BIRZEIT UNIVERSITY

Faculty of Engineering & Technology
Electrical & Computer Engineering Department

ENCS4370

Computer Architecture

(First Semester – 2024/2025)

Report #1

Pipelined Processor

Prepared by:
Suhaib Sawalha 1220251
Abd Alhameed Maree 1220775
Mahmoud Zabade 1222936

Instructor: Dr. Ayman Hroub
Section: 1
Date: 13/1/2025

Abstract

The project presents the design and implementation of 16-bit instruction set architecture (ISA) using a 5-stage pipeline processor. The ISA includes 15 instructions divided into three types: R-type, I-type and J-type, each with its own specific format and control signals. The processor has 8 16-bit general purpose registers, 16-bit PC (Program Counter), 16-bit RR (Return Register), and 8 16-bit performance registers to track the instructions and clocks. The word size in the processor is 16-bit, the instruction memory and data memory are word-addressable. The ALU handles 5 different types of operations.

The processor is a RISC processor, as it supports fixed size and simple instructions that are capable to execute any program. Key instructions include logical operations (AND, SLL, SRL, FOR), arithmetic operations (ADD, SUB), memory access (LW, SW) and control flow (JUMP, CALL, RET, conditional branches, FOR).

The design procedure involved developing detailed data-path and control units, along with block diagrams and truth tables for the control signals. A five-stage pipeline was incorporated to improve performance, consisting of fetch, decode, ALU, memory access, and write-back stages. Verification included creating an extensive testbench and various code sequence to confirm the processor's accuracy and completeness.

Table of Contents

List of Figures

List of Tables

1 Theory

1.1 Set of operation

The processor supports set of operations to make sure the instruction set works and is flexible. These operations are divided into four types: arithmetic, logical, memory access, and control flow instructions. Each type has a specific function, allowing the processor to do many different tasks.

1.1.1 Arithmetic Operation

Arithmetic operation includes addition and subtraction, which are essential for numerical computations. These operations are performed using 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.
- ADDI (Add Immediate): Computes the sum of a register and an immediate value.
- FOR (For loop): decrements the counter of the loop by one.

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.
- SLL (Logical shift left): Shift the bits of a register to left by a specific number of positions, filling the vacated bit position with zeros.
- SRL (Logical shift right): Shift the bits a register to the right by a specified number of positions, filling, filling the vacated bit position with zeros.

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.

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 (For loop): Branch to the first instruction in the loop scope if the loop counter is not 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 three distinct instruction types: R-type, I-type and J-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 which equals to zero in case of R type instructions, destination register (Rd), and two source registers (Rs and Rt). And a function field to determines the specific operation of the instruction

Opcode (15-12)	Rd (11-9)	Rs (8-6)	Rt (5-3)	Function (2-0)

Figure 1-1: R-type instruction format

1.2.2 I-type Instructions

I-type instructions include four-bit Opcode, immediate values for arithmetic and logical operations, as well as for load, branch operations and for loop. In logical instructions, the immediate is zero extended while it is sign-extended for all other instruction. In the case of BEQ and BNE the branch target is calculated as: Branch target = Current PC + sign extended immediate. The format includes an opcode, destination register (Rt), source register (Rs)

Opcode (15-12)	Rs (11-9)	Rt (8-6)	Signed Imm (5-0)

Figure 1-2: I-type instruction format

1.2.3 J-type Instructions

J-type instructions handle jumps, calls and returns. The format includes a four-bit opcode (the opcode is 1 for all J-Type instruction), three-bit Function. This field determines the specific operation of the instruction and a 9-bit offset to calculate the target address for jumps, returns and calls. The jump target address is calculated by concatenating the PC and the nine-bit offset

Opcode (15-12)	9-bit offset (11-3)	Function (2-0)

Figure 1-3: J-type instruction format

1.3 Processor Architecture

The processor has a 16-bit word and instruction size, with eight 16-bit general-purpose registers. It includes a 16-bit Program Counter (PC) and a 16-bit Return Register (RR) for function call management. The architecture features separate memories for instructions and data. It supports three instruction types: R-type (Register), I-type (Immediate), and J-type (Jump). The processor uses word-addressable memory, allowing efficient memory access.

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

Table 2-1: RTL Design

Instruction Type	Stage	RTL
R-type	IF (Instruction Fetch)	inst = instMem [PC] PC = PC + 1
	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 + 1
	ID (Instruction Decode)	Source = Reg [inst [11:9]] Immediate = zero_extend (inst [5:0]) (for ANDI) Immediate = sign_extend (inst [5:0]) (for ADDI) DestinationAddress = inst [8:6] Opcode = inst [15:12]
	EX (Execute)	AluResult = Source + Immediate (for ADDI) AluResult = Source & Immediate (for ANDI)
	MEM (Memory Access)	Not required
	WB (Write Back)	RegFile [DestinationAddress] = AluResult
I-type (Load/Store)	IF (Instruction Fetch)	inst = instMem [PC] PC = PC + 1
	ID (Instruction Decode)	source = Reg [inst [11:9]] Immediate = inst [5:0] DestinationAddress = inst [8:6] Opcode = inst [15:12]
	EX (Execute)	AluResult = source + Immediate
	MEM (Memory Access)	LW: MemData = DMem [AluResult] SW: DMem [AluResult] = RegFile [DestinationAddress]
	WB (Write Back)	LW: RegFile [DestinationAddress] = MemData
I-type (For)	IF (Instruction Fetch)	inst = instMem [PC] PC1 = PC + 1
	D (Instruction Decode)	RepeatedInstructionAddress = Reg [inst [11:9]] LoopCounter = Reg [inst [8:6]] Opcode = inst [15:12]

		If (Opcode == FOR && LoopCounter! = 0) Then PC = RepeatedInstructionAddress else PC = PC1
	EX (Execute)	AluResult = LoopCounter - 1
	MEM (Memory Access)	Not required
	WB (Write Back)	LoopCounter = AluResult
I-type (Branch)	IF (Instruction Fetch)	inst = instMem [PC] PC1 = PC + 1
	ID (Instruction Decode)	PcRelativeOffset = sign_ext (inst [5:0]) AluOperand1 = Reg [inst [11:9]] AluOperand2=Reg [int [8-6]] Opcode = inst [15:12] If (Zero) Then PC = PC + PcRelativeOffset Else PC1
	EX (Execute)	Not required
	MEM (Memory Access)	Not required
	WB (Write Back)	Not required
J-type	IF (Instruction Fetch)	inst = instMem [PC] PC1 = PC + 1

	ID (Instruction Decode)	Opcode = inst [15:12]
		Offset = inst [11-3]
		Function = inst [2-0]
		If (Opcode == 1 && function =1)
		Then RR = PC1
		If (Opcode == 1 && function =0) PC = {PC [15:9], offset}
		If (Opcode == 1 && function =1) PC = PC + 1
		If (Opcode == 1 && function =2) PC = RR

	EX (Execute)	Not required
	MEM (Memory Access)	Not required
	WB (Write Back)	Not required

2.2 Single Cycle Processor

The single-cycle processor was designed first and then turned into a pipelined-processor; this step was done to ensure that every essential concept in the data path would work as expected.

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:

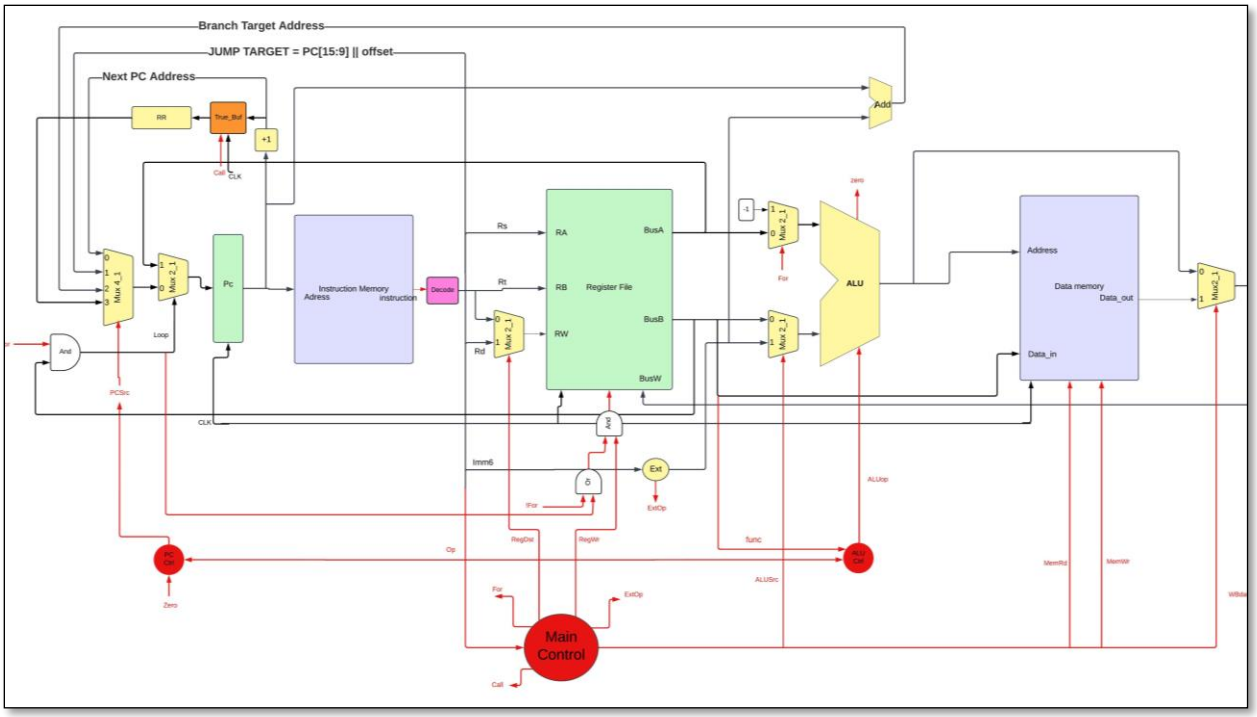

Figure 2-1: Single Cycle Processor Datapath

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 1, pointing to the next instruction, since each instruction is 16 bits (1 word).
- **Return Register (RR):** The RR register is used to store the value of PC+1 in case of "CALL" instruction to return it when the instruction "RET".
- **Buffer:** This buffer is used to control the assigning of the RR register to write it only when "CALL" register is called.
- **First Mux:** This multiplexer is used to choose the value of the next PC depending on the PC control unit.
- **Second Mux:** This multiplexer is used to choose the final value of the next PC if there is a "FOR" instruction.

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 (Rs and Rt) 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.

- **Extender:** For I-type instructions, the extender sign-extends or zero-extends the immediate value to match the operand size required by the ALU.
- **First Mux:** This Mux is used to choose between Rs from the register file and -1 in case of "FOR" instruction, and this would be the first input of the ALU.
- **Second Mux:** This Mux is used to choose between Rt and Immediate in case of R-type and I-type instructions, and this would be the second input of the ALU.

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).
- **ALU Control:** This is used to determine the ALU operation that would be done to the inputs.

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 Main 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, For 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:

Table 2-2: Main Control Signals Truth Table

Inst/signal	Call	For	RegDst	RegWr	ExtOp	ALUSrc	MemRd	MemWr	WBdata
AND	0	0	1	1	X	0	0	0	0
ADD	0	0	1	1	X	0	0	0	0
SUB	0	0	1	1	X	0	0	0	0
SLL	0	0	1	1	X	0	0	0	0
SRL	0	0	1	1	X	0	0	0	0
ANDI	0	0	0	1	0	1	0	0	0
ADDI	0	0	0	1	1	1	0	0	0
LW	0	0	0	1	1	1	1	0	1
SW	0	0	X	0	1	1	0	1	X
BEQ	0	0	X	0	1	0	0	0	X
BNE	0	0	X	0	1	0	0	0	X
FOR	0	1	0	1	X	0	0	0	0
JMP	0	0	X	0	X	X	0	0	X
CALL	1	0	X	0	X	X	0	0	X
RET	0	0	X	0	X	X	0	0	X

- **Call:** Indicates that the instruction is “CALL”.
- **For:** Indicates that the instruction is “FOR”.
- **RegDst**: Indicates the destination register that would be written on:
  - **0:** the destination register is Rt.
  - **1:** the destination register is Rd.
- **RegWr:** Indicates whether the instruction would write on the register file or not.
- **ExtOp:** Indicates whether the extension would be 0-extention or 1-extension.
- **ALUSrc:** Indicates the second ALU input:
  - **0:** the input is BusB from the register file.
  - **1:** the input is the extended immediate.
- **MemRd:** Indicates whether the instruction would read from the memory or not.

- **MemWr:** Indicates whether the instruction would write on the memory or not.
- **WBdata:** Indicates the source of the data that would written on the register file:
  - **0:** the alu result would be written.
  - **1:** the output of the memory would be written.

The "FOR" instruction is a bit tricky, it would only write in the register file only when Rt does not equal 0, so there is a control signal called **RegWrF**, the signal is the true signal that would be entered to the register file, to find its expression there is another control signal defined named **Loop:**

- **Loop:** This control signal indicates that the for loop is going to an iteration, this done by taking the **and** result between the **FOR** and **Rt**, this indicates that the instruction is "FOR" and **Rt** does not equal zero, so an iteration would happen and the **PC** would equal to **Rs**.
- **RegWrF:** This control signal indicates that the register file would be written into, this is done by taking the Loop signal (which indicates that Rt isn't 0 so it would be written as Rt – 1) or if the instruction would write on the register file and it is not the "FOR" instruction.

Table 2-3: Main Control Signals Boolean Expressions

Signal	Boolean Expression
Call	(opcode = 0001) ^ (func = 001)
For	(Opcode = 1000)
RegDst	(Opcode = 0000)
RegWR	(Opcode = 0000) v (opcode = 0010) v (opcode = 0011) v (opcode = 0100) v (opcode = 1000)
ExtOp	(opcode = 0011) v (opcode = 0100) v (opcode = 0101) v (opcode = 0110) v (opcode = 0111)
ALUsrc	(opcode = 0010) v (opcode = 0011) v (opcode = 0100) v (opcode = 0101)
MemRd	(opcode = 0100)
MemWr	(opcode = 0101)
WBdata	(opcode = 0100)
Loop	For ^ Rt
RegWrF	RegWrF ^ (~For v Loop)

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:

Table 2-4: ALU Control Signals Truth Table

Instruction	ALUop
AND	AND (000)
ADD	ADD (001)
SUB	SUB (010)
SLL	SLL (011)
SRL	SRL (100)
ANDI	AND (000)
ADDI	ADD (001)
LW	X
SW	X
BEQ	X
BNE	X
FOR	ADD (001)
JMP	X
CALL	X
RET	X

- **ALUOp:** Determines the specific operation the ALU will perform (AND, ADD, SUB, SLL, SRL). 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:

Table 2-5: PC Control Signals Truth Table

instruction	**zero**	**PCSrc**
AND	X	0
ADD	X	0
SUB	X	0
SLL	X	0
SRL	X	0
ANDI	X	0
ADDI	X	0
LW	X	0
SW	X	0
BEQ	0 1	0 2
BNE	0 1	2 0
FOR	X	0
JMP	X	1
CALL	X	1
RET	X	3

Table 2-6: PC Control Signals Boolean Expression

Signal	**Boolean Expression**
PCSrc[0]	(opcode = 0001)
PCSrc[1]	(opcode = 0110 ^ Zero) v (opcode = 0111 ^ ~zero) v (opcode = 0001 ^ func = 010)

The following represents the meaning of the value of PCSrc:

- **00:** The next instruction is at PC + 1.
- **01:** The next instruction is at the jump target = PC[15:9] || offset.
- **10:** The next instruction is at the branch target = PC + sign extended immediate.
- **11:** The next instruction is at the Return Register (RR).

This is not the final PC value; it depends on the control signal Loop:

- **Loop = 0**, the value of the PC would stay the same.
- **Loop = 1**, the value of the PC is **Rs**.

2.3.4 Hazard Controls

There are special control signals that are added to the pipeline process to resolve the hazards that can appear: structural hazards, data hazards and control hazards. These controls are: **ForwardA**, **ForwardB** to control the data forwarding, **stall** to make stall cycles when there is a dependency that is not solved by the forwarding, **kill** to kill the instruction if it is fetched and was not supposed to execute in branching and jumping, **zero** This control signal is to used to compare two registers values in order to branch or not.

```
if (Rs == Rd2 && Ex_RegWr)
    ForwardA = 2'b01;
else if (Rs == Rd3 && Mem_RegWr)
    ForwardA = 2'b10;
else if (Rs == Rd4 && WB_RegWr)
    ForwardA = 2'b11;
else
    ForwardA = 2'b00;

if (Rt == Rd2 && Ex_RegWr)
    ForwardB = 2'b01;
else if (Rt == Rd3 && Mem_RegWr)
    ForwardB = 2'b10;
else if (Rt == Rd4 && WB_RegWr)
    ForwardB = 2'b11;
else
    ForwardB = 2'b00;

stall = (MemRd_Ex === 1'b1) && ((ForwardA === 2'b01) || (ForwardB === 2'b01));
```

Figure 2-2: Forwarding and Stall Control Signals

```
kill = ((Op == `BEQ) && (BusA_forward == BusB_forward)) || ((Op == `BNE) && (BusA_forward != BusB_forward))
               || (Op == `JMP) || (Op == `CALL) || (Op == `RET) || (Op == `FOR && Loop == 1'b1);
zero = (Op == `BEQ || Op == `BNE) && (BusA_forward == BusB_forward);
```

Figure 2-3: Kill and Zero 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, a detailed explanations of each stage and the corresponding stage buffers is provided, followed by a discussion on how hazards are handled in the pipeline.

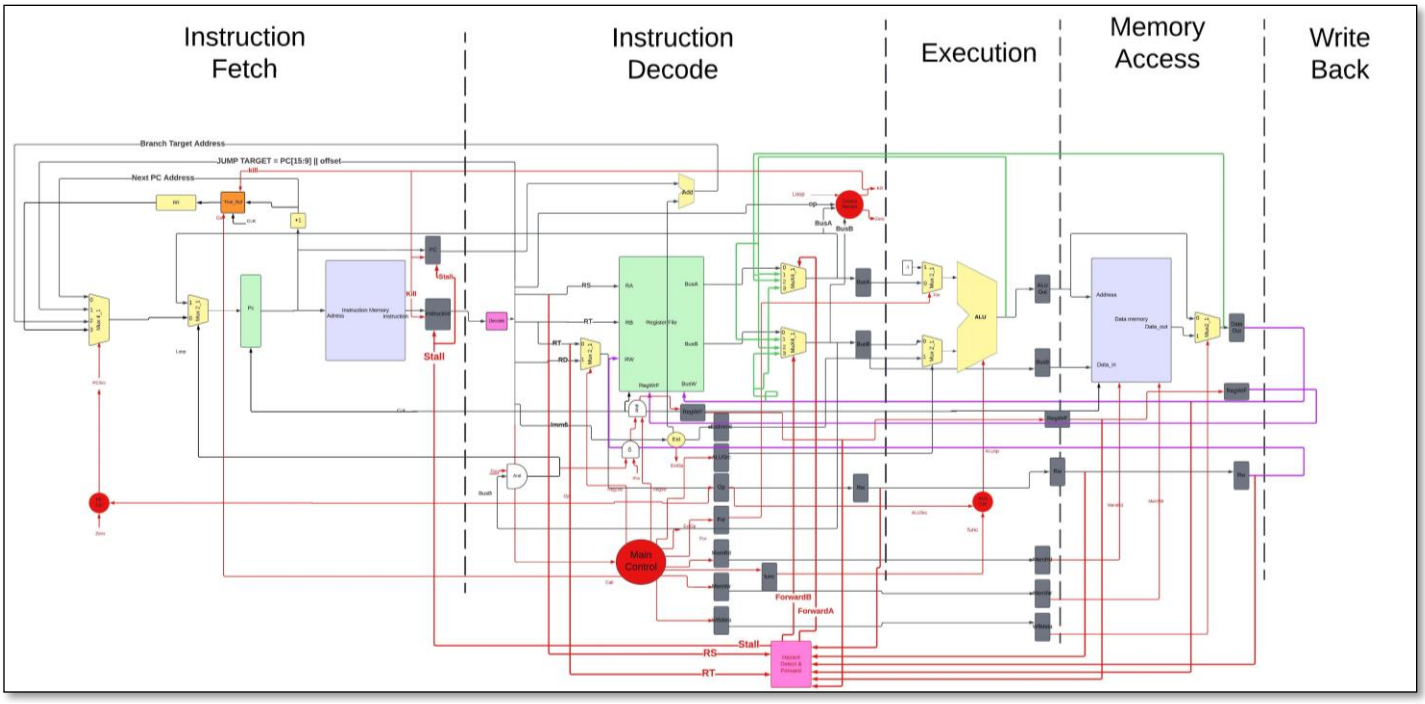

Figure 2-4: Pipeline Processor Datapath

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.

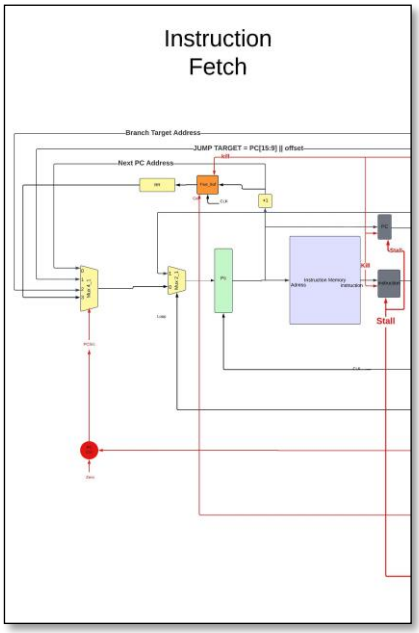

Figure 2-5: Instruction Fetch Stage

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 instruction is at PC + 1.
    - **Input 1:** The next instruction is at the jump target = PC[15:9] || offset.
    - **Input 2:** The next instruction is at the branch target = PC + sign extended immediate.
    - **Input 3:** The next instruction is at the Return Register (RR).

This is not the final PC value; it depends on the control signal Loop:

- **Loop = 0**, the value of the PC would stay the same.
- **Loop = 1**, the value of the PC is the value of **Rs** that is taken from the forwarding.

- **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 1 to point to the next instruction.
- **Return Register (RR):** The RR register is used to store the value of PC+1 in case of "CALL" instruction to return it when the instruction "RET".
- **PC Control Unit:** The PC control unit generates the PCSrc from the Opcode that is generated in the Instruction Decode stage.
- **Buffers:** There are two buffers that go from Instruction Fetch stage to Instruction Decode stage, the PC and the instruction, these buffers take the values, stall signal and kill signal. The buffers will pass a bubble if the instruction to be killed, and the value itself otherwise.

Operations:

- The PC provides the address to the Instruction Memory.
- The instruction is fetched from the Instruction Memory.
- The PC is incremented by 1 to point to the next instruction.
- The PC is stored in the RR register if the instruction is "CALL".

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.

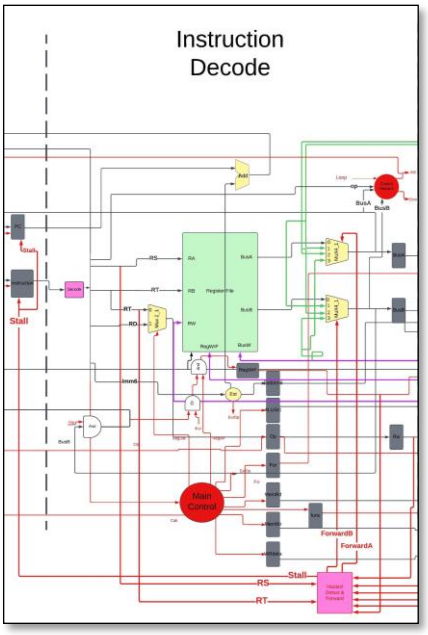

Figure 2-6: Instruction Decode Stage

Components:

- **Decode Unit:** Thie unit is responsible of decoding the instruction that is provided by the Instruction Fetch stage and split it into components depending on the type of it (I-type, R-type or J-type) as described in the Theoretical part.
- **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.
- **Mux 2x1:** Choose between **Rt** and **Rd** as the register to write on depending on the **RegDst** control signal.
- **Hazard Unit:** The hazard unit is used to determine the stall cycles and forwarding, these signals are determined by the opcode and the values of the registers in the next stages.
- **Control Hazard Unit:** The control hazard unit is used to determine if the next instruction would be killed and if the branch would be taken.
- **Forward Mux:** Two Mux 4x1 (ForwardA and ForwardB) these muxes are used to get the final values that would be sent to the ALU as inputs after the forwarding is done.
- **Sign Extender:** Extends a 6-bit immediate value to a signed/unsigned 16-bit value, allowing for proper handling of signed arithmetic operations in the processor.
- **Adder Component:** Calculates the target address for branch instructions, determining the next instruction address when a branch is taken.

- **Buffers:** The control signals are passed to the next stage (Execute Stage) and the final values of the registers after forwarding.

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 inputs are generated and given to the Execute Stage.
- **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.

2.4.4 Execution

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.

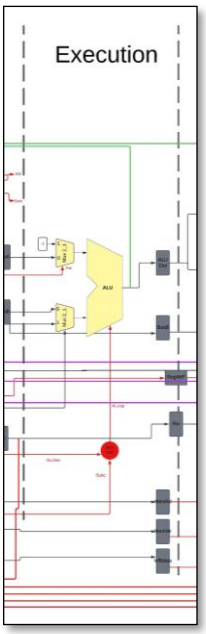

Figure 2-7: Execution Stage

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.
- **First Mux 2x1:** This mux is used to determine the output between the value of the register passed by the buffer and -1, which would be chosen if the instruction is "FOR" instruction. This is done by **For** control signal which is passed from the previous stage using buffer.
- **Second Mux 2x1:** This mux is used to determine the output between the value of the register passed by the buffer and the Immediate value passed by the buffer too. This is done by **ALUSrc** control signal which is passed from the previous stage using buffer.
- **ALU Control Unit:** The control unit uses **Opcode** and **func** control signals passed from the previous stage using buffers to determine the signal **ALUop**, which would be passed to the ALU to determine the operation that the ALU would apply to its inputs.
- **Buffers:** The ALU passes its output to a buffer to pass it to the next stage, it also passes other control signals to the next stage to be used.

Operations:

- The ALU performs the operation specified by the ALUop signal using the operands selected by the two 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.

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.

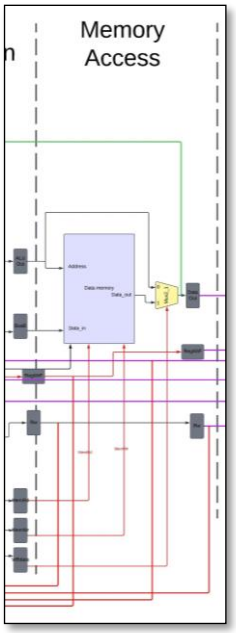

Figure 2-8: Memory Stage

Components:

- **Data Memory:** Stores values in address and allow to read and write from it (RAM).
- **Mux 2x1:** To choose from the ALU output and Memory output to write on the register depending on the value of **WBdata** signal passed from the previous stage using buffer.
- **Buffers:** The final result to write on a register and other control units relevant is passed to the next stage using buffers.

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.

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.

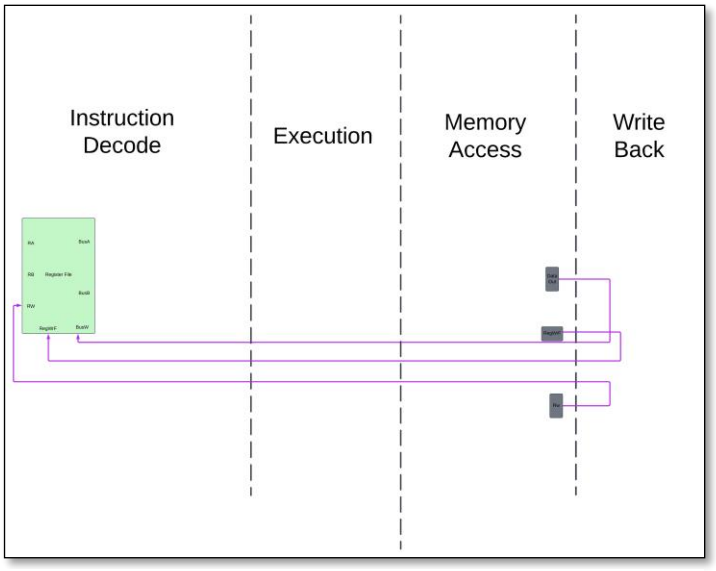

Figure 2-9: Write Back Stage

Write Back stage takes the values of: **RegWrF** signal to determine if register file would be written on it, **RW** which is the register that would be written on and **DataOut** which is the value that would written on the register. All of them are passed directly to the register file.

3 Verification

To verify the pipelined processor design and its Verilog code, 2 test cases that uses every instruction in the Instruction Set Architecture (ISA) are used. A C++ code was written to transform assembly code to machine code (0's and 1's).

3.1 Tests

3.1.1 Test 1

The following code stores five values (6, 14, 24, 2, 5) in the memory and read them after that two times: once with "FOR" instruction and the other using "BNE". Inside the code itself, the control instructions are tested too by the register 6, the expected behavior is to get the first value 1 then 7 the 15.

```
ADDI R1 R0 6
SW R1 R0 0
ADDI R1 R0 14
SW R1 R0 1
ADDI R1 R0 23
SW R1 R0 2
ADDI R1 R0 2
SW R1 R0 3
```

```
ADDI R1 R0 5
SW R1 R0 4
ADDI R3 R0 4
ADDI R5 R0 12
LW R4 R2 0
ADDI R2 R2 1
FOR R5 R3
ADDI R7 R0 7
BEQ R6 R7 2
ADDI R6 R0 7
ADDI R3 R0 4
ADDI R2 R0 0
LW R4 R2 0
ADDI R2 R2 1
BNE R2 R3 -2
CALL 26
ADDI R6 R0 7
JMP 28
ADDI R6 R0 1
RET
ADDI R6 R0 15
```

The following can be noticed from this test case:

- It tests all structural hazards by default.
- It tests “Read after Write” hazard that is solved by the forwarding.
- It tests all types of control hazards.

The code transforms this to the following machine code:

```
0011000001000110
0101000001000000
0011000001001110
0101000001000001
0011000001010111
0101000001000010
0011000001000010
0101000001000011
0011000001000101
0101000001000100
0011000011000100
0011000101001100
0100010100000000
0011010010000001
1000101011000000
```

0011000111000111
0110111110000010
0011000110000111
0011000011000100
0011000010000000
0100010100000000
0011010010000001
0111011010111110
0001000011010001
0011000110000111
0001000011100000
0011000110000001
0001000000000010
0011000110001111

The following are the simulation results after running the program:

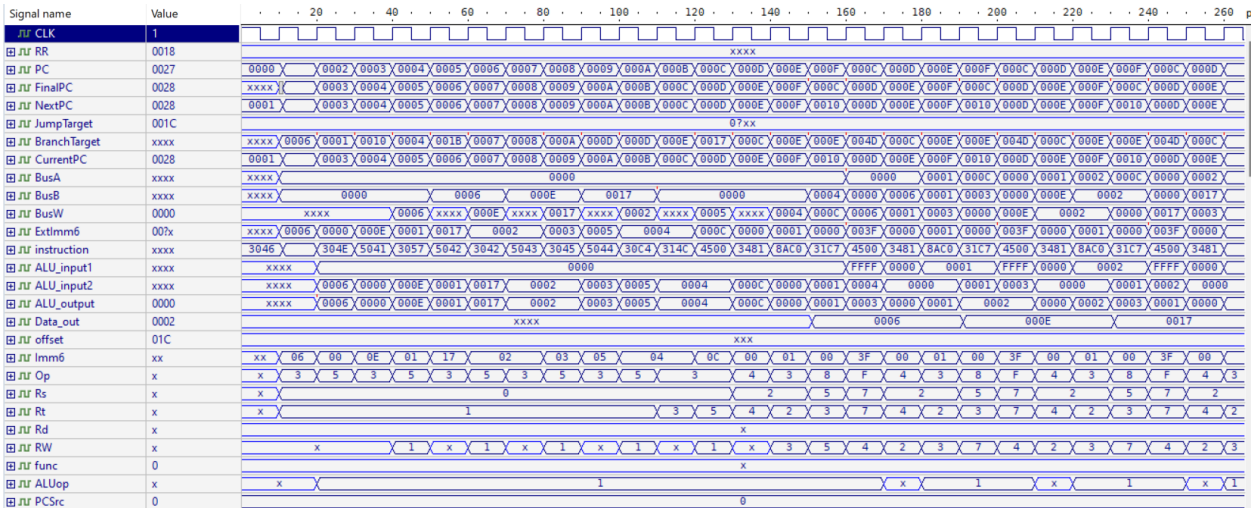

Figure 3-1: Test 1 waveform 1

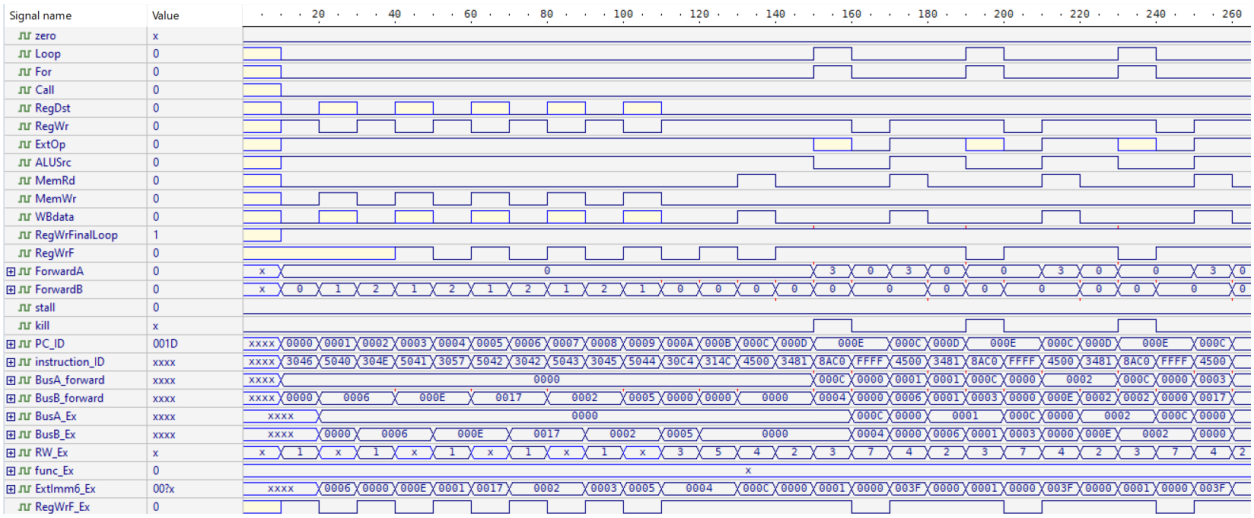

Figure 3-2: Test 1 waveform 2

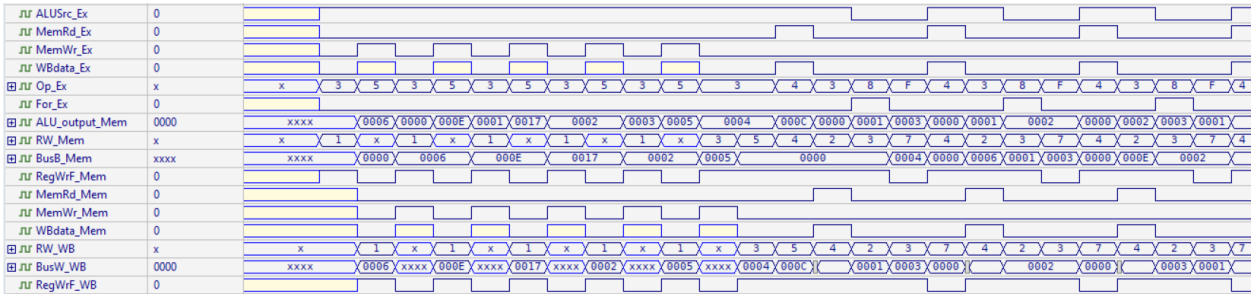

Figure 3-3: Test 1 waveform 3

The waveforms mainly used for debugging when the output is not as expected.

To ensure that the program is acting as expected, the register values are printed whenever one of the changes, the following are the output registers:

```
# KERNEL: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000110 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000001110 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000010111 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000010 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

Figure 3-4: Test 1 output 1

Figure 3-4 shows the values stored in the memory, as they are first stored in R1 then to memory.

```
# KERNEL: 0000000000000000 0000000000000101 0000000000000000 0000000000000100 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000000 0000000000000100 0000000000000000 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000000 0000000000000100 0000000000000110 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000001 0000000000000100 0000000000000110 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000001 0000000000000011 0000000000000110 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000001 0000000000000011 0000000000001110 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000010 0000000000000011 0000000000001110 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000010 0000000000000010 0000000000001110 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000010 0000000000000010 0000000000010111 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000011 0000000000000010 0000000000010111 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000011 0000000000000001 0000000000010111 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000011 0000000000000001 0000000000000010 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000100 0000000000000001 0000000000000010 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000100 0000000000000000 0000000000000010 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000100 0000000000000000 0000000000000101 0000000000001100 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000101 0000000000000101 0000000000000000 0000000000000101 0000000000001100 0000000000000000 0000000000000000
```

Figure 3-5: Test 1 output 2

Figure 3-5 shows the following when execute the “FOR” instruction:

- R2 is acting as the index of the memory, it can be seen that its values are (0, 1, 2, 3, 4).
- R3 is storing the number of iterations of the for loop, it can be seen that its value is 4, as the first iteration is always executed.
- R4 is storing the values of the memory values (Mem[R2]) which are (6, 14, 24, 2, 5).
- R5 is storing the position of beginning of each for loop iteration.

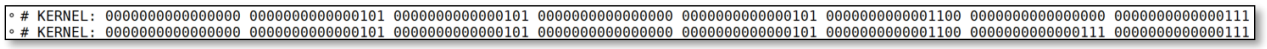

Figure 3-6: Test 1 output 3

Figure 3-6 shows the value of R7 changes, then a “BEQ” instruction is applied between R6 and R7, but as they are not equal the branch is not taken and the following instruction would be applied to assign R7 to R6.

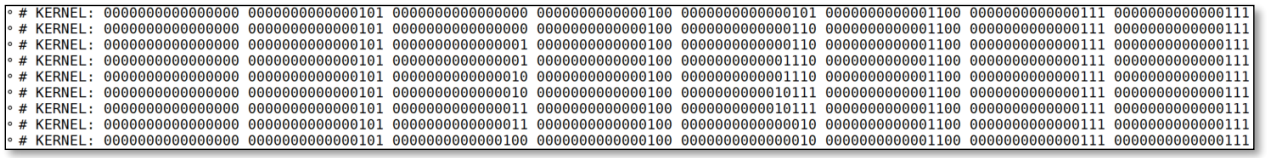

Figure 3-7: Test 1 output 4

Figure 3-7 shows the following when execute the “BNE” instruction:

- R2 is acting as the index of the memory, it can be seen that its values are (0, 1, 2, 3).
- R3 is storing the number 4 to take only the first 4 elements stored in the array to compare it with R2.
- R4 is storing the values of the memory values (Mem[R2]) which are (6, 14, 24, 2).

Figure 3-8: Test 1 output 5

Figure … shows how R6 is changing according to the order of jump instructions:

- First going to CALL instruction that has ADDI to give it the value “1”, then RET to return to the next instruction after the CALL.
- The instruction after the CALL is ADDI to give it the value “7”, then JMP.
- The JMP is telling the PC to jump to another ADDI that gives the value “15”.

```
# KERNEL: Executed Instructions:    50
# KERNEL: Load Instructions:        9
# KERNEL: Store Instructions:       5
# KERNEL: ALU Instructions:         42
# KERNEL: Control Instructions:     13
# KERNEL: Number of Clocks:         101
# KERNEL: Stall Cycles:             10
# KERNEL: Killed Instructions:      10
```

Figure 3-9: Test 1 Performance Registers

Figure 3-9 shows the results of the performance registers that gives counts of the instructions.

3.1.2 Test 2

This test is mainly used to test the Load Delay hazard and solve it using stall cycles. The test is just about storing some values in the memory and load value from the memory and some small calculations.

```
ADDI R1 R0 3
SW R1 R0 0
LW R2 R0 0
ADD R3 R1 R2
ADD R4 R1 R3
LW R5 R4 -9
ADD R6 R4 R5
ADD R7 R6 R6
SRL R7 R7 R1
SLL R6 R6 R1
```

The code shows the load delay hazard that is solved by giving a stall cycle.

The machine code for the above code is:

```
0011000001000011
0101000001000000
0100000010000000
0000011001010001
0000100001011001
0100100101110111
0000110100101001
0000111110110001
0000111111001100
0000110110001011
```

The following are the simulation results after running the program:

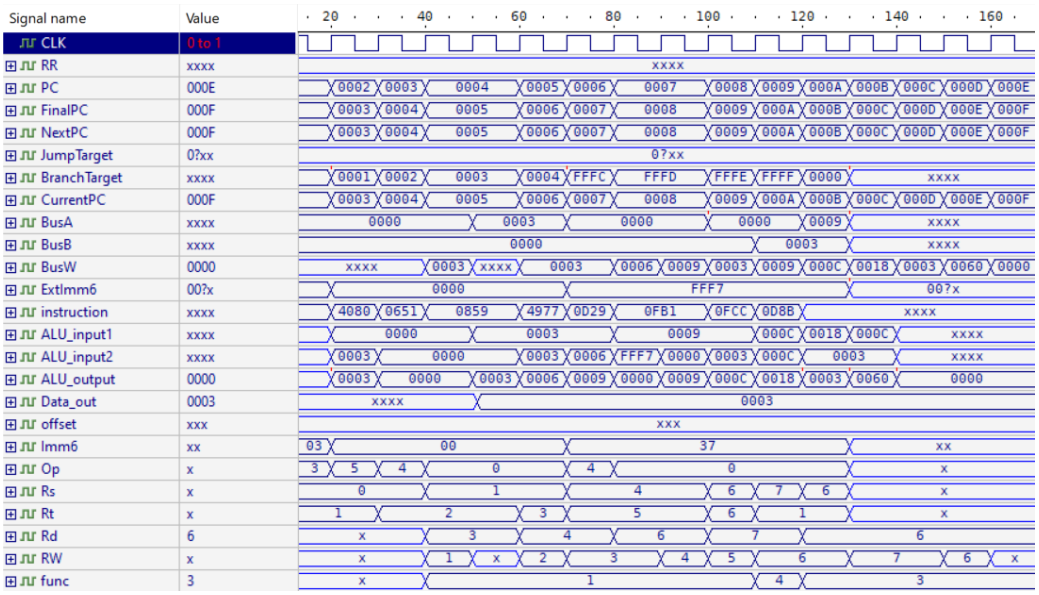

Figure 3-10: Test 2 waveform 1

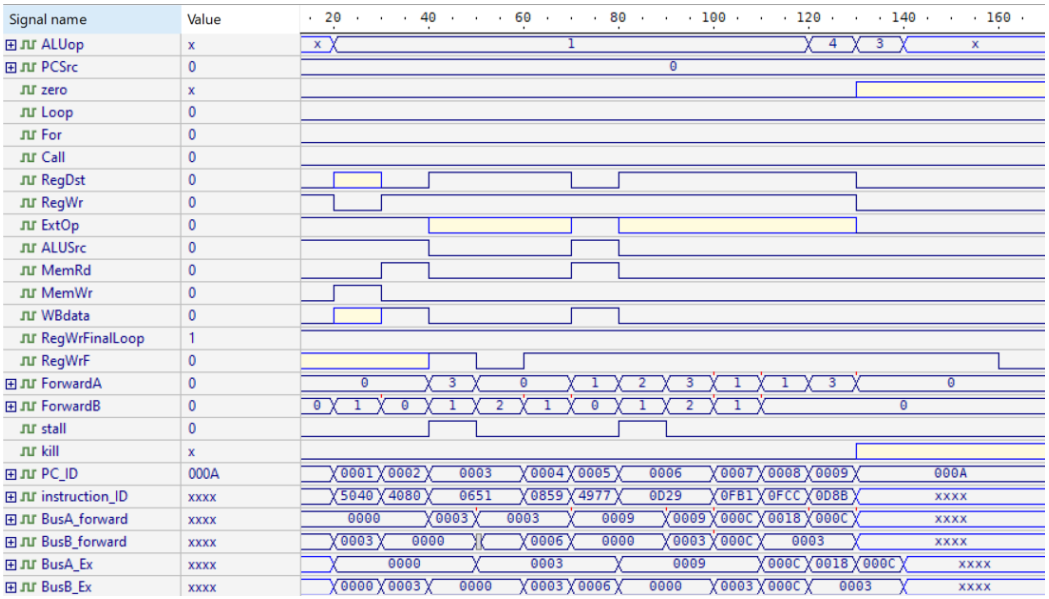

Figure 3-11: Test 2 waveform 2

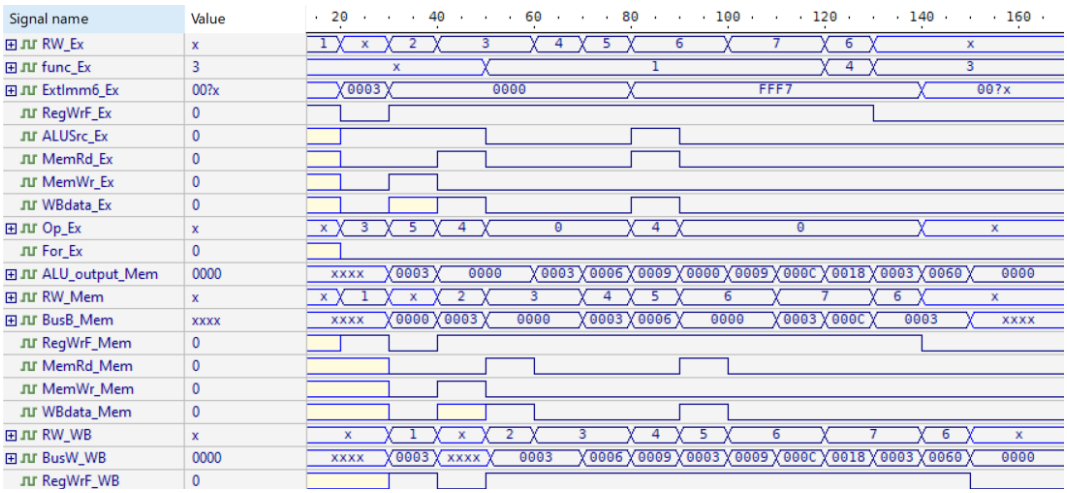

Figure 3-12: Test 2 waveform 3

The waveforms mainly used for debugging when the output is not as expected.

To ensure that the program is acting as expected, the register values are printed whenever one of the changes, the following are the output registers:

```
# KERNEL: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000011 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000110 0000000000000000 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000110 0000000000001001 0000000000000000 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000110 0000000000001001 0000000000000011 0000000000000000 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000110 0000000000001001 0000000000000011 0000000000001001 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000110 0000000000001001 0000000000000011 0000000000001100 0000000000000000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000110 0000000000001001 0000000000000011 0000000000001100 0000000000011000
# KERNEL: 0000000000000000 0000000000000011 0000000000000011 0000000000000110 0000000000001001 0000000000000011 0000000001100000 0000000000000011
```

Figure 3-13: Test 2 output

Figure … shows the results of running the program:

- R1 = R0 + 3 = 3
- Mem[R0] = Mem[0] = R1 = 3
- R2 = Mem[R0] = Mem[0] = 3
- R3 = R1 + R2 = 3 + 3 = 6 load delay
- R4 = R1 + R3 = 3 + 6 = 9
- R5 = Mem[R4 – 9] = Mem[0] = 3
- R6 = R4 + R5 = 9 + 3 = 12 load delay
- R7 = R6 + R6 = 12 + 12 = 24
- R7 = R7 >> R1 = 24 >> 3 = 3
- R6 = R6 << R1 = 12 << 3 = 96

The output of the code gives the exact expected output.

For the performance registers:

```
# KERNEL: Executed Instructions:    10
# KERNEL: Load Instructions:        2
# KERNEL: Store Instructions:       1
# KERNEL: ALU Instructions:         10
# KERNEL: Control Instructions:     0
# KERNEL: Number of Clocks:         18
# KERNEL: Stall Cycles:             2
# KERNEL: Killed Instructions:      0
```

Figure 3-14: Test 2 Performance Registers

The performance registers values represent the following:

- There are 10 executed registers.
- There are 2 load instructions.
- There are 1 store instruction.
- All instructions use the ALU.
- The are no control instructions.
- The clocks given are 18.
- There are 2 stall cycles (as there is 2 load delay hazards).
- There are no killed instructions.

4 Teamwork

Design Process and Diagram

- The design process was a collaborative effort involving brainstorming and block diagram design. Responsibilities for the Register Transfer Notation (RTN) were divided as follows:
    - **Mahmoud**: Wrote the RTN for R-type instructions.
    - **Suhaib**: Wrote the RTN for I-type instructions.
    - **Abd Alhameed**: Wrote the RTN for J-type instructions and ensured the design diagram was accurate and well-organized.

Implementation in Verilog
The implementation process was thoroughly discussed and shared among all team members. Although tasks were divided, the entire team agreed on a unified implementation approach beforehand. The individual contributions were as follows:

- **Suhaib**: Designed the data path, controllers, and hazard modules. He also supervised the implementation process, ensured correctness, and assisted in debugging the data-path module.
- **Abd Alhameed**: Developed the ALU, program counter (PC) control, data path, and hazard modules. He contributed to debugging and refining the data-path module.
- **Mahmoud**: Focused on the main control unit, hazard modules, register file, memory modules, and other basic components. He also assisted in debugging the data-path module.

Simulation and Testing

- **Suhaib**: Created the testbench in Verilog, provided test cases for the report, and wrote a C++ program to convert instructions into binary code.
- **Mahmoud and Abd Alhameed**: Suggested ideas to enhance the testbench's thoroughness and provided additional test cases.

Report Writing
The entire team collaborated on writing the report, ensuring equal contributions from all members.

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 and J-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 data-path 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 shows cases a robust and scalable pipelined processor architecture that meets the specified requirements, providing a solid foundation for further enhancements and extensions in future projects.