Advanced Computer Architectures
Theory

Christian Rossi

Academic Year 2023-2024

Abstract

The course topics are:

- Review of basic computer architecture: the RISC approach and pipelining, the memory hierarchy.
- Basic performance evaluation metrics of computer architectures.
- Techniques for performance optimization: processor and memory.
- Instruction level parallelism: static and dynamic scheduling; superscalar architectures: principles and problems; VLIW (Very Long Instruction Word) architectures, examples of architecture families.
- Thread-level parallelism.
- Multiprocessors and multicore systems: taxonomy, topologies, communication management, memory management, cache coherency protocols, example of architectures.
- Stream processors and vector processors; Graphic Processors, GP-GPUs, heterogeneous architectures.

Contents

CHAPTER 1

Introduction

1.1 Architectures' classification

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

- *Single Instruction Single Data*: utilized by uniprocessor systems.
- *Multiple Instruction Single Data*: although theoretically possible, this architecture lacks practical configurations.
- *Single Instruction Multiple Data*: features a straightforward programming model with low overhead and high flexibility, commonly employed in custom integrated circuits.
- *Multiple Instruction Multiple Data*: known for its scalability and fault tolerance, this architecture is utilized by 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). Tasks are divided into a sequence of discrete instructions executed sequentially, allowing only one instruction to be processed at any given moment. This arrangement is illustrated by the single instruction single data architecture.

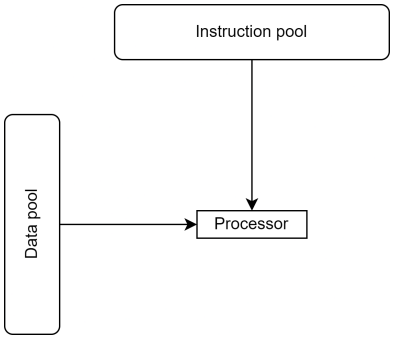


Figure 1.1: Single Instruction Single Data (SISD)

In a single instruction single data architecture:

- *Single instruction*: only one instruction is processed by the CPU in each clock cycle.
- *Single data*: 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. Single instruction single data architecture architectures represent the most prevalent type of computers.

1.1.2 Single instruction multiple data

In the single instruction multiple data architecture, the following characteristics apply:

- *Single instruction*: all processing units execute the same instruction simultaneously during each clock cycle.
- *Multiple data*: 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.

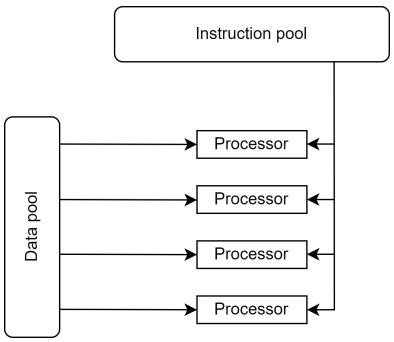


Figure 1.2: Single Instruction Multiple Data

1.1.3 Multiple instructions architectures

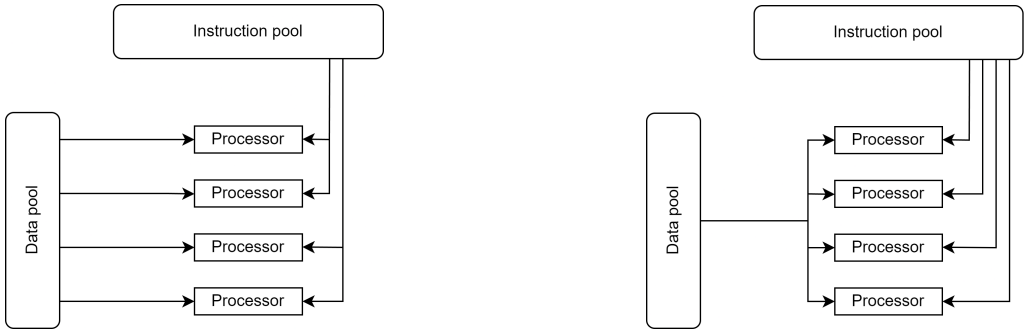
Hardware parallelism can be achieved through various methods:

- *Instruction-level parallelism*: this method harnesses 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*: these architectures leverage data-level parallelism by executing a single instruction across a set of 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 capitalizes on parallelism among largely independent tasks specified by either the programmer or the operating system.

Currently, the most common type of parallel computer features:

- *Multiple instruction*: each processor may execute a different instruction stream.
- *Multiple data*: each processor may operate with a distinct data stream.

Execution in parallel computing can occur synchronously or asynchronously, and it may be deterministic or non-deterministic.



(a) Multiple Instruction Multiple Data (b) Multiple Instruction Single Data

Figure 1.3: Possible architectures for hardware parallelism

CHAPTER 2

MIPS

2.1 Characteristics

MIPS embodies the principles of Reduced Instruction Set Computer (RISC) architecture, focusing on streamlined execution by employing simple instructions within 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:

- Loading data from memory into registers.
- Storing data from registers into memory.

A pipeline architecture is a pivotal technique aimed at performance optimization. It capitalizes on the concurrent execution of multiple instructions derived from a sequential execution flow.

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

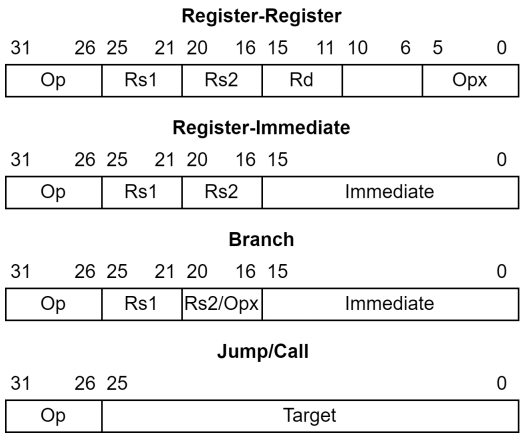


Figure 2.1: MIPS instruction set architecture

2.1.1 MIPS CPU

Within a MIPS CPU, the datapath encompasses the necessary components such as storage, functional units (FUs), and interconnects to execute desired operations effectively. In this setup, control points serve as inputs while signals serve as outputs.

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

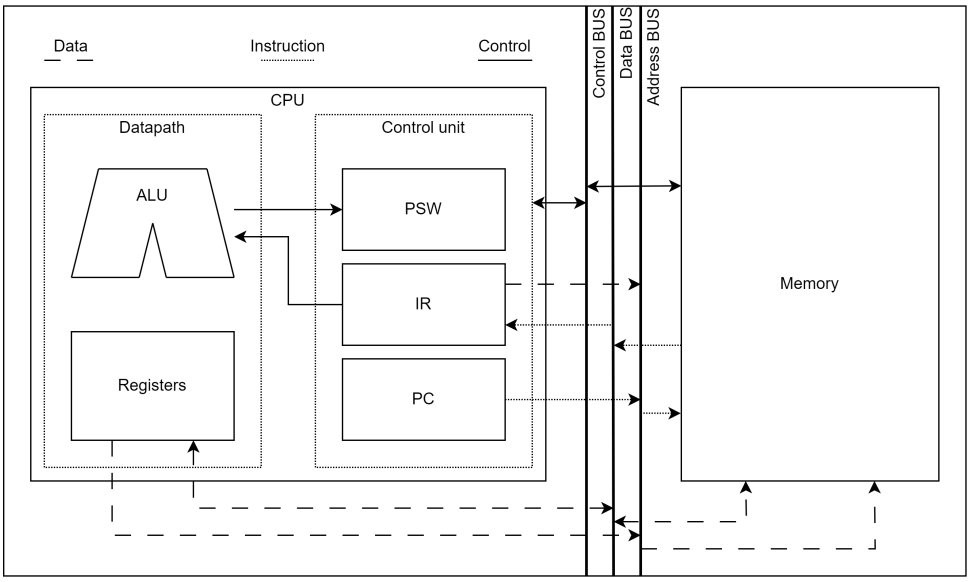


Figure 2.2: MIPS CPU

2.1.2 Program execution

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 lower-level state machines transitioning states with each cycle.

2.2 MIPS instruction execution

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

1. *Instruction fetch cycle*:
   - Transfer the content of the program counter register to the instruction memory and retrieve the current instruction.
   - Update the program counter to the next sequential address by incrementing it by 4 (since each instruction occupies 4 bytes).
2. *Instruction decode and register read cycle*:
   - Decode the current instruction using fixed-field decoding.
   - Access the register file to read one or two registers as specified by the instruction fields.

- Perform sign-extension of the offset field of the instruction if necessary.

3. *Execution cycle*:

   - For register-register ALU instructions, the ALU performs the specified operation on the operands retrieved from the register file (RF).
   - For register-immediate ALU instructions, the ALU performs the specified operation on the first operand retrieved from the RF and the sign-extended immediate operand.
   - For memory reference instructions, the ALU computes the effective address by adding the base register and the offset.
   - For conditional branches, it compares the two registers read from the RF and calculates the potential branch target address by adding the sign-extended offset to the incremented program counter (PC).

4. *Memory access*:

   - Load instructions entail a read access to the data memory using the effective address.
   - Store instructions require a write access to the data memory using the effective address to store the data from the source register read from the RF.
   - Conditional branches may update the content of the PC with the branch target address if the conditional test evaluates to true.

5. *Write-back cycle*:

   - Load instructions write the data retrieved from memory into the destination register of the RF.
   - ALU instructions store the ALU results into the destination register of the RF.

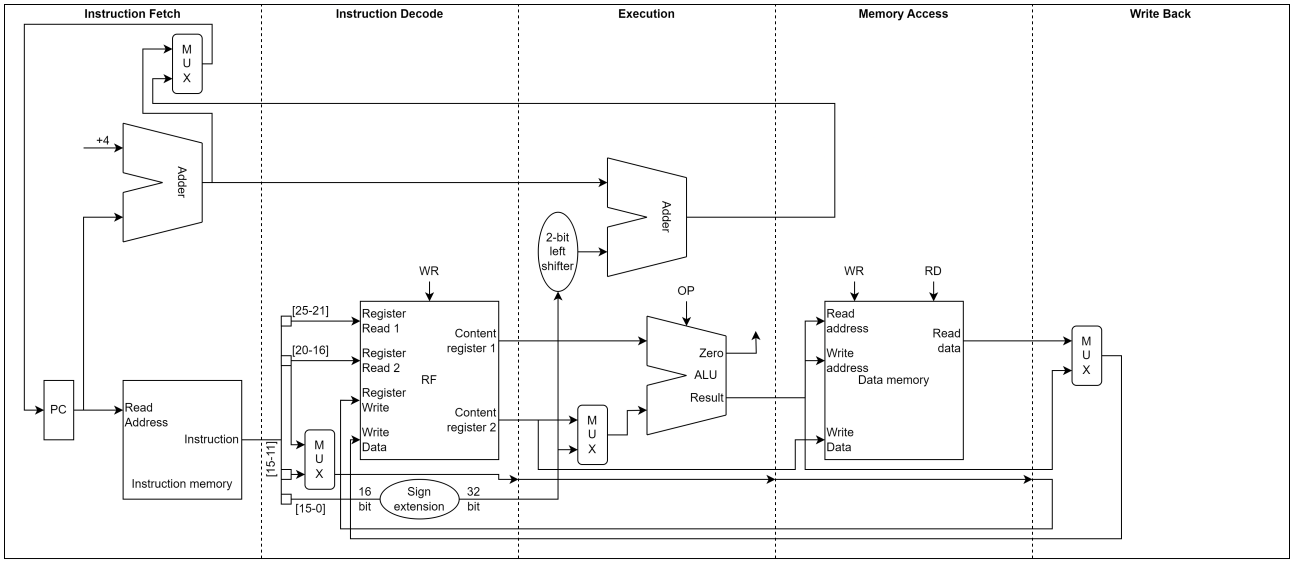


Figure 2.3: MIPS CPU architecture

Below are the durations of each instruction:

Instruction type	Instruction memory	Register read	ALU operations	Data memory	Write back	Total latency
ALU instruction	2	1	2	0	1	$6ns$
Load	2	1	2	2	1	$8ns$
Store	2	1	2	2	0	$7ns$
Conditional branch	2	1	2	0	0	$5ns$
Jump	2	0	0	0	0	$2ns$

The duration of each clock cycle is determined by the critical path established by the load instruction, denoted as $T = 8ns$ (equivalent to a frequency of $f = 125MHz$). 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. This device enables multiple inputs to access a module and allows the selection of one input among several based on the configuration of control lines.

In the multi-cycle implementation of CPU, the execution of instructions spans across multiple cycles, with MIPS typically utilizing five cycles. The fundamental cycle is shorter at $2ns$, leading to an instruction latency of $10ns$. 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.
- 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.

2.3 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: instructions enter at one end, traverse through the stages, and exit from the other end, akin to an assembly line. This facilitates a continuous flow of instruction execution, leading to improved efficiency and throughput.

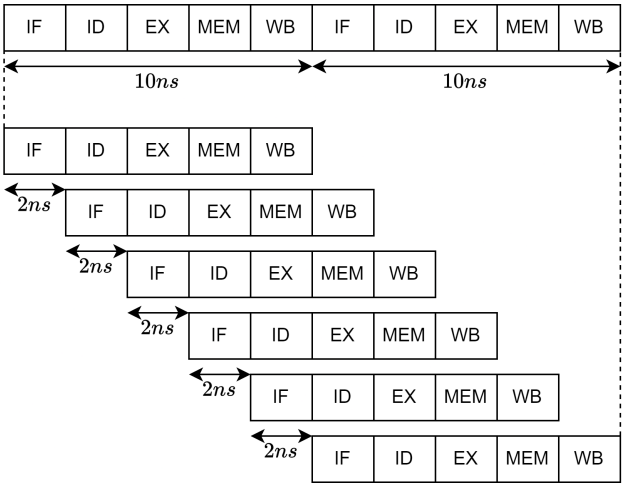


Figure 2.4: Sequential execution and pipelining execution

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 slower stage of the pipeline, typically $2ns$.

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.

In the ideal scenario, we compare a single-cycle unpipelined CPU1 with a clock cycle of $8ns$ to a pipelined CPU2 with five stages of $2ns$ each. In this case the latency (total execution time) of each instruction is increased from $8ns$ to $10ns$ due to the pipeline overhead. However, the throughput (number of instructions completed in a given time unit) is enhanced by four times: CPU1 completes one instruction every $8ns$, while CPU2 completes one instruction every $2ns$.

In the ideal scenario, when comparing a multi-cycle unpipelined CPU3 consisting of five cycles of $2ns$ each to a pipelined CPU2 with five stages of $2ns$ each. The latency (total execution time) of each instruction remains constant at $10ns$. However, the throughput (number of instructions completed in a given time unit) is enhanced by five times: CPU3 completes one instruction every $10ns$, while CPU2 completes one instruction every $2ns$.

2.3.1 Possible issues

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 write-back 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.

If dependent instructions are executed closely within the pipeline, data hazards become more prevalent.

2.3.2 Data hazards

Read after write A data hazard of the "read after write" type occurs when an instruction j attempts to read an operand before instruction i has written to it. This hazard, known as a "dependence" in compiler terminology, arises from a genuine necessity for communication between instructions. The potential solutions for mitigating this hazard include:

- *Compilation techniques*:
  - Insertion of "nop" (no operation) instructions.
  - Instruction scheduling: the compiler arranges instructions to ensure that dependent instructions are not placed too close together. It attempts to intersperse independent instructions among dependent ones. If independent instructions cannot be found, the compiler inserts "nop" instructions.
- *Hardware techniques*:
  - Insertion of "bubbles" or stalls in the pipeline.
  - Data Forwarding or Bypassing: this technique involves using temporary results stored in the pipeline registers instead of waiting for the results to be written back to the register file. Multiplexers are added at the inputs of the ALU to fetch inputs from pipeline registers, thus avoiding the need to insert stalls in the pipeline.

Utilizing forwarding allows for resolving this conflict without introducing stalls in most cases. However, for load/use hazards, it is imperative to insert one stall to properly address the issue.

Write after write A data hazard of the "write after write" type arises when instruction j writes operand before instruction i writes it. This situation results in write operations being executed in an incorrect order. Notably, this type of hazard does not occur in the MIPS pipeline since all register write operations take place in the write-back stage. Compiler writers classify this hazard as an output dependence.

Write after read A data hazard of the "write after write" type arises when an instruction j writes operand before instruction i reads it. This scenario leads to the possibility of reading an incorrect value. However, such hazards do not occur in the MIPS pipeline because operand read operations take place in the instruction decode stage, while write operations occur in the write-back stage. Similarly, assuming that register writes in ALU instructions occur in the fourth stage and that two stages are needed to access the data memory, some instructions might read operands too late in the pipeline. Compiler writers classify this hazard as an anti-dependence.

CHAPTER 3

Performance evaluation

3.1 Introduction

Developing software has grown increasingly challenging, reaching a point where manually managing all constraints has become nearly impossible. With computational power at unprecedented levels, processor cores are abundant, yet energy consumption has emerged as a critical limitation. Hence, there's a pressing need for software to be mindful of energy usage and space constraints.

3.2 Speed measures

To compare the speed of two computers we can use two metrics:

- *Computer system user* tries to minimize elapsed time for program execution:

$$\text{response time} : \text{execution time} = \text{time\_end} - \text{time\_start}$$

- *Computer center manager* tries to maximize the completion rate. In other words it tries to maximize the throughput that is the total amount of work done in a given time.

The two metrics can be compared with the formula:

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

This equality holds if there are no overlaps, otherwise we will have a greater throughput.

Usually we consider the frequent case because it is often simpler and can be done faster than the infrequent case.

Theorem 3.2.1 (*Amdahl's law*)**.** *The performance improvement of a system that can be achieved by optimizing a certain part of the system is limited by the fraction of time during which that part is actually utilized.*

Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected. We have that:

$$\text{Speedup}_{overall} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

As a result the best we could ever hope to do is:

$$\text{Speedup}_{overall} = \frac{1}{1 - \text{Fraction}_{enhanced}}$$

Example:
Consider a new CPU 10x faster. Consider a I/O bound server, so 60% time waiting for I/O. The overall speedup will be:

$$\text{Speedup}_{overall} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56$$

Apparently, its human nature to be attracted by 10X faster, vs. keeping in perspective its just 1.6X faster.

Corollary 3.2.1.1 (*Amdahl's law*)**.** *If an enhancement is only usable for a fraction of a task we can't speed up the task by more than the reciprocal of 1 minus the fraction*

3.3 Performance measures

The system's performance can be described as follows:

- *Response time*: this encompasses the latency resulting from completing a task, which includes factors like disk accesses, I/O activity, and operating system overhead. The elapsed time is calculated as the sum of CPU time and I/O wait time:

$$\text{elapsed time} = \text{CPU time} + \text{I/O wait}$$

- *CPU time*: this includes the time spent waiting for I/O operations and corresponds to the CPU's processing time. It can be calculated as follows:

$$\text{CPU time } (P) = \frac{\text{clock cycles needed to execute } P}{\text{clock frequency}}$$

$$\text{CPU time } (P) = \text{clock cycles needed to execute } P \times \text{clock cycles time}$$

SLIDE 42

CHAPTER 4

Caches and memory

4.1 Memory hierarchy

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.

During the period from 1980 to 1986, DRAM latency decreased by 9% annually, while CPU performance saw a steady increase of 1.35 times per year. Post-1986, CPU performance accelerated further to 1.55 times per year, while DRAM performance remained relatively constant.

With the advent of recent multicore processors, the design of memory hierarchy has become increasingly critical.

> **Example:**
> Consider the Intel Core i7 processor, which boasts the capability to generate two references per core clock cycle. With a total of four cores operating at a clock frequency of $3.2\,GHz$, it can achieve a good throughput. Specifically, it can handle 25.6 billion 64-bit data references per second and 12.8 billion 128-bit instruction references per second, resulting in a combined throughput of $409.6\,GB/s$. However, this remarkable processing power highlights a contrast with the DRAM bandwidth, which is merely 6% of the total throughput, amounting to a modest $25\,GB/s$.

To address this challenge, several solutions are necessary:

- Implementation of multi-port, pipelined caches to enhance data access efficiency.
- Adoption of a two-level cache structure per core to optimize data retrieval.
- Integration of a shared third-level cache directly on the chip to further streamline memory access.

In modern high-end microprocessors, the on-chip cache capacity has surpassed 10 MB, albeit at the cost of significant area and power consumption.

The ultimate goal of memory hierarchy is to create the illusion of a vast, speedy, and cost-effective memory system that 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.

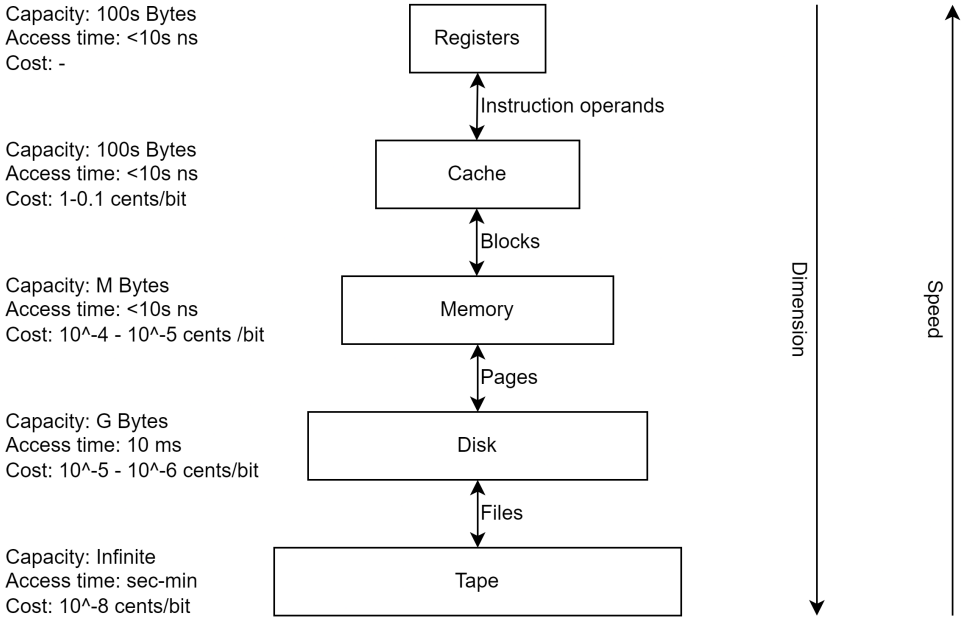


Figure 4.1: Memory hierarchy

4.2 Principle of 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*: this property suggests that if a memory location is accessed, it is probable that it will be accessed again in the near future.

2. *Spatial locality*: this property indicates that 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 predictability. 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:

- If a match is found in the cache (HIT), the data copy is retrieved from the cache and returned.

- If the address is not found in the cache (MISS), 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 encompasses 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.

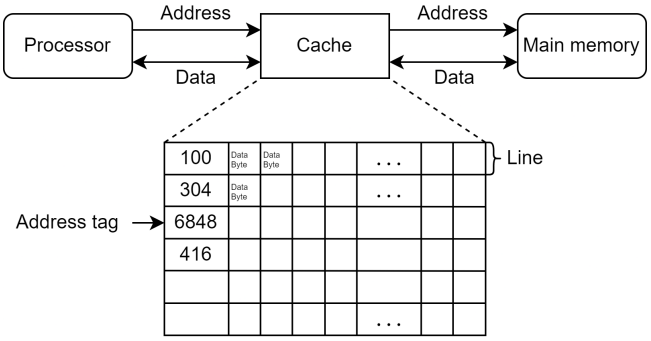
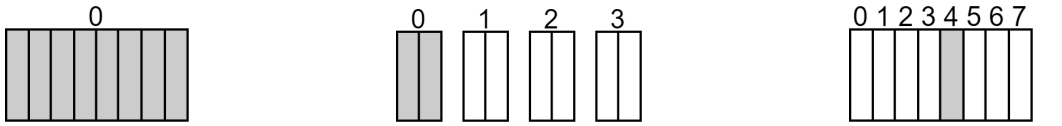


Figure 4.2: Cache and processor interaction

4.2.1 Block placement

Depending on the chosen memory type, the block numbered 12 can be positioned as follows:

- *Fully associative*: it can be placed anywhere within the memory.
- *Two-way set associative*: it can be placed anywhere within set zero, which corresponds to 12 mod 4.
- *Direct mapped*: it can only be placed into block four, determined by 12 mod 8.



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

Figure 4.3: Possible placement of blocks

In this diagram, the placement of block 12 is illustrated based on the different memory configurations mentioned above.

4.2.2 Block identification

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 is a potential solution to 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. However, some caching applications may not utilize this field.

Figure 4.4: 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.

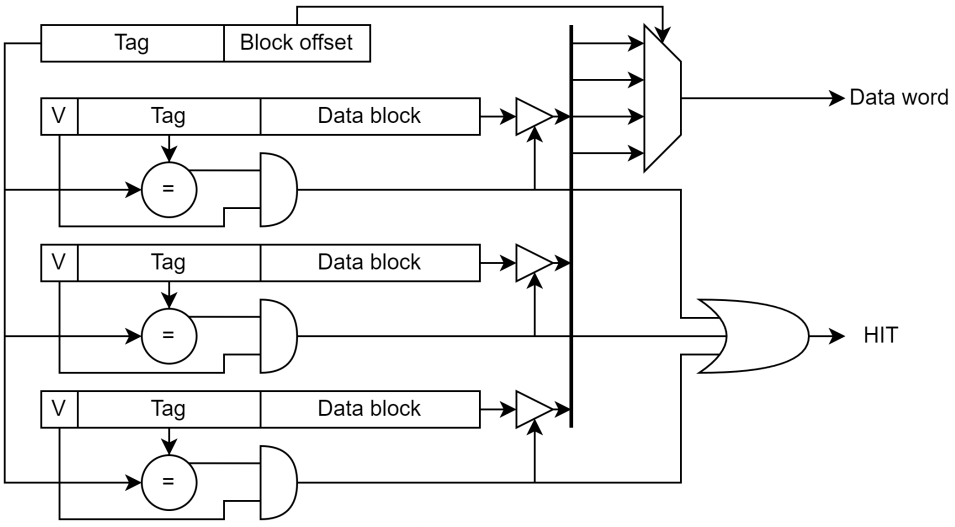


Figure 4.5: Fully associative cache

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

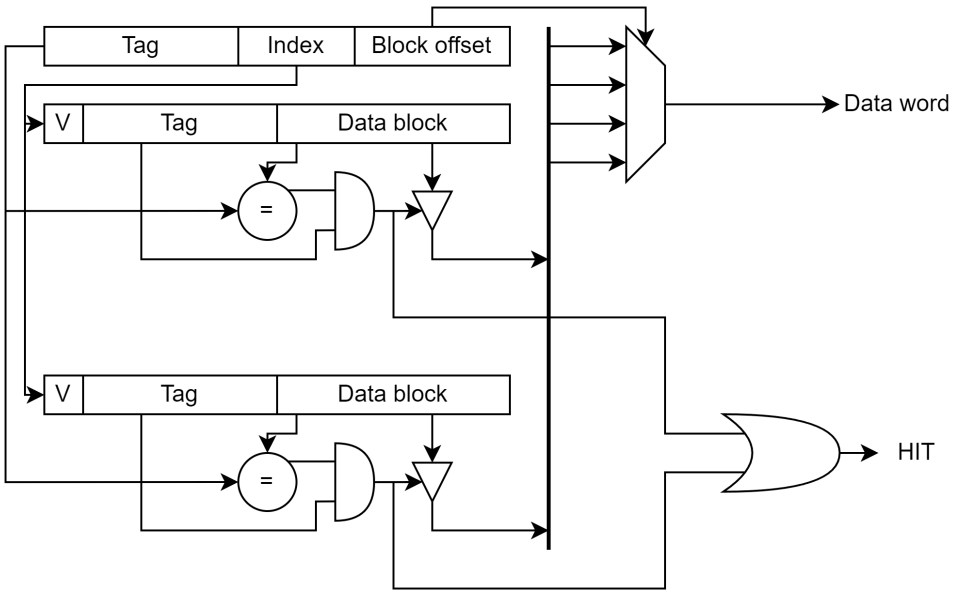


Figure 4.6: Two-way set associative cache

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

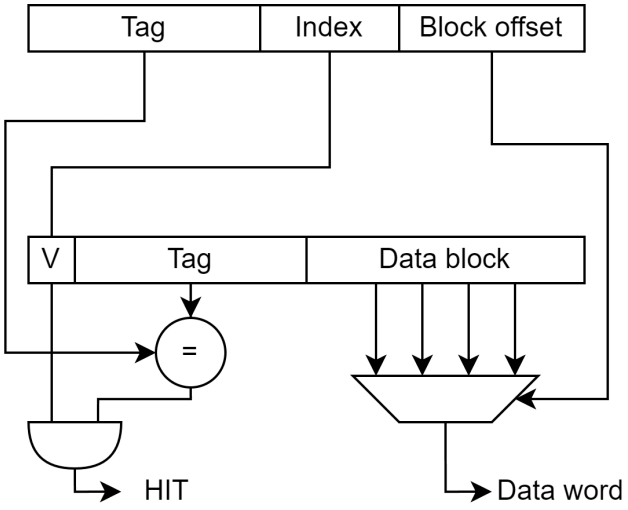


Figure 4.7: Direct mapped cache

Each type of cache has its own specific structure, with varying degrees of associativity and corresponding requirements for indexing and tagging.

4.2.3 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 because 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.

Each of these replacement policies has its advantages and trade-offs, and the choice depends on factors such as cache size, associativity, and the desired balance between complexity and performance.

4.2.4 Write strategy

In the event of a cache hit, we have two options for handling writes:

- *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, the 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, we also have two alternatives:

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

- *Write allocate* (also known as fetch on write): 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, and 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.