storing and loading data. At **35 ns memwrite** active, `writedata` equals `5` (the value of `R2`), confirming that the `sw` instruction correctly stores the value at the appropriate memory address. At **45 ns**, `readdata` equals `5`, verifying that the value was correctly loaded back from memory, ensuring the `lw` instruction works properly.
- **55-75 ns:** Further testing of memory instructions continues. At **55 ns memwrite** active, `writedata` equals `0x0a` (the value of `R4`), and at **75 ns**, `readdata` equals `0x0a`, confirming that the data was correctly stored and loaded.
- **75-95 ns:** Similar testing is performed for `R6`. At **75 ns memwrite** active, `writedata` is `0x0f` (the value of `R6`), and at **95 ns**, `readdata` is `0x0f`, verifying correct memory operations.
- **95-105 ns:** The ALU performs a subtraction: `aluout = R2 - R4 = 0xffffffff...`, which is not equal to zero, so the PC increments by 4 to `0x28`. This confirms the PC update for the case when the branch condition is not met.
- **105-115 ns:** The ALU performs another subtraction: `aluout = R2 - R2 = 0`, which is equal to zero, so the PC is updated to `PC + 4 + BranchAddr = 0x34`. This confirms that the branch instruction (`beq`) is functioning correctly.
- **115-125 ns:** A jump instruction is executed, setting the PC to `0x04` by calculating `PC = {PC+4[31:28], address, 2'b0}`.
- **125 ns - End:** The instructions repeat, starting from address `0x04`.

Test case 3: Verify the complete integration of all modules in the top-level design.

#	Assembly	Description	Address
main:	addi R2, R0, 5	initialize R2 = 5	0
	addi R3, R0, 12	initialize R3 = 12	4
	addi R7, R3, -9	initialize R7 = 3	8
	or R4, R7, R9	R4 = 3 | 5 = 7	c
	and R5, R3, R4	R5 = 12 & 7 = 4	10
	add R5, R4, R7	R5 = 4 + 7 = 11	14
	beq R5, R7, end	shouldn't be taken	18
	slt R4, R3, R4	R4 = 12 < 7 = 0	1c
	beq R4, R0, around	should be taken	20
	addi R5, R0, 0	shouldn't happen	24
around:	slt R4, R7, R2	R4 = 3 < 5 = 1	28
	add R7, R4, R5	R7 = 1 + 11 = 12	2c
	sub R7, R7, R2	R7 = 12 - 5 = 7	30
	sw R7, 68(R3)	[80] = 7	34
	lw R2, 80(R0)	R2 = [80] = 7	38
	j end	should be taken	3c
	addi R2, R0, 1	shouldn't happen	40
end:	addi R4, R2, 2	R4 = 7 + 2 = 9	44

Table 6: Intruction in test case 3

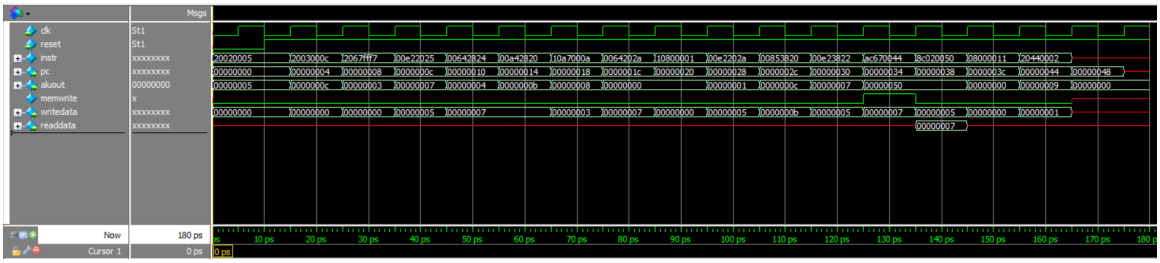

Figure 14: Waveform of the mips processor

Simulation and Analysis

- **0-35 ns:** The simulation begins by using I-type instructions to initialize data in the registers: `R2 = 5`, `R3 = 12`, `R6 = 13`.
- **35-65 ns:** Test R-type and observe the expected result in the table 6
- **65-75 ns:** Check `beq` function when the two values are not equal, so the PC = PC + 4 = 32'h1c
- **85-95 ns:** The `beq` function compares the two values in R4 and R0; they are equal, so PC = PC + 4 + BranchAddr = 32'h28.
- **125-145 ns:** The memory instructions. At **125 ns**, **memwrite** is active, `writedata` equals `7` (the value of `R7`), confirming that the `sw` instruction correctly stores the value at the appropriate memory address. At **135 ns**, `readdata` equals `7`, verifying that the value was correctly loaded back from memory, ensuring the `lw` instruction works properly.
- **145-155 ns:** Execute jump instruction. The PC will turn to the end address (32'h44).

7 Conclusion and Future work

7.1 Conclusion

1. The processor has been designed and tested to perform the basic functionalities of a MIPS processor, including:

- Performing basic arithmetic and logical operations
- Reading/writing registers
- Accessing memory through load/store instructions
- Executing control flow instructions such as branch and jump

2. The test cases have ensured that the values of ALUout, WriteData, Readdata, and PC are all correct as expected, confirming the validity of the design.

7.2 Future Development

- **Instruction set expansion**: Currently, only a basic instruction set is supported. Additional instructions such as multiply, divide, logical shifts, etc., should be added.
- **Performance improvement**: Research and implement multi-cycle or pipelined architectures to increase the processing speed.
- **External Memory Interface**: Develop an IP (Intellectual Property) module to interface the processor with an external memory, such as SRAM or DRAM. This would allow the processor to access a larger memory space beyond the limited on-chip memory.
- **Architecture extension**: Investigate enhancements like adding cache, supporting multithreading, etc., to improve overall performance.

With these improvements, the single-cycle MIPS processor will become more complete and better serve embedded applications and computer architecture research.