



Birzeit University
Faculty of Engineering & Technology
Department of Electrical & Computer Engineering
ENCS4370 | Computer Architecture

Multicycle RISC Processor in Verilog

Prepared By:
Tariq Odeh - 1190699
Wasim Atta – 1200920
Eyab Ghifari - 1190999

Instructor:
Dr. Ayman Hroub & Dr. Aziz Qaroush

Second Project – Multicycle RISC Processor Report

Birzeit
January, 2024

Abstract

This project involves the design and implementation of a Multi-Cycle RISC processor using Verilog HDL, characterized by a 32-bit instruction size and equipped with 16 general-purpose 32-bit registers. The processor's architecture boasts a segmented memory layout, supporting diverse instruction types: R-type, I-type, J-type, and S-type, thereby enhancing computational flexibility. Central to its functionality is a multi-cycle Datapath, encompassing five distinct stages, governed by a state machine-based control unit. This initiative aligns with both educational objectives and practical aspects of contemporary processor design, offering deep insights into the complexities of real-world processor architecture and bridging theoretical principles with hands-on application in Multi-Cycle RISC processor design.

Contents

English Abstract	I
Table of Contents	II
List of Tables	V
List of Figures	VI
1 Introduction	1
1.1 Overview	1
1.2 Motivation	2
1.3 Instruction Format	3
1.3.1 R-Type (Register Type)	3
1.3.2 I-Type (Immediate Type)	3
1.3.3 J-Type (Jump Type)	3
1.3.4 S-Type (Stack)	4
1.4 Instruction Sets	4
1.5 Report Outline	5
2 Components	6
2.1 Memory	7
2.1.1 Instruction Memory	7
2.1.2 Data Memory	7
2.2 Register File	8
2.3 Arithmetic Logic Unit (ALU)	8
2.4 Control Unit	9
2.5 Clock Organizer	9
2.6 Special Purpose Registers	10
2.6.1 Program Counter (PC)	10

2.6.2	Stack Pointer (SP)	10
2.7	Extender	11
2.8	Multiplexer	12
2.8.1	Multiplexer 4x1	12
2.8.2	Multiplexer 2x1	12
2.9	Adder	13
2.9.1	One Input Adder	13
2.9.2	Two Input Adder	13
2.10	Subtractor	14
2.11	Concatenation	14
3	Design and Implementation	15
3.1	Full Data Path	15
3.2	The System's Control Signals	16
3.2.1	Signals Meanings	16
3.2.2	Main Control Unit	17
3.2.3	ALU Control Unit	19
3.2.4	PC Control Unit	20
3.3	State Machine Diagram	21
4	Testing and Validation	22
4.1	Overview	23
4.2	Memory	23
4.2.1	Instruction Memory	23
4.2.2	Data Memory	24
4.3	Register File	25
4.4	Arithmetic Logic Unit (ALU)	26
4.5	Control Unit	27
4.6	Clock Organizer	29
4.7	Special Purpose Registers	31
4.7.1	Program Counter (PC)	31
4.7.2	Stack Pointer (SP)	32
4.8	Extender	32
4.9	Multiplexer	33
4.9.1	Multiplexer 4x1	33

4.9.2	Multiplexer 2x1	33
4.10	Adder	34
4.10.1	One Input Adder	34
4.10.2	Two Input Adder	34
4.11	Subtractor	35
4.12	Concatenation	35
4.13	All System	36
5	Teamwork, Conclusion and Future Work	39
5.1	Teamwork	39
5.2	Conclusion and Future Work	39
6	Appendix	40

List of Tables

3.1	Description of Signals and Effects	16
3.2	Main Control Unit Signals	17
3.3	Opcode and ALUOp Table	19
3.4	PC Control Unit Signals Table	20
3.5	State Transition Table	21

List of Figures

1.1	R-Type Instruction Format	3
1.2	I-Type Instruction Format	3
1.3	J-Type Instruction Format	4
1.4	S-Type Instruction Format	4
2.1	Instruction Memory Diagram	7
2.2	Data Memory Diagram	8
2.3	Register File Diagram	8
2.4	ALU Diagram	9
2.5	Control Unit Diagram	9
2.6	Clock Organizer Diagram	10
2.7	PC Diagram	10
2.8	SP Diagram	11
2.9	Extender Diagram	11
2.10	4x1 Multiplexer Diagram	12
2.11	2x1 Multiplexer Diagram	12
2.12	One Input Adder Diagram	13
2.13	Two Input Adder Diagram	13
2.14	Subtractor Diagram	14
2.15	Concatenation Diagram	14
3.1	Full Data Path Diagram	16
3.2	State Machine Diagram	21
4.1	Instruction Memory Testbench	24
4.2	Data Memory Testbench	25
4.3	Register File Testbench	26
4.4	ALU Testbench	26

4.5	Control Unit Testbench	29
4.6	Clock Organizer Testbench	31
4.7	PC Testbench	32
4.8	SP Testbench	32
4.9	Extender Testbench	33
4.10	Mux4x2 Testbench	33
4.11	Mux2x1 Testbench	34
4.12	One Input Adder Testbench	34
4.13	Two Input Adder Testbench	34
4.14	Subtractor Testbench	35
4.15	Concatenation Testbench	35
4.16	First code sequences in the ISA	36
4.17	Waveform for first code	36
4.18	The output-1 for first code	37
4.19	The output-2 for first code	37
4.20	Second code sequences in the ISA	38
4.21	Waveform for Second code	38
4.22	The output-1 for second code	38
4.23	The output-2 for second code	38

Chapter 1

Introduction

Contents

1.1 Overview	1
1.2 Motivation	2
1.3 Instruction Format	3
1.3.1 R-Type (Register Type)	3
1.3.2 I-Type (Immediate Type)	3
1.3.3 J-Type (Jump Type)	3
1.3.4 S-Type (Stack)	4
1.4 Instruction Sets	4
1.5 Report Outline	5

1.1 Overview

This project involves designing and testing a simple Multi-Cycle RISC processor using Verilog HDL, focusing on developing a processor with a 32-bit instruction size. This processor includes 16 general-purpose 32-bit registers, from R0 to R15, and features specialized 32-bit registers for both the program counter (PC) and the stack pointer (SP), enhancing its computational capabilities.

The processor's memory layout comprises three distinct segments: a static data segment, a code segment, and a stack segment operating on a LIFO (Last In, First Out) basis. This design allows explicit stack manipulation through push/pop instructions, adding to the processor's functionality. Furthermore, the processor has separate physical memories for instructions and data, with the data memory storing both the static data segment and the stack segment.

Supporting four instruction types (R-type, I-type, J-type, and S-type), the processor offers versatility in its operations. It is designed to be word-addressable and requires generating specific signals from the ALU to determine conditional branch outcomes, such as taken or not taken, factoring in signals like zero, carry, and overflow.

Central to the processor's architecture is its multi-cycle Datapath, incorporating five stages: Instruction Fetch, Instruction Decode, Execution, Memory Access, and Write

Back. Each stage plays a vital role in the instruction cycle, ensuring efficient and systematic processing.

The development of a control unit, using a state machine approach, is a significant aspect of this project. It is essential for managing the Datapath and generating control signals based on the instruction type and the Datapath's current state.

This project offers practical experience in processor design and a comprehensive understanding of instruction set architecture and multi-cycle processor design. It represents a fusion of theoretical and practical learning, aimed at mastering the complexities of designing a Multi-Cycle RISC processor with Verilog HDL.

1.2 Motivation

In this project, a simple Multi-Cycle RISC processor is meticulously designed to meet the evolving demands of processor architecture. The design incorporates several specific features:

- **Instruction and Word Size:** The processor features a 32-bit instruction size, aligning with contemporary computing standards and enabling a broad spectrum of operations.
- **General-Purpose Registers:** It includes 16 general-purpose 32-bit registers (R0 to R15), providing ample computational capacity and efficient instruction processing.
- **Specialized Registers:** Incorporates a 32-bit program counter (PC) and a 32-bit stack pointer (SP), with the SP pointing to the stack's topmost empty element, enhancing the processor's stack management capabilities.
- **Memory Segmentation:** Features a segmented memory layout, including static data, code, and stack segments, with the stack operating on a LIFO basis.
- **Instruction Types:** Supports four instruction types - R-type, I-type, J-type, and S-type - enhancing the processor's adaptability for various computational tasks.
- **Separate Data and Instruction Memories:** Implements separate physical memories for instructions and data, with the data memory storing both the static data segment and the stack segment.
- **Memory Addressability and Endianness:** The processor is word-addressable and adheres to big endian byte ordering, ensuring compatibility with a range of systems and software.

The motivation behind this design is to create a processor that is not only educational but also reflective of real-world processor architecture. By balancing traditional and modern design elements, the project aims to provide practical insights into the operational aspects of a Multi-Cycle RISC processor.

This project emphasizes a multi-cycle approach and the development of a sophisticated control unit using state machine methodology. The goal is to delve into the complexities

of processor design, moving beyond basics to explore control logic and state management intricacies. Ultimately, this project bridges theoretical learning with practical application, fostering a deep understanding of Multi-Cycle RISC processor design.

1.3 Instruction Format

The processor's Instruction Set Architecture (ISA) includes four distinct instruction formats: R-type, I-type, J-type, and S-type, each with a unique structure.

1.3.1 R-Type (Register Type)

- Opcode (6 bits)
- Rd (4 bits): Destination register
- Rs1 (4 bits): First source register
- Rs2 (4 bits): Second source register
- Unused (14 bits)

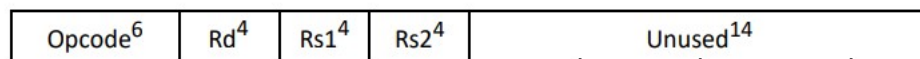


Figure 1.1: R-Type Instruction Format

1.3.2 I-Type (Immediate Type)

- Opcode (6 bits)
- Rd (4 bits): Destination register
- Rs1 (4 bits): First source register
- Immediate (16 bits): Unsigned for logic instructions, signed otherwise
- Mode (2 bits): Specifies load/store instruction modes, with options for post-increment and unused modes

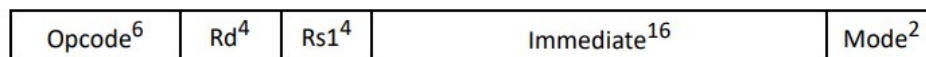


Figure 1.2: I-Type Instruction Format

1.3.3 J-Type (Jump Type)

- Opcode (6 bits)
- Jump Offset (26 bits): For instructions like 'jmp' and 'call', calculating the target address from the PC and offset

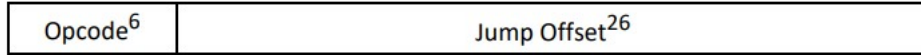


Figure 1.3: J-Type Instruction Format

1.3.4 S-Type (Stack)

- Opcode (6 bits)
- Rd (4 bits): Stack operation register
- Unused (22 bits): For stack operations like 'push' and 'pop'

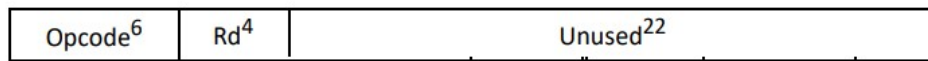


Figure 1.4: S-Type Instruction Format

1.4 Instruction Sets

The processor's instruction set is integral to its functionality, comprising diverse categories such as R-Type for arithmetic and logical operations, I-Type for immediate value operations, B-Type for conditional branching, J-Type for control flow, and S-Type for stack manipulations. Each instruction type is tailored to specific computational needs, enhancing the processor's operational versatility. The detailed instruction set, outlined in the table below, demonstrates the processor's capability to handle a wide range of tasks efficiently.

No.	Instr	Meaning	Opcode
R-Type Instructions			
1	AND	$\text{Reg(Rd)} = \text{Reg(Rs1)} \& \text{Reg(Rs2)}$	000000
2	ADD	$\text{Reg(Rd)} = \text{Reg(Rs1)} + \text{Reg(Rs2)}$	000001
3	SUB	$\text{Reg(Rd)} = \text{Reg(Rs1)} - \text{Reg(Rs2)}$	000010
I-Type Instructions			
4	ANDI	$\text{Reg(Rd)} = \text{Reg(Rs1)} \& \text{Imm16}$	000011
5	ADDI	$\text{Reg(Rd)} = \text{Reg(Rs1)} + \text{Imm16}$	000100
6	LW	$\text{Reg(Rd)} = \text{Mem}[\text{Reg(Rs1)} + \text{Imm16}]$	000101
7	SW	$\text{Mem}[\text{Reg(Rs1)} + \text{Imm16}] = \text{Reg(Rd)}$	000110
8	LW.POI	$\text{Reg(Rs1)} = \text{Mem}[\text{Reg(Rs1)}]; \text{Reg(Rs1)} = \text{Reg(Rs1)} + 1$	000111

9	BGT	if (Reg(Rd) > Reg(Rs1)) { Next PC = PC + sign_extended (Imm16) } else { PC = PC + 1 }	001000
10	BLT	if (Reg(Rd) < Reg(Rs1)) { Next PC = PC + sign_extended (Imm16) } else { PC = PC + 1 }	001001
11	BEQ	if (Reg(Rd) == Reg(Rs1)) { Next PC = PC + sign_extended (Imm16) } else { PC = PC + 1 }	001010
12	BNE	if (Reg(Rd) != Reg(Rs1)) { Next PC = PC + sign_extended (Imm16) } else { PC = PC + 1 }	001011
J-Type Instructions			
13	JMP	Next PC = [PC[31:26], Immediate26]	001100
14	CALL	Next PC = [PC[31:26], Immediate26]; PC + 1 is pushed on the stack	001101
15	RET	Next PC = top of the stack	001110
S-Type Instructions			
16	PUSH	Rd is pushed on the top of the stack	001111
17	POP	The top element of the stack is popped, and it is stored in the Register	010000

This table provides a concise overview of the processor's instruction set, illustrating the opcode and functionality of each instruction type, essential for understanding the processor's capabilities and design.

1.5 Report Outline

The report commences with **Introduction**, providing an overview, project motivation, and a detailed discussion on instruction formats including R-Type, I-Type, J-Type, and S-Type, as well as the instruction sets used. Following this, the **Components** chapter dives into the essential elements of the system, covering Instruction and Data Memory, Register File, Arithmetic Logic Unit (ALU), Control Unit, Clock Organizer, and Special Purpose Registers like the Program Counter and Stack Pointer. It also examines Extenders, Multiplexers, Adders, Subtractors, and the Concatenation process. The **Design and Implementation** chapter delves into the system's design and logic, focusing on the full data path, control signals, and the state machine diagram. In the **Testing and Validation** chapter, a critical analysis of each component's performance is conducted, including Memory, Register File, ALU, Control Unit, and overall system evaluation. The report concludes with **Teamwork, Conclusion, and Future Work**, reflecting on collaborative efforts, summarizing findings, and suggesting future research directions. An **Appendix** supplements the report with additional supporting materials and data.

Chapter 2

Components

Contents

2.1	Memory	7
2.1.1	Instruction Memory	7
2.1.2	Data Memory	7
2.2	Register File	8
2.3	Arithmetic Logic Unit (ALU)	8
2.4	Control Unit	9
2.5	Clock Organizer	9
2.6	Special Purpose Registers	10
2.6.1	Program Counter (PC)	10
2.6.2	Stack Pointer (SP)	10
2.7	Extender	11
2.8	Multiplexer	12
2.8.1	Multiplexer 4x1	12
2.8.2	Multiplexer 2x1	12
2.9	Adder	13
2.9.1	One Input Adder	13
2.9.2	Two Input Adder	13
2.10	Subtractor	14
2.11	Concatenation	14

2.1 Memory

The memory in this processor is divided into two parts: Instruction Memory, which stores and fetches various 32-bit instructions, and Data Memory, crucial for storing and accessing data vital for the processor's operations.

2.1.1 Instruction Memory

The Instruction Memory is designed to store a predetermined collection of 32-bit instructions. It functions in sync with the clock signal, fetching instructions using a 32-bit program counter (PC) address. This memory is preloaded with a diverse range of instructions, including R-Type, I-Type, J-Type, and S-Type operations, each with parameterized opcodes for efficient access and execution. This configuration guarantees a dependable and prompt retrieval of instructions, perfectly synchronized with the processor's clock cycles, underscoring the crucial role of memory in facilitating efficient instruction management and execution within the processor's architecture.

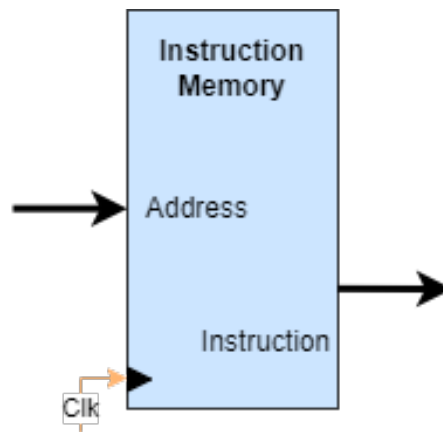


Figure 2.1: Instruction Memory Diagram

2.1.2 Data Memory

The data memory module within this implementation serves as a crucial component for storing and accessing data essential to the processor's execution of instructions. It comprises various inputs and signals, including a 32-bit address input, 32-bit data input, MemRead (a signal for data retrieval), MemWrite (a signal for data storage), SAddress (for specifying data destination), and a clock input. When MemRead is active, the module retrieves data from the specified memory address and outputs it as a 32-bit data stream. Conversely, when MemWrite is activated, the module writes the provided 32-bit data to the designated memory location. Synchronization with the processor's clock via the clock input ensures that data operations occur at the precise moments required for seamless functionality.

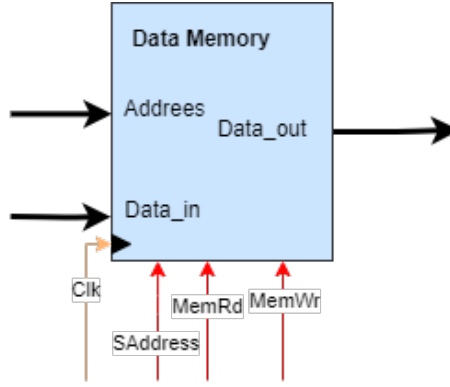


Figure 2.2: Data Memory Diagram

2.2 Register File

The register file is a crucial component of the processor, storing 16 32-bit general-purpose registers (R0 to R15). It offers four input ports for source and destination register addresses, each represented by 4-bit values. Signals like RegWr1 and RegWr2 enable writing to specific registers, while BusA and BusB provide 32-bit data outputs for reading operations. When the RegWrite signal is active, the register file writes data from BusW1 and BusW2 to the designated registers. This component plays a pivotal role in executing instructions that involve register operations, ensuring efficient read and write access for smooth processor execution.

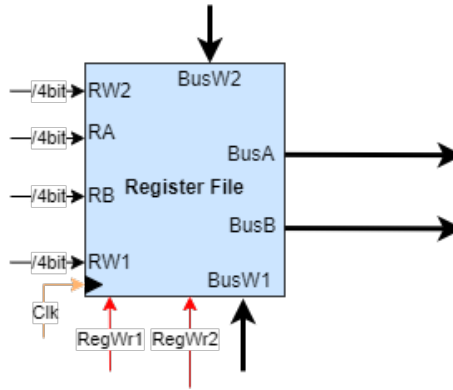


Figure 2.3: Register File Diagram

2.3 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a pivotal component of the Multi-Cycle RISC processor, responsible for executing arithmetic and logical operations. This ALU is equipped with two 32-bit input operands and utilizes a 2-bit operation code (Op) to select the required operation. The ALU outputs a 32-bit result and also provides two flags: the Zero flag, which indicates whether the result is zero, and the Sign flag, derived from the most significant bit to signify the result's sign in 2's complement form. In this design, the ALU handles three key operations: bitwise AND, addition, and subtraction. Subtraction is uniquely implemented using the same circuitry as addition by negating the second input

through two's complement, involving the inversion of bits and addition of one.

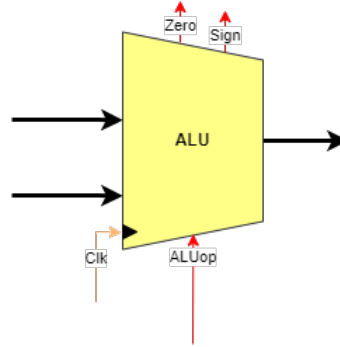


Figure 2.4: ALU Diagram

2.4 Control Unit

The control unit is a pivotal component responsible for the precise sequencing of instruction execution within the processor. It efficiently manages and coordinates 14 crucial control signals (PCSrc, RegS, RegWr1, RegWr2, ALUop, ExtOp, ALUSrc, Rdata, SAddress, StaS, MemRd, MemWr, PopSrc, WBdata) that are vital for the synchronization of the fetch, decode, and execution phases. These control signals govern the flow of data and the execution of operations across various processor components, including instruction memory, register file, ALU, and data memory, ensuring the smooth execution of instructions and proper data handling. Employing a control path and a finite-state machine, the control unit regulates the processor's state during each instruction phase, generating the necessary control signals to read instructions, decode opcodes, retrieve operands, and execute operations. This meticulous orchestration guarantees the efficient processing of instructions and the accurate management of data throughout the processor's operation.

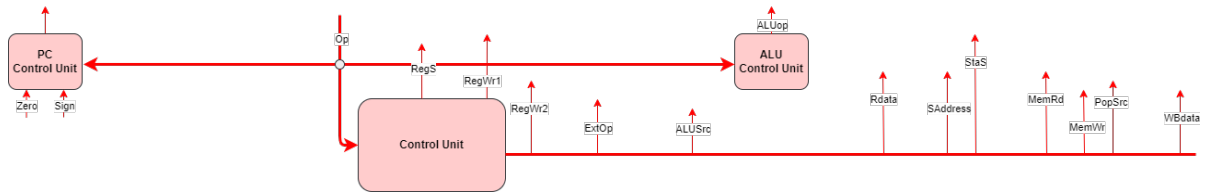


Figure 2.5: Control Unit Diagram

2.5 Clock Organizer

The clock organizer is a pivotal module within the processor, converting it into a multicycle, finite state machine. Its primary role involves allocating the correct number of clock cycles to each stage of instruction execution, ensuring that every instruction receives the precise duration needed for its completion. In contrast to the single-cycle approach, the clock organizer considers two key inputs: the clock signal (clk) and the opcode (instruction type) to determine the subsequent execution stage. This dual-input mechanism facilitates a more sophisticated and customized approach to instruction processing, ultimately enhancing the efficiency and performance of the processor.

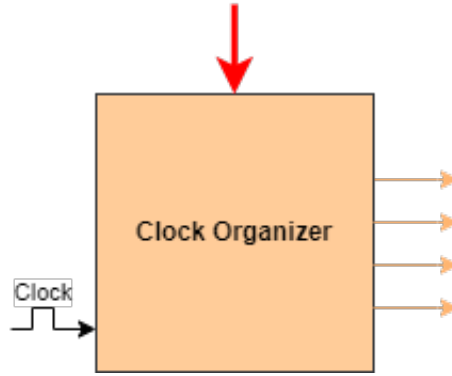


Figure 2.6: Clock Organizer Diagram

2.6 Special Purpose Registers

The Multi-Cycle RISC processor features two key registers: the Program Counter (PC) for instruction sequencing and control, and the Stack Pointer (SP) for managing stack operations and addressing.

2.6.1 Program Counter (PC)

In Multi-Cycle RISC processor design, the Program Counter (PC) plays a crucial role in fetching instructions and controlling program execution. It gets input from a multiplexer with four options: normal sequential progression ($PC + 1$) for regular instructions, jump instruction address, branch target address for conditional branches, and return address from the stack for function returns. This setup allows the PC to efficiently manage instruction fetching and determine the program's execution order, whether it's following a linear sequence or handling complex control flow scenarios like jumps and branches.

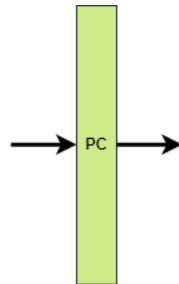


Figure 2.7: PC Diagram

2.6.2 Stack Pointer (SP)

The Stack Pointer (SP) is a vital component in the Multi-Cycle RISC processor, specifically designed to oversee the stack, a fundamental data structure in computing. The SP keeps track of the top of the stack's address, usually representing the next available storage slot. In the processor's design, the stack operates on a Last In, First Out (LIFO) basis, where the most recently added item is the first to be removed. The SP's role is particularly critical in stack-related operations, especially with S-Type instructions like 'push' and 'pop'. During a 'push' operation, the SP updates to point to a new top po-

sition after storing the value of a specified register (Rd). Conversely, during a 'pop', it retrieves the topmost element from the stack and adjusts the stack pointer accordingly. In the given module, the SP starts with an initial address and adapts dynamically based on input data, reflecting the current state of the stack. This dynamic SP adjustment is essential for effectively managing function calls, returns, and temporary data storage in the processor.

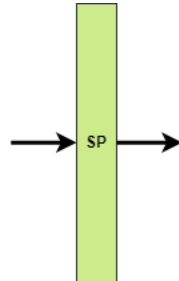


Figure 2.8: SP Diagram

2.7 Extender

The Extender in the Multi-Cycle RISC processor adapts 16-bit immediate values to the processor's 32-bit architecture. It extends the input to 32 bits, matching the processor's data path. The Extender can perform signed extension by copying the sign bit to the upper 16 bits for maintaining the value's sign, or it can do unsigned extension by adding 16 zeros to the upper half, preserving the original value. This dual function enables the Extender to handle both signed and unsigned immediate values, critical for various instructions such as arithmetic operations and memory access. Its straightforward design and versatility are essential for the processor's effective execution of a wide range of instructions.

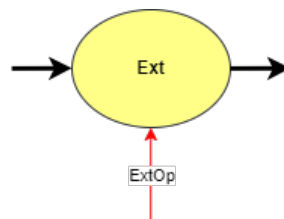


Figure 2.9: Extender Diagram

2.8 Multiplexer

The multiplexer in the system includes two types: the Mux4x1, which selects from four inputs using a 2-bit signal, and the Mux2x1, choosing between two inputs based on a control signal. Both are essential for decision-making in digital circuits and processor architectures.

2.8.1 Multiplexer 4x1

The Mux4x1 is an extension of the Mux2x1, offering a choice between four input signals using a 2-bit select signal. It takes four inputs (I0, I1, I2, I3) and uses the 2-bit 'select' signal to determine the output. This allows for complex decision-making based on the select bits, making it valuable in managing multiple data sources in intricate digital systems. Its ability to handle up to four inputs with a customizable width is crucial in advanced digital circuits and computing systems.

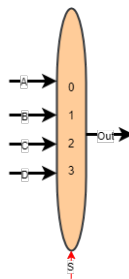


Figure 2.10: 4x1 Multiplexer Diagram

2.8.2 Multiplexer 2x1

The Mux2x1 is a digital circuit that chooses between two inputs using a control signal. It has I0 and I1 inputs, with 'select' determining the output. When 'select' is 1, it outputs I1; otherwise, it's I0. It's versatile and can handle various data sizes, making it useful in digital circuits and processor architectures.

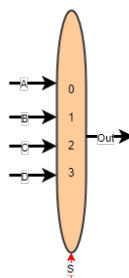


Figure 2.11: 2x1 Multiplexer Diagram

2.9 Adder

The adder in the system comprises two variants: the One-Input Adder, which increments a single 32-bit input by one, useful for tasks like array indexing, and the Two-Input Adder, which adds two 32-bit inputs, crucial for a wide range of computing tasks including arithmetic and data processing in ALUs.

2.9.1 One Input Adder

The One-Input Adder is a basic digital circuit that increments a value by one. In its simplest form (Adder_1), it takes a single 32-bit input (A) and produces an output that's the input value plus one. This adder is essential for tasks like array indexing and counter incrementing in iterative processes. Its simplicity makes it versatile and efficient, commonly used in digital circuit designs, including processor architectures for managing sequential operations.

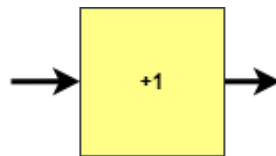


Figure 2.12: One Input Adder Diagram

2.9.2 Two Input Adder

The Two-Input Adder is a vital component in digital systems, performing the fundamental arithmetic operation of addition. The Adder_2 module exemplifies this by taking two 32-bit inputs (A and B) and producing an output that's the sum of these inputs. This adder is essential for a broad spectrum of computing tasks, from basic arithmetic to complex operations like address calculation and data processing. Its capacity to handle two separate inputs and generate a sum makes it indispensable in processor arithmetic logic units (ALUs), playing a significant role in executing various instructions.

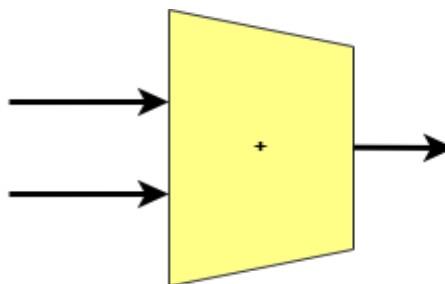


Figure 2.13: Two Input Adder Diagram

2.10 Subtractor

The One-Input Subtractor, as depicted in the Subtractor_1 module, is a digital circuit designed to decrement a value by one. It takes a single 32-bit input (A) and outputs a value that is one less than the input. This subtractor is particularly useful in digital systems for operations like counting down, loop control in decrementing iterations, and simple arithmetic operations. The one-input subtractor's design is straightforward yet effective for tasks requiring decremental adjustments, making it a valuable component in diverse computing applications, especially where precise control over single-step decrements is necessary.

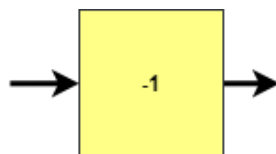


Figure 2.14: Subtractor Diagram

2.11 Concatenation

The Concatenator is a specialized digital component in processor design, utilized for combining different pieces of data into a single, larger data unit. In the context of the 'Concatenator' module, it merges a 26-bit input (A) and a 6-bit input (B) to produce a 32-bit output. This operation is achieved by concatenating the 6-bit input B in front of the 26-bit input A, forming a continuous 32-bit binary sequence. Such a component is particularly useful in scenarios where parts of data or addresses need to be assembled from smaller fragments, such as constructing full addresses from offset and base parts in jump and branch instructions within a processor. The Concatenator's ability to seamlessly merge different bits of data underpins its utility in efficiently handling complex data manipulations and addressing schemes in digital circuits and computing systems.

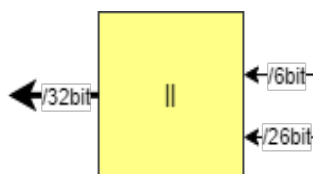


Figure 2.15: Concatenation Diagram

Chapter 3

Design and Implementation

Contents

3.1 Full Data Path	15
3.2 The System's Control Signals	16
3.2.1 Signals Meanings	16
3.2.2 Main Control Unit	17
3.2.3 ALU Control Unit	19
3.2.4 PC Control Unit	20
3.3 State Machine Diagram	21

3.1 Full Data Path

The fully interconnected datapath represents the culmination of seamlessly integrating all components and interconnections within the Multi-Cycle Processor with Memory. This comprehensive integration encompasses both the data path and control signal designs. In the initial instruction fetch stage, we introduced essential components such as the program counter and an incrementing adder to generate the subsequent instruction address, which then served as input for the instruction memory. Subsequently, the output from the instruction memory seamlessly flowed into the decode stage. Within the instruction decode stage, the register file played a pivotal role in extracting values from the general-purpose registers. These extracted register values were then seamlessly forwarded to the ALU stage, where the appropriate operation, as dictated by the instruction, was meticulously executed. Following this, the resulting data underwent either storage or retrieval from the data memory. Moreover, if there was write-back data, it was thoughtfully preserved within the registers in the register file. While this provides a high-level overview of the datapath stages, it's crucial to emphasize that each instruction type necessitates tailored handling and optimization within the datapath design to ensure the utmost efficiency and performance of the processor.

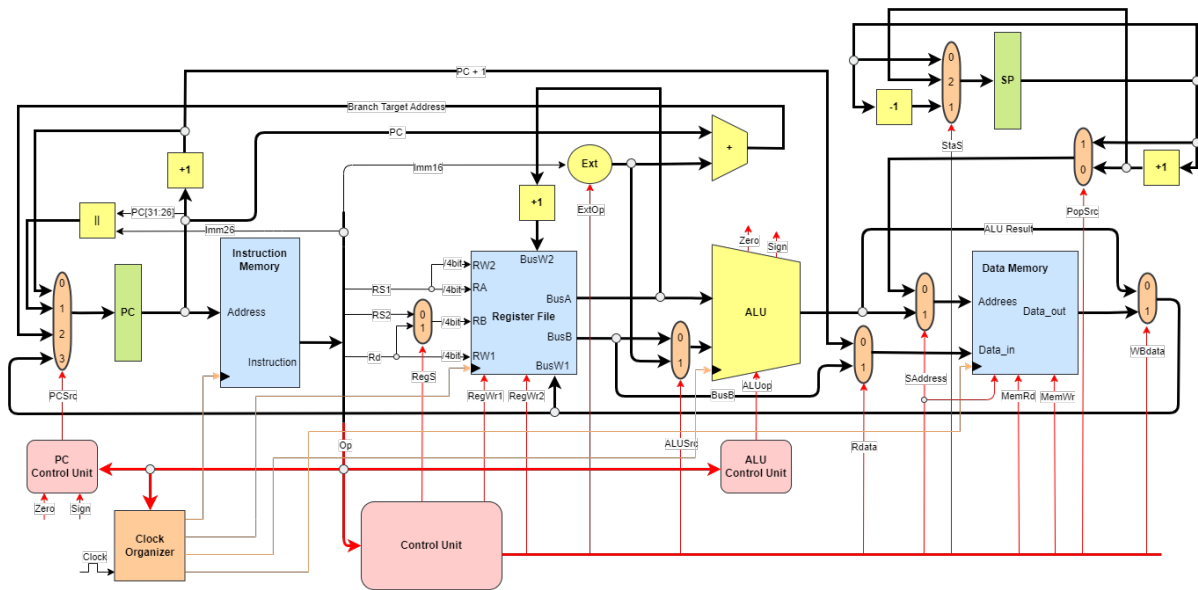


Figure 3.1: Full Data Path Diagram

3.2 The System's Control Signals

3.2.1 Signals Meanings

As previously mentioned, the system encompasses fourteen control signals, each of which carries a distinct value determined by factors such as the instruction type, instruction function, and the presence of a stop bit. The following control signals are outlined along with their respective meanings:

Table 3.1: Description of Signals and Effects

Signal	Effect when '0'	Effect when '1'	Effect when '2'	Effect when '3'
PCSrc	The next PC = PC + 1	The next PC = {PC[31:26], Immediate26}	The next PC = PC + sign extended (Imm16)	The next PC = top of the stack
RegS	Destination register = Rs2	Destination register = Rd	-----	-----
RegWr1	Destination register Rd not written	Destination register Rd is written with the data on BusW1	-----	-----
RegWr2	Destination register Rs1 not written	Destination register Rs1 is written with the data on BusW2	-----	-----
ALUOp	The operation is AND	The operation is ADD	The operation is SUM	-----
ExtOp	16-bit immediate is zero-extended	16-bit immediate is sign-extended	-----	-----
ALUSrc	Second ALU operand is the value of register Rs2 that appears on BusB	Second ALU operand is the value of the extended 16-bit immediate	-----	-----
Rdata	Data_in to the memory is PC + 1	Data_in to the memory is BusB	-----	-----
SAddress	Address into memory is SP	Address into memory is ALU address	-----	-----
StaS	No Change in SP	SP decrement	Next PC = top of the stack	Rd is pushed on the top of the stack
MemRd	Data memory is NOT read	Data memory is read Data_out ← Memory[address]	-----	-----
MemWr	Data Memory is NOT written	Data memory is written Memory[address] ← Data_in	-----	-----
PopSrc	Pass the SP value	Pass the SP + 1 value	-----	-----
WBdata	BusW = The top element of the stack	BusW = ALU result	BusW = Data_out from Memory	-----

3.2.2 Main Control Unit

The table presented below illustrates the values of all control signals for each instruction, with some of them being designated as 'x' to signify that their precise value is not of utmost importance or has minimal impact on the instruction's operation.

Table 3.2: Main Control Unit Signals

Op	RegS	RegWr1	RegWr2	ExtOp	ALUSrc	Rdata	SAddress	MemRd	MemWr	StaS	PopSrc	WBdata
AND	0 = Rs2	1	0	X	0 = BusB	X	X	0	0	00	X	0 = ALU
ADD	0 = Rs2	1	0	X	0 = BusB	X	X	0	0	00	X	0 = ALU
SUB	0 = Rs2	1	0	X	0 = BusB	X	X	0	0	00	X	0 = ALU
ANDI	X	1	0	0	1 = Imm	X	X	0	0	00	X	0 = ALU
ADDI	X	1	0	1	1 = Imm	X	X	0	0	00	X	0 = ALU
LW	X	1	0	1	1 = Imm	X	1	1	0	00	X	1 = Data_out
LWPOI	X	1	1	1	1 = Imm	X	1	1	0	00	X	1 = Data_out
SW	1 = Rd	0	0	1	1 = Imm	1	1	0	1	00	X	X
BGT	1 = Rd	0	0	1	0 = BusB	X	X	0	0	00	X	X
BLT	1 = Rd	0	0	1	0 = BusB	X	X	0	0	00	X	X
BEQ	1 = Rd	0	0	1	0 = BusB	X	X	0	0	00	X	X
BNE	1 = Rd	0	0	1	0 = BusB	X	X	0	0	00	X	X
JMP	X	0	0	X	X	X	X	0	0	00	X	X
CALL	X	0	0	X	X	0	0	0	1	10 = SP+1	0	X
RET	X	0	0	X	X	X	0	1	0	01 = SP-1	1	1 = Data_out
PUSH	1	0	0	X	X	1	0	0	1	10 = SP+1	0	X
POP	X	1	0	X	X	X	0	1	0	01 = SP-1	1	1 = Data_out

Each signal is assigned a Boolean value based on its corresponding Boolean expression, determined by instruction type, function, and stop bit, as previously mentioned.

- **RegS:** SW || BGT || BLT || BEQ || BNE || PUSH
- **RegWr1:** AND || ADD || SUB || ANDI || ADDI || LW || LW.POI || POP
- **RegWr2:** LW.POI
- **ExtOp:** ADDI || LW || LW.POI || SW || BGT || BLT || BEQ || BNE
- **ALUSrc:** ANDI || ADDI || LW || LW.POI || SW
- **Rdata:** SW || PUSH
- **SAddress:** LW || LW.POI || SW
- **MemRd:** LW || LW.POI || RET || POP
- **MemWr:** SW || CALL || PUSH
- **StaS[0]:** RET || POP
- **StaS[1]:** CALL || PUSH
- **PopSrc:** RET || POP
- **WBdata:** LW || LW.POI || RET || POP

3.2.3 ALU Control Unit

The "ALUOp" signal has distinct effects based on its value. When it's '0', the ALU performs the AND operation. When set to '1', it triggers the ADD operation. When it's '2', it instructs the ALU to perform the SUB operation. However, when the value is '3', there is no specified effect, and it remains inactive or undefined in this context.

Table 3.3: Opcode and ALUOp Table

Op	Opcode Value	ALUOp	2-bit Coding
AND	000000	AND	00
ADD	000001	ADD	01
SUB	000010	SUB	10
ANDI	000011	AND	00
ADDI	000100	ADD	01
LW	000101	ADD	01
LW.POI	000110	ADD	01
SW	000111	ADD	01
BGT	001000	SUB	10
BLT	001001	SUB	10
BEQ	001010	SUB	10
BNE	001011	SUB	10
JMP	001100	X	X
CALL	001101	X	X
RET	001110	X	X
PUSH	001111	X	X
POP	010000	X	X

- **ALUOp[0]:** ADD || ADDI || LW || LW.POI || SW
- **ALUOp[1]:** SUB || BGT || BLT || BEQ || BNE

3.2.4 PC Control Unit

The "PCSrc" signal dictates the program counter's (PC) behavior: '0' increments PC by 1, '1' combines bits 31 to 26 with Immediate26, '2' increments PC by sign-extended Imm16, and '3' sets PC to the top of the stack for subroutine operations.

Table 3.4: PC Control Unit Signals Table

OP	Zero Flag	Sign Flag	PCSrc
AND	X	X	00 = Increment PC
ADD	X	X	00 = Increment PC
SUB	X	X	00 = Increment PC
ANDI	X	X	00 = Increment PC
ADDI	X	X	00 = Increment PC
LW	X	X	00 = Increment PC
LW.POI	X	X	00 = Increment PC
SW	X	X	00 = Increment PC
BLT	0	0	10 = Branch Target Address
BLT	X	1	00 = Increment PC
BLT	1	X	00 = Increment PC
BGT	0	1	10 = Branch Target Address
BGT	1	X	00 = Increment PC
BGT	X	0	00 = Increment PC
BEQ	1	X	10 = Branch Target Address
BEQ	0	X	00 = Increment PC
BNE	1	X	00 = Increment PC
BNE	0	X	10 = Branch Target Address
JMP	X	X	01 = Jump Target Address
CALL	X	X	01 = Jump Target Address
RET	X	X	11 = Top of the stack address
PUSH	X	X	00 = Increment PC
POP	X	X	00 = Increment PC

3.3 State Machine Diagram

A systematic approach was used to create the state machine diagram for a multi-cycle processor. We first analyzed the sequence of operations and their conditions, identifying the necessary states for the diagram. Key operations in each state, such as updating the Program Counter or writing to the register file, were detailed. To validate the diagram's accuracy, we ran simulations of processor activities, which helped in constructing the state table and ensuring the outcomes matched our expectations.

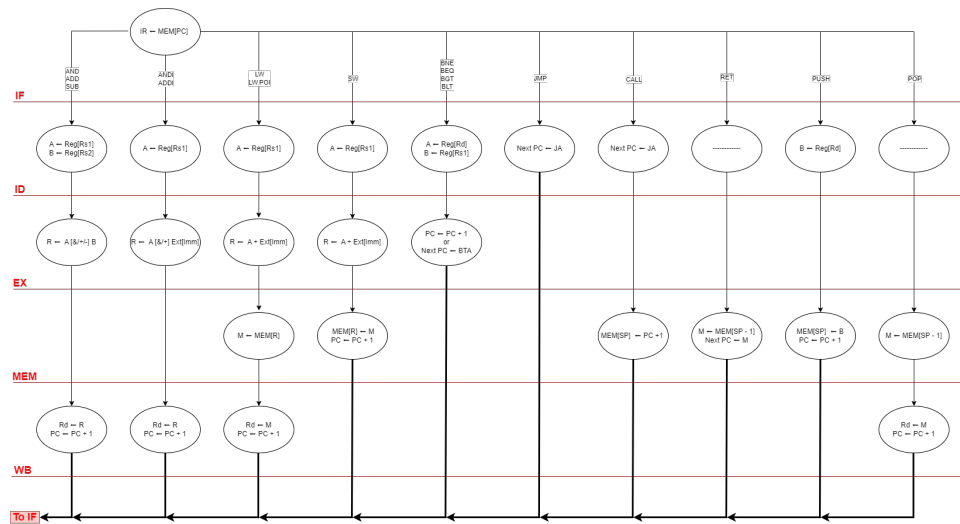


Figure 3.2: State Machine Diagram

Table 3.5: State Transition Table

Current Stage	State Code	Next Stage	Next State Code	Instructions Cases
Instruction Fetch	000	Instruction Decode	001	Always
Instruction Decode	001	Execute	010	R-Type, I-Type
		Instruction Fetch	000	JMP
		Memory	011	CALL, RET, S Type
Execute	010	Register Write	100	R-Type, ANDI, ADDI
		Memory	011	LW, LW.POI, SW
		Instruction Fetch	000	BNE, BEQ, BGT, BLT
Memory	011	Register Write	100	LW, LW.POI, POP
		Instruction Fetch	000	SW, CALL, RET, PUSH
Register Write	100	Instruction Fetch	000	Always

Chapter 4

Testing and Validation

Contents

4.1 Overview	23
4.2 Memory	23
4.2.1 Instruction Memory	23
4.2.2 Data Memory	24
4.3 Register File	25
4.4 Arithmetic Logic Unit (ALU)	26
4.5 Control Unit	27
4.6 Clock Organizer	29
4.7 Special Purpose Registers	31
4.7.1 Program Counter (PC)	31
4.7.2 Stack Pointer (SP)	32
4.8 Extender	32
4.9 Multiplexer	33
4.9.1 Multiplexer 4x1	33
4.9.2 Multiplexer 2x1	33
4.10 Adder	34
4.10.1 One Input Adder	34
4.10.2 Two Input Adder	34
4.11 Subtractor	35
4.12 Concatenation	35
4.13 All System	36

4.1 Overview

Testbenches provided a crucial platform for designers to evaluate modules in a controlled setting before incorporating them into larger systems, ensuring the modules functioned correctly. Understanding the module's requirements, input and output signals, and the conditions under which it was meant to operate was key to constructing effective Testbenches. By simulating various scenarios and stimuli, designers were able to verify that their modules performed as expected and identify and address errors early in the design phase. Thus, Testbenches played an essential role in validating the precision and reliability of Verilog designs. Test environments were created for each module to simulate a selection of possible scenarios.

4.2 Memory

4.2.1 Instruction Memory

This module represents an instruction memory, which stores instructions for a processor. It uses a clock signal and an address input to output the corresponding instruction.

```
1 module InstructionMemory(  
2     input wire clk,  
3     input [31:0] address,  
4     output reg [31:0] instruction  
5 );  
6  
7  
8     // Define the space of memory with 32-bit width  
9     reg [31:0] MemoryInst [0:1024];  
10  
11     // Define Register Names (R0 to R15)  
12     parameter R0 = 4'b0000, R1 = 4'b0001, R2 = 4'b0010, R3 = 4'b0011, R4 = 4'b0100,  
13         R5 = 4'b0101, R6 = 4'b0110, R7 = 4'b0111, R8 = 4'b1000, R9 = 4'b1001,  
14         R10 = 4'b1010, R11 = 4'b1011, R12 = 4'b1100, R13 = 4'b1101, R14 = 4'b1110,  
15         R15 = 4'b1111;  
16  
17     // R-Type Instructions Opcode Values  
18     parameter AND = 6'b000000, ADD = 6'b000001, SUB = 6'b000010;  
19     // I-Type Instructions Opcode Values  
20     parameter ANDI = 6'b000011, ADDI = 6'b000100, LW = 6'b000101, LWPOL = 6'b000110,  
21         SW = 6'b000111, BGT = 6'b001000, BLT = 6'b001001, BEQ = 6'b001010, BNE =  
22         6'b001011;  
23     // J-Type Instructions Opcode Values  
24     parameter JMP = 6'b001100, CALL = 6'b001101, RET = 6'b001110;  
25     // S-Type Instructions Opcode Values  
26     parameter PUSH = 6'b001111, POP = 6'b010000;  
27  
28     // Initialize the memory with instructions  
29     initial begin  
30         MemoryInst[1] = {ADDI, R1, R0, 16'd5, 2'b00};  
31         MemoryInst[2] = {ADDI, R2, R0, 16'd3, 2'b00};  
32         MemoryInst[3] = {ADDI, R3, R0, 16'd5, 2'b00};  
33         MemoryInst[4] = {ADDI, R4, R0, 16'd5, 2'b00};  
34         MemoryInst[5] = {BLT, R2, R1, 16'd4, 2'b00};  
35         MemoryInst[6] = {ADD, R10, R2, R1, 14'd0};  
36         MemoryInst[7] = {PUSH, R1, 22'd0};  
37         MemoryInst[8] = {POP, R4, 22'd0};  
38         MemoryInst[9] = {ADD, R8, R4, R2, 14'd0};  
39  
40     end  
41  
42     // Get the instruction  
43     always @(posedge clk) begin  
44         instruction = MemoryInst[address];
```

```

45     $display("MemoryInst: clock = %b, pc_in = %d , Output = %0b", clk, address,
46             instruction);
47     end
48 endmodule

```

This testbench validates the Instruction Memory module's ability to output the correct instruction based on the given address. By simulating different address inputs and observing the corresponding instruction outputs, the testbench confirms that the module reliably retrieves the correct instruction for each address, a critical operation for instruction execution.

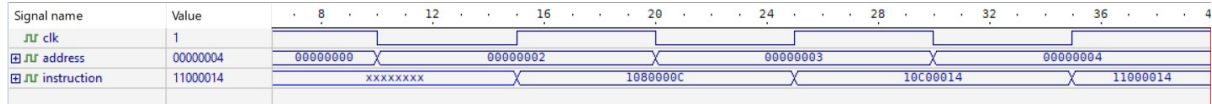


Figure 4.1: Instruction Memory Testbench

4.2.2 Data Memory

This module simulates a data memory unit in a processor. It handles read and write operations based on the input signals and addresses. The memory is segmented for data and stack based on the address range and 'SAddress'.

```

1 module Data_memory (
2     input clk,
3     input [31:0] address,
4     input MemWr,
5     input MemRd,
6     input [31:0] data_in,
7     input SAddress,
8     output reg [31:0] data_out
9 );
10
11 reg [31:0] memory [0:600];          //Assign 600 address location to the memory
12                                     // 500 for data (PC) and 100 for stack (SP)
13 always @(posedge clk) begin
14     // Condition for Write
15     if (MemWr && ((SAddress == 0 && address > 500) || (SAddress == 1 && address < 500))
16         ) begin
17         memory[address] = data_in;
18     end
19     // Condition for Read
20     if (MemRd && ((SAddress == 0 && address > 500) || (SAddress == 1 && address < 500))
21         ) begin
22         data_out = memory[address];
23     end
24     // Condition for wrong address
25     else if (MemRd) begin
26         data_out = 32'h00000000;
27     end
28     $display("DataMemory: clock = %b, address = %b , data_in = %d , MemWr = %b, MemRd =
29             %b, SAddress = %b,data_out = %d", clk, address, data_in ,MemWr,MemRd,SAddress,
30             data_out);
31 end
32 endmodule

```

The Data Memory module's testbench assesses its capability to handle read and write operations with respect to the given address, data input, and control signals. It tests various scenarios, including writing to and reading from different memory addresses, to ensure that the module correctly manages data storage and retrieval.

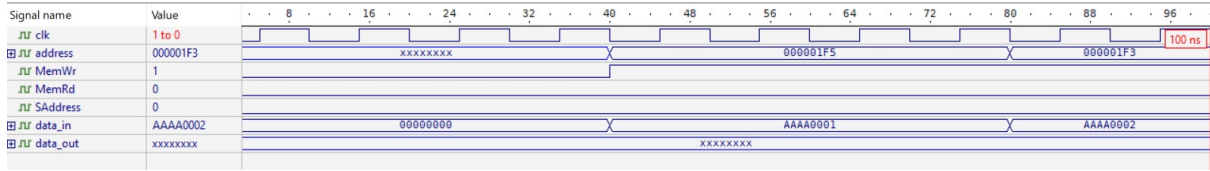


Figure 4.2: Data Memory Testbench

4.3 Register File

This module simulates a register file in a CPU, with read and write functionalities. It has two separate clock signals for reading and writing, input registers for read (RA, RB) and write (RW1, RW2), and buses (BusW1, BusW2, BusA, BusB) for data transfer.

```

1 module Registers_file(
2     input wire clk1,
3     input wire clk2,
4     input [3:0] RA, RB, RW1, RW2,
5     input [31:0] BusW1, BusW2,
6     input RegWr1, RegWr2,
7     output reg [31:0] BusA, BusB
8 );
9
10 reg [31:0] Reg [0:15]; // Register file with 16 registers
11
12
13 initial begin
14     for (int i = 0; i < 16; i = i + 1) begin
15         Reg[i] = 0; // Initialize all registers in the file to 0
16     end
17 end
18
19
20 always @(posedge clk1) begin
21     BusA = Reg[RA]; // Use non-blocking assignment (<=) for BusA
22     BusB = Reg[RB]; // Use non-blocking assignment (<=) for BusB
23     $display("READ Registers_file: clock = %b, RA_in = %d, RB_in = %d, RW1 = %d, RW2 = %d, BusW1 = %d, BusW2 = %d, BusA = %d, BusB = %d, Reg[RW1] = %d, Reg[RB] = %d", clk1, RA, RB, RW1, RW2, BusW1, BusW2, BusA, BusB, Reg[RW1], Reg[RB]);
24 end
25
26 always @(posedge clk2) begin
27     if (RegWr1 && RW1 != 0) begin
28         Reg[RW1] = BusW1; // Use non-blocking assignment (<=) to update the register
29     end
30
31     if (RegWr2 && RW2 != 0) begin
32         Reg[RW2] = BusW2; // Use non-blocking assignment (<=) to update the register
33     end
34
35     $display("WRITE Registers_file: clock = %b, RA_in = %d, RB_in = %d, RW1 = %d, RW2 = %d, BusW1 = %d, BusW2 = %d, BusA = %d, BusB = %d, Reg[RW1] = %d, Reg[RB] = %d", clk2, RA, RB, RW1, RW2, BusW1, BusW2, BusA, BusB, Reg[RW1], Reg[RB]);
36 end
37
38 endmodule

```

The testbench for the Registers File module ensures that the module accurately reads from and writes to the specified registers. It checks the module's ability to output the correct data from given register addresses and to update registers with provided data. This testbench is essential for verifying the register file's role in storing and providing CPU data.

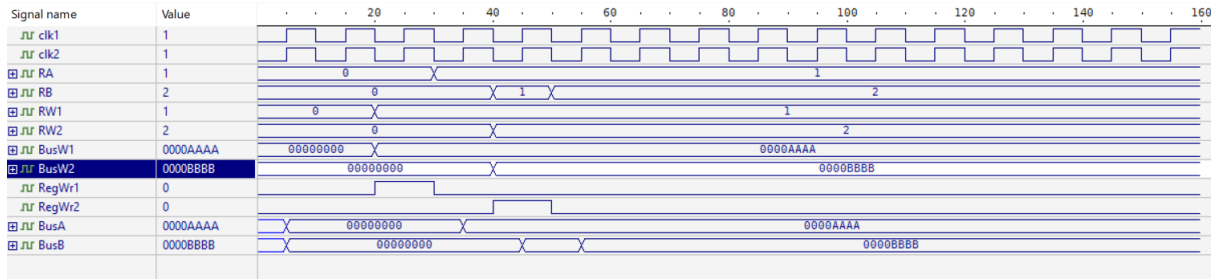


Figure 4.3: Register File Testbench

4.4 Arithmetic Logic Unit (ALU)

Performs arithmetic (addition, subtraction) and logical (AND) operations based on a 2-bit opcode. The ALU is the core of computational operations in a processor.

```

1 module ALU(
2     input [31:0] A, B,           // 32-bit inputs
3     input [1:0] Op,             // 2-bit operation code
4     output reg [31:0] Result,    // 32-bit result
5     output Zero, Sign           // Zero and Sign flags
6 );
7
8 // Define operation codes
9 parameter AND_OP = 2'b00,
10            ADD_OP = 2'b01,
11            SUB_OP = 2'b10;
12
13 // Perform the operation based on the operation code
14 always @(A, B, Op) begin
15     case (Op)
16         AND_OP: Result = A & B;
17         ADD_OP: Result = A + B;
18         SUB_OP: Result = A - B;
19         default: Result = 32'b0;
20     endcase
21 end
22
23 // Set Zero and Sign flags
24 assign Zero = (Result == 32'b0) ? 1'b1 : 1'b0;
25 assign Sign = Result[31]; // MSB is the sign bit for 2's complement numbers
26
27 endmodule

```

ALU_tb runs several scenarios, testing each operation (ADD, SUB, AND) and the correct setting of Zero and Sign flags, ensuring the ALU responds correctly to different inputs and operations.

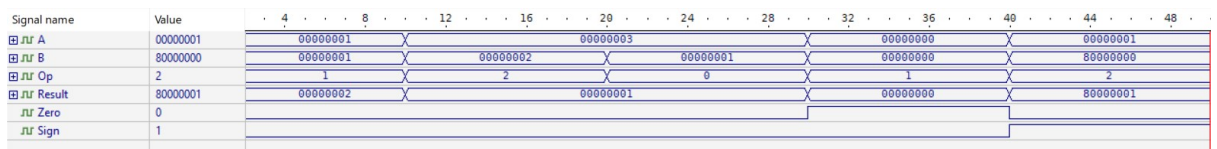


Figure 4.4: ALU Testbench

4.5 Control Unit

This module is the heart of the processor's control system. It generates control signals based on the opcode to control the operation of other components in the processor.

```
1 module Control_unit(  
2     input [5:0] Op,  
3     input zerof, signf,  
4     output reg RegS, RegWr1, RegWr2, ExtOp, ALUSrc, Rdata, SAddress, MemRd, MemWr,  
5         PopSrc, WBdata,  
6     output reg [1:0] StaS, ALUOp,  
7     output reg [1:0] PCSrc  
8 );  
9  
10    // R-Type Instructions  
11    reg AND = 0, ADD = 0, SUB = 0;  
12    // I-Type Instructions  
13    reg ANDI = 0, ADDI = 0, LW = 0, LWPOL = 0, SW = 0, BGT = 0, BLT = 0, BEQ = 0, BNE =  
14        0;  
15    // J-Type Instructions  
16    reg JMP = 0, CALL = 0, RET = 0;  
17    // S-Type Instructions  
18    reg PUSH = 0, POP = 0;  
19  
20    always @(*) begin  
21        // Default values for control signals  
22        RegS = 0; RegWr1 = 0; RegWr2 = 0; ExtOp = 0; ALUSrc = 0;  
23        Rdata = 0; SAddress = 0; MemRd = 0; MemWr = 0; PopSrc = 0;  
24        WBdata = 0;  
25  
26        // Default value for StaS, ALUOp, and PCSrc  
27        StaS = 2'b00;  
28        ALUOp = 2'b00;  
29        PCSrc = 2'b00;  
30  
31        // Reset all control signal assignments to zero  
32        AND = 0; ADD = 0; SUB = 0;  
33        ANDI = 0; ADDI = 0; LW = 0; LWPOL = 0; SW = 0; BGT = 0; BLT = 0; BEQ = 0; BNE =  
34            0;  
35        JMP = 0; CALL = 0; RET = 0;  
36        PUSH = 0; POP = 0;  
37  
38        case (Op)  
39            6'b000000: AND = 1;  
40            6'b000001: ADD = 1;  
41            6'b000010: SUB = 1;  
42            6'b000011: ANDI = 1;  
43            6'b000100: ADDI = 1;  
44            6'b000101: LW = 1;  
45            6'b000110: LWPOL = 1;  
46            6'b000111: SW = 1;  
47            6'b001000: BGT = 1;  
48            6'b001001: BLT = 1;  
49            6'b001010: BEQ = 1;  
50            6'b001011: BNE = 1;  
51            6'b001100: JMP = 1;  
52            6'b001101: CALL = 1;  
53            6'b001110: RET = 1;  
54            6'b001111: PUSH = 1;  
55            6'b010000: POP = 1;  
56            default: begin // Set default values to 0  
57                AND = 0; ADD = 0; SUB = 0;  
58                ANDI = 0; ADDI = 0; LW = 0; LWPOL = 0; SW = 0; BGT = 0; BLT = 0; BEQ =  
59                    0; BNE = 0;  
60                JMP = 0; CALL = 0; RET = 0;  
61                PUSH = 0; POP = 0;  
62            end  
63        endcase  
64  
65        // Additional logic for control signal assignments  
66        RegS = SW || BGT || BLT || BEQ || BNE || PUSH;  
67        RegWr1 = AND || ADD || SUB || ANDI || ADDI || LW || LWPOL || POP;
```

```

64     RegWr2 = LWPOL;
65     ExtOp = ADDI || LW || LWPOL || SW || BGT || BLT || BEQ || BNE;
66     ALUSrc = ANDI || ADDI || LW || LWPOL || SW;
67     Rdata = SW || PUSH;
68     SAddress = LW || LWPOL || SW;
69     MemRd = LW || LWPOL || RET || POP;
70     MemWr = SW || CALL || PUSH;
71     WBdata = LW || LWPOL || RET || POP;
72
73     // Logic for StaS assignment
74     if (RET || POP) begin
75         StaS = 2'b01; // Assign StaS to 01
76     end else if (CALL || PUSH) begin
77         StaS = 2'b10; // Assign StaS to 10
78     end
79
80     // Additional logic for ALUOp assignment
81     if (ADD || ADDI || LW || LWPOL || SW) begin
82         ALUOp = 2'b01; // Assign ALUOp to 01
83     end else if (SUB || BGT || BLT || BEQ || BNE) begin
84         ALUOp = 2'b10; // Assign ALUOp to 10
85     end
86
87     // Additional logic for PCSrc assignment
88     case (Op)
89         // BGT
90         6'b001000:
91             if (zerof == 0 && signif == 1) begin
92                 PCSrc = 2'b10;
93             end
94             else begin
95                 PCSrc = 2'b00;
96             end
97
98         // BLT
99         6'b001001:
100             if (zerof == 0 && signif == 0) begin
101                 PCSrc = 2'b10;
102             end
103             else begin
104                 PCSrc = 2'b00;
105             end
106
107         // BEQ
108         6'b001010:
109             if (zerof == 1) begin
110                 PCSrc = 2'b10;
111             end
112             else begin
113                 PCSrc = 2'b00;
114             end
115
116         // BNE
117         6'b001011:
118             if (zerof == 0) begin
119                 PCSrc = 2'b10;
120             end
121             else begin
122                 PCSrc = 2'b00;
123             end
124
125         6'b001100: PCSrc = 2'b01; // JMP
126         6'b001101: PCSrc = 2'b01; // CALL
127         6'b001110: PCSrc = 2'b11; // RET
128         default:
129             PCSrc = 2'b00;
130     endcase
131 end
132 endmodule

```

This testbench verifies the Control Unit module's functionality in generating the correct control signals based on various operation codes. It tests a range of instructions to ensure

that the control unit appropriately sets the control signals for each type of operation, confirming its critical role in directing the processor's actions.

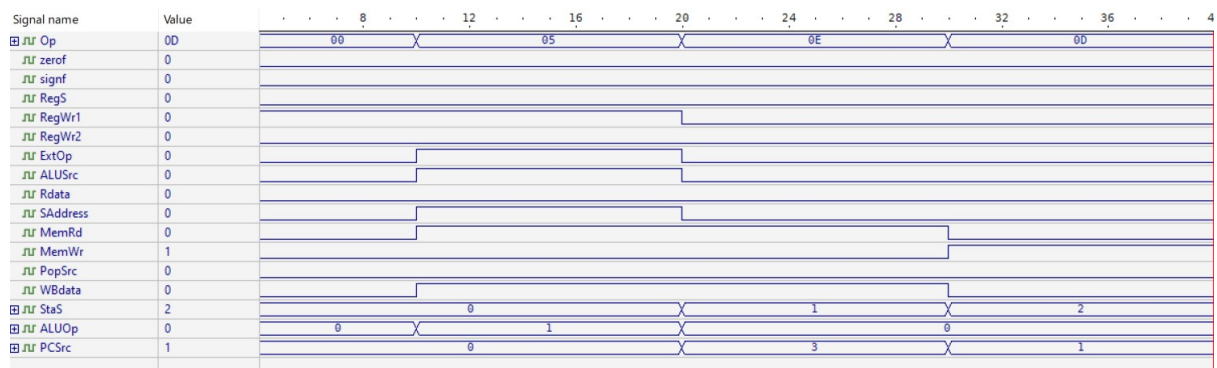


Figure 4.5: Control Unit Testbench

4.6 Clock Organizer

This module is responsible for organizing the clock signals for different stages of the processor's operation, such as instruction fetch, decode, execute, memory access, and write-back.

```

1 module Clock_organizer(
2     input clock,
3     input [5:0] Op,
4     output reg IF, ID, MEM, EX, WB, IFF
5 );
6
7     // Opcode Parameters
8     parameter AND = 6'b000000, ADD = 6'b000001, SUB = 6'b000010;
9     parameter ANDI = 6'b000011, ADDI = 6'b000100, LW = 6'b000101, LWPOL = 6'b000110,
10     SW = 6'b000111, BGT = 6'b001000, BLT = 6'b001001, BEQ = 6'b001010, BNE =
11     6'b001011;
12     parameter JMP = 6'b001100, CALL = 6'b001101, RET = 6'b001110;
13     parameter PUSH = 6'b001111, POP = 6'b010000 ;
14
15     // Internal State
16     reg [2:0] state;
17     reg en = 0;
18     // Initialization block
19     initial begin
20         state = 3'b000; // Initialize state
21         IF = 1; // Initialize all output signals
22         ID = 0;
23         MEM = 0;
24         EX = 0;
25         WB = 0;
26         IFF <= 0;
27     end
28
29     // State Machine
30     always @(posedge clock) begin
31         case (state)
32             3'b000: begin
33                 state <= 3'b001;
34                 IF <= 0;
35                 ID <= 1;
36                 EX <= 0;
37                 MEM <= 0;
38                 WB <= 0;
39                 IFF <= 0;
40             end

```

```

41     3'b001: begin
42         if (Op == JMP) begin
43             state <= 3'b000;
44             IFF <= 1;
45         #2ns IF <= 1;
46             ID <= 0;
47             EX <= 0;
48             MEM <= 0;
49             WB <= 0;
50
51         end
52         else if (Op == CALL || Op == RET || Op == PUSH || Op == POP) begin
53             state <= 3'b011;
54             IF <= 0;
55             ID <= 0;
56             EX <= 0;
57             MEM <= 1;
58             WB <= 0;
59         end
60         else begin
61             state <= 3'b010;
62             IF <= 0;
63             ID <= 0;
64             EX <= 1;
65             MEM <= 0;
66             WB <= 0;
67         end
68     end
69
70     3'b010: begin
71         if (Op == BNE || Op == BEQ || Op == BGT || Op == BLT) begin
72             state <= 3'b000;
73             IFF <= 1;
74         #2ns IF <= 1;
75             ID <= 0;
76             EX <= 0;
77             MEM <= 0;
78             WB <= 0;
79
80         end
81
82         else if (Op == LW || Op == LWPOL || Op == SW) begin
83             state <= 3'b011;
84             IF <= 0;
85             ID <= 0;
86             EX <= 0;
87             MEM <= 1;
88             WB <= 0;
89         end
90         else begin
91             state <= 3'b100;
92             IF <= 0;
93             ID <= 0;
94             EX <= 0;
95             MEM <= 0;
96             WB <= 1;
97         end
98     end
99
100    3'b011: begin
101        if (Op == LW || Op == LWPOL || Op == POP) begin
102            state <= 3'b100;
103            IF <= 0;
104            ID <= 0;
105            EX <= 0;
106            MEM <= 0;
107            WB <= 1;
108        end
109        else begin
110            state <= 3'b000;
111            IFF <= 1;
112        #2ns IF <= 1;
113            ID <= 0;

```

```

114         EX <= 0;
115         MEM <= 0;
116         WB <= 0;
117
118     end
119 end
120
121     3'b100: begin
122         state <= 3'b000;
123         IFF <= 1;
124         #2ns IF <= 1;
125         ID <= 0;
126         EX <= 0;
127         MEM <= 0;
128         WB <= 0;
129
130     end
131
132 endcase
133 end
134 endmodule

```

The Clock Organizer module's testbench is focused on confirming the correct sequencing and timing of operations based on the operation codes. By simulating various instruction types, the testbench checks whether the module appropriately sets the clock signals for each stage of the instruction cycle, validating its role in synchronizing CPU operations.

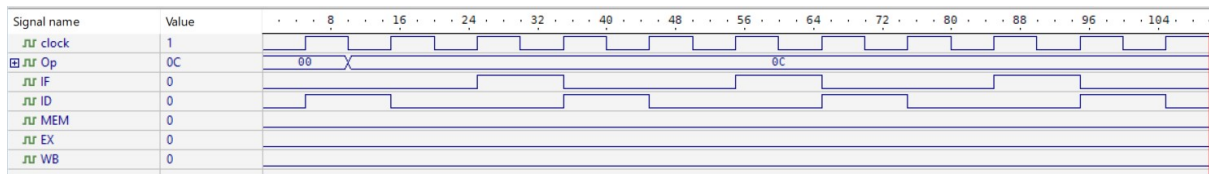


Figure 4.6: Clock Organizer Testbench

4.7 Special Purpose Registers

4.7.1 Program Counter (PC)

The PC module is designed to store and update a 32-bit address, typically used in a processor to keep track of the address of the next instruction to execute.

```

1 module PC(
2     input wire [31:0] input_data,
3     output wire [31:0] output_data
4 );
5
6     reg [31:0] register = 32'b0; // Initialize register to zero
7
8     always @(input_data) begin
9         register <= input_data; // Load input data when it changes
10    end
11
12    assign output_data = register;
13
14 endmodule

```

The testbench for the Program Counter (PC) module checks the module's ability to store and update its internal register based on the input data. It sequentially inputs different 32-bit values and verifies if the output matches the input after a clock cycle. This ensures that the PC correctly updates its value to point to the next instruction in a sequence.



Signal name	Value	4	8	12	16	20	24	28
 <code>input_data</code>	80000000	12345678	X	ABCDEF01	X	80000000		
 <code>output_data</code>	80000000	12345678	X	ABCDEF01	X	80000000		

Figure 4.7: PC Testbench

4.7.2 Stack Pointer (SP)

The SP module functions similarly to the PC but is used to track the top of the stack in memory. It's a crucial component for stack operations in many CPU architectures.

```

1 module SP(
2     input wire [31:0] input_data,
3     output wire [31:0] output_data
4 );
5
6     reg [31:0] register1 = 32'd500; // Initialize register to 500
7
8     always @(input_data) begin
9         register1 <= input_data; // Load input data when it changes
10    end
11
12    assign output_data = register1;
13
14 endmodule

```

The Stack Pointer (SP) module's testbench is designed to validate the module's updating mechanism. Similar to the PC testbench, it provides different 32-bit values and checks if the SP correctly updates its internal register. The testbench ensures that the SP accurately maintains the stack's top address, crucial for stack operations in a CPU.



Signal name	Value	4	8	12	16	20	24	28
 <code>input_data</code>	80000000	12345678	X	ABCDEF01	X	80000000		
 <code>output_data</code>	80000000	12345678	X	ABCDEF01	X	80000000		

Figure 4.8: SP Testbench

4.8 Extender

This module performs either sign extension or zero extension on a 16-bit input to produce a 32-bit output. This is important in systems where different bit-widths are integrated.

```

1 module Extender(
2     input wire [15:0] in, // 16-bit input
3     output wire [31:0] out, // 32-bit output
4     input wire is_signed // Parameter for signed/unsigned extension
5 );
6
7     // Perform sign extension or zero extension based on is_signed
8     assign out = is_signed ? {{16{in[15]}}, in} : {16'b0, in};
9
10 endmodule

```

Extender_tb tests both sign extension and zero extension, verifying

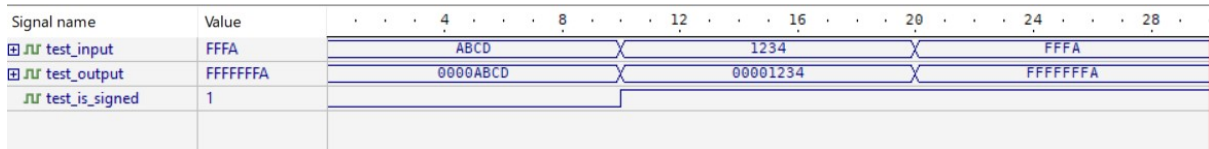


Figure 4.9: Extender Testbench

4.9 Multiplexer

4.9.1 Multiplexer 4x1

This module selects one of four 32-bit inputs based on a 2-bit select signal. It's more versatile for larger data selections.

```

1 module Mux4x1 #(parameter WIDTH = 32) (
2     input wire [1:0] select,
3     input wire [WIDTH-1:0] I0, I1, I2, I3,
4     output wire [WIDTH-1:0] out
5 );
6
7     assign out = (select[1]) ? ((select[0]) ? I3 : I2) : ((select[0]) ? I1 : I0);
8 endmodule

```

Mux4x1_tb tests all possible select signal combinations to ensure the correct input is selected each time.

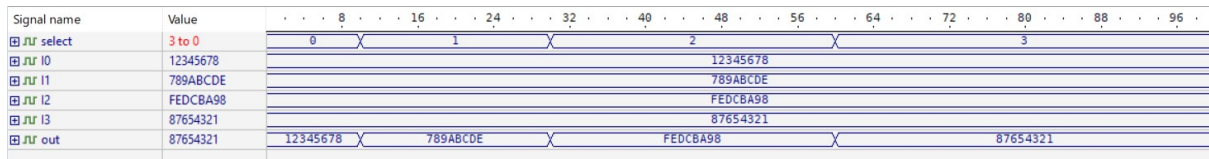


Figure 4.10: Mux4x2 Testbench

4.9.2 Multiplexer 2x1

This module selects one of two 32-bit inputs based on a single-bit select signal. Multiplexers are fundamental in digital circuits for routing signals.

```

1 module Mux2x1 #(parameter WIDTH = 32) (
2     input wire select,
3     input wire [WIDTH-1:0] I0, I1,
4     output wire [WIDTH-1:0] out
5 );
6
7     // Selects I1 if select is true (1), otherwise selects I0.
8     assign out = (select) ? I1 : I0;
9
10 endmodule

```

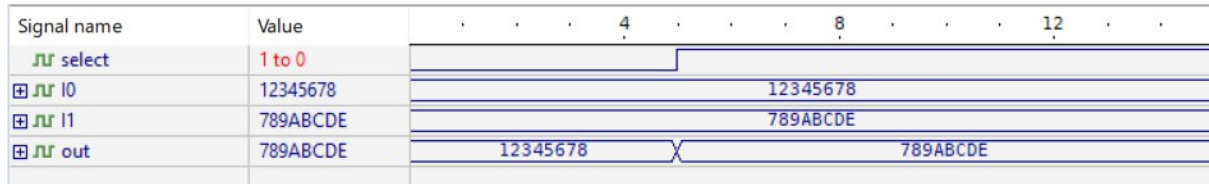


Figure 4.11: Mux2x1 Testbench

4.10 Adder

4.10.1 One Input Adder

Increments a 32-bit number by 1. Useful in counters and loop control.

```

1 module Adder_1(
2     input [31:0] A,
3     output [31:0] out
4 );
5     assign out = A + 1; // Adds 1 to input A and assigns the result to out
6 endmodule

```

Adder_1_tb verifies that the module correctly increments the input value by 1.

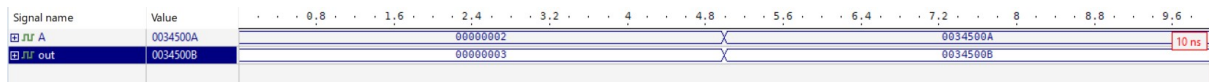


Figure 4.12: One Input Adder Testbench

4.10.2 Two Input Adder

Adds two 32-bit numbers. Adders are critical in arithmetic operations in processors.

```

1 module Adder_2(
2     input [31:0] A, B,
3     output [31:0] out
4 );
5     assign out = A + B; // Adds inputs A and B and assigns the result to out
6 endmodule

```

Adder_2_tb tests the addition of different pairs of numbers, verifying the module correctly adds 32-bit values.

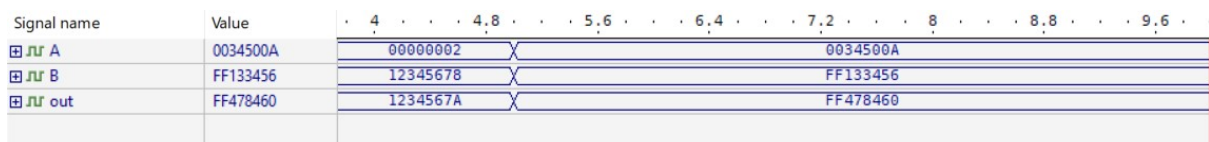


Figure 4.13: Two Input Adder Testbench

4.11 Subtractor

Decrements a 32-bit number by 1. Similar to the adder but for decrementing.

```

1 module Subtractor_1(
2     input [31:0] A,
3     output [31:0] out
4 );
5     assign out = A - 1; // Subtracts input B from input A and assigns the result to out
6 endmodule

```

Signal name	Value	4	4.8	5.6	6.4	7.2	8	8.8	9.6
A	FF133456	12345678						FF133456	
out	FF133455	12345677						FF133455	

Figure 4.14: Subtractor Testbench

4.12 Concatenation

This module takes two inputs, A (26 bits) and B (6 bits), and concatenates them to form a 32-bit output. It's used in systems where bits from different sources need to be combined into a single word.

```

1 module Concatenator(
2     input [25:0] A, // 26-bit input
3     input [5:0] B, // 6-bit input
4     output [31:0] out // 32-bit output
5 );
6     assign out = {B, A}; // Concatenates B (6-bit) and A (26-bit) to form a 32-bit output
7 endmodule

```

The Concatenator_tb sets specific values to A and B, and initiates the simulation. It verifies that the concatenation of A and B produces the correct 32-bit output.

Signal name	Value	400	800	1200	1600	2000	2400	2800	3200	3600	4000	4400	4800
A	3CDEF12						3CDEF12						
B	3A						3A						
out	EBCDEF12						EBCDEF12						

Figure 4.15: Concatenation Testbench

4.13 All System

To test the entire system, we need to create a comprehensive test bench and validate the correctness of its data. In this context, a specific test bench was developed and provided for the instruction fetch stage to initiate the testing process.

```

1 // First code sequences in the ISA
2 ADDI R1, R0, 7; //R1=7
3 ADDI R2, R0, 2; //R2=2
4 CALL F; // GO TO LABEL F
5
6 PUSH R2; // put 2 to stack
7 POP R11; // pop 2 to R11
8 ADD R12, R11, R2; //R12 = 2+2 = 4
9 JMP END;
10 F:
11 SUB R3, R1, R2; //R3 = 7-2 = 5
12 ADD R4, R3, R1; //R4 = 5+7 = 12
13 AND R5, R1, R4; // R5 = 0...00111 AND 0...01100 =0...00100 = 4
14 ANDI R6, R1, 12; // same above R6 = 4
15 RET; //
16 END:

//code test 1
MemoryInst[0] = {ADDI, R1, R0, 16'd7, 2'b00}; //R1=7
MemoryInst[1] = {ADDI, R2, R0, 16'd2, 2'b00}; //R2=2
MemoryInst[2] = {CALL, 26'd7}; // GO TO LABEL F IN POSTION 6 IN PC

MemoryInst[3] = {PUSH, R2, 22'd0}; // put 2 to stack
MemoryInst[4] = {POP, R11, 22'd0}; // pop 2 to R11
MemoryInst[5] = {ADD, R12, R11, R2, 14'd0}; //R12 = 2+2 = 4
MemoryInst[6] = {JMP, 26'd12}; // GO TO LABEL EN
// LAPEL F:
MemoryInst[7] = {SUB, R3, R1, R2, 14'd0}; //R3 = 7-2 = 5
MemoryInst[8] = {ADD, R4, R3, R1, 14'd0}; //R4 = 5+7 = 12
MemoryInst[9] = {AND, R5, R1, R4, 14'd0}; // R5 = 0...00111 AND 0...01100 =0...00100 = 4
MemoryInst[10] = {ANDI, R6, R1, 16'd12, 2'b00}; // same above R6 = 4
MemoryInst[11] = {RET, 26'd0}; // back to line after call

// LAPEL END

```

Figure 4.16: First code sequences in the ISA

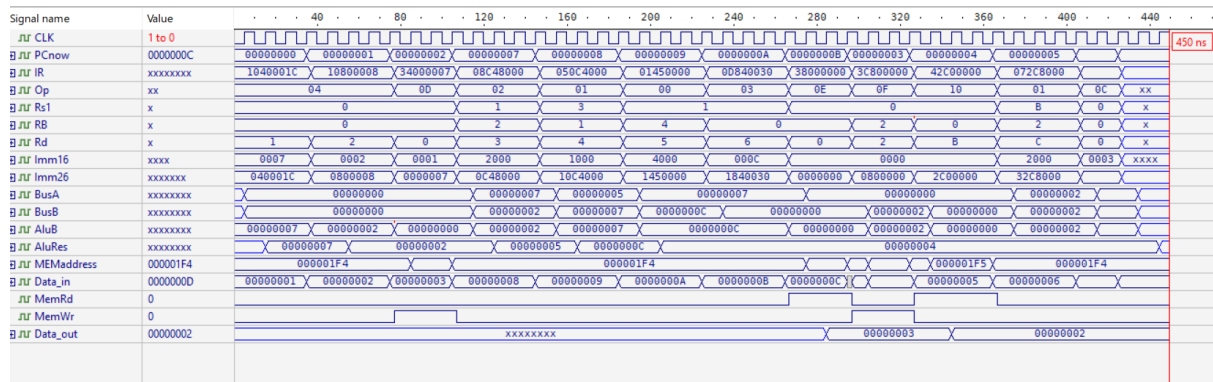


Figure 4.17: Waveform for first code


```

//code test 2
MemoryInst[0] = {ADDI, R1, R0, 16'd7,2'b00}; //R1=7
MemoryInst[1] = {ADDI, R2, R0, 16'd2,2'b00}; //R2=2
MemoryInst[2] = {ADDI, R5, R0, 16'd4,2'b00}; //R5=4

MemoryInst[3] = {JMP, 26'd8}; // go to label L

// LAPEL PUS
MemoryInst[4] = {LWPOL, R7, R2, 16'd4,2'b00}; // R7 <--mem[R2+4] R7 must = 7 and R2=R2+1
MemoryInst[5] = {LW, R8, R2, 16'd4,2'b00}; // R8 <--mem[R2+4] R8 must = 4
MemoryInst[6] = {ADD, R10,R7, R8, 14'd0}; // R10 = 7+4 =11
MemoryInst[7] = {JMP, 26'd12}; // GO TO LABEL EN

// LAPEL L
MemoryInst[8] = {SW, R1, R2, 16'd4,2'b00}; //store R1 =7 in postion mem[R2+4]
MemoryInst[9] = {SW, R5, R2, 16'd5,2'b00}; //store R5 =4 in postion mem[R2+5]
MemoryInst[10] = {BEQ, R5, R2, 16'd2,2'b00}; // go to label END
MemoryInst[11] = {BGT, R5, R2, 16'b111111111111001,2'b00}; // got to label PUS

// LAPEL END

```

Figure 4.20: Second code sequences in the ISA

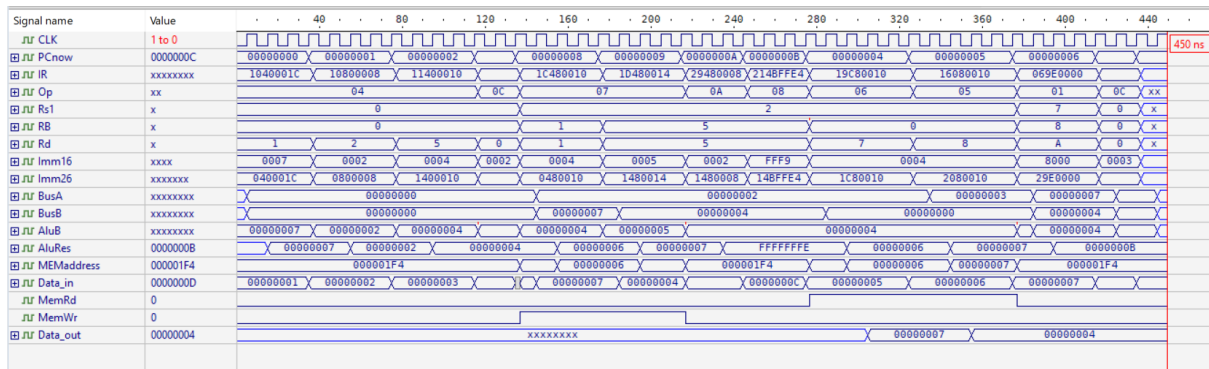


Figure 4.21: Waveform for Second code

```

# KERNEL: MemoryInst:      clock = 1, pc_in = 0 , Output = 0001000001000000000000000000000011100
# KERNEL: READ Registers_file: clock = 1, RA_in = 0, RB_in = 0, RW1 = 1, RW2 = 0, BusW1 = x, BusW2 = x, BusA = 0, BusB = 0
# KERNEL: ALU:              clock = 1, in1 = 0, in2 = 7 ALU_Result = 7, Zero = 0 , Sign = 0
# KERNEL: WRITE Registers_file: clock = 1, RA_in = 0, RB_in = 0, RW1 = 1, RW2 = 0, BusW1 = 7, BusW2 = 1, BusA = 0, BusB = 0
# KERNEL: MemoryInst:      clock = 1, pc_in = 1 , Output = 000100001000000000000000000000001000
# KERNEL: READ Registers_file: clock = 1, RA_in = 0, RB_in = 0, RW1 = 2, RW2 = 0, BusW1 = 7, BusW2 = 1, BusA = 0, BusB = 0
# KERNEL: ALU:              clock = 1, in1 = 0, in2 = 2 ALU_Result = 2, Zero = 0 , Sign = 0
# KERNEL: WRITE Registers_file: clock = 1, RA_in = 0, RB_in = 0, RW1 = 2, RW2 = 0, BusW1 = 2, BusW2 = 1, BusA = 0, BusB = 0
# KERNEL: MemoryInst:      clock = 1, pc_in = 2 , Output = 000100010100000000000000000000001000
# KERNEL: READ Registers_file: clock = 1, RA_in = 0, RB_in = 0, RW1 = 5, RW2 = 0, BusW1 = 2, BusW2 = 1, BusA = 0, BusB = 0
# KERNEL: ALU:              clock = 1, in1 = 0, in2 = 4 ALU_Result = 4, Zero = 0 , Sign = 0
# KERNEL: WRITE Registers_file: clock = 1, RA_in = 0, RB_in = 0, RW1 = 5, RW2 = 0, BusW1 = 4, BusW2 = 1, BusA = 0, BusB = 0
# KERNEL: MemoryInst:      clock = 1, pc_in = 3 , Output = 001100000000000000000000000000001000
# KERNEL: READ Registers_file: clock = 1, RA_in = 0, RB_in = 0, RW1 = 0, RW2 = 0, BusW1 = 4, BusW2 = 1, BusA = 0, BusB = 0
# KERNEL: MemoryInst:      clock = 1, pc_in = 8 , Output = 000110001001000000000000000000001000
# KERNEL: READ Registers_file: clock = 1, RA_in = 2, RB_in = 1, RW1 = 1, RW2 = 2, BusW1 = 4, BusW2 = 1, BusA = 2, BusB = 7
# KERNEL: ALU:              clock = 1, in1 = 2, in2 = 4 ALU_Result = 6, Zero = 0 , Sign = 0
# KERNEL: DataMemory:      clock = 1, address = 00000000000000000000000000000000110 , data_in = 7 , MemWr = 1, MemRd = 0 ,data_out = x
# KERNEL: MemoryInst:      clock = 1, pc_in = 9 , Output = 00011010100100000000000000000010100
# KERNEL: READ Registers_file: clock = 1, RA_in = 2, RB_in = 5, RW1 = 5, RW2 = 2, BusW1 = 6, BusW2 = 3, BusA = 2, BusB = 4
# KERNEL: ALU:              clock = 1, in1 = 2, in2 = 5 ALU_Result = 7, Zero = 0 , Sign = 0
# KERNEL: DataMemory:      clock = 1, address = 00000000000000000000000000000000111 , data_in = 4 , MemWr = 1, MemRd = 0 ,data_out = x
# KERNEL: MemoryInst:      clock = 1, pc_in = 10 , Output = 0010100101001000000000000000001000
# KERNEL: READ Registers_file: clock = 1, RA_in = 2, RB_in = 5, RW1 = 5, RW2 = 2, BusW1 = 7, BusW2 = 3, BusA = 2, BusB = 4
# KERNEL: ALU:              clock = 1, in1 = 2, in2 = 4 ALU_Result = 4294967294, Zero = 0 , Sign = 1
# KERNEL: MemoryInst:      clock = 1, pc_in = 11 , Output = 001000010100101111111111111100100

```

Figure 4.22: The output-1 for second code

```

# KERNEL: READ Registers_file: clock = 1, RA_in = 2, RB_in = 5, RW1 = 5, RW2 = 2, BusW1 = 4294967294, BusW2 = 3, BusA = 2, BusB = 4
# KERNEL: ALU:              clock = 1, in1 = 2, in2 = 4 ALU_Result = 4294967294, Zero = 0 , Sign = 1
# KERNEL: MemoryInst:      clock = 1, pc_in = 4 , Output = 0001100110010000000000000000001000
# KERNEL: READ Registers_file: clock = 1, RA_in = 2, RB_in = 0, RW1 = 7, RW2 = 2, BusW1 = 3, BusA = 2, BusB = 0
# KERNEL: ALU:              clock = 1, in1 = 2, in2 = 4 ALU_Result = 6, Zero = 0 , Sign = 0
# KERNEL: DataMemory:      clock = 1, address = 00000000000000000000000000000000110 , data_in = 5 , MemWr = 0, MemRd = 1 ,data_out = 7
# KERNEL: WRITE Registers_file: clock = 1, RA_in = 2, RB_in = 0, RW1 = 7, RW2 = 2, BusW1 = 7, BusW2 = 3, BusA = 2, BusB = 0
# KERNEL: MemoryInst:      clock = 1, pc_in = 5 , Output = 0001011000001000000000000000001000
# KERNEL: READ Registers_file: clock = 1, RA_in = 2, RB_in = 0, RW1 = 8, RW2 = 2, BusW1 = 7, BusW2 = 3, BusA = 3, BusB = 0
# KERNEL: ALU:              clock = 1, in1 = 3, in2 = 4 ALU_Result = 7, Zero = 0 , Sign = 0
# KERNEL: DataMemory:      clock = 1, address = 00000000000000000000000000000000111 , data_in = 6 , MemWr = 0, MemRd = 1 ,data_out = 4
# KERNEL: WRITE Registers_file: clock = 1, RA_in = 2, RB_in = 0, RW1 = 8, RW2 = 2, BusW1 = 4, BusW2 = 4, BusA = 3, BusB = 0
# KERNEL: MemoryInst:      clock = 1, pc_in = 6 , Output = 0000011010011110000000000000000000
# KERNEL: READ Registers_file: clock = 1, RA_in = 7, RB_in = 8, RW1 = 10, RW2 = 7, BusW1 = 7, BusW2 = 4, BusA = 7, BusB = 4
# KERNEL: ALU:              clock = 1, in1 = 7, in2 = 4 ALU_Result = 11, Zero = 0 , Sign = 0
# RUNTIME: Info: RUNTIME_0068 code.v (1153): $finish called.

```

Figure 4.23: The output-2 for second code

Chapter 5

Teamwork, Conclusion and Future Work

5.1 Teamwork

In our team, we embraced a collaborative approach to design and build various components of our project. Initially, we worked together to develop the data paths, signals, state diagram, and tables. This joint effort laid a strong foundation for our work. Subsequently, we divided the tasks of writing component codes and testing benches among ourselves. Tariq was responsible for the Data Memory, Extender, Program Counter (PC), and Stack Pointer (SP). Wasim handled the Instruction Memory, Register File, and Subtractor, while Eyab took charge of the Arithmetic Logic Unit (ALU), Multiplexer, Adder, and Concatenation. Following this, we reconvened to collectively work on the Control Unit and Clock Organizer, ensuring these integral parts received our united focus. In the final stage, we interconnected all the parts, culminating in a design and final code that was a product of our united efforts and presence. Moreover, the creation of the project report was a collaborative effort as well, with each of us contributing our insights and findings. This project was not only a testament to the strength of teamwork in technical development but also in effective communication and documentation, exemplifying our shared responsibility and unity throughout the entire process.

5.2 Conclusion and Future Work

The Multi-Cycle Processor architecture represents a significant advancement in processing efficiency and flexibility. By dividing complex instructions into multiple cycles, it optimizes hardware use and speeds up instruction execution. The potential addition of pipelining, which allows parallel execution stages, promises even greater efficiency. However, this introduces certain hazards, necessitating the use of techniques like forwarding and branch prediction for effective management.

Through this project, we have gained a deeper understanding of the MIPS CPU, particularly in terms of module connectivity and testing. Looking forward, next step involves refining the architecture with a pipeline approach, aiming to further reduce the cycles per instruction (CPI) and enhance overall CPU performance. This future endeavor will leverage our learned insights and contribute to the ongoing evolution of processor design.

Chapter 6

Appendix

All the codes are attached in the ZIP file.

