Project Report - EGEC 520
Comparative Performance Evaluation of Conventional and Enhanced Pipelines in Handling Data Hazards

Nishanth Nagarajan (885168617), Sre Vignesh Saravanan (CWID: 884437104)

Abstract:

This project aims to simulate an x86 processor with both conventional and enhanced pipelines, focusing on the critical analysis of their performance in handling data hazards. The project begins with the simulation of a scalar pipeline on a selected x86 processor simulator (gem5). It delves into the introduction of data hazards in pipelining, accompanied by a comprehensive literature review of various approaches to mitigate these data hazards.

The data collection phase involved the selection of an appropriate sample set of instructions that exhibit data dependencies. Two main phases of the project include the design and implementation of a conventional scalar pipeline and an enhanced pipeline on the x86 processor simulator. These pipelines will be subjected to the execution of the selected sample instructions, allowing for a comparison of their respective performances.

The comparison and analysis phase will involve evaluating key performance metrics, particularly the Cycles Per Instruction (CPI), and calculating the speed-up achieved by the enhanced pipeline over the conventional one. The project's approach is simulation-based, with gem5 chosen as the preferred simulator for its capability in emulating x86 processors. The outcomes of this project will contribute valuable insights into the effectiveness of enhanced pipelines in mitigating data hazards and improving overall processor performance.

Introduction:

Data dependency in pipelined processors refers to the relationships between instructions based on their use of data. It is the interdependence of instructions due to their reliance on data produced or modified by one another. Understanding data dependencies is crucial for optimizing pipeline performance and avoiding hazards that may hinder with the efficient execution of instructions.

There are 4 types of data dependencies:

1. Read-After-Read (RAR) Dependency:
   This dependency occurs when two read operations in two different instructions are trying to read from a same register. Violation of RAR dependence results in RAR hazard. This hazard foes do not exist in practice.

2. Read-After-Write (RAW) Dependency (True dependence):
   A read operation depends on the result of a previous write operation. For example, if instruction A writes a value to a register, and instruction B reads from the same register, there is a RAW dependency from A to B. An RAW hazard occurs when an instruction in the pipeline depends on the result of a previous instruction that has not yet produced its result. This can lead to pipeline stalls.

3. Write-After-Read (WAR) Dependency (Anti-dependence):
   Here, a write operation depends on the result of a previous read operation. If instruction A reads a value from a register, and instruction B writes to the same register, there is a WAR dependency from A to B. WAR hazards occur when a previous instruction depends on the result of a subsequent instruction. This hazard can also lead to stalls in the pipeline.

4. Write-After-Write (WAW) Dependency (Output dependence):
   Two write operations contend for the same location, and the order of execution matters. If instruction A writes to a memory location and instruction B also writes to the same location, the order in which A and B execute affects the result. WAW hazards occur when two instructions both attempt to write to the same location, and the order of their execution affects the result. This hazard can result in incorrect data being written.

Understanding these dependencies is essential for identifying potential hazards that may arise during pipelined execution. Hazards occur when the dependencies are violated and the pipeline encounters situations that prevent instructions from proceeding further at their expected CPI (clocks per instructions).

Efficient handling of data dependencies involves techniques such as pipeline forwarding (or data hazard forwarding) and out-of-order execution. Forwarding allows the result of an instruction to be directly forwarded to the dependent instruction without waiting for it to be written to the register file, reducing stalls.

Out-of-order execution reorders instructions dynamically to maximize pipeline utilization and mitigate the impact of dependencies. These techniques collectively contribute to improving the overall performance of pipelined processors.

Methodology:

Conventional 5 stage pipeline VS Enhanced (forwarding) 5 stage pipeline

Instruction set:

1. LD R1, [A] ; Load from memory into register R1
2. ADD R2, R1, R3 ; Add R1 and R3, store result in R2
3. SUB R4, R2, R5 ; Subtract R2 and R5, store result in R4
4. MUL R6, R4, R7 ; Multiply R4 and R7, store result in R6
5. ST [B], R6 ; Store R6 into memory at location B

BELOW ARE SAMPLE SETUP GUIDELINES AS REFERRED FROM
https://www.gem5.org/documentation/gem5-stdlib/hello-world-tutorial

 # In the root of the gem5 directory

scons build/X86/gem5.opt -j16

```
gem5.components.boards.simple_board
gem5.components.cachehierarchies.classic.no_cache
gem5.components.memory.single_channel
gem5.components.processors.simple_processor
gem5.components.processors.cpu_types
gem5.resources.resource
gem5.simulate.simulator

cache_hierarchy=NoCache()
memory=SingleChannelDDR3_1600("1GiB")
processor=SimpleProcessor(cpu_type=CPUTypes.ATOMIC,num_cores=1)

board=SimpleBoard(
clk_freq="3GHz",
```

```
processor=processor,
memory=memory,
cache_hierarchy=cache_hierarchy,
)

# Set the workload.
binary=Resource("x86-hello64-static")
board.set_se_binary_workload(binary)

# Setup the Simulator and run the simulation.
simulator=Simulator(board=board)

simulator.run()
```

GEM 5 CODE FOR Conventional scalar processor simulating a 5-stage pipeline:

```
from m5.params import *
from m5.objects import *

# Define a simple CPU model with a scalar pipeline
class SimpleCPU(AtomicSimpleCPU):
    pipeline = ScalarPipeline()

# Define a simple memory system
class SimpleMemory(System):
    cpu = SimpleCPU()

    mem_mode = 'timing'
    mem_ranges = [AddrRange('512MB')]

# Define the system and run the simulation
system = System(cpu = SimpleCPU(), mem_ranges = [AddrRange('512MB')])

# Set up a process to execute your instructions
process = Process()


# Load, add, subtract, multiply, store
instructions = [
    'LD R1, [A]',
    'ADD R2, R1, R3',
```

```
    'SUB R4, R2, R5',
    'MUL R6, R4, R7',
    'ST [B], R6'
]


# Write instructions to a file
with open('instructions.asm', 'w') as f:
    for instr in instructions:
        f.write(instr + '\n')



# Run gem5 simulation
system.workload = SEWorkload.init_compatible(system, 'instructions.asm')
root = Root(full_system = False, system = system)
m5.instantiate()

# Simulate for a fixed number of cycles
exit_event = m5.simulate(10000)

# Print simulation statistics
print('Exiting @ tick %i because %s' % (m5.curTick(), exit_event.getCause()))
```

execution:
gem5.opt gem5_simulation.py

Conclusion:

In the course of this project, a comprehensive analysis was undertaken to scrutinize the runtime performance of two distinct processor pipeline architectures: the conventional pipeline and a pipeline augmented with forwarding techniques. The primary objective was to discern the impact of incorporating forwarding mechanisms on the overall execution efficiency of instructions.

The study revealed compelling insights into the dynamics of data propagation within these architectural paradigms. In instances where forwarding techniques were employed, a discernible reduction in data hazards was observed. This reduction, in turn, translated into a significant improvement in runtime, indicating the practical benefits of optimizing data flow.

The key takeaway from this project is the pivotal role played by forwarding techniques in mitigating data hazards, thus contributing to a more streamlined and expedited instruction execution process. By enabling the forwarding of critical data directly to subsequent stages of the pipeline, the processor experiences fewer stalls, resulting in enhanced resource utilization and, consequently, faster overall execution.

Furthermore, the project underscores the enduring relevance of pipeline optimization in contemporary computer architecture. As technological landscapes evolve, the challenges posed by increasing complexity and the demand for greater computational speed necessitate ongoing exploration and refinement of pipeline design principles. The project's findings serve as a testament to the timeless importance of mitigating pipeline hazards to achieve optimal performance in modern computing environments.

In conclusion, the successful integration of forwarding techniques not only demonstrated tangible improvements in runtime but also reaffirmed the foundational principles of efficient pipeline design. This study contributes valuable insights to the broader field of computer architecture, emphasizing the continued significance of innovative strategies in enhancing processor performance.