Birzeit University
Faculty of Engineering & Technology
Department of Electrical & Computer Engineering

Pipelined Processor Design

Prepared By:
Mohammed Milhem: 1211053
Faris Abu Farha: 1200546
Tareq Zaghal: 1210158

Supervised By:
Dr. Ayman Hroub

A Computer architecture final project submitted for the ENCS4370 course

Birzeit
January 2025

Abstract

This report presents a simple pipelined Reduced Instruction Set Computer (RISC) processor design and verification using Verilog hardware description language. The processor features a 16-bit architecture with three distinct instruction types: R-Type, I-Type, and J-Type, supported by an instruction set for modular design. The datapath and control path were developed to facilitate efficient instruction execution, integrating a 16-bit program counter (PC), return register (RR), and eight general-purpose registers.

The processor includes dedicated physical memories for instructions and data, with support for word-addressable memory and performance monitoring registers. These registers track key execution metrics such as the number of executed instructions, clock cycles, and pipeline stall cycles. A five-stage pipelined architecture was employed, instruction fetch, decode, execution, memory access, and write-back stages, designed to optimize performance.

Verification was conducted using a comprehensive test bench, simulating various instruction sequences and monitoring execution outputs to ensure functional correctness. Design choices, including control signal generation and modular implementation, were validated against simulation results, demonstrating alignment with project specifications.

This report outlines the design methodology, implementation details, simulation results, and testing approach, providing a detailed account of the project's development and achievements.

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Theoretical Background

This chapter presents a brief background on the project requirements and the necessary theoretical knowledge.

Contents

1.1 Set of Operations

A minimal set of operations is implemented, maintaining the simplicity and flexibility of the processor instruction set. These operations are categorized to arithmetic, logical, memory access, and control flow. These categories are used for different purposes, allowing the processor to perform 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.
- **ADDI (Add Immediate):** Computes the sum of a register and an immediate value.
- **SUB (Subtraction):** Computes the difference between two registers.

1.1.2 Logical Operations

Logical operations perform bitwise manipulation, crucial for tasks such as masking and set- ting 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.
- **SLL (Shift Left Logical):** Computes the logical shift left of the value in register Rs by the number of bits specified in register Rt and stores the result in register Rd.
- **SRL (Shift Right Logical):** Computes the logical shift right of the value in register Rs by the number of bits specified in register Rt and stores the result in register Rd.

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 a memory address added to an immediate, into a register.
- **SW (Store Word):** Stores a word from a register into a memory address added to an immediate.

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.
- **BEQ (Branch if Equal):** Branches if two registers are equal.
- **BNE (Branch if Not Equal):** Branches if two registers are not equal.
- **FOR Rs, Rt:**

- Initiates a loop with a target address stored in Rs and the number of iterations stored in Rt.
- Rs stores the loop target address, i.e., the address of the first instruction in the loop block.
- Rt stores the initial number of the loop iterations, i.e., the initial value of the loop counter.
- The Rt register is decremented at the end of each iteration. The loop exits when the content of the Rt register becomes zero.
- The immediate field is ignored in this instruction.

1.2 Instruction Set Architecture (ISA)

An Instruction Set Architecture(ISA) is an interface between the software that runs on the CPU and its underlying hardware. Defining the set of instructions a processor can execute. The ISA specifies 16-bit architecture, and defines four instruction types : R-type instructions, I-type instructions, J-type instructions and S-type instructions. These instructions are operations of various types such as arithmetic, logical, memory access, and control flow.

1.2.1 R-type Instructions

R-Type (Register Type) instructions are a class of instructions in which the operation is performed between registers. The instruction format consists of a 4-bit opcode, which is set to zero for all R-Type instructions, followed by a 3-bit destination register (Rd), a 3-bit first source register (Rs), a 3-bit second source register (Rt), and a 3-bit function field. The function field specifies the specific operation to be executed, such as addition, subtraction, or logical operations. These instructions typically involve arithmetic or logical operations between the contents of registers and store the result in the destination register. Figure 1.1 shows the R-Type instruction.

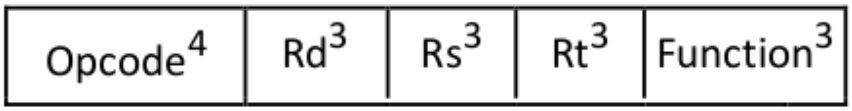

Figure 1.1: R-Type instruction

1.2.2 I-type Instructions

I-Type instructions involve an operation between a register and an immediate value. The format includes a 4-bit opcode, a 3-bit source register (Rs), a 3-bit destination register (Rt), and a 6-bit signed immediate value. For logical instructions, the immediate value is zero-extended, while for others it is sign-extended. In branch instructions like BEQ and BNE, the branch target is calculated by adding the sign-extended immediate to the current program counter (PC). Figure 1.2 shows the I-Type instruction.

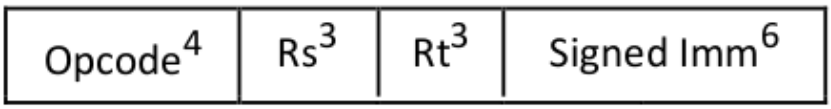

Figure 1.2: I-Type instruction

1.2.3 J-Type

J-Type instructions are used for jumps and include a 4-bit opcode (set to 1 for J-Type), a 3-bit function field defining the specific operation, and a 9-bit offset. The jump target address is calculated by concatenating the upper bits of the program counter (PC[15:9]) with the 9-bit offset. Figure 1.3 shows the J-Type instruction.

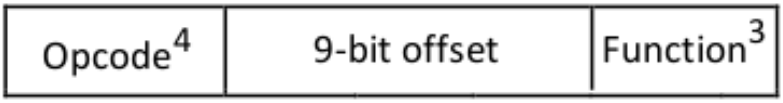

Figure 1.3: J-Type instruction

1.3 Processor Architecture

The processor architecture consists of eight 16-bit general-purpose registers (R0 to R7). It also features a 16-bit program counter (PC) and a return register (RR) for storing the return address during function calls. The processor includes separate data and instruction memories, both of which are word-addressable. The memory uses little-endian byte ordering, and the processor supports three types of instructions: R-Type, I-Type, and J-Type, facilitating modular design and efficient execution.

1.4 Pipeline Stages

The processor uses a 5-stage pipeline to improve instruction performance: fetch, decode, execute, memory access, and write back. Every stage in it is on a special section of the processing instructions.

1. **Fetch:** The fetch stage fetches the next instruction from the instruction memory using the program counter (PC).
2. **Decode:** In the decode stage, the fetched instruction is interpreted to determine the opcode, immediate values, and the source and destination registers.
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 guides 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 executes the arithmetic and logical operations required from different instructions.

1.7 Verification

Involves writing a testbench, simulating the operation of the processor and validating its correctness if the processor is functioning correctly. A variety of test programs are run to verify that every instruction functions properly, with results measured against expected outcomes.

Chapter 2

Design and Implementation

Contents

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 table shows the detailed RTL (Register Transfer Level) description for each instruction type, explaining the step-by-step process.

Instruction Type	**Stage**	**RTL**
R-type	IF	$inst = instMem[PC]$ $PC = PC + 1$
	ID	$AluOperand1 = Reg[inst[8:6]]$ $AluOperand2 = Reg[inst[5:3]]$ $DestinationAddress = inst[11:9]$ $Opcode = inst[15:12]$ $function = inst[2:0]$
	EX	$AluResult = Operation(AluOperand1, AluOperand2)$
	MEM	Not required
	WB	$RegFile[DestinationAddress] = AluResult$
I-type (ALU)	IF	$inst = instMem[PC]$ $PC = PC + 1$
	ID	$Immediate = zero\_extend(inst[5:0])$ (for ANDI) $Immediate = sign\_extend(inst[5:0])$ (for ADDI) $Op1 = Reg[inst[8:6]]$ $DestinationAddress = inst[11:9]$ $Opcode = inst[15:12]$
	EX	$AluResult = Op1 + Immediate$ (for ADDI) $AluResult = Op1 \& Immediate$ (for ANDI)
	MEM	Not required
	WB	$RegFile[DestinationAddress] = AluResult$
I-type (Branch)	IF	$inst = instMem[PC]$
	ID	$Immediate = sign\_extend(inst[5:0])$ $AluOperand1 = Reg[inst[8:6]]$ $AluOperand2 = inst[11:9]$ $Opcode = inst[15:12]$
	EX	$AluResult = Sub(AluOperand1, AluOperand2)$ $(Flags for branch decision updated here)$
	MEM	Not required
	WB	$RegFile[DestinationAddress] = AluResult$
I-Type (Load/ Store)	IF	$inst = instMem[PC]$ $PC = PC + 1$
	ID	$Immediate = sign\_extend(inst[5:0])$ $AluOperand1 = inst[11:9]$ $Opcode = inst[15:12]$

Instruction Type	**Stage**	**RTL**
	EX	$AluResult = AluOperand1 + Immediate$
	MEM	$LW : MemData = DMem[AluResult]$ $SW : DMem[AluResult] = RegFile[DestinationAddress]$
	WB	$LW : RegFile[DestinationAddress] = MemData$
J-type	IF	$inst = instMem[PC]$
	ID	$Opcode = inst[15:12]$ $ReturnAddress = PC$ $JumpOffset = inst[11:3]$ $function = inst[2:0]$ $RET : PC = RR$ $CALL : PC = PC[15:9], JumpOffset$ $RR = currentPc$ $JMP : PC = PC[15:9], JumpOffset$
	EX	Not required
	MEM	Not required
	WB	Not required
I-type (FOR)	IF	$inst = instMem[PC]$
	ID	$BranchAddress = reg(inst[11:9])$ $ALUoperand1 = reg(inst[8:6])$ $Opcode = inst[15:12]$ $(flagZero == 0) : PC = bus2$
	EX	$ALUres = Decrement(ALUoperand1)$
	MEM	Not required
	WB	Not required

Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) are the five stages of the processor design pipeline. These steps are followed by every kind of instruction, guaranteeing a well-organised and effective execution flow. The presented RTL description illustrates the systematic approach to processor design by outlining the precise operations carried out at every stage for different instruction types.

2.2 Single Cycle Datapath

The single-cycle datapath plays a crucial role in the CPU's architecture, allowing the processor to complete the execution of each instruction within a single clock cycle. The diagram shows the key components and connections that make this functionality possible. Figure 2.1 outlines the main components of the data path and describes how components interact.

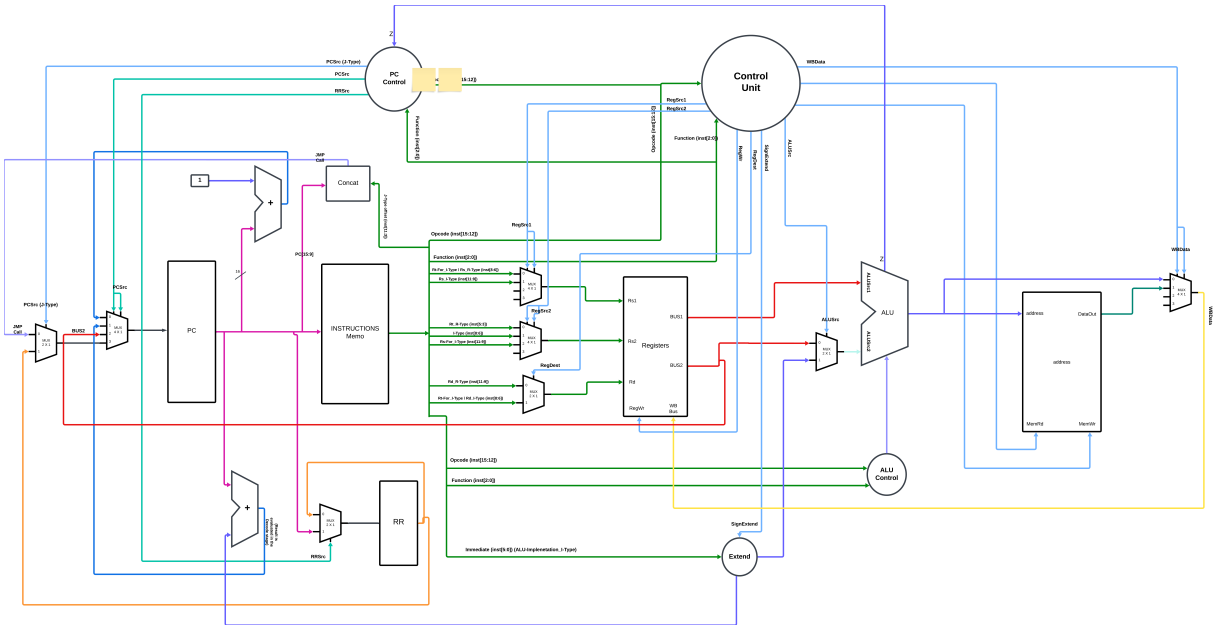

Figure 2.1: Single cycel datapath

2.2.1 Instruction Fetch (IF)

- **Program Counter (PC)**: The Program Counter holds the address of the next instruction to be retrieved from the instruction memory. It automatically updates each clock cycle to point to the subsequent instruction.
- **Instruction Memory**: This component stores the set of instructions that the CPU executes. During the fetch stage, the instruction located at the address specified by the PC is retrieved.
- **Adder**: An adder is responsible for incrementing the PC by 1 to point to the next instruction since each instruction is 16 bits (equivalent to a word).

2.2.2 Instruction Decode (ID)

- **Registers**: The register file comprises eight general-purpose 16-bit registers ($R0$ to $R7$), along with a dedicated return register (RR). During this stage, the source registers ($Rs1$ and $Rs2$) specified in the instruction are accessed. For jump or call instructions, the return register is also used to store the address to which control should return.
- **Instruction Fields**: The instruction is divided into several fields, including the opcode, source registers, destination register, and immediate values, which are utilized in subsequent processing stages.

2.2.3 Execution (EX)

- **ALU**: The Arithmetic Logic Unit carries out the arithmetic and logical operations specified by the opcode. The operands are derived from the registers or immediate values, and the ALU produces the result accordingly.

- **Extender**: For instructions of type I, the extender either sign-extends or zero-extends the immediate value to match the operand size needed for processing by the ALU.

2.2.4 Memory Access (MEM)

- **Data Memory (DMEM)**: This stage handles data memory access for load and store instructions. The ALU calculates the address, and data is either retrieved from or written to this location.
- **Multiplexer (MUX2_1)**: A multiplexer determines which data source should be selected for storage in the data memory or the register file.

2.2.5 Write Back (WB)

- **Register File**: The final stage involves writing the result obtained from the ALU or the data memory back into the register file, completing the instruction cycle.

2.2.6 Control Unit

- **Control Signals**: Based on the instruction's opcode, the control unit generates appropriate signals to regulate the multiplexers, ALU, memory, and register file operations, ensuring accurate execution of the instruction.

2.2.7 Branch and Jump Handling

- **PC Control**: For branch and jump instructions, the Program Counter (PC) is updated based on the target address, which is calculated by the ALU or derived from immediate values. In the case of 'CALL' instructions, the return address (the address of the instruction immediately after the call) is stored in the return register (RR).
- **Return Register**: For 'RET' instructions, the return register (RR) is used to restore the PC, allowing the processor to continue execution from the instruction following a subroutine call.
- **Branch Condition**: The ALU generates condition flags (e.g., zero, carry, overflow) that determine whether branch instructions should be executed.

2.2.8 Resolving branch and for instructions

- During the decode stage, a multiplexer (MUX) is utilized to select between branch signals.
- For branch instructions, the subtractor computes $R1 - R2$.
- If the instruction is not a branch, the value in $R1$ is decremented.
- A zero flag (Z flag) is generated and passed to the program counter (PC), where it is appropriately handled.

2.2.9 Summary

The single-cycle datapath architecture ensures that each instruction is executed within one clock cycle. It is a straightforward yet effective design for fundamental processors. The accompanying figure demonstrates how the components are interconnected to facilitate the efficient execution of instructions in a single cycle.

2.3 Single Cycle Control Units

The control unit is responsible for producing the required control signals to coordinate the processor's operations. It interprets the instruction's opcode and configures the control signals for various components like the ALU, memory, and registers accordingly.

2.3.1 Main Control Signals

The main control signals generated by the control unit are summarized in Figure 2.2.

Instruction	Opcode	Func	RegSrc1	RegSrc2	RegDest	RegWr	ALUSrc	WBData	SignExtend	MemRd	MemWR	Branch_OR_For	NoOp	ALUInst	CntInst
AND	0000	000	0	00	0	1	0	0	x	0	0	x	0	1	0
ADD	0000	001	0	00	0	1	0	0	x	0	0	x	0	1	0
SUB	0000	010	0	00	0	1	0	0	x	0	0	x	0	1	0
SLL	0000	011	0	00	0	1	0	0	x	0	0	x	0	1	0
SRL	0000	100	0	00	0	1	0	0	x	0	0	x	0	1	0
ANDI	0010	NA	1	xx	1	1	1	0	0	0	0	x	0	1	0
ADDI	0011	NA	1	xx	1	1	1	0	1	0	0	x	0	1	0
LW	0100	NA	1	xx	1	1	1	1	1	1	0	x	0	0	0
SW	0101	NA	1	xx	x	0	1	x	1	0	1	x	0	0	0
BEQ	0110	NA	1	01	x	0	0	x	1	0	0	0	0	0	1
BNE	0111	NA	1	01	x	0	0	x	1	0	0	0	0	0	1
FOR	1000	NA	0	10	1	1	x (add decrement AlUop)	0	x	0	0	1	0	0	1
JMP	0001	000	x	xx	x	0	x	x	x	0	0	x	0	0	1
CALL	0001	001	x	xx	x	0	x	x	x	0	0	x	0	0	1
RET	0001	010	x	xx	x	0	x	x	x	0	0	x	0	0	1
NoOP(Buble)	1111	x	x	xx	x	0	x	x	x	0	0	x	1	0	0

Figure 2.2: Main Control Unit Signals

- **Instruction**: The operation or command that the CPU needs to execute (e.g., ADD, SUB).
- **Opcode**: The binary code that represents the specific operation.
- **Func**: A field used in some instructions to specify additional operation details (e.g., function of the ALU).
- **RegSrc1**: Specifies the first source register for the instruction (e.g., Rs1 in R-type instructions).
- **RegSrc2**: Specifies the second source register (e.g., Rs2 in R-type instructions).

- **RegDest**: Specifies the destination register where the result will be stored (e.g., Rd in R-type instructions).
- **RegWr**: Indicates whether a register write operation occurs.
- **ALUsrc**: Determines the second operand for the ALU.
- **WBData**: Specifies the source of data for writing back to the register file .
- **SignExtend**: Indicates whether the immediate value should be sign-extended.
- **MemRd**: Indicates if data memory should be read.
- **MemWR**: Indicates if data memory should be written to.
- **Branch_OR_For**: Determines if the instruction is a branch or a jump operation.
- **NoOp**: Specifies whether the instruction is a no-operation instruction.
- **ALUinst**: Specifies the ALU operation to be performed.
- **Cntlnst**: Specifies the control signals to configure the datapath elements.
- **StoreALUOpFw**: used for forwarding from memory stage to input of ALU stage for execution.
- **StoreDataInFw**: used for forwarding from memory stage to DataIn ALU stage for execution.
- **ForwardBus1,2**: it is a mux selection to select from the different forwarding stages in the decode stage.

2.3.2 ALU Control

The ALU control unit produces signals that dictate the ALU's operation according to the opcode of the instruction. Table 2.2 provides a summary of these ALU control signals.

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. Table 2.3 summarizes the PC control signals:

Table 2.2: ALU Operation Unit

Instruction	**Opcode**
AND	and(000)
ADD	add(001)
SUB	sub(010)
SLL	sll(011)
SRL	srl(100)
ANDI	and(000)
ADDI	add(001)
LW	add(001)
SW	add(001)
BEQ	x (resolved in D)
BNE	x (resolved in D)
FOR	Decrement(101)
JMP	x
CALL	x
RET	x

Table 2.3: PCsrc Control Unit

ID/Opcode	**Flags**	**PCSrc**	**PCSrc(J-Type)**	**RRSrc**
BEQ	(Z=1)	01	x	0
BNE	(Z=0)	01	x	0
FOR	(Z=0)	10	x	0
JMP	x	01	0	0
CALL	x	11	0	1
RET	x	11	1	0
ALL REST	x	00	x	0

- **ID/Opcode**:
  - Represents the **instruction identifier** or **opcode** for each instruction.
  - Specifies the **type of instruction** (e.g., BEQ, BNE, JMP, etc.) or its corresponding opcode value.
  - Helps identify the instruction being executed in the processor.
- **Flags**:
  - Indicates the **condition or status flags** used by the instruction.
  - Examples:
    - `(Z=1)` means the instruction checks if the **zero flag** is set (e.g., BEQ).
    - `(Z=0)` means the instruction checks if the **zero flag is not set** (e.g., BNE).
  - Flags are used for **conditional operations** like branching.
- **PCSrc**:
  - Specifies the **source of the next Program Counter (PC)** value.
  - Determines how the PC is updated for the next instruction.
  - Examples:
    - `01` means the PC is updated based on a **branch target**.
    - `10` means the PC is updated for a **loop instruction**.
    - `00` means the PC is updated normally (`PC + 1`).
- **PCSrc(J-Type)**:
  - Specific to **J-type instructions** (jump, call, return).
  - Indicates how the PC is updated for jump-related instructions.
  - Examples:
    - `0` means the PC is updated based on a **jump target**.
    - `1` means the PC is updated using the **return address** stored in the Return Register (RR).
- **RRSrc**:
  - Specifies the **source for updating the Return Register (RR)**.
  - The RR is used to store return addresses during function calls.
  - Examples:
    - `0` means the RR is **not updated**.
    - `1` means the RR is updated with the **return address** (e.g., during a CALL instruction).

Table 2.4: Instruction Control Signals

ID/Opcode	KillF
Branch	1
J-type	1
FOR	1
ELSE	0

2.3.4 Instruction Control

Table 2.4 describes control signals for specific instructions or instruction types, focusing on the KillF signal for pipeline control.

- **ID/Opcode**:
  - Represents the **instruction identifier** or **opcode**.
  - Examples: `Branch`, `J-type`, `FOR`, `ELSE`.
- **KillF**:
  - Indicates whether the instruction **kills the fetch stage**.
  - `1` means the fetch stage is killed.
  - `0` means the fetch stage is not killed.

2.4 Pipelining

2.4.1 Datapath

The processor's pipeline datapath is divided into five distinct stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage is responsible for specific tasks that enable the efficient execution of instructions. Detailed descriptions of each stage, along with their respective stage buffers, are provided below, followed by an explanation of how the pipeline addresses hazards. Figure 2.3 shows the implemented datapath.

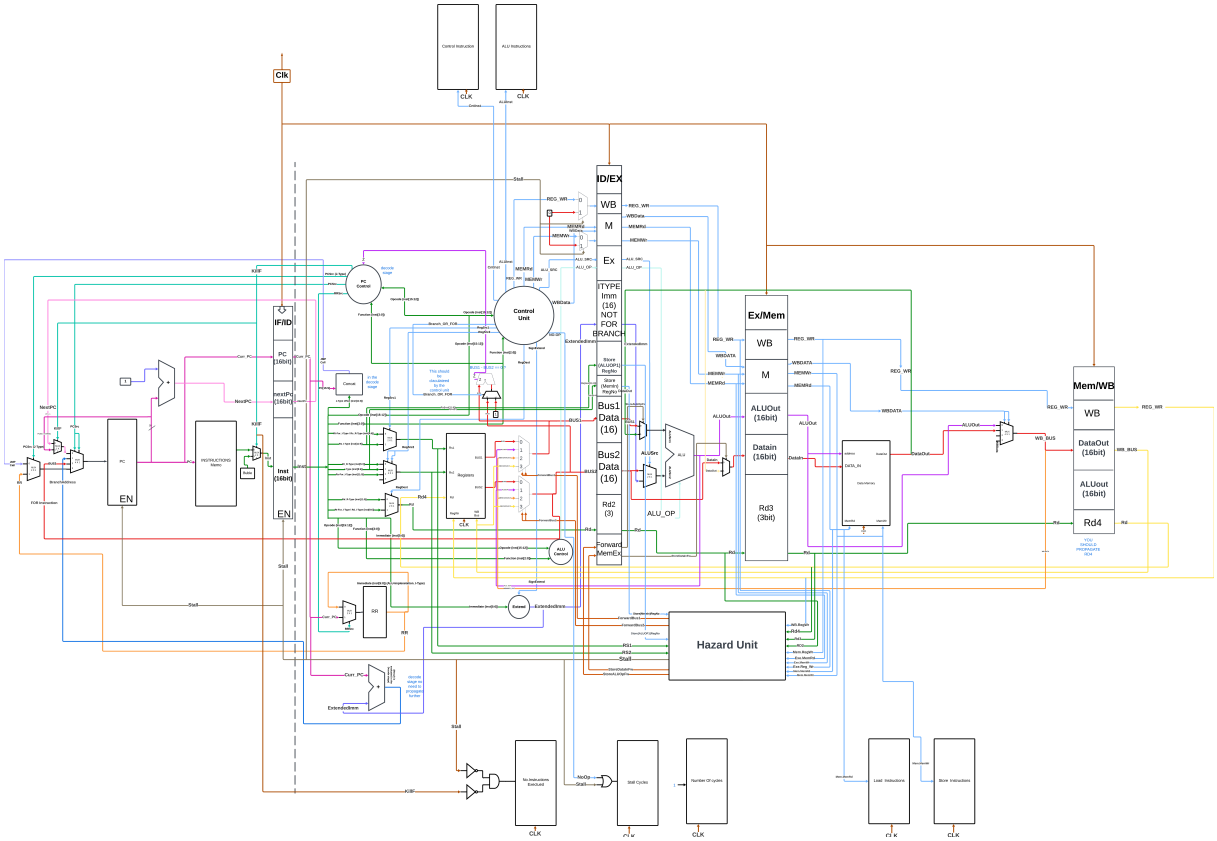

Figure 2.3: Pipelined datapath of the 16-bit Processor

2.4.2 Instruction Fetch (IF)

The Instruction Fetch stage is tasked with obtaining the next instruction from instruction memory. Its key components include the Program Counter (PC) and the Instruction Memory. Components of the IF is shown in Figure 1.2.

- **Program Counter (PC):** The Program Counter holds the address of the next instruction to be fetched. It is updated based on the value of the *PCSrc* flag, a 2-bit signal generated by the PC control logic during the Instruction Decode (ID) stage. The *PCSrc* flag determines the source for the next PC value, selecting from four possible inputs:

– **Input 0:** The next sequential instruction address, PC + 1, for regular instruction flow.
– **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 RR, used for the *RET* (return from subroutine) instruction.

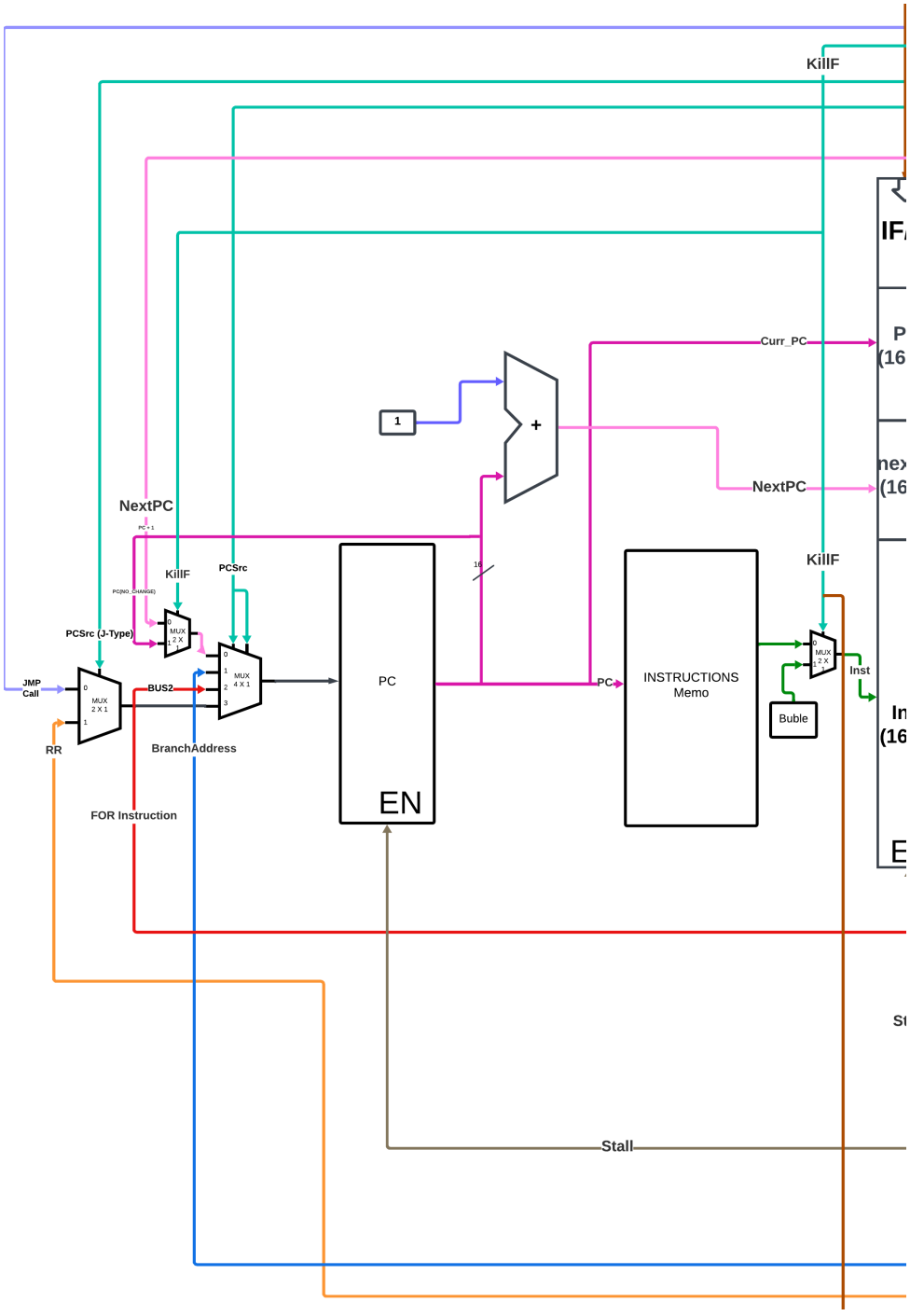

Figure 2.4: Pipeline Data Path Instruction Fetch Stage

- **Instruction Memory:** Responsible for storing all the processor's instructions. The instruction at the address specified by the Program Counter (PC) is retrieved during this stage.
- **Adder:** Updates the PC by adding 1 to its current value, thereby pointing to the subsequent instruction.
- **2-to-1 Multiplexer (16-bit output, 1-bit stall input):** Selects the instruction to be passed to the IF/ID buffer. It takes two possible inputs:
  - **Input 0:** The normal instruction fetched from the instruction memory.
  - **Input 1:** A bubble (no-operation) instruction, utilized to stall the pipeline when required.

  The stall signal determines which input is chosen.

Operations:

- The Program Counter (PC) provides the address to access the instruction from the memory.
- The instruction is fetched from the Instruction Memory.
- The Adder increments the PC by 1, preparing it for the next instruction fetch.

Stage Buffer:

- **IF/ID Register:** Stores the fetched instruction, the updated PC value for the next stage, and the original PC for reference.

2.4.3 Instruction Decode (ID)

During the Instruction Decode stage, the fetched instruction is analyzed to determine the opcode, identify the source and destination registers, and extract any immediate values. The register file is then accessed to retrieve the data stored in the source registers. Figure 2.5 shows the Instruction Decode stage components.

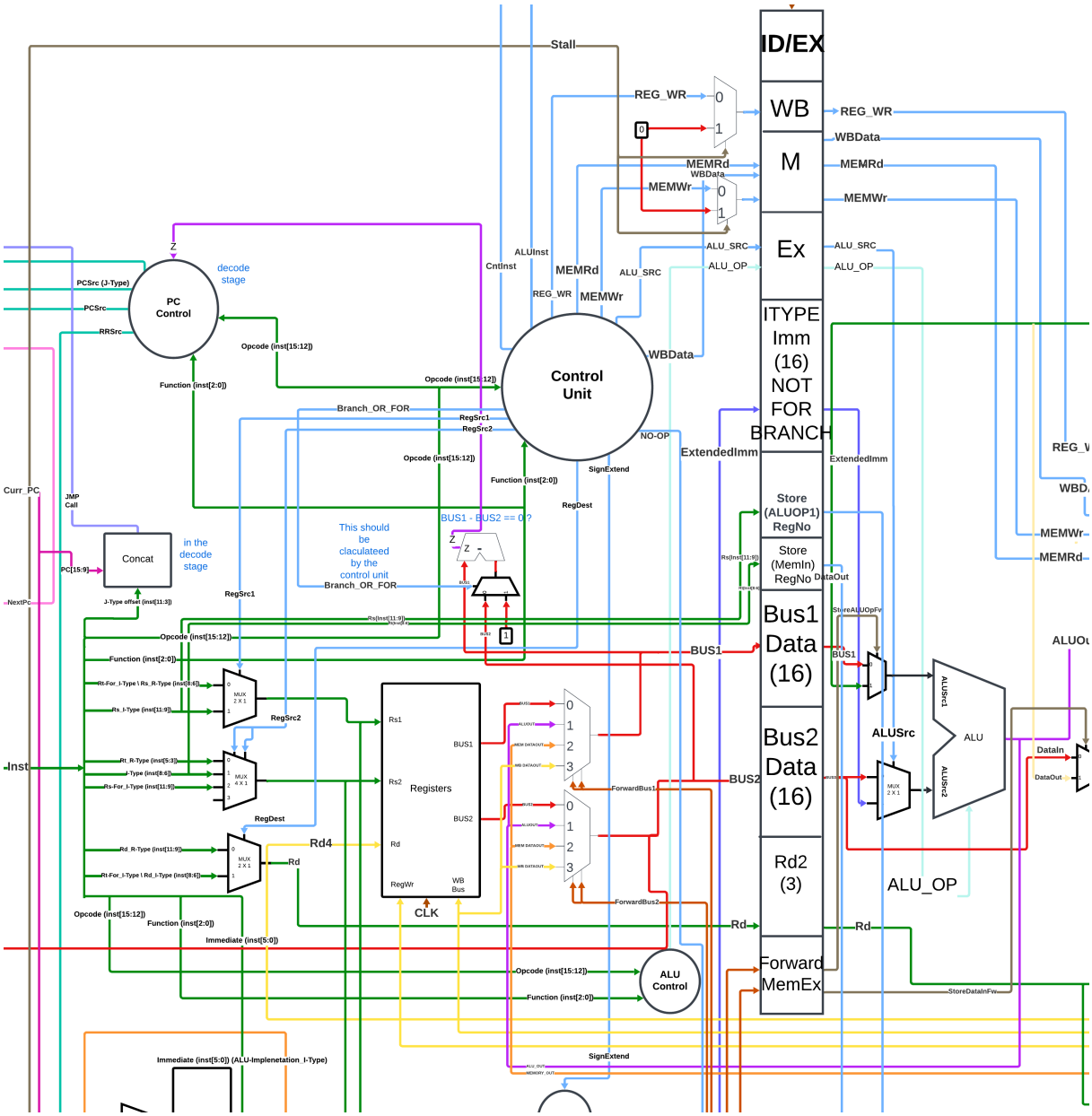

Figure 2.5: Pipeline Data Path Instruction Decode Stage

Components:

- **Register File:** Contains eight 16-bit general-purpose registers (R_0 to R_7). This serves as the storage location for temporary data required during execution.

- **Control Unit:** Responsible for generating control signals based on the decoded instruction, ensuring the processor components execute their assigned tasks in coordination.

- **PC Control:** Manages the Program Counter (PC) by generating control signals based on the decoded instruction. It ensures the PC is updated correctly to maintain the proper execution sequence. Figure 2.5 shows the implemented Instruction

Decode stage.

- **Multiplexers:**
  - **RegSrc1 2x1 Mux:** Determines the source for R_s2:
    - **0:** Instruction[8:6] (Rt-For I-Type Rs R-Type).
    - **1:** Instruction[11:9] (Rs I-Type).
  - **RegSrc2 4x1 Mux:** Determines the source for R_s1:
    - **0:** Instruction[5:3] (R-TYPE).
    - **1:** Instruction[8:6] (I-TYPE).
    - **2:** Instruction[11:9](Rs-For I-Type)
    - **3:** 0.
  - **RegDest 2x1 Mux:** Selects the destination register (R_d):
    - **0:** Instruction[11:9] (Rd R-Type).
    - **1:** Instruction[8:6] (Rt-For I-Type Rd I-Type).
  - **ForwardBus1 Mux:** Selects the Bus1 Data input for ID/EX:
    - **0:** Bus1 (output of the register file).
    - **1:** ALU result.
    - **2:** Memory output.
    - **3:** Write-Back bus.
  - **ForwardBus2 Mux:** Selects the Bus2 Data input for ID/EX:
    - **0:** Bus2 (output of the register file).
    - **1:** ALU result.
    - **2:** Memory output.
    - **3:** Write-Back bus.
- **RegWr 2X1 Mux:** Passes the RegWr signal:
  - **0:** RegWr signal.
  - **1:** 0.
- **MemWr 2X1 Mux:** Passes the MemWr signal:
  - **0:** MemWr signal.
  - **1:** 0.
- **Sign Extender:** Extends a 6-bit immediate value to a 16-bit signed format, ensuring compatibility for arithmetic operations.

- **ALU Control:** Generates control signals for the ALU based on the instruction opcode, directing it to perform specific operations like addition, subtraction, or logical tasks.
- **Adder Component:** Computes the target address for branch instructions, ensuring accurate branching during execution.
- **Subtractor Component:**
  - Assists PC control by providing arithmetic signals for branch predictions.
  - Computes branch target addresses, similar to the adder but for arithmetic cases.

Operations:

- **Instruction Decoding:** Splits the instruction into opcode, source registers, destination register, and immediate value, enabling proper execution flow.
- **Register Access:** Retrieves values from the source registers in the register file for use in operations.
- **Control Signal Generation:** Produces control signals to manage components like the ALU, multiplexers, and PC control.
- **Sign Extension:** Converts immediate values to a 16-bit format, aligning data size for uniform processing.
- **ALU Configuration:** Sets the ALU operation based on control signals to perform tasks required by the instruction.
- **Branch Target Calculation:** Computes branch and jump addresses using adder and subtractor components.
- **Multiplexer Control:** Selects appropriate data paths to ensure correct inputs for components during execution.

Stage Buffer:

- **ID/EX Register:** Holds the decoded opcode, source register values, and additional information, ensuring seamless data transfer between the Instruction Decode (ID) and Execute (EX) stages.

2.4.4 Execution (EX)

The Execution stage handles the arithmetic and logical operations defined by the instruction. Key components in this stage include the ALU, multiplexers, and forwarding units, which work together to execute the required operations efficiently. Figure 2.6 shows the implemented execution stage components.

Components:

- **ALU:** The Arithmetic Logic Unit (ALU) executes arithmetic and logical operations as determined by the 2-bit ALUOp control signal, which specifies the operation type. The ALU processes two input operands:
  - **ALUSrc1:** The first operand, sourced from the ALUsrc1MUX. This operand may be directly forwarded or retrieved from the register file.
  - **ALUSrc2:** The second operand, also sourced from the previous stage buffer. A 2-to-1 multiplexer with a 16-bit output determines whether this operand is taken from the previous buffer or is an immediate value from I-type instructions.
- **2-to-1 Multiplexer (ALUSrc1):** Selects the first ALU operand from Bus2 data, or forwarded values.
- **2-to-1 Multiplexer (ALUSrc2):** Selects the second ALU operand from Bus2 data, immediate values, or other forwarded values.
- **Hazard Unit:** The Hazard Unit identifies and addresses hazards to guarantee proper instruction execution. It produces the required control signals and assesses the need for forwarding. By directing the appropriate data to the ALU inputs, the Hazard Unit efficiently resolves data hazards.
- **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 previous stage buffer.
  - **Input 1:** The forwarded value.

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).
  - Control flags required for the memory and write-back stages.

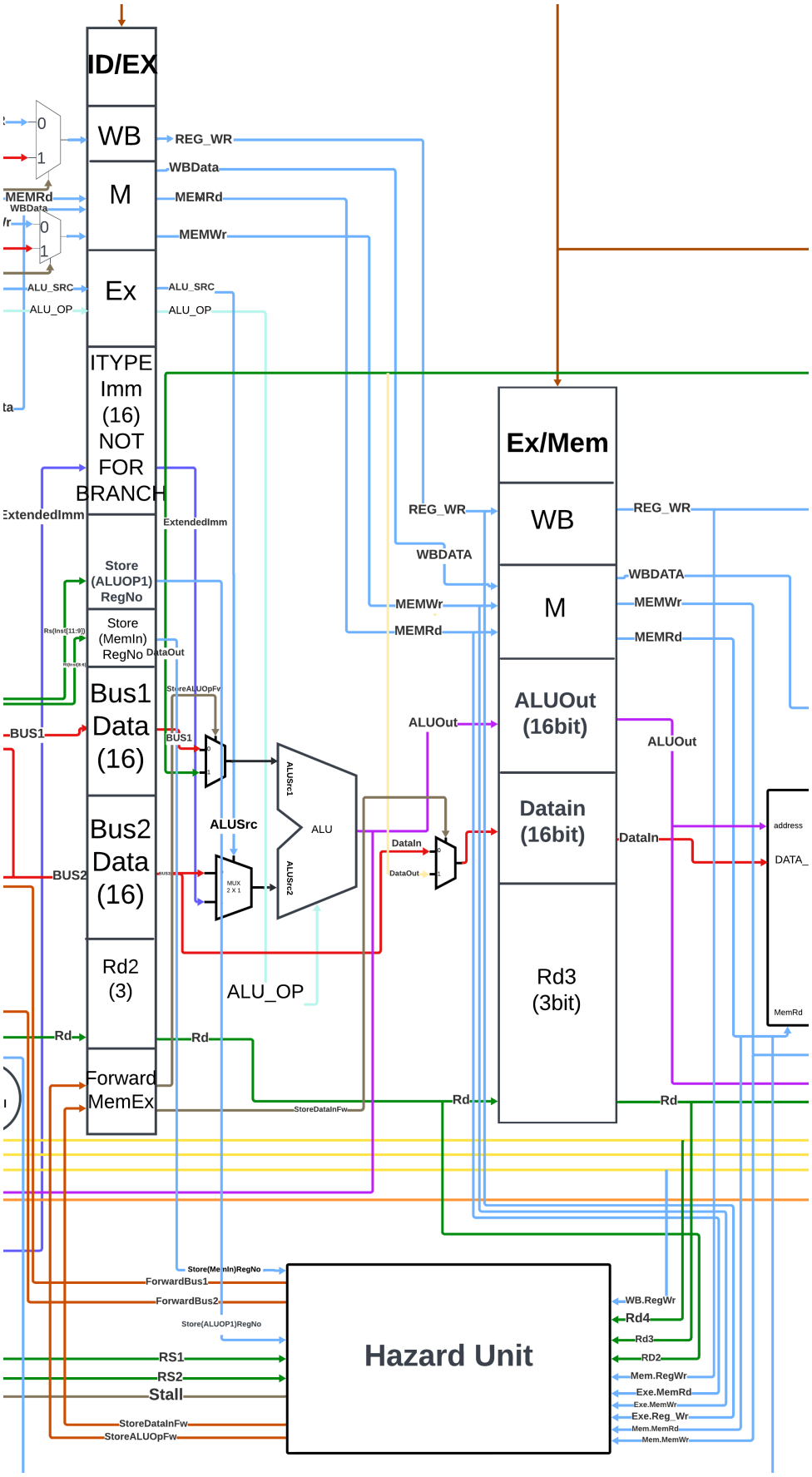

Figure 2.6: Pipeline Data Path Execution Stage

2.4.5 Memory Access (MEM)

During the Memory Access stage, the data memory is utilized to perform load and store operations. The address, determined by the ALU, is used to either retrieve data from or write data to the memory. This stage plays a vital role in managing memory operations and maintaining data integrity throughout instruction execution. Figure 2.7 shows the implemented memory stage components.

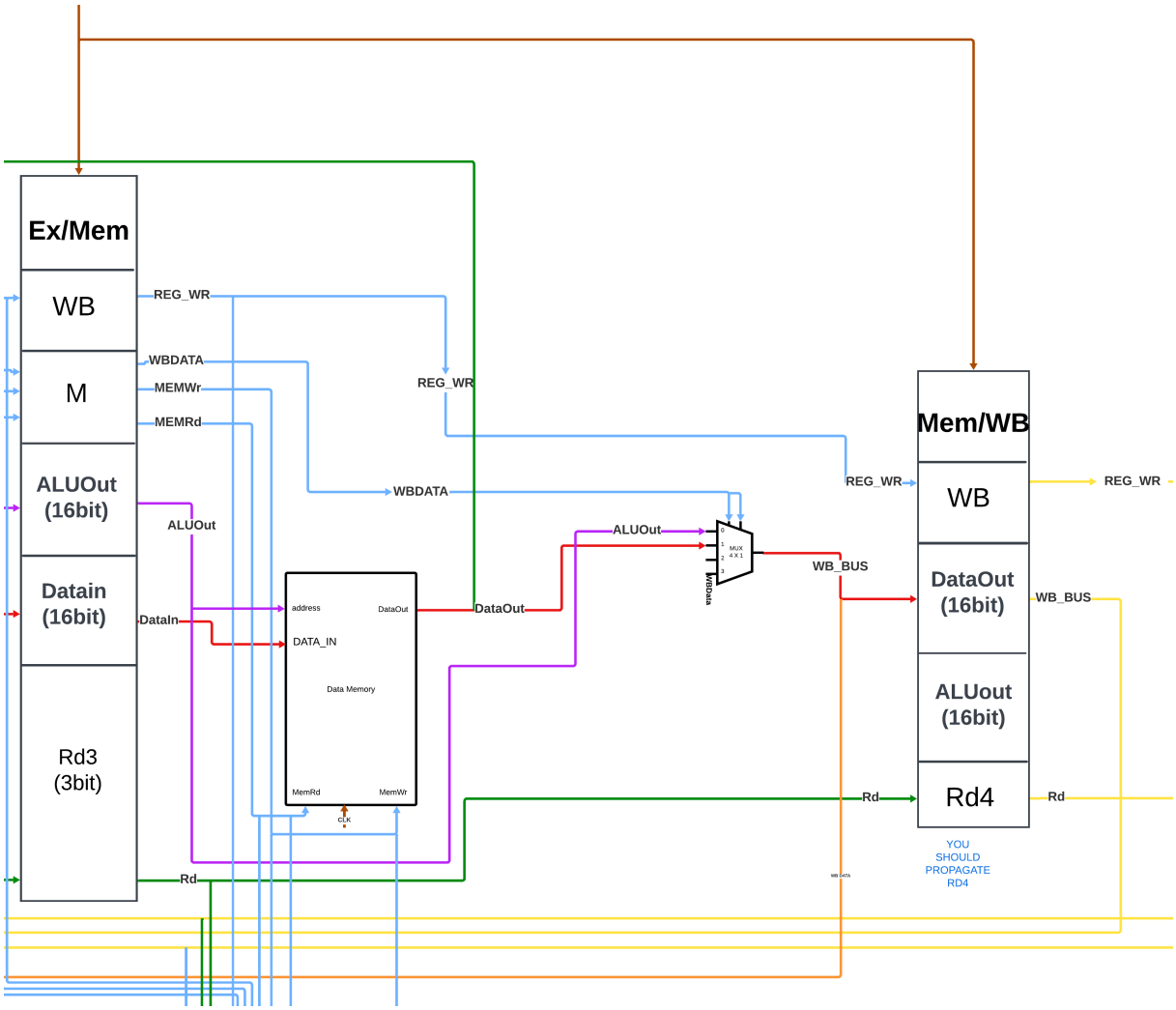

Figure 2.7: Pipeline Data Path Memory Stage

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.
  - **2/3** zero.

- **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.

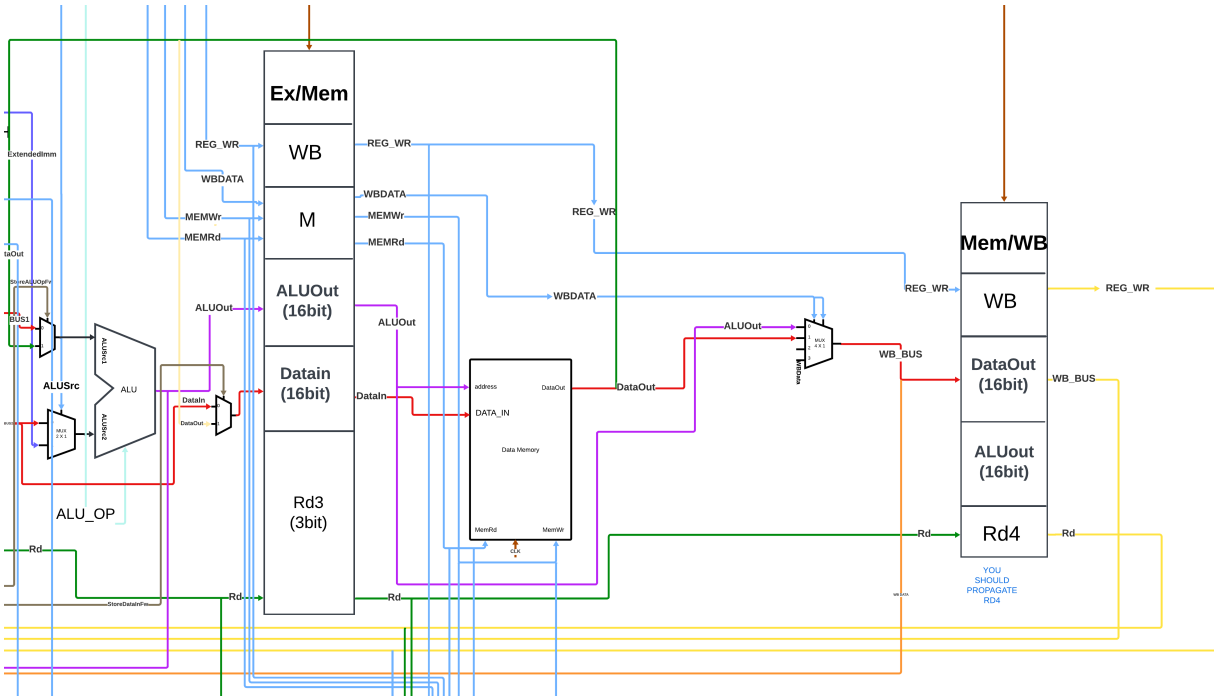

Figure 2.8: Pipeline Data Path Write Back Stage

2.5 Pipelining 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. Table 2.5 shows Boolean equations for the PC control signals.

Table 2.5: Signal Assignments and Boolean Expressions

Signal	**Boolean Expression**
branchTaken	`((IDOpcode == ‘BEQ) && (Z == 1)) || ((IDOpcode == ‘BNE) && (Z == 0))`
forTaken	`(IDOpcode == ‘FOR) && (Z == 0)`
jump	`(IDOpcode == ‘JTYPE) && ((IDFunction == ‘FunJMP) || (IDFunction == ‘FunCALL))`
RRSrc	`(IDOpcode == ‘JTYPE) && (IDFunction == ‘FunCALL)`
ret	`(IDOpcode == ‘JTYPE) && (IDFunction == ‘FunRET)`
PCsrcJType	`jump ? 1’b0 : 1’b1`
PCsrc	`branchTaken ? 2’b01 : ...`

2.5.2 Hazard Control Unit

The Hazard Control Unit effectively manages data hazards by generating forwarding and stall signals to ensure accurate instruction execution. Forwarding is employed from the memory stage to the execute stage to address load/store hazards efficiently. Table 2.6 presents the Boolean equations used for these control signals.

Table 2.6: Forward and Stall Signal Conditions

Forward/Stall Signal	**Condition**
ForwardBus1 = 0	ForwardBus1 != 1 && ForwardBus1 != 2
ForwardBus1 = 1	Rs1 == Rd2 && Exs.RegWr == 1
ForwardBus1 = 2	Rs1 == Rd3 && Mem.RegWr == 1
ForwardBus1 = 3	Rs1 == Rd4 && WB.RegWr == 1
ForwardBus2 = 0	ForwardBus2 != 1 && ForwardBus2 != 2
ForwardBus2 = 1	Rs2 == Rd2 && Exs.RegWr == 1
ForwardBus2 = 2	Rs2 == Rd3 && Mem.RegWr == 1
ForwardBus2 = 3	Rd2 == Rd4 && WB.RegWr == 1
StoreALLOp1	store\ALLOp1\RegNo == Rd3 && Exs.MemWr == 1 && Mem.MemRd == 1
StoreDetailnPw	store(MemIn)\RegNo == Rd3 && Exs.MemWr == 1 && Mem.MemRd == 1
Stall = 1	Ex.MemRd && (ForwardBus1 == 1 || ForwardBus2 == 1) && MemWr

Chapter 3

Simulation and Testing

Contents

3.1 Performance Counters

To evaluate the performance of the pipelined processor, several counters were implemented to monitor key metrics:

1. **Instruction Count:** Tracks the total number of instructions executed by the processor. This counter helps in measuring throughput and identifying any anomalies in instruction flow.
2. **ALU Operation Count:** Counts the number of arithmetic and logical operations performed by the ALU. It provides insights into the processor's utilization and workload distribution.
3. **Pipeline Stall Count:** Measures the number of stall cycles encountered due to hazards or dependencies. This counter is crucial for assessing pipeline efficiency and identifying optimization opportunities.
4. **Store Instruction Count:** Records the number of store operations executed. This counter aids in analyzing memory access patterns and evaluating the efficiency of memory-bound operations.

These counters are updated dynamically during execution and can be observed in the simulation results to validate pipeline behavior and identify potential bottlenecks. For example, the stall counter provides evidence of pipeline hazards, while the instruction count ensures all instructions are processed correctly.

3.2 Tested Assembly Code Sequence

Listing 3.1: Testcase

```
; Testcase

LW R2, R1, 1    ; Load word from address (R1 + 1) into R2
LW R3, R2, 0    ; Load word from address (R2 + 0) into R3
ADD R4, R2, R3  ; Add R2 and R3, store result in R4
SUB R5, R4, R3  ; Subtract R3 from R4, store result in R5
SW R4, R1, 12   ; Store word from R4 to address (R1 + 12)
SW R5, R0, 18   ; Store word from R5 to address (R0 + 18)
ADDI R1, R1, 1  ; Increment R1 by 1
JMP 0           ; Jump to address 0
ADDI R2, R2, 5  ; Increment R2 by 5 (killed)
```

3.3 Simulation Results

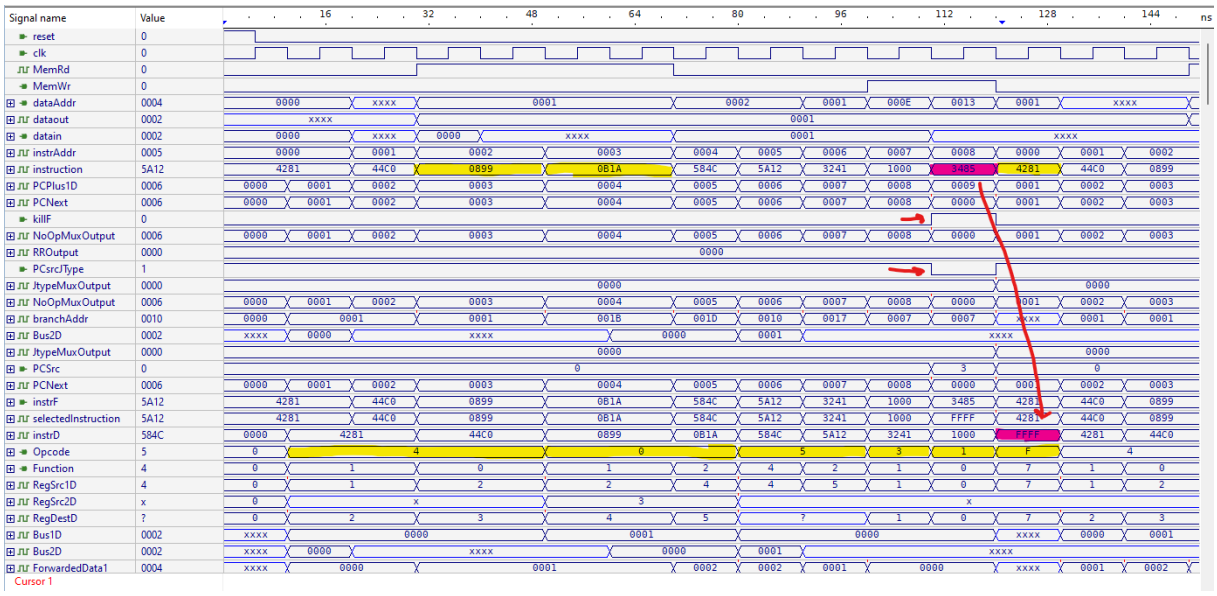

Figure 3.1: Testing datapath test case

Figure 3.1 illustrates the values of various wires in our design. Observing the instruction wire (Yellow), we notice that instructions '0899' and '0B1A' remain in the same position due to stall cycles caused by the load delay slot. Additionally, the jump instruction and the killed-fetched instruction (Pink) are highlighted, indicating the occurrence of a jump. This is further validated by the changes in the 'killF' signal and the 'PCSrcJType', confirming that the jump was successfully executed.

Moreover, the killed instruction is converted into a no-operation ('opcode 1111') and propagates through the pipeline, as indicated by the red arrow. Finally, in the opcode row at the bottom, we can observe that all instructions are correctly decoded and segmented throughout the pipeline.

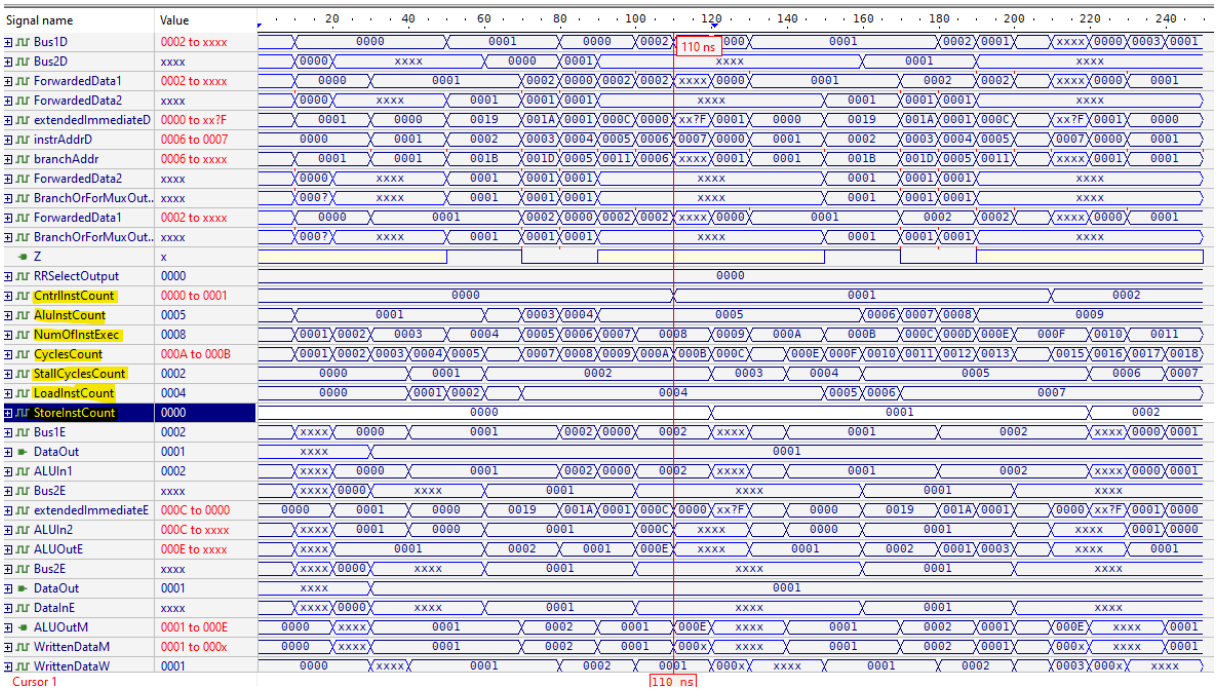

Figure 3.2: special regesters values

Examining Figure 3.2 and focusing on the second iteration (to the left of the red line, excluding the first iteration due to incorrect initial signal values), we observe the following:

- The `cntrlInstCount` is incremented when the jump instruction is encountered.
- The `AluInstCount` increases from 5 to 9, indicating that 4 ALU instructions were executed during the second iteration. This aligns with the test case, as the number of executed instructions is shown to be 8, matching the expected value.
- The total cycle count equals the number of executed instructions plus 2 stall cycles, as reflected in the stall cycles counter.
- The `store` counter experienced initialization issues during the first iteration, but in the second iteration, it correctly counts two store instructions, which matches the expected value from the test case.

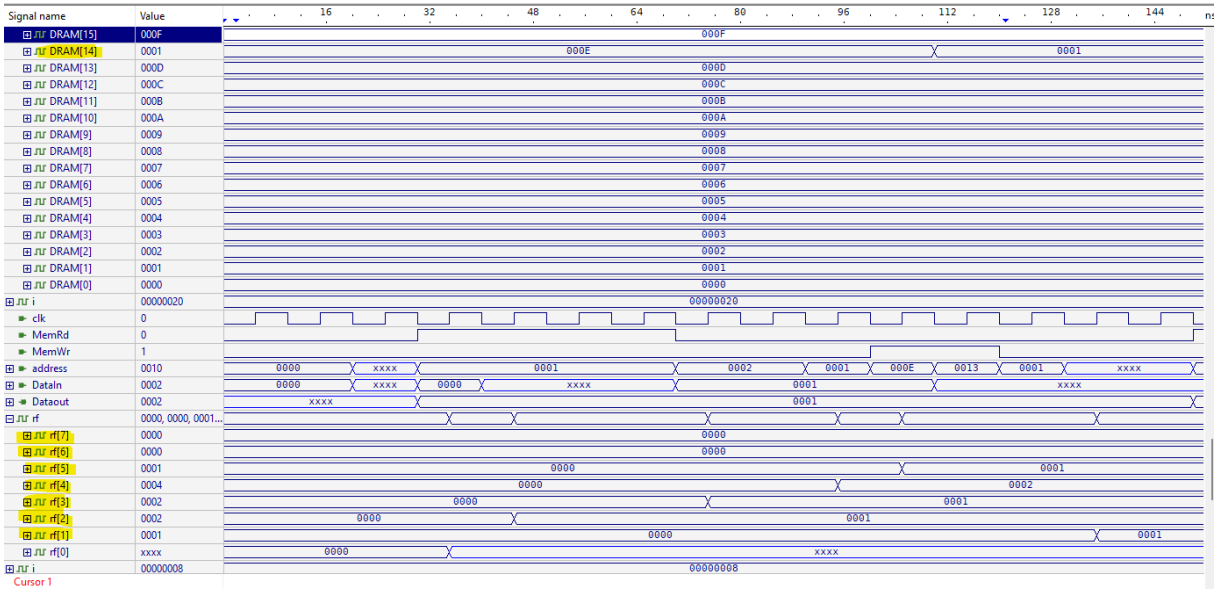

Figure 3.3: changes on regfile and memory

The DMEM was initialized such that each cell's value corresponds to its address (e.g., address 0 contains 0, address 1 contains 1, and so on). Figure 3.3 shows the chanes on register file and memory

- **R2** is loaded with the value 1 because it retrieves its value from the memory location $(R1 + 1)$, where $R1 = 0$.
- **R3** is also loaded with the value 1, as it retrieves its value from the memory location $(R2 + 0)$, where $R2 = 1$.
- **R4** is computed as $R2 + R3$, resulting in $1 + 1 = 2$.
- **R5** is calculated as $R4 - R3$, yielding $2 - 1 = 1$.
- Finally, the **SW** instruction stores the value of $R1$ (which is 1) at memory address $(R4 + 12)$, calculated as $2 + 12 = 14$.

Chapter 4

Conclusion

This project aimed to design and verify a simple pipelined RISC processor using Verilog, based on a 16-bit architecture with support for R-Type, I-Type, and J-Type instructions. The processor was designed with a five-stage pipeline (fetch, decode, execute, memory access, and write-back) and included key components such as a 16-bit program counter (PC), return register (RR), and eight general-purpose registers. Dedicated instruction and data memories were implemented to ensure efficient operation, along with performance monitoring registers to track execution metrics like the number of instructions executed, clock cycles, and pipeline stall cycles.

While the design and implementation phases were completed successfully, the testing phase encountered several challenges. Despite the development of a comprehensive test-bench and the simulation of various instruction sequences, some issues were identified during verification. These issues prevented the processor from fully meeting all functional requirements as outlined in the project specifications. Specifically, certain instructions and pipeline behaviors did not perform as expected, indicating areas for further refinement and debugging.

The project highlighted the complexities of processor design and the importance of thorough testing and validation. While the design methodology and modular implementation demonstrated promising results, the testing phase revealed gaps that need to be addressed in future iterations. Lessons learned from this experience will inform improvements in control signal generation, pipeline optimization, and verification strategies.

In conclusion, while the project achieved significant progress in designing a pipelined RISC processor, the testing phase underscored the need for further refinement to achieve full functional correctness. This experience provides valuable insights for future work in processor design and verification.

Bibliography