PROJECT PHASE 2 DOCUMENTATION

Pipelined Implementation of 32bit RISC-V instruction execution

CS204

Computer Architecture

INTRODUCTION:-

RISC-V architecture is a popular and open-source instruction set architecture (ISA) known for its simplicity, modularity, and extensibility. Pipelining is a technique used in modern processors to enhance performance by allowing multiple instructions to be executed simultaneously in different stages such as fetch, decode, execute, memory access, and write-back.The project aimed to implement a pipelined execution approach for 32-bit RISC-V instructions to improve the performance of RISC-V processors.The pipeline is designed to efficiently fetch instructions from memory, decode them to determine the operation, execute them to compute the result, and store the results back to the appropriate destination.

The code has been written in c++ programming language.

INPUT:-

Input to the simulator is MEM file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space.   For example:

```
0x50 0x00000093

0x4A 0x00100113

0x8C 0x00A00193
```

OUTPUT:-

Output.txt file is the output file. This file shows the values in Registers, Values at the memory addresses.

Data structure:-

Data structures play an important role in pipelined implementation to facilitate efficient data handling and processing within the pipeline stages. In a pipelined implementation, variables can refer to different types of data elements that are used for various purposes within the pipeline stages. Here are some examples of variables commonly used in a pipelined implementation:-

1. **Instruction fetch variables:-** These variables may include the instruction address, instruction buffer, and program counter (PC) to keep track of the current instruction being fetched from memory, and to store the fetched instruction for further processing in subsequent pipeline stages. Such as PC, loop, inst, immI etc.

2. **Instruction decode variables:-** These variables may include the opcode, operands, and other relevant information extracted from the fetched instruction during the instruction decode stage. These variables are used to decode the instruction and determine the type of operation to be performed. Such as opcode, funct7, funct3 etc.

3. **Register variables:-** These variables may include the source and destination registers for instructions that involve register operations, such as register-to-register data transfers or arithmetic and logical operations. These variables are used to track the operands and results of instructions that operate on registers. Such as rd, rs1, rs2 etc.

4. **Buffer variables:-** These variables may include buffers used for temporary storage of data between pipeline stages. For example, variables to hold data waiting to be written back to memory or registers, or temporary data that needs to be passed between pipeline stages, such as data being processed by an arithmetic unit or a data cache.

5. **Control variables:-** These variables may include flags, counters, or other control signals used to manage the flow and control of instructions through the pipeline. For example, variables to track the status of instruction execution, handle stalls or hazards, manage branch instructions, or control the pipeline flush operations. Such as BranchTargetSelect, ResultSelect, RFWrite, ALUOperation, OP2Select, MemOp etc.

6. **Prediction variables:-**These variables may include data used for branch prediction, such as history tables, target buffers, and counters, to predict the outcome of branch instructions and determine the next instruction to fetch. Such as IsBranch, branchOperation, branchAdd etc.

SIMULATOR FLOW:-

A pipelined simulator is a type of computer simulation that uses a pipeline architecture to execute instructions in parallel, resulting in increased performance and throughput. The flow of a pipelined simulator typically involves several stages, each dedicated to a specific step in the instruction execution process.
Here some important stages:-

1. **Instruction Fetch:-** In this stage, the simulator fetches the next instruction from the memory or instruction cache. The instruction is typically fetched from the address pointed to by the program counter (PC), which is a register that keeps track of the address of the next instruction to be executed.

2. **Instruction Decode:-** In this stage, the fetched instruction is decoded to determine the type of instruction and its operands. This stage involves parsing the instruction and identifying the opcode, registers, and immediate values involved in the instruction. The decoded information is used in subsequent stages for instruction execution.

3. **Execution:-** In this stage, the decoded instruction is executed. The ALU (Arithmetic Logic Unit) performs arithmetic or logical operations on the operands, and the result is stored in a temporary register. This stage may also involve memory access for load/store instructions.

4. **Memory Access (MEM):-** In this stage, memory operations, such as load or store instructions, are performed. The memory access stage may involve reading data from or writing data to the main memory or data cache.

5. **Write Back:-** In this stage, the results of the executed instruction are written back to the appropriate registers or memory locations. For example, the result of an arithmetic operation may be written back to a register file, while the result of a load instruction may be written back to a register or memory location.

6. **Branch:-** In this stage, branch instructions are evaluated to determine if a branch is taken or not. If a branch is taken, the PC is updated to the target address of the branch instruction, and the pipeline is flushed to start fetching and executing the new instruction sequence from the target address.

7. **Hazard Detection:-** In this stage, hazards, which are dependencies between instructions that can cause incorrect results or stalls in the pipeline, are detected. Common hazards include data hazards, where an

instruction depends on the result of a previous instruction that is not yet available, and control hazards, where the outcome of a branch instruction is not yet known.

DATA HAZARD:-

Data hazards, also known as data dependencies, are situations where the correct execution of an instruction depends on the availability of data from a previous instruction that has not yet completed its execution. Data hazards can occur in a pipelined processor when multiple instructions are being executed concurrently in different stages of the pipeline.

Data hazards can arise in three forms:-

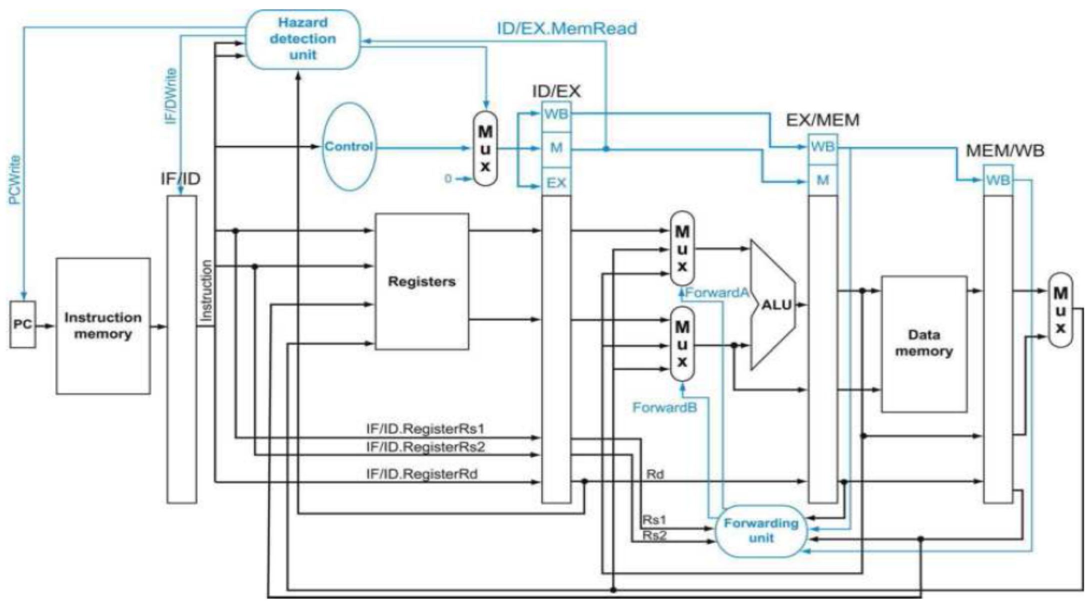
1. **Read-after-Write (RAW) hazard:-** This occurs when an instruction tries to read data from a register or memory location that has been written by a previous instruction, but the write has not yet been completed. This can result in the instruction reading stale or incorrect data, leading to incorrect results.

2. **Write-after-Read (WAR) hazard:-** This occurs when an instruction tries to write data to a register or memory location that is being read by a subsequent instruction. This can result in the subsequent instruction using incorrect or inconsistent data, leading to incorrect results.

3. **Write-after-Write (WAW) hazard:-** This occurs when two or more instructions try to write data to the same register or memory location at the same time or overlapping time periods. This can result in the data being written in an incorrect order, leading to inconsistent or incorrect results.

Methods to handle data hazards:-

rHandling data hazards is an important aspect of processor design to ensure correct and efficient execution of instructions in a pipelined implementation. Data hazards occur when an instruction depends on the result of a previous instruction that has not yet been computed or written back to the register file. This can lead to incorrect results or stalls in the pipeline, resulting in reduced performance.



Methods

1. **Stalling:-** When a data hazard is detected, the pipeline can be stalled by inserting NOP (No Operation) instructions to delay the execution of subsequent instructions until the required data is available.

2. **Forwarding:-** Forwarding, also known as bypassing, involves transmitting the result of an instruction directly to subsequent instructions that depend on that result, bypassing the need to wait for it to be written back to the register file.

3. **Compiler Optimizations:-** Compiler optimizations can be applied at the software level to rearrange instructions or use different registers to avoid data hazards.

CONTROL HAZARDS:-

Control hazards, also known as branch hazards or control flow hazards, occur when the outcome of a conditional branch instruction is not known until the instruction has reached the later stages of the pipeline. This can result in incorrect execution or stalls in the pipeline, leading to reduced performance.

HANDAL CONTROL HAZARD:-

Handling control hazards is an important aspect of processor design, and here are some techniques :-

1. **Branch prediction:-** Branch prediction is a technique used to predict the outcome of a branch instruction before it is actually resolved. Predicted outcomes are used to speculatively fetch and execute instructions along the predicted path, even before the actual outcome is known. This helps to minimize the pipeline stalls and maintain instruction throughput. There are different types of branch predictors, such as static, dynamic, and tournament.

2. **Branch Target buffer :-** A BTB is a cache that stores the target addresses of recently executed branch instructions. When a branch instruction is fetched, the BTB is checked for a match, and if found, the target address is used to fetch instructions from the predicted target path, reducing the stalls caused by control hazards.

3. **Delayed branching:-** In this technique, the instruction following a branch instruction is allowed to execute regardless of the outcome of the branch. The result of the branch is then used to either commit or discard the instructions fetched speculatively along the predicted path. If the branch is predicted correctly, the pipeline continues without stalls. Otherwise, the pipeline is flushed, and instructions from the correct path are fetched.

4. **Pipeline flushing:-** When a control hazard is detected, such as a mispredicted branch, the pipeline may need to be flushed, meaning that all instructions in the pipeline after the mispredicted branch are discarded, and the correct instructions are fetched and processed again. This can introduce a delay in the instruction execution, but it ensures correct program execution.

Test

We test the simulator with following assembly programs:

1. Fibonacci Program
2. Sum of the array of N elements.
3. Bubble Sort Program.

Group members:

1. SUSHIL KUMAR (2021CSB1136)
2. GYANENDRA MANI(2021CSB1090)
3. RAKESH MEENA (2021CSB1126)