Report on Instruction Pipeline Simulator

May 17, 2024

1 Introduction

This report presents an analysis of the Instruction Pipeline Simulator, a software tool developed to simulate the behavior of a simplified processor architecture. The simulator is capable of executing a set of predefined instructions stored in memory and provides insights into the operation of the processor at each clock cycle.

2 Methodology

The methodology employed in this project involves the design and implementation of a pipelined processor simulator using the C programming language. The key components of the methodology include:

2.1 Instruction Set Architecture (ISA) Design

The project starts with defining an Instruction Set Architecture (ISA) comprising a set of instructions, registers, and memory organization. The ISA determines the operations supported by the processor and the format of instructions.

2.2 Program Development

The simulator is developed using the C programming language to create a functional model of a pipelined processor. The implementation includes the following steps:

- **Instruction Encoding and Decoding:** Instructions are encoded into binary format based on the ISA. The simulator includes functions to decode binary instructions into meaningful operation codes and operands.
- **Instruction Memory:** An array is used to represent the instruction memory of the simulated processor. Instructions are loaded into memory from an external file using file I/O operations.

- **Execution Pipeline:** The simulator implements a basic pipeline with fetch, decode, and execute stages. Each stage is represented by a separate function, allowing for modular design and easy modification.
- **Register File and Data Memory:** Arrays are used to simulate the register file and data memory of the processor. These memory elements store intermediate results and data accessed by instructions.
- **Control Logic:** The simulator includes logic to control the flow of instructions through the pipeline, manage hazards, and update the processor state.

2.3 Pipeline Simulation

The main focus of the project is on simulating the operation of a pipelined processor. The simulator iterates over the instruction memory, simulating the fetch, decode, and execute stages for each instruction. It tracks the state of the processor, including register values, program counter, and flags, at each clock cycle.

2.4 Branch and Jump Hazard Handling

To accurately model pipeline hazards, the simulator detects and handles branch and jump instructions. When a branch or jump condition is encountered, the simulator flushes the pipeline to prevent the execution of incorrect instructions. It updates the program counter (PC) to the target address specified by the branch or jump instruction.

2.5 Results Analysis

After simulating the execution of all instructions, the simulator displays the final state of the processor, including register values and any flags set during execution. The results are analyzed to verify the correctness of the pipeline simulation and identify any discrepancies or performance bottlenecks.

2.6 Software Tools

The project utilizes various software tools for development and testing, including text editors for writing code, compilers for building the simulator, and debuggers for identifying and fixing errors. Version control systems are used to manage code revisions and facilitate collaboration among team members.

2.7 Fetch

The Fetch stage is responsible for retrieving instructions from memory. In the simulator, the fetch function reads the instruction pointed to by the program counter (PC) from the instruction memory array. It then increments the program counter to prepare for the next instruction fetch. This stage is essential for supplying instructions to the pipeline for subsequent processing.

2.8 load instruction from file

change the assembly code from instruction file to binary and put it in the memory.

2.9 Decode

During the Decode stage, fetched instructions are decoded to determine the operation to be performed and the operands involved. In the simulator, the decode function takes the fetched instruction as input and extracts the opcode along with its operands. The opcode is used to identify the type of operation to be executed, while the operands specify the registers or memory locations involved in the operation. This stage prepares the instruction for execution by mask first 4 bits in opcode , second 6 bits for operand 1 , third 6 bits in operand 2.

2.10 Execute

The Execute stage is where the actual computation specified by the instruction takes place. In the simulator, the execute function performs the operation specified by the opcode using the provided operands.there is a condition that checks that if the opcode , operand 1 and operand 2 equal 0 then there was a branch or jump and will be not executed.This stage includes arithmetic operations (such as addition, subtraction, and multiplication), logical operations (such as AND, OR), memory access operations (such as load and store), and control flow operations (such as branch and jump). Additionally, the execute stage is responsible for updating the processor's state, including register values and status flags.

2.11 Update Flags

After executing an instruction, the Update Flags stage updates the status flags of the processor based on the result of the operation. Status flags provide information about the outcome of arithmetic and logical operations, such as whether the result is zero, negative, or has caused overflow or carry. In the simulator, the updateFlags function calculates and updates these flags according to the result of the executed instruction. This information is crucial for subsequent conditional branching and decision-making within programs.

3 Results

The simulator was tested with various instruction sequences to evaluate its performance and accuracy. The results of these tests demonstrate the following key findings:

- **Correctness**: The simulator accurately executes instructions according to the predefined opcode definitions. Results of arithmetic and logical operations are consistent with expected behavior.
- **Pipeline Hazards**: Branch and jump instructions introduce pipeline hazards that require flushing the pipeline to maintain program correctness. The simulator handles these hazards by discarding partially executed instructions and adjusting the program counter accordingly.
- **Flag Updates**: Status flags such as zero, sign, carry, and overflow are correctly updated based on the results of executed instructions. This allows for effective condition checking and branching within programs.

Overall, the Instruction Pipeline Simulator demonstrates efficient instruction execution and flag management, making it a valuable tool for understanding and analyzing processor behavior.

4 Conclusion

In conclusion, the Instruction Pipeline Simulator provides a comprehensive platform for simulating the execution of instructions in a pipelined processor architecture. Its modular design, accurate instruction execution, and effective handling of pipeline hazards make it a useful tool for educational purposes and processor performance analysis.