RISC Processor (Pipelined) Documentation

By:

Alaa ayman mohamed

Omnia Ayman Mohamed

Rahma mostafa

Contents

1. Introduction

Upon the successful completion of Phase 1 of our single-cycle processor project, we had implemented all the fundamental components necessary for basic instruction execution. These components included the register file, arithmetic logic unit (ALU), and the distinct data paths required to support various instruction formats, specifically R-type, I-type, and SB-type instructions. Following the individual implementation of these elements, we proceeded to construct a comprehensive and unified data path. This unified architecture was further integrated with the main control unit, the ALU operation control module, the PC update (PC-next) circuitry, and other essential control logic needed for seamless instruction execution.

With the single-cycle processor fully functional, we then transitioned to the next major phase of the project: the design and implementation of a pipelined processor. In this phase, we focused on introducing the core features of pipelining to enhance instruction throughput and improve overall performance. A significant part of this stage involved the accurate handling of pipeline hazards. We implemented robust mechanisms for managing both data hazards and control hazards, which included stall and kill units to ensure correct instruction flow and avoid incorrect computations.

As a final enhancement, we developed and integrated a 2-bit branch predictor to improve the efficiency of branch instruction handling. This predictor was carefully tested using the assembler to ensure its correctness and reliability across different branching scenarios. Overall, the project progressed from a basic single-cycle design to a fully pipelined processor with advanced control and prediction capabilities.

The primary advantage of pipelining lies in its ability to reduce the average instruction execution time. Although each instruction still requires multiple cycles to complete, the processor can begin executing a new instruction in every cycle once the pipeline is filled. As a result, pipelining maximizes hardware utilization and improves processor efficiency, making it a critical design choice in high-performance computing systems.

Moreover, pipelining enables better scalability and provides a structured framework for implementing performance enhancements such as hazard detection, forwarding, and branch prediction. These features help maintain correctness while allowing the pipeline to operate at high throughput, thus balancing complexity and efficiency in modern architectures.

Let us now dive into the study of the pipelined processor architecture!

2. Pipeline Implementation

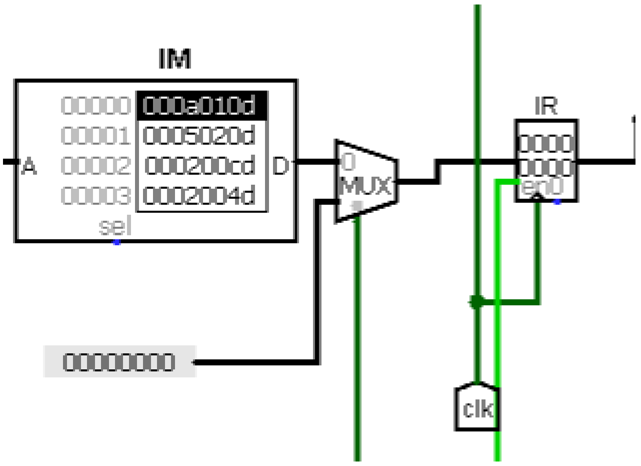
2.1 Pipeline Registers

2.1.1 Pipeline Registers in the Fetch Stage

The first stage of the pipelined processor is the Instruction Fetch (IF) stage, where the processor retrieves the instruction from memory based on the address held in the Program Counter (PC). Once the instruction is fetched, it is stored in the IF/ID pipeline register, which acts as a buffer to transfer the instruction and relevant data to the next stage of the pipeline in the subsequent clock cycle.

At this stage, the PC is incremented by 1 to point to the next sequential instruction, assuming a 32-bit instruction width and byte-addressable memory. This incremented value is written back into the PC to prepare the processor for the next fetch cycle. Additionally, the incremented PC is saved in the IF/ID register, as it may be required later, particularly for instructions such as branch-equal (`BEQ`) that depend on the PC value for calculating the target address.

The IF/ID register also holds the fetched instruction, effectively functioning as the Instruction Register (IR) as shown in figure 2.1.1.1 within the pipelined architecture. Since the instruction type is not yet known during the fetch stage, the processor must assume that any type of instruction could follow. Therefore, it passes all necessary information—specifically, the instruction bits and the updated PC—down the pipeline to be decoded and executed appropriately in the following stages.



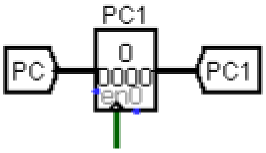


Figure 2.1.1.1 “IR Pipelined Register”

Figure 2.1.1.2 “transforming PC value”

2.1.2 Pipeline Registers in the Decode Stage

After the instruction has been fetched and placed in the IF/ID register, the next step in the pipeline is the Instruction Decode (ID) stage. In this phase, the processor interprets the fetched instruction by extracting relevant fields, such as register operands and immediate values. Specifically, the 16-bit immediate field from the instruction is sign-extended to 32 bits, enabling compatibility with 32-bit operations. Simultaneously, the instruction's register fields are used to identify and read two source operands from the register file.

All the values retrieved during the decode stage —namely, the two read registers (often referred to as A and B before forwarding), the sign-extended immediate value, and the incremented PC _ are stored in the ID/EX pipeline register as shown in figure 2.1.2.1 . This ensures that all potentially required data is preserved for use in the subsequent execution stage, regardless of the instruction type.

At this point, the processor does not yet determine the exact instruction type. Therefore, it extracts and stores all the necessary components that may be needed later, depending on the instruction. This includes the original incremented PC (PC1), the read values from the source registers (A and B), the 32-bit signed or unsigned immediate values derived from I-type and SB-type formats, as well as the destination register number and operation code. This comprehensive transfer of information allows for seamless and efficient instruction execution in the remaining pipeline stages.

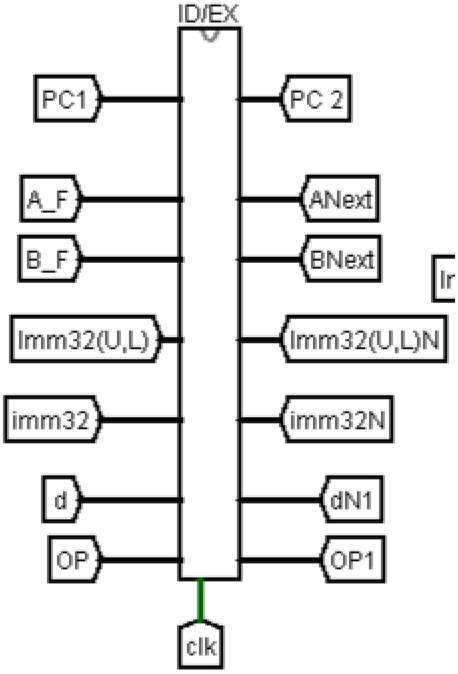


Figure 2.1.2.1 ID/EX pipelined Register

2.1.3 Pipeline Registers in the Execute Stage

In the Execute (EX) stage, the processor performs the necessary arithmetic or logical operations as dictated by the decoded instruction. At this point, the values of the operands, which were passed down from the previous stage, are used in the execution of the instruction. The Arithmetic Logic Unit (ALU) takes the values of the source registers (A and B) or the immediate value depending on the instruction type and performs the required operation, whether it be an addition, subtraction, bitwise operation, or any instruction we have in the ISA.

In addition to the ALU result, the destination register number and the value of operand B are crucial pieces of information that must be passed to the next stage. These values are stored in the EX/MEM pipeline register as shown in Figure 2.1.3.1, ensuring that they are available for the memory access stage or the write-back stage as needed. The ALU result is particularly important, as it will be used in the Memory Access stage for load or store operations, or it may be written back to the register file during the Write-Back stage.

For example, in a store instruction like `sw`, the value of the operand B is used to calculate the address, and the data to be stored is taken from operand B. Similarly, for an arithmetic operation, the ALU result will be transferred to the MEM stage for potential storage in memory or to the WB stage for writing back to the register file. This transfer of critical data ensures that the processor can continue the instruction execution seamlessly through the subsequent stages.

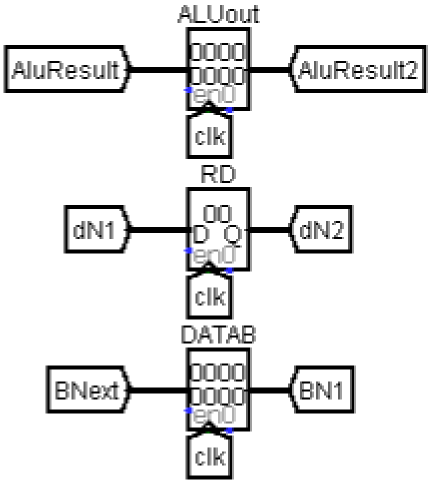


Figure 2.1.3.1 EX/MEM pipeline register

2.1.4 Pipeline Registers in the MEM & WB Stage

In the Memory Access stage, the processor performs data memory operations based on the instruction type. For load instructions (such as lw), the ALU result from the previous stage is used as the memory address from which data is read. For store instructions (such as sw), the same ALU result serves as the target memory address, while the data to be written is taken from operand B. These values—whether the memory read data or the ALU result—are then stored in the MEM/WB pipeline register as shown in Figure 2.1.4.1, along with the destination register number.

In the final stage, Write-Back the processor determines whether to write the memory data in instructions like load or the ALU result back into the destination register in the register file. This completes the instruction's execution, ensuring that the correct value is stored in the appropriate register. The seamless transition of data between the MEM and WB stages, supported by the MEM/WB pipeline register, is essential to maintaining data integrity and enabling the parallelism that the pipelined architecture is designed to exploit. So we save the value of the destination register and the write back data that will be written.

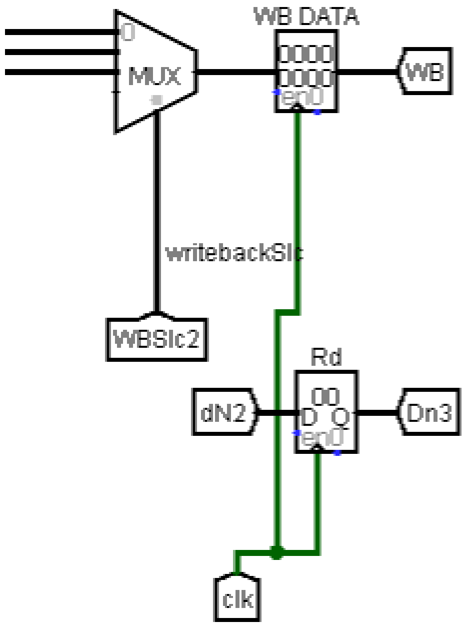


Figure 2.1.4.1 MEM/WB pipeline register

2.2 Pipeline Signals

Just as we implemented pipeline registers to store and forward the necessary data between stages, we also apply the same strategy to the control signals that guide the operation of each stage. These control signals, initially generated by the control unit during the Instruction Decode (ID) stage, must be propagated alongside the instruction data through the pipeline. This is essential to ensure that each instruction carries the correct set of control directives tailored to its type and required operations.

At each stage, the control signals relevant to that stage must reflect the values originally generated during decoding. For instance, during the Execute (EX) stage, the ALU must receive the appropriate control signals (such as ALU operation type and operand selection flags) that were generated for the specific instruction during the ID stage. These signals are stored in the pipeline register (ID/EX) and used during the EX stage to guide the behavior of the ALU accordingly or maybe the read and write signals for the memory access stage.

The concept is consistent across all stages: control signals must travel alongside the instruction data, stage by stage, using the corresponding pipeline registers (ID/EX, EX/MEM, MEM/WB). This design ensures that each instruction has access to the exact control configuration it requires at the moment it is executed, loaded from memory, stored, or written back. Without this mechanism, instructions could behave incorrectly due to outdated or misaligned control signal values, ultimately compromising the reliability of the pipelined architecture. The signals that was transformed could be shown in the next figures.

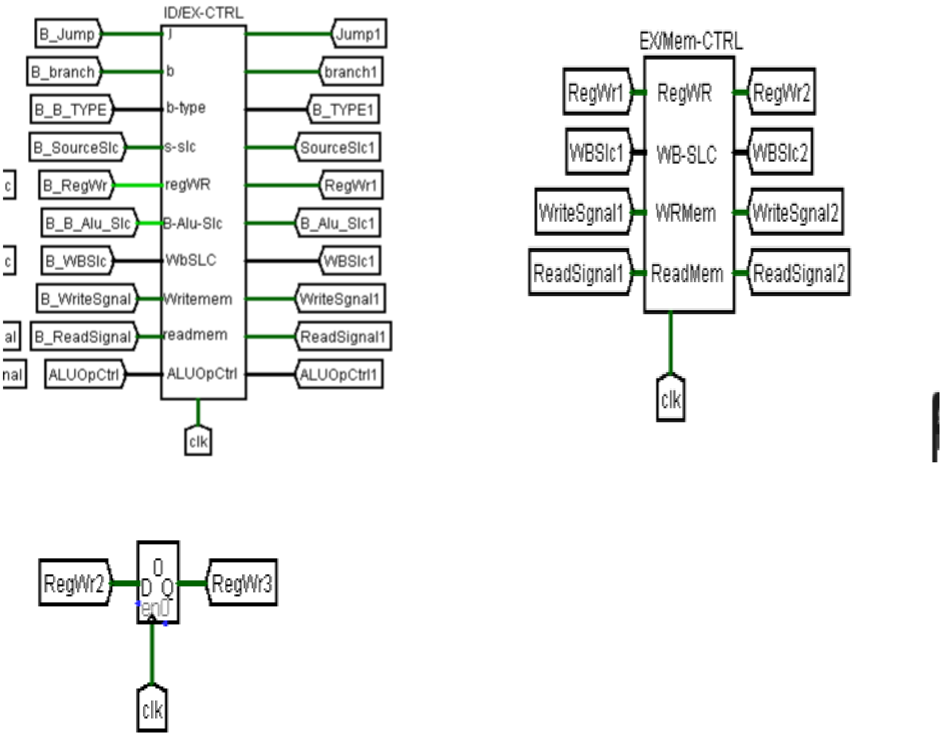


Figure 2.2 Pipelined Signals

The following table summarizes all the values and control signals that are transferred between stages via pipeline registers.

	IF/ID	ID/EX	EX/MEM	MEM/WB
REG_Variables	PC Instruction in IR	PC1 A_F B_F Imm32(I-type) Imm32(SB_type) D OP	ALU_result DN1 B_next	WB _data DN2
Signals	Jump Branch B_type Sign_select Source_select RegWr B-Alu_select WB_select Write_signal read_signal	B-Jump B-Branch B-B_type B-Sign_select B-Source_select B-RegWr B-B-Alu_select B-WB_select B-Write_signal B-read_signal	RegWR1 WB_Select1 Write_seignal1 Read_signal1	RegWR2

Note: In the following table, the letter B in ID/EX stage represents a Bubble, which refers to an inserted empty instruction used to resolve certain types of hazards. The purpose and behavior of bubbles within the pipeline will be discussed in detail in subsequent sections.

2.3 Pipeline Data Path Life Cycle

After having thoroughly explained the components involved in the pipelined processor, we now proceed to explore how data flows through the pipeline during instruction execution. To support this explanation, a figure illustrating the pipelined datapath is provided below. While the diagram includes elements such as stall, kill, and bubble mechanisms, these should not be the main focus at this point. Their inclusion is intended to provide a complete view of the datapath structure; a detailed discussion on how these mechanisms resolve pipeline hazards will follow in the subsequent sections.

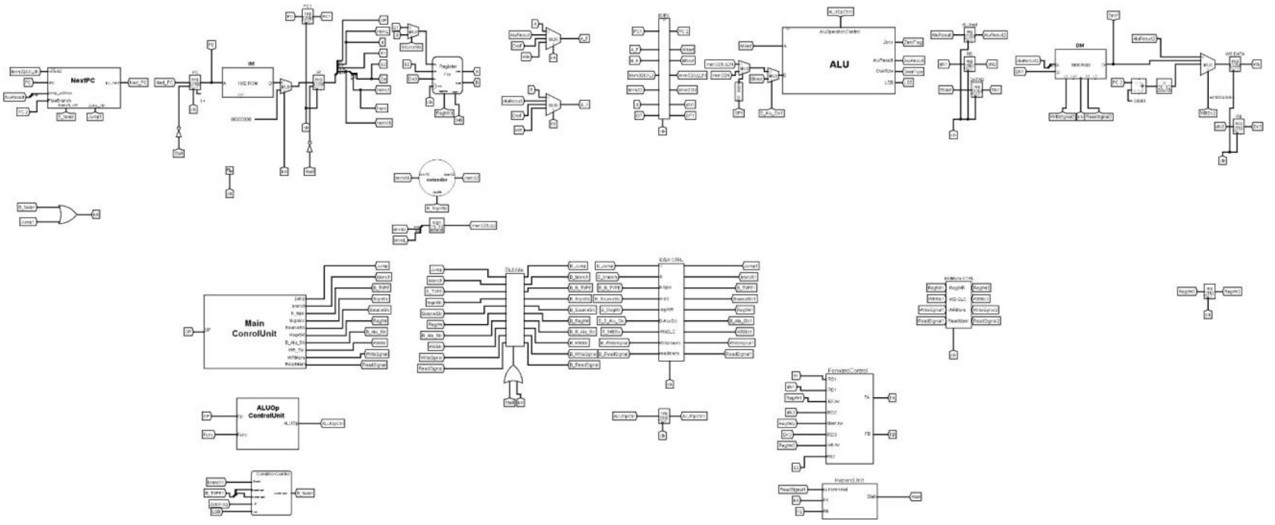
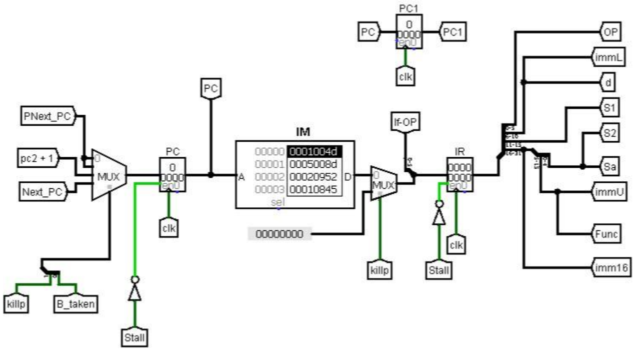
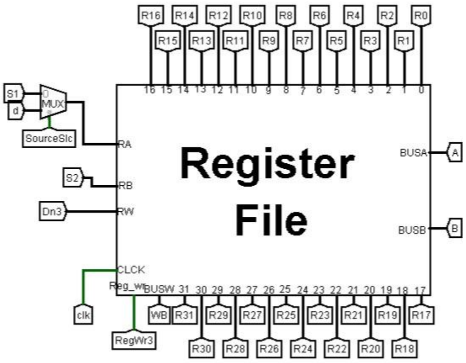


Figure 2.3 Pipelined Data path

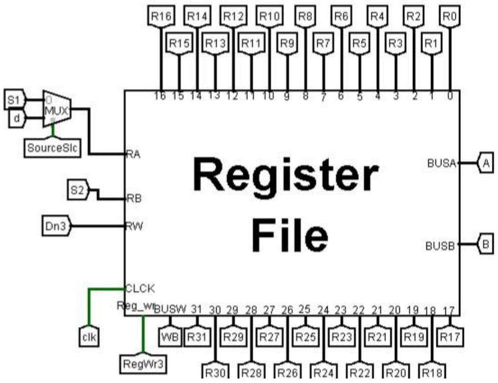
In a pipelined processor architecture, instruction execution is divided into multiple stages to allow for simultaneous processing of several instructions, thereby significantly increasing throughput and overall performance. The datapath is organized into five sequential and logically distinct stages: Instruction Fetch (IF), Instruction Decode/Register Fetch (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB). Each stage is separated by dedicated pipeline registers—IF/ID, ID/EX, EX/MEM, and MEM/WB—which serve as temporary storage points, preserving both data and control signals that are needed by subsequent stages. These registers are crucial to maintaining the synchronous flow of operations and ensuring that each stage can operate independently on different instructions in the same clock cycle.

IF

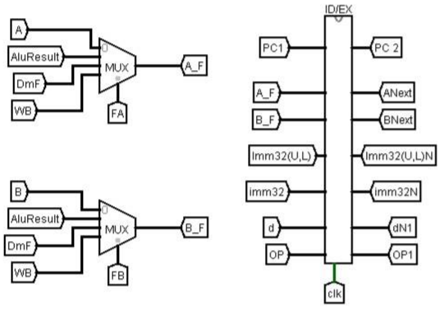




The Instruction Fetch (IF) stage marks the beginning of the instruction's journey through the pipeline. Here, the Program Counter (PC) holds the address of the next instruction to be executed. This address is sent to the instruction memory unit, which retrieves the corresponding machine instruction. Simultaneously, the PC is incremented by 1 and will be stored in the IF/ID pipeline register, allowing the next instruction to be fetched in the following cycle while the current one proceeds to the decode stage. This stage is purely combinational, and its output is stabilized by the IF/ID register to be safely consumed in the next stage.

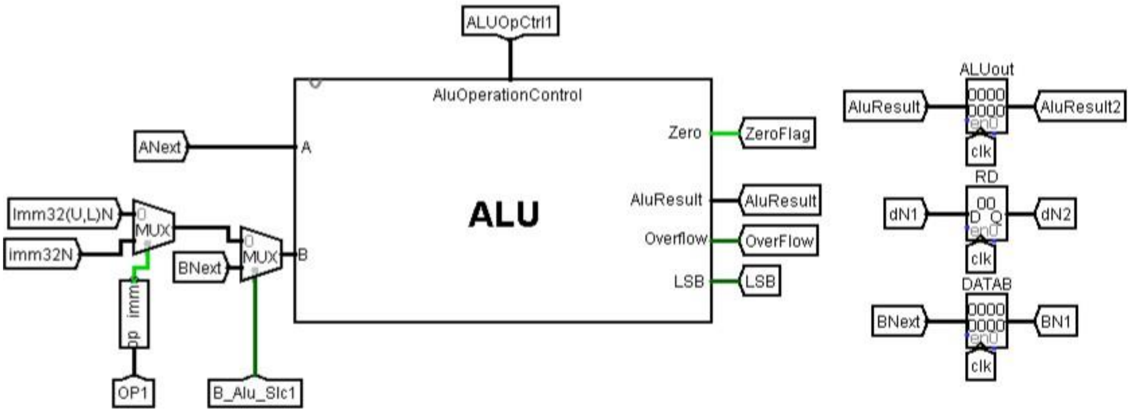


ID



In the Instruction Decode/Register Fetch (ID) stage, the instruction is broken down into its constituent fields—opcode, source and destination register identifiers, immediate values and function codes for R-type instructions. The register file is accessed to read the values of the source registers identified in the instruction. For instructions that utilize immediate values (e.g., I-type or SB-type instructions), the immediate field is extracted and sign-extended to 32 bits to match the architecture's word size. The control unit is also activated in this stage and generates all necessary control signals (such as RegWrite, MemRead, MemWrite, ALUSrc, Branch, etc.) based on the opcode. All of the decoded information, including the read register values (commonly denoted as A and B), the extended immediate values, the destination register identifiers, the incremented PC, and the generated control signals, are transferred to the next pipeline stage through the ID/EX register. This transfer ensures that the EX stage receives all the necessary inputs in the following cycle without having to re-access earlier stages.

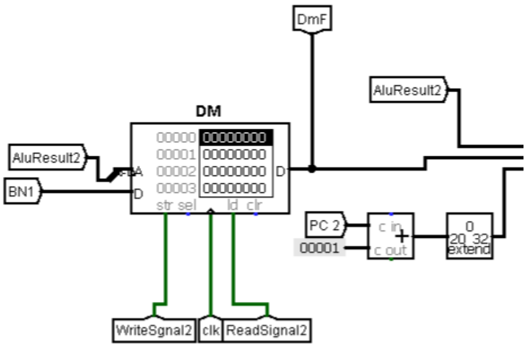
EX



Next, in the Execute (EX) stage, actual computation begins. The Arithmetic Logic Unit (ALU) receives its operands, which may be sourced either directly from the register file (A and B) or from a combination of one register value and an immediate operand, based on the control signal ALUSrc. The ALU Control Unit, which interprets signals from both the main control unit and the instruction's function field, determines the exact operation to perform (e.g., addition, subtraction, logical operations, etc.). The output of the ALU is the primary computational result of the instruction and is passed forward for further use. For certain instructions like store (sw), the second operand (register B value) is also carried forward, as it will be needed for writing into data memory in the next stage. The identity of the destination register (either Rd or Rt depending on the instruction type and RegDst signal) is also forwarded. All of this information—the ALU result, operand B for store instructions, and the target register—is packaged into the EX/MEM pipeline register, along with the relevant control signals such as MemRead, MemWrite, and RegWrite.

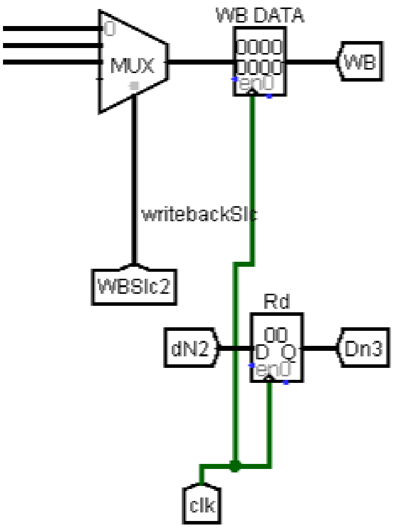
MEM

The Memory Access (MEM) stage interacts with data memory. If the instruction is a load (lw), the memory address (produced by the ALU in the EX stage) is used to fetch a word from memory, which is stored in the MEM/WB register for writing back to the register file. If the instruction is a store (sw), the value from operand B is written to the address computed by the ALU. For R-type instructions, which do not involve memory, the ALU result is simply forwarded unchanged. The control signals are again carried forward to guide the final write-back process. This stage is especially sensitive to memory-related hazards, and its correct execution



depends on prior data being available and no conflicts occurring from instructions ahead or behind in the pipeline.

Finally, the Write-Back (WB) stage is responsible for updating the register file. Based on the MemtoReg control signal, the value to be written is selected either from the memory (for lw instructions) or directly from the ALU output (for R-type or immediate arithmetic instructions). This value is written into the destination register whose address has been passed along through the pipeline stages. This marks the end of the instruction's lifecycle in the pipeline.



Throughout this entire process, control signals—generated during the ID stage—are carefully passed along with the data through each pipeline register to ensure consistent and correct instruction behavior. Without these signals, the subsequent stages would lack the context needed to perform their operations. This separation and passing of control and data across stages not only improves modularity and clarity but also enables instruction-level parallelism, allowing the processor to work on multiple instructions at different stages of execution simultaneously.

The pipelined design, while highly efficient, introduces potential hazards—namely structural hazards (resource conflicts), data hazards (when instructions depend on the results of prior instructions), and control hazards (due to branches). Though this detailed data flow overview assumes an ideal hazard-free environment, actual implementation requires mechanisms such as data forwarding, stalling, pipeline flushing, and bubble insertion to handle these challenges. These mechanisms, which are integral to ensuring correct execution without sacrificing performance, will be explored in depth in subsequent sections.

3. data Hazard Forwarding

In our **Risc pipelined processor**, instructions are executed in a pipelined fashion, meaning multiple instructions are overlapped in execution to improve performance. However, this overlap can introduce **hazards**, which are situations that prevent the next instruction from executing during its designated clock cycle. Two important mechanisms to deal with these hazards are hazard detection and forwarding (also known as bypassing).

3.1. Hazards in Risc

There are three main types of hazards:

1. **Data Hazards:**
   - Occur when an instruction produces a result that is required by a subsequent instruction.
   - The dependent instruction must wait until the preceding instruction completes its data read or write operation.
   - These hazards arise from data dependencies between instructions in the pipeline.
2. **Control Hazards:**
   - Arise from instructions that alter the program's control flow, such as branches and jumps.
   - The decision to change control flow often depends on the outcome of a preceding instruction.
   - These hazards introduce delays in determining the correct next instruction to execute.
3. **Structural Hazards:**
   - Occur when multiple instructions compete for the same hardware resource during the same clock cycle.
   - These hazards result from resource contention due to insufficient hardware to support all concurrent operations in the pipeline.

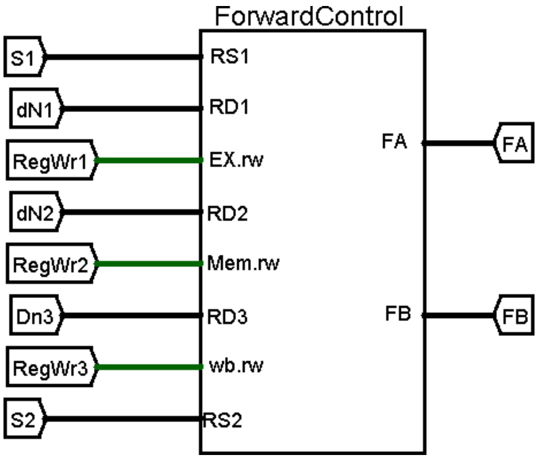
Hazard detection and forwarding specifically deal with **data hazards**.

3.2. Forwarding (Bypassing) Logic

Forwarding logic (also known as bypassing) is a key component in a pipelined Risc processor used to resolve **data hazards**—specifically **Read After Write (RAW)** hazards—without stalling the pipeline. It allows data to be forwarded (bypassed) from later stages of the pipeline to earlier ones, so dependent instructions can proceed without waiting for the register write-back to complete.

3.2.1. Purpose:

To avoid stalling the pipeline when a value needed by an instruction is still in the pipeline but has already been computed (in a later stage).



3.2.2. How It Works:

Forwarding logic detects when an instruction in the EX-stageneeds a value that is currently in the ALU or MEM or WB stages and redirects ("forwards") that value to the EX-stage without waiting for it to be written back to the register file.

3.2.3. Hardware Design:

Here's a simplified breakdown of the forwarding unit:

- **Inputs:**
    - Source register IDs from the ID-stage: RS1, RS2.
    - Destination register IDs from EX and MEM and WB stages: **EX.RD**(RD1), **MEM.RD**(RD2) and **WB.RD3**(RD3).
    - Write-back enable signals: EX.RegWr, MEM.RegWr, WB.RegWr.

- **Logic**:
    - If ((RS1 != 0) and (RS1 == RD1) and (EX.RegWr)) ForwardA = 1
    - Else if ((RS1 != 0) and (RS1 == RD2) and (MEM.RegWr)) ForwardA = 2
    - Else if ((RS1 != 0) and (RS1 == RD3) and (WB.RegWr)) ForwardA = 3
    - Else ForwardA = 0

    - If ((RS2 != 0) and (RS2 == RD1) and (EX.RegWr)) ForwardB = 1
    - Else if ((RS2 != 0) and (RS2 == RD2) and (MEM.RegWr)) ForwardB = 2
    - Else if ((RS2 != 0) and (RS2 == RD3) and (WB.RegWr)) ForwardB = 3
    - Else ForwardB= 0

- **Multiplexer Design**

  To implement forwarding, use 2 multiplexers in the ID stage:

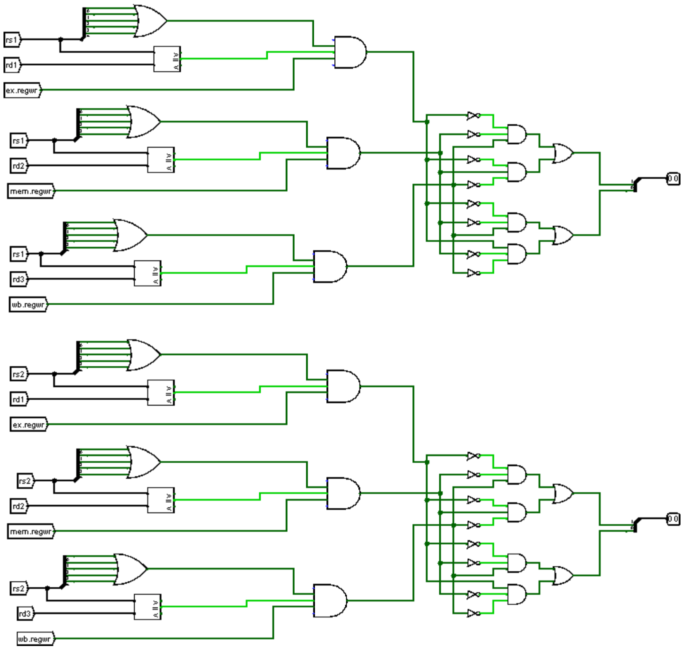
-two before ID/EX Register: one for Bus A and the other for Bus B.

-Select between:

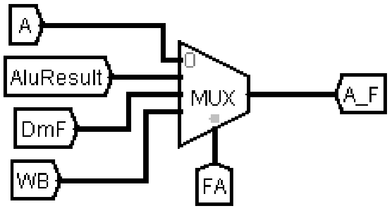
- Register file output (normal path)
- ALU result from EX
- Data from MEM
- Data from WB

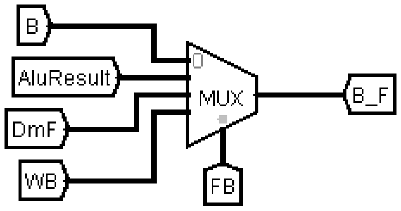
3.2.4. Hardware Block Diagram (Description)

- **Forwarding Unit:**
  - Takes in the `RegWrite` and `rd` from EX, MEM and WB stages.
  - Compare rd with `rs1`, `rs2` from ID stage.
  - Outputs `ForwardA` and `ForwardB` control signals.



- **Multiplexers:**
  - At ID (before ID/EX Register) at BusA and Bus B in ID stage
  - Controlled by `ForwardA` and `ForwardB`
  - Inputs:
    - 00: From register file (normal)
    - 01: From ALU result
    - 10: From MEM
    - 11: From WB





3.2.5. Limitations

Forwarding can't help in **Load-Use hazards**:
Forwarding cannot resolve load-use hazards because the load instruction does not produce its result until the MEM stage. Therefore, if a subsequent instruction depends on the loaded data and is positioned immediately after the load instruction, it must be stalled to prevent incorrect execution.

4. Load delay and stall

In pipelined processor architectures, one of the critical challenges to maintaining correct instruction execution is the handling of *data hazards*. A specific and frequent type of data hazard is the load-use hazard, which occurs when an instruction attempts to use data loaded from memory by a preceding **lw** (load word) instruction before the memory access has completed. Due to the nature of pipelining, this data is not yet available for forwarding at the time it is needed by the dependent instruction. To prevent the use of incorrect or uninitialized data, the processor must delay the execution of the dependent instruction. This is achieved by implementing a stalling mechanism.

In this project, we have implemented a dedicated Hazard Detection Unit that detects load-use hazards in the ID stage and inserts a one-cycle stall when necessary. The following sections detail the design, implementation, verification, and integration of this hazard mitigation mechanism.

4.1 Theoretical Background

A *load-use hazard* arises when the destination register of a load instruction (currently in the EX stage) matches either source operand register of the instruction in the ID stage. In such a case, the data will not be available in time for correct operation due to the memory latency, even with forwarding logic in place.

This scenario necessitates a stall to allow the lw instruction to complete the memory access and write-back before the dependent instruction proceeds. The stall ensures data correctness without introducing any structural or control hazards.

4.2 Design Objectives

The design of the hazard detection unit adheres to the following objectives:

1. **Accurate Detection**: Identify only the cases where a true data dependency exists between a load instruction and a subsequent instruction.
2. **Minimal Performance Impact**: Insert a stall only when it is absolutely necessary, i.e., for load-use dependencies.
3. **Pipeline Compatibility**: Seamlessly integrate with the pipeline control signals to freeze the IF/ID register and the Program Counter (PC) during the stall cycle.

4.3 Functional Description

Inputs:

- **ex_memread** *(1 bit)*: A control signal generated in the EX stage indicating that the current instruction is a memory read operation.

- **FA and FB** *(2 bits each)*: Register addresses of the source operands of the instruction in the ID stage.

Output:

- **Stall** *(1 bit)*: A control signal that, when asserted (1), causes a one-cycle stall in the pipeline by freezing the PC and the IF/ID pipeline register.

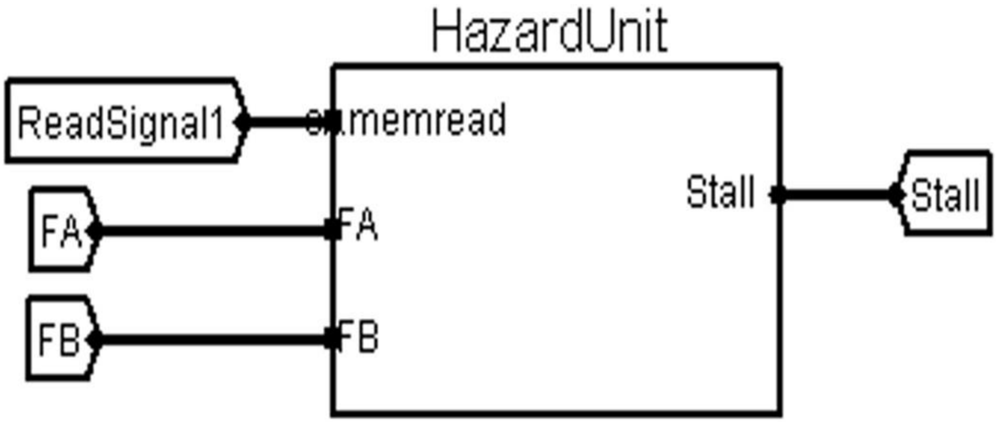


Figure 4.1

4.4 Hardware Implementation

The Hazard Detection Unit is implemented as a combinational logic block that compares register addresses using logic gates. The output is the Stall signal.

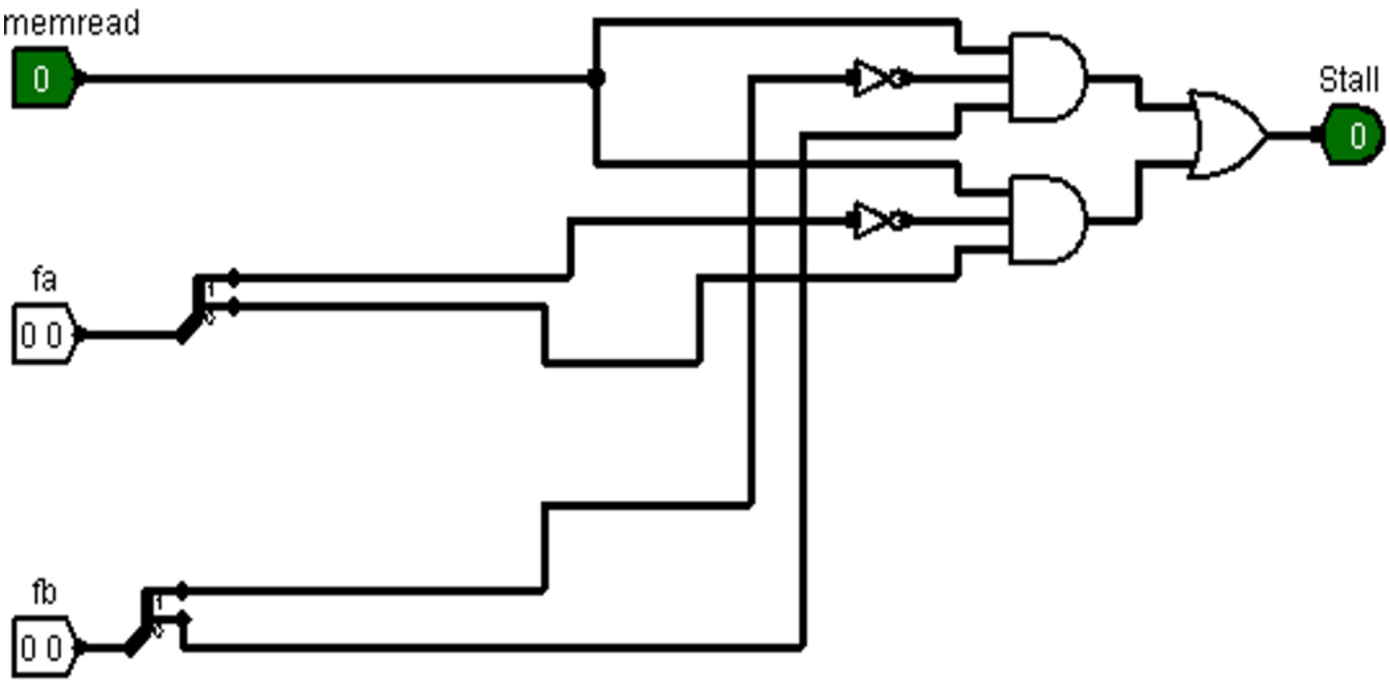


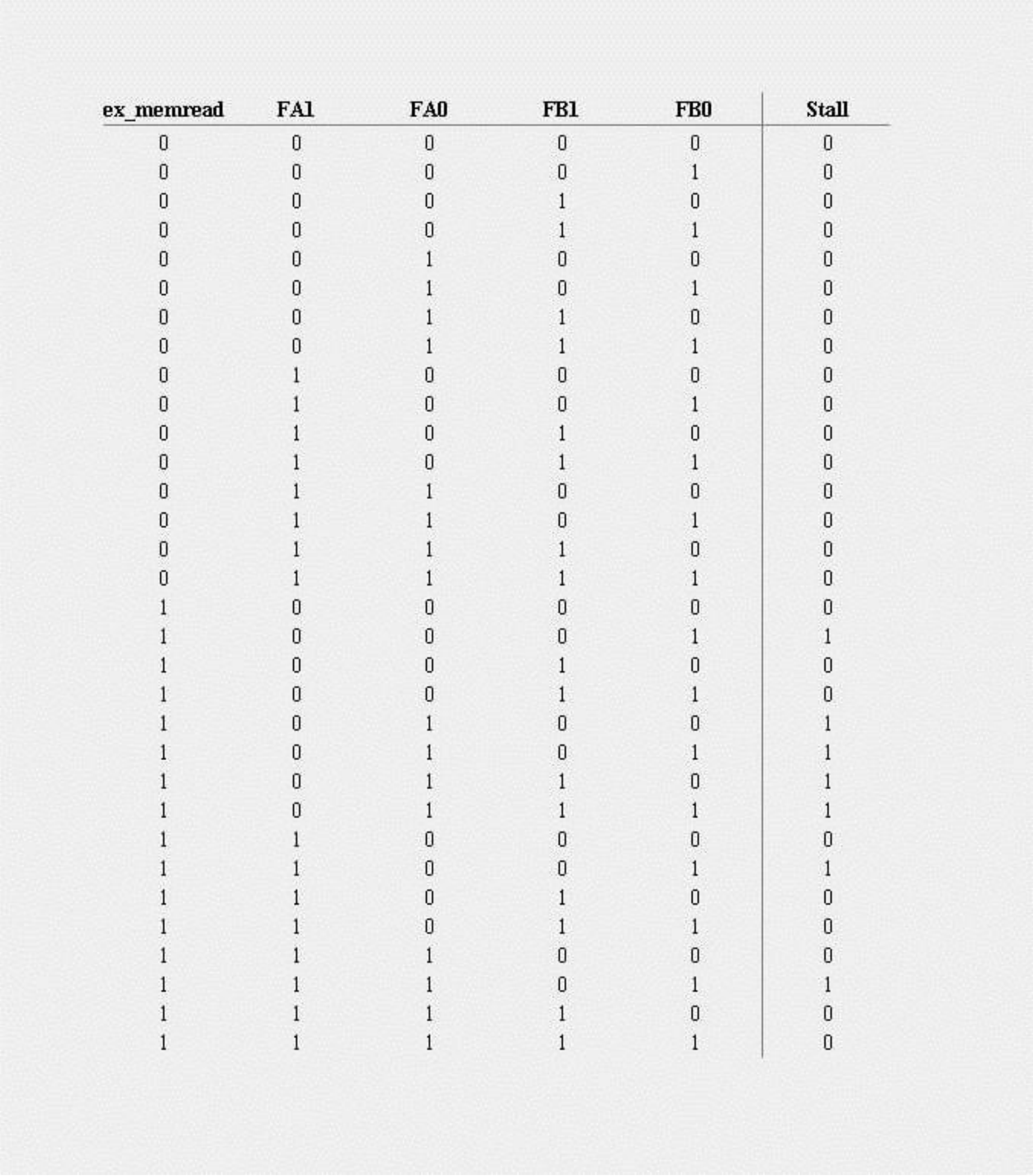
Figure 4.2

Table 4.1: Stall Signal Output for All Input Combinations

ex_memread	FA1	FA0	FB1	FB0	Stall
0	X	X	X	X	0
1	0	1	X	X	1
1	1	0	X	X	1
1	X	X	0	0	0
1	X	X	0	1	1
1	X	X	1	0	1
1	X	X	1	1	0
1	1	1	X	X	0

This table confirms that a stall is only triggered under the precise conditions described above, validating the logic against incorrect or unnecessary stalling.

A full image of the truth table is provided in Figure 4.2.



ex_memread	FA1	FA0	FB1	FB0	Stall
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	0	1	1
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	1
1	1	1	1	0	0
1	1	1	1	1	0

Figure 4.3

4.5 Pipeline Integration

The Stall signal is fed back into the Instruction Fetch (IF) stage to:

- Freeze the Program Counter (PC) from incrementing.
- Insert a NOP or bubble in the EX stage to delay execution.

This integration, where the Stall signal connects to PC enable, is shown in Figure 4.4, and insert a NOP is shown in Figure 4.5.

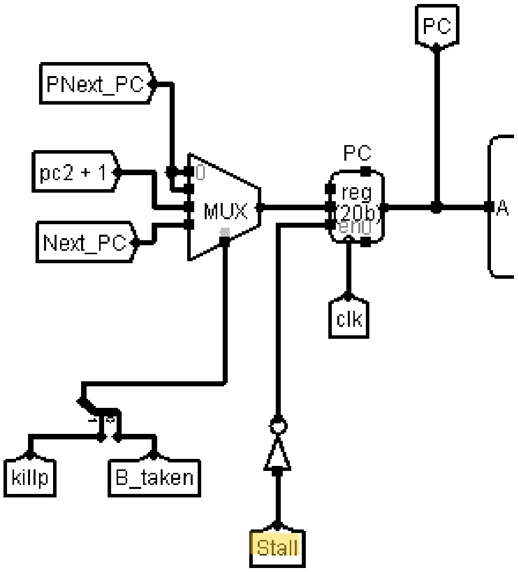


Figure 4.4

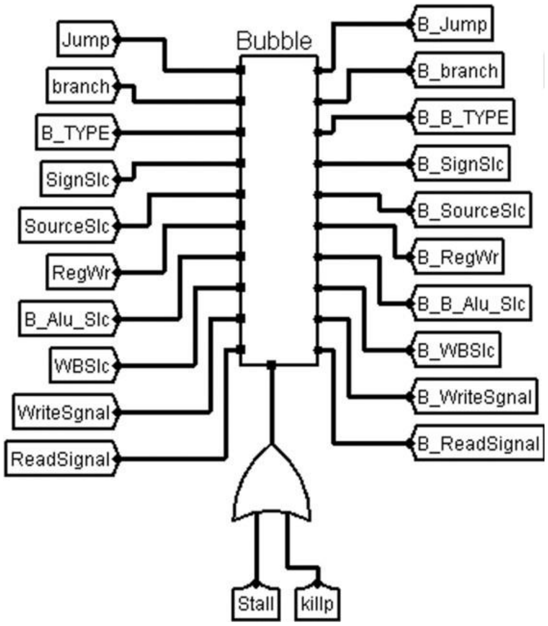


Figure 4.5

BUT HOW TO INSERT A BUBBLE?

To implement a bubble in our pipeline architecture, we introduced a controlled stalling mechanism is shown in Figure 4.6, that effectively inserts a NOP (No Operation) instruction into the pipeline when a load-use hazard is detected. This is achieved by first concatenating all relevant control signals from the Decode (ID) stage into a single composite signal bus. This bus is then passed through a 2-to-1 multiplexer, which selects between the normal signal values and a pre-defined zero vector based on the Stall control signal. When a hazard is detected and Stall is asserted, the multiplexer outputs the zero vector instead of the actual instruction signals. This vector propagates to the Execute (EX) stage, where it is interpreted as a NOP, effectively creating a bubble in the pipeline. The bubble allows the load instruction in the EX stage to complete without interference, thereby resolving the hazard. After the multiplexer, the signal bus is split back into individual signals, preserving the modular design of the pipeline. This method ensures correct execution flow while maintaining a clean and hardware-efficient implementation of hazard handling.

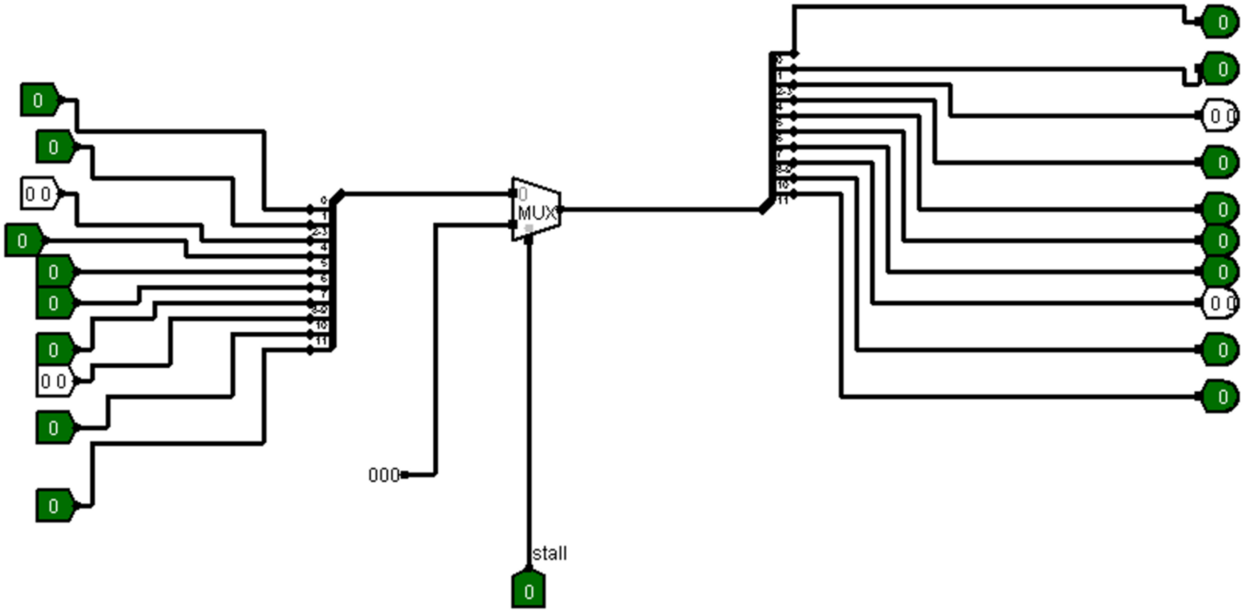


Figure 4.6

5. Control Hazards

In pipelined processor architectures, instructions are executed in overlapping stages to improve performance. However, this parallelism introduces various hazards, including control hazards, which occur when the flow of instruction fetch depends on the outcome of a branch or jump instruction. These hazards arise because the branch target address and decision are typically not known until a later stage in the pipeline.

Failure to handle control hazards correctly can result in incorrect instruction execution and corruption of program state. This chapter discusses the nature of control hazards, the associated performance implications, and the detailed implementation of control hazard resolution in our pipelined processor.

5.1 Nature of Control Hazards

In our 5-stage pipeline (IF → ID → EX → MEM → WB), the Program Counter (PC) is updated every cycle to fetch the next instruction. When a control transfer instruction—such as a conditional branch (BEQ, BNE, BLT, etc.) or an unconditional jump (JALR)—is executed, it may redirect the PC to a new target address. However, the decision to perform this redirection is not resolved until the EX stage, after operand comparison and branch condition evaluation.

By the time the EX stage resolves the control transfer:

- The next instruction has already been fetched (IF stage),
- And its control signals have already been decoded (ID stage).

If the control transfer is taken, the instructions in IF and ID must be discarded. This results in what is known as a control hazard or branch hazard.

5.2 Pipeline Impact and the Two-Cycle Penalty

Because our pipeline resolves branches and jumps in the EX stage till now and does not implement branch prediction or delay slots, it inherently suffers a two-instruction penalty when a control transfer is taken.

The sequence of events is as follows:

Cycle	IF	ID	EX	Outcome
t	BEQ			Branch fetched
t+1	NEXT1	BEQ		Instruction fetched normally
t+2	NEXT2	NEXT1	BEQ	Branch resolved (taken)
t+3	Target	(flushed)	(flushed)	Resume at correct target

- NEXT1 and NEXT2 are invalid instructions fetched along the wrong path.
- They must be killed (flushed) and replaced with NOPs.
- This results in a 2-cycle control hazard stall whenever a branch or jump is taken.

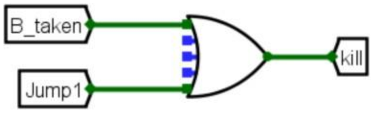
5.3 Flush Logic and Kill Signal Generation

To resolve control hazards, our processor implements a Kill-based flushing mechanism, driven by signals generated in the EX stage. The Kill signal is asserted when a control transfer is confirmed.

5.3.1 Kill Signal Definition

The Kill signal is computed as:

```
Kill = BranchTaken OR Jump
```



Where:

- BranchTaken is output from the Condition Control Unit in the EX stage,
- Jump is a control signal for unconditional jump instructions (JALR), also determined in EX.

This signal captures all control flow transfers that require flushing previous stages.

5.4 Implementation of Instruction Flushing

The Kill signal is used to flush both the IF and ID stages to prevent incorrectly fetched or decoded instructions from executing.

5.4.1 Instruction Register Flushing (IF Stage)

In the IF stage, a multiplexer is inserted between the output of the Instruction Memory (IM) and the Instruction Register (IR) as shown in Figure5.1.

- **MUX Inputs**:
  - Input 0: Normal instruction from Instruction Memory
  - Input 1: Zero constant
- **MUX Select**:
  - Driven by Kill
    → If Kill = 1, the zero constant is passed to IR
    → If Kill = 0, the normal instruction is passed

This ensures that the instruction fetched during a mispredicted control transfer is invalidated.

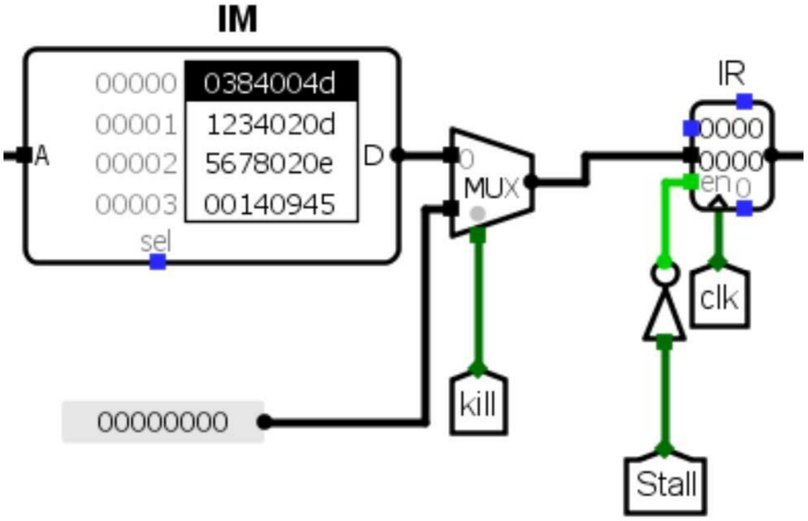


Figure5.1

5.4.2 Bubble Triggering and Instruction Invalidation

For the instruction in the decode stage, bubble insertion is triggered using the same Kill signal. Rather than duplicating bubble logic, the Kill signal is simply combined with the Stall signal (from the load hazard unit), and the result is passed to the bubble control system as shown in Figure5.2.

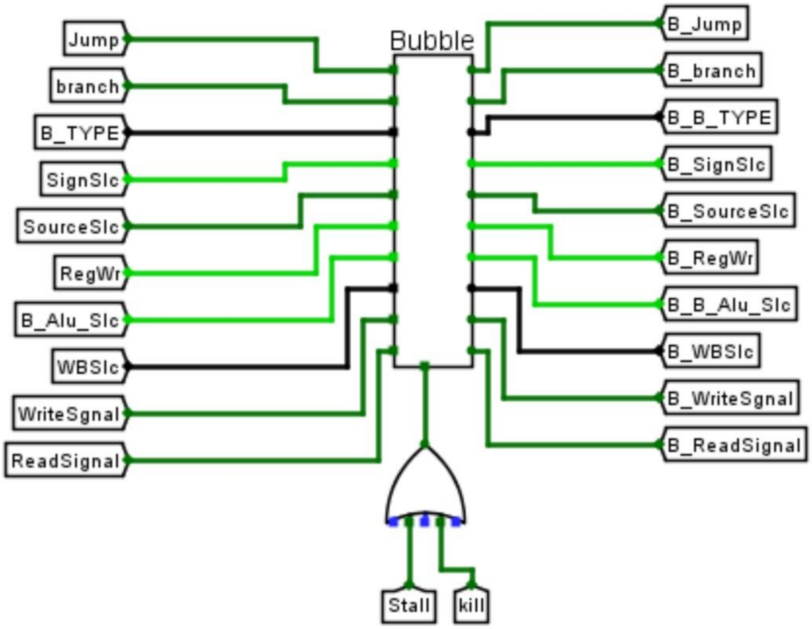


Figure5.2

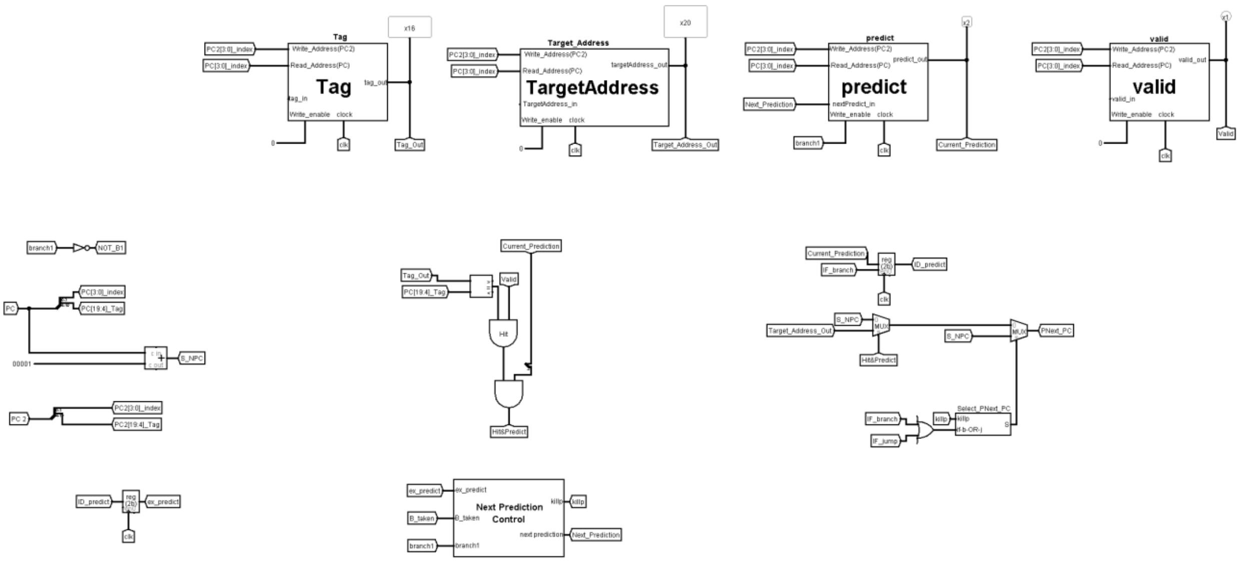
Note: The unified bubble insertion mechanism and its detailed implementation are fully described in Section 4.5. This system ensures that control and data hazards share a consistent and efficient flushing interface

6. 2 Bit Dynamic Branch Prediction in RISC Processor

6.1. Overview

In our pipelined Risc processor, control hazards caused by branch instructions can significantly degrade performance by introducing stalls. To mitigate this, branch prediction techniques are used to guess the outcome of a branch instruction before it is resolved. One effective method is the 2-bit dynamic branch prediction, which improves accuracy over simple static or 1-bit schemes by maintaining a 2-bit saturating counter for each branch instruction.

2-bit Dynamic Branch Predictor



6.2. Principle of Operation

The 2-bit predictor uses a small **finite state machine (FSM)** to predict whether a branch will be taken or not taken. Unlike 1-bit predictors, which change prediction with a single incorrect outcome, the 2-bit predictor requires two consecutive mispredictions to switch its state, thereby reducing the impact of occasional anomalies in branching behavior.

6.2.1 State Machine

The 2-bit predictor operates in four states:

State	Meaning	Prediction
00	Strongly Not Taken	Not Taken
01	Weakly Not Taken	Not Taken
10	Weakly Taken	Taken
11	Strongly Taken	Taken

Transition Rules:

- If the branch is taken, increment the counter (up to 11).
- If the branch is not taken, decrement the counter (down to 00).

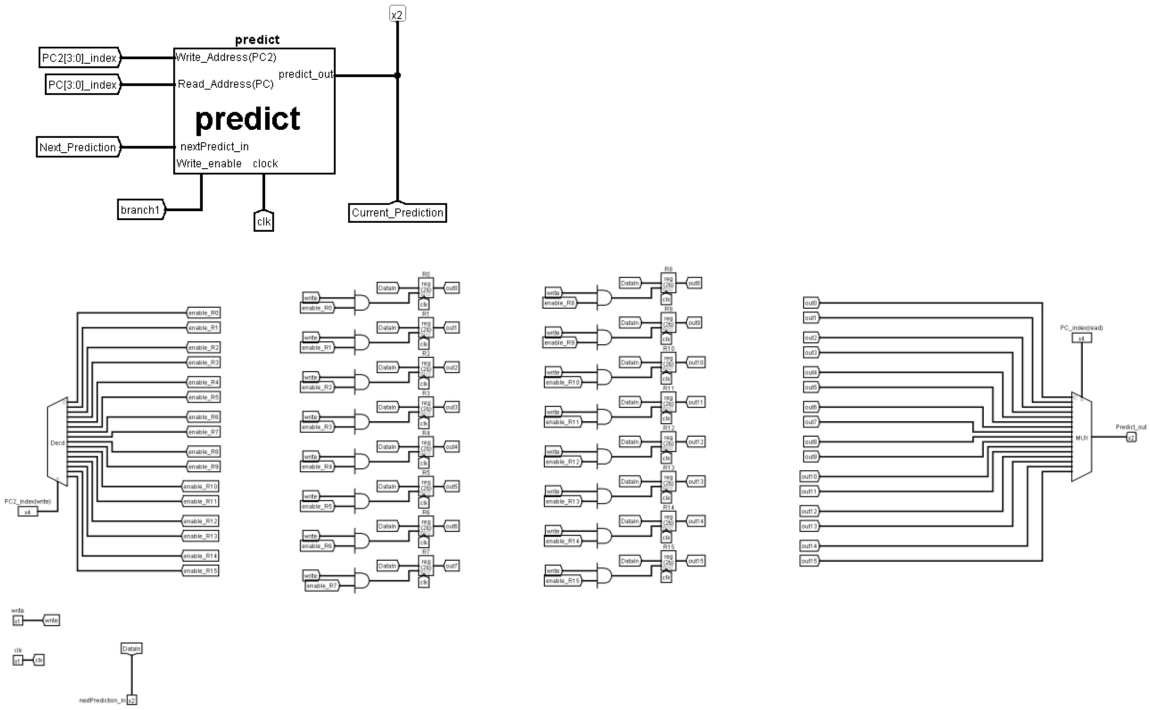
6.3. Hardware Design

6.3.1. Components

1. **Branch History Table (BHT)**:

   - A small memory array indexed by the lower bits (the first four bits) of the program counter (PC).

   - Each entry contains a 2-bit counter representing the prediction state.



(the above figure is the design of predict block)

2. **PC Indexing Logic**:

   - Extracts the relevant bits from the PC to access the BHT.

   - Typically, only the least significant bits are used to reduce hardware size.

3. **Prediction Logic**:

   - Reads the BHT entry at fetch stage.

   - If the state is 10 or 11, predict "taken"; otherwise, predict "not taken".

4. **Update Logic**:

   - At the branch EX stage, the actual outcome is known.

   - The corresponding BHT entry is updated based on the outcome, modifying the 2-bit counter.

6.3.2. Timing and Placement

- **Fetch Stage (IF)**: BHT is read using PC to make the prediction.

- **Execution Stage (EX)**: Actual branch decision is made and BHT is updated with the outcome.

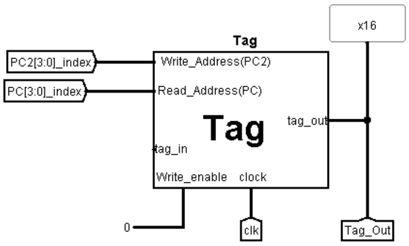
6.4. How does it work

FIRST:

We Create BTB Storage by Using four blocks of registers modules:

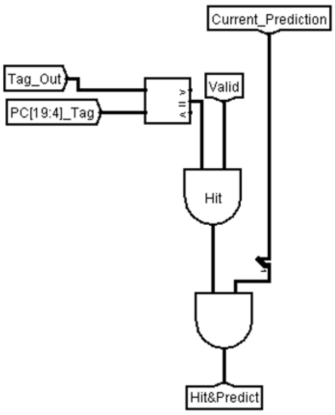
- Tag (16 bit)
- Target Address (20 bit)
- Predict (2 bit)
- Valid (1 bit)

All 4 blocks of registers use the same address: PC/PC2 [3:0].



SECOND:

In IF Stage – Predict Branch:



1. **Extract Index and Tag from PC:**
   - PC_index = PC[3:0] → block address
   - PC_tag = PC[19:4] → Compare with stored tag
2. **Read from blocks using PC_index:**
   - stored_tag from Tag block
   - stored_target from Target block
   - stored_counter from Counter block
   - valid_bit from Valid block
3. **Check for BTB Hit:**
   - BTB_Hit = (stored_tag == PC_tag) AND (valid_bit == 1)
4. **Make Prediction:**
   - predicted_taken = stored_counter[1] (MSB)
   - If BTB_Hit AND predicted_taken == 1 : → Set next_PC = stored_target

     Else: → Set next_PC = PC + 1
5. **Pass info to later stages (save PC, stored_counter)**
   - PC (IF stage) **-> IF/ID pipeline register ->** PC1 (ID stage) **-> ID/EX pipeline register ->** PC2 (EX stage)
   - Predict (IF stage) **-> IF/ID register ->** ID_predict (ID stage) **-> ID/EX register ->** EX_predict (EX stage)

THIRD:

In EX Stage – Resolve and Update:

1. **Inputs needed in EX stage:**
   - EX_PC (PC2): original PC of branch instruction
   - EX_taken (B_taken): actual branch outcome (1 or 0)
   - EX_target: actual branch target address
   - EX_is_branch (branch 1): control signal
   - EX_predict: the stored prediction
2. **Extract again:**
   - PC2_index = EX_PC2[3:0]
   - PC2_tag = EX_PC2[19:4]
3. **Write to block at PC2_index:**
   - Predict block -> write next prediction:

   -Read current value (stored_counter)

   -If EX_taken == 1: increment (up to 11)

   -If not: decrement (down to 00)

   - Write updated value at PC2_index according to Counter Update Logic:

     To update the 2-bit predictor:

     ○ From EX stage:

     -get stored_counter from the second

      Predict register(ID/EX register)

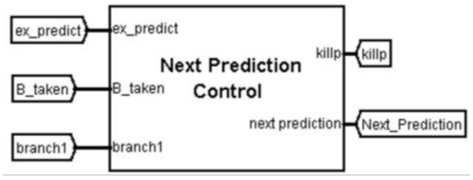
     -get EX_is_branch(branch1)

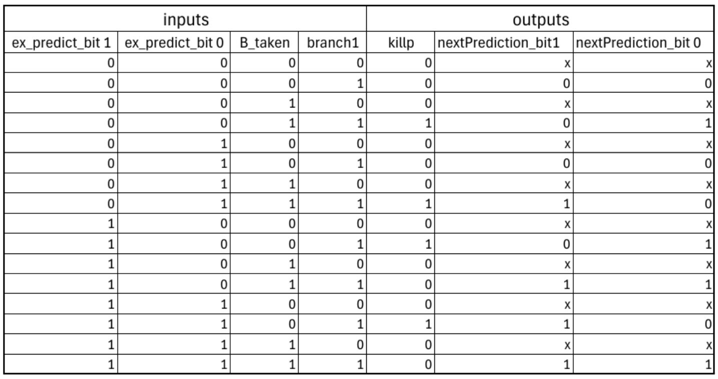
     -get B_taken

     And according to the next truth table:

     -we get killp

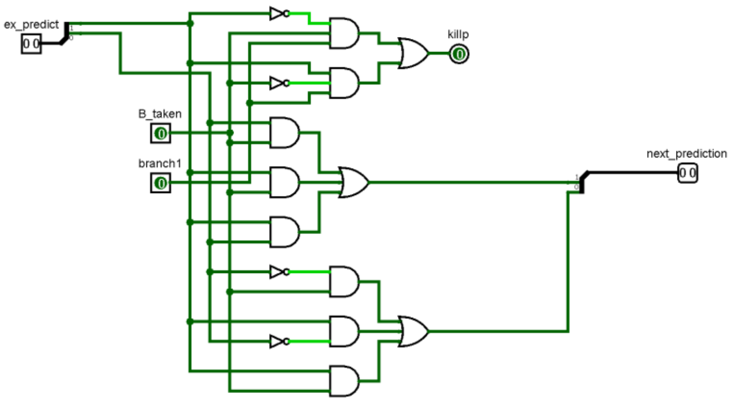
     -we get nextPrediction





inputs				outputs		
ex_predict_bit 1	ex_predict_bit 0	B_taken	branch1	killp	nextPrediction_bit1	nextPrediction_bit 0
0	0	0	0	0	x	x
0	0	0	1	0	0	0
0	0	1	0	0	x	x
0	0	1	1	1	0	1
0	1	0	0	0	x	x
0	1	0	1	0	0	0
0	1	1	0	0	x	x
0	1	1	1	1	1	0
1	0	0	0	0	x	x
1	0	0	1	1	0	1
1	0	1	0	0	x	x
1	0	1	1	0	1	1
1	1	0	0	0	x	x
1	1	0	1	1	1	0
1	1	1	0	0	x	x
1	1	1	1	0	1	1

(the above table is the Truth Table of next prediction control)



(The combinational circuit of next prediction control)

4. **Write Enable for predict block = EX_is_branch (branch1)**
   Branch1 = write enable for the predict block

FOURTH:

EX Stage - Misprediction Handling:

1. **From stored data in EX stage:**
   - predicted_taken(ex_predict) = stored_counter
   - mispredicted = EX_taken(B_taken) XOR predicted_taken(ex_predict)
2. **If mispredicted == 1:**
   - Kill the instructions that were incorrectly fetched :Convert Next1(that is IF stage)and Next2 (ID stage) into bubbles
   - Correct the PC:

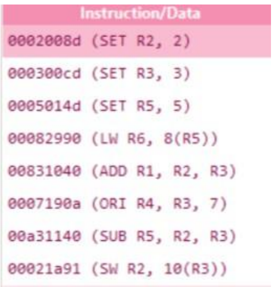
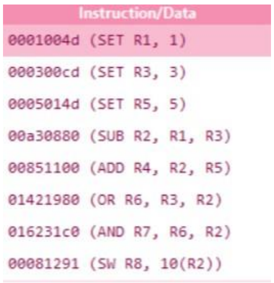
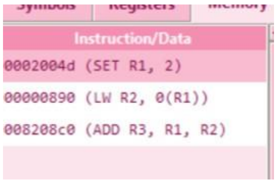
   If EX_taken == 1: set PC = EX_target

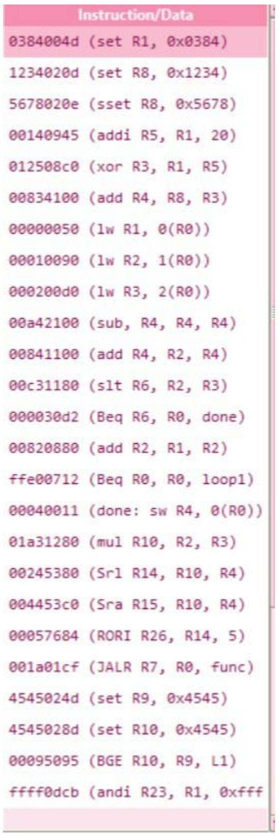
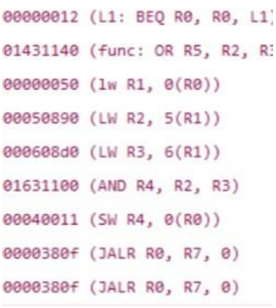
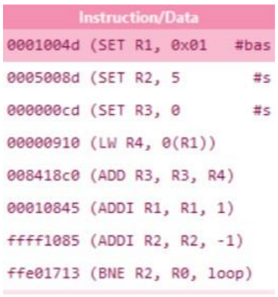
   Else: set PC = EX_PC + 1

7.Testing and verification

Now that we have completed the processor in all aspects, it's time to begin testing. We will conduct several tests, including general pipeline functionality, data hazards and forwarding, load delays, control hazards, as well as the *testCode* task and array sum operations.

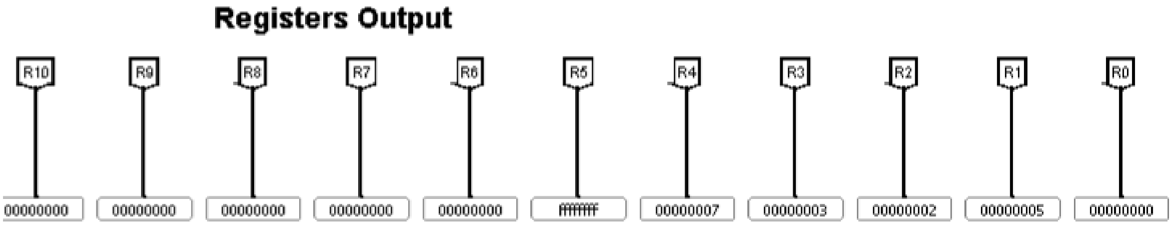
Program Num	Test Subject	Program	Result

1	General Pipeline	 0002008d (SET R2, 2) 000300cd (SET R3, 3) 0005014d (SET R5, 5) 00082990 (LW R6, 8(R5)) 00831040 (ADD R1, R2, R3) 0007190a (ORI R4, R3, 7) 00a31140 (SUB R5, R2, R3) 00021a91 (SW R2, 10(R3))	Works Well
2	Data hazards	 0001004d (SET R1, 1) 000300cd (SET R3, 3) 0005014d (SET R5, 5) 00a30880 (SUB R2, R1, R3) 00851100 (ADD R4, R2, R5) 01421980 (OR R6, R3, R2) 016231c0 (AND R7, R6, R2) 00081291 (SW R8, 10(R2))	Works Well
3	load delay	 0002004d (SET R1, 2) 00000890 (LW R2, 0(R1)) 008208c0 (ADD R3, R1, R2)	Works Well

4	testCode	 Instruction/Data 0384004d (set R1, 0x0384) 1234020d (set R8, 0x1234) 5678020e (sset R8, 0x5678) 00140945 (addi R5, R1, 20) 012508c0 (xor R3, R1, R5) 00834100 (add R4, R8, R3) 00000050 (lw R1, 0(R0)) 00010090 (lw R2, 1(R0)) 000200d0 (lw R3, 2(R0)) 00a42100 (sub, R4, R4, R4) 00841100 (add R4, R2, R4) 00c31180 (slt R6, R2, R3) 000030d2 (Beq R6, R0, done) 00820880 (add R2, R1, R2) ffe00712 (Beq R0, R0, loop1) 00040011 (done: sw R4, 0(R0)) 01a31280 (mul R10, R2, R3) 00245380 (Srl R14, R10, R4) 004453c0 (Sra R15, R10, R4) 00057684 (RORI R26, R14, 5) 001a01cf (JALR R7, R0, func) 4545024d (set R9, 0x4545) 4545028d (set R10, 0x4545) 00095095 (BGE R10, R9, L1) ffff0dcb (andi R23, R1, 0xfff  00000012 (L1: BEQ R0, R0, L1 01431140 (func: OR R5, R2, R 00000050 (lw R1, 0(R0)) 00050890 (LW R2, 5(R1)) 000608d0 (LW R3, 6(R1)) 01631100 (AND R4, R2, R3) 00040011 (SW R4, 0(R0)) 0000380f (JALR R0, R7, 0) 0000380f (JALR R0, R7, 0)	Works Well
5	array sum	 Instruction/Data 0001004d (SET R1, 0x01 #bas 0005008d (SET R2, 5 #s 000000cd (SET R3, 0 #s 00000910 (LW R4, 0(R1)) 008418c0 (ADD R3, R3, R4) 00010845 (ADDI R1, R1, 1) ffff1085 (ADDI R2, R2, -1) ffe01713 (BNE R2, R0, loop)	Works Well

Processor Outputs

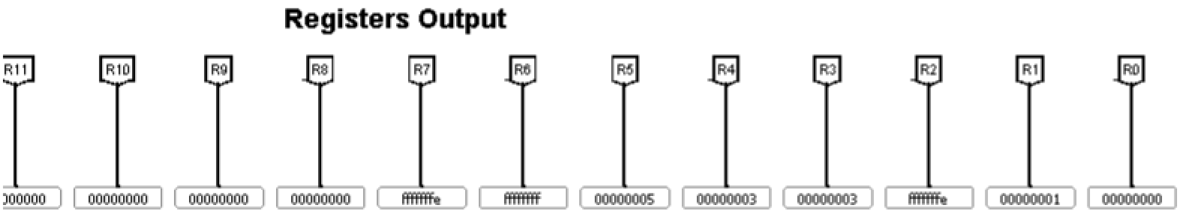
Program1



Registers Output

R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
00000000	00000000	00000000	00000000	00000000	ffffffff	00000007	00000003	00000002	00000005	00000000

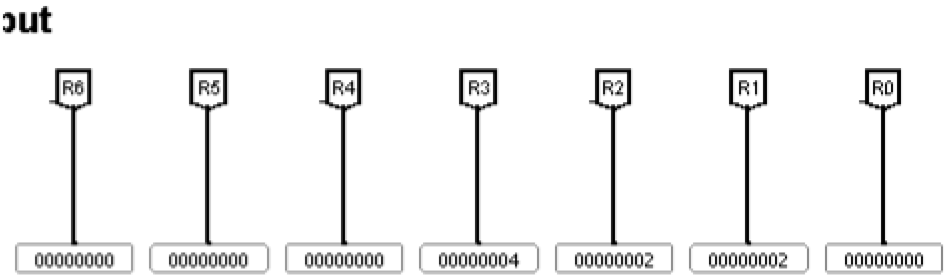
Program2



Registers Output

R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
000000	00000000	00000000	00000000	fffffffe	ffffffff	00000005	00000003	00000003	fffffffe	00000001	00000000

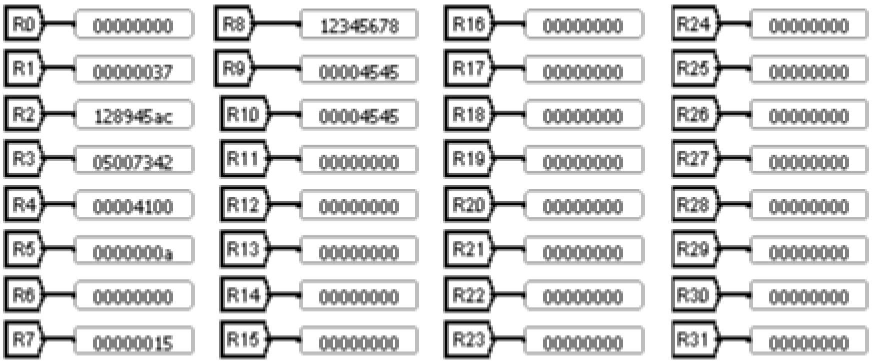
Program3



ut

R6	R5	R4	R3	R2	R1	R0
00000000	00000000	00000000	00000004	00000002	00000002	00000000

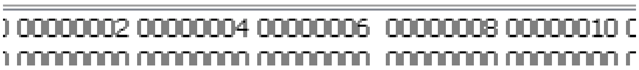
Program4



R0	00000000	R8	12345678	R16	00000000	R24	00000000
R1	00000037	R9	00004545	R17	00000000	R25	00000000
R2	128945ac	R10	00004545	R18	00000000	R26	00000000
R3	05007342	R11	00000000	R19	00000000	R27	00000000
R4	00004100	R12	00000000	R20	00000000	R28	00000000
R5	0000000a	R13	00000000	R21	00000000	R29	00000000
R6	00000000	R14	00000000	R22	00000000	R30	00000000
R7	00000015	R15	00000000	R23	00000000	R31	00000000

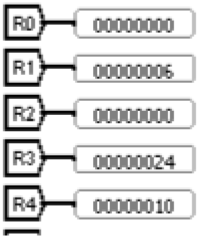
Program5

Data memory



```
) 00000002 00000004 00000006  00000008 00000010 C
```

Processor Output



```
R0 00000000
R1 00000006
R2 00000000
R3 00000024
R4 00000010
```

6. Workflow

6.1 Meetings

Day	Period	Place	What We Have Done
Thursday 1/5/2025	10AM: 9PM	Offline	Phase2: - Pipeline Design - Data Hazards - Load Delay - Control Hazard - Try 2 Bit Branch Predictor
Sunday 4/5/2025	3PM : 9PM	Offline	Try 2 Bit Branch Predictor in more details
Monday 5/5/2025	5PM : 9PM	Offline	Complete 2 Bit Branch Predictor

6.2 Contribution

Omnia Ayman	Alaa Ayman	Rahma Mostafa
- Pipeline design - Documentation - Video	- Pipeline design - Documentation - Video	- Pipeline design - Documentation - Video