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.

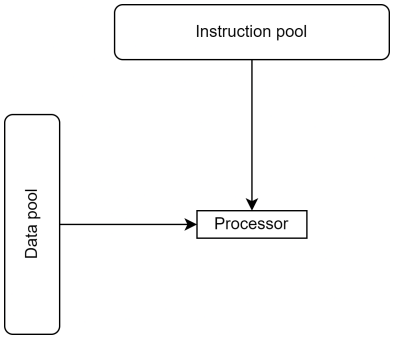


Figure 1.1: Single Instruction Single Data (SISD)

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.

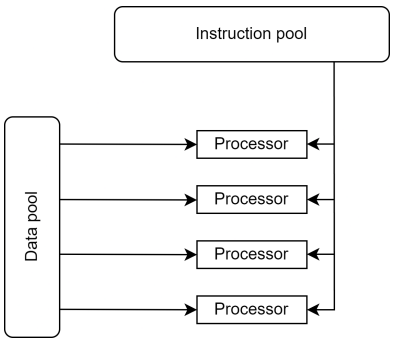


Figure 1.2: Single Instruction Multiple Data

1.1.3 Multiple instructions architectures

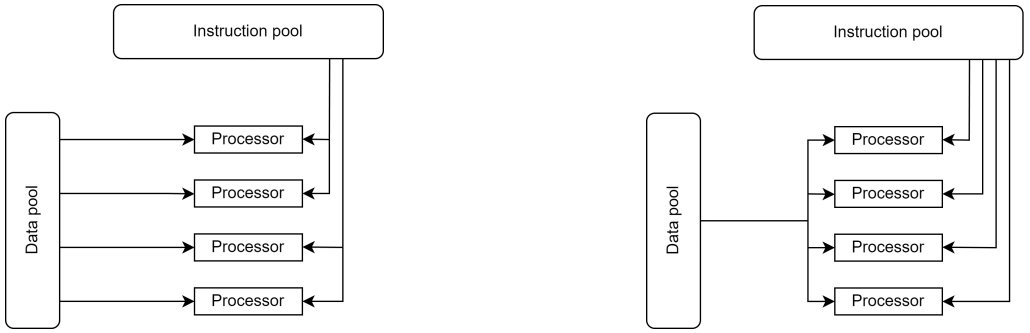
Hardware parallelism can be achieved through various methods:

- *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.

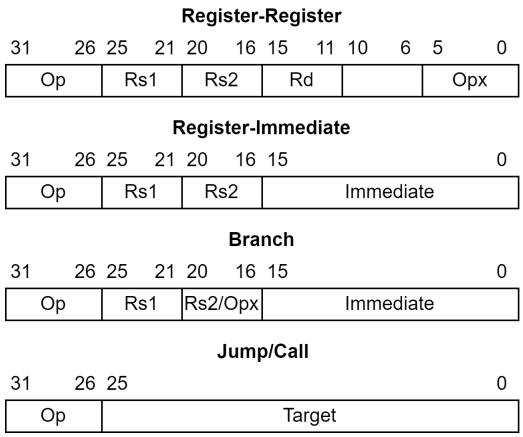


Figure 2.1: MIPS instruction set architecture

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.

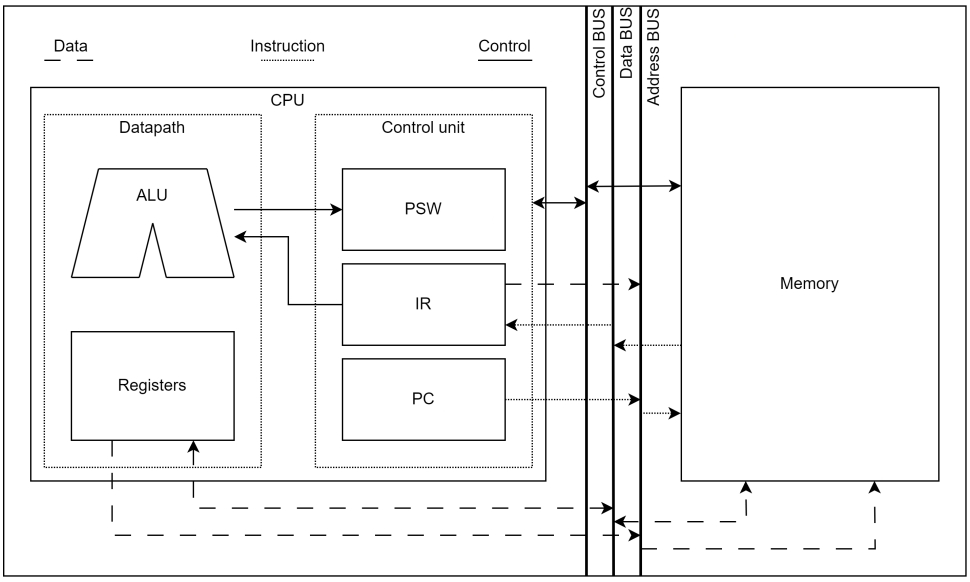


Figure 2.2: MIPS CPU

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.

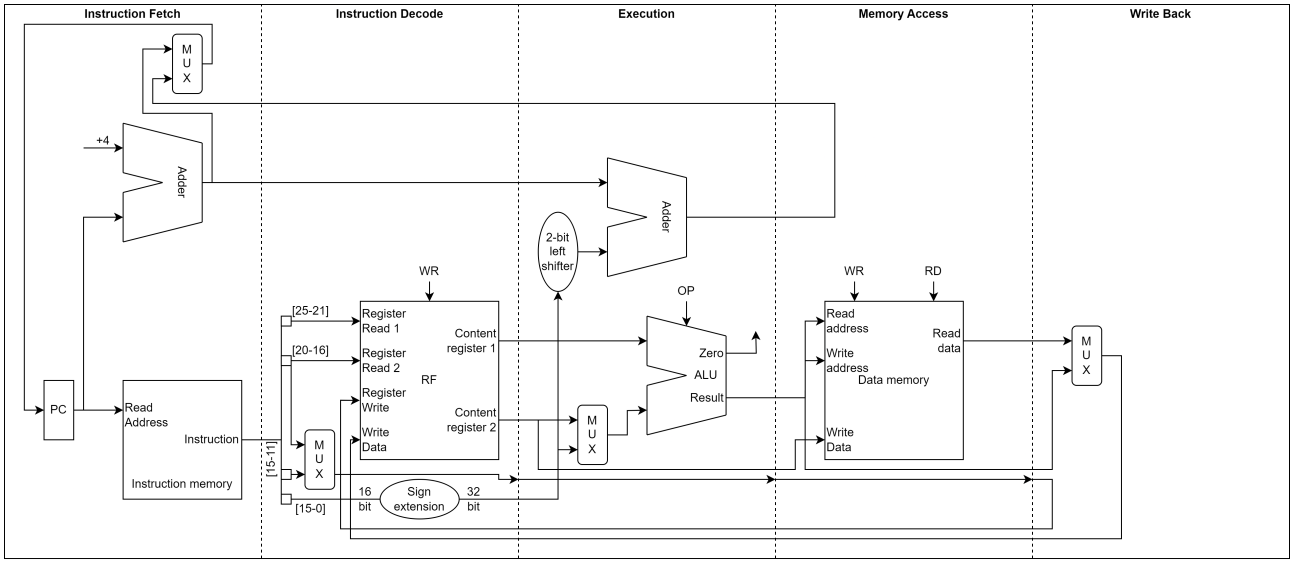


Figure 2.3: MIPS CPU architecture

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.

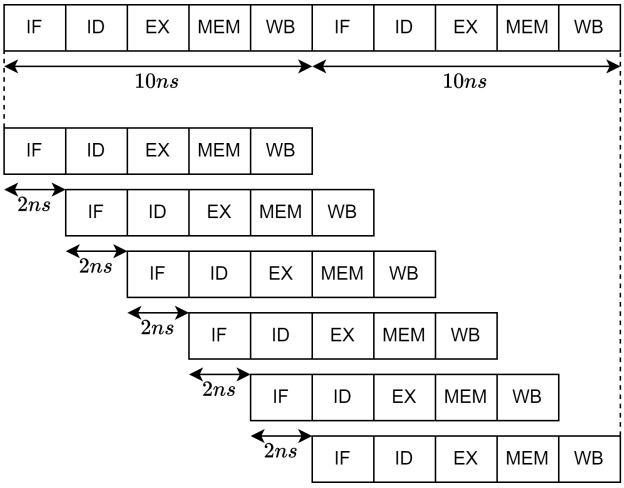


Figure 2.4: Sequential execution and pipelining execution

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

Creating software has become progressively more difficult, to the extent that manually handling all the constraints has become almost unmanageable. Despite the abundance of processor cores due to the unprecedented computational power, energy consumption has become a crucial limitation. Consequently, there is an urgent requirement for software to prioritize energy efficiency and be mindful of space constraints.

3.2 Speed measures

To evaluate the speed of two computers, we can utilize two metrics:

- *User perspective* Users aim to minimize the elapsed time for program execution, which is measured by the response time:

$$T_{response} : T_{execution} = T_{end} - T_{start}$$

- *System manager perspective*: system managers aim to maximize the completion rate, or throughput, which refers to the total amount of work done within a specified timeframe.

These metrics can be compared using the formula:

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

This equality holds true if there are no overlaps; otherwise, a greater throughput is achieved.

Typically, we focus on the common case as it's usually simpler and quicker to assess than the uncommon case.

Theorem 3.2.1 (*Amdahl law*)**.** *The performance improvement of a system achieved by optimizing a particular part of the system 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 remainder of the task remains unaffected. The overall speedup can be calculated as:

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

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

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

Example:
Considering a new CPU that is 10 times faster, and an I/O-bound server where 60% of the time is spent waiting for I/O, the overall speedup would be:

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

It's important to note that despite the allure of a 10x speed increase, the actual improvement is only 1.6x. This illustrates a fundamental aspect of human perception regarding speed improvements.

Corollary 3.2.1.1 (*Amdahl's law*)**.** *If an enhancement is only applicable for a fraction of a task, the task's speed cannot be increased by more than the reciprocal of one minus that fraction.*

3.3 Performance measures

The system's performance can be characterized by the following components:

- *Response time*: this encompasses the latency incurred while completing a task, which includes factors such as disk accesses, I/O activity, and operating system overhead. The elapsed time is determined by the sum of CPU time and I/O wait time:

$$T_{elapsed} = T_{CPU} + T_{I/O\,wait}$$

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

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

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

3.3.1 CPU time

The CPU time can be determined by the following formula:

$$T_{CPU} = \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{IC}} \cdot \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{CPI}} \cdot \underbrace{\frac{\text{Seconds}}{\text{Cycle}}}_{\text{CT}}$$

Here are the components involved:

- *Instruction count* (IC): this denotes the number of instructions executed, not the static code size. It is influenced by the algorithm, compiler, and Instruction Set Architecture (ISA).

- *Cycles per instructions* (CPI): determined by the ISA and CPU organization, this metric accounts for overlap among instructions (pipelining) which reduces this term.
- *Cycle time* (CT): determined by technology, organization, and circuit design.

The objective of CPU performance is to minimize time, which is the product of these three terms because they are interconnected.

Cycles per instruction As mentioned, cycles per instruction are defined as:

$$\text{CPI}(P) = \frac{\text{clock cycles needed to execute } P}{\text{number of instructions}}$$

We can also define the CPU time for every single instruction as:

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

Thus, better results are obtained by allocating resources to the instructions where time is spent.

3.3.2 Other metrics

Other metrics utilized to evaluate hardware performance include:

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

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

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

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

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

  Assuming floating-point operations are independent of compiler and ISA, MFLOPS can be a reliable metric for numeric codes based on the matrix size, determining the number of floating-point operations in a program. However, it's not always reliable due to factors like missing instructions or optimizing compilers.

3.4 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 microbenchmarks: these are representative program fragments that are 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.

3.4.1 System performance evaluation cooperative

The System Performance Evaluation Cooperative was established in 1989 to address bench marketing issues. SPEC2000, for example, is based on 12 integer and 14 floating-point programs, and it is utilized to evaluate the CPU, memory architecture, and compilers' compute-intensive performance.

3.4.2 Benchmark problems

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 bench marketing pressure incentivizes vendors to optimize compiler/hardware/software to specific benchmarks. Therefore, benchmarks need to be periodically refreshed.

3.4.3 Alternatives

A straightforward method for comparing relative performance is to use the total execution time of the two 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 frequencies of execution, the weighted arithmetic mean is used:

$$\left\{\sum_{i=1}^{n} \text{weight}(i) \cdot \text{time}(i)\right\} \div \frac{1}{n}$$

In general:

- Use the arithmetic mean for times.
- Use the harmonic mean if rates must be used.
- Use the geometric mean if ratios must be used.

3.5 Energy and power

Energy and power consumption pose constraints for a variety of systems, including embedded systems, IoT devices, and mobile devices due to battery capacities, as well as for desktops, servers, and HPC clusters in terms of energy costs. Consequently, it becomes necessary to establish an energy and power budget for these systems. Achieving optimization in this regard requires a thorough understanding of the underlying phenomenon.

3.5.1 Energy consumption

Thermal Design Power (TDP) serves as a metric to characterize sustained power consumption. It is used as a target for power supply and cooling systems and typically falls between peak power and average power consumption. The clock rate can be dynamically reduced to limit power consumption, and measuring energy per task often provides a more accurate assessment.

Various techniques are employed to reduce dynamic power consumption:

- 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, needs to be considered. To mitigate static power, a technique called power gating is commonly employed.

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.

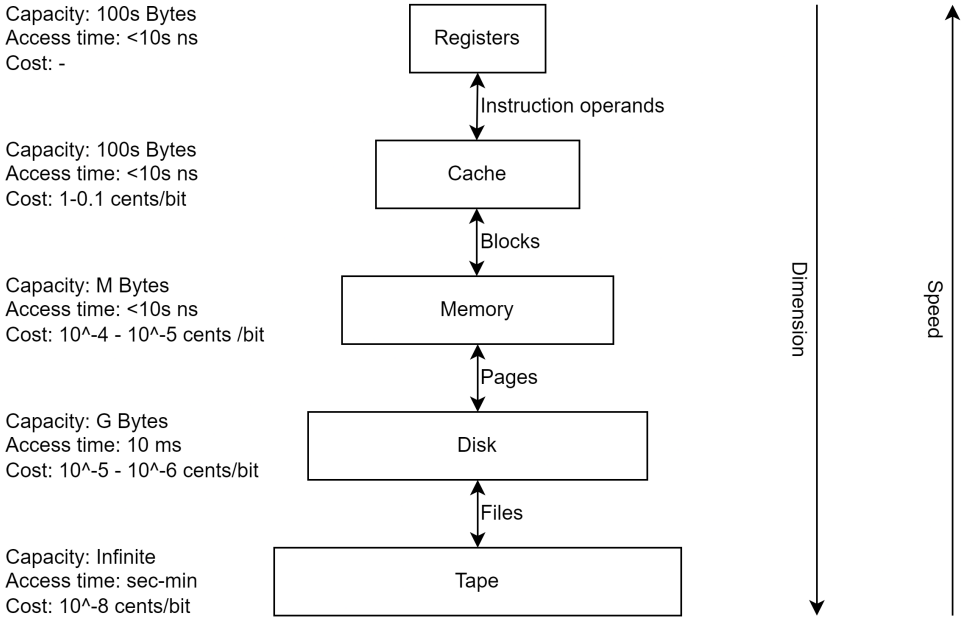


Figure 4.1: Memory hierarchy

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.

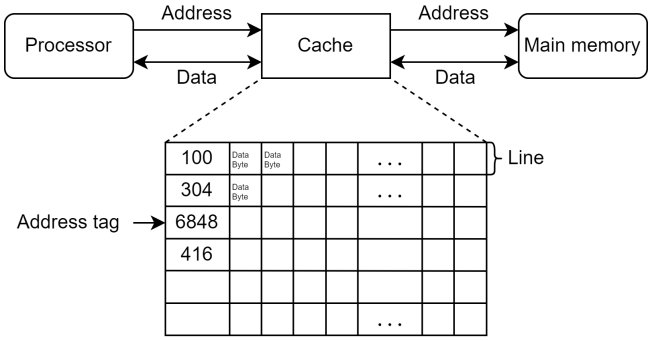
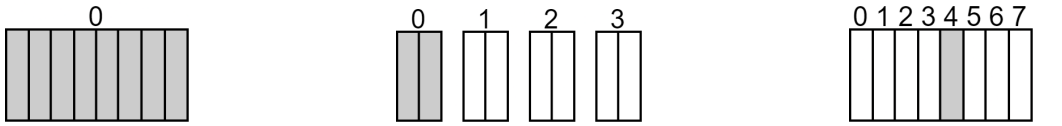


Figure 4.2: Cache and processor interaction

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.

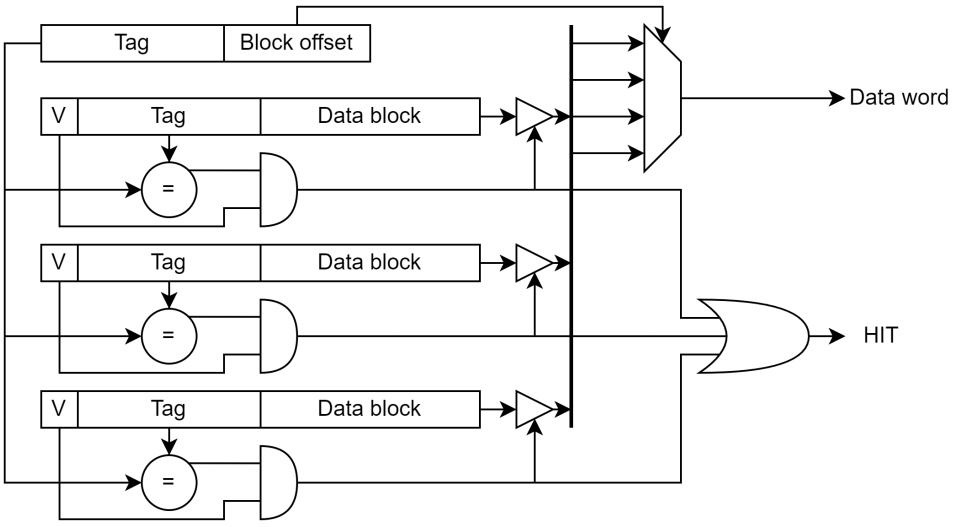


Figure 4.5: Fully associative cache

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

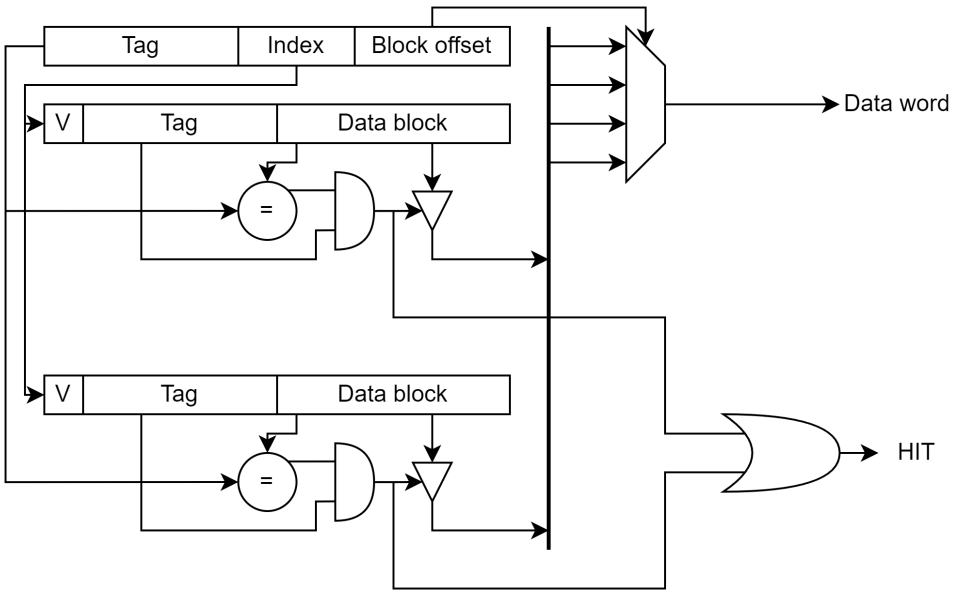


Figure 4.6: Two-way set associative cache

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

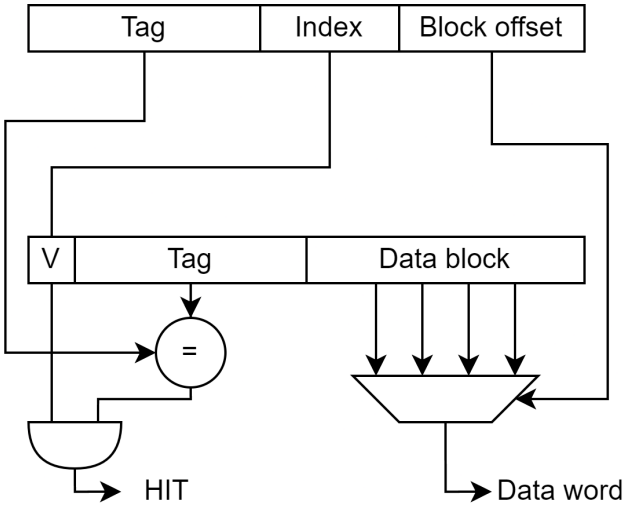


Figure 4.7: Direct mapped cache

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.

We can also use a write buffer for write-through caches to avoid CPU stalls.

4.3 Cache performance

Definition (*Memory stall cycles*)**.** Memory stall cycles is the number of cycles in which the CPU is not working (stalled) waiting for a memory access.

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 now compute:

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

$$\text{Memory stall cycles} = \text{Number of misses} \cdot \text{Miss penalty}$$

We can simplify by averaging reads and writes:

$$\text{Memory stall cycles} = \text{IC} \cdot (\frac{\text{Memory accesses}}{\text{Instruction}}) \cdot \text{Miss rate} \cdot \text{Miss penalty}$$

This value is independent of the hardware implementation, but dependent on the architecture.

On the other hand the average access time is computed as:

$$\begin{aligned} T_A &= H_{rate} \cdot H_{time} + M_{rate} \cdot M_{time} \\ &= H_{rate} \cdot H_{time} + M_{rate} \cdot (H_{time} + M_{penalty}) \\ &= H_{time} \cdot (H_{rate} + M_{rate}) + M_{rate} \cdot M_{penalty} \\ &= H_{time} + M_{rate} \cdot M_{penalty} \end{aligned}$$

4.3.1 Basic cache optimizations

The access time for a cache can be optimized by:

- Reducing M_{rate}:
  - Larger Block size (compulsory misses):
  - Larger Cache size (capacity misses):
  - Higher Associativity (conflict misses):
- Reducing $M_{penalty}$:
  - Multilevel Caches:
- Reducing H_{time}:
  - Giving Reads Priority over Writes:

4.3.2 Cache design

Within the realm of cache design, various factors interplay: cache size, block size, associativity, replacement policy, and the choice between write-through and write-back mechanisms. Determining the best option involves finding a balance influenced by access patterns (workload and usage) and technological expenses. Often, simplicity emerges as the preferred solution.

The performance metrics used to evaluate the performance are:

- Latency is concern of cache.
- Bandwidth is concern of multiprocessors and I/O.
- Access time: time between read request and when desired word arrives.
- Cycle time: minimum time between unrelated requests to memory.

DRAM used for main memory, SRAM used for cache.

SRAM SRAM memory requires low power to retain bit and requires six transistors for each bit.

DRAM The DRAM must be re-written after being read and also periodically refreshed (each row simultaneously every eight milliseconds). However, this type of memory requires only one transistor per bit. The address lines are multiplexed:

- Upper half of address: row access strobe (RAS).
- Lower half of address: column access strobe (CAS).

Flash memory Flash memory belongs to the category of EEPROM, necessitating block erasure prior to overwrite. It retains data without power, constituting non-volatile storage. With a finite number of write cycles, it falls in price between SDRAM and disk storage. Although slower than SRAM, it outpaces traditional disk speeds.

Optimizations The Amdahl law states that the memory capacity should grow linearly with processor speed. Unfortunately, memory capacity and speed has not kept pace with processors. To overcome this issue some optimizations are possible:

- Multiple accesses to same row.
- Synchronous DRAM: added clock to DRAM interface, and burst mode with critical word first.
- Wider interfaces.
- Double data rate (DDR).
- Multiple banks on each DRAM device.

4.4 Virtual memory

Virtual memory is utilized to confine processes within their allocated memory space boundaries. This architecture serves multiple functions:

- It facilitates user mode and supervisor mode.
- It safeguards specific CPU state components.
- It incorporates mechanisms for transitioning between user mode and supervisor mode.
- It provides tools for restricting memory accesses.
- It includes a Translation Lookaside Buffer (TLB) for address translation.

4.4.1 Virtual machines

Virtual memory's concept enables the creation of virtual machines. These machines support isolation and security, allowing multiple unrelated users to share a computer. This capability is made feasible by the processors' raw speed, which mitigates the associated overhead.

Virtual machines enable the presentation of different Instruction Set Architectures (ISAs) and operating systems to user programs.

The software responsible for system virtual machines is called a hypervisor, with individual virtual machines operating under it referred to as guest virtual machines. Each guest operating system maintains its set of page tables:

- The hypervisor introduces a layer of memory between physical and virtual memory, termed real memory.
- The hypervisor maintains a shadow page table mapping guest virtual addresses to physical addresses. This necessitates the hypervisor's ability to detect changes made by the guest to its page table, which naturally occurs if accessing the page table pointer is a privileged operation.

CHAPTER 5

Exception handling

5.1 Introduction

Definition (*Interrupt*)**.** An interrupt refers to an external or internal event necessitating processing by another system program.

Typically unexpected or infrequent from the program's perspective, interrupts can stem from various causes:

- *Asynchronous*: stemming from external events, such as input/output device service requests, timer expirations, power disruptions, or hardware failures.
- *Synchronous*: arising from internal events (exceptions), including undefined opcodes, privileged instructions, arithmetic overflows, FPU exceptions, misaligned memory access, virtual memory exceptions (such as page faults, TLB misses, and protection violations), and traps (system calls).

5.1.1 History

The first system to incorporate exceptions was the Univac-I in 1951, where an arithmetic overflow would either trigger the execution of a two-instruction fix-up routine at address 0 or, optionally, cause the computer to halt.

The Univac 1103, modified in 1955, introduced external interrupts for gathering real-time wind tunnel data.

The DYSEAC in 1954 was the first system with I/O interrupts, featuring two program counters, and an I/O signal facilitated switching between them. Additionally, it was the first system to incorporate DMA (Direct Memory Access) by I/O devices.

5.2 Taxonomy

Exceptions can be categorized as follows:

- *Synchronous* or *asynchronous*: asynchronous events stem from devices external to the CPU and memory, manageable after the current instruction completes (easier to handle).

- *User requested* or *coerced*: user requested exceptions are predictable and treated similarly to exceptions, utilizing the same mechanisms for saving and restoring state; they are handled after instruction completion. Coerced exceptions arise from hardware events beyond the program's control.
- *User maskable* or *user nonmaskable*: the mask dictates whether the hardware responds to the exception.
- *Within instructions* or *between instructions*: exceptions occurring within instructions are typically synchronous as the instruction initiates the exception. The instruction must halt and restart. Asynchronous exceptions between instructions arise from critical situations, leading to program termination.
- *Resume* or *terminate*: terminating events result in program execution always halting after the interrupt. Resuming events allow program execution to continue after the interrupt.

5.3 Interrupts

Asynchronous interrupt In case of asynchronous interrupt an I/O device signals the need for attention by activating a prioritized interrupt request line. Upon the processor's decision to handle the interrupt:

1. Execution halts at instruction I_i of the current program, ensuring completion of all instructions up to I_{i-1} (precise interrupt).
2. The processor stores the program counter (PC) value of instruction I_i in a dedicated register (EPC).
3. Interrupts are disabled, and control shifts to a specified interrupt handler operating in kernel mode.

The interrupt handler:

- Before enabling interrupts to accommodate nested interrupts:
  - It saves the program counter (PC) to facilitate nested interrupts.
  - Requires an instruction to transfer the PC into general-purpose registers (GPRs).
  - Requires a mechanism to temporarily block further interrupts until the PC is saved.
- It retrieves information about the interrupt cause from a designated status register.
- Utilizes a specialized indirect jump instruction called Return-From-Exception (RFE), which:
  - Enables interrupts.
  - Restores the processor to user mode.
  - Reinstates 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. This necessitates undoing the impact of one or more partially executed instructions. However, in the scenario of a system call trap, the instruction is deemed as fully executed. This involves a special jump instruction that transitions to privileged kernel mode.

5.3.1 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 you can effectively resume execution from the interrupt point and obtain the correct outcome.

Precise interrupts are desirable for several reasons. They facilitate the restart of various interrupt and exception types. Additionally, 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: less state must be preserved when unloading processes, and restarts are quicker.

5.4 Exception handling in five stage pipelines

Exceptions may arise at various stages within the pipeline. However, the recommended approach for interrupt handling is to minimize pipeline interruption as much as possible.

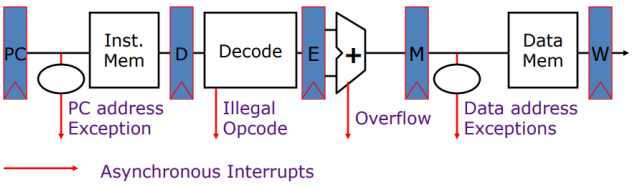


Figure 5.1: Exception origins

To address this, instructions in the pipeline can be tagged to indicate whether they cause exceptions or not. 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. However, managing interrupt conditions becomes complex due to requirements such as supervisor mode switching and saving one or more program counters (PCs).

Additionally, optimizing instruction fetch to start fetching instructions from the interrupt vector may be challenging due to these complexities.

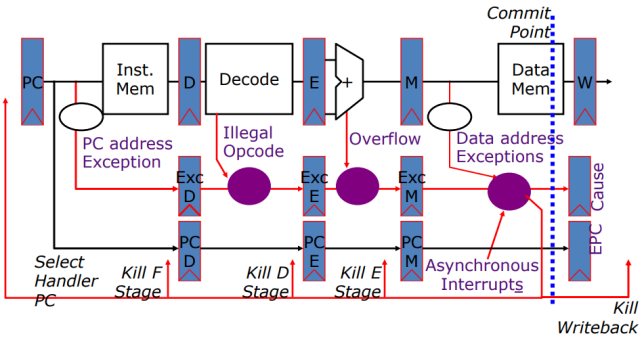


Figure 5.2: Exception handling

Maintain exception flags within the pipeline until reaching the commit point (M stage). 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, cause and EPC registers are updated, all pipeline stages are terminated, and the handler PC is injected into the fetch stage.

Jim Smith's paper explores various techniques for achieving precise interrupts, including in-order instruction completion, reorder buffer, and history buffer methodologies.

5.4.1 Speculations on exceptions

The possible methods to anticipate the exceptions are:

- *Prediction mechanism*: due to 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, enabling 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 6

Branch prediction

6.1 Introduction

Every branch incurs a single stall to fetch the correct instruction flow: either the (PC+4) or the branch target address. Our goal is to predict the outcome of a branch instruction as early as possible to enhance performance. The performance of a branch prediction technique depends on three key factors:

1. *Accuracy*: this is determined by the percentage of incorrect predictions made by the predictor.
2. *Cost of misprediction*: this refers to the time lost executing unnecessary instructions due to an incorrect prediction (misprediction penalty). This cost increases notably in deeply pipelined processors.
3. *Branch frequency*: the frequency of branches within the application plays a significant role. Accurate branch prediction is particularly crucial in programs with higher branch frequencies.

Various methods address the performance degradation resulting from branch hazards:

- *Static branch prediction techniques*: predefined actions for each branch remain constant throughout program execution, determined at compile time.
- *Dynamic branch prediction techniques*: decisions leading to branch prediction may alter during program execution.

In both approaches, caution is essential to avoid altering the processor state until the branch outcome is definitively determined.

6.2 Static branch prediction techniques

Static Branch Prediction is employed in processors where it's anticipated that the branch behavior remains highly predictable at compile time. Additionally, Static Branch Prediction can complement dynamic predictors. Several techniques fall under Static Branch Prediction:

- *Branch always not taken* (predicted-not-taken).
- *Branch always taken* (predicted-taken).
- *Backward taken forward not taken* (BTFNT).
- *Profile-driven prediction.*
- *Delayed branch.*

6.2.1 Branch always not taken

In Static Branch Prediction, when assuming that the branch will not be taken, the sequential instruction flow that has been fetched can proceed as if the branch condition was not met.

- If the condition in the instruction decode (ID) stage indicates that the branch will not be taken (correct prediction), performance is preserved.
- However, if the condition in the ID stage indicates that the branch will be taken (incorrect prediction), the next instruction that has already been fetched must be flushed (turned into a nop), and execution restarts by fetching the instruction at the branch target address, incurring a one-cycle penalty.

6.2.2 Branch always taken

An alternative approach is to treat every branch as taken. Once the branch is decoded and the branch target address is computed, we assume the branch is taken and immediately commence fetching and executing at the target.

This predicted-taken scheme is advantageous for pipelines where the branch target is ascertainable before the branch outcome. However, in the MIPS pipeline, the branch target address isn't determined earlier than the branch outcome. Hence, there is no benefit in adopting this approach for this particular pipeline.

6.2.3 Backward taken forward not taken

The prediction relies on the direction of the branch:

- Backward-going branches are anticipated as taken.
- Forward-going branches are expected as not taken.

6.2.4 Profile-driven prediction

Branch prediction relies on profiling information gathered from previous executions. This approach can incorporate compiler hints to enhance prediction accuracy.

6.2.5 Delayed branch

The compiler statically arranges an independent instruction in the branch delay slot. The instruction in the branch delay slot executes regardless of whether the branch is taken or not. Assuming a branch delay of one cycle (as in MIPS), there is typically only one delay slot. While some deeply pipelined processors may have longer branch delays, most processors with delayed branches feature a single delay slot, as it is often challenging for the compiler to fill more than one delay slot.

For MIPS, the compiler consistently schedules an independent instruction after the branch. For instance, a preceding add instruction without any impact on the branch may be placed in the branch delay slot.

The behavior of the delayed branch remains consistent irrespective of whether the branch is taken or not:

- If the branch is untaken, execution 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 the instruction placed in the branch delay slot is valid and beneficial. There are three typical strategies for scheduling the branch delay slot:

1. *From before*: an independent instruction from before the branch is placed in the branch delay slot. The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not.
2. *From target*: the branch delay slot is filled with an instruction from the target of the branch. This is often accompanied by using a register (e.g., $1) in the branch condition to prevent certain instructions (e.g., add) from being moved after the branch. This strategy is preferred for branches that are likely to be taken, such as loop branches (backward branches).
3. From fall-through: The branch delay slot is occupied by an instruction from the fall-through path, where the branch is not taken. Similar to the from target strategy, this may involve using a register (e.g., $1) in the branch condition to control instruction placement. This strategy is favored for branches that are unlikely to be taken, such as forward branches.

For optimizations to be valid in both target and fall-through cases, it must be acceptable to execute the moved instruction when the branch takes an unexpected direction. This means that while the instruction in the branch delay slot is executed, the work done by it is wasted, yet the program still executes correctly. For instance, if the destination register is an unused temporary register when the branch takes an unexpected direction.

Generally, compilers are able to fill approximately 50% of delayed branch slots with valid and useful instructions, with the remaining slots filled with nops. However, in deeply pipelined processors where delayed branches span multiple cycles, it becomes more challenging to populate all slots with useful instructions. The primary constraints on delayed branch scheduling stem from:

- Limitations on the instructions that can be scheduled in the delay slot.
- The compiler's capability to accurately predict the branch outcome statically.

To enhance the compiler's ability to populate the branch delay slot, many processors have introduced a canceling or nullifying branch. This instruction indicates the predicted branch direction:

- When the branch behaves as predicted, the instruction in the delay slot executes normally.
- However, if the branch prediction is incorrect, the instruction in the delay slot is replaced with a nop (flushed).

This approach enables the compiler to be less conservative when filling the delay slot.

6.3 Dynamic branch prediction techniques

The main idea of dynamic branch prediction techniques is to leverage past branch behavior to forecast future outcomes.

We utilize hardware to dynamically anticipate branch outcomes, where predictions adapt based on real-time branch behavior during execution.

Initially, we implement a basic prediction scheme and then explore methods to enhance prediction accuracy. Dynamic branch prediction integrates two key mechanisms:

- *Branch outcome predictor*: determines the likelihood of branch direction (taken or not taken) 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 Program Counter (PC) increments.
- In the case of a taken branch, the BTP provides the target address.

6.3.1 Branch outcome predictor

Branch history table The branch history table is a table featuring one bit for each entry, indicating whether a branch was recently taken or not. It is indexed by the lower portion of the branch instruction's address.

Prediction involves assuming the correctness of a hint, initiating fetching in the predicted direction. Should the hint prove incorrect, the prediction bit is flipped and stored anew. Subsequently, the pipeline is cleared, and the accurate sequence is executed.

This table lacks tags, resulting in every access being a hit. Moreover, it's plausible for the prediction bit to have been set by another branch sharing the same lower-order address bits; however, this doesn't impact prediction accuracy as it merely serves as a hint.

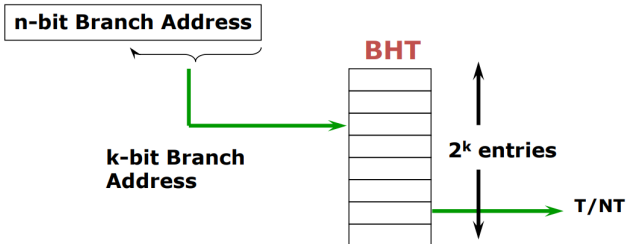


Figure 6.1: Branch history table structure

Accuracy A misprediction arises when either the prediction is erroneous for a particular branch or when the same index has been referenced by two distinct branches, and the previous history pertains to the other branch. To mitigate this issue, increasing the number of rows in the BHT or employing a hashing function (e.g., as in GShare) can be effective solutions.

One-bit branch history table The 1-bit BHT exhibits a notable limitation in scenarios such as loop branches. Even if a branch is predominantly taken throughout a loop but is not taken once, the 1-bit BHT may mispredict twice instead of once. This scheme results in two erroneous predictions:

- At the conclusion of the loop iteration, where the prediction bit indicates a taken branch, contradicting the need to exit the loop.
- Upon re-entering the loop, following the first loop iteration's end, the branch should be taken to maintain loop execution. However, the prediction bit suggests exiting the loop, stemming from the previous execution of the loop's final iteration where the prediction bit was flipped.

6.3.2 Branch target predictor

Branch target buffer Branch target buffer (BTB), also known as branch target predictor, functions as a cache that stores the anticipated branch target address for the instruction following a branch. During the Instruction Fetch (IF) stage, the BTB is accessed using the instruction address of the fetched instruction, which could potentially be a branch, to index the cache. Typical entries in the BTB include:

- The exact address of a branch.
- The predicted target address.

The predicted target address is typically represented as PC-relative.

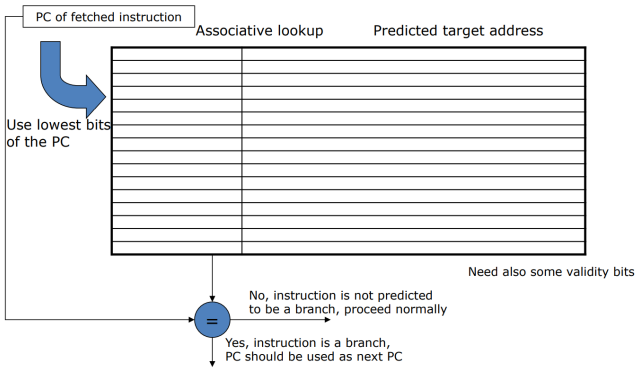


Figure 6.2: Branch target buffer structure