



Department of Electrical & Computer Engineering
ENCS4370 - Computer Architecture

Pipelined RISC Processor Design

Prepared by: Ameer Rabie 1211226
Mumen Anbar 1212297
Ghazi Haj Qassem 1210778
Date: June 17, 2025

Abstract

This project presents the design and implementation of a 16-bit instruction set architecture (ISA) and a 5-stage pipelined processor. The ISA includes four instruction types—R-type, I-type, J-type, and S-type—each with specific formats and a shared opcode field. The processor features eight 16-bit general-purpose registers, with R0 hardwired to zero, a 16-bit program counter (PC), and separate data and instruction memories that are byte-addressable and use little-endian byte ordering. The ALU generates signals to calculate branch outcomes, including zero, carry, and overflow flags.

The processor supports a reduced set of instructions to simplify the design while retaining sufficient complexity for meaningful programs. Key instructions include logical operations (AND, OR), arithmetic operations (ADD, SUB), memory access (LW, SW), and control flow (JMP, CALL, RET, conditional branches).

The design process involved creating a detailed datapath and control path, with block diagrams and truth tables for the control signals. A five-stage pipeline was implemented to enhance performance, comprising fetch, decode, ALU, memory access, and write-back stages. Verification included developing a comprehensive testbench and multiple code sequences to ensure the processor's correctness and completeness.

Simulation results confirm that the processor accurately executes the instruction set, with snapshots of the simulator validating the implementation. The project demonstrates a systematic approach to processor design, emphasizing correctness, completeness, and efficient execution of the ISA.

Contents

1	Theoretical Background	1
1.1	Set of Operations	1
1.1.1	Arithmetic Operations	1
1.1.2	Logical Operations	1
1.1.3	Memory Access Operations	1
1.1.4	Control Flow Operations	2
1.2	Instruction Set Architecture (ISA)	3
1.2.1	R-type Instructions	3
1.2.2	I-type Instructions	3
1.2.3	J-type Instructions	3
1.2.4	S-type Instructions	3
1.3	Processor Architecture	3
1.4	Pipeline Stages	4
1.5	Control Path	4
1.6	ALU and Condition Branch Signals	4
1.7	Verification	4
2	Procedure	5
2.1	RTL Design	5
2.2	Single Cycle Datapath	8
2.2.1	Instruction Fetch (IF)	8
2.2.2	Instruction Decode (ID)	9
2.2.3	Execution (EX)	9
2.2.4	Memory Access (MEM)	9
2.2.5	Write Back (WB)	9
2.2.6	Control Unit	9
2.2.7	Branch and Jump Handling	9
2.3	Single Cycle Control Units	10
2.3.1	Main Control Signals	10
2.3.2	ALU Control	13
2.3.3	PC Control	13
2.4	Pipelining	15
2.4.1	Datapath	15
2.4.2	Instruction Fetch (IF)	16
2.4.3	Instruction Decode (ID)	18
2.4.4	Execution (EX)	21
2.4.5	Memory Access (MEM)	23
2.4.6	Write Back (WB)	25
2.5	Pipeline Control Units	26
2.5.1	PC Control Unit	26
2.5.2	Hazard Control Unit	26
3	Verification	27
3.1	Tested Assembly Code Sequence	27
3.2	Test Bench Code	30
3.3	Simulation Results	31

3.3.1	Testcase #1	31
3.3.2	Testcase #2	36
3.3.3	Testcase #3	38
3.4	Analysis of Results	40
3.4.1	Testcase #1	40
3.4.2	Testcase #2	43
3.4.3	Testcase #3	46
4	Teamwork	48
5	Conclusion	49

List of Figures

2.1	Single-Cycle Data Path of the 16-bit Processor	8
2.2	Pipeline Data Path of the 16-bit Processor	15
2.3	Pipeline Data Path Instruction Fetch Stage	16
2.4	Pipeline Data Path Instruction Decode Stage	18
2.5	Pipeline Data Path Execution Stage	21
2.6	Pipeline Data Path Memory Stage	23
2.7	Pipeline Data Path Write Back Stage	25
3.1	Simulation Console Output	31
3.2	Waveform Results for the first part of the simulation (0-24 ns)	31
3.3	Data memory contents for the first part of the simulation (0-24 ns))	32
3.4	Waveform Results for the second part of the simulation (24-50 ns)	32
3.5	Data memory contents for the second part of the simulation (24-50 ns)	33
3.6	Waveform Results for the second part of the simulation (48-74) ns)	33
3.7	Data memory contents for the second part of the simulation (48-74) ns)	34
3.8	Waveform Results for the fourth part of the simulation (74-82 ns)	34
3.9	Waveform Results showing final memory state (74-82 ns)	35
3.10	Instruction Memory.	35
3.11	Waveform Results (0-32 ns)	36
3.12	Waveform Results (32-64 ns)	36
3.13	Instruction Memory Contents	37
3.14	Register File Contents	38
3.15	Data Memory Contents	38
3.16	Instruction Memory Contents	39

List of Tables

2	Main Control Unit Signals.	10
3	Boolean Expressions of Control Unit Signals.	12
4	ALU Operation Unit.	13
5	PCsrc Control Unit.	13
6	Boolean Expressions for ALU Control Signals.	14
7	Boolean Expressions for PC Control Signals	14
8	Boolean Expressions for PC Control Unit Signals	26
9	Boolean Expressions for Hazard Control Unit Signals	26

List of Listings

1	Testcase 1	27
2	Testcase 2	28
3	Testcase 3	29
4	Test Bench	30

1 Theoretical Background

1.1 Set of Operations

The processor supports a fundamental set of operations to ensure a functional and versatile instruction set. These operations are categorized into arithmetic, logical, memory access, and control flow instructions. Each category serves a distinct purpose, enabling the processor to perform a wide range of tasks.

1.1.1 Arithmetic Operations

Arithmetic operations include addition and subtraction, which are essential for numerical computations. These operations are performed using the ALU.

- **ADD (Addition):** Computes the sum of two registers.
- **SUB (Subtraction):** Computes the difference between two registers.
- **ADDI (Add Immediate):** Computes the sum of a register and an immediate value.

1.1.2 Logical Operations

Logical operations perform bitwise manipulation, crucial for tasks such as masking and setting specific bits.

- **AND (Logical AND):** Computes the bitwise AND of two registers.
- **ANDI (AND Immediate):** Computes the bitwise AND of a register and an immediate value.

1.1.3 Memory Access Operations

Memory access operations allow the processor to load data from and store data to memory.

- **LW (Load Word):** Loads a word from memory into a register.
- **SW (Store Word):** Stores a word from a register into memory.
- **LB (Load Byte):** Loads a byte from memory into a register with zero or sign extension.

1.1.4 Control Flow Operations

Control flow operations manage the sequence of instruction execution, enabling conditional and unconditional jumps.

- **JMP (Jump):** Unconditionally jumps to a specified address.
- **CALL (Call):** Calls a subroutine and saves the return address.
- **RET (Return):** Returns from a subroutine.
- **BGT (Branch if Greater Than):** Branches if one register is greater than another.
- **BGTZ (Branch if Greater Than Zero):** Branches if a register is greater than zero.
- **BLT (Branch if Less Than):** Branches if one register is less than another.
- **BLTZ (Branch if Less Than Zero):** Branches if a register is less than zero.
- **BEQ (Branch if Equal):** Branches if two registers are equal.
- **BEQZ (Branch if Equal to Zero):** Branches if a register is equal to zero.
- **BNE (Branch if Not Equal):** Branches if two registers are not equal.
- **BNEZ (Branch if Not Equal to Zero):** Branches if a register is not equal to zero.

1.2 Instruction Set Architecture (ISA)

An Instruction Set Architecture (ISA) serves as the interface between software and hardware, defining the set of instructions that a processor can execute. The ISA for this project is a 16-bit architecture with four distinct instruction types: R-type, I-type, J-type, and S-type. These instructions encompass a range of operations including arithmetic, logical, memory access, and control flow.

1.2.1 R-type Instructions

R-type instructions perform arithmetic and logical operations. The format includes fields for the opcode, destination register (R_d), and two source registers (R_{s1} and R_{s2}). An unused field is reserved for potential future use.

Opcode (15-12)	R_d (11-9)	R_{s1} (8-6)	R_{s2} (5-3)	Unused (2-0)
-------------------	-----------------	-------------------	-------------------	-----------------

1.2.2 I-type Instructions

I-type instructions include immediate values for arithmetic and logical operations, as well as for load and branch operations. The format includes an opcode, destination register (R_d), source register (R_{s1}), a mode bit, and an immediate value.

Opcode (15-12)	Mode (11)	R_d (10-8)	R_{s1} (7-5)	Immediate (4-0)
-------------------	--------------	-----------------	-------------------	--------------------

1.2.3 J-type Instructions

J-type instructions handle jumps and calls. The format includes an opcode and a 12-bit offset to calculate the target address for jumps and calls.

Opcode (15-12)	Jump Offset (11-0)
-------------------	-----------------------

1.2.4 S-type Instructions

S-type instructions manage memory storage. The format includes an opcode, source register (R_s), and an immediate value representing the memory address.

Opcode (15-12)	R_s (11-9)	Immediate (8-0)
-------------------	-----------------	--------------------

1.3 Processor Architecture

The processor architecture comprises eight 16-bit general-purpose registers (R_0 to R_7), with R_0 hardwired to zero. It also includes a 16-bit program counter (PC) and separate data and instruction memories. Memory is byte-addressable and uses little-endian byte ordering.

1.4 Pipeline Stages

To enhance performance, the processor implements a 5-stage pipeline: fetch, decode, execute, memory access, and write-back. Each stage performs a specific function in processing instructions.

1. **Fetch:** The fetch stage retrieves the next instruction from the instruction memory using the program counter (PC).
2. **Decode:** The decode stage interprets the fetched instruction, identifying the opcode, source, and destination registers, and any immediate values.
3. **Execute:** The execute stage performs the operation specified by the instruction, utilizing the Arithmetic Logic Unit (ALU) for arithmetic and logical operations. The ALU generates necessary signals, such as zero, carry, and overflow.
4. **Memory Access:** The memory access stage handles load and store operations, interacting with the data memory to read or write values.
5. **Write-Back:** The write-back stage updates the destination register with the result of the operation performed in the execute stage.

1.5 Control Path

The control path orchestrates the processor's operations by generating control signals based on the decoded instruction. It ensures correct data flow and operation sequencing across the pipeline stages.

1.6 ALU and Condition Branch Signals

The ALU performs arithmetic and logical operations and generates condition signals (zero, carry, overflow) that are critical for branch instructions. These signals determine the outcome of conditional branches (taken or not taken), influencing the next PC value.

1.7 Verification

Verification involves creating a testbench to simulate the processor's operation and validate its correctness. Multiple test programs are executed to ensure that each instruction operates correctly, with results compared against expected outcomes.

2 Procedure

2.1 RTL Design

The design and implementation of the 16-bit pipelined processor involve several key stages, each playing a crucial role in the instruction execution process. Below is the detailed RTL (Register Transfer Level) description for each instruction type, explaining the step-by-step process.

Instruction Type	Stage	RTL
R-type	IF (Instruction Fetch)	$inst = instMem[PC]$ $PC = PC + 2$
	ID (Instruction Decode)	$AluOperand1 = Reg[inst[8:6]]$ $AluOperand2 = Reg[inst[5:3]]$ $DestinationAddress = inst[11:9]$ $Opcode = inst[15:12]$
	EX (Execute)	$AluResult = Operation(AluOperand1, AluOperand2)$
	MEM (Memory Access)	Not required
	WB (Write Back)	$RegFile[DestinationAddress] = AluResult$
I-type (ALU)	IF (Instruction Fetch)	$inst = instMem[PC]$ $PC = PC + 2$
	ID (Instruction Decode)	$Op1 = Reg[inst[7:5]]$ $Immediate = zero_extend(inst[4:0])$ (for ANDI) $Immediate = sign_extend(inst[4:0])$ (for ADDI) $DestinationAddress = inst[10:8]$
	EX (Execute)	$AluResult = Op1 + Immediate$ (for ADDI) $AluResult = Op1 \& Immediate$ (for ANDI)
	MEM (Memory Access)	Not required
	WB (Write Back)	$RegFile[DestinationAddress] = AluResult$

Instruction Type	Stage	RTL
I-type (Load/Store)	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 2
	ID (Instruction Decode)	Op1 = Reg[inst[7:5]] Immediate = inst[4:0] DestinationAddress = inst[10:8]
	EX (Execute)	AluResult = Op1 + Immediate
	MEM (Memory Access)	LW: MemData = DMem[AluResult] LBu: MemData = DMem[AluResult] LBs: MemData = DMem[AluResult] SW: DMem[AluResult] = RegFile[DestinationAddress]
	WB (Write Back)	LW: RegFile[DestinationAddress] = MemData LBu: RegFile[DestinationAddress][7:0] = MemData LBs: RegFile[DestinationAddress] = sign_extend(MemData)
I-type (Branch)	IF (Instruction Fetch)	inst = instMem[PC]
	ID (Instruction Decode)	PcRelativeOffset = sign_ext(inst[4:0]) AluOperand1 = Reg[inst[10:8]] if (inst[11] == 1) AluOperand2 = Reg[0] else AluOperand2 = Reg[inst[7:5]]
	EX (Execute)	AluResult = Sub(AluOperand1, AluOperand2) (Flags for branch decision updated here)
	MEM (Memory Access)	Not required
	WB (Write Back)	Not required

Instruction Type	Stage	RTL
J-type	IF (Instruction Fetch)	inst = instMem[PC]
	ID (Instruction Decode)	Opcode = inst[15:12] ReturnAddress = PC JumpOffset = inst[11:0] « 1
	EX (Execute)	RET: PC = Reg[7] CALL: PC = {PC[15:12], JumpOffset} Reg[15] = ReturnAddress JMP: PC = {PC[15:12], JumpOffset}
	MEM (Memory Access)	Not required
	WB (Write Back)	Not required
S-type	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 2
	ID (Instruction Decode)	Rs = Reg[inst[11:9]] Immediate = sign_extend(inst[8:0]) Opcode = inst[15:12]
	EX (Execute)	DMem[Rs] = Immediate
	MEM (Memory Access)	Not required
	WB (Write Back)	Not required

The processor design follows a 5-stage pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each instruction type undergoes these stages, ensuring a structured and efficient execution flow. The RTL description provided outlines the specific operations performed at each stage for various instruction types, demonstrating the systematic approach to processor design.

2.2 Single Cycle Datapath

The single-cycle data path is an essential aspect of the CPU's architecture, enabling the processor to execute each instruction in one clock cycle. The data path illustrated in the figure includes several key components and connections that facilitate this operation. Here, we provide an explanation of the data path components and their interactions:

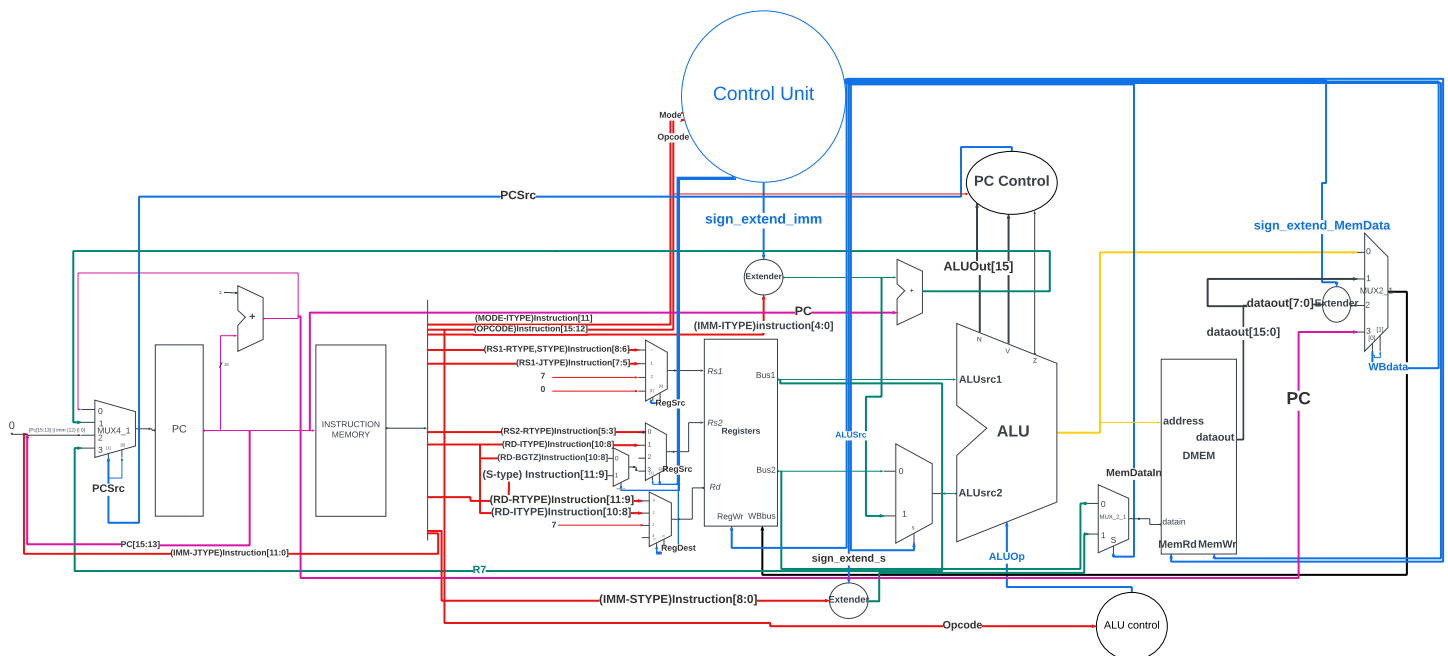


Figure 2.1: Single-Cycle Data Path of the 16-bit Processor

2.2.1 Instruction Fetch (IF)

- **Program Counter (PC):** The PC holds the address of the next instruction to be fetched from the instruction memory. It is updated every clock cycle to point to the next instruction.
- **Instruction Memory:** The instruction memory stores the instructions to be executed by the CPU. The instruction at the address specified by the PC is fetched during this stage.
- **Adder:** An adder is used to increment the PC by 2, pointing to the next instruction, since each instruction is 16 bits (2 bytes).

2.2.2 Instruction Decode (ID)

- **Registers:** The register file consists of eight 16-bit general-purpose registers (R0 to R7). During this stage, the source registers (Rs1 and Rs2) specified by the instruction are read.
- **Instruction Fields:** The instruction is divided into fields (opcode, source registers, destination register, immediate values) to be used in subsequent stages.

2.2.3 Execution (EX)

- **ALU:** The Arithmetic Logic Unit (ALU) performs arithmetic and logical operations on the operands provided by the registers or immediate values. It generates the result based on the instruction's operation code (opcode).
- **Extender:** For I-type instructions, the extender sign-extends or zero-extends the immediate value to match the operand size required by the ALU.

2.2.4 Memory Access (MEM)

- **Data Memory (DMEM):** The data memory is accessed for load and store instructions. The address is calculated by the ALU, and the data is either read from or written to this address.
- **Multiplexer (MUX2_1):** A multiplexer selects between different data sources to be written to the data memory or the register file.

2.2.5 Write Back (WB)

- **Register File:** The result from the ALU or the data memory is written back to the register file, completing the instruction execution cycle.

2.2.6 Control Unit

- **Control Signals:** The control unit generates the necessary control signals based on the opcode of the instruction. These signals control the operation of the multiplexers, ALU, memory, and register file to ensure the correct execution of the instruction.

2.2.7 Branch and Jump Handling

- **PC Control:** For branch and jump instructions, the PC is updated based on the target address calculated by the ALU or immediate values.
- **Branch Condition:** The ALU generates condition flags (zero, carry, overflow) used to determine the outcome of branch instructions.

The single-cycle data path design ensures that each instruction is executed within one clock cycle, making it a simple yet efficient architecture for basic processors. The figure illustrates the interconnected components that facilitate the seamless execution of instructions in a single cycle.

2.3 Single Cycle Control Units

The control unit generates the necessary control signals to orchestrate the operations of the processor. Based on the opcode of the instruction, it sets the control signals for the different components such as the ALU, memory, and registers.

2.3.1 Main Control Signals

The main control signals generated by the control unit are summarized in the table below:

Instruction	Opcode	m	RegSrc1	RegSrc2	RegDest	extend_imm	RegWr	ALUsrc
AND	0000	X	00	00	00	X	1	0
ADD	0001	X	00	00	00	X	1	0
SUB	0010	X	00	00	00	X	1	0
ADDI	0011	X	01	01	01	1	1	1
ANDI	0100	X	01	01	01	0	1	1
LW	0101	X	01	01	01	1	1	1
LBu	0110	0	01	01	01	1	1	1
LBs	0110	1	01	01	01	1	1	1
SW	0111	X	01	01	XX	1	0	1
BGT	1000	0	01	01	XX	1	0	0
BGTZ	1000	1	11	01	XX	1	0	0
BLT	1001	0	01	01	XX	1	0	0
BLTZ	1001	1	11	01	XX	1	0	0
BEQ	1010	0	01	01	XX	1	0	0
BEQZ	1010	1	11	01	XX	1	0	0
BNE	1011	0	01	01	XX	1	0	0
BNEZ	1011	1	11	01	XX	1	0	0
JMP	1100	X	XX	XX	XX	X	0	X
CALL	1101	X	XX	XX	10	X	1	X
RET	1110	X	10	XX	XX	X	0	X
Sv	1111	X	11	11	XX	X	0	0

Table 2: Main Control Unit Signals.

- **m**: Indicates the mode bit for certain instructions (e.g., load byte with zero or sign extension).
- **RegSrc1, RegSrc2**: Selects the source registers for the ALU operation.
- **RegDest**: Specifies the destination register.
- **extend_imm**: Indicates if the immediate value should be sign-extended.
- **RegWr**: Enables writing to the register file.
- **ALUsrc**: Determines if the ALU second operand is a register value or an immediate value.

Signal	Boolean Expression
RegSrc1[1]	$(\text{Opcode} = 0011)$ $\vee (\text{Opcode} = 0100) \vee (\text{Opcode} = 0101) \vee (\text{Opcode} = 0110)$ $\vee (\text{Opcode} = 1000 \wedge \neg m) \vee (\text{Opcode} = 1001 \wedge \neg m)$ $\vee (\text{Opcode} = 1010 \wedge \neg m) \vee (\text{Opcode} = 1011 \wedge \neg m)$
RegSrc1[0]	$(\text{Opcode} = 0000) \vee (\text{Opcode} = 0001) \vee (\text{Opcode} = 0010)$ $\vee (\text{Opcode} = 1000 \wedge m) \vee (\text{Opcode} = 1001 \wedge m)$ $\vee (\text{Opcode} = 1010 \wedge m) \vee (\text{Opcode} = 1011 \wedge m)$
RegSrc2	$(\text{Opcode} = 0011) \vee (\text{Opcode} = 0100) \vee (\text{Opcode} = 0101)$ $\vee (\text{Opcode} = 0110) \vee (\text{Opcode} = 0111)$ $\vee (\text{Opcode} = 1000) \vee (\text{Opcode} = 1001) \vee (\text{Opcode} = 1010)$ $\vee (\text{Opcode} = 1011)$
RegDest[1]	$(\text{Opcode} = 1101)$
RegDest[0]	$(\text{Opcode} = 0000) \vee (\text{Opcode} = 0001) \vee (\text{Opcode} = 0010)$ $\vee (\text{Opcode} = 0011) \vee (\text{Opcode} = 0100)$ $\vee (\text{Opcode} = 0101) \vee (\text{Opcode} = 0110)$
sign_extend_imm	$(\text{Opcode} = 0011) \vee (\text{Opcode} = 0101) \vee (\text{Opcode} = 0110)$ $\vee (\text{Opcode} = 1000) \vee (\text{Opcode} = 1001)$ $\vee (\text{Opcode} = 1010) \vee (\text{Opcode} = 1011)$
RegWr	$(\text{Opcode} = 0000) \vee (\text{Opcode} = 0001) \vee (\text{Opcode} = 0010)$ $\vee (\text{Opcode} = 0011) \vee (\text{Opcode} = 0100)$ $\vee (\text{Opcode} = 0101) \vee (\text{Opcode} = 0110)$ $\vee (\text{Opcode} = 1101)$

Signal	Boolean Expression
ALUsrc	$(\text{Opcode} = 0011) \vee (\text{Opcode} = 0100)$ $\vee (\text{Opcode} = 0101) \vee (\text{Opcode} = 0110)$ $\vee (\text{Opcode} = 0111)$
MemDataIn	$(\text{Opcode} = 0111)$
MemRd	$(\text{Opcode} = 0101)$ $\vee (\text{Opcode} = 0110)$
MemWr	$(\text{Opcode} = 0111)$
Sign_Extend_Data	$(\text{Opcode} = 0110 \wedge m)$
WBdata[1]	$(\text{Opcode} = 1101)$
WBdata[0]	$(\text{Opcode} = 0101)$ $\vee (\text{Opcode} = 0110)$

Table 3: Boolean Expressions of Control Unit Signals.

2.3.2 ALU Control

The ALU control unit generates signals that control the operation of the ALU based on the instruction's opcode. The table below summarizes the ALU control signals:

Instruction	ALUOp
AND	and(01)
ADD	add(00)
SUB	sub(10)
ADDI	add(00)
ANDI	and(01)
LW	add(00)
LBu	add(00)
LBs	add(00)
SW	add(00)
BGT	xx
BGTZ	xx
BLT	xx
BLTZ	xx
BEQ	xx
BEQZ	xx
BNE	xx
BNEZ	xx
JMP	xx
CALL	xx
RET	xx
Sv	add(00)

Table 4: ALU Operation Unit.

- **ALUOp:** Determines the specific operation the ALU will perform (e.g., AND, ADD, SUB). The ALUOp signal is derived from the instruction opcode and is used to set the operation mode of the ALU.

2.3.3 PC Control

The PC control unit determines the next value of the PC based on the instruction and the result of the ALU. The table below summarizes the PC control signals:

IDOpcode	Flags	PCSrc
BGT	(Z == 1) (N != V)	01
BGTZ	(Z == 1) (N != V)	01
BLT	(N == V)	01
BLTZ	(N == V)	01
BEQ	(Z == 1)	01
BEQZ	(Z == 1)	01
BNE	(Z == 0)	01
BNEZ	(Z == 0)	01
JMP	—	10
CALL	—	10
RET	—	11
Else	—	00

Table 5: PCSrc Control Unit.

- **Flags:** Conditions derived from the ALU status flags:
 - **Zero (Z):** Indicates that the ALU result is zero.
 - **Negative (N):** Indicates that the ALU result is negative.
 - **Overflow (V):** Indicates that an arithmetic overflow has occurred.
- **PCSrc:** Specifies the source for the next Program Counter (PC) value:
 - **01:** Branch target address.
 - **10:** Jump target address.
 - **11:** Return address from a call.
 - **00:** Sequential address (PC + 2).

ALU Control Logic Equations

Signal	Boolean Expression
ALUOp[1]	(Opcode = SUB)
ALUOp[0]	(Opcode = AND) \vee (Opcode = ANDI)

Table 6: Boolean Expressions for ALU Control Signals.

PC Control Logic Equations

Signal	Boolean Expression
PCSrc[1]	(Opcode = JMP) \vee (Opcode = CALL) \vee (Opcode = RET)
PCSrc[0]	(Opcode = BGT) \vee (Opcode = BGTZ) \vee (Opcode = BLT) \vee (Opcode = BLTZ) \vee (Opcode = BEQ) \vee (Opcode = BEQZ) \vee (Opcode = BNE) \vee (Opcode = BNEZ)

Table 7: Boolean Expressions for PC Control Signals

2.4 Pipelining

2.4.1 Datapath

The pipeline data path of our processor is divided into five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage is designed to perform specific tasks to facilitate the efficient execution of instructions. Below, we provide detailed explanations of each stage and the corresponding stage buffers, followed by a discussion on how hazards are handled in the pipeline.

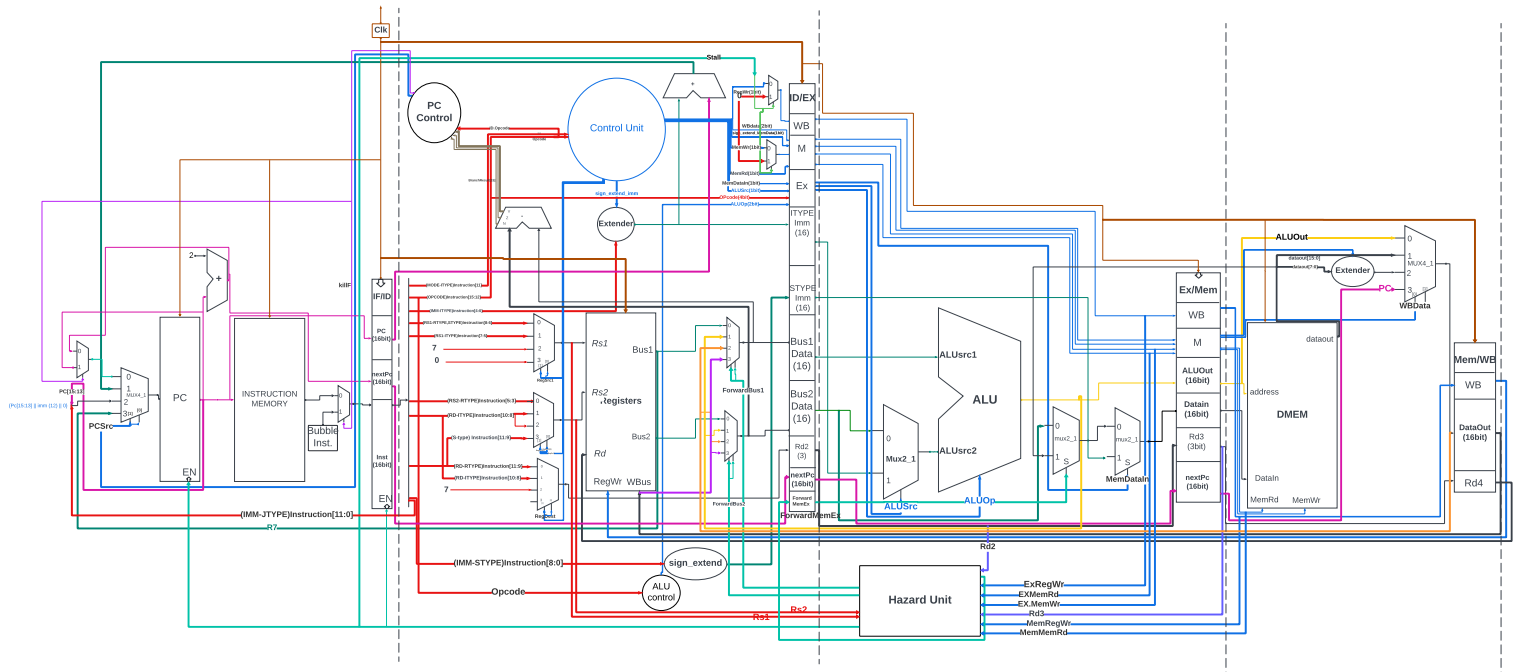


Figure 2.2: Pipeline Data Path of the 16-bit Processor

2.4.2 Instruction Fetch (IF)

The Instruction Fetch stage is responsible for retrieving the next instruction from the instruction memory. The main components involved in this stage include the Program Counter (PC) and the Instruction Memory.

Components:

- **Program Counter (PC):** The Program Counter (PC) holds the address of the next instruction to be fetched. The PC is updated based on the value of the PCSrc flag, which is a 2-bit signal determined by the PC control logic during the Instruction Decode (ID) stage. The PCSrc flag selects the source of the next PC value from four possible inputs:
 - **Input 0:** The next sequential instruction address, $PC + 2$, for regular instruction flow.

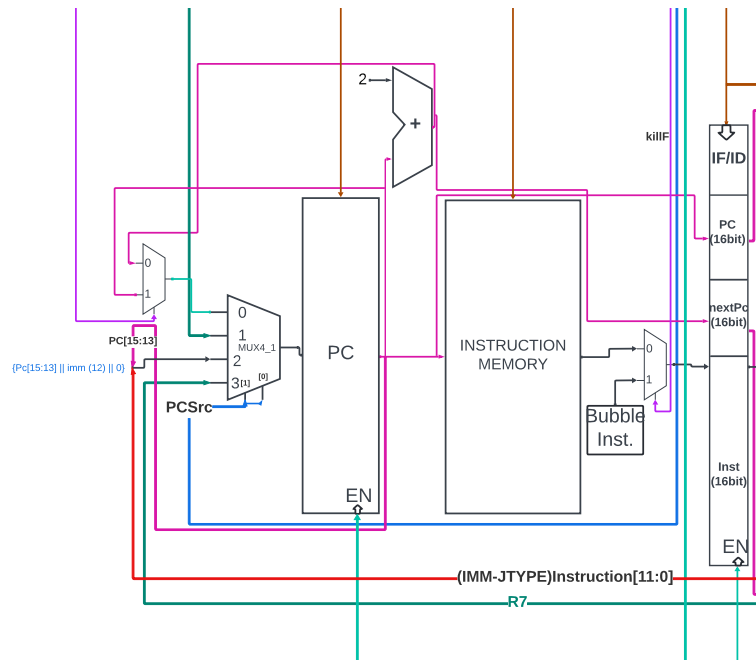


Figure 2.3: Pipeline Data Path Instruction Fetch Stage

- **Input 1:** The branch target address calculated during the ID stage, used for branch instructions.
- **Input 2:** The jump target address provided by the ID stage, used for jump instructions.
- **Input 3:** The value stored in register $R7$, used for the RET (return from subroutine) instruction.
- **Instruction Memory:** Stores the instructions to be executed by the processor. The instruction is fetched from the address provided by the Program Counter (PC).
- **Adder:** Increments the PC by 2 to point to the next instruction.

- **2-to-1 Multiplexer (16-bit output, 1-bit stall input):** This multiplexer determines the instruction that will be forwarded to the IF/ID buffer. It has two inputs:
 - **Input 0:** The normal instruction read from the instruction memory.
 - **Input 1:** A bubble (no-op) instruction used for stalling the pipeline.The selection between these inputs is controlled by the stall signal.

Operations:

- The PC provides the address to the Instruction Memory.
- The instruction is fetched from the Instruction Memory.
- The PC is incremented by 2 to point to the next instruction.

Stage Buffer:

- **IF/ID Register:** Holds the fetched instruction, the incremented PC value for the next stage, and the original PC.

2.4.3 Instruction Decode (ID)

In the Instruction Decode stage, the fetched instruction is decoded to identify the opcode, source registers, destination register, and immediate values. The register file is accessed to read the values of the source registers.

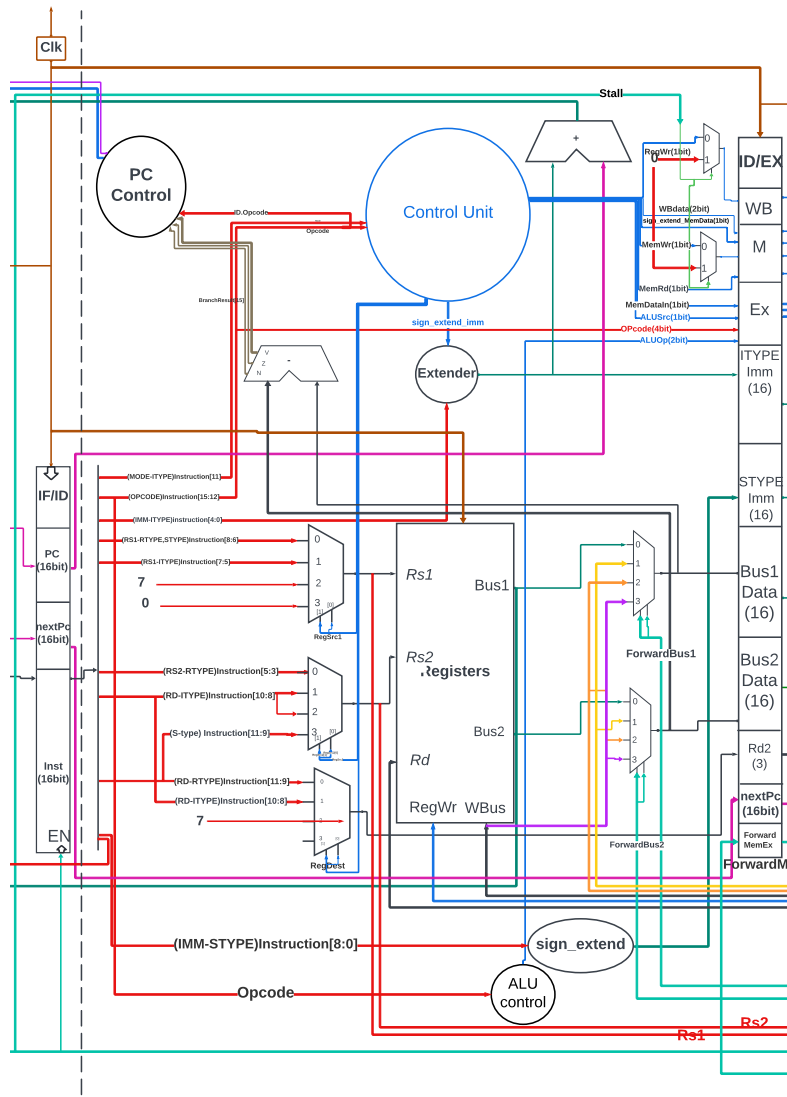


Figure 2.4: Pipeline Data Path Instruction Decode Stage

Components:

- **Register File:** Consists of eight 16-bit general-purpose registers (R0 to R7). This is where all the temporary data required by the processor is stored and accessed during execution.
- **Control Unit:** Generates control signals based on the decoded instruction. It orchestrates the operations of the processor by signaling the appropriate components to perform their tasks.
- **PC Control:** Generates control signals related to the Program Counter (PC) register based on the decoded instruction. It handles the update and manipulation of the PC, ensuring the correct sequence of instruction execution.
- **Multiplexers:**
 - **Five (4-to-1) Multiplexers:** Used to select from multiple data sources based on control signals, ensuring the correct data paths are followed during operations as the following:
 - * **RegSrc1 Mux:** Chooses the source of R_{S1} as follows:
 - 0 Instruction[8:6] (R-TYPE, S-TYPE).
 - 1 Instruction[7:5] (I-TYPE).
 - 2 R_7 (constant register).
 - 3 R_0 (zero register).
 - * **RegSrc2 Mux:** Chooses the source of R_{S2} as follows:
 - 0 Instruction[5:3] (R-TYPE).
 - 1 2 Instruction[10:8] (for R_d in I-TYPE).
 - 3 R_7 (constant register).
 - * **RegDest Mux:** Chooses the destination register R_d as follows:
 - 0 Instruction[11:9] (R-TYPE).
 - 1 Instruction[10:8] (I-TYPE).
 - 2 R_7 (constant register).
 - * **ForwardBus1 Mux:** Chooses the value of Bus1 Data part in ID\EX as follows:
 - 0 Bus1 (output of register file).
 - 1 ALU result.
 - 2 MemoryOut (output of data memory).
 - 3 Write-Back bus.
 - * **ForwardBus2 Mux:** Chooses the value of Bus2 Data part in ID\EX as follows:
 - 0 Bus2 (output of register file).
 - 1 ALU result.
 - 2 MemoryOut (output of data memory).
 - 3 Write-Back bus.
 - **Two (2-to-1) Multiplexers:** Used to propagate values to be used in the memory stage, facilitating efficient data flow and memory access as the following:

- * **WB Mux:** Propagates the signal of RegWr as follows:
 - 0 RegWr Signal.
 - 1 R_0 .
- * **Mem Mux:** Propagates the signal of MemWr as follows:
 - 0 MemWr Signal.
 - 1 R_0 .
- **Sign Extender:** Extends an 8-bit immediate value to a signed 16-bit value, allowing for proper handling of signed arithmetic operations in the processor.
- **ALU Control:** Generates control signals for the Arithmetic Logic Unit (ALU) based on the decoded instruction's opcode. This determines the specific operation the ALU will perform (e.g., addition, subtraction, logical operations).
- **Adder Component:** Calculates the target address for branch instructions, determining the next instruction address when a branch is taken.
- **Subtractor Component:**
 - Generates arithmetic signals for the PC Control, aiding in branch prediction and other PC-related calculations.
 - Calculates the target address of branch instructions, similar to the adder but used in specific arithmetic conditions.

Operations:

- **Instruction Decoding:** The instruction is decoded into its fields: opcode, source registers, destination register, and immediate value. This decoding process is crucial for determining the operation type.
- **Register Access:** Accesses the register file to read values from the source registers, which are necessary inputs for arithmetic or logical operations.
- **Control Signal Generation:** Generates control signals based on the decoded instruction, influencing the ALU, multiplexers, and PC Control.
- **Sign Extension:** Extends immediate values to the required 16-bit format, ensuring uniform data size for processing.
- **ALU Preparation:** The ALU Control generates signals to configure the ALU for the specific arithmetic or logical operation required by the instruction.
- **Branch Target Calculation:** The Adder and Subtractor calculate branch and jump addresses, facilitating conditional flow control.
- **Multiplexer Selection:** Multiplexers select appropriate data paths, ensuring correct inputs for the ALU and other processing stages based on control signals.

Stage Buffer:

- **ID/EX Register:** Stores opcode, source register values, and other decoded information for passing between the Instruction Decode (ID) and Execute (EX) stages in the pipeline.

2.4.4 Execution (EX)

The Execution stage is responsible for performing arithmetic and logical operations specified by the instruction. The main components involved in this stage include the ALU, multiplexers, and forwarding units.

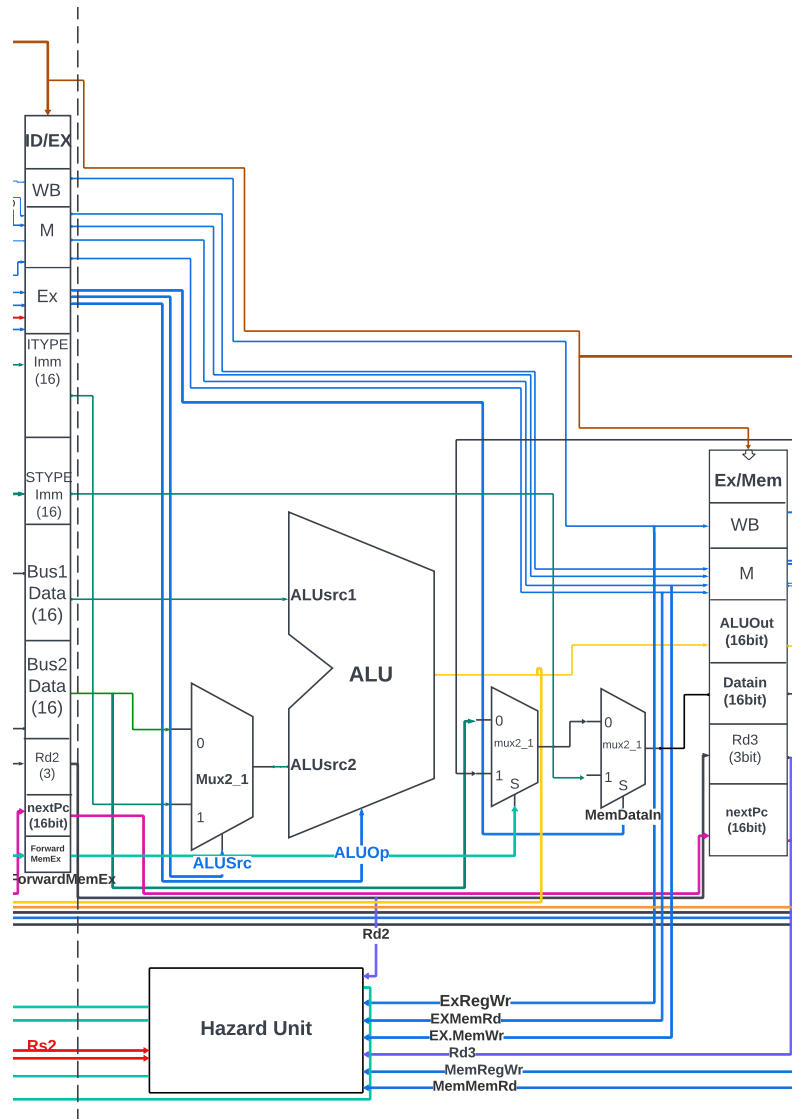


Figure 2.5: Pipeline Data Path Execution Stage

Components:

- **ALU:** The Arithmetic Logic Unit (ALU) is responsible for performing arithmetic and logical operations. It operates based on the ALUOp control signal, which is 2 bits wide and specifies the operation to be performed. The ALU takes two operands as inputs:
 - **ALUSrc1:** The first operand for the operation, which comes from the previous stage buffer. This operand can either be forwarded or read from the register file.
 - **ALUSrc2:** The second operand for the operation, which also comes from the previous stage buffer. However, a 2-to-1 multiplexer with a 16-bit output is used to select between the input from the previous buffer and the immediate value from I-type instructions.
- **2-to-1 Multiplexer (ALUSrc2):** Selects the second ALU operand from Bus2 data, immediate values, or other forwarded values.
- **Hazard Unit:** The Hazard Unit detects and manages hazards to ensure correct instruction execution. It generates the necessary control signals and determines if forwarding is required. By selecting the correct data to be forwarded to the ALU inputs, the Hazard Unit resolves data hazards effectively.
- **2-to-1 Multiplexer (Memory Data In):** Selects between two inputs for the data to be written to memory:
 - **Input 0:** The value read from the register file or forwarded and stored in the previous stage buffer.
 - **Input 1:** The immediate value used in the SV instruction.
- **2-to-1 Multiplexer (Memory Data In Forwarding):** This multiplexer selects the data to be written to memory from two possible sources:
 - **Input 0:** The result from the Memory Data In multiplexer.
 - **Input 1:** A forwarded value from the memory stage, used when a store instruction follows a load instruction and the data to be stored is the same as the data that was just loaded (i.e., from the same register).

Operations:

- The ALU performs the operation specified by the ALUOp signal using the operands selected by the ALUSrc1 and ALUSrc2 multiplexers.
- The ALU result is stored in the EX/MEM register along with the destination register identifier and any control signals needed for the Memory Access stage.

Stage Buffer:

- **EX/MEM Register:** This register holds several important pieces of information for the subsequent stages:
 - The result of the ALU operation.
 - The identifier of the destination register (Rd3).
 - The data to be written to memory (DataIn).
 - The next program counter value (nextPC).
 - Control flags required for the memory and write-back stages.

2.4.5 Memory Access (MEM)

In the Memory Access stage, the data memory is accessed for load and store operations. The address is calculated by the ALU, and the data is either read from or written to this address. This stage is critical for handling memory operations and ensuring data integrity during the execution of instructions.

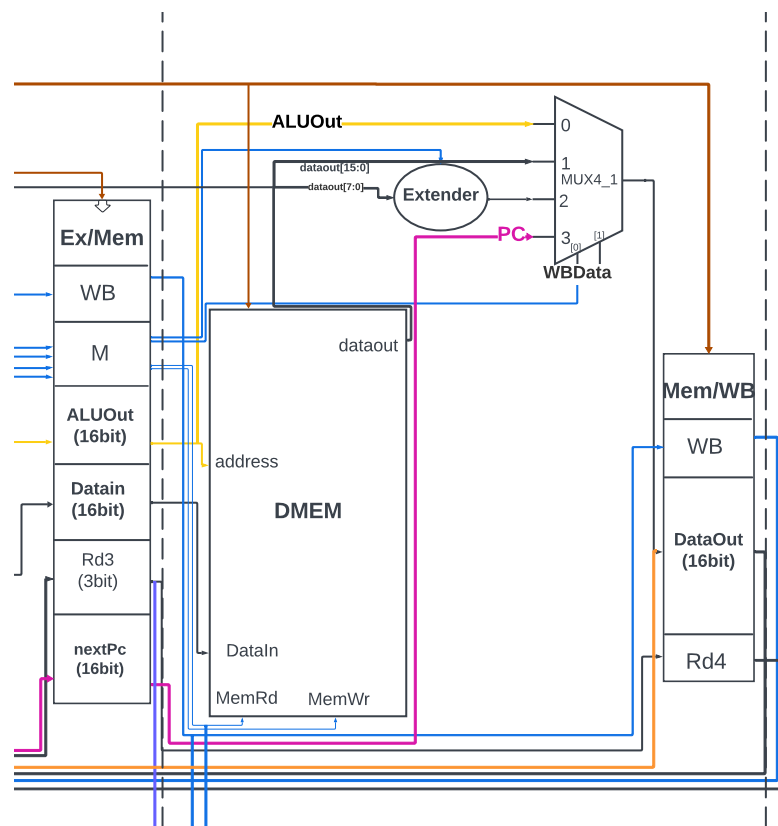


Figure 2.6: Pipeline Data Path Memory Stage

Components:

- **Data Memory (DMEM):**
 - Stores data that can be read from or written to by load and store instructions.
 - Supports byte (half-word), and word access.
- **WBData Mux:** Chooses the value of the WBbus to write on the register file\EX based on the following control signals:
 - **0** ALU result.
 - **1** DataOut (output of data memory) **without** extension.
 - **2** DataOut (output of data memory) **with** extension.
 - **3** Program Counter (PC).
- **Extender:**
 - Extends the least significant 8 bits from data memory by either signed or unsigned extension.
 - Ensures proper handling of smaller data types when loading them into registers.

Operations:

- **Load Instructions:**
 - Data is read from the address calculated by the ALU.
 - For 8-bit loads, a signed extension may be applied using the extender unit.
- **Store Instructions:**
 - Data is written to the address calculated by the ALU.
 - Ensures the correct data is stored in the correct memory location.

Stage Buffer:

- **MEM/WB Register:**
 - Holds the data read from memory or the ALU result.
 - Stores the destination register information for the Write-Back (WB) stage.
 - Ensures smooth data transition between the Memory Access and Write-Back stages.

2.4.6 Write Back (WB)

The Write Back stage updates the destination register with the result stored in the buffer by the previous stage, completing the instruction execution cycle.

Components

- **Register File:** The destination register is updated with the result if the write-back flag is set.

Operations

- The result from the buffer is written back to the destination register.

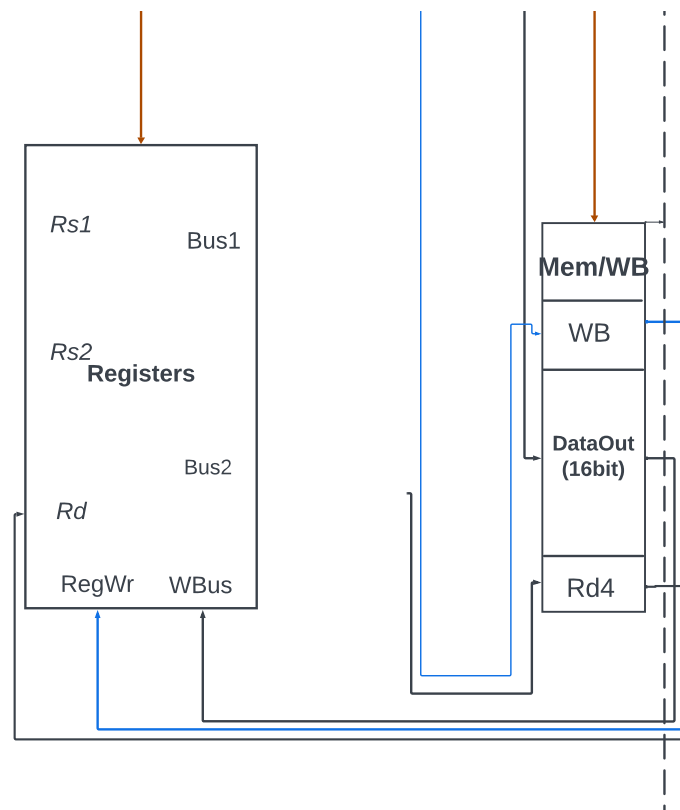


Figure 2.7: Pipeline Data Path Write Back Stage

Stage Buffer

- **WB Register:** Holds the final result to be written back to the register file, the address of the destination register (*Rd4*), and the write-back enable (*WB*) signal.

2.5 Pipeline Control Units

2.5.1 PC Control Unit

The PC Control Unit determines the next value of the Program Counter (PC) based on various instruction types and conditions. Below are the Boolean equations for the PC control signals:

PC Control Logic Equations

Signal	Boolean Expression
PCSrc[1]	$(\text{Opcode} = \text{JMP}) \vee (\text{Opcode} = \text{CALL}) \vee (\text{Opcode} = \text{RET})$
PCSrc[0]	$(\text{Opcode} = \text{BGT}) \vee (\text{Opcode} = \text{BGTZ}) \vee (\text{Opcode} = \text{BLT}) \vee (\text{Opcode} = \text{BLTZ}) \vee (\text{Opcode} = \text{BEQ}) \vee (\text{Opcode} = \text{BEQZ}) \vee (\text{Opcode} = \text{BNE}) \vee (\text{Opcode} = \text{BNEZ})$
Branch	$(\text{Opcode} = \text{BGT}) \vee (\text{Opcode} = \text{BGTZ}) \vee (\text{Opcode} = \text{BLT}) \vee (\text{Opcode} = \text{BLTZ}) \vee (\text{Opcode} = \text{BEQ}) \vee (\text{Opcode} = \text{BEQZ}) \vee (\text{Opcode} = \text{BNE}) \vee (\text{Opcode} = \text{BNEZ})$
J-type	$(\text{Opcode} = \text{JMP}) \vee (\text{Opcode} = \text{CALL}) \vee (\text{Opcode} = \text{RET})$

Table 8: Boolean Expressions for PC Control Unit Signals

2.5.2 Hazard Control Unit

The Hazard Control Unit handles data hazards by generating forwarding and stall signals to ensure correct instruction execution. Below are the Boolean equations for the forwarding and stall signals:

Hazard Unit Logic Equations

Signal	Boolean Expression
ForwardBus1 = 0	$\text{ForwardBus1} \neq 1 \wedge \text{ForwardBus1} \neq 2$
ForwardBus1 = 1	$\text{Rs1} \neq 0 \wedge \text{Rs1} == \text{Rd2} \wedge \text{ExRegWr}$
ForwardBus1 = 2	$\text{Rs1} \neq 0 \wedge \text{Rs1} == \text{Rd3} \wedge \text{MemRegWr}$
ForwardBus2 = 0	$\text{ForwardBus2} \neq 1 \wedge \text{ForwardBus2} \neq 2$
ForwardBus2 = 1	$\text{Rs2} \neq 0 \wedge \text{Rs2} == \text{Rd2} \wedge \text{ExRegWr}$
ForwardBus2 = 2	$\text{Rs2} \neq 0 \wedge \text{Rs2} == \text{Rd3} \wedge \text{MemRegWr}$
ForwardME = 1	$\text{Ex.MemRd} \wedge \text{D.MemWr} \wedge \text{Rd2} == \text{Rd3}$
Stall = 1	$\text{Ex.MemRd} \wedge (\text{ForwardBus1} == 1 \vee \text{ForwardBus2} == 1) \wedge \neg \text{ForwardME}$

Table 9: Boolean Expressions for Hazard Control Unit Signals

3 Verification

The verification of the pipelined MIPS processor was performed using a test bench and simulation tools. The results of the simulations confirm the correct functionality of the processor. The test bench code used for the verification and the results of the simulations are discussed below.

3.1 Tested Assembly Code Sequence

The following assembly code sequence was tested to verify the functionality of the pipelined MIPS processor:

Listing 1: Testcase 1

```
; Testcase #1
JMP main

sum:
    ADD r5, r5, r3
    ADDI r3, r3, -1
    BNE r3, r0, -4
    RET

main:
    SW r0, 5
    LW r2, r0, 0
    SW r2, r0, 2
    LBU r3, r0, 2
    ANDI r5, r0, 0
    CALL sum
    SW r5, r0, 4
    ADDI r6, r5, -15
    ADDI r6, r6, -15
    SW r6, r0, 6
    LBS r1, r0, 6
```

Listing 2: Testcase 2

```
; Testcase #2
JMP main

; computes the max of r1 and r2, and stores the result in r3.
max:
    BGT r2, r1, 6
    ADD r3, r0, r1
    RET
r2_is_max:
    ADD r3, r0, r2
    RET

; computes the min of r1 and r2, and stores the result in r4.
min:
    BLT r2, r1, 6
    ADD r4, r0, r1
    RET
r2_is_min:
    ADD r4, r0, r2
    RET

; finds the sign of r5, and stores the result in r6.
; positive: 0, negative: 1
sign_of:
    BLTZ r5, 6
    ADDI r6, r0, 0
    RET
r5_is_negative:
    ADDI r6, r0, 1
    RET

main:
    ADDI r1, r0, 5
    ADDI r2, r0, 7
    CALL max
    CALL min
    ADD r5, r0, r4
    CALL sign_of
    ; attempt to write to r0
    ADDI r0, r2, 7
```

Listing 3: Testcase 3

```
; Testcase #3
; data hazards
ADDI r1, r0, 3
ADDI r2, r1, 4
AND r3, r2, r1
SUB r4, r2, r3

Sv r0, -4
LBs r5, r0, 0
ADD r6, r5, r4
ANDI r1, r5, 0
SW r6, r0, 2
```

3.2 Test Bench Code

The following Verilog code was used as the test bench for the pipelined MIPS processor:

Listing 4: Test Bench

```
1 module testbench();
2     reg          clk;
3     reg          reset;
4     reg [1:0]     ctr;
5     wire [15:0]  datain, dataAddr;
6     wire MemWr;
7
8     // instantiate device to be tested
9     top dut(clk, reset, datain, dataAddr, MemWr);
10    // initialize test
11    initial
12        begin
13            reset <= 1; # 1; reset <= 0;
14            ctr <= 0;
15        end
16    // generate clock to sequence tests
17    always
18        begin
19            clk <= 1; # 1; clk <= 0; # 1;
20        end
21    // check results
22    always@(negedge clk)
23        begin
24            if(MemWr) begin
25
26                if(((dataAddr != 0 | datain != 5) & ctr == 0) |
27                    ((dataAddr != 2 | datain != 5) & ctr == 1) |
28                    ((dataAddr != 4 | datain != 15) & ctr == 2) |
29                    ((dataAddr != 6 | datain != 16'b1111111111110001) & ctr == 3))
30                    begin
31                        $display("Simulation failed. Values: address=%x, data=%x,
32                                ctr=%d, time=%d", dataAddr, datain, ctr, $time);
33                    end
34                else begin
35                    $display("write passed. address=%x, data=%x, ctr=%d, time=%d",
36                            dataAddr, datain, ctr, $time);
37                end
38
39                if(ctr == 3) begin
40                    $display("simulation passed!");
41                    $stop;
42                end
43
44                ctr = ctr + 1;
45            end
46        end
47 endmodule
```

3.3 Simulation Results

The simulation results show the correct operation of the pipelined MIPS processor. The following figures illustrate the waveforms and the values of various signals during the simulation.

3.3.1 Testcase #1

```

Console
# Waveform file 'untitled.awc' connected to 'c:/My_Designs/Pipelined_MIPS/PipelinedMIPS/src/wave.asdb'.
# run 100 ns
# KERNEL: write passed. address=0000, data=0005, ctr=0, time= 11
# KERNEL: write passed. address=0002, data=0005, ctr=1, time= 17
# KERNEL: write passed. address=0004, data=000f, ctr=2, time= 71
# KERNEL: write passed. address=0006, data=fff1, ctr=3, time= 77
# KERNEL: simulation passed!
# RUNTIME: Info: RUNTIME_0070 MIPS.v (46): $stop called.
# KERNEL: Time: 77 ns, Iteration: 1, Instance: /testbench, Process: @ALWAYS#31_2@.
# KERNEL: Stopped at time 77 ns + 1.
# endsim
# VSIM: Simulation has finished.
>

```

Figure 3.1: Simulation Console Output

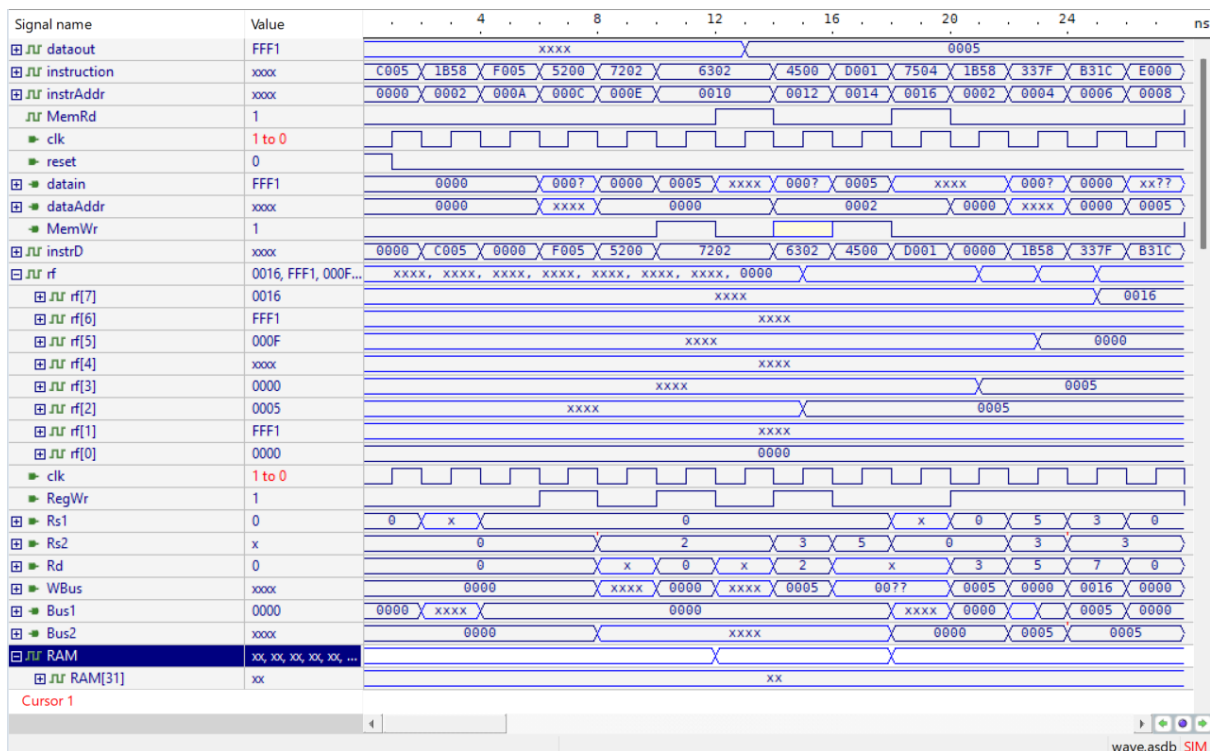


Figure 3.2: Waveform Results for the first part of the simulation (0-24 ns)

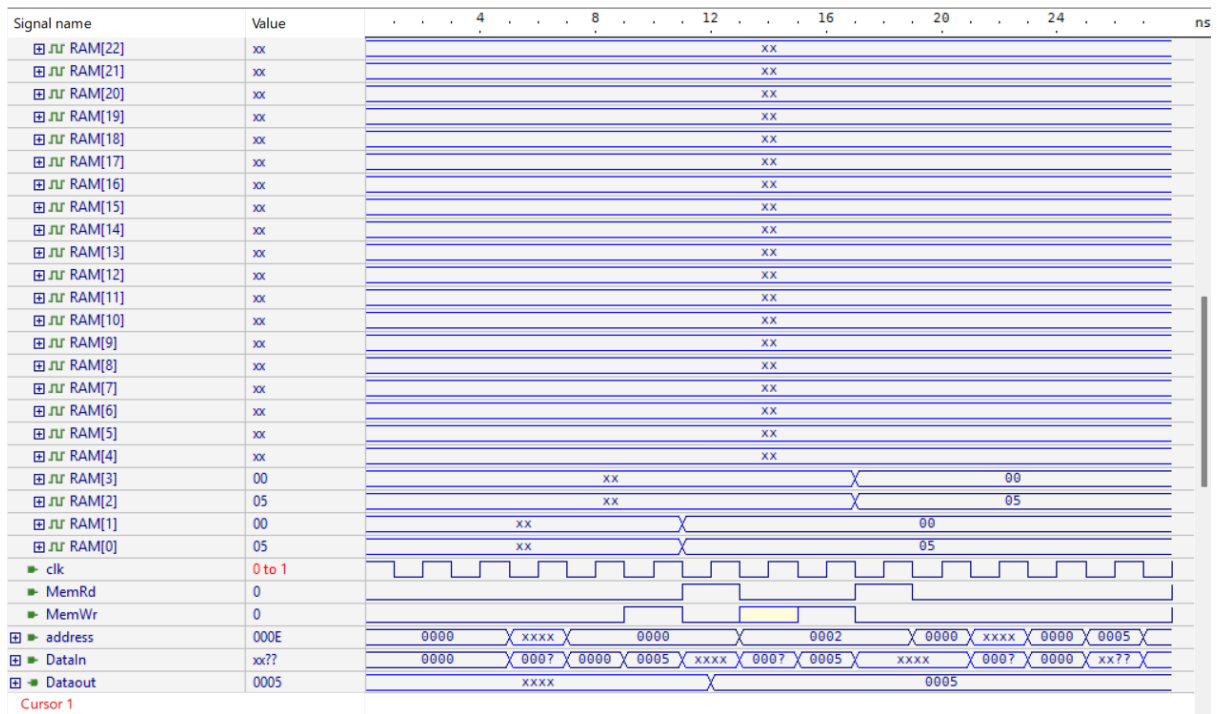


Figure 3.3: Data memory contents for the first part of the simulation (0-24 ns))

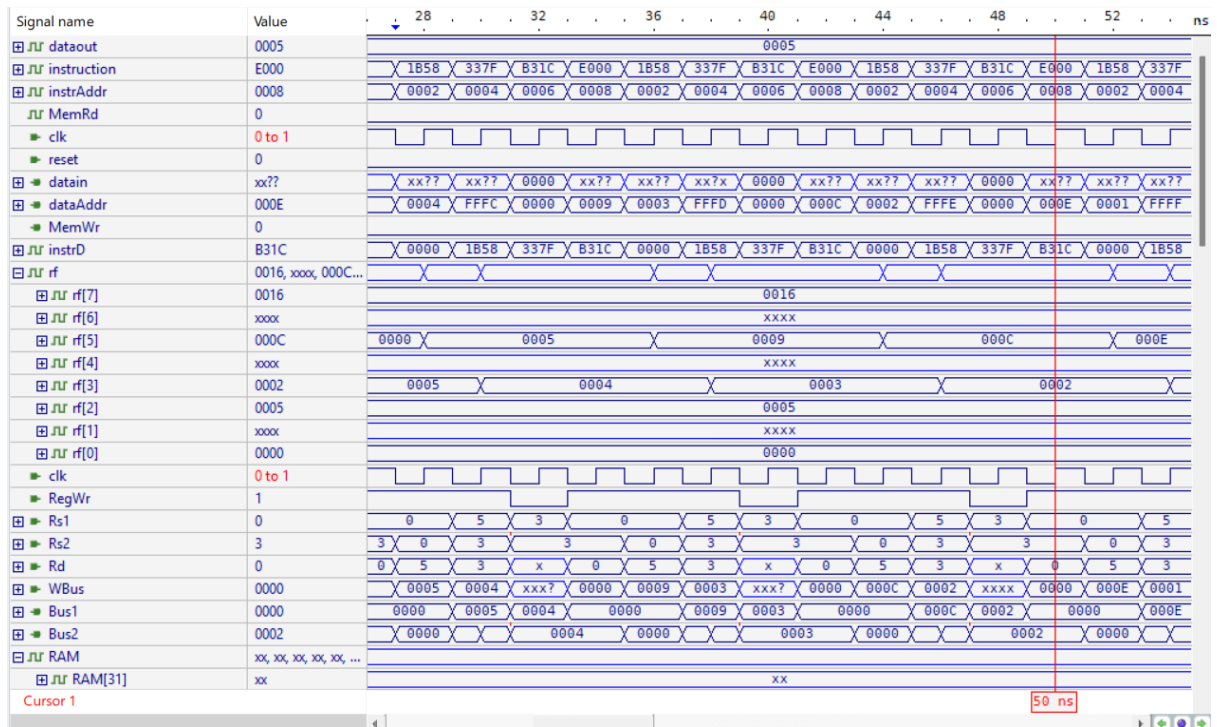


Figure 3.4: Waveform Results for the second part of the simulation (24-50 ns)

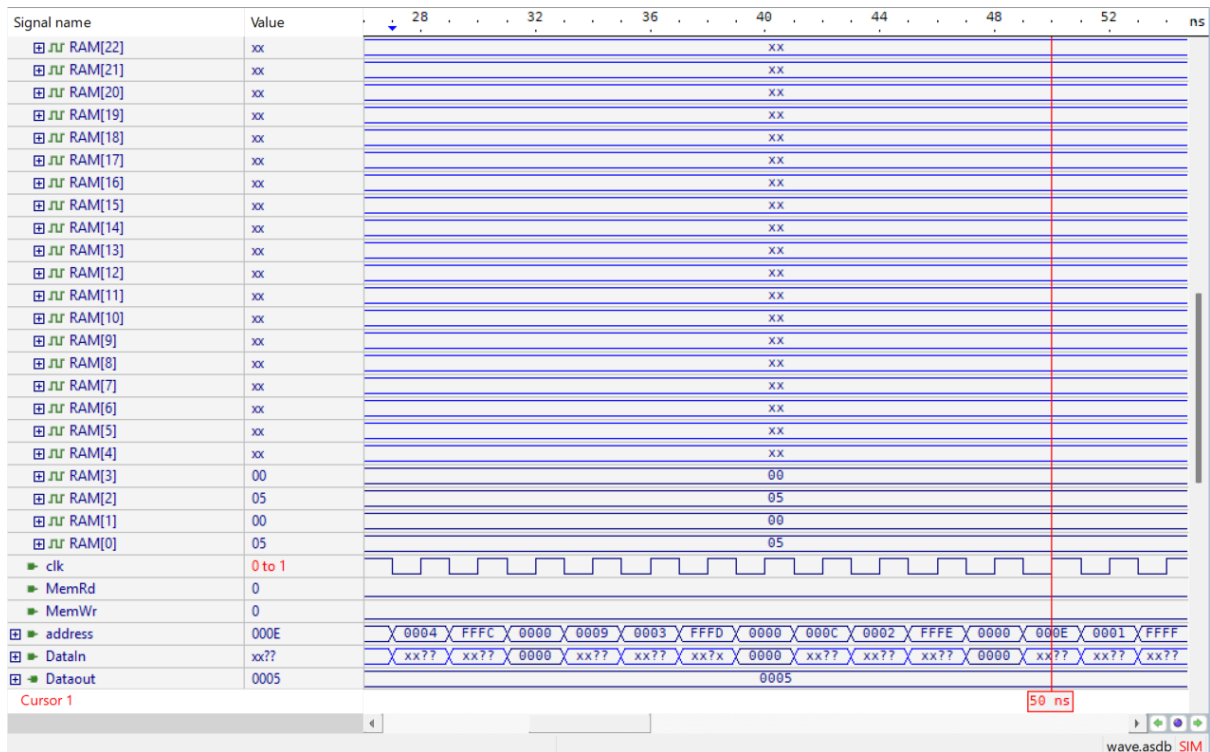


Figure 3.5: Data memory contents for the second part of the simulation (24-50 ns)

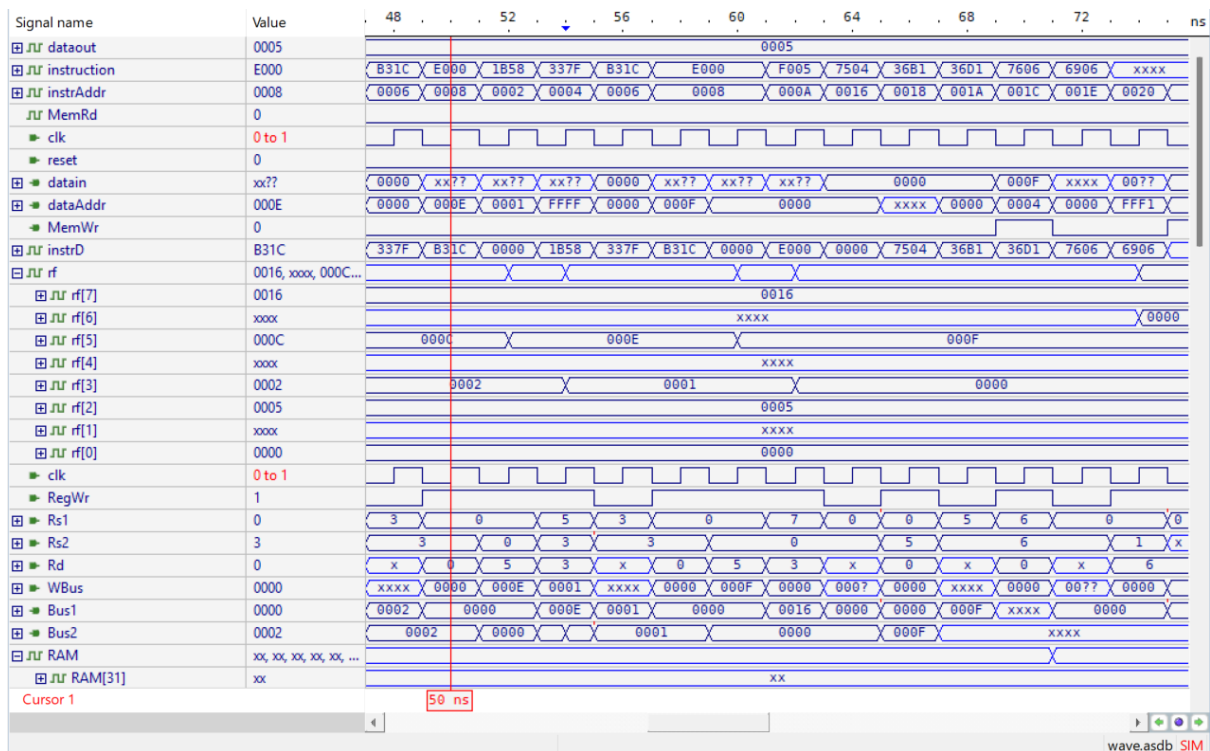


Figure 3.6: Waveform Results for the second part of the simulation (48-74 ns)

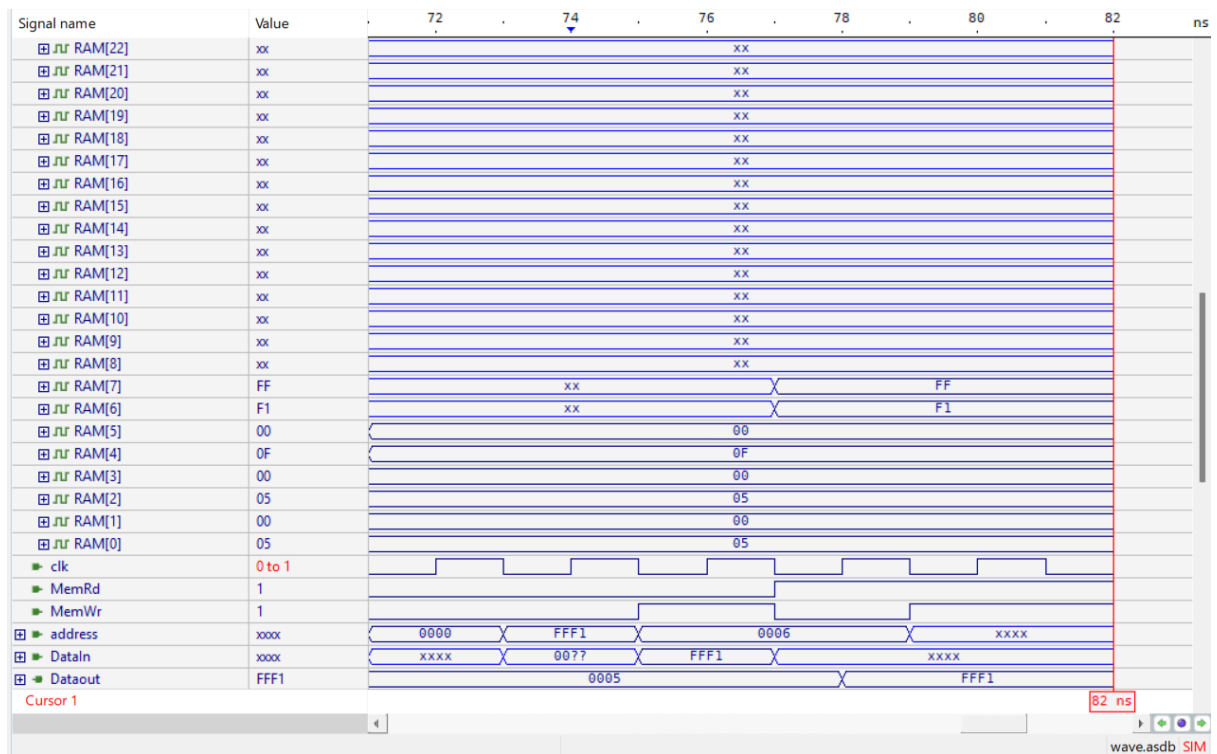


Figure 3.9: Waveform Results showing final memory state (74-82 ns)

RAM[15]	72	RAM[31]	69
RAM[14]	02	RAM[30]	06
RAM[13]	52	RAM[29]	76
RAM[12]	00	RAM[28]	06
RAM[11]	F0	RAM[27]	36
RAM[10]	05	RAM[26]	D1
RAM[9]	E0	RAM[25]	36
RAM[8]	00	RAM[24]	B1
RAM[7]	B3	RAM[23]	75
RAM[6]	1C	RAM[22]	04
RAM[5]	33	RAM[21]	D0
RAM[4]	7F	RAM[20]	01
RAM[3]	1B	RAM[19]	45
RAM[2]	58	RAM[18]	00
RAM[1]	C0	RAM[17]	63
RAM[0]	05	RAM[16]	02

Figure 3.10: Instruction Memory.

3.3.2 Testcase #2

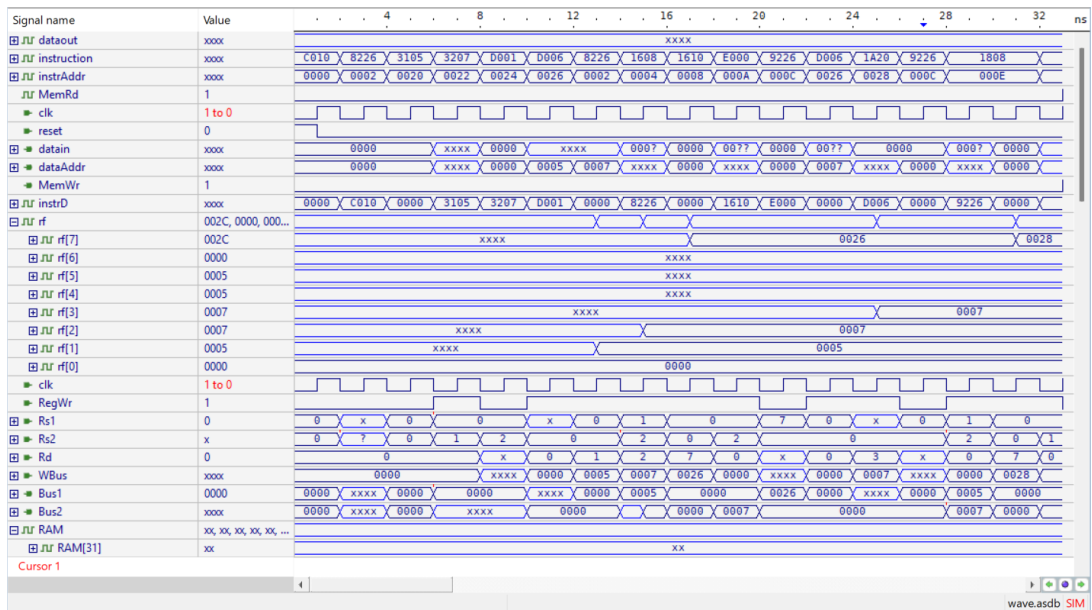


Figure 3.11: Waveform Results (0-32 ns)

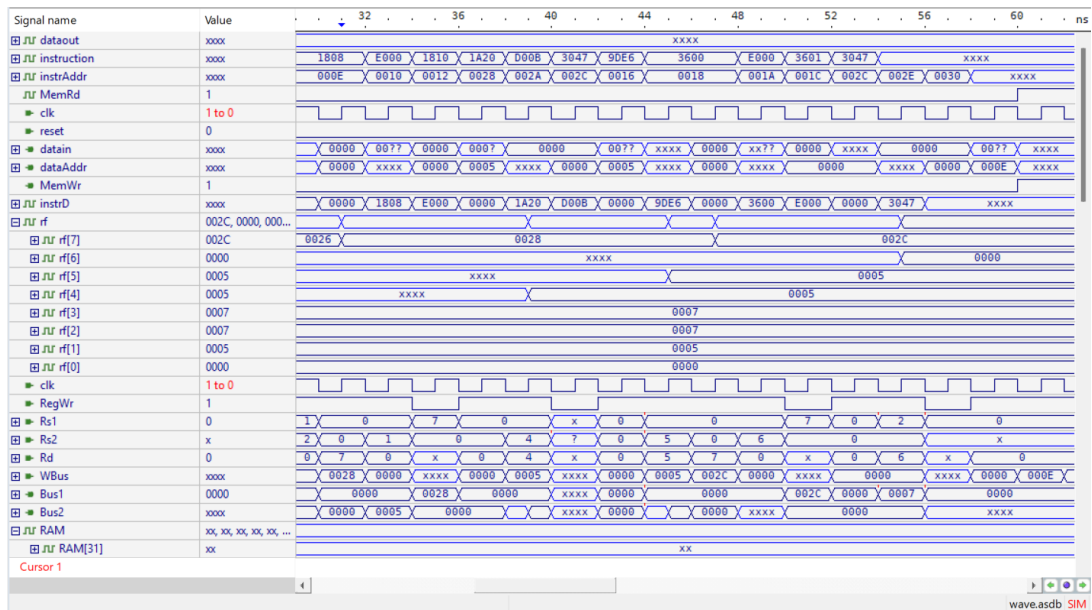


Figure 3.12: Waveform Results (32-64 ns)















































⊕  RAM[22]	E6	⊕  RAM[45]	30
⊕  RAM[21]	E0	⊕  RAM[44]	47
⊕  RAM[20]	00	⊕  RAM[43]	D0
⊕  RAM[19]	18	⊕  RAM[42]	0B
⊕  RAM[18]	10	⊕  RAM[41]	1A
⊕  RAM[17]	E0	⊕  RAM[40]	20
⊕  RAM[16]	00	⊕  RAM[39]	D0
⊕  RAM[15]	18	⊕  RAM[38]	06
⊕  RAM[14]	08	⊕  RAM[37]	D0
⊕  RAM[13]	92	⊕  RAM[36]	01
⊕  RAM[12]	26	⊕  RAM[35]	32
⊕  RAM[11]	E0	⊕  RAM[34]	07
⊕  RAM[10]	00	⊕  RAM[33]	31
⊕  RAM[9]	16	⊕  RAM[32]	05
⊕  RAM[8]	10	⊕  RAM[31]	E0
⊕  RAM[7]	E0	⊕  RAM[30]	00
⊕  RAM[6]	00	⊕  RAM[29]	36
⊕  RAM[5]	16	⊕  RAM[28]	01
⊕  RAM[4]	08	⊕  RAM[27]	E0
⊕  RAM[3]	82	⊕  RAM[26]	00
⊕  RAM[2]	26	⊕  RAM[25]	36
⊕  RAM[1]	C0	⊕  RAM[24]	00
⊕  RAM[0]	10	⊕  RAM[23]	9D

Figure 3.13: Instruction Memory Contents

3.3.3 Testcase #3



Figure 3.14: Register File Contents

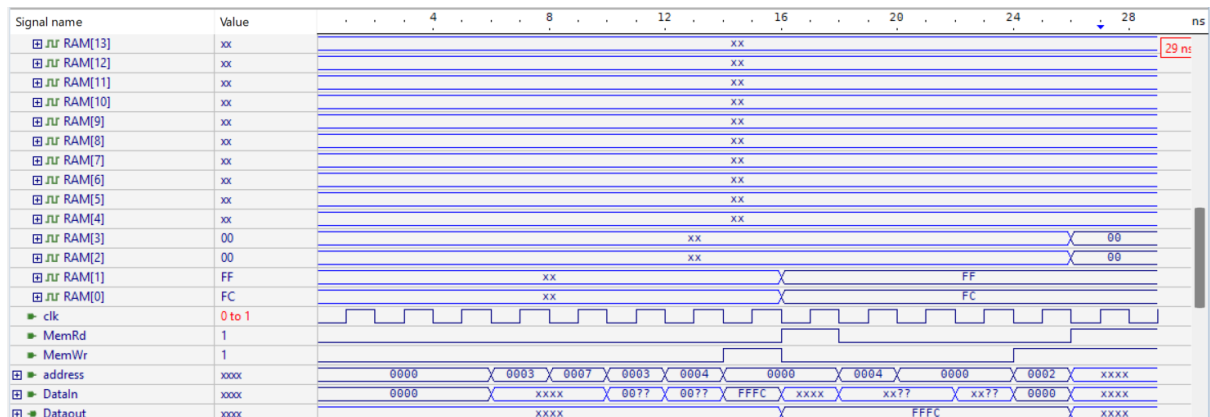


Figure 3.15: Data Memory Contents

⊕ RAM[17]	76
⊕ RAM[16]	02
⊕ RAM[15]	41
⊕ RAM[14]	A0
⊕ RAM[13]	1D
⊕ RAM[12]	60
⊕ RAM[11]	6D
⊕ RAM[10]	00
⊕ RAM[9]	F1
⊕ RAM[8]	FC
⊕ RAM[7]	28
⊕ RAM[6]	98
⊕ RAM[5]	06
⊕ RAM[4]	88
⊕ RAM[3]	32
⊕ RAM[2]	24
⊕ RAM[1]	31
⊕ RAM[0]	03

Figure 3.16: Instruction Memory Contents

3.4 Analysis of Results

The following formula will also be used along the waveforms and testbench to test the correctness. # of clock cycles = # of pipeline stages + # of instructions - 1 + # of stall cycles
of stall cycles = (# of branch or J-type instructions + # of loads with hazards except store after load) * # of stall cycles

3.4.1 Testcase #1

The console output (Figure 3.1) indicates that the simulation passed successfully with the expected write operations performed at the specified addresses and data values. The waveform results (Figures 3.2 to 3.7) show the detailed timing and values of various signals during the simulation. Key observations include:

- **Figure 3.2:**
 - The clk signal toggles correctly with a period of 2 ns.
 - The instruction signals are applied correctly at each clock cycle.
 - Initial instructions and register file (rf) updates are visible.
 - Execution of JMP main and beginning of main instructions are shown.
- **Figure 3.3:**
 - Continued application of instructions.
 - Memory addresses and data transitions are visible.
 - Execution of SW, LW, and LBU instructions are shown.
 - Data output (dataout) shows expected values for load operations.
- **Figure 3.4:**
 - Further progression of instruction execution.
 - Memory read and write operations are highlighted.
 - Register file contents reflect expected changes.
 - Execution of ANDI, CALL sum, and SW instructions are shown.
- **Figure 3.5:**
 - Final stages of instruction execution.
 - Register file updates complete, and memory data output shows the final values.
 - Execution of ADDI instructions and the final SW instruction is shown.
- **Figure 3.6:**
 - Memory read and write operations are highlighted.
 - Changes in RAM contents due to write operations.
 - The MemRd and MemWr signals indicate memory operations.
 - Execution of LBS instruction and its effect on register r1.
- **Figure 3.7:**
 - Final memory state is shown.
 - Accurate read and write cycles are displayed.
- **Figure 3.10**
 - Stored in the instruction memory is the correct sequence of instructions machine code.

Analysis by Calculating the Number of Taken Clock Cycles

of cycles wasted on filling the pipeline = 4 cycles

of executed instructions = 8 cycles

of stall cycles = $2 \times 3 + 4$ cycles

Summing all that up yields:

of clock cycles = $4 + 5 \times 3 + 13 + 8 = 40$

This goes along with the simulation results, where the last instruction `LBs` finishes its write-back stage by writing the stored value to the register `r1`. Note that `r1` changes its value at the time 80 ns. Since the clock period is 2 ns. The # of clock cycles taken from the simulation is $80/2 = 40$, which confirms our claim.

Explanation of the Assembly Code and Hazards Covered

The tested assembly code sequence aims to verify various aspects of the pipelined MIPS processor, including arithmetic operations, memory access, and control flow. Here's an explanation of the code:

- **JMP main:** Jumps to the main label to begin execution.
- **sum:** A subroutine to sum values and decrement a counter.
 - **ADD r5, r5, r3:** Adds the values of `r5` and `r3`, storing the result in `r5`.
 - **ADDI r3, r3, -1:** Decrements the value of `r3` by 1.
 - **BNE r3, r0, -4:** Branches back to the `ADD` instruction if `r3` is not equal to `r0`.
 - **RET:** Returns from the subroutine.
- **main:** The main program execution starts here.
 - **SW r0, 5:** Stores the value 5 at the address in `r0`.
 - **LW r2, r0, 0:** Loads the value from the address in `r0` into `r2`.
 - **SW r2, r0, 2:** Stores the value of `r2` at the address `r0 + 2`.
 - **LBu r3, r0, 2:** Loads a byte from the address `r0 + 2` into `r3`.
 - **ANDI r5, r0, 0:** Performs an AND operation between `r0` and 0, storing the result in `r5`.
 - **CALL sum:** Calls the `sum` subroutine.
 - **SW r5, r0, 4:** Stores the value of `r5` at the address `r0 + 4`.
 - **ADDI r6, r5, -15:** Adds -15 to the value of `r5`, storing the result in `r6`.
 - **ADDI r6, r6, -15:** Adds -15 to the value of `r6`, storing the result in `r6`.
 - **SW r6, r0, 6:** Stores the value of `r6` at the address `r0 + 6`.
 - **LBs r1, r0, 6:** Loads a signed byte from the address `r0 + 6` into `r1`.

Hazards Covered in the Code

- **Data Hazards:**
 - Forwarding paths and stalls were verified through the data dependencies in instructions like `ADDI r6, r5, -15` which depends on the result of `SW r5, r0, 4`.
 - The `sum` subroutine involves multiple dependent instructions.
- **Control Hazards:**
 - The `BNE r3, r0, -4` instruction in the `sum` subroutine introduces a control hazard that is resolved by causing one stall cycle to calculate the branch outcome, both the next PC address and whether the branch is taken or not in the decode stage. The good thing about this branch is that it is calculated as taken the number of

times the sum is done, but the last time is untaken. Thus, this testcase tests both taken and untaken branches.

- The JMP main, CALL sum and RET cause one stall cycle to determine the next PC address in the decode stage.

3.4.2 Testcase #2

The waveform results (Figures 3.11 to 3.13) show the detailed timing and values of various signals during the simulation. Key observations include:

- **Figure 3.11:**
 - The clk signal toggles correctly with a period of 2 ns, with each instruction fetched at the next cycle.
 - The value of r1 first turns 5 as a result of ADDI r1, r0, 5.
 - The value of r2 is 7 as a result of ADDI r2, r0, 7.
 - The value of r3 is 7 as a result of CALL max, since $\max(7,5) = 7$.
- **Figure 3.12:**
 - The value of r5 is 5 as a result of ADD r5, r0, r4.
 - The value of r6 is 0 as a result of CALL sign.
 - The value of r0 is still zero, since any write attempt to it will be discarded.
- **Figure 3.13:**
 - Stored in the instruction memory is the correct sequence of instructions machine code.

Analysis by Calculating the Number of Taken Clock Cycles

of cycles wasted on filling the pipeline = 4 cycles

of executed instructions = 8 cycles

of stall cycles = $2 \times 3 + 4$ cycles

Summing all that up yields:

of clock cycles = $4 + 8 + 3 \times 3 + 2 \times 3 + 4 = 31$

This matches the simulation results, where the last instruction SW r6, r0, 2 finishes its execution at time = 62 ns. The number of clock cycles taken from the simulation is $62/2 = 31$, which confirms our claim.

Explanation of the Assembly Code and Hazards Covered

The tested assembly code sequence aims to verify various aspects of the pipelined MIPS processor, including arithmetic operations, memory access, and control flow. Here's an explanation of the code:

- **JMP main:** Jumps to the main label to begin execution.
- **max:** A subroutine to return the max of two numbers.
 - **BGT r2, r1, 6:** Jumps to the instruction whose address is PC+6 under the condition that r2 is greater than r1.
 - **ADD r3, r0, r1:** Adds the value in r0 (zero) to the value in r1, and the result is stored in r3. It acts like a move instruction.
 - **RET:** Returns from the subroutine; it jumps to the instruction whose address is stored in r7.
 - **ADD r3, r0, r2:** Adds the value in r0 (zero) to the value in r2, and the result is stored in r3. It acts like a move instruction.
 - **RET:** Returns from the subroutine; it jumps to the instruction whose address is stored in r7.
- **min:** A subroutine to return the max of two numbers.
 - **BLT r2, r1, 6:** Jumps to the instruction whose address is PC+6 under the condi-

tion that r2 is less than r1.

- **ADD r4, r0, r1**: Adds the value in r0 (zero) to the value in r1, and the result is stored in r4. It acts like a move instruction.
- **RET**: Returns from the subroutine; it jumps to the instruction whose address is stored in r7.
- **ADD r4, r0, r2**: Adds the value in r0 (zero) to the value in r2, and the result is stored in r4. It acts like a move instruction.
- **RET**: Returns from the subroutine; it jumps to the instruction whose address is stored in r7.
- **sign_of**: A subroutine to return the max of two numbers.
 - **BLTZ r5, 6**: Jumps to the instruction whose address is PC+6 under the condition that r2 is less than r0 (zero).
 - **ADDI r6, r0, 0**: Adds the value in r0 (zero) to the immediate value (zero), and the result is stored in r6. It acts like a load immediate instruction.
 - **RET**: Returns from the subroutine; it jumps to the instruction whose address is stored in r7.
 - **ADDI r6, r0, 1**: Adds the value in r0 (zero) to the immediate value (one), and the result is stored in r6. It acts like a load immediate instruction.
 - **RET**: Returns from the subroutine; it jumps to the instruction whose address is stored in r7.
- **main**: The main program execution starts here.
 - **ADDI r1, r0, 5**: Adds the value in r0 (zero) to the immediate value (five), and the result is stored in r1. It acts like a load immediate instruction.
 - **ADDI r2, r0, 7**: Adds the value in r0 (zero) to the immediate value (seven), and the result is stored in r2. It acts like a load immediate instruction.
 - **CALL max**: Jumps to the instruction whose address is at the label max, and stores the return address in r7.
 - **CALL min**: Jumps to the instruction whose address is at the label min, and stores the return address in r7.
 - **ADD r5, r0, r4**: Adds the value in r0 (zero) to the value in r4, and the result is stored in r5.
 - **CALL sign_of**: Jumps to the instruction whose address is at the label sign_of, and stores the return address in r7.
 - **ADDI r0, r2, 7**: This instruction tries to add r2 to 7 and write the result to r0, but the result would be discarded and r0 remains zero.

Hazards Covered in the Code

- **Data Hazards:**
 - Forwarding paths and stalls were verified through the data dependencies in instructions like **ADDI r6, r5, -15** which depends on the result of **SW r5, r0, 4**.
 - The sum subroutine involves multiple dependent instructions, verifying the hazard detection and resolution logic.
- **Control Hazards:**
 - The branch instructions in the code: **BGT r2, r1, 6**, **BLT r2, r1, 6** and **BLTZ r5, 6** all introduce a control hazard that is resolved by causing one stall cycle to calculate the branch outcome, both the next PC address and whether the branch is taken or not in the decode stage.

- The JMP main, CALL max, CALL min, CALL sign_of and RET cause one stall cycle to determine the next PC address in the decode stage.

3.4.3 Testcase #3

The waveform results (Figures 3.14 to 3.16) show the detailed timing and values of various signals during the simulation. Key observations include:

- **Figure 3.14:**
 - The clk signal toggles correctly with a period of 2 ns, with each instruction fetched at the next cycle.
 - The value of r1 first turns to 3 as a result of `ADDI r1, r0, 3`.
 - The value of r2 becomes 7 as a result of `ADDI r2, r1, 4`.
 - The value of r3 is calculated as 7 from `AND r3, r2, r1`.
 - The value of r4 is calculated as 0 from `SUB r4, r2, r3`.
- **Figure 3.15:**
 - Memory addresses and data transitions are visible.
 - The `SW r6, r0, 2` instruction stores the value of r6 in memory.
 - Data output (dataout) shows expected values for load operations.
 - The value of r5 is loaded from memory as a result of `LBs r5, r0, 0`.
 - The value of r6 is calculated as 7 from `ADD r6, r5, r4`.
- **Figure 3.16:**
 - Memory read and write operations are highlighted.
 - Register file contents reflect expected changes.
 - Execution of `ANDI r1, r5, 0` resets r1 to 0.

Analysis by Calculating the Number of Taken Clock Cycles

of cycles wasted on filling the pipeline = 4 cycles

of executed instructions = 9 cycles

of stall cycles = 1 cycles

Summing all that up yields:

of clock cycles = $4 + 9 + 1 = 14$ cycles

This goes along with the simulation results, where the last instruction `SW r6, r0, 2` finishes its memory stage at time = 26 ns, therefore the write-back stage finishes at 28 ns. The of clock cycles taken from the simulation is $28/2 = 14$, which confirms our claim.

Explanation of the Assembly Code and Hazards Covered

The tested assembly code sequence aims to verify various aspects of the pipelined MIPS processor, including arithmetic operations, memory access, and control flow. Here's an explanation of the code:

- **ADDI r1, r0, 3:** Adds the immediate value 3 to r0 (zero), and stores the result in r1.
- **ADDI r2, r1, 4:** Adds the immediate value 4 to the value in r1, and stores the result in r2.
- **AND r3, r2, r1:** Performs a bitwise AND operation between r2 and r1, storing the result in r3.
- **SUB r4, r2, r3:** Subtracts the value in r3 from r2, and stores the result in r4.
- **Sv r0, -4:** Stores the value of r0 into memory at address -4.
- **LBs r5, r0, 0:** Loads a byte from memory address 0 into r5.
- **ADD r6, r5, r4:** Adds the values in r5 and r4, and stores the result in r6.
- **ANDI r1, r5, 0:** Performs a bitwise AND operation between r5 and the immediate value 0, storing the result in r1.

- **SW r6, r0, 2:** Stores the value of r6 into memory at address 2.

Hazards Covered in the Code

- **Data Hazards:**
 - Forwarding paths and stalls were verified through the data dependencies in instructions like `ADDI r2, r1, 4` which depends on the result of `ADDI r1, r0, 3`.

4 Teamwork

- **Design Process and Diagram:**
 - The design process was performed in all parts by us all, including brainstorming and block design. Register transfer notation was divided as follows: Ameer wrote the register transfer notation of R-type instructions, Ghazi wrote the I-type instructions, and Mumen wrote both the J-type and S-type instructions. Ameer helped overseeing that the design diagram is correct and neat.
- **Implementation in Verilog:** The implementation process was discussed and shared among us all. Even though other partners did different things, everyone in the team agreed on a unified implementation procedure beforehand.
 - **Ghazi:** top, cpu, datapath, controller and hazard modules. Ghazi also helped overseeing the implementation process and that everything was done right.
 - **Ameer:** memories, pc control and hazard modules. He also helped in debugging the datapath module.
 - **Mumen:** main control unit, register file and other basic modules.
- **Simulation and Testing:**
 - **Ghazi:** wrote the testbench in Verilog and provided the testcases in the report.
 - **Ameer, Mumen:** came up with an idea for a testbench to be thorough enough and provided additional testcases.
- **Report Writing:**
 - **Mumen, Ameer:** wrote the theory and procedure sections. Mumen also helped in overseeing the report layout, contents and style.
 - **Ghazi:** wrote the verification section.

5 Conclusion

In this project, we successfully designed and implemented a 16-bit pipelined RISC processor with a comprehensive instruction set architecture (ISA) that supports R-type, I-type, J-type, and S-type instructions. Our processor features a 5-stage pipeline, including instruction fetch, decode, execute, memory access, and write-back stages, ensuring efficient and systematic instruction execution.

Throughout the design process, we emphasized the importance of a detailed and accurate datapath and control path, incorporating essential components and control signals to achieve desired functionality. We addressed potential hazards in the pipeline by implementing effective hazard handling mechanisms.

Verification of our processor involved rigorous testing using a comprehensive testbench and multiple code sequences. The simulation results validated the correctness and completeness of the processor, confirming its capability to accurately execute the designed instruction set.

This project demonstrates a structured approach to processor design, emphasizing the principles of correctness, completeness, and efficient execution. Our implementation showcases a robust and scalable pipelined processor architecture that meets the specified requirements, providing a solid foundation for further enhancements and extensions in future projects.