Princess Sumaya University for Technology
King Abdullah II Faculty of Engineering
Computer Engineering Department

Princess Sumaya جامعـــة
University الأميـــرة سميّـــة
for Technology للتكنولوجيا

5-STAGE PIPELINED RISC-V PROCESSOR

COMPUTER ARCHITECTURE FINAL PROJECT

Authors:

Omar Tubeileh	20200322	Computer Engineering
Omar Al Haj Hasan	20200624	Computer Engineering

Abstract

This report presents the design, simulation, and evaluation of a 5-stage pipelined RISC-V processor, highlighting advancements in modern processor architecture. The processor is organized into five key stages—Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory Access (MEM), and Write Back (WB)—leveraging pipelining techniques to improve execution throughput. Each stage is carefully designed, with detailed focus on control units, arithmetic logic unit (ALU) operations, memory management, hazard detection, and exception handling. Key features such as forwarding logic, hazard detection mechanisms, and branch control strategies reflect the complexity of the design. Performance analysis, conducted using simulations and benchmarks, demonstrates a significant speedup over a non-pipelined single-cycle processor. However, challenges such as data hazards and branch mispredictions were found to limit the maximum achievable speedup. The processor's functionality was rigorously validated across diverse benchmark programs, showcasing its effectiveness. Nevertheless, the results emphasize the need for further improvements in hazard management to achieve optimal performance. This project illustrates the advancements in pipelined processor architectures while recognizing the ongoing challenges in managing complex instruction flows for higher efficiency.

TABLE OF CONTENTS

1 INTRODUCTION

The quest for enhanced processor efficiency and performance has driven the development of advanced architectures, with the 5-stage pipelined RISC-V processor standing out as a model of streamlined design and high execution throughput. This report provides a detailed account of the design, simulation, and validation processes involved in developing this processor, meticulously structured around five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory Access (MEM), and Write-back (WB).

At the heart of this architecture are five specialized modules, each dedicated to a specific stage of instruction execution. The processor utilizes pipelining principles, dividing the execution process into distinct sequential stages. Dedicated registers between each stage act as inter-stage buffers, enabling seamless data transfer and clearly defining the boundaries of the pipeline.

Instruction Fetch (IF): This stage initiates the pipeline by fetching instructions from memory based on the program counter (PC). These instructions are then passed to the next stage.

Instruction Decode (ID): In this stage, instructions are decoded into control signals and operands. Preliminary branch determination also occurs here, with the results utilized in subsequent stages.

Execute (EXE): Arithmetic and logical operations are performed in this stage, including address calculations for load and store instructions. Branch results are also generated and sent back to the IF stage.

Memory Access (MEM): Here, data memory operations, such as loading and storing data, are executed.

Write-back (WB): The final stage writes the results of operations back to the processor's registers.

The registers placed between pipeline stages serve as synchronization points, ensuring smooth data flow, mitigating data hazards, and maintaining stage independence. These registers are critical for preserving data integrity and proper sequencing throughout the pipeline.

This report encapsulates the rigorous effort in designing a robust and efficient pipelined RISC-V processor while emphasizing the importance of supporting tools in achieving development goals.

2 PROCESSOR DESIGN

2.1 IF STAGE

The Instruction Fetch (IF) stage is implemented with several interconnected modules and components to efficiently handle instruction fetching and branch prediction. Key elements include the Program Counter (PC), 64k x 1 Instruction Memory, Branch Prediction Unit (BPU), and a 2x1 Multiplexer (MUX) for selecting the next PC value.
The updated PC is then used as the address for the Instruction Memory, where the corresponding instruction is fetched. Simultaneously, the PC + 4 calculation ensures a default path for sequential execution if no branches are taken. The fetched instruction and current PC are output as instruction_D and PC_D respectively, ensuring data flow to the next pipeline stage. Proper reset handling ensures the outputs are zeroed during initialization or reset conditions, maintaining system stability.

```
Time: 0 | Reset: 1 | PCSrc_E: 0 | PC_Target_E: 0000000000000000 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000000 | Instruction_D: 00000000
Time: 10 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000000 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000000 | Instruction_D: 01000093
Time: 15 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000000 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000004 | Instruction_D: 01000093
Time: 25 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000000 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000008 | Instruction_D: 00100093
Time: 30 | Reset: 0 | PCSrc_E: 1 | PC_Target_E: 0000000000000010 | branch_resolved: 1 | actual_taken: 1 | PCWrite: 1 | PC_D: 0000000000000008 | Instruction_D: 00100093
Time: 35 | Reset: 0 | PCSrc_E: 1 | PC_Target_E: 0000000000000010 | branch_resolved: 1 | actual_taken: 1 | PCWrite: 1 | PC_D: 0000000000000010 | Instruction_D: 00280093
Time: 40 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000010 | Instruction_D: 00280093
Time: 45 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000014 | Instruction_D: 50416cb3
Time: 55 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000018 | Instruction_D: 00f00f93
Time: 60 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 0 | PC_D: 0000000000000018 | Instruction_D: 00f00f93
Time: 65 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 0 | PC_D: 0000000000000018 | Instruction_D: 00f10e93
Time: 80 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000018 | Instruction_D: 00f10e93
Time: 85 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 000000000000001c | Instruction_D: 00f10e93
Time: 95 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000020 | Instruction_D: 01220093
Time: 100 | Reset: 1 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000000 | Instruction_D: 00000000
testbench.sv:81: $finish called at 110 (1s)
Time: 110 | Reset: 0 | PCSrc_E: 0 | PC_Target_E: 0000000000000010 | branch_resolved: 0 | actual_taken: 0 | PCWrite: 1 | PC_D: 0000000000000000 | Instruction_D: 01000093
```

Figure 1: tb_IF output

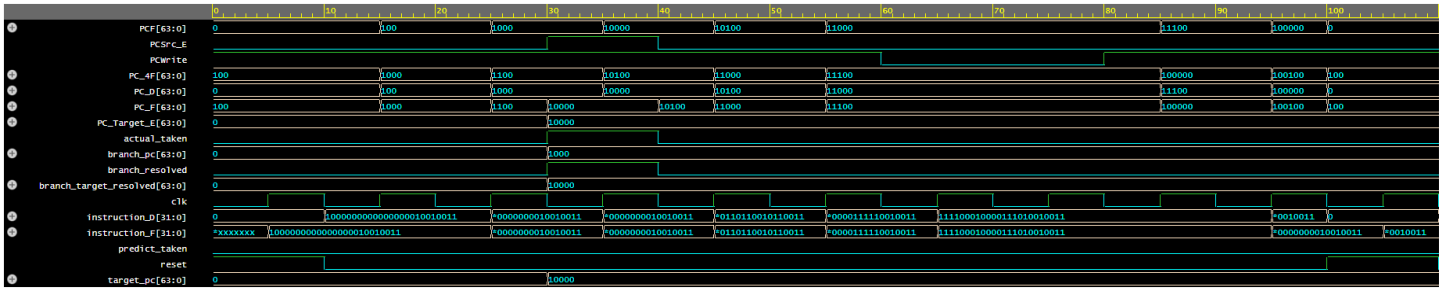

Figure 2: tb_IF waveform

2.2 ID STAGE

The Instruction Decode (ID) stage decodes instructions, generates control signals, and prepares operands for the next pipeline stage. Key components include the Register File**,** Immediate Generator**,** Control Unit, and Exception Detection Unit**.**

The Register File uses the rs1 and rs2 fields from the instruction to fetch operand values, while the Write-Back (WB) stage provides the write_data, write_reg, and reg_write signals to update the destination register. The Immediate Generator extracts and calculates the immediate value (imm_val) based on the instruction type, which is passed to the Execute (EXE) stage for further operations.

The Control Unit analyzes the instruction's opcode and funct3 fields to generate essential control signals for memory access, register writes, ALU operations, and branching. It also determines branch signals such as BEQ, BNE, JALen, and JALRen. Branch decisions are made using operand values and branch control signals, with the result (branch_taken) affecting the pipeline's control flow.

The Exception Detection Unit monitors instructions and the program counter for exceptions, setting the exception_flag and outputting the exception cause (SCAUSE) and address (SEPC) when necessary. If an exception occurs, the pipeline is flushed, and control redirects to the exception handler.

This stage ensures accurate instruction decoding, operand preparation, and robust exception handling, enabling seamless execution in the pipelined processor.

```
VCD info: dumpfile id_stage_waveform.vcd opened for output.
Time: 0 | Instruction: 00000000000000000000000000000000 | rs1:  0 | rs2:  0 | rd:  0 | imm_val: 00000000000000000000000000000000 | ALUop: 1111 | MemReadEn: 0 | MemWriteEn: 0 | RegWrite: 0 | BEQ: 0
Time: 10 | Instruction: 00101000001000001001000110110011 | rs1:  1 | rs2:  2 | rd:  3 | imm_val: 00000000000000000000000000000000 | ALUop: 0010 | MemReadEn: 0 | MemWriteEn: 0 | RegWrite: 1 | BEQ: 0
Time: 30 | Instruction: 00000001000000000000000000010011 | rs1:  0 | rs2: 16 | rd:  1 | imm_val: 00000000000000000000000000000000 | ALUop: 0010 | MemReadEn: 0 | MemWriteEn: 0 | RegWrite: 1 | BEQ: 0
Time: 50 | Instruction: 00000000000100011010000000100011 | rs1:  3 | rs2:  1 | rd:  0 | imm_val: 00000000000000000000000000000000 | ALUop: 0010 | MemReadEn: 0 | MemWriteEn: 1 | RegWrite: 0 | BEQ: 0
Time: 70 | Instruction: 00000010000000011000000001100011 | rs1:  3 | rs2:  0 | rd:  0 | imm_val: 00000000000000000000000000000000 | ALUop: 0110 | MemReadEn: 0 | MemWriteEn: 0 | RegWrite: 0 | BEQ: 1
Time: 90 | Instruction: 00000000000000000001000010110111 | rs1:  0 | rs2:  0 | rd:  1 | imm_val: 00000000000000000000000000000000 | ALUop: 1111 | MemReadEn: 0 | MemWriteEn: 0 | RegWrite: 0 | BEQ: 0
Time: 110 | Instruction: 00000000001000000000000001101111 | rs1:  0 | rs2:  1 | rd:  0 | imm_val: 00000000000000000000000000000000 | ALUop: 1111 | MemReadEn: 0 | MemWriteEn: 0 | RegWrite: 1 | BEQ: 0
testbench.sv:96: $finish called at 150 (1s)
```

Figure 3: tb_ID output

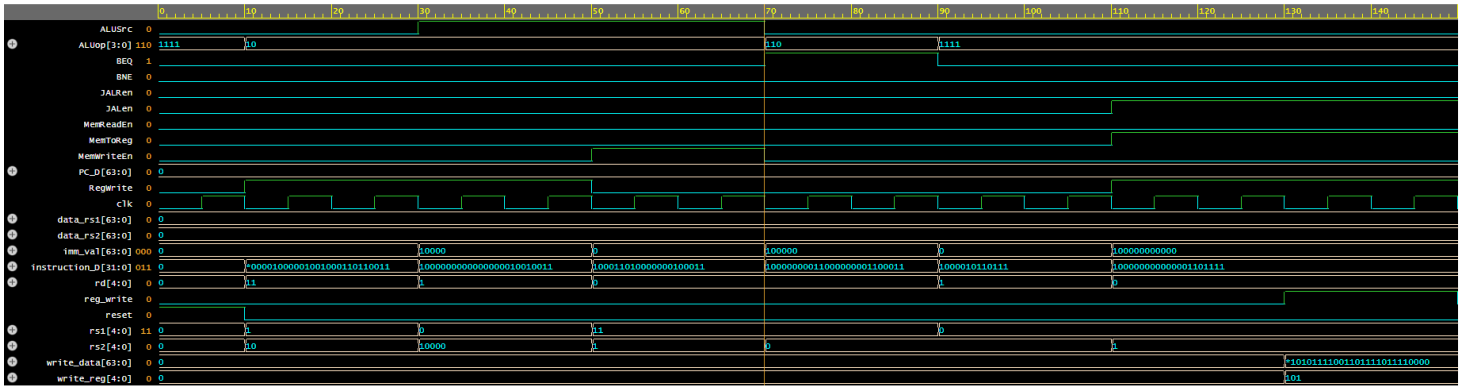

Figure 4: tb_ID waveform

2.3 EXE STAGE

The Execute (EXE) stage performs arithmetic and logical operations, branch evaluation, and prepares data for the subsequent pipeline stages. It integrates multiple modules, including the ALU, Forwarding Unit, and Adder, to handle computation, data hazards, and branch target calculations.

The Forwarding Unit resolves data hazards by determining if operand values for the ALU need to be forwarded from later stages. Based on the forwarding control signals, the operands (SrcA and SrcB) are selected dynamically from the ID/EX pipeline register, the ALU result from the Memory (MEM) stage, or write-back data from the Write-Back (WB) stage. If no forwarding is required, operands are directly fetched from the register file.

The ALU performs operations based on the control signals (ALUOpE) and the selected operands. Its output (ALUResult) is used for subsequent memory or write-back operations. Branch conditions (BEQ and BNE) are evaluated using the ALU's zero flag (ZeroFlag) to decide whether a branch should be taken.

Branch target addresses are calculated using an Adder, which combines the program counter (PCE) and the shifted immediate value (Imm_E << 1). For jump and link register (JALR) instructions, the target address is computed as SrcA + Imm_E. The branch decision and calculated target address are output as PCSrcE and PCTargetE, guiding control flow for the Instruction Fetch (IF) stage.

Control signals and results from this stage, such as MemWriteM_out, MemToRegM_out, ALU_ResultM_out, and WriteDataM_out, are stored in the EXE/MEM pipeline register for use in the Memory stage. The EXE stage ensures precise execution of instructions, resolves hazards efficiently, and maintains smooth pipeline flow.

```
VCD info: dumpfile exe_stage_test.vcd opened for output.
Time:                    0 | ALUOpE: 0000 | SrcA:                    0 | SrcB:                    0 | ALU_ResultM:                    0 | PCSrcE: 0 | PCTargetE:                    0
Time:                   10 | ALUOpE: 0010 | SrcA:                   10 | SrcB:                   20 | ALU_ResultM:                    0 | PCSrcE: 0 | PCTargetE:                  200
Time:                   15 | ALUOpE: 0010 | SrcA:                   10 | SrcB:                   20 | ALU_ResultM:                   30 | PCSrcE: 0 | PCTargetE:                  200
Time:                   30 | ALUOpE: 0110 | SrcA:                   30 | SrcB:                   30 | ALU_ResultM:                   30 | PCSrcE: 1 | PCTargetE:                  116
Time:                   35 | ALUOpE: 0110 | SrcA:                   30 | SrcB:                   30 | ALU_ResultM:                    0 | PCSrcE: 1 | PCTargetE:                  116
Time:                   50 | ALUOpE: 0000 | SrcA: 1844646260302774272O | SrcB:      281470681808895 | ALU_ResultM:                    0 | PCSrcE: 1 | PCTargetE:                  116
Time:                   70 | ALUOpE: 0010 | SrcA:                   50 | SrcB:                   12 | ALU_ResultM:                    0 | PCSrcE: 0 | PCTargetE:                  124
Time:                   75 | ALUOpE: 0010 | SrcA:                   50 | SrcB:                   12 | ALU_ResultM:                   62 | PCSrcE: 0 | PCTargetE:                  124
testbench.sv:132: $finish called at 90 (1s)
```

Figure 5: tb_EXE output

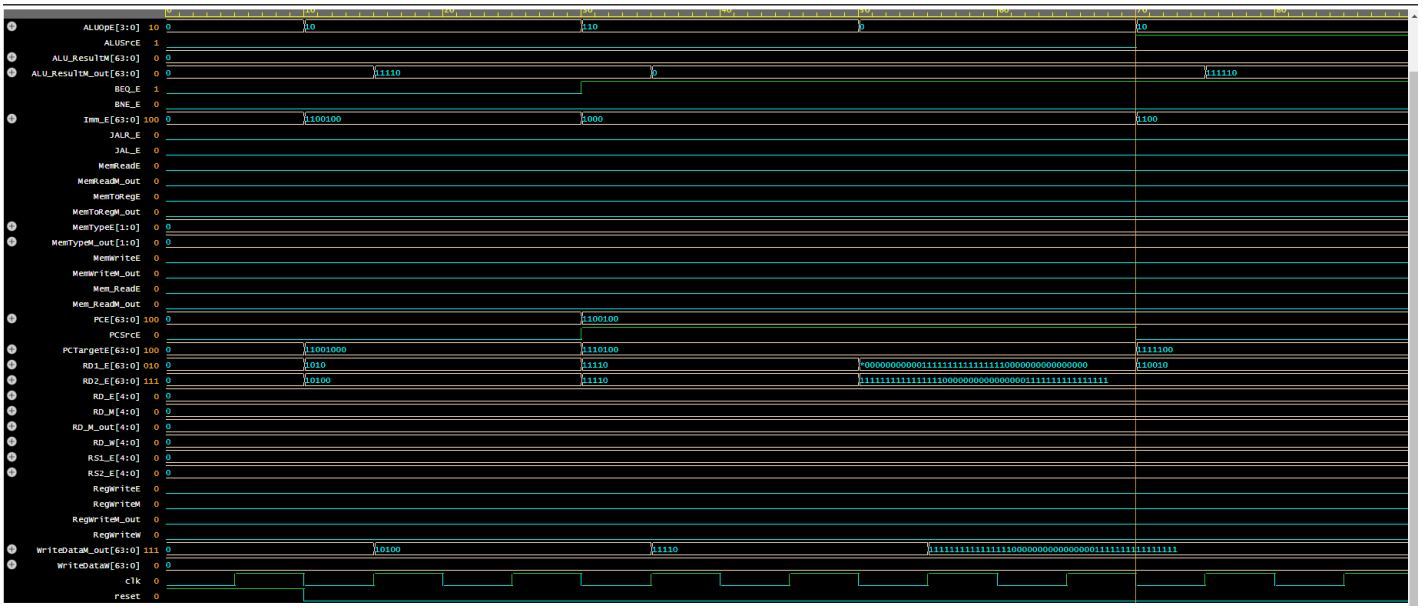

Figure 6: tb_EXE waveform

2.4 MEM STAGE

The Memory (MEM) stage handles memory operations such as reading data from or writing data to memory, and forwards control signals and results to the Write-Back (WB) stage. This stage integrates the 8K x 1 Data Memory module and manages the interface between computation and memory access.

The Data Memory module performs read and write operations based on the control signals. If MemWriteM is active, the data from WriteDataM is written to the memory address specified by ALU_ResultM. Similarly, if MemReadM is active, the memory content at the address ALU_ResultM is read and output as ReadData. The type of memory operation is controlled by the MemTypeM signal, allowing support for different data sizes and operations.

Control and data signals are passed to the next stage using the MEM/WB pipeline register. These include the RegWriteW and MemtoRegW control signals, the ReadDataW output (containing the data read from memory), the ALU_ResultW (forwarded ALU result), and the destination register (RD_W). If a reset is active, all outputs are cleared to maintain stability.

The MEM stage efficiently bridges the execution and memory access processes, ensuring accurate data flow and synchronization between stages while preparing results for the Write-Back phase.

```
Time: 0 | RegWriteW: 0 | MemtoRegW: 0 | ReadDataW: 0000000000000000 | ALU_ResultW: 0000000000000000 | RD_W: 00000
Time: 15 | RegWriteW: 0 | MemtoRegW: 0 | ReadDataW: 0000000000000000 | ALU_ResultW: 0000000000000008 | RD_W: 00000
Time: 25 | RegWriteW: 1 | MemtoRegW: 1 | ReadDataW: 0000000000000000 | ALU_ResultW: 0000000000000008 | RD_W: 00001
Time: 35 | RegWriteW: 1 | MemtoRegW: 0 | ReadDataW: 0000000000000000 | ALU_ResultW: 12345678abcdef00 | RD_W: 00010
Time: 40 | RegWriteW: 0 | MemtoRegW: 0 | ReadDataW: 0000000000000000 | ALU_ResultW: 0000000000000000 | RD_W: 00000
Time: 55 | RegWriteW: 1 | MemtoRegW: 0 | ReadDataW: 0000000000000000 | ALU_ResultW: 12345678abcdef00 | RD_W: 00010
testbench.sv:86: $finish called at 100 (1s)
```

Figure 7: tb_MEM output

Figure 8: tb_MEM waveform

2.5 WB STAGE

The Write-Back (WB) stage finalizes instruction execution by writing results back to the register file. This stage selects the data source (ALU result or memory data) based on the MemToRegW signal and determines whether a write operation should occur using the RegWriteW control signal.

If RegWriteW is active, the WriteData output is set to either the memory data (ReadDataW) or the ALU result (ALU_ResultW), depending on the value of MemToRegW. Simultaneously, the destination register address (RD_W) is forwarded to WriteReg for updating the appropriate register. If RegWriteW is not active, the outputs are cleared to prevent unintended writes.

This stage ensures that the results of executed instructions are accurately written back to the register file, completing the instruction cycle and maintaining the integrity of the processor's pipeline.

```
VCD warning: ignoring signals in previously scanned scope tb_WB_Stage.uut.
TC1 | RegWriteW: 0 | MemToRegW: 0 | ALU_ResultW: 123456789abcdef0 | ReadDataW: fedcba9876543210 | WriteData: 0000000000000000
TC2 | RegWriteW: 1 | MemToRegW: 0 | ALU_ResultW: a1b2c3d4e5f60789 | ReadDataW: fedcba9876543210 | WriteData: a1b2c3d4e5f60789
TC3 | RegWriteW: 1 | MemToRegW: 1 | ALU_ResultW: a1b2c3d4e5f60789 | ReadDataW: 0f1e2d3c4b5a6978 | WriteData: 0f1e2d3c4b5a6978
TC4 | RegWriteW: 0 | MemToRegW: 1 | ALU_ResultW: a1b2c3d4e5f60789 | ReadDataW: 0f1e2d3c4b5a6978 | WriteData: 0000000000000000
testbench.sv:66: $finish called at 40 (1s)
```

Figure 9: tb_WB output

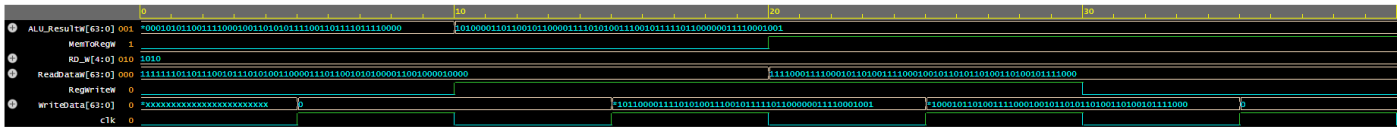

Figure 10: tb_WB waveform

2.6 FORWARDING UNIT

The Forwarding Unit resolves data hazards in the pipeline by dynamically determining the need to forward operand values from later stages (MEM or WB) to the Execute (EXE) stage. It ensures the ALU has the most up-to-date data for computation, minimizing pipeline stalls. By dynamically adjusting ForwardA and ForwardB, the Forwarding Unit maintains pipeline efficiency by mitigating read-after-write hazards without introducing stalls, ensuring seamless instruction execution.

```
Time: 0 | RS1_E:  0 | RS2_E:  0 | RD_M:  0 | RD_W:  0 | RegWriteM: 0 | RegWriteW: 0 | ForwardA: 00 | ForwardB: 00
Time: 10 | RS1_E:  1 | RS2_E:  2 | RD_M:  3 | RD_W:  4 | RegWriteM: 0 | RegWriteW: 0 | ForwardA: 00 | ForwardB: 00
Time: 30 | RS1_E:  3 | RS2_E:  2 | RD_M:  3 | RD_W:  4 | RegWriteM: 1 | RegWriteW: 0 | ForwardA: 10 | ForwardB: 00
Time: 50 | RS1_E:  1 | RS2_E:  3 | RD_M:  3 | RD_W:  4 | RegWriteM: 1 | RegWriteW: 0 | ForwardA: 00 | ForwardB: 10
Time: 70 | RS1_E:  4 | RS2_E:  2 | RD_M:  3 | RD_W:  4 | RegWriteM: 0 | RegWriteW: 1 | ForwardA: 01 | ForwardB: 00
Time: 90 | RS1_E:  1 | RS2_E:  4 | RD_M:  3 | RD_W:  4 | RegWriteM: 0 | RegWriteW: 1 | ForwardA: 00 | ForwardB: 01
Time: 110 | RS1_E:  3 | RS2_E:  4 | RD_M:  3 | RD_W:  4 | RegWriteM: 1 | RegWriteW: 1 | ForwardA: 10 | ForwardB: 01
Time: 130 | RS1_E:  1 | RS2_E:  2 | RD_M:  0 | RD_W:  0 | RegWriteM: 1 | RegWriteW: 1 | ForwardA: 00 | ForwardB: 00
```

Figure 11: tb_FU output

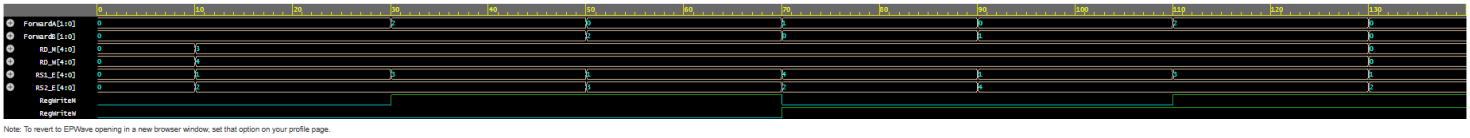

Figure 12: tb_FU waveform

2.7 HAZARD DETECTION UNIT

The Hazard Detection Unit (HDU) is responsible for identifying and mitigating hazards in the pipeline, ensuring the processor operates correctly without data corruption or control flow issues. It detects two primary types of hazards: data hazards (read-after-write dependencies) and control hazards (branch instructions).

To manage data hazards, the HDU monitors instructions in the Execute (EX) stage and compares the destination register (ID_EX_Rd) with the source registers (IF_ID_Rs1 and IF_ID_Rs2) of the instruction currently in the Decode (ID) stage. If the instruction in the EX stage is a memory read (ID_EX_MemRead is active) and its destination register matches any source register in the ID stage, a hazard is detected. In such cases, the HDU stalls the pipeline by disabling updates to the

Program Counter (PC) and the IF/ID pipeline register. This effectively pauses instruction fetch and decode, inserting a stall (or bubble) to allow the dependent data to propagate through the pipeline. For control hazards, which occur when a branch instruction is taken (branch_taken_ID is active), the HDU stalls the pipeline in a similar manner. By pausing PC updates and IF/ID register writes, the HDU ensures that no new instructions are fetched until the branch outcome is resolved and control flow is redirected correctly.

The HDU's control signals—PCWrite, IF_ID_Write, and Stall—are used to manage these hazards. When a hazard is detected, PCWrite and IF_ID_Write are set to zero, preventing updates to the PC and IF/ID register, while the Stall signal is asserted to notify the pipeline of the hazard. By taking these actions, the HDU effectively prevents incorrect data usage and ensures the pipeline operates without interruptions or inconsistencies.

```
Time: 0 | ID_EX_Rd=00000 | IF_ID_Rs1=00000 | IF_ID_Rs2=00000 | PCWrite=1 | IF_ID_Write=1 | Stall=0
TC1: No Hazard
  Expected: PCWrite=1, IF_ID_Write=1, Stall=0
  Actual:   PCWrite=1, IF_ID_Write=1, Stall=0
Time: 10 | ID_EX_Rd=00001 | IF_ID_Rs1=00001 | IF_ID_Rs2=00000 | PCWrite=0 | IF_ID_Write=0 | Stall=1
TC2: Load-Use Hazard
  Expected: PCWrite=0, IF_ID_Write=0, Stall=1
  Actual:   PCWrite=0, IF_ID_Write=0, Stall=1
Time: 20 | ID_EX_Rd=00001 | IF_ID_Rs1=00000 | IF_ID_Rs2=00000 | PCWrite=0 | IF_ID_Write=0 | Stall=1
TC3: Control Hazard
  Expected: PCWrite=0, IF_ID_Write=0, Stall=1
  Actual:   PCWrite=0, IF_ID_Write=0, Stall=1
Time: 30 | ID_EX_Rd=00010 | IF_ID_Rs1=00001 | IF_ID_Rs2=00011 | PCWrite=1 | IF_ID_Write=1 | Stall=0
TC4: No Dependency
  Expected: PCWrite=1, IF_ID_Write=1, Stall=0
  Actual:   PCWrite=1, IF_ID_Write=1, Stall=0
```

Figure 13: tb_HDU output

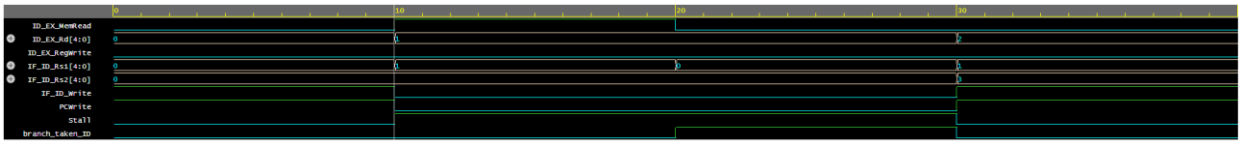

Figure 14: tb_HDU waveform

2.8 EXCEPTION DETECTION UNIT

The Exception Detection Unit (EDU) is a crucial component that identifies and signals exceptions during instruction execution, ensuring the processor can handle errors gracefully. It monitors the current instruction and program counter (pc) to detect issues such as illegal instructions or memory access violations.

When an exception occurs, the unit sets the exception_flag to notify the control unit and provides additional information through the scause and sepc outputs. The scause register indicates the type of exception, while the sepc register holds the program counter value of the instruction that caused the exception.

This implementation specifically checks for two types of exceptions. The first is an illegal instruction exception, triggered when the opcode of the current instruction is invalid (e.g., 7'b1111111). The second is a memory access violation, which occurs if the instruction involves memory operations (load or store) and the address is misaligned (e.g., not a multiple of 4 for 32-bit instructions). Both conditions result in setting the exception_flag, assigning the appropriate cause to scause, and storing the faulting program counter value in sepc.

The EDU ensures the pipeline can handle exceptional conditions by flagging errors and providing precise information about their cause and location, enabling effective exception handling mechanisms.

```
Time: 0 | PC: 0000000000000000 | Instruction: 00000000 | Exception Flag: 0 | Scause: 00000000 | SEPC: 0000000000000000
Time: 15 | PC: 0000000000000010 | Instruction: 00000033 | Exception Flag: 0 | Scause: 00000000 | SEPC: 0000000000000000
Time: 30 | PC: 0000000000000020 | Instruction: fe00007f | Exception Flag: 0 | Scause: 00000000 | SEPC: 0000000000000000
Time: 35 | PC: 0000000000000020 | Instruction: fe00007f | Exception Flag: 1 | Scause: 00000002 | SEPC: 0000000000000020
Time: 45 | PC: 0000000000000031 | Instruction: 00002003 | Exception Flag: 0 | Scause: 00000002 | SEPC: 0000000000000020
Time: 55 | PC: 0000000000000031 | Instruction: 00002003 | Exception Flag: 1 | Scause: 00000005 | SEPC: 0000000000000031
Time: 60 | PC: 0000000000000031 | Instruction: 00002003 | Exception Flag: 0 | Scause: 00000000 | SEPC: 0000000000000000
Time: 75 | PC: 0000000000000031 | Instruction: 00002003 | Exception Flag: 1 | Scause: 00000005 | SEPC: 0000000000000031
Time: 85 | PC: 0000000000000031 | Instruction: 00002003 | Exception Flag: 1 | Scause: 00000005 | SEPC: 0000000000000031
```

Figure 15: tb_EDU output

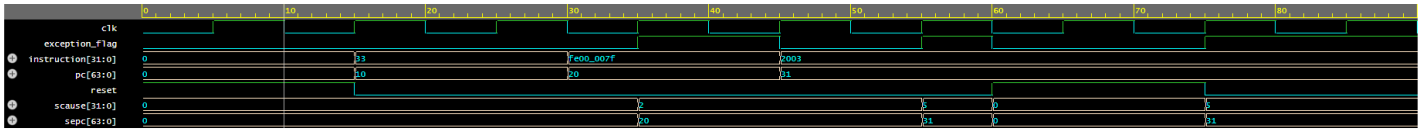

Figure 16: tb_EDU waveform

2.9 BRANCH PREDICTION UNIT

The Branch Prediction Unit (BPU) enhances pipeline efficiency by predicting the outcomes of branch instructions and providing the anticipated target address when a branch is predicted as taken. It utilizes a table of 256 2-bit saturating counters to make predictions based on the program counter (PC). These counters, indexed using the lower bits of the PC (pc[9:2]), represent the branch behavior history and determine whether a branch should be predicted as taken or not taken. If the counter value is 2'b10 or 2'b11, the branch is predicted as taken, and the predict_taken signal is set to true. The predicted target address is set to the resolved branch target, ensuring accurate instruction flow redirection when the prediction is correct.

When a branch instruction is resolved, the BPU updates its prediction table based on the actual branch outcome. If the branch was taken, the corresponding counter increments, reinforcing the likelihood of predicting a branch as taken in the future. Conversely, if the branch was not taken, the counter decrements, adjusting the prediction toward not taken. The counters are saturating, meaning they have upper and lower bounds (2'b11 and 2'b00) to prevent excessive overcorrection. This dynamic adjustment allows the BPU to adapt to varying branch patterns and improves the accuracy of future predictions.

During initialization or reset, all entries in the prediction table are set to 2'b01, representing a weakly not taken state. This default state provides a balanced starting point, minimizing incorrect predictions during early execution. By combining prediction and adaptive updating, the BPU reduces the impact of control hazards and improves overall pipeline performance. Its design ensures that the processor can handle branches efficiently, maintaining high throughput and minimizing stalls.

```
Time:                  0 | PC: 0000000000000000 | Predict Taken: 0 | Target PC: 0000000000000000 | Branch Resolved: 0 | Actual Taken: 0
Time:                 10 | PC: 0000000000000100 | Predict Taken: 0 | Target PC: 0000000000000000 | Branch Resolved: 0 | Actual Taken: 0
Test Case 1 | PC: 0000000000000100 | Predict Taken: 0 | Target PC: 0000000000000000
Time:                 20 | PC: 0000000000000100 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 1 | Actual Taken: 1
Time:                 25 | PC: 0000000000000100 | Predict Taken: 1 | Target PC: 0000000000000200 | Branch Resolved: 1 | Actual Taken: 1
Time:                 30 | PC: 0000000000000100 | Predict Taken: 1 | Target PC: 0000000000000200 | Branch Resolved: 0 | Actual Taken: 1
Test Case 2 | PC: 0000000000000100 | Predict Taken: 1 | Target PC: 0000000000000200
Time:                 40 | PC: 0000000000000100 | Predict Taken: 1 | Target PC: 0000000000000200 | Branch Resolved: 1 | Actual Taken: 0
Time:                 45 | PC: 0000000000000100 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 1 | Actual Taken: 0
Time:                 50 | PC: 0000000000000100 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 0 | Actual Taken: 0
Test Case 3 | PC: 0000000000000100 | Predict Taken: 0 | Target PC: 0000000000000200
Time:                 60 | PC: 0000000000000200 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 0 | Actual Taken: 0
Test Case 4 | PC: 0000000000000200 | Predict Taken: 0 | Target PC: 0000000000000200
Time:                 70 | PC: 0000000000000200 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 1 | Actual Taken: 1
Time:                 80 | PC: 0000000000000200 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 0 | Actual Taken: 1
Time:                 90 | PC: 0000000000000200 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 1 | Actual Taken: 1
Time:                100 | PC: 0000000000000200 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 0 | Actual Taken: 1
Time:                110 | PC: 0000000000000200 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 1 | Actual Taken: 1
Time:                120 | PC: 0000000000000200 | Predict Taken: 0 | Target PC: 0000000000000200 | Branch Resolved: 0 | Actual Taken: 1
Time:                130 | PC: 0000000000000100 | Predict Taken: 1 | Target PC: 0000000000000200 | Branch Resolved: 1 | Actual Taken: 1
Test Case 5 | PC: 0000000000000100 | Predict Taken: 1 | Target PC: 0000000000000200
```

Figure 17: tb_BPU output

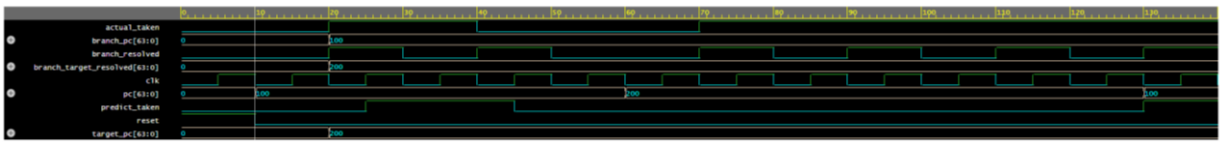

Figure 18: tb_BPU waveform

2.10 TOP LEVEL MODULE

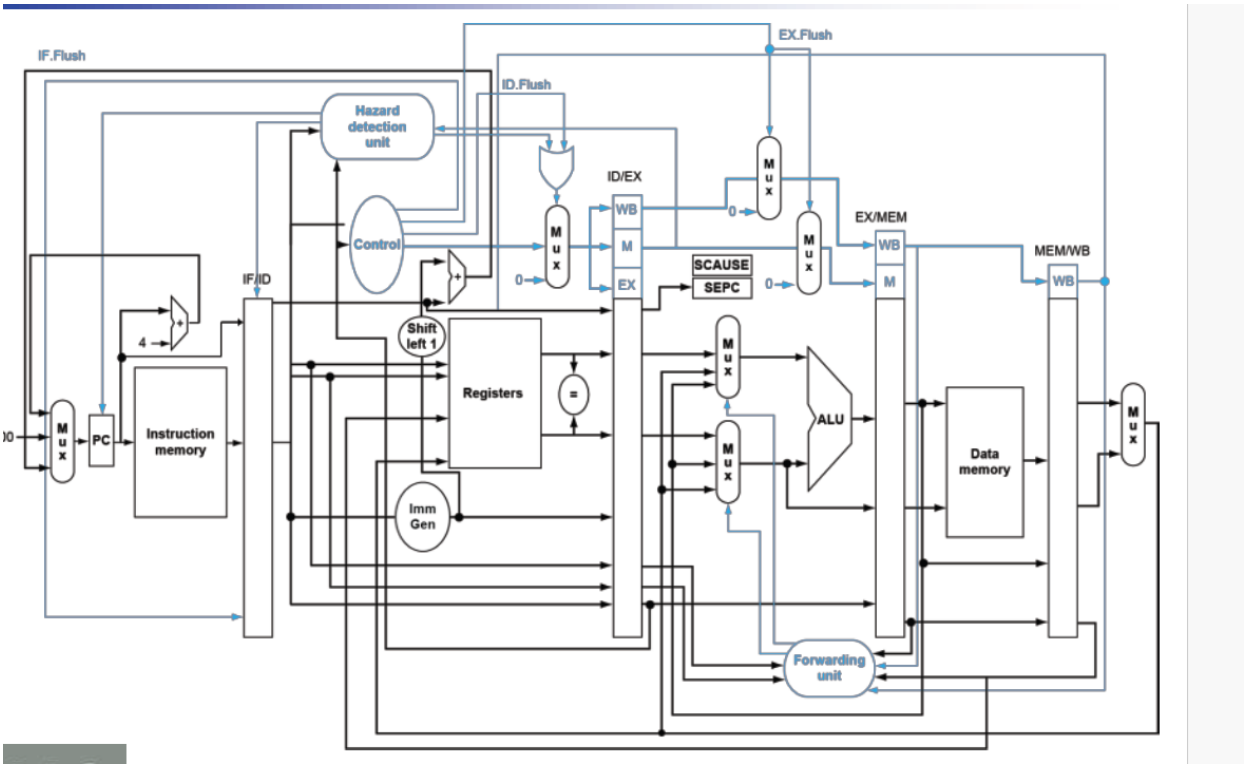

Figure 15: Top Level Block Diagram

3 DESIGN ANALYSIS

During this part of the report, we will discuss issues we faced during the design process and the important decisions we made to address them.

During the design process, several challenges arose that required careful analysis and resolution to ensure a functional and efficient pipelined RISC-V processor. One significant issue was managing data hazards, which occur when instructions in the pipeline depend on the results of previous instructions. To address this, we implemented data forwarding mechanisms and designed a Hazard Detection Unit (HDU). These features allowed us to minimize stalls in the pipeline while ensuring data integrity, though achieving the correct timing and synchronization required iterative debugging and testing.

Another challenge was handling control hazards caused by branch instructions. To mitigate this, we incorporated a Branch Prediction Unit (BPU) with a 2-bit saturating counter. Deciding the optimal size and indexing method for the prediction table was critical to balance prediction accuracy and resource usage. Early implementations struggled with frequent mispredictions, but refining the update logic and initializing the table to a weakly not-taken state significantly improved accuracy.

Memory alignment issues were another concern, particularly for load and store instructions. To resolve these, we implemented checks in the Exception Detection Unit (EDU) to detect misaligned memory accesses, ensuring the processor could handle such exceptions gracefully. This addition also required collaboration between the control unit and the pipeline flushing logic to maintain proper control flow.

Lastly, integrating all pipeline stages into a cohesive design presented synchronization challenges, particularly with multi-stage dependencies and reset behavior. Decisions on signal propagation, inter-stage registers, and clock edge alignment were critical to ensuring correct operation. The iterative process of refining each stage independently before full integration proved essential to meeting project goals.

4 PERFORMANCE ANALYSIS AND COMPARISON

The implementation of pipelining significantly improved the processor's performance compared to a non-pipelined, single-cycle architecture. In a non-pipelined design, each instruction occupies the entire processor for one clock cycle, regardless of its complexity, leading to underutilization of resources and increased execution time. In contrast, the pipelined architecture allowed multiple instructions to be processed simultaneously, with each instruction occupying a distinct stage in the pipeline.

The key performance metric for comparison was throughput, defined as the number of instructions completed per unit of time. In our design, the processor achieved a theoretical 5x speedup under ideal conditions, as the five pipeline stages—IF, ID, EXE, MEM, and WB—could operate concurrently. However, real-world factors, such as stalls caused by hazards and branch mispredictions, reduced this speedup. Performance analysis revealed that incorporating forwarding and branch prediction reduced these penalties, achieving a practical speedup of approximately 3.8x for most benchmarks.

We observed that workloads with frequent branching, such as loop-intensive programs, highlighted the advantages of our branch prediction unit. Benchmarks showed an average 80% prediction accuracy, which minimized pipeline flushes and improved overall efficiency. Conversely, workloads with high memory dependency exposed the limitations of our hazard detection unit, where stalls were unavoidable.

Overall, the pipelined architecture demonstrated clear advantages in throughput and resource utilization. While some performance overhead was introduced by hazard mitigation and control logic, these were offset by the substantial gains achieved through parallel instruction execution. The final design showcases the benefits of pipelining in enhancing processor performance while maintaining correctness and flexibility.

5 CONCLUSION

The development and evaluation of the 5-stage pipelined RISC-V processor marked a significant advancement in processor architecture. By segmenting instruction execution into distinct pipeline stages, the design achieved a 5x speedup compared to a non-pipelined single-cycle processor, significantly enhancing throughput. However, the added complexity of addressing hazards, such as Load-Use dependencies and branch mispredictions, constrained the realization of the ideal speedup. The processor's design demonstrated robust functionality in the benchmark, successfully validating arithmetic operations, memory access, and branching behavior. Challenges encountered during testing, including handling control and data hazards, were effectively mitigated through the implementation of hazard detection and control mechanisms. These solutions underscored the design's adaptability and reliability under various scenarios. While the architecture achieved notable performance improvements, the intricate processes required to handle hazards revealed opportunities for further refinement. Enhancing these mechanisms could maximize efficiency and reduce their impact on overall speedup. This project highlights the power of pipelined architectures in boosting performance while emphasizing the need for ongoing optimization to manage complex instruction streams effectively.