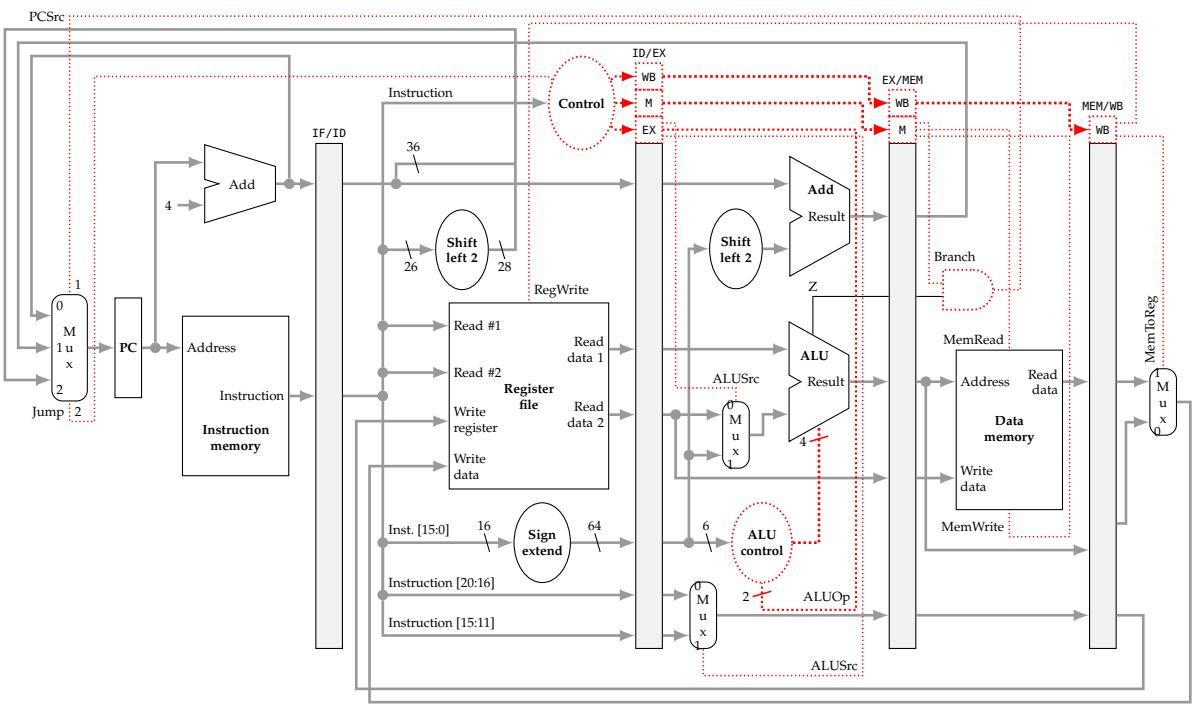
Juan Carlos Granda Candás
José María López López
Manuel García Vázquez
Julio Molleda Meré
Rubén Usamentiaga Fernández
Joaquín Entrialgo Castaño
Francisco Javier de la Calle Herrero

Computer Architecture



Textos universitarios ediuno

COMPUTER ARCHITECTURE

COMPUTER ARCHITECTURE

Juan Carlos Granda Candás
José María López López
Manuel García Vázquez
Julio Molleda Meré
Rubén Usamentiaga Fernández
Joaquín Entrialgo Castaño
Francisco Javier de la Calle Herrero

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Servicio de Publicaciones de la Universidad de Oviedo
ISNI: 0000 0004 8513 7929
Edificio de Servicios - Campus de Humanidades
33011 Oviedo - Asturias
985 10 95 03 / 985 10 59 56
servipub@uniovi.es
www.publicaciones.uniovi.es

ISBN: 978-84-17445-50-8

Short contents

Contents

Prologue

Current computers are very different to the initial idea proposed for the ENIAC by John von Neumann in 1945 based on the work of Presper Eckert and John William Mauchly. However, the general basic principles are the same.

Computers have evolved into machines with tremendous performance that have become essential in many fields. In fact, we have seen in recent times how the traditional look of computers has changed as they become more ubiquitous. Now it is possible to find computers in countless shapes, such as laptops, mobile devices, embedded systems, etc.

In this book, the authors try to illustrate how the basic principles of the von Neumann architecture are still applied to current computers, and what are the improvements that have been carried out to increase performance. In addition, the book also describes the influence of modern operating systems and the required virtualization support from the underlying hardware.

Gijón, July 19, 2022.

The authors.

Chapter 1

Introduction

In this chapter, an introduction to computer architecture is presented assuming the reader has a basic knowledge about computer fundamentals. The performance of a computer is a key factor in the design of computer architectures. Sometimes, performance is directly related to cost; however, not always a more expensive computer provides higher performance than a less expensive computer. Therefore, it is necessary to objectively assess the performance achieved by a computer to make an unbiased comparison.

This chapter addresses how to assess the performance of a computer. At the same time, it lays the foundations to better comprehend the performance improvements that will be proposed in this book for several components of the computer.

1.1 The computer

Before studying advanced topics of the computer, such as its architecture and performance measurement, it is necessary to review some basic concepts. The first section addresses the basic structure of a computer, specifying the main building blocks. Then, the abstraction levels used to facilitate both the construction and the programming of this complex machine are described. Finally, the basic principles guiding the development of computers are presented.

1.1.1 Basic structure

A computer is a machine that receives input information, processes this information using a program and generates new information through output media, with no human intervention.

A typical computer may be seen as a machine with several peripheral devices attached, such as a keyboard, a mouse and a screen. However, modern computers are presented in many different formats, from desktop computers or laptops to smartphones or embedded devices, such as those found in vehicles or electrical appliances.

Building a computer that can execute programs imposes the following needs:

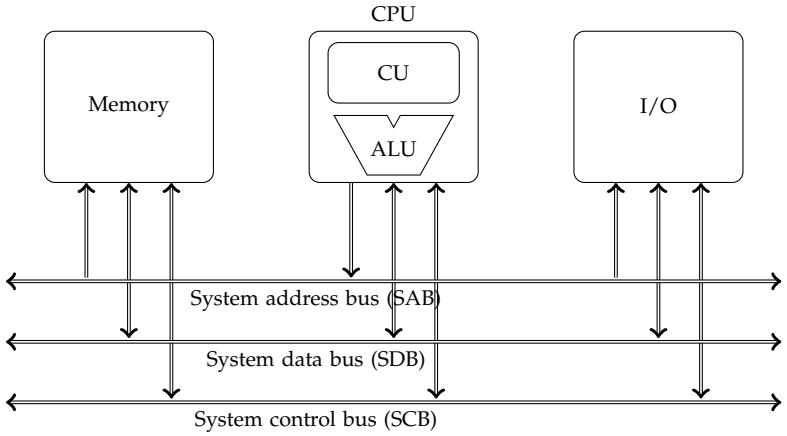


Figure 1.1: von Neumann architecture

- Storing the program to be executed. The program is composed of instructions that must be stored in a container inside the computer.
- Storing the program data. The input and output data of the program must also be stored in a container inside the computer.
- Reading and executing the program. The program instructions must be read from the container where they are stored and executed on the input data to generate the output data.
- Input/output (I/O). Both the program and the input data must be entered in the computer, either by a user or by an external system. In addition, it must be possible to show the output data to the user or send it to another system.

As in other fields of engineering, the design of a computer must follow certain requirements and meet restrictions on performance, cost and power consumption. Thus, the design of a computer is usually a compromise to meet such restrictions. Although several computer designs are possible, all of them have three components in common: memory, CPU and peripherals, as it is proposed in the von Neumann architecture (see figure 1.1). The memory constitutes the container where programs and data are stored using bit sequences; the CPU reads and executes programs; and the peripherals allow entering and extracting information to and from the computer. In the CPU, two are the most important elements:

- Control unit. It is used to decode the instructions of the program and generate the control signals required to execute these instructions.
- Arithmetic-logic unit (ALU). It is used to perform operations on binary data.

There also exist several buses to interconnect these elements. Buses are shared communication channels:

- System address bus (SAB). This bus is used by the CPU to indicate the location of the information that needs to be read from or written to the memory or I/O units.
- System data bus (SDB). This bus contains the information that has been read from or is going to be written to a specific location.
- System control bus (SCB). This bus is used to control the information flow between the computer components.

There exists a variation of the von Neumann architecture, called the Harvard architecture, which provides separate memories for data and code instead of using a single memory.

1.1.2 Multi-level machine

A great challenge in modern computers is the *conceptual distance* between the language understood by the computer —the language of the machine, encoded as a bit sequence— and the language used by programmers to implement algorithms. This distance is increasing due to two factors:

- The programmer is more comfortable using a language closer to the natural language to describe the solutions to problems, so that program development is simplified.
- The language of the computer should be as simple as possible, in order to use the fewer amount of hardware resources, and be highly efficient at the same time.

To deal with this problem, a new set of instructions closer to the natural language, L1, is commonly defined from the language of the machine, L0. Thus, language L1 is defined from the instructions available in language L0. This new language gives rise to a virtual machine, M1, built upon the physical machine, M0. Then, programmers can use M1 to develop programs in language L1.

To execute programs written in language L1 in machine M0 (that uses language L0), two non-exclusive techniques can be used:

- Translation. Before executing the program in language L1, it must be fully translated into L0 instructions.
- Interpretation. The program in language L1 is used as an input to a program written in language L0 (interpreter) that is in charge of executing the instructions written in L1 in machine M0 with no previous translation.

Although the definition of a new language simplifies program development by programmers, the distance between L1 and L0 must not be excessive for the translation and/or the interpretation to be possible. Thus, language L1 is still far from the natural language. Nevertheless, more levels can be defined to get closer to the natural language. Nowadays, computers usually define a set of common levels:

- Digital-logic level (level 0). It is based upon logic gates and constitutes the hardware of the computer.
- Microarchitecture level (level 1). This level is composed of the datapath and the control unit. In some CPUs, the operations on the datapath are controlled by means of a microprogrammed control unit, whereas others use wired control units. In the former, the microprogram is the interpreter of level 2 instructions. Current CPUs tend to avoid the use of microprograms, or at least use a hybrid approach where the wired part of the control unit is in charge of the simple level 2 instructions (usually, the most commonly used) and the microprogrammed part is in charge of the more complex level 2 instructions (the least commonly used).
- Instruction set architecture level (level 2). It is the specification defined by the processor manufacturer about the language of the CPU, the available registers, the addressing modes, etc.
- Operating system level (level 3). It is a hybrid level where many instructions belong to level 2 and others are interpreted (system service calls). This means that some instructions will reach level 2 unmodified and others will be substituted by operating system routines providing services to user programs. The operating system provides an abstraction layer of the lower levels so that programs written for a given operating system can be executed regardless on the underlying hardware.
- Assembly language level (level 4). It is the lowest level in which user programs can be written. It is the first symbolic language and is usually translated (compiled) into level 3. New abstractions and data types not available in level 3 can be defined in this level.
- Level 5 and upper levels. These levels define high level languages, such as C/C++. These languages enable the portability of programs from one operating system to another. It is possible to write programs in a high level language to be executed in several operating systems. However, if the language is not interpreted, programs may need to be recompiled for each target operating system.

Figure 1.2 shows the levels usually defined in a modern computer as well as the usual method to execute programs defined in a virtual machine in the immediately underlying machine.

1.1.3 Basic design principles

Next, several ideas applied to all areas of computer architecture are presented. These have lasted long after the first computer.

The basic principles that guide the design process of the architecture of computers are:

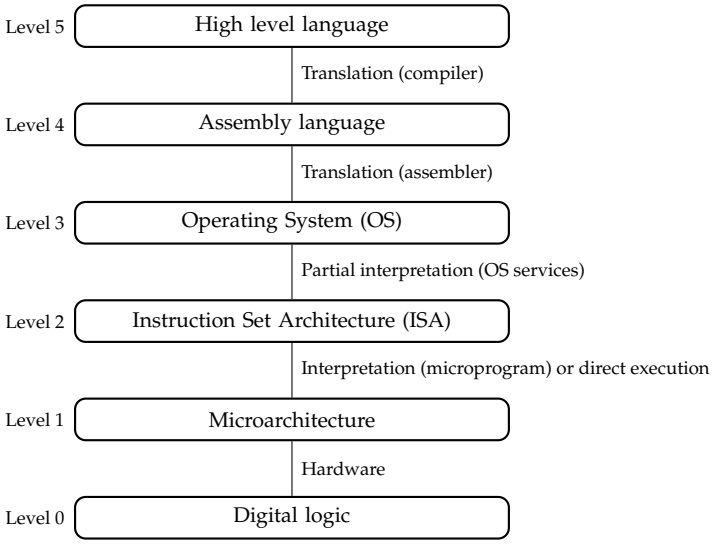


Figure 1.2: Abstraction levels in a modern computer

- Design for rapid change. Computer manufacturing technologies have greatly evolved and continue evolving every day. The number of logic gates per chip increases rapidly. The design of a new architecture must take into account future technology improvements to take advantage of all possible capabilities.
- Use abstraction. Designing a computer is an extremely complex task. Thus, abstraction is used to facilitate the process. Low level details are hidden to simplify the problem to higher levels. The current design of multi-level machines shown above is an example of the use of abstraction.
- Optimize for the common case. As the Amdahl's law (that will be studied in section 1.4.2) states, the improvement of the performance of a computer due to the improvement of the performance of a particular component is limited to the fraction of time when this component is used. Therefore, designs must optimize the most common cases (for example, the execution of common instructions).
- Improve performance via parallelism. When possible, parallelism must be used to improve performance. Manufacturing technologies are limited by physical constraints, limiting the speed of operation of a device. If the computer is designed allowing several components to work in parallel, its performance will be improved.
- Pipelining. It is a particular pattern of parallelism used in modern CPUs. Due to its importance, it will be covered in detail in chapter 2.
- Improve performance via prediction. Sometimes performance can be improved assuming that a task will be executed before actually knowing that it will be necessary to execute this task. If the prediction is correct, the time saved will

improve the performance of the system. Otherwise, the work done must be undone. The key relies on getting the prediction right most of the times.

- Hierarchy of memories. It is a key concept in the design of the memory system, which allows combining different technologies to get the best features of each technology. This will be covered in chapter 3.
- Dependability via redundancy. Although this book will not cover topics related to computer dependability, a common technique to achieve this feature is via redundancy of computer components.

1.2 Instruction set architecture

The design of a computer requires defining its architecture, which mostly refers to the architecture of its CPU. This architecture includes aspects such as the computer building elements, their behaviour and interactions, how these elements are interconnected or what input/output mechanisms are used.

Today, the architecture of a CPU, and by extension the architecture of the computer, usually refers to the instruction set architecture (ISA). The definition of the **instruction set architecture** of a CPU is the most critical design decision, since it determines all aspects that are visible to the programmer.[1] Among other things, the ISA defines the data types handled by the CPU (integers, real numbers, strings, etc.), the registers available to the programmer, the instruction set, or the address format and addressing modes.

The instruction set architectures have been classified into two groups: RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer). On one hand, RISC architectures define very simple instructions that can be executed quickly with minimum hardware. Examples of RISC architectures include ARM (vastly used in mobile devices), MIPS, IA-64 and PowerPC. On the other hand, CISC architectures define complex instructions capable of executing high level operations, facilitating the work of programmers. Examples of CISC architectures include x86 and Motorola 68 000.

The instruction set architecture of a CPU can implement one of the next four machine models, shown in figure 1.3.

- Stack. In a stack machine, the CPU has a set of registers arranged as a stack, as well as a special register called `TOS` (Top Of Stack). Operands of instructions are implicit and fetched from the register stack, starting from the one pointed by the `TOS` register. The result of the operation is also stored on the top of the register stack. For example, the assignment of the addition of two operands may be as follows:

```
push A
push B
add
pop  C
```

[1] Low level programmer, compiler developer or system developer.

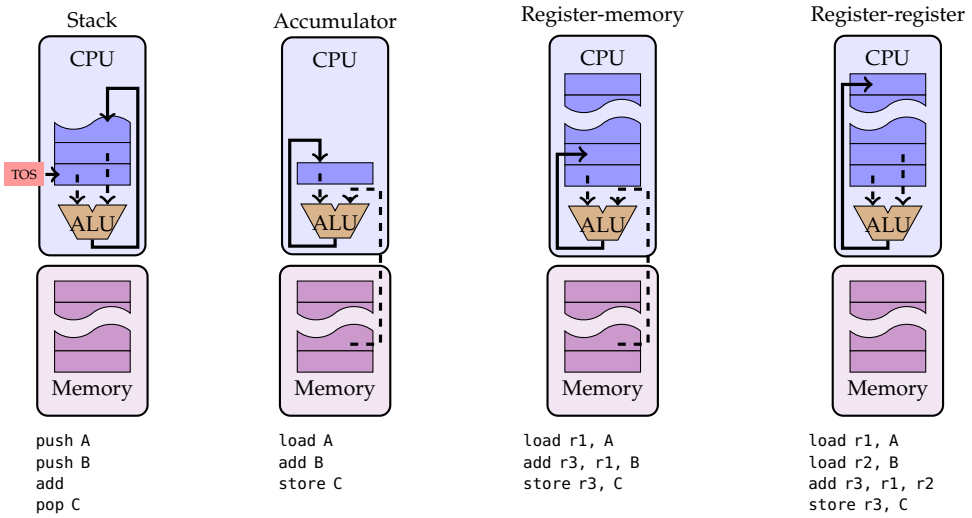


Figure 1.3: Machine models

At the beginning, the values of variables `A` and `B` are pushed on the register stack. Then, the add instruction is executed. This instruction pops the two top operands from the stack, adds them and stores the result back in the stack. The `pop` instruction pops the value pointed by the `TOS` register and stores it into variable `C`. The advantage of a CPU based on a stack machine model is that it requires a fewer amount of bits to encode instructions, since the instruction code does not require bits to identify operands.

- Accumulator. Accumulator machines provide a special register, called accumulator, that is implicitly used as an operand. The result of an operation is stored in the accumulator register. Hence, an assignment operation of an addition may be as follows:

```
load  A
add   B
store C
```

The first instruction loads the value of variable `A` into the accumulator register. Then, the addition is carried out, specifying the variable whose value will be added to the value of the accumulator register. The result is stored in the accumulator register and saved in variable `C` in the last instruction.

- Register-memory. In a register-memory machine one operand may be stored in a register of the CPU and the other in memory. The result of the operation is stored in another register of the CPU. Under these circumstances, an assignment operation of an addition may be as follows:

```
load  r1, A
add   r3, r1, B
store r3, C
```

The first instruction loads the value of variable `A` into register `r1`. Then, the addition between the value stored in this register and variable `B`, in memory, is carried out. The result is stored in register `r3`. Finally, the result of the addition can be stored in memory in variable `C` from register `r3`.

- Register-register, more commonly known as load/store. In a load/store machine all operands of instructions must be stored in registers inside the CPU. The assignment of an addition operation may be as follows:

```
load  r1, A
load  r2, B
add   r3, r1, r2
store r3, C
```

The first instructions load the operands in registers `r1` and `r2`. The `add` instruction must explicitly specify the registers containing the operands, as well as the register that will store the result. After the operation, the result can be stored in variable `C` from register `r3`. In this machine model, only `load` and `store` instructions can access memory.

Nowadays, ARM is the most commonly used architecture, since it is the preferred choice in mobile devices. This architecture was inspired by MIPS64, which will be used along this book. This architecture implements a load/store machine model.

1.3 Microarchitecture

The microarchitecture refers to the internal organization of a CPU that is hidden to programmers. Any instruction set architecture can be implemented defining different microarchitectures. For example, Intel Nehalem (Core i5) and AMD K10 (Athlon II) microarchitectures implement x86 instruction set architecture.

The microarchitecture defines features such as the number of functional units in the CPU (for instance, the number of arithmetic-logic units), the number and type of cache memories and the memory bus width. It also determines the organization, where different CPU types are possible: pipelined, superscalar or multicore, among others.

The main goal of a microarchitecture is taking advantage of existing manufacturing technologies to provide maximum performance in the implementation of a given instruction set architecture.

Chapter 2 describes several microarchitectures to implement a reduced version of the MIPS64 instruction set architecture.

1.4 Performance

In the previous paragraphs, the instruction set architecture of a computer was covered from the point of view of its functionality, that is, the instructions that can be

executed. However, performance is related to the number of instructions that a computer can execute in a given amount of time, which greatly depends on its microarchitecture.

This section introduces the concept of performance, as well as how performance is measured in a computer.

1.4.1 Concept of performance

A computer is a very complex system where many elements collaborate. It is difficult to define the concept of performance in a computer, since it depends on the perception of the user and the target operation of the computer. Thus, from the point of view of the user, the interest lies on the execution time of a task. In contrast, a system administrator is more interested in the number of tasks completed per time unit. Therefore, various metrics can be used to measure performance.

A **metric** is a magnitude that quantifies a measurable property of a system, so that computers can be compared using a metric. There exist different types of metrics depending on the property to be measured in the system.

The performance of a computer is usually related to its ability to do work. It is quantified mainly using two metrics: response time and throughput. **Response time** refers to the time taken by the computer to run a task, while **throughput** refers to the number of tasks completed per time unit.

$$\text{Throughput} = \frac{\text{Tasks completed}}{\text{Reference time}}$$

For both throughput and response time to make sense, it is necessary to clearly define the task on which they are measured. For example, the task can refer to the execution of a complete program, the execution of a single instruction, fetching a web page, responding a query to a database, etc. Thus, response time refers to the time between the start and the end of the task (program, instruction, web page or query), while throughput refers to the number of tasks (programs, instructions, web pages or queries) completed per time unit.

Throughput is the metric commonly used to measure the performance of a computer. In the case of a system capable of running only one task at a time, throughput and response time are inversely related. For example, if the average response time of a task is 1 ms, the system can execute $(1/0.001\,\text{s}) = 1000$ tasks per second. However, this relation is not true when the system is capable of running more than one task simultaneously.

This difference in the behaviour of the two metrics can be observed when a system is improved to increase performance. For example, in a system with a CPU executing instructions sequentially with a clock frequency of 1 GHz, the time taken to execute one instruction is t seconds. Two improvements of the system are devised:

- Replace the processor by another with exactly the same microarchitecture but with doubled clock frequency, that is, 2 GHz.

- Replace the processor by another with two cores at the same clock frequency and with the same microarchitecture as the original.

The result obtained applying these improvements would be the following:

- In the first scenario, since the processor operates using a doubled clock frequency, it will be able to execute instructions in half the time. Thus, the time taken to execute one instruction would be $\frac{t}{2}$ seconds. The response time is reduced by half, and hence twice the original number of instructions will be executed in the same amount of time; that is, the throughput is doubled.
- In the second scenario, the processor executes two instructions at the same time; thus, the throughput of the system is also doubled. However, as the CPU cores implement the same microarchitecture as the original, the time taken to execute one instruction is the same, that is, t seconds.

This example shows how the system performance is affected when introducing improvements. On one hand, it is possible to reduce response time, which implies also increasing throughput. To achieve this, organizational or technological improvements are required. On the other hand, throughput can be increased by increasing the parallelism of the components of the system, although response time is not usually decreased (it is even increased in occasions).

If the computer can run several tasks simultaneously, which is really common, the performance metric typically used is the throughput.

The final goal of metrics is to provide an instrument to compare the performance of computers. As an example, let A be a computer executing 30 tasks per second and let B be a computer executing 20 tasks per second. How many times is computer A faster than computer B? That is, what is the speedup in the performance of computer A compared to computer B?

$$\text{Speedup} = \frac{\text{Performance}_\text{A}}{\text{Performance}_\text{B}} = \frac{\text{Throughput}_\text{A}}{\text{Throughput}_\text{B}} = \frac{30}{20} = 1.5$$

The speedup in performance is the ratio between the performance of the two computers. It is said that computer A has a speedup of 1.5. A speedup value of 1 shows that the performance of both computers is the same. If the speedup is greater than 1, computer A has better performance than computer B and vice versa if the speedup is less than 1.

The term response time is often used when assessing the performance of the CPU executing programs. However, when the performance of memory or interconnection systems is assessed, the terms **latency** (referring to the time taken by the system to provide the required data) and **bandwidth** (referring to the amount of information that can be provided per time unit) are preferred instead of response time and throughput, respectively.

Although response time and throughput have been considered constant measurements, in most scenarios they must be approximated using random variables,

since they vary from measurement to measurement depending on the time of measure and the load of the computer at that time. Thus, for example, instead of taking a fixed response time, it is approximated using a random variable normally distributed $\sim \mathcal{N}(\mu, \sigma^2)$, characterized by the mean and the standard deviation.

Therefore, statistics is very important when measuring the performance of computers. As an example, if the response time of a program in a computer is measured 10 times (n) obtaining the following results:

Measurement	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
Response time (s)	3.2	2.9	3.1	3.0	2.8	3.1	3.2	3.0	3.3	2.7

the mean ($\overline{x}$) and the standard deviation (s) of the measurements, i.e., the sample mean and the sample standard deviation, can be computed:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{3.2 + 2.9 + 3.1 + 3.0 + 2.8 + 3.1 + 3.2 + 3.0 + 3.3 + 2.7}{10} = 3.03\,\text{s}$$

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}} = 0.189\,\text{s}$$

The population mean (μ) is estimated using the sample mean ($\overline{x}$), while the population standard deviation (σ) is estimated using the sample standard deviation (s). Assuming that the response time follows a normal distribution, the area in the interval $[\mu - 2\sigma, \mu + 2\sigma]$ covers approximately 95% of all the response times of the program; that is, the probability that the response time of the program lies within this interval[2] is 95%, in this example $[2.62, 3.38]$. This is shown in the probability density function of the response time in figure 1.4.

1.4.2 Amdahl's law

As described in section 1.1, a computer is composed of several components that work collaboratively to execute programs: CPU, memory, interconnection systems and input/output system. Improvements in computer performance can be achieved by improving components of the computer. However, improving the performance of a component p times does not increase the performance of the whole computer p times.

In a computer executing only one task at a time, the Amdahl's law states:

> *The speedup or gain (A) that can be obtained in the performance of a system due to the improvement of one of its components is limited by the fraction of the time that this component is used.*

[2] Actually, given the fact that the number of measurements is really short (<30) and the population standard deviation is unknown, the Student's t-distribution should have been used to compute the confidence interval. For simplicity, the normal distribution is used, and the interval $[\mu - 2\sigma, \mu + 2\sigma]$ is a valid approximation for the 95% confidence level.

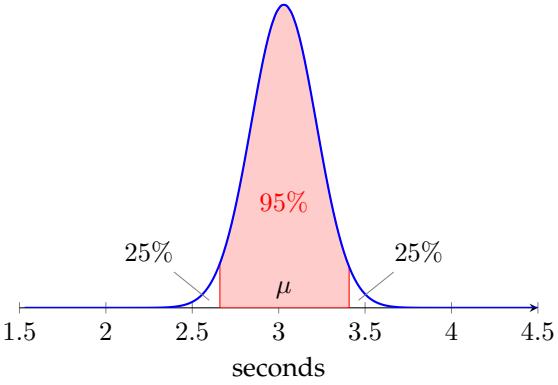


Figure 1.4: Probability density function of the response time

The response time obtained in the computer using the enhanced component is computed adding two factors: the response time introduced by the non-enhanced components and the response time introduced by the enhanced component, which is shortened according to the applied improvement:

$$\text{Response time}_{\text{enh}} = \text{Response time}_{\text{original}} \times \left((1 - \text{Fraction}_{\text{enh}}) + \frac{\text{Fraction}_{\text{enh}}}{\text{Speedup}_{\text{enh}}} \right)$$

Amdahl's expression can be deduced from the performance speedup obtained using the enhanced component compared to the performance using the original component:

$$A = \frac{\text{Performance}_{\text{enh}}}{\text{Performance}_{\text{original}}} = \frac{\text{Response time}_{\text{original}}}{\text{Response time}_{\text{enh}}}$$

And, thus:

$$A = \frac{1}{(1 - \text{Fraction}_{\text{enh}}) + \frac{\text{Fraction}_{\text{enh}}}{\text{Speedup}_{\text{enh}}}}$$

For example, a computer can execute a program in 100 seconds. In this execution time, 20 seconds are due to the execution of the CPU, 35 seconds are due to memory accesses, 40 seconds are due to input/output operations with the disk, and the rest of seconds are due to interconnection operations. What is the speedup in the execution of the program if the CPU is replaced by another three times faster? What is the speedup if the disk is replaced by another twice as fast? What is the new execution time in each of these scenarios?

Firstly, the fraction of time of the components that will be enhanced is computed:

$$\text{Fraction}_{\text{CPU}} = \frac{20}{100} = 0.2$$

$$\text{Fraction}_{\text{disk}} = \frac{40}{100} = 0.4$$

In the scenario where the CPU is replaced:

$$A = \frac{1}{(1 - 0.2) + 0.2/3} \approx 1.15$$

$$\text{Response time}_{\text{enh}} = \frac{\text{Response time}_{\text{original}}}{A} \approx \frac{100}{1.15} = 86.96 \text{ seconds}$$

In the scenario where the disk is replaced:

$$A = \frac{1}{(1 - 0.4) + 0.4/2} = 1.25$$

$$\text{Response time}_{\text{enh}} = \frac{\text{Response time}_{\text{original}}}{A} = \frac{100}{1.25} = 80 \text{ seconds}$$

As can be seen, the performance improvement achieved by replacing the disk by another twice as fast is higher than that obtained by replacing the CPU by another three times faster. This is because in this example the disk is used a larger fraction of time.

1.4.3 CPU performance

The performance of a computer depends on all of its components. However, the most complex component, which is also the most eligible to be optimized, is the CPU. Therefore, the analysis of the performance of a computer pays special attention to the performance of the CPU.

Comparing the performance of two CPUs requires defining the metrics to measure their performance. As seen above, response time and throughput are two commonly used metrics. However, these metrics make the comparison of the performance of two CPUs difficult in the real world.

The response time of a program shows the time between the start and the completion of the program and is variable. Not only does it include the time that the program is running, which is called **CPU time**,[3] but also other factors as shown in figure 1.5:

[3]Some sources use *execution time* to refer to response time and others to refer to CPU time. To avoid misunderstandings, this book uses response time as the time required to execute a program and CPU time as the effective running time in the CPU.

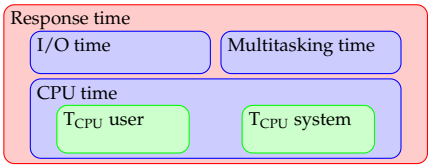


Figure 1.5: Factors affecting the response time of a program

- CPU time. This time greatly depends on the CPU, though other elements such as memory are also involved, since the execution of a program requires accessing the memory of the computer. The CPU time is usually divided in two: CPU user time and CPU system time. For example, in the execution of a program, services provided by the operating system are invoked, which are executed in a running mode of the CPU called privileged mode, and contribute to the CPU system time.
- Input/output waiting time. For example, a program must wait for a reading operation from the disk to be completed before continuing execution.
- Multitasking time. From time to time, the operating system must run tasks that are not associated with the current program, such as those related to managing computer resources. In addition, a multitasking operating system can run several tasks or programs simultaneously, not only the task whose response time is being measured.

From the above factors, the CPU time is the one providing a closer idea to the performance of the CPU. Thus, if response time is used to estimate the performance of a CPU, care must be taken to minimize the other factors affecting the response time.

An alternative to response time in order to measure the performance of the CPU involves measuring throughput in instructions per second in the execution of the program (usually in millions of instructions per second or MIPS). The main disadvantage is that MIPS measurements cannot be compared among CPUs implementing different instruction sets.

Measuring the performance of a CPU is very complex. Thus, the next section analyses the most important factors from a theoretical point of view.

Theoretical analysis of CPU time

In a single-threaded program, the CPU time depends on several factors. The first important factor is the clock period (T), which is the inverse of the frequency (f). Thus, clock period and clock frequency are commonly used interchangeably.

$$f = T^{-1} = \frac{1}{T}$$

Then, the CPU time in a program can be defined in relation to the clock frequency or clock period as follows:

$$T_{CPU} = \frac{\text{CPU cycles of the program}}{f} = \text{CPU cycles of the program} \times T$$

Clock cycles per instruction (CPI) shows the number of clock cycles required to complete each instruction of the program and can be computed as follows:

$$\text{CPI} = \frac{\text{CPU cycles of the program}}{\text{Program instructions}}$$

Using the CPI, the CPU time can be computed as follows:

$$T_{CPU} = \frac{\text{Program instructions} \times \text{CPI}}{f} = \text{Program instructions} \times \text{CPI} \times T$$

The above expression is referred to as the **iron law of performance** of a CPU. As can be seen, the CPU time depends on three related factors:

- Number of instructions of the program. This number depends on the instruction set and compilers. For example, programs in RISC architectures usually contain many more instructions than their counterparts in CISC architectures. In addition, the compiler is responsible for translating programs written in a high-level language into instructions of the CPU; thus, it plays a crucial role in the number of instructions of the program.

- The number of clock cycles per instruction (CPI). This factor depends on the internal organization of the CPU and the complexity of the instructions. CISC CPUs implement instructions that can perform complex operations; thus, programs tend to have fewer instructions. The main disadvantage is that these instructions require many clock cycles to be completed, that is, a high CPI. On the contrary, RISC CPUs implement very simple instructions requiring very few clock cycles, that is, a low CPI, but programs require a large number of instructions.

  The basic principle stated by Amdahl's law that the gain in performance is proportional to the fraction of time that the enhanced component is used can be utilized to reduce the CPI. In the above expression to compute the CPU time, the CPI is expressed as the average CPI of all the machine instructions of the program. Therefore, instruction set designers seek to reduce the CPI in the most commonly used instructions, since their influence in performance will be greater.

- Clock period. This factor expresses the speed of the CPU. In order to reduce the clock period, and increase the clock frequency, the manufacturing technology of the CPU should usually be improved. The clock period can also be reduced with organizational improvements such as the *pipeline*, as will be studied in the chapter related to the CPU.

From the iron law, the next expression can be deduced to compute the MIPS based on the CPI:

$$\text{MIPS} = \frac{1}{\text{CPI} \times T \times 10^6}$$

And, thus, the CPU time of a program can be expressed as follows:

$$\text{T}_{\text{CPU}} = \frac{\text{Program instructions}}{10^6 \times \text{MIPS}}$$

1.4.4 Benchmarks

A *benchmark* is a program designed to evaluate the performance of a computer or one of its components. Benchmarks are common for evaluating the performance of the CPU and the memory system.

As shown above, most of the time the performance of a computer cannot be computed directly, but it must be estimated based on measurements. The results of these measurements vary according to the workload of the computer. A benchmark imposes some *workload* on the computer or one of its components to assess performance. Therefore, results provided by two benchmarks on the same computer may be different.

A benchmark should impose a workload on the computer similar to the usual workload the computer is used for. For example, to evaluate the performance of a computer oriented to office tasks, the workload of the benchmark should mimic the workload of office programs in the computer. Another example: the performance evaluation of a workstation oriented to develop programs in C should use a benchmark with a representative workload of compilers, linkers, IDEs, etc. Therefore, a benchmark is designed to evaluate the performance of a computer under certain circumstances. A computer can get a low performance level when measured using a benchmark with a given workload, and a high performance level if evaluated with a different workload.

There exist two types of benchmarks depending on the workload:

- Application benchmarks. They are based on real programs used in the usual workload of the computer. The main advantage of this type is that these benchmarks assess the performance under real use conditions of the computer. However, the main disadvantage is their complex reproducibility, so measurements may contain great variability.
- Synthetic benchmarks. They are programs specifically created to reproduce the most common operations used in real programs; though they are not real programs. The main advantage of this type of benchmarks is their reproducibility. However, the main disadvantage is that the obtained results do not always match the performance in the execution of real programs.

	Throughput C1	Throughput C2
Benchmark B1	5	10
Benchmark B2	5	2

Table 1.1: Sample performance assessment using two benchmarks

	$A_{C1/C2}$	$A_{C2/C1}$
Benchmark B1	0.5	2
Benchmark B2	2.5	0.4
Arithmetic mean (B1, B2)	1.5	1.2
Geometric mean (B1, B2)	1.12	0.89

Table 1.2: Speedups computed using individual becnhmarks or statistical aggregations

There exist benchmarks to assess the performance of several parts of the computer, such as the CPU, the memory system, the graphics interface, the disk, etc.

The performance of two computers can be compared based on the results provided by a benchmark, expressing the speedup of one computer compared to the other for the workload of the benchmark. Nevertheless, running different benchmarks is sometimes required to assess the performance of the computer using different workloads, obtaining several speedups. In this scenario, a computer may show higher performance for a given workload, and lower for another. Table 1.1 shows the result of the execution of two benchmarks, B1 and B2, on two computers, C1 and C2.

The performance shown by computer C1 using benchmark B1 is lower than that shown by computer C2, as can be seen in throughput values. However, using benchmark B2, the result is different: computer C1 shows higher performance than computer C2. These values can be seen in the speedups $A_{C1/C2}$ (computer C1 compared to computer C2) and $A_{C2/C1}$ (computer C2 compared to computer C1), shown in table 1.2.

When it is necessary to summarize the performance of a computer compared to another using only one value, the average of all speedups may be used. The arithmetic mean is an option. In this case, the aggregated speedup of computer C1 compared to computer C2 results $A_{C1/C2} = (0.5 + 2.5)/2 = 1.5$, and thus, computer C1 is 1.5 times faster than computer C2. Nevertheless, if the aggregated speedup of computer C2 compared to computer C1 is computed, the result is $A_{C2/C1} = (2 + 0.4)/2 = 1.2$, so computer C2 is 1.2 times faster than computer C1. As can be seen, the obtained results are contradictory, and they depend on the computer selected as the reference. Therefore, the arithmetic mean cannot be used to aggregate speedups.

To compute an aggregate speedup from the speedups obtained in n benchmarks, the geometric mean is usually used according to the next expression:

$$A = \sqrt[n]{A_1 \cdot A_2 \cdot ... \cdot A_n}$$

As can be seen in table 1.2, the results obtained using the geometric mean are coherent. The speedup of computer C1 compared to computer C2 is 1.12, and thus, computer C1 performs 1.12 times faster than computer C2. If computer C2 is used as the reference, it is 0.89 times faster than computer C1, or the other way round, computer C1 is $(1/0.89) = 1.12$ times faster than computer C2. Therefore, the aggregated speedup obtained using the geometric mean is coherent regardless of the computer used as the reference.

Chapter 2

The CPU

This chapter is focused on studying the main features in which the design of modern CPUs is based on. To illustrate this, the MIPS64 architecture is used. The chapter begins with a brief description of the MIPS64 instruction set architecture; then, some possible implementations are depicted. After that, the design principles of modern CPUs are addressed:

- Performance improvements, based on the concept of pipeline (segmentation of instructions), described in sections 2.3 to 2.6.
- Support to multitasking operating systems, described in section 2.7.
- Support to virtualization, introduced in section 2.8.

This chapter will mention energy management in a CPU briefly, since it is also a critical feature in modern computers.

2.1 MIPS64 architecture

To illustrate the concepts covered in this chapter, the MIPS64 architecture is used. The main reason for this choice is its simplicity. It is a RISC architecture implementing a load/store machine model. This means that the CPU operates on internal registers, and every operand must be fetched from memory and stored in one of these registers before making operations with it. Furthermore, only load and store instructions can access the memory of the computer, leading to a very simple implementation as it will be shown.

MIPS64 is an extension of the MIPS32 architecture. MIPS64 is fully compatible with MIPS32, so programs written for the latter can be run in both architectures. MIPS64 processors can be found in video game consoles and network devices. MIPS64 has great similarities to the ARMv8 architecture, another RISC architecture, that is commonly found in smartphones and a large number of embedded systems.

Nowadays, RISC has become the design paradigm of most CPUs. All of them, including the MIPS64 CPU, are based on the same design principles:

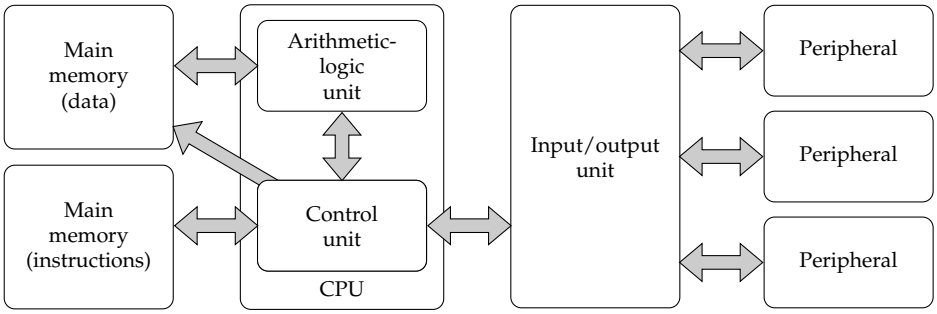


Figure 2.1: Harvard architecture

- Very simple and fixed-size instructions, so they can be decoded really fast.
- Few and simple addressing modes, simplifying the hardware design.
- Large number of general purpose registers, so many variables of the program may be stored in registers, reducing the number of memory accesses and, thus, improving performance.

MIPS64 is a 64-bit architecture with a very simple instruction set. Introduced in 1991, it was considered the first 64-bit architecture. It features a set of 32 integer registers, 64-bit wide each, called `R0`, ..., `R31`; `R0` always stores 0 and `R31` is reserved to store the return address in some jump instructions. There are also 32 floating-point registers, 64-bit wide each, called `FP0`, ..., `FP31`. MIPS64 also defines two status registers, one for integer operations and antoher for floating-point operations.

Memory is byte-addressable using 64-bit addresses. This means that a MIPS64 CPU can address 2^{64} bytes, that is, 16 EiB (16 exbibytes). As MIPS64 follows a load-/store machine model, read (load) and write (store) operations from/to memory are always carried out on general purpose registers (`Rx` or `FPx`).

MIPS64 implements a Harvard model in its internal memory configuration. This means that MIPS64 CPUs feature independent memories for data and instructions that can be simultaneously accessed. This model is shown in figure 2.1, where the two memories can be seen. Thus, the CPU has the memory lines replicated.

2.1.1 Data types

The MIPS64 architecture provides native support for integer and real numbers. It is important to emphasize *native*, since even though the CPU does only understand these data types, different abstraction layers in the computer (assembly, operating system, programming language, etc.) may define other data types implemented on top of these. For instance, a *string* data type could be defined from an integer data type, as a sequence of integers.

MIPS64 supports integers in two's complement of the next sizes, shown in figure 2.2:

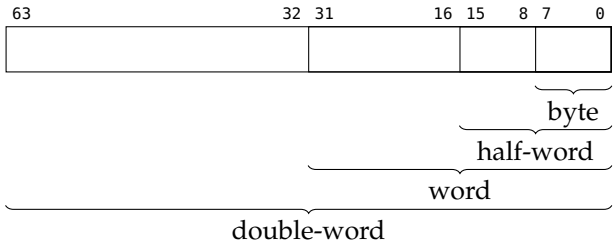


Figure 2.2: Data types supported by MIPS64 architecture

- **byte**: 8-bit integer.
- **half-word**: 16-bit integer.
- **word**: 32-bit integer.
- **doble-word**: 64-bit integer.

When load and store operations are performed using integer registers (Rx), one byte, one half-word (2 bytes), one word (4 bytes), or one double-word (8 bytes) can be accessed in memory. When loading a data item smaller than 64 bits (byte, half-word or word) it is possible to do it with or without sign. If the load operation is performed without sign, the register is initially reset to 0 and then the data item is copied into the low part of the register. If the load operation is performed with sign, the data item is copied to the low part of the register and then the sign is extended, that is, the most significant bit of the data item is replicated in the most significant bits of the register that have not been modified by the data item. This is shown in figure 2.3, where half-word 90BDh is loaded into a register. When the operation is carried out without sign, the 64-bit register is loaded with 0000 0000 0000 90BDh. However, if the load operation is carried out with sign, the value will be FFFF FFFF FFFF 90BDh. In any case, memory accesses to more than one byte must be aligned, that is, the address must be multiple of 2 to access half-words, multiple of 4 to access words, and multiple of 8 to access double-words. Figure 2.4 shows two memory accesses to two words, one aligned and one misaligned. As can be seen, the latter is trying to access a memory address which is not multiple of 4 (the size in bytes of a word).

Another important issue to take into account when a data item larger than one byte is stored in memory is the order of bytes, or byte endianness. In a MIPS64 CPU the order can be set by hardware (so it is called a *bi-endian* architecture). There are two alternatives: *big-endian* or *little-endian*. In the aligned example in figure 2.4, the word is interpreted as BF9A 10A3h in little-endian and as A310 9ABFh in big-endian.

Regarding floating-point numbers, the MIPS64 CPU provides support for two sizes: single precision (32-bit word) and double precision (64-bit double-word), both encoded using IEEE-754. As with integers, loading floating-point registers must be done using aligned addresses.

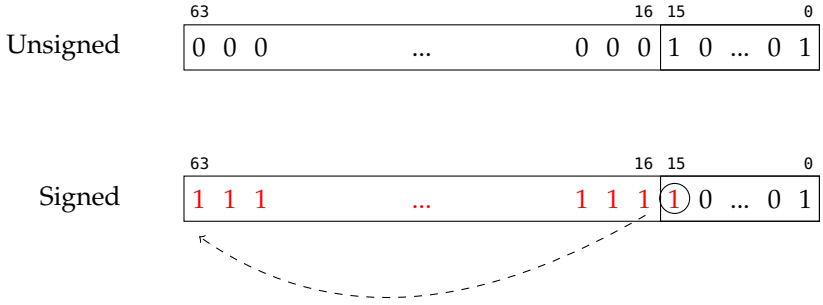


Figure 2.3: Loading the half-word 90BDh in a register without sign (*zero-extended*) and with sign (*sign-extended*)

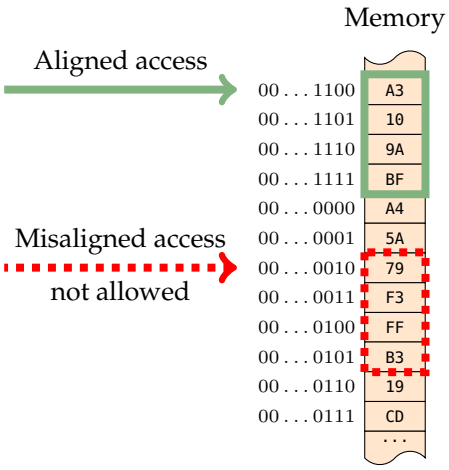


Figure 2.4: Aligned and misaligned memory accesses to memory words

2.1.2 Instruction set

As any other architecture, MIPS64 defines an instruction set composed of arithmetic, logical, move and jump instructions, among others. Being a RISC architecture, the MIPS64 instruction set is very simple compared to other instruction sets, such as for example that implemented by x86 CPUs. This simplicity allows using a fixed-size instruction code, requiring simpler hardware and facilitating the instruction code loading and decoding. In addition, as described in chapter 1, simple instructions achieve a low CPI factor, that is, they can be executed in very few clock cycles.

The MIPS64 architecture defines, among others, the following instruction types:

- Load and store instructions. They are used to read and write from and to memory. A general purpose register is always involved (integer or floating-point). Only this type of instructions can access memory.
- Arithmetic-logic instructions. These instructions carry out arithmetic or logical operations on integers or floating-point numbers.
- Flow-control instructions. These instructions modify the sequential flow of execution of a program. There are two types: (unconditional) jumps, which are always taken, and (conditional) branches, which are taken depending on a particular condition.
- Other instructions. They include bit shifts, integer-to-real conversion, etc.

Although this chapter uses the MIPS64 architecture to illustrate several concepts, just a small portion of the instruction set, which is described in appendix A, is considered.

Addressing modes

CPU instructions require operands, and thus, several bits of the instruction code are reserved to identify them. The addressing modes of a CPU determine how the operands are obtained from these bits. The operands can be in three different locations; all of them have advantages and disadvantages.

Operands can be stored in memory. The main advantage of this location is the great amount of memory positions available in the computer. However, the disadvantages include high access time and a large number of bits in the instruction code in order to identify the memory address of the operand.

Operands can also be located in registers inside the CPU. This alternative is faster, since the access time to a CPU register is negligible compared to the access time to memory. In addition, very few bits are required in the instruction code to identify the register (addressing 32 integer registers requires 5 bits). The main disadvantage of this option is the very limited number of registers available in a CPU compared to the number of memory positions.

Finally, instead of using several bits of the instruction code to reference the operand (in memory or in a register), these bits may encode the value of the operand

in the instruction code directly. In this case, the access to the operand is immediate. This is only valid for constants, since they are encoded at the same time as the instruction. Another disadvantage is the large number of bits required in the instruction code, and that they are only valid for source operands, that is, the destination operand of an instruction cannot be stored in the instruction code as an immediate value.

MIPS64 provides five addressing modes:

- Immediate. In this case, the value of the operand is stored in the instruction code. For example, the next instruction performs the addition of the 64-bit integer stored in register `r8` with number -3 and stores the result in register `r4`. In this case, constant -3 is encoded in two's complement in the instruction code directly.

```
daddi r4, r8, -3
```

- Register. In this mode, the number of the register is included in the instruction code. Hence, in the above example, in addition to constant -3, the instruction code will include 5 bits identifying the source register (`r8`) and other 5 bits identifying the destination register (`r4`). Using this addressing mode, the value of the operand is located in the referenced register.

- Base and offset (indexed). In this mode, the operand is located in a memory position referenced by the result of the addition of a value in a register and a constant encoded in the instruction code. Next, several examples of loading a double-word are shown:

```
ld r12, 200(r0)  ; direct addressing to address 200 (r0 is always 0)
ld r12, 0(r3)    ; indirect addressing to the address contained in r3
ld r12, 350(r3)  ; indexed addressing to the address contained in r3 + 350
```

  When `r0` is used as the base register this is called *direct addressing*. Since `r0` is always 0, the address where the operand is stored is provided only by the value of the constant. However, if the value of the constant is 0 this is called *indirect addressing*, as the base register determines the position of the operand in memory. In any case, it must be taken into account that the addition of the base register and the offset must provide an aligned memory access. In the above example, adding 350 and `r3` should provide a value multiple of 8.

- PC-relative. This addressing mode is used in branches to calculate the destination address as the addition of the program counter (`PC`) and a value computed from a constant encoded in the instruction code. The next example shows a branch that branches when the contents of both registers are the same.

```
beq r2, r9, -5
```

  The constant shows the number of instructions to jump, since this number is shifted two positions to the left (which is the same as multiplying by 4, the size in bytes of the instruction code) before being added to the program counter.

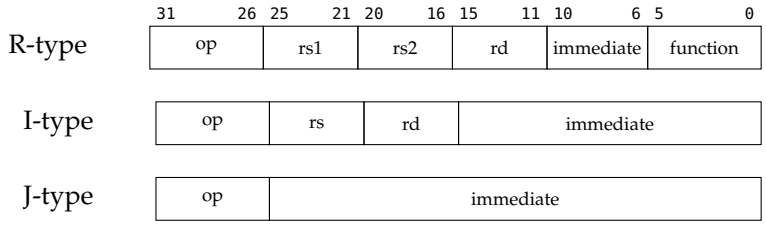


Figure 2.5: Instruction encoding in MIPS64

In this case, $-5 \times 4 = -20$. Thus, the number of positions to add to the PC is multiple of 4, which guarantees an aligned access to the instruction memory.

Using this addressing mode, branches can jump up to $\pm$128 KiB.

- Pseudo-direct. This addressing mode is used in jumps to compute the destination address. This address results from the concatenation of 26 bits of the instruction code shifted two positions to the left (28 bits), and the 36 $(64 - 28)$ most significant bits of the program counter. Next, an example of a jump is shown:

```
j 1000
```

In this example, assuming that PC contains D000 0000 0000 0000h when the jump instruction is executed, the destination address is computed taking the 36 most significant bits of this value (D 0000 0000h) and concatenating them with the result of shifting the constant two positions to the left (1000×4 = 4000 = FA0h), which results in a value for PC of D000 0000 0000 0FA0h (always multiple of 4).

Using this addressing mode, the instruction memory is organized in areas of 256 MiB. Modifying adequately the value of the constant, it is possible to jump to any instruction stored in the same area as the one currently pointed by PC. In the above example, address FA0h in the 256 MiB area pointed by PC is accessed.

Instruction encoding

As mentioned above, all instructions in the MIPS64 architecture are encoded using the same number of bits. Specifically, the length of the instruction code of all instructions is 32 bits. They are encoded following one of the three types shown in figure 2.5. Arithmetic and logical instructions are R-type instructions. Load and store, as well as branches, are I-type instructions. Finally, jumps are J-type instructions.

The 32 bits of the instruction code are divided into several fields:

- op. It is present in all types of instruction and represents the operation code (opcode), which identifies the operation carried out by the instruction.
- function. It specifies the variant of the operation identified in op. For example, the operation code is the same for the **dadd** and **dsub** instructions (0), and they only differ in the function code: 2Ch and 2Eh, respectively.

Instruction	Type	Description
`ld Rd,imm_16(Ri)`	I	Load double-word
`sd Rs,imm_16(Ri)`	I	Store double-word
`beq Rs1,Rs2,imm_16`	I	Branch on equal registers
`daddi Rd,Rs,imm_16`	I	Add a double-word and an immediate value
`j imm_26`	J	Jump
`dadd Rd,Rs1,Rs2`	R	Add double-words
`dsub Rd,Rs1,Rs2`	R	Subtract double-words
`and Rd,Rs1,Rs2`	R	Logical AND operation
`or Rd,Rs1,Rs2`	R	Logical OR operation
`slt Rd,Rs1,Rs2`	R	Set on less than (`if Rs1<Rs2 Rd=1; else Rd=0`)

Table 2.1: Instructions that will be implemented by the microarchitecture

- `rs/rs1`. It is the number of the register containing the first source operand.
- `rs2`. It is the number of the register containing the second source operand.
- `rd`. It is the number of the register acting as the destination operand.
- `immediate`. It represents immediate values encoded with the rest of the instruction code. This immediate value is used, for instance, for bit shifting in R-type instructions, for branches in I-type instructions and for jumps in J-type instructions.

2.2 Single-cycle microarchitecture

Once the instruction set architecture of a CPU has been defined, the MIPS64 architecture in this case, it is now the moment to implement it. As described in chapter 1, various implementations are possible. The simplest implementation involves a single-cycle microarchitecture, where each instruction only requires one clock cycle to be executed.

To illustrate the key ideas of this microarchitecture, a subset of the MIPS64 instruction set architecture will be implemented: load and store instructions (only using double-words), basic arithmetic instructions with integers (excluding bit shifting instructions, multiplication and division), a branch instruction (**`beq`**), and a jump (`j imm_26`) instruction. Instructions operating with floating-point numbers are excluded. Table 2.1 shows the instructions that will be implemented.

Every microarchitecture is composed of two key elements: the datapath and the control unit. The former is the executive part of the CPU and carries out the operations indicated by instructions: integer addition, logical operations, jumps, etc. The latter is in charge of generating the required control signals, in the correct order, for each instruction to be executed in the datapath.

The control unit receives some information from the datapath: the operation code of the instruction to be executed and some status flags; and generates several control signals that reach the functional units composing the datapath.

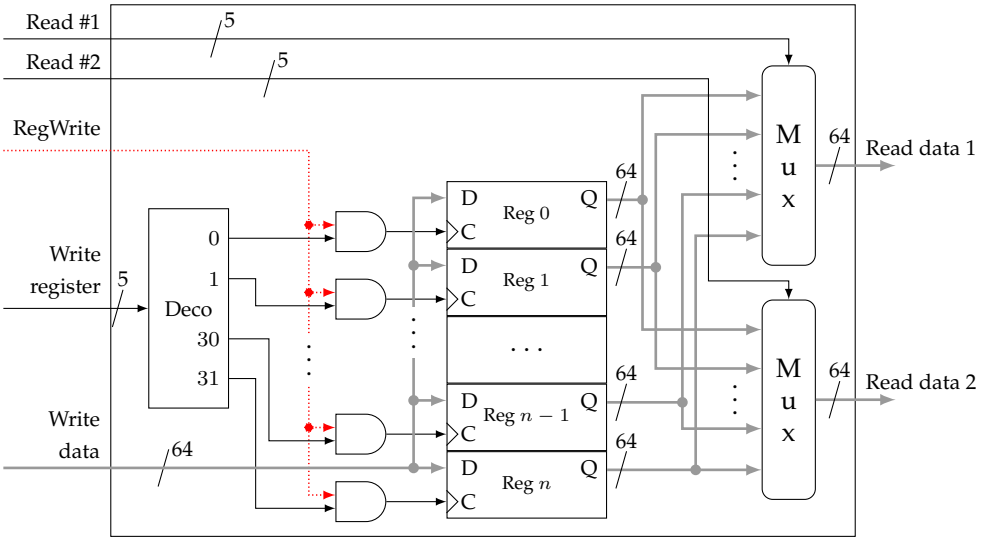


Figure 2.6: Possible register file implementation

The next sections will cover the construction of the datapath following an incremental design; the control unit for this datapath is described in appendix B.

2.2.1 Functional units

Firstly, the functional units required to implement the datapath must be identified by taking into account the instructions that will be implemented.

Thanks to the simplicity of the MIPS64 instruction set, most of instructions behave pretty similarly, which facilitates the identification of the functional units. Common steps in the execution of instructions are:

- Fetching the instruction code. Initially, the instruction code pointed by the program counter (`PC`) must be read from the instruction memory. Thus, a register with this purpose must exist in the datapath. MIPS64 addresses are 64-bit wide; thus, the `PC` register is also 64-bit wide and is implemented using 64 D-type flip-flops. This register is read to fetch the next instruction to be executed from memory and it is written to point to the next instruction, either the next in sequential order or any other if the sequential execution flow is modified as a result of a branch or a jump. This register can be implemented in the register file together with the general purpose registers.

- Register access. Most of instructions require reading one or more registers. Registers are usually implemented as positions in a multiport register file. This is an ultra-fast memory in which several locations can be accessed at the same time. The register file constitutes another functional unit. Figure 2.6 depicts a possible implementation of this unit.

Control bits	Operation
0000	AND
0001	OR
0010	addition
0110	subtraction
0111	set on less than

Table 2.2: ALU operation selection through the control bits

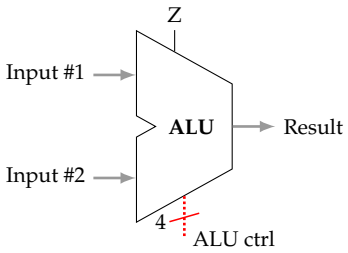


Figure 2.7: Arithmetic-logic unit

As can be seen in figure 2.6, a register file contains several registers. The reduced MIPS64 architecture contains 32 64-bit integer registers; each register is implemented using 64 D-type flip-flops. The inputs and outputs of each register in the figure reach all its 64 flip-flops. The content of the registers is always available on lines labelled with `Q`. In MIPS64 some instructions require reading two registers at the same time, such as in instruction **`dadd r2, r1, r6`**, where **`r1`** and **`r6`** must be read. Thus, two multiplexers are located at the output of the registers. Using lines `Read #1` and `Read #2`, the two registers to be read are selected. Values read from the two registers are dumped in lines `Read data 1` and `Read data 2`, respectively. Each of the two multiplexers receives 5 inputs to select one of the 32 registers. In writing operations, registers can modify their content when line `C` is set and lines `Write data` contain the 64 bits to be written. Line `C` of the register to be written is set through the line `RegWrite` at the same time that the number of the register is indicated in lines `Write register`.

- ALU. Another key component of the datapath is the arithmetic-logic unit, where arithmetic operations with integers and logical operations are carried out. The ALU in a MIPS64 CPU manages 64-bit operands. This unit receives two input operands and provides a 64-bit result. In addition, it will set the `Z` (`Zero`) flag if the result of the operation is 0. Four control input lines indicate the operation to be carried out (addition, subtraction, AND, OR, or set on less than).[1] The possible values and the meaning of these control inputs are shown in table 2.2. Figure 2.7 shows a schematic representation of this functional unit.

[1] The design of the ALU is not covered in this text.

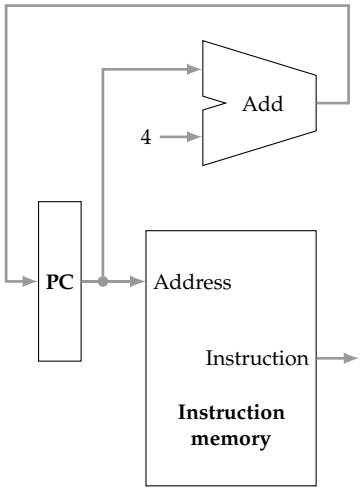


Figure 2.8: Instruction fetch and PC increment

Furthermore, some instructions will use other elements such as sign extenders (used to replicate the integer sign bit to generate 64-bit values) and bit shifters, for example jumps and branches.

2.2.2 Single-cycle datapath

This section covers the implementation of the datapath to support the subset of the MIPS64 instruction set shown in table 2.1. This implementation will be done incrementally.

As mentioned above, the first operation that all instructions must carry out is fetching the instruction code. Figure 2.8 shows the portion of the datapath which performs this operation. The value of the program counter (PC) is used to access the instruction memory and get the instruction code. This value represents the memory address where the instruction code is located. Once the instruction code is read, it is available in the lines labelled as Instruction. Finally, the program counter must be incremented in 4 units, since all instructions codes are 4-byte wide (4 memory locations). To do so, a specific adder is used, where one of its input operands is hard-coded to 4. This process of fetching the instruction code and incrementing the program counter is repeated every clock cycle.

Then, many instructions require reading one or two registers. For example, R-type instructions (see figure 2.5) require reading two registers, whereas I-type instructions require reading only one register. In any case, accessing the register file is required.

Taking into account the subset of instructions to be implemented, R-type instructions are arithmetic operations, such as **dadd r1, r2, r3**, and logical, such as **or r5, r9, r6**. These instructions read two registers, operate with them using the ALU, and finally move the result to another register. Figure 2.9 shows a possible implementation of this part of the datapath. In R-type instructions, the identifiers of

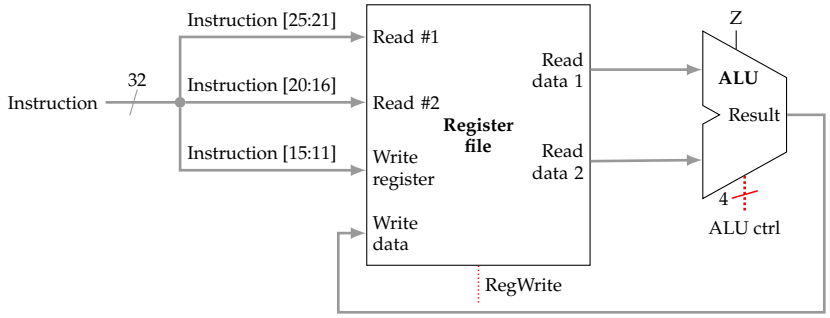


Figure 2.9: Execution of R-type instructions

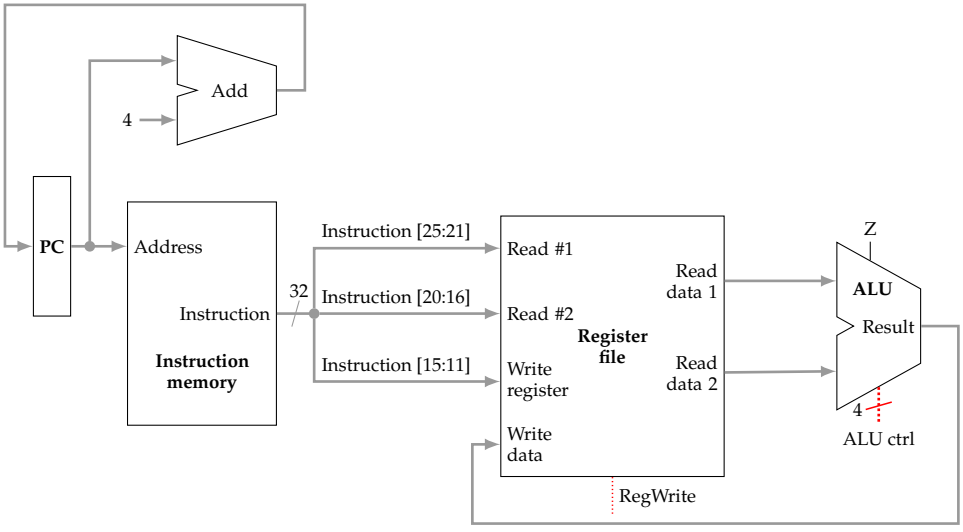


Figure 2.10: Datapath supporting R-type instructions (arithmetic instructions with integers and logical instructions)

registers to be read are encoded in the instruction code using 5 bits. The first operand is encoded in bits 25 to 21, whereas the second is encoded in bits 20 to 16. The values read from these registers must be sent to the ALU to operate with them. The result of the operation will be stored in a register identified by bits 15 to 11. The control signals that determine the behaviour of the register file and the ALU are also shown. These are set by the control unit to execute instructions in the datapath. In this case, `RegWrite` signal is set to store the result of the ALU in the specified register. A possible implementation of the control is covered in appendix B.

The first version of the datapath supporting R-type instructions can be obtained by combining the logic required to fetch the instruction and increment `PC` (figure 2.8) and the logic required to execute R-type instructions (figure 2.9). This version of the datapath is shown in figure 2.10.

The next type of instructions are loads and stores, which are I-type instructions, such as **`ld r3`**`, 200(`**`r4`**`)` and **`sd r6`**`, 12(`**`r5`**`)`. They always involve the use of a register

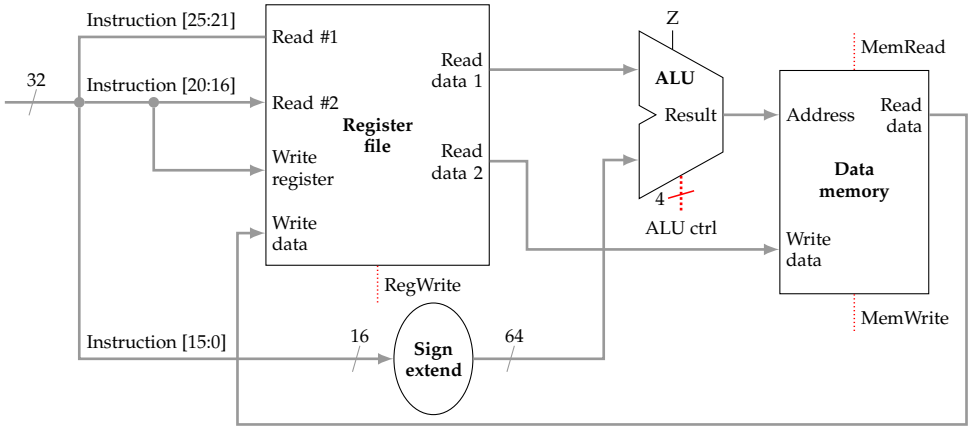


Figure 2.11: Execution of load and store instructions

(destination in the case of load, and source in the case of store), a base register and a 16-bit immediate value. The final address is computed as the addition of the base register and the immediate value with its sign extended to 64 bits. Figure 2.11 shows the required logic to support this type of instructions.

In a load instruction, the base register is obtained from bits 25 to 21 of the instruction code. This register must be added, using the ALU, to the immediate value stored in bits 15 to 0 extended to 64 bits. Hence, a sign extension unit is required. After the addition, the result is the memory address to be read. Then, `MemRead` signal must be set. Once read, the data item is copied into lines labelled as `Write data` of the register file. The register to be written is specified in bits 20 to 16 of the instruction code. Finally, `RegWrite` signal is set to store the value in the specified register.

The initial step of a store instruction is the same as in a load instruction. The memory address is obtained in the same way, that is, adding the base register and the immediate value. Differences start here. Store operations require reading a second register, encoded in bits 20 to 16 of the instruction code. The content of this register is copied into `Write data` memory lines. After setting `MemWrite` signal, the data item is stored in memory.

The logic required to execute the load and store instructions, shown in figure 2.11, cannot be directly combined with the basic datapath shown in figure 2.10. For instance, in arithmetic and logical instructions, the destination register is specified in bits 15 to 11 of the instruction code, and the value to be written in `Write data` lines of the register file is the result provided by the ALU, whereas in the load instructions the destination register is encoded in bits 20 to 16 and the value of `Write data` lines is read from memory. Therefore, multiplexers must be added with appropriate control lines to identify these situations and determine the values to use. Figure 2.12 shows the datapath using three multiplexers to manage these situations.

The multiplexer attached to `Write register` lines of the register file is used to select with `RegDst` control line the field identifying the destination register in the instruction code (bits 20 to 16 in loads and bits 15 to 11 in arithmetic-logic operations). This control line will be set in the execution of arithmetic and logical operations and

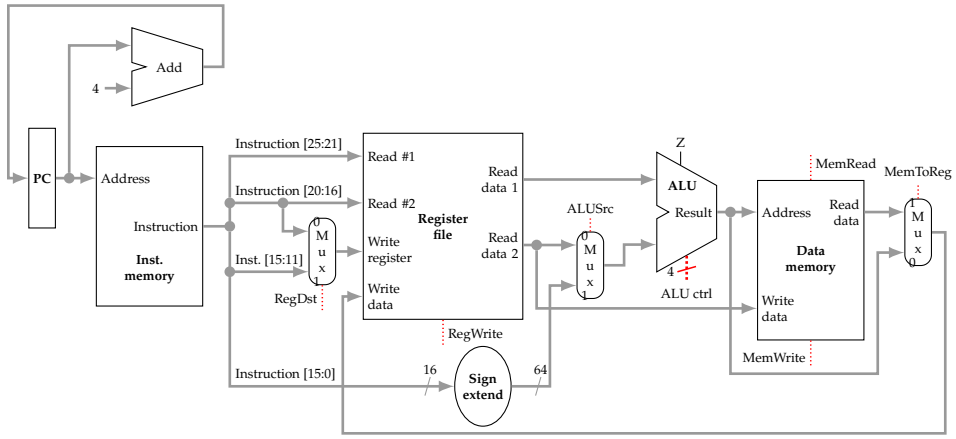


Figure 2.12: Datapath supporting R-type and load/store instructions

will be reset in loads. Another multiplexer attached to the second input of the ALU is required, since a second register or an immediate value may be used as an operand. This multiplexer is operated through `ALUSrc` line, which will be set in loads and stores, and reset in arithmetic and logical operations. Finally, another multiplexer is attached to the input of `Write data` lines of the register file, just to the right of the data memory. This multiplexer selects, through `MemToReg` line, the data item to copy to `Write data` lines: either the result provided by the ALU in arithmetic and logical instructions, or the value read from memory in load instructions.

The next instructions to add to the datapath are branches. For simplicity, only instruction **`beq`** `Rs1,Rs2,imm_16` is considered. This instruction branches if the contents of the two registers are equal. The target address is computed as the addition of the program counter and the immediate value shifted two positions to the left[2] and sign extended. An important detail is that the `PC` register has already been incremented in 4 units to point to the next instruction before the addition. Therefore, branch instructions perform two operations: check whether the branch condition is true (the ALU is used for this) and compute the target address of the branch (a specific adder is required since the ALU is in use at this moment). Figure 2.13 shows the required logic to include the branch instruction in the datapath.

The two registers read are input operands to the ALU. To check whether both operands are equal, a subtraction operation is performed. In this case, the `Z` flag will be set to 1 if the result is 0, that is, if both operands are equal. In addition, the immediate value used to compute the branch target address is extended to 64 bits and shifted two bits to the left (equivalent to multiplying by 4). Finally, the shifted value is added to the value contained in the program counter, which currently points to the next instruction to be executed (the instruction after **`beq`**). The branch is taken and the `PC` will be written with the target address if the condition is met (`Z` flag) and the `Branch` control signal is set. A generic control signal called `PCSrc` can be defined (to deal with several branch conditions), which will select the value to write in the

[2]This is done for the value to add to `PC` to be multiple of 4, that is, the width of the instruction code.

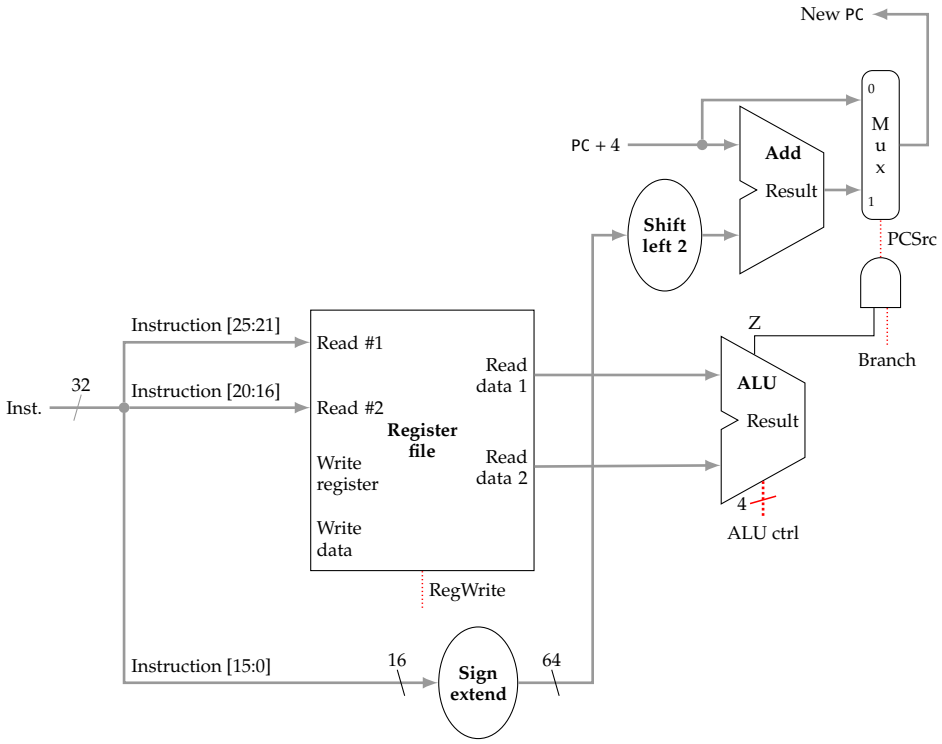


Figure 2.13: Execution of **beq** branch instruction

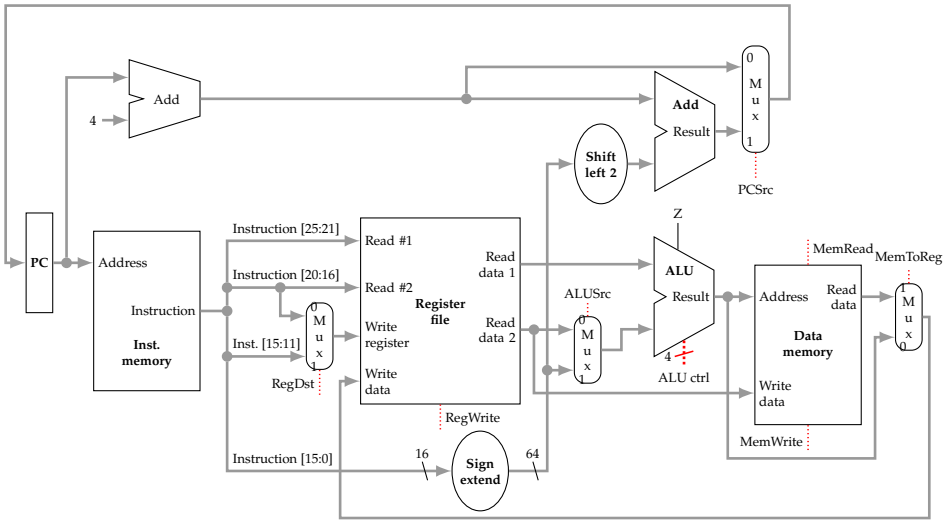


Figure 2.14: Datapath supporting R-type, load/store and **beq** branch instructions

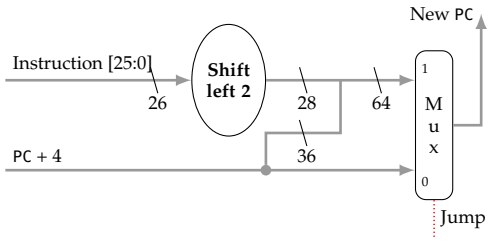


Figure 2.15: Execution of `j` jump instruction

PC register using a multiplexer attached to the output of the adder. The selection will be the address of the next instruction if the branch is not taken, or the branch target address if the branch is taken.

The logic required by the branch instruction can be easily combined in the datapath. Figure 2.14 shows the result of the combination. As can be seen, no additional elements are required.

Finally, jump instructions will be added to the datapath. Again, only one instruction is considered for simplicity: `j imm_26`. This instruction modifies the value of PC; it is an always-taken jump. The jump target address is computed with the 26-bit immediate value, shifted two bits to the left (total 28 bits), concatenated with the 36 most significant bits of the PC register to create a 64-bit value. It must also be taken into account that the value of PC has already been incremented before jumping. The logic required to implement the jump instruction is shown in figure 2.15. The control signal labelled as `Jump` governs this logic. It is set when a jump instruction is executed. Otherwise, the value written in the program counter is the original value incremented in 4 units.

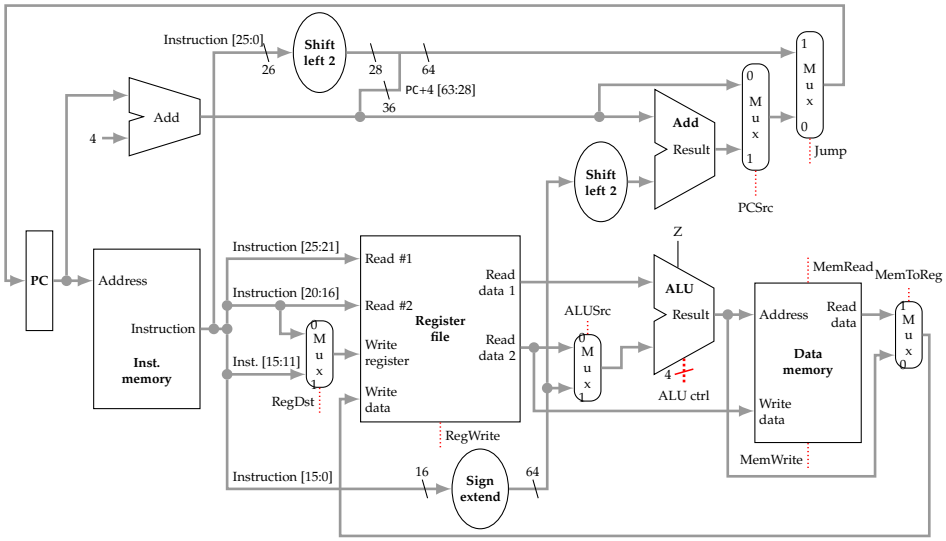


Figure 2.16: Datapath supporting R-type, load/store, **beq** branch and **j** jump instructions

No additional elements are required to include the logic to execute the jump instruction within the datapath. Nevertheless, it must be taken into account that the new value for the program counter can be determined by the logic implementing the jump or by the multiplexer attached to the output of the branch adder. Figure 2.16 shows the complete implementation of the datapath supporting the execution of the instructions listed in table 2.1.

Appendix B describes the design of the control unit required by the proposed datapath.

2.2.3 Deficiencies

Although a single-cycle implementation is feasible, modern CPUs do not follow this approach. Although the single-cycle CPU achieves a CPI of 1, the clock cycle must be large enough to allow the execution of any instruction, either a simple or a complex instruction. An example of a simple instruction is an addition, which accesses the instruction memory, the register file, the ALU and again the register file to store the result. In contrast, an example of a complex instruction is a load, which accesses the same functional units as the add but also the data memory.

All instructions will require one clock cycle; thus, it would not be possible to optimize the execution of the most common instructions. Optimizing the execution of all instructions will be the only solution to reduce the clock cycle, and thus, the execution time of programs, as dictated by the iron law of performance.

Microarchitecture	CPI	Clock cycle length (T)
Single-cycle	1	Long
Multi-cycle	>1	Short
Pipelined	1	Short

Table 2.3: Microarchitectures and clock cycles

2.3 Pipelined microarchitecture

A single-cycle microarchitecture executes every instruction in one clock cycle, but the cycle needs to be long in order to accommodate all the operations performed by the datapath. Alternatively, a multi-cycle microarchitecture, where instructions are divided into steps, is possible. Each of these steps requires one cycle, but shorter than the cycle in the single-cycle microarchitecture.

One of the foremost objectives in the CPU design is to reduce the instruction execution time, making CPU time as short as possible. CPU time comes from the iron law of performance, described in chapter 1 with the following expression:

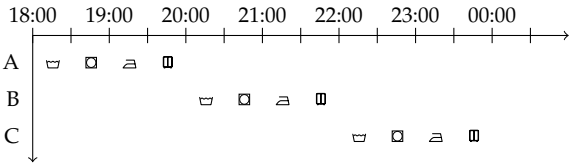
$$\mathrm{T_{CPU}} = \text{Program instructions} \times \text{CPI} \times T$$

The microarchitecture does not affect the first factor (program instructions) because this factor is determined by the instruction set. Thus, the objective becomes to execute instructions with a low CPI without increasing the clock cycle. In this section, a technique called pipelining is presented. It allows executing instructions with an ideal CPI of 1, while maintaining a short clock cycle. For this, the execution of instructions is divided into stages working in parallel, so that the CPU executes several instructions simultaneously, albeit in different stages. Table 2.3 summarizes the characteristics of several types of microarchitectures.

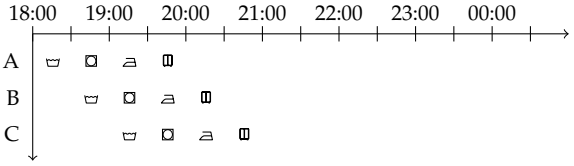
A great advantage of pipelining is that it is completely transparent to software. Instructions run faster without modifying applications or the operating system. Hence, almost all modern CPUs implement this technique.

Conceptually, pipelining is simple and applied in countless areas. An example of pipelining is the process of doing laundry, which may be divided into the following stages:

- Place one dirty load of clothes in the washing machine.
- Take the clothes from the washing machine and put them into the dryer.
- Take the clothes from the dryer and iron.
- Put the clothes in the wardrobe.



(a) Non-pipelined



(b) Pipelined

Figure 2.17: Non-pipelined versus pipelined laundry

In the non-pipelined (sequential) version of the process, the second dirty load would start only when the clothes of the first load were placed in the wardrobe. Intuitively, this is inefficient, as resources such as the washing machine, the dryer and the iron would be idle most of the time. An example of the non-pipelined version of the process is shown in figure 2.17a.

In contrast, in the pipelined version of the process, the second load of clothes would be put into the washing machine just after moving the first load from the washing machine to the dryer. Only when the washing machine and the dryer were finished would the first load be ironed, while the second load would go in the dryer and the third load would get into the washing machine, and so on. This process is depicted in figure 2.17b.

In the pipeline implementation it is important to bear in mind the need for all the stages to progress in parallel, regardless of the time required by each stage. For example, the washing machine could have a cycle of 40 minutes, the dryer 50 minutes, the ironing could take 45 minutes and placing the clothes in the wardrobe could take 20 minutes (40, 50, 45, 20). This means that the washing machine will wait 10 minutes for the dryer, which is the slowest stage, to finish in order to proceed with the next load. It is clear that the most favourable situation is when all the stages are busy all the time, i.e., when the stages are balanced, requiring the same time to complete. In any other case, the advance of the pipeline is given by the slowest stage.

The pipeline technique improves throughput (number of loads per time unit), but not response time (time to perform a single load). For example, assuming the ideal case where all the stages are balanced and need 30 minutes to complete, each load still needs to go through the four stages, which requires 2 hours in total. However, if the number of loads to be done is high, there will be a time when all the stages of the laundry will be working in parallel, thus, every 30 minutes a new load will be placed in the wardrobe. This implies an ideal speedup of 4, the number of stages, which means that ideally it will be possible to perform 4 times more loads per time unit with the pipelined version of the process. In this ideal case, times with idle

stages are not taken into account (at the beginning, until the first load of clothes is put in the wardrobe, and at the end, since the last load of clothes gets into the dryer).

However, it is extremely difficult to achieve the ideal situation where all the stages require the same time, i.e., they are balanced. In practice, the improvement in throughput is lower and response time increases due to the waiting times between stages. Returning to the initial example where each stage requires a different time (40, 50, 45, 20), some direct calculations can be made. The time necessary to perform a load in non-pipelined mode is calculated as the addition of the times of each stage: 155 minutes. Considering now the pipelined version, all the stages move forward in parallel, so the minimum time of the stages will be 50 minutes. From this time, the real gain in terms of throughput can be calculated (ignoring transient periods with idle stages), which is less than or equal to the ideal in any case.

$$A_{\text{Throughput}} = \frac{155}{50} = 3.1$$

Response time is calculated by adding the times of all the stages, i.e., 200 minutes, so this performance metric worsens compared to the non-pipelined version. It is important to note that pipelining improves throughput but not response time in the execution of instructions: the latter could get even worse.

The same principles apply to the execution of instructions in order to build pipelined CPUs. In the case of MIPS, instruction execution is commonly divided into five stages:

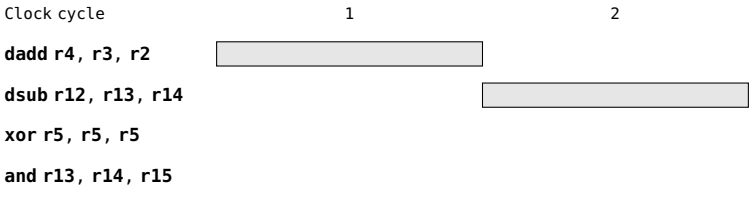
1. `IF` (**I**nstruction **F**etch): fetching instruction from memory. The instruction code pointed to by the program counter is read from the instruction memory and the program counter is incremented by 4, making it point to the next instruction.

2. `ID` (**I**nstruction **D**ecode): decoding instruction and reading registers acting as instruction operands. In this stage, the instruction is decoded, the register file is accessed to read the two source registers (they are always read, although they may be discarded later because the instruction does not use them), the sign of the 16-bit immediate value is extended (even if the instruction is not I-type) and the 26-bit immediate value shifted (even if the instruction is not J-type). In this stage the program counter is also updated with the destination address of jump instructions.

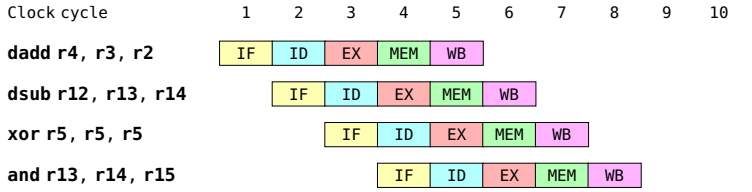
3. `EX` (**EX**ecute): execution in the ALU and computation of the target address of branches. The ALU is used during arithmetic-logical instructions, the evaluation of the condition of a branch and the computation of the memory address in the case of load/store instructions.

4. `MEM` (**MEM**ory): access to memory in load/store instructions. In this stage the data memory is accessed. In addition, the program counter is updated with the previously computed target address in branches.

5. `WB` (**W**rite **B**ack): write the result back to the register file. The register file is written after an ALU operation or after a load from memory.



(a) Non-pipelined execution



(b) Pipelined execution

Figure 2.18: Non-pipelined and pipelined execution

The time required for all stages to finish defines the clock cycle in a pipelined CPU. The pipelined datapath moves forward with each clock cycle. Thus, the shorter the duration of the stages, the lower the clock cycle of the CPU and, as a result, the higher the working frequency. A direct consequence is that the CPU frequency can be increased simply by increasing the pipeline depth, that is, increasing the number of stages. However, as previously stated, it is extremely difficult to design deep balanced pipelines, even ignoring the complexity of controlling a deep pipeline, as branches give rise to new problems, as will be seen later.

As a summary, although the pipeline does not improve the execution time of instructions individually (in fact, it usually worsens it), it does increase the throughput of the CPU when executing instructions. It is important to note that throughput is the most important performance metric from a practical point of view, because programs are composed of a large number of instructions that must be executed as quickly as possible, so the execution time of a single instruction is less important.

Figure 2.18a shows an example of representation for the non-pipelined execution of instructions in a single-cycle CPU. All instructions are executed sequentially and require a long clock cycle. In contrast, figure 2.18b depicts the execution in the pipelined CPU. In this case, the CPI observed under ideal conditions is also 1, but with a much shorter clock cycle. The pipelined version is assumed to be derived from the single-cycle version by dividing the execution into 5 perfectly balanced stages. Hence, the clock cycle of the pipelined CPU is exactly one fifth the clock cycle of the single-cycle CPU. It can be noted that in the time required by the pipelined CPU to execute the four instructions, the single-cycle CPU is still executing the second instruction.

In theory, if the execution of instructions is divided into 10 stages instead of 5, throughput will be multiplied by 10, i.e., the number of instructions per second that the pipelined CPU can execute in comparison to the non-pipelined version. However, the balance of stages and the pipeline management by the control unit become

more complex as the number of stages increases. In practice, CPUs used in today computers usually implement between 10 and 20 pipeline stages. For example, the Intel Core i7 processor based on *Nehalem* microarchitecture has a 14-stage pipeline.

In addition to the huge challenge of balancing the pipeline stages, which makes it difficult to reach the theoretical maximum speedup in a pipelined CPU (equivalent to the number of stages), there are other issues, called **pipeline hazards**. Pipeline hazards decrease even more the speedup obtained by pipelined CPUs and will be studied later using a pipelined implementation of the MIPS64 architecture as a reference.

2.3.1 Pipelined datapath

Figure 2.19 shows the single-cycle datapath developed in section 2.2.2, identifying the components of the 5 stages described in the previous section that implement the pipeline. The balance of the stages in this proposal is high. On one hand, the three slowest stages are `IF`, `EX` and `MEM`, since they access memory or use the ALU. On the other hand, stages `ID` and `WB`, which access the register file, are the fastest. `ID` also decodes the instruction and computes jump addresses. However, it is important to remember that all stages must move forward at the same time, so the CPU clock cycle duration is imposed by the slowest stage.

As can be seen in figure 2.19, the instruction code and the data it operates with move from the left side of the datapath to the right side with two exceptions. Firstly, the result of an operation in the ALU or the data read from memory must be written to the register file during the `WB` stage. Secondly, when `PC` is updated as a result of a jump or a taken branch. As will be seen later, data moving from left to right in the datapath does not have undesirable effects on the instructions currently executing, in contrast to the movements in the opposite direction.

Another point to consider is that not all instructions make use of the resources of all the stages, but they must go through each of these 5 stages anyway. For example, instruction `dadd r3, r2, r1` must go through each of the 5 stages, even if it does not access memory in its `MEM` stage. It simply goes through this stage without taking any action.

In order for all the stages in the pipeline to work on a different instruction and progress at the same time, it is necessary to introduce new elements in the datapath. These are the so-called **pipeline registers**, which work as buffers between stages. Figure 2.20 shows the datapath with the pipeline registers. At the beginning of the cycle, pipeline registers and `PC` are read to obtain the information that each stage requires to work. At the end of each clock cycle, pipeline registers and `PC` store the information placed at their inputs. In this way, data generated by the stages progress from one pipeline register to the next and to the `PC` in each clock cycle. In addition, a slight modification is introduced compared to the datapath shown in the figure 2.19 for simplicity. The two multiplexers that determine the value to be written to `PC` are combined into one with three entries: `PC`+4, the target address of the branch and the target address of the jump.

Back to the analogy of doing laundry, a pipeline register would be the basket where clothes would be placed once washed by the washing machine until they are put into the dryer.

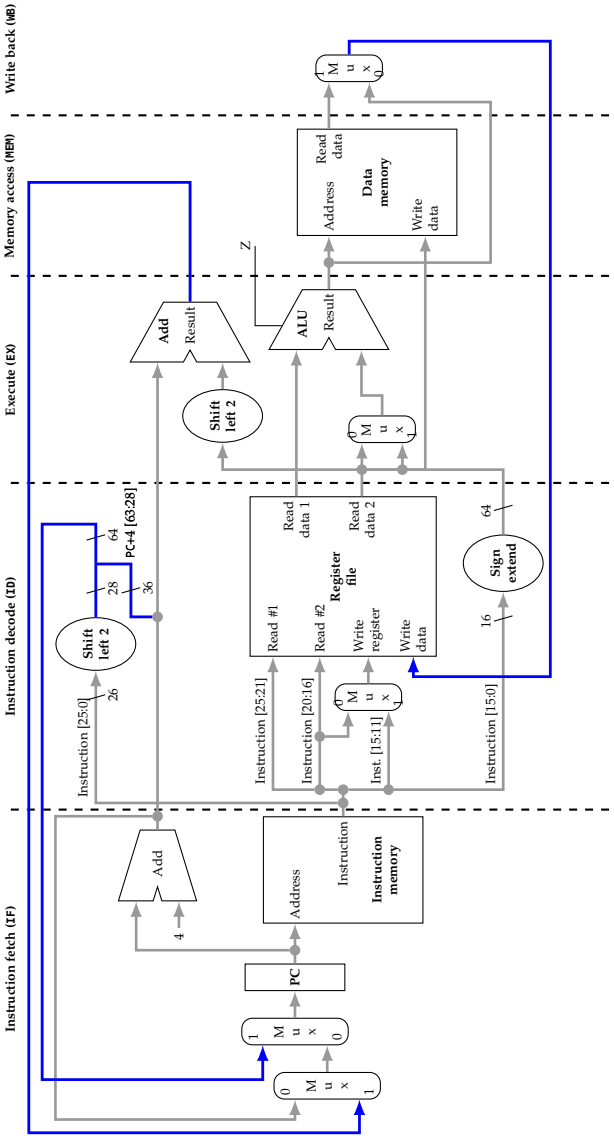


Figure 2.19: Single-cycle datapath identifying the stages of the pipelined datapath

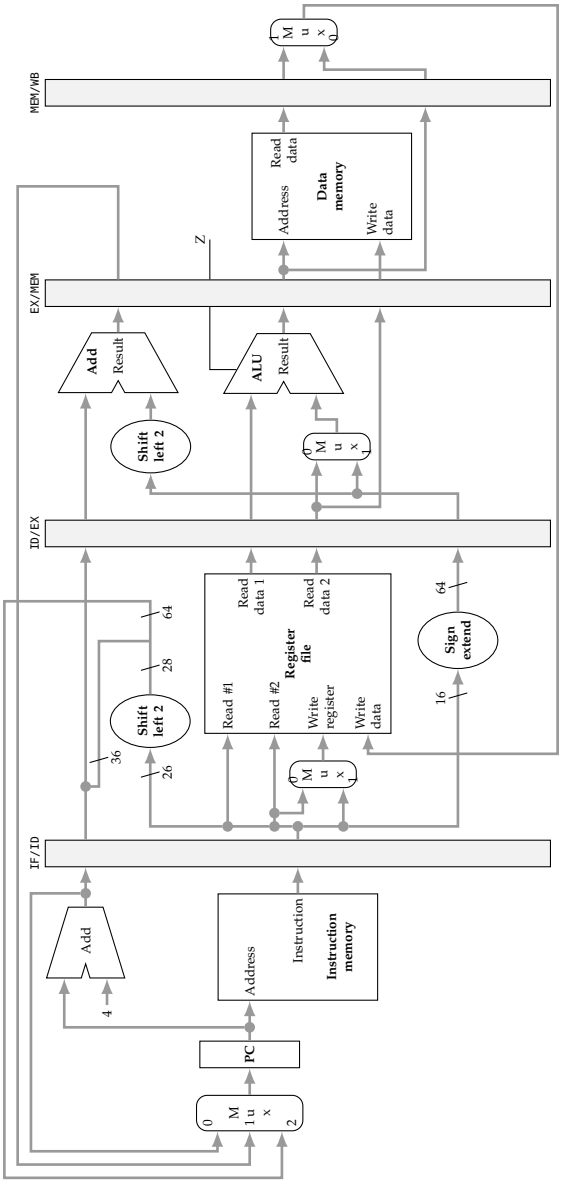


Figure 2.20: Pipelined datapath

Pipeline registers must be large enough to store all the data that flows from one stage to another. In the implementation of figure 2.20, register IF/ID must have a size of 96 bits (64 bits for the incremented PC value and 32 for the instruction code), register ID/EX has a size of 256 bits, register EX/MEM has a size of 193 bits and register MEM/WB has a size of 128 bits.

Starting from this initial implementation, some modifications are necessary to introduce the control of the pipeline. This is described in appendix B.

2.3.2 Pipeline hazards

There are situations, called pipeline hazards, that may require the parallel execution of instructions to stop temporarily. An instruction is stalled in a pipeline stage whenever some condition for completing the stage is not met. In this case, the input pipeline register is not modified at the end of the clock cycle, so the stage is repeated with the same instruction in the next clock cycle. This means that the previous stages must also be stalled so their output pipeline registers are not modified. However, the instructions that are in the stages after the stage that caused the stall may continue.[3]

A stalled stage should not be confused with an idle stage. In the idle state, the stage does not have valid information in its input pipeline register. In this situation, the stage does not appear in the execution timelines in the considered clock cycle.

Pipeline hazards can be classified into three types: structural hazards, data hazards and control hazards. The following sections detail each of these types of hazards.

Structural hazards

Structural hazards appear when hardware cannot execute in parallel instructions that require the use of the same resource.

Structural hazards are not possible in the pipelined datapath proposed in figure 2.20, since each stage uses resources independent of the rest of the stages. To illustrate the consequences of a structural hazard, a datapath with a single memory for data and instructions, instead of two, is considered while executing the following code snippet.

```
ld   r11, 120(r0)
dadd r2,  r3,  r6
xor  r15, r15, r15
dsub r4,  r1,  r9
and  r12, r13, r10
```

Figure 2.21 shows the execution timeline of the previous code snippet considering a single memory.[4] As can be seen, the pipelined execution is ideal until it is time to proceed with the MEM stage of instruction `ld r11, 120 (r0)`, at cycle #4, which requires reading data from memory (the rest of the instructions do not have operands

[3] The stage causing the stall resets its output pipeline register so the next stage is idle in the next clock cycle.

[4] This would imply modifying the datapath by adding a new multiplexer to select the address to use in memory accesses: the one that comes from PC, or that computed in EX.

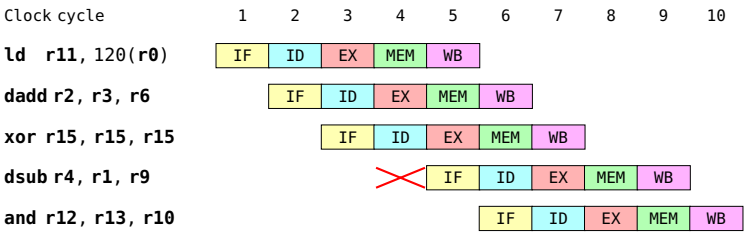


Figure 2.21: Pipelined execution assuming a single memory

in memory). As the memory is unique, the IF stage of instruction **dsub r4, r1, r9** cannot fetch the instruction code at the same time, so a stall occurs and the stage is postponed to the next clock cycle. Pipeline stalls (a single clock cycle in the example) are propagated to the following instructions, which have been delayed. For convenience, structural stalls are represented by a cross mark to distinguish them from other types of stalls instead of repeating the stage. It can also be seen in the figure how the ID stage is idle in cycle 5 (the one after the stall), hence it does not appear in the timeline.

Detecting structural hazards and stalling the pipeline require the use of additional hardware in the MIPS64 microarchitecture studied, which depends on the implementation. This is beyond the scope of this text.

Data hazards

A data dependency is a situation where an instruction refers to the data (register or memory location) of a preceding instruction. A data dependency may lead to stalling the pipeline when an instruction depends on the result of another instruction that is still in the pipeline. This is clearly seen in the following code snippet.

```
dadd r2,  r3,  r4
dsub r1,  r10, r2
or   r12, r11, r14
xor  r18, r18, r18
```

The first instruction adds the values contained in registers **r3** and **r4** and stores the result in register **r2**. The following instruction uses this calculated value to perform the subtraction. The issue appears when the second instruction reads **r2** (ID stage), since the result of the addition has not been written to the register yet. This result is written during the WB stage of the first instruction, so it is necessary to stall the pipeline until the correct value can be read from **r2**.

Figure 2.22 shows the timeline of instructions taking into account the stall produced by the data hazard. The ID stage of **dsub** must be repeated until the correct value of **r2** is read at clock cycle #5. Register **r2** is written in the WB stage of **dadd** at the same time as it is read in the ID stage of **dsub**. This behaviour is possible since the writing to the register file is performed in WB during the first half of the clock cycle and the reading in ID is carried out during the second half of the clock cycle.

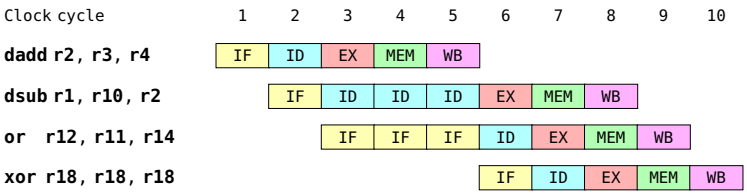


Figure 2.22: Pipelined execution with a data stall

However, this overlap of writing and reading operations in the same clock cycle is not possible in the case of memory.

It can also be observed how the following instruction (`or`) is still in `IF` until instruction `dsub` can continue.

An important detail to keep in mind is that it is not necessary for two instructions exhibiting data dependence to be consecutive so that there is a stall. In the previous example, the stall would occur even if an instruction existed between them. Pipeline stalls produced by data dependencies between instructions reduce the performance speedup of the pipeline. In the previous example, it takes 10 cycles to run the code instead of the 8 cycles it would require without the stall.

The above data dependency is only one example of the different types that can appear. Next, other types of data dependencies that can occur between instructions are formally described.

- Read After Write (RAW). It is the dependency that appeared in the previous code snippet. It occurs when an instruction reads a register that has been previously written by a previous instruction. This type of dependency is also known as true dependency.

  The control unit must detect this dependency and check if there is a hazard. In that case, it must stall the pipeline during the necessary cycles until the hazard disappears. In the case of the MIPS64 microarchitecture presented, data hazards due to register operands are possible, but not due to memory operands. Memory is accessed in the `MEM` stage of load and store instructions (two `MEM` stages cannot be carried out at the same time). The following code fragment shows this situation.

  ```
  sd r3, 120(r0)
  ld r5, 120(r0)
  ```

  The `ld` instruction reads memory location `120(r0)` after it has been written by `sd`. It is not necessary to check if this dependency leads to a stall, since the reading and writing of data memory occurs in the same stage, `MEM`, and `sd` arrives at this stage before `ld`. This beneficial feature is not the result of chance, but it is a consequence of a RISC instruction set where only load and store instructions access the data memory. This is an example on how ISAs are designed to simplify the pipeline, reducing costs and improving performance.

- Write After Read (WAR). WAR dependencies are also called antidependencies. This code snippet is an example of WAR data dependency.

```
dsub r1,  r10, r2
dadd r2,  r3,  r4
or   r12, r11, r14
```

  According to the code semantics, the writing in registry `r2` by `dadd` must come after reading this register by `dsub`.

  Writing of registers in the MIPS64 microarchitecture is performed in stage `WB`, while reading occurs in stage `ID`. Since instruction `dsub r1, r10, r2` will get to stage `ID` before instruction `dadd r2, r3, r4` reaches stage `WB`, this pipeline hazard is not possible.

  In the case of memory operands, WAR dependencies can never cause pipeline stalls, since both reading and writing are carried out in the same stage: `MEM`.

  However, as seen later in this chapter, one of the pipeline improvement techniques is out-of-order execution. Using this technique, the execution of the write instruction could start before the reading, so it would be necessary to check for hazards due to WAR dependencies.

- Write after write (WAW). It defines the order of two writes on the same operand. It is also called output dependency. An example of WAW dependency is shown below.

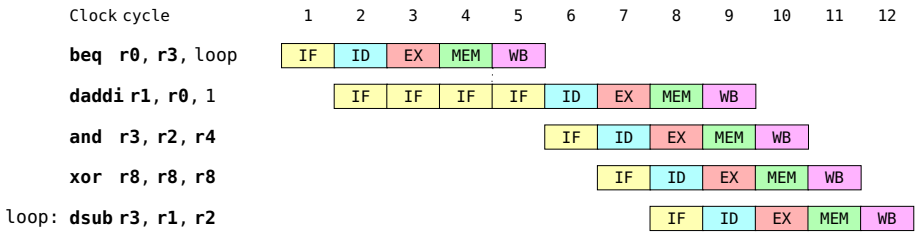
```
dsub r1, r10, r2
and  r1, r3,  r4
xor  r7, r1,  r6
```

  According to the code semantics, the writing in register `r1` by `and` must happen after the writing in this register by `dsub`. Otherwise, instruction `xor`, or any other coming later that uses `r1`, would receive an incorrect value. Again, in the case of the proposed MIPS64 microarchitecture, this type of dependency can be ignored, as there is only one stage writing to the register file: `WB`. The same applies to memory operands; only `MEM` writes to memory. However, this dependency may lead to pipeline hazards in the event of out-of-order execution as with WAR dependencies.

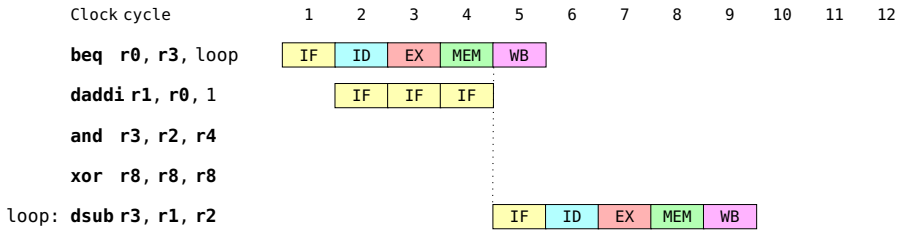
Dealing with data hazards requires hardware support within the MIPS64 microarchitecture. The precise way to do it depends on the implementation and is beyond the scope of this text.

Control hazards

Control hazards occur when the execution flow changes, such as in function calls, jumps, branches, interrupts or exceptions. Under these conditions, the sequential execution flow is interrupted, and this prevents the CPU from executing the initial stages of the following instructions until the target address is resolved. The following code snippet illustrates this situation.



(a) Not-taken branch



(b) Taken branch

Figure 2.23: Control hazard in the pipelined execution of a branch

```
      beq   r0, r3, loop
      daddi r1, r0, 1
      and   r3, r2, r4
      xor   r8, r8, r8
loop: dsub  r3, r1, r2
```

The first instruction is a branch. Depending on whether the branch is taken or not, the next instruction to execute will be **dsub** or the instruction after the branch (**daddi**), respectively. The problem is that PC is updated with the address of the next instruction in the MEM stage of **beq**. Thus, it is necessary to stall the pipeline until determining the next instruction to execute. Figure 2.23 shows the execution timeline of the previous code under the assumptions of branch taken or not taken.

It is interesting to note that when the branch condition is false, the ID stage of **daddi** could be moved from cycle #6 to cycle #5, reducing the stall from 3 to 2 clock cycles. At the end of cycle #4, IF has already fetched instruction **daddi**, so it would be unnecessary to repeat the fetching for this instruction. However, this optimization would complicate the MIPS64 microarchitecture, since it would be necessary to take into account whether the branch condition is true or not to advance the ID stage. This optimization will not be considered for simplicity.

In the case of jumps, PC is updated in the ID stage, so the duration of the stall would be shorter: one cycle. If instruction **beq** in the previous code snippet were replaced by **j** loop, the execution timeline would be that shown in figure 2.24.

Control hazards are a particular case of RAW dependency related to the PC register. The target address and the condition of branches are evaluated in EX. However, it is not until the next stage, MEM, when PC is updated, so for practical purposes the

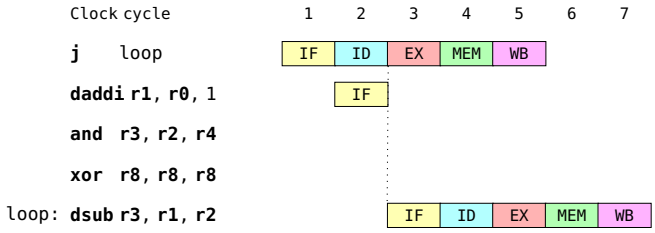


Figure 2.24: Control hazard in the pipelined execution of a jump

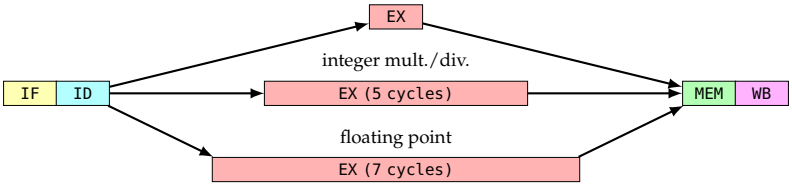


Figure 2.25: Integer and floating-point multi-cycle operations

evaluation of a conditional branch occurs in the MEM stage. Thus, the stage writing PC, MEM, comes after the stage reading PC, IF, hence the data hazard (on the PC register). The same applies in the case of jumps, but now the stall is shorter, since the modification of PC occurs in ID.

As with RAW hazards, control stalls require repeating the affected stage (IF) and the previous ones (which do not exist) until the value of the PC register is available.

2.3.3 Multi-cycle operations

Up to now, it has been considered that the MIPS64 microarchitecture implements instructions that require a single cycle in the EX stage. For example, the ALU in the EX stage can perform addition and subtraction of integers in one clock cycle. The problem arises when trying to implement more complex operations such as multiplication and division of integers, or floating-point operations. All these operations require more time (theoretically several clock cycles) to complete.

A simple solution would be to increase the duration of the clock cycle to accommodate the longest operation to be carried out. However, this would drastically reduce performance, since it would slow down the entire pipeline. A more suitable solution is to add multi-cycle EX stages to perform complex operations. For example, a multi-cycle EX stage could be added to implement integer multiplications and divisions and another for floating-point operations, as depicted in figure 2.25.

If the instruction performs a multiplication or division of integers, the multi-cycle EX stage for integer multiplication/division will be used, as shown in the figure. On the other hand, if it is a floating-point operation, the EX stage for floating-point operations will be used instead. In any other case, the usual EX stage would be used.

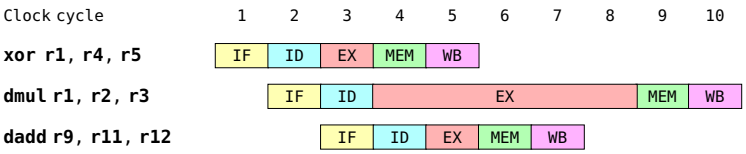


Figure 2.26: Performance improvement when using several execution units

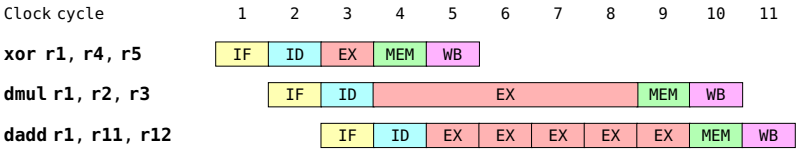


Figure 2.27: Example of a WAW stall produced by multi-cycle execution units

The use of several execution units enables some performance improvements. For example, an integer addition instruction can be issued to its execution unit while an integer multiplication instruction is being executed in the corresponding unit. To support this improvement, every execution unit must have its own input pipeline register. This situation is shown in figure 2.26 for the following code, assuming that the EX stage for integer multiplication/division requires five clock cycles.

```
xor  r1, r4,  r5
dmul r1, r2,  r3
dadd r9, r11, r12
```

Instruction **dadd** enters the EX stage (it is issued) before the preceding instruction **dmul** leaves the same stage, because it uses a different execution unit, which is available. It is said that both instructions are issued and executed in-order (they go into the EX stage and start this stage in the order of the program).[5] In addition, instruction **dadd** terminates before the preceding instruction **dmul**, so both instructions terminate or are retired out-of-order. In any case, instructions are always fetched (IF) and decoded (ID) in order, but issue, execution (EX) and retirement (MEM and WB) can be out-of-order (OoO).

Out-of-order issuing, execution and retirement improve performance at the expense of complicating the microarchitecture, since handling data dependencies becomes more complex. For example, if instruction **dadd r9, r11, r12** were replaced by instruction **dadd r1, r11, r12**, the latter could not terminate before the preceding instruction, since there is a WAW dependency between both instructions on register **r1**. Otherwise, the value of register **r1** at the end of the code snippet would be wrong. The result is a 4-cycle stall due to the writing of **r1**, shown as blanks in figure 2.27. This stall increases the execution time by one clock cycle.

[5] Technically, issuing refers to sending an instruction to the corresponding execution unit, while execution is the moment when the instruction is being executed in this unit. Buffers may exist to make these two processes independent.

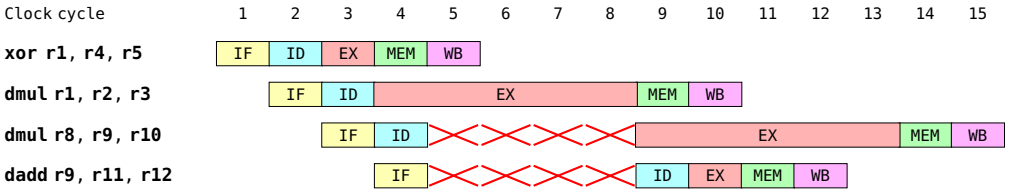


Figure 2.28: Execution on non-pipelined multi-cycle units

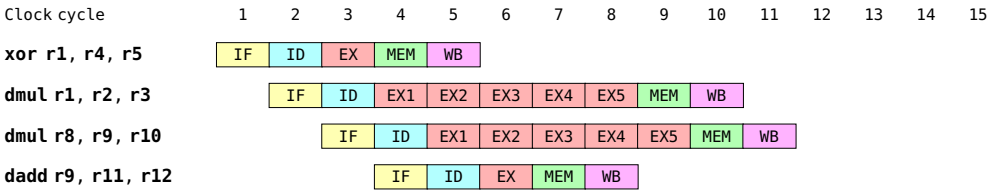


Figure 2.29: Execution on pipelined multi-cycle units

The execution of multi-cycle operations brings another effect that did not exist in the MIPS64 microarchitecture: structural stalls are now possible in the EX stage. To illustrate this point, the following code snippet is shown as an example.

```
xor  r1, r4,  r5
dmul r1, r2,  r3
dmul r8, r9,  10
dadd r9, r11, r12
```

The second **dmul** instruction cannot enter the execution stage until the first **dmul** leaves it, giving rise to a structural stall of 4 cycles. The execution timeline is shown in figure 2.28. This stall involves a performance penalty, because it increases the CPI. Observe how the structural stall propagates to the following instructions in the pipeline; **dadd r9, r11, r12** in this case. This restriction will be removed in the section devoted to the superscalar microarchitecture.

In order to reduce or remove the structural stall, it is possible to pipeline the multi-cycle multiplication/division stage in as many stages as cycles needed, in such a way that they work in parallel. Note that the pipelining technique can be applied to any digital system, not just the CPU. Figure 2.29 shows the execution once the integer multiplication/division unit has been pipelined. A significant improvement can be observed, since the stall due to the structural hazard between the two multiplication instructions disappears.

2.3.4 Exception handling

Exceptions introduce changes in the normal flow of program execution. Depending on the architecture, the names faults or interrupts are also used. Once an exception occurs, the execution of the program is stopped, an operating system routine is

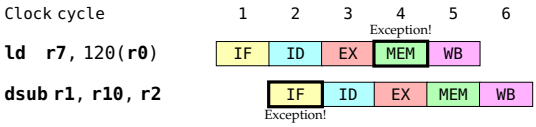


Figure 2.30: Multiple exceptions in the pipelined execution

executed and, in general, the execution of the program resumes where it was left. Exceptions can occur during the execution of instructions or after they finish.

For example, an instruction that divides by zero or an instruction with an invalid instruction code cannot be executed. In both cases an exception is triggered. In the context of the pipeline, interrupts requested by peripheral interfaces are also considered exceptions.

It is desirable that pipelined microarchitectures behave similarly to non-pipelined ones when an exception occurs. Instructions in the pipeline following the one that generated the exception should stop. If the exception occurs in the middle of an instruction, it should also stop. In contrast, the preceding instructions should be completed. When this goal is achieved, it is said that the pipelined microarchitecture supports **precise exceptions**.

Current CPUs support precise exceptions, but not some old pipelined CPUs, which required some support by exception handlers. In those cases, the operating system was responsible for the correct execution of instructions in the presence of exceptions.

The problem of exceptions in pipelined microarchitectures is that they can appear in different stages of various instructions. For example, the following program might result in an exception at the `MEM` stage of **`ld`** and also an exception at the `IF` stage of **`dsub`**, as shown in figure 2.30.

```
ld   r7, 120(r0)
dsub r1, r10, r2
```

The exception in **`ld`** occurs after the exception in **`dsub`**, but it should be processed before, because **`ld`** appears earlier in the program. If exceptions were processed just at the moment they appear, they would not be precise.

The solution consists in processing exceptions not at the moment they occur, but later when the execution of the instruction reaches the last stage: `WB`. Thus, the exception of **`ld`** arrives to `WB` before the exception of **`dsub`**. For the instruction to advance until `WB`, the pipeline should not stall, disabling the writings in registers and memory for the instruction raising the exception and all that follow in the pipeline, so that these instructions cannot modify the state of the machine.

Another problem associated with exceptions is the out-of-order retirement of instructions: an instruction that raises an exception may arrive to the stage `WB` later than another instruction that is after it in the code. Figure 2.31 shows this situation.

In order to implement precise exceptions, instructions must be retired in-order. Retirement of instructions is carried out when they write (that is, in stages `MEM` and `WB`) and therefore they modify the state of the machine. Figure 2.32 illustrates the

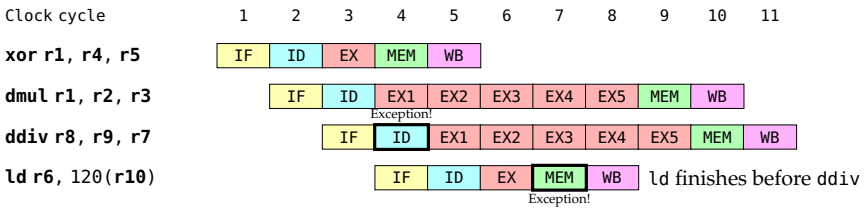


Figure 2.31: Out-of-order retirement may result in non-precise exceptions

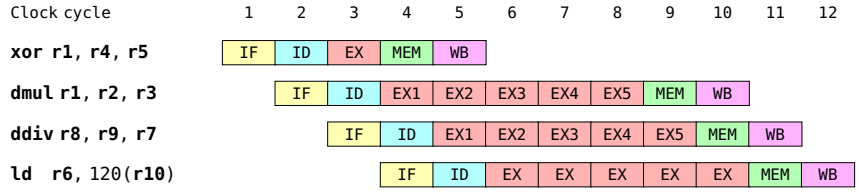


Figure 2.32: Precise exceptions require in-order retirement

same previous example with in-order retirement and how this restriction reduces the performance.

Although the support of precise exceptions requires the retirement of instructions in-order, instructions can be issued and executed out-of-order, offering opportunities for performance improvement, especially in the case of superscalar CPUs, which will be studied in section 2.4.2.

2.3.5 Reducing data hazard stalls

Pipelined CPUs may retire a maximum of one instruction per clock cycle under ideal conditions. However, as seen before, there are hazards that prevent the pipeline from achieving this value in practice, so that the CPI of these CPUs will always be greater than 1. For the case of the presented MIPS64 pipelined microarchitecture, only stalls due to data dependencies and control hazards are possible. These stalls result in penalties of three clock cycles at most, which causes an increase in the average CPI of programs.

This section shows the basic techniques used to reduce the impact of data dependencies between instructions to bring the performance of the pipeline close to its theoretical maximum. These are illustrated on the studied MIPS64 microarchitecture. The techniques to minimize stalls for control hazards will be seen in the following section. In this section, three techniques will be proposed to reduce or eliminate stalls due to data hazards:

- Instruction scheduling. The scheduling can be performed by the compiler, so it does not require changes in the microarchitecture, or by the hardware directly.
- Forwarding paths. They are implemented in hardware and are transparent to software.

- Register renaming. It is also implemented in hardware and is transparent to software.

All these techniques eliminate or reduce stalls without altering the semantics of the program, that is, without changing the results of the program. Register renaming focuses on eliminating dependencies, while the other two techniques maintain the dependencies but focus on reducing the stalls they produce.

At this point, it is worth remembering the different types of data dependencies:

- Read After Write (RAW). A memory location or a register cannot be read until a preceding instruction that writes to the memory location or register has written the new value.
- Write After Read (WAR). An instruction reading from a memory location or register must be executed before a subsequent write instruction. This restriction must be imposed when out-of-order execution is allowed.
- Write After Write (WAW). If one instruction that writes into a memory location or register precedes another write instruction, out-of-order execution cannot cause that the execution of the first write is after the second one.

RAW dependencies are also called true dependencies, because they involve a data transfer from one instruction to another within the program semantics, unlike WAR and WAW dependencies. These last two are artificial dependencies produced by the register reuse carried out by compilers.

Instruction scheduling

Instruction scheduling is a technique that involves reordering the instructions to execute so that data dependencies do not produce stalls and, at the same time, the program semantics are preserved. This involves placing independent instructions between two instructions that maintain a dependency, "moving the dependent instructions away" and thus avoiding pipeline stalls, or at least reducing their duration. In any case, data dependencies of the original program must be preserved.

The following code snippet is used to illustrate RAW dependencies in the MIPS64 microarchitecture. In this case, this code results in a stall of two clock cycles:

```
dadd r2,  r3,  r4
dsub r1,  r10, r2
or   r12, r11, r14
```

The first two instructions depend on register `r2`. However, instruction `or r12, r11, r14` is independent, because it uses different registers. For this reason, this instruction could be placed between the dependent instructions without changing the semantics of the program, as can be seen below.

```
dadd r2,  r3,  r4
or   r12, r11, r14
dsub r1,  r10, r2
```

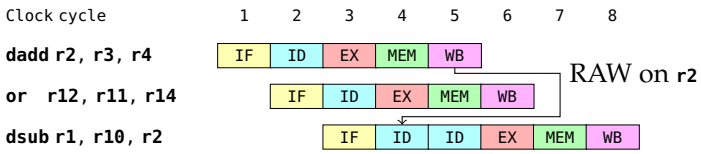


Figure 2.33: Timeline after reordering instructions

The stall produced by the data dependency is reduced from two cycles to one cycle, as can be seen in figure 2.33, where the `ID` stage of **`dsub`** is repeated. In fact, if two independent instructions were placed between instructions **`dadd`** and **`dsub`** there would be no stall at all.

This technique can be implemented at the software level by the compiler or at the hardware level. In the first case, it is known as static instruction scheduling, since it is done at compile time. The disadvantage of being implemented in the compiler is not only the increase of its complexity, but also that it may be necessary to recompile programs to adapt them to each microarchitecture, even though the CPU ISA does not change.

When instruction scheduling is implemented at the hardware level, the microarchitecture is responsible for the reordering of instructions. Decisions are taken on-the-fly, as instructions are executed and dependencies appear. This technique is called dynamic instruction scheduling.

Forwarding paths

Forwarding paths are connections in the datapath used to reduce the stalls due to RAW dependencies, which are only possible between registers (never memory locations) in the studied MIPS64 microarchitecture. Register writing takes place in the first half of the `WB` stage, while reading takes place in the second half of the `ID` stage. The longest stall is two cycles and appears when the write instruction and the read instruction are consecutive, as in the following program, depicted in figure 2.22 of page 47.

```
dadd r2,  r3,  r4
dsub r1,  r10, r2
or   r12, r11, r14
xor  r18, r18, r18
```

The value of register **`r2`**, required by **`dsub`**, is calculated when **`dadd`** goes through the `EX` stage, but it is written to **`r2`** in the first half cycle of the `WB` stage. If it were possible to bring the result of stage `EX` to the input of this same stage on the next clock cycle, instead of that provided by the `ID` stage, **`dsub`** could enter the `EX` stage on the next clock cycle, removing the stall.

Figure 2.34 shows the execution of the previous code when a forwarding path is available from the output of the `EX` stage to the input of the same stage.

The implementation of this forwarding path in the datapath is simple. It suffices to include a multiplexer at the input of the `EX` stage to select the value of the register

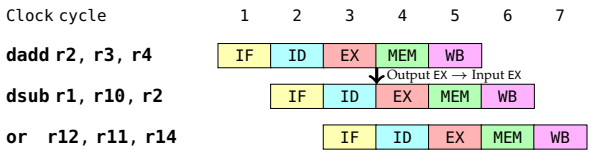


Figure 2.34: Execution with forwarding path Output EX → Input EX

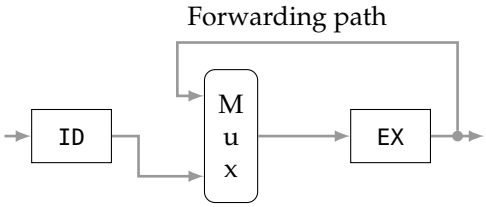


Figure 2.35: Implementation of forwarding path Output EX → Input EX

at the output of the ID stage (output of pipeline register ID/EX) or at the output of the EX stage (output of pipeline register EX/MEM). Figure 2.35 conceptually shows the modifications of the datapath to support this forwarding path.

When the dependent instructions are not consecutive, it is necessary to define another forwarding path. For example, in the following code snippet there is an independent instruction between the two dependent instructions.

```
dadd r2,  r3,  r4
xor  r6,  r11, r10
dsub r1,  r10, r2
or   r12, r11, r14
```

When **dsub** is about to enter the EX stage, the forwarding path from the output of the EX stage to the input of the same stage must be disabled, because the value of register **r2** is not at the output of EX, but at the output of the next stage, the MEM stage. It should be noted that with each clock cycle the results of the EX stage move through the pipeline registers EX/MEM and MEM/WB until finally reaching the WB stage.

Figure 2.36 shows the execution of the previous program when a forwarding path is available from the output of the MEM stage (output of pipeline register MEM/WB) to the input of the EX stage. In this case, the one-cycle stall disappears, since the value of **r2** is available at the beginning of the EX stage.

If two instructions were located between the dependent instructions, the situation would be analogous to the following:

```
dadd r2,  r3,  r4
xor  r6,  r11, r10
ld   r7,  120(r15)
dsub r1,  r10, r2
or   r12, r11, r14
```

In this case, a new forwarding path from the output of the WB stage to the input of the EX stage is not necessary, since **dsub** reads the correct value of register **r2** during

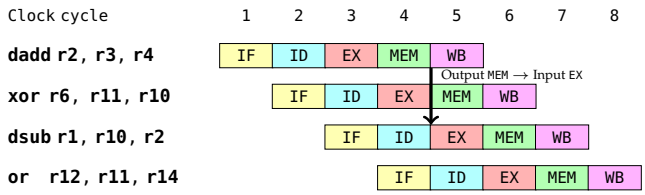


Figure 2.36: Execution with forwarding path Output MEM → Input EX

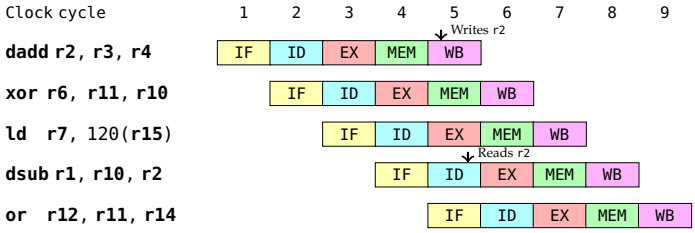


Figure 2.37: Forwarding path Output WB → Input EX is not necessary

the second half of the cycle in the ID stage. This value has been previously written during the first half of the cycle by the WB stage of **dadd**. Figure 2.37 illustrates this situation.

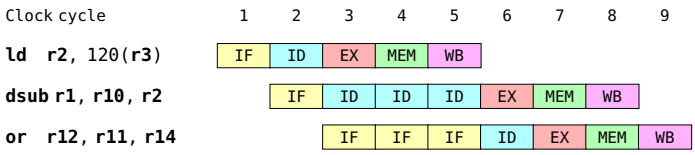
The forwarding paths considered so far assume that instructions writing into registers are arithmetic-logical, such as **dadd r2, r3, r4**. The aforementioned forwarding paths are also valid with other types of instructions that write into registers, with the exception of load instructions. In the case of a load instruction, the value of the destination register is available just at the end of the MEM stage, so the only possible forwarding path is Output MEM → Input EX. For example, this forwarding path is enabled in figure 2.38, which allows a reduction of the stall from two clock cycles to only one for the next code:

```
ld   r2,  120(r3)
dsub r1,  r10, r2
or   r12, r11, r14
```

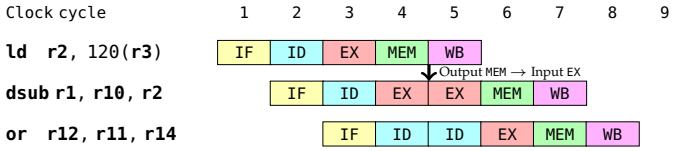
A detail that may go unnoticed at first glance in figure 2.38 is the fact that the pipeline is stalled in the EX stage instead of the ID stage because of the data dependency. The reason is the high complexity of the control unit to decide when the pipeline should be stalled in the ID stage in microarchitectures that go beyond the basic pipelined microarchitecture.

When forwarding paths are available, data dependency stalls occur at the EX stage instead of at the ID stage. This results in a much simpler implementation. The EX stage stalls when it hasn't received any of the values it needs, either from its input pipeline register or from any forwarding path.

An additional consequence of delaying the data stall to the EX stage is the performance improvement when there are multiple execution units. Stalling the pipeline

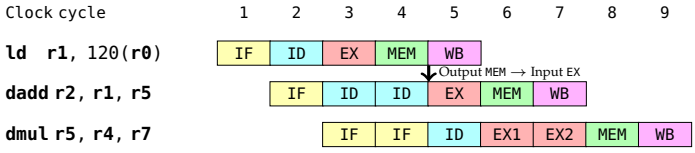


(a) RAW dependency with initial load instruction

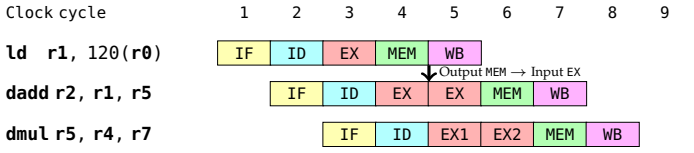


(b) Forwarding path Output MEM → Input EX with initial load instruction

Figure 2.38: Stall reduction for a RAW dependency with initial load instruction



(a) Stall in the ID stage



(b) Stall in the EX stage

Figure 2.39: Performance improvement by stalling in the EX stage

in the EX stage allows the execution of subsequent instructions that use other execution units, as shown in figure 2.39, where instruction **dmul r5, r1, r7** uses a two-cycle pipelined execution unit.

Since the stall does not occur in the ID stage of **dmul r5, r1, r7**, but in the EX1 stage, the next instruction (**dadd r2, r4, r5**), which uses a different execution unit, can enter the ID stage and then execute, without having to wait for the add **r2, r1, r5** instruction to finish the stall.

So far, the case where a single forwarding path is activated during the execution of an instruction has been considered, but in general, an instruction with two source registers may require two forwarding paths to be activated at the same time, or even at different moments. This situation is exemplified in figure 2.40. The EX stage in clock cycle 5 is stalled, since it requires the value of register **r2**, which is not available until the end of the clock cycle. However, despite being stalled, it should fetch

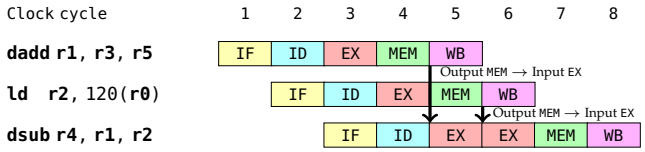
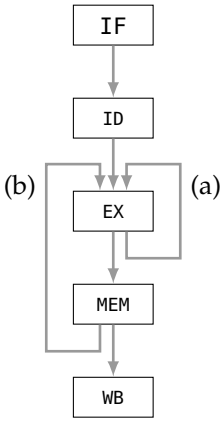


Figure 2.40: Activation of two forwarding paths in the same instruction



(a) Non-load write instruction, followed by a read instruction

(b) Three options:

- Non-load write instruction and a read instruction, both separated by one instruction
- Consecutive load and read instructions**
- Load and read instructions separated by one instruction

** In this case, a one-cycle stall remains

Figure 2.41: Summary of forwarding paths

the value of register `r1` using the forwarding path Output `MEM` → Input `EX` which is activated at the beginning of the cycle.

To summarize, forwarding paths Output `EX` → Input `EX` and Output `MEM` → Input `EX` can reduce or even eliminate the stalls produced by RAW dependencies. Figure 2.41 schematically shows the forwarding paths for both a non-load write instruction and a load instruction.

Register renaming

Unlike instruction scheduling and forwarding paths, register renaming does not try to minimize the effect of data dependencies, but to eliminate them. Specifically, it seeks to eliminate dependencies of type WAR and WAW, which could produce stalls when the CPU executes instructions out of order.

WAR and WAW dependencies are not part of the program semantics. They are created by the compiler when associating storage to program variables because of a very limited number of registers in the architecture.[6] Whenever possible, compilers use registers instead of memory locations, since access is much faster, which results in a shorter execution time. For this reason, when the compiler uses a register for a variable and detects that the variable is no longer used from a certain point in the program, it recycles the associated register, i.e., the register is used to store another

[6] For example, when applying optimizations such as loop unrolling that replicates the body of loops to avoid jumps/branches and therefore control hazards.

```
a = 1;
b = 3 * a;
c = 7;
d = 6;
```

```
(1) ori  r5, r0, 1  ; r5 <- a
(2) ori  r6, r0, 3  ; r6 <- 3
(3) dmul r4, r6, r5 ; r4 <- b
(4) ori  r4, r0, 7  ; r4 <- c
(5) ori  r6, r0, 6  ; r6 <- d
```

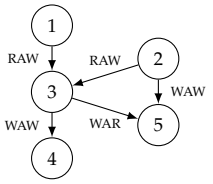


Figure 2.42: Example of dependencies due to register recycling by compilers

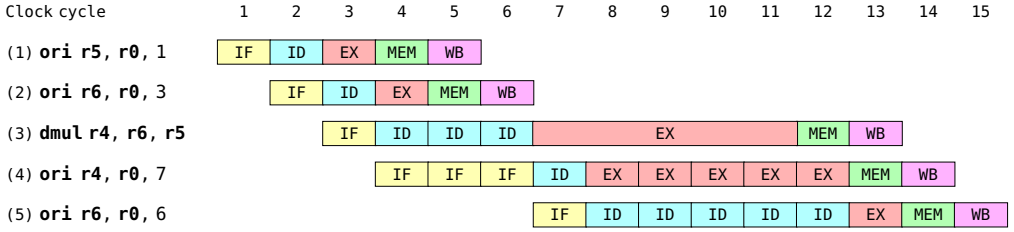


Figure 2.43: Execution with register recycling

program variable, which may lead to the appearance of WAR and WAW dependencies.

For example, figure 2.42 shows a source program fragment and its assembly translation, as well as the resulting dependency graph. In the graph, a dependency is indicated with an arrow, where the pointed instruction depends on the instruction on the arrow start. In addition to true dependencies (RAW), WAR and WAW dependencies can be observed, caused by the recycling of registers `r4` and `r6`. Register `r4` is initially used to store variable `b` and is later recycled to store variable `c`. Register `r6` is initially used to store constant 3 and is later recycled to store variable `d`.

The execution of the previous program in a MIPS64 CPU without precise exceptions is as shown in figure 2.43. It has been assumed that the integer multiplication unit requires 5 clock cycles. A WAW stall can be observed; `ori r4, r0, 7` cannot finish until previous instruction `dmul r4, r6, r5` does.

Suppose that the compiler has not recycled the registers, by storing variable `c` in register `r7` and variable `d` in registry `r8`. The program and the dependency graph would become those shown in figure 2.44, which would result in a lower number of dependencies between instructions. By removing the recycling of registers, the WAW dependency between instructions `dmul r4, r6, r5` and `ori r4, r0, 7` disappears, and also the associated stall, so the execution time is reduced from 14 to 13 clock cycles, as shown in figure 2.45.

Anyway, register recycling cannot be avoided, since CPUs usually have a reduced number of registers. *A priori*, a solution to the problem would be to increase the number of CPU registers to avoid recycling. However, this solution is not viable, because increasing the number of registers means a major change in the architecture that finally leads to software compatibility issues. In addition, even in the case of new architectures that are not conditioned by backward compatibilities, having a large number of registers is inadvisable, since identifying them requires more bits

```
a = 1;
b = 3 * a;
c = 7;
d = 6;
```

```
(1) ori  r5, r0, 1  ; r5 <- a
(2) ori  r6, r0, 3  ; r6 <- 3
(3) dmul r4, r6, r5 ; r4 <- b
(4) ori  r7, r0, 7  ; r7 <- c
(5) ori  r8, r0, 6  ; r8 <- d
```

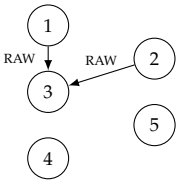


Figure 2.44: WAR and WAW dependencies are removed by removing register recycling

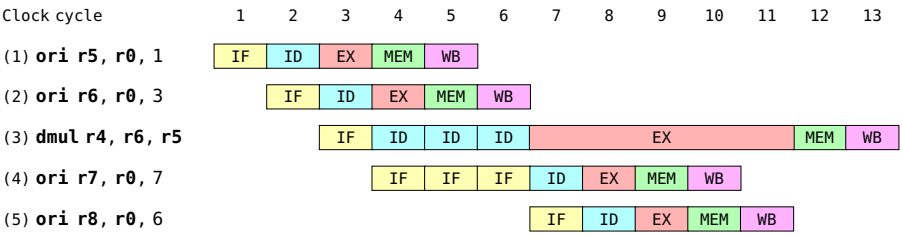


Figure 2.45: Execution without register recycling

and longer instruction codes. For example, in the MIPS64 architecture, moving from 32 registers to 256 registers involves going from 5 to 8 bits to identify registers. If it were necessary to identify 3 registers, as is the case of many MIPS64 instructions, it would finally force to increase the instruction code length from 32 to 64 bits.

The usual solution to the problem of register recycling is register renaming at hardware level. The idea is that architectural registers, from **`r0`** to **`r31`** in the case of MIPS64, are symbolic registers that are associated to physical registers. In this way the architecture does not change, because the registers seen by the programmer are the architectural registers, but the hardware can take advantage of a larger set of physical registers. As an example, it will be assumed that the MIPS64 CPU has 64 physical registers for integers ranging from `rr0` to `rr63` and also 64 floating-point physical registers ranging from `rf0` to `rf63`.

The CPU has a data structure associated to register renaming, called renaming table, which evolves during the execution of instructions. The initial state of the renaming table for a MIPS64 microarchitecture can be seen in table 2.4.

The table relates each architectural register to a physical register. In addition, for each physical register there is a `Counter` field that indicates the number of pending instructions to be terminated that use the physical register as a source operand. Physical registers can be found in three states:

- Allocated and used. In this case, the physical register is used as a source operand in at least one instruction that has not finished. This state is identified by a greater than zero `Counter` field. In addition, the physical register is associated to an architectural register, so it is not available for a future renaming.

Architectural register	Physical register	Counter
r0	rr0	0
.....		
r31	rr31	0
-	rr32	0
.....		
-	rr63	0
f0	rf0	0
.....		
f31	rf31	0
-	rf32	0
.....		
-	rf63	0

Table 2.4: Initial state of the renaming table before executing any instruction

- Allocated and unused. In this case, the physical register is associated to an architectural register, but it is not being used as a source operand in any pending instruction. This state implies a value of zero in the `Counter` field of the physical register. Since it is associated to an architectural register, it is not available for a future renaming. This is the initial state of physical registers `rr0` to `rr31` and `rf0` to `rf31`.
- Available. The physical register is available for future register renaming. It can by identified by a zero `Counter` field and no architectural register associated to it. This is the initial state of physical registers `rr32` to `rr63` and `rf32` to `rf63`.

From the table above, the operation of register renaming is as follows:

1. At the CPU reset, architectural registers **`r0`** to **`r31`** and **`f0`** to **`f31`** are associated to physical registers with the same index, which are allocated and unused, so their `Counter` field is set to `0`. For example, **`r6`** is initially associated to `rr6` and there is no pending instruction reading `rr6`. The rest of the physical registers are available (`rr32` to `rr63` and `rf32` to `rf63`), so their `Counter` field is set to zero and they are not associated to any architectural register. This initial situation is shown in table 2.4.
2. Available physical registers are organized in two FIFO queues. The first queue initially contains registers `rr32` to `rr63`, while the second queue contains registers`rf32` to `rf63`.
3. In the second half of the `ID` stage, source and destination architectural registers are identified during instruction decoding.

Source architectural registers are replaced by the associated physical registers, obtained from the renaming table. In addition, `Counter` fields are incremented to reflect that a new instruction references them. If the same physical register appears several times in the instruction as a source operand, its `Counter` will be incremented only by one unit, since it refers to the number of pending instructions that read the register.

The destination architectural register is renamed by associating it to an available physical register that is extracted from the corresponding FIFO queue. At the same time, its association with the previous physical register must be deleted, since an architectural register must be associated to a single physical register. It is important to keep in mind that register **`r0`** is never renamed, since it cannot act as a destination operand in instructions. Register **`r0`** always contains a zero value.

4. When an instruction is retired, specifically in the first half of its `WB` stage, the `Counter` field of its source physical registers is decremented. If the physical register appears more than once, it is only decremented by one unit. If the `Counter` field of any source physical register gets a zero value and does not have an associated architectural register, it will be available to be used in a future renaming and it is added to the corresponding FIFO queue.

Figure 2.46 provides an example of register renaming using the same program as in figure 2.42. Although in such a simple example it would be easy to avoid register recycling at compile time (as shown in figure 2.44), in general it is not always possible, since the number of available registers is very limited and some data dependencies are caused by branches that cannot be resolved in compilation time.

Expressions as `ri→rrj:k` are used to indicate changes in the renaming table. This indicates that architectural register `ri` is associated to physical register `rrj` and its counter is set to `k`. When a physical register is available for future renamings, the architectural register does not appear in the expression and its counter is zero, i.e., `rrj:0`.

To analyse the renaming of registers in the example, it is necessary to focus on the operations performed by stages `ID` and `WB` for every clock cycle, starting from the initial state of table 2.4.

- Clock cycle #2. `ID` stage of **`ori r5`**, **`r0`**, `1`. The architectural destination register, **`r5`**, must be renamed, so it will no longer be associated to physical register `rr5`, but to the first physical register available in the FIFO queue. Thus, the renaming table is updated with `r5→rr32:0`. From now on, any reference to source register **`r5`** becomes a reference to the associated physical register `rr32`.

- Clock cycle #3. Something similar happens to register **`r6`** at the `ID` stage of **`ori r6`**, **`r0`**, `3`, so the renaming table is updated with `r6→rr33:0`.

- Clock cycle #4. The destination register should be renamed and the counters of the physical registers associated to the source architectural registers incremented at the `ID` stage of **`dmul r4`**, **`r6`**, **`r5`**,. However, since a RAW dependency is identified on registry **`r6`** that leads to a pipeline stall, the modifications in

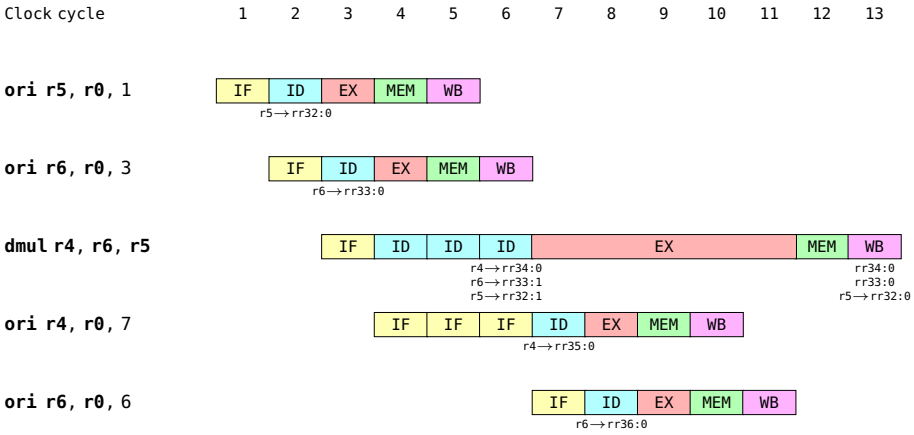


Figure 2.46: Program execution with register renaming by hardware

the renaming table are delayed to clock cycle #6, once `rr33` physical register is written by **`ori r6, r0, 3`**.

- Clock cycle #5. Instruction **`ori r5, r0, 1`** is retired, so the counters of its physical source registers should be decremented by one unit. Since there are no physical registers in use in this case (register **`r0`** is not considered), the renaming table is unmodified.

- Clock cycle #6. Instruction **`ori r6, r0, 3`** is retired without affecting the renaming table.

  The stall due to the RAW dependency also finishes in this cycle, so the renaming table is modified at the `ID` stage of **`dmul r4, r6, r5`**. Firstly, destination register is renamed, so **`r4`** becomes associated to physical register `rr34` (`r4→rr34:0`). Secondly, the counters of the physical registers associated to architectural registers **`r6`** and **`r5`** are incremented, i.e., `rr33` and `rr32`, respectively (`r6→rr33:1` and `r5→rr32:1`).

- Clock cycle #7. Register **`r4`** is renamed with `r4→rr35:0` at the `ID` stage of **`ori r4, r0, 7`**.

- Clock cycle #8. Register **`r6`** is renamed with `r6→rr36:0` at the `ID` stage of **`ori r6, r0, 6`**.

- Clock cycle #13. Instruction **`dmul r4, r6, r5`** is retired, so the counter field of its source physical registers is decremented by one unit, i.e., registers `rr33` and `rr32`. Physical registers `rr34` and `rr33` are no longer associated to any architectural register and now have a zero counter (there are no pending instructions reading them), so they become available and are added to the FIFO queue. The counter of physical register `rr32` also becomes zero, but unlike the previous physical registers, it is associated to an architectural register, register **`r5`**, so it is unavailable.

In short, the code actually executed on the microarchitecture is shown below:

```
ori  rr32, rr0,  1
ori  rr33, rr0,  3
dmul rr34, rr33, rr32
ori  rr35, rr0,  7
ori  rr36, rr0,  6
```

It can be verified that there are only true data dependencies (RAW) in this code, since WAR and WAW dependencies have been eliminated thanks to the renaming of registers. Thus, it was possible to reduce the execution time from 14 clock cycles (figure 2.43) to 13 clock cycles (figure 2.46). One single cycle may seem a poor improvement, but it is not so small taking into account the low number of instructions in the example.

Although the number of physical registers is usually high, all WAR and WAW dependencies may not be removed in some complex programs due to the high number of physical registers that would be required.

2.3.6 Reducing control hazard stalls

The execution of branches requires to calculate the target address and evaluate the condition. The new value of the program counter is set in the `MEM` stage. In contrast, in the case of jumps, the new value of `PC` is set in the `ID` stage.

When a branch instruction is decoded in the `ID` stage, it prevents the subsequent instructions from entering the pipeline until it leaves the `MEM` stage. However, the instruction following the branch cannot be prevented from entering stage `IF`, because at that moment the branch instruction has not been identified yet, so that the result of this stage is discarded. In total, there is a stall of three clock cycles. This stall is shown in the example of figure 2.23.

Next, some techniques for reducing the stalls due to control hazards are presented.

Early evaluation of branches

The *early evaluation of branches* or *early branches* is a technique to reduce the penalty of branches, advancing branch evaluation (computing the target address and evaluating the condition) in the pipeline, for example to stage `ID`, which results in a stall of one cycle, the same as for jumps.

Advancing the evaluation of branches requires some changes in the datapath. The logic to calculate the target address of the branch is moved from the `EX` stage to the `ID` stage. In addition, a comparator must be added to verify whether the branch condition is met. With these changes, the affected part of the datapath required to support the **`beq`** branch instruction would be that shown in figure 2.47. Figure 2.48 shows an example of execution with early evaluation of branches.

The implementation of figure 2.47 poses some challenges. Registers are read during the second half of the clock cycle in the `ID` stage (and written during the first half in `WB`). This means that the `ID` stage must carry out additional operations during the

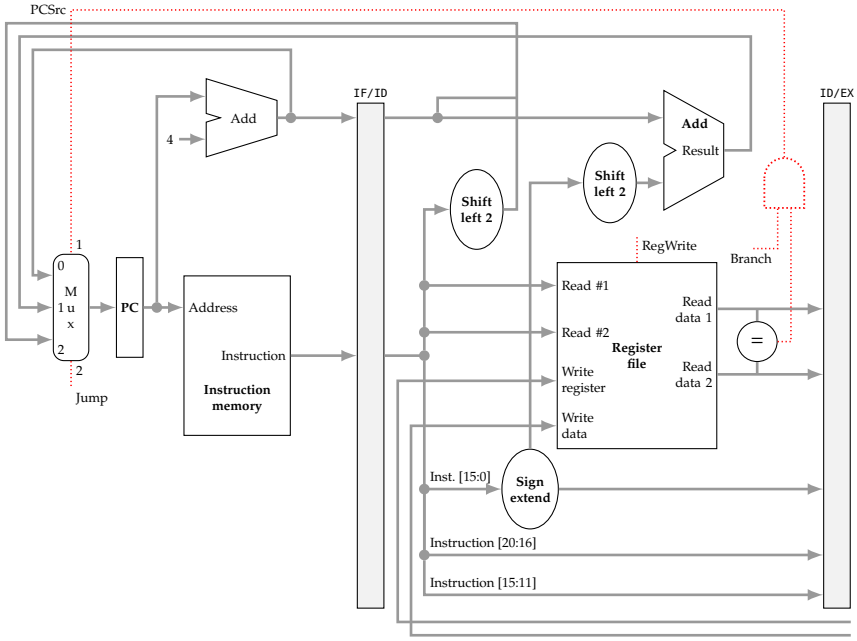


Figure 2.47: `IF` and `ID` stages for early evaluation of **`beq`** branches

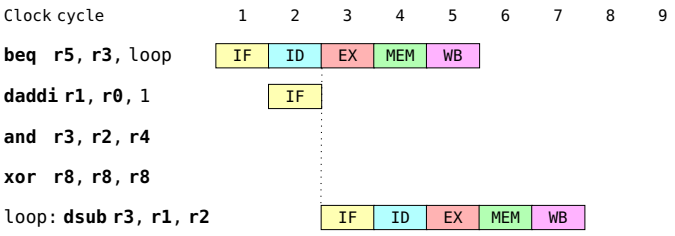


Figure 2.48: A control hazard stall with early branches assuming the branch is taken

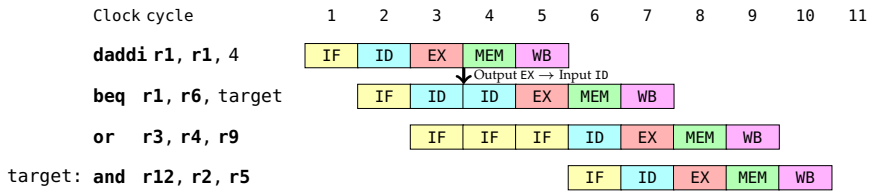


Figure 2.49: Early branches require a new forwarding path: Output `EX` → Input `ID`. In this example, branch is not taken.

second half of the clock cycle. It must compare the value of the registers, so this may enlarge the clock cycle to accommodate this new logic. Thus, early branches are difficult to implement efficiently, as they may imply a longer clock cycle. The key is to move the logic to compute branches to early stages in the pipeline without affecting the clock cycle. It must be noted that the pipeline clock cycle depends on the longest stage.

Another problem derived from the early evaluation of branches is that a new forwarding path is necessary. The branch instruction needs the values of source registers in the `ID` stage, and these can be generated in stages `EX` and `MEM` as a result of arithmetic-logic operations or memory loads, respectively. For the second case, the register would be written in the `WB` stage during the first part of the next cycle and could be read in the second part of that cycle. However, a new forwarding path is required for the first case: Output `EX` → Input `ID`. Figure 2.49 shows an example that uses this new forwarding path. Note that it is still necessary to stall the pipeline for one clock cycle.

Early branches are applicable to simple microarchitectures with few stages, as is the case of the MIPS64 microarchitecture presented. However, they are not effective in the case of microarchitectures with a greater number of stages.

Branch prediction

Branch prediction is the most commonly used technique to reduce stalls due to control hazards. It consists in predicting whether the branch is taken or not in such a way that at the end of the `IF` stage the program counter contains the memory address of the instruction that should be executed next.

It is a speculative execution technique that tries to predict the result of a branch. Based on this prediction, the CPU executes instructions that should be executed after the branch is evaluated otherwise. Performance improves in the case of a correct prediction, as it has not been necessary to stall the pipeline waiting for the evaluation of the condition and the computation of the target address. On the contrary, in the case of an incorrect prediction, instructions have been executed improperly, so their effects must be reverted.

To illustrate branch prediction, the following code snippet will be used.

```
  beq  r4,  r5,  target
  dadd r9,  r0,  r7
  dsub r11, r10, r5
  or   r12, r5,  r14
target:
  dadd r7,  r1,  r3
```

The simplest prediction technique is called *always not taken* and it assumes that branch conditions are never met. When a branch instruction is decoded, the control hardware of the CPU always assumes that the branch condition is not fulfilled and continues executing the instructions that follow the branch without stalling the pipeline, which also simplifies branch handling. Once the branch reaches the `MEM` stage (or `ID` if early branches is used) and the branch condition is evaluated, there are two possibilities:[7]

- Branch is not taken. In this case the prediction was correct (hit), so is the execution of instructions following the branch. Pipeline has not stalled, so the improvement is evident (three clock cycles). This situation is shown in figure 2.50a.
- Branch is taken. In this case the prediction was incorrect (miss). Instructions `dadd r9, r0, r7`, `dsub r11, r10, r5` and `or r12, r5, r14` must stop progressing in the pipeline, so they are discarded by flushing the pipeline. The stall is three cycles as when there was no branch prediction. This situation is shown in figure 2.50b.

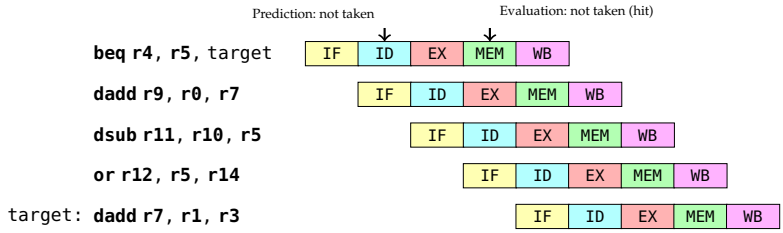
For example, if the branch conditions are met in half the branches of a program, the average stall reduces from three clock cycles to just half, with the consequent improvement of CPI.

An interesting feature of the studied MIPS64 microarchitecture is that stages `IF`, `ID` and `EX` do not modify the status of registers or memory, so in the case of a prediction miss, writings do not need to be reverted. Memory and register writings are carried out in stages `MEM` and `WB`, respectively. This favourable feature is not accidental, but it comes from of a very careful design of the microarchitecture. In deep pipelines, instructions that are executed speculatively are labelled so that they cannot make definitive writings in registers or memory. Only when the branch is resolved the writings become definitive if the prediction was correct, or discarded if it was incorrect.

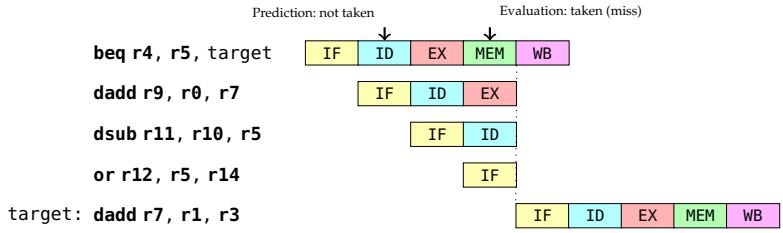
The always-not-taken branch prediction technique is very basic and is currently of little use. Some experimental studies have shown that, on average, two out of three branches are taken and one is not, so the efficiency of the algorithm would be about 33%. Of course, these figures are only indicative.

Modern pipelined CPUs implement many more stages than the MIPS64 pipelined microarchitecture presented with the aim of improving performance. As the number of stages increases, the penalties of prediction misses are greater. They involve longer stalls due to the instructions that must be discarded and higher energy consumption due to the execution of these unnecessary instructions. Thereby,

[7] It is necessary to remember that the evaluation of the condition is carried out in the `EX` stage, but the `PC` register is modified in the `MEM` stage, so branch outcome is obtained in the `MEM` stage in practice.



(a) Prediction hit



(b) Prediction miss

Figure 2.50: Behaviour with always-not-taken prediction

sophisticated branch prediction techniques are used going far beyond than assuming that branches are never taken.

Branch prediction is an essential topic in the design of CPUs because of its influence on performance. Enormous research effort has been made on this subject. Branch prediction algorithms of modern CPUs are very complex and outside the scope of this text. However, it is interesting to describe a branch prediction algorithm much more effective than the simple always not taken.

Firstly, it is convenient to meditate about the nature of the always-not-taken prediction algorithm. It belongs to the family of algorithms called static, because once they arrive at a branch instruction their behaviour is always the same for that instruction, regardless the outcome of the branch. The biggest drawback of static prediction algorithms is that they do not take into account the history of the branch instruction, that is, whether the branch was taken or not in previous executions. For example, consider this loop of 100 repetitions:

```
  .....
  ori   r1, r0, 100 ; r1 = 100
startf:
  daddi r1, r1, -1  ; r1 = r1 - 1
  bnez  r1, startf  ; jump to startf if r1 is not zero
endf:
  .....
```

In the first 99 executions of **bnez** the condition is met, but not in execution 100 when the loop terminates. If the history of the branch were taken into account, the CPU could miss predicting the first branch or even the second, but later, once the history of the branch is analyzed, it is clear that the best prediction is to assume that

the branch is taken. The opposite situation could occur and the analysis of the branch could lead to assume that the branch is not taken in loops with different structure. This is the key idea that dynamic predictors incorporate: they take into account the history of the branch to predict whether the branch will be taken or not in the current execution.

One of these dynamic predictors is the two-bit predictor, which uses a table, called BHT (Branch History Table) or BPB (Branch Prediction Buffer), to store the history of branches. It is also necessary to have a second table, called BTB (Branch Target Buffer), where target addresses are stored for both branches and jumps.

For the sake of simplicity, from now on it is assumed that the information contained in the BTB is included in the BHT, so that only a single table, the BHT, is necessary for the operation of the predictor. Each time a branch/jump instruction is executed for the first time, an entry in the BHT is created that contains these fields:

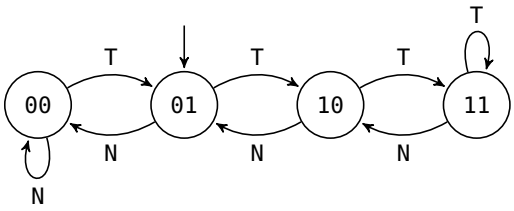
- Memory address of the branch/jump instruction (or its least significant bits). It matches the value of `PC` at the beginning of the `IF` stage for the branch/jump instruction. This field is used in the `IF` stage to obtain the entry in the BHT while fecthing the instruction code from the instruction memory at the same time. Therefore, the reading of the BHT does not imply any performance penalty, since it is done in parallel with the reading of the instruction memory.
- Target address of the branch/jump. In the studied MIPS64 microarchitecture, the target address of a branch is calculated in the `EX` stage (when not using early branches), but it does not become effective until the `MEM` stage, when the `PC` register is updated. It is also in the `MEM` stage where the target address is written to the BHT the first time the branch instruction is executed. In the case of jumps, or branches with early evaluation, the writing is done in the `ID` stage.
- Prediction bits. They are used to predict whether a branch is taken or not based on its past behaviour.

The two-bit branch predictor uses two bits in each BHT entry. These two bits can be considered a counter that increments with each branch taken and decrements with each branch not taken, saturating to `00` or `11`. Based on the value of this counter, these four states are possible:

- `00` (*strong not taken*).
- `01` (*weak not taken*).
- `10` (*weak taken*).
- `11` (*strong taken*).

The most significant bit determines the prediction. If it is `0` (states strong not taken and weak not taken) the prediction is that the branch is not taken. If the bit is `1` (states strong taken and weak taken) the prediction is that the branch is taken.

The leftmost part of figure 2.51 shows the state diagram of the predictor, where the initial state is `01` (weak not taken). The right part of the figure shows the BHT.



BHT

Memory address	Target address	History
1010101...	0011101...	01
0111010...	1100010...	10
1010101...	0111011...	11

Figure 2.51: State diagram and branch history table for a two-bit predictor

Several changes in the microarchitecture are necessary to implement the two-bit branch predictor. Every clock cycle, the `IF` stage checks if there is an entry in the BHT with the current `PC` address. If there is an entry, it is said that a BHT hit occurs, and a BHT miss otherwise.

If a BHT miss occurs in a jump instruction the pipeline stalls for one cycle, since the target address of the jump is unknown until the end of stage `ID`. At that time, the BHT is updated to store the target address of the jump.

If the BHT miss occurs with a branch instruction, it is assumed that the branch is not taken, so the pipeline will be flushed if finally the branch must be taken. If the prediction is correct there is no stall, but if it is incorrect there will be a stall of three clock cycles, or only one if early branch evaluation is used. In any of the above cases, the BHT is updated, either in the `MEM` stage or the `ID` stage.

If a BHT hit occurs with a jump instruction, the `PC` is written with the target address in the BHT entry. Since both the BHT check and the `PC` writing are performed in the `IF` stage, no stall is required.

If a BHT hit occurs with a branch instruction, the most significant bit in the prediction bits determines whether the branch should be taken. If it is predicted that the branch is taken, the `PC` is written with the target address stored in the BHT entry. If it is predicted that the branch is not taken, the value `PC+4` is written in the `PC`. All this process is done in the `IF` stage. Subsequently, it is checked the prediction correctness in the `MEM` stage or the `ID` stage when using early branches. If the prediction was correct, execution continues without stalling. On the contrary, if it was incorrect, it is necessary to flush all the instructions that erroneously entered the pipeline, leading to a stall of three clock cycles, or only one when using early branches. In any of the above cases the BHT is updated, either in the `MEM` stage or the `ID` stage.

The operation of the two-bit predictor is illustrated with the following example (see figure 2.52). Early branch evaluation in the `ID` stage and all the forwarding paths are assumed.

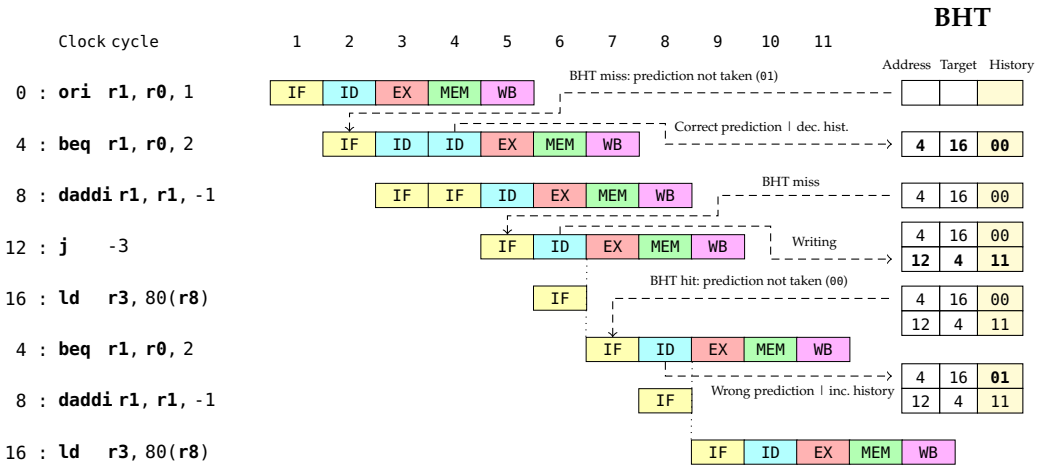


Figure 2.52: Operation of the two-bit branch predictor

```
  ori   r1, r0, 1 ; r1 = 1
startf:
  beq   r1, r0, endf
  daddi r1, r1, -1
  j     startf
endf:
  ld    r3, 80(r8)
```

For the sake of simplicity, figure 2.52 depicts only the BHT queries of jump/branch instructions, although BHT queries would be performed in all instructions,[8] since instructions have not been decoded yet in the `IF` stage.

- The BHT table is empty the first time **`beq`** is executed, so a BHT miss occurs. Therefore, the branch is predicted as not taken and `PC` is loaded with the address of the following instruction (from address 8), which enters the pipeline in cycle #3. In cycle #4, once the RAW dependency on **`r1`** is resolved, the prediction is verified to be correct, i.e., the branch is not taken. The entry corresponding to the branch is written into the BHT table. The history for this branch will be set to `00` (strong not taken) as the branch is not taken.

- A BHT miss also occurs in the first execution of `j`, so the jump must be processed as usual in the `ID` stage. Once the jump target address has been calculated, `PC` and the target address of the jump instruction in the BHT are updated. In addition, the history for this jump is set to `11` (strong taken) in the BHT entry (the prediction is always to jump).[9]

- The branch instruction is reached again in the second iteration, but this time a BHT hit occurs. Since the stored history is strong not taken, it is predicted that

[8] Instructions that are not branches or jumps will always generate a BHT miss.

[9] It does not make any sense predicting jumps, as they are always taken. This is a consequence of combining BHT and BTB in a single table. Jumps only use BTB.

the branch is not taken. In the next clock cycle the prediction is checked to be incorrect, so the instruction that entered incorrectly the `IF` stage (`daddi r1, r1, -1`) is flushed and the value of the branch target in the BHT is copied to `PC`. This results in one-cycle stall. Finally, the history counter in the BHT table is increased.

Experimental studies using typical programs have shown that a two-bit predictor like this correctly predicts about 90% of branches. However, current pipelined CPUs require even higher percentages than those achieved with this predictor in order to reduce the CPI penalty. The behaviour of many branches also depends on previously executed branches, so the most modern predictors take into account not only the branch instruction to be predicted but also other branch instructions.

2.3.7 Pipeline depth

The instruction pipeline is a technique that exploits the parallelism that exists between instructions, which allows to execute several instructions at the same time, as many as stages in the pipeline. For this reason, the independence of the instructions in a program is key to achieve a high degree of parallelism and good performance in pipelined CPUs.

A possible way to improve performance is increasing the pipeline depth, i.e., increasing the number of stages in the pipeline. Thus, the clock period is reduced and therefore the CPU time. For example, if the number of stages is multiplied by two, the duration of the stages will be half the original time ideally, doubling the frequency and halving the cycle time, assuming as usual that all stages have the same duration. According to the iron law of performance, CPU time would be reduced by half. In theory, the higher the number of stages, the lower the CPU time and the higher the CPU performance. However, as the number of stages increases, negative effects appear that diminish the improvement in performance:

- There is an increase in the execution time of an instruction caused by a more complex CPU hardware. Pipeline registers are arranged between the pipeline stages. These pipeline registers introduce a delay due to the time required to be written. In addition, by increasing the number of stages there are more stages that require simultaneous access to the register file and memory, which implies the use of multiport register files and memories. The access delays and the hardware cost of multiport registers and memories increase rapidly with the number of ports.
- The penalty coming from control stalls caused by branches and jumps increases rapidly as the number of stages increases. In the pipelined MIPS64 microarchitecture, branches are evaluated in the `MEM` stage (without early branches), so the cost of a control stall is three clock cycles in this case: the distance between stages `IF` and `MEM` in the pipeline. The larger the number of stages of the pipeline, the further this distance and the penalty in clock cycles.
- As the number of stages increases, the clock frequency also increases, since all the stages require a shorter clock cycle. A small increase in the clock frequency

brings a large increase in the power dissipated and, at present, there are limits on the power that a CPU can dissipate, which is known as the *power wall*. Section 2.5 provides more details about these limits.

Considering all these limitations, modern general purpose CPUs implement pipelines between 10 and 20 stages.

2.4 Multiple instruction issue

As shown in the previous section, a pipelined CPU achieves a CPI equal to 1 under ideal conditions, or in other words, achieves a throughput of one instruction per clock cycle. To improve the performance of the CPU it is necessary to reduce the product of the three factors that provide the CPU time in the iron law of performance:

$$\mathrm{T_{CPU}} = \text{Program instructions} \times \text{CPI} \times T$$

For a given ISA, the number of program instructions depends on the compiler and can be assumed constant for the same program. Therefore, the options for improvement are to reduce the clock period or to devise microarchitectures able to achieve CPI values lower than one, that is, execute more than one instruction per clock cycle. In the previous section it was indicated that there are practical limits on the clock period derived from the power wall. Taking these constraints into account, the remaining option to improve performance consists in executing more than one instruction per clock cycle.

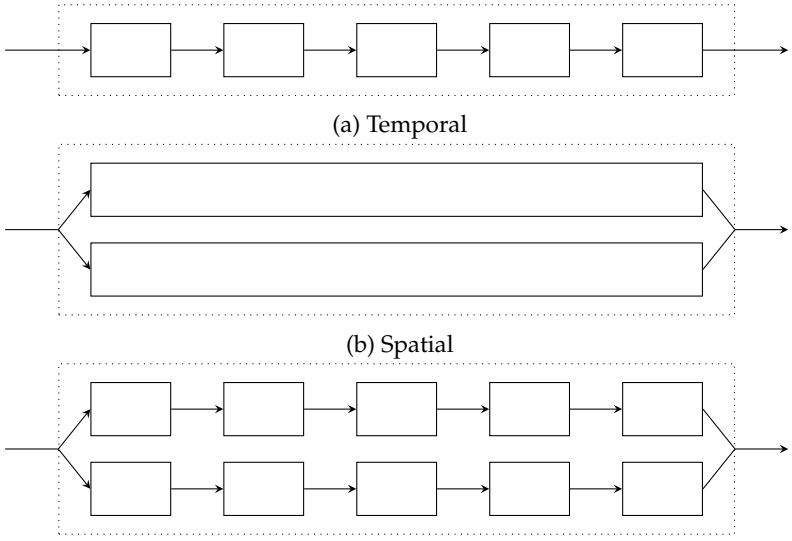
The number of instructions per clock cycle is called Instructions Per Cycle, or IPC, and is directly related to CPI, since IPC $= 1/\text{CPI}$. Therefore, the interest will focus on designing microarchitectures to achieve IPC values greater than one. For this, it is necessary to exploit instruction parallelism even more.

One way to exploit instruction parallelism is the pipeline, where each stage of the datapath is specialized in a part of the instruction and allows executing several instructions simultaneously. The pipeline uses a type of parallelism called temporal parallelism, schematized in figure 2.53a.

Another form of parallelism, called spatial parallelism, is possible. It consists in replicating the CPU datapath to allow the execution of several complete instructions at the same time. Figure 2.53b shows this other type of parallelism.

Finally, the temporal and spatial parallelism can be combined by replicating the pipelined datapath, giving rise to several execution pipelines, as shown in figure 2.53c. This technique is known as multiple instruction issue and allows the CPU to execute more than one instruction per clock cycle. The number of instructions that can be processed in the same stage at the same time is called the issue width. For example, for the datapath of figure 2.53c, the issue width is 2, which would allow reaching an IPC of 2 (or a CPI of 0.5) in ideal conditions, without needing to double the clock frequency.

Although figure 2.53c conceptually represents a multiple instruction issue CPU, it does not correspond to the actual organization of this type of CPUs. Before showing a more realistic organization, it is convenient to establish the objectives and restrictions of multiple instruction issue CPUs:



(a) Temporal

(b) Spatial

(c) Temporal and spatial

Figure 2.53: Types of parallelism in the execution of instructions

1. Instruction scheduling. Having several pipelines, even with different execution units, it is necessary to correctly schedule the execution of instructions. Not only is it necessary to decide which pipeline processes each instruction, but also to solve cross-data dependencies that may appear between the instructions of different pipelines. Instruction scheduling may rely upon the compiler, the CPU or both.

2. Each pipeline stage must be able to process at least as many instructions as the width of the multiple-issue CPU.

There are basically two alternatives to implement multiple instruction issue, which differ in how the responsibility of dealing with the above problems is shared between the compiler and the CPU: multiple issue with static scheduling and multiple issue with dynamic scheduling.

In the multiple issue with static scheduling, the compiler packages several operations in a very long instruction code, each one associated to one of the CPU execution units. For example, if the CPU has 4 integer units, the instruction could include four independent register addition operations. The compiler is responsible for carrying out the packaging of operations to take advantage of program parallelism while preserving instruction dependencies. Microarchitectures following this approach are known as **VLIW** (Very Long Instruction Word). In this type of microarchitectures the compiler must fit to the microarchitecture, so any change, even keeping the same instruction set architecture, requires an important effort adapting the compiler.

In multiple-issue CPUs with dynamic scheduling, the hardware is responsible for decision making in terms of instruction scheduling and hazard resolutions. However, to a lesser extent, the compiler collaboration can help to improve performance.

This microarchitecture is called **superscalar**,[10] as opposed to a scalar microarchitecture, which can execute at most one instruction per clock cycle. The main difference with VLIW microarchitectures is that superscalar CPUs guarantee the correct execution of instructions without compiler assistance. In a VLIW CPU, the correct execution of instructions is not guaranteed without the help of the compiler.

Currently, most modern CPUs are superscalar, so the rest of this section focuses on them.

Instruction level parallelism is a key program parameter to take advantage of multiple-issue microarchitectures and, in particular, of superscalar microarchitectures. For this reason it is discussed in more detail below.

2.4.1 Instruction level parallelism

The independence of instructions in a program is of great importance to obtain a good performance with pipelined CPUs, but even more in the case of multiple-issue CPUs. Control hazards can be reduced using branch prediction, structural hazards by replicating units or implementing multiport devices, but data hazards are part of the program semantics. The pipeline continously stalls when there are many nearby instructions with data dependencies. This penalizes performance.

Instruction level parallelism or **ILP** is a term used to refer to the level of independence between instructions in a program. Figure 2.54 shows two code snippets with two opposite situations in terms of instruction level parallelism. In the leftmost snippet, all instructions except the first one could theoretically suffer data stalls (although some are avoidable using register renaming). The bottom graph represents the instruction dependencies. In this situation, the performance of a multiple-issue CPU would be very poor, that is, the IPC value would be much lower than the CPU issue width.

The rightmost code snipped in figure 2.54 has no data dependencies, so it takes full advantage of multiple-issue CPUs, making their performance close to the ideal. The corresponding graph appears just below. It can be observed that all the nodes are disconnected, as there are no dependencies.

Dependency graphs provide visual information not only about instruction dependencies, but also about the minimum execution time of a program. The longest path within the graph approximates the minimum execution time. Although the hardware available were unlimited, that is, a CPU with unlimited issue width, the execution time could not be reduced below that associated with the longest path, which is called the critical path.

In summary, the program ILP, or analogously, the dependencies between instructions, imposes theoretical limits regarding the minimum execution time of a program, even with unlimited hardware. After all, if some instructions have to wait for others, it is not possible to execute all of them in parallel.

[10] There are also superscalar CPUs with static scheduling, but they will not be considered, since their operation is similar to VLIW CPUs. In this text the term superscalar CPU will be used to refer to a superscalar CPU with dynamic instruction scheduling.

```
(1) dadd r1, r3, r4      (1) dadd r3, r2, r1
(2) xor r2, r5, r1       (2) xor r6, r4, r1
(3) dsub r2, r4, r7      (3) dsub r8, r4, r7
(4) movz r7, r4, r6      (4) movz r9, r4, r5
(5) slt r9, r7, r6       (5) slt r10, r7, r5
```

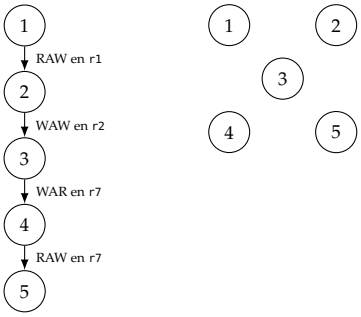


Figure 2.54: Examples of instruction level parallelism

2.4.2 Superscalar microarchitecture

Figure 2.55 shows a two-issue superscalar microarchitecture for a MIPS64 CPU at block level. The organization of a CPU with a greater width would be similar. Instruction execution is divided into five stages:

- Instruction fetch (`IF`).
- Instruction decoding (`ID`).
- Dispatch (`DT`).
- Execution (`EX`).
- Retirement (`RT`).

The stages do not exactly match those of the studied pipelined microarchitecture, but most are conceptually similar. Stages `IF`, `ID` and `EX` are analogous to those of the pipelined microarchitecture. The retirement stage, `RT`, performs a function analogous to that of stages `MEM` and `WB` as a whole. The dispatch stage, `DT`, does not match any of the stages in the pipelined CPU. All the stages can process two instructions at the same time, except the `EX` stage that has six execution units: two for integers and logical operations, one for load/store memory operations, one for branch evaluation, one for floating-point operations and another one for integer multiplication/division. These execution units require different number of cycles. In the case of requiring more than one cycle, the units are pipelined to improve performance. Depending on the combination of instructions, up to six instructions can be simultaneously in the `EX` stage.

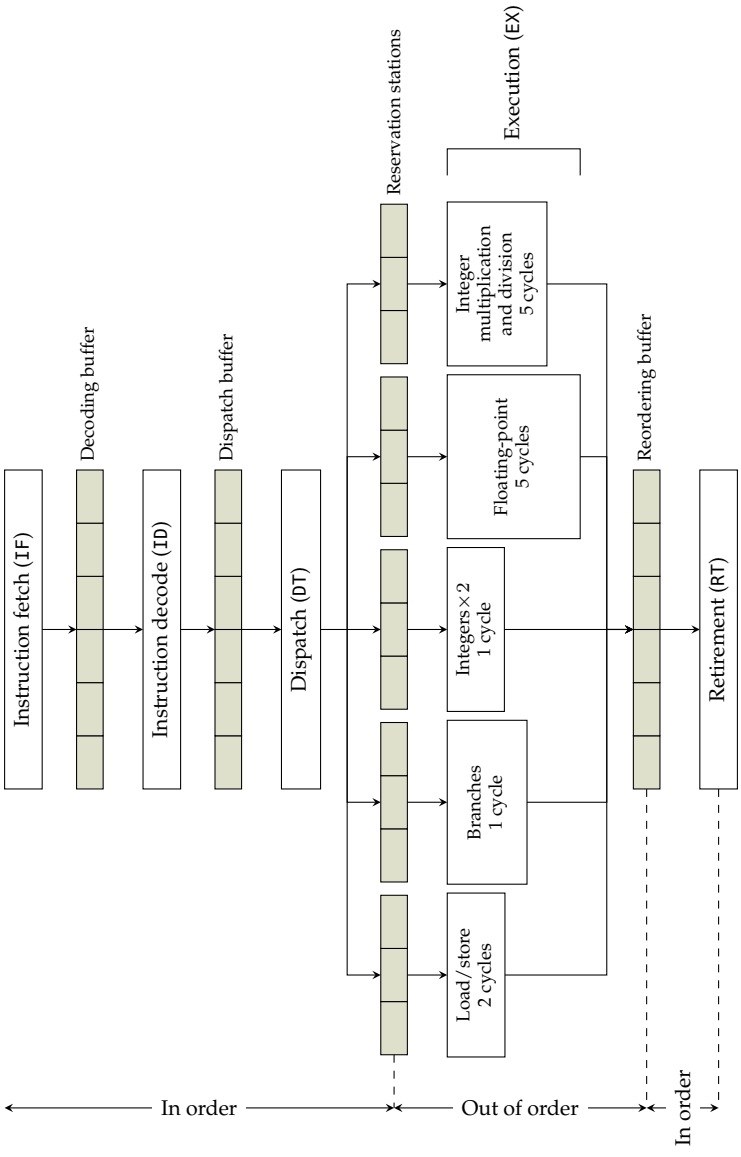


Figure 2.55: Organization of a superscalar CPU

Multiport buffers are placed between all the stages that allow to carry out several reads and writes in parallel. These are registers with more capacity and complexity than pipeline registers.

It is important to note that superscalar CPUs are divided into three sections, each comprising one or several stages:

- In-order initial section. Instructions are read from memory, decoded and dispatched following the program order.
- Out-of-order issue and execution. Instructions are issued and executed out-of-order in the corresponding execution unit to take full advantage of instruction level parallelism.
- In-order final section. Instructions are retired following the order indicated by the program, so memory and register writes occur in the proper order, also providing support to precise exceptions.

Next, the stages of the superscalar CPU will be described in detail.

- `IF` fetches from the program memory a number of instructions equal to the issue width of the superscalar CPU, in this case 2. Two consecutive instructions are fetched at the same time, so a double-word is read (64 bits), starting from the address given by `PC`.

  Instruction fetches are aligned to 8 bytes (64 bits). This has some consequences when changing the sequential flow of the program, since jump and branch target addresses are aligned to 4 bytes.

  Branch prediction is also performed in this stage as explained in section 2.3.6.

- `ID` is one of the most complex stages. It decodes two instructions (as many as the width of the CPU) at the same time, reads the available source registers of both instructions (up to 4 simultaneously), tags the missing registers (because they are affected by RAW dependencies) and renames the destination registers (to remove WAR and WAW data dependencies).

- `DT` extracts up to two instructions from the dispatch buffer and distributes them to buffers called reservation stations, associated to execution units. Typically, there is a reservation station associated with each type of execution unit. Each of the reservation stations can store several instructions that await processing by an associated execution unit. If the required reservation station is full when the instruction needs to be distributed, a structural stall occurs.

  Reservation stations are multiport buffers that deserve special attention. Each reservation station receives instructions of a certain type. Entries in reservation stations store the instruction together with the value of the available source registers and tags for the registers not yet available. For example, in figure 2.55 there are reservation stations for load/store instructions, branches, integer arithmetic-logical operations, floating-point instructions and integer multiplication/division. The reservation stations define the boundary between the first pipeline part, in order, and the out-of-order issue.

An instruction in a reservation station waits until all its source operands are ready. While an instruction is waiting for its operands, the forwarding bus is continuously checked, where the instructions publish the value of destination registers when they leave the execution stage. Once an entry in a reservation station detects a match between a published register and one tagged in the waiting instruction, the value of this register is stored in the entry of the reservation station. When all the instruction operands are available, the instruction is issued to be executed as soon as an execution unit is available.

- EX. Superscalar CPUs have several specialized execution units. Instructions are executed out of order in these units. The order is determined by the instruction issue and the availability of an execution unit. Once instructions are executed, they are stored in the reordering buffer and the value of the register it modifies is published on the forwarding bus. In addition, if an exception has occurred in the execution of the instruction, the instruction is tagged within the reordering buffer.

  Once a instruction leaves the EX stage it has not still finished, but the publication of the modified register allows instructions with a RAW dependency on the register to be issued. As can be seen, it is a generalization of the forwarding path concept.

  In the case of load or store instructions, the memory address is calculated and the memory access operation is stored in the reordering buffer along with other instruction information, such as source or destination registers.

- RT extracts instructions from the reordering buffer in the program order. In this way, in addition to support precise exceptions, the compliance of RAW, WAW and WAR dependencies between memory operands is guaranteed.[11]

  An instruction must be tagged as non-speculative to be retired. When a branch prediction is made, the instructions following the branch are tagged as speculative. Later, when the branch condition is evaluated, the executed instructions change from speculative to non-speculative and may be retired if the prediction was correct. Otherwise, speculative instructions are discarded.

  In the particular case of store instructions, writes are not carried directly to data memory, but are carried to a storage buffer within the CPU in the program order. Although the operation of this buffer is outside the objectives of this chapter, it is worth mentioning that its objective is to accelerate memory readings and writings.

In order to understand the implications in the execution of instructions in a superscalar microarchitecture, figure 2.56 shows the execution of 8 instructions of a program in the proposed superscalar MIPS64 CPU:

- The first two instructions are fetched from memory at the same time in the IF stage. The CPU reads 64 bits that correspond to the codes of these two instructions.

[11] Compliance of memory dependencies is no longer trivial with out-of-order execution as occurred with the studied MIPS64 pipelined microarchitecture.

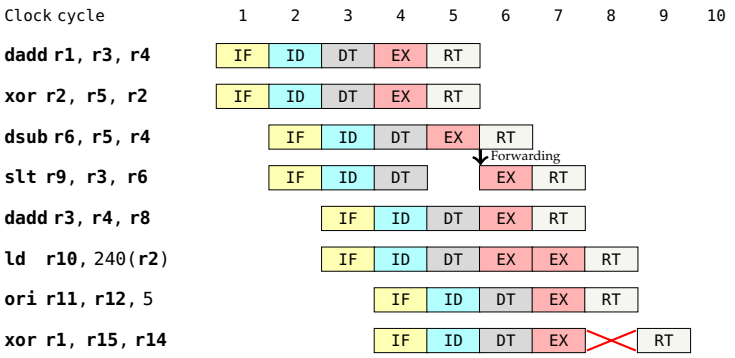


Figure 2.56: Superscalar execution

The two instructions are decoded at the same time in cycle #2, it is verified that there are no dependencies between them, destination registers are renamed and the 4 source registers are read from the register file. If some instruction depended on the result of a previous instruction, it would be tagged to indicate this situation (this is not the case).

The two instructions are distributed to the reservation stations in cycle #3. In this case, since one is an arithmetic operation on integers and the other a logical operation, both instructions are distributed to the reservation station of the integer execution units.

In cycle #4 both instructions are issued and executed in the integer units (each of them in one of the two available units). At the end of this cycle the instructions, already executed, are stored in the reordering buffer at the output of the EX stage to keep the program order. The calculated values of `r1` and `r2` are published on the forwarding bus in case they are required by any instruction waiting at a reservation station.

During cycle #5 the results of both instructions are written in the register file following the program order.

- The next two instructions are fetched in cycle #2. These are two integer arithmetic instructions, so they could be executed at the same time. However, there is a RAW dependency between the two on register `r6`. For this reason, `slt r9, r3, r6` must wait at the reservation station until `dsub r6, r5, r4` computes the value of `r6` in cycle #5.

  In cycle #6, the value of register `r6` is available to `slt r9, r3, r6` through the forwarding bus, so this instruction is issued, executed in an integer execution unit and retired one cycle later (in the RT stage).

- The following instructions are fetched in cycle #3. One is an integer arithmetic instruction and the other a load instruction, so they can be executed at the same time in an integer unit and the load/store unit, respectively. This last unit requires 2 cycles, while the integer unit needs only one cycle. Hence, one instruction is retired in cycle #7 and the other in cycle #8.

It is important to note that there are three instructions in the EX stage during cycle #6; a number greater than the CPU width. Two instructions are using the integer execution units and the third is in the load/store unit. The same happens in cycle #7.

- The following two instructions are fetched in cycle #4. They are two logical instructions, so they could be executed at the same time in the two available integer execution units. However, the load instruction `ld r10, 240 (r2)` is retired in cycle #8, so only the first of these two new instructions can be retired at the same cycle. Instruction `xor r1, r15, r14` must wait until cycle #9 to be retired. A structural stall occurs, since two instructions are retired in cycle #8.

Ignoring the four-cycle initial transient period, the execution of the 8 previous instructions requires 5 cycles (from cycles #5 to #9), which results in IPC $= 8/5 = 1.6$, which is equivalent to CPI $= 0.625$, lower than the theoretical minimum limit of 1 for a pipelined scalar CPU.

Influence of speculative execution on security

As mentioned, it is usual to use speculative execution techniques such as branch prediction and out-of-order execution to obtain a high performance in current architectures. This means that the effects of instructions executed speculatively must be reverted in occasions. For example, when a branch is mispredicted or an exception occurs during an instruction executed out of order. This can be very complicated and, in fact, has been the cause of a series of vulnerabilities in mainstream processors (especially by Intel and AMD, but also by ARM). The first to be found, in 2017, were called Meltdown and Spectre, and combine speculative execution with other architectural concepts such as caches, privilege levels and exception management.

Although the details on how they are exploited are different in each case, the general idea is always the same: trying to guess values (such as passwords or credit card numbers) that should not be accessible to a program by speculatively executing instructions that access memory addresses forbidden for that program. When it is discovered that these instructions should not be executed, their effects are reverted, but they leave a trace of their speculative execution. This trace may be a change in the execution time of the program. By measuring the changes in the execution time, theoretically inaccessible values can be obtained through what is called a side-channel, a name used because information is not accessed directly through the main channel, but indirectly.

These vulnerabilities are forcing CPU designers to modify their designs, trying to avoid these attacks. For example, one of the attacks is based on improperly training the branch prediction tables from a program thread. Measuring the changes in execution times, this thread can guess data from another program. One solution to this problem is to empty the branch prediction tables each time the running thread is changed. Another alternative is to have independent branch prediction tables for each thread. In any case, the two alternatives raise performance penalties or increase costs, which means new and very complex problems for CPU designers.

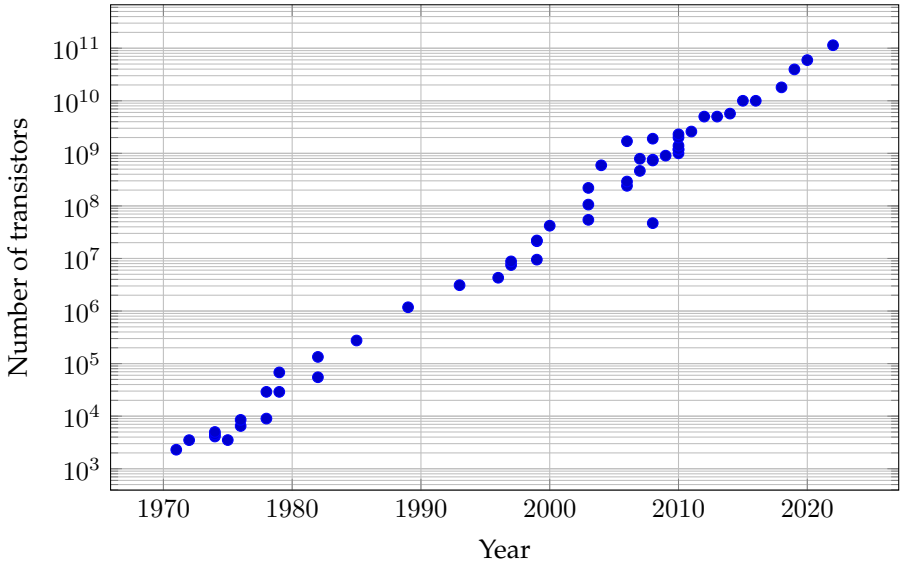


Figure 2.57: Number of transistors in processors for the past few decades

2.5 Moore's law

The CPU microarchitectural and manufacturing improvements discussed in the previous sections pursue a common objective: to increase CPU performance. Unfortunately, current technology is reaching some limits. This section describes these limits.

The term Moore's law refers to the idea that the computational power will increase dramatically, while its cost will be reduced, at an exponential rate. It is based on a prediction made by Gordon Moore (co-founder of Intel) in 1965.

The prediction is based on the improvements in the manufacturing process used for processors.[12] Processors are manufactured using integrated circuits, also referred to as chips or microchips, which are made from silicon glass wafers. Circuit components, such as transistors or cables, are etched onto the surface of the wafer using chemical techniques. A single wafer is used to build as many chips as possible in order to make the most of its surface and save costs.

Over the years the manufacturing process has been improving. The manufacturing process is designated by the feature size, i.e., the size of the smallest feature (a wire, a transistor, or some other such circuit component) that can be etched onto the surface of a wafer. Improvements in the manufacturing process allow the manufacturer to build smaller and faster transistors, and thus, including more transistors per unit area. Figure 2.57 shows the evolution in the number of transistors of processors over the years using logarithmic scale.

[12] In this text, the term CPU is used to refer to a processing unit, while the term processor refers to the device that has its own die and includes one or more CPUs. When the die includes a single CPU, processor and CPU are used interchangeably.

These improvements in the manufacturing process are reflected in the factors of the iron law of performance that provides the CPU time. They are analyzed in detail next.

$$T_{CPU} = \text{Instructions per program} \times \text{CPI} \times T$$

Having more transistors provides the opportunity for introducing improvements in the microarchitecture, such as those shown in previous sections. During the early years of development of superscalar CPUs, the extra available transistors were used to design superscalar CPUs with an increasing issue width until the degree of parallelism of the programs was exhausted. Programs have a limited ILP level, thus at some point increasing the issue width does not reduce the CPI and, therefore, does not improve performance. The consequence from the point of view of the CPU manufacturer is that further complicating the microarchitecture does not provide any benefit, as it does not result in performance improvements. Current superscalar CPUs usually have an issue width less than or equal to four.

Having faster transistors provides the opportunity for increasing the working frequency, and thus, the performance of a microarchitecture, as it reduces the clock period. The drawback is that the energy consumption of the processor increases exponentially with the working frequency. Higher working frequency means more power consumed and, therefore, more power dissipated. The dissipation of power above 100 W in devices of approximately one square centimeter provokes thermal issues that require very expensive technological solutions.

Figure 2.58 shows the increase in processors frequency for the last decades using logarithmic scale. Due to power consumption and heat dissipation, the increase in frequency stopped approximately in 2005. The technological limit that prevents frequency from continuously increasing is referred to as the power or energy wall. This power wall limits the frequencies from increasing well beyond 4 GHz.

Considering the maximum ILP of programs and the impossibility of increasing the working frequency, the exponential growth of computational power that was correctly predicted until then by Moore's law slowed down years ago. Moreover, the introduction of new manufacturing processes requires increasingly large investments by manufacturing companies, thus, these new technologies tend to appear more distant to each other. This does not mean that CPU performance has not increased since then, there has been improvements, but much more slowly, using other forms of parallelism, described in section 2.6.

Taking into account the limitations of ILP and power, there are fundamentally three ways to take advantage of manufacturing improvements of integrated circuits, as can be seen in figure 2.59.

1. Manufacture the same functionality in smaller area and maintain frequency. The reduction of the surface is an advantage in terms of cost, since more processors can be produced per wafer. In addition, it has another important advantage: energy consumption is reduced because it is proportional to the surface. Energy consumption is a key parameter in the processors of mobile and IoT (Internet of Things) devices, so this is an interesting option.

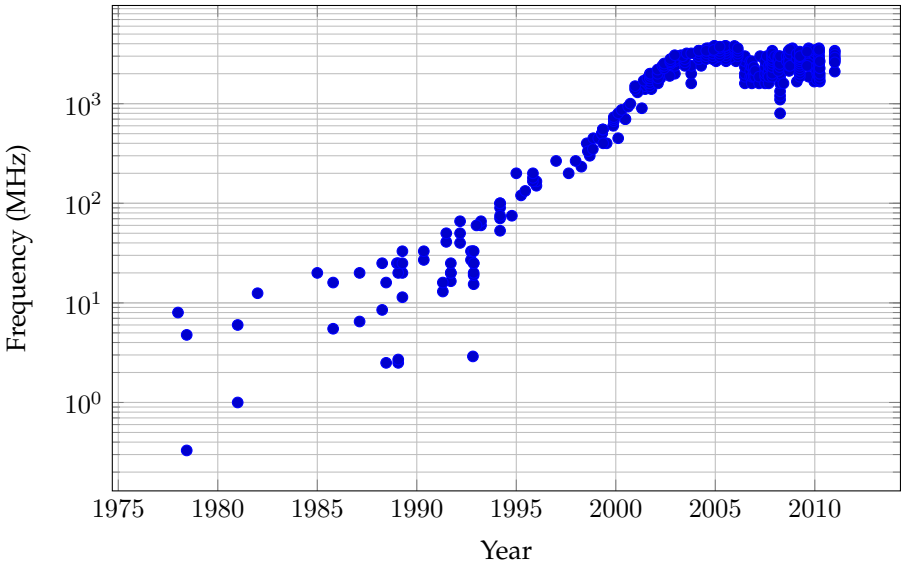


Figure 2.58: Frequency in Intel processors for the past few decades

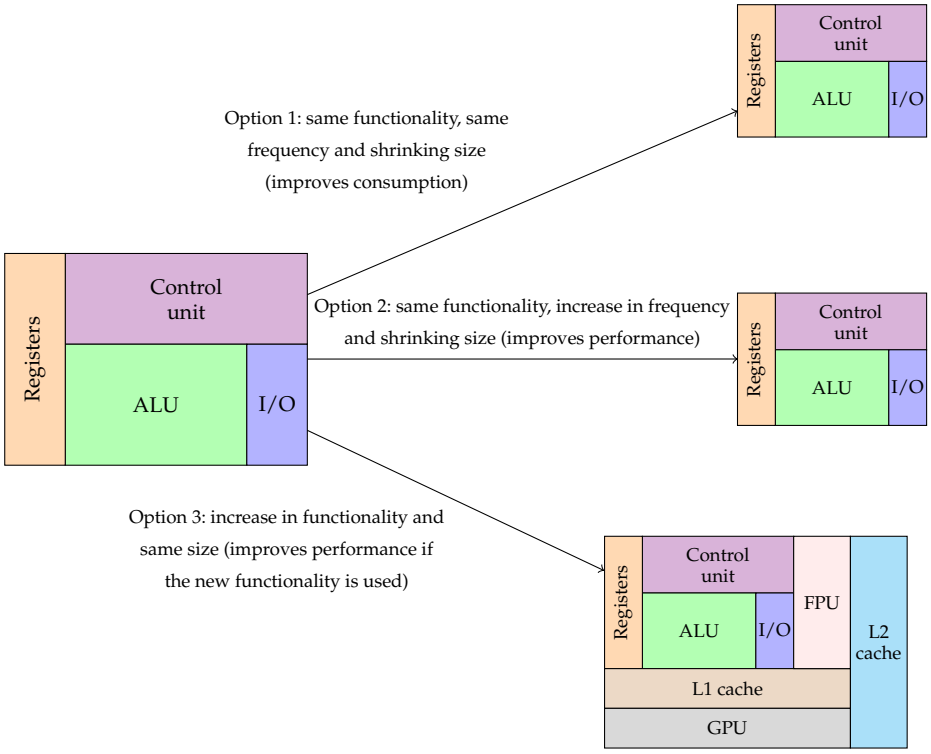


Figure 2.59: Options for improvement when shrinking the manufacturing process

2. Manufacture the same functionality in smaller area and increase frequency. The idea behind this option is to take advantage of the reduction in consumption, obtained when reducing the area, to increase frequency, which again increases consumption, until the reduction on one side is compensated with the increase on the other. The problem with this option is that reductions in consumption due to the reduction of the surface are quickly absorbed by small increments of frequency. The reduction of power is proportional to the reduction of surface, but it grows exponentially with frequency. For this reason, the possible frequency increases with manufacturing improvements are very small.

3. Add more functionality for the same surface and maintain frequency. The increase in the number of transistors makes possible the integration of other components of the computer into the processor, such as I/O devices, memories, etc. This achieves performance improvements, since communications between these components and the processor are faster. For example, the integration of part of the memory inside the processor chip reduces memory access time. Another possibility when adding more functionality is to integrate several CPUs within the same chip, resulting in multicore configurations. This type of configuration is described in section 2.6.

In summary, manufacturing improvements were traditionally used to implement increasingly sophisticated microarchitectures and increase clock frequency. Once this path was exhausted, new techniques began to appear where increases in performance are also obtained, although more modest. Next section describes some of these techniques.

2.6 Multi-threaded CPU

The usual way to increase performance beyond the limits imposed by manufacturing technology is to increase the level of parallelism of the system. So far, instruction level parallelism has been used to achieve performance improvements, but there are other forms of parallelism, described in Flynn's taxonomy.

2.6.1 Flynn's taxonomy

Flynn's taxonomy is a classification of computer architectures proposed in the 1970s and still used today. It is based on the number of concurrent instructions and data streams available, and considers four large groups: SISD, SIMD, MISD and MIMD. Below are the main characteristics of each group.

SISD (Single Instruction, Single Data). SISD describes the organization of a computer that executes a single stream of instructions on a single data stream. This is the case of von Neumann and Harvard architectures.

SIMD (Single Instruction, Multiple Data). SIMD describes the organization of a system that executes a single stream of instructions on multiple data streams, that is, it executes the same instruction on several data items at the same time, which results in a significant performance improvement. For example, if it is necessary to add two vectors of four elements, considering the corresponding components, to obtain a third vector, rather than using a loop to iterate four times across the sequence of values adding each corresponding component, a single SIMD instruction could be used to perform all four additions at the same time. This implies a great advantage in terms of the time required for instruction fetch and execution.

An example of a SIMD system is the graphic processing unit (GPU) of a personal computer, since it is common for graphics procedures to apply the same operation on several pixels of an image. The CPUs used in current personal computers also have SIMD instructions. Multimedia and signal processing applications usually take greater advantage of SIMD instructions for exhibiting this type of parallelism.

MISD (Multiple Instruction, Single Data). MISD describes the organization of a system that executes multiple instruction streams on a single data stream.

MISD systems do not use parallelism to achieve performance improvements, but for fault tolerance, since they execute the same instruction in parallel on the same data item to detect errors at run-time. They are used in critical systems, such as aviation systems.

MIMD (Multiple Instruction, Multiple Data). MIMD describes the organization of a system that executes multiple instruction streams over multiple data streams. This is the generalization of von Neumman or Harvard architectures.

There are two alternatives to implement a MIMD system with several CPUs and several memory devices:

- MIMD using distributed memory. It consists in replicating the CPU/memory pairs and connecting them through a communication network. Each memory device can only be accessed by the CPU assigned to it.
- MIMD using shared memory. It consists in building a set of CPUs and memory devices so that any CPU can access any memory device through an interconnection mechanism.

Current processors mostly follow a shared memory MIMD approach. All CPUs share the system memory. This is the case of multicore systems, which are presented in section 2.6.3. This approach, based on the execution of several programs (or threads) concurrently, increases performance beyond the limits established by ILP and the power wall.

2.6.2 Thread level parallelism

A thread is the minimum unit an operating system can schedule for execution. Each program has at least one thread, but some programs may have multiple threads. In this case, the CPU performance can be improved by running multiple threads sharing the CPU in what is called multi-threading. This technique takes advantage of task parallelism or thread level parallelism (TLP). The CPU only executes one thread at any time, so threads must share CPU time. The switches between threads in the CPU must be done very fast in order to make this technique efficient.

Thread level parallelism is different from instruction level parallelism, exploited by the CPU pipeline or superscalar architectures. Two fundamental differences between both types of parallelism are:

- ILP improves the performance of each individual thread, regardless of the number of threads. Thus, the performance of a single-thread program is automatically improved, without changes in the programming paradigm.
- TLP, on the other hand, only improves performance when several threads are executed at the same time. Unlike ILP, TLP is not transparent to programs; it requires transforming single-thread programs into multi-thread programs in order to take advantage of the performance improvements. Multi-thread programming is significantly more complex than single-thread, as it requires taking the synchronization between multiple threads into account.

An evolution of multi-threading used in many current CPUs is simultaneous multi-threading (SMT). The underlying idea is that CPUs offer a high number of parallel functional units that are not always used at the same time, either because of the type of instructions being executed, or the program has a low ILP. Thus, it is possible to simultaneously run several threads instead of sharing CPU time. However, not all the functional units required to execute any combination of instructions in multiple threads are replicated. The best-known implementation of SMT is the Intel HyperThreading technology, where two threads may be executed simultaneously in a single CPU. This is only possible under certain conditions, so the performance increase is limited.

2.6.3 Multicore processors

As indicated above, at the beginning of the first decade of the 21st century large increases in frequency ceased to be significant due to energy issues and heat dissipation. Thus, the most commonly used method to increase performance was to transform architectures with a single CPU into shared memory MIMD architectures following the scheme shown in figure 2.60. CPUs based on this type of architecture share the main memory, in such a way that the memory access time is similar for all CPUs. For this reason, this type of architecture is often referred to as Shared Memory Processors (SMP) or Uniform Memory Access (UMA).

A multicore or multicore processor refers to a processor that contains two or more CPUs, or cores. In this way, the extra transistors provided by the improvements in

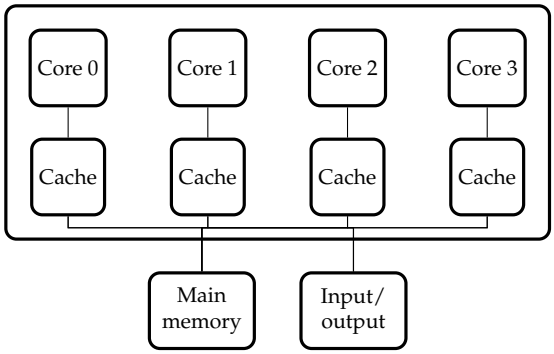


Figure 2.60: Basic architecture of a multicore processor

manufacturing technology are used to integrate several CPUs within the same chip, obtaining new performance improvements despite the maximum ILP of programs and limitations in power dissipation.

Nowadays, almost all multicore processors are SMPs, since they usually share a single physical address space. Multicore technology is common for all types of processors in desktops and laptops, and it is also present even in mobile phones and tablets.

Multicore processors can also implement SMT techniques, so each processor core can execute more than one thread simultaneously. For example, a processor with four cores that implements SMT in such a way that it can schedule two threads on each core offers the operating system the possibility to execute up to eight simultaneous threads. However, these eight threads are not equivalent, as it is not the same to schedule two threads within the same core as in different cores. When two threads are scheduled within the same core, sometimes they will be able to run simultaneously and sometimes not (when they require a non-replicated functional unit). When two threads are scheduled in different cores, they will not compete for the same functional units, thus, they will be able to run simultaneously.

Multicore processors provide better performance in terms of throughput. However, this technique has limitations. Using many cores gives rise to a new wall: the memory wall. Many cores accessing memory simultaneously cause memory to become a bottleneck. This effect is mitigated with the use of caches. As shown in figure 2.60, each core has usually one or more levels of memory, called cache memory. These caches are essential to reduce the dependency on memory. However, considering the current growth rate in the number of cores, it is estimated that the number of cores will be limited by the memory wall in the future. The cache memory is thoroughly studied in chapter 3.

In general, there are three walls that limit performance: the power wall, the memory wall, and the ILP wall. The combination of these three walls is known as the brick wall, and it is a way to describe the technological limitations that prevent computational power from maintaining the growth at the same pace as in the past.

2.7 Multitasking OS support

So far, this chapter has presented an example of instruction set architecture: the MIPS64 architecture. Based on this architecture, different microarchitectures have been developed with the aim of improving performance within technological limitations. While performance is a very important part of a CPU, there are other important aspects related to the CPU functionality. One of the requirements of any modern CPU is the support for multitasking operating systems. Broadly speaking, functionality is linked to the instruction set architecture, that is, it indicates what the CPU can do, while performance refers to the speed this functionality is performed. The latter depends on the microarchitecture. Nevertheless, some of the CPU features supporting multitasking operating systems have consequences in terms of performance.

Basic concepts of multitasking operating systems are introduced in this section before presenting the support of the architecture for them.

2.7.1 Introduction to multitasking OS

A user does not usually use a bare computer, that is, a computer without any software installed. In general, an operating system is installed in the computer. The combination of a computer and the operating system constitutes a machine that provides a series of services to user applications, which interact with the hardware through a perfectly defined API (Application Programming Interface). This makes application programming much easier, since the operating system provides a hardware-independent interface. The operating system also becomes a manager of computer hardware resources in order to provide services to applications. The relation between hardware, operating system and applications is represented in figure 2.61.

Virtually all current operating systems are multitasking OSs. This means that the set formed by the computer and the OS supports multiprogramming, that is, it can run several programs concurrently.

A multitasking operating system is a program with access to all the resources of the machine. It provides services for the rest of the programs, which are generally referred to as tasks.[13] At any given time, assuming a single CPU or single-core processor, either a task or the operating system will be running. Figure 2.62 shows an example of execution of three tasks and the operating system. As can be seen, the execution time of the operating system is very low compared to that of the tasks.[14] Over a period of 50 milliseconds, each task is executed from two to three times. This is perceived by the user as the simultaneous execution of the three tasks, although in reality one task or the operating system is in execution at any time. Pausing and saving the state of a task and restoring the state of a different task to resume execution is called a context switch.

Figure 2.62 shows how to change the running task, that is, perform a context switch. This requires a transfer of control to the operating system, i.e., transferring

[13] The term task is used as an abstraction for the concepts of process and thread, commonly used in operating systems.

[14] This is not true when the operating system is executing the code of a complex service requested by a task.

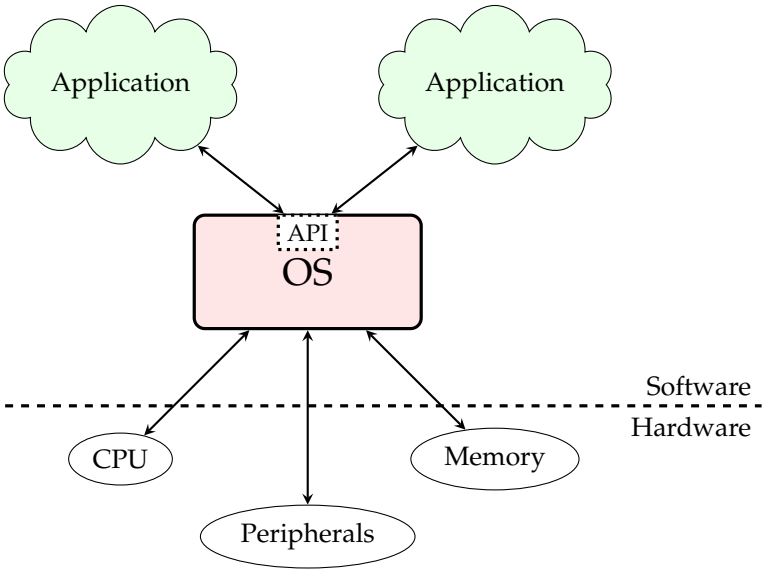


Figure 2.61: Relation between hardware, operating system and applications

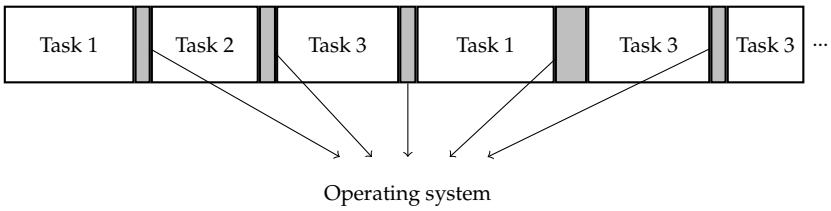


Figure 2.62: Timeline for the execution of the OS and three tasks

the execution to the operating system. A multitasking operating system takes control of the machine when one of these three situations occur:

1. A task requests a service from the operating system by making a system call.
2. An interrupt occurs.
3. An exception occurs.

A handler or routine of the operating system is executed in any of these situations. Some issues need to be considered:

- Every time a program makes a system call, or an interrupt or an exception occurs, the handler may modify the state of the tasks. For example, it can revoke the control of the CPU from one task to grant it to another, as can be seen in figure 2.62.
- System calls, interrupts and exceptions can be nested. For example, an exception may be generated during the execution of an interrupt handler, or an interrupt handler may be also interrupted to execute the handler of a higher priority interrupt that has just occurred.
- It is necessary to be careful with the nomenclature used in real CPUs, since there is no common agreement in the technical literature. In some CPUs, system calls, interrupts and exceptions are included under the generic term exceptions, as has been done in this text in previous sections. Others use the generic term traps. There are other CPUs where system calls are known as software interrupts, such as x86 CPUs. In the rest of this text the terms used are *system calls*, *interrupts* and *exceptions*.
- A timer is required to generate a periodic interrupt every T time units. This ensures that the operating system regains control of the machine in the worst case every T time units. This prevents a task from monopolizing the CPU when no exceptions or interrupts are triggered.

The following sections describe the three mechanisms for transferring control to the operating system.

Calling operating system services

It consists in an application calling a routine of the operating system that provides a service. For example, when a program needs to open a disk file, it makes a system call. It is important to understand that a program cannot access the disk directly; it must always do it using services provided by the operating system.

There are two types of instructions related to system calls. Generically, they are called `Sysret` and `Syscall`, although their real name depends on the architecture.

- `Syscall`. Instruction to request an operating system service through its API. The control of the CPU is transferred to the operating system when a task executes this instruction.

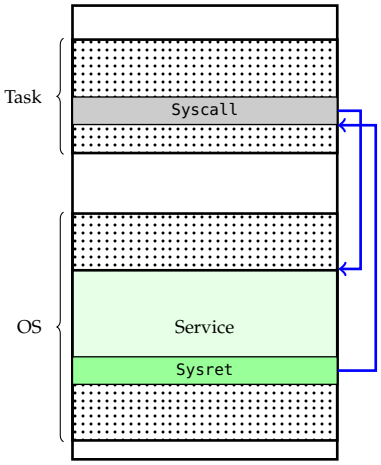


Figure 2.63: Call and return of an operating system service

- `Sysret`. Instruction to return from a system call. The operating system executes this instruction just before returning control of the CPU to the task that requested the service (or another task if a context switch occurs). The operating system accesses the memory space of the task between the execution of `Syscall` and `Sysret`. For example, in the case of opening a file, it writes information about whether the file opening operation was successful in a data structure of the task. After executing `Sysret`, the next instruction after `Syscall` is executed.

Figura 2.63 shows how the call and return of an operating system service occurs.

Interrupts

Interrupts are a hardware mechanism used by a peripheral interface to ask the CPU to temporarily interrupt the execution in progress in order to execute an interrupt service routine. This routine, or interrupt handler, is part of the operating system. As with systems calls, the last instruction of the routine is `Sysret`. Then, the CPU continues execution from where interrupted.

For example, interrupts occur every time a packet of data is received through a network interface, the user presses a key on the keyboard or the system timer expires.

Figure 2.64 shows the execution flow of the CPU during an interrupt. The steps are the following:

- An interrupt can occur at any time and the CPU does not know *a priori* when it will happen.
- The execution of the current instruction is completed and the CPU jumps to the interrupt service routine, which is part of the operating system.
- The interrupt service routine is executed until instruction `Sysret` is reached.

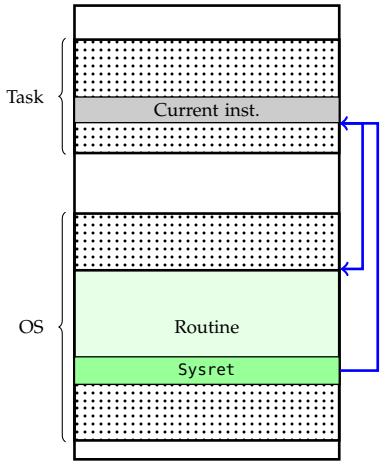


Figure 2.64: Execution flow during an interrupt

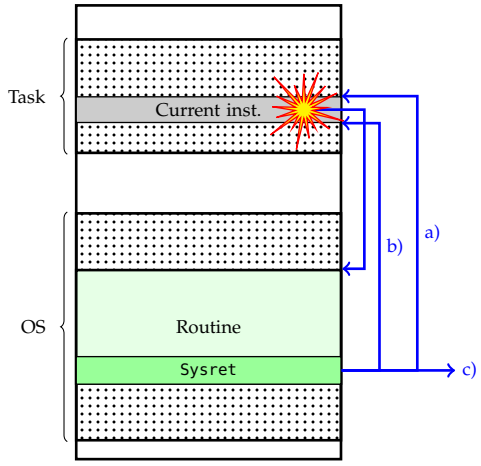


Figure 2.65: Execution flow during an exception

- After `Sysret`, the execution of the task is restored. The CPU executes the next instruction after the one it was executing when it was interrupted.

Exceptions

The purpose of exceptions is to transfer control to the operating system when an abnormal situation occurs during the execution of an instruction. Contrary to what happened with interrupts, it is not a peripheral that originates an exception, but the CPU. There are many situations that can trigger an exception. For example, when an instruction that attempts to perform a division by zero is executed, or when the CPU tries to execute an invalid instruction code.

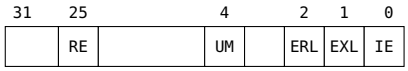
Figure 2.65 shows the execution flow of the CPU during an exception. The steps are the following:

- During the execution of an instruction an exception is triggered.
- Control is transferred to the exception handling routine, which is part of the operating system.
- The exception handling routine is executed until instruction `Sysret` is reached.
- `Sysret` generally returns to:
  a) the same instruction causing the exception, or
  b) the instruction just after the instruction that caused the exception, or
  c) it cannot return.

2.7.2 Support for multitasking OSs

CPU support for multitasking operating systems establishes a series of minimum functional requirements that must be implemented in the architecture. These requirements are the following:

1. There are privileged instructions in the instruction set that only the operating system should be able to execute. These instructions modify the operating mode of the CPU or change its configuration. A clear example is an instruction that disables interrupts. If a task could execute this privileged instruction, the system as a whole would stop responding to external events, such as the interrupts requested by the keyboard interface when a key is pressed. On the other hand, the operating system must be able to execute any instruction required to manage the resources of the machine.
2. The memory of the OS must be protected from reads and writes by user tasks. The operating system is another program stored in the memory of the computer. If a user task could write in the memory area associated with the operating system, it could modify its operation, taking control of the machine or even causing a malfunction. If a task could read the memory of the operating system, it could access private information from another user, or even perform reads on the peripheral interfaces, which are mapped in the memory area of the operating system.
3. The memory of a task must be protected from reads and writes performed by another task. A task should not have access to the memory area of another task, as it could access private data. Writing in the memory of another task should not be allowed either, since it can modify its operation.
4. Whenever a system call, interrupt or exception occurs, the CPU must provide mechanisms to save the execution state. Thus, execution can be restored where it was once the service routine is completed.



IE: Interrupt Enable
EXL: Exception Level
ERL: Error Level
UM: User Mode
RE: Reverse Endian

Figure 2.66: Some bits of the status register of the MIPS64 architecture

2.7.3 Support for multitasking OSs in MIPS64

The functional requirements that must be implemented in the architecture in order to support multitasking OSs have been introduced. Next, an example with a real architecture is presented: the MIPS64 architecture. As will be seen, new instructions and registers will appear that were not necessary before.

Levels of privilege

The execution privilege level (or mode), or protection ring, is the state of the CPU that indicates which instructions can and cannot execute. The most privileged level is usually associated with the operating system, while the least privileged corresponds to user tasks. In the former, all the instructions in the instruction set can be executed, while in the latter only a subset of the instruction set can be executed.

The privilege level of a CPU is defined by one or more control bits. In the case of the MIPS64 architecture there are basically two levels of privilege:

- User. Level in which user tasks are executed.
- Kernel. Level in which the operating system is executed.

The privilege level is defined by bits `UM`, `EXL` y `ERL` of the status register. Figure 2.66 shows these bits in the status register.[15] In user mode, the value of the bits is as follows: `UM`=1, `EXL`=0 and `ERL`=0. In kernel mode, the value of the bits is as follows: `UM`=0, `EXL`=1 and `ERL`=1.

One of the privileged instructions in the MIPS64 architecture is `di`, which disables interrupts. This instruction can only be executed in kernel mode. An exception is triggered if it is executed in user mode.

The microarchitecture changes the state of bits `UM`, `EXL` and `ERL` to switch from user mode to kernel mode when an interrupt, system call, or exception occurs during the execution of a task. The instruction `syscall` is used to make a system call.

The instruction used to finish the execution of a service routine is `eret`. This instruction modifies the value of bits `UM`, `EXL` and `ERL` to switch the CPU to user mode.

[15] This figure also shows the interrupt flag, `IE`, and bit `RE`, which determines if the CPU is working in little endian or big endian mode.

Memory protection

In order to protect the memory of the operating system from user tasks, the MIPS64 architecture reserves a specific range of addresses. Any address in the range 4000 0000 0000 0000h to FFFF FFFF FFFF FFFFh can only be accessed in kernel mode, that is, only by the operating system. OS data and code located in this range are protected from user tasks. User tasks are asigned the range 0000 0000 0000 0000h to 3FFF FFFF FFFF FFFFh. When a task tries to access a memory address outside this range, either with a load or a store instruction, an exception is triggered.

In order to protect the memory of one task from other tasks, the MIPS64 CPUs incorporate a memory management unit that implements a paged virtual memory system. The paged virtual memory is used to assign a unique address range to each task, which does not overlap with those of other tasks. When a task tries to access an address from a range of addresses of a different task, an exception is triggered. Paging will be studied in chapter 3.

Saving the execution state

When an interrupt, system call or exception occurs, it is necessary to save the execution state in order to be restored later and resume execution. In the case of the MIPS64 architecture there are two related registers, `EPC` (also called `Exception PC`) and `cause register`.

`EPC` stores the address of the instruction to execute when the routine finishes. It can be the address of the instruction after the one that caused the interrupt, system call or exception, or the address of the same instruction that caused an exception.

The `cause register` stores a value that identifies the interrupt, system call or cause of the exception.

When an interrupt, system call or exception occurs, interrupts are disabled to prevent another interrupt from overwriting `EPC` and `cause register`. The time while interrupts remain disabled should be as short as possible. Thus, the service routine immediately creates a copy of `EPC` and `cause register` in a secure location and re-enables interrupts.

Returning from a service routine is carried out by writing to `EPC` the previously saved return address and then executing `eret`. Then, the CPU switches to user mode and resumes execution in the instruction whose address was saved in `EPC`.

2.8 Virtualization support

Virtualization is a technique that allows to run several operating systems with their applications on the same machine. It has innumerable advantages and is used both in servers and desktop computers. In this section, a brief introduction to virtualization is provided first. Next, an example of virtualization is presented, in this case with the x86 architecture.

2.8.1 Introduction to virtualization

Virtualization is a general concept where a virtual (fictitious) entity is presented as if it were real. In computer science, it is used in many areas; the closest to computer architecture is what is usually called hardware virtualization, where virtual machines are created as duplicates of the real, or physical, machine. Operating systems can be installed in these virtual machines as if they were physical machines. This has many advantages, such as running different operating systems on the same hardware at the same time, isolating applications from each other, or easily migrating entire systems (operating system and applications) between different physical machines.

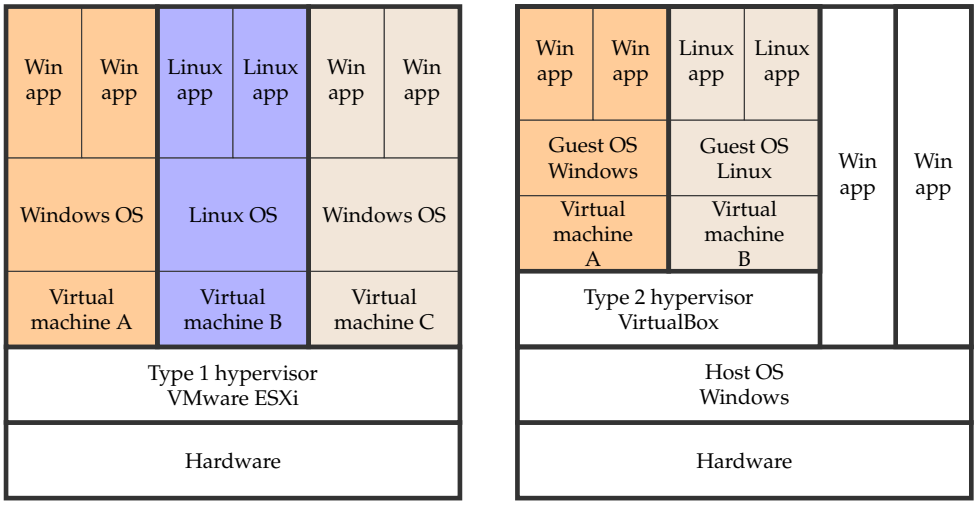
At the beginning, computer architectures were designed assuming that only a single OS had to be supported. The OS would manage hardware resources between different applications. Operating systems, in turn, were designed assuming they were managing hardware that was not shared with other operating systems. Therefore, it was not possible to run two different operating systems at the same time on a single physical machine. The solution was to add an abstraction layer that supported virtualization.

An hypervisor or Virtual Machine Monitor (VMM) refers to the component that interacts directly with the real hardware and presents a virtual hardware to the operating systems of virtual machines, multiplexing the hardware (real) between several virtual machines. There are two types of hypervisor (see figure 2.67):

- Type 1: the hypervisor is a small software layer that runs directly on the real hardware and has the sole purpose of managing resources between virtual machines; it is not dedicated to execute other applications.
- Type 2: the hypervisor is an application that runs on an operating system called host, which manages the real hardware and can run other applications. The hypervisor is responsible for providing virtual machines where the other operating systems will be executed, which are called guests.

Type 2 hypervisors provide less performance than those of type 1, since the host operating system represents an additional software layer. However, unlike type 1 hypervisors, which run on very specific hardware, type 2 hypervisors have the advantage of running on almost any hardware, since the host operating system provides the necessary support.

Initially, most architectures did not support virtualization. However, even in that case a less efficient VMM could be built, running the guest operating systems with the privilege level of the tasks and using the technique called trap-and-emulate. In this way, each time a guest operating system executed a privileged instruction, an exception was triggered that was captured by the hypervisor, emulating the behaviour of the instruction on real hardware and then returning the control to the guest operating system. Even using the trap-and-emulate technique there were instructions that caused no exceptions when executed in non-privileged mode, but they changed their behaviour when executed in privileged mode. This type of instructions is called sensitive. To solve this issue, it was necessary to use a technique called binary translation, which translated sensitive instructions into other non-sensitive instructions



(a) Type 1 (b) Type 2

Figure 2.67: Types of hypervisor

before they were executed by the CPU. The main problem with the trap-and-emulate technique and binary translation is the considerable reduction of performance in the execution of the virtual machine.

Another more efficient alternative than using trap-and-emulate and binary translation techniques is paravirtualization. Basically, it consists in modifying the host operating system by replacing its privileged code fragments with calls to hypervisor services. Therefore, a relationship is established between the hypervisor and the guest operating system analogous to that between an operating system and a user task. The biggest drawback of paravirtualization is that it requires modifying the guest operating system, which is difficult. Moreover, this is not feasible if the source code is not available.

Nowadays, general-purpose CPUs provide hardware support for virtualization in such a way that trap-and-emulate and binary translation techniques are no longer necessary, which reduces the gap between the performance of the virtual machine when compared to the real machine. Although in this case paravirtualization is not necessary, some operating systems implement a reduced version, so that during its execution they can determine if they are running directly on the hardware or on a paravirtualized hypervisor, thus they can optimize its operation. For example, the VirtualBox hypervisor provides the KVM paravirtualization option that allows to optimize the performance of virtual machines running modern versions of Linux.

2.8.2 Virtualization support in the x86 architecture

Modern versions of the x86 architecture have introduced hardware support to virtualization in the form of new instructions and modes of operation. The first improvements are included under a set of extensions called VT-x, in the case of Intel, and

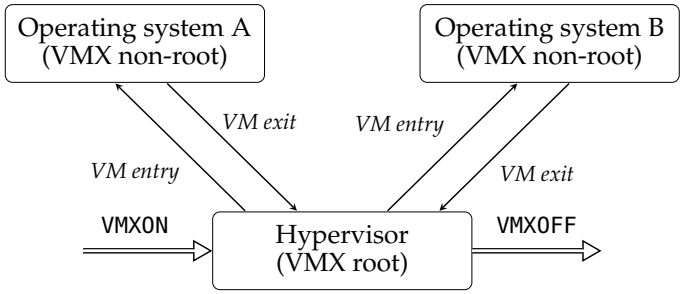


Figure 2.68: Control transfer with the VT-x extension

AMD-V, in the case of AMD. Both extensions are incompatible, so the implementation of a hypervisor in the x86 architecture usually requires identifying the supported extension. Basically, these extensions introduce an additional level of privilege, more privileged than the level of privilege associated with operating systems, as well as a set of new instructions and data structures related to virtualization.

The operation of virtualization is illustrated using Intel VT-x as an example. When the architecture includes this extension, the CPU implements a new operation mode associated with virtualization, called VMX. The CPU enters this mode by executing a new instruction, `VMXON`, and leaves this mode using another new instruction, `VMXOFF`. Once the CPU is in VMX mode, it can be in two operating sub-modes:

- VMX root. Mode in which the hypervisor is executed with access to the complete instruction set. In systems that use type 2 hypervisors, the host operating system is also executed with this privilege level.
- VMX non-root. Mode in which the operating systems of all the virtual machines are executed with access to a subset of the instruction set (larger than the subset of user tasks).

Figure 2.68 shows the changes between operation modes using the new instructions included in the VT-x extension.

In VMX root mode, the CPU has instruction `VMLAUNCH` to start a virtual machine, transferring the control to the corresponding operating system. The transfer of control from the hypervisor to the operating system is called VM entry. Once the `VMLAUNCH` instruction is executed, the CPU enters VMX non-root mode.

Once the operating system takes control of the system, it runs normally until it executes certain instructions. At that moment, the control of the system is transferred from the operating system to the hypervisor, which is called VM exit. Once the hypervisor takes over, it can properly handle the instruction that caused the VM exit using the trap-and-emulate technique. The operating system then regains control of the machine when the hypervisor executes `VMRESUME`, which results in a new type of VM entry.

VT-x and AMD-V extensions significantly improve the performance of virtual machines, improving the management of the CPU. However, there are other components of the computer that suffer performance issues derived from virtualization that

are not alleviated with these extensions. This is the case of virtual memory and the input/output system. In the case of virtual memory, architectures implement new extensions, which are described in section 3.5, while in the case of the input/output system they are described in section 4.4.

Chapter 3

The memory hierarchy

This chapter shows how the computer memory system is built using different storage technologies, trying to obtain a fast, cheap and high-capacity memory system. Nowadays, this is achieved by combining three different memory technologies:

- SRAM memory. Used in the cache memory.
- DRAM memory. Used in the main memory.
- Secondary storage. Managed with the virtual memory.

3.1 Introduction

The memory system is the second most important component in the computer after the CPU. The memory system is used to store the instructions and data that constitute programs. Therefore, it needs to be large in order to store many programs with many instructions and much data. In addition, it needs to be fast, since the CPU accesses memory at least once per instruction: in the instruction fetch stage. If the instruction is a load or store instruction, a second access is required, this time to read or write data, respectively. In general, the CPU is faster than the memory, so instructions accessing memory may stall, waiting for the memory system.

Speed differences between CPU and memory have widened with time, as they have experienced different technological advances. The speed of the CPU has greatly increased, so has the number of processor cores. On the contrary, memory has evolved mainly in density, that is, in storage capacity, but to a lesser extent in speed. Figure 3.1 shows the evolution in terms of speed followed by both the CPU and the memory.[1]

When the CPU waits for the memory, performance decreases rapidly. This decrease is even greater when considering recent trend to include more and more cores

[1]These figures are a mere approximation, since measuring the speed in MHz is misleading. The performance of CPUs working at the same clock frequency can be very different.

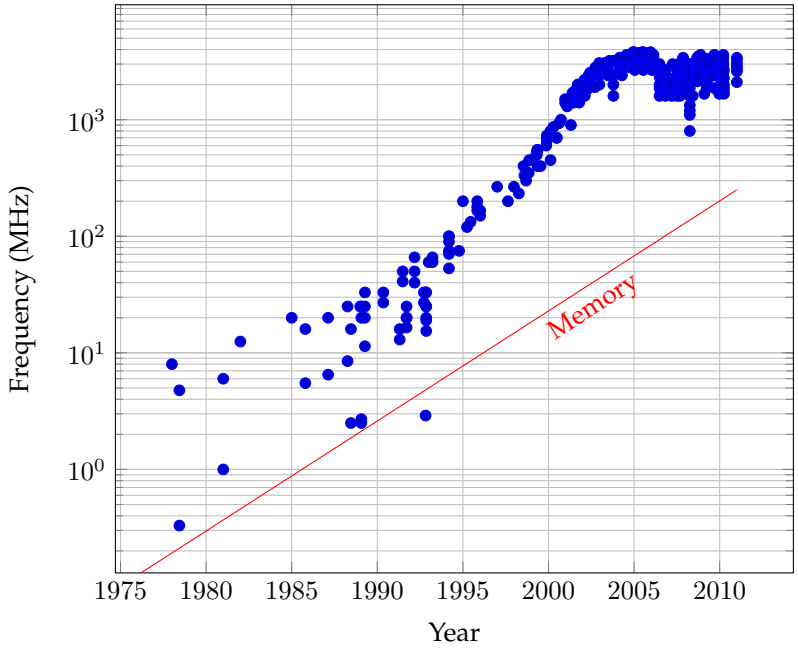


Figure 3.1: Evolution in terms of speed followed by both CPU and memory

in the processor. This causes the effective performance of the CPU to decrease because it needs to wait for memory.

For example, a pipelined CPU working at a frequency of 1 GHz could be capable, ideally, of executing instructions with a CPI of 1, that is, it could execute 1000 MIPS. However, if assuming the real CPI of this CPU is 1.3 because it needs to wait for memory in load and store instructions (without considering instruction fetches), the actual CPU would be able to execute only 769.23 MIPS.

To solve the problem of size and speed, the available options to build a memory system are listed below:

- Static RAM technology (SRAM). The basic cell is a flip-flop.
- Dynamic RAM technology (DRAM). The basic cell is a capacitor and requires refresh.
- Secondary storage (magnetic disks and solid state drives).

Among these three alternatives, SRAM is the fastest, with an access time close to the CPU clock cycle. DRAM memory is 5 times slower than SRAM. Magnetic storage is extremely slow, up to 10 million times slower. Finally, flash memory (SSD drives) is around 100 times faster than magnetic storage, but its access time is still far from those of SRAM and DRAM.

Considering access time, there is no doubt: the memory of the computer should be built with SRAM. However, there is an additional constraint that has not yet been

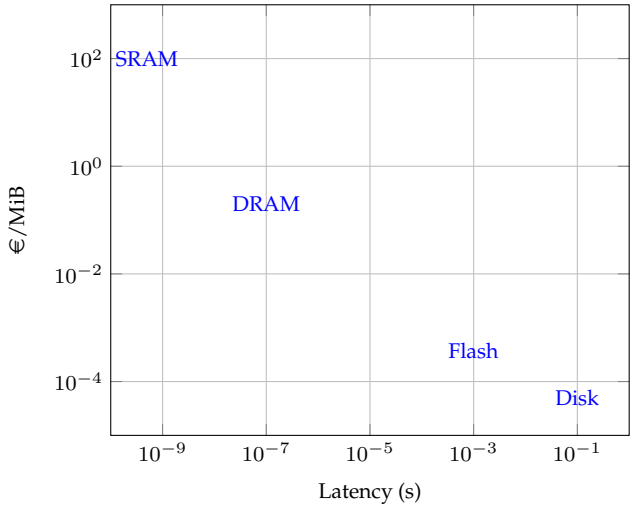


Figure 3.2: Comparison between cost and speed for different memory technologies

mentioned: the cost should be low. This is determined both by the physical size of the basic cell (amount of material needed to store a bit), and the energy consumed. Considering physical size the characteristics of the three technologies are the following:

- SRAM has the largest cell size, requiring at least 6 transistors and is the one that consumes the most energy.
- DRAM only needs one transistor to build the basic cell and requires less energy consumption.
- In magnetic storage the basic cell is made of a few elements of magnetic material, allowing very large storage densities. Cells are passive, that is, they do not consume energy to maintain information. Solid state drives are based on flash technology, which is also passive.

Figure 3.2 compares the two previous criteria in logarithmic scale: speed of the memory, expressed as latency, and cost, expressed in € per MiB.

As an example, representative technologies available in 2017 can be considered.

- SRAM.- SRAM, as currently used, is built into the processor and its cost could only calculated as the cost difference of two identical processors except for SRAM size. Intel Core i3 4170T and 4370T processors are considered with 3 MiB and 4 MiB of cache (SRAM), respectively. The access speed is approximately 1 nanosecond. Costs are approximately 135,€ and 163 €, respectively. This makes it possible to estimate the additional MiB of SRAM as 28 €.
- DRAM.- The Kingston memory module HX421C14FB/8 has a capacity of 8 GiB, access time of 6.6 nanoseconds and an approximated cost of 75 €. This results in a cost of 0.0092 €/MiB.

- Magnetic storage.- The Western Digital hard disk WD10EZEX (Serial ATA/600) of 1 TB has an average read access time of 8 milliseconds and a cost of approximately 50 €, resulting in a cost of 0.00005 €/MB.

The following alternatives would exist for building a memory system of 16 GiB for a computer considering the above costs:

- If built using SRAM, an extremely fast memory is obtained with an access time of 1 nanosecond and an approximate cost of 459 000 €.
- In contrast, a disk of 1 TB or more could be used for only 50 €. However, the access time would be approximately of 8 milliseconds, 8 million times slower than SRAM.
- If the memory system is built using DRAM, the approximate cost would be 150 € and it would be approximately 6 times slower than SRAM.

It can be concluded that there is no memory technology that solely satisfies all the desirable requirements for memory. SRAM is the fastest, but very expensive; magnetic storage is very economical and has a huge capacity, but it is extremely slow (the same happens with memory based on flash storage); DRAM is in the middle, not fast enough, not big enough, but not too expensive. How to build computer memory? The solution is to combine the previous technologies, taking the best of each one. This is what is known as a memory hierarchy.

3.2 Memory hierarchy

Three objectives are mainly considered when building the memory system of a computer:

- High speed.
- High capacity.
- Low cost.

The problem is that these objectives cannot be achieved at the same time with a single memory technology. The solution to the problem lies in combining fast and small memories with slow and large memories, in such a way that:

- Fast and small memories contain the data with the highest access probability.
- Slow and large memories contain data with less access probability.

This approach achieves high capacity, since it contains large memories. In addition, the memory system will be fast, because the fast memories are accessed in most cases.

The resulting size of combining memories in this way is not cumulative. The fast and small memories contain a copy of some data stored in the large memories so, from the point of view of the CPU, the size of the memory system is defined by the size of the largest memory.

Two conditions must be met in order to implement the previous solution:

- It is necessary to know data with higher probability of future access *a priori*, so that it can be stored in the small and fast memories.
- It is necessary for data with higher probability of future access not to be much, so it fits in the small and fast memories. Thus, future memory accesses will be served by fast memories in most cases.

Meeting these conditions is possible thanks to the principle of locality (or locality of reference) observed in the execution of programs by any CPU. This principle is divided in two:

- Principle of spatial locality. If a memory address is accessed at a given moment, it is likely that a nearby memory address will be accessed shortly thereafter. An example of spatial locality is when instructions are fetched during the sequential execution of a program. An example of spatial locality in the access to data appears when accessing a vector sequentially.
- Principle of temporal locality. If a memory address is accessed at a given moment, it is likely that the same memory address will be accessed shortly thereafter. An example of temporal locality in accesses to instructions appears in the execution of loops, where the same instruction is fetched repeatedly to execute it several times. An example of temporal locality in data accesses appears when accessing an iteration counter during the execution of a loop.

An analogy to the memory hierarchy is that of a sale business (the memory system). This business consists of a small store (small and fast memory) in the city center, where the customer (the CPU) is personally attended and a large warehouse in the city outskirts (large and slow memory). If the tastes of the customer (locality principle) are known, the small store can contain the products (memory positions) that the customer will probably request in order to serve them quickly. In addition, the business has a large number of products, since it has a large store.[2]

The combination of different memory technologies is carried out at different levels of capacity and speed, creating what is known as the memory hierarchy. Figure 3.3 shows a memory hierarchy using three levels. This example will be used throughout this chapter. The numerical values are just an example, as they depend on technology. For example, a MIPS64 CPU can access bytes, half-words, words or

[2]The ideal solution from the point of view of performance would be to have a large store in the city center, but that would be very expensive; the same occurs with the memory system.

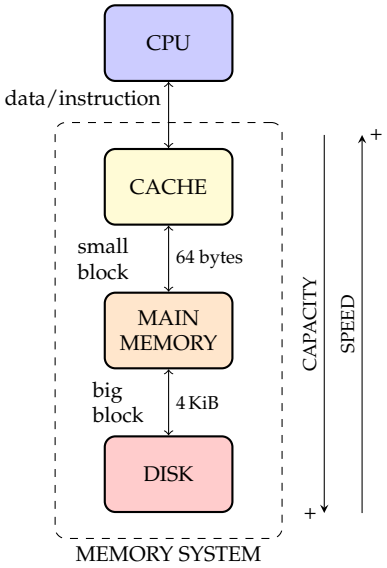


Figure 3.3: Memory hierarchy of three levels

double-words. In the case of using multiple issue, the width of the communication channel between the CPU and the cache must allow simultaneous reads of as many instructions as the issue width. In addition, pre-fetching techniques can also be implemented when using wide communication channels with caches, where several instructions are read at the same time to take advantage of the aforementioned spatial locality principle.

The cache is built using SRAM, which is very fast but with low capacity. In practice, the cache level can be in turn divided into sub-levels.

The main memory is built using DRAM, which is of medium speed with average capacity.

The furthest level of the hierarchy is usually built using magnetic disks or solid state drives, which are slow but have a large storage capacity. Only a part of the device is used.

The working principle of the memory hierarchy is very simple. When the CPU needs to read the content of a memory address, it accesses the cache level. If the content of the address is in the cache (cache hit), the CPU accesses it very quickly. This is the most usual situation, since most of the accesses of the CPU to the memory system are served immediately by the cache.

On the contrary, if the content of the required memory address is not in the cache (cache miss), a set of consecutive memory positions that includes the required address (or addresses if accessing more than one byte) is transferred from the main memory to the cache memory. The set of consecutive memory positions is called a block. Once the transfer is completed, the CPU accesses the cache. In general, any time a miss occurs in level N of the memory system, the block containing the required position is copied from level N+1 to level N and then the access is performed.

It should be noted that the unit of information transferred between two consecutive levels of the hierarchy is the block. The size of the block does not have to be the same between different levels of the hierarchy. For example, the size of the block used between the disk and the main memory is much larger than the size of the block used between the main memory and the cache.

Misses can be chained at different levels. For example, if the CPU needs to fetch data that is not in the cache, it tries to copy the block containing this data from the main memory to the cache. However, if the block is not in the main memory, another miss occurs in the main memory, and a larger block is copied from disk to the main memory. Then, the block from the main memory is copied to the cache.

Next, an example describes how the memory hierarchy is able to simultaneously provide low cost, high capacity and high speed. A memory system of 6 GiB is considered using three technologies with the following features:

- 4 MiB of SRAM used as a cache with access time $t_c = 1$ nanosecond.
- 3 GiB of DRAM used as the main memory with access time $t_p = 5$ nanoseconds per byte.
- 6 GiB of disk with access time $t_c = 11$ milliseconds, regardless of the size of the disk block.[3]

The size of the block between the cache and the main memory, B_{cp}, is 64 bytes.

The hit rate when the CPU accesses the cache, A_c, is 0.999, that is, 99.9%. The hit rate when accessing the main memory, A_p, is 0.9999999, that is, 99.99999%.

The cost of SRAM is 28 € per MiB, the cost of DRAM is 0.0092 € per MiB and the cost of the disk is 0.00005 € per MB.

1. What is the cost of the memory system?
2. What is the average read time of the memory system tr_{cpd}?

The cost is easily calculated considering the amount of memory of each type:

$$\text{cost} = 4 \times 28 + 0.0092 \times 3072 + 0.00005 \times 6144 \times \frac{2^{20}}{10^6} = 140.58\,€$$

As for the average read time of the memory system, it is calculated considering three possibilities: the data is in the cache, it is in main memory but not in the cache, or it is solely in the disk. The cases where the data does not exist or is inaccessible are not considered.

In the first case, a cache hit occurs when the data is in the cache and the read time is equal to the average access time of the cache.

[3] In practice, the block size is around 4 KiB. The penalty to access the first byte is higher than the others. In fact, this time is the one that mostly provides the access time of the block.

In the second case, the data is in the main memory but not in the cache, so a cache miss occurs. As a result, a complete block is fetched from the main memory to the cache. Although the transfer time of the block from the main memory to the cache depends on the width of the communication channel between both memories among other factors, it will be estimated proportionally to the number of bytes to be transferred for simplicity.

In the third case, the data is solely in the disk, which implies a cache miss and another miss in the main memory occur. A block would be copied from the disk to the main memory, and then a block would be copied from the main memory to the cache. For simplicity, it is assumed that the transfer of data from the disk to the main memory can be performed simultaneously with the transfer from the main memory to the cache. Similarly, only the time needed to copy the block from the main memory to the cache is considered, without including the subsequent read of the cache for the required data, that causes a cache hit this time.

The mathematical expressions related to each of the above cases is shown next:

$$tr_{cpd} = \begin{cases} \text{cache hit} & \Rightarrow A_c \cdot t_c \\ \text{cache miss and hit in M.M.} & \Rightarrow (1 - A_c) \times A_p \times (t_p \cdot B_{cp}) \\ \text{cache miss and miss in M.M.} & \Rightarrow (1 - A_c) \times (1 - A_p) \times t_d \end{cases}$$

This can also be expressed as:

$$tr_{cpd} = A_c \cdot t_c + (1 - A_c) \times [A_p \cdot t_p \cdot B_{cp} + (1 - A_p) \cdot t_d]$$

where:

tr_{cpd} is the average read time of the memory system.

A_c is the cache hit rate.

t_c is the access time of the cache.

A_p is the hit rate of the main memory in case of a cache miss.

t_p is the access time to a single byte in the main memory.

B_{cp} is the number of bytes to be transferred from the main memory to the cache for a cache miss.

$t_p \times B_{cp}$ is the time required to transfer a block from the main memory to the cache.

t_d is the time required to transfer a block from the disk to the main memory.

Substituting the value for each variable in the previous expression results in $tr_{cpd} = 1.32\,\text{ns}$.

As can be seen, the result is a memory system with an average read time similar to that of the cache, with a capacity of 6 GiB and a reasonable cost.

The key to obtain an average read time close to that of the cache memory lies in the principle of locality, since it ensures high hit rates when accessing the cache and the main memory. If the cache hit rate were 0.9 and the main memory hit rate 0.999, the average read time would be 1.13 microseconds, even much higher than the access time of the main memory.

It must be noted that these expressions only apply to read accesses. In the case of write accesses, write policies used in each hierarchy level must considered. Concretely, cache write policies are described in section 3.3.4.

3.3 Cache memory

This section introduces the operation of the cache memory and its communication with main memory. In addition, an example of organization of the cache in the x86 architecture is presented.

3.3.1 Preliminary concepts

The cache memory is the fastest memory in the system. Currently, it is implemented inside the processor chip using SRAM, which results in a high cost.

The cache is organized into lines that contain blocks, all of them containing the same number of bytes. These bytes are located in consecutive memory addresses. Every time the CPU tries to access a memory address that is not in any line of the cache, a cache miss occurs. The consequence is that the access is suspended momentarily, the block containing the address (or addresses if the access is to several bytes) is copied from main memory to the cache memory, and then the access is resumed. This means that the instruction that is accessing is temporarily stalled waiting for the cache. Therefore, a cache miss increases the CPI, so performance is reduced.

From the point of view of the interaction with the cache memory, main memory (more specifically, the memory address space) can be seen as a large store of blocks of the same size as those that can be stored in cache lines. However, this does not mean that the memory address space is organized in blocks.

For example, a cache of 32 bytes organized in lines of 4 bytes will store 8 blocks. If the size of the memory address space is 256 bytes, the cache will perceive this address space as 64 blocks of 4 bytes each, as shown in figure 3.4.

A memory address issued by the CPU is divided into two fields:

- The block number.
- The offset in the block.

The memory block where the issued address is included can be obtained taking into account that the least significant bits of the address are used to define the offset of the byte within the block and, therefore, the rest of the bits provide the memory block number. This is also shown in figure 3.4.

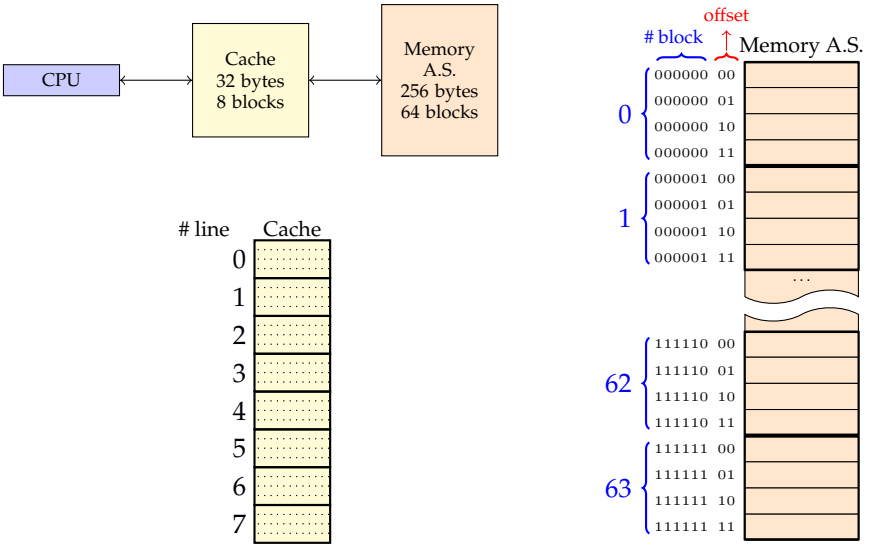


Figure 3.4: Memory address space and organization of the cache memory in lines

The cache memory includes a controller that manages reads and writes in the cache and checks whether a memory access operation is a hit or a miss.

In general, the operation of the cache memory is defined by the following characteristics:

- Placement policy. It establishes to which cache line or lines each block of memory can be copied (placed).
- Replacement policy. It establishes which line of the cache memory should be replaced to hold a memory block when a cache miss occurs. This only applies to cases where the memory block can be placed in more than one cache line.
- Write policy. It establishes the relationship between the writes in the cache and the next level of the memory hierarchy. The fundamental objective of this policy is to ensure consistency between data of both memory levels, affecting the performance of the system as low as possible.

3.3.2 Placement policies

Basically, there are three placement policies:

- Direct mapped cache
- Fully associative cache
- Set associative cache

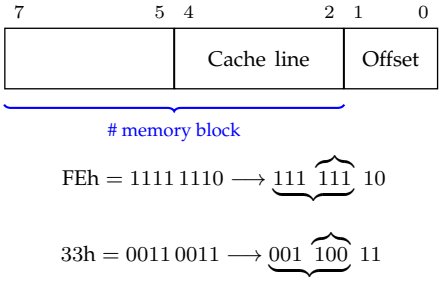


Figure 3.5: Examples of obtaining the memory block number and the cache line in a direct mapped cache

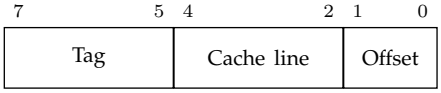


Figure 3.6: Fields in a memory address when using a direct mapped cache

Direct mapped cache

The direct mapped cache structure is the simplest. Each memory block is always mapped to the same cache line according to the following expression:

Cache line = (Memory block) MOD (Number of lines in the cache)

In practice, the number of lines in the cache is a power of two (2^x), so the cache line is obtained by taking the least significant x bits of the binary number that represents the memory block. Figure 3.5 shows examples about the calculation of the memory block number and the cache line given the organization shown in figure 3.4.

When a memory block is copied to a line in the cache memory, a set of bits that uniquely identifies the memory block is written together with the block. This set of bits is called a tag. The memory address is then divided into fields, as indicated in figure 3.6 for a direct mapped cache.

Each line of the cache also includes a valid bit (v) that indicates whether the block stored in the line is valid. For example, all the lines of the cache memory are marked as not valid (v=0) during the initialization process of the computer.

Figure 3.7 shows the internal structure of a direct mapped cache. The memory address is divided into fields to initially get the line number. Then, the tag field of the address is compared to the tag stored in the cache line. If they match, the address offset field indicates the byte of the line to be accessed. Figures 3.8 to 3.16 show the operation of this type of cache when the CPU performs several reads of one byte.

Fully associative cache

A direct mapped cache is very simple to implement. Its biggest drawback comes from the strict mapping of memory blocks to cache lines, because there is a single

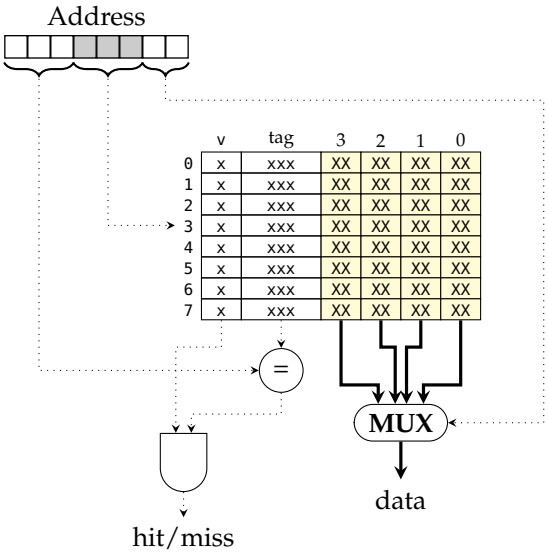


Figure 3.7: Structure of a direct mapped cache

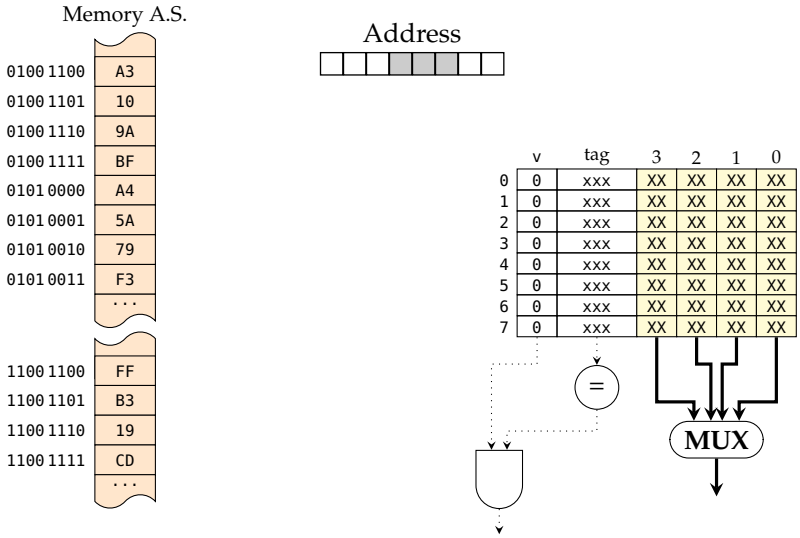


Figure 3.8: Direct mapped cache: initial state with an empty cache

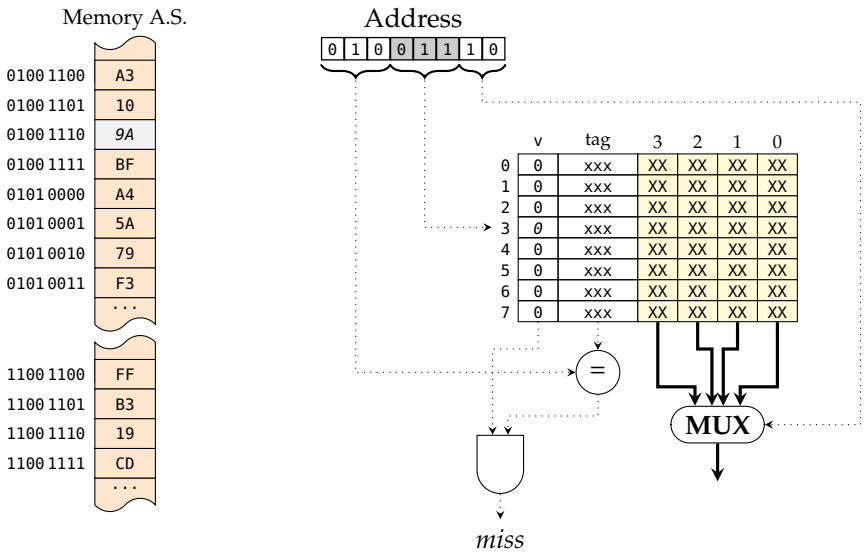


Figure 3.9: Direct mapped cache: read request for a byte at address 4Eh, which leads to a cache miss

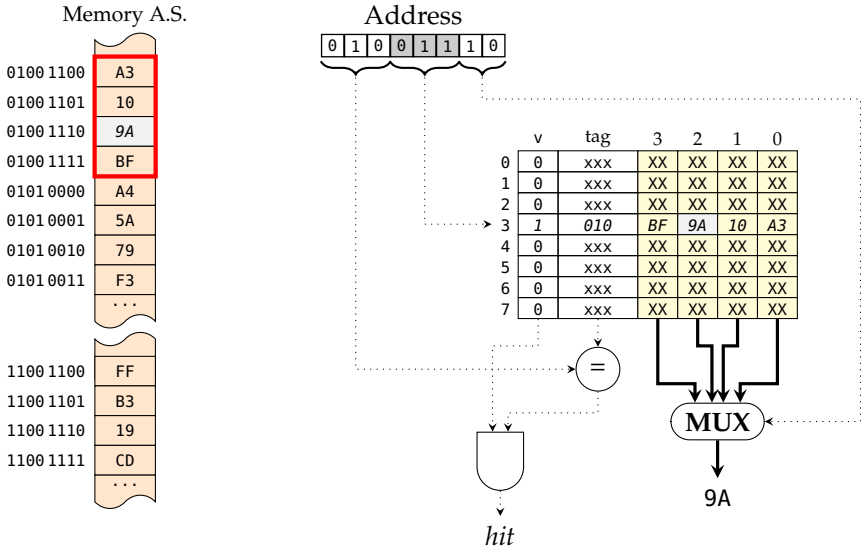


Figure 3.10: Direct mapped cache: cache line 3 is loaded and the cache provides the requested byte

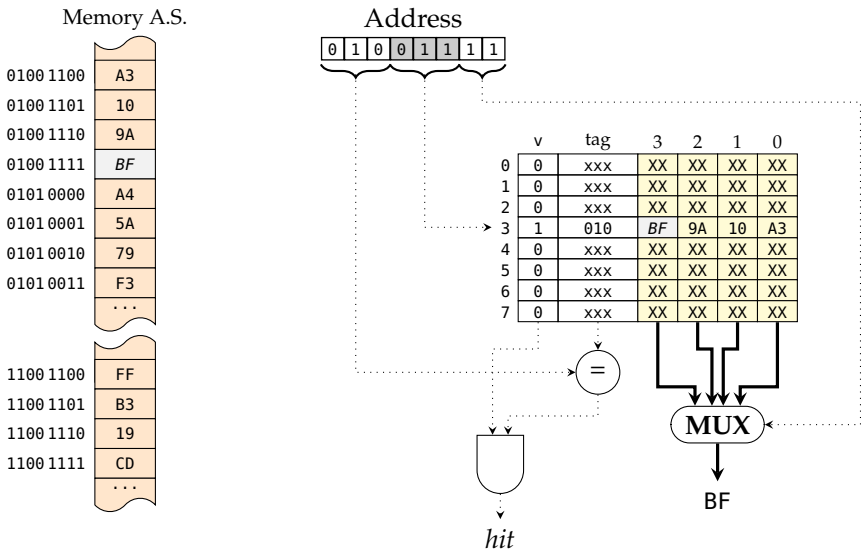


Figure 3.11: Direct mapped cache: read request for a byte at address 4Fh, which leads to a cache hit

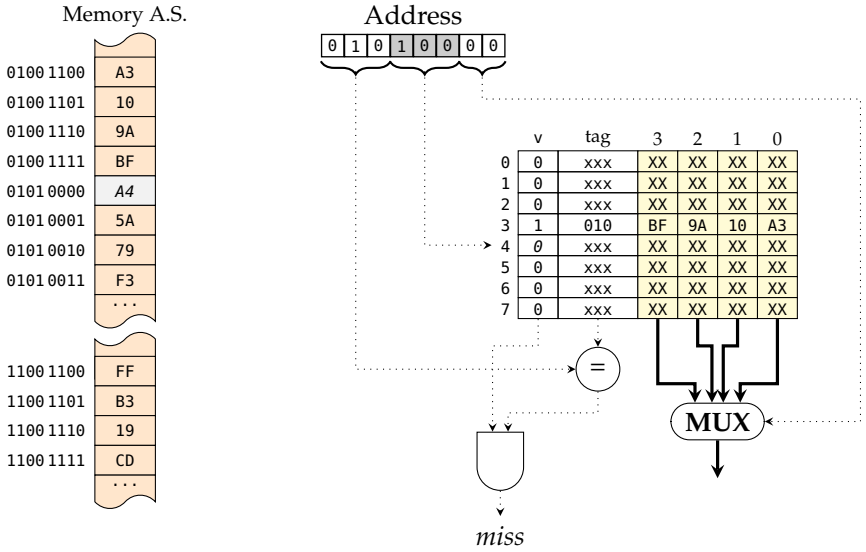


Figure 3.12: Direct mapped cache: read request for a byte at address 50h, which leads to a cache miss

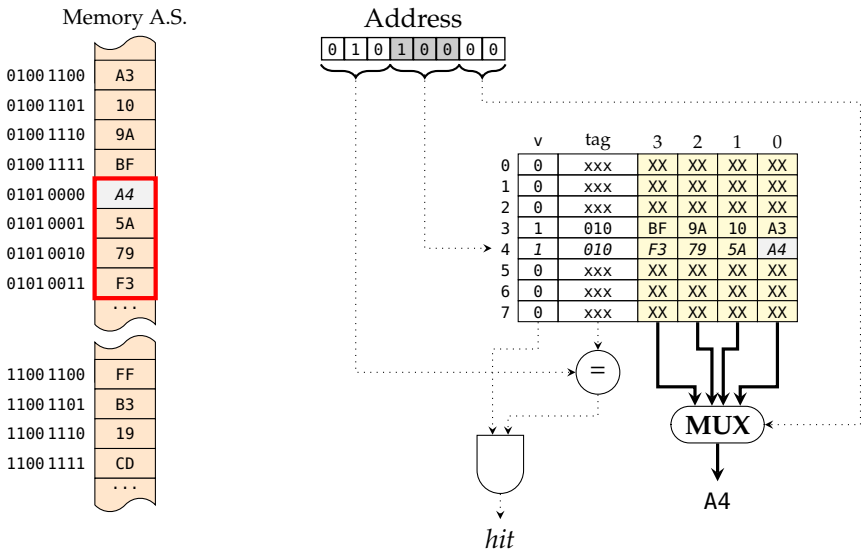


Figure 3.13: Direct mapped cache: cache line 4 is loaded and the cache provides the requested byte

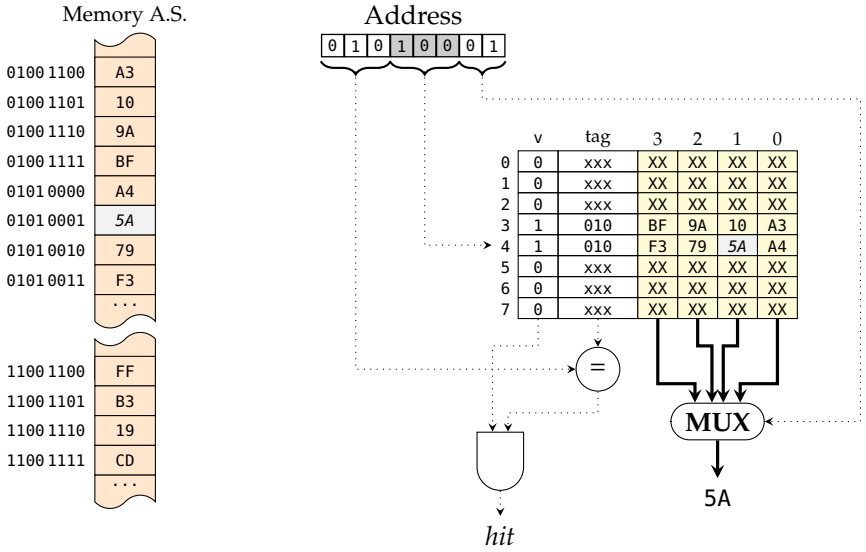


Figure 3.14: Direct mapped cache: read request for a byte at address 51h, which leads to a cache hit

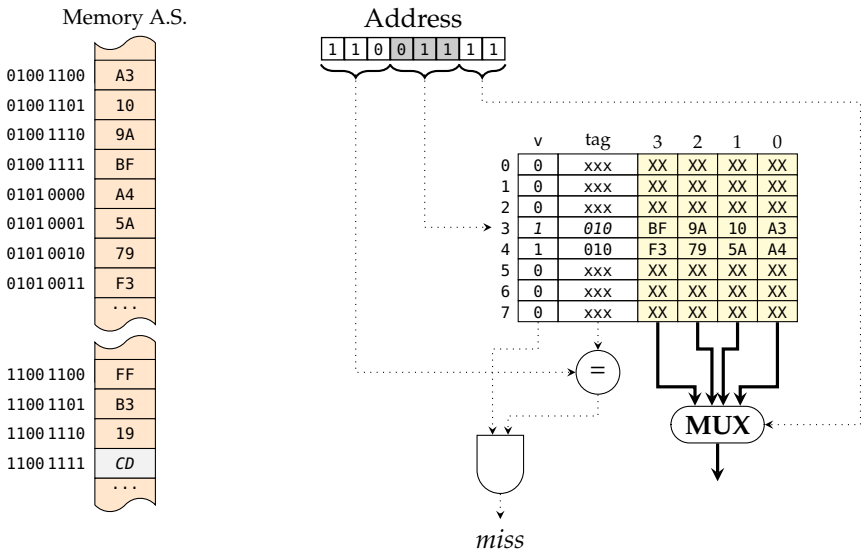


Figure 3.15: Direct mapped cache: read request for a byte at address CFh, which leads to a cache miss

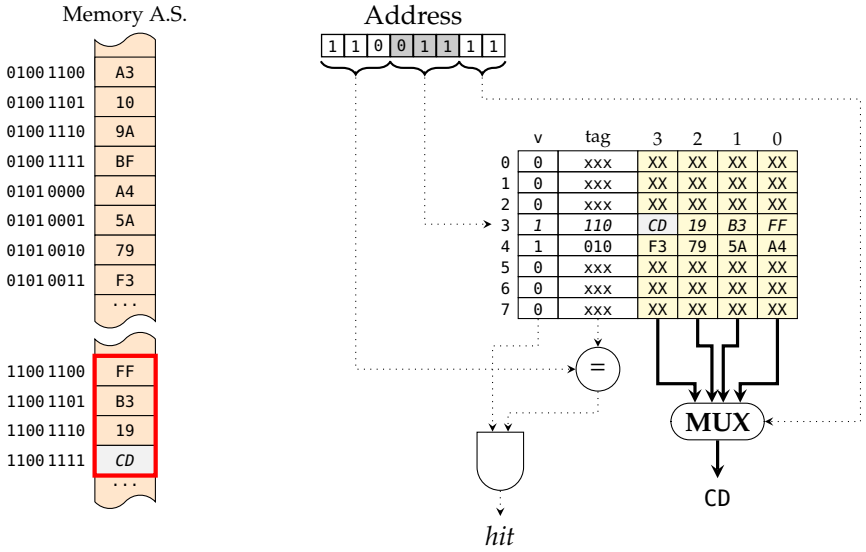


Figure 3.16: Direct mapped cache: replacement of block in line 3 and the cache provides the requested byte

Figure 3.17: Fields in a memory address when using a fully associative cache

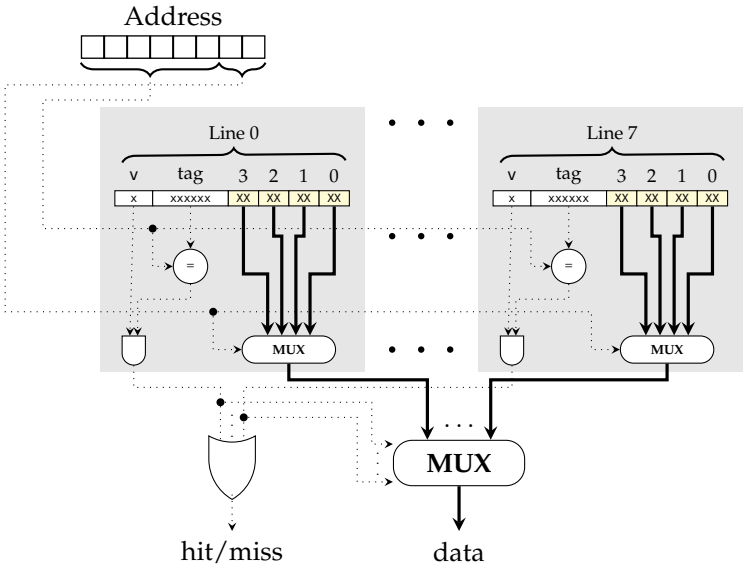


Figure 3.18: Structure of a fully associative cache

possibility. Each block of memory can only be placed in one cache line. If the CPU accesses two memory blocks mapped to the same cache line very frequently, there will be frequent cache misses, which reduces CPI and, consequently, performance.

To alleviate this issue, a fully associative cache is totally flexible: a block of memory can be mapped to any cache line. In this case, the tag included in the cache line matches the memory block number. Figure 3.17 illustrates the division into fields of a memory address under this placement policy.

Figure 3.18 shows the internal structure of a fully associative cache.

Set associative cache

A fully associative cache provides optimal performance, as it imposes no restrictions when mapping memory blocks to cache lines. However, it presents a serious drawback: the implementation cost is unaffordable for large caches, basically due to the high number of comparators that it uses. In this case one comparator per line is needed, when a direct mapped cache only needs one for the entire cache. A set associative cache is a trade-off between a direct mapped cache and a fully associative cache, as shown in figure 3.19.

A set associative cache is divided into sets, each of which contains a fixed number of lines. Each block of memory can be mapped to any of the lines of a single set. The

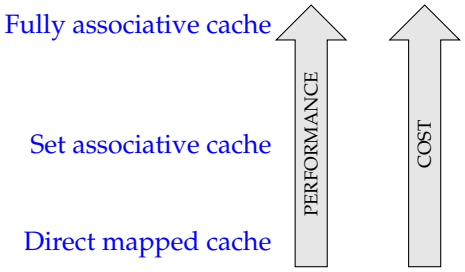


Figure 3.19: Comparison between placement policies

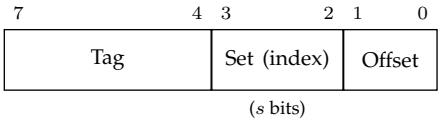


Figure 3.20: Fields in a memory address when using a set associative cache

number of lines that make up a set is called the number of ways. The number of sets in the cache is chosen as a power of two, 2^s, where s is the number of bits used to represent the set number. Using this type of cache, the s least significant bits of the memory block number provide the set, and the remaining bits the tag, as illustrated in figure 3.20.

Therefore, direct mapped caches and fully associative caches are set associative caches with different levels of set associativity. A direct mapped cache is a one-way set associative cache, while a fully associative cache is a set associative cache with one single set containing all the lines in the cache. In this case, the number of ways matches the number of lines in the cache, so the associativity is maximum.

Figure 3.21 shows how block mapping is performed in a set associative cache.

Figure 3.22 shows the organization of a set associative cache. The figure shows how the memory address is divided into fields to initially obtain the set number. Next, the tag field of the address is compared to the tags stored in all the ways of the set. If a match occurs, the address offset field indicates the byte of the line to be accessed.

Figures 3.23 to 3.25 show the operation of this type of cache when the CPU performs several reads of one byte.

3.3.3 Replacement policies

Every time there is a cache miss, a block is copied from the memory address space to a cache line, possibly replacing a previously cached block.

In a direct mapped cache, there is no doubt about which block to be replaced. However, in the case of a set associative cache, there are several candidate blocks to be replaced. The replacement policy determines which particular block is replaced. Two heuristics are considered:

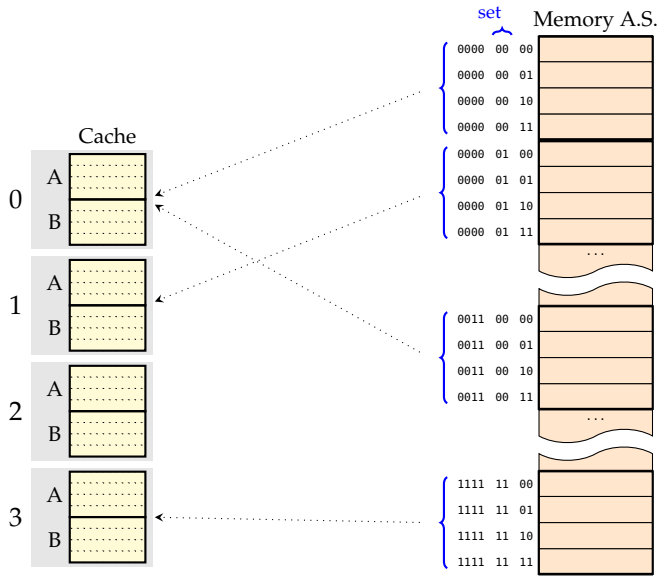


Figure 3.21: Block mapping in a set associative cache

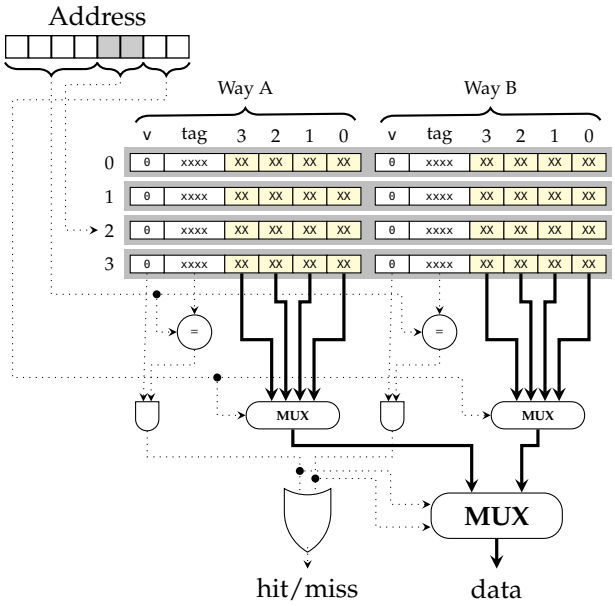


Figure 3.22: Structure of a set associative cache

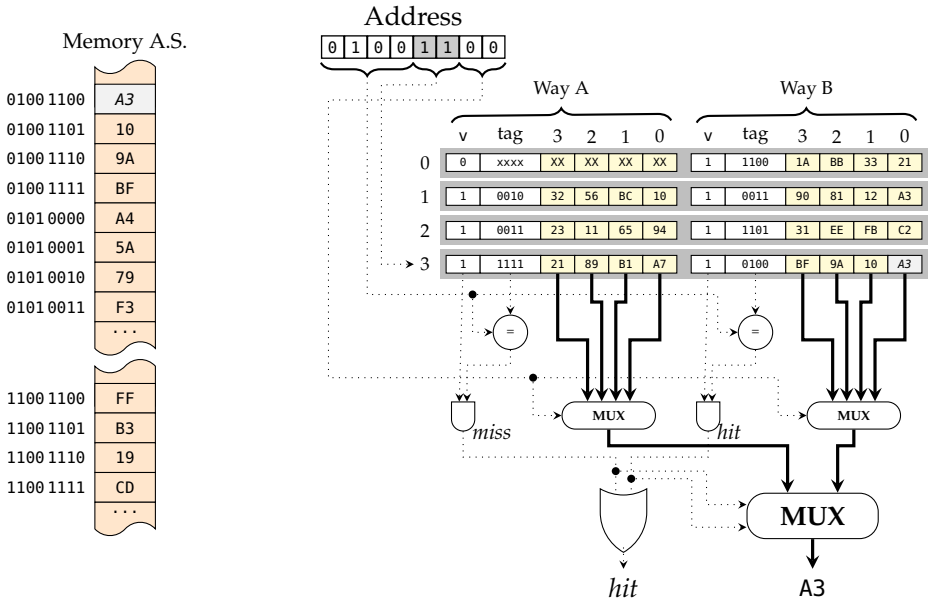


Figure 3.23: Set associative cache: read request for a byte at address 4Ch, which leads to a cache hit

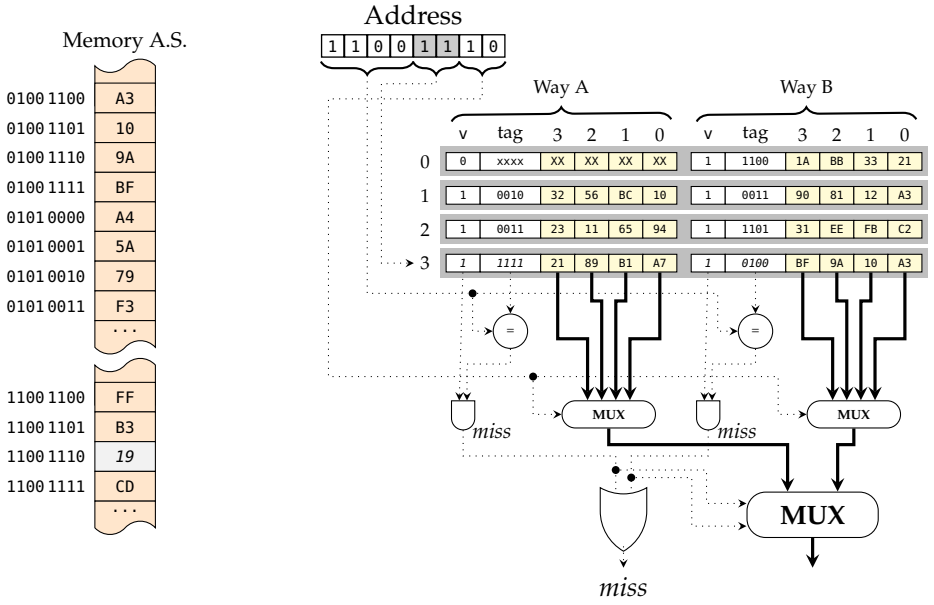


Figure 3.24: Set associative cache: read request for a byte at address CEh, which leads to a cache miss

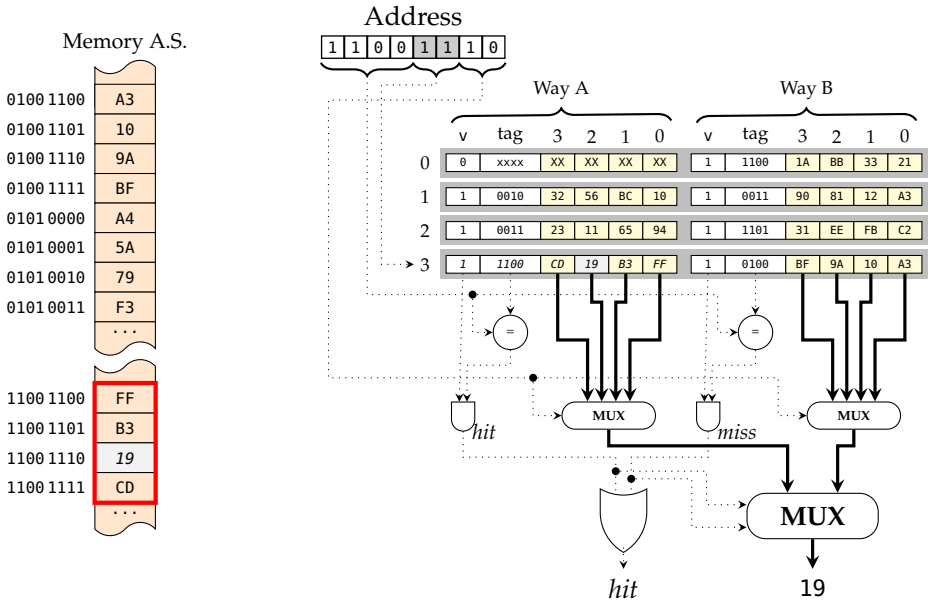


Figure 3.25: Set associative cache: replacement of the block in way A, set 3, and the cache provides the requested byte

- Least Recently Used (LRU). Each cache line stores additional bits that indicate the least recently accessed block, which is then selected for replacement.
- Random replacement. The block to be replaced is randomly chosen among the candidate blocks.

3.3.4 Write policies

There are two main write policies:

- Write-through.
- Write-back.

In a write-through cache, every write operation is performed in the cache and in the memory address space at the same time.

On the other hand, in a write-back cache data is initially only written to the cache. The data is written to the memory address space only when the block containing it is replaced in the cache. In a write-back cache there are coherence issues that will be addressed later in this chapter.

Write accesses, as read accesses, can result in cache hits or misses. Cache hits are handled as the write policy dictates: data is only written to the cache in the case of write-back caches, and it is written to the cache and the memory address space simultaneously in write-through caches.

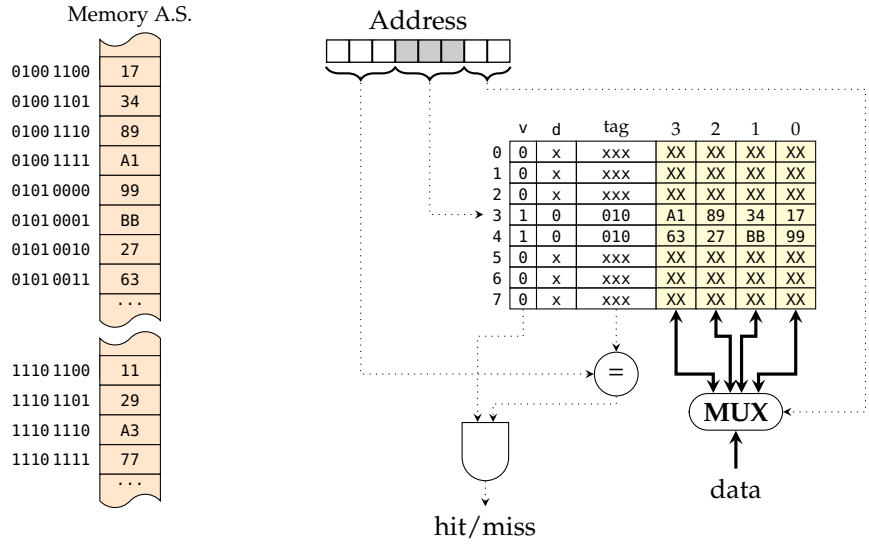


Figure 3.26: Operation of a write-back cache: initial state of the cache and the memory address space

However, the difference appears with write cache misses. In this case, there are two options: write allocate or no write allocate. When using write allocate, the behaviour for writings is identical to that for readings: the block is copied into cache memory before writing. With no write allocate, the block is not copied to the cache, data is only written in the memory address space, avoiding the cost of copying the entire block to the cache previously.

There are different variations, but the most common are write-back with write allocate and write-through with no write allocate.

The performance of a write-back cache is usually higher than the performance of a write-through cache, at the expense of greater hardware complexity for keeping data coherence.

Figures 3.26 to 3.30 show the operation of a write-back cache. When using this write policy, one more bit is associated with each cache line, the dirty bit (`d`), which is set every time the CPU writes to the line.

Write operations can generate coherence issues between different copies of the information stored in several levels of the memory hierarchy, as will be analyzed in section 3.3.6.

3.3.5 Cache memory organization

There are two determining factors in the organization of a cache memory:

- Number of cache levels.
- Separation into data and instruction caches.

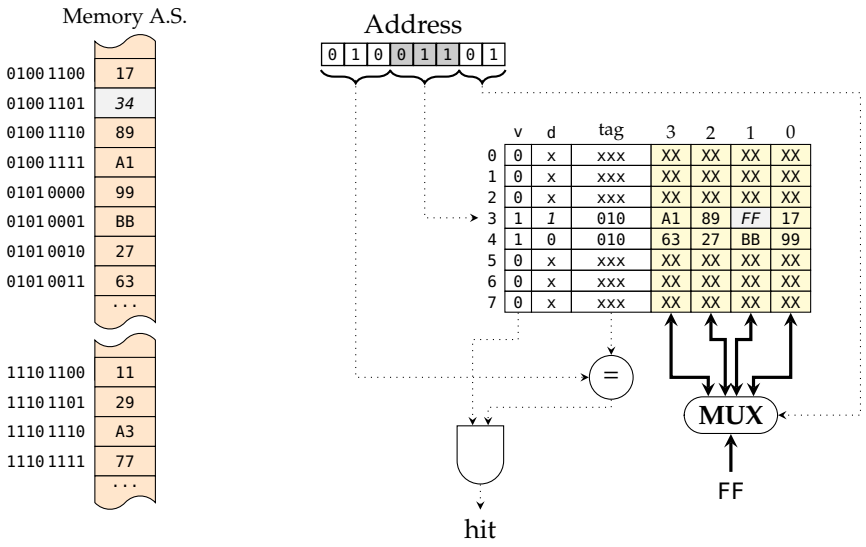


Figure 3.27: Operation of a write-back cache: Request to write one byte at address 4Dh; cache hit, the block is marked as dirty

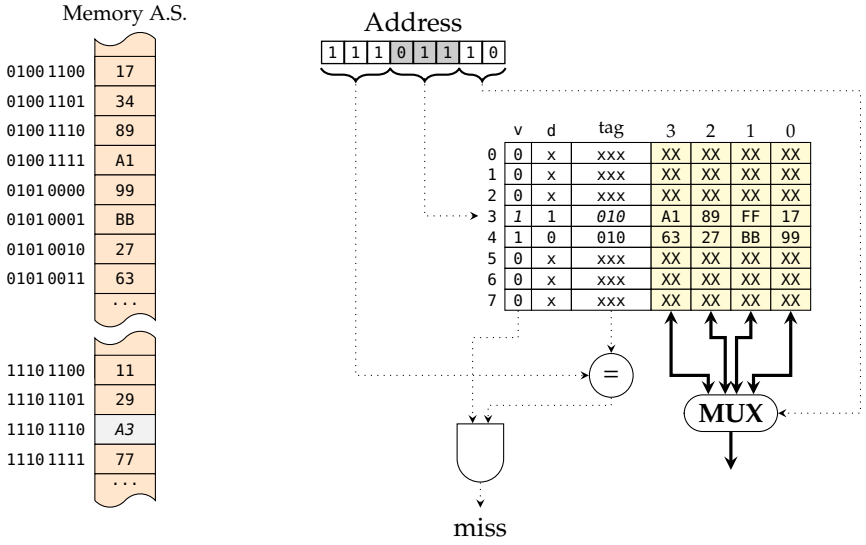


Figure 3.28: Operation of a write-back cache: Request to read one byte at address EEh, which results in a cache miss

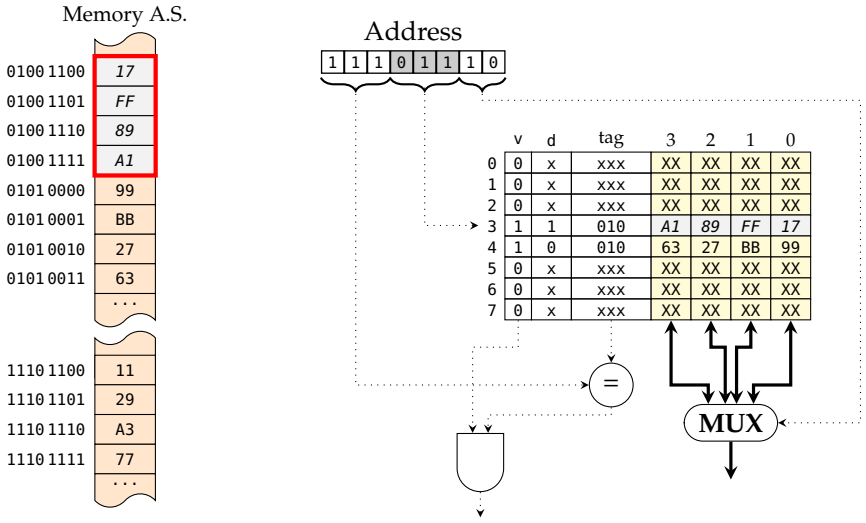


Figure 3.29: Operation of a write-back cache: updating a dirty block in the memory address space

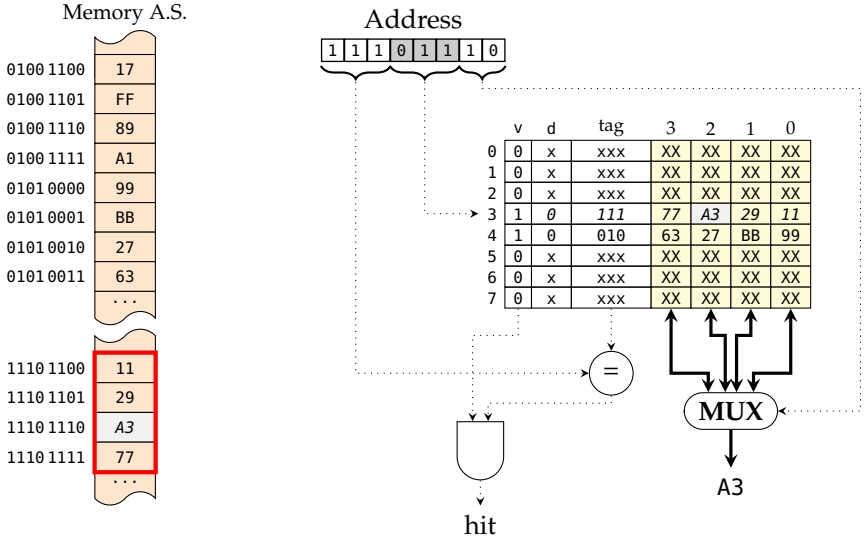


Figure 3.30: Operation of a write-back cache: replacement of block at line 3 and the cache provides the requested byte

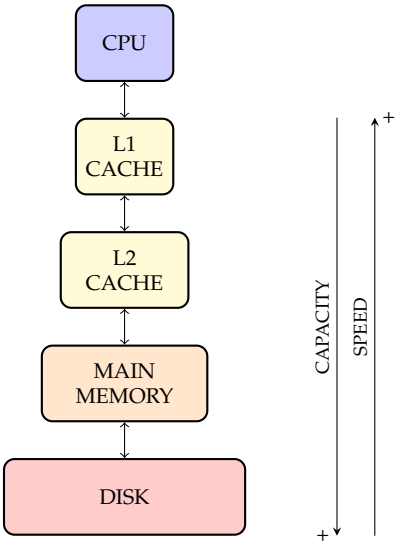


Figure 3.31: Memory hierarchy with two cache levels

Initially when caches were introduced, the most common approach was to use a single cache memory. However, the most common approach nowadays is to have two or three cache levels in order to increase hit rate and reduce latency. These levels are called L1, L2, etc., where the L1 level is the closest to the CPU and the smallest, and the last level cache (LLC) refers to the furthest-level cache, usually shared by all the cores. Figure 3.31 shows a memory hierarchy with several levels of cache. In this example, L2 cache is LLC.

L2 and higher caches cut down the performance penalty from a cache miss in L1. The latency difference between the L1 cache and the main memory is too high. The L2 cache is of greater capacity than the L1 cache, so it is also slower, but still much faster than main memory. In theory, the more cache levels between the CPU and main memory, the better performance of the memory system, but at a higher cost.

As previously mentioned, the MIPS64 microarchitecture incorporates independent memories for instructions and data. This means that a MIPS64 CPU includes two L1 caches: one for instructions and another for data, as opposed to a single cache for data and instructions. It could occur that the data cache had free lines, while the instructions cache was generating cache misses because it is full. On the contrary, with a unified cache, the cached blocks are automatically divided between data and instructions, so that if fewer blocks of data and more of instructions are needed, they are distributed efficiently and transparently between the lines of the cache.

Originally, cache memories used to be unified, since they obtain a better performance in terms of hit rate. However, current L1 caches are usually divided due to the high degree of parallelism of CPUs. The split cache makes a greater degree of parallelism possible, since a stage of the pipelined CPU can write data to memory while another stage can simultaneously fetch an instruction code.

3.3.6 Cache coherence

The CPU always works with a copy of information stored in the memory address space. This copy resides in the cache. If other components of the computer can access the memory address space, then coherence problems appear.

The principle of cache coherence states that reads by a CPU at a memory address must match the last value written at that memory address by any other component of the computer.

Single-level cache coherence with a single CPU

Cache coherence issues in single-CPU systems appear when peripheral interfaces are mapped in the memory address space. For example, if the interface of a keyboard is mapped in the memory address space and the user presses a key, the key code is stored in one of the positions of the memory address space. The problem is that if the memory block containing the interface address is cached, the cache memory is not aware of this writing and, therefore, does not contain updated data. If the CPU accesses the keyboard interface to obtain the code of the key pressed, the key code is received from the cache and is incorrect.

The solution in this case is simple: memory addresses mapped to peripheral interfaces are marked as non-cacheable. The cache memory directly forwards accesses to any of these positions to the memory address space.

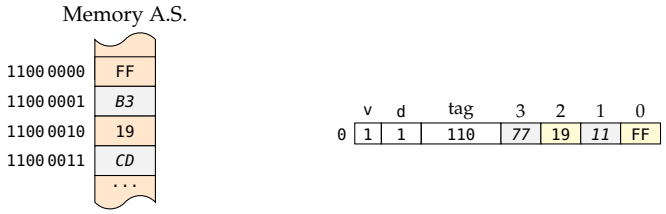
Another more complex coherence issue appears when there are peripheral interfaces with DMA (Direct Memory Access) capability.[4] For example, the disk interface writes a requested data sector into main memory. This could require writing 1 KiB in contiguous memory positions. Marking that area as non-cacheable solves coherence issues but causes another problem: performance is considerably reduced, as it requires accessing the memory address space instead of the cache memory. Accessing a few bytes, such as in the case of the keyboard interface is not problematic, but accessing large portions of memory incurs in a serious performance penalty.

In the case of peripheral interfaces with DMA capability, the following coherence issues appear:

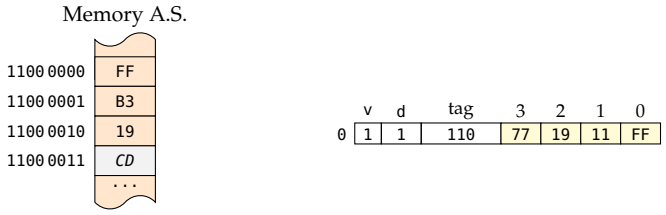
- A memory block that had been previously cached has been modified by a peripheral. Reading the block will provide non-updated data, as the CPU reads from the cache memory.
- The CPU modifies a cache line using the write-back policy. If a peripheral reads this block from main memory using DMA before replacing it in the cache, the data will not be valid, as it is not updated. This issue does not appear when the write-through policy is used.

The described cache coherence issues are usually solved using the snooping technique. Using this technique, the cache controller continuously observes the read and write control lines of the memory address space, as well as the address lines. In the

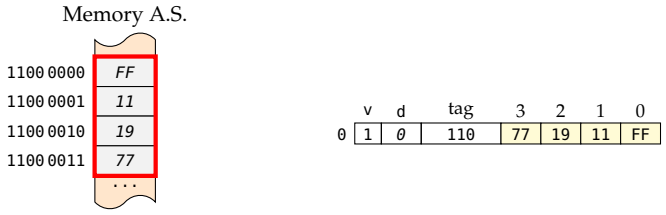
[4] The DMA technique is studied in the input/output chapter.



(a) Initial state: the dirty block in the cache line differs from the memory address space



(b) Peripheral performs read on address C3h. The controller detects this address belongs to a modified line and stops the read operation



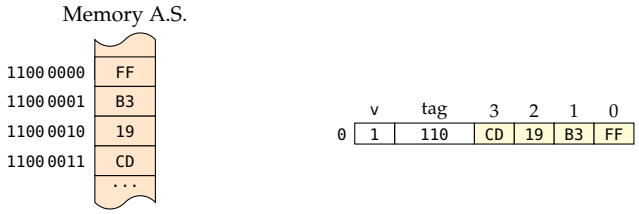
(c) The controller updates the block in the memory address space (bit d=0), then resumes the read operation

Figure 3.32: Cache coherence when a peripheral interface performs a read operation and using a write-back cache

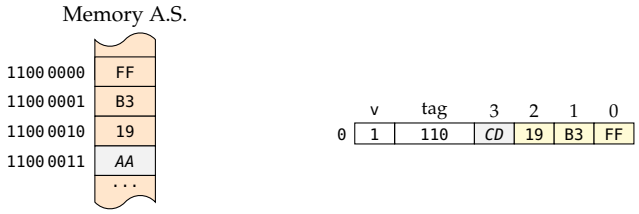
first case, when it detects that a peripheral is writing a memory block that is cached, it invalidates the cache line where the block is located by resetting the valid bit. In the second case, the cache controller temporarily stops the memory read by the peripheral, updates the memory block that was cached and then resumes the read operation by the peripheral.

Figures 3.32 to 3.34 show the three cases where coherence issues between cache memory and the memory address space appear.

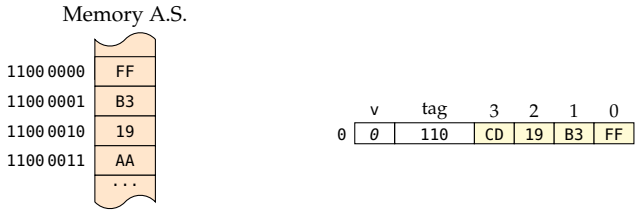
The example issue shown in figure 3.32 for a write-back cache also appears for a write-through cache (figure 3.33). However, in the former there is a coherence issue that does not appear for the write-through cache when the interface of a DMA-capable peripheral writes to a cached memory block marked as dirty. The solution is shown in figure 3.34.



(a) Initial state

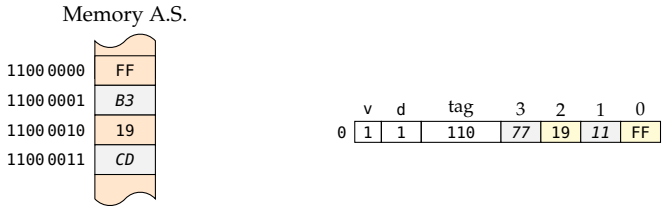


(b) Peripheral writes data AAh at address C3h

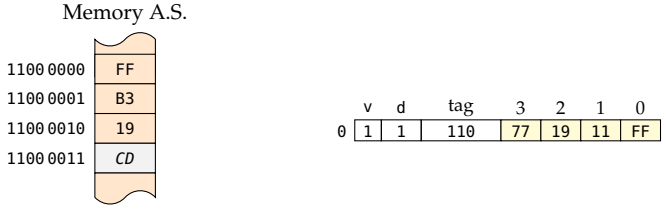


(c) The controller detects that this address is cached and invalidates the line

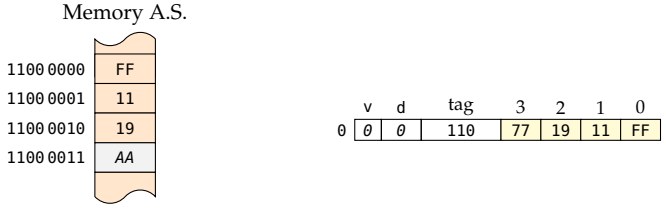
Figure 3.33: Cache coherence when a peripheral interface performs a write operation and using a write-through cache



(a) Initial state: the dirty block in the cache line differs from the memory address space



(b) Peripheral initiates write operation of AAh at address C3h. The controller detects this address belongs to a modified line and stops the write operation



(c) The controller updates the block in the memory address space (bit d=0), invalidates the line (bit v=0) and then resumes the write operation

Figure 3.34: Cache coherence when a peripheral interface performs a write operation and using a write-back cache

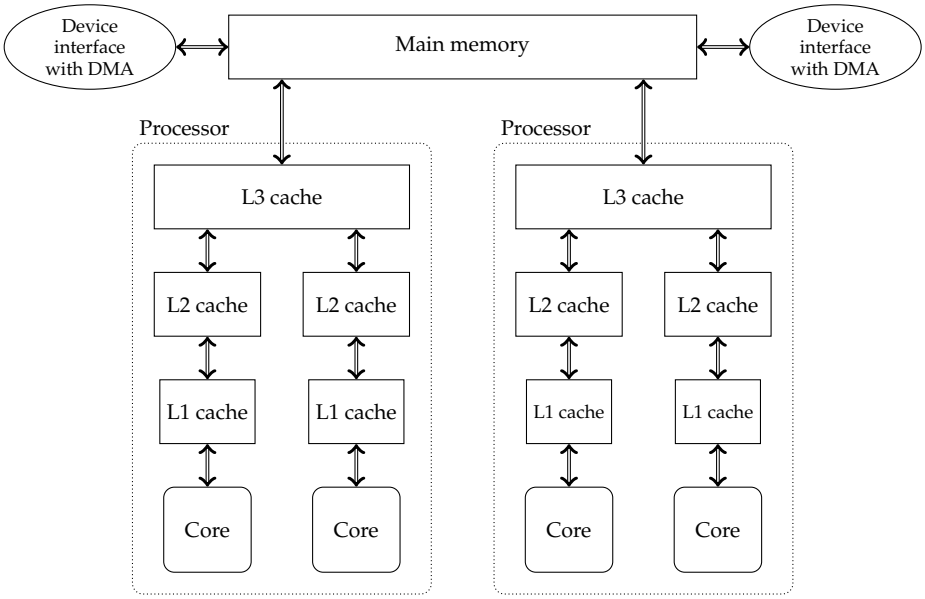


Figure 3.35: Example of multi-processor with several cores and levels of cache per processor

Multi-level cache coherence with multiple CPUs

Previous sections described how coherence issues come from peripheral interfaces with DMA capability in single-level, single-CPU systems. In many cases, computers have several processors each of which incorporates several cores with various cache levels. Figure 3.35 shows an example of this type of system.

For the sake of simplicity, interfaces with DMA capacity will not be considered, as they can be assimilated to CPUs without cache in terms of cache coherence.

In this scenario, the memory system as a whole is accessed by multiple CPUs to perform reads and writes. In practice, due to performance issues, a write-back policy is used, so that the writing performed by a CPU in a memory address appears in its L1 cache and does not migrate to further levels of cache or other system memories. The written data is not directly visible to other CPUs even they are co-located in the same processor. The situation can be even worse if the written memory address has already been previously written by another CPU.

These problems appear when programs share memory, which is the most common approach in systems with several CPUs. Given all this complexity, more sophisticated coherence methods are needed than those described for a single CPU.

Maintaining cache coherence influences the performance of the system, since memory access is critical for the execution of the programs. The general strategy is to prevent these methods from interfering the system when accessing non-shared memory addresses, and reducing their impact when accessing memory addresses shared with other CPUs.

The fundamental problem comes from memory writes, as memory reads do not generate cache coherence issues. When a CPU writes to an address that is in the L1 cache, the address could also be in L2 and L3 caches of the same CPU, or at any of the cache levels of other CPUs. There are two alternatives:

- **Write-update**. The data written to the L1 cache of a CPU is updated in all caches of the system where the address is cached, including other caches of the same CPU and those of the rest of the CPUs.
- **Write-invalidate**. The data is written to the L1 cache of the CPU and all the lines with the cached address are marked as not valid in the rest of the caches of the system. Thus, a cache miss will occur and an updated block will be received when accessing the address in other caches.

In practice, the write-invalidate protocol is used, as it is much faster to invalidate cache lines than to update them. In addition, several consecutive writes are often made before a read, so the penalty of maintaining coherence is even lower.

Whatever the chosen alternative, write-update or write-invalidate, any read or write operation can provoke changes in multiple caches of the system and main memory. For this reason, information must be transmitted to the rest of the caches of the CPU and the rest of CPUs for updating their states in order to maintain coherence. There are two basic ways to transmit coherence information:

- Snooping. Any read or write operation in a cache is visible to all system caches and main memory. Thus, if a CPU writes in its L1 cache, all the system caches observe the writing address and if it is included in any of its lines they mark the line as invalid. *A priori* this approach may be seen as inefficient, considering the high number of caches that coexist in a multi-processor and multicore system. However, multiple optimizations make this approach feasible. For example, when a cache line is cached only in the L1 cache of a CPU, it can be read or written without transmitting coherence information to other caches. This is also the most common case, since the vast majority of memory accesses are made to non-shared addresses.
- Directory. There is a hardware device that manages the coherence of the memory system and stores coherence information for all the caches. All read or write operations on caches or main memory that may affect coherence require a prior query or modification of the directory.

In systems with few CPUs, snooping is the most common method, but the directory method is used in large-scale multi-processor systems.

MESI protocol

The MESI coherence protocol is the basis of many current coherence protocols.[5] This section illustrates the main operation in systems with several CPUs without providing all the implementation details.

The MESI coherence protocol is a write-invalidate protocol commonly used with snooping. In this protocol, a cache line can be in one of these four coherence states:

[5]For example, Intel uses a variation called MESIF, while AMD uses another variation called MOESI.

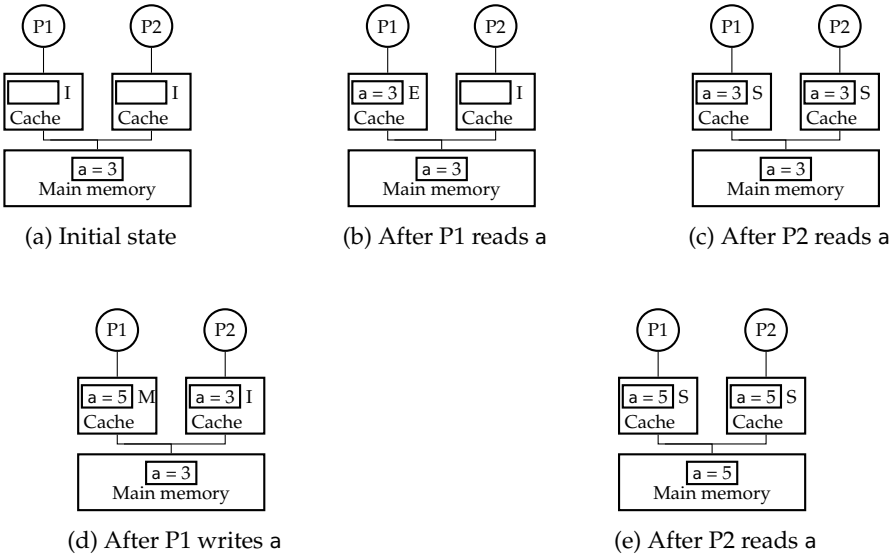


Figure 3.36: Example of operation of the MESI protocol

- Modified (M). The cache line is valid, it has been modified and, therefore, is not coherent with main memory. The block contained in the line is not stored in other cache memories.
- Exclusive (E). The line is valid, it has not been modified and, therefore, is coherent with main memory. The block contained in the line is not stored in other cache memories.
- Shared (S). The line is valid, it has not been modified and, therefore, is coherent with main memory. The block contained in the line can also be found in other cache memories.
- Invalid (I). The cache line is invalid. It does not contain any block.

Every time a CPU performs a memory operation (read or write) in the cache, it can change the MESI status of a line in its L1 cache and any other system cache.

Figure 3.36 illustrates the state changes of the cache lines where a shared variable a is located. The system contains two CPUs, P1 and P2, each with a single-level cache. Initially, the block containing the variable is not cached (figure 3.36a).

When P1 first reads the variable, a cache miss occurs and the memory block that contains the variable is copied to a cache line (figure 3.36b). Because the memory block is not in the cache of the other CPU, the line is marked as exclusive (E). The next time P1 reads the variable, a cache hit occurs and it is not necessary to update the coherence state of any system cache.

A subsequent read of the variable by P2 will provoke a cache miss, but in this case in the cache of this CPU. However, unlike the previous case, the block is in the P1 cache in addition to main memory. The block is copied to the P2 cache from

main memory and the lines of both caches are marked as shared (S), as shown in figure 3.36c.

A later write to the variable by P1 modifies the cache line but it does not update main memory (given the write-back policy). In addition, the corresponding cache line in the P2 cache is marked as invalid (I), and the cache line in P1 is marked as modified (M). The state of the system is shown in figure 3.36d.

Finally, a cache miss will occur again if P2 reads again the variable. Since the block is modified in the P1 cache, the block is updated in main memory and then copied to the P2 cache. Finally, both lines are marked as shared (figure 3.36e).

Most memory accesses by a CPU are to non-shared memory addresses that are cached in lines of its L1 cache, in the modified (M) or exclusive (E) state. Therefore, any read or write on these addresses does not require transmitting coherence information to other caches of the system, thus preventing the coherence protocol from introducing any penalty when accessing non-shared memory addresses. This minimizes the effects on performance caused by the maintenance of cache coherence.

3.3.7 Cache memory in the PC

The characteristics of the cache greatly vary from one processor to another. Currently, processors have two or three levels of cache: L1, L2 and L3. L1 is a split cache for data and instructions, and L2 is a unified cache that can be shared among several cores or specific to each of them as L1. L3 is unified and shared by the entire processor.

All cache levels follow a set associative placement policy.

As for the physical location, all cache levels are inside the processor.

For example, in the Intel Core i5-6600K, with four cores, the cache memory features the following characteristics:

- L1 data cache per core (4x).
  - Size: 32 KiB.
  - Line size: 64 bytes.
  - Ways: 8.
  - Number of sets: 32 KiB / (8 $\times$ 64) bytes = 64.
- L1 instruction cache per core (4x).
  - Size: 32 KiB.
  - Line size: 64 bytes.
  - Ways: 8.
  - Number of sets: 32 KiB / (8 $\times$ 64) bytes = 64.
- Unified L2 cache per core (4x)
  - Size: 256 KiB.
  - Line size: 64 bytes.

  - Ways: 4.
  - Number of sets: 256 KiB / (4 × 64) bytes = 1024.
- Unified L3 cache
  - Size: 6 MiB.
  - Line size: 64 bytes.
  - Ways: 12.
  - Number of sets: 6 MiB / (12 × 64) bytes = 8192.

3.4 Virtual memory

Not only is virtual memory a memory management technique that enables the use of disk (or any secondary storage in general) as a level of the memory hierarchy, but it also provides many other characteristics that will be detailed in this section.

Once the general concepts about virtual memory have been introduced, the paging technique is described. This technique enables a straightforward implementation of virtual memory. In fact, this is the technique used in most cases nowadays.

3.4.1 Introduction

Virtual memory makes it possible to use main memory as a cache of secondary storage devices (hard disk drives or solid state drives). They are the furthest level from the CPU in the memory hierarchy, as shown in figure 3.37.

As can be seen, secondary storage devices constitute the biggest and slowest level of the memory hierarchy. The capacity of the memory system is significantly increased without sacrificing speed in most of the memory accesses by using this new level. This is achieved thanks to the principle of locality, introduced in section 3.2. However, virtual memory is much more than a technique for increasing the capacity of the memory system. It also provides a rich functionality based on the concept of virtual addresses.

Programs executed on CPUs supporting multitasking operating systems work with virtual addresses. These addresses are translated into physical addresses by the Memory Management Unit (MMU) to be sent to memory devices.

These two types of addresses, virtual and physical, generate virtual and physical address spaces respectively. Figure 3.38 represents the translation process from a virtual to a physical address.

The virtual address space represents the whole set of virtual addresses that can be formed. Each task in the system has its own virtual address space, which is independent from the virtual address spaces of other tasks. This means that a task can read or write in any of its virtual addresses without affecting any other task. Figure 3.39 schematically shows the address spaces of two tasks, T_1 and T_2, which are two instances of the same program executed at the same time.

The virtual addresses of variables `var` of tasks T_1 and T_2 are exactly the same. However, the execution of the statement `var = var + 1;` in task T_1 affects the value

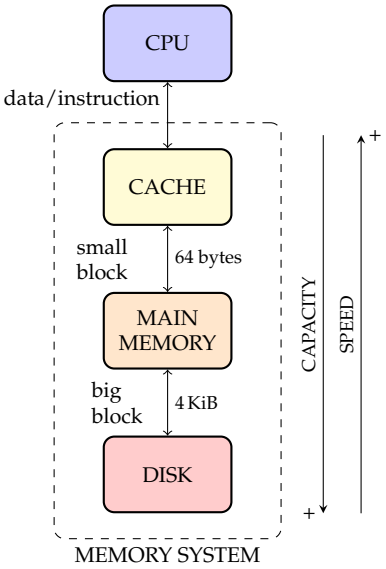


Figure 3.37: Memory hierarchy of three levels

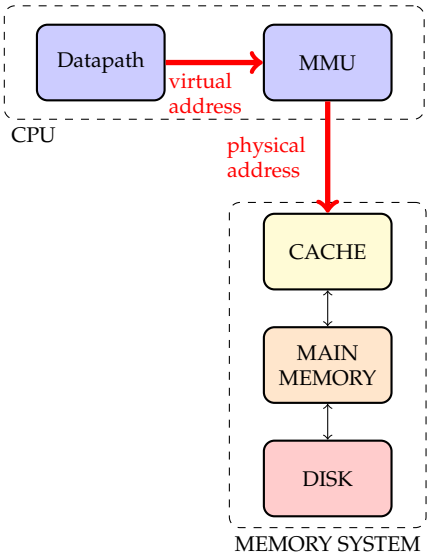


Figure 3.38: Translation of a virtual address into a physical address

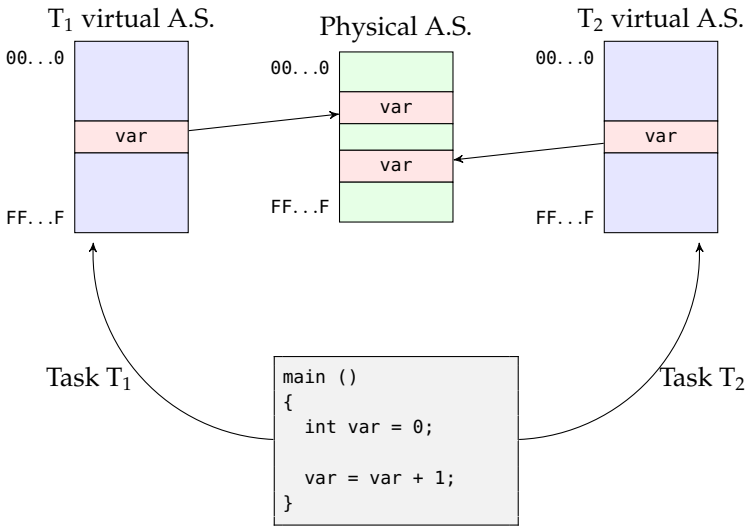


Figure 3.39: Example showing the independence of different virtual address spaces

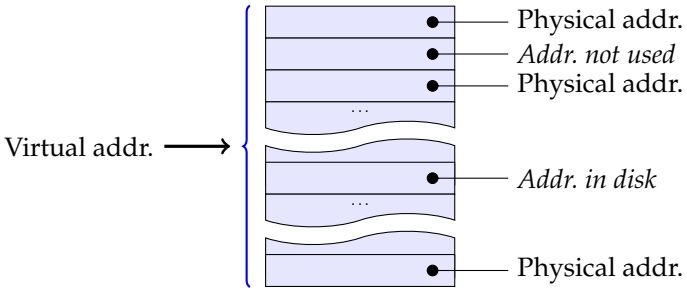


Figure 3.40: Translation table of a task

of its variable `var`, but not the variable of task T_2. The reason is that, even when both variables have the same virtual address, they have different physical addresses, that is, different storage addresses in the memory devices. The addresses associated to programs by development tools (compilers, linkers, etc.) are virtual addresses. As an example, the MIPS64 instruction `ld r3, 80(r7)` stores the double-word located at the virtual address `80 + r7` in the register `r3`.

The physical address space represents the whole set of physical addresses that can be issued, and is unique for the whole system. This space is partially occupied by memory devices. The MMU translates virtual addresses into physical addresses. To do so, the MMU has a translation table from virtual addresses to physical addresses for each task. Figure 3.40 shows a simplified translation table of a task. This table makes it possible to associate a virtual address to a physical address (in main memory), but also to a location in disk. This increases the capacity of the memory system as it will be shown in section 3.4.2.

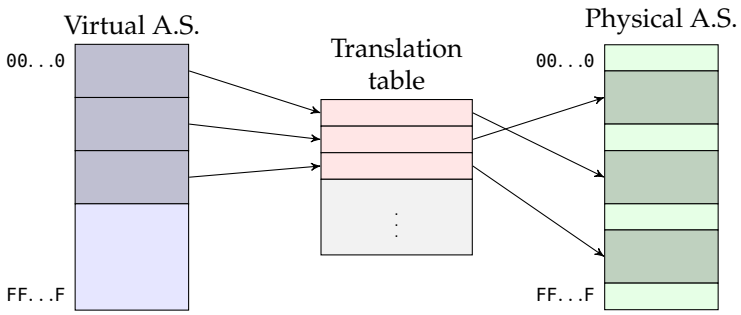


Figure 3.41: Translation from virtual addresses into physical addresses using blocks

3.4.2 Paged virtual memory

The association of virtual addresses to physical addresses is not performed for each address individually, because it would be too cumbersome and inefficient (the translation table would be huge). Instead, the virtual address space is divided into blocks of consecutive addresses that are translated into blocks of consecutive physical addresses, as shown in figure 3.41.

The most used virtual memory technique is paged virtual memory or paging. The blocks, called pages, have a fixed size defined by hardware.

Using paging, a virtual address is composed of two fields:

- Virtual page number.
- Offset.

Figure 3.42 shows as an example a virtual address of 32 bits, with 12 bits used as offset (3 hexadecimal digits) and 20 bits used to identify the virtual page (5 hexadecimal digits).

The size of the page is defined by the number of bits used as offset. In the example, the size of a page is $2^{12} = 4$ KiB, assuming memory is byte-addressable.

The page number field identifies a virtual memory page within all the pages that form the virtual address space. This can be seen in the left side of figure 3.42. There are 2^{20} pages in total from 00000h to FFFFFh.

The offset identifies a virtual address within a page. In fact, the offset added to the base virtual address of the page provides the virtual address.

As an example, the right side of figure 3.42 shows the placement of virtual address 0000 2A10h in the virtual address space based on its virtual page number and offset fields.

In the same way, physical addresses are composed of two fields:

- Page frame number (or physical page number).
- Offset.

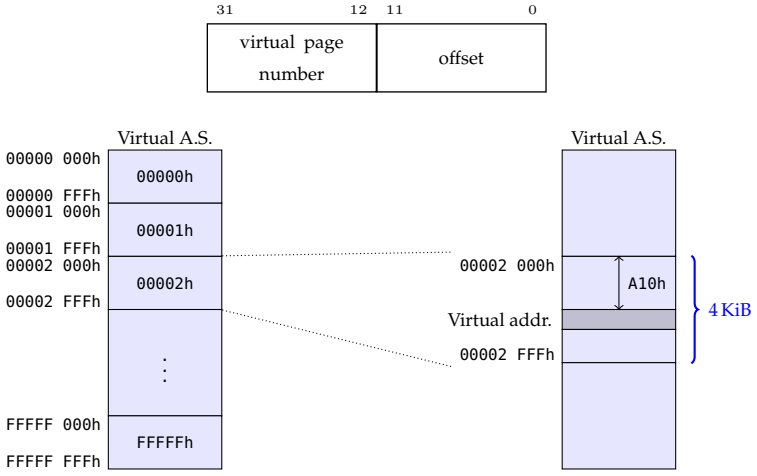


Figure 3.42: Virtual address space organization and placement of a virtual address based on virtual page number and offset fields

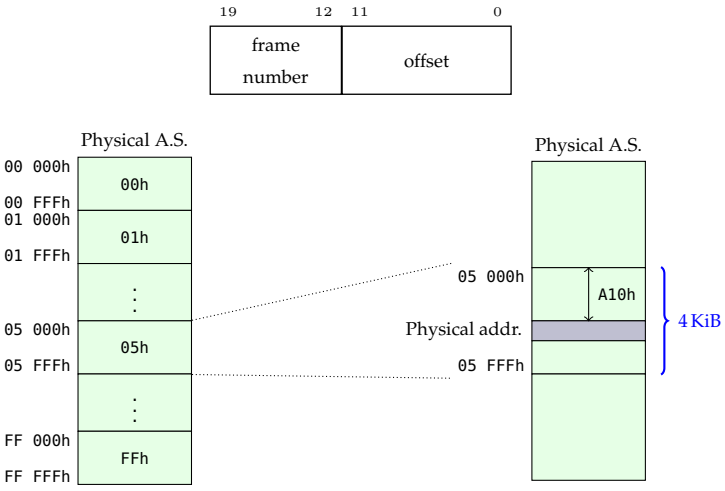


Figure 3.43: Physical address space organization and placement of a physical address based on page frame number and offset fields

Figure 3.43 shows a physical address of 20 bits, composed of 12 bits for the offset (3 hexadecimal digits) and 8 bits to identify the page frame number (2 hexadecimal digits).

The number of bits used for the offset in the physical address is the same as in the virtual address. This is not by chance, because the page frame is the block of physical memory, which must have the same size as the block of virtual memory (the page).

The page frame number field identifies a page frame within all the frames of the physical address space. This can be seen in the left side of figure 3.43. There are 2^8 page frames in total, from 00h to FFh.

The offset identifies a physical address within a page frame. In fact, the offset added to the base physical address of the page frame provides the physical address.

The location of a physical address within the physical address space based on the page frame number and offset fields is analogous to the case of virtual addresses. The right side of figure 3.43 shows an example: address 05A10h.

Various aspects related to paging are described in the next subsections. First, the typical structure of a translation table is shown: the page table. Once the details of the page table are known, the configuration of page tables for mapping the tasks and the operating system into physical and virtual address spaces is analyzed. Finally, the main functionalities provided by paging are reviewed: memory protection, memory sharing and increasing the capacity of the memory system.

The page table

The page table provides a way to translate virtual addresses into physical addresses, associating virtual pages to page frames or disk locations. Each task has its own page table managed by the operating system. The page table of a running task is located in physical memory starting from a position pointed by a control register in the CPU, commonly called the page table register.

The page table has as many entries as virtual pages compose the virtual address space. For example, in the case of the virtual addresses shown previously, 20 bits used for the page number field and 12 for the offset field, each page table has 2^{20} entries (more than 1 million entries). Each entry of the page table typically contains these fields:

- Presence bit. It indicates if the page is in memory. When this bit is not active it means that the page is on disk, inside the paging file, or it has no associated storage. In this last case the page cannot be accessed.
- Page frame / disk location. This field indicates the page frame associated to the virtual page when the presence bit is active. Otherwise, the operating system may give any use to this field. In this case, it usually provides the disk location associated to the virtual page. Operating systems have areas in disk, or in any secondary storage device, reserved for implementing paging. For the sake of simplicity, these areas will be called paging file.[6] An invalid disk location can be used to indicate a virtual page without any associated storage.

[6]Windows has one or more paging files, while Unix operating systems have one or more paging partitions, known as swapping partitions.

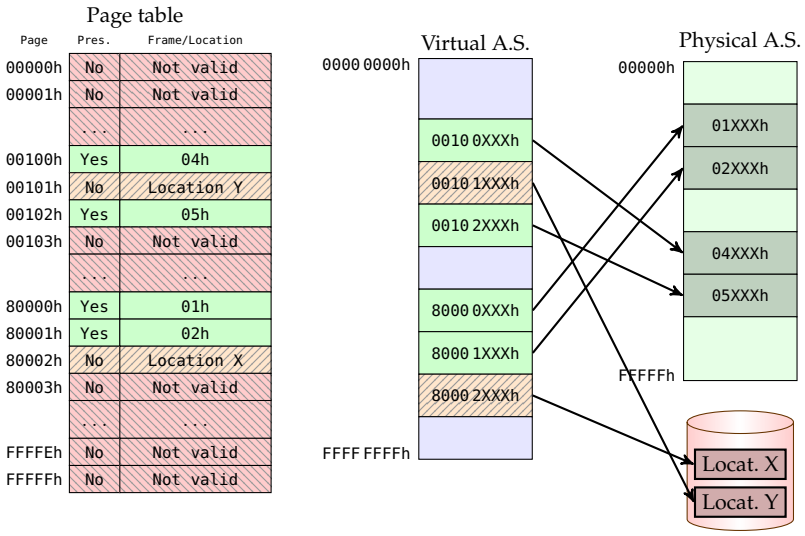


Figure 3.44: Page table of a task

- Protection bits. They set page permissions such as write permission and access privileges.
- Status bits. There is a written page bit or dirty bit, analogous to the dirty bit in the cache, which is used during the replacement of a virtual page in physical memory. If the page has not been written, it can be deleted from memory without updating its content in the paging file. Another typical status bit is the accessed page bit. Its purpose is tracking the accesses to the virtual pages. This information is useful for the operating system when implementing replacement policies like LRU.

Figure 3.44 shows an example of a task page table using the virtual and physical address spaces previously presented, this is, pages of 4 KiB in size, 20 bits to identify the page and 8 bits to identify the page frame. In addition to the page table, the figure shows the virtual address space of the task, the physical address space of the system and the paging file. As can be seen, several fill patterns have been used to differentiate the types of virtual pages. There are pages like 00000h represented using a grated pattern descending to the right that have no associated physical storage and, therefore, cannot be read or written. Their presence bit is not active and they have an invalid disk location associated. Other pages, like 00100h, represented with a solid fill pattern, have a presence bit active, so they have a page frame associated. In this case the associated page frame is 04h. Finally, there are pages represented using a grated pattern ascending to the right that have the presence bit set to 0, but they have a page frame / disk location field that points to a location in the paging file. This is the case of page 00101h, which is stored in location Y in the paging file. To complete the description of the page table, figure 3.45 shows an example of translating a virtual address to the associated physical address using the described page table.

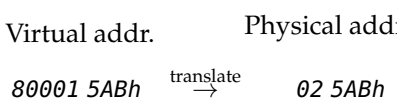


Figure 3.45: Translation example from a virtual address to a physical address

Memory placement of tasks and OS

Regardless of the programming language, the source code of a program is made of code and data. This program is the input for development tools like compilers and linkers, which generate the machine code. This machine code contains the encoding of data and instructions and some additional information that must be handled by the operating system, such as the initialization of some CPU registers. The main idea is that development tools map instructions and data in the virtual address space, associating virtual pages to instructions and data.

When the operating system loads a task in memory to execute it, it assigns physical storage to the virtual pages associated to data and code of the task. Some of these pages could be placed in physical memory and others in the paging file. To do so, the operating system must keep an exhaustive accounting of the free locations in memory and the paging file. Then, the operating system modifies the page table accordingly to reflect the placement of the virtual pages. This technique makes it possible to correctly execute tasks regardless of where they are located in memory and simplifies task loading by the operating system.

Some operating systems include security mechanisms that prevent virtual addresses from being set during compiling and linking time. In this case, relocatable addresses are generated and the operating system assigns both virtual and physical addresses to them during the loading process. This way, virtual and physical addresses change from one execution to another.

The use of paging simplifies the work of development tools. The reason is simple: as each task has its own translation table, the whole virtual address space si available to map code and data. Because of this, compilers and linkers can generate the code of a program ignoring any other program executed at the same time.

Figure 3.46 shows an example of a simple program written in C and the assignment of virtual pages carried out by development tools in a real system.

The program has a global variable called `var`. This variable is composed of 1500 single precision real numbers, which means that each one needs 32 bits, so 6000 bytes are needed to store the variable. The program was compiled and linked in a machine that uses 32-bit virtual addresses and pages of 4 KiB. Therefore, 2 pages are needed to store the global variable. These pages are 00426h and 00427h. Additionally, there is a local variable called `i` in the `main()` function that is placed in the stack. The stack has page 0012Fh associated, as shown in the figure. The code of the `main()` function has page 00400h associated. Finally, it must be noted that the virtual page assignment carried out by development tools is a bit more complex than shown. For example, call and return code to and from the `main()` function has not been taken into account.

On the other hand, the operating system is typically mapped in the virtual address space of every task. The reason is that when an exception, interrupt, or system

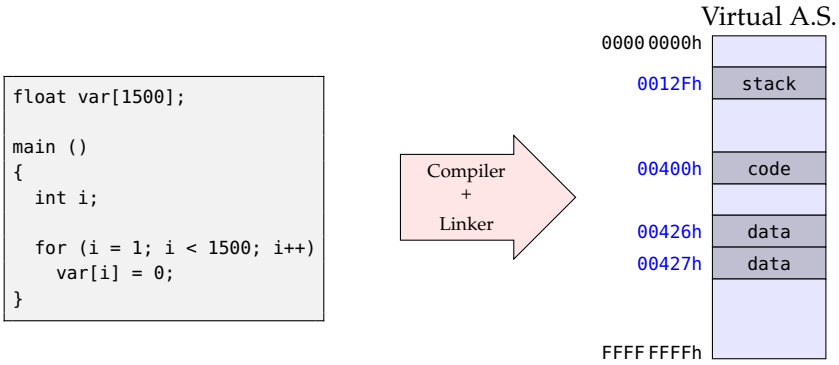


Figure 3.46: Example of code and data location of a program in the virtual address space of a task

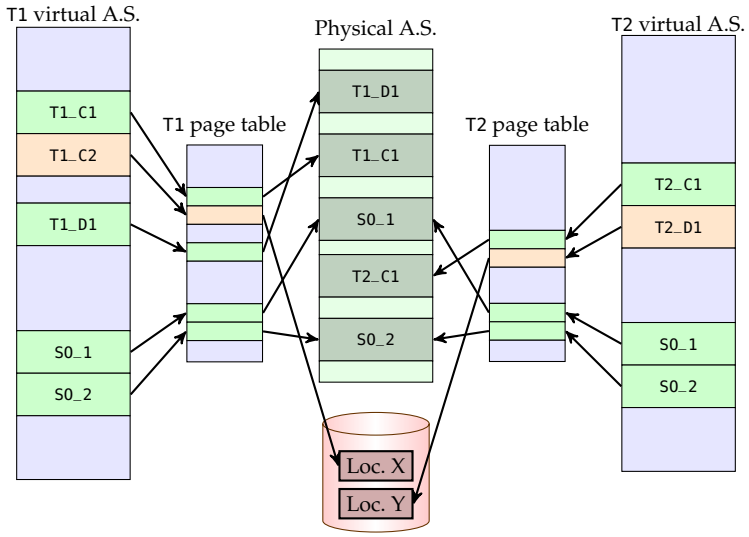


Figure 3.47: Mapping of the operating system in the virtual address spaces of tasks

call occurs, it is necessary to execute the operating system code using the current page table, which can be the table of any task. Figure 3.47 shows a situation where there are two tasks `T1` and `T2`, each one with its own page table, where there are some pages occupied by the operating system.

Task `T1` has two virtual pages for code (`T1_C1` and `T1_C2`) and one virtual page for data (`T1_D1`), while task `T2` has one code page (`T2_C1`) and one data page (`T2_D1`). The figure also shows the assignment of page frames and paging file locations to the virtual pages of the tasks. In particular, all the pages have a page frame associated except for pages `T1_C2` and `T2_D1`, which are stored in the paging file in locations `X` and `Y` respectively.

Figure 3.47 shows the mapping of the operating system in the virtual address space of every task. As the control transfer between a task and the operating system does not imply a change in the page table register, the page table used by the operating system is the same as the one used by the task that gave it the control. This means that the placement of the operating system in the virtual address space is the same for all tasks. As can also be seen, the page frames associated to the operating system are shared by the tasks, because it makes no sense to copy the operating system as many times as tasks are running in the system.

Memory protection

Paging makes it possible to easily achieve memory protection. To do so, three mechanisms are implemented:

- Independence among task virtual address spaces.
- Access type.
- Privilege level.

The fisrt of these mechanisms is the independence among task virtual address spaces. Figure 3.47 shows the organization of the virtual address spaces of two tasks `T1` and `T2` and the operating system. It also shows how the virtual pages of the tasks and the operating system are mapped in the physical address space and the paging file. The operating system sets up the page tables of `T1` and `T2` so that there is not overlapping between the page frames associated to their virtual pages. In the same way, the disk location of the virtual pages of `T1` and `T2` is different, avoiding any kind of overlapping. Therefore, any read or write by `T1` in its virtual pages `T1_C1`, `T1_C2` and `T1_D1` will not affect `T2`. Conversely, any read or write by `T2` in its virtual pages `T2_C1` and `T2_D1` will not affect `T1`. The independence among task virtual address spaces protects each task from the others and also simplifies development tools as previously mentioned.

Another protection mechanism provided by paging is the addition of access types to virtual pages. For example, an entry in the page table associated to a virtual page of a task contains a field that specifies if the virtual page can be read, read and written, only executed, etc. This allows the protection of certain virtual pages against unappropriated accesses.

The last protection mechanism provided by paging consists in the association of a minimum privilege level to each virtual page. This privilege level represents the minimum privilege level at which an instruction should be executed to be able to access a virtual page. Most of CPUs associate two possible levels to the virtual pages: a low privilege level to the pages that can be accessed by tasks and the operating system and a high privilege level to the pages that can only be accessed by the operating system.

Figure 3.48 shows the protection bits associated to the virtual pages of `T1` and the operating system.

As can be seen, all code pages are set as read only, while data pages are set as read and write. The pages associated to the operating system can only be accessed when the CPU is executing instructions with kernel privilege level, that is, while the operating system is executed.

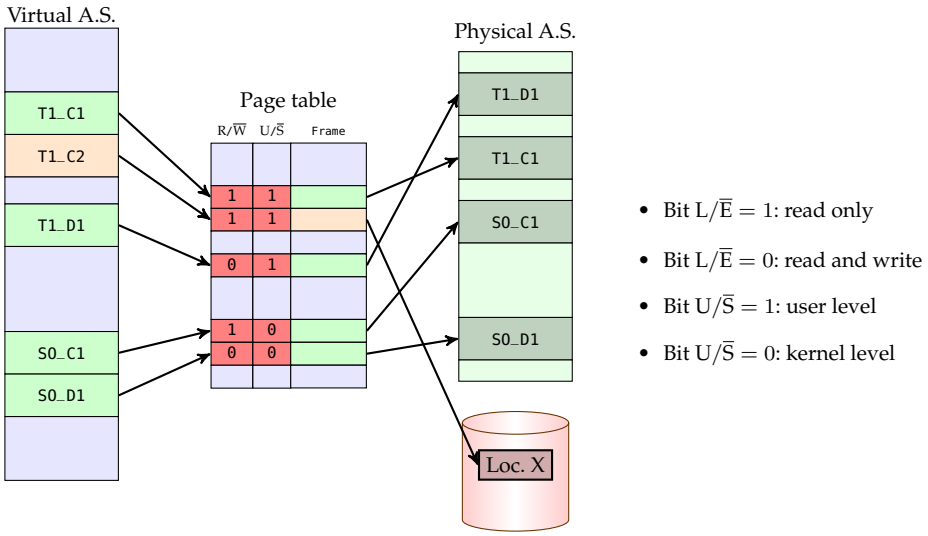


Figure 3.48: Protection bits associated to the pages of a task and the operating system

Memory sharing

Another important feature of paging is its support to memory sharing.

The operating system enables memory sharing between tasks by programming the page table of each task appropriately. Memory sharing makes it possible to have a high speed communication mechanism between tasks and to avoid duplicities in physical memory. For example, as shown in figure 3.47, all the tasks share the physical memory associated to the operating system, avoiding the need of having as many copies of those pages as tasks in the system. Another example of duplicity prevention is the use of dynamic-link libraries. If two or more tasks use the same library, they do not need to have its own copy of the library in physical memory, just a single instance shared between all the tasks is needed.

Figure 3.49 shows how shared memory areas can be created during the execution of programs. At a given time, task `T1` requests a shared page frame associated to its virtual page `T1_D2` to the operating system.[7] Assuming that the operating system grants the request, it programs the entry of the page table of `T1` associated to that virtual page properly, so that it points to the granted page frame. From that moment on, `T1` can read and write its virtual page `T1_D2` because it has physical storage associated. Later, task `T2` asks the operating system to share the previous page frame and to have it associated to its virtual page `T2_D2`. Once the operating system grants the request, the page frame becomes shared between `T1` and `T2`. This means that anything that `T1` writes on its virtual page could be read by `T2` using its virtual page `T2_D2` and vice versa.

[7] This is done through a call to special services of the operating system

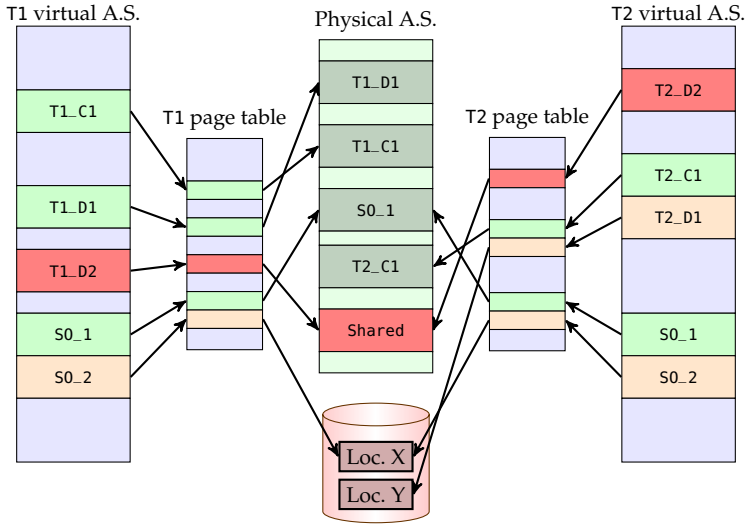


Figure 3.49: Memory page shared between two tasks

Increasing the capacity of the memory system

The objetive is having a large capacity memory system for:

- Being able to execute programs larger than the size of the installed main memory.
- Being able to execute, at the same time, many programas that would need more memory than the installed main memory.

This increase in the capacity of the memory system is achieved using main memory as a cache of the disk. When the CPU issues a virtual address, the page number field is used as an index in the page table of the task. If the virtual page is associated with a page frame, the MMU generates the physical address corresponding to the virtual address and the data is accessed directly in physical memory. On the contrary, if the page table shows that the virtual page has a disk location associated, an exception is triggered. This exception is called a page fault. As consequence, the exception handler, which is part of the operating system, is executed and the page is moved from the disk to main memory. Finally, the operating system modifies the entry in the page table so that it shows the new situation. Now, the page is stored in main memory and if the access is repeated it will not trigger an exception.

Although paging makes it possible to increase the capacity of the memory system beyond the capacity of main memory, the extra capacity is gained at the expense of reducing the performance of the system. During the page faults it is necessary to move disk blocks from disk to main memory, which requires much time compared to the time needed to access main memory. Next, the operations that happen when a page fault occurs are described in detail.

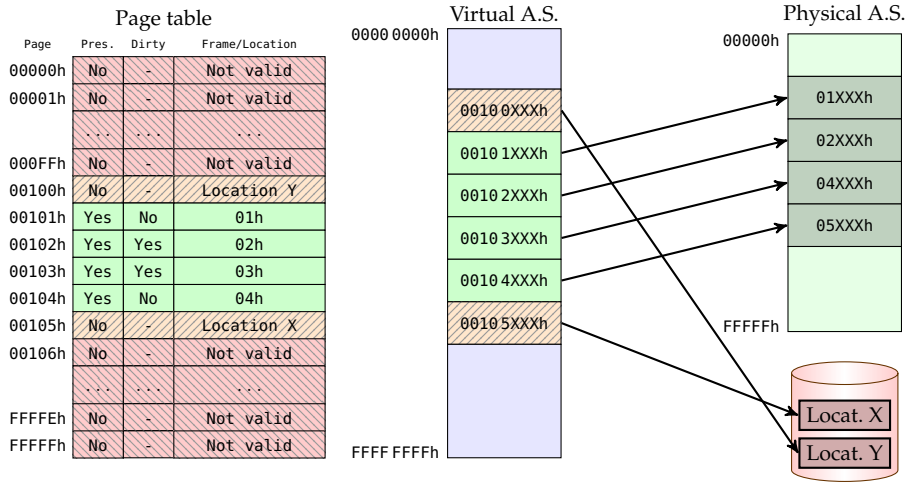


Figure 3.50: Initial situation before the page fault

A page fault occurs when the CPU issues a virtual address of a page that is not located in physical memory. This situation is easily detected by the CPU hardware, as the presence bit is not active. When this happens, the CPU triggers a page fault exception that is managed by the page fault handler, which is part of the operating system.

In order to illustrate the steps followed by the page fault handler the initial situation shown in figure 3.50 is assumed. Next, these steps are enumerated.

A) The cause of the page fault is analyzed. There are two possible causes: address of a page without any associated storage or address of a page located in disk.

B) An address in a page that does not have associated storage means that it is not in physical memory nor in the paging file. In this case, a memory access violation exception is triggered. For example, when a program tries to access address 000F F05Eh (page 000FFh) using the page table from figure 3.50.

C) If the address is stored in disk, like for example address 0010 51B3h (page 00105h), the next operations are carried out:

   C1) The operating system determines in which page frame is going to be loaded the page that is stored in disk. To do so, a replacement policy is used (LRU for example), in the same way as for the cache memory. The algorithm may choose between one of the four page frames assigned to the task or another page frame that has not belonged to the task. The set of page frames assigned to a task is called the working set. Page frame 02h is chosen in the example.

   C2) If the page to be replaced has been modified (bit *dirty* is active), as it happens in the example (page 00102h), the page is saved to disk before it is replaced and the associated entry in the page table is updated. The new situation is shown in figure 3.51.

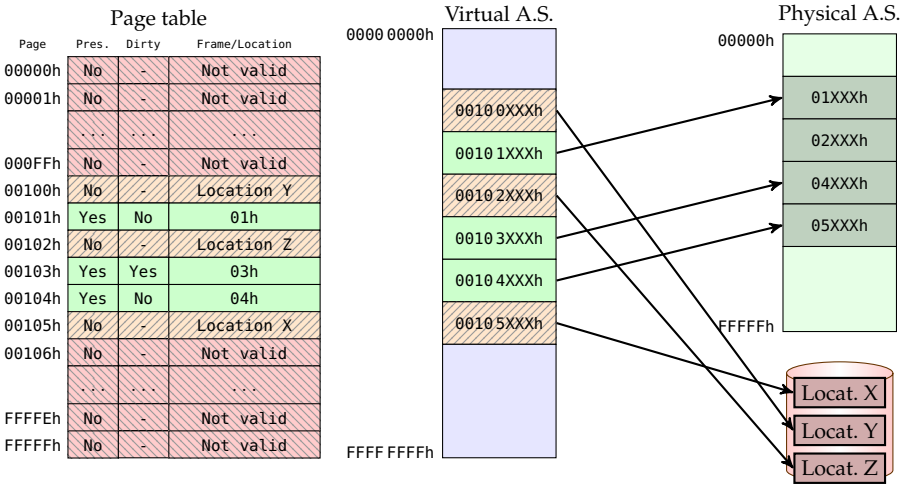


Figure 3.51: Movement of modified page 00102h to disk before it is replaced

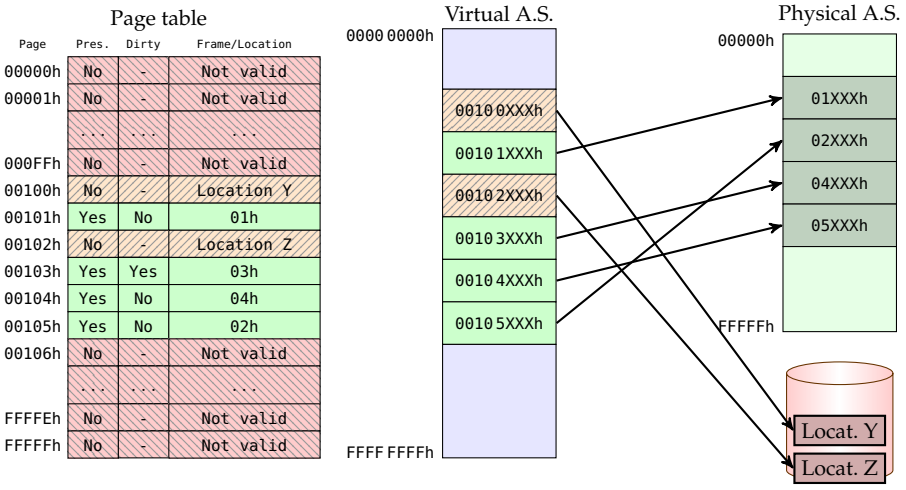


Figure 3.52: Loading page 00105h from disk

C3) The requested page is loaded from disk and the corresponding page table entry is updated. The new situation is shown in figure 3.52.

C4) The page fault handler returns to the instruction that triggered the exception once the requested address is located in the physical memory. The same instruction is executed again, but it will not trigger a page fault exception this time.

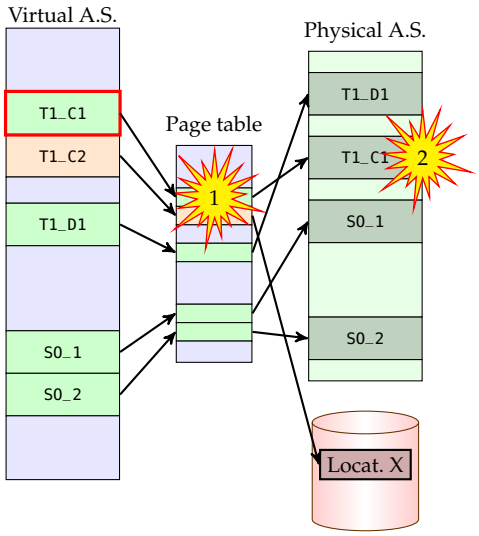


Figure 3.53: Memory access using paging without TLB

3.4.3 The TLB

So far, aspects related to the functionality of the virtual memory, paging in particular, have been studied. However, the use of this technique has important consequences in the machine performance that must be taken into account.

Every access to a virtual address by the CPU implies, in fact, two accesses to memory using paging: one to the page table and another one to the associated physical address. Figure 3.53 illustrates this issue.

Task `T1` accesses a byte of memory in the virtual page `T1_C1`, which requires:

1. Accessing physical memory to read the entry in the page table associated to virtual page `T1_C1`.
2. Accessing the physical address resulted from the translation of the virtual address using the entry in the page table.

Obviously, this significantly reduces the performance of the system. Supposing that there is not cache memory, each access to a virtual address will result in two accesses to main memory.

Thanks to the principle of locality, a task often accesses a few virtual pages, whose entries can be stored in a small specific cache memory in the CPU. This is usually a fully associative cache and is called Translation Lookaside Buffer (TLB). This way, the access to a virtual address is reduced to one access to the TLB and one access to main memory. Taking into account that the access time to the TLB is far less than the access time to main memory, the former can be considered negligible. Therefore, the use of paging, theoretically, should not affect the performance of the system if the TLB is used.

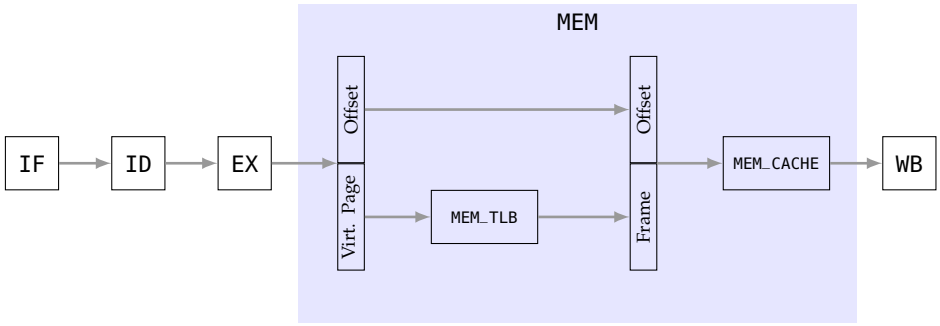


Figure 3.54: Pipelining in the access to TLB and cache

The management of a TLB miss is usually carried out by the hardware, in the same way it is done in general purpose caches. However, architectures usually provide privileged instructions related to the TLB, for example, to invalidate all the entries in the TLB or just some specific entry.

The content of the page table register changes, so does the page table on every context switch, when the execution changes from a task to another. Therefore, the operating system must invalidate all the entries in the TLB. Then, several TLB misses occur (at least one for the data area, one for the code area and another one for the stack) that slow down the resuming of programs after context switches. The usual solution consists in adding a set of bits to each TLB entry that identifies the owner task. A special sequence of these bits can be reserved for identifying some entries as global, that is, valid for all tasks. These global entries could be used to translate the addresses associated to the operating system, because its addresses do not change when switching between tasks. Of course, the TLB must have a bigger capacity if only one is used for all tasks.

The existence of the general purpose cache memory between the CPU and main memory was ignored intentionally when the concept of TLB was introduced for simplifying the explanation. Nevertheless, the coexistence of the TLB and the cache has great implications from the performance point of view. The access time to the TLB can not be considered negligible anymore, because the memory access time is often now equal to the access time of the cache, which is similar to the access time of the TLB. Therefore, one access to virtual memory becomes one access to the cache and another one to the TLB, which is equivalent to two accesses to cache and is an important performance loss. This loss is unacceptable so different solutions are proposed next.

Figure 3.54 illustrates one possible solution to the performance issue using a pipelined CPU like the one described in the chapter dedicated to the CPU. The original `MEM` stage is pipelined in two stages, `MEM_TLB` and `MEM_CACHE`, each one requiring one clock cycle. The former obtains the page frame number based on the virtual page number and the latter accesses the cache using the obtained physical address.

In ideal conditions, stages `MEM_TLB` and `MEM_CACHE` can work in parallel thanks to the pipelining of the memory access, so the access time to memory will be one clock cycle. However, the efficiency of this solution is limited, because it requires many

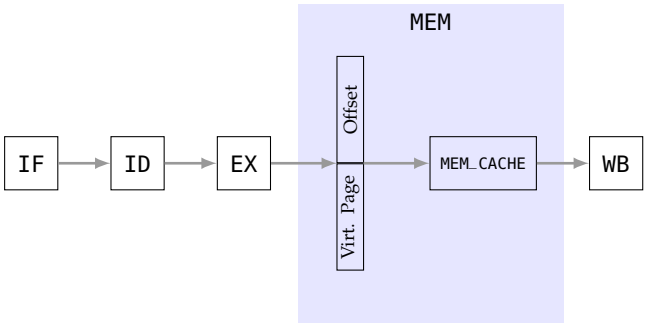


Figure 3.55: Virtually indexed and virtually tagged cache (V/V)

consecutive memory accesses. This is why the solution used in practice is virtual caches, which are introduced below.

Virtual caches

Virtual caches are caches that totally or partially use virtual addresses to work. The idea is that caches working with virtual addresses do not need to wait for the TLB to obtain the page frame associated to the virtual page.

Before studying virtual caches deeper, it should be reminded that every single memory block that is cached is identified by two elements:

- Set (or index). Cache lines are organized in sets, so a memory block can be placed in any line of the set. The number of lines or ways of the set is called degree of associativity. Direct mapped placement and fully associative placement are particular cases of the set associative placement.
- Tag. It allows the identification of a memory block between all that can be placed in the same cache set.

Caches described so far are indexed and tagged with bits from the physical address. They are called P/P caches (physically indexed / physically tagged). Both the cache set and the tag come from the physical address, so it is necessary to access the TLB before accessing the cache. This kind of caches lead to a performance loss, even when the access to the TLB and the cache are pipelined. However, there are some alternatives that are always applied to the L1 cache. The reason is that the performance issue derived from the address translation only arises in the level of cache closest to the CPU. Cache levels L2, L3 and successive do not show this issue as they always work with physical addresses, so they are ignored when dealing with address translation.

V/V caches are virtually indexed and virtually tagged caches. These caches exclusively work with virtual addresses, so its is not necessary to wait for the TLB. Figure 3.55 shows the structure of a virtually indexed and virtually tagged cache.

The L1 cache serves the data when accessing a cached virtual address. Therefore, the access time to memory is one clock cycle, not two, in case of a cache hit, so the

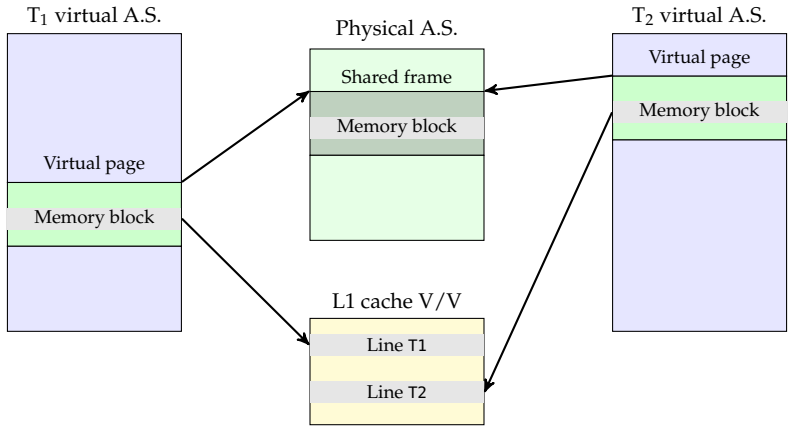


Figure 3.56: Synonyms issue in virtual caches

performance issue is solved. If a cache miss occurs, an access to the TLB is needed to translate the virtual address into the physical address in order to access the L2 cache, which works with physical addresses, and deal with the L1 cache miss.

Although the use of L1 V/V caches solves the performance issue they introduce some additional issues.

- Each cache line should include a task identifier in order for the cache to store data from different tasks, similarly to the TLB with physical caches. If not, the cache should be invalidated with every context switch, as a virtual address may be mapped to different physical addresses in different tasks.
- The cache must manage memory protection. The TLB is not used in the case of a cache hit, so the cache must do this management. To do so, each cache line must have some bits to indicate the access type ($R/\overline{W}$) and the privilege level ($U/\overline{S}$) that must be consistent with the bits associated to the virtual page in the page table.
- The synonyms issue shows up. This issue requires special attention and is illustrated in figure 3.56.

  Tasks `T1` and `T2` share a page frame using different virtual pages. Since the virtual addresses of the shared memory area are different for each task, the same memory block in that area can be placed in two different sets of the cache at the same time, one with the identifier of task `T1` and the other with the identifier of `T2`. This is possible because the virtual addresses are different. Besides the wasted cache space, a coherence issue may appear because the same memory block is placed twice in cache. If `T1` writes in the shared memory block, it will write in the cache line with the identifier of `T1`. Then, if `T2` reads the content of the shared memory block, it will read the cache line with the identifier of `T2`, so it will not read the updated value.

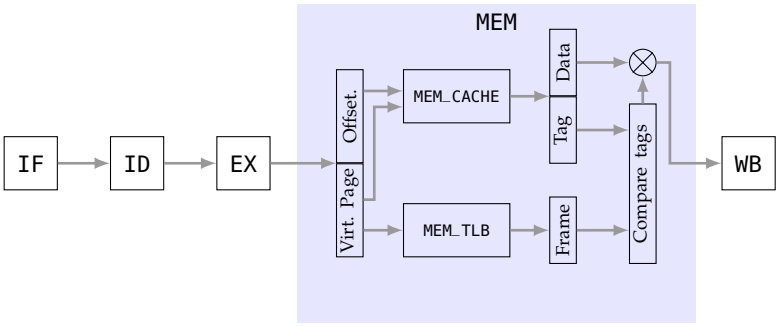


Figure 3.57: Virtually indexed and physically tagged cache (V/P)

A solution to this issue is invalidating the L1 cache (after updating the modified blocks in L2) with every context switch. Another solution is to look for synonyms in the L1 cache at the time of writing and invalidate the lines with synonyms. In both cases the cost in terms of performance in unacceptable.

As seen so far, using physical caches (P/P) or virtual caches (V/V) causes performance issues. In practice, the solution used consists in mixing both strategies using a virtually indexed and physically tagged L1 cache (V/P). Figure 3.57 shows the structure of this cache.

The cache is virtually indexed, which means that the virtual address is used to select the cache set where the virtual address can be placed. The novelty compared to V/V caches is that the tag field of a cache line does not comes from the virtual address in this case, but from the physical address. Therefore, the tag stored in all the lines in the set must be compared with the one obtained from the physical address to check if a virtual address is cached. In order to get the tag from the physical address, the TLB is accessed in parallel using the virtual page as usual, providing the page frame from which the physical tag is obtained. As the TLB and the cache are accessed in parallel there are no performance penalties; the access time to memory is one clock cycle.

V/P caches have the benefit compared to V/V caches that every memory access requires the use of the TLB, so no privilege management or any other cache access restriction is needed. However, they share the synonyms issue with V/V caches. The same memory block in a page frame shared by two tasks `T1` and `T2` may be placed in different sets for each task because the cache is virtually indexed. Nevertheless, the solutions in this case are much more simple and efficient than for the case of V/V caches.

Even though there are several possible solutions, the most common is to limit the size of the L1 cache and/or increment its associativity (number of ways). If the size of the page multiplied by the number of ways is greater or equal to the size of the cache, the set where the virtual address is cached can be obtained from the offset of the virtual address, which is the same as the offset of the physical address, so it does not depend on the virtual page number. In other words, if several tasks share a memory block, this block will be placed in the same cache set no matter the task, so the synonyms (duplicities) cannot exist. This solution is the chosen in many

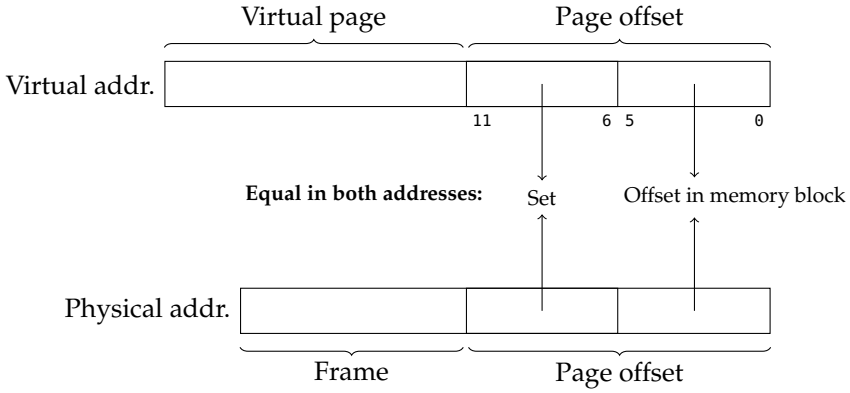


Figure 3.58: Obtaining the cache set from the offset of the physical address

PC processors by Intel and AMD. For example, each core of the Intel Core i5-6600K processor has a L1 data cache of 32 KiB, with 8 ways per set and the minimum page size is 4 KiB. Using this example, figure 3.58 shows how the cache set can be obtained directly from the offset in the address, so it does not depend on the virtual page. The situation is analogous to removing the arrow from the virtual page to the `MEM_CACHE` stage in figure 3.57.

The result of multiplying the size of the page by the number of ways of the cache is 4 KiB $\times$ 8 = 32 KiB, which is equal to the size of the cache. The problem with this strategy is the increase in the cost of the L1 cache, which grows rapidly with its degree of associativity. However, this cost is affordable with current manufacturing techniques.

3.5 Virtualization support

Virtual memory is an example of virtualization where a unique physical resource, the physical address space, is shown as multiple spaces of virtual addresses, one for each task. Each task sees the "illusion" that it has a complete address space for itself, but in fact, it is sharing a physical address space with other tasks. The MMU with correct management by the operating system makes that "illusion" possible.

A new virtualization level is needed when virtual machines are used, because several operating systems must share a single physical address space at the same time. This sharing is managed by the hypervisor by introducing a new virtualization level between the virtual memory (managed by the operating system) and the physical memory (managed by the hypervisor). Therefore, three address spaces are defined:

- Guest Virtual Addresses (GVAs). The addresses seen by a task.
- Guest Physical Addresses (GPAs). They are not real physical addresses. They are managed by the operating system as the memory assigned to the virtual machine.

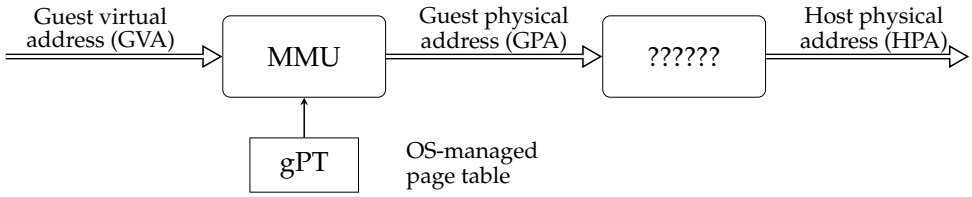


Figure 3.59: Translation between the three address spaces

- Host Physical Addresses (HPAs). These are the real physical addresses of the system, managed by the hypervisor.

Therefore, in order to translate a virtual address of a task into a physical address of the system two consecutive translations are needed, as seen in figure 3.59:

- Guest virtual address (GVA) → Guest physical address (GPA). This translation is carried out by the MMU using the page table managed by the operating system as shown in section 3.4.2. This page table is called guest physical table (gPT) in this context.
- Guest physical address (GPA) → Host physical address (HPA). This translation creates a problem because the CPU does not have any device that can perform it.

The solutions used in the x86 architecture are presented below.

The first solution to the translation problem between guest physical addresses and host physical addresses is based on shadow page tables. Each task has its own table called shadow page table, or sPT, managed by the hypervisor, that contains the direct translation between a guest virtual address (GVA) and the corresponding host physical address (HPA). Therefore, the MMU works with the shadow page table (sPT) instead of the guest page table (gPT) as shown in figure 3.60.

The hypervisor must intercept any writing the operating system performs to a guest page table (gPT) for the operating system to have the illusion that it is managing the host physical address space. This can be done by setting all pages as read-only, so a page fault exception will be triggered when the operating system tries to modify an entry. The handler of the page fault exception, executed by the hypervisor, will modify the proper entry in the shadow page table. Additionally, it is necessary that the hypervisor intercepts any management instruction over the TLB the operating systems execute such as the invalidation of entries in the TLB.

One of the issues of using shadow page tables is the great amount of context switches that occur between operating systems and the hypervisor in order to manage paging, which degrades performance. Manufacturers of x86 CPUs, as part of the hardware virtualization support, have developed a technique called Second Level Address Translation (SLAT), also known as nested paging, in order to alleviate this performance issue.

In a first stage, when a task accesses a virtual address (GVA) using nested paging, the MMU translates it to a guest physical address (GPA) using its gPT, which is

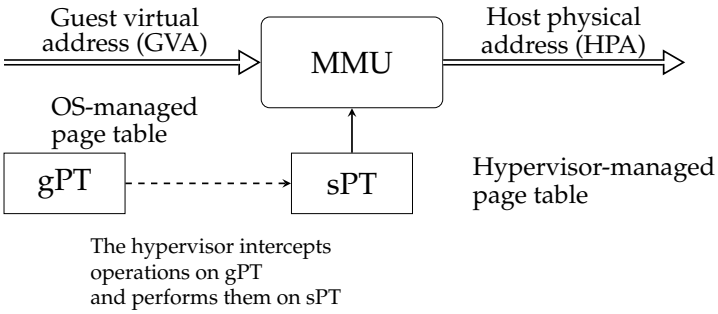


Figure 3.60: Translation using shadow page tables

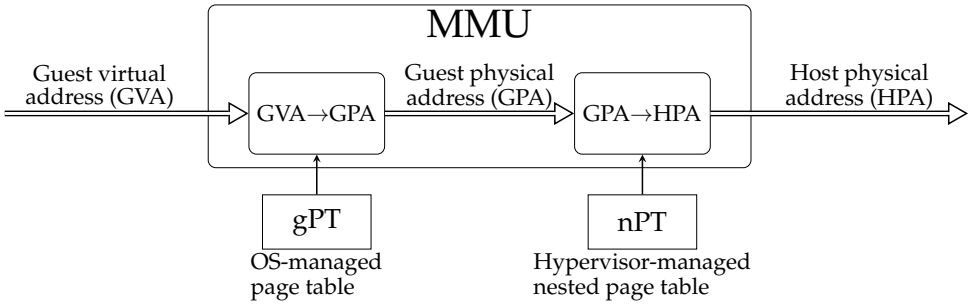


Figure 3.61: Translation using nested paging

managed by the host operating system or the hypervisor. The host operating system can write in that table and perform the usual TLB management operations without meddling of the hypervisor.

In a second stage, the MMU translates the resulting address to a host physical address (HPA) using a second translation table called nested page table (nPT). This second table is under control of the hypervisor. Figure 3.61 gives a general outline of the translation of the addresses of a task using nested paging.

Using the nested paging technique, the TLB stores the direct translation between guest virtual addresses (GVAs) and host physical addresses (HPAs) during the execution of the operating system and its tasks.

Nested paging does not always provide a performance increase comparing to shadow page tables. The cost of a TLB fault is greater using nested paging than using shadow page tables because the number of page table entries read from memory is doubled: there is a page read from the gPT and another one from the nPT.

Chapter 4

The input/output system

The definition of the computer architecture is completed with the third block, known as the I/O system. This block of the architecture is essential in every computer, since it is in charge of the communication between the computer and its environment. Unlike the other blocks of the architecture described before (CPU and memory), the I/O system is pretty heterogeneous because the communication with the environment can involve different types of interlocutor (people or other systems) and each interlocutor can carry out communications with different characteristics (sound, images, printed text, etc).

The communication of the computer with its environment usually involves three elements:

- Peripheral devices. They are the devices that perform the communication with the environment: screen, mouse, keyboard, network devices, etc.
- Interfaces. They are the computer components in charge of performing the adaptation between the binary information handled by the computer and the type of signal the peripheral uses.
- The interconnection system. This is the set of technologies and devices used for moving information. Not only is it used for communication with the environment (communication between CPU and memory is done also throught it), but it is also responsible for the variety of the existing technologies to adapt to the requirements of the communications with the peripheral devices.

The three previous elements participate in the communication between the computer and the environment. In practice they are usually studied separately. Firstly, the I/O interfaces will be studied. Secondly, the interconnection system will be presented. Finally, peripheral devices will be briefly reviewed.

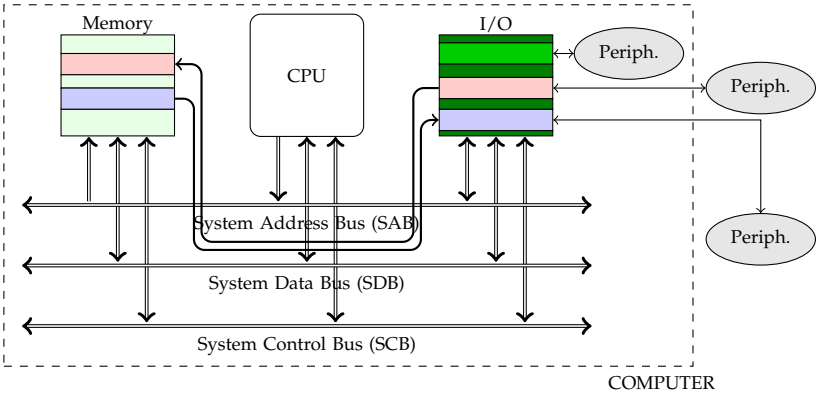


Figure 4.1: Input and output operations in the von Neumann computer

4.1 I/O interfaces

The interfaces of all the peripheral devices of the system are part of the I/O system. Usually, each peripheral has one interface, but one single interface can handle several peripherals.[1]

In general, any I/O operation consists in transferring information between the memory system and the peripheral device. The direction of the communication is set always from the computer point of view. Thus, an input operation consists in transferring information from the peripheral through its interface to the memory system. An output operation consists in transferring information from the memory system to a peripheral through its interface.

Figure 4.1 illustrate the I/O operations in a von Neumann computer. The components of the computer are placed inside the dotted line. It can be seen that some peripheral devices are included in the computer.[2]

4.1.1 Mapping in address spaces

In order to perform I/O operations the CPU needs to access the interfaces to read or write information. This is possible by assigning addresses to the different elements of the interfaces (registers or ports). These addresses are implemented as locations in the address spaces of the CPU. As a reminder, the address space is the set of all the addresses that the CPU can form. I/O operations consist in the reading or writing in the addresses where the interfaces are mapped. The effect of the read/write operations in these addresses depends on the interface. For example, the value written in a certain position of the video interface is converted into data that flows to the screen making some information to show up.

[1] This is the case of the PS/2 interface for the keyboard and mouse, the USB interface or the *Firewire* interface.

[2] There are some peripheral devices like hard-disk drives that are used to support some inner functions of the computer and not to communicate it with the environment. These devices are usually placed inside the computer

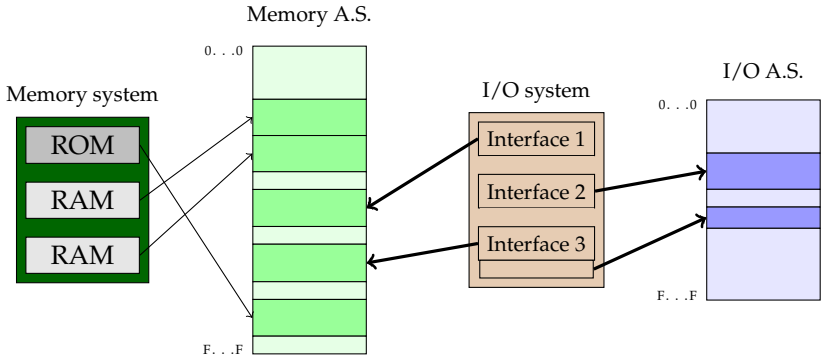


Figure 4.2: Example of location in address spaces

In practice, every CPU has at least one address space, called the memory address space. In this space, among other things, the main memory of the system is mapped. Computers whose CPU has a single address space map the interfaces of all peripheral devices in the memory address space. This technique, known as memory mapped I/O, consists in reserving some addresses to be assigned to the interfaces of the peripheral devices. This way, I/O accesses are simplified converting them in simple memory accesses that are carried out with the same instructions. However, the disadvantage is that the amount of available memory addresses for the main memory is reduced.

There are some CPU, such as those implementing the x86 architecture, that have an additional memory address space called the I/O address space. In this case, interfaces can also be mapped in the I/O address space. This mechanism is called separated address space. It has the advantage that no memory addresses are sacrificed for mapping interfaces, but specific instructions are needed to access to the I/O address space. Figure 4.2 shows a possible location of the memory system and the I/O system in address spaces.

4.1.2 Protection

From the I/O point of view, what differentiates a CPU that supports multitasking operating systems from another that does not is the possibility of limiting the access to the interfaces of peripheral devices. These limitations prevent tasks from accessing interfaces directly. Only the operating system can access interfaces.

Protecting interfaces mapped in the memory address space is usually carried out by assigning a kernel privilege access level to the pages where the interfaces are mapped. This way, only the operating system can access those interfaces.

Read and write operations in the I/O address space are carried out using specific instructions. For example, instructions `in` and `out` in the x86 architecture. The protection of the I/O address space is achieved by allowing the execution of these instructions only at kernel level.

Figure 4.3 shows how an application that wants to access interfaces in a multitasking operating system will not be able to do it directly, but only through the

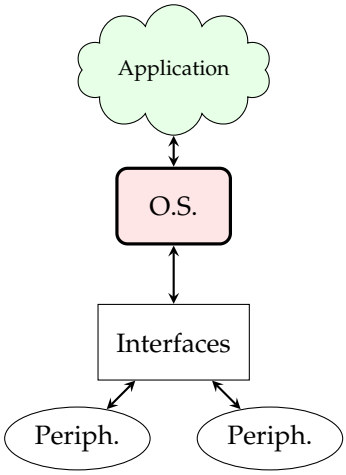


Figure 4.3: I/O access in multitasking operating systems

services provided by the operating system. Even in systems that do not support multitasking, the access to the interfaces is usually done through services provided by the operating system.

4.1.3 I/O techniques

As commented, I/O operations consist mainly in transferring information between the interfaces of the peripheral devices and the memory.

An issue to be solved is how the CPU is synchronized with the I/O operation. Both the CPU and peripheral devices can start I/O operations. In the first case, the program gets to an instruction that involves an I/O operation. For example, calling the C function `fprintf` that writes to a file. In this case, the CPU starts the I/O operation. In the second case, at any time the interface of a peripheral device has some data provided by the peripheral device, but the CPU can not know, *a priori*, when this will happen. For example, the CPU can not know when the user will push a key or move the mouse. Then, it is said that the peripheral device or its interface starts the I/O operation.

Once the I/O operation is started, data must be moved between the peripheral interface and the memory. This transfer can be done by the CPU (executing load and store operations), or some other devices can be used to free the CPU from doing these tasks and, therefore, leaving it available to execute other tasks.

The existing techniques to carry out I/O operations are enumerated next from the least to the most efficient:

- Pooling-based I/O.
- Interrupt-based I/O.
- Direct Memory Access or DMA-based I/O.
- I/O processors.

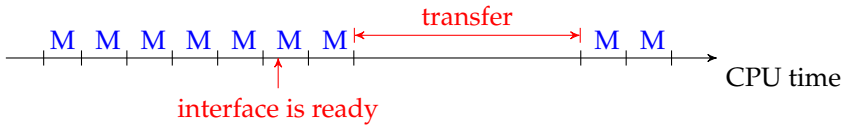


Figure 4.4: Pooling-based I/O technique

Pooling-based I/O

This is the simplest I/O technique, but it is also the worst from the performance point of view.

In general, interfaces have at least one status register and one data register. This technique is based on the CPU pooling continuously the status register of an interface. This allows the CPU to know when the interface is ready, that is, when it has available data in the data register or it is ready for getting new data through the data register. Once the interface is ready, the data transfer between the interface and the memory is performed by the CPU. This is depicted in figure 4.4. As can be seen, while the interface is not ready, the CPU is in a loop pooling the status register constantly. Once the interface is ready, the CPU gets out of the loop.

This technique can be applied simultaneously to more than one interface. To do so, the CPU must pool sequentially the status registers of all the interfaces.

This technique is very inefficient because it keeps the CPU busy pooling the status register until an interface is ready. It can be applied to small single-task operating systems where the performance is not essential. However, it is not applied in personal computers or servers in which multitasking and performance are very important features. It is not used either in battery-powered devices due to the great power consumption it produces.

Interrupt-based I/O

It is a frequently used technique that avoids pooling interfaces to know when they are ready. The idea is simple: instead of having the CPU pooling the status register of the interface, the interface informs the CPU when it is ready. Thus, the CPU does not waste time on pooling the status register.

When the interface of the peripheral is ready, for example when a data is available, it generates an interrupt request to the CPU. At the end of each instruction, the CPU checks if an interrupt has been raised. If there is no interrupt, it keeps executing the program with the next instruction. If there is an interrupt, it executes a small piece of code associated to the interface, where it typically transfers data between the memory and the interface. Once the execution of that piece of code finishes, the CPU returns to the program where it left it. The small piece of code associated with an interface is called interrupt service routine or interrupt handler.

Figure 4.5 shows the execution of the program composed of instructions `A`, `B`, `C`, `D` and `E` and the interrupt service routine composed of the instructions `1`, `2` and `3` when a interrupt is produced during the execution of the instruction `C`.

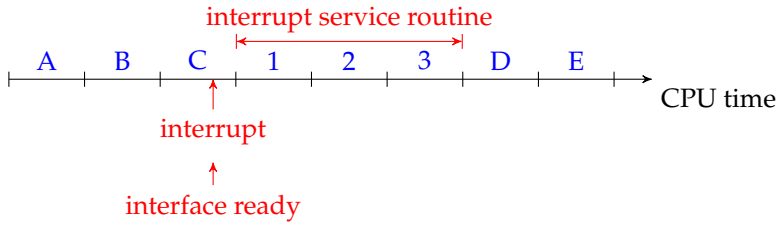


Figure 4.5: Execution of a program with an interrupt

The management of interrupts in a pipelined CPU is similar to the management of exceptions, as seen in section 2.3.4.

The implementation of interrupts in a given architecture poses several issues:

- Priorization. When two or more interfaces raise an interrupt during the execution of the same instruction it must be chosen which one is served first. To do so, priorities must be set between the interfaces of the peripherals.
- Interrupt disabling. There are situations when the CPU must not be interrupted during the execution of a certain piece of code. For example, during the modification of the table page. If an interrupt service routine starts its execution before the modification of the page table is completed, it can take down the whole system. Of course, there must be some mechanisms to enable the interrupts and return to the previous situation.
- Interface identification. It is common that a processor has a limited number of interrupt request lines. For example, x86 compatible processors have a single request line for interrupts that is shared with all interfaces. Therefore, once a interrupt is raised there must be some mechanism to identify which interface (the one with higher priority) has raised the interrupt and so which interrupt service routine must be executed.

Direct Memory Access (DMA)

Interrupt-based I/O has a disadvantage, which has more impact in multitasking systems when the data flow between interfaces and memory is high. The CPU is in charge of the data transfer between interfaces and memory, so it cannot execute other programs while the data transfer is being done.

The main idea of the DMA is freeing the CPU from the data transfer. Thus, this data transfer is done by a device called DMA controller. The classic DMA controller is an element that simply moves information between interfaces and memory.

Figure 4.6 shows the data flows in an input operation with and without DMA in a multitasking operating system with 3 user tasks. Task `T2` needs data from a peripheral device and asks the operating system for it. `T2` stays suspended until the operating system provides that data.

If using the interrupt-based I/O technique, the operating system would generate a context switch, making the CPU execute another task, task `T1` for instance. Once the

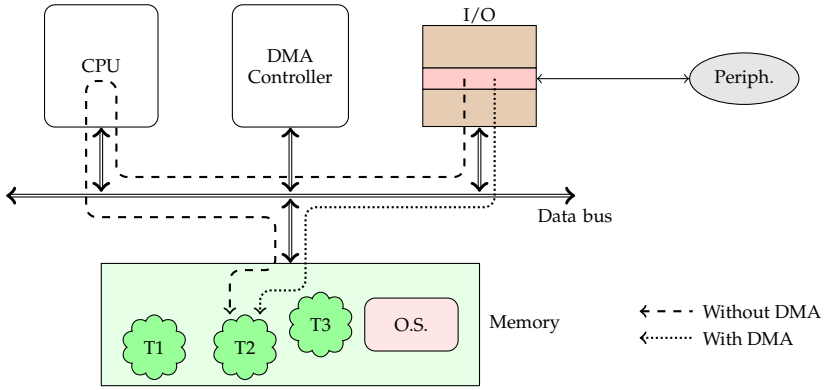


Figure 4.6: Data transfer during an input operation with and without DMA in a multitasking operating system

interface is ready, it would raise an interrupt. The CPU would stop executing task `T1` and would execute the interrupt service routine associated to the peripheral device. In this routine, the CPU would transfer the data from the interface to memory and wake up task `T2`. This would result in a new context switch. The CPU is in charge of the data transfer in this case.

If using the DMA technique, a context switch is also done in order to resume the execution of task `T1`. The difference is that when the data is ready, it is copied from the interface to the memory by the DMA controller while the CPU is still executing task `T1`. When the data transfer finishes, an interrupt is raised, but in this case the handler must not move the data, but simply wake up task `T2`. The system performance has been improved.

It must be taken into account that DMA only has advantages if the system supports multitasking or if it is single-task and the DMA controller can transfer data faster than the CPU.

The usual steps carried out using DMA I/O are enumerated below.

1. The CPU executing the operating system programs the DMA controller indicating the operation that must be done (reading or writing to the interface), the amount of data to transfer and the addresses involved in the transfer.

2. The CPU executing the operating system programs the interface of the peripheral device indicating that DMA must be used.

3. When the interface is ready to transfer, it notifies the DMA controller with a DMA request. The DMA controller takes the control of the buses and transfers the data between the interface and the memory.

4. At the end of the transfer, the CPU is notified using an interrupt.

The classic DMA controller is a device with the following characteristics:

- No intelligence. It simply copies the specified amount of data from some addresses to others.
- It is a shared device. All the interfaces with DMA capabilities share the DMA controller/s.

Nowadays, the classic DMA has evolved into what is known as bus mastering. With this technique, each interface has its own controller capable of acting as bus master, i.e., it is capable of controlling the buses and using them in the same way the CPU does. This controller associated with an interface is in charge of doing the data transfer between the interface and the memory, but it is more adaptable than the classic DMA controller because it is an intelligent device. It can even take the initiative to do transfers. At the end of the transfer the controller notifies the CPU using interrupts.

I/O processors

I/O processors are one step further in the evolution of I/O. The main idea is to extend the DMA controller giving it its own CPU. In this case, the main CPU will only indicate the commands to carry out. The I/O processor is in charge of all the I/O process, which often involves the execution of a program in the I/O processor. At the end of the transfer the I/O processor will notify that the operation is finished. These are interfaces with bus mastering capabilities that include one or more CPUs with high processing capacity. In some cases, they are more complex than the main CPU.

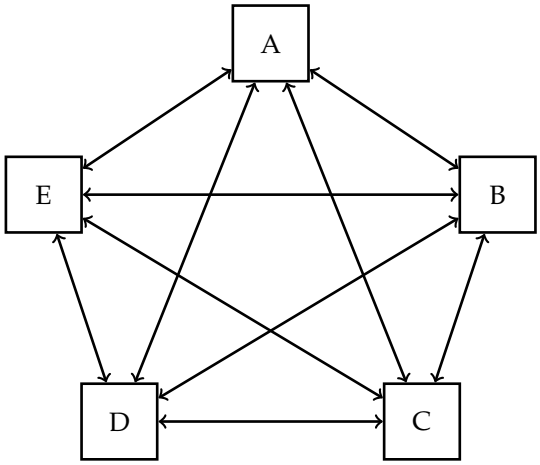
This is the case of graphics accelerators. They receive high level commands from the CPU such as moving an application window from one point on the screen to another. This way, the CPU does not have to update the state of the pixels of the screen occupied by the window before and after the movement. This effect is specially important in games and 3D animation programs.

Another example is image encondig/decoding cards and sound cards that free the CPU from the work of doing this processing, allowing it to do other tasks.

The current trend in I/O is using I/O processors, so the computer is converted into a multi-processor system composed of specialized processors.

4.2 The interconnection system

The components of the von Neumann machine exchange information through communication channels known as data, address and control buses. In general, the actual composition of the communication channels of the computer is known as the interconnection system of the computer. Nowadays, the interconnection system is heterogeneous, including different technologies, motivated in many cases by the diversity of interfaces and communication characteristics.



$$\#\,\text{channels} = \frac{\#\,\text{devices} \times (\#\,\text{devices} - 1)}{2}$$

Figure 4.7: Point-to-point connections between devices of the computer

4.2.1 Topologies

The computer is composed of devices that need to connect to each other. This is the case of the CPU, the memory system and the interfaces of peripheral devices. The computer also includes some control devices that are mapped in the address spaces like timers, interrupt controllers, etc. The interfaces of the peripherals need to be connected to peripherals as well.

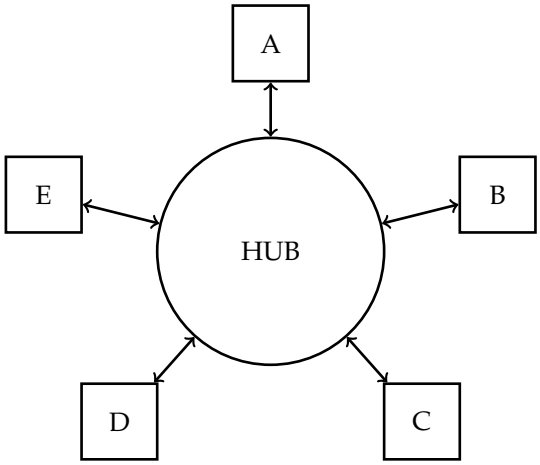
The easiest way to connect two devices is through a point-to-point channel. Using this kind of connection, if two devices `A` and `B` need to communicate, there is a dedicated channel that connects exclusively `A` and `B`. Figure 4.7 shows the point-to-point channels that are needed to communicate devices `A`, `B`, `C`, `D` and `E`.

This interconnection technique presents some advantages:

- Easy implementation.
- High bandwidth.
- Parallel transfers are possible.

However, it also presents some disadvantages:

- As seen in figure 4.7, 10 channels are required in order to connect 5 devices between them. If there were 20 devices, the number of channels needed would be 190. When there is a need to communicate many devices, the number of channels required increases rapidly, so does the cost.



channels = # of devices

Figure 4.8: Star topology among devices of the computer using a hub

- When a device wants to send the same information to more than one device, it needs to do the transfer as many times as destination devices.

Clearly, the most important disadvantage of point-to-point channels is the great number of channels required. A solution to this problem consists in using and deploying a star topology. Figure 4.8 shows the connection between devices `A`, `B`, `C`, `D` and `E` using a hub.

This topology requires a single channel for each device. Each time a device wants to send information to another, it sends it to the hub with the information that indentifies the destination device. The hub uses this information to forward information to the destination device.

The star topology has the following advantages compared to point-to-point channels:

- It requires a smaller amount of channels. There are as many point-to-point channels between the hub and the devices as devices.
- If the hub allows it, it is possible to transmit the information coming from a device to several destination devices at once.

The main disadvantage of this topology is the need of the hub. This hub must be fast enough and must allow simultaneous transfers between devices in order not to become a bottleneck. For instance, it would be advisable that the hub would allow simultaneous communication between devices `A` and `B` and devices `E` and `C` in figure 4.8.

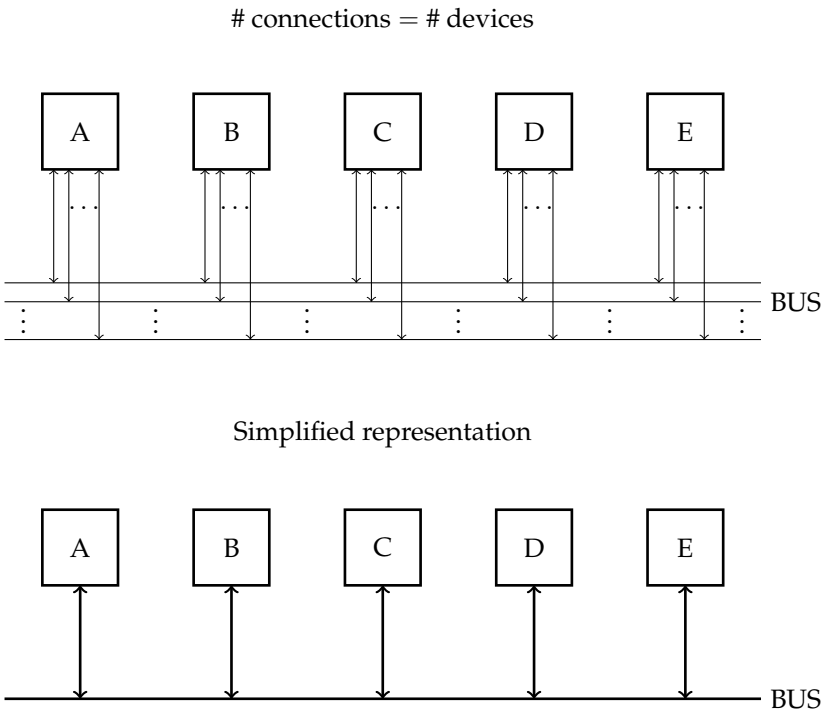


Figure 4.9: Bus topology among devices of the computer

Another connection topology is the bus topology. A bus is just a communication channel shared by several devices. Figure 4.9 shows the connection of devices `A`, `B`, `C`, `D` and `E` using a bus.

It can be seen that the number of connections to the bus is the same as the number of devices, which makes the bus connection technique inexpensive. Any two devices connected to the bus communicate with each other through it. When a device wants to send the same information to more than one destination device, it can send it only once in broadcast mode.

However, this technique has some disadvantages. Sharing the physical connection medium prevents parallel transfers, so the bus may become a bottleneck. Besides, the construction of the bus using parallel lines limits its frequency due to interference issues between lines and, therefore, limits its bandwidth.

The connection among all the devices of a computer through a single interconnection mechanism presents many disadvantages.

- The computer is composed of many devices with different capacities and characteristics. For instance, an mechanism appropriated for receiving data from a network interface with 1 Gbps of bandwidth is not appropriated for a keyboard interface, that has a bandwidth of several bytes per second.
- When a great number of devices are connected to a bus, its effective transfer capacity is reduced by the overhead produced by communication protocols. This problem also appears in hub topologies to a lesser extent.

In order to solve these issues, the three topologies are combined in practice. Figure 4.10 shows an example of this. It can be seen that there are hubs that manage not only point-to-point channels but also buses.

4.2.2 Characteristics

A set of cualitative characteristics of interconnection technologies are listed below.

- Synchronization. Technologies can be divided into synchronous and asynchronous. Synchronous technologies are characterized by the presence of a clock signal in their control lines. This clock signal is used to serialize the steps of communication. For example, using a point-to-point channel, the transmitter puts the information in the data lines and generates a clock signal so that the receiver can know where each data unit transmitted starts and finishes.

  Asynchronous technologies include additional control lines, such as a line that is activated when there are some data available to be read. In general, these control lines are activated by the transmitter and the receiver following a protocol that allows the communication between devices.

  In general, synchronous connections are simpler, which makes their implementation easier and provides a higher bandwidth. However, the length of the communication channel must be short or the clock frequency low, because the clock signal is deformed while it goes through the channel, which can make it unrecognizable. This degradation increases with the distance and the clock frequency in an effect known as clock skew.

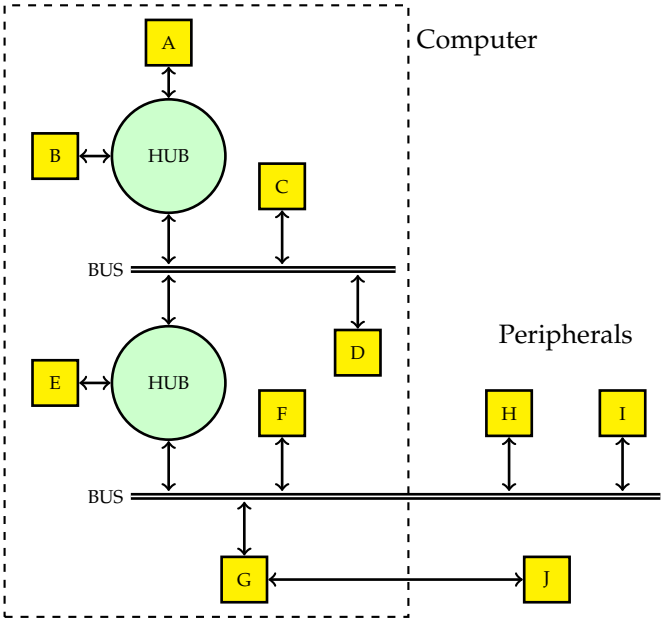


Figure 4.10: Combination of topology techniques

- Data size. The main function is transferring data. Each read or write operation involves the transfer of a data item of a certain size in bits.
- Serial or parallel. A parallel connection has as many data lines as bits in a transferred data item. On the contrary, a serial connection has less number of lines, usually one, and all the bits that compose the data item are transmitted sequentially. For instance, if the size of the data is 32 bits and the connection has only one data line, the 32 bits have to be transferred sequentially over the same data line.
- Multiplexed or not multiplexed. With the objective of reducing the number of lines, it is common to use certain lines for several functions depending on the moment. For instance, it is common to multiplex data and address lines. This way, during a write operation an address is put on the lines and, afterwards, the data to be written is put on the same lines. Multiplexing reduces the number of lines, also reducing the cost and improving reliability. However, it usually implies less bandwidth compared to a not multiplexed version.
- Theoretical bandwidth. It represents the maximum transfer rate in ideal conditions. It is usually expressed in Megabits per second (Mbps) or Gigabits per second (Gbps).
- Maximum length. The maximum physical length of the connection cannot be arbitrary. The speed propagation of the electric signals through the wire is limited: they move a little slower than the speed of light.[3] This effect is specially

[3] Almost 3 ns are required to travel a meter.

critical in synchronous connections because of the clock skew effect. It must be taken into account that wires and printed circuit tracks work as antennas that emit and receive electromagnetic noise, which can produce incorrect signals in the wires. The mentioned effects are bigger the longer the wire is.

- Plug and Play (PnP). PnP technologies define mechanisms to know the identity of the connected devices and the hardware resources they need (positions in the address spaces, interrupt request lines, etc.). The idea is that the system automatically configures these devices when they are connected, so that the process is transparent to the user in order to avoid conflicts.
- Hot plug. There are interconnection technologies that allows the connection of devices while the system is running.

4.2.3 PCI Express (PCIe)

All technologies used in the computer, including the most successful must be replaced by new technologies at some point. The main cause relies on the continuous improvement on the speed of computer elements.

This is the case of the PCI specification, precursor of PCI Express. The initial specification of PCI had improved its capabilities for many years. However, its limitations have made it almost disappear from the market. The gap left by PCI was filled by another connection technology called PCI Express launched in 2002.

PCI and PCI Express have many differences at a physical level. However, at the beginning they were compatible at a software level. The main idea was that all the software developed for PCI devices could be used by PCI Express devices. This general idea of backward compatibility is very important in the computer industry each time a new technology is launched or an existing technology is improved.

Next, some of the PCI Express 1.0 characteristics at a physical level are enumerated as an example of an interconnection system:

- PCI Express is a point-to-point connection. Each PCI Express connection is a single bidirectional connection between two devices.
- PCI Express is a serial connection. Each byte is sent bit by bit sequentially using two lines in a differential way.[4] However, each PCI Express connection can have several serial lanes working in parallel.
- Each PCI Express lane can transmit 2 Gbps in one direction and 2 Gbps in the opposite direction at the same time, so it has a theoretical bandwidth of 4 Gbps.
- Each PCI Express connection can have 1, 2, 4, 8, 12, 16 or 32 lanes, according to 1x, 2x, 4x, 8x, 12x, 16x or 32x suffixes. Therefore, the maximum theoretical bandwidth of PCI Express is 128 Gbps. Devices with different number of lanes can be connected, since a handshake takes place in order to decide the number

[4] One bit is transmited as a difference of voltage between two lines. This way the transmission is immune to interferences, because they affect both lines similarly, keeping the voltage difference between them and therefore information.

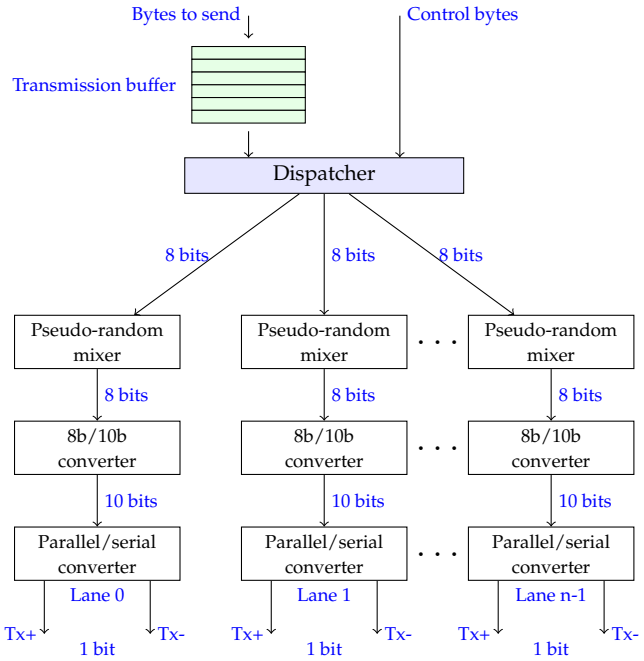


Figure 4.11: Processing of a packet to be sent by a PCI Express device

of lanes used before starting the communication. For instance, if a 4-lane device is connected to other with 12 lanes, the communication will be carried out using 4 lanes.

- The number of pins used by PCI Express is low. PCI Express uses only 2 lines per lane. This means a great reduction in terms of cost as well as an improvement in its reliability.

Figure 4.11 shows how the transmitter of a PCI Express device processes a data packet that is sent to another PCI Express device. The processing carried out by the receiver is the inverse.

The packet is a set of bytes stored in a transmission buffer of the transmitter. It must be taken into account that, together with the data, some control information is also sent. There is a dispatcher in charge of selecting a byte of data or a byte of control, progressively emptying the transmission buffer in the first case.

Each byte of the byte sequence generated by the dispatcher (data and control bytes) is dealt among the communication lanes.

Each lane has a pseudo-random mixer, an 8b/10b converter and a parallel to serial converter.

The pseudo-random mixer changes the order of the bytes it receives in order to avoid patterns of repeated bytes, which distributes the electromagnetic radiation of `Tx+` and `Tx-` lines of the lane in a wide range of frequencies.[5]

[5]This way, the electromagnetic radiation that near conductors receive is similar to white noise, which is less harmful than noise focused in some few frequencies.

Version	Year	Theoretical bandwidth	Effective bandwidth	Encoding
PCIe 1.0	2003	2.5 Gbps	2 Gbps	8b/10b
PCIe 1.1	2005	2.5 Gbps	2 Gbps	8b/10b
PCIe 2.0	2007	5 Gbps	4 Gbps	8b/10b
PCIe 2.1	2009	5 Gbps	4 Gbps	8b/10b
PCIe 3.0	2010	8 Gbps	7.88 Gbps	128b/130b
PCIe 3.1	2014	8 Gbps	7.88 Gbps	128b/130b
PCIe 4.0	2017	16 Gbps	15.75 Gbps	128b/130b
PCIe 5.0	2019	32 Gbps	31.5 Gbps	128b/130b

Table 4.1: PCIe versions and bandwidths

The 8b/10b converter transforms each byte in a sequence of 10 bits. The main objective of this transformation is to make the distance between the transitions from `0` to `1` and from `1` to `0` not more than a certain number of bits. The reason is that these transitions are used to synchronize the receiver clock with the 2.5 GHz clock of the transmitter. Without this encoding, the distance between consecutive transitions can be randomly long such as when there are many consecutive 00h bytes or FFh bytes. Taking into account the transmitter clock frequency and the fact that in each clock cycle a bit is transferred, the theoretical bandwidth (one way) is 2.5 Gbps. However, the effective bandwidth is 2 Gbps due to the two bits added during the 8b/10b conversion.

The parallel to serial converter receives words of 10 bits and transmit their bits serially using lines `Tx+` and `Tx-`. A logical one is a positive voltage difference between `Tx+` and `Tx-`, while a logical zero is a negative voltage difference.

Over the years, the PCIe technology has evolved and improved its capabilities in successive versions that provide great improvements. In table 4.1 the main versions of PCIe launched until the writing of this text are listed together with their one-way bandwidth for one lane.

4.3 Peripheral devices

This section studies the devices that enable the computer to communicate with its environment. These devices are called peripherals and communicate with the computer through interfaces whose work is to adapt the operational characteristics of each peripheral to the digital operation of the computer.

4.3.1 Introduction

Although it is hard to establish a classification of peripherals due to the amount of different criteria that could be used, the most general classification is based on the direction of the information flow. Periherals can be classified as follows:

- **Input peripherals**. Their main task is to enter information in the computer. The most used are the keyboard and the mouse. However, some other devices belong to this group: magnetic stripe readers, optical detectors (marks, stripes, dots or characters), voice recognition devices, light pens, joysticks, scannners, digital cameras, etc.
- **Output peripherals**. Their main task is to present information to the user. The most common output peripheral is the screen, but there are other devices in this group: printers, plotters, any kind of specific display, augmented reality devices, etc.
- **Input/Output peripherals**. These peripherals are used both to enter information in the computer and to extract information from the computer. In this group the most representative elements are communication peripherals like network cards. Devices such as sound cards and touchscreens also belong to this group.
- **Auxiliary memory or storage peripherals**. This group of peripherals is a particular case of input/output peripherals because they are used both to enter and to extract information to/from the computer. They are used as storage devices. Magnetic disks, optical disks (CD, DVD and Blu-ray), magnetic tapes, magnetic-optic disks, USB drives and SSD devices belong to this group.

Storage peripherals (hard disk drives and solid state drives) will be studied in more depth due to their importance in current computers.

4.3.2 Hard disk drive

Hard disk drives, or HDDs, are devices that enable the storage of information in a non-volatile way, which means that information persists even without a power supply. These devices are composed of several platters linked by a shaft spinning together at speeds that can reach 15 000 rpm. The surfaces of these platters are coated with ferromagnetic material that can store information. There is a magnetic head in each side of a platter that can read and write information. The heads are attached to an arm connected to a mechanic structure that is used to move them over the platter radially outward and inward. This radial movement, together with the rotation of the disk, allows the heads to reach any point of the platters. A scheme of the components of a disk is shown in figure 4.12.

Information organization

The organization of the information determines how the information must be placed so it can be accessed. This organization is shown in figure 4.13. The following elements can be identified in the figure:

- Tracks. They are concentric circles on the surface of a platter. They are numbered from 0 to $(\text{Track count} - 1)$. Number 0 is reserved for the most external track.

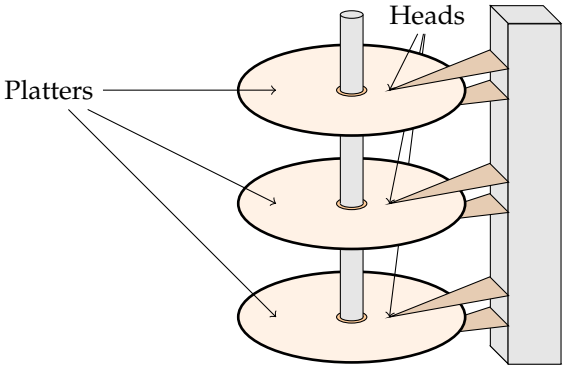
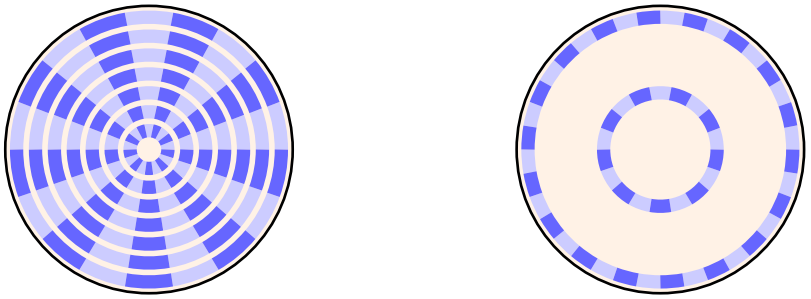


Figure 4.12: Parts of a hard disk drive



(a) Constant number of sectors

(b) Constant information density

Figure 4.13: Distribution of information in a platter

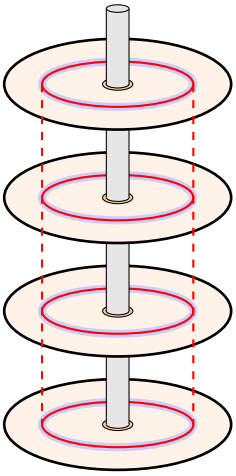


Figure 4.14: Cylinder definition in a hard disk drive

- Each of these tracks is divided into small segments called sectors. Each sector contains a certain amount of bytes, usually 512 or 1024.

  The sector is the minimum unit that can be read or written in a hard disk (even when only one byte is needed, the whole sector is read or written).

- Information can be stored in one or in both faces of a platter. Technically, they are called surfaces.

- A cylinder is defined as the set of tracks with the same number in all surfaces. That is, all the tracks that are in the same vertical axis. Figure 4.14 shows a representation of a cylinder.

Some operating systems define another unit called cluster. A cluster puts together two or more sectors and represents the minimum size of disk that can be used to perform a read or write operation in the file system.

During the low level formatting operation, the heads write information necessary to identify tracks and sectors on each platter. This information will be used to locate information in the disk when reading and writing.

After the low level formatting, the high level formatting is performed. This operation basically consists in copying in a specific location of the disk the information needed to boot the computer (master boot record and partition table) and a data structure (file system) that can be used as an index to find files in the disk. In this table, the number of the cluster where each file starts is stored.[6] Clusters are numbered consecutively and each cluster has its own surface, track and sector coordinates. These values are sent to the disk controller to access the contents of the cluster.

Finally, a consideration about the number of sectors that exist in a platter must be made. Figure 4.13a shows how the previous definition means that there are the

[6] If the file takes up more than one cluster, the following cluster number is stored at the end of each cluster. The last cluster ends with a special sequence that indicates the end of the file.

same number of sectors in each track. The same amount of information is stored (512 bytes) in each sector, so information density is lower in the most external sectors, which are bigger, whereas information density increases in the inner sectors. This is a waste of ferromagnetic material. An alternative consists in keeping information density constant. This way, the most external tracks would have more sectors than inner tracks as can be seen in figure 4.13b. This way of storing information makes the most of the magnetic surface.

Basic parameters

The most important parameters to consider in a disk are the following:

1. **Capacity**. It is the number of bytes that can be stored in the drive:

$$\text{Total capacity} = \frac{\text{Bytes}}{\text{Sector}} \times \frac{\text{Sectors}}{\text{Track}} \times \frac{\text{Tracks}}{\text{Surface}} \times \frac{\text{Surfaces}}{\text{Unit}}$$

   The actual capacity is slightly lower because a small part of the capacity is dedicated to store the information used for locating the stored data.[7]

2. **Access time**. It is the amount of time needed to access a piece of information in a read or write operation. This requires a sequence of actions that have their own cost in terms of time:

   - Placing the read/write head over the right track without oscillations in a stable way. The time needed to do this is called track seek time.
   - Waiting until the starting point of the required sector passes under the read/write head. This time is called rotational delay or latency. The latency depends on the rotational speed of the drive; statistically, this is estimated as one half of the rotational period.
   - Transferring (reading or writing) the bytes while they are passing under the head. The time needed to do this is called transfer time.

   Considering these three steps, the access time can be quantified using this expression:

$$\text{Access time} = \text{Seek time} + \text{Latency} + \text{Transfer time}$$

3. **Information density**. Once the physical dimensions of the disk and its storage capacity are known, some metrics can be obtained that represent the amount of information that can be stored related to the physical space it uses. These metrics can be seen in figure 4.15 and are defined as follows:

   TPI: Tracks per inch. It indicates the number of tracks stored per unit of radial length of the platter.

   BPI: Bits per inch. It indicates the number of bits stored per length unit of the track. If the number of sectors is constant, this density increases for inner tracks.

[7] Data structures of the file system.

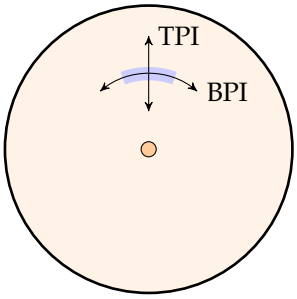


Figure 4.15: Definition of storage density metrics

4. **Buffer**. Hard disk drives have a component called disk buffer. When the drive receives a read request, the drive accesses the track where the information is stored and instead of reading just the requested sectors, it copies all the information of the track to the disk buffer and the request is served from there. The objective of the buffer is to act as a cache of the disk drive. Again, the locality principle comes into play.

4.3.3 Solid state drive

A solid state drive or SSD is a device that is used to store information in a persistent way like hard disk drives. Unlike HDDs, the storage in a SSD device is based on semiconductors. This is the same principle used in the main memory of the computer: it has memory cells based on transistors that can store one bit.

Current SSD devices use flash memory, that are based on NAND gates as basic cells. NAND cells are slower than DRAM memory, used in the main memory of the computer, but as they do not have any mechanic part they are still much faster than hard drives. Depending on the technology applied, one cell can store one, two, or three bits (the voltage range would be divided into 2, 4 or 8 levels). The capacity increase achieved with each technology comes with a speed cost: the higher the amount of bits per cell, the slower the access to information due to the need to distinguish from one voltage level to another.

The technology used for storing information in SSD devices is the same used as in USB pen drives. The differences are their capacities and the interfaces they use to be connected to the computer.

Information organization

NAND cells are organized in a matrix in a SSD device. Each row of the matrix is known as a page and the whole matrix is called block. Usual page sizes are 2 KiB, 4 KiB, 8 KiB or 16 KiB and each block is composed of 128 or 256 pages. This means that the size of a block goes from 256 KiB to 4 MiB.

The SSD operation is related to the organization of information, but there are differences between reading, writing and overwriting.

- The smallest piece of information that can be read from a SSD drive is a page. The reading process is very fast.
- Writing operations are also carried out at page level when the page is empty. Even being slower than reading, writing is a very fast process.
- A different case is when a writing operation must be done in a page that is not empty (it was previously written), for instance, when updating the content of a file. In this case the process is not the same, as the previous information must be erased at block level. An option is to copy the content of the block to main memory, erase the block in the SSD, perform the update and write the whole block back to the SSD. Another option is to transfer the content of the modified block to an empty block and set the old one as available.

Overwritings and lack of space when the drive is getting full are the causes that make these devices slower as time goes by. The controller of the SSD drive must balance writings to avoid this performance degradation as much as possible.

4.3.4 Comparison between hard disk drives and solid state drives

The main difference between both types of devices is that HDDs are based on the spinning movement of the disk and the movement of their reading heads. On the other hand, SSDs do not have any mobile part. This aspect influences in the following ways:

- SSDs are faster than HDDs, specially for reading operations, as they do not have mobile parts. This is the reason why SSDs are used to store the operating system, since mainly reads are carried out during the booting process.
- SDDs are much quieter than HDDs, as they do not have any mobile part.
- SSDs are also more reliable in hostile environments. Since they have no mobile parts, they are not affected by vibrations, humidity, etc. In fact, the first usage of SSDs was in industrial environments where HDDs were not reliable.
- SSDs are much smaller than HDDs with the same capacity because they do not need mechanical support.

HDDs also have some advantages compared to SSDs:

- HDDs have higher storage density, which allows them to have higher capacity than SSDs (at the same cost). The persistence of information is compromised when the size of the NAND gates is reduced in the SSD drives.
- The cost per GiB in HDDs is much less than in SDDs, so storage in HDDs is cheaper.
- Finally, the writings and overwritings in SSDs affect the life cycle of their blocks. However, the number of operations needed to cause a fault is increasing with the improvements in SSD controllers.

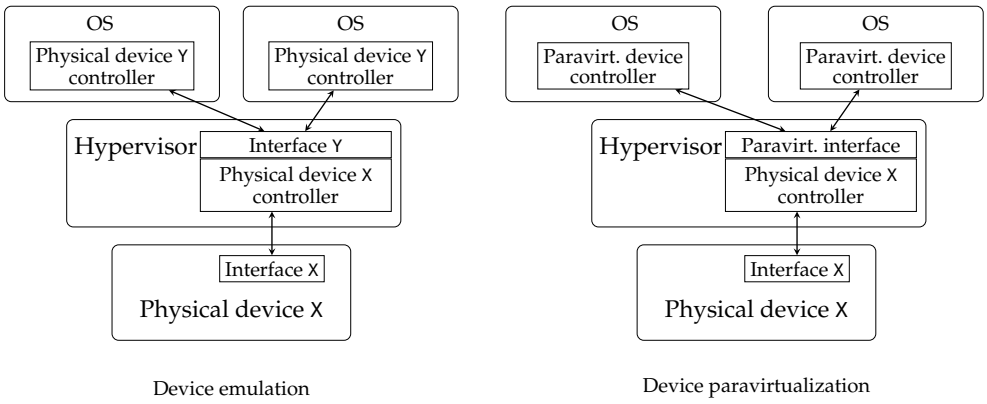


Figure 4.16: Software-based I/O virtualization

4.4 I/O virtualization

Virtual machines must share the access to physical I/O devices in a virtualized system, ideally without causing security or performance issues. The operating system executed in each virtual machine must have the "illusion" of having total control over devices through their interfaces, as it happens in non-virtualized systems.

For instance, in a server with a physical network interface card (NIC) and several virtual machines, each virtual machine should has its own virtual NIC used to send messages without interfering with the virtual NICs of the other virtual machines. Operations requested to the virtual NICs must be served by the physical NIC, so this device is being virtualized.

As with CPU virtualization, I/O virtualization can be done using software, applying the following techniques, which are illustrated in figure 4.16.

- Device emulation. In this option, the I/O virtualized devices are emulated by software. The device controller or driver used by the operating system is the driver of a physical interface. For example, if a network interface is virtualized in a way that it is compatible with the Intel 82540EM interface, the operating system installed in the virtual machine will use the same driver that would be used on a physical machine using the Intel 82540EM interface.

  In order to emulate devices, the hypervisor uses the trap-and-emulate technique. Any read or write operation over interface Y carried out by driver Y in the figure (executed by the operating system) is trapped by the hypervisor and translated into operations over physical interface X. Similarly, any operation that physical interface X should notify must be transfered to driver Y through the hypervisor. Apart from the overhead produced by the translation between two different interfaces, X and Y, these operations imply several context switches between the operating system and the hypervisor, making this virtualization technique poor in terms of performance.

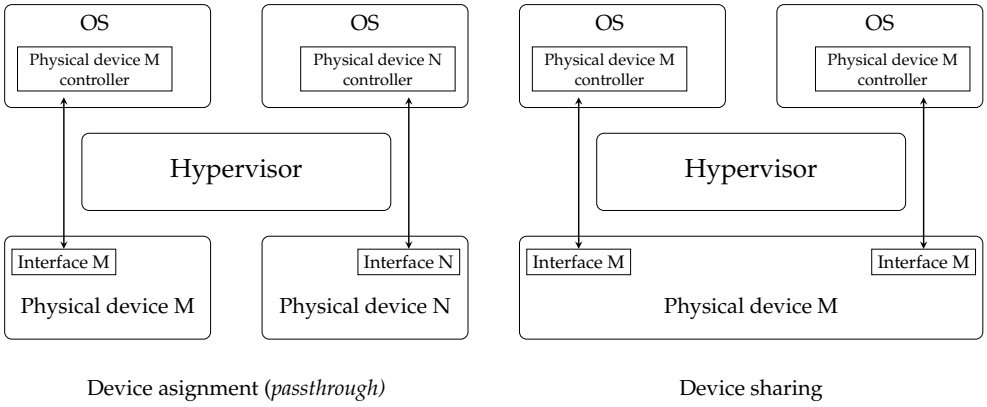
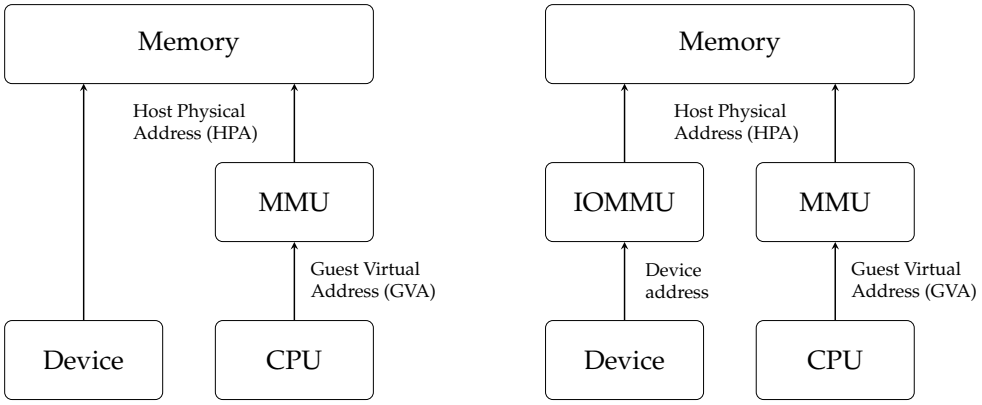


Figure 4.17: Hardware-assisted I/O virtualization

- Device paravirtualization. This is a variant of device emulation, but in this case the virtual machine exposes a paravirtualized interface to the operating system. This is an interface that does not match any physical interface and that is adapted for coordinating with the hypervisor more efficiently. This alternative improves performance compared to device emulation and does not require any change in the operating system, in contrast to what was studied about CPU paravirtualization. However, the main disadvantage is that it requires a specific driver that must be installed in the operating system. Besides, the performance is still low, although slightly better than the performance obtained with device emulation.

In order to reduce performance issues, some hardware-assisted I/O virtualization techniques have been developed. Their main objective is to communicate virtual machines and I/O devices with the minimum hypervisor intervention. These techniques are described bellow and are illustrated in figure 4.17.

- Device assignment or passthrough. The hypervisor assigns the physical device to the virtual machine so that it can access directly to the physical device without any intervention from the hypervisor, which implies a great improvement in performance. A type 1 hypervisor using this technique does not need to support the device driver, which expands the scope of compatible hardware. In the same way, the host operating system does not need to support the device driver with type 2 virtualization, only the guest operating system does. The only issue of this technique is that a physical device can be assigned to a single virtual machine. For instance, this cannot be applied if two virtual machines should share the access to a single physical NIC.
- Device sharing. This is a variant of the device assignment that uses the direct access to the device and also shares it between virtual machines. The requirement is that devices must expose multiple interfaces, so that each one can be directly assigned to a different virtual machine. This is the case of PCI Express devices that have the Single Root I/O Virtualization (SR-IOV) extension,



(a) Without device addresses remapping (b) With device addresses remapping

Figure 4.18: Remapping of DMA devices

which means that they have multiple PCI Express independent interfaces. For instance, a PCI Express NIC with four interfaces could be assigned to four virtual machines that will perceive it as their own independent NIC.

The implementation of device assignment, or sharing, presents several problems that must be solved by hardware. As an example, a network interface using DMA will be used (figure 4.18a). At a given time, the operating system executed in a virtual machine programs the network interface in order to send a packet stored in memory at a physical address. In the context of a virtualized system, this physical address is the guest physical address or GPA. The hypervisor will not intercept this configuration operation, so the network interface will be configured with this address and will read the packet starting from it. The network interface works using real physical addresses so it will access to a host physical address or HPA equal to the GPA given by the operating system, hence producing a system malfunction.

In order to solve the previous issue among others, modern computers based on x86 architecture include a I/O Memory Management Unit (IOMMU), analogous to the MMU for the virtual memory. It is located between the I/O interfaces and the memory. The IOMMU translates addresses generated by a DMA device into host physical addresses (HPA) before they reach the memory. This technique is called DMA remapping. Figure 4.18b shows this new situation. Each device has its own associated table, analogous to the page table, that the IOMMU uses to translate GPA addresses into HPA addresses. In order to speed up the translation of GPA into HPA, devices also have translation caches called IOTLBs, analogous to the TLB used for paging.

The IOMMU provides a huge improvement of performance in virtualized environments, but despite what happens with other hardware-assisted I/O virtualization techniques, it requires support from the operating system. However, this is not a problem because the IOMMU is also used in non-virtualized environments to add some security to devices. It is possible to set memory access restrictions for I/O de-

vices as it was done in page tables used by the MMU, improving the security of the system against a bad programmed or malicious device.

Appendix A

PyMIPS64 instruction set

Naming conventions

`RsX`	64-bit integer register used as source operand.
`Rd`	64-bit integer register used as destination operand.
`Ri`	64-bit integer register used for computing a memory address.
`FsX`	64-bit floating-point register used as source operand.
`Fd`	64-bit floating-point register used as destination operand.
`imm_X`	Immediate value of X bits.
`sign_ext`	Sign extension up to 64 bits.
`zero_ext`	Sign extension up to 64 bits, adding zeros on the left.
`[]`	Memory position.

The instruction set shown here is the one supported by the simulator WinMIPS64, which is a simplification of the one supported by MIPS64. In addition, some instruction codes are different. These differences will be highlighted in the following tables.

Arithmetic instructions

Instruction	Operation	Description
nop		No operation
daddi Rd,Rs,imm_16	Rd←Rs+sign_ext(imm_16)	Adds the doble word and the immediate value, trapping on overflow
dadd Rd,Rs1,Rs2	Rd←Rs1+Rs2	Adds double words, trapping on overflow
dsub Rd,Rs1,Rs2	Rd←Rs1−Rs2	Subtracts double words, trapping on overflow
dmul[1] Rd,Rs1,Rs2	Rd←Rs1×Rs2	Multiplies double words
ddiv[1] Rd,Rs1,Rs2	Rd←Rs1÷Rs2	Divides double words
add.d Fd,Fs1,Fs2	Fd←Fs1+Fs2	Adds double-precision floating-point values
sub.d Fd,Fs1,Fs2	Fd←Fs1−Fs2	Subtracts double-precision floating-point values
mul.d Fd,Fs1,Fs2	Fd←Fs1×Fs2	Multiplies double-precision floating-point values
div.d Fd,Fs1,Fs2	Fd←Fs1÷Fs2	Divides double-precision floating-point values

Logical instructions

Instruction	Operation	Description
andi Rd,Rs,imm_16	Rd←Rs AND zero_ext(imm_16)	Logical AND with immediate value
ori Rd,Rs,imm_16	Rd←Rs OR zero_ext(imm_16)	Logical OR with immediate value
xori Rd,Rs,imm_16	Rd←Rs XOR zero_ext(imm_16)	Logical XOR with immediate value
and Rd,Rs1,Rs2	Rd←Rs1 AND Rs2	Logical AND
or Rd,Rs1,Rs2	Rd←Rs1 OR Rs2	Logical OR
xor Rd,Rs1,Rs2	Rd←Rs1 XOR Rs2	Logical XOR
slt Rd,Rs1,Rs2	If Rs1<Rs2 : Rd←1 else : Rd←0	Signed integer comparison

[1] In MIPS64, the instructions `dmult`, `dmultu`, `ddiv` and `ddivu` only have two source operands and store the result in two 64-bit registers of the CPU: `HI` and `LO`. The multiplication results in a 128-bit value that is stored in both registers, while the quotient of the division is stored in `LO` and the reminder in `HI`. These registers can be accessed with the instructions `mfhi Rd` and `mflo Rd`. In the pyMIPS64 simultator, the result of the multiplication is 64 bit long and only the quotient of the division is stored.

Load instructions

Instruction	Operation	Description
ld Rd,imm_16(Ri)	Rd←[Ri+sign_ext(imm_16)]	Loads double word
lb Rd,imm_16(Ri)	Rd←sign_ext([Ri+sign_ext(imm_16)]$_{[7..0]}$)	Load byte with sign extension
lh Rd,imm_16(Ri)	Rd←sign_ext([Ri+sign_ext(imm_16)]$_{[15..0]}$)	Loads half word with sign extension
lw Rd,imm_16(Ri)	Rd←sign_ext([Ri+sign_ext(imm_16)]$_{[31..0]}$)	Loads word with sign extension
lbu Rd,imm_16(Ri)	Rd←zero_ext([Ri+sign_ext(imm_16)]$_{[7..0]}$)	Loads byte without sign extension
lhu Rd,imm_16(Ri)	Rd←zero_ext([Ri+sign_ext(imm_16)]$_{[15..0]}$)	Loads half word without sign extension
lwu Rd,imm_16(Ri)	Rd←zero_ext([Ri+sign_ext(imm_16)]$_{[31..0]}$)	Loads word without sign extension
l.d Fd,imm_16(Ri)	Fd←[Ri+sign_ext(imm_16)]	Loads double-precision floating-point value

Store instructions

Instruction	Operation	Description
sd Rs,imm_16(Ri)	[Ri+sign_ext(imm_16)]←Rs	Stores double word
sb Rs,imm_16(Ri)	[Ri+sign_ext(imm_16)]←Rs$_{[7..0]}$	Stores byte
sh Rs,imm_16(Ri)	[Ri+sign_ext(imm_16)]←Rs$_{[15..0]}$	Stores half word
sw Rs,imm_16(Ri)	[Ri+sign_ext(imm_16)]←Rs$_{[31..0]}$	Stores word
s.d Fs,imm_16(Ri)	[Ri+sign_ext(imm_16)]←Fs	Stores double-precision floating-point value

Jump instructions

Instruction	Operation	Description
j imm_26	PC←PC$_{63..28}$|imm_26<<2	Jumps to an address within a 256 (2^{28}) MiB region

Branch instructions

Instruction	Operation	Description
beq Rs1,Rs2,imm_16	If Rs1=Rs2 : PC←PC+sign_ext(imm_16<<2)	If equal, branches to an address within a ±128 kiB region
bne Rs1,Rs2,imm_16	If Rs1≠Rs2 : PC←PC+sign_ext(imm_16<<2)	If not equal, branches to an address within a ±128 kiB region
beqz Rs,imm_16	If Rs=0 : PC←PC+sign_ext(imm_16<<2)	If register is 0, branches to an address within a ±128 kiB region
bnez Rs,imm_16	If Rs≠0 : PC←PC+sign_ext(imm_16<<2)	If register is not 0, branches to an address within a ±128 kiB region

Appendix B

The control unit

This appendix describes the design of the control unit required by the datapath presented in this book.

B.1 Single-cycle control unit

The control unit governs all the operations in the datapath. There are mainly two types of implementation of control units: wired and microprogrammed. In a wired control unit each control signal is obtained as a result of a logical function; the inputs of this function are the instruction code and the flags generated by the ALU. Its design and implementation are similar to those of any combinational digital system. In contrast, the status of all signals is represented through a control word in a microprogrammed control unit. The control unit has an internal memory storing control words; each instruction code references one of these words (in the single-cycle scenario).

Regardless of the type of implementation, the purpose of the control unit is the same: set and reset the control signals that govern the execution of instructions in the datapath. For simplicity, only the design using the truth table of the control unit is covered. Its implementation is out of the scope of this book.

A technique commonly used to improve the performance of the control unit, and the performance of the CPU as a consequence, consists on dividing the control in several units. Thus, a reduced control unit can be proposed to manage the ALU. This control unit will manage the control signals (4 lines) reaching the ALU depending on the instruction to be executed.

Load and store instructions require the ALU to compute the memory address to read or write and add the base register and the immediate value. Arithmetic and logical operations are R-type instructions with the same value in the `op` field; they only differ in the value of the field `function`, as can be seen in figure 2.5. They use the ALU depending on the specific operation (add, subtract, AND, OR, etc.). Finally, the branch instruction uses the ALU to compare the values of two registers and check whether the condition is met.

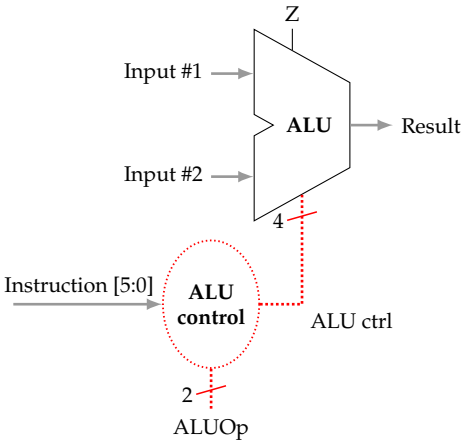


Figure B.1: ALU control unit

Instruction	Op	Function	ALU operation	ALUOp	ALU ctrl
ld	load	–	add	00	0010
sd	store	–	add	00	0010
beq	branch on equal	–	subtract	01	0110
daddi	add imm. (I)	–	add	00	0010
dadd	R-type	100000	add	10	0010
dsub	R-type	100010	subtract	10	0110
and	R-type	100100	AND	10	0000
or	R-type	100101	OR	10	0001
slt	R-type	101010	set on less than	10	0111

Table B.1: ALU operation and control signals

Under these circumstances a two-line control signal can be defined, called `ALUOp`, allowing the main control unit to ask the control unit of the ALU to perform an operation: `00` for addition; `01` for subtraction; `10` for the operation referred in the `function` field of an arithmetic-logic instruction code. The reduced control unit for the ALU is shown in figure B.1. Furthermore, table B.1 lists the instructions of the MIPS64 implemented subset that use the ALU, as well as the values of the control signals required depending on the `op` and `function` fields of the instruction code.

The design of the main control unit must take into account all the control signals added to the datapath. Many of them govern multiplexers to select from different data sources. Being two-input multiplexers, only one-line control signal is required. Table B.2 lists the control signals in the datapath shown in figure 2.16, as well as the effect produced when set to `1` or `0`.

The values of the control signals can be inferred based on the behaviour of the datapath in the execution of instructions described in section 2.2.2. Table B.3 shows the values of the control signals in the implemented instructions (X means that the value of the control signal can either be `1` or `0`). It must be taken into account that `Branch` signal is considered instead of the generic `PCSrc` signal in the table.

Signal	Effect when reset (0)	Effect when set (1)
RegDst	The number of the destination register is in bits 20 to 16	The number of the destination register is in bits 15 to 11
RegWrite	None	The register specified in Write register is written with data from Write data
ALUSrc	The second operand of the ALU is in the second register	The second operand of the ALU is the immediate value extented to 64 bits
PCSrc	The value of PC is computed as PC + 4	The value of PC is computed as the target address of the branch
Jump	The PC register is loaded with the value already computed	The PC register is loaded with the target address of the jump
MemRead	None	The memory address in Address is read
MemWrite	None	Write data is written to the memory address specified in Address
MemToReg	The value to be written to the register file comes from the ALU	The value to be written to the register file comes from the data memory (Read data)

Table B.2: Control signals generated by the control unit

Inst.	RegDst	RegWrite	ALUSrc	Branch	Jump	MemRead	MemWrite	MemToReg	ALUOp
R-type	1	1	0	0	0	0	0	0	10
ld	0	1	1	0	0	1	0	1	00
sd	X	0	1	0	0	0	1	X	00
beq	X	0	0	1	0	0	0	X	01
daddi	0	1	1	0	0	0	0	0	00
j	X	0	X	X	1	0	0	X	XX

Table B.3: Values of the control signals for each instruction

Therefore, the datapath, together with the required control, is shown in figure B.2. Next, the control signals generated in the execution of the following instruction are shown as an example:

```
dadd r2, r3, r6
```

The execution of this instruction requires only one clock cycle, when the next operations are carried out:

- When the rising edge in the clock cycle is produced, the next instruction to be executed is fetched from the instruction memory using the address contained in PC.
- The instruction code is copied into lines Instruction.
- Registers are read (r3 and r6) from the register file (Read #1 and Read #2).
- The values of the registers are available in Read data 1 and Read data 2.
- ALUSrc line is set to 0 to indicate that the instruction operates with the contents of the two registers.
- The value 10 is sent to lines ALUOp to notify the ALU control unit that it is an R-type instruction. According to the function field in the instruction code, the ALU control unit sends the value 0010 to the ALU to add the operands.

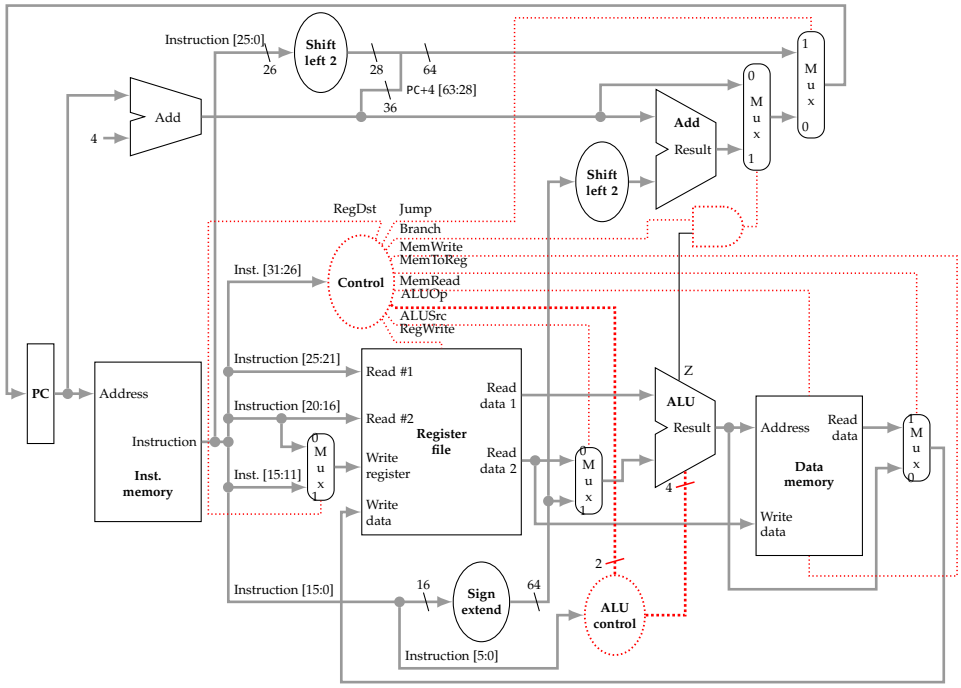


Figure B.2: Datapath supporting R-type, load/store, **`beq`** branch and `j` jump instructions, together with the control logic

- `MemToReg` line is set to `0` to indicate that the value to be written to the register file is the result of the ALU.
- `RegDst` line is set to `1` to indicate that the number of the destination register is encoded in bits 15 to 11 of the instruction code.
- `RegWrite` line is set to `1` to write to the register file.
- At the same time, `Branch` and `Jump` lines are set to `0` for the program counter to be incremented in 4 units, as well as the lines associated with the data memory, `MemRead` and `MemWrite`, since the memory is not used.

B.2 Pipelined control unit

The implementation of the control unit takes the single-cycle control unit and the datapath shown in figure 2.20 as a starting point. Figure B.3 shows the datapath with the control signals identified, as well as a small modification to correctly execute instructions that store their result in the register file (load, arithmetic and logical instructions). These instructions require the field of the instruction code that encodes the destination register during stage `WB`. For this reason, it is necessary to propagate along the datapath, using the pipeline registers, the fields of the instruction code

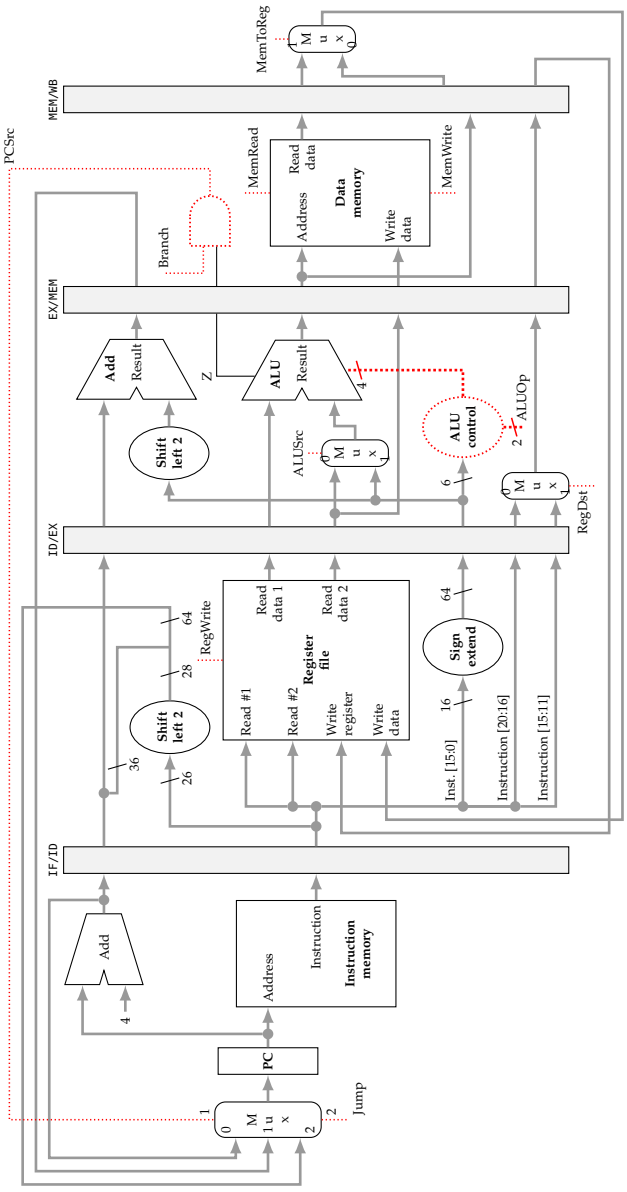


Figure B.3: Pipelined datapath with control signals

needed by stages after ID. This modification increases the width of segmentation register ID/EX from 256 to 266 bits.

According to this representation of the datapath, almost all instructions behave in the same way during the first two stages (IF and ID). Instructions are fetched in IF and decoded in ID. For this reason, the instruction to be executed is unknown in the first stage and partially during the second stage. Only during ID the type of

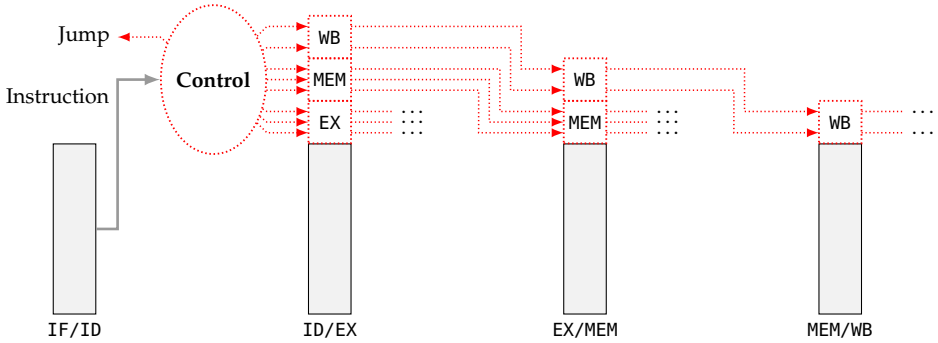


Figure B.4: Control signals propagation through the pipelined datapath

instruction to be executed is identified. An exceptional case is the jump instruction. The jump address target is computed during `ID` and, as soon as the instruction is decoded and the jump is identified, the program counter register is modified. However, taking into account that the instruction is not identified until it is decoded, jump target address is calculated for all instructions, regardless they are jumps or not.

Control signals generated to execute an instruction must be propagated through the datapath in the same way as data. Thus, it is necessary to increase the sizes of pipeline registers to include control signals, which are generated during the decoding stage (`ID`). Since stages `IF` and `ID` are the same for all the instructions, except for jumps, most control signals are propagated through pipeline registers `ID/EX`, `EX/MEM` and `MEM/WB` as depicted in figure B.4.

The resulting pipelined datapath, including the control unit, is shown in figure B.5. This implementation ignores pipeline hazards, so instructions can be ideally executed with CPI equals to 1.

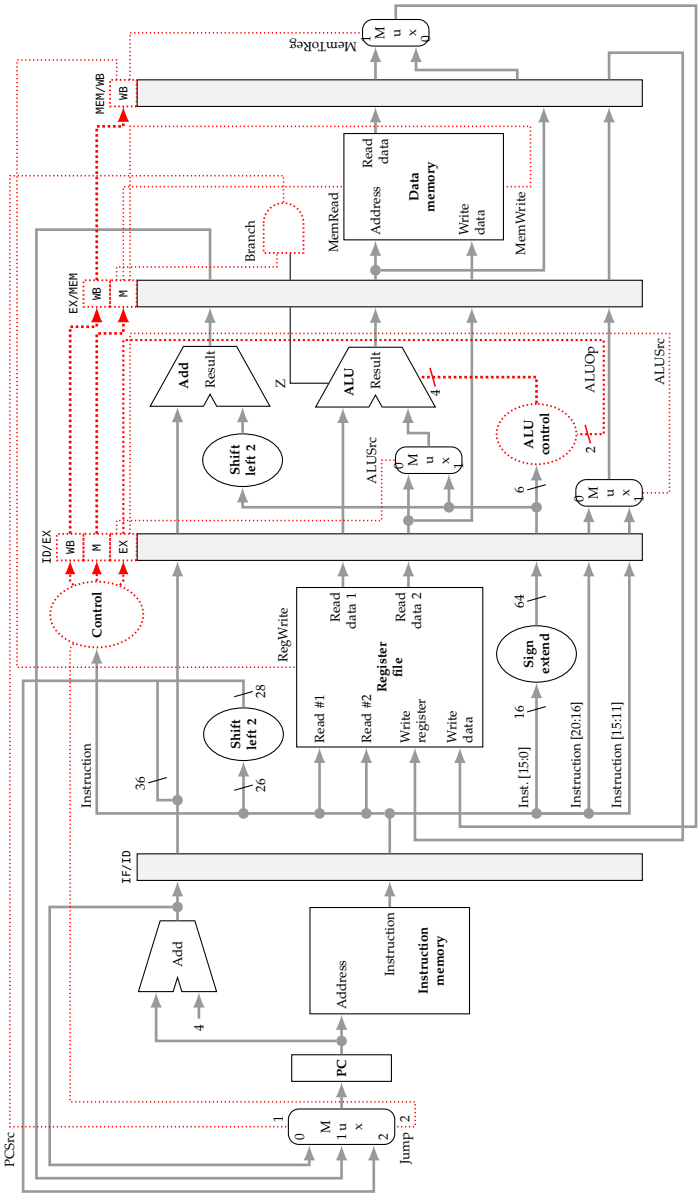


Figure B.5: Pipelined datapath and control unit ignoring pipeline hazards

Bibliography

[1] D.A. Patterson, J.L. Hennessy. *Computer organization and design. The hardware/-software interface, 5th edition*. Morgan Kaufmann, 2014. ISBN: 978-0124077263.

[2] J.L. Hennessy, D.A. Patterson. *Computer architecture. A quantitative approach, 5th edition*. Morgan Kaufmann, 2012. ISBN: 978-0123838728.

[3] G.E. Moore. *Moore's law at 40*. Understanding Moore's law: four decades of innovation. Chemical Heritage Foundation, 2006. ISBN: 978-0941901413.

[4] Julio Ortega, Mancia Anguita, Alberto Prieto. *Arquitectura de computadores*. Paraninfo, 2005. ISBN: 849-7322746.

[5] W. Stallings, A.C. Vargas. *Organización y arquitectura de computadores: diseño para optimizar prestaciones*. Prentice Hall, 2001. ISBN: 978-8420529936.

[6] R.R. Schaller. *Moore's law: past, present and future*. IEEE Spectrum, vol. 34(6), pp. 52–59, 1997.

[7] V.C. Hamacher, Z.G. Vranesic, S.G. Zaky, M.L.F. García, G.Q. Vieyra. *Organización de computadoras*. McGraw-Hill, 1987. ISBN: 968-4220588.

Computer Arquitecture

Nowadays, computers are ubiquitous. They can be found in uncountable situations and applications, from the traditional personal computers to smartphones and embedded devices for industrial automation. The latter are the basis of a new digital revolution that is allowing, among other advances, smart cities, self-driving cars and the Internet of Things (IoT).

The aim of this book is presenting the techniques involved in developing current computers from three points of view: performance improvement, operating system support and virtualization support. For this, the contents are organized in four chapters: introduction to computer architecture, CPU, memory hierarchy and input/output system.

This book is oriented to serving as bibliography in university courses about Computer Architecture, within the studies of Computer and Software Engineering, where a basic understanding of the internals of computers is assumed.

ISBN 978-84-17445-50-8
9 788417 445508

ediuno

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo