Advanced Computer Architectures

Christian Rossi

Academic Year 2023-2024

Abstract

The course covers several key topics in computer architecture. It begins with a review of fundamental concepts, such as the RISC approach, pipelining, and the memory hierarchy. Students will learn about performance evaluation metrics for computer architectures and techniques for optimizing performance in both processors and memory. The course delves into instruction-level parallelism, discussing static and dynamic scheduling, superscalar architectures, their principles and challenges, and VLIW architectures, with examples from various architecture families. Additionally, the course covers thread-level parallelism and explores multiprocessor and multicore systems, including their taxonomy, topologies, communication management, memory management, and cache coherency protocols, with examples of different architectures. Finally, the course examines stream processors, vector processors, graphics processors, GP-GPUs, and heterogeneous architectures.

Contents

CHAPTER 1

Introduction

1.1 Computer architectures

In 1966, Michael Flynn introduced a taxonomy for categorizing the architecture of calculators. This classification divides architectures into four categories:

- *Single Instruction Single Data* (SISD): utilized by uniprocessor systems.
- *Multiple Instruction Single Data* (MISD): although theoretically possible, this architecture lacks practical implementations.
- *Single Instruction Multiple Data* (SIMD): features a straightforward programming model with low overhead and high flexibility, commonly employed in custom integrated circuits.
- *Multiple Instruction Multiple Data* (MIMD): known for its scalability and fault tolerance, this architecture is used in off-the-shelf microservices.

1.1.1 Single Instruction Single Data

The traditional concept of computation involves writing software for serial execution, typically on a single computer with a lone Central Processing Unit (CPU). In this model, tasks are divided into a sequence of discrete instructions that are executed sequentially, allowing only one instruction to be processed at any given moment.

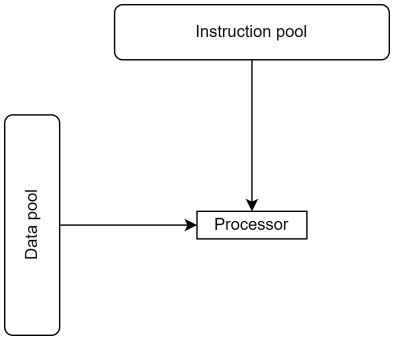


Figure 1.1: SISD architecture

In an SISD architecture, only one instruction is processed by the CPU in each clock cycle, and only one data stream is utilized as input during each clock cycle. Execution in this setup is deterministic, meaning the outcome is predictable and follows a defined sequence of steps. SISD architectures represent the most prevalent type of computers.

1.1.2 Single Instruction Multiple Data

In the SIMD architecture, all processing units execute the same instruction simultaneously during each clock cycle, but each processing unit can handle a different data element independently. This architecture is particularly well-suited for specialized problems with a high level of regularity, such as graphics and image processing.

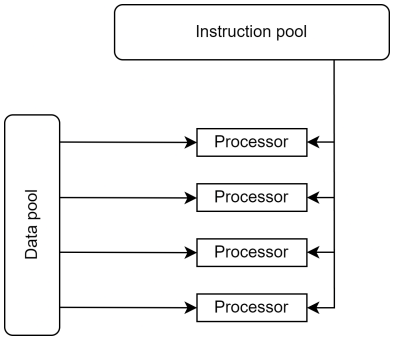


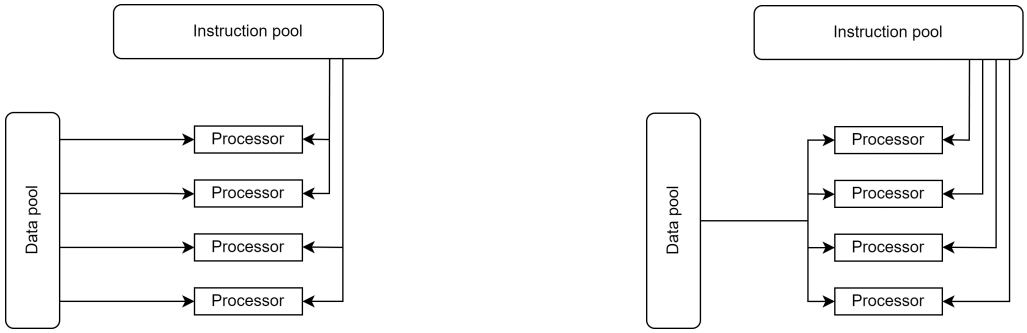
Figure 1.2: SIMD architecture

1.1.3 Multiple Instructions Multiple Data

Hardware parallelism can be achieved through various methods:

- *Instruction-level parallelism*: this method leverages data-level parallelism at different levels. Compiler techniques such as pipelining exploit modest-level parallelism, while speculation techniques operate at medium levels of parallelism.
- *Vector architectures and Graphic Processor Units* (GPU): these architectures utilize data-level parallelism by executing a single instruction across multiple data elements simultaneously.
- *Thread-level parallelism*: this approach exploits either data-level or task-level parallelism within a closely interconnected hardware model that enables interaction among threads.
- *Request-level parallelism*: this method takes advantage of parallelism among largely independent tasks specified by either the programmer or the operating system.

Currently, the most common type of parallel computer features processors that can each execute different instruction streams and operate on distinct data streams. Execution in parallel computing can occur synchronously or asynchronously, and it may be deterministic or nondeterministic.



(a) MIMD architecture (b) MISD architecture

Figure 1.3: Possible architectures for hardware parallelism

1.2 Microprocessor without Interlocked Pipeline Stages

MIPS exemplifies the principles of Reduced Instruction Set Computer (RISC) architecture, emphasizing streamlined execution through simple instructions and a condensed basic cycle. This design aims to enhance the efficiency of Complex Instruction Set Computer (CISC) CPUs.

As a load-store architecture, MIPS operates such that Arithmetic Logic Unit operands are sourced exclusively from the CPU's general-purpose registers, precluding direct retrieval from memory. Dedicated instructions are thus essential for operations between registers and main memory.

MIPS is a pipeline architecture, designed for performance optimization by enabling the concurrent execution of multiple instructions from a sequential execution flow.

Additionally, the Instruction Set Architecture (ISA) of MIPS includes a defined set of operations, instruction formats, supported hardware data types, named storage, addressing modes, and sequencing protocols.

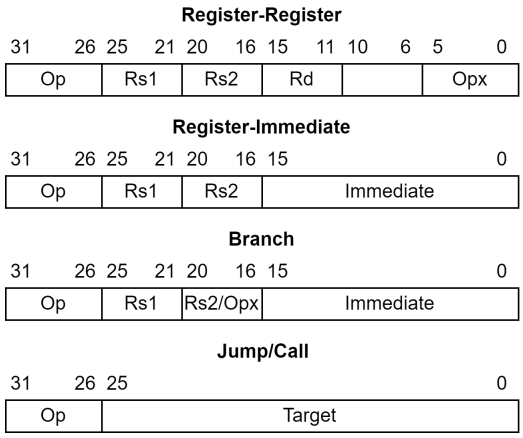


Figure 1.4: MIPS ISA

1.2.1 MIPS hardware

Within a MIPS CPU, the data path includes necessary components such as storage, Functional Units (FUs), and interconnects to effectively execute operations. In this setup, control points serve as inputs while signals serve as outputs.

The controller, functioning as a state machine, coordinates activities within the data path by directing operations based on the desired function and the signals received.

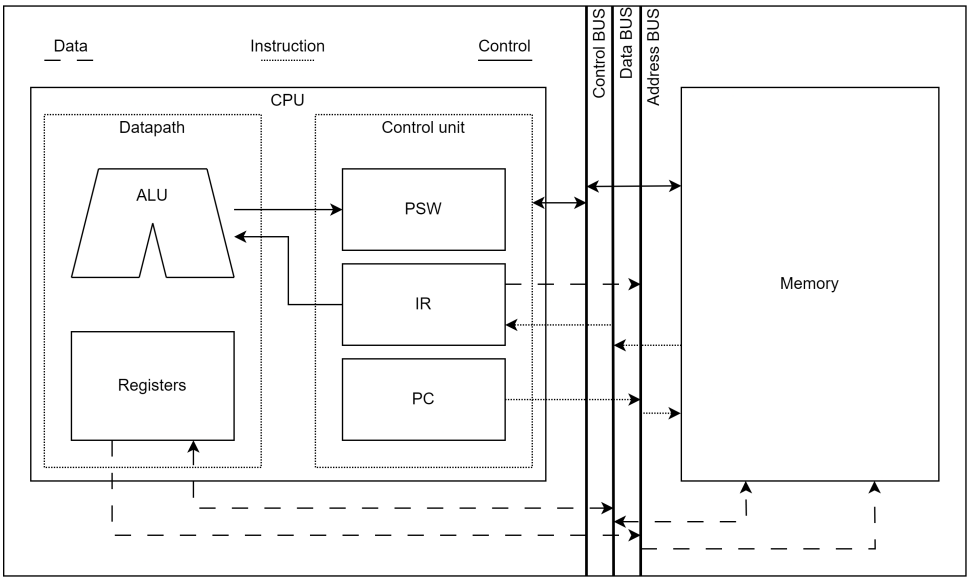


Figure 1.5: MIPS CPU

At the core, a program is segmented into instructions, with the hardware focusing on individual instructions rather than the entire program. At a lower level, the hardware divides instructions into clock cycles, with state machines transitioning states with each cycle.

1.2.2 MIPS workflow

Each instruction within the MIPS subset can be executed within a maximum of five clock cycles, as outlined below:

1. *Instruction Fetch cycle* (IF): The content of the Program Counter (PC) is transferred into the instruction memory, and the instruction to be executed is retrieved. Then, the PC is incremented by four to point to the next instruction.

2. *Instruction Decode cycle* (ID): the executing instruction is decoded, and the necessary registers are loaded. If needed, a sign extension of the offset field is performed.

3. *Execution cycle* (EX): arithmetic operations are performed between registers or between a register and an immediate value. If dealing with a branch or a memory instruction, the address with the offset is computed in this stage. For branches, the registers are compared to determine if the branch is taken or not.

4. *Memory access cycle* (MEM): for load operations, the value in memory is stored into a register, while for store operations, the register value is saved into main memory. If executing a branch, the value of the PC is updated to the branch address.

5. *Write Back cycle* (WB): the memory value is saved into the register, finalizing the load operation, and the result of arithmetic operations is saved into the register.

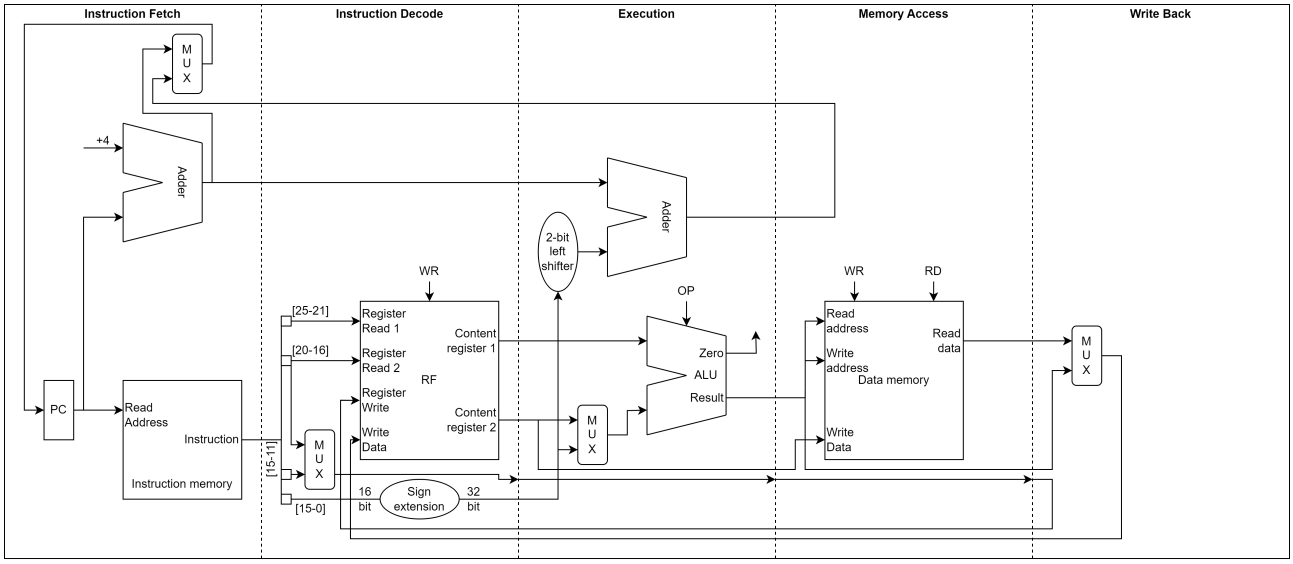


Figure 1.6: MIPS architecture

The duration of each clock cycle is determined by the critical path established by the load instruction, that is of 8 ns. We assume a single-clock cycle execution for each instruction, wherein each module is utilized once within a cycle. Modules utilized more than once within a cycle necessitate duplication for efficiency. Furthermore, to ensure separate functionality, an instruction memory distinct from the data memory is required.

Certain modules must be duplicated, while others are shared across different instruction flows. To facilitate sharing a module between two distinct instructions, a multiplexer is utilized.

In the multi-cycle implementation of CPU, the execution of instructions spans across multiple cycles, with MIPS typically utilizing five cycles. Key aspects of the multi-cycle CPU implementation include: each phase of instruction execution necessitates a clock cycle, modules can be utilized multiple times per instruction across different clock cycles, allowing for potential module sharing, and Internal registers are required to retain values for subsequent clock cycles. These registers store data to be utilized in future stages of the instruction execution process.

1.2.3 MIPS pipelining

Pipelining is an optimization method aimed at enhancing performance by overlapping the execution of multiple instructions originating from a sequential execution flow. It capitalizes on the inherent parallelism among instructions within a sequential instruction stream.

The fundamental concept involves breaking down the execution of an instruction into distinct phases, also known as pipeline stages. Each stage requires only a portion of the time needed to complete the instruction. These stages are interconnected to form a pipeline, leading to improved efficiency and throughput.

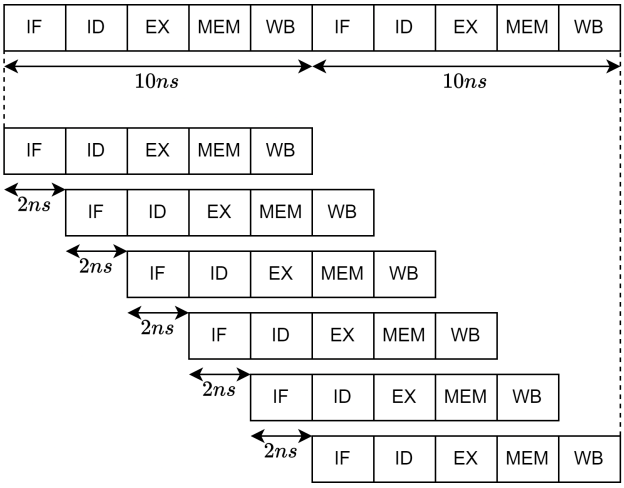


Figure 1.7: Sequential execution and pipelining execution

In pipelining, each stage of the pipeline corresponds to the time required to advance an instruction by one clock cycle. It's crucial to synchronize the pipeline stages, with the duration of a clock cycle determined by the slowest stage of the pipeline.

The objective is to achieve a balance in the length of each pipeline stage. When stages are perfectly balanced, the ideal speedup resulting from pipelining is equal to the number of pipeline stages. This ensures optimal utilization of the pipeline, enhancing overall performance and efficiency.

1.2.4 Hazards

A potential concern arises due to the two-stage nature of the Register File: read access during the instruction decode stage and write access during the WB stage. When a read and a write operation target the same register within the same clock cycle, it necessitates the insertion of a stall to prevent issues.

Definition (*Optimized pipeline*)**.** An optimized pipeline is achieved when the Register File read operation takes place in the second half of the clock cycle, while the Register File write operation occurs in the first half of the clock cycle.

Another potential issue is the occurrence of hazards within the pipeline. Hazards arise when there is a dependency between instructions, and the pipelining process causes a change in the order of accessing operands involved in the dependency, thereby preventing the next instruction from executing during its designated clock cycle. Hazards diminish the performance from the ideal speedup achieved by pipelining. Hazards can be categorized into three main types:

- *Structural hazards*: these occur when different instructions attempt to use the same resource simultaneously. For example, there may be a conflict when both instructions require access to a single Memory Unit for instructions and data.
- *Data hazards*: these occur when an instruction tries to use a result before it is ready. For instance, an instruction might depend on the result of a previous instruction that is still in the pipeline.
- *Control hazards*: these occur when a decision regarding the next instruction to execute is made before the condition for the decision is evaluated. For instance, issues arise during conditional branch execution.

1.2.5 Data hazards

Read After Write Read After Write (RAW) hazard occurs when an instruction j attempts to read an operand before instruction i has written to it. Potential solutions for mitigating this hazard include:

- *Compilation techniques*: using no-operation instructions (nop) or instruction scheduling to rearrange instructions so that dependent instructions are not placed too closely together. If all instructions are dependent, inserting nops may be necessary.
- *Hardware techniques*: Inserting stalls in the pipeline or utilizing data forwarding. Data forwarding involves using temporary results stored in pipeline registers instead of waiting for results to be written back to the Register File. This technique can often resolve conflicts without introducing stalls. However, for load and use hazards, a stall may still be required to properly resolve the issue.

Write After Write Write After Write (WAW) hazard arises when instruction j writes operand before instruction i writes to it. This situation can lead to incorrect order of write operations. Notably, this type of hazard does not occur in the MIPS pipeline since all register write operations occur in the WB stage.

Write After Read Write After Read (WAR) hazard arises when an instruction j writes operand before instruction i reads from it. However, such hazards do not occur in the MIPS pipeline because operand read operations occur in the instruction decode stage, while write operations occur in the WB stage.

1.3 Performance analysis

Developing software has become increasingly complex, to the point where manually managing all constraints has become nearly unfeasible. Despite the proliferation of processor cores and unprecedented computational power, energy consumption has emerged as a critical limitation. Consequently, there is an urgent need for software to prioritize energy efficiency and consider space constraints.

1.3.1 Swiftness metrics

To assess the speed of computers, we examine the system from two perspectives: user and system manager.

User perspective From the user's standpoint, the goal is to minimize program execution time, measured as the response time:

$$T_{\text{response}} = T_{\text{end}} - T_{\text{start}}$$

System manager perspective System managers aim to maximize throughput, which is the total amount of work completed within a specified time frame and is inversely related to the response time of the program:

$$T_{\text{throughput}} = \frac{1}{T_{\text{response}}}$$

This relationship holds true when there are no overlaps; otherwise, higher throughput can be achieved.

Swiftness enhancement Improving system speed typically focuses on optimizing common cases, as they are generally easier and quicker to assess than less frequent cases.

Theorem 1.3.1 (*Amdahl law*)**.** *The speedup of a system due to improving one part of it is limited by the fraction of time that part is utilized.*

Suppose an enhancement E accelerates a fraction F of the task by a factor S, while the remaining task remains unchanged. The overall speedup S_{overall} can be calculated as:

$$S_{\text{overall}} = \frac{1}{(1 - F_{\text{enhanced}}) + \frac{F_{\text{enhanced}}}{S_{\text{enhanced}}}}$$

Example:
Consider a scenario where a new CPU is ten times faster, and in an I/O-bound server where 60% of the time is spent waiting for I/O:

$$S_{\text{overall}} = \frac{1}{(1 - F_{\text{enhanced}}) + \frac{F_{\text{enhanced}}}{S_{\text{enhanced}}}} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56$$

This example illustrates that despite a CPU being ten times faster, the actual improvement in performance is only about 1.56 times, due to the limitations imposed by the fraction of time spent waiting for I/O.

Corollary 1.3.1.1 (*Amdahl's law*)**.** *If an enhancement applies to only a fraction of a task, the maximum speedup that can be achieved is:*

$$S_{\text{max}} = \frac{1}{1 - F_{\text{enhanced}}}$$

These principles underscore the importance of understanding and managing bottlenecks in system performance to achieve meaningful improvements in efficiency and speed.

1.3.2 Performance metrics

The response time of a system measures the latency incurred while completing a task and is the sum of the input/output time and the CPU time:

$$T_{\text{response}} = T_{\text{CPU}} + T_{\text{I/O}}$$

Here, T_{CPU} represents the processing time of the given instruction or program. It is calculated as the ratio of the clock cycles needed to perform the operations to the clock frequency of the processor:

$$T_{\text{CPU}} = \frac{\text{clock cycles needed}}{\text{processor clock frequency}}$$

This formula can be expanded as:

$$T_{\text{CPU}} = \underbrace{\#\text{instructions}}_{\text{IC}} \cdot \underbrace{\frac{\#\text{clock total}}{\#\text{instruction}}}_{\text{CPI}} \cdot \underbrace{\text{cycle duration}}_{\text{CT}}$$

The components involved in this calculation are:

- *Instruction count* (IC): the number of instructions executed, influenced by the algorithm, compiler, and ISA.
- *Cycles per instructions* (CPI): determined by the ISA and CPU organization, this metric accounts for the overlap among instructions. The CPI for a set of instructions is calculated as:

$$T_{\text{CPU}} = \text{CT} \cdot \sum_{i=1}^{n} \text{CPI}_i \cdot \text{I}_i$$

- *Cycle time* (CT): determined by the technology, organization, and circuit design.

Other metrics used to evaluate hardware performance include:

- *Million of instructions per second* (MIPS):

$$\text{MIPS} = \frac{\text{IC}}{T_{\text{execution}} \cdot 10^6} = \frac{\text{Clock frequency}}{\text{CPI} \cdot 10^6}$$

  MIPS quantifies the rate of operations per unit time, with faster machines having higher MIPS ratings.

- *Million of floating point operations* (MFLOPS):

$$\text{MFLOPS} = \frac{\text{Floating point operations in a program}}{T_{\text{CPU}} \cdot 10^6}$$

  Assuming floating-point operations are independent of the compiler and ISA, MFLOPS is a reliable metric for numerical codes, depending on the matrix size to determine the number of floating-point operations in a program.

1.3.3 Power metrics

Energy and power consumption impose constraints on a variety of systems, making it essential to establish an energy and power budget for these systems.

Thermal Design Power (TDP) is a metric used to characterize sustained power consumption. It serves as a target for power supply and cooling systems and typically falls between peak power and average power consumption. To manage power consumption, the clock rate can be dynamically reduced, and measuring energy per task often provides a more accurate assessment. Various techniques are employed to reduce dynamic power consumption, including:

- Optimizing existing processes.
- Implementing dynamic voltage-frequency scaling.
- Employing low-power states for DRAM and disks.
- Utilizing methods like overclocking and turning off cores.

Additionally, static power consumption, which scales with the number of transistors, must be considered. To mitigate static power, power gating is commonly employed.

1.3.4 Cost metrics

In addition to performance, hardware also incurs costs. High-volume production offers several benefits, including a faster learning curve, increased manufacturing efficiency, and decreased research and development costs per unit produced. Commodities refer to identical products offered by numerous vendors in substantial quantities. These products are characterized by their low cost due to high production volume and competition among suppliers.

1.3.5 Benchmarks

The conventional method for conducting performance tests on programs involves using benchmarks. In this methodology, certain groups select programs available to the community to measure performance. These programs are executed on machines, and their performance is reported, allowing for comparison with reports from other machines. The most commonly used benchmarks include:

- *Real programs*: these are representative of real workloads and provide the most accurate way to characterize performance. Occasionally, modified CPU-oriented benchmarks may eliminate I/O operations.
- *Kernels* or micro benchmarks: these are representative program fragments useful for focusing on individual features.
- *Synthetic benchmarks*: similar to kernels, these benchmarks attempt to match the average frequency of operations and operands from a large set of programs.
- *Instruction mixes* for CPI.

he System Performance Evaluation Cooperative (SPEC) was established in 1989 to address benchmarking issues. Benchmarks may not be representative if the workload is I/O bound, rendering certain benchmarks like SPECint ineffective. Benchmarks also become obsolete over time, and aging benchmarks can be problematic as benchmarking pressure incentivizes vendors to optimize systems for specific benchmarks.

A straightforward method for comparing relative performance is to use the total execution time of the programs. Another option is to calculate the arithmetic mean of the execution times, which is valid only if the programs run equally often. If the programs have different execution frequencies, the weighted arithmetic mean is used:

$$\sum_{i=1}^{n} \frac{\mathrm{w}_i \cdot \mathrm{time}_i}{n}$$

In general, the arithmetic mean is used for times, the harmonic mean if rates must be used, and the geometric mean if ratios must be used.

1.4 Exceptions

Definition (*Interrupt*)**.** An interrupt is an external or internal event that necessitates processing by another system program.

Interrupts, typically unexpected or infrequent from the program's perspective, can arise from various causes. Exceptions can be categorized as follows:

a. *Synchronous*: events from devices external to the CPU and memory, manageable after the current instruction completes, making them easier to handle.

b. *Asynchronous*: events initiated by the current instruction.

a. *User requested*: predictable and treated similarly to exceptions, using the same mechanisms for saving and restoring state; handled after instruction completion.

b. *Coerced*: arise from hardware events beyond the program's control.

a. *User maskable*: the mask determines whether the hardware responds to the exception.

b. *User non-maskable*: exceptions that cannot be ignored by the user.

a. *Within instructions*: typically synchronous as the instruction initiates the exception, requiring the instruction to halt and restart.

b. *Between instructions*: asynchronous exceptions arising from critical situations, often leading to program termination.

a. *Terminate*: program execution always halts after the interrupt.

b. *Resume*: program execution continues after the interrupt.

Asynchronous interrupt In the case of an asynchronous interrupt, an I/O device signals the need for attention by activating a prioritized interrupt request line. When the processor decides to handle the interrupt, it completes all instructions before the current one, saves the PC in a dedicated register called the Exception Program Counter (EPC), disables all interrupts, and then transfers control to a specified interrupt handler operating in kernel mode. Before enabling interrupts to accommodate nested interrupts, the interrupt handler:

- Saves the PC into general-purpose registers
- Temporarily blocks further interrupts until the PC is saved
- Retrieves information about the interrupt cause from a designated status register.

The interrupt handler then uses a specialized indirect jump instruction called Return-From-Exception (RFE) to: enable interrupts, restore the processor to user mode, and reinstate hardware status and control state

Synchronous interrupt A synchronous interrupt, also known as an exception, is triggered by a specific instruction. Typically, the instruction cannot finish execution and must be restarted after handling the exception, necessitating the undoing of one or more partially executed instructions. However, in the case of a system call trap, the instruction is considered fully executed, involving a special jump instruction that transitions to privileged kernel mode.

1.4.1 Precise interrupt

Definition (*Precise interrupt*)**.** An interrupt or exception is deemed precise when a singular instruction (or interrupt point) exists at which all preceding instructions have finalized their state, and no subsequent instructions, including the interrupting instruction, have altered any state.

This implies that execution can effectively resume from the interrupt point, yielding the correct outcome. Precise interrupts are desirable for several reasons:

- They facilitate the restart of various interrupt and exception types
- They simplify the process of determining the exact cause of the interruption

While restartability doesn't mandate preciseness, it significantly enhances the ease of restarting. Preciseness notably streamlines the task for the operating system, requiring less state to be preserved when unloading processes and allowing for quicker restarts.

1.4.2 Exceptions in pipeline

Exceptions can arise at various stages within the pipeline. The recommended approach for handling these interrupts is to minimize pipeline interruption as much as possible.

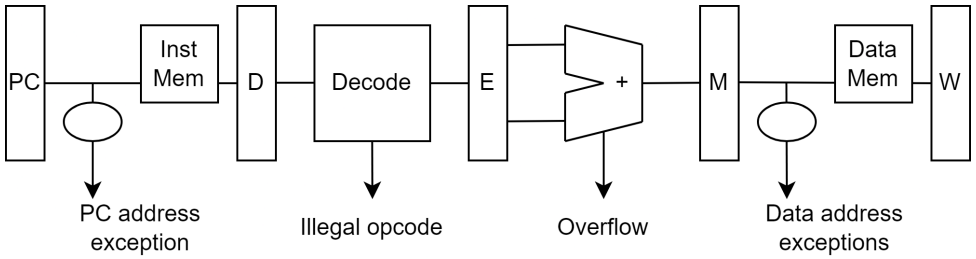


Figure 1.8: Exception origins

To manage this, instructions in the pipeline can be tagged to indicate whether they cause exceptions. This tagging process waits until the memory stage concludes before flagging an exception. Interrupts are then represented as no-operation (nop) placeholders inserted into the pipeline instead of regular instructions.

In cases where a nop is flushed, it is assumed that the interrupt condition persists. Managing interrupt conditions can be complex due to requirements such as switching to supervisor mode and saving one or more PCs.

Optimizing the instruction fetch to start fetching instructions from the interrupt vector can also be challenging due to these complexities.

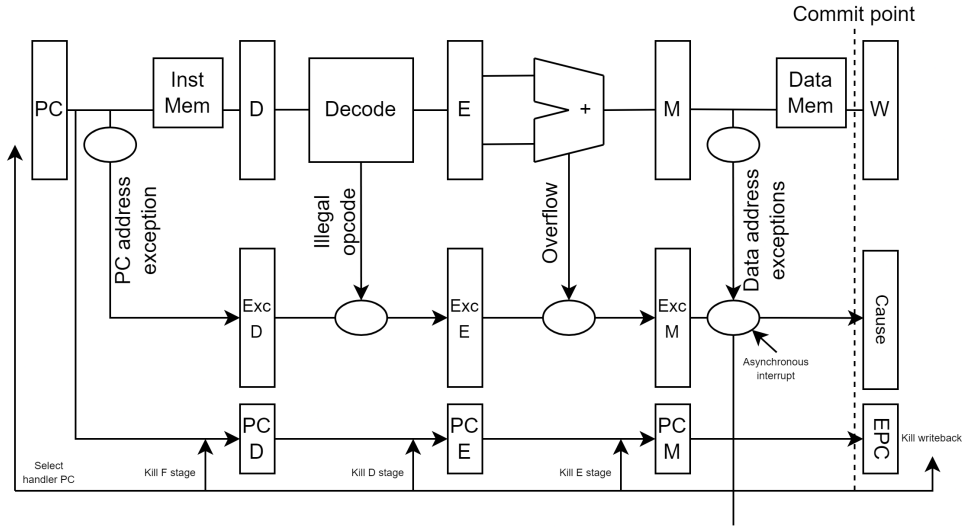


Figure 1.9: Exception handling

Exception flags are maintained within the pipeline until reaching the commit point. Exceptions occurring in earlier pipeline stages take precedence over later ones for a particular instruction. External interrupts are injected at the commit point, overriding any other exceptions present.

If an exception occurs at the commit point, the cause and EPC registers are updated, all pipeline stages are terminated, and the handler PC is injected into the fetch stage.

1.4.3 Exceptions prediction

Methods to anticipate exceptions include:

- *Prediction mechanism*: given the infrequent occurrence of exceptions, a straightforward prediction of no exceptions yields high accuracy.
- *Verification of prediction mechanism*: exceptions are identified at the conclusion of the instruction execution pipeline, utilizing specialized hardware tailored for different exception types.
- *Recovery mechanism*: architectural state is exclusively written at the commit point, allowing the discarding of partially executed instructions after an exception. Following the exception, the pipeline is flushed, and the exception handler is initiated.
- *Bypassing*: bypassing facilitates the utilization of uncommitted instruction outcomes by subsequent instructions.

CHAPTER 2

Caches and memory

2.1 Introduction

Since 1980, there has been a notable divergence in performance between CPUs and DRAM. To bridge this gap, architects introduced small, high-speed cache memories between the CPU and DRAM, establishing a memory hierarchy. With the advent of recent multicore processors, the design of memory hierarchy has become increasingly critical. To address this challenge, several solutions are necessary: multi-port pipelined caches, two-level cache structure per core, and shared third-level cache on chip: Integrating a shared third-level cache directly on the chip further streamlines memory access.

The ultimate goal of the memory hierarchy is to create the illusion of a vast, speedy, and cost-effective memory system. This allows programs to access a memory space scalable to the size of the disk, with speeds comparable to register access. Achieving this necessitates the establishment of a memory hierarchy comprising various technologies, costs, and sizes, each with distinct access mechanisms.

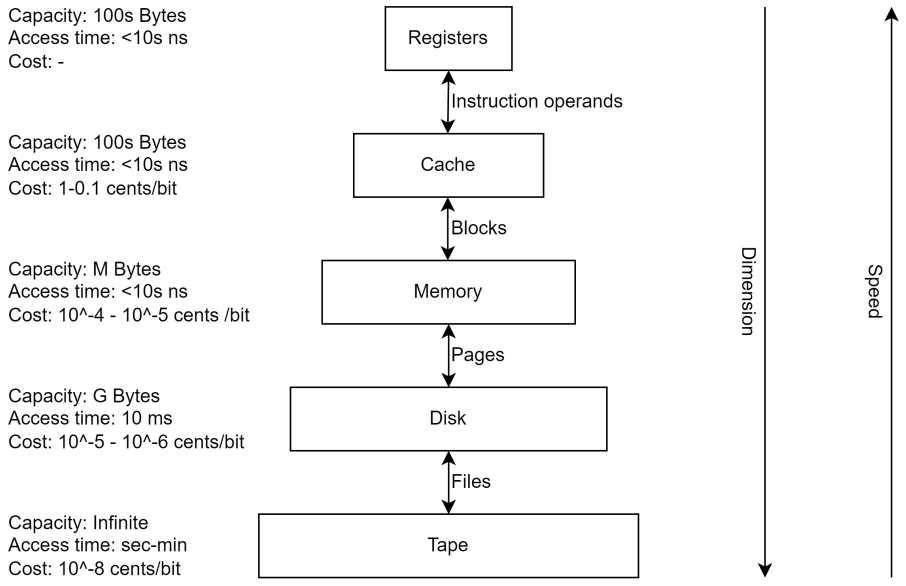


Figure 2.1: Memory hierarchy

2.2 Cache locality

The principle of locality asserts that programs tend to access only a fraction of the total address space at any given moment. This principle is further elaborated through two key properties:

1. *Temporal locality*: if a memory location is accessed, it is probable that it will be accessed again in the near future.

2. *Spatial locality*: if a memory location is accessed, it is likely that nearby locations will also be accessed in the near future.

Caches leverage both forms of locality. They exploit temporal locality by retaining the contents of recently accessed memory locations, anticipating their future use. Additionally, they exploit spatial locality by pre-fetching blocks of data surrounding recently accessed locations, capitalizing on the likelihood of adjacent memory access.

When examining a processor address, the cache tags are searched to locate a match. Subsequently, one of the following actions occurs:

- *Cache hit*: if a match is found in the cache, the data copy is retrieved from the cache and returned.

- *Cache miss*: if the address is not found in the cache, a block of data is read from the main memory. There is a wait period, after which the data is returned to the processor, and the cache is updated accordingly.

Based on these operations, several metrics can be defined.

Definition (*Hit rate*)**.** The hit rate is the fraction of accesses that are found in the cache.

Definition (*Miss rate*)**.** The miss rate is the complement of the hit rate, indicating the fraction of accesses that result in cache misses.

Definition (*Hit time*)**.** The hit time comprises the time required for RAM access along with the time needed to determine whether the access resulted in a hit or miss.

Definition (*Miss time*)**.** The miss time comprises the time necessary to replace a block in the cache and the time taken to deliver the block to the processor.

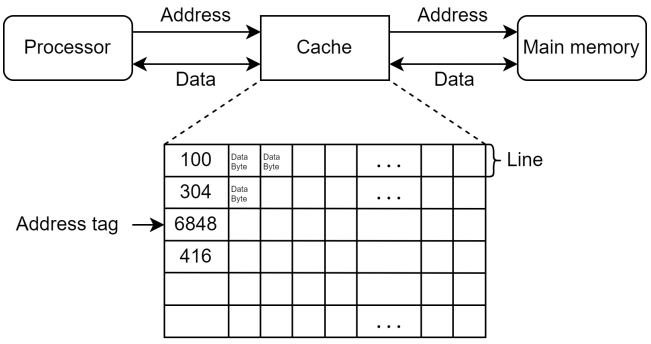
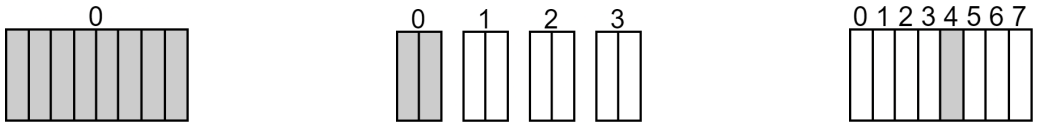


Figure 2.2: Cache and processor interaction

2.2.1 Cache blocks

Depending on the chosen cache type, the placement of a block number can be as follows:

- *Fully associative*: the block can be placed anywhere within the cache.
- *Two-way set associative*: the block can be placed anywhere within set zero, which corresponds to $n \mod 4$.
- *Direct mapped*: the block can only be placed into block four, determined by $n \mod 8$.



(a) Fully associative (b) Two-way set associative (c) Direct mapped

Figure 2.3: Possible blocks placement

2.2.2 Block read

A cache miss can occur for several reasons:

1. *Compulsory miss* (cold start or process migration): this happens during the first access to a block, such as during a cold start or when a process migrates. It's essentially an unavoidable aspect of system operation, and there's little that can be done to mitigate it.
2. *Capacity miss*: this occurs when the cache is unable to accommodate all the blocks accessed by the program. Increasing the cache size can help reduce the frequency of these misses.
3. *Conflict miss* (collision): multiple memory locations are mapped to the same cache location, resulting in conflicts. This can be addressed by either increasing the cache size or increasing associativity, which allows more flexibility in mapping memory locations to cache locations.
4. *Coherence Miss* (invalidation): this type of miss occurs when another process, such as I/O operations, updates memory, leading to inconsistencies in cached data. Ensuring cache coherence mechanisms are in place can help mitigate this issue.

To locate a block, the cache index is used to select the set to search within, while the tag identifies the actual copy. If no matching candidates are found, a cache miss is declared. The structure of a memory address typically includes a field for selecting data within a block.

Figure 2.4: Memory address general structure

Increasing associativity reduces the index size and expands the tag. Fully associative caches, for example, do not have an index field and can directly access any block in the cache.

The fully associative cache, characterized by requiring only a tag and block offset, is depicted as follows.

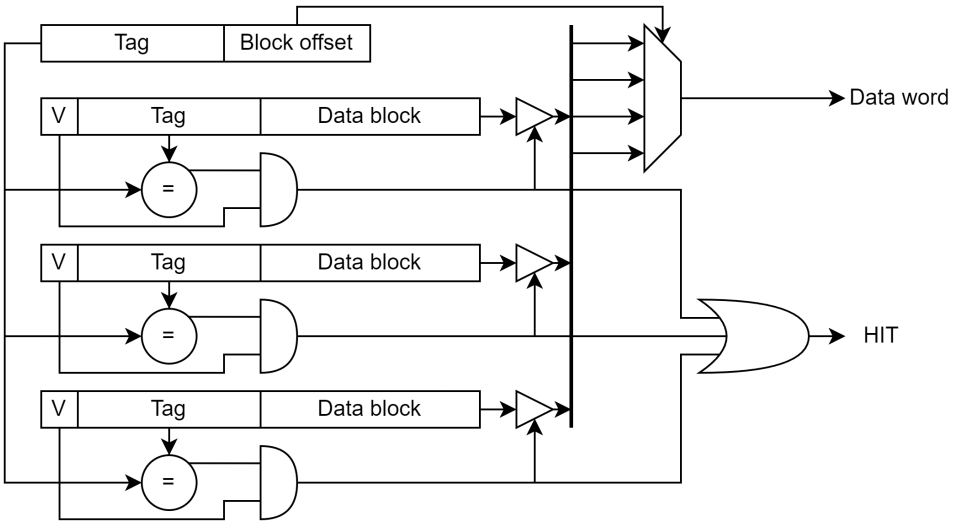


Figure 2.5: Fully associative cache

The two-way set associative cache, which necessitates a tag, index, and block offset, is represented as follows.

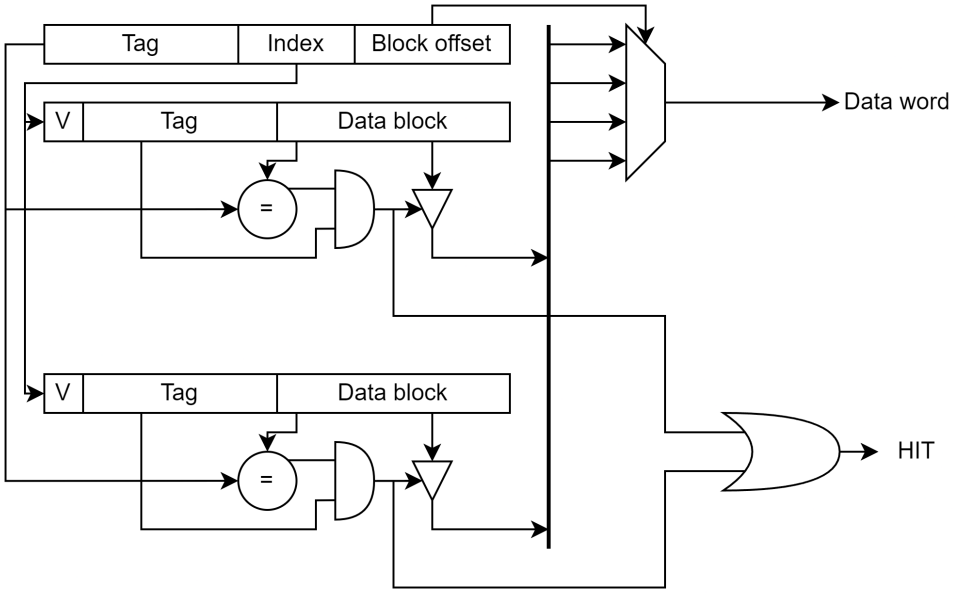


Figure 2.6: Two-way set associative cache

The direct-mapped cache, which also requires a tag, index, and block offset, is illustrated as follows.

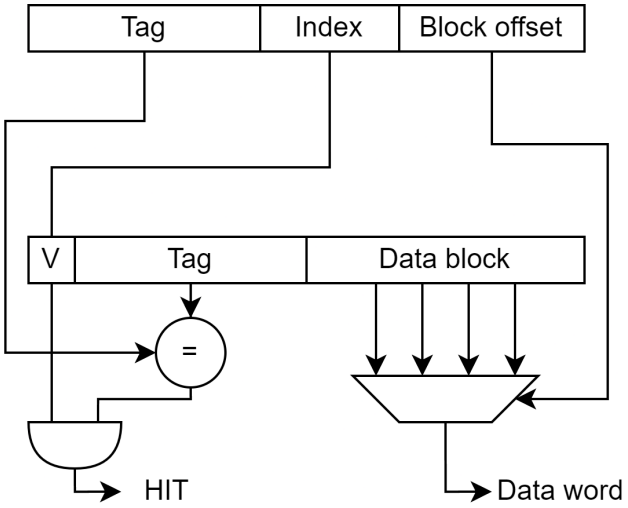


Figure 2.7: Direct mapped cache

2.2.3 Block replacement

In the context of cache misses, block replacement is straightforward for direct-mapped caches. However, for set-associative or fully associative caches, the choice of replacement policy has a significant impact since replacements only occur upon misses. Here are some commonly used replacement policies:

- *Random*: blocks are replaced randomly, without any specific order or pattern.
- *Least Recently Used* (LRU): this policy replaces the block that has been accessed least recently. Although effective, implementing LRU requires tracking the access history of each block, making it feasible only for caches with a few sets due to the computational overhead.
- *First In First Out* (FIFO): blocks are replaced based on the order they were brought into the cache. FIFO is commonly used in highly associative caches where keeping track of access history for LRU may not be practical.

2.2.4 Block write

When handling writes, we have two main options for a cache hit:

- *Write through*: this strategy involves writing the data both to the cache and to main memory simultaneously. While this approach typically results in higher traffic, it simplifies cache coherence management.
- *Write back*: with this approach, data is written only to the cache. The corresponding entry in main memory is updated only when the cache block is evicted. A dirty bit per block helps reduce traffic by indicating whether the block in the cache has been modified.

In the case of a cache miss, there are two alternatives:

- *No write allocate*: data is written directly to main memory without being fetched into the cache.

- *Write allocate*: in this scenario, the data is fetched into the cache upon a write miss.

The most common combinations of these strategies are:

- *Write through with no write allocate*: data is written to both the cache and main memory simultaneously. In the event of a write miss, no data is brought into the cache.
- *Write back with write allocate*: data is written only to the cache, and in the case of a write miss, the data is fetched into the cache before being modified.

Additionally, a write buffer can be used for write-through caches to avoid CPU stalls.

2.3 Cache performance

Definition (*Memory stall cycles*)**.** Memory stall cycles represent the number of cycles during which the CPU is idle, waiting for memory access to complete.

We assume that the cycle time includes the time necessary to manage a cache hit and that during a cache miss, the CPU is stalled. We can compute:

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{memory stall cycles}) \cdot \text{clock cycle time}$$

$$\text{memory stall cycles} = \text{number of misses} \cdot \text{miss penalty}$$

Simplifying by averaging reads and writes:

$$\text{memory stall cycles} = \text{IC} \cdot \left(\frac{\text{memory accesses}}{\text{instruction}}\right) \cdot \text{miss rate} \cdot \text{miss penalty}$$

This value is architecture-dependent but hardware-independent.

The average access time is computed as:

$$T_{\text{access}} = H_{\text{time}} + M_{\text{rate}} \cdot M_{\text{penalty}}$$

Optimization The access time for a cache can be optimized by:

- Reducing M_{rate}: achieved through larger block size, larger cache size, or higher associativity.
- Reducing M_{penalty}: achieved by using multilevel caches.
- Reducing H_{time}: achieved by giving reads priority over writes.

2.3.1 Cache design

In cache design, factors such as cache size, block size, associativity, replacement policy, and the choice between write-through and write-back mechanisms play significant roles. The best option involves balancing access patterns (workload and usage) and technological expenses. Often, simplicity emerges as the preferred solution.

SRAM Static RAM (SRAM) memory requires low power to retain data and uses six transistors for each bit.

DRAM Dynamic RAM (DRAM) must be rewritten after each read and periodically refreshed (each row every eight milliseconds). It requires only one transistor per bit. The address lines are multiplexed into upper and lower half addresses.

Flash memory Flash memory is a type of EEPROM, requiring block erasure prior to overwrite. It retains data without power, classifying it as non-volatile storage. It has a finite number of write cycles and falls in price between SDRAM and disk storage. Although slower than SRAM, it outpaces traditional disk speeds.

Optimizations According to Amdahl's Law, memory capacity should grow linearly with processor speed. However, memory capacity and speed have not kept pace with processors. To address this issue, several optimizations are possible:

- Multiple accesses to the same row.
- Synchronous DRAM: Adds a clock to the DRAM interface and supports burst mode with critical word first.
- Wider interfaces.
- Double Data Rate (DDR) memory.
- Multiple banks on each DRAM device.

2.4 Virtual memory

Virtual memory is a crucial architecture used to confine processes within their allocated memory space boundaries, providing several key functions:

- *User mode and supervisor mode*: facilitates distinct execution modes for user applications and privileged operating system tasks.
- *CPU state protection*: safeguards critical components of the CPU state from unauthorized access.
- *Mode transition mechanisms*: implements mechanisms for transitioning between user and supervisor modes securely.
- *Memory access control*: provides tools to restrict memory accesses, ensuring security and isolation.
- *Translation Lookaside Buffer* (TLB): accelerates virtual to physical address translation by caching recent mappings.

2.4.1 Virtual machines

Virtual memory forms the foundation for creating virtual machines (VMs), which enhance isolation and security in computing environments where multiple users or tasks share a single physical machine. The key functionalities of VMs include:

- *Isolation and security*: ensures that each VM is isolated from others, preventing interference and ensuring security for diverse workloads.

- *Support for different architectures*: presents distinct ISAs and operating systems to user programs concurrently.

Hypervisor and guest VM A hypervisor, or VM monitor (VMM), is software responsible for managing and running multiple guest VMs on a single physical machine. Key aspects of MV management include:

- *Guest page tables*: each guest operating system maintains its own set of page tables, which map virtual addresses used by applications running inside the guest to physical addresses in the real memory managed by the hypervisor.
- *Real memory*: the hypervisor introduces a layer of abstraction between physical memory and virtual memory used by guest VMs.
- *Shadow page tables*: to efficiently manage address translations, the hypervisor maintains shadow page tables, which mirror the guest's page tables but map directly to physical memory. This allows the hypervisor to detect changes made by the guest to its page tables, ensuring consistency and security.

Virtual memory and VMs enable efficient resource utilization and flexibility in managing computing resources, making them essential components in modern computing environments.

CHAPTER 3

Branch prediction

3.1 Introduction

In computer architecture, handling branches efficiently is crucial for optimizing processor performance. Branches introduce potential stalls in instruction execution as the correct path of execution is determined. The aim is to predict the outcome of branch instructions as early as possible to minimize these stalls and improve overall performance. The effectiveness of branch prediction techniques hinges on three primary factors:

1. *Accuracy*: this metric reflects the percentage of correct predictions made by the branch predictor. Higher accuracy reduces unnecessary stalls caused by mispredictions.

2. *Cost of misprediction*: refers to the penalty incurred when a branch prediction turns out to be incorrect. In deeply pipelined processors, mispredictions can lead to significant performance degradation due to the time lost in executing unnecessary instructions.

3. *Branch frequency*: the frequency at which branches occur within a program. Programs with frequent branches benefit significantly from accurate prediction techniques to maintain performance.

To mitigate the performance impact of branch hazards, several methods are employed:

- *Static branch prediction*: these techniques rely on predefined actions for branches that remain constant throughout program execution. Predictions are determined statically at compile time based on heuristics or program structure.

- *Dynamic branch prediction* : unlike static techniques, dynamic approaches adjust branch predictions during program execution based on runtime behavior and historical data. This adaptability improves prediction accuracy for varying branch conditions.

3.2 Static branch prediction

Static branch prediction is employed in processors where branch behavior is expected to remain predictable at compile time. It can also complement dynamic predictors by providing a baseline prediction strategy.

3.2.1 Branch always not taken

Branch always not taken assumes that every branch instruction will not be taken. In the pipeline, if the branch condition is evaluated to false during the ID stage, the sequential instruction flow proceeds without any penalties. If the branch condition is true (an incorrect prediction), the pipeline must be flushed. This involves discarding the fetched instructions after the branch and fetching the correct instructions from the branch target address, incurring a one-cycle penalty.

3.2.2 Branch always taken

Branch always taken predicts that every branch will be taken. Once the branch is decoded and its target address computed, the processor immediately begins fetching and executing instructions from the target address. This approach is advantageous in pipelines where the branch target address is known early enough to initiate fetching without waiting for the branch outcome. However, in some pipelines like MIPS, where the branch target address isn't determined before the branch outcome, this approach offers no benefit.

3.2.3 Backward taken forward not taken

Backward taken forward not taken prediction strategy categorizes branches based on their direction. Backward-going branches, such as loops, are predicted as taken, while forward-going branches, like conditional jumps to subsequent code, are predicted as not taken.

3.2.4 Profile-driven prediction

Profile-driven prediction utilizes profiling information gathered from previous executions to improve branch prediction accuracy. This approach leverages data on branch behavior collected during runtime to predict future branch outcomes more effectively. Additionally, compilers can provide hints to further refine these predictions based on the program's characteristics and execution patterns.

3.2.5 Delayed branch

In a delayed branch scenario, the compiler strategically places an independent instruction into a designated branch delay slot. This instruction executes regardless of whether the branch is taken or not. Typically, processors like MIPS have a single-cycle delay slot, while more complex pipelines may feature longer delays, though filling multiple slots can be challenging for compilers. For MIPS, the compiler routinely schedules an independent instruction immediately following the branch. The behavior of the delayed branch is consistent:

- If the branch is not taken, the processor proceeds with the instruction following the branch.
- If the branch is taken, execution continues at the branch target.

The compiler's task is to ensure that the instruction placed in the delay slot is both valid and beneficial. There are three primary strategies for scheduling the delay slot:

1. *From before*: an independent instruction preceding the branch is placed in the delay slot. This instruction always executes, irrespective of the branch outcome.

2. *From target*: the delay slot is filled with an instruction from the branch target. To prevent certain instructions from being moved after the branch, a register in the branch condition may be used. This strategy is effective for branches likely to be taken, such as loop branches.

3. *From fall-through*: qn instruction from the fall-through path, where the branch is not taken, occupies the delay slot. Similar to the from target strategy, this may involve using a register in the branch condition to control instruction placement. It's favored for branches unlikely to be taken, such as forward branches.

In deeply pipelined processors with multi-cycle delays, fully populating delay slots with useful instructions becomes more challenging. The primary challenges in scheduling delayed branches include: constraints on which instructions can occupy the delay slot, and the compiler's ability to accurately predict branch outcomes statically.

To mitigate these challenges, some processors employ a canceling or nullifying branch instruction. This approach indicates the predicted branch direction:

- When the prediction is correct, the instruction in the delay slot executes normally.
- If the prediction is incorrect, the instruction is replaced with a nop.

3.3 Dynamic branch prediction

The main idea of dynamic branch prediction techniques is to leverage past branch behavior to forecast future outcomes. Dynamic branch prediction integrates two key mechanisms:

- *Branch outcome predictor*: determines the likelihood of branch direction based on historical behavior.
- *Branch target predictor*: forecasts the target address for a taken branch.

These prediction modules collaborate within the instruction fetch unit, aiding in the prediction of the subsequent instruction to fetch from the instruction cache: if the branch isn't taken, the PC increments; otherwise the BTP provides the target address.

3.3.1 Branch History Table

The Branch History Table (BHT) is a data structure where each entry consists of a single bit, representing whether a branch was recently taken or not. This table is pivotal in branch prediction mechanisms.

The BHT operates by predicting the outcome of branches based on recent history:

- *Initial prediction*: when encountering a branch, the BHT provides a prediction based on its stored history. If the history indicates that the branch has been frequently taken, the BHT predicts that the branch will be taken again.
- *Execution flow*: the processor begins fetching instructions in the predicted direction, assuming the correctness of the prediction.

- *Validation*: if the prediction proves correct, execution proceeds smoothly. However, if the branch prediction is incorrect the prediction bit associated with that branch in the BHT is updated to reflect the correct outcome. Then, the pipeline is flushed to discard any instructions fetched based on the incorrect prediction and the correct instructions are then fetched and executed based on the accurate branch outcome.

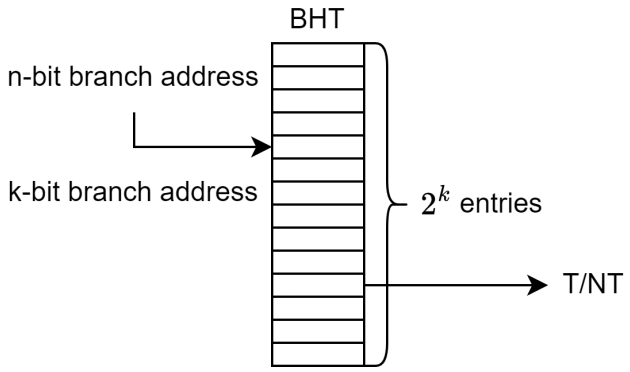


Figure 3.1: BHT structure

Mispredictions can occur due to: incorrect predictions for specific branches, or conflicting history at the same index when multiple branches reference it, leading to inaccurate predictions. To mitigate mispredictions:

- *Increase BHT size*: by expanding the number of rows in the BHT, more branches can be tracked simultaneously, reducing the likelihood of conflicting histories.
- *Hashing function*: employing a hashing function for indexing can distribute branches more evenly across the BHT, minimizing index collisions and improving prediction accuracy.

One bit BHT The 1-bit BHT has a notable limitation, especially evident in loop branches. Even if a branch is mostly taken throughout a loop but is not taken once, the 1-bit BHT may mispredict twice instead of once. This results in two erroneous predictions:

- At the end of the loop iteration, where the prediction bit suggests a taken branch, contradicting the need to exit the loop.
- When re-entering the loop after the first iteration's end, the prediction bit indicates an exit from the loop, stemming from the final iteration's flipped prediction bit.

Two bit BHT In contrast, the 2-bit BHT requires two consecutive prediction failures before altering its prediction. For a loop branch, during the final iteration, there is no need to change the prediction. Each index in the table employs 2 bits, encoding the four states of a finite state machine.

Higher bit BHT In the case of an n-bit BHT, each entry in the prediction buffer requires an n-bit saturating counter. This counter ranges from 0 to $2^n - 1$. If the counter equals or exceeds half of its maximum value, the branch is predicted as taken; otherwise, as untaken. Similar to the 2-bit scheme, a taken branch increments the counter, while an untaken branch decrements it. Research suggests that 2-bit predictors perform nearly as effectively as higher bit predictors.

3.3.2 Branch target predictor

The Branch Target Buffer (BTB), also referred to as a branch target predictor, acts as a cache that stores the predicted target address for instructions following a branch. During the IF stage, the BTB is accessed using the instruction address of the fetched instruction, which may include a branch, to index the cache. Typical entries in the BTB include the exact address of a branch, and the predicted target address.

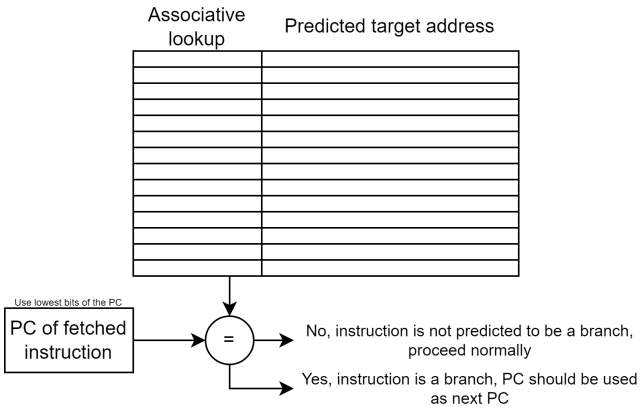


Figure 3.2: Branch target buffer structure

3.3.3 Correlating branch predictors

BHT predictors rely solely on the recent behavior of a single branch to forecast its future behavior. Correlating branch predictors, however, operate on the principle that recent branches exhibit correlations. This means that not only the current branch under consideration but also other recent branches can influence the prediction of the current branch. Predictors that utilize the behavior of other branches to make predictions are known as correlating predictors or 2-level predictors.

In a general (m, n) correlating predictor, the system maintains a history of the last m branches to select from 2^m BHTs, each being an n-bit predictor.

Accuracy A 2-bit predictor without global history can be seen as a $(0, 2)$ predictor. Comparing the performance of a simple 2-bit predictor to a $(2, 2)$ correlating predictor it often outperforms a 2-bit predictor regardless of the number of entries.

Two-level adaptive branch predictors In two-level adaptive branch predictors:

- The first-level history is stored in one or more k-bit shift registers called the Branch History Register (BHR), which records outcomes of the k most recent branches.
- The second-level history is stored in one or more tables known as the Pattern History Table (PHT), consisting of two-bit saturating counters.

The BHR is used to index the PHT to determine which 2-bit counter to use for prediction. Several types of two-level predictors include: BHT (local), GAs (global), and GShare (combination of local and global).

CHAPTER 4

Hardware-level parallelism

4.1 Introduction

Instruction-Level Parallelism (ILP) involves the simultaneous execution of independent instructions. Achieving ILP requires:

- Absence of structural hazards.
- No stalls due to Read After Write (RAW), Write After Read (WAR), or Write After Write (WAW) dependencies.
- No stalls due to control dependencies.

The CPI for a pipeline is calculated using

$$\text{CPI}_{\text{pipeline}} = \text{CPI}_{\text{ideal pipeline}} + \text{stalls}_{\text{structural}} + \text{stalls}_{\text{data hazard}} + \text{stalls}_{\text{control}}$$

In this formula, $\text{CPI}_{\text{ideal pipeline}}$ denotes the optimal performance attainable by the pipeline.

Performance limitations due to hazards arise in several ways:

- Structural hazards demand additional hardware resources.
- Data hazards require solutions such as forwarding and compiler scheduling.
- Control hazards can be addressed through early evaluation, program counter adjustments, delayed branching, and prediction techniques.

The influence of these hazards becomes more pronounced with longer pipelines. Pipelining enhances instruction throughput rather than reducing execution latency.

4.2 Complex pipelining

Achieving high performance in pipelining introduces significant complexity, particularly when dealing with long latency or partially pipelined Floating Point Units (FPUs), multiple function and Memory Units (MUs), memory systems with variable access times, and precise exception handling.

A potential solution to these challenges is the complex in-order pipeline. This pipeline introduces a delay in the WB stage to ensure that all operations have the same latency to the WB stage. It avoids over-subscription of write ports, allowing one instruction to enter and exit each cycle. Furthermore, it enforces in-order instruction commitment, simplifying the implementation of precise exception handling.

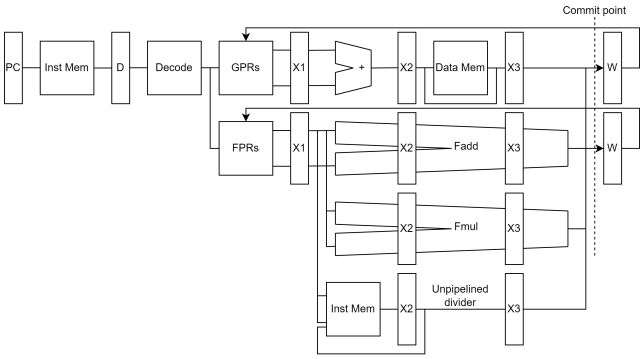


Figure 4.1: Complex in order pipeline

However, several issues arise with this approach. Structural conflicts can occur during the execution stage if certain FPUs or MUs are not pipelined and take longer than one cycle. Structural conflicts also emerge at the WB stage because different functional units may have variable latencies. Additionally, out-of-order write hazards can occur due to these varying latencies.

4.3 Dependency

To achieve higher performance within a given technology, it is crucial to extract more parallelism from the program. This involves detecting and resolving dependencies and scheduling instructions to maximize execution parallelism with the available resources.

Dependencies among instructions are key to determining the level of parallelism in a program. If two instructions are dependent on each other, they cannot execute simultaneously and must be executed sequentially or with limited overlap. There are three types of dependencies: name, data, and control dependencies.

4.3.1 Name dependency

A name dependency occurs when two instructions use the same register or memory location without any data flow between them. There are two types of name dependencies between an instruction i preceding instruction j:

- *Anti-dependence* (WAR): occurs when j writes to a register or memory location that instruction i reads.
- *Output dependence* (WAW): arises when both i and j write to the same register or memory location.

Name dependencies differ from true data dependencies as there is no data flow between the instructions.

Register Renaming If the names used in the instructions can be altered, the instructions do not conflict. Detecting dependencies through memory locations is more challenging since two addresses might refer to the same location but appear different. Register renaming is easier to implement and can be accomplished statically by the compiler or dynamically by the hardware.

4.3.2 Data dependency

Data dependencies can potentially create data hazards, but the actual hazard and the number of stalls required to eliminate it depend on the pipeline. There are three types of data hazards:

- *RAW hazards*: true data dependence.
- *WAW hazards*: output dependence
- *WAR hazards*: anti-dependence.

It is important to note that data dependencies are inherent to the program, while hazards are specific to the pipeline.

4.3.3 Control dependency

Control dependencies dictate the order of instruction execution, preserved by:

- Executing instructions in program order to ensure that an instruction before a branch executes before the branch.
- Detecting control hazards to ensure that an instruction dependent on a branch is not executed until the branch direction is known.

While preserving control dependence is a simple way to maintain program order, it is not the most critical property that must be preserved for execution.

4.4 Scheduling

Two essential properties must be maintained to ensure program correctness, typically achieved by preserving both data and control dependencies. The first is exception behavior, which ensures that any changes in the execution order of instructions do not affect how exceptions are triggered within the program. The second is data flow, which pertains to the actual movement of data values among instructions, ensuring that the correct results are produced and consumed.

There are two main strategies to support ILP: dynamic scheduling and static scheduling. Dynamic scheduling relies on hardware to identify and exploit parallelism within the program, while static scheduling depends on software to identify potential parallelism beforehand.

4.4.1 Dynamic scheduling

Dynamic scheduling involves hardware reordering instruction execution to mitigate pipeline stalls while upholding data flow and exception behavior. The main advantages of this approach include handling cases where dependencies are unknown at compile time, simplifying compiler complexity, and facilitating efficient execution of compiled code on different pipeline architectures. However, these benefits come with costs such as a significant increase in hardware complexity, higher power consumption, and the potential for imprecise exceptions. In

dynamic scheduling, instructions are fetched and issued in program order, but execution begins as soon as operands are available, potentially allowing out-of-order execution. This introduces the possibility of WAR and WAW data hazards and implies out-of-order completion unless a reorder buffer is present to ensure in-order completion.

4.4.2 Static scheduling

In static scheduling, compilers use sophisticated algorithms for code scheduling to harness ILP. While the ILP available within a basic block is often limited, significant performance improvements can be achieved by exploiting ILP across multiple basic blocks, transcending branches. Static scheduling involves the compiler detecting and resolving dependencies by reordering code to avoid conflicts. The compiler's output is typically dependency-free code, which is well-suited for architectures like Very Long Instruction Word (VLIW) processors. However, there are limits to ILP exploitation, including unpredictable branches, variable memory latency such as unpredictable cache misses, code size explosion, and increased compiler complexity.

4.5 Scoreboard

Consider a data structure that maintains records of all instructions across Functional Units (FUs). Before dispatching an instruction at the issue stage, several checks are necessary:

1. The availability of the required FUs.
2. The availability of input data, considering RAW hazards.
3. The safety of writing to the destination, taking into account WAR and WAW hazards.
4. The detection of structural conflicts at the WB stage.

To manage these checks, we introduce a data structure called the Correct Issues Table to keep track of the status of FUs. The fields in this table include:

Name	**Busy**	**Op**	**Dest**	**Src1**	**Src2**
Int Mem					
Add1 Add2 Add3					
Mult1 Mult2					
Div					

At the issue stage, an instruction consults this table using specific rules:

- To check if the required FUs are available, the busy column is consulted.
- For RAW hazard detection, the destination column is searched for the instruction's sources.
- For WAR hazard detection, the source columns are searched for the instruction's destination.

- For WAW hazard detection, the destination column is searched for the instruction's destination.

An entry is added to the table if no hazard is detected and is removed after the WB stage.

4.5.1 Scoreboard features

Scoreboard efficiently manages instruction execution while ensuring proper handling of data dependencies. Here are its key features:

- Instructions are dispatched in sequential order to FUs as long as there are no structural hazards or WAW hazards.
- If a structural hazard occurs or no FUs are available, execution is stalled.
- Each register has only one pending write operation at a time.
- Instructions may execute out-of-order to handle RAW hazards, waiting for input operands before execution.
- To prevent WAR hazards, instructions wait until preceding instructions have read the output register, and the result remains in the FU until the register is free for writing.

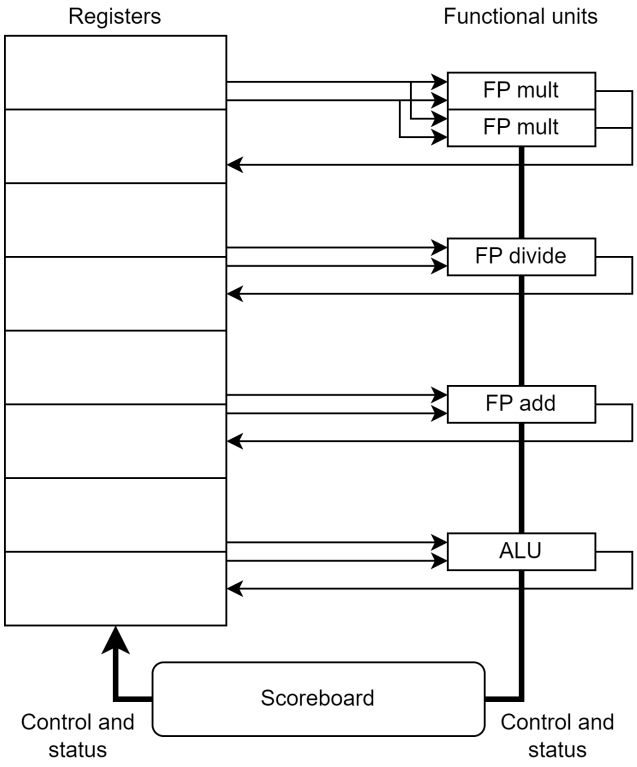


Figure 4.2: CDC6600 Scoreboard

Scoreboard serves as the central hub for hazard management in the pipeline architecture. All instructions are routed through Scoreboard for processing. It orchestrates the timing for instructions to access their operands and commence execution, continuously monitoring hardware changes to resolve stalled instructions. Additionally, it governs the timing for instructions to

commit their results, optimizing performance and ensuring efficient instruction flow within the updated pipeline framework.

Scoreboard plays a crucial role in managing hazards and dependencies in the instruction pipeline. Without register renaming, it must detect WAW hazards and stall the issuance of new instructions until the conflicting instruction completes execution. To accommodate multiple instructions in the execution phase, the system requires either multiple execution units or pipelined execution units. Scoreboard maintains the state of operations and tracks dependencies to ensure correct execution sequencing and hazard detection.

4.5.2 Scoreboard structure

The ID stage stage is now divided into two distinct phases:

1. *Issue*: this phase is responsible for instruction decoding and checking for structural hazards.

2. *Read operands*: this phase holds instructions until data hazards are resolved, with an out-of-order execution approach.

Although instructions are issued sequentially, operand reading occurs out of order. This innovative approach, facilitated by Scoreboard, allows instructions without dependencies to execute, enhancing overall efficiency and throughput.

The structure of Scoreboard consists of several components:

1. *Instruction status.*

2. *Functional Unit status*: this indicates the state of each FU.

3. *Register result status*: Indicates which FU will write to each register. If no pending instructions will write to a register, this field is left blank.

4.5.3 Scoreboard control

The four stages of Scoreboard control are:

1. *Issue*: during this stage, instructions are decoded and checked for structural hazards. Instructions are dispatched to the functional unit if the corresponding unit is available and no other active instruction has the same destination register (avoiding WAW hazards). If a structural hazard or a WAW hazard is detected, instruction issuance is stalled until these hazards are resolved.

2. *Reading operands*: in this stage, instructions wait until there are no data hazards before reading operands. A source operand is considered available if no earlier issued active instruction will modify it, or if a functional unit is currently writing its value into a register. Once the source operands are available, Scoreboard instructs the functional unit to read the operands from the registers and begin execution. RAW hazards are dynamically resolved during this stage, enabling out-of-order execution. This model does not involve data forwarding.

3. *Execution*: the functional unit begins operations on the provided operands. Upon completing the operation, it informs Scoreboard about the execution's conclusion. FUs are

defined by their latency (the time required to complete one operation effectively) and their initiation interval (the number of cycles necessary between issuing two operations to the same FU).

4. *Write result*: after execution completion, Scoreboard checks for WAR hazards. If no WAR hazards are detected, the results are written. If a WAR hazard is identified, the instruction is stalled. The design assumes overlap between issuing instructions and writing results is permitted.

4.5.4 Summary

The core idea behind Scoreboard is to enable instructions behind a stall to proceed, allowing for the decoding, issuing, and reading of operands. This design achieves a speedup of 2.5 compared to systems without dynamic scheduling and a speedup of 1.7 by reorganizing instructions at the compiler level. However, the benefits are constrained by the slow memory (lacking cache) of CDC 6600.

Limitations of CDC 6600 Scoreboard include the lack of forwarding hardware, limitation to instructions within a basic block (leading to a small window), a limited number of FUs (particularly in integer and load and store units, leading to structural hazards), the inability to issue instructions on structural hazards, waiting for WAR hazards, and measures to prevent WAW hazards.

4.6 Tomasulo

Tomasulo is a dynamic scheduling approach designed to enable continued execution despite dependencies. Developed at IBM for the IBM 360/91, it emerged three years after the CDC 6600, aiming to achieve high performance without relying on specialized compilers.

4.6.1 Tomasulo features

In Tomasulo's design, control logic and operand buffers are decentralized, contrasting with the centralized approach of the Scoreboard algorithm. Buffers for operands are called Reservation Stations, where each instruction is represented as an entry.

Operands are replaced with values or pointers in a technique known as Register Renaming, which helps to avoid WAR and WAW hazards. Tomasulo's Reservation Stations provide more flexibility than traditional registers, enabling optimizations beyond compiler capabilities. Results are distributed to other functional units via a Common Data Bus (CDB), which carries both data and its corresponding source. Additionally, load and store operations are treated as functional units within the algorithm.

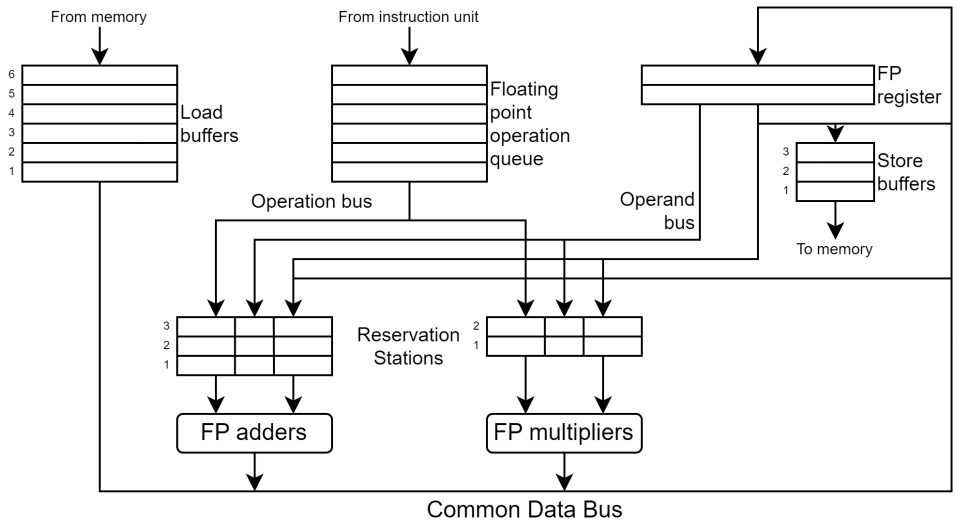


Figure 4.3: Tomasulo data structure

4.6.2 Tomasulo structure

In Tomasulo, the following components are key:

- *Reservations Stations*: these stations include several components:
  - *Tag*: identifies the Reservation Station.
  - *OP*: specifies the operation to be executed.
  - V_j, V_k: values of the source operands.
  - Q_j, Q_k: pointers to Reservation Stations producing V_j, V_k.
  - *Zero value*: indicates that the source operand is already available in either V_j or V_k.
  - *Busy*: indicates whether the Reservation Station is currently occupied.

  Note that for each operand, only one of the V-field or Q-field is valid at a time.

- *Register File and the store buffer*: both include a value and a pointer field. The pointer field indicates the Reservation Station producing the result to be stored in the Register File or store buffer. If the pointer is zero, it indicates no active instructions are producing the result, meaning the Register File or store buffer content is valid.

- *Load buffers*: these buffers include an address field and a busy field. They hold information related to memory address calculation for load and store operations. Initially, they contain the instruction offset, and after address calculation, they store the effective address.

- *Store buffers*: similar to load buffers, store buffers also feature an address field and manage store operations within the algorithm.

4.6.3 Tomasulo control

Tomasulo's algorithm unfolds across three main stages:

- *Issue*: retrieve an instruction from the instruction queue. If it's a floating-point operation, check for available Reservation Stations to avoid structural hazards. Perform Register Renaming to resolve WAW hazards, linking the Register File to instruction due to in-order issuance.
- *Execution*: execute the instruction once both operands are available. If the operands are not ready, monitor the CDB for results, delaying execution until operands are ready to avoid RAW hazards. Multiple instructions may become ready simultaneously for the same FU. For load and store instructions, perform a two-step process: compute the effective address and store it in the load or store buffer. Loads in the load buffer execute as soon as the MU is available, while stores in the store buffer wait until the value is ready before being sent to the MU.
- *Write*: once the result is available, write it onto the CDB. Distribute the result to the Register File and all Reservation Stations (including store buffers) waiting for this result. Stores write data to memory during this stage, marking Reservation Stations as available for the next instructions.

Load and store Loads and stores undergo effective address computation in a FU before proceeding to their respective load and store buffers. Loads require a second execution step to access memory before transitioning to the write result stage, sending the value from memory to the Register File and or Reservation Stations. Stores complete their execution in the write result stage by writing data to memory.

Load and store operations can be executed out of order, provided they access different memory locations. Hazards may occur if they access the same memory location. To detect such hazards, the CPU must have computed the data memory addresses associated with any earlier memory operation. When a load executes out of order with a previous store, it's assumed the address was computed in program order. nce the load address is computed, it's compared with the address fields in active store buffers. If there's a match, the load isn't sent to the Load buffer until the conflicting store completes.

Stores must check for matching addresses in both load and store buffers, a process called dynamic disambiguation, as opposed to the static disambiguation performed by the compiler. This approach requires significant hardware, including fast associative buffers in each Reservation Station, and a single CDB may limit performance.

4.6.4 Summary

Feature	**Tomasulo**	**Scoreboard**
Issue window size	14	5
Structural hazards	No	No
WAW and WAR	Implicit Register Renaming	Stalls
Control type	Distributed on Reservation Stations	Centralized
Results handling	Broadcasted from FUs	Written to registers
Loop unrolling	Yes	No

Both approaches distribute control and buffers with FUs, as opposed to centralizing them in the Scoreboard. In Tomasulo's method, Reservation Stations serve as buffers for pending operands. Registers in instructions are replaced by values or pointers to Reservation Stations through Register Renaming, avoiding WAR and WAW hazards. Tomasulo can use more Reservation Stations than registers, enabling optimizations beyond compiler capabilities.

Results are sent to FUs from Reservation Stations via a CDB that broadcasts results to all FUs. Additionally, load and store operations are treated as FUs with associated Reservation Stations. Tomasulo allows integer instructions to proceed past branches, enabling floating-point operations beyond the basic block in the floating-point queue.

4.7 Explicit Register Renaming

Explicit Register Renaming uses a physical register file larger than what the ISA specifies. The core idea is to allocate a new physical destination register for each instruction that writes a result, resembling the Static Single Assignment (SSA) form used by compilers but implemented in hardware. This approach effectively eliminates the risk of WAR or WAW hazards.

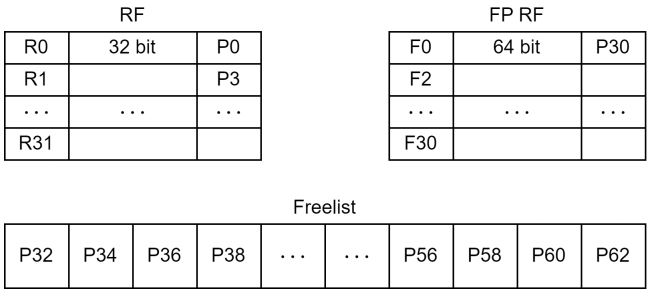


Figure 4.4: Explicit Register Renaming

Expanding the physical register file beyond the ISA register file size involves the following steps:

1. Upon issuance, each instruction that produces a result is assigned a new physical register from the available free list.

2. When a physical register is no longer needed, it is returned to the free list for future use.

Explicit Register Renaming requires maintaining a translation table that maps ISA registers to physical registers. When an instruction writes to a register, the corresponding entry in the translation table is updated with a new physical register from the pool. Physical registers are released when they are no longer used by any active instructions.

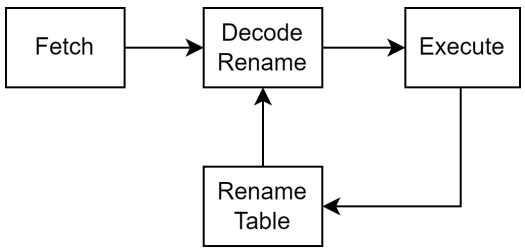


Figure 4.5: Explicit Register Renaming mechanism

In the unified physical register file approach, architectural registers are integrated into a single physical register file during the decoding stage, without accessing register values. FUs interact with this unified register file, accessing and updating both committed and temporary registers during execution. Committing an instruction involves updating the mapping of architectural registers to physical registers without moving any data.

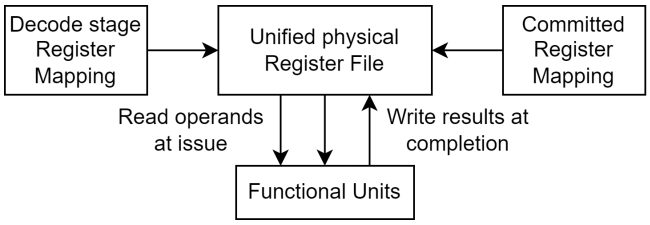


Figure 4.6: Unified physical Register File

4.7.1 Explicit Register Renaming features

Instruction commit includes the following steps:

1. Mark the mapping between an architectural register number and a physical register number as non-speculative, finalizing it.
2. Release any physical registers used to store the previous value of the architectural register.
3. De-allocating registers involves:
    - Ensuring a physical register no longer corresponds to an architectural register and has no further outstanding uses before freeing it.
    - A physical register remains associated with an architectural register until that register is overwritten.
    - Checking if any source operand in the FU queue corresponds to the register. If not, the register can be de-allocated.

    Alternatively, the processor can wait until another instruction that writes to the same architectural register commits. This method might slightly prolong the usage of a physical register but is easier to implement.

4.7.2 Explicit Register Renaming structure

Explicit Register Renaming requires specific components to manage the mapping between architectural registers and physical registers efficiently:

- *Renaming map*: this is a straightforward data structure that provides the mapping from architectural register numbers to corresponding physical register numbers. Whenever an instruction writes to an architectural register, the renaming map is updated to reflect the physical register holding the result.
- *Instruction commit*: during the commit phase of an instruction, the renaming map is updated permanently. This update indicates that the physical register holding the destination value now corresponds to the actual architectural register.

To enforce in-order commit, a ReOrder Buffer (ROB) is employed. The ROB ensures that instructions are committed in the same order they were fetched, despite possible out-of-order execution.

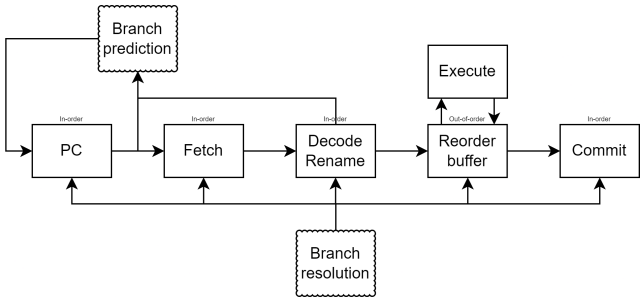


Figure 4.7: Explicit Register Renaming

Explicit Register Renaming offers several advantages:

- *Decoupling Renaming from scheduling*: this approach provides flexibility in pipeline design. The processor can adopt various scheduling methodologies, such as issuing multiple operations per cycle or implementing algorithms like Tomasulo or Scoreboard. It also allows for standard forwarding or bypassing techniques to be employed.
- *Single Register File access*: retrieval of data from a single Register File is facilitated, eliminating the need to bypass values from the ROB. This simplifies pipeline management and enhances overall efficiency.
- *Widespread adoption*: many processors have adopted variants of explicit Register Renaming. This technique has proven effective in improving performance and managing dependencies in modern CPU architectures.

Efficient access to the translation table is ensured by deploying a physical Register File with a capacity that exceeds the specifications set by the ISA. This design allows for quick determination of available physical registers, which is essential for seamless operation. In cases where no registers are available, the instruction issuance process halts temporarily until registers become free.

While Reservation Stations are not necessary for Register Renaming alone, many contemporary processors combine explicit Register Renaming with Tomasulo-like Reservation Stations to optimize control over execution flow and resource allocation.

4.7.3 Scoreboard with Explicit Register Renaming

Explicit Register Renaming involves using a physical Register File that surpasses the number of registers specified by the ISA. Here's how it operates:

- *Translation table*: a translation table is employed to maintain the mapping between ISA registers and physical registers. When an instruction writes to a register, the corresponding entry in the table is updated with a new physical register from the free list. Physical registers are marked as free when they are not in use by any active instructions.
- *Pipeline structure*: the pipeline structure can resemble a standard pipeline with stages.

In Scoreboard control stages with Explicit Renaming, the process unfolds as follows:

1. *Issue*: during this stage, instructions are decoded, and structural hazards are checked. New physical registers are allocated to store the results of instructions. Instructions are issued in program order to facilitate hazard checking. If no free physical registers are available or if a structural hazard is detected, issuance of instructions is delayed.

2. *Read operands*: this stage waits until all hazards are resolved before proceeding to read operands. Here, all dependencies (RAW hazards) are resolved as instructions wait for data to be written back.

3. *Execution*: FUs begin executing instructions as soon as they receive operands. Once execution is complete and the result is ready, notification is sent to the Scoreboard.

4. *Write result*: in this final stage, execution concludes, and the results are written to their respective destinations.

Explicit Register Renaming enhances CPU performance by effectively eliminating WAR and WAW hazards, improving the efficiency and throughput of instruction execution. This technique ensures that each instruction operates on its own set of physical registers, thereby eliminating the possibility of such hazards.

4.7.4 Summary

Comparing Explicit Register Renaming with the Reorder Buffer (ROB) reveals distinct characteristics and trade-offs:

- *Committing instructions*: Register Renaming simplifies the process of committing instructions compared to ROB. In Register Renaming, instructions are committed by updating the translation table, indicating the final architectural register for each physical register used.

- *De-allocating registers*: Register Renaming introduces complexity when de-allocating physical registers. This complexity arises because physical registers are managed dynamically based on instruction lifetimes.

- *Dynamic mapping*: the dynamic mapping of architectural to physical registers in Register Renaming adds intricacy to processor design and debugging processes. It allows for more flexible handling of data dependencies but requires careful management of register allocation and de-allocation.

- *Adoption in architectures*: Register Renaming is implemented in various architectures. These processors typically incorporate additional physical registers beyond those specified by the ISA, ranging from 20 to 80 registers.

Multiple issue Handling multiple instructions simultaneously, particularly with dependencies, is critical for dynamically scheduled superscalar processors:

- *Issue logic complexity*: designing the logic to manage dependencies among multiple instructions within a single clock cycle is highly complex. As the number of instructions issued per cycle increases, the complexity grows exponentially.

- *Basic strategy*: superscalar processors employ techniques such as assigning Reservation Stations and reorder buffer entries for each instruction in the next issue bundle. Dependencies are analyzed, and the reservation table is updated accordingly.
- *Simultaneous execution*: these operations are executed in parallel within a single clock cycle, ensuring efficient utilization of processor resources and maximizing throughput.

Superscalar Register Renaming In the context of Superscalar Register Renaming with a two-issue capability:

- *Allocation during decode*: instructions are allocated new physical destination registers during the decode stage, enabling parallel execution.
- *Operand renaming*: source operands are renamed to the physical registers holding the most recent values, reducing RAW hazards and ensuring correct data dependencies.
- *Execution unit view*: execution units exclusively use physical register numbers, simplifying the handling of data dependencies and ensuring efficient execution.
- *Hazard checking*: superscalar architectures must check for RAW hazards between instructions issued in the same cycle. This process is integrated with the operand renaming and allocation mechanisms.

Speculation and energy efficiency Speculative execution enhances performance by reducing execution time, leading to overall energy savings despite increased power consumption during speculation:

- *Power consumption*: speculation initially increases power consumption due to increased activity, but it can reduce the overall energy consumption by accelerating program execution.
- *Mis-speculation impact*: the impact of mis-speculation varies depending on the type of code. Scientific code typically experiences minimal mis-speculation, while integer code may see more substantial impacts, averaging around 30% in some cases.
- *Prediction techniques*: modern processors employ sophisticated prediction techniques for branches, data dependencies, and memory accesses. Techniques like branch prediction using structures such as the BTB and BHT help enhance performance by accurately predicting future instructions.

Final considerations Explicit Register Renaming involves using a physical Register File larger than that specified by the ISA, decoupling renaming from scheduling and offering flexibility in resolving RAW hazards. It relies on a translation table to dynamically map architectural registers to physical registers, contributing to efficient parallel execution despite the inherent complexities in managing the translation table.

Achieving efficient parallelism in real hardware remains challenging despite these advancements, highlighting the ongoing efforts in processor architecture to balance performance, power consumption, and energy efficiency.

4.8 Instruction-Level Parallelism limits

Enabling multiple instructions to start execution within each clock cycle necessitates fetching more instructions per cycle and effectively managing data and control dependencies. Superscalar processors exemplify this capability by issuing multiple instructions per clock cycle, ranging from one to eight instructions simultaneously. Instruction scheduling can be handled either by compilers or by hardware mechanisms like Tomasulo's algorithm. The success of superscalar architectures has shifted the focus from CPI to Instructions Per Clock cycle (IPC) as a primary performance metric. In an ideal scenario, where every available instruction slot is utilized in every cycle, the ideal CPI equals 1 divided by the issue width.

Ideal machine structure Assumptions for an ideal machine:

1. *Register Renaming*: assume infinite virtual registers are available, and all WAW and WAR hazards are effectively mitigated through renaming.
2. *Branch prediction*: assume perfect branch prediction with no mis-predictions, ensuring streamlined execution flow.
3. *Jump Prediction*: assume perfect prediction of jumps, facilitating flawless speculation and an infinite buffer of readily available instructions.
4. *Memory-address alias analysis*: assume perfect knowledge of memory addresses, enabling freedom to reorder stores before loads unless their addresses conflict.
5. *Uniform latency*: assume uniform one-cycle latency for all instructions across all functional units, allowing unlimited instruction issuance per clock cycle.
6. The CPU can issue an unlimited number of instructions simultaneously, with the capability to foresee computation arbitrarily far ahead.
7. There are no restrictions on the types of instructions that can be executed in a single cycle, including loads and stores.
8. All functional unit latencies are uniform at one cycle, and any sequence of dependent instructions can issue on successive cycles.
9. Perfect caches are assumed, implying all loads and stores complete in one cycle.

These assumptions yield highly optimistic results regarding ILP, as such an ideal CPU configuration is not achievable in practice.

4.8.1 Limits

Window size Dynamic analysis plays a crucial role in striving towards perfect branch prediction, especially since achieving such precision at compile time remains unattainable. For a CPU with perfect dynamic scheduling, several conditions ideally need to be fulfilled:

1. *Lookahead capability*: the CPU must possess the ability to look far ahead into the instruction stream to identify a suitable set of instructions for issuance while accurately predicting all branches.

2. *Register Renaming*: all uses of registers should undergo renaming to prevent hazards such as WAW and WAR.

3. *Dependency analysis*: the CPU should analyze potential data dependencies among instructions within the issue packet and rename registers as necessary to resolve these dependencies.

4. *Memory dependency handling*: it must detect and manage memory dependencies among issuing instructions effectively to ensure correct execution order and data coherence.

5. *FU replication*: to enable maximal instruction throughput, the CPU should provide sufficient replicated FUs so that all ready instructions can execute simultaneously.

The size of the window significantly impacts the complexity and efficiency of dynamic scheduling. In modern CPUs, constraints arise due to the limited number of registers and the requirement to search for dependent instructions while adhering to in-order issue policies. Therefore, all instructions within the window must remain within the processor's active processing range. Present-day CPUs typically feature window sizes ranging from 32 to 200 instructions. Larger window sizes increase the number of comparisons necessary to identify raw data dependencies.

CPU Several factors contribute to limiting CPU performance, including the number of FUs, buses, and ports in the Register File. These limitations determine the maximum number of instructions that can be issued, executed, or committed in a single clock cycle, often significantly fewer than the window size. Achieving the maximum possible instructions per cycle, which can reach up to six in certain cases, remains rare. Processors range in width from single-issue to configurations supporting up to six-issue executions. However, selecting the optimal 8 or 16 instructions to execute each cycle poses a formidable challenge due to the inherent complexity involved.

4.8.2 Superscalar and VLIW processors

Dynamically-scheduled superscalar processors represent the forefront of commercial general-purpose computing. Conversely, VLIW processors have found success primarily in embedded media processors for consumer electronics.

Increasing issue rates from the current typical range of 3-6 instructions per clock to 6-12 instructions would likely necessitate a processor capable of handling:

- Multiple data memory accesses per cycle (3-4).
- Branch resolutions per cycle (2-3).
- Register renaming and access to over 20 registers per cycle.
- Fetching 12-24 instructions per cycle.

Implementing these capabilities is intricate and often involves trade-offs with maximum clock rates and increased power consumption. Most performance-enhancing techniques tend to escalate power usage. The critical consideration revolves around whether a technique is energy-efficient—enhancing performance without disproportionately increasing power consumption.

Techniques employing multiple issue processors generally exhibit inefficiencies:

- Issuing multiple instructions introduces logic overhead that scales faster than the issue rate.
- This discrepancy between peak issue rates and sustained performance widens over time.
- Transistor switching correlates with peak issue rates, while sustained performance depends on a different metric, resulting in an escalating energy-per-performance ratio.

4.8.3 Summary

To advance performance in the future, we need to consider the limitations of ILP (ILP) and explore explicit parallelism that programmers can directly leverage, as opposed to relying solely on implicit parallelism exploited by compilers and hardware. However, several challenges remain:

- *Processor-memory performance gap*: this gap poses a significant hurdle to achieving further performance gains.
- *VLSI scaling problems*: challenges related to wiring and other VLSI scaling issues need to be addressed.
- *Energy and leakage problems*: increasing energy consumption and leakage present additional concerns.

Despite these challenges, alternative forms of parallelism offer potential solutions such as multicore and SIMD architectures.

CHAPTER 5

Software-level parallelism

5.1 Introduction

ILP aims to execute individual machine operations concurrently to enhance performance, transparently speeding up execution without user intervention.

In contrast, parallel processing involves employing multiple processors to handle distinct parts of a program independently, aiming to improve execution speed and efficiency, but with user awareness. To achieve performance gains beyond a single CPI using parallel processing, techniques like Superscalar architecture or Very Long Instruction Word (VLIW) are utilized.

5.1.1 Very Long Instruction Word

VLIW architecture utilizes a fixed number of instructions arranged in wide templates scheduled by the compiler. A significant milestone in VLIW development was the HP and Intel joint agreement in 1999/2000, leading to the creation of the Intel IA-64 architecture, known as Explicitly Parallel Instruction Computing (EPIC).

In VLIW, the processor can execute multiple operations per cycle, all predetermined by the compiler, contrasting with Superscalar architectures. This method reduces hardware complexity by eliminating the need for scheduling hardware and accommodating variable latency instructions. It also ensures explicit parallelism and a single control flow by prohibiting instruction reordering.

Here, an operation denotes a computation unit akin to an instruction in sequential architectures. However, instructions in VLIW encompass multiple operations slated for simultaneous execution, determined by compiler scheduling. Each VLIW instruction mandates constant operation latencies and guarantees intra-instruction parallelism, thereby obviating cross-operation RAW dependencies and data interlocks.

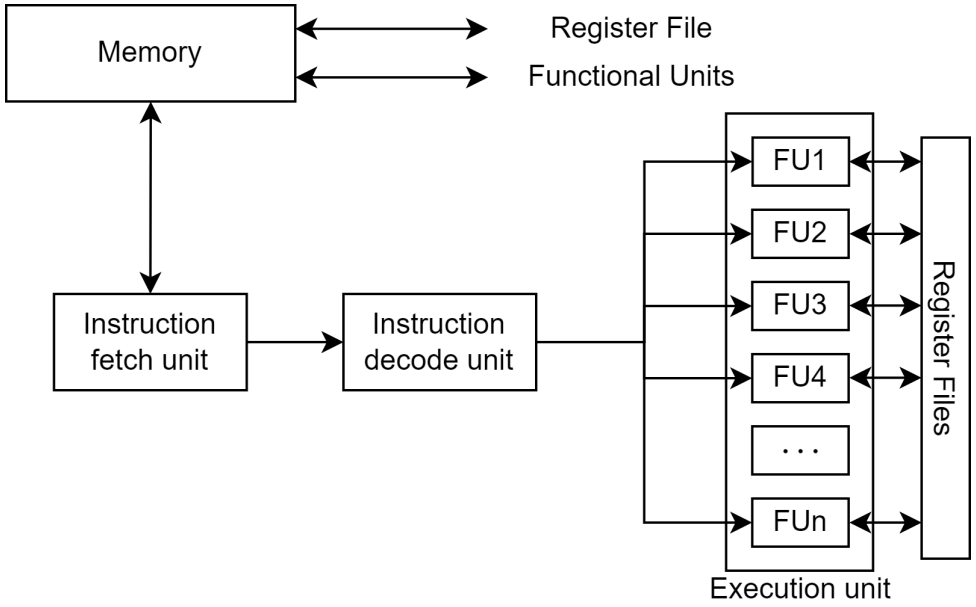


Figure 5.1: VLIW machine configuration

5.2 Very Long Instruction Word compiler

The VLIW compiler plays a crucial role in optimizing instruction scheduling to maximize parallel execution, leveraging both ILP and LLP. This entails mapping instructions to the FUs of the machine while considering timing constraints and task dependencies. The compiler ensures intra-instruction parallelism and schedules operations to prevent data hazards, often inserting explicit no-operation instructions (nop). Its primary objective is to minimize the total execution time of the program. To enhance ILP, two main strategies are employed: static and dynamic scheduling.

5.2.1 Static scheduling

Static scheduling aims to maintain a full pipeline in single-issue pipelines or utilize all FUs in each cycle within VLIW architectures.

Compilers employ advanced algorithms for code scheduling to exploit ILP effectively. Within a basic block parallelism is inherently limited by data dependencies. Therefore, maximizing performance necessitates exploiting ILP across multiple basic blocks, including those spanning branches.

Static scheduling involves the compiler identifying and resolving dependencies through code reordering. The compiler outputs dependency-free code, which is typical for VLIW processors.

Several methods are used for static scheduling, including:

- *Simple code motion*: reordering instructions to avoid stalls.
- *Loop unrolling and loop peeling*: increasing loop body size to reduce overhead and improve parallelism.
- *Software pipelining*: overlapping the execution of operations from different iterations of a loop.
- *Global code scheduling*: optimizing code across multiple basic blocks.

- *Trace scheduling*: identifying and optimizing frequently executed paths.
- *Super-block scheduling*: grouping basic blocks to optimize their execution as a single unit.
- *Hyper-block scheduling*: extending super-blocks to include predicated instructions for increased parallelism.
- *Speculative trace scheduling*: scheduling instructions based on probable execution paths, even if they involve branches.

Summary VLIW architecture benefits from straightforward hardware requirements, simplifying design and maintenance. It supports easy scalability by extending the number of FUs. Moreover, proficient compilers can detect and exploit parallelism adeptly, optimizing performance gains. However, VLIW demands a substantial register count to keep all FUs active and requires ample storage for operands and results. It necessitates high data transport capacity between FUs, Register Files, and memory, as well as high bandwidth between the instruction cache and fetch unit, resulting in larger code sizes. Challenges include binary compatibility and the need for profiling to assess branch probabilities, adding complexity to the build process. Static scheduling is also challenging for unpredictably branched code paths, as optimal schedules may vary.

5.2.2 Trace Scheduling

Trace scheduling optimizes sequences of instructions, known as traces, within a control flow graph. A trace is defined as a loop-free sequence of basic blocks that represents an execution path for specific input conditions. The execution frequency of a trace depends on the input data, with some traces being more frequently executed than others.

The process of trace scheduling begins by identifying a sequence of basic blocks that corresponds to the most commonly taken branch path. This selection is guided by profiling feedback or compiler heuristics. Once identified, the entire trace is scheduled as a cohesive unit, with additional fix-up code inserted to manage branches that exit the trace.

A significant limitation of trace scheduling is its inability to extend beyond loop boundaries. This limitation is typically addressed through techniques like loop unrolling, which allows scheduling to continue past loops but may lead to increased code size and performance overhead due to the management of loop iteration boundaries.

Despite these challenges, trace scheduling offers notable benefits by prioritizing the optimization of frequently executed paths. This approach ensures that the most commonly used traces receive optimal scheduling. During scheduling, traces are treated similarly to basic blocks, without requiring special handling for branches.

5.2.3 Code Motion

In addition to managing fix-up code, trace scheduling imposes constraints on code movement within traces. It is crucial to preserve the dataflow integrity of the program and maintain exception behavior. his preservation is achieved by addressing two primary dependencies: data dependencies and control dependencies.

To mitigate control dependencies, two effective techniques are commonly employed:

- *Predicate instructions* (hyper-block): these instructions allow bypassing branches to maintain flow within a trace.
- *Speculative instructions* (speculative): these instructions enable moving an operation before a branch conditionally, anticipating the outcome to maintain flow.

These strategies enhance scheduling efficiency within traces, optimizing performance while ensuring program correctness across varying execution scenarios.

CHAPTER 6

Thread-level parallelism

6.1 Multithreading

Modern processors often struggle to efficiently utilize their execution resources due to issues such as memory conflicts, control hazards, branch mispredictions, and cache misses. Addressing these challenges individually often provides limited effectiveness. Therefore, there is a need for a comprehensive latency-tolerance solution capable of masking all sources of latency to significantly improve performance.

6.1.1 Parallel programming

Explicit parallelism involves organizing applications into concurrent and communicating tasks. Operating systems support various types of tasks, with processes and threads being the most prevalent. Multitasking implementations vary based on the characteristics of the processor, whether it is single-core, single-core with multithreading support, or multicore.

Multithreading allows multiple threads to share the functional units (FUs) of a single processor by overlapping their execution. Each thread requires independent state duplication, including separate copies of the Register File, PC, and, for running independent programs, separate page tables. Memory sharing occurs through virtual memory mechanisms, which inherently support multiple processes. Hardware is optimized for rapid thread switching, which is significantly faster than switching between full processes. Thread switching can occur in two main ways:

- *Fine-grained multithreading*: the processor switches between threads at every instruction, interleaving the execution of multiple threads. This requires the CPU to be capable of changing threads every clock cycle, necessitating duplicated hardware resources.
- *Coarse-grained multithreading*: the processor switches between threads only during extended stalls, such as when there is a cache miss. Two threads share many system resources, and switching between them requires several clock cycles to save and restore context. This approach does not impact the throughput of a single thread under normal conditions and does not require extremely rapid thread-switching capabilities. However, it does not reduce throughput loss for short stalls, as the CPU must drain the pipeline before switching to a new thread.

6.1.2 Thread-level parallelism

ILP and TLP exploit different forms of parallelism within a program. Processors designed for ILP often experience idle FUs due to stalls or dependencies in code execution. TLP addresses this by providing independent instructions from multiple threads to keep the processor busy during these stalls, effectively utilizing otherwise idle FUs. Modern superscalar processors can leverage both ILP and TLP simultaneously.

The primary motivation behind TLP is that modern CPUs possess more functional resources than a single thread can fully utilize. Techniques like Register Renaming and dynamic scheduling enable the processor to issue independent instructions from different threads without concerns about dependencies, which are managed by hardware mechanisms.

Simultaneous Multithreading (SMT) further builds upon these concepts by leveraging existing hardware mechanisms in dynamically scheduled processors. This includes a large set of virtual registers to accommodate register sets from multiple threads, and Register Renaming to ensure unique register identifiers. Instructions from multiple threads can be interleaved in the data path without confusion over their sources and destinations across threads. Out-of-order completion enhances hardware utilization by allowing instructions to complete as soon as their operands are available.

Implementation of SMT typically involves adding a per-thread renaming table and maintaining separate Program Counters (PCs) for each thread. Independent commitment is supported by logically segregating a reorder buffer for each thread. This dynamic adaptation enables the CPU to execute instructions from each thread as needed, efficiently utilizing all available functional units. The number of threads can significantly increase the CPU's potential issue opportunities per cycle, constrained primarily by resource availability and demand imbalance.

6.2 Thread-level parallelism

Improving the performance and clock frequency of a single core has become increasingly challenging due to several factors:

- *Heat dissipation*: managing the heat generated by high-frequency operations is problematic.
- *Speed of light limitations*: physical limits on light speed affect transmission speeds in wires.
- *Design complexity*: designing and verifying deep pipelines is complex and requires large design teams.
- *Multithreaded applications*: many modern applications inherently support multithreading, demanding better support for parallel execution.

ILP architectures like superscalar and VLIW are designed to exploit fine-grained ILP but are not well-suited for supporting large-scale parallel systems. CPUs that attempt to issue multiple instructions per cycle have become exceedingly complex, with diminishing returns in extracting additional parallelism. As a result, extracting parallelism at higher levels has become more attractive.

To achieve higher performance, the next step involves process- and thread-level parallel architectures. This approach connects multiple microprocessors within a complex system to efficiently handle large-scale parallel tasks.

Definition (*Parallel architecture*)**.** A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems quickly.

The focus shifts from designing faster single processors to replicating processors to enhance performance. Parallel architecture extends traditional computer architecture by incorporating a communication framework that includes:

- *Abstractions*: defining the hardware and software interfaces.
- *Efficient structures*: developing various structures to efficiently realize these abstractions.

By leveraging parallel architectures, systems can better support multithreaded applications and efficiently handle large-scale computational tasks, overcoming the limitations of ILP and deep pipeline designs.

6.2.1 Vector processing

Vector processors are specialized for performing high-level operations on linear arrays of numbers, known as vectors. A typical vector processor comprises a pipelined scalar unit (which may be out-of-order or VLIW) alongside a dedicated vector unit. There are two primary types of vector processors:

- *Memory memory vector processors*: all vector operations involve memory-to-memory transfers.
- *Vector register processors*: vector operations occur primarily between vector registers, except for load and store operations. This type is analogous to load-store architectures.

Vector processors excel in multimedia processing tasks such as compression, graphics, audio synthesis, and image processing.

6.3 MIMD architectures

MIMD architectures have gained popularity due to their versatility. They can serve as single-user machines optimized for high performance on specific applications, act as multiprogrammed multiprocessors capable of executing many tasks concurrently, or combine these functionalities. Modern MIMD systems typically utilize standard CPUs, where each processor fetches its own instructions and operates on its own data, often employing off-the-shelf microprocessors. This scalability allows for a variable number of processor nodes, offering advantages in terms of both cost and performance. While MIMD systems excel in single-user high-performance tasks and multi-programmed environments, ensuring fault tolerance remains a critical concern.

To effectively utilize an MIMD system with n processors, there must be at least n threads or processes available for execution. These threads are either identified by the programmer or generated by the compiler, encapsulating parallelism within threads—a concept known as TLP. Threads can range from large, independent processes to parallel iterations of loops, with the degree of parallelism defined by software, unlike in superscalar CPUs where it is hardware-determined.

6.3.1 MIMD taxonomy

Existing MIMD machines are categorized into two primary classes based on the number of processors involved, influencing their memory organization and interconnection strategies:

- *Centralized shared-memory architectures*: these systems support up to several dozen processor chips. They feature large caches, multiple memory banks, and are commonly known as Symmetric Multiprocessors (SMP) with Uniform Memory Access (UMA) architecture.
- *Distributed memory architectures*: designed to support a larger number of processors, these systems require high-bandwidth interconnects. They facilitate complex data communication among processors but offer scalability beyond what centralized architectures can provide.

In MIMD architectures, memory models define how processors access and communicate with memory:

- *Shared address space*: processors communicate through shared variables in memory. Communication management is implicit, handled through load and store operations. This model is the oldest and most prevalent in parallel computing. It's important to note that shared memory does not necessarily imply a single centralized memory structure.
- *Private address space*: processors communicate by sending and receiving messages. Communication is managed explicitly through send and receive operations to access private memory. One advantage of this setup is the absence of cache coherency issues among processors, which can be a significant challenge in shared memory systems.

The MIMD memory model can be further categorized into:

- *Shared programming model*: a program consists of threads of control, which can be dynamically created during execution in certain languages. Each thread possesses a set of private variables and a set of shared variables. Threads communicate implicitly by reading and writing shared variables and synchronize on these shared variables for coordination.
- *Private programming model*: a program comprises named processes, typically fixed at startup. Each process operates within its own local address space without shared data. Logically shared data is partitioned across local processes. Processes communicate explicitly via send/receive pairs, with coordination implicit in each communication event. The Message Passing Interface (MPI) is a widely used software implementation of this model.

The choice between shared memory and message passing hinges on the specific characteristics of the program. Both models possess Turing-complete communication capabilities.

Shared memory provides advantages such as implicit communication via loads and stores, low overhead when cached, and simpler control over data placement within the caching system. However, scaling shared memory systems can be challenging, requiring synchronization operations and potentially complicating cache coherence.

On the other hand, message passing, especially in massively parallel processors, requires software to manage data layout and handle remote data access through message requests and replies. This model incurs higher software overhead due to the involvement of the operating system in message handling. While sending messages can be relatively inexpensive, receiving messages can be costly due to the necessity of polling or interrupt handling.

Bus-based symmetric shared memory Bus-based symmetric shared memory architectures are prevalent in the server market and increasingly adopted in desktop systems. They are particularly appealing for throughput servers and parallel programs due to their fine-grain resource sharing, uniform access facilitated by loads and stores, automatic data movement capabilities, and coherent cache replication. These architectures effectively extend uni-processor mechanisms for data access, offering a cost-effective and potent solution. A key feature of these architectures is the extension of the memory hierarchy to seamlessly support multiple processors.

6.3.2 Cache coherence

Shared memory machines are typically categorized into two main types: non-cache coherent and hardware cache coherent systems. Hardware cache coherent machines are capable of working with any data placement, although performance may vary. Optimization efforts often target critical sections of code. These systems utilize load and store instructions for data communication, avoiding operating system involvement and resulting in low software overhead. Special synchronization primitives are commonly used. In large-scale systems, logically distributed shared memory is implemented through physically distributed memory modules.

Shared-memory architectures cache both private data and shared data. Caching shared data reduces access latency and required memory bandwidth, while also alleviating contention during simultaneous read operations by multiple processors. However, issues arise due to private processor caches potentially containing copies of the same variable, leading to the cache coherence problem.

Coherence ensures that any read operation retrieves the most recent write to a shared location. While a strict implementation of this principle is challenging, a practical approach requires that all processors eventually see all writes in the correct order, a concept known as serialization. This coherence is maintained by following two rules:

1. If processor P writes x and processor P_1 subsequently reads x, P's write must be visible to P_1 if the read and write are sufficiently spaced apart without intervening writes to x.

2. Writes to the same location must be serialized, ensuring that all processors observe these writes in the same order.

Memory consistency further complicates this scenario, as it specifies the order of reads and writes across different memory locations. In practice:

- A write operation is not considered complete until all processors have observed its effect.

- The order of writes relative to other memory accesses remains unchanged.

In coherent multiprocessors, copies of the same data are typically present in several caches. Caches support both migration and replication of shared data items:

- *Migration*: data items can be moved to a local cache for transparent use, reducing access latency and bandwidth demand on shared memory.

- *Replication*: copies of shared data items are made in local caches to reduce latency and contention during simultaneous read operations.

To enforce cache coherence, hardware based solutions like cache-coherence protocols are essential. These protocols manage the status of shared data blocks across processors. There are two primary classes of coherence protocols: snooping protocols and directory-based protocols.

6.4 Snooping protocols

In snooping protocols, each cache controller snoops the bus to determine if it has a copy of the block requested. Each cache that holds a copy of the shared block also maintains the status of its sharing independently, without centralized state management.

Requests for shared data are broadcasted to all processors due to the caching information residing in each processor's cache. This broadcast mechanism ensures that caches are updated with the latest state of shared data.

Snooping protocols are well-suited for centralized shared memory architectures, particularly in small-scale multiprocessors with a single shared bus. This architecture allows efficient coordination and synchronization among caches without the need for centralized control, leveraging the bus as a communication medium for maintaining cache coherence.

Snoopy cache The concept of a Snoopy cache, introduced by Goodman in 1983, revolves around caches actively monitoring DMA transfers and other bus transactions to maintain coherence. The core idea is for each cache to do the right thing based on the observed bus transactions. Snoopy cache tags are designed to be dual-ported, facilitating efficient monitoring of bus transactions without significantly impacting processor performance.

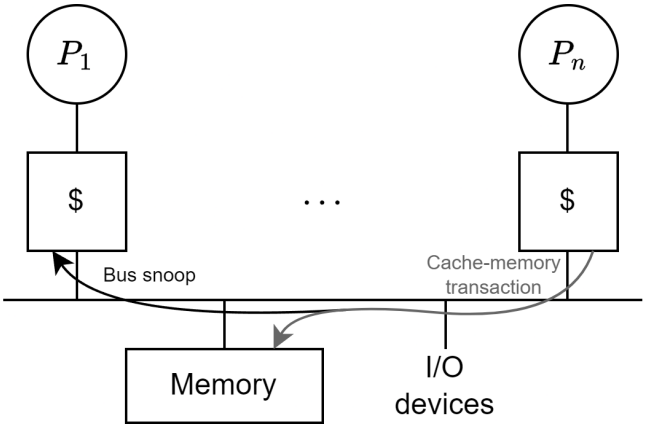


Figure 6.1: Snoopy cache

In a Snoopy cache architecture:

- The bus serves as a broadcast medium, and each cache maintains awareness of the data it holds.
- Cache controllers snoop on all transactions on the shared bus. If a transaction involves a block that a cache contains, the cache controller takes action to maintain coherence—such as invalidating, updating, or supplying data based on the block's current state and the protocol in use.

Due to the frequent checking of cache address tags during bus transactions, interference with processor operations can occur, potentially leading to stalls when the cache is unavailable.

To mitigate interference and maintain processor performance, the address tag portion of the cache is duplicated specifically for snooping activities. This duplication typically involves adding an extra read port to the address tag portion of the cache, allowing snooping operations to proceed independently of normal cache accesses.

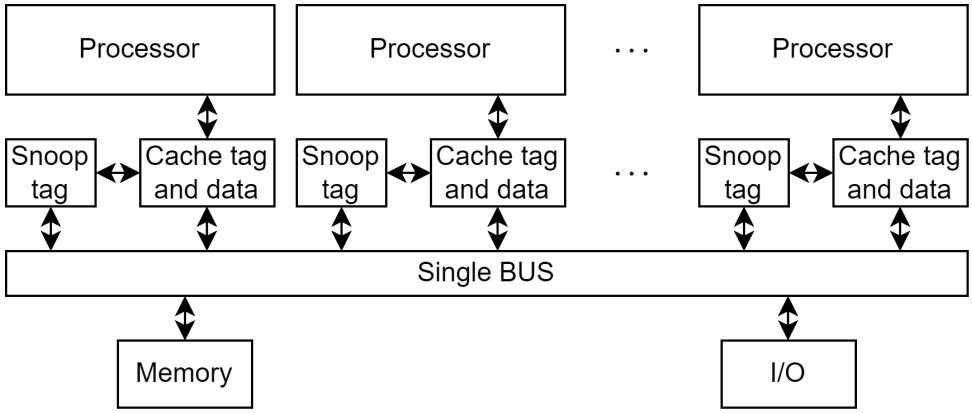


Figure 6.2: Snoop tag

Snooping protocols are categorized based on their handling of write operations:

- *Write invalidate protocol*: when a processor writes to a block, all other caches holding copies of that block are invalidated. This ensures that only one copy of the data is active at any given time.
- *Write update protocol*: when a processor writes to a block, the updated data is broadcasted to all other caches holding copies of that block. This allows multiple caches to maintain coherent copies of the data.

6.4.1 Write invalidate protocol

In the Write Invalidate Protocol, the process of updating data is managed through the following steps:

1. The writing processor initiates an invalidation signal over the shared bus, instructing all other caches to invalidate their copies of the data before it modifies its local copy.
2. After issuing the invalidation signal, the writing processor is free to update its local data until another processor requests it.
3. All caches connected to the bus check if they possess a copy of the data. If so, they invalidate the corresponding data block to maintain coherence.

This protocol supports multiple processors reading from the data concurrently but restricts writing to one processor at a time. Initially, the bus is utilized solely for the first write operation to invalidate other copies. Subsequent writes do not require bus activity unless a cache with the data block is accessed. In case of a read miss:

- *Write through*: ensures memory is always updated immediately.
- *Write back*: involves snooping in caches to locate the most recent copy of the data.

MESI protocol The MESI Protocol governs the state management of cache blocks with four distinct states:

- *Modified*: the block is dirty and exclusive to the cache holding it. It cannot be shared, and this cache is the sole holder with writable access.

- *Exclusive*: the block is clean, and only one cache has a copy. It is not shared with other caches.
- *Shared*: the block is clean, and multiple caches have copies of it.
- *Invalid*: the block contains no valid data and is not currently in use.

In both the Shared and Exclusive states, the memory holds an up-to-date version of the data. When a write operation occurs:

- Writing to an Exclusive block does not necessitate sending an invalidation signal over the bus because no other caches hold copies of the block.
- Writing to a Shared block requires invalidating all other copies of the block in the caches to maintain coherence.

The MESI Protocol optimizes cache coherence by efficiently managing the states of cache blocks, ensuring data integrity and minimizing bus traffic in shared-memory architectures.

6.4.2 Write update protocol

In Write update protocol, the writing processor broadcasts new data over the bus. Each cache checks if it holds a copy of the data. If a copy exists, it updates all copies with the new value.

This approach necessitates continuous broadcast of write operations to shared data. Unlike the write-invalidate protocol, which removes all copies except for a single local copy for subsequent writes, this protocol resembles write-through because all writes propagate over the bus to update shared data copies.

An advantage of this protocol is the reduced latency in caches, as new values appear sooner. This ensures that memory is always up-to-date, minimizing read misses.

6.4.3 Typical cache configuration

The majority of commercial cache-based multiprocessors commonly utilize:

- *Write-back caches*: these caches reduce bus traffic by allowing multiple processors on a single bus. They use a snooping scheme to determine the most recent data value of a cache block in case of a cache miss. Each processor monitors every address placed on the bus. If a processor discovers it holds a dirty copy of the requested cache block, it responds with that cache block.
- *Write invalidate protocol*: this protocol is employed to conserve bus bandwidth. It ensures write serialization due to the bus being a single point of arbitration. Therefore, a write to a shared data item cannot complete until it gains bus access.

6.4.4 Cache consistency

Definition (*System sequential consistency*)**.** A system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program.

Sequential consistency means that any execution yields the same result as if all processor operations were executed in a specific sequential order, respecting the program's specified order.

In practice, ensuring sequential consistency requires processors to observe data writes from other processors in a particular order. This consistency model guarantees that memory accesses from different processors are interleaved in a way that preserves the program's intended order.

To achieve sequential consistency, processors may delay completing memory accesses until all invalidations resulting from those accesses are finished. This ensures that all processors see a consistent view of memory at all times.

While maintaining sequential consistency can impact performance, it is typically not a significant issue for most programs that are inherently synchronized or structured to handle such consistency constraints effectively.

Synchronization A program is considered synchronized when all access to shared data is ordered by synchronization operations. Synchronization becomes necessary whenever concurrent processes exist, even in a uniprocessor system. There are two primary classes of synchronization:

- *Producer consumer*: a consumer process waits until a producer process has produced data.
- *Mutual exclusion*: ensures that only one process can use a resource at any given time.

Locks In the Sequential Consistency (SC) model, regular loads and stores (and fences in weaker models) are adequate for implementing mutual exclusion. However, this approach can lead to inefficient and complex code. As a result, atomic Read Modify Write (RMW) instructions have been introduced into Instruction Set Architectures (ISAs) to support efficient mutual exclusion.