JoSDC'24

المسابقة الوطنية لتصميم الشرائح الإلكترونية

Jordan National Semiconductors Design Competition

Development Phase Report - JoSDC'24

Team Name		
Silicore		
Team Members		
#	Name	Email
1	Alanoud Alsalem	ala20210371@std.psut.edu.jo
2	Hasan Al-Hasnawi	hassanhasnawi99@gmail.com
3	Lojain Hamdan	loj20210576@std.psut.edu.jo
4	Nedal Abu-Haltam	nedalhaltam1@gmail.com
5	Qabas Ahmad	qab20210786@std.psut.edu.jo

Jordan National Semiconductor Design Competition (JoSDC'2024)

Silicore's Pipelined Implementation of a MIPS Processor

By

Nedal Abu-Haltam	Alanoud Alsalem	Hasan Al-Hasnawi	Lojain Hamdan	Qabas Ahmad
Computer Engineering	Computer Engineering	Electronics Engineering	Computer Engineering	Computer Engineering
nedalhaltam1@gmail.com	ala20210371@std.psut.edu.jo	hassanhasnawi99@gmail.com	loj20210576@std.psut.edu.jo	qab20210786@std.psut.edu.jo

Amman, Jordan

12 / 2024

Acknowledgments

We have the deepest of gratitude for our university Princess Sumaya University for Technology PSUT for granting us this invaluable opportunity in which we were trained for two weeks about the intricacies of computer architecture which helped us in the completion of this phase.

We extend our heartfelt gratitude to the Crown Prince Foundation (CPF), the Ministry of Digital Economy and Entrepreneurship (MODEE), the Fifth Organization, and the Information and Communications Technology Association (INT@J) for granting us the opportunity to participate in this competition. We deeply appreciate your support and the resources provided, which made this training a truly enriching experience.

We want to thank our professors, whose invaluable guidance during our university courses provided us with the knowledge and skills needed to excel in this competition and in our field. We also wish to thank our mentors, who tirelessly supported us throughout our progress, offering priceless advice and coaching us across various domains.

Ultimately, the successful completion of this project was made possible through the dedication and expertise of a remarkable team. Grand appreciation is extended to everyone who contributed their skills and insights. The support and diligence throughout were nothing short of notable.

Abstract

This report aims to showcase the design of a 32-bit MIPS processor. Two fundamental architectures are implemented, evaluated and compared using a variety of tools and integrating both hardware and software components to create a comprehensive system. Starting with a single-cycle processor in which the design focuses on correct functionality of the required instructions. Transitioning to a pipelined implementation, a 6-stage pipeline is developed to enhance performance and throughput. Additionally, an assembler and a cycle accurate simulator were built representing the software component which complemented the hardware design and demonstrating the system-level interactions with the hardware. Furthermore, various evaluation metrics were used to compare the performance of single cycle and pipelined processors. The findings demonstrate the advantages of pipelining in processor design underscoring the importance of iterative refinement in tackling complex challenges in computer architecture.

Table of Contents

List of Figures

List of Tables

1 Executive Summary

This report details the implementation, development and evaluation of a 32-bit pipelined MIPS processor undertaken by Team Silicore as part of the JoSDC'24 competition. The project represents a comprehensive study of processor design, from foundational single-cycle to advanced pipelined architectures. In addition to the hardware, software tools were developed to support it.

The primary goal was to design a MIPS processor that improves instruction throughput and efficiency while addressing the limitations single cycles have. With that information, our design processor is as follows:

- Implementing a single cycle processor as the foundation for understanding core functionalities.
- Evolving the design into a pipelined processor using a 6-stage architecture – Fetch, Decode, execute(separated into forwarding and executing), Memory and Write Back.
- Developing complementary software tools like the assembler and cycle accurate simulator to enhance the testing, debugging and performance evaluation.

Hardware Design

After fully learning about the MIPS architecture, the main Datapath components were implemented which includes program counter, instruction memory, control unit, Arithmetic Logic Unit and many more.

Those components were connected to create the single cycle implementation, with thorough testing and debugging to ensure its correct functionality.

Through that process, the shortcomings and inefficiencies of single cycle was predominant which led to the development and implementation of the pipeline.

Several modules were added to the pipeline to ensure both efficiency and correctness. Hazard Resolution was an important one as it is caused by reading and writing on the same value and it directly affects the correctness of the design as efficiency falls short if the output was wrong. Branch Prediction unit was added to minimize the number of wasted cycles that results from flushing the instructions from the pipeline to fetch the correct ones. Inter-stage registers were used to store the information needed for the successive stage.

Software Design

A real-time assembler with a user-friendly graphical interface was built to offer immediate feedback on assembly code conversion as well as validity and errors. This addition results in reduced debugging complexity and enables seamless conversion of assembly language into machine code. Another developed software is the cycle-accurate simulator, allowing real-time analysis of performance validation of benchmarks and compatibility with single-cycle and pipelined architectures. It also generates universal memory initialization files for simulation and prototyping.

Testing and Evaluation

Rigorous validation tests and simulations confirmed the pipelined processor's superior performance compared to the single-cycle design. Benchmarking showed significant improvements, such as reducing execution time for BM3 by 28.3% and achieving a clock frequency increase from old initial operating frequency of 64.91 MHz to 82.99 MHz through optimizations like refined mux design and a sixth pipeline stage. The assembler ensured smooth hardware simulations by converting programs into machine code and memory files, with a HALT instruction added for accurate cycle counts.

Critical paths, initially causing slack violations up to -11.088 ns, were optimized through structural improvements, branch unit refinements, and clock adjustments and reaching so far to a positive setup slack of 0.081. Functionality testing using JOSDC committee benchmarks confirmed the reliable execution of all instructions, while the Cycle-Accurate Simulator and Verilog testbenches validated outputs like register values and timing waveforms.

The pipelined processor consistently outperformed the single-cycle design, especially for complex workloads with higher instruction counts. Throughput increased across most benchmarks.

Challenges and Solutions

The transition from single cycle processor to a pipelined architecture posed several challenges such as managing hazards (data, control and structural hazards), implementing branch predictions to minimize stalls and creating modular software tools adapting to iterative changes.

To ensure all cases were covered, additional test cases were added to ensure that all possible outcomes are dealt with correctly and efficiently.

Contributions and Learning

This project combined theoretical knowledge and hands-on design, emphasizing both hardware and software co-development. The team navigated the intricacies of computer architecture while balancing academic workloads, highlighting their resilience and collaborative spirit. Key takeaways include:

1. Practical experience in processor design and verification.
2. Enhanced understanding of pipelining concepts and their impact on performance.
3. Skills in software tool development for system validation.

Conclusion

The project successfully met all requirements and demonstrated the benefits of pipelining in modern processor design. The combined efforts of hardware and software teams resulted in a functional, efficient, and validated system that reflects advanced principles of computer architecture. This achievement underscores the value of iterative design and highlights the team’s ability to tackle complex engineering challenges effectively.

This report not only provides a technical overview of the project but also showcases the dedication and expertise of Team Silicore in advancing processor design methodologies. The work lays a strong foundation for future phases of the competition and the professional growth of the team members.

2 Introduction

Scientists and Engineers across decades have worked meticulously to craft powerful central processing units (CPU) that are fast and simultaneously consume a reasonable amount of power. As a consequence, the implementation of the CPU varied significantly and scientists used several approaches to achieve that; from evolving at a transistor-level to implementing deeper and deeper pipelines to now introducing several cores on a single chip. This report will show how an implementation of a MIPS processor evolved from its logical blocks to single cycle implementation to pipelined implementation.

There are two distinct philosophies in CPU architecture design, CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer). CISC architecture is characterized by a large set of instructions, each instruction is able to perform multiple operations. It prioritizes reducing code size and easier programming in behalf of slower execution and design complexity. RISC on the other hand focuses on a smaller set of simple instructions, it prioritizes speed of execution but increases code size.

The MIPS (Microprocessor without Interlocked Pipeline Stages) processor is a prominent example of a RISC architecture that is designed to optimize performance through efficiency and simplicity. The instruction set consists of instructions categorized into three types: R-Type, I-Type and J-Type. MIPS processors have various applications and are widely used in embedded systems, consumer electronics, networking equipment and gaming consoles.

This report will show how an implementation of a MIPS processor evolved from its logical blocks to single cycle implementation to pipelined implementation.

2.1 **Objectives**

Implementing MIPS architecture processor that supports various ALU, Memory and Branch operations.

The single-cycle MIPS processor introduced the basic concept of instruction processing and laid the foundation by executing each instruction in a single clock cycle. However, this simplicity limited the clock speed, making it difficult to meet the needs of modern computing workloads.

The pipelined MIPS processor solved this problem by breaking instruction processing into sequential stages, increasing throughput and efficiency by allowing multiple instructions to be processed simultaneously.

In order to obtain proper functionality of the processor and still have proper efficiency, the implemented CPU deals with various hazards like data hazards, control hazards and structural hazards.

2.2 Design Achieved

We consider a simple version of MIPS that uses Harvard architecture. Harvard architecture uses separate memory for instruction and data. MIPS (Microprocessor without Interlocked Pipeline Stages) is a family of RISC instruction set architectures developed as part of VLSI research program at Stanford University in the early 80s.

Important features of MIPS Architecture include:

- It utilizes a simplified instruction set allowing for faster execution and easier pipeline implementation.
- Operations are done via registers with specific load and store instructions for memory access.
- Each instruction is 32-bits in length which simplifies fetching and decoding.

2.2.1 Phase I:

Debugging pre-implemented single cycle 32-bit processor that handles a subset of instructions: add, addi, sub, and, or, slt, lw, sw, beq, jump, the goal is to ensure proper functionality of the processor. Concurrently, the code for the single cycle processor was being developed and tested rigorously.

2.2.2 Phase II:

Improving upon the 32-bit single cycle processor such that it supports all the instructions in table (1) and it handles the implementation of the pipeline and all the hazards resulting from it via forwarding and stalling.

3 MIPS Reference Architecture

Scientists and Engineers across decades have worked meticulously to craft powerful central processing units (CPU) that are fast and simultaneously consume a reasonable amount of power. As a consequence, the implementation of the CPU varied significantly and scientists used several approaches to achieve that; from evolving at a transistor-level to implementing deeper and deeper pipelines to now introducing several cores on a single chip. This report will show how an implementation of a MIPS processor evolved from its logical blocks to single cycle implementation to pipelined implementation all the way to the superscalar implementation.

3.1 Instruction Set

The instruction set is represented in 32 bits categorized as a RISC (Reduced Instruction Set Computer) architecture; each instruction has different fields that indicate the functionality needed from the CPU.

- R-Format: This instruction has three registers (two of them are operands and one is destination, each taking 5 bits), shift amount and function code which identifies the specific R-Format Instruction.

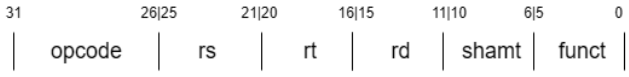


Figure 1: R-type Instruction Format in MIPS Architecture.

- I-Format: This instruction has two registers (one operand and one destination) and a constant value immediately present in the instruction.

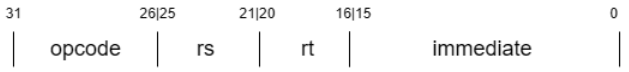


Figure 2: I-type Instruction Format in MIPS Architecture.

- J-Format: This instruction has a (part of) address in the instruction.

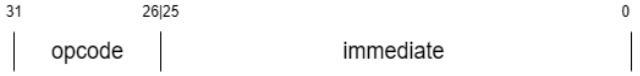


Figure 3: J-type Instruction Format in MIPS Architecture.

- FR-Format: Similar to R-Format, the MIPS processor supports floating point operations in which the operands are not integers but floating point. It will not be a part of this processor's implementation.

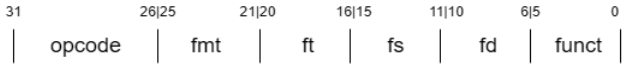


Figure 4: FR-type Instruction Format in MIPS Architecture.

- FI-Format: Similar to I-Format, it is the floating-point operation for immediate value operations. It will also not be a part of this processor's implementation.

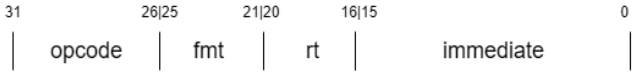


Figure 5: FI-type Instruction Format in MIPS Architecture.

The exact instructions implemented in this processor can be seen in the following table:

Table 1: the implemented instructions and pseudo instructions.

Branches and Jumps	Arithmetic and Logic	Load and Store
BEQ	ADD, ADDI, ADDU	LW
BNE	SUB, SUBU	SW
BLTZ	AND, ANDI	
BGEZ	OR, ORI	
JUMP (J)	XOR, XORI	
JR	NOR	
JAL	Shift Right Logical (SRL)	
	Shift Left Logical (SLL)	
	Set Less Than (SLT)	
	Set Less Than Immediate (SLTI)	
	Set Greater Than (SGT)	

3.2 Memory Organization and Management

The MIPS architecture employs a structured approach crucial for efficient data handling and program execution. It utilizes a 32-bit flat memory model, allowing addressing up to 4 gigabytes of memory. Flat memory model refers to a memory addressing paradigm in which memory appears to the program as a single contiguous address space and the CPU can directly and linearly address all of the available memory locations without bank switching.

MIPS have a limited number of general-purpose registers, each capable of holding 32-bit value, they serve as fast-access storage for data being processed by the CPU, they are critical for the Datapath because all operations must occur on data stored in registers so values stored in the memory need to be loaded into the registers to be used for operations.

- Loading Data: Data in memory is transferred into registers using load instructions.
- Processing Data: Arithmetic and logical operations are executed on the data in the registers.
- Storing Data: Results can be written back to memory using store instructions.

MIPS balances performance with flexibility through a flat addressing model, strict alignment requirements and reliance on the limited number of registers in its operation.

4 Design

4.1 Hardware Components

4.1.1 Program Counter (PC)

It is a register that holds the address of the next instruction to be executed, it automatically increments after every instruction is fetched unless a branch or jump instruction changes its flow.

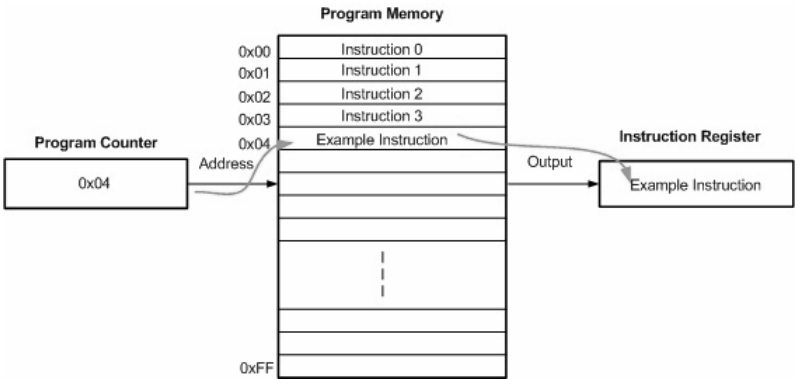


Figure 6: Program Counter Functionality Diagram.

It takes as input a 32-bit address which is the address of the next instruction as well as the clock signal, reset signal and enable signal and it outputs the address that is stored in the register at the positive edge of the clock.

4.1.2 Instruction Memory

A memory that contains the instructions of the program that the CPU executes, the program counter decides which instruction is to be fetched and sends the address to the instruction to retrieve that instruction.

It uses Harvard architecture instruction set, in which the instruction memory is independent and separate from the data memory.

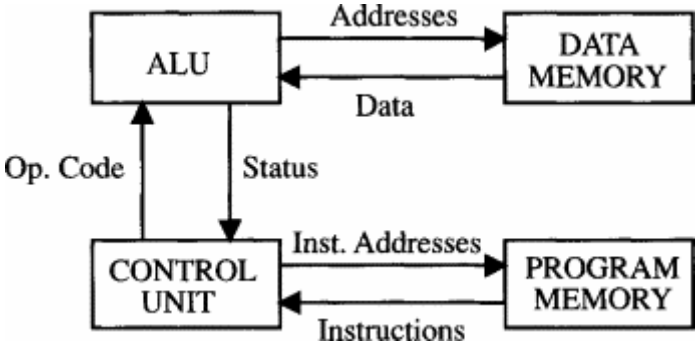


Figure 7: Harvard Architecture CPUs.

The module takes the 32-bit address as an input which uses the first 10 bits to access the location of the memory containing the instruction that needs to be executed and it returns the instruction it reads for the Datapath to be decoded.

4.1.3 Control Unit

The Control Unit interprets the fetched instruction and generates signals to control and orchestrate the operation of remaining Datapath components. The control unit has a pivotal role of instruction decoding, control signals generation and combinational logic.

The control signals generated - based on the opcode of the instruction:

- Register Write: Indicating whether data should be written back to a register.

```
output regwrite;
```

- Memory Read/Write: Controlling access to data memory for load and store operations.

```
output memread;
output memwrite;
```

- ALU Operand: Specifying the source of the second operand in arithmetic and logical operations.

```
output is_oper2_immed;
```

- Branching: Managing branch instructions to alter the program counter (PC).

```
output is_beq, is_bne, is_jr, is_jal, is_j;
```

The module that determines the operation of the ALU is implemented separately from the control unit to ensure a modular design. It takes as input the opcode and specifies the ALU's operation as follows:

Table 2: The operation code for all operations performed by the ALU.

ALU Opcode	Operation
0	Addition, Memory Load and Store, Jump Operations
1	Subtraction
2	Logical AND operation
3	Logical OR operation
4	Logical XOR operation
5	Universal-gate NOR operation
6	Shift Left Logical
7	Shift Right Logical
8	Set Less Than
9	Set Greater Than

4.1.4 Register File

A small, fast storage array containing general-purpose registers that store intermediate data and operands for the ALU. The register file typically has two read ports and one write port, allowing two source operands to be read simultaneously for the ALU.

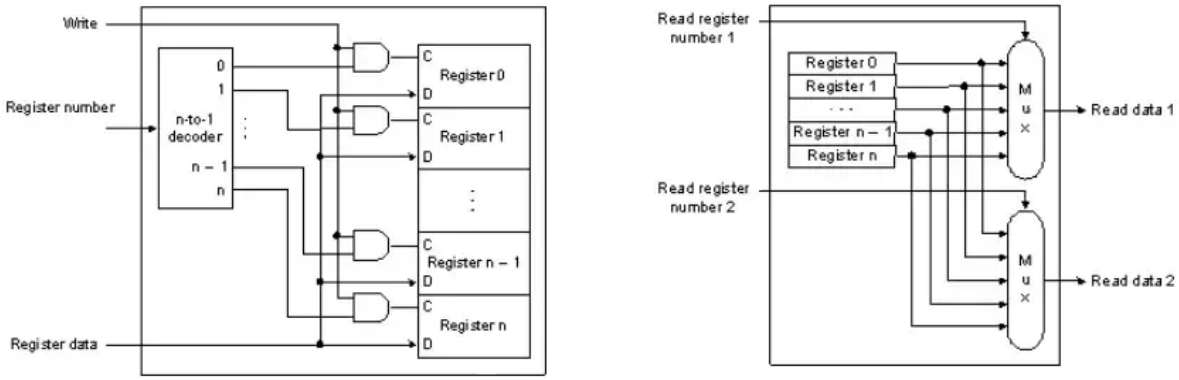


Figure 8: Register File Diagram and Abstract Unit.

It takes as input the first and second read register as well as the destination register. It also takes the value to be written in the destination register if the instruction requires it like load and arithmetic operations.

It outputs the contents of the first and second read register.

4.1.5 Arithmetic Logic Unit (ALU)

The ALU performs arithmetic and logical operations, such as addition, subtraction, logical operations like bitwise AND, OR, and shifts. It processes the operands from the register file or immediate values from the instruction. The ALU's output is directed to either a register or memory, depending on the instruction.

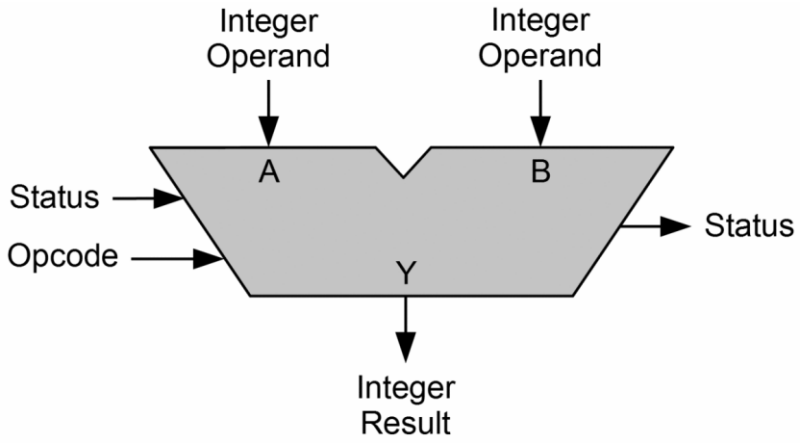


Figure 9: ALU diagram with inputs and outputs.

Table 3: the instructions done by the ALU.

Operation	Description
`ADD x3, x1, x2`	Adds register operands x1 and x2 and stores the result in register x3

`SUB x3, x1, x2`	Subtract register x2 from register x1 and stores the result in register x3
`AND x3, x1, x2`	Performs logical AND for x1 and x2 and stores the result in x3
`OR x3, x1, x2`	Performs logical OR for x1 and x2 and stores the result in x3
`XOR x3, x1, x2`	Performs logical XOR for x1 and x2 and stores the result in x3
`NOR x3, x1, x2`	Performs logical NOR for x1 and x2 and stores the result in x3
`SLL x3, x1,imm`	Shifts left x1 value imm(immediate) times and stores the result in x3
`SRL x3, x1,imm`	Shifts right x1 value imm(immediate) times and stores the result in x3
`SLT x3, x1, x2`	Sets x3 to 1 if x1 is less than x2
`SGT x3, x1, x2`	Sets x3 to 1 if x1 is greater than x2

4.1.6 Multiplexers (MUXs)

Multiplexers are used in the Datapath to select inputs based on the signals from the control unit derived from the instruction. They allow for resource optimization such that multiple data inputs are routed to a single output.

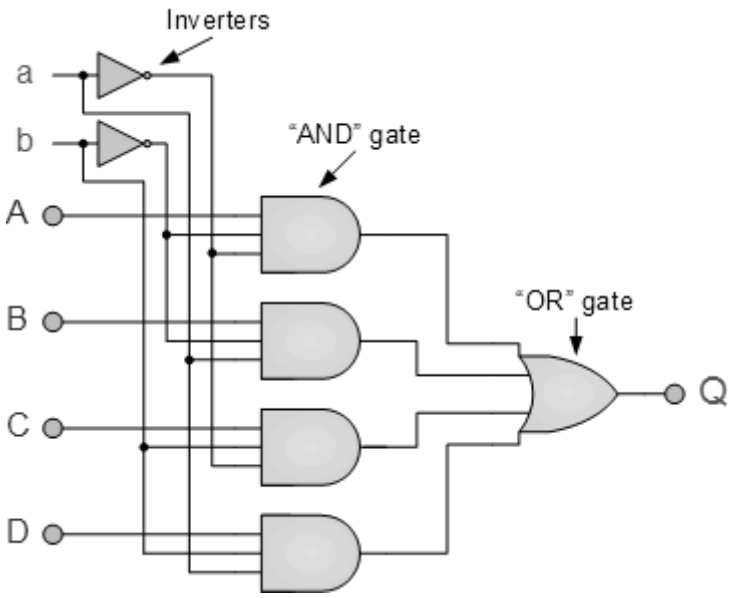


Figure 10: 4x1 MUX components and circuitry.

Common MUX uses include:

- ALU Operand: is it a register value for R-Format instructions or immediate value for I-Format instructions?

- Next Instruction: is it an Incremented PC such that the flow of instructions is retained or is it a Branch target?
- Write-Back on Register: is it ALU result or Memory data to be written back into the register file?

4.1.7 Data Memory

Data Memory stores and retrieves data during load and store operations. This component is separate from the instruction memory in Harvard architectures.

This module takes the address and data to be written for write operations as 32-bit inputs and outputs the 32-bit memory data for read operation. The data memory is word addressable with size 1Kx32 such that its width can store the size of the data. The module takes the address to be 32 bits but only the first 10 bits are extracted and used to address the actual memory.

4.1.8 Immediate Generation Unit

A sign extender is used to increase the bit-width of an immediate value, typically in instructions like load or branch. Implemented in the immediate generation unit, it is pivotal for the shamt in R-Format instructions and in the immediate in I-Format and J-Format instructions.

4.1.9 Branch/Jump Target Calculators

- **Branch Target**: Adds the PC and an offset to compute the branch target address.
- **Jump Target**: Address is decoded from the instruction.

4.2 Single Cycle CPU Implementation

A single-cycle processor executes a complete instruction in one cycle, takes the instruction when the clock rises, decodes, executes, and accesses it during the high-level period, and writes it back when the clock falls. It uses one clock-period per instruction and the clock-period is set by the total delay of the slowest instruction.

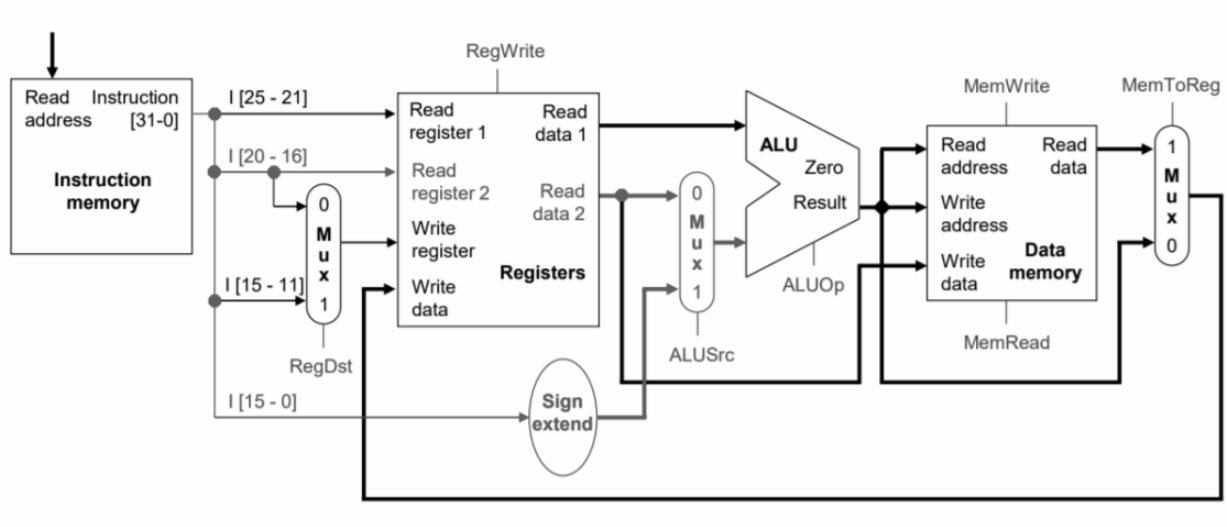


Figure 11: Simple Datapath diagram of a single cycle CPU.

In this section, we will walk through the Datapath to gain a deeper understanding and appreciation for the foundational implementation of today's processors.

In a single cycle processor, a single instruction goes through the entire Datapath and only uses the hardware it needs to do its operation, it provides a simplified understanding of how a CPU functions suitable for educational purposes.

4.2.1 Cycle Time Execution and Halt Instruction

Every instruction takes exactly one clock cycle, resulting in a Cycle Per Instruction (CPI) of 1. It simplifies the control logic since all operations occur within one cycle. However, the cycle must be long enough to accommodate the slowest instruction in the architecture, even the simpler instructions will take as long as the longest instruction.

Calculating the number of cycles is crucial for determining CPU efficiency, emphasizing the need to halt operations when a program ends. If not, the number of cycles could increase unnecessarily without any instructions being executed. To manage this, a halt instruction (HLT) is employed to terminate the continuous execution cycle. This mechanism ensures that no additional instructions are fetched beyond the halt instruction, leading to accurate cycle count calculations. The opcode of the HLT instruction is:

$$halt\_inst \; = \; 12'hFC0;$$

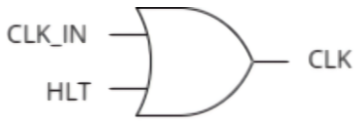


Figure 12: Halt instruction implementation circuitry.

The aim of the halt instruction is to disconnect the clock to stop counting the cycles so the processor decodes the instruction to generate a halt signal. This signal indicates that the CPU should stop executing further instructions. The halt signal is then combined with the clock signal using an OR gate. When the halt signal is active, the output of the OR gate prevents the counting of additional clock cycles, effectively pausing cycle counting. It is essential for both performance and power management as CPU resources are not wasted.

4.2.2 Control Logic in Single Cycle

The control unit is needed to produce all necessary control signals for every instruction in a single cycle which requires a well-designed combinational logic circuit that interprets the opcode and generates appropriate signals for each component.

4.2.3 Pseudo Instructions

Pseudo instructions are a significant aspect of assembly language programming as it provides a level of abstractions to simplify coding while not changing the instruction set architecture.

Pseudo instructions are not recognized by the hardware, they are representations of an instruction that the assembler translates into one or more machine code instructions – allowing for more human-readable code.

An example of a pseudo instruction to ask is branch greater than or equal:

```
BGEZ  <r1>, <label>
```

which the assembler translates to:

```
SLT  <r3>, <r1>, <r0>

BEQ <r3>, <r0>, <label>
```

4.2.4 Efficiency vs. Simplicity – Disadvantages

While simple, this architecture can be inefficient for complex instructions that require more time than simpler ones, as all instructions must wait for the longest operation to complete.

Since each functional unit is used once per instruction cycle, there may be a need for multiple instances of certain units (like ALUs) to handle different operations simultaneously, which can increase complexity and cost.

To summarize the downsides of single cycle implementation, it does not utilize the resources enough to get the maximum benefit of the architecture. In addition, the frequency in which the instructions are processed is low. Because of that, specialists started looking into ways to efficiently use the hardware components and process at a higher frequency, ensuring that no component is idle while another one is busy, which brought into being the concept of pipelining – to divide an instruction into stages and execute the stages in parallel as detailed in the following section.

4.3 Pipelined CPU Implementation

Pipelining is an essential technique used in modern processors to enhance performance by enabling multiple instructions to be processed at once. Instead of executing one instruction at a time in a linear fashion, pipelining divides the process into separate stages. Each stage completes part of an instruction, allowing different instructions to overlap and be processed concurrently at different stages of their execution. This parallelism maximizes the throughput and performance of the processor, enabling faster execution compared to single-cycle designs, where each instruction must be completed before the next can begin.

To optimize our pipelined design, we introduced a six-stage architecture:

1. **Instruction Fetch (IF):** Retrieves the instruction from memory.
2. **Instruction Decode (ID):** Decodes the instruction and prepares operands for execution.
3. **Forwarding Stage (EX1):** Manages data dependencies by forwarding data from later stages to ensure proper operand availability.
4. **Execution Stage (EX2):** Performs the actual operation using the ALU and resolves branch decisions.
5. **Memory Access (MEM):** Reads from or writes to memory for load and store instructions.
6. **Write Back (WB):** Writes the final result back to the register file.

   The decision to split the execution stage into **EX1** (forwarding) and **EX2** (execution and branch resolution) addresses critical path bottlenecks and enables higher operating frequencies. For a detailed analysis of the design choices and their performance implications, please refer to the **Comparative Evaluation** section.

4.3.1 Instruction Fetch (IF)

The **Instruction Fetch (IF) stage** is the first step in the pipelined processor, responsible for retrieving the next instruction from memory. It calculates the Program Counter (PC), which determines the memory address of the instruction to be fetched. This calculation can involve either incrementing the PC for sequential instructions or selecting a new target address for branches or jumps. The updated PC value is stored in a register and used to fetch the instruction from memory. The fetched instruction is then passed to the next pipeline stage (Instruction Decode) for further processing.

This stage includes three modules:

1. **Program Counter (PC) Source Multiplexer (PC_src_mux)**

An 8-to-1 multiplexer that determines the next value for the program counter (pc_reg_in) based on the control signal (pc_src). It supports sequential execution, branching and jump instructions by selecting the appropriate input. The selected value is passed to the PC_register, ensuring the correct instruction is fetched during the next cycle. Note, PFC stands for program flow control to denote that this changes the flow of the program

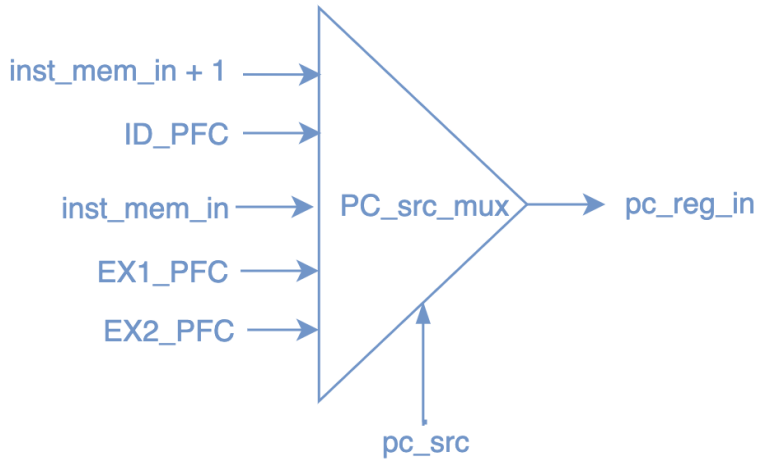


Figure 13: 8-to-1 PC Source Multiplexer.

Table 4: PC_src control signals and their corresponding inputs, sources, and descriptions.

PC_src	Input	Source	Description
000	inst_mem_in + 1'b1	Instruction Fetch (IF)	Default case (no branch, jump, or halt). Sequential PC value (current PC + 1) for normal instruction flow.
001	ID_PFC	Decode Stage (ID)	Predicted branch or unconditional jump (j, jal). (Next PC=pc+inst[9:0] or inst[9:0])
010	inst_mem_in	Instruction Fetch (IF)	Current PC value. Used when the CPU is halted, freezes the PC to halt the processor by fetching the same instruction repeatedly until halt is resolved. (Next PC=inst_mem_in)
011	EX1_PFC	EX1 (Forwarding Stage)	Jump register (jr) resolved in EX1.(Next PC=RegisterFile[Rs], forwarded)
100	EX2_PFC	EX2 (Execution Stage)	Correct branch target after misprediction resolved in EX2. (Next PC=EX2_PFC)

2. **Program Counter Register (PC_register):**

The PC register is critical for maintaining the current PC value, ensuring the fetch stage can address the instruction memory accurately for fetching instructions. It holds the current PC value and updates it based on the **pc_src_mux** output (**pc_reg_in**). Its output is the current PC value, used to index the instruction memory (**inst_mem_in**). The new value of the PC is written into the PC register on the rising edge of the clock, provided that the PC_write signal is active. If a reset signal (rst) is asserted, the PC is cleared to its initial value.

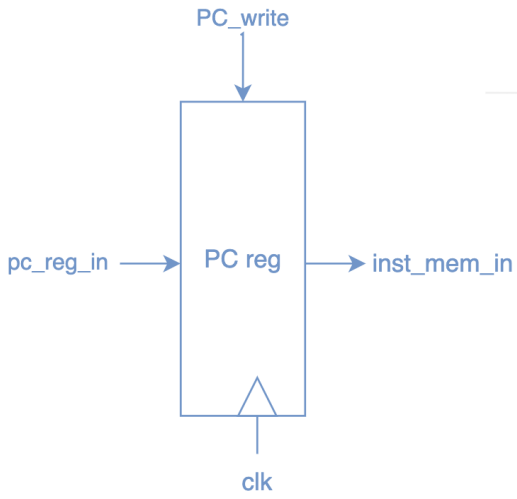


Figure 14: Program Counter Register diagram.

3. Instruction Memory (IM)

The instruction memory is word-addressable and asynchronous, it allows for efficient fetching of instructions, supporting up to 1024 instructions with 32-bit widths. Stores program instructions and retrieves the instruction at the current PC value. Takes as input the PC, **inst_mem_in**, from the PC_register output, and returns the 32-bit instruction (**inst**) fetched from memory.

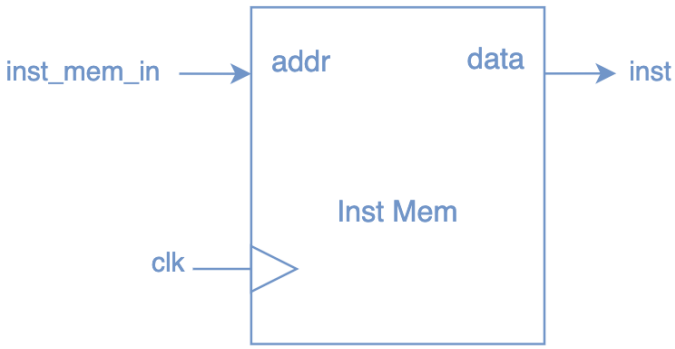


Figure 15: Instruction Memory diagram.

4.3.2 IF/ID Register

The **IF/ID buffer** acts as an intermediate pipeline register that connects the Fetch (IF) and Decode (ID) stages in a pipelined processor. It holds the program counter (PC) and the fetched instruction (IF_INST) while parsing the instruction into its opcode and register indices (rs1_ind, rs2_ind, rd_ind). The buffer updates its contents on every clock cycle unless stalled (if_id_Write is low) or reset (rst is high). It flushes its contents (clearing outputs) when IF_FLUSH is asserted, typically due to exceptions, pipeline flush requirements, or branch mispredictions. By decoupling the IF and ID stages, the buffer ensures independent execution while enabling hazard handling, branch prediction, and pipeline synchronization.

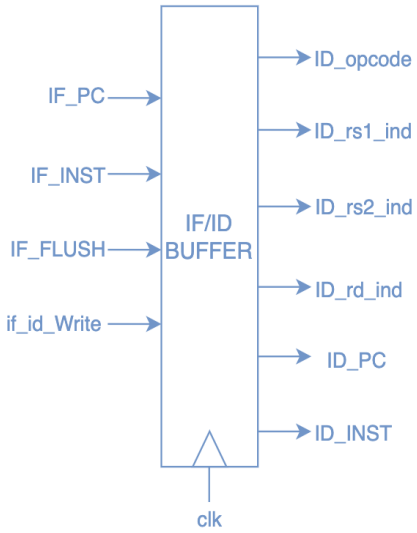


Figure 16: IF/ID buffer diagram

4.3.3 Instruction Decode Stage (ID)

The **Instruction Decode** (ID) Stage is responsible for interpreting the fetched instruction, extracting the opcode, source/destination register indices, and immediate values. It determines the control signals needed for subsequent pipeline stages and performs hazard detection to ensure the pipeline runs smoothly. Additionally, this stage calculates branch target addresses, contributing to branch prediction and resolution.

This stage includes five modules:

1. **Register File (reg_file)**

The **Register File** is a critical component in the pipeline, providing fast access to 32 general-purpose registers, each 32 bits wide. It supports simultaneous reads and a single write per clock cycle. The inputs rd_reg1 and rd_reg2 specify the registers to read, and the corresponding outputs rd_data1 and rd_data2 provide their values. The wr_reg input specifies the register to write, and wr_data is the data to be written, controlled by the reg_wr signal. During reset, all registers are initialized to zero. Register 0 is hardwired to always hold the value 0, maintaining consistency regardless of write attempts. For hazard-free operation, if the register being written matches a

read register, the module forwards the write data directly to the read output, this ensures that if a write is occurring to a register that is also being read in the same clock cycle, the new data is available immediately for the read operation, which prevents the use of stale data.

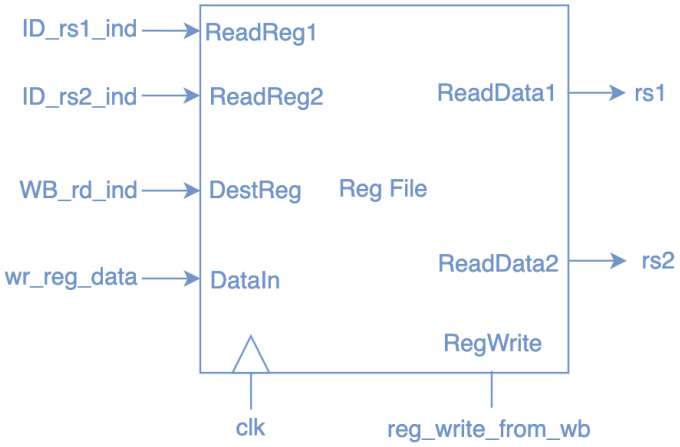


Figure 17: Register File Diagram.

2. Immediate Generation unit (immed_gen)

The **immediate generation unit** generates a 32-bit extended immediate value (Immed) based on the instruction type and opcode, with a default value of zero if no condition is met. For R-format shift instructions (sll, srl), the 5-bit shift amount (Inst[10:6]) is zero-extended to 32 bits. For I-format logical instructions (andi, ori, xori), the 16-bit immediate (Inst[15:0]) is zero-extended, while for arithmetic and branch instructions (addi, lw, sw, beq, bne, blt, etc.), the 16-bit immediate is sign-extended to preserve signed operations. For J-format instructions (j, jal), the 26-bit target address (Inst[25:0]) is zero-extended.

The module does not shift the immediate left by 2 for branch instructions because, in word-addressable memory systems, addresses already represent words, and the immediate inherently specifies word offsets, simplifying branch target calculations.

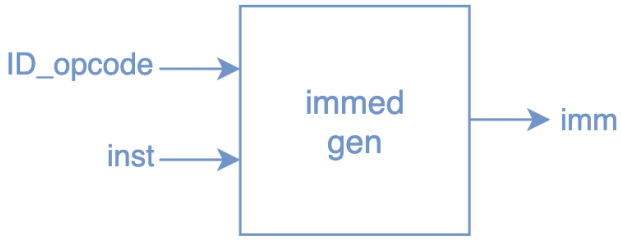


Figure 18: Immediate Generation unit diagram.

3. Control Unit

The **control unit** generates control signals based on the opcode to guide the execution of instructions in the pipeline. It determines if the instruction uses an immediate operand (is_oper2_immed), applicable to instructions like addi, andi, lw, and sw. It identifies branch instructions with signals like is_beq and is_bne, corresponding to beq and bne instructions. The regwrite signal enables register writes for instructions that modify registers, such as arithmetic and load operations, while being disabled for branches, jumps, and memory store (sw). The memread and memwrite signals handle memory operations, with memread active for load (lw) and memwrite active for store (sw). Unlike the base design, the ALU operation decoding is done in a separate module, maintaining modularity and flexibility.

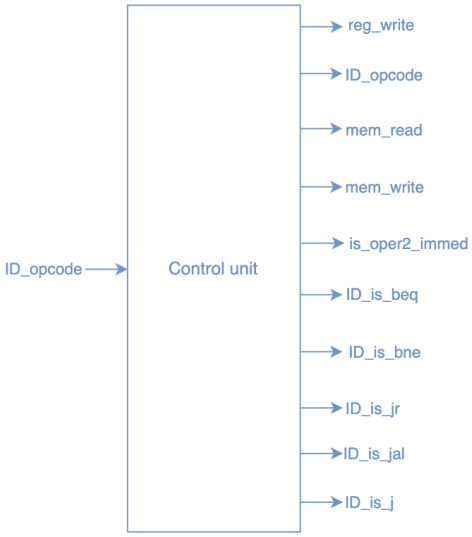


Figure 19: Control Unit diagram

4. Branch Resolver

The **Branch Resolver** is responsible for determining the source of the program counter (PC) for the next instruction. It ensures correct control flow during normal execution, branch instructions, and jumps, as well as recovery from mispredictions. The module receives inputs from both the decode (ID_opcode) and execute (EX1_opcode, EX2_opcode) stages, the Wrong_prediction signal indicating a branch misprediction, and the reset (rst) and clock (clk) signals. It outputs the PC_src signal, which selects the appropriate source for the next PC, and propagates the predicted signal (from the Branch Predictor Unit) as predicted_to_EX to the execute stage.

The PC_src output is a 3-bit signal that controls the multiplexer for selecting the next PC value based on the following conditions:

- **3'b000 (inst_mem_in + 1):** Default case for normal sequential instruction fetching.
- **3'b001 (ID_PFC):** Chosen for unconditional jump instructions (j, jal), or when a branch is predicted to be taken.
- **3'b010 (inst_mem_in):** Selected when the current instruction is hlt_inst (halt), which stops further instruction fetching.
- **3'b011 (EX1_PFC):** Used for jump register (jr) instructions, where the next PC is taken from a register value.
- **3'b100 (EX2_PFC):** Chosen when a branch misprediction occurs (Wrong_prediction is high), redirecting the PC to the correct target address.

The Branch Resolver integrates the outputs from the Branch Predictor Unit to ensure that the correct program flow is followed.

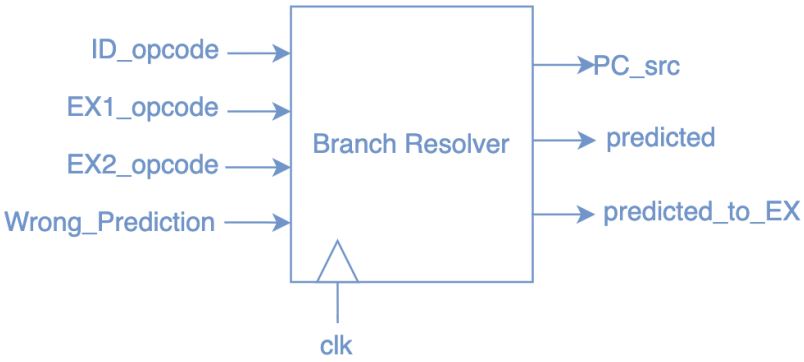


Figure 20: Branch Resolver unit diagram.

5. Branch Predictor Unit (BPU)

The **Branch Predictor Unit (BPU)** is a 2-bit finite state machine (FSM) that predicts whether a branch will be taken or not. The FSM is initialized in the **00 (Not Taken, NT)** state, and its state transitions are governed by the Wrong_prediction signal. If the previous branch prediction was correct (Wrong_prediction is 0), the FSM maintains its current state. If the prediction was wrong (Wrong_prediction is 1), the state changes to reflect the opposite outcome (from NT to T or vice versa).

Since the Wrong_prediction signal is determined in the **execute** stage, the opcode of the instruction currently in the execute stage (EX2_opcode) is used as an input for the branch predictor. This ensures that only branch instructions affect the state of the FSM within the branch predictor. The output predicted from the BPU indicates whether the branch will be taken (1) or not (0), and is based on the current state of the FSM:

- If the state is **NT** (00 or 01), the predicted signal will be 0, meaning the branch is not taken.
- If the state is **T** (10 or 11), the predicted signal will be 1, meaning the branch is taken.

On the other hand, branch instructions in the **decode** stage need to use the predicted signal, which is an output from the branch predictor. Hence, the opcode of the instruction in the **decode stage** (ID_opcode) is also required to ensure that the prediction of branch taken is only applied to branch instructions.

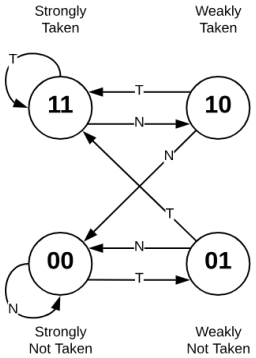


Figure 21: Branch Predictor unit state diagram.

6. Stall Detection Unit

The **Stall Detection Unit** is a vital control module that identifies scenarios where the pipeline must pause or flush instructions, inserting a stall (NOP), to handle hazards or incorrect predictions. It generates control signals to either allow normal operation or temporarily halt instruction flow through four primary conditions:

1. Branch misprediction (Wrong_prediction), the pipeline must be corrected to resume proper execution flow. In this scenario, the program counter (PC_Write) and IF/ID register (if_id_Write) are enabled to allow fetching and decoding of new instructions. The IF/ID stage is not flushed (if_id_flush = 0) since it will be updated with the correct instruction. However, the ID/EX stage is flushed (ID_EX1_flush and ID_EX2_flush are set) to remove the mis predicted instruction and prevent it from being executed. This ensures that the effects of the misprediction are cleared while the pipeline continues with the correct instructions.

2. Load-use hazards; where a load instruction in EX1 or EX2 involves a register that the current instruction in the ID stage depends on. This scenario indicates a hazard because the load's result is not yet available for forwarding. To resolve this, the pipeline is stalled: the program counter (PC_Write) and IF/ID register (if_id_Write) are disabled to prevent new instruction fetches and updates, while the ID/EX1 stage is flushed (id_ex1_flush = 1) to remove the dependent instruction. The IF/ID stage is preserved (if_id_flush = 0) since its instruction will proceed after the stall.

3. Jump register (jr) instructions; when a jr (jump register) instruction is encountered in the IF/ID stage, the pipeline halts fetching new instructions (PC_Write = 0) to calculate the jump target. The IF/ID register is allowed to update (if_id_Write = 1), and flushes the IF/ID stage (if_id_flush = 1). The ID/EX1 and EX2 stages are not flushed (id_ex1_flush = 0 and id_ex2_flush = 0), to avoid disrupting unrelated pipeline stages.

4. Normal Operation; when no hazards are detected, the pipeline operates normally, with all stages updated as required (PC_Write = 1, IF_ID_Write = 1).

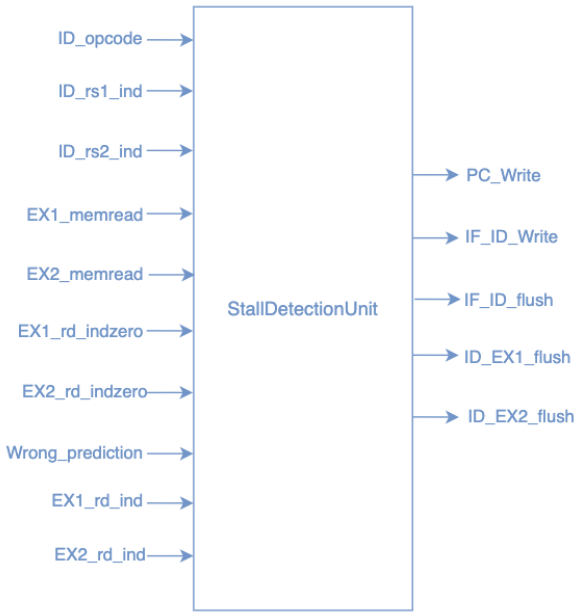


Figure 22: Stall Detection unit diagram.

4.3.4 ID/EX1 Buffer

The **ID/EX1 Buffer** is a pipeline register that bridges the Decode (ID) and Forwarding (EX1) stages, preserving all relevant instruction information across clock cycles while handling control signals and potential flush conditions. It transfers decoded fields such as the opcode (ID_opcode), source and destination register indices (ID_rs1_ind, ID_rs2_ind, ID_rd_ind), and immediate values (ID_Immed) to their EX1 stage equivalents. Additionally, it propagates the program counter (ID_PC) and the Program Flow Control (ID_PFC) incremented by 1. Control signals like ID_regwrite, ID_memread, ID_memwrite, and branch-related flags (ID_is_beq, ID_is_bne, etc.) are also stored and passed forward. If a flush condition (ID_FLUSH) or reset (rst) occurs, all outputs are set to zero, ensuring no invalid or incorrect data enters the forwarding stage.

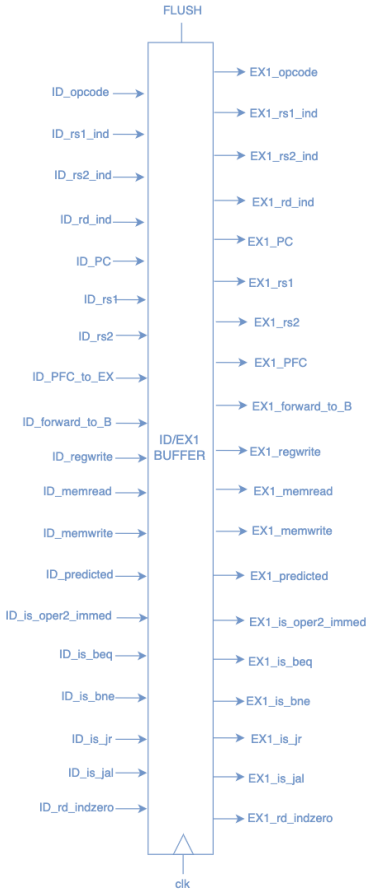


Figure 23: ID/EX1 Buffer diagram.

4.3.4 Forwarding Stage (EX1)

The forwarding stage is an essential mechanism in the pipeline architecture to handle data hazards caused by dependencies between instructions at different stages. It dynamically selects intermediate results from the execute (EX2), memory (MEM), or write-back (WB) stages to provide the required operands for subsequent instructions without pipeline stalls. This functionality is implemented using multiplexers for operand selection and forwarding control units that generate the selection signals based on the detected hazards.

The forwarding stage resolves three types of hazards:

- **ID Hazard**: Occurs when the current instruction in the EX1 stage requires data that is being computed in the EX2 stage. This typically happens when an instruction in EX2 writes to a register, and the destination register of that instruction matches a source register in the EX1 instruction.

- **EX Hazard**: Detected when the current instruction depends on data being finalized in the MEM stage. For example, an ALU operation result or memory load value may still be in the MEM stage but is needed by the current instruction in EX1. The condition for this hazard is that MEM writes to a register, and the destination register matches a source register of the current instruction in EX1.

- **MEM Hazard**: Happens when the required operand for the current instruction is being written to the register file by the WB stage. Instead of waiting for the write-back process to complete, the forwarding stage directly routes the value from the WB stage to EX1. The condition for this hazard is that WB writes to a register, and the destination register matches a source register of the current instruction in EX1.

In addition to resolving hazards, the forwarding stage also determines the next program counter value for jr instruction. If the instruction is a jr (jump register), the target address is taken from the forwarded first operand(R[rs]), the output of the first multiplexer (alu_oper1).

The main components of the forwarding stage include:

1. **ALU Operand 1 Multiplexer (alu_oper1)**

   This 4-to-1 multiplexer selects the first operand (oper1) for the ALU. The selection signal, alu_selA, is generated by the **forwardA** module. The forwardA module analyzes hazards at various stages and determines the appropriate source for oper1.

   - **00**: No hazard; operand is fetched from the register file (rs1).
   - **01:** MEM hazard; operand is forwarded from the WB stage – data to writeback.
   - **10:** EX hazard; operand is forwarded from the MEM stage – ALU result.
   - **11**: ID hazard; operand is forwarded from the EX-stage - ALU result

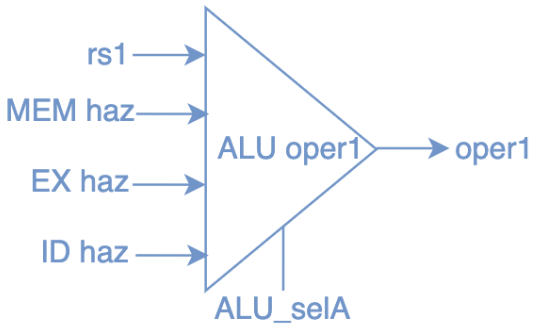


Figure 24: ALU Operand 1 mux diagram.

2. **ALU Operand 2 Multiplexer (alu_oper2)**

   This 4-to-1 multiplexer selects the second operand (oper2) for the ALU. The output, oper2, is selected based on the control signal alu_selB, generated by the forwardB module.

   - **00:** No hazard; if the instruction specifies an immediate value, it is used directly as oper2, or the value is taken from the register file (rs2).
   - **01:** MEM hazard; operand is forwarded from the WB stage – data to writeback.
   - **10:** EX hazard; operand is forwarded from the MEM stage – ALU result.

- **11**: ID hazard; operand is forwarded from the EX-stage - ALU result

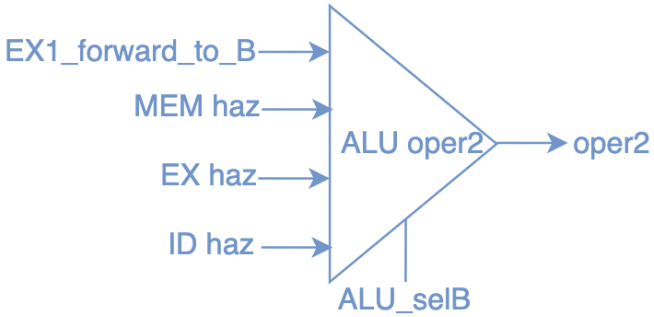


Figure 25: ALU Operand 2 mux diagram.

3. **Store Value Forwarding (store_rs2_mux)**

   The store_rs2_mux multiplexer resolves the value to be stored in memory for store instructions. The store_rs2_forward control signal (generated by the forwardC module) determines the input source:

   - **00**: No hazard; operand is fetched from the register file - rs2_in, the value of the source register.
   - **01:** MEM hazard; operand is forwarded from the WB stage – data to writeback.
   - **10:** EX hazard; operand is forwarded from the MEM stage – ALU result.
   - **11**: ID hazard; operand is forwarded from the EX-stage - ALU result

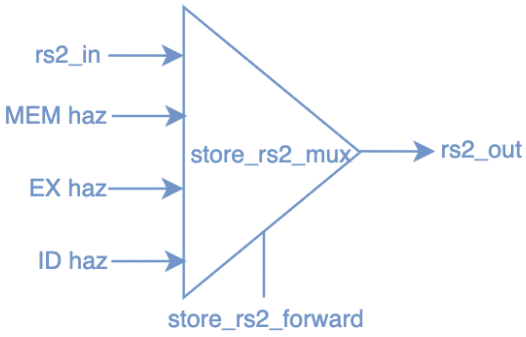


Figure 26: Store Value Forwarding mux diagram.

4.3.5 EX1/EX2 Buffer

The **EX1/EX2 Buffer** serves as a pipeline register between the forwarding stage (EX1) and the execution stage (EX2). It ensures seamless data flow by storing and passing crucial control signals and operands required for execution. The buffer retains ALU inputs (ALU_OPER1, ALU_OPER2), opcode, register indices (rs1_ind, rs2_ind, rd_ind), the program counter (PC), and data forwarding signals (forward_to_B). Control signals such as register write, memory read/write, branch indicators (is_beq, is_bne), and special instructions like Jump and Link (is_jal) are also stored to facilitate accurate operation in the execution stage. A flush mechanism clears the buffer in case of mispredictions or resets, maintaining pipeline consistency. This module is essential for coordinating operations across the pipeline stages.

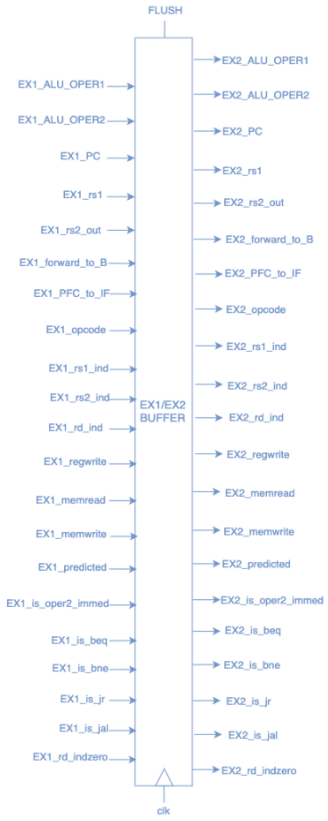


Figure 27: EX1/EX2 Buffer diagram.

4.3.6 Execution Stage (EX2)

The **Execution (EX) stage** is a critical component of the pipeline responsible for performing arithmetic, logical, and control operations based on decoded instructions. It is the stage where computations take place, and decisions regarding control flow are made. The EX-stage takes inputs such as operands, the current program counter (pc), and an opcode from the decode stage and produces outputs such as the computed result (alu_out) and control signals indicating branch decisions or prediction mismatches (Wrong_prediction).

The modules involved in the execution stage include:

1. ALU Operation Decoder (ALU_OPER)

   The **ALU_OPER** module determines the specific operation the ALU should perform based on the instruction's opcode. It translates the 12-bit opcode into a 4-bit ALU_OP control signal, which directly maps to the operation codes supported by the ALU. This module categorizes a wide range of instructions into appropriate ALU operations:

   - **0000 (add)**: Arithmetic addition (add, addi, addu) and address calculations (lw, sw, jal, jr, j).
   - **0001 (sub)**: Arithmetic subtraction (sub, subu).
   - **0010 (and)**: Bitwise AND operations (and_, andi).

- **0011 (or)**: Bitwise OR operations (or_, ori).
- **0100 (xor)**: Bitwise XOR operations (e.g., xor_, xori).
- **0101 (nor)**: Bitwise NOR operations (e.g., nor_).
- **0110 (shift left)**: Logical left shift (sll).
- **0111 (shift right)**: Logical right shift (srl).
- **1000 (less-than)**: Signed comparison for less-than (slt, slti).
- **1001 (greater-than)**: Signed comparison for greater-than (sgt).

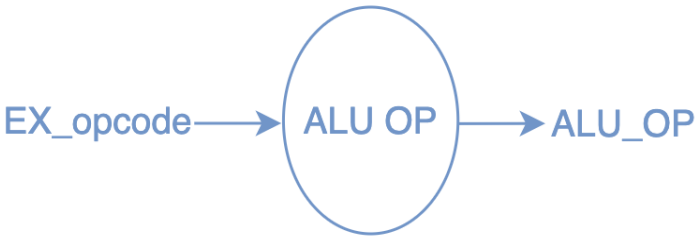


Figure 28: ALU Operation Decoder diagram.

2. **Arithmetic Logic Unit (**ALU)
   The ALU module performs arithmetic and logical operations on two 32-bit inputs, A and B, as determined by a 4-bit ALU operation code (ALUOP) provided by the ALU_OPER module. The result of the operation is stored in the res output, while the carry flag (CF) is updated for addition, subtraction, and shifts to indicate overflow or carry out. For shifting operations, the number of positions is determined by B, and care is taken to avoid overflows by checking if the shift exceeds the bit-width. The zero flag (ZF) is set if the result is zero.

   However, the final output, alu_out, which is used in the execution stage, is conditionally assigned. If the instruction is a Jump and Link (JAL), alu_out is assigned the value of pc + 1 to calculate the correct jump address for control flow operations. If the instruction is not JAL, alu_out is simply assigned the value of res, which holds the result of the ALU operation.

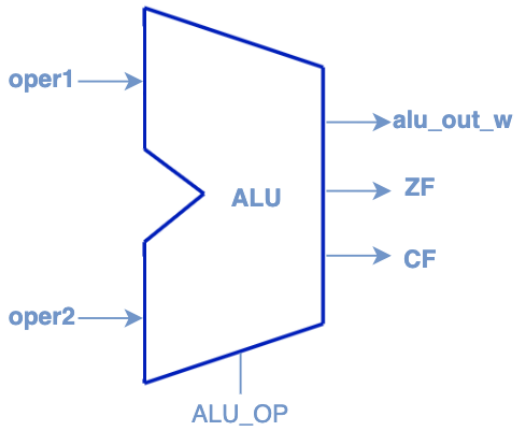


Figure 29: ALU module diagram.

3. **Branch Decision Unit (BDU)**

The BDU is designed to determine the outcome of conditional branch instructions. Its purpose is to evaluate whether a branch condition is satisfied based on the relationship between two operands (oper1 and oper2) and specific branch control signals (is_beq for branch-equal and is_bne for branch-not-equal).

The BDU works by comparing the two operands to determine equality. A submodule, compare_equal, performs this comparison by applying XOR gates across corresponding bits of oper1 and oper2. The result indicates whether the operands are equal. Using this comparison, the BDU computes the branch decision:

- If is_beq is active and the operands are equal, the branch is taken (is_beq_taken).
- If is_bne is active and the operands are not equal, the branch is taken (is_bne_taken).

The final branch decision (BranchDecision) is the logical OR of is_beq_taken and is_bne_taken, indicating whether a branch should be taken based on the conditions specified by the instruction. This result, combined with branch prediction signals, is used to assess whether the pipeline has correctly predicted the branch direction or needs to handle a misprediction (Wrong_predcition).

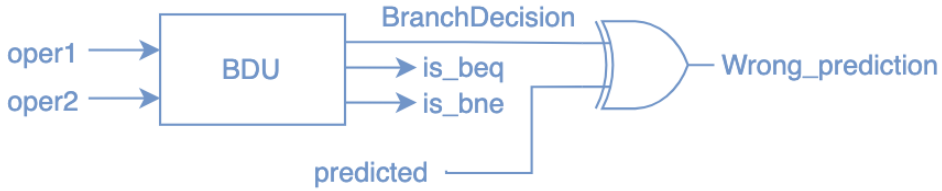


Figure 30: Branch Decision Unit diagram.

4. **Compare Equal (compare_equal)**

The **Compare Equal** module is a fundamental submodule within the BDU. Its purpose is to determine if two 32-bit operands, a and b, are equal by performing a bitwise comparison. This result is a critical input for conditional branching decisions, especially for instructions like branch-equal (is_beq) and branch-not-equal (is_bne).

The module works by implementing a bitwise XOR operation on each corresponding bit of the two operands. The XOR operation outputs 0 if the bits are the same and 1 if they differ. The results of these XOR operations are then fed into a large NOR gate. The NOR gate checks if all XOR outputs are 0, indicating that all bits in a and b are identical. If true, the output out is set to 1, signifying that a and b are equal; otherwise, it is set to 0. This design is both efficient and hardware-friendly, leveraging combinational logic to produce a single-cycle result.

4.3.7 EX2/MEM Buffer

The EX2/MEM buffer is a critical pipeline register that facilitates the smooth transition of data and control signals from the Execution (EX2) stage to the Memory (MEM) stage. It stores and forwards the results of the ALU operation (EX2_ALU_OUT), the value to be written to memory in the case of store instructions (EX2_rs2), and essential metadata such as the program counter (EX2_PC) and opcode (EX2_opcode). Additionally, it carries control signals like EX2_memread, EX2_memwrite, and EX2_regwrite, which guide the operations in the MEM stage. The EX2_rd_ind (destination register index) and related metadata (EX2_rd_indzero, EX2_rs1_ind, EX2_rs2_ind) ensure proper forwarding and hazard detection in subsequent stages. Sensitive to clock and reset signals, the buffer latches input signals on the clock's positive edge unless EX2_FLUSH is asserted, which clears the buffer to handle mispredictions. This mechanism ensures only valid data is passed forward, maintaining pipeline integrity. Outputs from the buffer, such as MEM_ALU_OUT and MEM_rs2, directly enable memory access and data forwarding in the MEM stage, ensuring synchronization between pipeline stages.

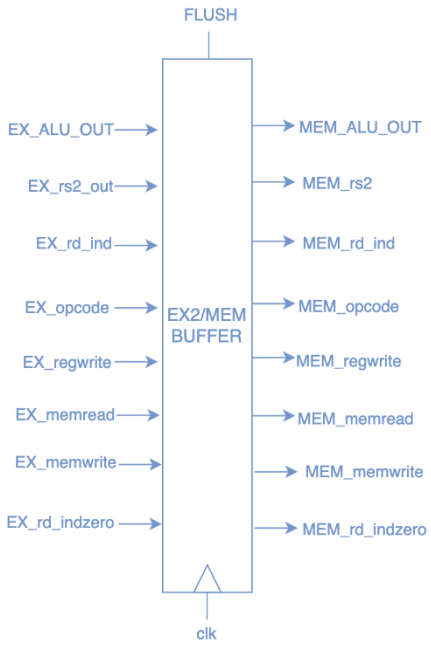


Figure 31: EX2/MEM Buffer diagram.

4.3.8 Memory Stage (MEM)

The **Memory Stage (MEM)** is responsible for performing memory access operations, including reading from or writing to the data memory, as required by load (lw) and store (sw) instructions. The stage receives the memory address (addr) calculated in the execution stage and the data to be written (wdata) for store operations. It uses the **Data Memory (DM)** module to handle the memory interactions.

Data Memory (DM)

The **Data Memory (DM)** is a 32-bit wide, 1024-entry memory array, providing a total storage capacity of **4 KB**. It is designed to support read and write operations for instructions like lw (load word) and sw (store word) in the memory stage of the pipeline. The addr input specifies the memory address, while Data_In and Data_Out are used for writing data to and reading data from

memory, respectively. The lower 10 bits of the 32-bit address are used to index the memory, allowing access to any of the 1024 entries. On a positive clock edge, if the WR (write) signal is asserted, the data from Data_In is written to the specified memory location. Simultaneously, Data_Out outputs the data stored at the indexed address, enabling single-cycle memory access. The module initializes memory with predefined values via DM_INIT.INIT for simulation and testing in VS Code and a MIF file if Quartus is used in simulation.

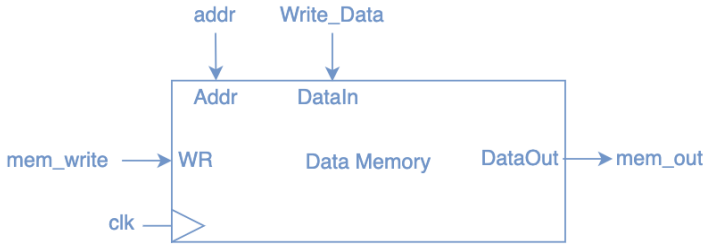


Figure 32: Data Memory module diagram.

4.3.9 MEM/WB Buffer

The **MEM/WB buffer** serves as the final pipeline register, transferring data and control signals from the **Memory (MEM)** stage to the **Write-Back (WB)** stage. It stores critical outputs, including the ALU result (MEM_ALU_OUT), the data read from memory (MEM_Data_mem_out), and register indices (MEM_rd_ind, MEM_rs1_ind, MEM_rs2_ind). The control signals (MEM_memread, MEM_memwrite, MEM_regwrite) are also passed through this buffer to govern operations in the WB stage. On each clock cycle, the buffer updates its outputs (WB_ALU_OUT, WB_Data_mem_out, etc.) with the corresponding MEM-stage values, ensuring accurate data propagation. Additionally, the **hlt** signal is generated in this buffer when a halt instruction (hlt_inst) is detected, marking program termination. This halt signal is integrated into the halt logic, which uses a nor gate to stop the processor clock, effectively halting execution and ensuring safe program termination.

The **halt logic** is designed to stop the processor's operation by controlling the clock signal when a halt condition is detected. It takes two inputs: input_clk, the main clock signal driving the system, and hlt, a signal asserted when a halt instruction (hlt_inst) is encountered, typically during the MEM/WB stage. The nor gate ensures that the output clock (clk) is forced to 0 when the hlt signal is high, effectively stopping the processor. If the hlt signal is low, the nor gate allows the clk to follow the input_clk, enabling normal processor operation. This simple mechanism halts execution in a controlled manner, ensuring the processor stops processing instructions after a halt command, which is crucial for safely terminating the system's operation.

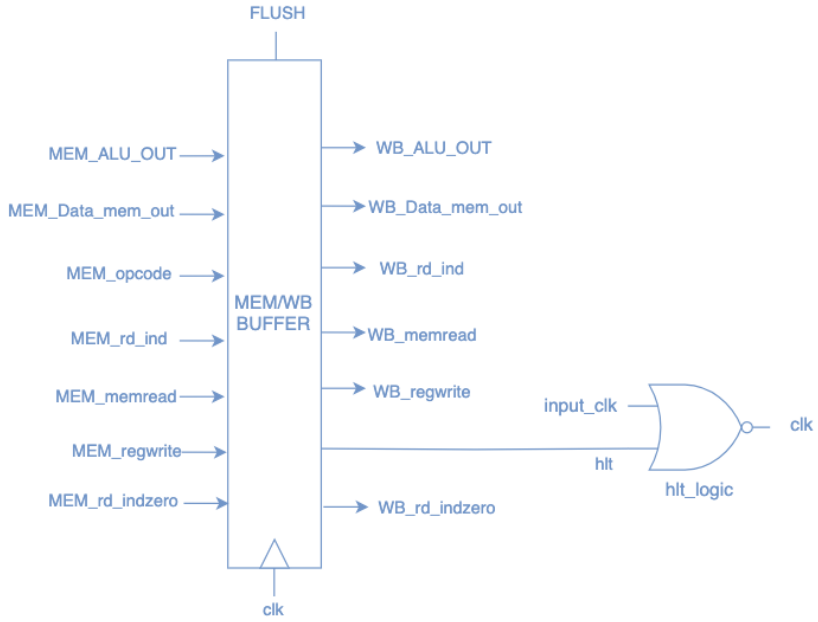


Figure 33: MEM/WB Buffer diagram.

4.3.10 Write-Back Stage (WB)

The **Write-Back (WB) stage** is the final stage in the pipeline, where the results of computations or memory operations are written back to the register file. It takes two inputs: mem_out, the data read from memory for load (lw) instructions, and alu_out, the result of arithmetic or logical operations performed in the execution stage. The mem_read control signal determines the source of the data to be written back. If mem_read is asserted, the data from mem_out is selected; otherwise, alu_out is used. The output, wdata_to_reg_file, is forwarded to the register file as the write data for the destination register. This stage ensures that the results of executed instructions are correctly updated in the register file, enabling subsequent instructions to access accurate and up-to-date values.

4.3.11 Instruction Flow

The first stage in the pipeline is the **Instruction Fetch (IF)** stage, where the instruction is fetched from memory. The **Program Counter (PC)** holds the address of the instruction to be fetched. The address is passed to the instruction memory module, which retrieves the instruction. The **PC_src_mux** decides the next PC value based on the branch or jump conditions, forwarding predictions, or a simple increment. If the instruction is part of a branch or jump, this stage adjusts the PC accordingly. The fetched instruction is then passed to the next stage through the **IF_ID_buffer**, where it is prepared for decoding. Additionally, the PC is updated and stored in the **PC_register** for future use.

As the fetched instruction reaches the **Instruction Decode Stage (ID)**, the **REG_FILE** reads the values of **rs1** and **rs2** based on the instruction's source register indices, **ID_rs1_ind** and **ID_rs2_ind**. These register values are used for further computations and decisions. The **Immed_Gen_unit** decodes the

instruction and generates the immediate value if required, which is either passed to **forward_to_B** if the instruction uses an immediate operand or remains as **rs2** otherwise. The **BranchResolver** checks if the instruction is a branch (e.g., **BEQ**, **BNE**) and evaluates whether the branch should be taken based on the **predicted** outcome, adjusting the **PC_src** for the next instruction. The **StallDetectionUnit** identifies any hazards, ensuring the pipeline behaves correctly by controlling when the instruction should be stalled or flushed, such as in case of incorrect branch predictions or data hazards. The control signals such as **reg_write**, **mem_read**, **mem_write**, and others are determined by the **control_unit**, guiding how the instruction should proceed. The instruction also calculates the **PFC_to_IF** and **PFC_to_EX** values, which dictate the next program counter locations based on branch decisions. Finally, if any branch is predicted to be taken, the instruction flow is adjusted accordingly, and any necessary pipeline flushes are triggered to correct mispredictions. Decoded values, control signals, and immediate data are passed to the ID_EX_buffer1 for the next stage.

In the **Forwarding (EX1)** stage, the instruction's operands are forwarded to the appropriate execution stages to handle data hazards. Specifically, the values of rs1 and rs2 are selected using forwarding modules A, B, and C. **Forwarding Module A** selects the operand for **oper1** based on the hazard signals (id_haz, ex_haz, or mem_haz) through the alu_selA control signal. Similarly, **Forwarding Module B** selects **oper2** based on the forwarding logic (id_haz, ex_haz, or mem_haz) controlled by alu_selB. **Forwarding Module C** handles the forwarding of the rs2 value via the store_rs2_forward control signal. If the instruction is a jump (is_jr), the **EX_PFC** (Program Counter) is forwarded to the IF stage. The output of this stage, including forwarded operands and intermediate results, is passed to the next stage through the **ID_EX_buffer2** module.

As the instruction reaches the **Execution Stage (EX2)**, it brings along its operands, **oper1** and **oper2**, along with the **PC** (Program Counter) and the **opcode**. First, the opcode is passed to the **ALU_OPER** module, which determines the type of operation that needs to be performed, producing the **ALU_OP** control signal. Next, the two operands are sent to the **ALU**, which executes the operation defined by **ALU_OP**. If the instruction involves a **jump and link** (such as **JAL**), the **PC** is incremented by 1 and is forwarded as the result instead of the ALU computation. At the same time, if the instruction is a branch (like **BEQ** or **BNE**), the **BranchDecision Unit** evaluates the condition using **oper1** and **oper2**. It generates a **BranchDecision** indicating whether the branch should be taken or not. The decision is compared with the **predicted** branch outcome, and if there's a mismatch, the **Wrong_prediction** signal is set, which can trigger corrections in the pipeline. Results of these operations are stored in the **EX_MEM_buffer** module, which passes the data to the memory stage.

The instruction proceeds to the **Memory Stage (MEM)**, it arrives with the address that was calculated earlier, in the **EX2 stage**, and any data to be written to memory (if it's a store instruction). If the instruction is a load operation, the **mem_read** signal is asserted, prompting the **data memory** module (**DM**) to retrieve the data from the memory location specified by the **addr**. The retrieved data is then passed back as **mem_out**. If the instruction is a store operation, the **mem_write** signal is asserted, causing the **Write_Data** (passed from the execution stage) to be written to the specified memory address. The **mem_out** value (for load operations) or the updated memory (for store operations) will be passed onto the next stage. Results from this stage are buffered in the **MEM_WB_buffer** module for use in the write-back stage.

The final stage of execution is handled by the **Write-Back stage (WB)**. This stage writes the results of the instruction back to the register file. For load instructions, the data fetched from memory is written to the destination register. For computational instructions, the ALU result is written back. By updating the

register file, the instruction completes its life cycle, and its result becomes available for subsequent instructions.

4.3.11 Final Datapath

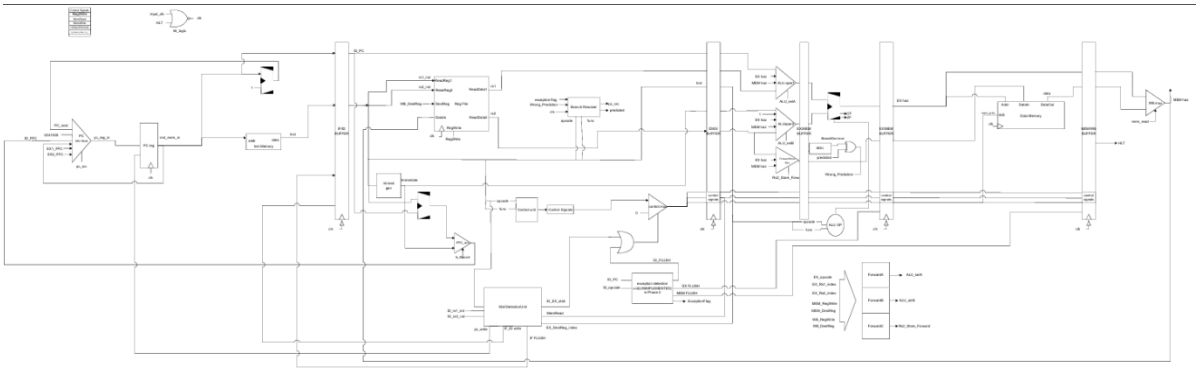


Figure 34: final Datapath diagram.

please refer to the drar.io file or the pdf file for clearer view of the complete data path

4.4 Coding and Software Development

Integrating software into CPU development is essential for creating efficient and effective computing systems. This process involves several key components, including building an assembler, a cycle-accurate simulator, and a real-time cycle-accurate simulator. The assembler translates high-level code into machine language, enabling the CPU to execute instructions. A cycle-accurate simulator provides detailed insights into how instructions interact with the CPU's architecture over time, allowing developers to optimize performance and identify bottlenecks. Meanwhile, a real-time cycle-accurate simulator simulates the CPU's behavior under actual operating conditions, offering a more dynamic testing environment.

To facilitate these developments, libraries such as LibAN, LibCPU, and MIPS Assembler can be utilized. LibAN offers tools for assembly language processing, while LibCPU provides essential functions for simulating CPU operations like the ability to simulate the single cycle and the pipeline architectures. The MIPS Assembler is particularly useful for translating MIPS architecture code into executable formats. Together, these tools enhance the integration of software into CPU design, ensuring that both hardware and software components work seamlessly together for optimal performance.

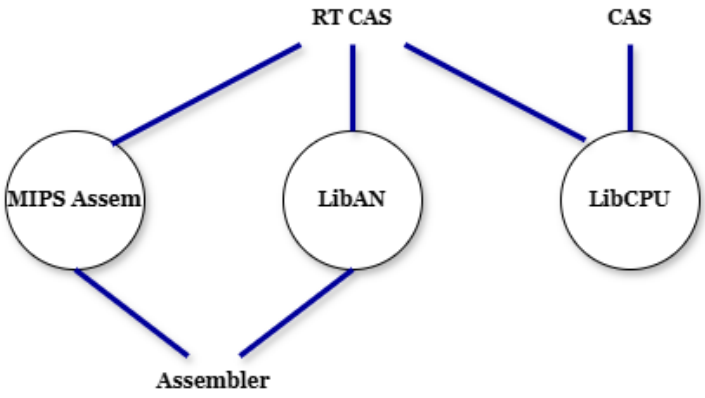


Figure 35: Libraries needed for the software portion.

4.4.1 Assembler

An assembler is a software program that translates assembly language, a low-level programming language, into machine code, which is the binary format that a computer's processor can execute.

This implemented assembler is a Windows Forms desktop application that utilizes the developed library for assembling the program based on the MIPS instruction set and supports all the instructions of the CPU. The design favors practicality and efficiency as translating assembly to machine code is a tedious task even with console application – since using a console requires re-running the program and using the new machine code if anything is changed in the initial code.

Another downside for console applications is the fragility when using a text file as an input. Assuming that the program runs correctly with no bugs, it is prone to incorrect assembly code if the file itself was modified.

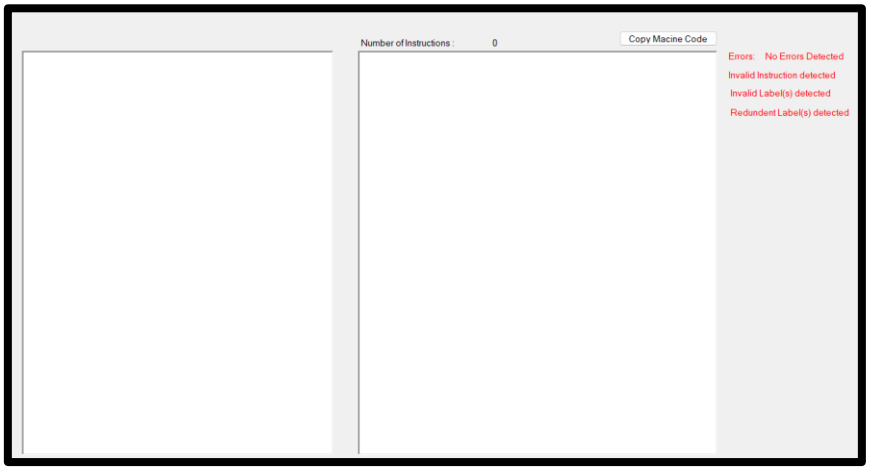


Figure 36: The simple user interface of the assembler.

Creating a user interface (UI) for an assembler application that effectively communicates syntax errors and provides functionality for copying machine code is essential for enhancing the user experience.

To streamline the process of using this CPU, the assembling of the instructions happens in real time as the code is being written. It checks the syntax and displays a "no error detected" label if it is correct. However, if there is an error, it will display the type of the error:

- Invalid Instruction if not recognized by the assembler.

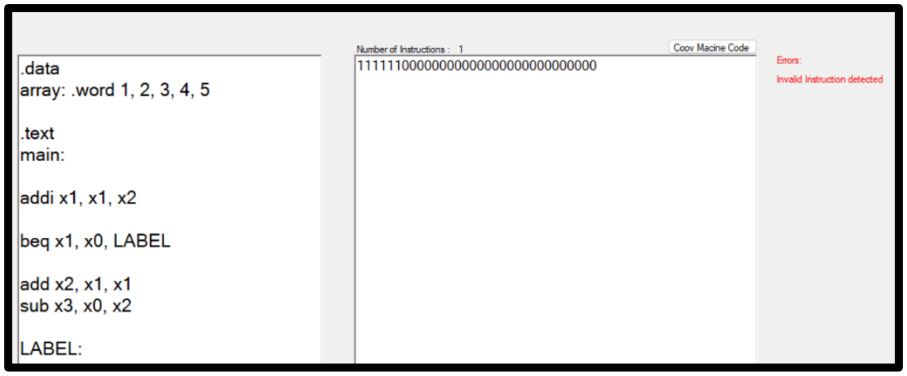


Figure 37: Output of invalid instruction.

- Invalid or Redundant Label if there are duplicate labels that could lead to ambiguity.

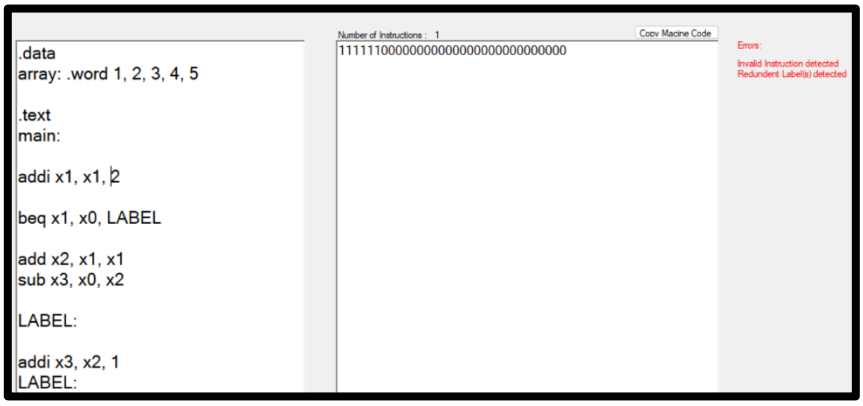


Figure 38: Output of invalid label.

The output of the assembler is the machine code which can be represented in both binary or hexadecimal numbers, with a button allowing the code to be copied directly.

Bin: "{binary machine code}", Hex:{hexadecimal machine code}; // corresponding instruction

It will also display the number of instructions in the program – all done as the code is getting updated at real time, giving instant feedback and analysis which is especially helpful in writing the benchmarks and algorithms.

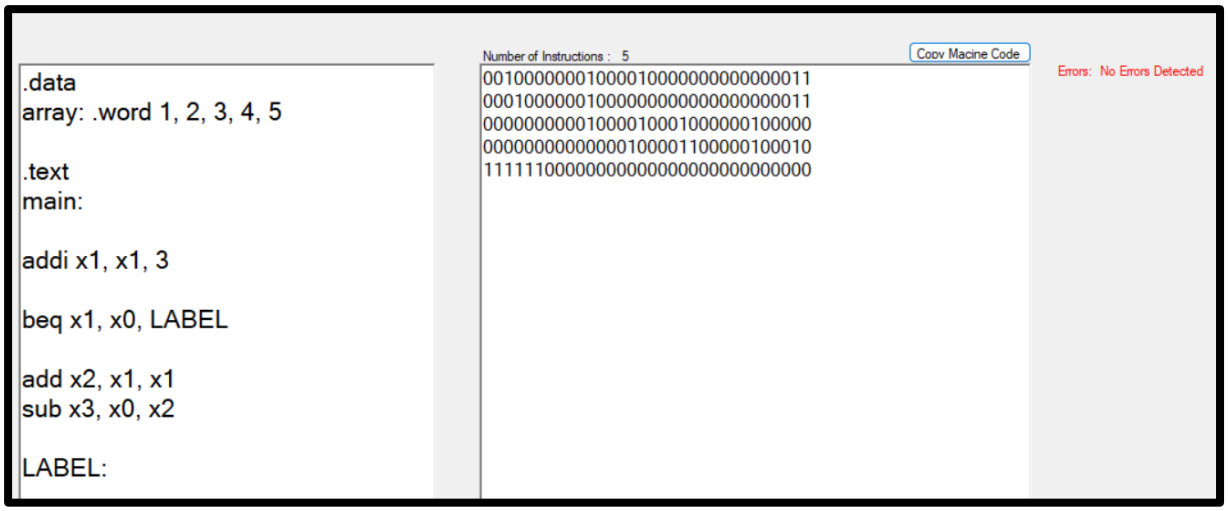


Figure 39: Output of a clean bug-free assembly code.

By pressing the "copy machine code" button, the following output is copied:

```
Bin: "00100000001000010000000000000011", Hex: 0x20210003; // addi x1 3
Bin: "00010000001000000000000000000011", Hex: 0x10200003; // beq x1 x0 3
Bin: "00000000001000010001000000100000", Hex: 0x00211020; // add x2 x1
Bin: "00000000000000100001100000100010", Hex: 0x00021822; // sub x3 x0 x2
Bin: "11111100000000000000000000000000", Hex: 0xFC000000; // hlt
```

Programming Language and Tools:

The primary programming language used for developing the Windows Forms application is C# as it provides robust support for GUI developments. The target language is assembly which is parsed and translated into machine code. The integrated development environment (IDE) used for coding, debugging and testing is Visual Studio. Git is used for version control to manage changes in the codebase and maintain a history of project developments.

Code Flowchart and Algorithms:

The application is initialized by loading necessary libraries, user input is captured from textboxes then each line of code is checked for correct instructions, correct non-redundant labels.

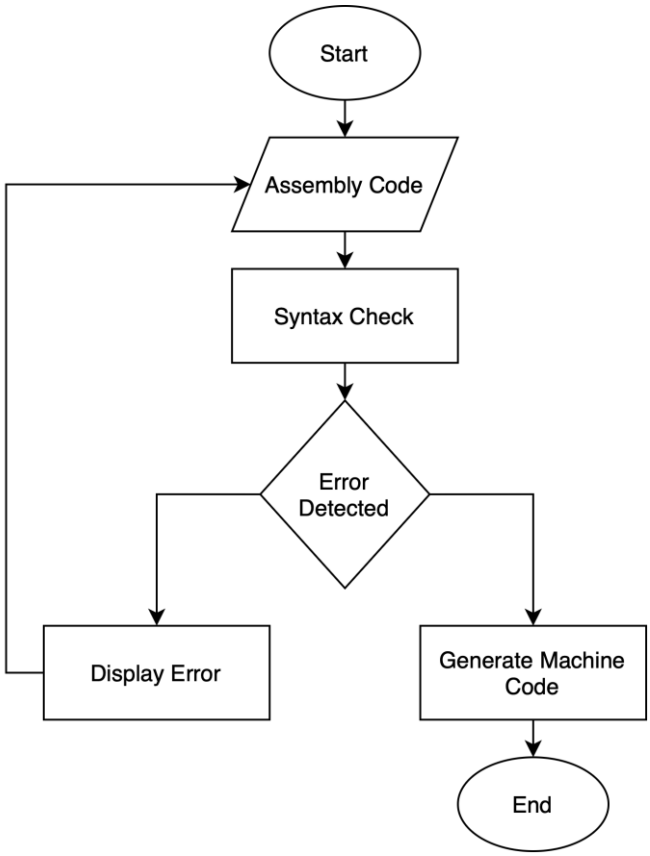


Figure 40: Flowchart of how the assembler works.

To begin writing code in the assembler and translating into machine code, the .text directive should be included to specify the starting point for executing instructions. The corresponding machine code for the

written program is displayed on the right side, with a HLT instruction automatically appended as the final instruction. This ensures that the CPU halts execution and the simulation stops counting the number of consumed cycles.

4.4.2 Cycle Accurate Simulator

The **Cycle-Accurate Simulator (CAS)** is a versatile tool designed to simulate MIPS-based CPU architectures with cycle-level precision. It models the processor at the clock cycle level, accurately reflecting the timing and behavior of each hardware component. It captures the flow of instructions through the pipeline stages, the state of registers and memory at each cycle, and the occurrence of pipeline stalls and hazards. This level of simulation is essential for performance analysis, debugging, and verification.

We developed the CAS as a console application. In the next section, we introduce the Real-time cycle-accurate simulator (RT CAS), with common implementation details as the regular CAS. The simulator is implemented as a modular, scalable library, allowing for easy integration, as it can be imported into various programs and facilitates future enhancements. It supports both single-cycle and pipelined processor designs, allowing users to compare and analyze the performance and functionality of each architecture using metrics like instruction count, cycles, and stalls.

The CAS is implemented as a C# library (libCPU.cs) with two primary CPU models:

1. **SingleCycle**: Implements a CPU that executes one instruction per clock cycle..
2. **CPU5STAGE**: Simulates a pipelined CPU with hazard detection and forwarding, processing multiple instructions simultaneously across fetch, decode, execute, memory, and write-back stages.

Both models rely on shared data structures and utilities provided by the **MIPS** class, which include:

- **Instruction Representation**: Instruction struct captures the details of each instruction. It includes fields for machine code (mc), opcode, registers (rs, rt, rd), immediate values, and the instruction format (R, I, J). The instruction decoding and execution is done using the functions:
    1. **decodemc(string mc, int pc)**: Decodes the binary machine code (mc) into a structured Instruction object. It identifies the opcode, function code, instruction format (R, I, J), register indices (rs, rt, rd) and immediate values.
    2. **get_inst_aluop(Mnemonic mnem)**: Maps a mnemonic (e.g., add, sub) to the corresponding ALU operation.
    3. **execute_inst(Instruction inst)**: Executes ALU operations such as addition, subtraction, bitwise operations, and comparisons based on the decoded instruction.
    4. **get_oper1, get_oper2**: Retrieve operands for ALU execution based on instruction format and immediate values.

- **Memory and Registers**: Instruction Memory (IM), Data Memory (DM), and Register File (RF) are initialized and managed using the function **InitMipsCPU**:

1. **Instruction Memory (IM)**: Loaded with program instructions, padded with NOP if the instruction count is less than 1024.
2. **Data Memory (DM)**: Pre-allocated memory blocks initialized with default values.
3. **Register File (regs)**: A 32-register array initialized to zeros.

- **Hazard Handling**: Data hazards are addressed using forwarding logic, which provides operands directly from later pipeline stages. If forwarding is not possible, the simulator inserts bubbles (stalls) to delay execution. Control hazards, arising from branches or jumps, are handled using branch prediction and flushing mechanisms to correct mispredictions, by flushing invalid instructions and updating the PC, and implements a branch resolver (BranchResolver). It uses functions to detect and handle hazards, which include:

    1. **detect_exception(Instruction inst, Stage stage)**: Identifies invalid operations, such as, division by zero, out-of-bound memory access, and invalid opcode or function code.
    2. **InsertBubble(string source)**: Introduces bubbles (stalls) in the pipeline to resolve hazards.
    3. **handle_exception(Exception e)**: Redirects the pipeline to the exception handler address (HANDLER_ADDR) and clears pipeline stages.

The Run method in both SingleCycle and CPU5STAGE loops through the instruction memory until:

- A hlt instruction is encountered.
- An exception is raised.
- The instruction memory is exhausted.

The Program class in Program.cs serves as the entry point for the simulator. It parses command-line arguments to determine the CPU type (SingleCycle or PipeLined) and input/output file paths. The instructions are loaded into memory, and the appropriate CPU model is instantiated. The Run() method executes the simulation, capturing performance metrics such as cycle count, pipeline stalls, and register/memory states. Results are written to the specified output file for analysis.

The CAS outputs a detailed log of the simulation, including register contents, memory states, and cycle counts. Registers display both signed and unsigned values, aiding debugging and verification. The cycle count and stall metrics provide insights into the efficiency of the pipeline. These outputs serve as a comparison metric for optimizing processor designs and understanding the trade-offs between single-cycle and pipelined architectures.

The simulator's modular design makes it scalable and extensible. New instructions, features like floating-point operations, or advanced branch prediction algorithms can be integrated with minimal effort. Additionally, its transition to real-time simulation is straightforward due to its well-structured and abstracted implementation.

4.4.3 Real-Time Cycle Accurate Simulator

A real-time-cycle-accurate simulator is a specialized tool built upon the standalone cycle accurate simulator program to deliver timely feedback. It is designed for reusability and adaptability across various applications. Its use cases include evaluating new architectural features or optimizations, analyzing system performance and observing how changes in code affect performance metrics.

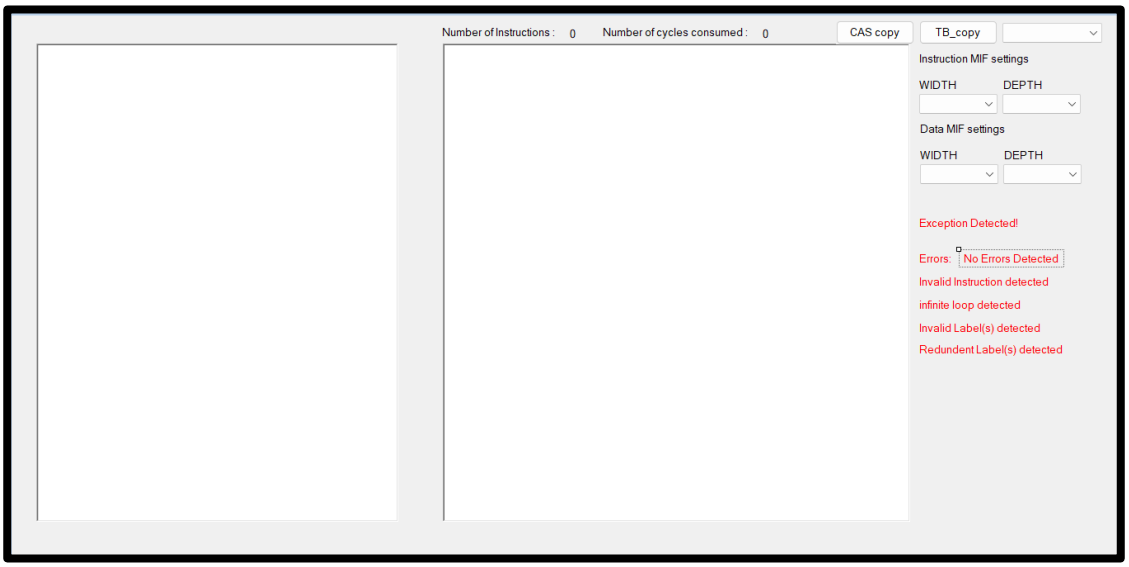


Figure 41: Real-time CAS user interface.

The simple user interface helps facilitate new users' introduction to the simulator with clear labels.

By initializing the data memory, the value of the entries will be displayed after the register file values.

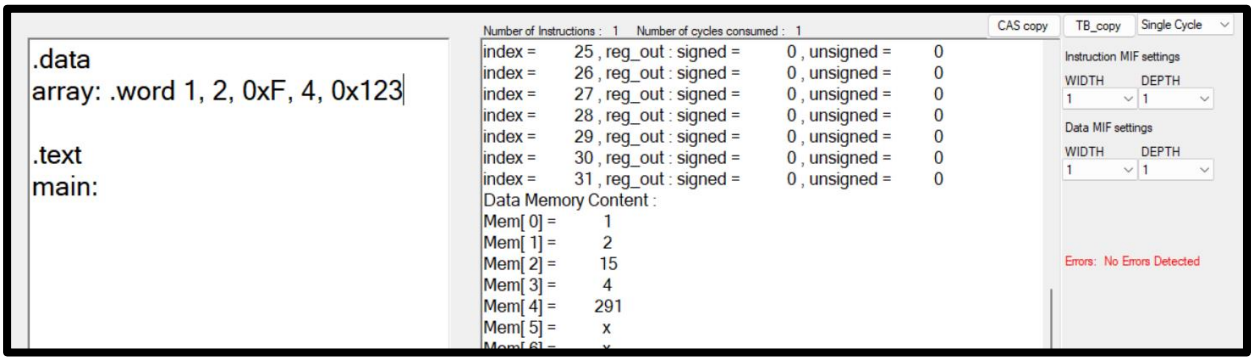


Figure 42 Output of initializing the data memory.

Then, inside the text directive, an arbitrary program is written to display the functionality of the real-time CAS.

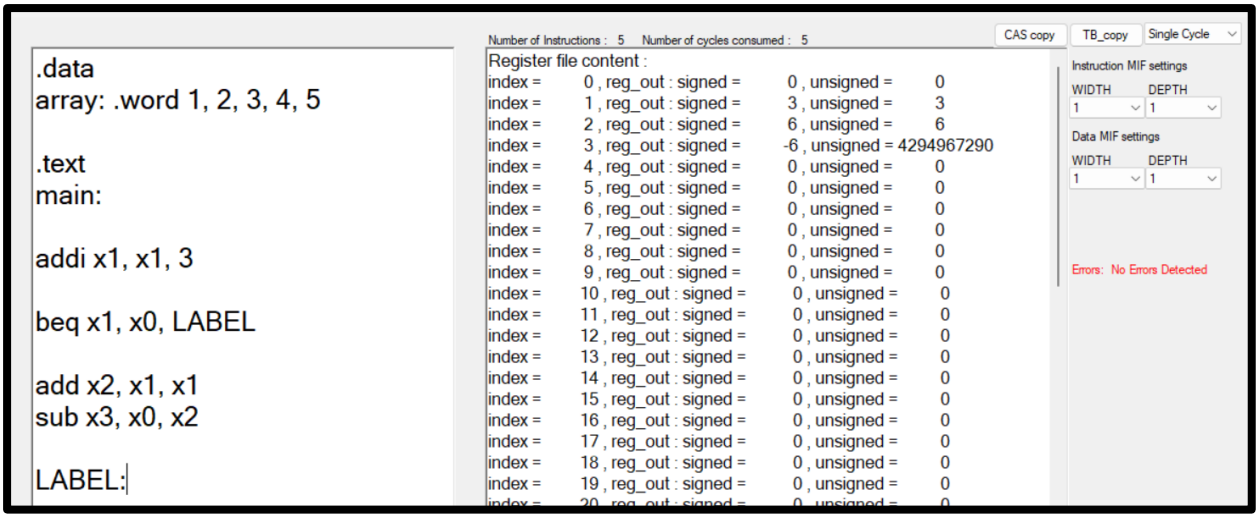


Figure 43 Register values after the execution of a simple program.

The real-time CAS contains many features to help in the process of validation and debugging, including:

- It is compatible with both single cycle and pipelined implementations of the CPU to execute a program. The output of both implementations is the same. However, the number of cycles consumed will differ.
- Similar to the assembler, it displays the time of the error such as wrong instruction or redundant label. However, it also contains the following:
    - The exception indicator which is triggered if an exception was thrown during the execution of the program e.g. invalid memory access.
    - The infinite loop indicator which is triggered to terminate the program if execution time took longer than usual.
- The buttons are for developmental purposes curated to the programmer to facilitate the process of testing different models as much as possible.
    - The CAS copy button copies the machine code of the program written in the text directive. From the previous example if pasted, the resulting output is:

```
"00100000001000010000000000000011", // addi x1 x1 3
"00010000001000000000000000000011", // beq x1 x0 label
"00000000001000010001000000100000", // add x2 x1 x1
"00000000000000010000110000100010", // sub x3 x0 x2
"11111100000000000000000000000000", // hlt
```

    - The Testbench copy (TB_copy) button allows for:
        - Copying the code in a way such that it can be pasted directly into the program memory.

```
InstMem[  0] <= 32'h20210003; // addi x1 x1 3
InstMem[  1] <= 32'h10200003; // beq x1 x0 label
InstMem[  2] <= 32'h00211020; // add x2 x1 x1
InstMem[  3] <= 32'h00021822; // sub x3 x0 x2
InstMem[  4] <= 32'hFC000000; // hlt
```

```
DataMem[0] = 32'd1;
DataMem[1] = 32'd2;
DataMem[2] = 32'd3;
DataMem[3] = 32'd4;
DataMem[4] = 32'd5;
```

- Generating two Memory Initialization Files (MIFs) for the instruction memory (program's machine code for simulation or FPGA prototyping) and for the data memory initializations (initial values before execution).

The importance of MIFs file lies in them being universal and not tied to a specific syntax, they are automatically generated in the same directory as the project's executable file.

The settings of the MIFs require configuration which can be done using the real-time CAS, as shown in the figure.

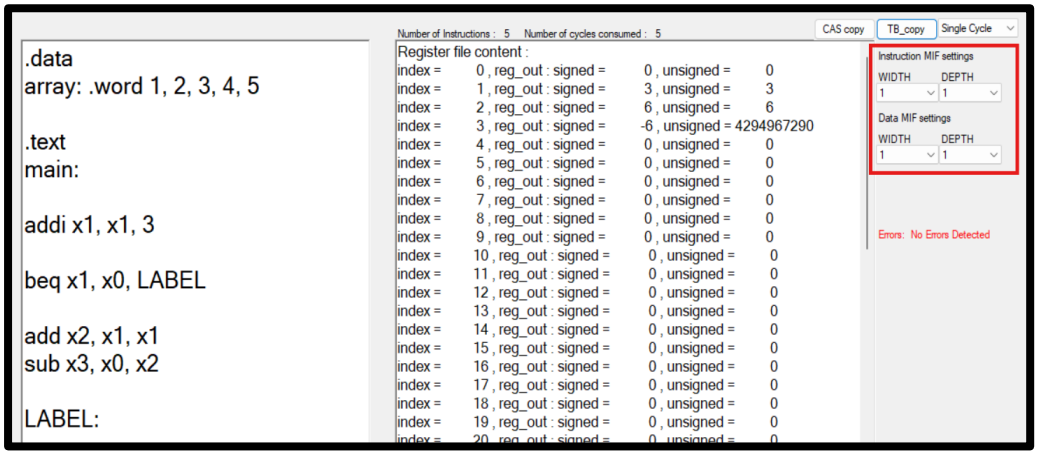


Figure 44: Settings of the Instruction and data MIFs.

5 Evaluation & Results

5.1 JOSDC Committee Benchmarks

To validate the functionality and reliability of the processor design, a structured approach to testbench execution was implemented. This process involved several key steps, automated using a custom Bash script (Run.sh). The script streamlined the workflow, ensuring consistency across all benchmarks, reducing manual effort, and minimizing errors. Each benchmark was organized in its own dedicated folder, containing the program file and placeholders for the output files generated during the testbench process.

During the preprocessing phase, the assembler tool converted the high-level program files into machine code and data memory files. Additionally, the assembler generated various file formats. These included Memory Initialization Files (MIF), which are used for configuring instruction and data memory in hardware simulations, raw machine code files that directly represent the compiled program in binary form, and custom initialization files (INIT), which were specifically designed for inclusion in Verilog modules during hardware simulations. Furthermore, a HALT instruction was added to the end of each benchmark, ensuring proper termination and allowing for accurate cycle count measurements.

The Cycle Accurate Simulator (CAS) was used to perform software simulation of the benchmarks, producing critical outputs such as register values and memory states at the end of execution. These outputs served as a reference point for validating the hardware's functionality. For hardware simulation, Verilog testbenches were used to execute the processor design. These simulations, performed using tools like iverilog, produced waveform files (.vcd) that visually depicted the processor's internal operations, such as control signals and data paths, enabling detailed verification.

An essential feature of the process was the automated file comparison mechanism. This step ensured that the outputs generated by the single-cycle CPU, pipelined CPU, and CAS were consistent. Any discrepancies in the benchmark results were automatically flagged and logged, providing a clear pathway for debugging and validation.

Finally, the overall structure and generated files (MIF, machine code, and INIT) were designed to be highly reusable. The folders containing these outputs can be used across various projects. This adaptability ensures that the workflow is not limited to this specific project but can be applied to a broader range of CPU validation and development tasks, making it a valuable resource for any user.

5.1.1 Data Manipulation Benchmark

```
    .data
value: .word 0x5                    # sample data for loading

    .text
main:
    # Immediate instructions to initialize registers
    ADDI $1, $0, 0xA
    ORI  $2, $0, 0xB
    XORI $3, $0, 0xC

    # Basic ALU operations
    ADD  $4, $1, $2
    SUB  $5, $4, $3
    AND  $6, $1, $3
    OR   $7, $2, $3
    NOR  $8, $2, $3
    XOR  $9, $1, $2

    # Comparison operations
    SLT  $10, $1, $2
    SGT  $11, $3, $1

    # Shift operations
    SLL  $12, $1, 2
    SRL  $13, $2, 1

    # Load and store word instructions
    LW   $14, 0x0($0)
    SW   $4, 0x0($0)
```

Figure 45: shows the MIPS assembly code for the Data Manipulation benchmark.

The benchmark above is a simple benchmark to test data manipulation instructions without branch or jump instructions. It has a minimal number of dependencies to test the general functionality of each instruction. Notice that this is the benchmark without the HALT instruction that was previously mentioned. Even though it is not found in **Figure 1**, its automatically added during the automated process of simulation.

Through the process explained before, the machine code for the instruction set in **Figure 1** is shown below:

```
00100000000000010000000000001010
00110100000000010000000000001011
00111000000000011000000000001100
00000000001000100010000000100000
00000000100000110010100000100010
00000000001000110011000000100100
00000000100000110011100000100101
00000000100000111010000000100111
00000000001000100100100000100110
00000000001000100101000000101010
00000000011000010101100000101011
00000000000000001011000010000000
00000000000000010011010000100010
10001100000011100000000000000000
10101100000001000000000000000000
11111100000000000000000000000000
```

Figure 46: shows the machine code for Data Manipulation Benchmark instruction set.

To make sure that both the single cycle and pipelined processor are correct, the waveform files for both processors are going to be analyzed below.

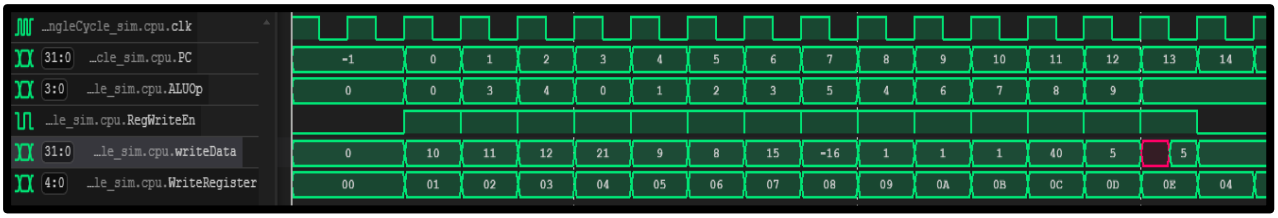


Figure 47: Waveform diagram for the single-cycle CPU during the Data Manipulation benchmark.

The waveform illustrates the execution of instructions in the single-cycle CPU, with key signals such as the clock (clk), program counter (PC), ALU operation (ALUOp), and register-related signals (RegWriteEn, WriteData, WriteRegister) displayed for verification.

The PC increments sequentially throughout the waveform, reflecting the CPU's proper instruction fetching process. Each increment of the PC corresponds to the address of the next instruction in the instruction memory. This confirms that the single-cycle CPU is progressing through the instructions without delays or jumps, as no branch or jump instructions are executed in this specific testbench. The correct sequencing of the PC values verifies that the fetch stage is functioning as expected.

The "ALUOp" signal changes to match the operation specified by the current instruction. For example, the waveform indicates operations such as addition, subtraction, and logical AND/OR, depending on the instruction being executed. For the first instruction, "ALUOp" was set to 4'b0000 indicting that the ALU was used for addition which aligns with the first instruction. This behavior demonstrates that the control signals governing the ALU are being generated correctly by the control unit.

The "RegWriteEn" signal is high during cycles where the CPU writes back the result of an operation to the register file. "RegWriteEn" is only zero for the last instruction which is a correct as it is a store instruction. Alongside this, the "WriteRegister" and "WriteData" signals indicate which register is being updated and the value being written, respectively. The observed behavior shows that the correct destination registers are selected, and the expected values are written. This confirms that the write-back stage operates as intended, ensuring that the results of ALU operations are stored for subsequent instructions.

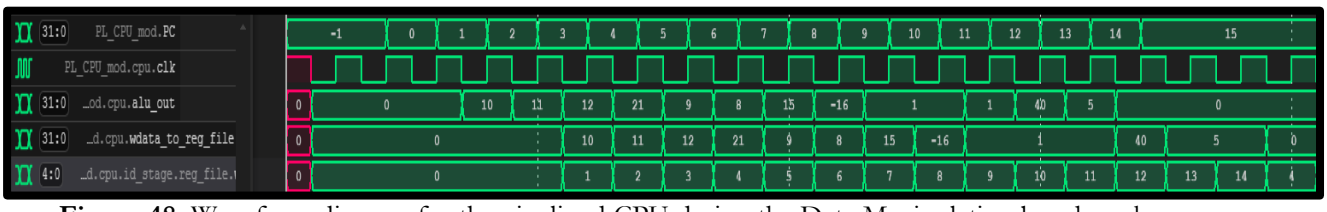


Figure 48: Waveform diagram for the pipelined CPU during the Data Manipulation benchmark.

The waveform for the pipelined CPU illustrates the execution of the Data Manipulation benchmark, emphasizing the parallel processing enabled by the pipelined architecture. Unlike the single-cycle CPU, the pipelined CPU overlaps the execution of multiple instructions across its stages, improving throughput.

Similarly to the single-cycle CPU, the pipeline CPU is progressing through the instructions without delays or jumps, as no branch or jump instructions are executed in this specific testbench. The correct sequencing of the PC values verifies that the fetch stage is functioning as expected.

The “alu_out” signal reflects the results of arithmetic or logical operations performed during the execution stage. The observed waveform corresponds accurately to the expected results of instructions such as addition, subtraction, or logical AND/OR operations, confirming that the execution stage is functioning correctly and producing valid outputs.

The “wdata_to_reg_file” signal indicates the data being written back to the register file during the write-back stage. This data matches the expected results from previous ALU operations or memory load instructions, confirming the accuracy of the write-back stage. Additionally, the “wr_reg” signal identifies the destination register being updated. The consistency between this signal and the expected instruction sequence ensures that the pipeline correctly targets and updates the appropriate registers.

The overlapping nature of instruction execution in the pipeline is evident in the waveform. It demonstrates that the stages—fetch, decode, execute, memory access, and write-back—operate in parallel, maintaining correct timing and avoiding data hazards. The signals indicate effective communication between pipeline stages, with results from earlier instructions being forwarded correctly when necessary.

```
Register file content :
index =          0 , reg_out : signed =           0 , unsigned =          0
index =          1 , reg_out : signed =          10 , unsigned =         10
index =          2 , reg_out : signed =          11 , unsigned =         11
index =          3 , reg_out : signed =          12 , unsigned =         12
index =          4 , reg_out : signed =          21 , unsigned =         21
index =          5 , reg_out : signed =           9 , unsigned =          9
index =          6 , reg_out : signed =           8 , unsigned =          8
index =          7 , reg_out : signed =          15 , unsigned =         15
index =          8 , reg_out : signed =         -16 , unsigned = 4294967280
index =          9 , reg_out : signed =           1 , unsigned =          1
index =         10 , reg_out : signed =           1 , unsigned =          1
index =         11 , reg_out : signed =           1 , unsigned =          1
index =         12 , reg_out : signed =          40 , unsigned =         40
index =         13 , reg_out : signed =           5 , unsigned =          5
index =         14 , reg_out : signed =           5 , unsigned =          5
```

Figure 49: Register file content for all CPU designs after the Data Manipulation benchmark.

```
Data Memory Content :
Mem[ 0] =          21
Mem[ 1] =           x
Mem[ 2] =           x
```

Figure 50: Data Memory content for all CPU designs after the Data Manipulation benchmark.

The register file and data memory content obtained from the Cycle Accurate Simulator, Verilog simulations, and for both the single-cycle and pipelined CPUs are identical. This consistency across all implementations confirms the correctness of the CPU designs and ensures that both architectures adhere to the expected instruction execution flow.

The results align perfectly with the expected outcomes derived from the program's instructions. For instance, register $1 holds the value 10, as initialized by the immediate instruction (ADDI $1, $0, 0xA). Similarly, registers $2 and $3 hold values 11 and 12, respectively, matching the ORI and XORI instructions.

Arithmetic and logical operations were validated successfully, with register $4 containing 21, confirming the addition of $1 and $2, and register $5 holding 9, verifying the subtraction of $3 from $4. Logical operations such as AND, OR, and XOR are also correctly reflected in registers $6, $7, and $8.

Comparison and shift operations demonstrate the CPU's ability to handle these instructions accurately. Registers $10 and $11 hold the value 1, representing true conditions for SLT and SGT instructions. Similarly, shift operations (SLL, SRL, SRA) produced expected values in registers $12, $13, and $14.

Memory operations are validated by the accurate execution of load and store instructions. For example, the load word instruction correctly loaded the value 21 from Mem[0] into register $14, and the store word instruction stored the same value from $14 back into Mem[0], as reflected in the memory content.

These results collectively validate the accuracy and correctness of both CPU implementations for the Data Manipulation benchmark.

5.1.2 Control Flow Benchmark

```
        .text
main:
                # Initialize registers with immediate values
                ADDI $1, $0, -5
                ADDI $2, $0, 5
                ADDI $3, $0, 5
                ADDI $4, $0, 21

L1:
                #  Test BEQ
                BEQ $2, $3, L2
                NOP             # No operation / SLL $0, $0, 0
                JAL L1
L2:
                # Test BNE
                BNE $2, $3, L3
L3:
                NOP
L4:
                # Test BLTZ
                BLTZ $1, L5
                NOP
                JAL L4
L5:
                #  Test BGEZ
                BGEZ $2, L6
                NOP
                JAL L5
L6:
                #  Test JAL
                JAL L7
                NOP
                JAL L6
L7:
                #  Test JR
                JR $4
                NOP
                JAL L7
L8:
                NOP             # end of test cases
```

Figure 51: shows the MIPS assembly code for the Control Flow benchmark.

This is a simple benchmark to test control flow instructions ('BEQ', 'BNE', 'BLTZ', 'BGEZ', 'JAL', 'JR') with minimal dependencies to test the functionality of each control flow instruction.

It is important to know that in this benchmark, the immediate value in the instruction (ADDI $4, $0, 84) was modified from 84 to 21. This adjustment was necessary because the CPU being tested uses a word-

addressable instruction memory rather than a byte-addressable one, as the original benchmark assumes. Therefore, the instruction in **Figure 8** is written as (ADDI $4, $0, 21).

Figure 52: Waveform diagram for the single-cycle CPU during the Control Flow benchmark.

The waveforms captured from the single-cycle CPU illustrate the execution of branch instructions, with

critical signals such as the PC, branch controller operands (operand1 and operand2), and the branch decision signal "PCsrc" displayed to verify the correctness of the branch logic. These signals provide insight into how the CPU evaluates branch conditions and determines whether to update the PC or proceed sequentially.

For branch instructions such as BEQ (branch if equal) and BNE (branch if not equal), the waveforms show that operand1 and operand2 are compared. When the condition is satisfied, the PCsrc is set to 1, and PC is updated to the branch target address. For example, at the fifth instruction (PC=4) after the ADDI instructions used for initialization have been performed, a "BEQ" instruction is executed with "operand1" and "operand2" both equal to 5. The branch controller sets "PCsrc" to 2'b01, causing the program counter to jump to the branch target address, confirming the proper functioning of the branch logic.

On the other hand, for instructions where the branch condition is not met, the "PCsrc" signal remains at 2'b00, and the PC increments sequentially. The instruction due next after performing the first BEQ branch is at PC = 7. The BNE equal instruction compares the $2 and $3 and branches to the target address if they are not equal. It can be seen that at PC=7, the branch controller operands are the contents of $2 and $3 and they are equal. As a result, the "PCsrc" signal is equal to 2'b00 since they are not equal and the PC increments sequentially.

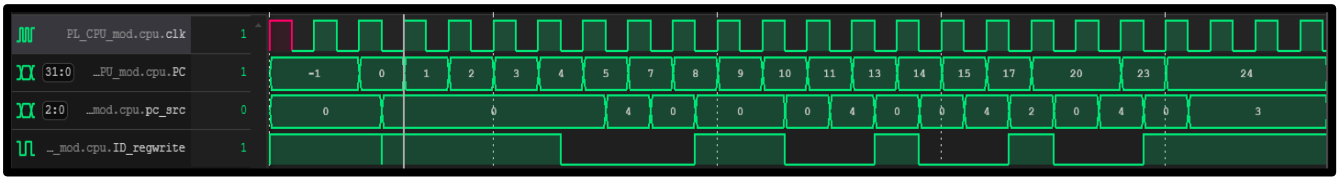
Other branch instructions such as BLTZ and BGEZ were also tested. The waveforms validate that the branch controller correctly interprets signed comparisons. For example, at the instruction PC = 8, during a BLTZ instruction, "operand1" interprets $1 and has a value of -5, satisfying the branch condition. The branch controller sets "PCsrc" to 2'b01, leading to a program counter update to the specified branch target. After branching to the target address, the BGEZ instruction is due next at PC = 12. The value stored at $2, equal to 5, satisfies the condition for the branch as it is greater than zero. Therefore, the "PCsrc" is set to 2'b01 and the branch is performed.

At PC=15, the "JAL" instruction is performed correctly as the "PCsrc " is set to 2'b01 and the PC due after equals to 18 which is the correct address. At PC=18, The "JR" instruction branches to the address stored in $4 equal to 21. This is verified as the PC value updates to 21 after the "JR" instruction as seen in **Figure 51**.

These observations confirm that the branch logic in the single-cycle CPU is functioning as intended. The waveforms demonstrate that the CPU correctly evaluates branch conditions, updates the program counter

when necessary, and continues sequentially otherwise. This alignment between the expected and observed behavior validates the implementation of the branch unit in the single-cycle CPU.

Figure 53: Waveform diagram for the pipelined CPU during the Control Flow benchmark.



The provided waveform demonstrates the execution of branch instructions in the pipelined CPU, with the "pc_src" signal playing a key role in managing program counter updates during control flow instructions. At PC=4, a branch instruction (e.g., BEQ) is executed successfully, as the "pc_src" signal reflects the correct decision to update the program counter to the branch target address. This confirms that the pipeline resolves the branch condition accurately and redirects execution flow appropriately.

In scenarios where the branch condition is not satisfied, such as the BNE instruction at PC=7, the "pc_src" signal remains consistent, ensuring sequential execution without interruptions. This behavior demonstrates that the pipeline correctly bypasses unnecessary branch actions while maintaining program flow.

The waveform also captures jump instructions such as JAL and JR. At PC=15, the JAL instruction correctly updates the program counter to the target address, as reflected by the "pc_src" signal. Similarly, during the JR instruction at PC=18, the program counter updates to the value stored in register $4 (PC=21), validating the pipeline's ability to handle control transfer instructions accurately.

Overall, the waveform confirms the pipelined CPU's control flow mechanisms function as intended, with proper program counter updates and seamless instruction execution.

```
Register file content :
index =          0 , reg_out : signed =           0 , unsigned =          0
index =          1 , reg_out : signed =          -5 , unsigned = 4294967291
index =          2 , reg_out : signed =           5 , unsigned =          5
index =          3 , reg_out : signed =           5 , unsigned =          5
index =          4 , reg_out : signed =          21 , unsigned =         21
```

Figure 54: Register file content for all CPU designs after the Control Flow benchmark.

The register file content shown in **Figure 53** demonstrates the correctness of the executed instructions for both the single-cycle and pipelined CPUs. The outputs, generated either through the Cycle Accurate Simulator (CAS) or Verilog simulations, are identical, affirming the consistency and correctness of the CPUs' implementation across both architectures.

5.1.3 Sum of Numbers Benchmark

```
    .text
main:
    # Initialize registers
    ORI $2, $0, 0x1
    ORI $3, $0, 0x0
    ORI $4, $0, 0xA

SUM_LOOP:
    ADD  $3, $3, $2
    ADDI $2, $2, 1
    SLT  $5, $4, $2
    BEQ  $5, $0, SUM_LOOP

    SW   $3, 0x0($0)

END:
    NOP                         # End program
```

Figure 55: MIPS assembly code for the Sum of Numbers benchmark.

This instruction set implements a program to calculate the sum of integers from 1 to 10 using a loop. It initializes key registers to store the current number being added, the cumulative sum, and the upper limit for the summation. The program performs iterative addition and stores the result in memory once the summation is complete.

In the initialization section, the program sets up the required values. Register $2 is initialized to 1, representing the current number to be added. Register $3 is set to 0 to accumulate the sum, and register $4 is set to 10 (in hexadecimal 0xA), which serves as the upper limit for the summation process. These initializations prepare the CPU for the subsequent loop operations.

The main computation occurs in the summation loop, labeled SUM_LOOP. During each iteration, the program adds the value in $2 to the cumulative sum stored in $3 using the ADD instruction. Then, $2 is incremented by 1 via the ADDI instruction, advancing to the next number in the sequence. A comparison is made using the SLT instruction to check if $2 has exceeded the upper limit stored in $4. If $2 is still within the range, the BEQ instruction branches back to the start of the SUM_LOOP, continuing the iterative addition process.

Once the loop completes, the program proceeds to store the final result. The cumulative sum, held in $3, is written to memory address 0x0 using the SW instruction. This marks the conclusion of the computation. Finally, the program halts with a NOP instruction at the labeled END, signifying no further operations are to be executed.

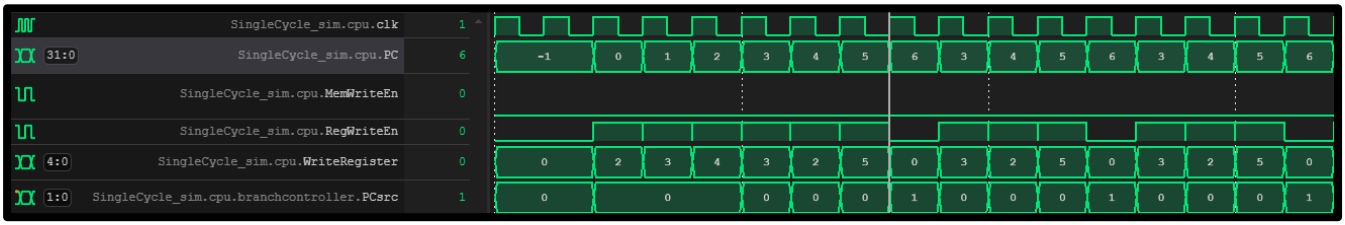


Figure 56: Waveform diagram for the single-cycle CPU during the Sum of Numbers benchmark (1).

Figure 57: Waveform diagram for the single-cycle CPU during the Sum of Numbers benchmark (2).

For clarity, only a portion of the program execution is shown—specifically, the initialization and the first two loop iterations in **Figure 55**, and the final two loop iterations leading to the program's conclusion in **Figure 56**. The waveforms accurately depict the execution of the program on the single-cycle CPU. They show the successful branching at PC = 6, where the program evaluates the branch condition for the BEQ instruction. For 10 iterations, the branch condition is satisfied. This directs the program counter to return to PC = 3, ensuring the loop executes as expected.

The "RegWriteEn" signal is asserted at the correct instructions, aligning with operations that write results to the register file, such as arithmetic and immediate operations. Additionally, the destination registers indicated by the "WriteRegister" signal are accurate, confirming that the write-back stage correctly updates the intended registers with the expected results.

The "MemWriteEn" signal is asserted exclusively during the SW instruction, which stores data from a register into memory. This behavior demonstrates that memory operations are performed accurately and only when required by the program, further validating the correctness of the CPU's implementation.

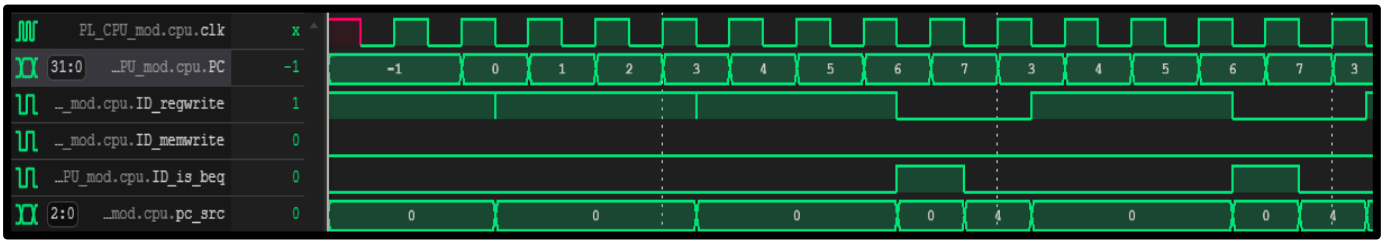


Figure 58: Waveform diagram for the pipelined CPU during the Sum of Numbers benchmark (1).

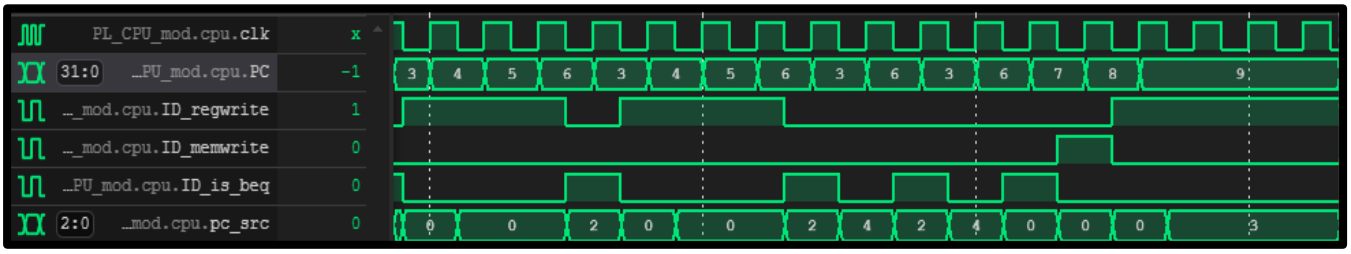


Figure 59: Waveform diagram for the pipelined CPU during the Sum of Numbers benchmark (2).

Notice that the signals "ID_regwrite," "ID_memwrite," and "ID_is_beq" are all captured at the instruction decode stage. This approach is maintained across all benchmarks to avoid the complexity and reduced readability that would arise from analyzing signals at multiple stages of the pipeline. By focusing on a single stage, the interpretation of results becomes more straightforward and efficient.

The PC correctly loops back to instruction PC=3 for 10 iterations due to the branch condition being met at PC=6. The PC correctly increments at the 10th iteration and does not branch and perform the instruction. The "ID_regwrite" signal is asserted at the appropriate instructions, confirming correct register updates, while "ID_memwrite" is only high during the SW instruction, verifying accurate memory operations. These waveforms validate the pipeline CPU's behavior for both loop iterations and program completion. In addition, "ID_is_beq" shows that the branch instructions taken are due to the BEQ instruction. It can be noticed that at branch instructions, that "ID_is_beq" is asserted showing that the operand comparisons being made are correct.

```
Register file content :
index =          0 , reg_out : signed =           0 , unsigned =           0
index =          1 , reg_out : signed =           0 , unsigned =           0
index =          2 , reg_out : signed =          11 , unsigned =          11
index =          3 , reg_out : signed =          55 , unsigned =          55
index =          4 , reg_out : signed =          10 , unsigned =          10
index =          5 , reg_out : signed =           1 , unsigned =           1
```

Figure 60: Register file content for all CPU designs after the Sum of Numbers benchmark.

```
Data Memory Content :
Mem[ 0] =          55
Mem[ 1] =           x
Mem[ 2] =           x
```

Figure 61: Data Memory content for all CPU designs after the Sum of Numbers benchmark.

The register values reflect the correct outputs of arithmetic and immediate instructions, while the data memory content confirms that the SW instruction successfully stored the value 55 in Mem[0]. This accuracy verifies the correct functionality of the CPU's instruction execution and memory operations.

5.1.4 Binary Search Benchmark

```
    .data
myArray:    .word 1, 4, 5, 7, 9, 12, 15, 17, 20, 21, 30
arraySize: .word 11

    .text
#Initailiztion
addi $1, $0, 0x0
addi $2, $0, 0xB
addi $3, $0, 0x7

loop:
    slt $7, $2, $1
    bne $0, $7, notFound
    add $4, $2, $1
    srl $5, $4, 1

    #sll $5, $5, 2 For byte addressable memory

    lw  $6, 0x0($5)

    #srl $5, $5, 2 For byte addressable memory

    beq $3, $6, found
    slt $6, $6, $3
    beq $6, $0, leftHalf
    j   rightHalf

leftHalf:
    add $2, $5, 0xFFFF      # "FFFF=-1"
    j   loop

rightHalf:
    addi $1, $5, 0x1
    j    loop

found:
    add $8, $0, $5
    j finish

notFound:
    addi $8, $0, 0xFFFF
    j finish

finish:
    NOP
```

Figure 62: MIPS assembly code for the Binary Search benchmark.

The binary search benchmark tests the CPU's ability to execute the binary search algorithm on a sorted array stored in memory. It efficiently narrows the search range by adjusting pointers for the left and right halves of the array until the target value is found or determined to be absent. The result, either the index of the target or a "not found" indicator, is stored in a register. This benchmark validates the CPU's handling of arithmetic operations, memory access, branching, and control flow.

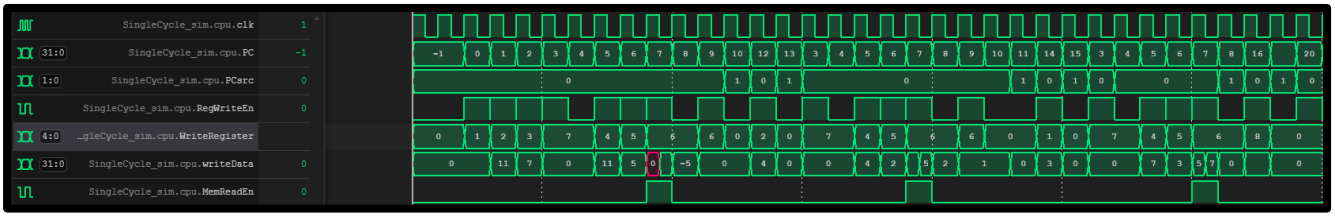


Figure 63: Waveform diagram for the single-cycle CPU during the Binary Search benchmark.

The waveform confirms the correct execution of the binary search benchmark on the single-cycle CPU by analyzing key control and data signals. The “PCsrc” signal correctly reflects the branching behavior, ensuring that the PC updates to the appropriate addresses during branch instructions.

The “RegWriteEn” signal is asserted during the correct cycles, indicating that the register file is updated only when necessary. The “writeData” signal confirms that the data generated by the CPU during

operations or loaded from memory matches the expected results. The observed values for "WriteRegister" confirm that the correct registers are being updated as required by the instructions in the benchmark.

The "MemReadEn" signal is asserted during memory load (lw) instructions, verifying that the CPU accesses memory only when required. The alignment of the MemReadEn signal with the corresponding PC values confirms that memory read operations are performed at the correct points in the program execution.

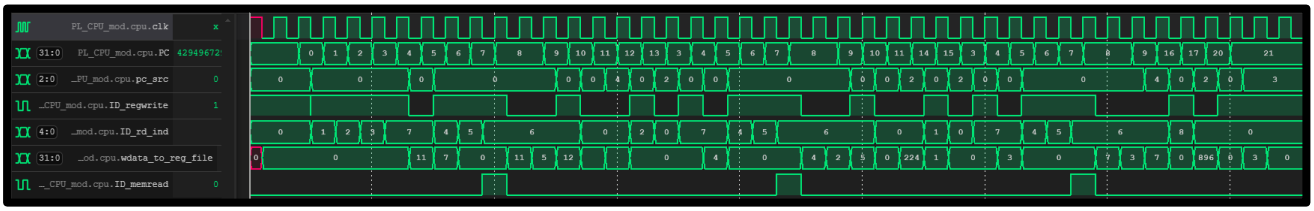


Figure 64: Waveform diagram for the pipelined CPU during the Binary Search benchmark.

The PC signal accurately tracks the execution flow of the program, showing both sequential increments and branching events. The "pc_src" signal reflects the branching behavior of the pipeline, ensuring that the program counter is updated correctly for control flow changes, such as jumps and branches.

The "ID_regwrite" signal is asserted during cycles where write-back to the register file is necessary. Its alignment with specific PC values demonstrates that the pipeline updates the register file only when required, adhering to the benchmark's execution flow. The "ID_rd_ind" signal specifies the destination register being updated, which matches the expected behavior for instructions writing data to registers.

The "wdata_to_reg_file" signal provides the data being written back to the register file. The values align with the results of prior operations or memory load instructions, verifying that the data being stored is accurate and consistent with the program's requirements.

The "ID_memread" signal is asserted during memory load (lw) instructions, confirming that memory read operations are performed at the appropriate points in the program. This ensures that the pipeline accesses memory only when necessary, maintaining efficiency and correctness.

```
Register file content :
index =          0 , reg_out : signed =          0 , unsigned =          0
index =          1 , reg_out : signed =          3 , unsigned =          3
index =          2 , reg_out : signed =          4 , unsigned =          4
index =          3 , reg_out : signed =          7 , unsigned =          7
index =          4 , reg_out : signed =          7 , unsigned =          7
index =          5 , reg_out : signed =          3 , unsigned =          3
index =          6 , reg_out : signed =          7 , unsigned =          7
index =          7 , reg_out : signed =          0 , unsigned =          0
index =          8 , reg_out : signed =          3 , unsigned =          3
```

Figure 65: Register File content for all CPU designs after the Binary Search benchmark.

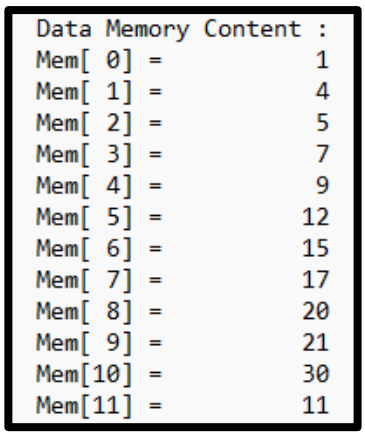


Figure 66: Data Memory content for all CPU designs after the Binary Search benchmark.

The initialization registers, such as $1 and $2, were correctly loaded with their respective starting values as per the program instructions. Register $1 was initialized to 1, representing the starting index of the array, while register $2 was initialized to 11, representing the ending index.

As the binary search algorithm progressed, the register values were updated to reflect intermediate computations, including the calculation of the mid-point index and adjustments to the search range. These computations were carried out correctly, with registers holding the intermediate results necessary for the binary search logic.

Ultimately, the target value 7, which we are searching for in the array, is correctly loaded into register $6. This demonstrates that the CPU accurately performed the necessary arithmetic, memory access, and branching operations to locate the target. This result validates the implementation of the benchmark, confirming that the registers are updated as expected at each stage of the binary search for both the pipelined and single-cycle CPUs through the software and hardware implementations.

5.1.5 Max and Min in Array Benchmark

```
    .data
Array: .word 0x10, 0xF, 0x5, 0x9, 0x20, 0x19, 0x4, 0x1E, 0x9, 0xB

    .text
main:
    # Initialize registers
    ORI  $2 , $0, 0x0
    ADDI $20, $0, 0xA
    XORI $31, $0, 0x1
    ANDI $5 , $0, 0x0
    LW   $10, 0x0($5)
    LW   $15, 0x0($5)

LOOP:
    ADDI $2,  $2, 1
    SGT  $25, $20, $2
    BNE  $25, $31, END

    # Choose one of these Insertion based on your memory
    # For Word addressable        # For byte addressable
    # ADD  $5, $2, $0             # SLL  $5, $2, 2

    LW   $16, 0x0($5)
    SGT  $26, $16, $10
    BEQ  $26, $0, MIN
    OR   $10, $16, $0
    J    LOOP

MIN:
    SLT  $27, $16, $15
    BEQ  $27, $0, LOOP
    ADD  $15, $16, $0
    J    LOOP

END:
    NOP # (NOP equals to SLL $0, $0, 0)
```

Figure 67: MIPS assembly code for the Max and Min in Array benchmark.

The Max and Min in Array benchmark aims to identify the maximum and minimum values in a given array stored in memory. The program initializes registers with starting values for comparison, then iterates through the array, loading each element into a register, and updating the maximum and minimum values as needed. Once the loop completes, the final maximum and minimum values are stored in designated registers. Since the array is stored in byte-addressable memory, the instruction SLL $5, $2, 2 is used for correct indexing.

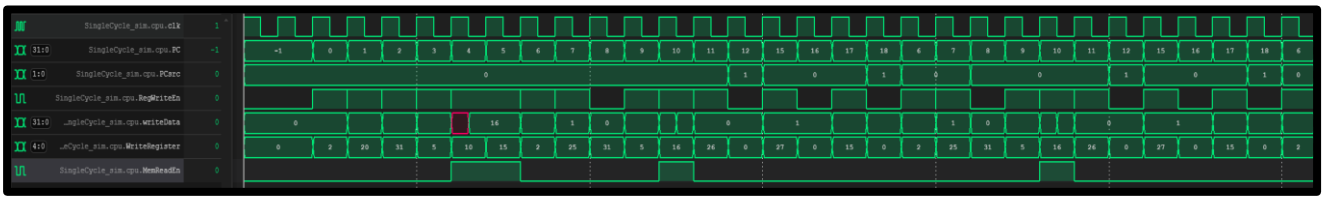


Figure 68: Waveform diagram for the single-cycle CPU during the Max and Min in Array benchmark(1).

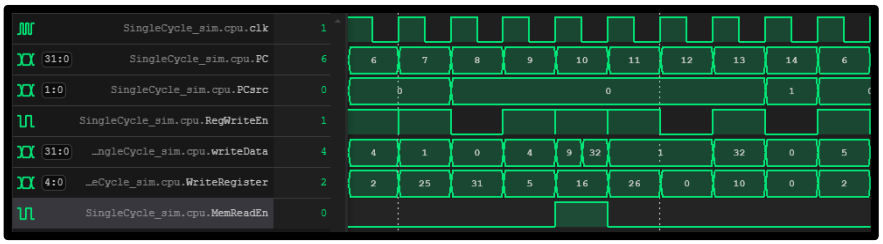


Figure 69: Waveform diagram for the single-cycle CPU during the Max and Min in Array benchmark(2).

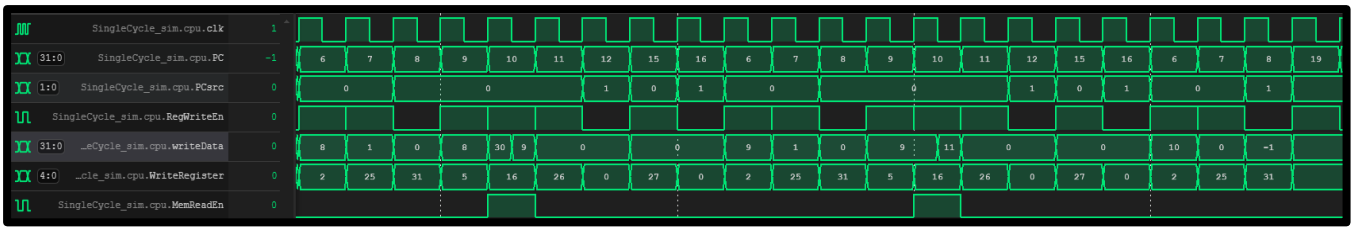


Figure 70: Waveform diagram for the single-cycle CPU during the Max and Min in Array benchmark(3).

Figure 67 shows the waveform signals during the first two loop iterations of the benchmark. **Figure 68** shows the 4th loop iteration of the benchmark. **Figure 69** shows the last 3 loop iterations and end of the benchmark.

The waveforms confirm that branching for all branch instructions was executed correctly and as expected. The PC updates appropriately, reflecting the intended control flow dictated by the branch conditions. Branches at instructions where PC = 14, PC = 16, and PC = 18 are all performed correctly under the required conditions, looping back to PC = 6. The "MemReadEn" signal is enabled only during l(w) instructions, demonstrating correct memory read behavior. Additionally, the "writeData" and "WriteRegister" signals verify that the correct data is being written back to the designated registers at their respective addresses, with the process accurately controlled by the "RegWriteEn" signal.

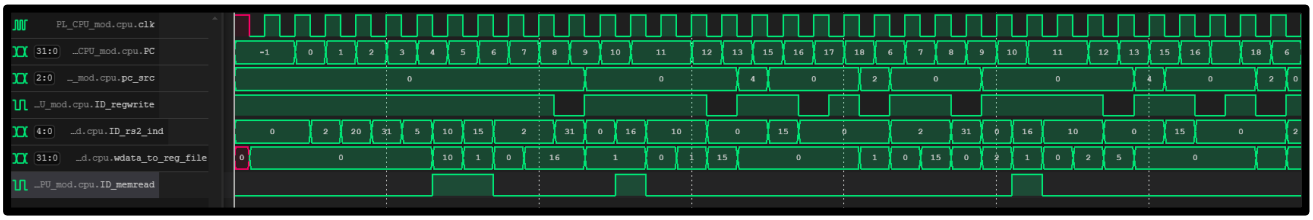


Figure 71: Waveform diagram for the pipelined CPU during the Max and Min in Array benchmark (1).

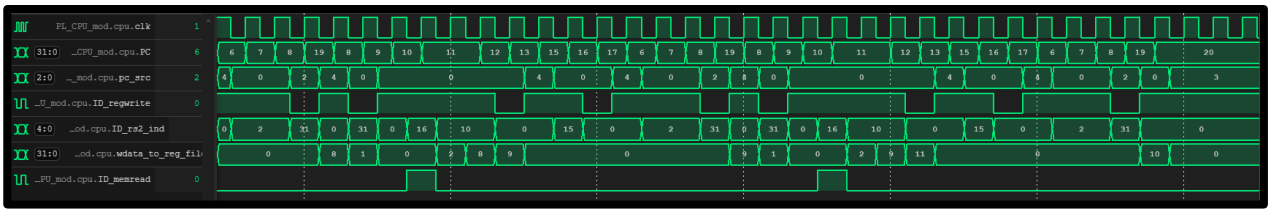


Figure 72: Waveform diagram for the pipelined CPU during the Max and Min in Array benchmark (2).

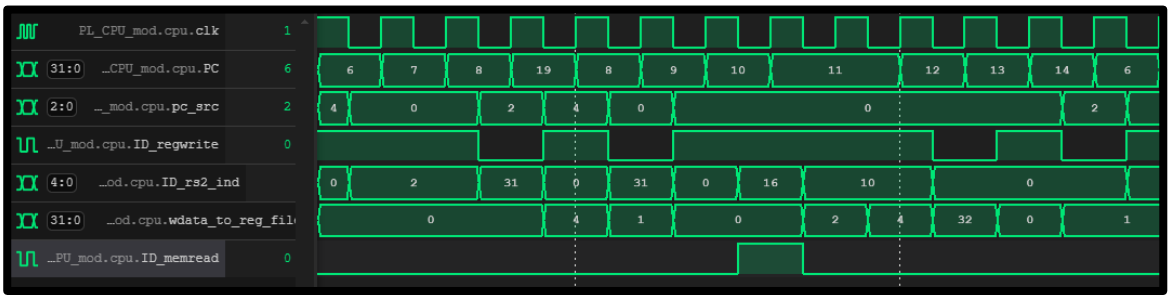


Figure 73: Waveform diagram for the pipelined CPU during the Max and Min in Array benchmark (3).

In a similar way to the single-cycle CPU, The PC signal accurately tracks the execution flow of the program, showing both sequential increments and branching events. The "pc_src" signal reflects the branching behavior of the pipeline, ensuring that the program counter is updated correctly for control flow changes, such as jumps and branches. The "ID_regwrite" signal is asserted during cycles where write-back to the register file at specific PC values demonstrating that the pipeline updates the register file only when required. The "ID_rd_ind" signal indicates the correct destination registers during execution. The "ID_memread" signal is asserted during memory load (lw) instructions, confirming that memory read operations are performed at the appropriate points in the program.

It is important to note that the branch predictor module in the pipelined CPU may occasionally make incorrect predictions. For instance, in Figure 32, when PC=8, the subsequent instruction executed is at PC=19, indicating a wrong prediction. However, the CPU handles this incorrect prediction efficiently, and the PC is successfully restored to 8, ensuring that the prog ram flow remains correct despite the misprediction.

```
Register file content :
index =          0 , reg_out : signed =           0 , unsigned =          0
index =          1 , reg_out : signed =           0 , unsigned =          0
index =          2 , reg_out : signed =          10 , unsigned =         10
index =          3 , reg_out : signed =           0 , unsigned =          0
index =          4 , reg_out : signed =           0 , unsigned =          0
index =          5 , reg_out : signed =           9 , unsigned =          9
index =          6 , reg_out : signed =           0 , unsigned =          0
index =          7 , reg_out : signed =           0 , unsigned =          0
index =          8 , reg_out : signed =           0 , unsigned =          0
index =          9 , reg_out : signed =           0 , unsigned =          0
index =         10 , reg_out : signed =          32 , unsigned =         32
index =         11 , reg_out : signed =           0 , unsigned =          0
index =         12 , reg_out : signed =           0 , unsigned =          0
index =         13 , reg_out : signed =           0 , unsigned =          0
index =         14 , reg_out : signed =           0 , unsigned =          0
index =         15 , reg_out : signed =           4 , unsigned =          4
index =         16 , reg_out : signed =          11 , unsigned =         11
index =         17 , reg_out : signed =           0 , unsigned =          0
index =         18 , reg_out : signed =           0 , unsigned =          0
index =         19 , reg_out : signed =           0 , unsigned =          0
index =         20 , reg_out : signed =          10 , unsigned =         10
index =         21 , reg_out : signed =           0 , unsigned =          0
index =         22 , reg_out : signed =           0 , unsigned =          0
index =         23 , reg_out : signed =           0 , unsigned =          0
index =         24 , reg_out : signed =           0 , unsigned =          0
index =         25 , reg_out : signed =           0 , unsigned =          0
index =         26 , reg_out : signed =           0 , unsigned =          0
index =         27 , reg_out : signed =           0 , unsigned =          0
index =         28 , reg_out : signed =           0 , unsigned =          0
index =         29 , reg_out : signed =           0 , unsigned =          0
index =         30 , reg_out : signed =           0 , unsigned =          0
index =         31 , reg_out : signed =           1 , unsigned =          1
```

Figure 74: Register File content for all CPU designs after the Max and Min in Array benchmark.

```
Data Memory Content :
Mem[ 0] =          16
Mem[ 1] =          15
Mem[ 2] =           5
Mem[ 3] =           9
Mem[ 4] =          32
Mem[ 5] =          25
Mem[ 6] =           4
Mem[ 7] =          30
Mem[ 8] =           9
Mem[ 9] =          11
```

Figure 75: Data Memory content for all CPU designs after the Max and Min in Array benchmark.

The data memory content shows the array contents that have been provided in the benchmark. The maximum and minimum values are 32 and 4 respectively. From the register contents in **Figure 34**, it can be noticed that register initializations have been performed correctly. Moreover, the maximum and minimum values in the array should be stored at registers $10 and $15 respectively. The benchmark has been successfully performed by software and hardware versions of the pipelined and single-cycle CPUs as the maximum value 32 is stored at $10 and the minimum value 4 is stored at $15.

5.1.6 Insertion Sort Benchmark

```
        .data
Arr: .word 0x5, 0x7, 0x2, 0xF, 0xA, 0x10, 0x30, 0x1, 0xFF, 0x55

        .text

main:
         $20, $0, $0
        addi $1, $0, 2
        addi $22, $0, -1

FOR_LOOP:

        sub $31, $1, $20
        bgez $31, EXIT
        lw   $8, 0x0($1)
        addi $2, $1, -1

WHILE_LOOP:

        sgt  $3, $2, $22
        beq  $3, $0, EXIT_WHILE

        lw   $5, 0x0($2)
        sgt  $4, $5, $8
        and  $7, $3, $4
        beq  $7, $0, EXIT_WHILE

        addi $6, $2, 1
        sw   $5, 0x0($6)
        addi $2, $2, -1
        j    WHILE_LOOP

EXIT_WHILE:

        addi $2, $2, 1
        sw   $8, 0x0($2)
        addi $1, $1, 1

        j    FOR_LOOP

EXIT:
```

Figure 76: MIPS assembly code for the Insertion Sort Benchmark.

NOTE: the committee insertion sort didn't work as intended so here we refer to it as the tested program but we wrote our version of insertion sort to test the CPUs on such benchmark. This program implements a sorting mechanism to rearrange elements in an array stored in memory. It iterates through the array using nested loops, with the outer "FOR_LOOP" traversing each element and the inner "WHILE_LOOP" comparing and swapping elements based on specific conditions.

The "FOR_LOOP" ensures all elements are processed by iterating over the array. The inner "WHILE_LOOP" compares the current array element with previously accessed elements to find and maintain the correct order. When a condition is satisfied, elements are swapped using the load and store word instructions.

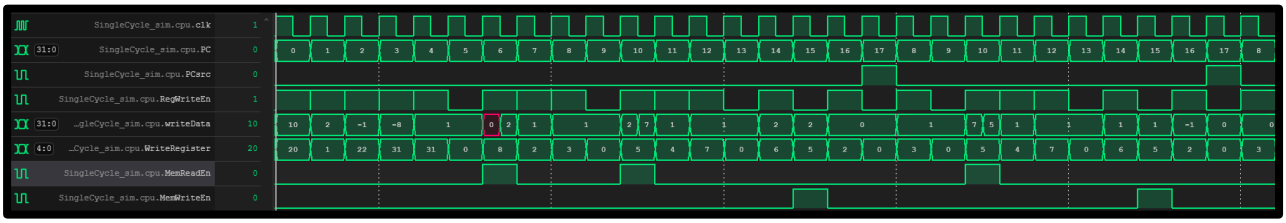


Figure 77: Waveform diagram for the single-cycle CPU during the Insertion Sort benchmark.

The PC updates sequentially and correctly during the first two iterations of the sorting algorithm. Branching is performed accurately when conditions require a jump, as evidenced by the values of the "PCsrc" signal.

The "RegWriteEn" signal is asserted during instructions that update registers, with corresponding "WriteRegister" and "WriteData" signals showing the expected data being written to the correct registers. Similarly, the "MemReadEn" and "MemWriteEn" signals are correctly enabled for lw and sw instructions, confirming proper memory access during the sorting process.

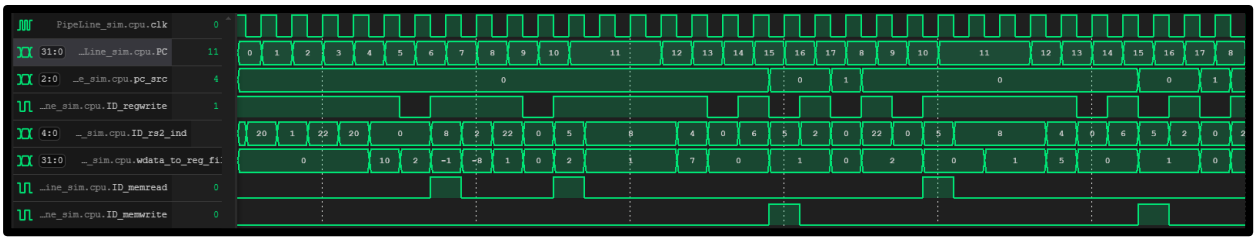


Figure 78: Waveform diagram for the pipelined CPU during the Insertion Sort benchmark.

Similar to the single-cycle CPU, the pipelined CPU executes the benchmark as expected. The signals used to demonstrate the waveform have already been explained in previous benchmarks. The CPU preforms as expected, accurately handling branching, memory access, and register updates while maintaining correct control flow and data integrity throughout the execution.

```
Register file content :
index =          0 , reg_out : signed =          0 , unsigned =          0
index =          1 , reg_out : signed =         10 , unsigned =         10
index =          2 , reg_out : signed =          8 , unsigned =          8
index =          3 , reg_out : signed =          1 , unsigned =          1
index =          4 , reg_out : signed =          0 , unsigned =          0
index =          5 , reg_out : signed =         48 , unsigned =         48
index =          6 , reg_out : signed =          9 , unsigned =          9
index =          7 , reg_out : signed =          0 , unsigned =          0
index =          8 , reg_out : signed =         85 , unsigned =         85
index =          9 , reg_out : signed =          0 , unsigned =          0
index =         10 , reg_out : signed =          0 , unsigned =          0
index =         11 , reg_out : signed =          0 , unsigned =          0
index =         12 , reg_out : signed =          0 , unsigned =          0
index =         13 , reg_out : signed =          0 , unsigned =          0
index =         14 , reg_out : signed =          0 , unsigned =          0
index =         15 , reg_out : signed =          0 , unsigned =          0
index =         16 , reg_out : signed =          0 , unsigned =          0
index =         17 , reg_out : signed =          0 , unsigned =          0
index =         18 , reg_out : signed =          0 , unsigned =          0
index =         19 , reg_out : signed =          0 , unsigned =          0
index =         20 , reg_out : signed =         10 , unsigned =         10
index =         21 , reg_out : signed =          0 , unsigned =          0
index =         22 , reg_out : signed =         -1 , unsigned = 4294967295
```

Figure 79: Register File content for all CPU designs after the Insertion Sort benchmark.

```
Data Memory Content :
Mem[ 0] =          1
Mem[ 1] =          2
Mem[ 2] =          5
Mem[ 3] =          7
Mem[ 4] =         10
Mem[ 5] =         15
Mem[ 6] =         16
Mem[ 7] =         48
Mem[ 8] =         85
Mem[ 9] =        255
```

Figure 80: Data Memory content for all CPU designs after the Insertion Sort benchmark.

The register and memory content snapshots confirm the correct execution of the Insertion Sort benchmark on both the single-cycle and pipelined CPUs. The register values align with the expected updates during sorting, reflecting accurate computation and control flow. Similarly, the sorted array in memory demonstrates the benchmark's successful completion, validating the proper handling of data manipulation and memory access across all tested configurations.

5.2 Silicore Benchmarks

To further validate our designs, we have included additional benchmarks to test the CPU's performance under different scenarios. These benchmarks focus on evaluating the CPU's ability to handle iterative computations and complex data manipulations. As with the committee benchmarks, these tests involve cycle-accurate simulations, waveform analysis, and validation of the final register and memory contents. The results from these benchmarks serve to reinforce the correctness and robustness of the CPU's functionality across a broader range of applications.

5.2.1 Bubble Sort Benchmark

```
# initializing the memory with reversed order array and with duplicates and negatives
.text

main:
#/ range : -10 : 10 (duplicates)
          addi $1, $0, 10

# number of elements = 42 (0 -> 41)
# $11 = 41 = 4 * $1 + 1
          sll $11, $1, 2
          addi $11, $11, 1

# index
          addi $2, $0, 0
          add $3, $0, $1

init:
          sub $8, $2, $11
          bgez $8, EXITINIT
          sw $3, 0($2)
          sw $3, 1($2)
          addi $2, $2, 2
          addi $3, $3, -1
          beq $0, $0, init
EXITINIT:

          addi $3, $0, 0
L1:
          beq $3, $11, EXITL1

# j
          addi $4, $0, 0
          add $7, $0, $11
          sub $7, $7, $3
L2:
          beq $4, $7, EXITL2

          lw $5, 0($4)
          lw $6, 1($4)
          sub $8, $5, $6
          bltz $8, else
          sw $5, 1($4)
          sw $6, 0($4)
else:

          addi $4, $4, 1
          beq $0, $0, L2
EXITL2:

          addi $3, $3, 1
          beq $0, $0, L1
EXITL1:
```

Figure 81: MIPS assembly code for the Bubble Sort benchmark.

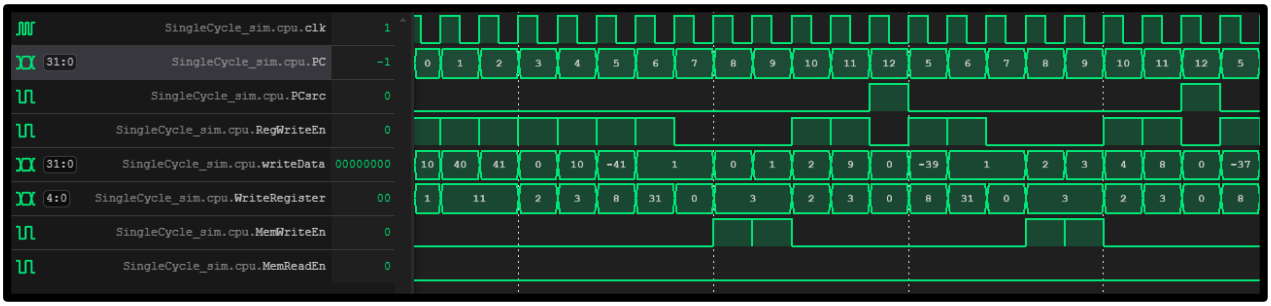


Figure 82: Waveform diagram for single-cycle CPU during Bubble Sort benchmark (1).

Figure 83: Waveform diagram for single-cycle CPU during Bubble Sort benchmark (2).

The waveforms confirm that the Bubble Sort benchmark executes as intended on the single-cycle CPU. **Figure 81** highlights the initialization phase, where array values are correctly written to memory with the "RegWriteEn" signal enabling proper register updates. The "WriteRegister" and "writeData" signals demonstrate accurate data handling during this phase.

Figure 82 showcases the sorting phase, where array elements are loaded, compared, and swapped as necessary. The assertions of "MemReadEn" and MemWriteEn" during memory operations, along with the branch signals, confirm proper execution of the nested loops and control flow of the algorithm.

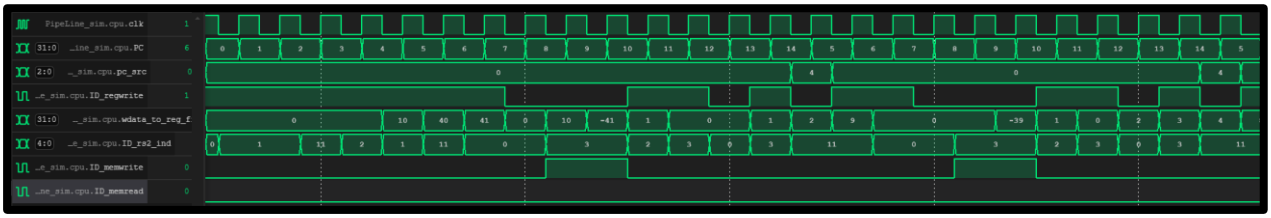


Figure 84: Waveform diagram for pipelined CPU during Bubble Sort benchmark (1).

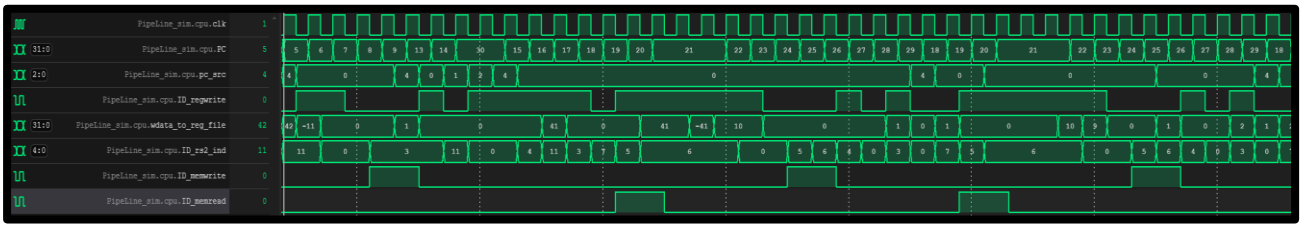


Figure 85: Waveform diagram for pipelined CPU during Bubble Sort benchmark (2).

```
Register file content :
index =          0 , reg_out : signed =           0 , unsigned =          0
index =          1 , reg_out : signed =          10 , unsigned =         10
index =          2 , reg_out : signed =          42 , unsigned =         42
index =          3 , reg_out : signed =          41 , unsigned =         41
index =          4 , reg_out : signed =           1 , unsigned =          1
index =          5 , reg_out : signed =         -10 , unsigned = 4294967286
index =          6 , reg_out : signed =         -10 , unsigned = 4294967286
index =          7 , reg_out : signed =           1 , unsigned =          1
index =          8 , reg_out : signed =           0 , unsigned =          0
index =          9 , reg_out : signed =           0 , unsigned =          0
index =         10 , reg_out : signed =           0 , unsigned =          0
index =         11 , reg_out : signed =          41 , unsigned =         41
```

Figure 86: Register File content for all CPU designs after the Bubble Sort benchmark.

```
Data Memory Content :
Mem[ 0] =         -10
Mem[ 1] =         -10
Mem[ 2] =          -9
Mem[ 3] =          -9
Mem[ 4] =          -8
Mem[ 5] =          -8
Mem[ 6] =          -7
Mem[ 7] =          -7
Mem[ 8] =          -6
Mem[ 9] =          -6
Mem[10] =          -5
Mem[11] =          -5
Mem[12] =          -4
Mem[13] =          -4
Mem[14] =          -3
Mem[15] =          -3
Mem[16] =          -2
Mem[17] =          -2
Mem[18] =          -1
Mem[19] =          -1
Mem[20] =           0
Mem[21] =           0
Mem[22] =           1
Mem[23] =           1
Mem[24] =           2
Mem[25] =           2
Mem[26] =           3
Mem[27] =           3
Mem[28] =           4
Mem[29] =           4
Mem[30] =           5
Mem[31] =           5
Mem[32] =           6
Mem[33] =           6
Mem[34] =           7
Mem[35] =           7
Mem[36] =           8
Mem[37] =           8
Mem[38] =           9
Mem[39] =           9
Mem[40] =          10
Mem[41] =          10
```

Figure 87: Data Memory content for all CPU designs after the Bubble Sort benchmark.

All CPU designs have successfully performed the benchmark as indicated by the sorted array contents shown in **Figure 86** and verified through register and memory state outputs matching expected results.

5.2.2 Fibonacci Benchmark

```
.text

main:
# the final Fibonacci value is in register $3
# the number of iteration is specified by the value
# of register $6

          addi $4, $0, 0
          addi $2, $0, 0
          addi $3, $0, 1
# if we increased the number of iterations we will see clearly that the prediction works even better
# it will introduce a number of bubbles (NOPs) but these will be less over large number of iterations
          addi $6, $0, 45
          jal FIB

          j END
FIB:
          add $5, $3, $0
          add $3, $2, $3
          add $2, $5, $0
          addi $4, $4, 1
          beq $4, $6, FINISH
          j FIB
FINISH:
          jr $31

END:
```

Figure 88: MIPS assembly code for Fibonacci benchmark.

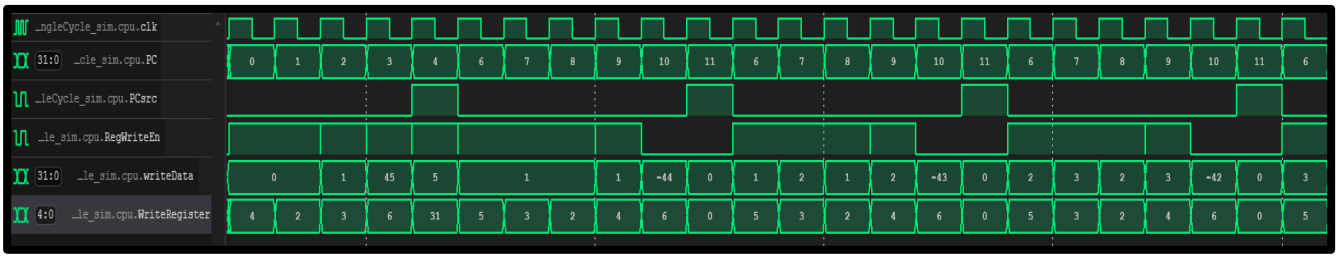


Figure 89: Waveform diagram for single-cycle CPU during Fibonacci benchmark.

Figure 90: Waveform diagram for pipelined CPU during Fibonacci benchmark.

```
Register file content :
index =          0 , reg_out : signed =           0 , unsigned =          0
index =          1 , reg_out : signed =           0 , unsigned =          0
index =          2 , reg_out : signed =  1134903170 , unsigned = 1134903170
index =          3 , reg_out : signed =  1836311903 , unsigned = 1836311903
index =          4 , reg_out : signed =          45 , unsigned =         45
index =          5 , reg_out : signed =  1134903170 , unsigned = 1134903170
index =          6 , reg_out : signed =          45 , unsigned =         45

 index =         31 , reg_out : signed =           5 , unsigned =          5
```

Figure 91: Register File content for all CPU designs after Fibonacci benchmark.

All CPU designs successfully executed the Fibonacci Benchmark, as confirmed by the correct sequence values stored in the register file, matching the expected results

5.3 Performance Optimization

Our timing analysis revealed several critical paths that posed challenges in meeting timing requirements. By examining the timing reports, we pinpointed critical paths that originated from the EX_MEM_buffer module and terminated in the IF_stage module (PC_register), with significant timing violations observed. For instance, initial paths showed a data arrival time of approximately 21.2 ns against a required time of 10 ns, resulting in slack violations of up to -11.088 ns. These delays primarily stemmed from the hazard control unit, forwarding muxes, and branch decision units.

We focused on addressing critical modules that contributed the most delay to the critical paths, starting with the muxes. Initially, we implemented muxes using case statements, if-else if constructs, and ternary operators, but all approaches resulted in similar delays. To overcome this, we revisited the design and rebuilt the muxes at the structural (gate) level, which led to a significant improvement, boosting the clock frequency by 10 MHz Beyond this, we targeted other critical elements of the path for optimization. We refined the forwarding muxes to improve the way data was passed from the memory stage and the write-back stage. Additionally, we optimized the branch decision unit to perform comparisons more efficiently, reducing delays in determining the validity of branch predictions.

To further minimize delay, we adjusted the read operation of the memory stage, making it edge-triggered to reduce its impact on the critical path. One of the most impactful steps was rethinking the clock operation itself. We recognized that the clock did not need to have a symmetric duty cycle or a fixed 50% ratio. Since the Datapath required more time during the low phase than the high phase, we adjusted the clock's duty cycle accordingly. This adjustment enabled us to push the maximum operating frequency to **82.99 MHz**, with positive setup slack of **+0.081 ns** and hold slack of **+0.389 ns**.

To optimize the design and further increase the operating frequency, we introduced a sixth pipeline stage. Specifically, we split the critical execution stage into two parts. The reason for this split is that the execution stage contains the logic required to handle data forwarding and perform branch calculations. By dividing this stage, we reduced the critical path. The first part of the new stage handles the forwarding logic, ensuring that data is forwarded from multiple stages, including the memory stage (execution hazard), the

write-back stage (memory hazard), and the execution stage itself (decode hazard). The second part of the new execution stage focuses on executing the instruction, including branch prediction and determination of whether the prediction was correct.

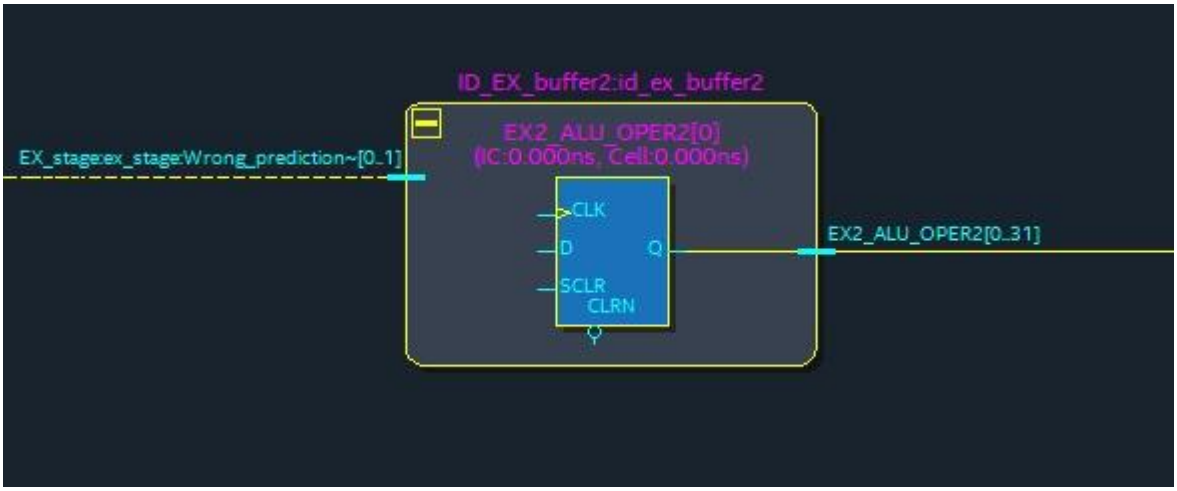


Figure 92: Critical path of the final design (1).

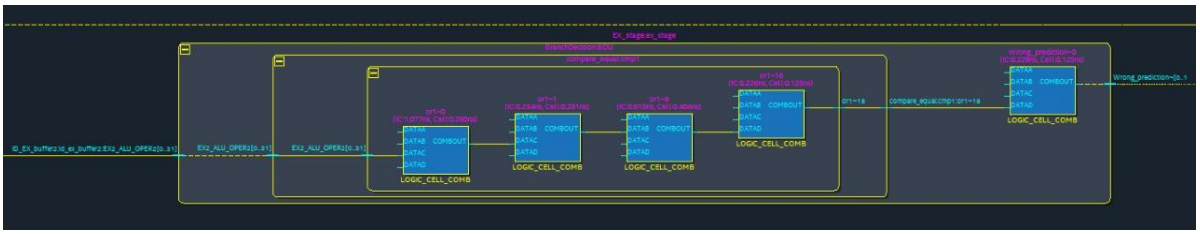


Figure 93: Critical path of the final design (2).

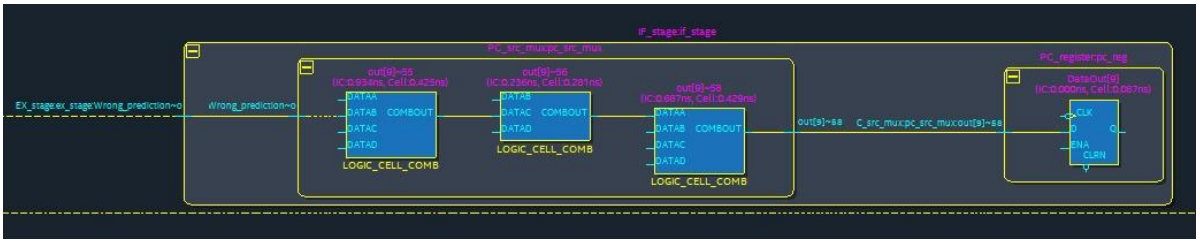


Figure 94: Critical path of the final design (3).

This modification significantly shortened the critical path and increased the maximum operating frequency to 82.99 MHz as a result, the setup slack became positive (+0.081 ns), and the hold slack improved to +0.389 ns. However, this optimization led to an increase in the average number of NOPs (no-operations or bubbles) inserted due to load-use hazards or wrong branch predictions. This is expected, as these stalls are necessary for maintaining correct execution. While the number of NOPs increased, the reduced critical path length and the higher clock frequency led to better overall performance, by increasing the overall throughput of the CPU.

Thus, when comparing the execution time of a program, we must consider both the number of cycles required to execute the program and the clock cycle time. The overall performance is a product of these two factors, with a notable improvement in performance after the critical path was optimized.

After optimizing the design, introducing a sixth pipeline stage, and tweaking the paths and computation methods for various parts of the critical path, the critical path following analysis with the timing analyzer is outlined as follows. The modules involved in this critical path are predicted to contribute the most delay, and the path begins at the **ID_EX2 buffer**, as shown in **Figure 91**, which holds the intermediate values that are passed to the next stage for execution. This buffer plays a key role in reducing delays by ensuring proper data forwarding for operands needed in the subsequent execution stages.

From the ID_EX2 buffer, the path moves into the actual execution stage, **EX2**, shown in **Figure 92**. In this stage, the module contributing the most delay is the **branch decision unit**, which contains the logic for comparing two forwarded operands from the previous stage. This comparison is critical for determining whether the branch prediction was incorrect. The result of this comparison is passed to the wrong_prediction signal, which is XORed with the branch prediction result.

Finally, the path leads to the **IF stage**, depicted in **Figure 93**, where the wrong_prediction signal is used to determine the next value of the program counter (PC). Based on this signal, the PC is directed to the correct calculated target address if the branch prediction was incorrect or allowed to continue normal execution if the prediction was accurate. This figure demonstrates how the PC source multiplexer integrates the wrong_prediction signal to adjust the control flow of the program.

In conclusion, the optimizations we introduced—including the sixth stage, refined execution stages, and the branch decision logic——resulted in a significant enhancement in the performance of the CPU. The reduction in the critical path length, along with the increase in the maximum operating frequency, demonstrated the effectiveness of our design changes. The modifications allowed us to achieve a substantial boost in clock speed, ultimately improving the overall performance of the system.

As the design will continue to evolve in future phases, we've identified several areas where further improvements can be made to enhance performance:

1. **Out-of-Order Execution**: Currently, our design executes instructions in a strict program order, which can create inefficiencies when there are data hazards. By implementing out-of-order execution, the CPU would be able to bypass stalled instructions and execute independent instructions whenever resources are available. This would take advantage of instruction-level parallelism (ILP) and significantly improve the CPU's throughput, allowing us to achieve a near-optimal cycles-per-instruction (CPI) value.

2. **Superscalar Design**: The current single execution unit can only handle one instruction per clock cycle, but with multiple functional units, the design could issue and execute multiple instructions simultaneously. This would improve overall throughput, and make better use of the CPU's resources to achieve higher execution rates.

3. **Advanced Branch Prediction**: While we have made improvements to the existing branch decision logic, enhancing the branch predictor could significantly reduce the number of pipeline stalls caused by mispredictions. Implementing more sophisticated prediction mechanisms—such as local branch prediction, global branch prediction, global with gselect, global with gshare, or even a tournament branch predictor—would improve prediction accuracy, minimizing branch misprediction penalties and increasing instruction flow.

4. **Parallel Execution and Multithreading**: Looking beyond single-thread performance, we could explore parallel execution techniques such as pipelining across multiple cores or using hardware threads to better utilize available resources and increase throughput.

By implementing these potential improvements, we could further enhance the efficiency and speed of the processor, paving the way for even more powerful and optimized CPU designs in the future.

5.4 Comparative Evaluation

Clock frequencies:

- Single-Cycle (SC): 27.78 MHz (36 ns as setup in the SDC file in Quartus)
- Pipelined (P): 82.99 MHz (12.05 ns as setup in the SDC file in Quartus)

Table 5: Comparison of single-cycle and pipelined processors in cycles, execution time, CPI and throughput across benchmarks.

Benchmark	Cycles (SC)	Cycles (P)	Execution Time (SC, ns)	Execution Time (P, ns)	CPI (SC)	CPI(P)	Throughput (SC, Million instructions /s)	Throughput (P, Million instructions /s)
BM1	16	21	576	253.05	1	1.3125	27.78	63.2
BM2	15	27	540	325.35	1	1.8000	27.78	46.1
BM3	46	57	1656	686.85	1	1.2391	27.78	67.0
BM4	34	49	1224	590.45	1	1.4412	27.78	57.6
BM5	98	151	3528	1819.55	1	1.5408	27.78	53.9
BM6	229	286	8244	3446.3	1	1.2489	27.78	66.4
BM7	9076	12813	326736	154396.7	1	1.4117	27.78	58.8
BM8	277	285	9972	3434.25	1	1.0289	27.78	80.7

To illustrate how these values were obtained, the calculations for the Max and Min Array benchmark (BM5) will be shown below.

For a single-cycle CPU, an instruction is executed every clock cycle.

$$CPI_{sc} = 1$$

Therefore, the number of instructions for each benchmark can be assumed to be the number of the single cycles consumed in the respective benchmark.

For BM5,

$$\#\ of\ instructions\ =\ Cycles\ (SC)\ =\ 98$$

To calculate the CPI for the pipelined CPU, follow the equation below:

$$CPI_P = \frac{Cycles\ (P)}{\#\ of\ instructions} = \frac{151}{98} \approx 1.5408$$

To calculate the execution time of the benchmark for each CPU, follow the steps shown:

$$T_{SC} = \frac{1}{f_{SC}} \times Cycles\ (SC) = \frac{1}{27.78\ MHz} \times 98 \approx 3528\ ns$$

$$T_P = \frac{1}{f_P} \times Cycles\ (P) = \frac{1}{82.99\ MHz} \times 151 \approx 1819.55\ ns$$

To calculate the throughput for each CPU while performing BM5:

$$TP_{SC} = \frac{f_{SC}}{CPI_{sc}} = \frac{27.78\ MHz}{1} = 27{,}780{,}000\ instructions/s$$

$$TP_P = \frac{f_P}{CPI_P} = \frac{82.99\ MHz}{1.5408} \approx 53{,}861{,}630\ instructions/s$$

Finally, there are two possible ways to calculate the speedup of the pipelined CPU in comparison with single-cycle CPU for BM5.

Benchmark	Speedup
BM1	2.28
BM2	1.66
BM3	2.41
BM4	2.07
BM5	1.94
BM6	2.39
BM7	2.15
BM8	2.90

$$Speedup = \frac{T_{SC}}{T_P} = \frac{TP_P}{TP_{SC}} = \frac{3528\ ns}{1819.55\ ns} = \frac{53{,}861{,}630\ instructions/s}{27{,}780{,}000\ instructions/s} \approx 1.938$$

Table 6: Speedup of pipelined CPU against single-cycle CPU at all benchmarks.

The calculations above have been repeated for all benchmarks to obtain **Table 5** and **Table 6**.

Now, it is possible to calculate the average speed up across all benchmarks to understand the improvement achieved.

$$Speedup_{avg} = \frac{\sum \quad Speedups}{\#\ of\ benchmarks}$$

$$Speedup_{avg} = \frac{2.28 + 1.66 + 2.41 + 2.07 + 1.94 + 2.39 + 2.15 + 2.90}{8} \approx 2.221$$

The results from our benchmarking and validation testing clearly demonstrate the contexts in which the pipelined processor excels over the single-cycle design. By analyzing performance metrics such as execution time, throughput, and cycle counts across various benchmarks, we observed that the pipelined processor outperforms the single-cycle processor, particularly for workloads with higher instruction counts and complex dependencies. However, this performance comes with increased complexity and resource utilization, presenting distinct advantages and disadvantages for both designs.

The single-cycle processor's greatest strength lies in its simplicity and predictability. Its control logic is straightforward, allowing for easy implementation and debugging. This simplicity translates into a smaller area footprint, with the single-cycle design utilizing only **2559 logic elements** compared to the pipelined processor's **3277 logic elements**. Additionally, its deterministic execution ensures predictable performance, which is particularly advantageous for workloads with uniform instruction latencies. For benchmarks with minimal dependencies, such as BM1 (arithmetic operations), the single-cycle processor performs efficiently. It completed BM1 in just 16 cycles, leveraging its low latency per instruction. However, its limited clock frequency of **27.78 MHz** restricted its execution time to **576 ns**, compared to **253.05 ns** for the pipelined processor operating at a higher clock speed of **82.99 MHz**

In contrast, the pipelined processor excels in scenarios involving complex dependencies and higher instruction counts, leveraging its ability to overlap instructions across pipeline stages. For example, in BM3, which involves iterative loops, the pipelined processor achieved a **2.41x speedup** over the single-cycle design by reducing execution time from **1656 ns** to **686.65 ns**. This advantage is even more apparent in control-intensive benchmarks like BM2 (branching) and BM4 (binary search). By utilizing branch prediction and resolving control hazards efficiently, the pipelined processor reduced execution times to **325.35 ns** and **590.45** respectively, achieving speedups of **1.66x** and **2.07x** over the single-cycle design. These gains highlight the pipelined processor's ability to maintain higher throughput even in the presence of hazards, as confirmed by validation tests and waveform analyses.

However, the complexity of the pipelined design introduces challenges. Managing hazards requires mechanisms like forwarding, stalling, and flushing, which increase control logic complexity and power consumption. While optimizations such as structural gate-level muxes and edge-triggered memory reads helped mitigate critical path delays, the pipelined processor's initial slack violations of **-11.088 ns** underscored the challenges of achieving high operational speeds. Despite these hurdles, iterative improvements allowed the pipelined design to reach a maximum clock frequency of **82.99 MHz given the positive slack**, significantly enhancing its performance metrics.

In summary, while the single-cycle design offers simplicity, lower area usage, and predictable performance for less demanding workloads, the pipelined processor excels in throughput and execution speed for complex and high-instruction-count workloads. The choice between these architectures depends on the specific workload characteristics, with the single-cycle design favoring simplicity and efficiency, and the pipelined processor catering to performance-critical scenarios.

6 Conclusion

This implementation of the MIPS processor highlights the collective effort of the team, each member spent a considerable time discussing and refining the project to ensure a heightened learning environment. The integration of the software portion such as an assembler and a cycle accurate simulator alongside the hardware portion provided a new perspective to the team's previous knowledge in the field.

The design started off with the implementation of a single cycle processor due to its simplicity and straightforward nature. It focused on the hardware components in the Datapath, ensuring correct and proper functionality of the instructions and the pseudo instructions. Then, to manage the flow of execution, halt instructions were added. During this period, the team noticed the shortcomings of the single cycle implementation at first hand, increasing the motivation of the team to start the pipelined implementation.

The 6-stage pipeline enhances the throughput and reflects on the goal of increasing the overall efficiency of the processor. Hazard detection unit is implemented to handle all the dependencies that the pipeline causes, as well as branch prediction techniques to avoid pipeline stalls.

Comprehensive and thorough verification was integral to the development process as various advanced simulations were employed to ensure full functionality enabling iterative improvements throughout the project.

At last, the team ensured that all the requirements for the MIPS processor are met and well-executed, reflecting the dedication and resilience of each member and their willingness to work and learn under pressure, balancing the load of their university courses and this invaluable project. This journey instilled valuable skills and insights that will shape the team's future endeavors in the upcoming phases and in the field of computer architecture.

7 References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2013.

[2] Hennessy, J. L., & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann.

[3] D. K. Dennis et al., "Single cycle RISC-V micro architecture processor and its FPGA prototype," in 017 7th International Symposium on Embedded Computing and System Design (ISED), Durgapur, India, 2017.