Advanced Micro-Architecture

Introduction

In modern computer architecture, improving processor performance is one of the most important design objectives. As software applications continue to grow in size and complexity, processors are required to execute a very large number of instructions efficiently and at high speed. Simply increasing hardware resources such as adding more transistors or increasing memory size is no longer sufficient. Instead, computer architects rely on advanced micro-architectural techniques to improve performance. Among these techniques, pipelining, deep pipelining, and branch prediction play a central role. These concepts are strongly interconnected. Pipelining improves instruction throughput, deep pipelining increases clock frequency, and branch prediction is essential to manage the control hazards introduced by deep pipelines. This document explains these concepts in a detailed and continuous manner so that students can understand both the theory and the practical motivation behind them.

Instruction Pipelining

Instruction pipelining is an implementation technique that allows multiple instructions to be overlapped during execution. Instead of completing one instruction fully before starting the next, pipelining divides the execution of instructions into a series of stages, with each stage performing a specific part of the instruction cycle. This concept is often compared to an assembly line in a factory. In an assembly line, the construction of a product is divided into multiple stages, and each stage performs one specific task. As a result, many products can be in different stages of production at the same time. Similarly, in a pipelined processor, many instructions are processed simultaneously, each at a different stage of execution.

In most classical RISC-based processors, the instruction execution cycle is divided into five main stages. The first stage is Instruction Fetch (IF), where the processor fetches the instruction from instruction memory using the Program Counter, which holds the address of the next instruction. The second stage is Instruction Decode (ID), where the fetched instruction is decoded and the required registers are read from the register file. The third stage is Instruction Execute (EX), where arithmetic or logical operations are performed, or where branch conditions and memory addresses are calculated. The fourth stage is Memory Access (MEM), where data memory is accessed if the instruction is a load or store operation. Finally, the Write Back (WB) stage writes the result of the instruction back into the destination register.

These stages can be represented as:

IF → ID → EX → MEM → WB

In a pipelined processor, these stages operate in parallel on different instructions. While one instruction is being executed in the EX-stage, another instruction may be in the ID stage, and a third instruction may be in the IF stage. This overlapping of execution increases throughput, which means that the processor can complete more instructions per unit time.

Performance of Pipelining

The main objective of pipelining is to improve throughput rather than reducing the latency of a single instruction. Latency refers to the time taken by one instruction to complete from start to finish, whereas throughput refers to the number of instructions completed per unit time.

Ideally, if an instruction execution process is divided into k pipeline stages, the time per instruction in a pipelined processor can be approximated as the time of a non-pipelined processor divided by k. This means that, in ideal conditions, pipelining can provide a speedup proportional to the number of stages.

The speedup achieved by pipelining can be expressed as the ratio of execution time without pipelining to the execution time with pipelining. As the number of instructions executed becomes very large, the speedup approaches the number of pipeline stages. However, in practical systems, hazards and pipeline overheads reduce the actual speedup.

Pipeline Hazards

Pipeline hazards are situations that prevent the next instruction in the pipeline from executing during its designated clock cycle. These hazards reduce the ideal performance of a pipelined processor.

Structural hazards occur when the hardware cannot support multiple instructions at the same time. For example, if the processor has a single memory unit that is shared by both instruction fetch and data access, a conflict may occur when both stages need memory access simultaneously.

Data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed its execution. A common example is a Read-After-Write dependency, where an instruction tries to read a register before the previous instruction has written the correct value into it. These hazards are typically handled using techniques such as forwarding (also known as bypassing) or stalling the pipeline.

Control hazards are caused by branch instructions. When a branch instruction is encountered, the processor does not immediately know which instruction should be fetched next. This uncertainty disrupts the normal flow of the pipeline and becomes a major problem in deeper pipelines.

Deep Pipelining (Super-pipelining)

Deep pipelining, also known as super-pipelining, is an advanced form of pipelining in which the number of pipeline stages is significantly increased. Instead of using a simple five-stage pipeline, deep pipelined processors may use ten, fifteen, twenty, or even more stages. A well-known example is the Intel Pentium 4 processor, which used a very deep pipeline to achieve high clock frequencies.

In deep pipelining, the original pipeline stages such as IF, ID, EX, MEM, and WB are further subdivided into smaller sub-stages. For example, the instruction fetch stage may be divided into IF1 and IF2, instruction decode into ID1 and ID2, and execution into EX1 and EX2. A conceptual representation of a deep pipeline can be shown as:

IF1 → IF2 → ID1 → ID2 → EX1 → EX2 → MEM1 → MEM2 → WB1 → WB2

The main motivation behind deep pipelining is to reduce the amount of work performed in each stage. When each stage performs a smaller amount of work, the delay of that stage is reduced. Since the clock cycle time is determined by the slowest stage in the pipeline, reducing the stage delay allows the processor to operate at a much higher clock frequency. As a result, more instructions can be processed per second, increasing throughput.

Trade-offs and Problems of Deep Pipelining

Although deep pipelining allows processors to achieve very high clock speeds, it introduces significant trade-offs. One major drawback is increased instruction latency. Because an instruction must pass through a larger number of stages, the total time taken for a single instruction to complete increases.

Another serious issue is the increased penalty caused by hazards. In deep pipelines, many more instructions are in flight at the same time. If a stall or hazard occurs, a larger number of instructions are affected. This problem is especially severe in the case of control hazards.

Branch misprediction becomes extremely costly in deep pipelines. If a branch is predicted incorrectly in a pipeline with twenty or thirty stages, the processor may need to flush all these stages. This means that many partially executed instructions are discarded, wasting a large number of clock cycles. This wasted time is known as the branch misprediction penalty, and it increases as the pipeline depth increases.

Branch Prediction and Its Importance

Branch prediction is a technique used to reduce the performance loss caused by control hazards. Instead of waiting for the branch decision to be resolved, the processor predicts whether the branch will be taken or not taken and continues fetching instructions accordingly. If the prediction is correct, the pipeline continues smoothly without interruption. If the prediction is incorrect, the incorrectly fetched instructions are flushed, and execution resumes from the correct path.

In deep pipelined processors, branch prediction is not optional; it is essential. Without branch prediction, the processor would have to stall the pipeline frequently, resulting in poor performance. Accurate branch prediction allows deep pipelines to operate efficiently by minimizing stalls and reducing the impact of control hazards.

Control Hazards and the Branch Problem

A control hazard occurs when the processor encounters a branch instruction that can change the normal sequential flow of execution. Branch instructions are very common in programs and appear in conditional statements, loops, and function calls. When a branch instruction is fetched into the pipeline, the processor does not immediately know which instruction should be executed next because the branch decision depends on the result of a comparison that is resolved later in the pipeline.

In a shallow pipeline, the branch decision is resolved relatively quickly, and only a small number of instructions may need to be discarded if the wrong path is taken. In a deep pipeline, however, the branch decision may be resolved much later, after many pipeline stages. During this time, the

processor continues fetching and partially executing instructions based on an assumed path. If this assumption turns out to be incorrect, all these instructions must be flushed from the pipeline. This flushing process wastes many clock cycles and significantly reduces performance. The number of wasted cycles due to a wrong branch decision is known as the branch penalty, and it increases as the pipeline becomes deeper.

Branch Prediction

Branch prediction is a technique designed to reduce the performance loss caused by control hazards in pipelined processors, especially those with deep pipelines. Instead of waiting for the branch instruction to be fully resolved, the processor predicts the outcome of the branch in advance. Based on this prediction, the processor continues fetching and executing instructions from the predicted path. If the prediction is correct, the pipeline continues smoothly without interruption. If the prediction is incorrect, the processor must discard the incorrectly executed instructions and restart execution from the correct path.

The primary goal of branch prediction is to keep the pipeline full and busy. In deep pipelines, stalling the pipeline until the branch decision is known would result in a large number of idle cycles, severely reducing performance. By predicting branch outcomes, the processor avoids unnecessary stalls and maintains high instruction throughput.

Static Branch Prediction

Static branch prediction techniques make predictions without using runtime information. The prediction strategy is fixed and does not change during program execution. One simple static approach is to always predict that a branch will not be taken, allowing the processor to continue executing the next sequential instruction. Another approach is to always predict that the branch will be taken. Some static methods use simple rules, such as predicting backward branches as taken because they are commonly associated with loops.

Static branch prediction has the advantage of being simple and requiring minimal hardware. However, its accuracy is limited because it does not adapt to the actual behavior of programs during execution. In deep pipelines, low prediction accuracy leads to frequent pipeline flushes and high performance penalties, making static prediction insufficient for modern high-performance processors.

Dynamic Branch Prediction

Dynamic branch prediction techniques use runtime information to make more accurate predictions. The processor monitors the actual outcomes of branch instructions and uses this history to predict future behavior. Over time, the predictor learns the patterns of branch behavior and improves its accuracy.

One common dynamic approach is the use of history-based predictors, such as one-bit and two-bit predictors. These predictors store information about whether a branch was taken or not taken in the past and use this information to make future predictions. More advanced dynamic predictors combine local and global branch histories to further improve accuracy.

Dynamic branch prediction significantly reduces the number of mispredictions in deep pipelines. Although it requires additional hardware and increases design complexity, the performance benefits far outweigh the costs, especially in processors with high clock speeds and deep pipelines.

Branch Misprediction and Penalty

To understand the cost of branch misprediction mathematically, it is useful to relate it to pipeline depth. When a branch is mis-predicted, all instructions that entered the pipeline after the branch must be flushed.

If the pipeline has N stages and the branch decision is resolved at stage k, then the approximate branch penalty can be expressed as:

Branch Penalty $\approx N - k$ cycles

In deep pipelines, N is large, which means the branch penalty is also large. This is why deep pipelines are highly sensitive to branch prediction accuracy.

Another important performance metric is Cycles Per Instruction (CPI). In an ideal pipelined processor, the CPI is close to 1. However, branch mispredictions increase CPI.

The effective CPI can be expressed as:

CPI = Ideal CPI + (Branch Frequency × Misprediction Rate × Branch Penalty)

This equation clearly shows that as pipeline depth increases, branch penalty increases, which in turn increases CPI if mispredictions occur. Therefore, accurate branch prediction is essential to maintain low CPI in deep pipelined processors.

A branch misprediction occurs when the predicted outcome of a branch does not match the actual outcome. When this happens, all instructions that were fetched and partially executed based on the incorrect prediction must be removed from the pipeline. The processor then fetches the correct instruction stream and resumes execution.

In deep pipelines, the penalty for a misprediction is very high because a large number of stages must be flushed. This makes accurate branch prediction essential. As pipeline depth increases, even a small reduction in prediction accuracy can lead to a significant drop in performance. Therefore, modern processors invest heavily in sophisticated branch prediction mechanisms to minimize misprediction penalties.

Relationship Between Deep Pipelines and Branch Prediction

Deep pipelining and branch prediction are tightly interconnected. Deep pipelines enable higher clock frequencies and improved throughput, but they also increase the severity of control hazards and branch penalties. Branch prediction addresses this problem by allowing the processor to make educated guesses about control flow and continue execution without waiting for branch resolution.

Without effective branch prediction, deep pipelines would spend a large amount of time stalled or flushing instructions, making them inefficient. As a result, branch prediction is not an optional feature but a fundamental requirement for the successful implementation of deep pipelined architectures.

Superscalar Processors

A superscalar processor is a processor that can issue and execute more than one instruction in a single clock cycle. In a traditional pipelined processor, even though multiple pipeline stages operate in parallel, only one instruction is issued per cycle. Superscalar processors remove this limitation by providing multiple execution units, such as multiple arithmetic logic units, load/store units, and floating-point units. The processor examines a group of instructions simultaneously and determines which instructions are independent of each other. Independent instructions are then dispatched to different execution units in the same clock cycle.

From a performance perspective, the main objective of a superscalar processor is to increase the number of instructions completed per cycle, often referred to as Instructions Per Cycle (IPC). By executing multiple instructions in parallel, the processor significantly improves throughput. However, this parallelism is limited by data dependencies, control dependencies, and hardware resource availability. Therefore, superscalar processors require complex hardware logic for instruction decoding, dependency checking, and scheduling. In examinations, students should clearly mention that superscalar execution exploits instruction-level parallelism (ILP).

Out-of-Order Execution

Out-of-order execution is a technique used to reduce pipeline stalls caused by data dependencies and long-latency operations. In an in-order processor, instructions are executed strictly in the order in which they appear in the program. If an instruction depends on the result of a previous instruction that has not yet completed, the entire pipeline must stall. Out-of-order processors avoid this inefficiency by allowing instructions that are ready for execution to proceed ahead of stalled instructions, even if they appear later in the program.

Although instructions may execute out of program order, the processor ensures that the final results are committed in the original program order. This guarantees correct program behavior and precise exceptions. Out-of-order execution improves overall performance by keeping execution units busy and minimizing idle cycles. From an exam point of view, it is important to emphasize that out-of-order execution improves utilization of hardware resources and works hand-in-hand with superscalar architectures.

Register Renaming

Register renaming is a key technique that enables efficient out-of-order execution. In programs, instructions use a limited set of architectural registers defined by the instruction set architecture. This limitation can create false dependencies between instructions, such as write-after-read (WAR) and write-after-write (WAW) hazards, even when there is no true data dependency. These false dependencies unnecessarily restrict instruction reordering.

Register renaming solves this problem by mapping architectural registers to a larger pool of physical registers inside the processor. Each time an instruction writes to a register, it is assigned a new physical register. As a result, multiple instructions that use the same architectural register

name can execute in parallel without interference. In examinations, students should mention that register renaming eliminates false dependencies and is essential for high-performance superscalar and out-of-order processors.

Single Instruction Multiple Data (SIMD)

Single Instruction Multiple Data, or SIMD, is a form of parallel processing where a single instruction operates on multiple data elements simultaneously. SIMD exploits data-level parallelism, which occurs when the same operation must be applied to a large set of similar data values. Instead of executing the same instruction repeatedly for each data element, SIMD allows the processor to process multiple elements in one operation.

SIMD is widely used in applications such as image processing, video encoding, scientific simulations, and machine learning. Modern processors include vector registers and specialized SIMD instruction sets, such as SSE and AVX in Intel architectures. From an exam perspective, it is important to note that SIMD improves performance and energy efficiency but is most effective when the program has regular, data-parallel workloads.

Multithreading

Multithreading is a technique that allows a single processor core to execute multiple threads of execution. A thread represents an independent sequence of instructions within a program. In many situations, a thread may stall due to cache misses, memory access delays, or branch mispredictions. During these stalls, processor resources may remain idle.

Multithreading improves processor utilization by allowing another thread to execute while one thread is waiting. There are different forms of multithreading, including coarse-grained, fine-grained, and simultaneous multithreading. Simultaneous multithreading allows instructions from multiple threads to be issued in the same clock cycle. Intel’s Hyper-Threading technology is a well-known example. In exams, students should emphasize that multithreading improves throughput and hides latency.

Homogeneous Multiprocessing

Homogeneous multiprocessing refers to a system that contains multiple processors or cores that are identical in terms of architecture, instruction set, and performance characteristics. All processors share the same memory and are capable of executing the same types of tasks. This is the most common form of multiprocessing found in modern multi-core CPUs.

In homogeneous multiprocessing systems, the operating system distributes processes and threads among the available processors to balance the workload and improve performance. Because all processors are identical, task scheduling and load balancing are simpler compared to heterogeneous systems. From an examination point of view, students should clearly state that homogeneous multiprocessing improves performance through parallel execution while maintaining programming simplicity.