Report on Modified Pipelined MIPS Processor

with Delayed Branches and Multi-Cycle Memory Access

Overview

This project simulates a 5-stage pipelined MIPS processor in Python with two key architectural enhancements:

1. Delayed Branch Execution using branch delay slots.
2. Multi-Cycle Memory Access for load/store instructions.

The simulation offers a detailed and modular view of pipelined MIPS behavior while introducing realistic execution delays. It provides timing tables, performance statistics, and accurately tracks stalls, memory latency, and branch delay effectiveness.

Pipeline Architecture

The processor uses a classic 5-stage MIPS pipeline:

- IF (Instruction Fetch): Fetches the instruction from memory.
- ID (Instruction Decode): Decodes the instruction and reads the required registers.
- EX (Execute): Performs arithmetic/logic operations or calculates addresses.
- MEM (Memory Access): Loads from or stores to memory.
- WB (Write Back): Writes results back to the register file.

Pipeline registers used for inter-stage communication: IF_ID, ID_EX, EX_MEM, MEM_WB.
Each stage reads data from the previous register and writes to the next, simulating stage-by-stage propagation.

Architectural Enhancements

1. Delayed Branch Execution:

- Implements branch delay slots by always executing the instruction following a branch.
- This avoids flushing the pipeline and improves instruction throughput.
- The simulator tracks how often this delay slot is utilized effectively (branch_delay_effective) vs total branch instructions (branch_delay_total).

2. Multi-Cycle Memory Access:

- Simulates variable latency in the MEM stage by randomly choosing 2 or 3 cycles.
- Load/store instructions cause stalls in the pipeline until memory access is complete.
- The EX stage is stalled when MEM is busy.
- This is implemented by tracking a mem_busy_cycles counter, managed via memory.py.

Data Path and Control Changes Made

Several modifications were made to simulate the delayed branching and multi-cycle memory functionality:

- Control Logic Adjustments: Identifies branch instructions and ensures the delay slot instruction is executed; checks memory latency and introduces stalls.
- Pipeline Stall Mechanism: Freezes pipeline registers when a stall is necessary; hazard detection prevents incorrect execution.
- Memory Access Logic: Simulates variable memory latency using random or fixed delay; mem_busy_cycles tracks ongoing memory ops.
- Statistics and Logging: Counters and logs added for measuring effectiveness and delay impact; enhanced stage-wise logging.

Code Breakdown by Module

main.py:

- Parses .asm input using parser module.
- Initializes register file and memory.
- Maintains pipeline stages and registers.
- Handles stalling and logs per cycle execution.
- Gathers performance statistics.

parser.py:

- Converts .asm to instruction list.
- Resolves labels and translates pseudoinstructions.

Report on Modified Pipelined MIPS Processor

with Delayed Branches and Multi-Cycle Memory Access

registers.py:

- Maintains and updates 32 general-purpose registers.
- Ensures $zero always reads as 0.

memory.py:

- Implements instruction and data memory.
- Simulates 2/3 cycle latency using random delays.
- Models memory as a sparse dictionary.

Sample Assembly Program

Includes loads, stores, branches, and arithmetic instructions.

Tests realistic behavior: delay slots, load-use hazards, multi-cycle memory.

Ensures correctness and pipeline order maintenance.

Performance Metrics (Reported by Simulation)

- Total Clock Cycles: Time taken to complete the program.
- Total Instructions Executed: Includes delay slot instructions.
- Stalls due to Loads: Cycles stalled waiting for load.
- Cycles Wasted due to Memory Delays: Memory delay overhead.
- Branch Delay Slot Effectiveness: Useful delay slot instructions executed.

Observations and Effects of Enhancements

- Delayed Branching: Saves flushes, improves throughput; requires scheduling.
- Multi-Cycle Memory: Simulates hardware delays accurately; stalls maintain correctness.
- Pipeline Hazards: Load-use stalls present, forwarding not implemented.
- Instruction Scheduling: Optimized code minimizes stalls and delay waste.

Possible Enhancements

- Add data forwarding to reduce stalls.
- Implement compiler scheduling for better delay slot usage.
- Add GUI-based visualization for pipeline simulation.

Conclusion

This project demonstrates a pipelined MIPS processor simulation with delayed branch execution and multi-cycle memory access. It offers insight into performance bottlenecks and execution timing, providing a foundation for advanced enhancements.

By modeling real-world behavior and tracking instruction progress, the simulator aids in understanding processor design and pipeline management challenges.

Prepared as part of the Computer Architecture Course Project

Contributors: Rohan K and Manvith

Institution: International Institute Of Information Technology - Bangalore