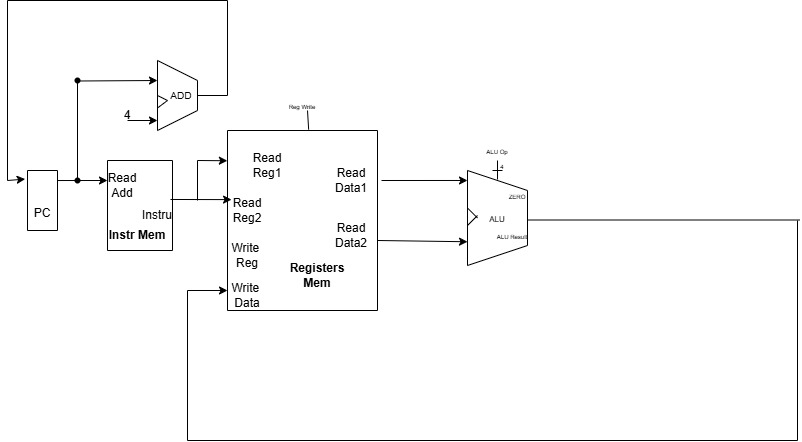
1. **Introduction**

This Instruction Set Architecture (ISA) is designed specifically for a bespoke RISC-V processor intended for AI-driven mobile financial applications in Lesotho. It focuses on affordable mobile devices, providing users with smart, secure, and user-friendly digital banking options. Such services play a vital role in improving financial inclusion in areas with scarce resources. R-type instructions will primarily be used for custom operations, while other types like I-type will handle immediate operations and loads. S-type will be utilized for data storage, B-type for conditional branches, and U/J-type will manage constants and jumps. The primary objective is to enable simple pipeline and support low power operations.

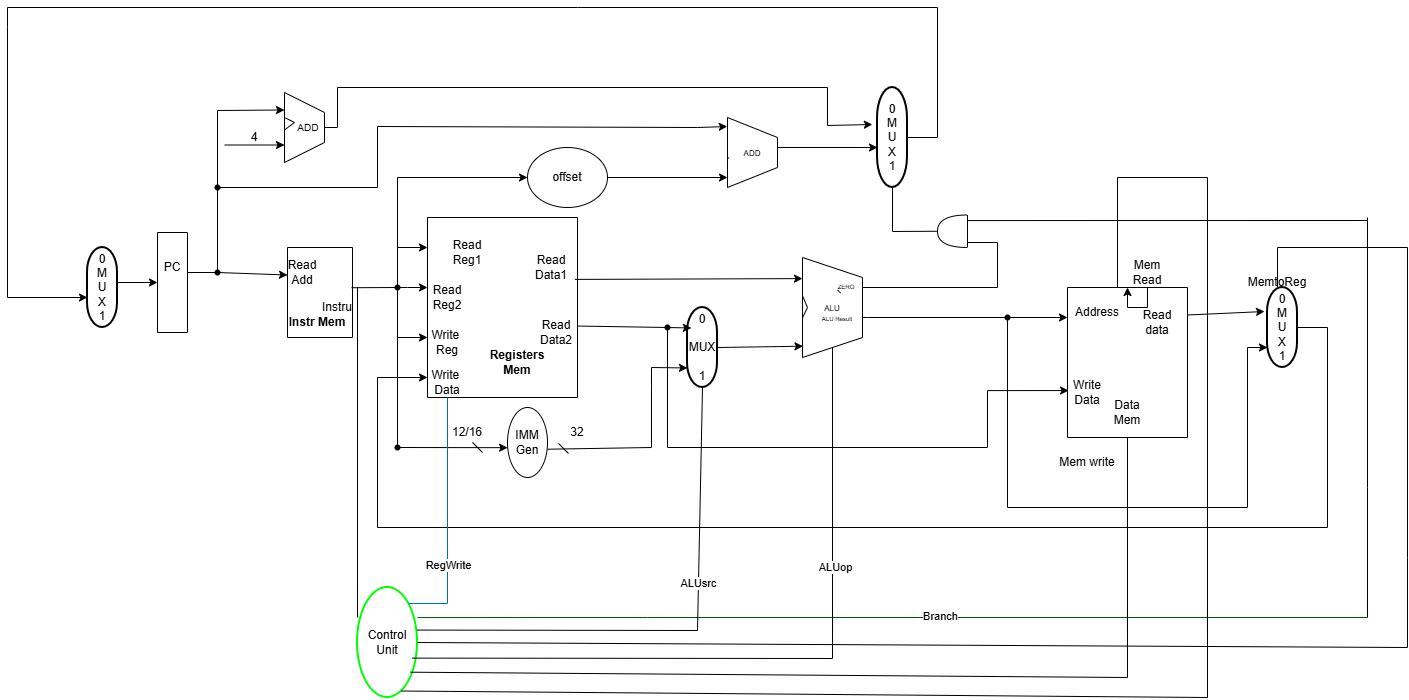
1. **Incremental Datapath Design**

**R-Type Instruction Execution**

****

A simplified processor datapath for executing the instructions. The processor begins by fetching an instruction from the Instruction Memory using the PC. This instruction is then decoded, and the register file is accessed using the specified register addresses (Read Reg1, Read Reg2) to read the source operands (Read Data1, Read Data2). If the instruction is the memory operation, the Data Memory is accessed for reading or writing data. For operations that produce a result (like arithmetic), the result is written back to a destination register in a register file, completing the execution cycle and preparing for the next instruction.

**B-Type Instruction Execution**

****

B-type instructions implement conditional branches for control flow. The Immediate Generator creates the offset, which is lifted left by one and added to the PC via the ADD block to compute the branch target. The ALU compares the two register operands based of function3, outputting a branch condition. A MUX then selects between PC+4 (no branch) and the computed target, feeding the next PC. The branch decision determines control flow without stalling

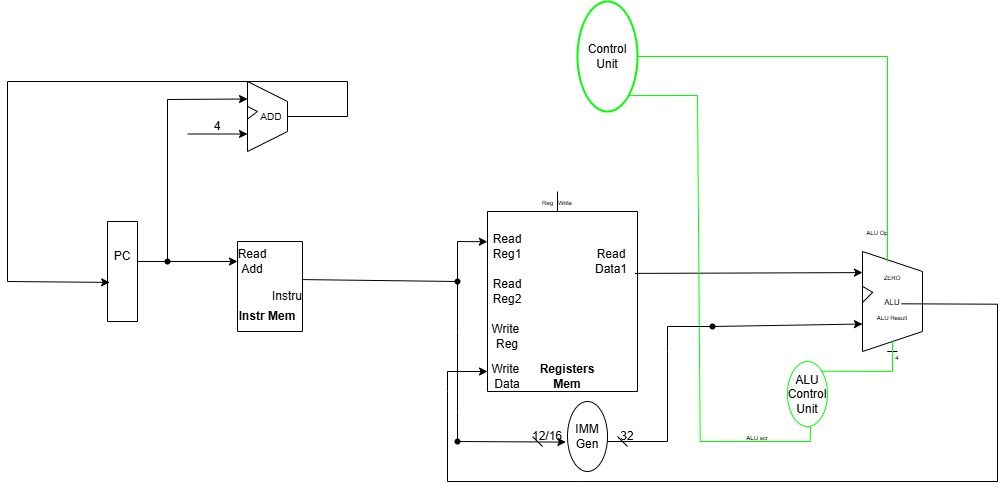
**S-type Instruction Execution**

**A diagram of a machine

AI-generated content may be incorrect.**

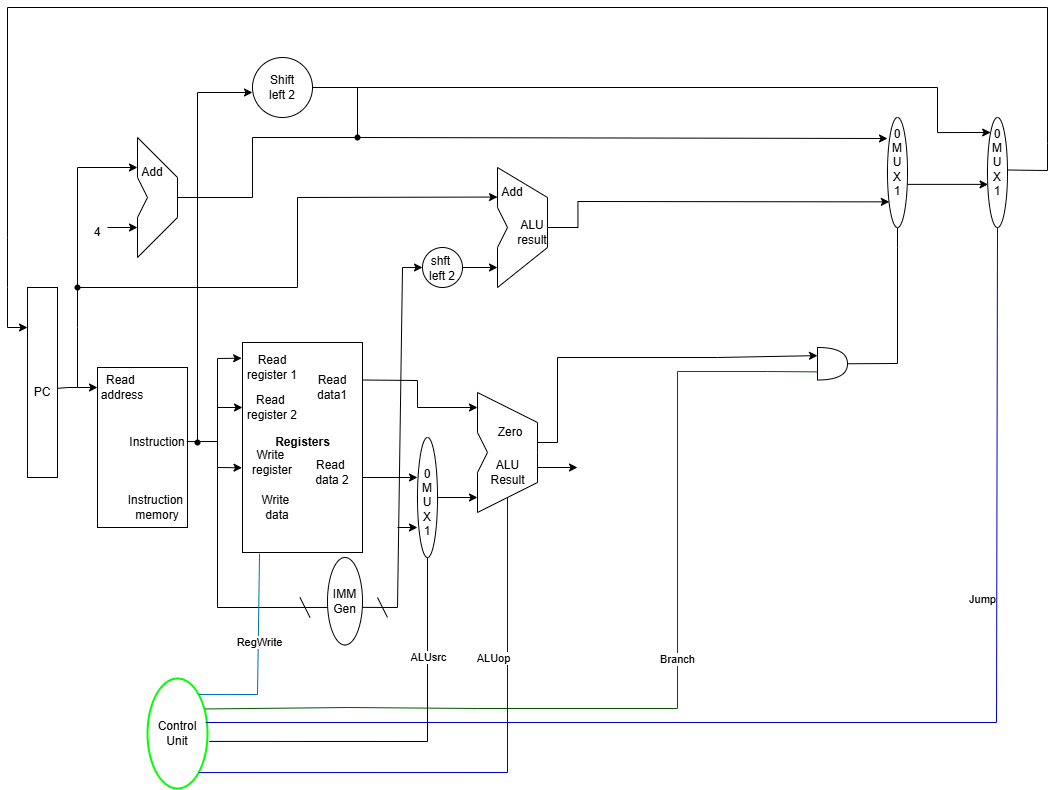
S-type write data to memory. During Decode, rs1 which is base address and rs2 which is data are read. The Immediate Generator extracts and sign-extends the offset, which combines the rs1 in the ALU to produce the memory address. During Memory stage, Memory Write is asserted and the data from rs2writes to the computed address with byte granularity (smallest addressable unit of memory) controlled by funct3.

**I-type Instruction Execution**

****

I-type use immediate operands for arithmetic and loads. During Decode, rs1 is read; The Immediate Generator sign-extends 12-bit immediate. For arithmetic (ADD). The ALU operates on rs1 and immediate, writing results to rd. for loads (LW) , the immediate combines the rs1 in the ALU to compute memory address, which fetches data during Memory stage with alignment based on funct3, then writes to rd.

**J-type instruction Execution**

****

J type instructions allow the program to transfer control to a different part of the code without returning to the original location using unconditional jumps. They are useful for implementation of loops, functions and sub-routines.

The datapath for J-type instructions involves the following steps:

1. Instruction Fetch (IF): The instruction is fetched from memory.

2. Instruction Decode (ID): The instruction is decoded, and the opcode is extracted.

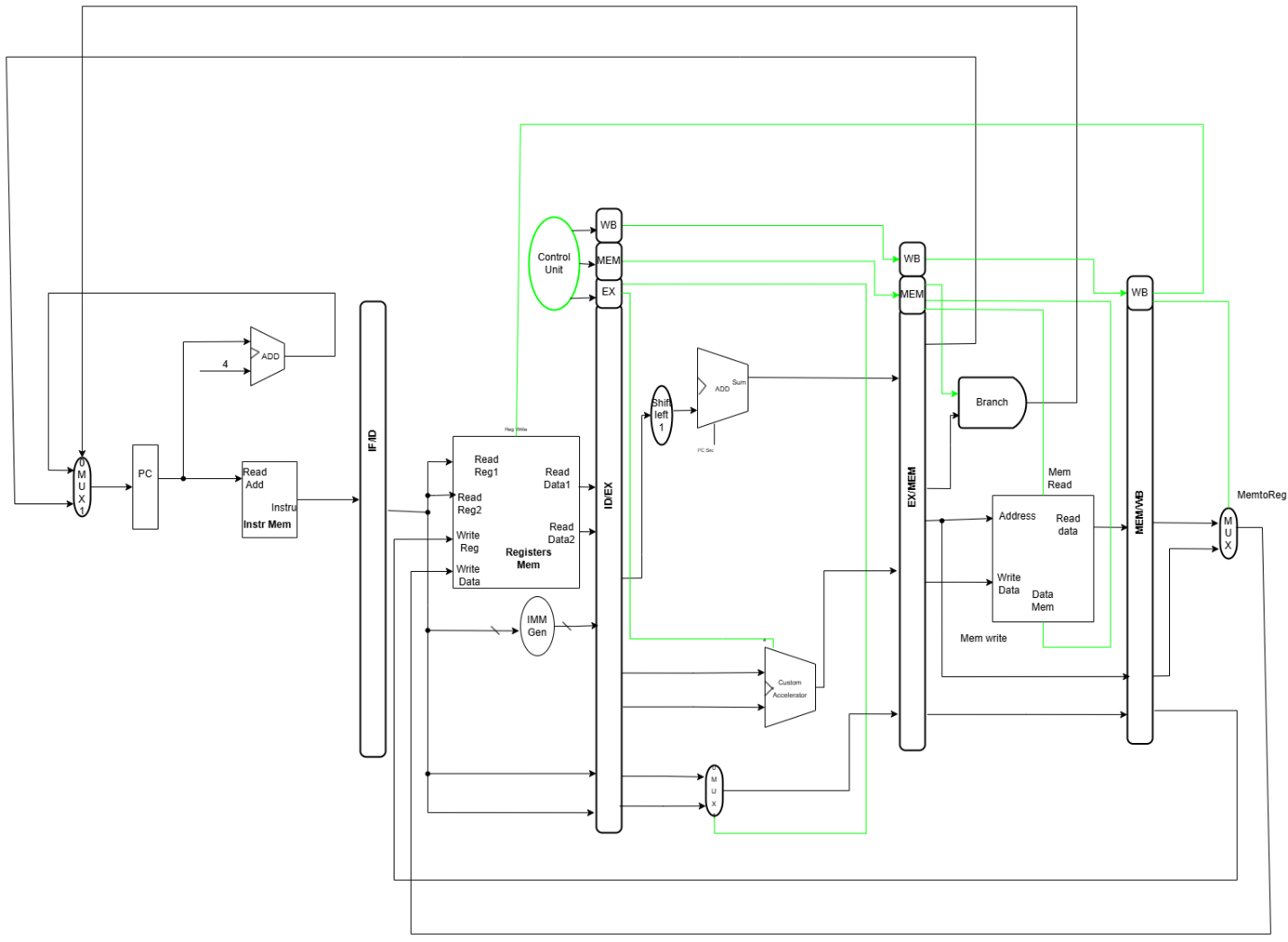
3. Jump Target Calculation: The jump target address is calculated by adding the immediate value (sign-extended) to the current PC (Program Counter).

- The immediate value is extracted from the instruction and sign-extended to 32 bits

- The PC is updated to the calculated jump target address.

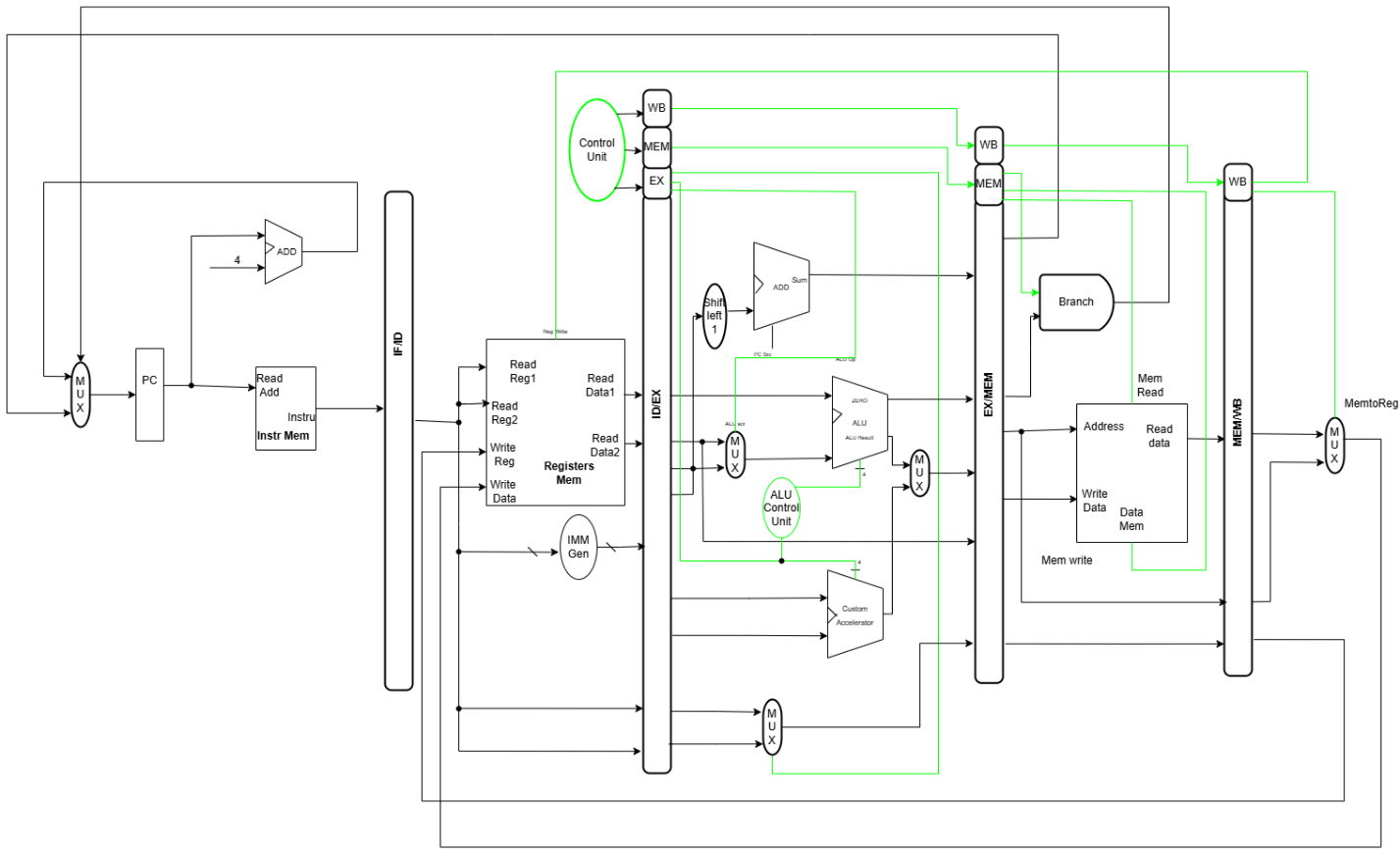
4. PC Update: The new PC value is written to the PC register

**Custom AI/Biometric Instruction Execution**

****

Custom instructions use reserved opcodes (0b1111011) with unique funct7 values. Operands (rs1, rs2) are read normally, but the operation routes to a custom accelerator instead of the standard ALU based on funct7 decoding. The accelerator performs specialized tasks (MFCCX for feature extraction, AUTHUSER for biometric verification) and writes results to rd in Write-Back, integrating seamlessly with the pipeline's forwarding mechanism.

1. **Unified Single-Cycle Datapath**

****

The control unit decodes opcode, funct3, and funct7 to generate signals (ALUSrc, RegWrite, MemRead, MemWrite, Branch) directing instruction execution. Multiplexers route data: ALUSrc selects register or immediate operands, MemtoReg chooses ALU result or memory data for write-back, and PCsrc selects sequential or branch targets. Forwarding monitors ALU and memory results, preventing hazards without stalling. All instruction formats flow through identical Fetch, Decode, Execute, Memory, and Write-Back stages, ensuring uniform treatment across the pipeline.

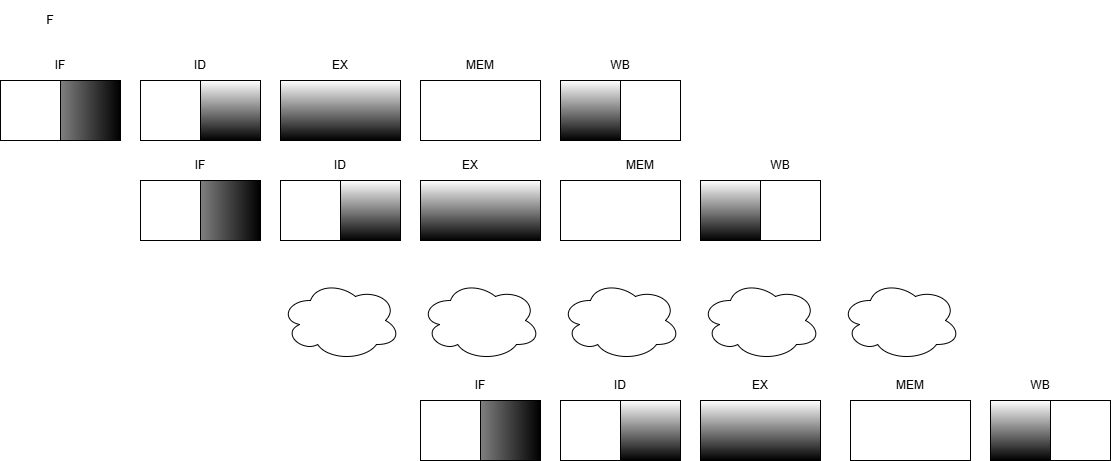
1. **Control Unit Design**

**Truth Table:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instr | ALUscr | MemToReg | RegWrite | MemRead | MemW | Brnch | SecStr | ALUOp |
| R | 0 | 0 | 1 | 0 | 0 | 0 | 0 | ADD/SUB/AND/OR |
| I | 1 | 1 | 1 | 1 | 0 | 0 | 0 | ADD |
| Sw | 1 | X | 0 | 0 | 1 | 0 | 0 | ADD |
| SStore | 1 | X | 0 | 0 | 1 | 0 | 0 | ADD |
| Beq | 0 | X | 0 | 0 | 0 | 1 |  | SUB |
| Custom | 0 | 1 | 1 | 1 | 1 | 1 | 1 | Custom |

1. **Pipelined Implementation**

Our pipelined implementation implementation encounters data hazards at various stages of many of our instructions. For such instructions, we introduced stalling to make sure that registers and data are available when needed. The hazards occurred when a register that is an output address of one instruction is also an needed in the next instruction that is piplined and reaches Instruction Decode beofre the previous instruction reaches Mem/WB stages. Outlined below is an example of one of our customized instructions, MATCHCHAR that encountered data hazards that were resolved by stalling.

****

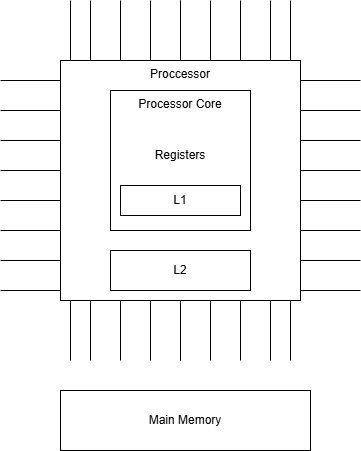
To enhance pipeline optimization, we have implemented a robust forwarding network with multiple bypass paths from all executing stages, EX, MEM and WB to inputs at the EX stage. This minimizes the need for stalls for all hazard but load-use hazard.

By Forwarding, instead of waiting for the result to be written back to the register file at the Write Back (WB) stage, the result is immediately forwarded from the output of an earlier stage such as EX or MEM stage, directly to the input for the EX stage of the dependent instruction. This maintains the idea throughput of one instruction per clock cycle.

For load-use hazards, where a value is loaded from memory and immediately used by the next instruction, we used stalling. This is a method where the pipeline freezes for one or more cycles waiting for a condition to be met, the freezing does not allow new instructions to be fetched

For control hazard, here a processor does not know which instruction to fetch next instruction until a branch condition is evaluated, or control decision is made, we will flush the instructions were incorrectly fetched, and then stall until the branch outcome is known, then fetch the correct instruction.

1. **Memory Hierarchy Design**

****The Memory Hierarchy will utilize L1I and L1D, L2 Cache and main memory.

The L1 Cache (Located in the CPU) is used to store most frequently accessed instructions and data for immediate use. It is in the CPU, thus being the fastest Cache memory because it uses the CPU speed. L1 instruction Cache stores the frequently used instruction, and it will be 32 bit in space. L1 Data Cache , 32-bit, stores frequently accessed data.

L2 Unified Cache (Unified meaning it stores both instructions and data). It will have a space of 256kb. It will be located outside the CPU, thus making it slower than L1 used to capture recent data access from the processor that was not caught by L1.

Main memory should be 2Gb to temporarily store data and instructions that the CPU is actively using, providing fast access for processing.

This type of memory hierarchy design is both cheaper and more efficient.

We will choose little endian over big endian. This is for making sure that software, hardware or systems design adheres to specified rules for how it should be specified rules for how it should be built (Ensures architectural compliance).

Advantages

* Allows ease of transmission and manipulation of bits.
* Ease of architectural operations

1. **Conclusion**

In conclusion, this project shows how a customized RISC-V RV32I processor can make technology more accessible, efficient, and affordable for everyday use in Lesotho. By focusing on specific needs like voice-driven financial transactions, signal strength prediction, and voice authentication, the design supports real-world applications that improve convenience and digital inclusion. The processor’s custom instructions and extensions allow it to handle lightweight AI and audio processing tasks efficiently, without the high power or cost of complex systems. Overall, this approach demonstrates how open, flexible hardware can help build smart, secure, and locally relevant technologies that bring Lesotho closer to its 4IR goals.

1. **References**
2. Setetemela, K.O. CS3520 Computer Organisation and Architecture I. Section 5 - Processor Design and Implementation. Department of Mathematics & Computer Science, National University of Lesotho.
3. Harris, S. (2021). DDCA Ch7 - Part 3: RISC-V Single-Cycle Processor Datapath: Extending Instructions. YouTube.