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. Creating a processor that supports various MIPS instructions including arithmetic, logic, memory access, and control instructions. MIPS processors are a family of RISC (Reduced Instruction Set Computer) architectures known for their simplicity and efficiency.

The pipeline architecture is divided into five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage is carefully designed to handle data hazards, control hazards, and ensure correct instruction flow through forwarding, stalling, and flushing mechanisms.

Finally, the individual modules were integrated into a main **risc_processor** 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 Pipeline Processor project aims to design and implement a simplified version of a MIPS processor, a widely recognized RISC (Reduced Instruction Set Computer) architecture. 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 Functional Implementation

- The pipelined microarchitecture divides the execution of an instruction into multiple stages, allowing different instructions to be processed simultaneously at different stages of the pipeline. This approach increases throughput and overall efficiency by overlapping the execution of instructions.
- **Advantages:** Multiple instructions are executed in parallel, significantly improving performance. The design is scalable and can handle higher instruction throughput compared to single-cycle architectures.
- **Disadvantages:** The complexity of the control unit increases, as it must handle hazards and ensure correct instruction flow. Additional hardware is required for forwarding, stalling, and flushing mechanisms to manage data and control hazards.
- The functional implementation of the pipelined MIPS processor can be described as follows:

  1. **Instruction Fetch (IF) Stage:** Getting instruction from memory by program counter (PC), and the PC is updated for the next instruction.
  2. **Instruction Decode (ID) Stage:** The fetched instruction is decoded, and the necessary control signals are generated. The operands needed for operation are also read from the register file. This stage may stall if control hazards are present.
  3. **Execution (EX) Stage:** The ALU performs the operation specified by the instruction. This stage may involve forwarding if data hazards are present.
  4. **Memory Access (MEM) Stage:** If the instruction involves memory access (e.g., 'lw' or 'sw'), the calculated address is used to read or write data from/to memory.
  5. **Write Back (WB) Stage:** The results from the EXE or MEM stage are written back to the appropriate register in the register file.
  6. **Hazard Unit:** To ensure correct instruction flow and data integrity, the pipeline employs mechanisms such as forwarding, stalling, and flushing. These techniques address data hazards (when Read after Write), control hazards (due to branch instructions).

  The modular design of the pipelined MIPS processor allows each stage to be independently developed, tested, and optimized. The pipeline structure maximizes instruction throughput, making it suitable for high-performance applications, while the hazard management techniques ensure the pipeline operates correctly even with dependencies and branch instructions.

3 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.

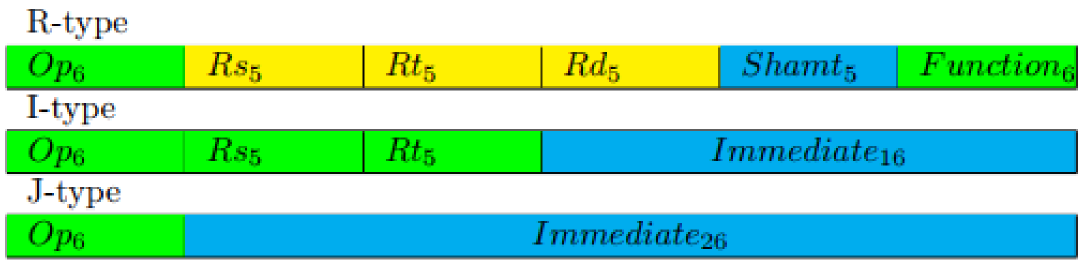

Figure 1: Mips Instruction Format

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	$R[\text{rd}] = R[\text{rs}] + R[\text{rt}]$	`0_hex / 20_hex`
and	R	$R[\text{rd}] = R[\text{rs}]\&R[\text{rt}]$	`0_hex / 24_hex`
or	R	$R[\text{rd}] = R[\text{rs}] \| R[\text{rt}]$	`0_hex / 25_hex`
sub	R	$R[\text{rd}] = R[\text{rs}] - R[\text{rt}]$	`0_hex / 22_hex`
slt	R	$R[\text{rd}] = (R[\text{rs}] < R[\text{rt}])?1 : 0$	`0_hex / 2A_hex`
nor	R	$R[\text{rd}] =\sim (R[\text{rs}] \| R[\text{rt}])$	`0_hex / 27_hex`
sll	R	$R[\text{rd}] = R[\text{rt}] << \text{shamt}$	`0_hex / 00_hex`
srl	R	$R[\text{rd}] = R[\text{rt}] >> \text{shamt}$	`0_hex / 02_hex`
addi	I	$R[\text{rt}] = R[\text{rs}] + \text{SignExtImm}$	`8_hex`
andi	I	$R[\text{rt}] = R[\text{rs}]\&\text{SignExtImm}$	`c_hex`
slti	I	$R[\text{rt}] = (R[\text{rs}] < \text{SignExtImm})?1 : 0$	`a_hex`
lw	I	$R[\text{rt}] = \text{Mem}[R[\text{rs}] + \text{SignExtImm}]$	`23_hex`
sw	I	$\text{Mem}[R[\text{rs}] + \text{SignExtImm}] = R[\text{rt}]$	`2b_hex`
beq	I	$\text{if}(R[\text{rs}] == R[\text{rt}])\text{PC} = \text{PC} + 4 + \text{BranchAddr}$	`4_hex`
bne	I	$\text{if}(R[\text{rs}] \neq R[\text{rt}])\text{PC} = \text{PC} + 4 + \text{BranchAddr}$	`5_hex`
j	J	$\text{PC} = \text{JumpAddr}$	`2_hex`

Table 1: MIPS Instructions

4 Architecture

Designing the pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. The output of the first stage become the input of the second stage, and so on. Besides, every stage operates on a different part of data, which allow multiple instructions to be in different stages of execution at the same time.

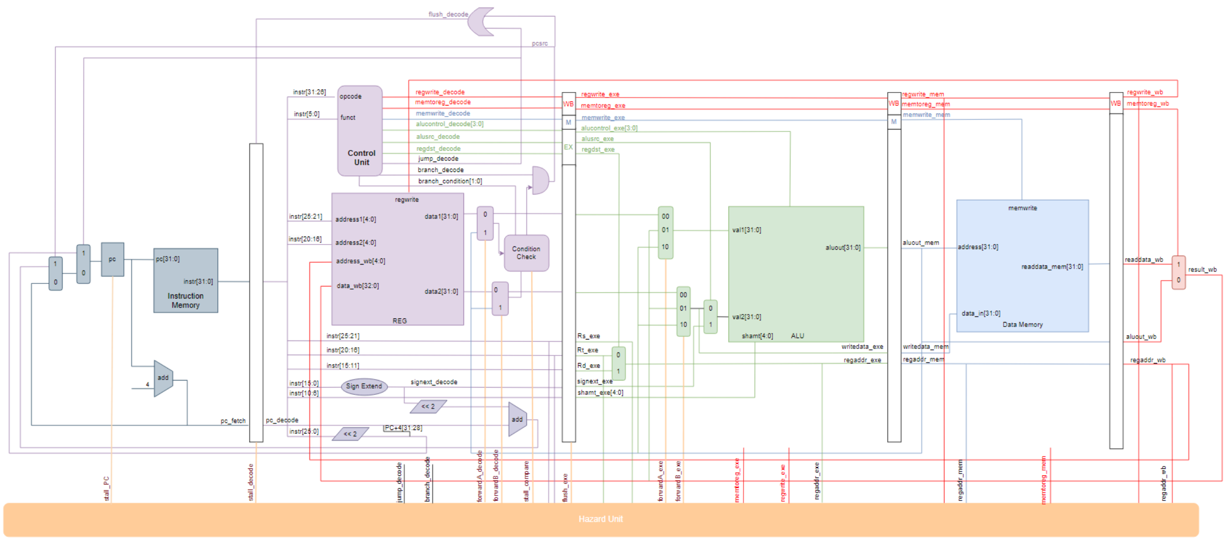

Figure 2: The Architecture of Mips Pipelined Processor

4.1 IF (Instruction Fetch) Stage

The **Instruction Fetch (IF)** stage is responsible for fetching the next instruction from memory. This stage primarily includes the Program Counter (PC), the Instruction Memory, Adder and Multiplexer (Mux) to select the correct next PC.

- **PC (Program Counter)**: A 32-bit register that holds the address of the current instruction being executed.
- **Instruction Memory**: A component of the computer system that stores the instructions of a program. The instruction to be fetched is based on the input of the Program Counter (PC).
- **Adder**: Increments the PC by 4 to point to the next instruction.
- **Mux**: Determining the correct address for the next instruction to be fetched

4.2 ID (Instruction Decode) Stage

The **Instruction Decode (ID)** stage is responsible for decoding the instruction fetched during the IF stage. This stage may involve determining branch or jump targets.

- **Control Unit**: Generates the control signals based on the opcode and function fields and that will guide the operation of subsequent stages in the pipeline.

- **Mux**: Select the correct register value when data hazard occur.
- **Condition Check**: Compare two register value and decide whether the branch should be taken or not.
- **Sign Extend**: Extends the immediate value of the instruction to 32 bits.
- **Register**: 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.

NAME	**NUMBER**	**USE**
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Table 2: Register Naming Conventions in MIPS

4.3 EXE (Execute) Stage

The **Execution (EXE)** stage performs the operation specified by the instruction, such as arithmetic or logic operations. This stage may involve determining the register destination for writing the result.

- **ALU (Arithmetic Logic Unit)**: The ALU takes two 32-bit operands and an operation control signal as inputs, and produces a 32-bit result.
- **2-to-1 Mux**: Select the correct register destination.
- **4-to-1 Mux**: Selects the correct values for the operands of the ALU.

4.4 MEM (Memory) Stage

The **Memory (MEM)** stage handles data memory operations, such as reading from or writing to memory. It's essential for load and store instructions.

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

4.5 WB (Write Back) Stage

The **Write Back (WB)** stage is where the results of operations or memory loads are written back to the register file. It selects the source of data that the to be written back.

4.6 Pipeline Register

In a pipelined MIPS processor, each stage of the pipeline is separated by **pipeline registers**. These registers store the intermediate data and control signals needed for the next stage, ensuring that each instruction can progress smoothly through the pipeline stages without interference from other instructions. In 5 stages pipeline, there are 4 pipeline registers namely *IF2ID, ID2EXE, EXE2MEM, and MEM2WB.* The registers are named for the two stages separated by that register. For example, the first pipeline register is *IF2ID* because it separates the instruction fetch and instruction decode stages.

- **IF2ID Register**: Holds the instruction fetched from memory during the **Instruction Fetch (IF)** stage and the incremented Program Counter (PC+4). This register ensures that the fetched instruction is passed correctly to the **Instruction Decode (ID)** stage in the next clock cycle.
- **ID2EXE Register**: Stores the decoded instruction information, control signals, and register values read during the **Instruction Decode (ID)** stage. It passes these to the **Execute (EX)** stage.
- **EXE2MEM Register**: Holds the results of the ALU operations performed in the **Execute (EX)** stage, along with control signals and data needed for memory access. It passes this information to the **Memory (MEM)** stage.
- **MEM2WB Register**: Stores the data retrieved from memory (or the ALU result if no memory access is required) and the control signals needed for the final **Write Back (WB)** stage.

4.7 Hazard Unit

The Hazard Unit is a component that is responsible for detecting and handling hazards that can occur during the execution of instructions. In a pipelined processor, multiple instructions are handled concurrently. When one instruction is *dependent* on the results of another that has not yet completed, a *hazard* occurs. There are two main types of hazards: **Data Hazard** and **Control Hazard**

4.7.1 Data Hazards

Data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed its passage through the pipeline. For example, consider the following sequence of instructions:

```
1. ADD R1, R2, R3   # R1 = R2 + R3
2. SUB R4, R1, R5   # R4 = R1 - R5
```

In this case, the *SUB* instruction depends on the result of the *ADD* instruction. If the *SUB* instruction reaches the Execute (EX) stage before the *ADD* instruction completes, it will use the old value of *R1*, leading to incorrect computation.

To solve data hazards, the following techniques are commonly used:

4.7.1.1 Forwarding (Data Forwarding/Bypassing)

- Forwarding is a technique where the result from Memory or WriteBack Stage is passed directly to a dependent instruction in the Execute stage before it is written back to the register file. This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage. The Figure 3 illustrates its operation.

- In cycle 4, \$4 is passed from the Memory stage of the *add* instruction to the Execute stage of the dependent *and* instruction. In cycle 5, \$4 is passed from the Writeback stage of the *add* instruction to the Execute stage of the dependent *or* instruction. The register file can be read and written in the same cycle so in this cycle, \$4 can be written to the register so the ID Stage of *and* instruction can get correct value of \$4.

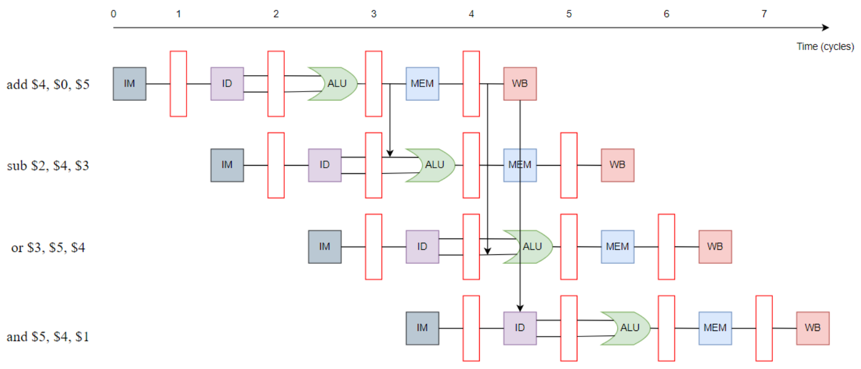

Figure 3: Pipeline diagram illustrating forwarding

4.7.1.2 Stalling (Pipeline Stall)

- Stalling is another method used to resolve data hazards by inserting "bubbles" or `NOP` (no-operation instructions) into the pipeline to delay the dependent instruction until the data is available.
- Stalling is used when an instruction tries to read a register following a *lw* instruction that writes the same register. The example of the instruction order is showed in Figure 4.
- In this case, the *sub* instruction will not get the value of \$2 from *lw* instruction because the data of the *lw* instruction is still being read from memory while the ALU is performing the operation for the *sub* instruction. Therefore, the pipeline must be stalled and wait when the \$2 read from the memory in cycle 4 and forward it to the *Execute* stage of *sub* instruction in cycle 5.

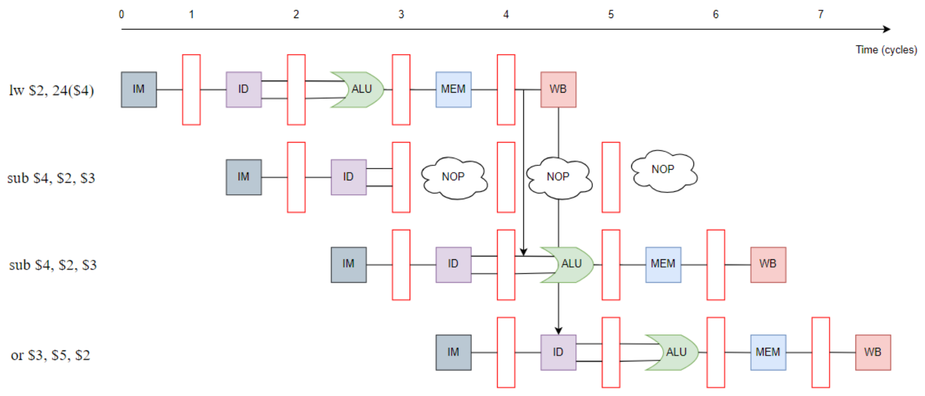

Figure 4: Pipeline diagram illustrating stall

4.7.2 Control Hazards

- **Control hazards** occur due to the pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched. One mechanism for dealing with the control hazard is to stall the pipeline until the branch decision is made. The figure 5 illustrates its principle.
- In cycle 2, if the branch is taken, the pipeline may have already fetched the *sub* instruction (since the *pc_branch* is calculated by the end of cycle 2, and the PC will be updated in cycle 3). If the branch is taken, the fetched *sub* instruction would need to be discarded, as the control flow would jump to the target of the branch (*around*). Therefore, if a branch or jump is taken, the pipeline processor will flush the next instruction to wait until *pc_branch* is fully computed.
- If the branch is not taken, the pipeline processor will continue execution without flushing, allowing the next instruction to execute as usual.

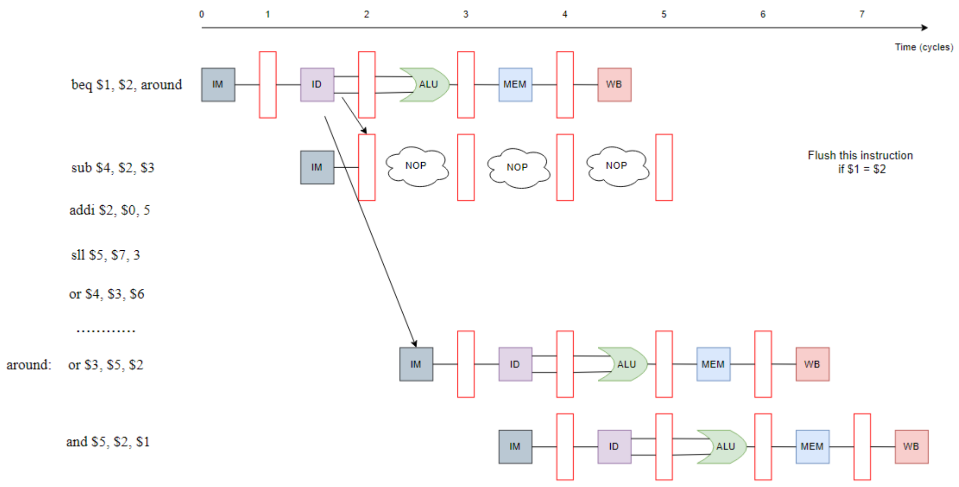

Figure 5: Pipeline diagram illustrating control hazard

5 Simulation and analysis

The testbench **risc_processor_tb** is designed to verify the execution of arithmetic, logical, shift, memory, branch and jump instructions. It ensures that the processor correctly performs arithmetic operations such as addition and subtraction, logical operations like AND and OR, and shift operations. It also verifies proper memory access for loading from and storing to memory, accurately evaluates branch conditions (e.g., `beq`, `bne`), and updates the program counter (PC) as needed for both branch and jump instructions.

In addition, the testbench validates the processor's handling of data hazards using techniques like forwarding and stalling, ensuring that instructions dependent on previous results (e.g., register-to-register data dependencies) are resolved without errors. It also checks the management of control hazards by testing branch prediction, pipeline flushing, and stalling mechanisms to guarantee correct instruction flow in the presence of branches.

#	Instruction	Description	Address	Assembly	Comment
begin:	`addi R2, R0, 5`	Initialize R2 = 5	0x00	0x20020005	
	`addi R3, R0, 12`	Initialize R3 = 12	0x04	0x2003000C	
	`addi R7, R0, 6`	Initialize R7 = 6	0x08	0x20070006	
	`addi R4, R0, 7`	Initialize R4 = 7	0x0c	0x20040007	
	`sub R6, R3, R2`	R6 = 12 - 5 = 7	0x10	0x00623022	
	`sll R5, R7, 3`	R5 = R7 << 3 = 48	0x14	0x000728C0	Data Hazard
	`sub R8, R5, R4`	R8 = 48 - 7 = 41	0x18	0x00444202	Data Hazard
	`and R5, R2, R6`	R5 = 5	0x1c	0x00462824	
	`sw R4, 4(R0)`	MEM[4] = 7	0x20	0xAC040004	
	`lw R2, 4(R0)`	R2 = MEM[4] = 7	0x24	0x8C020004	Data Hazard
	`sub R10, R8, R2`	R10 = 41 - 7 = 34	0x28	0x01025022	Data Hazard
	`beq R10, R5, end`	shouldn't be taken	0x2c	0x11450030	Control Hazard
	`or R4, R3, R6`	R4 = 15	0x30	0x00662025	
	`slti R3, R4, 10`	R3 = (R4 < 10) ? 1 : 0 = 0	0x34	0x2883000A	
	`beq R3, R0, around`	should be taken	0x38	0x10600002	Control Hazard
	`addi R5, R0, 0`	shouldn't be taken	0x3c	0x20050000	
	`slt R4, R7, R2`	shouldn't be taken	0x40	0x00E2202A	
around:	`sw R7, 68(R3)`	[68] = 6	0x44	0xAC670044	
	`lw R1, 68(R3)`	R1 = 6	0x48	0x8C610044	
	`bne R1, R7, end`	should be taken	0x4c	0x14270022	Control Hazard
	`nor R4, R6, R8`	R4 = -48	0x50	0x00C82027	
	`j end`	should be taken	0x54	0x08000018	Control Hazard
	`sub R7, R7, R2`	shouldn't be taken	0x58	0x00E23822	
end:	`addi R4, R2, 2`	R4 = 9	0x5c	0x00853820	

Table 3: Instruction Table with Hazard Analysis

5.1 Simulation and Analysis for the First 11 Instructions

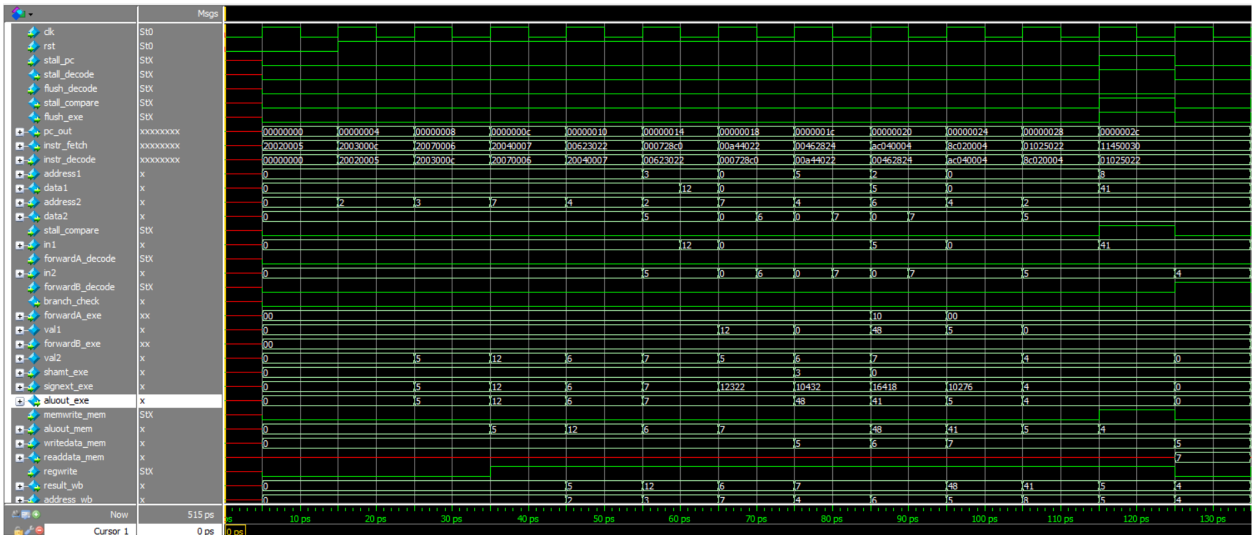

Figure 6: The first waveform of the pipelined processor

- **Cycle 1 (5ns):** *IF:* Fetch `addi R2, R0, 5`. *Others:* Reset active.
- **Cycle 2 (15ns):** *IF:* Fetch `addi R3, R0, 12`. *ID:* Decode `addi R2, R0, 5`.
- **Cycle 3 (25ns):** *IF:* Fetch `addi R7, R0, 6`. *ID:* Decode `addi R3, R0, 12`. *EX:* Execute `addi R2, R0, 5` (R2 = 5).
- **Cycle 4 (35ns):** *IF:* Fetch `addi R4, R0, 7`. *ID:* Decode `addi R7, R0, 6`. *EX:* Execute `addi R3, R0, 12` (R3 = 12).
- **Cycle 5 (45ns):** *IF:* Fetch `sub R6, R3, R2`. *ID:* Decode `addi R4, R0, 7`. *EX:* Execute `addi R7, R0, 6` (R7 = 6). *WB:* Write back `addi R2, R0, 5` (R2 = 5).
- **Cycle 6 - Cycle 8 (55ns - 75ns):** Arithmetic instructions execute without hazards.
- **Cycle 9 (85ns):** *IF:* Fetch `sw R4, 4(R0)`. *ID:* Decode `and R5, R2, R6`. *EX:* Execute `sub R8, R5, R4`. This stage encounters a data hazard because the execution depends on the result of the previous instruction. The data is forwarded from the memory stage to the ALU (forwardingA_exe = 10).
- **Cycle 10 - Cycle 11 (95ns - 105ns):** Instructions execute without hazards.
- **Cycle 12 (115ns):** *IF:* Fetch `beq R10, R5, end`. *ID:* Decode `sub R10, R8, R2`. This stage encounters a data hazard because the previous instruction is a `lw`, and the correct value will be available only during the memory stage. As a result, this instruction will be stalled (`stall_decode = 1, stall_pc = 1, flush_exe = 1`).. *EX:* Execute `lw R2, 4(R0)`.
- **Cycle 13 (125ns):** *IF:* Fetch `NOP`. *ID:* Decode `sub R10, R8, R2`. *MEM:* Load `lw R2, 4(R0)` (R2 = 7).

5.2 Simulation and Analysis of Instructions 12 to 18

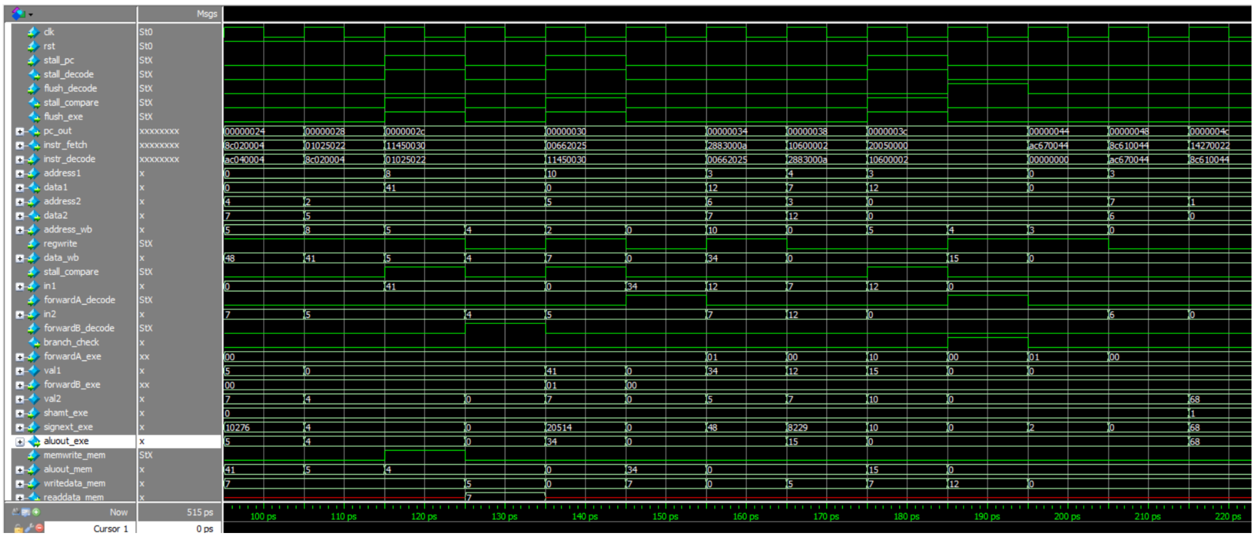

Figure 7: The seconde waveform of the pipelined processor

- **Cycle 14 (135ns):** *IF:* Fetch `or R4, R3, R6`. *ID:* Decode `beq R10, R5, end`, encounter control and data hazards (depends on the previous instruction), causing a stall (`stall_decode`, `stall_pc`, `flush_exe`, `stall_compare` = 1). *EX:* Execute `sub R10, R8, R2` (R10 = 41). *WB:* Write back the result of `lw R2, 4(R0)` (R2 = 7).

- **Cycle 15 (145ns):** *IF:* Fetch `or R4, R3, R6`. *ID:* Decode `beq R10, R5, end`, get correct values of R10 and R5; branch not taken (since $R10 \neq R5$, so `branch_check` = 0). *EX:* Execute `NOP`. *MEM:* Access memory (none for `sub R10, R8, R2`); `aluout_mem` forwarded to ID stage. *WB:* Write back result of `lw R2, 4(R0)` (R2 = 7).

- **Cycle 16 - Cycle 17 (155ns - 175ns):** No hazards; instructions executed normally.

- **Cycle 18 (175ns):** *IF:* Fetch `addi R5, R0, 0`. *ID:* Decode `beq R3, R0, around`, encounter control and data hazards (depends on the previous instruction), causing a stall (`stall_decode`, `stall_pc`, `flush_exe`, `stall_compare` = 1). *EX:* Execute `slti R3, R4, 10` (R3 = 0). *MEM:* Access memory (none for `or R4, R3, R6`); *WB:* Write back: none.

- **Cycle 19 (185ns):** *IF:* Fetch `addi R5, R0, 0`. *ID:* Decode `beq R3, R0, around`, get correct values of R3 and R0; branch taken (since $R3 = R0$, so `flush_decode` = 1). Also complete calculating `pc_branch`. *EX:* Execute `NOP`. *MEM:* Access memory (none for `slti R3, R4, 10`); *WB:* Write back for `or R4, R3, R6` (R4 = 15).

- **Cycle 20 (195ns):** *IF:* Fetch `sw R7, 68(R3)`. *ID:* Decode `NOP`. *EX:* Execute `NOP`. *MEM:* Access memory (none for `NOP`); *WB:* Write back for `or R4, R3, R6` (R3 = 0).

5.2.1 Simulation and Analysis of Instructions 18 to End

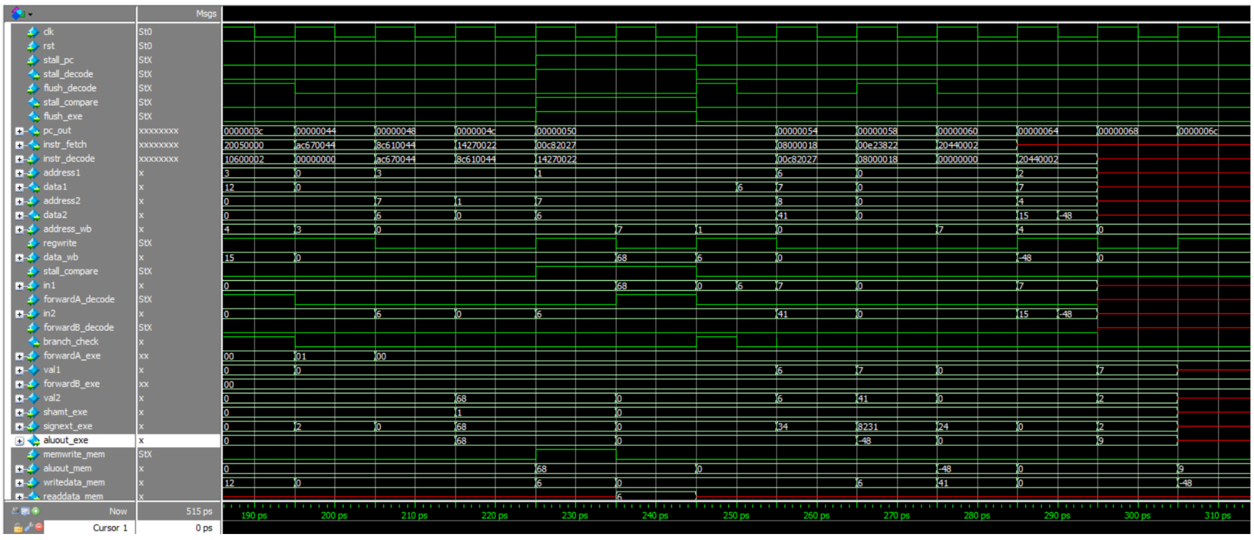

Figure 8: The last Waveform of the Pipelined Processor

- **Cycle 21 (205ns):** *IF:* Fetch `lw R1, 68(R3)`. *ID:* Decode `sw R7, 68(R3)`. *EX:* Execute `NOP`. *MEM:* Access memory (none for `NOP`). *WB:* Write back for `NOP`.
- **Cycle 22 (215ns):** *IF:* Fetch `bne R1, R7, end`. *ID:* Decode `lw R1, 68(R3)`. *EX:* Execute `sw R7, 68(R3)`. *MEM:* Access memory (none for `NOP`). *WB:* Write back for `NOP`.
- **Cycle 23 (225ns):** *IF:* Fetch `nor R4, R6, R8`. *ID:* Decode `bne R1, R7, end`, encounter control and data hazards (depends on the previous instruction), causing a stall (`stall_decode`, `stall_pc`, `flush_exe`, `stall_compare` = 1). *EX:* Execute `lw R1, 68(R3)`. *MEM:* Access memory `sw R7, 68(R3)` (MEM[68] = R7 = 6). *WB:* Write back for `NOP`.
- **Cycle 24 (235ns):** *IF:* Fetch `nor R4, R6, R8`. *ID:* Decode `bne R1, R7, end`, encounter control and data hazards (depends on the previous instruction), causing a stall (`stall_decode`, `stall_pc`, `flush_exe`, `stall_compare` = 1). *EX:* Execute `NOP`. *MEM:* Access memory `lw R1, 68(R3)` (MEM[68] = R7 = 6). *WB:* Write back (none for `sw R7, 68(R3)`).
- **Cycle 25 (245ns):** *IF:* Fetch `nor R4, R6, R8`. *ID:* Decode `bne R1, R7, end`. Get the correct data from R1 and compare it, but the branch shouldn't be taken (since $R1 == R7$). *EX:* Execute `NOP`. *MEM:* Access memory (none for `NOP`). *WB:* Write back (none for `lw R1, 68(R3)`).
- **Cycle 26 (255ns):** *IF:* Fetch `j end`. *ID:* Decode `nor R4, R6, R8`. *EX:* Execute `NOP`. *MEM:* Access memory (none for `NOP`). *WB:* Write back (none for `NOP`).
- **Cycle 27 (265ns):** *IF:* Fetch `sub R7, R7, R2`. *ID:* Decode `j end`. Stall the processor to complete calculating `pc_jump`. *EX:* Execute `nor R4, R6, R8` (R4 = -48). *MEM:* Access memory (none for `NOP`). *WB:* Write back (none for `NOP`).
- **Cycle 28 (275ns):** *IF:* Fetch `addi R4, R2, 2`. *ID:* Decode `NOP`. *EX:* Execute `NOP`. *MEM:* Access memory (none for `nor R4, R6, R8`). *WB:* Write back for `NOP`.

6 Conclusion and Future Work

6.1 Conclusion

- The pipelined processor has been designed and tested to enhance instruction throughput by overlapping the execution of multiple instructions. This includes:
  - Implementing stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB).
  - Enabling the processor to perform multiple operations simultaneously, improving overall performance.
  - Handling hazards (data, control, and structural) effectively to maintain pipeline efficiency.
  - Verifying that the outputs from various stages, such as ALU results, memory accesses, and control signals, operate as expected.
- Test cases have validated that the pipelined architecture maintains correct functionality, ensuring that the processor can execute a sequence of instructions without errors under various conditions.

6.2 Future Development

- **Hazard Mitigation Techniques**: Explore advanced techniques such as out-of-order execution and branch prediction to further minimize stalls and improve performance.
- **Pipeline Depth Optimization**: Investigate increasing or decreasing the number of pipeline stages to balance complexity and speed, optimizing the design according to specific application needs.
- **Support for Superscalar Execution**: Implement mechanisms to allow multiple instructions to be issued per cycle, enhancing throughput for parallel processing tasks.
- **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.

With these advancements, the pipelined processor will not only improve its capabilities but also provide a strong foundation for further research in computer architecture and embedded systems.