INTERNATIONAL UNIVERSITY OF SARAJEVO

Faculty of Engineering and Natural Sciences

Department of Software Engineering

CS304 Computer Architecture

5-STAGE PIPELINED CPU WITH HAZARD HANDLING

By Lejla Goralija,
Ali-Harun Neimarlija,
and Imran Mujkanović

Instructor: Dr. Ali Abd Almisreb

Sarajevo,
June, 2025.

Table of Contents

1. Introduction

This report details the design, simulation, and interactive visualization of a **5-stage pipelined Central Processing Unit** (CPU) that incorporates mechanisms to handle common pipeline hazards. **Pipelining** is a technique used in modern processors to improve performance by overlapping the execution of multiple instructions. Instead of processing each instruction sequentially through all stages, multiple instructions are in different stages of execution simultaneously. This approach can significantly increase the instruction throughput compared to a single-cycle design.

However, this overlap introduces challenges known as pipeline **hazards**, which can impede the smooth flow of instructions. These hazards include structural hazards (when multiple instructions need the same resource at the same time), data hazards (when an instruction depends on the result of a previous instruction that is not yet available), and control hazards (arising from branch instructions where the next instruction to be fetched is uncertain).

To address these issues, our simulated CPU implements various hazard handling techniques. For data hazards, we explore data forwarding (bypassing) and pipeline stalling. For control hazards arising from branch instructions, different branch prediction strategies, such as "Always Not Taken", "Always Taken", and more sophisticated 1-bit and 2-bit predictors, are implemented and can be toggled.

To facilitate a deeper understanding of these concepts, a comprehensive simulation and visualization tool has been developed using HTML and JavaScript. This web-based interface provides an animated view of the 5-stage pipeline (Instruction Fetch, Instruction Decode, Execute, Memory, Write Back), allowing users to observe instructions moving through the stages, identify when hazards occur, and see how the implemented hazard handling mechanisms resolve them. The software also visualizes the CPU's register file, memory, and the contents of the pipeline registers at each clock cycle. Users can step through the execution, observe the effects of enabling or disabling forwarding, and experiment with different branch prediction strategies. This interactive approach aims to provide an intuitive and engaging way to learn about the complexities of pipelined CPU design and hazard management. In this report we will have a deep dive into the architecture, hazard handling implementations, the software's functionality, and the insights gained from the simulation.

2. Understanding Pipelining and Performance

2.1. What is Pipelining?

Pipelining is a fundamental technique in computer architecture that allows for the overlapping execution of multiple instructions. In a car factory this would be done by adding a new part at each step in the production process, first the engine, then the transmission, electronics, door, paint etc. This allows for a much more time and cost-effective production of cars, instead of producing one start to finish. Similarly, in a pipelined CPU, the execution of an instruction is broken down into a series of sequential stages, and different instructions can be in different stages of their execution concurrently.

In our 5-stage pipeline, an instruction progresses through the following stages:

1. **Instruction Fetch (IF):** The instruction is fetched from memory.
2. **Instruction Decode (ID):** The instruction is decoded, and the required operands are read from the register file.
3. **Execute (EX):** Arithmetic and logical operations are performed, or the effective address for memory access is calculated.
4. **Memory Access (MEM):** Data is read from or written to memory.
5. **Write Back (WB):** The result of the operation is written back to a register.

Without pipelining, a CPU would execute each instruction completely before starting the next one. This is known as a single-cycle design, though it is important to note that even single-cycle designs involve internal steps, but nothing compared to pipelining. Pipelining allows us to start fetching the next instruction while the current one is still in the decode stage, and so on. Ideally, once the pipeline is "full," one instruction completes execution in each clock cycle.

Consider a non-pipelined execution. If each stage took, say, one clock cycle, then executing n instructions would take $n \times 5$ clock cycles. However, with a perfectly pipelined system, after the initial filling of the pipeline which takes a few cycles, each subsequent instruction can complete in one clock cycle. Thus, executing n instructions might ideally take closer to $5+(n-1)$ clock cycles. For a large number of instructions, this approaches n cycles, a significant improvement over the non-pipelined case.

The benefit of pipelining lies in its potential to increase the instruction throughput, which is the number of instructions completed per unit of time. While the latency, the time it takes for a single instruction to complete, might not necessarily decrease and can sometimes slightly increase due to pipeline overhead, the rate at which instructions finish increases dramatically.

2.2. Performance Gains: Pipelined vs. Single-Cycle CPU Designs

The performance advantage of a pipelined CPU over a single-cycle CPU can be significant. Let's analyze this more formally.

Assume each stage in our 5-stage pipeline takes approximately the same amount of time, say T. In a non-pipelined, single-cycle design where we conceptually complete an instruction in one long cycle, the cycle time would need to be long enough to accommodate the slowest stage, plus any overhead for inter-stage communication. If we simplify and assume the single-cycle CPU also breaks down its work into similar fundamental operations as the pipeline stages, then its cycle time would be roughly 5T. To execute n instructions, a single-cycle CPU would take approximately $n \times 5T$ time.

Now consider the 5-stage pipeline. Once the pipeline is full, one instruction completes every T time units (one clock cycle of duration T). To execute n instructions, the time taken would be approximately the time to fill the pipeline (5 cycles) plus the time for the remaining $n-1$ instructions to complete, so roughly $(5+n-1) \times T = (n+4) \times T$.

The speedup of a k-stage pipeline over a non-pipelined execution can be theoretically close to k. In our 5-stage case, the ideal speedup would be around 5. The actual speedup is often less due to factors like pipeline hazards and unequal stage lengths.

Let's quantify this with an example. Suppose each stage takes 1 ns.

- **Single-Cycle CPU:** Executes one instruction in approximately 5 ns. To execute 100 instructions would take 100×5 ns = 500 ns. The throughput is 1/5 instructions/ns = 0.2 instructions/ns.
- **5-Stage Pipelined CPU:** Ideally, after the initial 5 ns to fill the pipeline, each subsequent instruction completes every 1 ns. To execute 100 instructions would take approximately $(5+100-1) \times 1$ns = 104ns. The throughput is approximately 1/1 instruction/ns = 1 instruction/ns (once the pipeline is full).

The speedup in this idealized scenario for 100 instructions is $500/104 \approx 4.8$.

However, the presence of hazards complicates this picture. When a hazard occurs, the pipeline might need to stall, reducing the throughput. For instance, if a data hazard requires a one-cycle stall, then that clock cycle does not result in a completed instruction, impacting the overall performance. Similarly, branch mispredictions can lead to flushing the pipeline, effectively wasting the work done on incorrectly fetched instructions.

Despite these drawbacks, pipelining remains a crucial technique for achieving high performance in modern CPUs. The ability to overlap instruction execution significantly increases the rate at which instructions are completed, leading to substantial performance gains over single-cycle designs. Our simulation aims to illustrate these performance benefits and the challenges posed by hazards, as well as the effectiveness of hazard handling techniques in mitigating these performance losses.

3. The 5-Stage Pipelined CPU Architecture

The concept of a 5-stage pipeline revolves around the idea of taking a complex task, executing an instruction, and breaking it down into a sequence of simpler sub-tasks, each performed in a dedicated stage. This division allows for parallelism: while one instruction is being processed in one stage, another instruction can simultaneously be processed in a different stage. This divide and conquer is what leads to the potential for increased throughput.

In our 5-stage CPU, each instruction ideally moves through these stages in a sequence: IF → ID → EX → MEM → WB. At any given clock cycle, similarly to conveyor belt, up to five different instructions could be in the pipeline, each at a different stage of completion.

The resources that are used in each stage:

- **Instruction Fetch (IF):** Primarily uses the instruction memory and the Program Counter (PC). It needs to fetch the instruction located at the address pointed to by the PC. After fetching, the PC is typically updated to point to the next sequential instruction.
- **Instruction Decode (ID):** Requires access to the register file to read the operands specified in the instruction. It also involves decoding the instruction to determine the operation and generating control signals for the subsequent stages.
- **Execute (EX):** Utilizes the Arithmetic Logic Unit (ALU) for performing arithmetic and logical operations, as well as for address calculations.
- **Memory Access (MEM):** Interacts with the data memory to either read data (for load instructions) or write data (for store instructions).
- **Write Back (WB):** Needs to write the result back to the register file.

The beauty of pipelining is that, in an ideal scenario, each stage can operate independently or with minimal interference, on different instructions. For instance, while the ALU is performing an operation for one instruction in the EX stage, the instruction memory can fetch the next instruction for the IF stage.

However, this ideal scenario is often disrupted by pipeline hazards. These hazards arise when the smooth flow of instructions through the pipeline is stalled or needs to be altered. For example, if an instruction in the EX stage needs a value that is still being computed by a previous instruction in the EX stage (a data hazard), the subsequent instruction might need to wait.

The performance gain from pipelining comes from the fact that we can potentially complete one instruction per clock cycle once the pipeline is full. If each stage takes approximately one clock cycle, a non-pipelined processor would take roughly five clock cycles per instruction. Therefore, a 5-stage pipeline has the potential to achieve a speedup of up to 5 times compared to a non-pipelined processor with the same stage latencies.

Our simulation will visually demonstrate this flow, and how hazards can impact this smooth operation. It will also showcase the techniques used to mitigate these hazards and maintain performance.

3.1 Instruction Fetch (IF)

The Instruction Fetch (IF) stage is the first phase of the instruction execution pipeline. Its primary responsibility is to retrieve the instruction that is to be executed next from the main memory. This process is guided by the Program Counter (PC), which holds the memory address of the next instruction.

The IF stage can be further broken down:

1. **Address Generation:** The current value of the PC is used to generate the address of the instruction to be fetched.
2. **Instruction Memory Access:** This address is then sent to the instruction memory. The instruction memory, which stores the program's instructions, retrieves the instruction located at that address.
3. **Instruction Transfer:** The fetched instruction is then passed from the instruction memory to the next stage in the pipeline, which is the Instruction Decode (ID) stage.
4. **PC Update:** Immediately after fetching the current instruction, the PC is usually updated to point to the next instruction in the program sequence. For sequential execution, this typically involves incrementing the PC by the size of one instruction (in bytes). However, this update might be overridden later by control flow instructions like branches or jumps. The potentially updated PC value is often passed along with the fetched instruction to the next stage, as it might be needed for branch address calculations or in case of pipeline flushes due to control hazards.

In a pipelined system, while one instruction is being fetched in the IF stage, the previous instruction is in the ID stage, and so on. This overlap is what allows for the increased throughput.

However, the IF stage can be affected by certain issues. For example, if the instruction memory is slow or if there's a need to fetch an instruction from a location not readily available e.g., due to a cache miss, which is beyond the scope of our basic 5-stage pipeline but worth noting in a real system, the IF stage might stall, which in turn would stall the entire pipeline behind it.

In the context of control hazards like branches, the IF stage plays a crucial role in fetching the instruction at the predicted address. If the prediction is wrong, the instructions fetched in the IF stage and following stages based on the wrong prediction will need to be discarded, a pipeline flush, and the IF stage will then fetch the instruction from the correct address.

Our simulation will visualize the IF stage by showing an instruction being fetched from a conceptual instruction memory based on the PC's value and then moving to the next stage. When control hazards and branch prediction are involved, you'll see how the IF stage might fetch instructions along a predicted path and how it might be redirected if a misprediction occurs.

3.2 Instruction Decode (ID)

The Instruction Decode (ID) stage is the second stage in our 5-stage pipeline. Once an instruction has been fetched from memory in the IF stage, it moves to the ID stage where its meaning is determined, and the necessary preparations for the next execution are made.

Key operations performed in the ID stage include:

1. **Instruction Decoding:** The instruction's binary format is analyzed to identify the opcode and the operands it requires. The opcode dictates the operation to be performed like addition, subtraction, load, store, branch. The operands typically specify registers, immediate values, or memory addresses.
2. **Register Fetch:** If the instruction requires operands from the register file as is common for R-type and I-type instructions in many ISAs, the addresses of the source registers are identified from the instruction. The values stored in these registers are then read from the register file. These fetched register values are crucial for the Execute stage.
3. **Immediate Value Extraction:** For instructions that include an immediate value, a constant embedded within the instruction, common in I-type and some B-type instructions, this value is extracted and possibly sign-extended to the appropriate size for use in later stages.
4. **Branch Target Calculation:** For branch instructions (B-type), the target address to which the program might branch is often calculated at this stage. This typically involves adding an offset that is specified in the instruction, to the current Program Counter, the address of the next instruction.
5. **Control Signal Generation:** Based on the decoded instruction, the control unit generates the control signals that will be needed in the subsequent stages (EX, MEM, WB). These signals dictate things like the operation the ALU should perform, whether memory should be read or written, and whether the result should be written back to the register file.

The outputs of the ID stage, which are passed on to the Execute (EX) stage, typically include:

- The decoded instruction.
- The operands read from the register file.
- The immediate value.
- The calculated branch target address, if the instruction is a branch.
- The control signals for the following stages.

An important aspect handled in the ID stage is the initial detection of some data dependencies. If an instruction being decoded needs a register that is the destination of an instruction still in the pipeline but not yet in the Write Back stage, a data hazard, specifically a RAW hazard, might be detected here. The ID stage then plays a role in signaling the need for hazard handling, such as stalling the pipeline or initiating forwarding.

Our simulation will show the instruction being decoded, the registers being read (if applicable), and the preparation of the necessary data for the next stage. When hazards occur, the visualization will highlight the detection in this stage and the subsequent actions taken.

3.3 Execute (EX)

The Execute (EX) stage is the third stage in the 5-stage pipeline, and it's where the core operations of many instructions are carried out. This stage primarily involves the Arithmetic Logic Unit (ALU).

The main tasks performed in the EX stage include:

1. **ALU Operations:** For R-type e.g., ADD, SUB and I-type e.g., ADDI instructions, the ALU performs the arithmetic or logical operation specified by the instruction's opcode. The operands for the ALU are typically the register values fetched in the ID stage or an immediate value from the instruction.
2. **Address Calculation:** For memory access instructions LW, SW, the ALU calculates the effective memory address. This usually involves adding a base register value from the ID stage to an offset an immediate value from the instruction.
3. **Branch Condition Evaluation:** For branch instructions (B-type, e.g., BEQ), the ALU performs the comparison of the register values fetched in the ID stage. The result of this comparison like equal or not equal, determines whether the branch will be taken.

The output of the EX stage depends on the type of instruction:

- For R-type and I-type ALU instructions, the result of the ALU operation is passed to the next stage MEM or directly to WB if no memory access is needed.
- For load and store instructions, the calculated memory address is passed to the MEM stage. For store instructions, the data to be written to memory, which was fetched in the ID stage, is also passed along.
- For branch instructions, the result of the branch condition evaluation (taken or not taken) and the calculated branch target address, from the ID stage are passed along for control flow management.

The EX stage is also crucial for detecting certain types of data hazards that might not have been fully resolved in the ID stage. For example, if the ALU operation depends on a result that is still in the EX stage of a preceding instruction, forwarding logic would typically operate around this stage to supply the result directly.

Our simulation will visualize the ALU performing these operations. The EX stage is also where the effects of data forwarding will be evident, as results might bypass the register file and be used directly as inputs to the ALU.

3.4 Memory Access (MEM)

The Memory Access (MEM) stage is the fourth stage in our 5-stage pipeline. It is mainly involved with instructions that need to access the computer's data memory, namely load e.g., LW - Load Word and store e.g., SW - Store Word instructions. For other types of instructions like arithmetic, logical, and branches, this stage typically performs no operation or acts as a pass-through.

Here's what happens in the MEM stage:

1. **Load Instructions:** For a load instruction, the memory address calculated in the Execute (EX) stage is used to read data from the data memory. The data read from the specified memory location is then passed on to the Write Back (WB) stage.
2. **Store Instructions:** For a store instruction, the memory address calculated in the EX stage is used to write data to the data memory. The data to be written was typically fetched from a register in the Instruction Decode (ID) stage and passed through the pipeline stages to reach the MEM stage. After the write operation, the store instruction usually does not produce a result that needs to be written back to the register file.
3. **Non-Memory Instructions:** For instructions that do not involve memory access like R-type ALU operations or branches, the data received from the Execute stage is simply passed on to the next stage, Write Back for ALU results, or onwards for branch resolution.

It's important to note that the instruction and data memories are conceptually separate in this simplified view a characteristic of a Harvard architecture, though a pipelined CPU can also be implemented with a single unified memory, requiring careful handling of potential structural hazards if IF and MEM stages try to access it simultaneously.

In terms of hazards, the MEM stage is often where the data fetched by a load instruction becomes available. Instructions in earlier stages that depend on this data will need to either stall until this point or have the data forwarded from the MEM stage if our hazard handling includes this capability.

Our simulation will visualize the MEM stage by showing data being read from or written to the memory locations based on the address provided. For load instructions, you'll see data moving from the memory to the next stage. For store instructions, you'll see data being written into a memory location. For other instructions, you'll observe the data passing through.

3.5 Write Back (WB)

The Write Back (WB) stage is the final stage in our 5-stage pipeline. Its primary purpose is to write the results of an instruction back to the register file. Not all instructions produce a result that needs to be written back; for example, store instructions typically do not.

Here's what happens in the WB stage:

1. **Result Selection:** The data that needs to be written back to a register is selected. This data could have come from the Memory Access (MEM) stage in the case of a load instruction or directly from the Execute (EX) stage in the case of an ALU operation.
2. **Register Write:** The selected result is then written into the destination register specified by the instruction. This updates the state of the register file, making the result available for subsequent instructions.

For instructions that do not produce a result to be written to a register (like store instructions or some branch instructions), this stage essentially does nothing.

The Write Back stage is crucial for making the outcomes of computations and memory loads visible to other instructions. Data hazards can occur if an instruction needs to read a register that is being written to in the WB stage of a preceding instruction or will be written to in a later stage. Forwarding mechanisms often supply data from earlier pipeline stages to avoid stalls in such cases, but the ultimate write to the register file happens here.

4. Pipeline Hazards and Handling Mechanisms

The overlapping execution of instructions in a pipeline, while increasing throughput, can lead to situations known as pipeline hazards. These hazards prevent the next instruction in the instruction stream from executing during its designated clock cycle. Understanding and preventing these hazards when possible is crucial for achieving stability and the performance benefits of pipelining. There are different types of hazards that can occur in our 5-stage pipeline and specific techniques used to manage them.

4.1. Types of Pipeline Hazards

In a pipelined processor, the smooth flow of instructions can be disrupted by three primary types of hazards:

1. **Structural Hazards:** These occur when multiple instructions in the pipeline need to use the same hardware resource at the same time.
2. **Data Hazards:** These arise when an instruction depends on the result of a previous instruction that is still in the pipeline.
3. **Control Hazards:** These occur due to the changes in the program flow caused by instructions like branches and jumps.

4.1.1. Structural Hazards

Structural hazards arise when two or more instructions in the pipeline contend for the same hardware resource at the same time. If a resource is not fully pipelined (i.e., it can only be used by one instruction at a time), and multiple stages of different instructions need to access that resource simultaneously, a structural hazard occurs.

Consider our 5-stage pipeline (IF, ID, EX, MEM, WB) and potential shared resources:

- **Memory:** If the instruction memory and data memory were unified and accessible in the same cycle by both the Instruction Fetch (IF) stage and the Memory Access (MEM) stage, a structural hazard could occur if one instruction is being fetched while another is trying to access data.
- **Register File:** If the register file had only one read port and one write port, a structural hazard could occur if one instruction in the ID stage needs to read two registers while another instruction in the WB stage needs to write a result to a register, all in the same clock cycle. Modern designs typically have multiple read and write ports to mitigate this.

In a well-designed pipeline, structural hazards are often minimized or eliminated by ensuring that each stage has the necessary resources and that shared resources are used at different times or duplicated. For example, having separate instruction and data memories (a Harvard architecture) resolves the memory access conflict between IF and MEM stages. Similarly, having multiple read and write ports in the register file reduces contention during the ID and WB stages.

It is important to understand that structural hazards are a possibility in pipelined architectures and are often addressed at the architectural design level through resource allocation and management.

4.1.2. Data Hazards

Data hazards arise when an instruction needs to use the result of a preceding instruction that is still in the pipeline. This dependency can prevent the following instruction from executing correctly if the result is not yet available. There are three main types of data hazards:

- **Read After Write (RAW):** Occurs when an instruction tries to read a register or memory location before a preceding instruction has written to it. This is the most common type of data hazard.

```
Instruction 1: ADD R1, R2, R3  (writes to R1)
Instruction 2: SUB R4, R1, R5  (reads from R1)
```

In this example, Instruction 2 needs the result produced by Instruction 1. If Instruction 2 reaches the ID stage before Instruction 1 reaches the WB stage, the value read for R1 will be incorrect, it will be the old value.

- **Write After Read (WAR):** Occurs when an instruction tries to write to a register or memory location before a preceding instruction has read from it. WAR hazards are less common in typical in-order pipelines without write buffering that reorders writes.

```
Instruction 1: LW R1, 0(R2)   (reads from memory into R1)
Instruction 2: ADD R2, R3, R4  (writes to R2)
Instruction 3: SW R5, 0(R2)   (reads from R2)
```

Here, if Instruction 2 writes to R2 before Instruction 1 reads from memory using the original value of R2, a WAR hazard occurs with respect to R2 between Instructions 1 and 2. Similarly, there's a WAR hazard on R2 between Instructions 1 and 3. However, in a standard 5-stage pipeline where reads happen early in ID and writes happen late in WB, WAR hazards on registers are often naturally avoided for in-order execution.

- **Write After Write (WAW):** Occurs when two instructions try to write to the same register or memory location, and the writes occur in an order different from the program order. WAW hazards are also less common in simple in-order pipelines.

```
Instruction 1: ADD R1, R2, R3  (writes to R1)
Instruction 2: SUB R1, R4, R5  (writes to R1)
```

If Instruction 2 completes its WB stage before Instruction 1 (which shouldn't happen in a strictly in-order pipeline), a WAW hazard would occur, and the final value in R1 would be incorrect according to the program order.

Of these, RAW hazards are the most prevalent and the ones that pipelining needs to address most directly to maintain precision and performance. Techniques like forwarding (bypassing) and stalling are commonly used to handle RAW hazards.

4.1.3. Control Hazards (Branch Hazards)

Control hazards, also known as branch hazards, are caused by the pipelined execution of instructions that change the control flow of the program, such as branch instructions like conditional branches (BEQ, BNE) and jump instructions. When a branch instruction is encountered, the pipeline faces a decision, which instruction should be fetched next. Should it be the instruction immediately following the branch (assuming the branch is not taken), or the instruction at the branch target address (if the branch is taken).

The issue is that the outcome of a conditional branch, whether it's taken or not, is typically not known until the Execute (EX) stage, where the condition is evaluated. However, the Instruction Fetch (IF) stage needs to fetch the next instruction much earlier. This delay in determining the next instruction to fetch can lead to inefficiencies in the pipeline.

Consider the following sequence:

```
100: BEQ R1, R2, 116  ; Branch to address 116 if R1 equals R2
104: ADD R3, R4, R5
108: SUB R6, R7, R8
112: AND R9, R10, R11
116: OR R12, R13, R14
```

When the BEQ instruction is in the IF stage or shortly after, the CPU does not yet know if R1 equals R2. If it proceeds to fetch the instruction at address 104, assuming the branch is not taken, and then later finds out the branch should have been taken, the instructions fetched and partially executed like the ADD, SUB, AND, were fetched incorrectly and need to be discarded. This discarding of incorrectly fetched instructions is called flushing the pipeline. After flushing, the

CPU must then fetch the instruction at the correct target address (116 in this case), leading to a performance penalty.

Several techniques are used to handle control hazards:

- **Stalling:** The simplest approach is to stall the pipeline whenever a branch instruction is encountered. This means that the instructions following the branch in the pipeline are allowed to proceed, but no new instructions are fetched until the outcome of the branch is known typically after the EX stage. Once the branch outcome is determined, the pipeline either continues fetching sequentially or fetches from the branch target address. Stalling is easy to implement but can significantly reduce the performance benefits of pipelining as several cycles might be wasted waiting for the branch outcome.
- **Branch Prediction:** To reduce the performance penalty of stalling, processors often employ branch prediction. The idea is to guess whether a branch will be taken or not. Based on this prediction, the CPU fetches the next instruction from the predicted address. If the prediction is correct, the pipeline continues without any delay. However, if the prediction is incorrect, the pipeline needs to be flushed, and the instruction fetch restarts from the correct address, incurring a performance penalty equal to the pipeline flush.

Branch prediction can range from simple static predictions (e.g., always predict not taken, always predict taken) to more complex dynamic predictions that use the history of previous branch outcomes to make more informed guesses.

- **Delayed Branching:** This technique reorders instructions such that the instruction immediately following the branch is always executed, regardless of whether the branch is taken or not. The compiler tries to put a useful instruction that is independent of the branch outcome in this "delay slot". If a useful instruction can be placed there, the branch penalty can be reduced. However, finding such an instruction is not always possible, and this technique becomes less effective with deeper pipelines.

Our simulation demonstrates some of these control hazard handling techniques, particularly stalling and various branch prediction strategies. You'll be able to observe how the pipeline behaves when a branch is encountered and the impact of different prediction methods on the flow of instructions.

4.2. Hazard Resolution Strategies

Bearing in mind these types of pipeline hazards, we can see the common techniques used to resolve or mitigate their impact on the performance of a pipelined CPU.

4.2.1. Data Hazard Resolution

Data hazards, as we discussed, happen when an instruction depends on the result of a preceding instruction that is still in the pipeline. To maintain correct execution and minimize performance issues, several techniques are employed. The two primary methods are forwarding (also known as bypassing) and pipeline interlocks (stalling).

4.2.1.1. Forwarding (Bypassing)

Forwarding is a technique used to resolve RAW data hazards by providing the result of an instruction to a dependent instruction as soon as it is available, without waiting for the result to be written back to the register file. This typically involves adding extra data paths that "forward" the result from an internal pipeline register to the input of a subsequent instruction that needs it.

Consider the RAW hazard example again:

```
ADD R1, R2, R3
SUB R4, R1, R5
```

In a 5-stage pipeline, the ADD instruction produces the result in the Execute (EX) stage. This result is then passed to the Memory (MEM) stage and finally written back to R1 in the Write Back (WB) stage. However, the SUB instruction needs the value of R1 in its own Execute stage. If we simply waited for the ADD instruction to complete the WB stage, the SUB instruction would have to stall.

Forwarding addresses this by taking the result of the ADD instruction directly from the output of its EX stage or the MEM stage, if that's when the result is available and feeding it as an input to the EX stage of the SUB instruction. This bypasses the normal path through the register file, allowing the SUB instruction to proceed without waiting for the ADD to reach WB.

To implement forwarding, the CPU needs:

1. **Hazard Detection Logic:** To identify when a RAW hazard exists by comparing the destination register of an instruction in an earlier stage with the source registers of an instruction in a later stage.
2. **Forwarding Paths:** Extra hardware to route the result from the output of the EX or MEM stage back to the input of the ALU in the EX stage.

For example, if the SUB instruction in its ID stage decodes and finds that it needs the value of R1, and it detects that an ADD instruction currently in the EX stage is going to write to R1, the control logic will enable the forwarding path to send the output of the ALU from the ADD instruction directly to the ALU input for the SUB instruction in the next clock cycle.

Forwarding can handle many RAW dependencies, significantly reducing the need for stalls and improving performance. However, not all RAW hazards can be resolved by simple forwarding. For instance, a load instruction that reads from memory has its result available only at the end of the Memory (MEM) stage. If an instruction immediately following the load needs to use the loaded value in its EX stage, forwarding from the MEM stage might still require a stall cycle because the data is not available early enough in the pipeline for the subsequent instruction's EX stage. This leads us to the next technique: stalling.

4.2.1.2. Pipeline Interlocks (Stalling)

When a data hazard cannot be fully resolved by forwarding, or when the required data is not yet available in a forwardable stage, pipeline interlocks, or stalling, are used. Stalling involves temporarily halting the progress of one or more instructions in the pipeline to allow a preceding instruction to advance far enough to produce the needed data. This introduces "bubbles" (empty slots) into the pipeline, where no useful work is done, thus reducing the overall performance but ensuring correct execution.

Consider a load instruction followed immediately by an instruction that uses the loaded value:

```
LW R1, 0(R2)   ; Load a word from memory into R1
ADD R3, R1, R4  ; Add R4 to the value in R1
```

The LW instruction fetches the data from memory in the MEM stage, and the result is written to R1 in the WB stage. The ADD instruction in the very next cycle will be in its EX stage and will need the value of R1. If only forwarding from the MEM stage is available, the data might not arrive at the ALU input of the ADD instruction in time for the EX stage.

In such cases, a pipeline interlock is implemented. The ADD instruction (and any instructions following it in the pipeline) is stalled in the ID stage. This means it does not advance to the EX stage in the next clock cycle. Simultaneously, a "bubble" (a no-operation effectively) is inserted

into the EX stage. The LW instruction continues to proceed through the pipeline. In the subsequent clock cycle, the data from memory will be available at the end of the MEM stage and can be forwarded to the EX stage of the ADD instruction, which is now allowed to proceed.

The prerequisites for implementing stalls typically involve:

1. **Hazard Detection:** Identifying the data dependency that cannot be resolved by forwarding (e.g., a load followed by a dependent instruction in the next cycle).
2. **Control Logic:** To prevent the stalled instruction from advancing to the next stage and to insert a "bubble" into the preceding stage. This might involve holding the control signals of the stalled instruction and injecting a no-op into the pipeline.

While stalling ensures accuracy, it does come at a performance cost because the pipeline is not fully utilized during the stall cycles. The goal of architectural design and hazard handling techniques like forwarding is to minimize the frequency and duration of these stalls.

4.2.2. Control Hazard Resolution

Control hazards, arising from branch and jump instructions, pose a challenge to pipelined execution because the decision about the next instruction to fetch is not known until later in the pipeline, potentially after several subsequent instructions have already been fetched. We will explore the primary strategies for dealing with these hazards: stalling and branch prediction.

4.2.2.1. Stalling for Branches

One of the simplest ways to handle control hazards is to stall the pipeline whenever a branch instruction is encountered. When a branch instruction is decoded in the ID stage, the pipeline simply waits until the outcome of the branch, taken or not taken, and the target address, if taken, are determined, typically in the Execute (EX) stage. During this waiting period, no new instructions are fetched, and the instructions following the branch in the pipeline are allowed to continue. Once the branch outcome is known, the pipeline either continues fetching the next sequential instruction if the branch is not taken or starts fetching from the target address if the branch is taken.

The performance penalty of branch stalling is significant, especially in pipelines with many stages or programs with frequent branches. For a 5-stage pipeline, if the branch outcome is decided in the EX stage, it means that the IF and ID stages fetched instructions based on the assumption of sequential execution, which might be incorrect. These incorrectly fetched instructions would then need to be discarded, and the pipeline would have experienced a stall of several cycles.

For example, consider a branch instruction. While it's in the ID stage, the next instruction has already been fetched. If the branch outcome is only known at the end of the EX stage, the instruction fetched after the branch (and the one after that, if the pipeline is deep enough) might be the wrong one. Stalling would involve freezing the pipeline after the branch instruction enters the ID stage until the EX stage of the branch determines the next PC value. This means the IF stage would not fetch any new instructions during this stall. Once the branch outcome is known, the correct instruction fetch resumes.

While straightforward to implement, stalling significantly reduces the performance gains of pipelining by introducing idle cycles. Therefore, more advanced techniques like branch prediction are often used to mitigate the impact of control hazards.

4.2.2.2. Branch Prediction Strategies

To reduce the performance penalty associated with control hazards, particularly the stalls caused by waiting for the outcome of branch instructions, modern processors often use branch prediction. The goal of branch prediction is to guess whether a branch will be taken or not taken, and if taken, to predict the target address. Based on this prediction, the processor fetches and begins executing instructions along the guessed path.

If the prediction is correct, the pipeline continues without any interruption, effectively hiding the branch delay. However, if the prediction is incorrect, the pipeline must be flushed and the fetched instructions are discarded, then processor restarts from the correct address, causing a performance penalty. The effectiveness of a branch prediction scheme is determined by its accuracy, the higher the prediction accuracy, the lower the performance loss due to mispredictions.

There are various branch prediction strategies, ranging from simple static methods to more complex dynamic methods:

- **Static Branch Prediction:** These methods make the same prediction for all branches or classify branches based on their type or direction. Common static prediction strategies are:
    - **Always Not Taken:** Predicts that the branch will never be taken. This is often a reasonable default for forward branches in loops.
    - **Always Taken:** Predicts that the branch will always be taken. This might be more effective for backward branches, which often form loops.

- **Dynamic Branch Prediction:** These methods use the history of past branch outcomes to predict the outcome of the current branch. They tend to be more accurate than static methods but require more hardware to store the history. They include:
    - **1-bit Branch Prediction:** This scheme uses a 1-bit history register for each branch or a set of branches. This bit stores the outcome of the last execution of the branch, taken or not taken. The prediction for the next execution is the same as the last outcome. However, this predictor can suffer from mispredictions at the exit of a loop.
    - **2-bit Branch Prediction (Saturating Counter):** This is a more sophisticated approach that uses a 2-bit counter for each branch. The counter has four states. Incrementing the counter moves it towards the "strongly taken" state, and decrementing moves it towards the "strongly not taken" state. A prediction is made based on the higher two states (predict taken) or the lower two states (predict not taken). This 2-bit predictor is more resistant to single mispredictions, such as the last iteration of a loop.

Our simulation implements some of these branch prediction strategies, allowing you to observe their behavior and impact on pipeline flow. You'll be able to see how different strategies perform on the same sequence of instructions containing branches.

5. Simulation and Visualization Software

The simulation and visualization of our 5-stage hazard handling CPU are implemented using a modern web development stack. The user interface is constructed with HTML providing the structural foundation, styled responsively with Tailwind CSS, and made interactive and dynamic through the use of React with TypeScript. This combination allows for a rich, browser-based experience that effectively conveys the complexities of pipelined execution.

5.1. Animated Pipeline Visualization

A central element of the simulation is the animated visualization of the 5-stage pipeline. This is presented as a grid-based layout at the top of the user interface, clearly delineating the Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB) stages.

As the simulation progresses through clock cycles, each instruction currently being processed is represented as a distinct visual entity, often referred to as a "bubble." These bubbles move from left to right across the grid, transitioning through the pipeline stages in sequence. The animation provides an intuitive sense of the temporal parallelism achieved by pipelining, showing multiple instructions in different stages of execution simultaneously.

This allows users to easily track the progression of a specific instruction as it moves through the pipeline. This animated pipeline visualization serves as the primary tool for observing the CPU's behavior, making it easier to grasp concepts like instruction flow, pipeline depth, and the impact of hazards and their handling mechanisms in a dynamic and engaging way.

5.2. Instruction Set and Execution

The simulation showcases the principles of pipelining and hazard handling through the execution of a predefined set of 10 sample instructions. These instructions are hardcoded into the simulation for demonstration purposes and cover a range of common instruction types found in RISC-like architectures. The included instructions are designed to illustrate various aspects of CPU operation, including arithmetic and logical operations, memory access, and control flow changes.

The instruction set includes examples from the following categories:

- **R-type Instructions:** These instructions typically perform operations between two registers and store the result in a third register. Examples in your set include ADD and SUB. These are useful for demonstrating data dependencies (RAW hazards).
- **I-type Instructions:** These often involve an operation between a register and an immediate value or are used for memory access with an offset. Your set includes LW (Load Word) to read from memory into a register, and SW (Store Word) to write from a register to memory. LW instructions are particularly relevant for illustrating load-use data hazards and the need for stalling or forwarding.

- **B-type Instructions:** These are branch instructions that conditionally change the program's control flow based on the comparison of two registers. Your set includes BEQ (Branch if Equal), which is crucial for demonstrating control hazards and the effects of branch prediction strategies.

These 10 sample instructions are loaded into a simulated instruction memory at the start of the simulation. As the user steps through the clock cycles, the simulation fetches these instructions in order unless a branch is taken, and they progress through the pipeline stages visualized at the top. You can observe the operands being read from the simulated register file, the ALU performing operations, memory being accessed, and results being written back to registers. This controlled set of instructions allows for a focused study of how the pipeline handles different types of operations and the hazards they might introduce.

5.3. Registers and Memory Visualization

To provide a comprehensive view of the CPU's operational state, the simulation includes interactive visualizations of the register file and the data memory.

Registers: The simulation models a set of 32 general-purpose registers, a common feature in many modern architectures. These registers are displayed in the user interface, typically in a grid format for easy readability. Crucially, to make the simulation more realistic and to better illustrate data flow, these registers are initialized with diverse, non-zero values at the start of the simulation. As instructions execute and results are written back in the WB stage, these register values are updated in the visualization in real-time. This allows users to directly observe how instructions modify the CPU's register state. By tracking the changes in register values, one can follow the data dependencies between instructions and the effects of the Write Back stage.

Memory: Alongside the registers, the simulation also visualizes a segment of the main memory, consisting of 64 memory locations. Similar to the registers, these memory locations are initialized with various values at the beginning of the simulation. The memory visualization is essential for observing the effects of load (LW) and store (SW) instructions. When a LW instruction executes its MEM stage, the value read from the corresponding memory address is displayed. Conversely, when a SW instruction reaches the MEM stage, the data being written to a memory location is shown, and the memory content in the visualization is updated accordingly. This allows users to see how data moves between registers and memory under the control of load and store operations.

The visual presentation of both the registers and memory in grid layouts makes it straightforward to inspect their contents at any point during the simulation, aiding in the understanding of the CPU's internal state and the impact of the executed instructions.

5.4. Pipeline Register Visualization

In a pipelined CPU, the stages are separated by pipeline registers. These registers are essential for holding the intermediate results and control signals as an instruction moves from one stage to the next. For our 5-stage pipeline, we have pipeline registers between IF and ID (often labeled IF/ID), between ID and EX (ID/EX), between EX and MEM (EX/MEM), and between MEM and WB (MEM/WB).

The simulation provides a visualization of the contents of these pipeline registers at each clock cycle. This allows you to see exactly what information is being passed from one stage to the next for each instruction in the pipeline. The contents typically include:

- The instruction itself.
- Operands read from the register file (in ID/EX).
- Immediate values (in ID/EX).
- The result of the ALU operation (in EX/MEM).
- The data read from memory (in MEM/WB).
- Control signals needed by subsequent stages.

By observing the values held in these pipeline registers, you can gain a deeper understanding of how data flows through the pipeline and how the different stages interact. For instance, when a data hazard occurs, you might see an instruction in a later stage needing a value that is still held in a pipeline register of an earlier stage. Similarly, when forwarding happens, you can trace how the result bypasses the register file and is directly fed from the output of one pipeline register to the input of a functional unit in a later stage.

This visualization of the pipeline registers is a powerful tool for demystifying the internal workings of a pipelined CPU and the mechanisms of hazard handling.

5.5. Hazard Detection and Visualization

A crucial aspect of understanding a hazard-handling CPU is being able to identify when hazards occur and how they are managed. Our simulation includes a sophisticated hazard detection unit that actively monitors the instructions as they progress through the pipeline to identify:

- **RAW (Read After Write) Data Hazards:** These are detected by comparing the destination register of an instruction in an earlier pipeline stage with the source registers of an instruction in a later stage. If a match is found and the writing instruction has not yet reached the Write Back stage, a RAW hazard is flagged.
- **Control (Branch) Hazards:** These are detected when a branch instruction enters the pipeline. The uncertainty about the next instruction to fetch (whether to take the branch or not) constitutes a control hazard.
- **(Potentially) Structural Hazards:** While the design aims to minimize these, the detection logic would identify if two instructions require the same resource at the same time.

When a hazard is detected, the simulation provides us with feedback. This immediate information about the current state of the CPU is invaluable for learning why stalls or forwarding might be necessary and for observing these mechanisms in action.

5.6. Branch Prediction

To address control hazards, our simulation incorporates several branch prediction strategies that can be selected and toggled by the user. These include:

- **Always Not Taken:** The prediction is always that the branch will not be taken, and execution continues with the next sequential instruction.
- **Always Taken:** The prediction is always that the branch will be taken, and the next instruction is fetched from the calculated target address.
- **1-bit Predictor:** A 1-bit history is maintained for each branch. The prediction for the current execution is the same as the outcome of the last execution.
- **2-bit Predictor (Saturating Counter):** A 2-bit counter tracks the history of each branch, allowing for a more nuanced prediction that requires two consecutive mispredictions to change the prediction.

In the case of a misprediction, you will observe a pipeline flush, where the incorrectly fetched instructions are discarded, and fetching restarts from the correct target address. This allows for a direct comparison of the effectiveness of different branch prediction strategies.

5.7. Forwarding Toggle

To illustrate the impact of data forwarding on performance, our simulation includes a user-controllable toggle to enable or disable this feature. When forwarding (bypassing) is enabled, the results from the Execute or Memory stages are directly fed to subsequent instructions that need them, thus avoiding pipeline stalls due to RAW hazards. When forwarding is disabled, you can observe that RAW hazards will result in pipeline stalls, as the dependent instruction must wait for the producing instruction to reach the Write Back stage. This clearly demonstrates the performance advantages of data forwarding.

5.8. Step-Back and Reset

For enhanced control and learning, the simulation provides:

- **Step-Back:** This functionality allows the user to move the simulation back by one clock cycle. This is useful for re-examining the pipeline state and understanding the sequence of events leading to a particular situation, such as a hazard or a forwarding operation. The system likely maintains a history of the CPU's state to enable accurate reversal.
- **Reset:** This button returns the simulation to its initial state. The instruction memory, registers, and data memory are reset to their starting values, allowing users to easily restart the execution from the beginning.

5.9. Execution Log and WB Results

Throughout the simulation, a detailed execution log is maintained and displayed. For each clock cycle, this log records the activity in each pipeline stage, any detected hazards, and the actions taken (e.g., forwarding, stalling, prediction outcomes). Specifically, the results of the Write Back stage, including any updates to registers or memory, are logged and often highlighted. This textual log provides a precise, cycle-by-cycle account of the CPU's operation, complementing the visual animation.

5.10. Export Functionality

For more in-depth analysis or for sharing the results of a simulation run, users can export the current state of the simulation (including register and memory contents, and the state of the pipeline) along with the complete execution log. This data is typically exported in JSON format, which is easily parseable and can be used for further analysis or documentation.

6. Visuals of the Simulation Software and GUI

The following screenshots illustrate key features of our visualization software, that we have already talked about, and provide insights into the system's overall look, flow, and user experience. These images, though now separated for better clarity, collectively offer a comprehensive view of the simulation tools capabilities and user interface. The figures below showcase the primary outputs and interactive elements that help users understand and control the simulated CPU's pipeline behavior.

The first two figures – *Figure 1.* and *2.* – shows the dynamic representation of the pipeline's operation. The Animated Pipeline Visualization at the top uses visual "bubbles" to represent instructions as they move through the five pipeline stages (IF, ID, EX, MEM, WB). At the bottom of *Figure 2.*, various controls allow the user to interact with the simulation, such as stepping through cycles, toggling hazard details and forwarding, selecting branch prediction strategies, resetting the simulation, and exporting the simulation state.

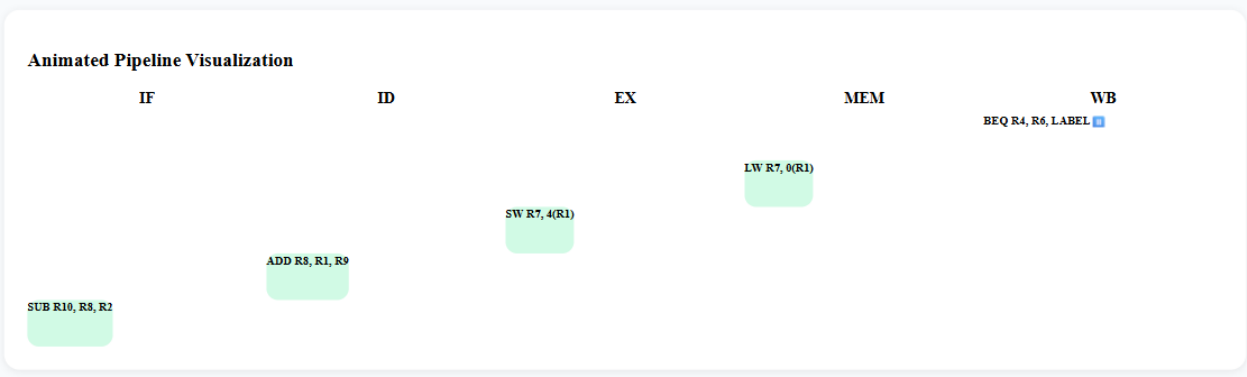

Figure 1. Animated Pipeline Visualization

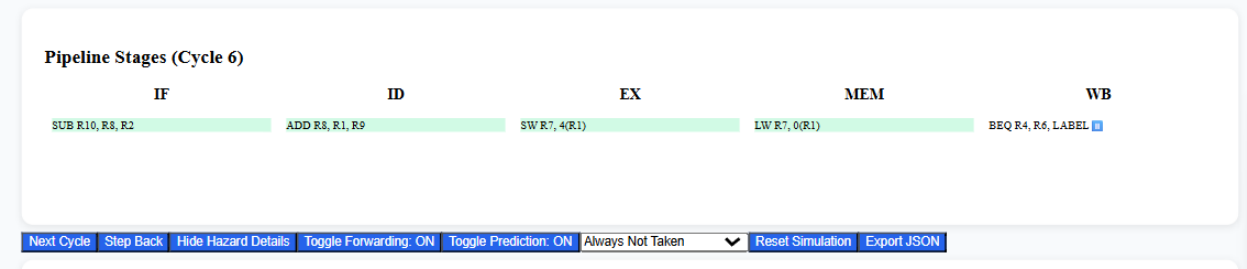

Figure 2. Pipeline Stages

Figure 3. and *Figure 4.* focus on the state of the simulated CPU's memory components and the user interface controls. The Registers section displays the current values of all 32 general-purpose registers. Similarly, the Memory section shows the contents of the 64 data memory locations.

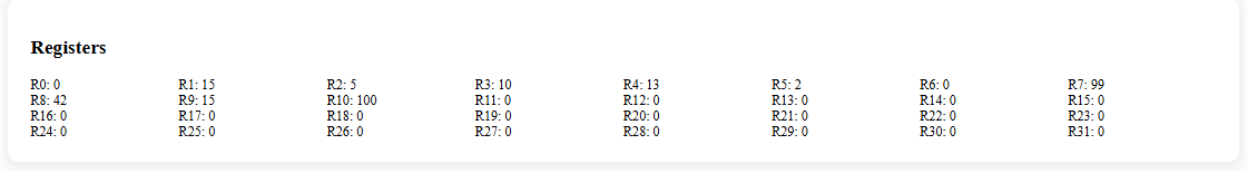

Figure 3. Registers

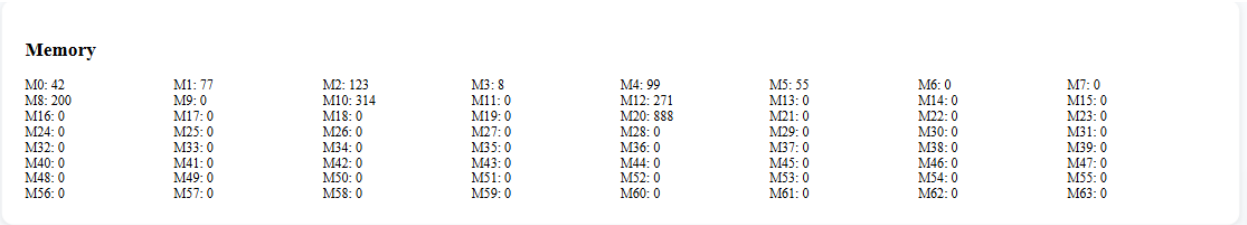

Figure 4. Memory

Lastly, *Figure 5.*, *Figure 6.*, *Figure 7.* and *Figure 8.* display several key textual outputs of the simulation. The Performance Metrics section provides a quantitative summary of the execution, such as the total number of cycles, the count of pipeline stalls, and the calculated CPI. Below this, the Hazard Explanations give a cycle-by-cycle account of what occurred, explicitly mentioning when hazards were encountered and how they were resolved (e.g., by stalling or flushing the pipeline). The Pipeline Registers section shows the data currently held in the inter-stage registers, offering a snapshot of the information being passed between consecutive pipeline stages. Finally, the Instruction Memory lists the sequence of instructions that the simulated CPU is executing.

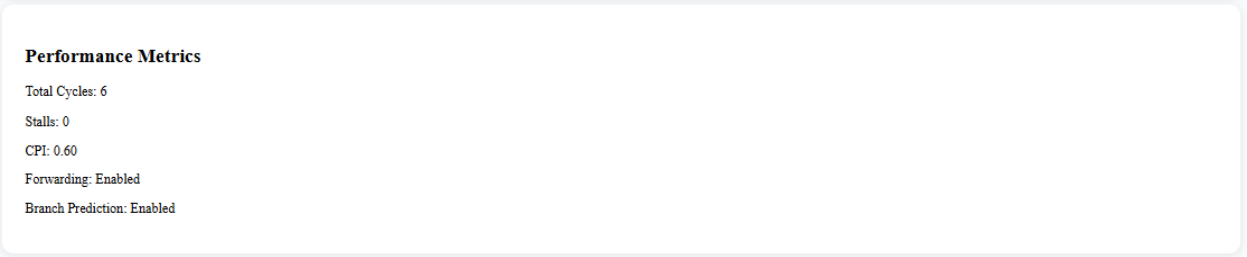

Figure 5. Performance Metrics

Hazard Explanations ❓

- Cycle 0 executed without stall.
- Cycle 1 executed without stall.
- Branch prediction correct at cycle 2.
- Cycle 3 executed without stall.
- Stalling at cycle 4 due to RAW hazard between "LW R7, 0(R1)" and "SW R7, 4(R1)".
- Cycle 4 executed without stall.
- Cycle 5 executed without stall.
- Cycle 6 executed without stall.
- Cycle 7 executed without stall.
- Cycle 8 executed without stall.
- Control hazard resolved by stalling at cycle 9. Flushing pipeline.
- Flushing pipeline at cycle 9 | Flushed instructions: BEQ R10, R7, LABEL2, SW R11, 12(R4), LW R11, 8(R4), SUB R10, R8, R2
- Flushing pipeline at cycle 10 | Flushed instructions: BEQ R10, R7, LABEL2, SW R11, 12(R4), LW R11, 8(R4)

Figure 6. Hazard Explanations

Pipeline Registers

IF/ID: -

ID/EX: -

EX/MEM: BEQ R10, R7, LABEL2

MEM/WB: SW R11, 12(R4)

Figure 7. Pipeline Registers

Instruction Memory

ADD R1, R2, R3
SUB R4, R1, R5
BEQ R4, R6, LABEL
LW R7, 0(R1)
SW R7, 4(R1)
ADD R8, R1, R9
SUB R10, R8, R2
LW R11, 8(R4)
SW R11, 12(R4)
BEQ R10, R7, LABEL2

Figure 8. Instruction Memory

7. Conclusion

This report has detailed the architecture of a 5-stage pipelined CPU designed to handle common pipeline hazards, and the accompanying simulation and visualization software developed using web technologies. The implementation of pipelining aimed to enhance instruction throughput by overlapping the execution of multiple instructions. However, this introduced the challenges of structural, data (RAW, WAR, WAW), and control hazards, which can impede the performance gains of pipelining.

To address these hazards, the simulated CPU incorporates several techniques. Data hazards, particularly RAW dependencies, are managed through data forwarding (bypassing) and pipeline interlocks (stalling). Control hazards arising from branch instructions are tackled using various branch prediction strategies, including static (Always Taken, Always Not Taken) and dynamic (1-bit, 2-bit) prediction, along with pipeline flushing upon mispredictions.

The HTML/JavaScript/React/TypeScript simulation provides an interactive and intuitive platform for understanding these concepts. The animated pipeline visualization allows users to observe the flow of instructions and the occurrence of hazards. The ability to toggle forwarding, select branch prediction strategies, step through execution, and view the register file, memory, and pipeline registers offers a hands-on approach to learning about computer architecture principles. The execution log and export functionality further support detailed analysis and understanding of the CPU's behavior.

Through this project, we have demonstrated the fundamental concepts of pipelining, the challenges posed by pipeline hazards, and the effectiveness of hazard handling mechanisms in maintaining performance and ensuring correct execution. The visualization software serves as a valuable educational tool, making the abstract concepts of computer architecture more tangible and accessible.

8. References

Patterson, D. A., & Hennessy, J. L. (2017). *Computer organization and design RISC-V edition: The hardware/software interface* (5th ed.). Morgan Kaufmann.

Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

Harris, D. M., & Harris, S. L. (2012). *Digital design and computer architecture* (2nd ed.). Morgan Kaufmann.

Mano, M. M., & Ciletti, M. D. (2017). *Digital design: With an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Stallings, W. (2018). *Computer organization and architecture: Designing for performance* (11th ed.). Pearson.