Essentials of computing systems

João M. Fernandes

Coleção Educação | Ciências, Engenharia e Tecnologia

Educação
Ciências, Engenharia
e Tecnologia

AUTOR
João M. Fernandes

COORDENAÇÃO EDITORIAL
Manuela Martins

FOTO CAPA
Iuzvykova Iaroslava/Shutterstock

PAGINAÇÃO EM LATEX
João M. Fernandes

EDIÇÃO UMinho Editora

LOCAL DE EDIÇÃO Braga 2022

ISBN 978-989-8974-60-0

DOI https://doi.org/10.21814/uminho.ed.33

Os conteúdos apresentados (textos e imagens) são da exclusiva responsabilidade dos respetivos autores.
© Autores / Universidade do Minho – Esta obra encontra-se sob a Licença Internacional Creative Commons Atribuição Compartilha Igual 4.0.

UNIVERSIDADE DO MINHO

Essentials of computing systems

João M. Fernandes

Coleção Educação | Ciências, Engenharia e Tecnologia

Preface

About the book

Computers were originally invented to solve all sort of mathematical problems. Nowadays, computers do much more than that and are present in all human activities. In fact, a computer is a fantastic machine capable of doing the most amazing tasks, if an appropriate program is provided. A computer system contains hardware and systems software that work together to run software applications. Interestingly, the underlying concepts that support the construction of a computer are relatively stable. In fact, (almost) all computer systems have a similar organisation, i.e., their hardware and software components are arranged in hierarchical layers (or levels) and perform similar functions. This book was written for programmers and software engineers who want to comprehend how the components of a computer work and how they affect the correctness and performance of their programs.

In fact, understanding how a computer works is a fundamental knowledge for software engineers to comprehend the principles governing the execution of the programs they develop or maintain. It is also a fundamental knowledge if one wants to optimise the performance of a program, to write a compiler, or to develop an embedded system. If a programmer knows exactly how the computer hardware operates, she can write efficient programs. For example, the method used to map main memory to cache can have a huge impact on the order by which array elements are accessed. Therefore, computer scientists and software engineers must understand how computer hardware interacts with software. This is the objective of this book, which is focused on describing the fundamental aspects of computers, by studying their organisation and structure.[1]

This book was originally written to support the courses on Computing Systems at Departamento de Informática, Universidade do Minho. The initial structure of these courses was defined by Alberto J. Proença, with the idea of allowing students to understand the main principles that govern the operation of a computer. This book still follows that idea, but was tuned to focus on the essential elements of computers that software engineers must master. Anyway, we hope that instructors from other universities will adopt this book for their courses.

[1] The distinction among these two terms is fuzzy and is out of the scope of this book.

This book addresses a very fundamental aspect related to the Computing discipline, so it is expected that any university-level student is able to understand its contents without major problems. Anyway, the reader is supposed to have a relatively good knowledge about how to program a computer in a high-level programming language, like C.

The book is structured in six chapters. Chapter 1 introduces the different levels which one can see a computer, representing each level a distinct abstraction of the computer. This chapter also describes the von Neumann architecture, which is followed by all modern computers. Chapters 2 and 3 address cover the different forms of discrete data found in computers. Chapter 2 introduces the main concepts and techniques related to the representation of textual information, instructions, images, and audio. Chapter 3 explains how numbers are represented in computers. Chapter 4 discusses the details of the IA32 assembly language and it discusses how high-level programs (written in C) get compiled into this form of machine code. Chapter 5 presents the main principles of caches, namely the different mapping functions and replacement algorithms. Chapter 6 explores how to make programs run faster via several different types of program optimisation. Each chapter has a set of exercises to ensure that students put in practice the concepts presented. The book provides answers to almost all those exercises to allow students to check their progress. A glossary, which provides brief definitions of all key terms from the chapters, is also included.

About the author

João Miguel Fernandes is full professor on software engineering at the Dep. Informatics, School of Engineering, UMinho.

João holds a 5-year degree in Informatics Engineering (1991), and a master degree in Informatics (1994), both from UMinho (Braga, Portugal). In May 2000, he has completed his Ph.D. thesis in Informatics/Computer Engineering, from UMinho, with a thesis entitled "An object-oriented methodology for embedded systems development."

He has been an invited researcher/professor at Universidade do Algarve (Portugal), University of Bristol (United Kingdom), Aarhus University (Denmark), TUCS (Turku Centre for Computer Science, Turku, Finland), Universitatea Tehnică Gheorghe Asachi (Iaşi, Romania), UFSC (Universidade Federal de Santa Catarina, Florianópolis, Brazil), ISCTEM (Mozambique), and ISTM (Angola).

João is the main author of the book "Requirements in engineering projects" (Springer, 2016) and co-editor of the book "Behavioral modeling for embedded systems and technologies: Applications for design and implementation" (IGI Global, 2009). He is the author of more than 130 scientific publications with peer revision on international conferences, journals and chapters of books. His publications have collected more than 2.000 citations. He is member of the Editorial Review Board of the Journal of Information Technology Research (IGI Publishing) and editor of Open Computer Science (De Gruyter/Springer). Additionally, he has already served as a scientific reviewer for an Addison-Wesley book, for scientific journals and for many symposia. He also regularly serves as a member of the Programme Committees of international conferences and workshops.

He has been involved in the organization of various international scientific events, including the 3rd Int. Conf. on Application of Concurrency to System Design (ACSD 2003), the 5th IFIP Int. Conf. on Distributed and Parallel Embedded Systems (DIPES 2006), the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009), the 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (PETRI NETS 2010), the 10th International Conference on Application of Concurrency to System Design (ACSD 2010), the 6th International Conference on Software Business (ICSOB 2015), and the Model-based Methodologies for Pervasive and Embedded Software (MOMPES) workshop series.

João conducts his research activities in Software Engineering, with a special interest in the following topics: software modeling, requirements engineering, embedded software, software process, bibliometrics. Within his research and teaching activities, he maintains regular collaborations with the industry. iFlow, a research project developed in cooperation with Bosch Portugal, received the 2016 Annual Logistics Excellence Award (PEL 2016) from the Portuguese Logistics Association (APLOG).

Acknowledgments

A number of colleagues have read the intermediate versions of the book and provided useful suggestions. In particular, I would like to thank Alberto Proença, Luís P. Santos, and Manuel Alves. I would also like to acknowledge Luís Gomes and João M.P. Cardoso, who acted as reviewers and gave useful feedback and many good suggestions, to ensure a quality textbook. The publishing team at UMinho Editora has been wonderful to work with and a special thanks goes to Carla Marques for the constant help and support. Alberto Simões and Bruno Dias helped me in adjusting the LaTeX styles to comply with the formats of the publisher. Lastly, and most importantly, I am deeply indebted to my wife Raquel and our two children Gonçalo and Constança. This book would not have been written without their patience and encouragement.

Table of Contents

Preface ... i

1 Computer systems ... 1
1.1 Levels of a computer ... 1
1.2 Organization of a computer ... 4
1.2.1 Central processing unit ... 5
1.2.2 Main memory ... 7
1.2.3 Secondary memory ... 8
1.2.4 I/O devices ... 9
1.3 Instruction-level parallelism ... 10
Exercises ... 13

2 Representation of information ... 15
2.1 Digital abstraction ... 15
2.2 Bits, bytes and words ... 16
2.3 Textual information ... 18
2.4 Machine-level instructions ... 20
2.5 Images ... 21
2.6 Audio ... 22
Exercises ... 23

3 Representation of numbers ... 25
3.1 Positional numeral systems ... 25
3.2 Octal and hexadecimal numbers ... 27
3.3 Conversions between different bases ... 29
3.4 Negative numbers ... 30
3.4.1 Sign-magnitude ... 31
3.4.2 One's-complement ... 31
3.4.3 Two's-complement ... 32
3.4.4 Excess representations ... 33
3.5 Two's-complement addition ... 34
3.6 Floating-point numbers ... 36

3.7 Binary codes for decimal numbers 41
Exercises 42

4 IA32 instruction-set architecture 47
4.1 Compilation of C code to assembly code 47
4.2 IA32 assembly language 49
4.2.1 Registers 50
4.2.2 Data types 51
4.2.3 Operands 51
4.2.4 Data movement instructions 52
4.2.5 Arithmetic and logical instructions 55
4.2.6 Control instructions 59
4.2.7 Procedures 63
4.2.8 Data structures 68
Exercises 71

5 Cache memory 75
5.1 Main principles 75
5.2 Mapping function 78
5.3 Replacement algorithm 82
Exercises 82

6 Code optimisations 85
6.1 Main principles 85
6.2 Limitations of the compilers 86
6.3 Machine-independent optimisations 88
6.3.1 Code motion 89
6.3.2 Elimination of unnecessary accesses to memory 90
6.3.3 Loop unrolling 91
6.3.4 Reduction of the number of procedure calls 93
6.4 Machine-dependent optimisations 93
6.5 Cache-oriented optimisations 95
Exercises 95

Solutions to exercises 99

Glossary 101

References 111

Index 113

Chapter 1
Computer systems

Abstract This chapter introduces the different levels at which one can see a computer. Each level represents a distinct abstraction of the computer. This division in layers permits those that work at a given level to be unaware of what happens in the other levels. This perspective allows unnecessary details to be omitted, which reduces complexity. This chapter also describe the von Neumann architecture, which is followed by all modern computers. It includes a CPU, the main memory, and some input/output devices.

1.1 Levels of a computer

A **computer** is a programmable device that can automatically execute a sequence of instructions on data once programmed for the task. It can store, retrieve, and process data according to the instructions. The sequence of instructions that describe the task to be performed is the **program**.

It is relevant here to distinguish a computer from a calculator. A device is considered as a computer if it has both of the following characteristics, and is a calculator if it lacks either or both (Fenwick, 2015, pp. 12–13):

- **Data-dependent instruction sequence**. A computer must include conditional branch constructors (like `IF-THEN-ELSE`, `SWITCH`, and `DO-WHILE`), whose execution depends on values that can only be calculated during the program execution (i.e., values that are not known while programming).
- **Data-dependent data selection**. A computer must be able to use data to determine the actual data to be used in calculations. This mechanism is used, for example, when accessing arrays and other types of structured variables.

The electronic circuits of each computer are able to directly execute a limited set of simple instructions, into which all its programs must be converted before they can be executed. These simple instructions are operations like:

- add two numbers;
- test if a number is equal to zero;

- copy a piece of data from one part to another;
- decide what instruction to execute next based on the evaluation of some logical condition.

These primitive instructions constitute a language, called a **machine language**, that programmers can use to operate the computer. The engineers that design a computer must decide what instructions to include in its machine language. Programming with machine languages is usually difficult, error-prone, and tedious for people, because they tend to be simple and cryptic. The machine language exists mainly to be executed by the computer and not necessarily to be understood by humans. Even a simple program may require many instructions, which make the program long and consequently hard to master. Fortunately, new languages can be constructed on top of the machine language to rise the abstraction level. This hierarchical approach is recursive, so computer systems can be designed in an organised, structured, and layered way. In general, a computer can be seen as a set of layers or levels, as shown in Fig. 1.1.

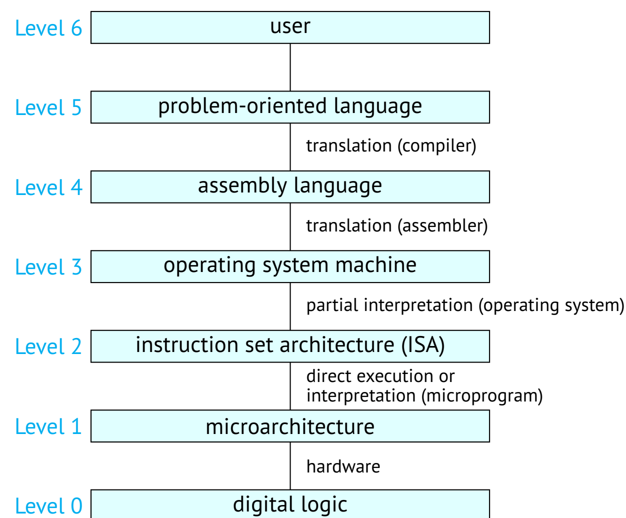


Fig. 1.1 A seven-level computer. The support method for each level and the supporting program is indicated below it.

Level 6, the user level, is what everyone who interacts with computers is familiar with. This level contains software applications, like text editors, web browsers, graphics programs, and games. The user at this level sees software applications that he can select to run for specific purposes. He can open a word editor to write a letter, a web browser to access a given website, or a game to have fun. Fortunately, the user can manipulate all these applications at this level, without having any idea on how the lower levels work.

Level 5, the problem-oriented language level, consists of languages, like C, C++, Java, Perl, Python, and PHP, which are appropriate to develop software applications. Such languages are called high-level languages. Programs written in these languages are generally

translated to levels 4 or 3 by translators, known as compilers, or interpreted. For example, Java programs are usually first translated to an ISA-like language, called Java byte code, which is later interpreted. In some cases, level 5 consists of an interpreter for a specific application domain. This level provides data and operations for solving problems in this domain, in terms that people familiar with the domain easily understand.

Level 4, the assembly language level, is really a symbolic form for one of the underlying languages. This level provides a method for humans to write programs for levels 3 and 2 in a form that is not as unpleasant as the virtual machine languages themselves. Programs in assembly language are first translated to level 3 or 2 language and then interpreted by the appropriate virtual or actual machine. The program that performs the translation is called an **assembler**.

Level 3 is usually a hybrid level. Most of the instructions in its language are also in level 2. This is totally admissible as an instruction at one level can also be present at other levels as well. Additionally, there is a set of new instructions, a different memory organisation, the ability to run two or more programs concurrently, and other features. The new facilities added at level 3 are carried out by an interpreter running at level 2, which is usually called as an **operating system**. The level 3 instructions that are identical to those on level 2 are executed directly by the microprogram (or hardwired control), not by the operating system. In other words, some of the level 3 instructions are interpreted by the operating system and some are interpreted directly by the microprogram. This justifies why this level is classified as hybrid.

Level 2 is the ISA (instruction set architecture) level. Every computer is accompanied by an instruction set manual that describes a strictly numerical language, the so-called machine language. A program at this level is machine code, which means that its instructions can be carried out interpretively by the microprogram or the hardware execution circuits. It is the lowest level representation of a program.

Level 1 is the microarchitecture level, where one sees a relatively small collection of registers, which forms a local memory, and an ALU (arithmetic logic unit), which can execute simple arithmetic operations. The registers and the ALU are interconnected to form a data path, through which the data flow. The basic operation of the data path consists of selecting one or two registers, having the ALU operate on them (e.g., adding them together), and storing the result back in some register.

The operation of the data path is usually controlled by a program called a **microprogram**. On machines with software control of the data path, the microprogram is an interpreter for the level 2 instructions. It fetches, examines, and executes instructions one after the other, using the data path to do so. For example, a `SUB` instruction is fetched from memory, its operands located and transferred to the registers, the subtraction computed by the ALU, and finally the result transferred to the destination. On a machine with hardwired control of the data path, similar steps are taken, but without an explicit stored program to control the interpretation of the level 2 instructions.

Finally, level 0, the digital logic level, represents the computer hardware. Its circuits execute the machine-language programs of level 1. Level 0 is composed of logic gates and wires. A gate is built from analog components, like transistors. A gate can be viewed as a digital device with a set of inputs that outputs some simple function, like logical AND or OR, of these inputs. This level also includes flip-flops, which are 1-bit memories that can

be used to store a bit (either a '0' or a '1'). A register can be assembled with, for example, 32 1-bit memories, which can be used to hold a 32-bit number.

A fundamental frontier exists between levels 4 and 3. The lowest three levels are intended primarily for running the interpreters and translators needed to support the highest levels. These interpreters/translators are developed by systems programmers, who are specialised in constructing new virtual machines. Levels 4 and above are intended for the applications programmer with a given problem to solve in a specific domain. Another change that occurs at level 4 is the method by which the higher levels are supported. Levels 2 and 3 are always interpreted. Levels 4, 5, and above are usually supported by translation/compilation. Another difference between levels 1 to 3 and levels 4, 5, and higher is the nature of the language provided. The machine languages of levels 1, 2, and 3 are numeric. Programs in these levels consist of long series of numbers (bits), which are appropriate for machines, but very hard for humans. At level 4 and above, the languages contain words and mnemonics that humans are able to easily understand.

To summarise, computers are designed as a series of levels, each one built on its predecessors. Each level represents a distinct abstraction of the computer, with different concepts, objects and operations. This division in layers is very convenient as it permits those that work with the computer at a given level to be unaware, if they wish, of what happens in the other levels. For example, a programmer that works with a level-5 language may not be aware of the principles and mechanisms that permit his programs to be translated to a level-4 language. By analysing or operating computers with this perspective, unnecessary details can be omitted, which reduces complexity.

From a programmer's perspective, it is relevant to understand how computers work at the lower levels. For most cases, knowing what happens in the underlying levels is not mandatory. However, when for example a program written in a high-level language does not execute as expected, is not running as fast as intended, or consumes too much energy, it is often necessary to analyse what is happening at the lower levels. This analysis is only possible if the programmers comprehend how computers operate at those levels.

1.2 Organization of a computer

A computer is a complex system that contains millions of elementary electronic components. To abstractly describe a computer, it is essential to follow an hierarchical approach. Almost all uniprocessor (or scalar) computers follow the so-called **von Neumann architecture**, which includes a CPU, the main memory, and some input/output devices, as shown in Fig. 1.2. The existence of two separate components, one for processing the data and a different one to store them, is a distinguished characteristics of this architecture. All transfers of information between these two units go through the unique channel connecting them, which constitutes a serious bottleneck. This architecture also introduces the stored program concept that makes computers to be general-purpose machines. This section introduces the major components of the computer and the way they are interconnected.

The von Neumann architecture has been extended in several ways to address some of its limitations. A well-known proposal, as illustrated in Fig. 1.3, is the inclusion of a **system bus**

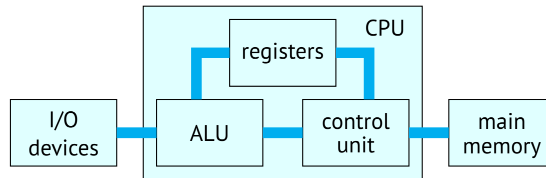


Fig. 1.2 The von Neumann architecture.

that connects the components of the computer. The system bus has three main elements: data bus, address bus, and control bus. The data bus moves data from main memory to the CPU registers (and vice versa). The address bus holds the address of the data that the data bus is currently accessing. The control bus carries the necessary control signals that specify how the information transfer is to take place.

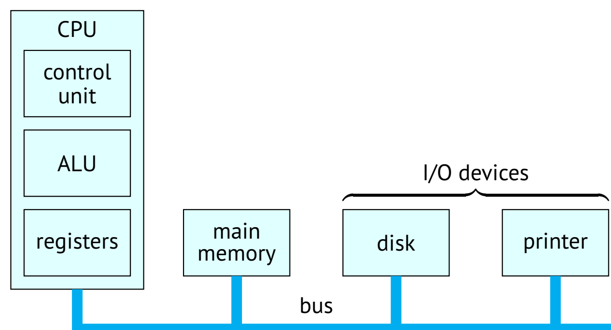


Fig. 1.3 The system bus model.

1.2.1 Central processing unit

The **central processing unit (CPU)**, also called processor, is the "brain" of the computer. On the machine language level, the operation of the CPU is quite straightforward. It is responsible for the execution of the program, stored as a sequence of machine language instructions in the main memory. Each instruction directs the CPU to perform some basic task, like subtracting two numbers or moving data from one register to another one. The CPU does all this mechanically, without understanding what is the purpose for executing the instruction. This means that the program must be complete and have no errors, since the CPU can only execute it.

The CPU is composed of three major components: control unit, ALU, and registers. The components are connected by a **bus**, which is a collection of electric wires for conducting address, data, and control signals. Buses allow the parallel movement of bits. At a given

moment in time, only one device may use the bus. Buses can be external to the CPU, connecting it to memory and input/output (I/O) devices, but also internal to connect its parts.

The CPU contains a small number of high-speed memory registers. A **register** is used to store temporary results (like a number) and status information. All the registers have usually the same size, that is, an equal number of bits. Registers can be read and written at a very high speed since they are internal to the CPU. A relevant register is the **instruction pointer (IP)**, also known as program counter (PC), which indicates the address of the next instruction to be executed. Another important register is the **instruction register (IR)**, which stores the instruction, codified according to the machine convention, currently being executed. Most computers have other registers as well, some of them general-purpose and others for specific purposes.

The **arithmetic logic unit (ALU)** performs basic operations such as addition, subtractions, and comparisons, which are necessary to execute the instructions. Usually, an ALU has two data inputs and one data output. The operations carried on by the ALU affect the status registers, which indicate attributes related to the last arithmetic or logic operation (e.g., if the result was negative or not). The operation that the ALU must perform is signalised by the control unit, based on the instruction that is executed in each instant.

The **control unit** is responsible for orchestrating the components of the computer to ensure that the instruction that is being executed in each moment produces its exact and expected effects. This is accomplished by activating in the right sequence the control signals that are sent to the computer components. From a conceptual point of view, the execution of an instruction by the CPU can be divided into the following steps:

1. Fetch the instruction from memory and store it into the IR.
2. Determine the type of instruction.
3. If a word in memory is used by the instruction, determine its location.
4. If needed, transfer the word from the memory into a CPU register.
5. Produce the effects of the instruction, i.e., perform some simple operation dictated by the instruction.
6. Change the IP to refer to the next instruction.

This process, called the **fetch-decode-execute cycle**, is repeated indefinitely, from the time that power is applied to the computer, until it is powered off. The CPU receives an instruction from the memory, decodes it, and executes it using data obtained from the memory or already available in the registers. Once the processor finishes the execution of an instruction, it starts the cycle again for the next instruction in the memory, except when a branch instruction was just executed. This cycle is a central concept to the operation of digital computers, since essentially they all just execute instructions, one after the other. The control unit can be seen as a program (a microprogram that can be implemented in hardware) that goes through the set of steps to execute the instructions of another program (i.e., the one that is being executed by the processor).

Most computer systems have a **clock** signal that acts like a heartbeat. The clock signals the passage of time within the computer. A clock emits periodically a pulse, i.e., it transmits a precise pulse with regular intervals of high and low values. The time between two consecutive ticks is called a clock cycle (or clock period), which represents one discrete time unit. The different operations performed by a processor, such as fetching an instruction or decoding it, are synchronised by the clock. All these operations begin with the pulse of the

clock. Therefore, the speed at which the instructions are executed by the processor depends on the frequency of the clock, measured in cycles per second or Hertz (Hz).

1.2.2 Main memory

The **main memory** is the component of the computers, where programs and data are both stored. A memory consists of a number of cells or locations, each of which can store a piece of information. Each cell has a number, called its **address**, by which programs can refer to it. A memory with n cells uses addresses from 0 to $n-1$. All memory cells have the same number of bits.

The main memory is a **random access memory (RAM)**, which stores a piece of data at a unique address and can recall the data upon presentation of the complete unique address. This is an important aspect, since microprocessors access the memory to fetch instructions and to read or write data in consecutive clock cycles. The addresses of these instructions and data need not be somehow related in a discernible way. In fact, RAM is able to handle random, or more precisely arbitrary, access, that is, any datum can be retrieved as quickly as any other, without favouring any particular location. RAM memories are volatile, which means that the information disappears if the supply voltage is turned off. In contrast, the contents of nonvolatile memory is preserved even after power is removed. For example, the flash memory used in digital cameras is nonvolatile, since the data is not lost when the power is turned off. Physically, the main memory consists of a collection of **dynamic random access memory (DRAM)** chips.

The **cell** is the smallest addressable unit. Almost all modern computers use 8-bit (1 byte) cells. Bytes are grouped into words. A computer with a 32-bit word has 4 bytes/word, while a computer with a 64-bit word has 8 bytes/word. The word is also important since most instructions operate on entire words, for example, adding two words together. Thus, a 32-bit machine has 32-bit registers and instructions for manipulating 32-bit words.

An issue that needs to be handled, when storing a multibyte value in a byte-addressable memory, is the order of the individual bytes. For example, the number $11\ 0000\ 0000_2$ (represented in base two as explained in Chapter 3) requires two bytes of memory, one storing the low set of eight bits $0000\ 0000_2$ and another one storing the high set of eight bits $0000\ 0011_2$ (leading zeros are added). If this value is to be stored in the memory at location 2104, there are two popular ordering alternatives as shown in Fig. 1.4.

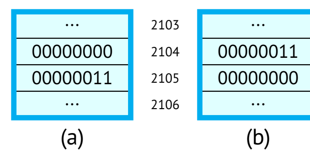


Fig. 1.4 Byte ordering for multibyte: (a) Little-endian, (b) Big-endian.

If the low order byte of a multibyte value is put first in memory (i.e, has the lowest addresses), **little endian** is the adopted alternative. Otherwise, if the high order bytes comes first, **big endian** is the selected order. In both cases, the two-byte number has address 2104, but occupies also the memory cell with address 2105.

Digital computers possess a great versatility. They are machines with finite hardware that can execute whatever programs their users wish.[1] This outstanding flexibility is related to the ability of a computer to load the programs in its internal memory. So, instructions and data are both stored in the memory. By changing the program in the memory, one changes the problem the computer is solving. This justifies the notion of a **stored program computer** (or general-purpose computer), one that can be used for any problem.[2] Additionally, during a computation, a given instruction may also be modified, i.e., changed to a different instruction, a characteristic that is not usually explored. This gives rise to **self-modifying code**, a concept that is particularly straightforward to perform at the machine-level, by directly writing new instructions over the existing ones in the memory. The stored program concept is a fundamental element of many computer models, namely the universal Turing machine and the von Neumann machine. The **Turing machine**, a mathematical model of a simple computer, can be used mainly to analyse the logical foundations of computer systems.

1.2.3 Secondary memory

The main memory of a computer is never enough, because people always need to store more information than it can hold. The traditional solution to store a great quantity of data is to use a memory hierarchy, as illustrated in Fig. 1.5. The top level includes the CPU registers, which constitute the fastest memories of the computer. Next come one or more cache memories, which according to Table 1.5, are on the order of 32 KiB to a few megabytes. Main memory is the next level, with sizes ranging from 1 GiB for low-end systems to hundreds of gibibytes for high-end ones. Afterwards, come local secondary storage, which include local disks, magnetic tape and optical disks for archival storage. The final level is remote secondary storage, which includes disks on remote servers that can be accessed over a network.

The overall goal of this hierarchy is to establish a memory system with cost almost as low as the cheapest level and speed almost as fast as the fastest level. All the data in a given level can be found in the next lower level. Each level in this hierarchy is smaller, faster and more expensive per bit than the lower ones. In fact, three parameters increase as the hierarchy is visited from the top to the bottom, as shown in Fig. 1.5 and in Table 1.1.

Firstly, the access time gets bigger. CPU registers can be accessed in a nanosecond or less. Cache memories use static random access memory (SRAM) technology and take a small multiple of CPU registers. Main memory accesses are typically 10 nanoseconds. The biggest

[1] Early computers had fixed programs. To modify the program, one was supposed to re-wire and re-structure the machine, which was very tedious and error-prone.

[2] There are problems that computers cannot solve. An interesting discussion about the limits of computers is presented by Harel (2000).

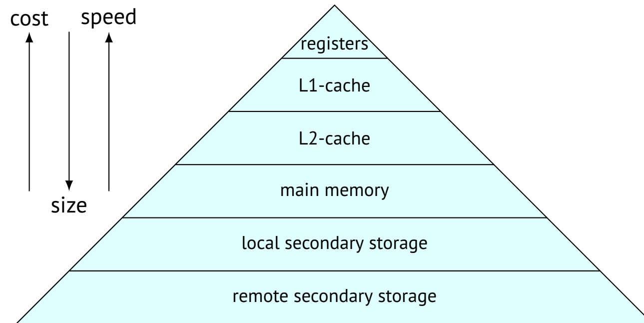


Fig. 1.5 A six-level memory hierarchy; Adapted from Bryant and O'Hallaron (2016).

level	1	2	3	4
name	registers	cache	main memory	disk storage
typical size	< 1KB	< 16 MB	< 512 GB	> 1 TB
implementation	CMOS	SRAM	DRAM	magnetic disk
access time (ns)	0.25–0.5	0.5–25	50–250	5,000,000
bandwidth (MB/sec)	50,000–500,000	5000–20,000	2500–10,000	50–500

Table 1.1 Typical levels in the memory hierarchy of a desktop computer; Adapted from Hennessy and Patterson (2007, p. C-3).

gap occurs in the interface between the main memory and the disk, as disk access times are at least 10 times slower for solid-state disks (and much slower for magnetic disks). Tape and optical disk access can be measured in seconds if the media have to be fetched and inserted into a drive.

Secondly, the storage capacity also increases. CPU registers are good for a small number of bytes. IA32, whose instruction set is described in detail in Chapter 4, has eight 4-byte generic registers, i.e., 32 bytes. Caches, which are presented and discussed in detail in Chapter 5, can store tens of megabytes, while main memories a few gigabytes. Solid-state disks have space for hundreds of gigabytes, and magnetic disks for terabytes. Tapes and optical disks are usually kept off-line, so their capacity depends mainly on the owner's budget.

Thirdly, the number of bits that one gets per dollar (or euro) increases down the hierarchy.

Programmers need to understand the memory hierarchy, because it has a big impact on the performance of the software programs. The more distant the data is to the CPU, the longer it takes to access it.

1.2.4 I/O devices

Computers are built so that they can be expanded with new I/O (input/output) devices, such as a hard disk, a keyboard, a mouse, a monitor, a printer, an audio output device, a network interface, a scanner. An **I/O device** is a hardware system used by humans or systems

to communicate with a computer. The CPU must communicate with and control all these devices. So, for each device in a system, there is a **device driver**, which consists of software that the CPU executes when it has to deal with that device. A new device can be installed by connecting the device into the computer and installing its device driver software. The device driver allows the CPU to communicate with the respective device.

As Fig. 1.2 shows, the I/O devices are connected to the computer through busses, which carry data, addresses, and control signals. An address directs the data to a specific device and probably to a particular element within that device. Control signals are used, for example, by a device to alert the CPU that data is available on the data bus.

When a given device produces data that needs to be processed, somehow the CPU needs to take some action. A simple solution, which is not very satisfactory, is for the CPU to keep checking for incoming data over and over. Whenever it detects that data was made available by the device, it processes it. This method is designated as **polling**, since the CPU repeatedly checks the input device to see whether it has any data. Polling is very simple, but is also very inefficient.

To avoid this inefficiency, interrupts are generally used instead of polling. An **interrupt** is a signal sent by a device to the CPU to request its attention. The CPU reacts by suspending what it is doing to answer to the interrupt. For this purpose, the CPU saves information about what it is currently doing so that it can return to it afterwards. This information consists of the contents of important internal registers such as the instruction pointer. Then the CPU jumps to some fixed memory location and begins executing the instructions stored there. Those instructions correspond to the interrupt handler that does the processing necessary to respond to the interrupt. This interrupt handler is a part of the device driver for the device that issued the interrupt. Once the interrupt is processed, the CPU returns to the process that was suspended. To this end, it restores its previously saved state. For example, whenever a key is pressed on the keyboard, an interrupt signal is sent to the CPU. The CPU responds by suspending what it is doing, reading the key that was pressed, processing it, and then returning to the task it was performing just before the key was pressed.

1.3 Instruction-level parallelism

Improving the characteristics of computers, like performance, cost, reliability, energy consumption, is a concern for computer engineers and architects, as they aim to construct the most efficient machines for the users. Performance is an inescapable issue, since making computers run faster is always relevant. One possible approach is to exploit parallelism, which can have different forms: instruction-level, data-level, and processor-level parallelism. This section discusses instruction-level parallelism with pipelining, which explores individual instructions to obtain higher throughput (i.e., more instructions per time unit).

With **pipelining**, the process of executing instructions is divided in stages, allowing multiple instructions to be overlapped in execution. Each stage is responsible for a part of the process and all can run in parallel, thus speeding up the process. Fig. 1.6 shows a possible five-stage pipeline. Stage 1 (FI) is responsible for fetching the instruction from the memory and to place it in the IR. Stage 2 (DI) decodes the instruction, to identify its type and what

operands are needed. Stage 3 (FO) fetches the operands, either from registers or from the memory. Stage 4 (EI) carries out the instruction. Stage 5 (SR) writes the result back to the proper destination.

Fig. 1.6 A 5-stage pipeline.

Fig. 1.7 illustrates the ideal operation of the pipeline. During the first clock cycle, stage 1 handles instruction 1, fetching it from memory. During cycle 2, stage 2 decodes instruction 1 and stage 1 fetches instruction 2. During cycle 3, stage 3 fetches the operands for instruction 1, stage 2 decodes instruction 2, and stage 1 fetches the third instruction. And so on. Thus, during the fifth cycle, stage 5 stores the result of instruction 1, while the other stages are operating on the subsequent instructions.

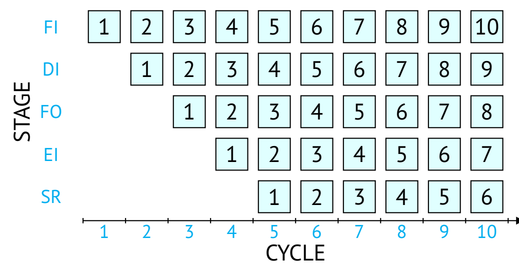


Fig. 1.7 An ideal five-stage pipeline in action.

The different stages operate in parallel, which permits this process to be accelerated. Pipelining increases the number of instructions that are simultaneously under execution. In fact, instruction **throughput**, which measures the number of instructions completed per time unit, is improved. This improvement occurs without the need to reduce the time that is required to complete each individual instruction, called the **latency**.

With a pipeline, all instructions must pass through each stage. If an instruction has no operand, it could skip stage 3, but to simplify the hardware and the timing, each instruction proceeds through all stages, even if not necessary. As Fig. 1.7 shows, in 10 cycles, six instructions are fully executed. In fact, with this 5-stage solution, after the pipeline is full, one instruction can be potentially executed at each clock cycle, i.e., the throughput is one clock cycle. This rate can be observed from cycles 5 to 10, in which six instructions are completed. Theoretically, if all stages take the same time, then there is a speed up of five, as one individual instruction has a latency equal to five cycles. In fact, the maximum speedup is equal to the number of stages.

The maximum theoretical speed up is however very unlikely to occur in practice, because it is difficult to balance the time it takes to complete each stage. All stages must be equally balanced, otherwise the faster stages must wait for the slower ones. Additionally, the following three major aspects also negatively affect the theoretical speed up:

1. resource conflicts,
2. data dependencies,
3. conditional branch statements.

A **resource conflict** greatly affects instruction-level parallelism. This type of conflict occurs whenever two pipeline stages need to simultaneously access the same resource (e.g., memory). For example, a conflict exists when a given stage is storing a value to the memory, while another one is fetching the instruction from memory. Fig. 1.8(a) shows this situation, because both the instruction fetch (FI) for instruction 3 and the operand fetch (FO) for instruction 1 need a simultaneous access to the memory. This conflict is usually resolved by permitting the operand fetch to occur, while waiting for the instruction fetch, as illustrated in Fig. 1.8(b). Certain conflicts can also be tackled by providing two separate pathways: one for data coming from memory and another for instructions coming from memory.

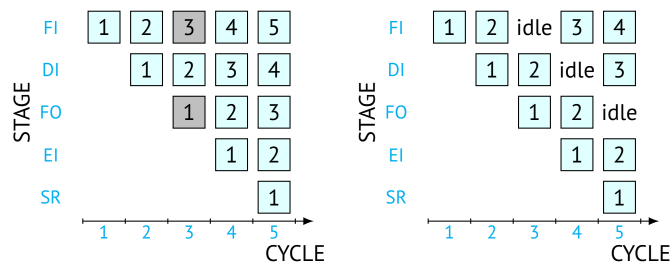


Fig. 1.8 A resource conflict in a 5-stage pipeline.

A **data dependency** occurs whenever the result of one instruction, not yet available, is also the operand of a subsequent instruction. There are some solutions to handle these conflicts. Special hardware can be incorporated to detect instructions whose source operands are destinations for instructions further up the pipeline. By inserting a delay (e.g., a `nop` instruction that does nothing) into the pipeline, sufficient time passes and the conflict disappears. Some architectures assume that this problem is solved by the compiler, which must reorder instructions, so that there is a delay in loading any conflicting data, but no effect on the program overall behaviour.

Problems in the pipeline can also be caused by a **branch instruction** that modifies the normal execution flow of a program. If the conditional branch is taken, the subsequent instructions must not be executed and the pipeline must be emptied. Some architectures predict the outcome of a conditional branch to try to identify the instructions that will be executed next. In some cases, compilers try to solve branching issues by rearranging the machine code to cause a delayed branch. An attempt is made to reorder and insert useful instructions, but, if not possible, `nop` instructions are inserted to keep the pipeline full.

Another alternative is to initiate fetches on both paths of the branch. When the branch is actually executed, the correct path is identified and the pipeline can be safely continued.

Exercises

Exerc. 1.1: Describe the following terms with your own words: **(a)** Compiler; **(b)** Interpreter; **(c)** Virtual machine.
Exerc. 1.2: Write a small program in a given programming language. Compile it and try to calculate the ratio of source code statements to the machine language instructions generated by the compilation process. Add different types of statements to the high-level program, one at a time, and check how the machine language program is affected.
Exerc. 1.3: On a big-endian computer, a 32-bit integer with value 00010010 00110100 01010110 01111000 is about to be stored in the memory at location 132,104. Indicate which memory cells are affected and which values are stored in each one.
Exerc. 1.4: Consider that part of the memory of a little-endian computer contains the values shown in the figure. Indicate the value of a 32-bit integer if it is read from the memory location 4365.

4362	0100 0011
4363	0111 0000
4364	0000 0011
4365	0001 0010
4366	1111 1111
4367	0000 0000
4368	0000 1111

Exerc. 1.5: In a stored-program computer, both the instructions and the data of a program are located in the main memory while it is executed. What are the possible implications if a program accidentally modifies the value that is stored in a memory cell that is related to an instruction?
Exerc. 1.6: In a factory, the production process of a given product goes through four steps: preparation, assembly, testing, and packaging. Those steps take the following times, in seconds, to be executed: preparation (20), assembly (30), testing (35), and packaging (35). Calculate the time needed to produce 1000 replica of the product by: **(a)** one single person; **(b)** four persons working in a pipeline.

Further reading

The history of computers is a very valuable source for better understanding many issues related to computers. Ceruzzi (1998) provides an excellent overview of the history of com-puting in the period 1945-2001. Another interesting material is available in the book edited by Rojas and Hashagen (2000), which includes detailed descriptions about the architectures of the first computers ever built.

The interested reader on issues related to computer organisation and design have several reference books to detail the material presented here. The authoritative book authored by Patterson and Hennessy (2014) deserves a special attention. A good source is the book by Bryant and O'Hallaron (2016), because it describes the concepts underlying all computer systems, but with a programmer's perspective. The book is focused on helping programmers to profit from their knowledge of a computer system to write better programs. Stallings (2019) also provide a very authoritative presentation of computer organisation and architecture.

A very detailed description of different types of secondary memory is presented by Tanenbaum and Austin (2013, Chapter 2). Another important reference for memory systems is provided by Jacob et al. (2008).

Chapter 2
Representation of information

Abstract The different forms of data found in computers may be grouped in three general categories: (1) the numbers used in arithmetic calculations, (2) the letters of the alphabet that are combined to form words in the writing systems of the languages, and (3) other types of discrete symbols that humans use for different purposes. All computers represent these types of data in binary-coded formats, i.e., with bits. Despite this fact, whenever the computer information is presented to humans, it must be transformed into numbers, letters of the alphabet or other symbols. This chapter introduces the main concepts and techniques related to the representation of information in a computer for the last two categories indicated above. The representation of numbers is the subject of the next chapter.

2.1 Digital abstraction

Analog (or continuous) systems process time-varying signals that can take on any value across a continuous range of voltage, current or other metric, as Fig. 2.1(a) shows. The same happens with digital systems, but the difference is that one pretends that they do not! Digital systems have signals (inputs and outputs) that are represented by discrete (i.e., non-continuous) values. Fig. 2.1(b) depicts a possible waveform of a signal of a digital system. The X axis represents time, while the Y axis is the measured voltage. In this case, the system has two possible values represented by -5V and +5V. This type of system is called binary digital system. Thus, a digital signal is modelled as taking on, at any instant, only one of two possible values, which are designated as '0' and '1' (or 'low' and 'high', 'false' and 'true', 'inactive' and 'active', 'down' and 'up', etc.).

There is nothing intrinsic to the digital approach that limits it to only two values. The fundamental aspect is that the set of possible values is finite. The simplest form of digital systems is binary, where there are two possible values for the signals. The more values that must be distinguished, the less separation between adjacent values, and the less reliable is the mechanism. Thus, the binary numeral system is the most reliable method for encoding digital information, since it is easier to distinguish two possible values with physical entities than, say, five or ten. In particular, binary-coded information is the simplest one to use

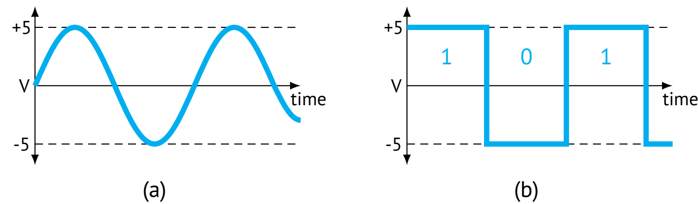


Fig. 2.1 (a) analog and (b) digital waveforms.

in digital computers, since it is easy to build electronic circuits that handle two alternative conditions that can naturally code values '0' and '1'. In electronic devices, the values '0' and '1' might be physically realised by two different voltage values (e.g., 0 volts vs. +5 volts). However, these two values can be represented with other technologies: magnetic polarisation (north vs. south), electrical current (flow vs. absence), relay logic (circuit open vs. circuit closed), fibre optics (light off vs. light on), pneumatic logic (fluid at low pressure vs. fluid at high pressure). The greatest advantage of digital systems is their rigorous formulation based on mathematical logic and Boolean algebra.

The fundamental advantage of digital systems with respect to analog ones is their ability to deal with electrical signals that have been degraded by transmission through the circuits. Due to the discrete nature of the output signals, a small variation in an input value is still interpreted correctly. In analog circuits, this behaviour does not occur as a slight error at an input generates an error at the output.

Digital circuits deal with analog voltages and currents. The **digital abstraction** allows analog behaviour to be ignored, so circuits can be modelled as if they really process 0s and 1s.

2.2 Bits, bytes and words

The digital abstraction hides the problems of the analog world by mapping the infinite set of real values for a physical quantity into two subsets. These two subsets correspond to two possible numbers or logic values: 0 and 1. Consequently, digital circuits can be analysed in their functionality using Boolean algebra. A logic value, 0 or 1, is often called a **bit**, a short form for binary digit.

In isolation, a single bit is not very useful, since it just permits to represent two possible values. However, if more values need to be represented, additional bits can be considered. When bits are grouped and coded, the elements of any finite set can be represented. This can be achieved by assigning some interpretation that gives meaning to the different possible bit patterns. With n bits, a maximum of 2^n different entities can be represented. Groups of bits can be used to encode numbers, as explained in Chapter 3. By using a standard character code, the letters and symbols in a document can also be encoded with a set of bits. Music, images, video and other media can also be digitally represented. In fact, all information that is processed by computers is represented by patterns of bits, often long

ones. In any case, a group of bits has no meaning of its own, nor anything that intrinsically indicates what its representation is. Its meaning is given by the instructions that uses those bits.

A block of 8 bits, designated as a **byte**, is the smallest addressable unit of memory in most computers. Each half of a byte is called a **nibble**, which can be represented by a 4-bit pattern or a hexadecimal digit (see Section 3.2).

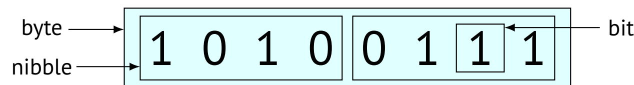


Fig. 2.2 A byte.

The unit symbol for the byte is the upper-case letter B.[1] For multiple-byte units, it is common to use systems based on powers of 10 or 2. Unfortunately, the nomenclature for these systems has promoted some confusion. Systems based on powers of 10 use the standard prefixes (kilo, mega, giga, ...) and the respective symbols (k, M, G, ...). Systems based on powers of 2 are supposed to use binary prefixes (kibi, mebi, gibi, ...) and the corresponding symbols (Ki, Mi, Gi, ...). Table 2.1 presents the metrics for both the decimal and the binary systems.

decimal			binary		
value		metric	value		metric
10^3	1000	kB kilobyte	2^{10}	1024	KiB kibibyte
10^6	1000^2	MB megabyte	2^{20}	1024^2	MiB mebibyte
10^9	1000^3	GB gigabyte	2^{30}	1024^3	GiB gibibyte
10^{12}	1000^4	TB terabyte	2^{40}	1024^4	TiB tebibyte
10^{15}	1000^5	PB petabyte	2^{50}	1024^5	PiB pebibyte
10^{18}	1000^6	EB exabyte	2^{60}	1024^6	EiB exbibyte
10^{21}	1000^7	ZB zettabyte	2^{70}	1024^7	KiB zebibyte

Table 2.1 Multiple-byte units.

A machine-level program views main memory as a large array of bytes. Every byte in the memory is identified by a unique number, its address. Fig. 2.3 shows a memory with 256 locations (or cells), each one storing one byte. To specify each memory location, a 8-bit address is needed. It is important to distinguish the memory address from its contents. In the figure, the contents of memory cell 0000 0100 is 1100 0110.

A **word** is the basic unit of data handled by a given family of computers. The sizes of words historically range from four bits, for early microprocessors like the Intel 4004, to 60 bits, for early mainframes (CDC 6600). Modern computers have word sizes of 16, 32,

[1] B is the symbol of the bel, a unit of logarithmic power ratio named after Alexander Graham Bell, creating a conflict. Since the bel is seldom used, little danger of confusion exists. Usually, it is used in its decadic fraction, the decibel (dB).

Address	Content
0000 0000	0010 1000
0000 0001	1110 1001
0000 0010	0001 0000
0000 0011	0100 0111
0000 0100	1100 0110
0000 0101	0100 1101
...	...
1111 1101	0000 0001
1111 1110	1111 0000
1111 1111	0100 1100

Fig. 2.3 A memory with 256 locations and addresses with 8 bits.

or 64 bits. The word size is a relevant characteristic of any specific processor or computer architecture.

2.3 Textual information

Text is the most common type of nonnumeric data that humans use, so computers need to represent it. In a computer, each alphanumeric character is represented by a bit pattern according to an established convention (code). The most commonly used character encoding standard is **ASCII** (American Standard Code for Information Interchange). The ASCII code is used in computers, telecommunications equipment, and other devices, and represents each character with a 7-bit string.

As Table 2.2 shows, the ASCII code includes 59 printable symbols: the digits 0 to 9, lowercase letters a to z, uppercase letters A to Z, and several punctuation symbols. Additionally, 33 non-printing control codes are considered, including for example the carriage return, the line feed and the tab. For example, lowercase j would be represented in the ASCII encoding by the 1101010 binary pattern.

ASCII encoding uses seven bits, but in practice, characters are not stored in groups of 7 bits. Instead, one ASCII symbol is stored in a byte, with the leftmost bit usually set to 0. Different manufacturers extended the ASCII code to take advantage of the 8th bit. The general idea was to have 128 additional characters for the bit patterns whose leftmost bit is 1. A string, like for instance the English word "Digital", is encoded in the C programming language by an array of ASCII characters, terminated by the null character, whose ASCII code is '00000000'. So, this word to be represented needs eight characters, i.e., 64 bits, as depicted in Fig. 2.4.

The set of symbols provided by ASCII is too short, since there are many different symbols in the various alphabets. ASCII does not include several accents used in European languages or larger alphabets, such as Cyrillic (the Russian alphabet) and Chinese Mandarin. Thus, modern computers use **Unicode**, which is a standard for the encoding, representation, and handling of text expressed in most of the world's writing systems. It provides the

binary	char	binary	char	binary	char	binary	char
0000000	NUL - null	0100000	space	1000000	@	1100000	`
0000001	SOH - start of header	0100001	!	1000001	A	1100001	a
0000010	STX - start of text	0100010	"	1000010	B	1100010	b
0000011	ETX - end of text	0100011	#	1000011	C	1100011	c
0000100	EOT - end of transmission	0100100	$	1000100	D	1100100	d
0000101	ENQ - enquiry	0100101	%	1000101	E	1100101	e
0000110	ACK - acknowledge	0100110	&	1000110	F	1100110	f
0000111	BEL - bell	0100111	'	1000111	G	1100111	g
0001000	BS - backspace	0101000	(	1001000	H	1101000	h
0001001	HT - horizontal tab	0101001	)	1001001	I	1101001	i
0001010	LF - line feed	0101010	*	1001010	J	1101010	j
0001011	VT - vertical tab	0101011	+	1001011	K	1101011	k
0001100	FF - form feed	0101100	,	1001100	L	1101100	l
0001101	CR - carriage return	0101101	-	1001101	M	1101101	m
0001110	SO - shift out	0101110	.	1001110	N	1101110	n
0001111	SI - shift in	0101111	/	1001111	O	1101111	o
0010000	DLE - data link escape	0110000	0	1010000	P	1110000	p
0010001	DC1 - device control 1	0110001	1	1010001	Q	1110001	q
0010010	DC2 - device control 2	0110010	2	1010010	R	1110010	r
0010011	DC3 - device control 3	0110011	3	1010011	S	1110011	s
0010100	DC4 - device control 4	0110100	4	1010100	T	1110100	t
0010101	NAK - negative acknowledge	0110101	5	1010101	U	1110101	u
0010110	SYN - synchronize	0110110	6	1010110	V	1110110	v
0010111	ETB - end transm. block	0110111	7	1010111	W	1110111	w
0011000	CAN - cancel	0111000	8	1011000	X	1111000	x
0011001	EM - end of medium	0111001	9	1011001	Y	1111001	y
0011010	SUB - substitute	0111010	:	1011010	Z	1111010	z
0011011	ESC - escape	0111011	;	1011011	[	1111011	{
0011100	FS - file separator	0111100	<	1011100	\	1111100	|
0011101	GS - group separator	0111101	=	1011101	]	1111101	}
0011110	RS - record separator	0111110	>	1011110	^	1111110	~
0011111	US - unit separator	0111111	?	1011111	_	1111111	DEL

Table 2.2 The ASCII table.

D	i	g	i	t	a	l	'\0'
01000100	01101001	01100111	01101001	01110100	01100001	01101100	00000000

Fig. 2.4 Binary pattern for the string "Digital".

capacity to represent all characters used for the written languages, since more than one million characters can be encoded. As of March 2020, Unicode has a total of 143,859 characters. Unicode 13.0 covers 154 modern and historic scripts, as well as multiple symbol sets and emoji. Unicode can be implemented by different character encodings, like UTF-8, UTF-16, and UTF-32. UTF-8 is currently the dominant encoding on the World Wide Web. Unicode is the representation scheme adopted by modern standards, such as CORBA 3.0, Java, LDAP, WML, and XML. The majority of the common-use characters fit into the first 64k code patterns, which just require two bytes (16 bits).

2.4 Machine-level instructions

A computer program is a sequence of instructions. At the machine-level, each instruction is represented by a bit pattern. Generally, it consists of an opcode and some additional information, such as where operands come from and where to store the results. Fig. 2.5 shows several possible formats for machine-level instructions. An instruction always has an opcode to indicate what operation is to be performed. The number of arguments (or operands) varies from zero to three, depending on the instruction and on the general format considered by the specific processor.

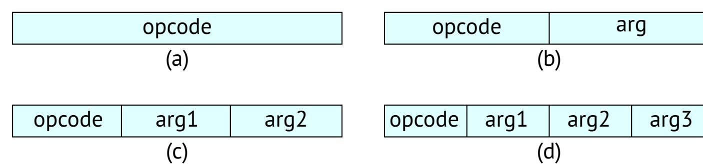
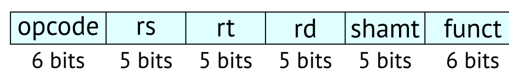


Fig. 2.5 Four common instruction formats: (a) Zero-argument instruction; (b) One-argument instruction; (c) Two-argument instruction; (d) Three-argument instruction.

On some machines, all instructions have the same length; on others there may be many different lengths. The opcode for each instruction type must be associated with a unique bit pattern, to identify it univocally. The instructions of the MIPS processor all have 32 bits. They are classified according to five different types (R, I, J, FR, FI). The R instructions have all the data values located in registers. The syntax of the R instructions is:

```
OP rd, rs, rt
```

where OP is the mnemonic for the particular instruction, rs and rt are the source registers, and rd is the destination register. The machine code for an R instruction has the following format:



An example of an R instruction is:

```
add $t1, $t2, $t3
```

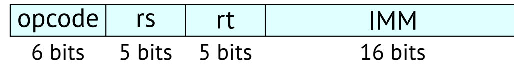
where \$t1, \$t2, and \$t3 are registers. This instruction adds the values of registers \$t2 and \$t3 and stores the result in register \$t1. This instruction is represented by the following bit pattern:

000000	01010	01011	01001	00000	100000

The I instructions are used when the instruction operates on an immediate value and a register value. Their syntax is:

```
OP rd, rs, IMM
```

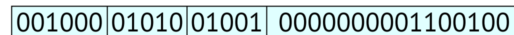
where OP is the mnemonic for the particular instruction, rs is the source register, IMM is an immediate value (with a maximum of 16 bits), and rd is the destination register. The machine code for an I instruction has the following format:



An example of an I instruction is an addition instruction that adds 100 to the contents of register $t2 and stores the result in register $t1:

```
addi $t1, $t2, 100
```

This instruction is represented by the following bit pattern:



2.5 Images

A digital image can be represented by a grid of small points. This image is a **raster image** (or simply bitmap image) as it is seen as an array of points. Each point is called a **pixel** (short for picture element) and is represented by a binary pattern. Black and white images represent, for example, black by '1' and white by '0'. Thus, each black and white image is a set of 0s and 1s. To draw the picture, a grid must be defined and the squares coloured accordingly. If an image has size 4x6 (in pixels), then an 'A' can be drawn with by the grid in Fig. 2.6(a), whose sequence of bits from bottom to top and from let to right is 01101001 10011111 10011001.

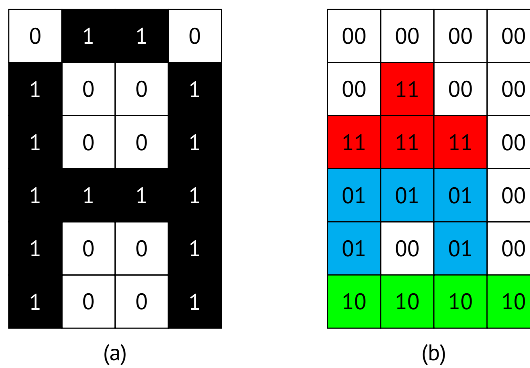


Fig. 2.6 (a) Black and white image, (b) coloured image.

The size of the image needs to be stored and is part of the metadata (data about data) of an image. Other examples of metadata are the date and the time when the image was created, the name of its creator, and the type of compression used to store the image.

Adding colours to a picture entails enlarging the number of bits that are used to represent each pixel. If two bits are used, four colours can be represented. For example, if the following encoding is used ('00' white, '01' blue, '10' green, and '11' red), the image in Fig. 2.6(b) is represented by the bit pattern 00000000 00110000 11111100 01010100 01000100 10101010. Many image formats use the RGB (Red-Green-Blue) colour model. RGB has three channels to define a wide assortment of colours. RGB was originally created to represent and display images in electronic devices, such as TV sets and computers, but it has also been used in photography and digital images. If each channel has 8 bits and all channels are at 0, a pure black colour is obtained. Similarly, if all channels are at 255, an all-white colour is defined. By adjusting the red, green, and blue channels, many different colour can be defined, including grays, whites, and blacks.

The number of bits used to store each pixel is called the **colour depth**. Images with more possible colours require obviously more bits to specify each one, so they are stored in larger files. The image quality depends on the **image resolution**, which is related to how close the pixels are. It is usually measured in dots per inch (dpi),[2] that is, the number of dots/pixels that can be placed in a line within the span of 1 inch (2.54 cm). In a low-resolution image, the pixels are larger so fewer are needed to fill the space. This results in images that may look pixelated. An image with a higher resolution has more pixels, so it looks better when it is zoomed in. The disadvantage of having more pixels is that the file size is larger.

A **vector image** uses scalable shapes, such as lines and curves. Coordinates and geometry are used to precisely define the elements of the image. It is more efficient than bitmaps at storing, for example, large areas of the same colour, since it does not need to store every pixel as a bitmap image does. Additionally, a vector image can be scaled (enlarged or reduced in size), without losing quality.

2.6 Audio

Sound waves in nature have an analog nature. To process sound (or audio), computers need to convert it into a digital format. Firstly, sound is recorded using a device, like a microphone, that translates sound waves into an electrical signal, as illustrated in Fig. 2.7(a). Then, periodic measurements of the level of that signal are registered, as shown in Fig. 2.7(b). This process that reduces a continuous-time signal to a discrete-time signal is called **sampling**. The value (or set of values) at a point in time is designated **sample**. The samples are then simply converted into binary, using a unique binary code. Afterwards, the digital sound can be processed by a computer as a sequence of bits.

The **sample resolution** is the number of bits used to represent the value of each sample. If more bits are available for each sample, then more levels of the signal can be captured, which improves the precision of the recording. When Fig. 2.7(b) and (c) are compared, it should be clear that the latter has higher precision. The digital sample in (b) has a low sampling resolution when compared to the digital sample in (c). The minimum step of two consecutive digital values in the former corresponds to 1 V (the value is rounded to the nearest whole number: 0, 1, 2, ...), while in the latter to 0.5 V (the value is rounded to

[2] Other metrics also exist, like dots per centimetre and pixels per inch.

the nearest half integer: 0, 0.5, 1.0, ...). The sample in (c) is thus more accurate, i.e., it has smaller rounding errors.

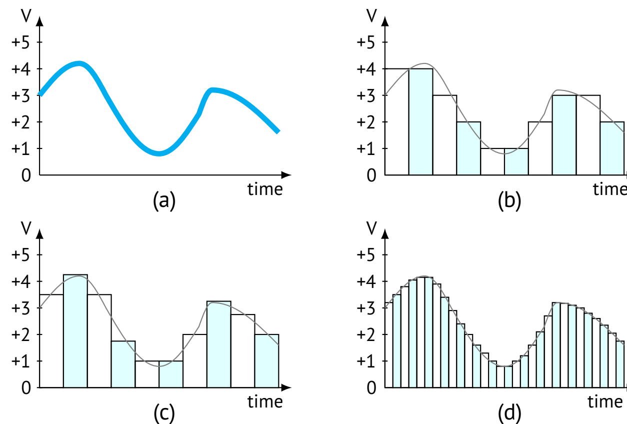


Fig. 2.7 (a) analog audio signal; digital signals with (b) a low sampling rate and a low sample resolution, (c) a high sampling resolution, (d) a high sampling rate.

The number of samples taken per second, measured in Hertz (Hz), is the **sampling rate**. The higher the sampling rate, the better the quality of the audio digital signal, because, if there are more samples, the original sound can be represented more accurately. When Fig. 2.7(b) and (d) are compared, it is clear that the latter has more quality, that is, it represents with more accuracy the original signal.

To determine the size s of a sound signal in bits, three values are multiplied: $s = f \times r \times t$. In this formula, f is the sampling rate in Hertz, r is the sample resolution measured in number of bits, and t is the duration of the signal in seconds.

Exercises

Exerc. 2.1: To encode Roman numbers (from 1 to 899), the following binary encoding for the symbols has been proposed: I (01), V (100), X (00), L (101), C (110), D (111). Indicate whether this encoding is valid and, if so, what Roman number is represented by the binary pattern 111101000101.
Exerc. 2.2: Decode the following ASCII string:
1010101 0101110 0100000 1001101 1101001 1101110 1101000 1101111.
Exerc. 2.3: A digital image has 128x128 pixels. Each pixel in the image stores information related to three channels (Red, Blue, Green). If each channel is capable of distinguishing 256 different tones, indicate the size in bytes of the image.
Exerc. 2.4: An image occupies 192 kibibytes and has dimensions of 256x512 pixels. Each pixel is represented by three unsigned integer values, which indicate the intensity of each

channel (Red-Green-Blue) in that pixel. Indicate, in binary and decimal, the maximum value that can be assigned to each of these integers, if they all have the same size.
Exerc. 2.5: The CYMK* subtractive colour system is formed by Cyan, Magenta, Yellow and Black and works due to the absorption of light, as the colours that are seen come from the part of the light that is not absorbed. Each pixel is represented by four 6-bit patterns that indicate the intensity in each channel. Indicate how many different colours a pixel can have, assuming that the "00000-" ("000000" and "000001") patterns cannot be used.
Exerc. 2.6: The SCB system for evaluating football players consists of three parameters: Strength, Courage and Braveness. Each parameter is represented by a binary pattern (of 7 bits each) that indicates the respective intensity. Indicate the number of different valid assessments with this system, if the "1111111" and "1111110" patterns represent evaluations that are still unknown or invalid, respectively.
Exerc. 2.7: Calculate the size in kB of a sound file, if the recording lasts exactly 2 minutes and it is sampled using a sampling rate of 50 kHz and a sample resolution of 8 bits. What is the size in KiB?
Exerc. 2.8: Calculate the sample resolution of a sound file with 4.5kB, if the recording lasts one minute with a sampling rate of 50 Hz.
Exerc. 2.9: Consider that a typical daily newspaper page contains 3700 Unicode characters including white spaces. **(a)** How many bytes are needed to encode a 32-page edition of that daily newspaper, if it includes on average 50 photos (1.2MiB each one)? **(b)** How many mebibytes are needed to store all the numbers of the newspaper published in a year? **(c)** If a library contains 1250 different daily newspapers, which have on average 50 years of publication, how many pebibytes are stored there?

Further reading

Information theory is a field that deserves to be studied by all computer scientists. Is is focused in studying the quantification, storage, and communication of information and was established by the works of Harry Nyquist, Ralph Hartley, and Claude Shannon. Extensive introductions to this field are provided by McEliece (2005) and Borda (2011).

Boolean algebra and logic design are classical topics in many courses related to the fundamentals of computers, but they are not addressed in this book. Boolean algebra allows the formal treatment of logic. This is relevant since the properties of electrical switching circuits can be represented by a two-valued Boolean algebra (called switching algebra). The interested reader is pointed to books by Katz (1994), Gajski (1997), and Wakerly (2001), which make a very good coverage of these fundamental topics.

The reader interested in understanding the Unicode is pointed to the book written by the Unicode Consortium (2007), which describes the 5th edition of this universal character encoding standard for written characters and text.

Chapter 3
Representation of numbers

Abstract This chapter introduces the different approaches and techniques used to represent numeric information in computers. The positional numeral systems are described. The use of numbers in the octal and hexadecimal systems and the conversion between different bases are discussed. Different alternatives to represent integer numbers, negative numbers, and real numbers are also presented. The representation of decimal numbers with binary codes closes the chapter.

3.1 Positional numeral systems

A **numeral system** (or number system) is a writing system for expressing numbers, that is, a notation for representing numbers of a given set, using digits or other symbols. To represent numbers, computers do not use the decimal numeral system used by humans, also called the Hindu–Arabic numeral system, but rather the binary numeral system. Each one is a **positional numeral system**, which represents any number by a sequence of juxtaposed digits.

A **digit** is a symbol used alone or in combinations to represent numbers in a positional numeral system. The position of each digit has a corresponding weight. The mathematical value of a number is provided by the weighted sum of all its digits. For decimal integer numbers, the weights are powers of ten equal to 10^i, where i is the position of the digit starting from the right, as occurs in this example:

$$4682 = 4 \cdot 1000 + 6 \cdot 100 + 8 \cdot 10 + 2 \cdot 1$$

Here, the value 10 is called the **radix** (or base) of the numeral system. The value of a decimal integer number D of the form $d_3d_2d_1d_0$ is:

$$D = d_3 \cdot 10^3 + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$$

Positional numeral systems can use any integer $r \geq 2$ for the base, and the digit in position i has weight r^i. So, the general format of a n-digit number D in such a system is:

$$D = d_{n-1}d_{n-2}\dots d_1 d_0$$

The value of this **natural number** is given by the following formula:

$$D = \sum_{i=0}^{n-1} d_i \cdot r^i \tag{3.1}$$

Digital devices adopt the binary base ($r = 2$), which uses two possible digits (0 and 1). The general format of a number B in such base is:

$$B = b_{n-1}b_{n-2}\dots b_1 b_0$$

The value of this natural number is:

$$B = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

The typical representation for natural numbers in digital computers involves a fixed number of bits, usually 8, 16, 32, or 64. Thus, these numbers have a fixed **range** that is defined as the difference between the largest and the smallest representable numbers in a given numeral system. A representation with n bits ranges from 0 to $2^n - 1$. With 8 bits, this range goes from 0 to +255.

The memory of a computer is organised as a linear array of bytes, each with its own unique address starting at zero. Memory addresses in the computer are obviously represented as binary numbers. If an address has m bits, the maximum number of addressable cells is 2^m. The number of bits in the address determines the maximum number of directly addressable cells in the memory and does not depend on the number of bits per cell. For example, one needs 10-bit addresses, either for a memory with 2^{10} cells of 8 bits each or for a memory with 2^{10} cells of 37 bits each.

The previous concepts all apply to natural numbers, but can also be extended to include fractions in any base system, which can be approximated using negative powers of a radix. The integer part of a number is separated from its fractional part by a **radix point**. In the decimal system, the radix point is called **decimal point**.

The next example shows a decimal number[1] that has two decimal digits to the right of the decimal point:

$$4682.51 = 4 \cdot 1000 + 6 \cdot 100 + 8 \cdot 10 + 2 \cdot 1 + 5 \cdot 0.1 + 1 \cdot 0.01$$

The value of a decimal number D of the form $d_2 d_1 d_0 . d_{-1} d_{-2}$ is:

$$D = d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2}$$

Binary fractions have a **binary point**. The general format of a number B in base 2, with m digits to the right of the radix point and n digits to the left, is:

[1] The term "decimal number" is often used as an equivalent to "decimal representation of a number", which is more precise; Likewise for other representations.

$$B = b_{n-1}b_{n-2}\dots b_1 b_0.b_{-1}b_{-2}\dots b_{-m}$$

The value of this number is:

$$B = \sum_{i=-m}^{n-1} b_i \cdot 2^i \tag{3.2}$$

This equation can be generalised to numbers represented in any base r.

$$D = \sum_{i=-m}^{n-1} b_i \cdot r^i \tag{3.3}$$

In positional numeral systems, the rightmost digit is designated as **least-significant digit (LSD)**, while the rightmost one is the **most-significant digit (MSD)**. For the particular case of binary numbers, the LSD is designated as **least-significant bit (LSB)**, while MSD is called as **most-significant bit (MSB)**.

Whenever one uses numbers in different bases, a subscript must be used to indicate which base is used in each case. Here, are some examples of binary numbers and their corresponding values in decimal.

$$\begin{aligned}
11001_2 &= 1\cdot16+1\cdot8+0\cdot4+0\cdot2+1\cdot1 &&= 25_{10}\\
111001_2 &= 1\cdot32+1\cdot16+1\cdot8+0\cdot4+0\cdot2+1\cdot1 &&= 57_{10}\\
11.101_2 &= 1\cdot2+1\cdot1+1\cdot0.5+0\cdot0.25+1\cdot0.125 &&= 3.625_{10}\\
0.1111_2 &= 1\cdot0.5+1\cdot0.25+1\cdot0.125+1\cdot0.0625 &&= 0.9375_{10}
\end{aligned}$$

3.2 Octal and hexadecimal numbers

Base 10 is important and useful, since humans use it in every day activities, and the same happens for base 2, as binary numbers are processed by digital devices. Since the binary notation is too verbose, the bases 8 (octal) and 16 (hexadecimal) are also popular among computer scientists, namely for documentation purposes. They are used to offer a shorthand representation for binary numbers, i.e., they allow long patterns of bits to be represented by shorter patterns of octal or hexadecimal digits.

Base 8 is adopted by the **octal numeral system**, whereas base 16 supports the **hexadecimal numeral system**. The octal system uses eight different digits: 0 to 7 (0, 1, 2, 3, 4, 5, 6, 7). Similarly, the hexadecimal numeral system uses 16 digits: 0 to 9 and the letters A to F (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). Table 3.1 shows the binary numbers from 0 to 10001 and their octal, decimal, and hexadecimal representations.

Bases 8 and 16 are useful for representing multi-bit numbers, since they are integer powers of 2 ($8 = 2^3, 16 = 2^4$). So, as indicated in Table 3.1, octal and hexadecimal digits are directly represented by 3-bit and 4-bit strings, respectively. Therefore, converting a binary number to octal is straightforward. Starting at the binary point and moving to the left, the bits are separated into groups of three and each group is replaced by the corresponding octal digit.

$$\begin{aligned}100010101001_2 &= 100\ 010\ 101\ 001_2 &= 4251_8\\ 11111100011101110_2 &= 011\ 111\ 100\ 011\ 101\ 110_2 &= 374356_8\end{aligned}$$

The method for converting a binary number to hexadecimal is similar, but the groups are composed of four bits:

$$\begin{aligned}100010101011_2 &= 1000\ 1010\ 1011_2 &= 8AB_{16}\\ 11111100011101110_2 &= 0001\ 1111\ 1000\ 1110\ 1110_2 &= 1F8EE_{16}\end{aligned}$$

binary	decimal	octal	3-bit string	hexa-decimal	4-bit string
0	0	0	000	0	0000
1	1	1	001	1	0001
10	2	2	010	2	0010
11	3	3	011	3	0011
100	4	4	100	4	0100
101	5	5	101	5	0101
110	6	6	110	6	0110
111	7	7	111	7	0111
1000	8	10	-	8	1000
1001	9	11	-	9	1001
1010	10	12	-	A	1010
1011	11	13	-	B	1011
1100	12	14	-	C	1100
1101	13	15	-	D	1101
1110	14	16	-	E	1110
1111	15	17	-	F	1111
10000	16	20	-	10	-
10001	17	21	-	11	-

Table 3.1 Different representations for binary numbers 0 to 10001.

In some of these conversions, additional 0s were added to the left, which do not change the value. The idea is to make the total number of bits be a multiple of 3 or 4, accordingly. Whenever a binary number contains digits to the right of the binary point, the method for converting into octal and hexadecimal is also easy. Starting at the binary point and moving to the right, the bits are again separated in groups of three (or four) and are replaced by the corresponding octal (hexadecimal) digit. Here, additional 0s can be added to the right of the rightmost bit, as this operation does not change the value of the number, as intended.

$$\begin{aligned}.11001_2 &= .110\ 010_2 &= .62_8\\ &= .1100\ 1000_2 &= .C8_{16}\end{aligned}$$

The conversions in the opposite direction (from octal or hexadecimal to binary) are also very easy. Each octal or hexadecimal digit is simply transformed into the corresponding 3- or 4-bit string.

$$\begin{aligned}1234_8 &= 001\ 010\ 011\ 100_2\\ 234.05_8 &= 010\ 011\ 100.000\ 101_2\end{aligned}$$

$$\begin{aligned} 1234_{16} &= 0001\ 0010\ 0011\ 0100_2 \\ \mathrm{AB3.05}_{16} &= 1010\ 1011\ 0011.0000\ 0101_2 \end{aligned}$$

Hexadecimal numbers are often used to represent addresses in the memory space of a digital computer. For instance, in a computer with 32-bit addresses, each position in the memory is specified with eight hexadecimal digits, like for example $\mathrm{A835B300}_{16}$. In many programming languages, like C, the prefix `0x` is used to indicated hexadecimal numbers, like `0xA835B300`.

3.3 Conversions between different bases

In general, the conversion from one base to another one cannot be accomplished by just replacing digits in a base to the corresponding digits in the other base. This only occurs when both bases are integer powers of the same number (e.g., 2, 4, 8, and 16; 3, 9, and 27). In the other cases, arithmetic operations are needed.

Equation 3.3 calculates the value of a number in base r, that has m digits to the right of the radix point and n digits to the left. The value can thus be calculated by converting each digit to the respective base-10 equivalent and by adding all these values. Some examples are given:

$$\begin{aligned} \mathrm{AB3}_{16} &= 10 \cdot 16^2 + 11 \cdot 16^1 + 3 \cdot 16^0 && = 2819_{10} \\ 2345_8 &= 2 \cdot 8^3 + 3 \cdot 8^2 + 4 \cdot 8^1 + 5 \cdot 8^0 && = 1253_{10} \\ 124.7_8 &= 1 \cdot 8^2 + 2 \cdot 8^1 + 4 \cdot 8^0 + 7 \cdot 8^{-1} && = 84.875_{10} \\ 120.01_3 &= 1 \cdot 3^2 + 2 \cdot 3^1 + 0 \cdot 3^0 + 0 \cdot 3^{-1} + 1 \cdot 3^{-2} && = 15.(1)_{10} \end{aligned}$$

The last example shows that fractions that contain repeating strings of digits to the right of the radix point in a given base may not necessarily have a repeating sequence of digits in another base.

Equation 3.1 can be rewritten in order to highlight a simple procedure for converting numbers to base 10

$$D = ((\ldots((d_{n-1}) \cdot r + d_{n-2}) \cdot r + \ldots) \cdot r + d_1) \cdot r + d_0 \tag{3.4}$$

When this formula is applied to hexadecimal number $A835_{16}$, the following expression can be used to calculate its value:

$$A835_{16} = (((10) \cdot 16 + 8) \cdot 16 + 3) \cdot 16 + 5$$

Algorithm 1 calculates the decimal value of an integer number represented with n digits in base b.

Equation 3.4 can also be used to convert a decimal number D to the equivalent number in base r. If D in Equation 3.4 is divided by r, d_0 is the remainder and the part inside parentheses corresponds to the quotient Q:

$$Q = (\ldots((d_{m-1}) \cdot r + d_{m-2}) \cdot r + \ldots) \cdot r + d_1 \tag{3.5}$$

Algorithm 1: calculation of the decimal value of a number in a given base

```
input  : A number d in base b with n digits (d_{n-1} d_{n-2} ... d_2 d_1 d_0)
output: The decimal value v of number d

v ← 0 ;
i ← n − 1 ;
while i ≥ 0 do
    v ← v · b + d_i ;
    i ← i − 1 ;
end
```

Since Q follows the same format as D, successive divisions by r yield successive digits of D, from right to left. Some examples illustrate how this method works for converting decimal number 177 to bases 2, 8, and 16:

$177 \div 2 = 88$ remainder 1
$88 \div 2 = 44$ remainder 0
$44 \div 2 = 22$ remainder 0
$22 \div 2 = 11$ remainder 0
$11 \div 2 = 5$ remainder 1
$5 \div 2 = 2$ remainder 1
$2 \div 2 = 1$ remainder 0
$1 \div 2 = 0$ remainder 1
$\boxed{177_{10} = 10110001_2}$

$177 \div 8 = 22$ remainder 1
$22 \div 8 = 2$ remainder 6
$2 \div 8 = 0$ remainder 2
$\boxed{177_{10} = 261_8}$

$177 \div 16 = 11$ remainder 1
$11 \div 16 = 0$ remainder 11
$\boxed{177_{10} = \mathrm{B}1_{16}}$

The methods just described can be followed to directly convert any number in a given base to any other base. However, often, it is easier to first convert to the decimal base and then to the target base. The next example shows how to convert 3410_5 to base 3:

$$3410_5 = 3 \cdot 5^3 + 4 \cdot 5^2 + 1 \cdot 5^1 + 0 \cdot 5^0 = 480_{10}$$

$480 \div 3 = 160$ remainder 0
$160 \div 3 = 53$ remainder 1
$53 \div 3 = 17$ remainder 2
$17 \div 3 = 5$ remainder 2
$5 \div 3 = 1$ remainder 2
$1 \div 3 = 0$ remainder 1
$\boxed{3410_5 = 122210_3}$

3.4 Negative numbers

Up this point, only natural (i.e., nonnegative) numbers were considered. Representing signed numbers, together with positive ones, requires additional issues to be addressed, namely the inclusion of a **sign bit**. There are many ways to represent negative numbers and this section discusses four alternatives: sign-magnitude, one's-complement, two's-complement, and excess representations.

3.4.1 Sign-magnitude

The most intuitive method, **sign-magnitude**, uses the MSB for the sign bit and the remaining bits for the **magnitude** of the number, i.e., its modulus or absolute value. By convention, a '1' in the sign bit indicates a negative number, whereas a '0' indicates a positive number (or zero). In a 8-bit representation, two symmetrical values just differ in the sign bit, as next illustrated:

$$+120_{10} = 01111000_2$$
$$-120_{10} = 11111000_2$$

It is mandatory to know how many bits are used to represent the numbers. The number 1100_2 is negative if four bits are used, but positive if instead the numbers are represented with six bits. In this case, extra 0s can be added to the left part of the number (001100_2).

A disadvantage of this method is the existence of two possible representations of zero ("+0" and "-0"). The sign-magnitude contains the same number of positive and negative numbers. A sign-magnitude integer representation with n bits ranges from $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$. With 8 bits, this range goes from -127 to $+127$.

Signed-magnitude calculations are performed basically with the same method as humans use with pencil and paper. As an example, consider the rules for addition:

1. If the signs are the same, add the magnitudes and use that same sign for the result;
2. If the signs differ, determine which operand has the larger magnitude. The sign of the result is the same as the sign of the operand with the larger magnitude, and the magnitude must be obtained by subtracting the smaller one from the larger one.

3.4.2 One's-complement

Complement numeral systems were created to make additions/subtractions faster and easier. The idea is to use complementation, which is easy to apply on binary numbers.

Let us analyse how this approach generally works for decimals. One decimal number can be subtracted from another by adding the difference of the subtrahend from all nines and adding back a carry. This is called taking the nine's complement of the subtrahend, or more precisely, finding the diminished radix complement of the subtrahend. Assume that one wants to calculate $165-43$. The difference of 43 from 999 is 956 and, in nine's complement arithmetic, $165-43 = 165+956 = (1)121$. The "carry" from the hundreds column is added back to the units place, yielding the correct result $165-43 = 121+1 = 122$.

This process works similarly in the **one's complement numeral system**. This system makes computer arithmetic simple and has the great advantage when compared with sign-magnitude that there is no need to handle sign bits separately. Anyway, one can still easily identify the sign of a number by checking its MSB. The one's-complements of binary numbers is computed the same way as for natural numbers, except that the weight of the MSB is $-2^{n-1}+1$ instead of -2^{n-1}. The next example shows how to compute directly the values of 6-bit one's-complement numbers:

$$011001_2 = 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 1 = 25_{10}$$
$$111001_2 = 1 \cdot (-31) + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 1 = -6_{10}$$

Given a number V in base r having n digits, the diminished radix complement of V is $r^n - 1 - V$. The range of representable integer numbers with n bits in the diminished radix complement representation is the same as for sign-magnitude one.

For a binary number, its **one's complement** is obtained by subtracting from all ones. For example, the one's-complement of 0101_2 is $1111 - 0101 = 1010_2$. The one's complement of a binary number can be obtained by just toggling all the bits (i.e., change the 1s to 0s and vice versa), an operation that is quite simple to implement in computer hardware.

Complement notation simplifies subtraction by turning it into addition. Additionally, it provides a method to represent negative numbers. The idea is that a negative number needs to be converted to its complement, which should have a '1' in the MSB. Positive numbers, which have a '0' in the MSB, are used as is, i.e., they are not converted to their complements. The next example illustrates these ideas.

$$+24_{10} = +(00011000_2) = 00011000_2$$
$$-24_{10} = -(00011000_2) = 11100111_2$$
$$-10_{10} = -(00001010_2) = 11110101_2$$

To subtract 10 from 24, one needs first to express the subtrahend (10) in one's-complement and then add it to the minuend (24). This effectively adds -10 to 24. The MSB will have a '0' or a '1' carry, which needs to be added to the LSB of the sum. This operation is designated **end carry-around** and results from the use of the diminished radix complement, in this case the one's-complement. The next example shows the bit operations to calculate $24 - 10$ and $10 - 24$ using one's-complement.

$1 \leftarrow$	1 1 1	$\Leftarrow$	carries	$0 \leftarrow$	1 1 1	$\Leftarrow$	carries
	0 0 0 1 1 0 0 0		(24_{10})		0 0 0 0 1 0 1 0		(10_{10})
	1 1 1 1 0 1 0 1		$+(-10_{10})$		1 1 1 0 0 1 1 1		$+(-24_{10})$
	0 0 0 0 1 1 0 1				1 1 1 1 0 0 0 1		
	+ 1				+ 0		
	0 0 0 0 1 1 1 0		(14_{10})		1 1 1 1 0 0 0 1		(-14_{10})

The range of representable integer numbers with n bits is $-(2^{n-1} - 1)$ through $+(2^{n-1} - 1)$. With 8 bits, this range goes from -127 to $+127$.

3.4.3 Two's-complement

Most digital computers do use the so-called **two's complement numeral system**. In fact, the radix complement is considered more intuitive than the diminished radix complement. Given a numeric value V in base r having n digits, the radix complement of V is defined to be $r^n - V$ for $V \neq 0$, and 0 for $V = 0$. With three decimal digits, the ten's complement of 43 is 957 $(10^3 - 43)$.

The decimal value for a **two's complement** number is computed the same way as for a natural number, except that the weight of the MSB is -2^{n-1} instead of $+2^{n-1}$. The next example shows how to compute directly the values of 6-bit two's-complement numbers:

$$\begin{aligned} 011001_2 &= 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 1 &= 25_{10} \\ 111001_2 &= 1 \cdot (-32) + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 1 &= -7_{10} \end{aligned}$$

A closer analysis shows that two's complement is just one's complement incremented by 1. To find the two's complement of a binary number, one just needs to flip bits and add 1. Since the subtrahend (the number that is complemented and added) is incremented at the outset, there is no end carry-around to perform. Any carries involving the MSB are simply discarded. Remember that only negative numbers need to be converted to two's complement notation, as the next example illustrates:

$$\begin{aligned} +24_{10} &= +(00011000_2) = 00011000_2 \\ -24_{10} &= -(00011000_2) = 11100111_2 + 1 = 11101000_2 \\ -10_{10} &= -(00001010_2) = 11110101_2 + 1 = 11110110_2 \end{aligned}$$

The next example shows the bit operations to calculate $24 - 10$ and $10 - 24$ using two's-complement. More details about addition with two's-complement numbers are given in section 3.5.

$1 \leftarrow$ 1 1 1	$\Leftarrow$ carries		$0 \leftarrow$ 1	$\Leftarrow$ carries
0 0 0 1 1 0 0 0	(24_{10})		0 0 0 0 1 0 1 0	(10_{10})
1 1 1 1 0 1 1 0	$+(-10_{10})$		1 1 1 0 1 0 0 0	$+(-24_{10})$
0 0 0 0 1 1 1 0	(14_{10})		1 1 1 1 0 0 1 0	(-14_{10})

The discarded carry in the example did not cause an erroneous result. An overflow occurs either if two positive numbers are added and the result is negative, or if two negative numbers are added and the result is positive. Formally, an integer **overflow** occurs when an arithmetic operation, like an addition, tries to create a numeric value that cannot be included in the range of the numeral system; either greater than the maximum representable value or smaller than the minimum one. When using two's complement notation, it is impossible to have overflow whenever adding a positive number with a negative one.

Two's-complement is the preferred choice for digitally representing signed numbers, since it does not require the addition and subtraction circuitry to examine the signs of the operands to determine whether to add or subtract. This property makes the system simpler to implement in hardware. Additionally, it has the best representation for 0 (all 0 bits), is self-inverting, and can be extended to larger numbers of bits. Finally, two's-complement numbers are added and subtracted by the same methods/algorithms as unsigned numbers with the same number of bits, so the same hardware can handle numbers in both cases. Its biggest drawback is the asymmetry in the range of values that can be represented, which may sound non-natural. The range of representable integer numbers with n bits is $-(2^{n-1})$ through $+(2^{n-1} - 1)$. For example, with 8 bits, this range goes from -128 to $+127$.

3.4.4 Excess representations

In an **excess-b representation**, an n-bit pattern, whose unsigned integer value is V ($0 \leq V < 2^n$) represents the signed integer $V - b$, where b is the **bias** (or offset) of the numeral system. The representable numeric values range from $-b$ to $2^n - 1 - b$. For example, with 8 bits and

$b = 100$, this range represents 256 different integer numbers, from -100 to $+155$. The main advantage of this digital coding scheme lies in the fact that the all-zero pattern corresponds to the minimal negative value and the all-one pattern to the maximal positive value. This facilitates the comparison of values, since a smaller numerical values has a bit pattern that is lexicographically smaller than that of a greater value. It also permits to represent unbalanced sets of consecutive negative and positive numbers (e.g., the subset from -10 to $+245$). From a mathematical point of view, the representation of natural numbers is an excess-0 representation.

Often, the bias for an n-bit binary word is $b = 2^{n-1}$, so that the number of negative numbers is the same as the number of positive numbers (including zero). With 8 bits, there is an excess-128 representation that includes the integer numbers from -128 to $+127$. The most common use of excess representations is in floating-point numeral systems, as explained in Section 3.6.

Table 3.2 shows, for four bits, the different numeral systems considered in this section, namely two excess representations ($b = 3$ and $b = 7$).

binary	un-signed	sign magn.	one's compl.	two's compl.	excess 3	excess 7
0000	0	0	0	0	-3	-7
0001	1	1	1	1	-2	-6
0010	2	2	2	2	-1	-5
0011	3	3	3	3	0	-4
0100	4	4	4	4	1	-3
0101	5	5	5	5	2	-2
0110	6	6	6	6	3	-1
0111	7	7	7	7	4	0
1000	8	-0	-7	-8	5	1
1001	9	-1	-6	-7	6	2
1010	10	-2	-5	-6	7	3
1011	11	-3	-4	-5	8	4
1100	12	-4	-3	-4	9	5
1101	13	-5	-2	-3	10	6
1110	14	-6	-1	-2	11	7
1111	15	-7	-0	-1	12	8

Table 3.2 Numerical values for binary numbers 0000 to 1111 in different representations.

3.5 Two's-complement addition

Adding values in the two's-complement numeral system requires the use of rules similar to the ones humans use to add decimals. An important difference is that the tables for binary numbers just contains 0s and 1s instead of decimal digits. Two decimal numbers are manually added, by adding each pair of digits at a time, starting with the LSDs of both numbers. If the sum is greater than 9, there is a carry that is transported to the next pair of digits

(the ones immediately to the left). When adding two binary numbers $A = a_{n-1}a_{n-2}\dots a_0$ and $B = b_{n-1}b_{n-2}\dots b_0$, the same procedure is applied, by starting to add the LSBs a_0 and b_0, with an initial carry c_0 equal to 0. This results in output carry bit c_1 and the sum bit s_1.

Table 3.3 shows the values of the sum s_i and the output carry bit c_{i+1}, for all possible combinations for the inputs (a_i, b_i, c_i). This process, formalised by algorithm 2, is repeated for all pairs of bits, proceeding from right to left.

a_i	b_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3.3 Addition of binary digits.

Algorithm 2: addition of two binary numbers

input : Two binary numbers a and b with n digits ($a_{n-1}a_{n-2}\dots a_1a_0$ and $b_{n-1}b_{n-2}\dots b_1b_0$)
output: The sum s of the input numbers

```
i ← 0 ;
c_0 ← 0 ;
while i < n do
    c_{i+1} ← carry(a_i, b_i, c_i) ;
    s_i ← sum(a_i, b_i, c_i) ;
    i ← i + 1 ;
end
```

Some examples of additions of two binary numbers are next given. Notice that an addition overflows whenever the signs of the addends are the same (both numbers are either positive or negative) and the sign of the sum is different from the addends' sign.

1	⇐	carries
0 1 0 1 1 0 1 0		(90_{10})
0 0 0 1 0 1 0 0		$+(20_{10})$
1 0 0 0 1 1 0 0		**overflow**

1	⇐	carries
0 1 0 1 1 0 1 0		(90_{10})
0 0 0 1 0 1 0 0		$+(-20_{10})$
0 1 0 0 0 1 1 0		(70_{10})

1 1 1 1	⇐	carries
1 0 1 0 0 1 1 0		(-90_{10})
1 1 0 0 1 1 1 0		$+(-50_{10})$
0 1 1 1 0 1 0 0		**overflow**

3.6 Floating-point numbers

Many numeral system assume that the radix point has a fixed position. In those cases, each number is a **fixed-point number**. When dealing with integers, the radix point is located on the right (i.e., after the LSD). So with five decimal digits, the values from 0 to 99,999 can be represented. This system represents 100 000 (10^5) different numbers, with every two consecutive numbers differing by 1. But one can consider the radix point to be in a different fixed position; for example, after the third decimal digit. In this case, the values range from 0 to 999.99. This system also represents 100000 different numbers, but every two consecutive numbers differ by 0.01. The value of a fixed-point number is essentially an integer that is scaled by an implicit specific factor determined by the position of the radix point. For example, the value 0.325 can be represented as 3250 in a fixed-point numeral system with scaling factor of 1/10 000 (10^{-4}), and the value 3,250,000 can be represented as 3250 with a scaling factor of 1000 (10^{+3}). So, these two numeral systems have different ranges, despite using the same number of decimal digits.

In several calculations, this type of equally separated numbers is not useful. An alternative is a floating-point numeral system, whose numbers have the following generic format:

$$\text{mantissa} \cdot \text{radix}^{\text{exponent}}$$

This format is useful for performing computations involving very large numbers ($V \gg 0$), numbers very close to 0 ($V \ll 1$), and more generally as an approximation to real arithmetic. A **floating-point number** can be seen as fixed-point number, indicated by the mantissa, whose radix point position is regulated by the exponent. The term floating-point is related to the fact that the radix point of a number can “float”, i.e., it can be placed anywhere relative to the significant digits of the number.

This format is used in the decimal scientific notation, in which numbers are expressed in two parts: a mantissa and an exponential part that indicates an integer power of ten. The value of the number is given by the product of these two parts. So, to express 22,538 in the scientific notation, one writes $2.2538 \cdot 10^4$. Scientific notation simplifies any arithmetic calculation that deals with very large or very small numbers. It serves also as the basis for floating-point computation in digital computers.

If this format is adopted again with five decimal digits, using three of them for the mantissa and the other two for the exponent, then the smallest representable number is $0 \cdot 10^0$ and the largest is $999 \cdot 10^{99}$.[2] So, the range size in this case is 10^{102}, which is much larger than the fixed-point alternatives previously discussed. The larger range of the floating-point numeral system is obtained at the expense of the number of significant digits that affect the precision of the numbers. Within a given subrange (specified by a unique exponent), there are fewer floating-point numbers than fixed-point numbers. Consider, for example, the range 1000 to 1999 in the 5-digit decimal alternatives discussed here. In the case of integers, exactly 1000 numbers $(1000, 1001, 1002, \ldots 1998, 1999)$ are represented. With floating-point numbers, 100 numbers $(1000, 1010, 1020, \ldots, 1990)$ are covered, which are represented as $100 \cdot 10^1$, $101 \cdot 10^1$, $102 \cdot 10^1$, $\ldots$ $199 \cdot 10^1$. Since computers always operate with a fixed number of bits, in general, floating-point representations offer more range and less precision than fixed-point ones.

In digital computers, floating-point numbers use bits and consist of three parts: a sign bit, an exponent part (representing a power of 2), and a mantissa (or significand). For illustration purposes, this section considers a 16-bit floating-point numeral system, with a 5-bit exponent, a 10-bit fractional part of the mantissa (so, a 11-bit mantissa as explained next), and a sign bit. Since both the mantissa and the exponent are signed values, both positive and negative numbers, as well fractions are covered. As shown in Fig. 3.1, the sign bit (s) is followed by the exponent (e), and finally by the fractional part of the mantissa (f).

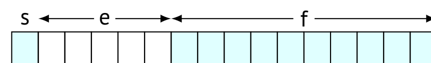


Fig. 3.1 Floating-point representation for 16 bits.

The value encoded by a bit pattern can be divided into three different cases, depending on the value of the exponent: normal numbers, subnormal numbers, and special values.

The value of a **normal number** (or normalised number) is calculated by the following equation, based on the three bit patterns (s, e, and f).

$$V = (-1)^s \times (1 + f) \times 2^{e-\text{bias}} \tag{3.6}$$

The **sign** bit s can obviously assume two possible values. If $s = 0$, then $(-1)^s = (-1)^0 = +1$, which implies that V is positive. If $s = 1$, then $(-1)^s = (-1)^1 = -1$, implying that V is negative. Notice that the other factors in Equation 3.6 are always positive.

The possible values for the exponents must include both positive numbers (to represent large numbers) and negative ones (to represent small numbers, i.e., near zero). The **exponent** e is encoded in an excess format. The bias value is a number near the middle of the range of possible values that is selected to represent zero. The bias typically equals $2^{k-1} - 1$, where k is the number of bits in the exponent.

[2] This format is not optimised in the sense that the mantissa is not normalised. Thus, some numbers have more than one representation. For instance, 2000 can be represented as $200 \cdot 10^1$, $20 \cdot 10^2$, and $2 \cdot 10^3$. It also does not include negative exponents. However, it serves for discussion purposes.

For the 16-bit floating-point numeral system, the bias is $2^{5-1}-1=15$, which is midway between 0 and 31. Any number larger than 15 in the exponent field represents a positive value for the real exponent. Values less than 15 indicate negative values. This is called an excess-15 representation, since one needs to subtract 15 to get the true value of the exponent.

For the numeral system here discussed, the value V of a normal number is given by Equation 3.7.

$$V=(-1)^s \times (1+f) \times 2^{\mathrm{e}-15} \tag{3.7}$$

This Equation only applies for exponents from 1 to 30. Exponents of all zeros ($00000_2 = 0_{10}$) or all ones ($11111_2 = 31_{10}$) are reserved for zero, subnormal numbers, or special values, like infinity, and are later described.

To avoid different possible representations of the same number, the **mantissa** $M = 1+f$ must be normalised. This normalisation requires the mantissa M to obey the restriction $1 \leq M < 2$.[3] This implies that the mantissa must always start with a nonzero bit. Since the only nonzero bit is '1', this constant bit is hidden, as its representation is totally useless. This bit is also the only one that is to the left of the binary point. So, the only part of the mantissa that needs to be represented in the bit pattern is its fractional part (f). Since there are 10 bits for this part, the mantissas are 11 bits long (they all have a '1' to the left of the binary point).

The next example shows how to represent the decimal number $+21.5_{10}$ in the 16-bit floating-point representation:

$$+21.5_{10} = +10101.1_2 = +10101.1_2 \cdot 2^0 = +1.01011_2 \cdot 2^4 = +1.0101100000_2 \cdot 2^{19-15}$$

Thus, $s=0$ (indicates non-negative numbers), $e=19_{10}=10011_2$, and $f=0101100000$ ($M=1.0101100000_2$). With these values, one can fill in the 16-bit pattern, as shown in Fig. 3.2.

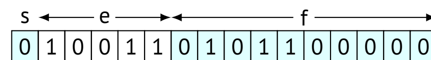


Fig. 3.2 16-bit floating-point representation for $+21.5_{10}$.

As previously indicated, Equation 3.7 does not apply for exponents 0 and 31. The all zeros (00000_2) exponent is reserved to represent subnormal numbers and zero. A **subnormal number** (or denormalised number) is a non-zero number with magnitude smaller than the smallest positive normal number. Its exponent value is fixed to be $1-bias$ (-14 in the example) and the mantissa M is restricted by the condition $0 \leq M < 1$ (there is no leading 1). So, for the all-zeros e exponent, the value V of a subnormal number is given by the following equation:

$$V=(-1)^s \times f \times 2^{-14} \tag{3.8}$$

[3] In the decimal scientific notation, the values are also normalised, but in this case the mantissa M is required to be $1 \leq M < 10$. One can generalise by stating that M must satisfy the condition $1 \leq M < r$, being r the base.

Since f can be also all-zeros (0000000000_2), this allows V to be 0. This is very convenient since the zero value, as shown in Fig. 3.3, is represented by an all-zeros bit pattern.[4] Zero cannot be represented by a normal number, since $M \geq 1$ and the other two factors in Equation 3.7 are never equal to zero. In fact, zero is considered a special value in a floating-point numeral system.

An **underflow** occurs when the absolute value of a number is smaller (that is, closer to zero) than the smallest absolute value representable in the floating-point numeral system. Underflow can theoretically be seen as negative overflow of the exponent of the floating-point number. For example, if the exponent part can represent values from -128 to $+127$, then a value with an exponent like -150 may cause underflow. In that case, the number can be represented as zero, which is the closest to the number under consideration.

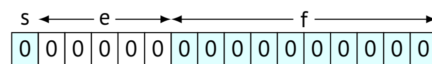


Fig. 3.3 16-bit floating-point representation for +0.

The all ones ($11111_2 = 31_{10}$) exponent is reserved for other special values. A **special value** can be used to indicate an error or to represent non-initialised data. If the fraction field f is equal to all zeros, the value represents infinity ($V = -\infty$, if $s = 1$; $V = +\infty$, if $s = 0$). Infinity can represent results that overflow (e.g., when multiplying two very large numbers or dividing by zero). When f is nonzero, the resulting value is a **not a number (NaN)**. Such value can be the result of an operation that cannot be given as a real number or as infinity, e.g., $\sqrt{-1}$.

Table 3.4 summarises the specific purposes for the largest and the smallest exponents in a floating-point numeral system, in function of the value of the mantissa.

exponent	M = 0	M ≠ 0
$000\ldots00$	$V = \pm 0$	subnormal
$111\ldots11$	$V = \pm\infty$	NaN

Table 3.4 The specific purposes for the largest and smallest exponents in a floating-point numeral system.

Fig. 3.4 shows for positive numbers how they are overspread, by considering for illustration purposes the six smallest exponents (00000_2 to 00101_2). With the exception of exponent 00000_2 (used for subnormal numbers), one can observe that the interval associated with a given exponent is larger when the exponent is bigger. In particular, the interval doubles if the exponent is incremented by one unit. For example, for exponents 00100_2 and 00101_2, the distance between two consecutive numbers is 2^{-21} and 2^{-20}, respectively. Since there are ten bits for the mantissa, the number of numbers represented in each interval is exactly

[4] In fact, there are two representations of zero, depending on the value of the sign bit. So, 0000000000000000_2 represents +0 and 1000000000000000_2 represents -0.

1024 and they are all equally separated. This means that the distance between two consecutive numbers is larger for larger exponents. Table 3.5 shows some important values for each exponent.

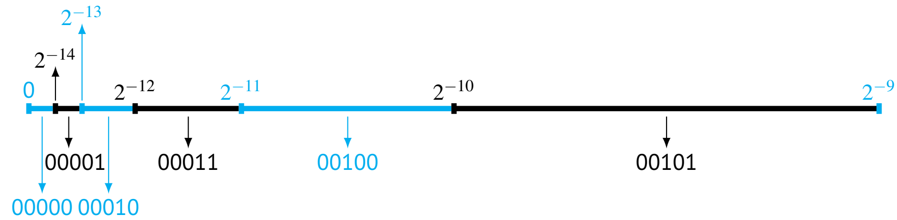


Fig. 3.4 The distribution of 16-bit floating-point numbers for six exponents (00000_2 to 00101_2), indicated below the line.

exponent	interval	smallest value	2nd smallest value	largest value	distance
$00000_2 = 0$	2^{-14}	0	2^{-24}	$1023 \cdot 2^{-24}$	2^{-24}
$00001_2 = 1$	2^{-14}	2^{-14}	$(1+2^{-10}) \cdot 2^{-14}$	$(1+1023 \cdot 2^{-10}) \cdot 2^{-14}$	2^{-24}
$00001_2 = 1$	2^{-13}	2^{-13}	$(1+2^{-10}) \cdot 2^{-13}$	$(1+1023 \cdot 2^{-10}) \cdot 2^{-13}$	2^{-23}
$00011_2 = 3$	2^{-12}	2^{-12}	$(1+2^{-10}) \cdot 2^{-12}$	$(1+1023 \cdot 2^{-10}) \cdot 2^{-12}$	2^{-22}
$00100_2 = 4$	2^{-11}	2^{-11}	$(1+2^{-10}) \cdot 2^{-11}$	$(1+1023 \cdot 2^{-10}) \cdot 2^{-11}$	2^{-21}
$00101_2 = 5$	2^{-10}	2^{-10}	$(1+2^{-10}) \cdot 2^{-10}$	$(1+1023 \cdot 2^{-10}) \cdot 2^{-10}$	2^{-20}
			...		
$11100_2 = 28$	2^{+13}	2^{+13}	$(1+2^{-10}) \cdot 2^{+13}$	$(1+1023 \cdot 2^{-10}) \cdot 2^{+13}$	2^{+3}
$11101_2 = 29$	2^{+14}	2^{+14}	$(1+2^{-10}) \cdot 2^{+14}$	$(1+1023 \cdot 2^{-10}) \cdot 2^{+14}$	2^{+4}
$11110_2 = 30$	2^{+15}	2^{+15}	$(1+2^{-10}) \cdot 2^{+15}$	$(1+1023 \cdot 2^{-10}) \cdot 2^{+15}$	2^{+5}

Table 3.5 16-bit floating-point numbers for a selection of exponents. The 'distance' column indicates the difference between two consecutive numbers for the respective exponent.

The 16-bit floating-point numeral system used in this section was introduced for simplicity and conceptual understanding. It corresponds to the half precision form that is part of the **IEEE 745 floating-point standard**, originally published by the Institute of Electrical and Electronic Engineers (IEEE) in 1985. This standard is also used for both single- and double-precision floating-point numbers. These two formats correspond to the C `float` and `double` datatypes, respectively. This standard was superseded in 2008 by IEEE 754-2008, and again in 2019 by minor revision IEEE 754-2019.

The IEEE-754 single-precision standard, represented in Fig. 3.5(a), uses an excess 127 bias over an 8-bit exponent. The mantissa is 23 bits. With the sign bit included, the total word size is 32 bits. Double-precision numbers use a signed 64-bit word consisting of an 11-bit exponent and 52-bit mantissa, as illustrated in Fig. 3.5(b). The bias is 1023. The ranges of numbers that can be represented in the IEEE single- and double-precision models are shown in Table 3.6.

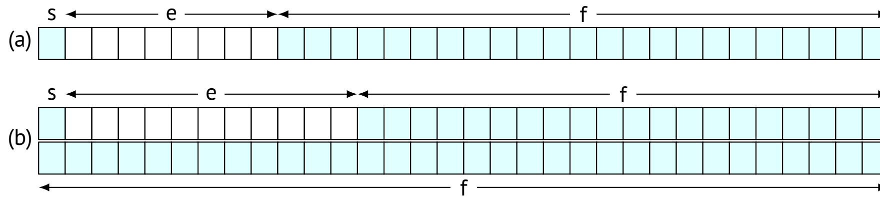


Fig. 3.5 IEEE 754 floating-point standard representation: (a) single-precision, (b) double-precision.

description	e	f	single precision value	single precision decimal	double precision value	double precision decimal
zero	00..00	0..00	0	0.0	0	0.0
smallest subnorm.	00..00	0..01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
largest subnorm.	00..00	1..11	$(1-\varepsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\varepsilon) \times 2^{-1022}$	2.2×10^{-308}
smallest norm.	00..01	0..00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
one	01..11	0..00	1×2^{0}	1.0	1×2^{0}	1.0
largest norm.	11..10	0..00	$(2-\varepsilon) \times 2^{127}$	3.4×10^{38}	$(2-\varepsilon) \times 2^{1023}$	1.8×10^{308}

Table 3.6 Representations and numeric values of some important single- and double-precision floating-point numbers.

3.7 Binary codes for decimal numbers

As already discussed, computers use binary patterns to represent numbers, which contrasts with humans that do prefer to use decimal digits. As such, some binary codes exist to facilitate the process of representing decimal numbers. The representations for decimal numbers do not modify the essential nature of digital circuits; they still handle signals with only two possible values (0 and 1). The idea is to represent a decimal number with a string of bits, where each group of contiguous bits represent one of its decimal digits. For example, if a 4-bit string is used to represent the decimal digits, one can assign '0000' to decimal digit 0, '0001' to decimal digit 1, and so on. So, a sequence of 4-bit strings represents a decimal number. For example, the sequence '010100001001' represents the decimal number 509.

There are $29\,059\,430\,400$ (or $\frac{16!}{6!}$) different codes to represent the 10 decimal digits with 4 bits (i.e., with 16 different bit patterns). The most obvious decimal code is called **binary-coded decimal (BCD)** and encodes the decimal digits 0 to 9 by the respective 4-bit unsigned binary representations 0000 through 1001. In this case, the code patterns 1010 through 1111 are not used. This code allows one byte to store two decimal digits in a packed BCD representation. For example, number 93 is represented by 10010011, with the first nibble (1001) representing decimal digit 9 and the last nibble (0011) representing decimal digit 3.

Table 3.2 shows some common decimal codes. The 2421 code is a weighted code, like BCD that uses the weights 8421. The 2421 code has the advantage of being self complemented, that is, the code word for the 9's-complement of any digit can be obtained by swapping its bits. For example, the 9's-complement of 6 (1100) is 3 (0011).

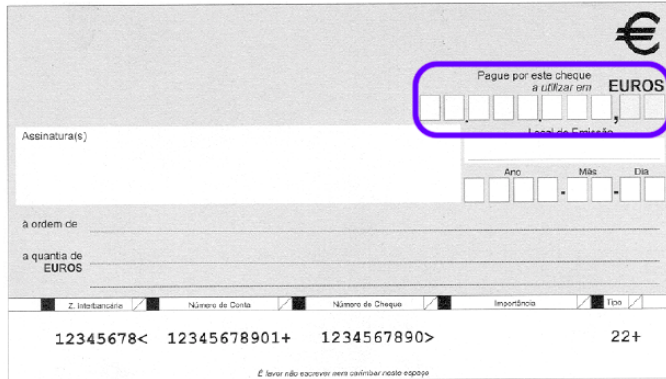
digit	BCD	2421	2-out-of-5	biquinary	1-out-of-10
0	0000	0000	01100	0100001	1000000000
1	0001	0001	11000	0100010	0100000000
2	0010	0010	10100	0100100	0010000000
3	0011	0011	10010	0101000	0001000000
4	0100	0100	01010	0110000	0000100000
5	0101	1011	00110	0100001	0000010000
6	0110	1100	10001	1000010	0000001000
7	0111	1101	01001	1000100	0000000100
8	1000	1110	00101	1001000	0000000010
9	1001	1111	00011	1010000	0000000001
Unused bit patterns					
	1010	0101	00000	0000000	0000000000
	1011	0110	00001	0000001	0000000011
	1100	0111	00010	0000010	0000000101
	1101	1000	00100	0000011	0000000110
	1110	1001	00111	0000100	0000000111
	1111	1010	...	...	...

Table 3.7 Some popular decimal codes.

Decimal digits can be represented with more than four bits, which is the case for 2-out-of-5 representation. Each decimal digit is represented by a 5-bit pattern, in which two bits are set to '1' and the other three to '0'. This type of representation is widely used by barcodes and in magnetic cards. Another example is the biquinary code. The first two bits indicated whether the digit is in the range 0–4 or 5–9, while the last five indicate which digit is represented. The use of more bits than strictly needed allows errors to be more likely detected. If, for example, one bit accidentally changes due, for example, to a transmission error, the resulting bit pattern does not represent a valid digit. In the biquinary code, only 10 combinations are valid. The remaining 118 can be flagged as errors if they appear. The 1-out-of-10 is another example of a code that uses more bits than the minimum. In this case only 10 out of 1024 possible code words are valid. Only the patterns with one (and only one) bit equal to '1' are valid.

Exercises

Exerc. 3.1: As the figure shows, bank cheques in Portugal have 10 boxes to indicate the amount to be paid. What are the minimum and the maximum values that can be written in a bank cheque?


€
Pague por este cheque
a utilizar em EUROS
Assinatura(s)
Local de Emissão
Ano Mês Dia
à ordem de
a quantia de
EUROS
Z. Interbancária Número de Conta Número de Cheque Importância Tipo
12345678< 12345678901+ 1234567890> 22+
É favor não escrever nem carimbar neste espaço

Exerc. 3.2: Represent the following decimal numbers as binary numbers: **(a)** 131; **(b)** 511; **(c)** 888; **(d)** 4096.
Exerc. 3.3: What is the largest natural number that can be represented with **(a)** 5, **(b)** 10, **(c)** 18, and **(d)** 32 bits?
Exerc. 3.4: List all the digits and their binary representation in base 13.
Exerc. 3.5: Convert the following binary numbers to hexadecimal: **(a)** 101111101101; **(b)** 1001 110110; **(c)** 11111111111; **(d)** 10100011110.
Exerc. 3.6: Convert the following hexadecimal numbers to binary: **(a)** BEEF; **(b)** 1000.FF; **(c)** ABC.DEF; **(d)** DAC.34.
Exerc. 3.7: Convert the following decimal numbers to base 5: **(a)** 77; **(b)** 131; **(c)** 511; **(d)** 1000.
Exerc. 3.8: Convert the following base-9 numbers to decimal: **(a)** 66; **(b)** 123; **(c)** 317; **(d)** 800.
Exerc. 3.9: Convert the following numbers from the given base to the indicated bases: **(a)** 66_{10} to bases 2, 7 and 9; **(b)** $13F.4_{16}$ to bases 10 and 12; **(c)** 1110010.1_2 to bases 3, 4, and 7; **(d)** $AB7_{13}$ to bases 2, 6, and 8.
Exerc. 3.10: A given computer is equipped with 1,073,741,824 bytes of memory. Why was this odd number chosen?
Exerc. 3.11: The St. Galen train station, in Switzerland, is equipped with a binary electronic clock to indicate the time of day (hours, minutes, seconds) on three lines. At what time the photograph was taken?

("Binary clock Swiss railway station" by BBCLCD is licensed under CC-BY-SA-4.0)

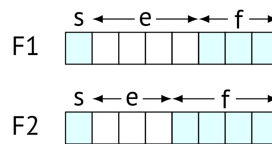
Exerc. 3.12: A 32-bit signed integer on a little-endian computer contains the numerical value of 3. If it is transmitted to a big-endian computer byte by byte and stored there, with byte 0 in byte 0, byte 1 in byte 1, and so on, what is its numerical value on the big-endian machine if read as a 32-bit unsigned integer?
Exerc. 3.13: As of 2018, Iceland had about 23,000 registered footballers (male and female). Calculate the minimum number of bits that allows representing this value with an integer encoded in signal and amplitude.
Exerc. 3.14: Monaco had, in 2013, 37,831 inhabitants. Calculate the minimum number of bits that are required to encode this value as a signed integer. What is the answer if the value is encoded as an unsigned integer?
Exerc. 3.15: A given company has 19 employees, who are paid every two weeks. It is necessary to register the number of half hours that each employee worked in each workday (Monday to Friday). For health reasons, the law does not permit an employee to work more than 12h in a day. Indicate the minimum number of bits needed to represent this information for two weeks.
Exerc. 3.16: An European institute aims to assign a code to each of its members. To this end, it was decided to use the format AA / HHHHH-BB, with A being a capital letter, H being a hexadecimal digit and B being a base 2 digit. The two letters indicate which of the 51 affiliated countries the member belongs to (e.g., BE for Belgium, PO for Portugal, LX for Luxembourg). The binary digits are used to encode the type of membership of the member with the Association (00: junior member, 01: regular member, 10: senior member). Indicate, for a given country, the maximum number of members that this code allows to register.
Exerc. 3.17: In 2014, the Spy's Gangnam Style video reached 2,147,483,648 views on YouTube. However, the number presented was negative. Explain why this happened and suggest the simplest solution to overcome it. To solve this issue, YouTube made an internal change that now allows the counters to go up to 9,223,372,036,854,775,807. Describe what approach was followed by the YouTube engineers.
Exerc. 3.18: Find the value of X, so that: **(a)** $23_X = 10101_2$; **(b)** $4X_7 = 35_9$.
Exerc. 3.19: Add the following natural numbers: **(a)** $110011_2 + 10101_2$; **(b)** $129\text{B}_{12} + 239_{12}$; **(c)** $\text{CBA}_{16} + 987_{16}$.
Exerc. 3.20: Indicate the ten's-complement of the following decimal numbers: **(a)** 1236; **(b)** $90\,037$; **(c)** $111\,122$.
Exerc. 3.21: Indicate the two's-complement of the following binary numbers: **(a)** 0011100_2; **(b)** 1100110011_2; **(c)** 00000001_2; **(d)** 11100000001_2.
Exerc. 3.22: Write the 8-bit sign-magnitude, one's-complement, two's-complement representations for decimal numbers: **(a)** $+18$; **(b)** $+121$; **(c)** -33; **(d)** -100.
Exerc. 3.23: Calculate the value of the 10-bit binary number $10110\ 00111_2$ in the following representations: **(a)** sign-magnitude; **(b)** one's-complement; **(c)** two's-complement; **(d)** excess-511.
Exerc. 3.24: Represent the number -233 in the following 10-bit representations: **(a)** sign-magnitude; **(b)** one's-complement; **(c)** two's-complement; **(d)** excess-511.
Exerc. 3.25: Perform binary subtraction by taking the two's-complement of the subtrahend: **(a)** $100110_2 - 111_2$; **(b)** $100110_2 - 10000_2$; **(c)** $1010101_2 - 11_2$; **(d)** $1000001_2 - 1000000_2$.
Exerc. 3.26: Add the following pairs of unsigned binary numbers, explicitly indicating the carries:

(a)	**(b)**	**(c)**	**(d)**
11010	111010	1001111010	1101011
+ 1010	+ 101010	+ 1011010	+ 1011000

Exerc. 3.27: Add the following pairs of 8-bit two's-complement numbers, explicitly indicating situations of overflow:

(a)	**(b)**	**(c)**	**(d)**
1001 1010	0111 1010	1101 1101	0110 1011
+ 1000 1010	+ 0110 1010	+ 1110 1101	+ 0101 1000

Exerc. 3.28: Consider two floating-point formats F1 and F2, with 8 bits, based on all the principles presented in Section 3.6, namely normal numbers, subnormal numbers, special values, etc.



(a) Indicate the mathematical expressions that can be used to calculate the normal numbers in both formats.
(b) For each format, indicate the bit patterns and the respective decimal value for i) the smallest positive subnormal number, ii) the largest subnormal number, iii) the smallest positive normal number, iv) one, and the v) largest normal number.
(c) Calculate the decimal values of the following bit patterns for the F1 format: i) 10110011, ii) 01111010, iii) 10010001, iv) 00000011, v) 11000001.
(d) Represent in the F1 format, the following values: i) -111.01_3, ii) 128_{10}, iii) 111.01_{10}, iv) -18C_{16}, v) 0.005_8.
(e) Convert the numbers represented in the F1 format into the F2 format: i) 00110011, ii) 11101001, iii) 00010000, iv) 11001110, v) 10000010. Overflow must be represented by $\pm\infty$, underflow by ± 0 and the roundings must be made to the closest value.
Exerc. 3.29: Write a C program that calculates the decimal value for a bit pattern that represents a floating-point number. The inputs, provided through the command line, are: the bit pattern (sequence of non-separated 0s and 1s), the number of bits of the exponent e, and the number of bits of the mantissa f. If no pattern is provided, the program lists a pair (bit pattern, decimal value) for all possible 2^{1+e+f} bit patterns.

Further reading

Precise and detailed analysis of the evolution of numeral systems and computer arithmetic is available in the seminal work by Knuth (1997). The book by Kneusel (2017) is related to numbers and how they are represented in and operated on by computers. It deals with standard representations of integers and floating-point numbers, and presents also several other number representations that are useful in specific contexts. Another interesting material about how to represent data, most particularly numbers, in computers is provided by Fenwick (2015). An relevant reference for floating-point numbers is the survey by Goldberg (1991).

Chapter 4
IA32 instruction-set architecture

Abstract Programming in a high-level language tends to be the preferred choice for programmers, since the abstraction level is higher than the one provided by machine-level languages. With the level of sophistication of the current compilers, the generated code is usually at least as good as the one an experienced assembly-language programmer can manually write. Anyhow, the ability to read and understand assembly code is important for professional programmers. This chapter presents the details of a particular assembly language (IA32) and discusses how high-level programs (written in C) get compiled into this form of machine code. Reading the assembly code generated by a compiler involves a set of skills that differ from the ones that are required to write assembly code by hand, since one must understand the transformations compilers usually make in converting the constructs/statements into machine code. Comprehending assembly code and how it relates to the original high-level code is the approach followed in this chapter to allow the reader to understand how computers execute programs, i.e., how they work.

4.1 Compilation of C code to assembly code

Programming in a high-level language, such as C, is productive and reliable, since the abstraction level is higher than the one provided by machine-level languages. When writing programs in low-level code, a programmer must specify how the program manages memory and which low-level instructions the program will execute. The level of sophistication of the current compilers makes programming in assembly-language almost unneeded. Indeed, it is very difficult for a skilled assembly-language programmer to identify parts of the assembly code generated by a compiler that can be optimised. Additionally, a program written in a high-level language is portable, i.e., it can be compiled and executed on a number of different machines, while assembly code is specific to a reduced set of machines. Anyhow, the ability to read and understand assembly code is a relevant skill for professional programmers.

By invoking the compiler with appropriate flags, it generates a file showing its output in assembly code. Assembly code is very close to the actual ISA-level code that computers

execute. Its main feature is that it is in a more readable textual format, compared to the binary format of object code. By reading this assembly code, the optimisation capabilities of the compiler and the underlying limitations of the generated code can be analysed.

The ISA level has a special importance for system architects: it is the interface between the software and the hardware. While it is theoretically possible to have the hardware directly executing programs written in C, C++, Java, or some other high-level programming languages, this does not seem to be a brilliant idea. As depicted in Fig. 4.1, the common approach that computer designers take is to have programs in several high-level languages compiled to the ISA level and to construct microprogrammed hardware that can execute ISA-level programs directly. The ISA level is thus the common intermediate form that establishes the interface between the compilers and the hardware. It constitutes the language that both have to understand.

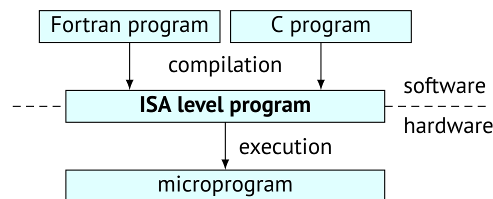


Fig. 4.1 The ISA level as the interface between software and hardware.

In order to run a C program with a computer, the individual C statements must be translated by other programs into a sequence of low-level machine-language instructions. These instructions are packed in a form called an executable object program and stored as a binary file that is saved in a disk.

In the command line, the translation from source file `prog.c` to its corresponding object file can be accomplished by invoking the `gcc` compiler:

```
> gcc prog.c
```

In fact, as Fig. 4.2 shows, four different programs are used to compile a C program: preprocessor, compiler, assembler, and linker.

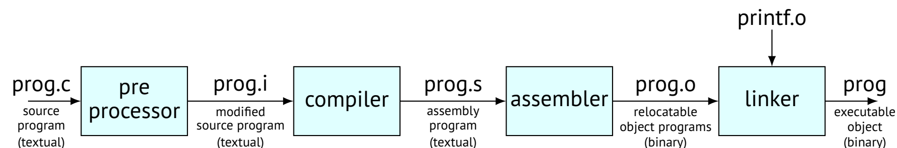


Fig. 4.2 The compilation system.

In the **preprocessing phase**, the original C program is modified according to directives that begin with the # character. For example, if an `#include <stdio.h>` directive is

in the C program being compiled, the preprocessor reads the contents of the header file `stdio.h` and inserts it directly into the program text. The same happens with `#define` directives that serve to define constants. If the program contains the `#define MAX 30` directive, all instances of the string "MAX" are replaced by the string "30." The preprocessor produces another C program, typically with the .i suffix.

The `gcc` compiler can be instructed to output the intermediate files that are produced, if the appropriate options are used. The file generated by the preprocessor can be obtained by invoking the `gcc` compiler as follows:

```
> gcc -E prog.c -o prog.i
```

During the **compilation phase**, the compiler translates the text file `prog.i` into the text file `prog.s`, which contains an equivalent assembly-language program. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. The assembly language is useful, since it constitutes a common language for compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language, as shown in Fig. 4.1.

The `gcc` compiler can also generate a file with the assembly code, as next indicated:

```
> gcc -S prog.c -o prog.s
```

In the **assembly phase**, the assembler translates `prog.s` into machine-language instructions, packages them in a format designated as a relocatable object program, and stores the result in the object file `prog.o`. This file is a binary file, whose bytes encode machine language instructions rather than characters. This file should not be open with a text editor, since it does not contain textual information, but rather binary codes.

To obtain the object file, `gcc` needs to be called with the following options:

```
> gcc -c prog.c -o prog.o
```

Usually, programs include other modules, like standard libraries. If the `prog.c` program calls, for instance, the `printf` function, it needs to include the respective standard C library provided by every C compiler (`stdio`). The `printf` function resides in a separate precompiled object file called `printf.o`, which must be merged, i.e., linked, with the `prog.o` file. So, during the **linking phase**, it is the responsibility of the linker to handle this merge. The result is the `prog` file, which is an executable.

To run the `prog` executable, the loader is invoked. The **loader** is part of the operating system and is responsible for loading programs and libraries. It loads programs into memory and prepares them for execution. Loading a program involves copying to the main memory the machine-level instructions that are saved in its executable file. Once loading is complete, the operating system passes the control to the loaded program code.

4.2 IA32 assembly language

IA-32, the acronym for "Intel Architecture, 32-bit", is the 32-bit version of the x86 instruction set architecture, developed by Intel and first implemented in the 80386 microprocessor. This

section provides a non-exhaustive description of the IA32 assembly language, that is, an overview of the must important aspects of this machine-oriented language. These aspects include the registers, the data types, the operands for the instructions, different groups of instructions (data movement, arithmetic, logical, control), the way procedures are handled, and how data structures are represented in IA32.

4.2.1 Registers

IA32 CPUs contain eight registers, each one storing 32-bit values (Fig. 4.3). These registers are used to store numeric values as well as pointers. For the most cases, the first six registers can be considered general-purpose ones, because there are no restrictions on their use. This general nature is just broken by some instructions that implicitly use some specific registers as sources and/or destinations. All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The two low-order bytes of the first four registers can also be accessed independently. The last two registers (ebp and esp) contain pointers to important places in the stack. They should only be altered according to the set of standard conventions for stack management.

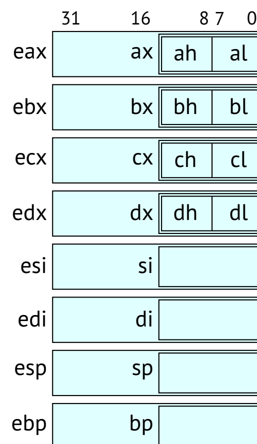


Fig. 4.3 The eight IA32 32-bit registers.

As indicated in Fig. 4.3, the low-order two bytes of the first four registers can be independently manipulated by byte-oriented instructions. This feature was provided in the 8086, a 16-bit microprocessor, to allow backward compatibility to the 8008 and 8080, two 8-bit microprocessors from the 1970 decade. Whenever a byte instruction updates one of these single-byte registers, the remaining three bytes of the full register remain unchanged. Similarly, the low-order 16 bits of each register can be read or written by word-oriented in-

structions. This characteristic results from the fact that IA32 inherits many features of all its predecessors, namely 16-bit microprocessors.

Another important register is the instruction pointer, referred to as eip in IA32. As already indicated, its purpose is to indicate the address in memory of the next instruction to be executed.

4.2.2 Data types

Due to IA32 being an expansion from a 16-bit architecture, Intel uses the term "word" to refer to a 16-bit data type. Based on this, 32-bit quantities are referred to as "double words."

Table 4.1 shows the machine representations used for the primitive data-types of C. Note that most of the common datatypes are stored as double words. This includes both regular and long integers (types `int` and `long`), whether or not they are signed. Additionally, all pointers (shown here as char *) are stored as 4-byte double words. Bytes are commonly used when manipulating string data (i.e., ASCII characters). As the table indicates, every operation has a single-character suffix denoting the size of the operand. For example, the `MOV` (move data) instruction has three variants: `MOVB` (move byte), `MOVW` (move word), and `MOVL` (move long/double word).

The assembly code discussed here follows the syntax followed by the GNU assembler (known as GAS). There exist some smaller differences with the assembler provided by Intel. Anyhow, it should not be difficult to spot those differences which are usually small.

C datatype	Intel datatype	GAS suffix	size (bytes)
`char`	Byte	b	1
`short`	Word	w	2
`int`	Double word	l	4
`unsigned`	Double word	l	4
`long int`	Double word	l	4
`unsigned long`	Double word	l	4
`char *`	Double word	l	4

Table 4.1 Sizes of standard datatypes.

4.2.3 Operands

Many instructions have operands, which are used to specify the source values for performing the operation and the destination location into which to store the result. IA32 supports a different number of operand forms, as indicated in Table 4.2, where Imm represents a constant, R_x the contents of a register, s a scale (1, 2, 4 or 8), and $M[addr]$ the contents of the memory in the location whose address is given by $addr$.

type	form	operand value	name
immediate	$Imm	Imm	constant
register	%R$_a$	R_a	register
memory	Imm	$M[Imm]$	absolute
	(%R$_b$)	$M[R_b]$	indirect
	Imm(%R$_b$)	$M[R_b + Imm]$	base+offset
	(%R$_b$,%R$_i$)	$M[R_b + R_i]$	indexed
	Imm(%R$_b$,%R$_i$)	$M[R_b + R_i + Imm]$	indexed
	(,%R$_i$,s)	$M[R_i \cdot s]$	scale indexed
	Imm(,%R$_i$,s)	$M[R_i \cdot s + Imm]$	scale indexed
	(%R$_b$,%R$_i$,s)	$M[R_b + R_i \cdot s]$	scale indexed
	Imm(%R$_b$,%R$_i$,s)	$M[R_b + R_i \cdot s + Imm]$	scale indexed

Table 4.2 Operands forms.

Basically, there are three major forms for the operands:

- **Immediate** is used to constant values, which can only be used as sources. Constants are written with a '$' followed by an integer using standard C notation, such as, `$-577` or `$0x1F`.
- **Register** is used to denote the contents of one of the registers, either with 32 bits (e.g., eax), 16 bits (e.g., ax), or 8 bits (e.g., ah). A register is specified with a leading '%' followed by its name. The notation R_a identifies register a and represents its contents.
- **Memory** is used to memory locations, according to an address that is calculated. Since the memory is viewed as a large array of bytes, the notation $M[addr]$ is used to note a reference to the value stored in memory starting at address $addr$. There are exactly nine different addressing modes to access memory. The most generic form is the last one in Table 4.2: Imm(R$_b$,R$_i$,s). It has four components: an immediate offset (Imm), a base register (R$_b$), an index register (R$_i$), and a scale factor (s), whose value must be 1, 2, 4, or 8. The effective address is then computed as $R_b + R_i \cdot s + Imm$. This general form facilitates the access to array elements. The other forms are simply special cases of this generic form, where some of the components are omitted. The omitted elements are assumed to have a zero value, but s that has 1 has its default value.

Generically, source values can be given as constants or read from registers or memory. Destination values can be written in either registers or memory.

4.2.4 Data movement instructions

Moving data is a fundamental operation in a computer. The term "move" must be seen as a synonym of "copy," since the values in the source remain there unchanged. Table 4.3 shows some IA32 instructions to move data, where s and d represent source and destination operands. Since IA32 has a maximum of two operands per instruction, in many cases, the destination operand also acts a source operand.

instruction	effect	description
MOV s, d	$d \leftarrow s$	move
movb		move byte
movw		move word
movl		move double-word
MOVZ s, d	$d \leftarrow zeroExt(s)$	zero-extending move
movzbw		move zero-extended byte to word
movzbl		move zero-extended byte to double-word
movzwl		move zero-extended word to long
MOVS s, d	$d \leftarrow signExt(s)$	sign-extending move
movsbw		move sign-extended byte to word
movsbl		move sign-extended byte to double-word
movswl		move sign-extended word to long
PUSH s	$esp \leftarrow esp - size(s)$	push
pushl	$M[esp] \leftarrow s$	push double-word
pushw		push word
POP d	$d \leftarrow M[esp]$	pop
popl	$esp \leftarrow esp + size(s)$	pop double-word
popw		pop word

Table 4.3 IA32 instructions for moving data.

Many IA32 instructions have three variants: one for bytes, one for words, and a third one for double-words. This occurs for the MOV instruction, whose variants (movb, movw, and movl) are differentiated by the last character (b, w, or l). The source operand s designates a value that is immediate, stored in a register, or stored in memory. The destination operand d designates a location that is either a register or a memory address. It is not possible for a MOV instruction to have both operands refer to memory locations. Thus, copying a value from one memory location to another one can be achieved with two instructions: the first one loads the source value into a register, and the second writes the register value to the destination. The next examples illustrate the five possible combinations of sources and destinations, with the movl instructions that handles operands with 32 bits:

```
movl $0x40, %eax           Immediate–Register
movl %ebp, %esp            Register–Register
movl (%edi,%ecx), %edx     Memory–Register
movl $-15, (%esp,%eacx)    Immediate–Memory
movl %ebx, -12(%ebp)       Register–Memory
```

To move data with the size of one or two bytes, IA32 provides the instructions movb and movw, respectively. When the operands are registers, they should have the adequate sizes.

```
movb $0x40, %bl            Immediate 8 bits
movw $0x40, %ax            Immediate 16 bits
movb (%edi,%ecx), %dh      Memory–Register
movw $-15, (%esp,%eacx)    Immediate–Memory
movw %bx, -12(%ebp)        Register–Memory
```

The suffixes b, w and l are not strictly necessary, when registers are involved, since they have well-know sizes. This does not occur for Immediate-Memory moves, since the sizes may not be evident.

The MOVZ and MOVS instructions both read the contents of the register or effective address as a word or byte and then extends to word or double-word. MOVZ adds zero's to the MSBs of the destination. It serves to convert an unsigned integer to a wider unsigned integer. MOVS extends the sign bit of the source to the MSBs of the destination, so it can be used to convert a signed integer to a wider signed integer. In both cases, the result is stored in the destination register.

Consider that al=00_{16}, bl=81_{16} and cx=$FFFF_{16}$. Note that the MSBs of al, bl, and cx are 0, 1, and 1, respectively. The next examples show, in the right part of each line, the values that are stored in the destination registers when the corresponding instructions are executed:

`movzbw %al, %dx`	dx $= 0000_{16}$
`movsbw %al, %dx`	dx $= 0000_{16}$
`movzbw %bl, %dx`	dx $= 0081_{16}$
`movsbw %bl, %dx`	dx $= FF81_{16}$
`movzbl %bl, %edx`	edx $= 00000081_{16}$
`movsbl %bl, %edx`	edx $= FFFFFF81_{16}$
`movzwl %cx, %edx`	edx $= 0000FFFF_{16}$
`movswl %cx, %edx`	edx $= FFFFFFFF_{16}$

The IA32 programming model also provides a **stack**, where temporary data can be stored. This stack is stored in some region of the main memory. It is accessed through the instructions pushl and popl.[1] These two instructions manipulate a single four-byte operand, either the data source for pushing or the data destination for popping. The stack grows downward such that the top element of the stack has the lowest address of all stack elements. Stacks are illustrated with the stack "top" shown at the top of the figures. The stack pointer esp holds the memory address of the top stack element.

Popping a double word includes reading from the top of the stack location and then incrementing the stack pointer by four. Thus, the instruction popl %ecx is functionally equivalent to the following pair of instructions:

```
movl (%esp), %cx
addl $4, %esp
```

Pushing a double-word value onto the stack involves first decrementing the stack pointer by four and then writing the value at the address of the new top of the stack. Therefore, the behaviour of the instruction pushl %edx is exactly the same as the one produced by the following instructions:

```
subl $4, %esp
movl %edx, (%esp)
```

Fig. 4.4 illustrates how the stack works. This example assumes that initially the contents of the stack and the relevant registers are as indicated in Fig. 4.4(a). When the instruction popl %ecx is executed, the contents of the memory in the address given by register esp (2108_{16}) is copied to register ecx. Additionally, the value in register esp is incremented by four, yielding $210C_{16}$. As indicated previously, information in processors is not moved but

[1] IA32 also provides instructions to use the stack with words (pushw and popw), but they are not discussed in this book. The stack, being in the memory, can also be accessed with other instructions, like MOV instructions.

rather copied, so the value 1234_{16} remains at memory location 2108_{16}, until it is eventually overwritten by another push operation. The top of the stack however is always considered to be the address indicated by esp. The result of the execution of this `popl` instruction is shown in Fig. 4.4(b).

Now, the instruction to be executed is `pushl %edx`. Firstly, the value in register esp is decremented by four, yielding 2108_{16}. The contents of register edx is copied to the memory in the address given by esp. Again, the move operation does not remove or delete the value that is stored in register edx; it was just copied to the destination. Fig. 4.4(c) shows the result of executing this `pushl` instruction.

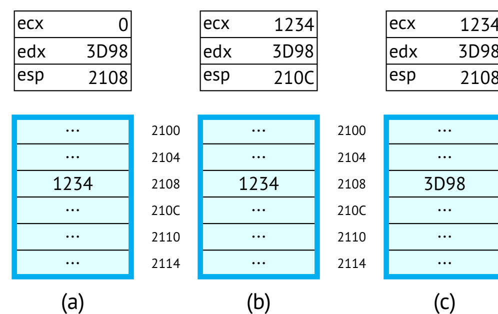


Fig. 4.4 A stack: (a) initial state; (b) state after executing `popl %ecx`; (c) state after executing `pushl %edx`. The values in the registers and that indicate memory addresses are represented in hexadecimal.

The fact that the stack is contained in the memory allows programs to access arbitrary positions within the stack, using the standard memory addressing mechanisms. For example, the instruction `movl 8(%esp),%ebx` copies to register ebx the third double word from the top of the stack.

4.2.5 Arithmetic and logical instructions

A computer must obviously be able to perform arithmetical and logical operations. Table 4.4 lists the most used IA32 instructions of this type.

The `LEAL` instruction has the same form of a memory-register `MOV` instruction, but it does not reference memory at all. The first operand, i.e., the source, indicates a memory reference, but instead of reading from the designated location, the instruction copies its effective address to the destination. This computation is indicated in Table 4.4 with the address() function (equivalent to the address/pointer operator & in the C language).

The single-operand form of the `IMUL` instruction executes a signed multiply of a byte, word, or double-word by the contents of the al, ax, or eax registers and stores the product in the ax, dx:ax or edx:eax registers, respectively. The two-operand form of `IMUL` executes a signed multiply of a register or memory word or double-word by a register word or double-word and stores the product in that register word or long word. There is also

instruction	effect	description
`LEAL s, d`	$d \leftarrow address(s)$	load effective address
`ADD s, d`	$d \leftarrow d + s$	add
`addb`		add bytes
`addw`		add words
`addl`		add double-words
`INC d`	$d \leftarrow d + 1$	increment
`incb`		increment byte
`incw`		increment word
`incl`		increment double-word
`SUB s, d`	$d \leftarrow d - s$	subtract
`subb`		subtract bytes
`subw`		subtract words
`subl`		subtract double-words
`DEC d`	$d \leftarrow d - 1$	decrement
`decb`		decrement byte
`decw`		decrement word
`decl`		decrement double-word
`NEG d`	$d \leftarrow -d$	two's-complement
`negb`		two's-complement byte
`negw`		two's-complement word
`negl`		two's-complement double-word
`NOT d`	$d \leftarrow {\sim}d$	one's-complement
`notb`		one's-complement byte
`notw`		one's-complement word
`notl`		one's-complement double-word
`IMUL s, d`	$d \leftarrow d \times s$	signed multiply
`imulb`		signed multiply bytes
`imulw`		signed multiply words
`imull`		signed multiply double-words
`IMUL d`	$d \leftarrow d \times R_a$	multiply
`imulb`		signed multiply bytes
`imulw`		signed multiply words
`imull`		signed multiply double-words
`IDIV s`	$R_d : R_a \leftarrow R_a / s$	signed division
`idivb`	$R_a \leftarrow R_b \% s$	signed division bytes
`idivw`		signed division words
`idivl`		signed division double-words
`SAR n, d`	$d \leftarrow d \gg n$	arithmetic right shift
`sarb`		arithmetic right shift byte
`sarw`		arithmetic right shift word
`sarl`		arithmetic right shift double-word
`SAL n, d`	$d \leftarrow d \ll n$	arithmetic left shift
`salb`		arithmetic left shift byte
`salw`		arithmetic left shift word
`sall`		arithmetic left shift double-word

Table 4.4 IA32 instructions for arithmetically operating the data.

a three-operand instruction that executes a signed multiply of a 16- or 32-bit immediate by a register or memory word or long and stores the product in a specified register word or long. The instruction MUL works in the same way as IMUL but executes an unsigned multiplication.

To better explain how these instructions operate, consider the following examples:

```
imulb 5(%edi)
mulw 8(%edi), %dx
imull $12345678, 4(%edi), %eax
```

The imulb instruction performs an 8-bit signed multiply of the al register and the byte stored in the memory address given by the esi register plus an offset of 5. The result goes into register ax. The mulw instruction executes a 16-bit unsigned multiply of the contents in the memory addressed given by the edi register plus an offset of 4 and the contents of the edx register. The result is stored in this register. Finally, the imull instruction calculates the signed product between the 32-bit constant 12345678 and the double-word contents in the memory address given by the edi register plus an offset of 4. The result is stored in the eax register.

IDIV executes signed division. It divides a 16-, 32-, or 64-bit register value (dividend) by a register or memory byte, word, or long (divisor). The size of the divisor implies the specific registers used as the dividend, quotient, and remainder, as indicated in Table 4.5. If the resulting quotient is too large to fit in the destination, or if the divisor is 0, an interrupt 0 is generated. Non-integral quotients are truncated toward 0. The remainder has the same sign as the dividend; the absolute value of the remainder is always less than the absolute value of the divisor. The instruction DIV works in the same way as IDIV but executes an unsigned division.

divisor size	dividend	quotient	remainder
byte	ax	al	ah
word	dx:ax	ax	dx
double	edx:eax	eax	edx

Table 4.5 Register implicitly involved in the IDIV and DIV instructions.

SAL left shifts (multiplies) the operand for a count and stores the product in that operand. The MSB is shifted into the carry flag; the low-order bit is cleared. The instruction SAR right shifts (signed divides) the operand for a count, leaving the MSB unchanged. In both cases, the shift count is specified by an immediate value or by the contents of register ecl. Fig. 4.5 illustrates these two arithmetic shift operations, considering their 8-bit versions.

Table 4.6 lists the most used IA32 instructions to perform logical operations.

The AND, OR, and XOR instructions perform, respectively, an AND, OR, and exclusive OR of each bit in the values specified by the two operands and stores the result in the second operand. Fig. 4.6 exemplifies the effects of these three instructions.

SHL is the same operation of SAL. They are synonymous mnemonics. SHR right shifts (unsigned divides) the operand for a count. It sets the MSB to 0. The shift count is specified

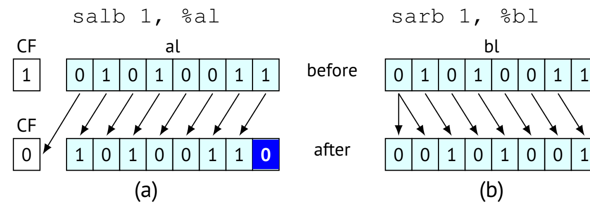


Fig. 4.5 The arithmetic shift instructions.

instruction	effect	description
`AND s, d`	$d \leftarrow d \& s$	bitwise AND
`andb`		bitwise logical AND bytes
`andw`		bitwise logical AND words
`andl`		bitwise logical AND double-words
`OR s, d`	$d \leftarrow d \| s$	bitwise OR
`orb`		bitwise logical OR bytes
`orw`		bitwise logical OR words
`orl`		bitwise logical OR double-words
`XOR s, d`	$d \leftarrow d \wedge s$	bitwise logical exclusive OR
`xorb`		bitwise logical exclusive OR bytes
`xorw`		bitwise logical exclusive OR words
`xorl`		bitwise logical exclusive OR double-words
`SHR n, d`	$d \leftarrow d \gg n$	logical right shift
`shrb`		logical right shift byte
`shrw`		logical right shift word
`shrl`		logical right shift double-word
`SHL n, d`	same as `SAL`	logical left shift
`shlb`	(see Table 4.4)	logical left shift byte
`shlw`		logical left shift word
`shll`		logical left shift double-word

Table 4.6 The IA32 instructions for logically operating the data.

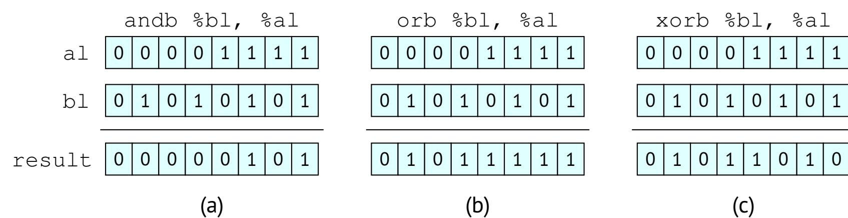


Fig. 4.6 The logical instructions.

by an immediate value or by the contents of register ecl. Fig. 4.7 illustrates the logical right shift operation, considering its 8-bit version.

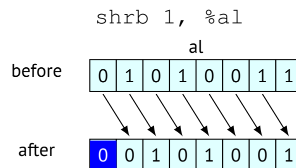


Fig. 4.7 The shift right logic instruction.

4.2.6 Control instructions

As already indicated, the biggest difference between a computer and a simple calculator is the ability of the former to make decisions and to use data whose values depend on data. This allows different instructions to be executed, based on the values computed during the program (i.e., not known in compile time). By default, the machine-level instructions are executed in sequence, i.e., according to the order in which they appear in the program. In some points, it is however necessary to break this default rule. This happens in constructs available in high-level programming languages like `IF-THEN-ELSE`, `SWITCH`, `FOR`, `REPEAT-UNTIL`, and `DO-WHILE` and function/procedure calls.

IA32 includes various instructions that are relevant to control the flow of execution, or seen from a different perspective to support those high-level constructs. Usually, the flow of control is dependent on a given condition, that is, on the values of some variables. If that condition is true, the program jumps to an instruction that is not the next one, thus breaking the default behaviour. Otherwise, the next instruction is executed.

A **condition code**, also known as status flag, is the mechanism used by processors to save a specific attribute related to the last arithmetic or logic operation. By testing a set of these flags, conditional jumps to specific locations into the code can be performed. IA32 provides the following condition codes:

CF (carry flag): The most recent operation generated a carry out of the most significant bit. This flag is used to detect overflow for unsigned operations.
OF (overflow flag): The most recent operation caused a two's-complement overflow, either negative or positive.
SF (sign flag): The most recent operation yielded a negative value.
ZF (zero flag): The most recent operation yielded zero.

Table 4.7 shows some of the most used IA32 instructions to test data.

The `CMP` instruction compares the two source operands, by subtracting the second operand from the first one, and sets the status flags according to the results. It does not alter either of the operands and can be used to determine if the second operand is greater

instruction	effect	description
`CMP s2,s1`	$s_1 - s_2$	compare
`cmpb`		compare bytes
`cmpw`		compare words
`cmpl`		compare double-words
`TEST s2,s1`	$s_1 \& s_2$	test
`testb`		test bytes
`testw`		test words
`testl`		test double-words

Table 4.7 The IA32 instructions for tests.

than, equal to, or less than the first operand. The `TST` instruction performs a bit-wise logical AND of the two operands. The result of a logical AND is 1 if the value of both operands is 1; otherwise, the result is 0. This instruction clears the OF and CF flags, and sets the SF, ZF and PF flags, according to the result.

Each `SET` instruction sets the destination (a single byte) to either 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate mnemonics for the same machine instruction, as indicated in Table 4.8.

instruction	synonyms	effect	set condition
`sete d`	`setz`	$d \leftarrow ZF$	Equal / zero ($= 0$)
`setne d`	`setnz`	$d \leftarrow \sim ZF$	Not equal / not zero ($\neq 0$)
`sets d`		$d \leftarrow SF$	Negative (< 0)
`setns d`		$d \leftarrow \sim SF$	Nonnegative (≥ 0)
`setg d`	`setnle`	$d \leftarrow \sim(SF \wedge OF) \& \sim ZF$	Greater (signed $>$)
`setge d`	`setnl`	$d \leftarrow \sim(SF \wedge OF)$	Greater or equal (signed $\geq$)
`setl d`	`setnge`	$d \leftarrow SF \wedge OF$	Less (signed $<$)
`setle d`	`setng`	$d \leftarrow (SF \wedge OF) \| ZF$	Less or equal (signed $\leq$)
`seta d`	`setnbe`	$d \leftarrow \sim CF \& \sim ZF$	Above (unsigned $>$)
`setae d`	`setnb`	$d \leftarrow \sim CF$	Above or equal (unsigned $\geq$)
`setb d`	`setnae`	$d \leftarrow CF$	Below (unsigned $<$)
`setbe d`	`setna`	$d \leftarrow CF \| ZF$	Below or equal (unsigned $\leq$)

Table 4.8 The IA32 `SET` instructions.

By default, instructions follow each other in the exact order they are listed in the program. IA32 provides instructions to break this normal (i.e., sequential) flow of execution (Table 4.9), allowing the program to "jump" to an instruction that is different from the next one. These jump destinations are generally indicated in assembly code by a label. They effectively jump to the destination, if a given condition does hold. Otherwise, the jump is not executed and the program continues as usual, i.e., with the next instruction. Some jump instructions have "synonyms" (alternate names for the same machine instruction).

The next example shows a piece of C code with an `IF-THEN-ELSE` construct and a behaviourally-equivalent piece of assembly code, which includes two jump instructions `jne` and `jmp`.

instruction	synonyms	condition	description
`jmp l`			Direct jump
`jmp *o`			Indirect jump
`je l`	`jz`	ZF	Equal / zero ($= 0$)
`jne l`	`jnz`	$\sim ZF$	Not equal / not zero ($\neq 0$)
`js l`		SF	Negative (< 0)
`jns l`		$\sim SF$	Nonnegative (≥ 0)
`jg l`	`jnle`	$\sim(SF \wedge OF) \& \sim ZF$	Greater (signed $>$)
`jge l`	`jnl`	$\sim(SF \wedge OF)$	Greater or equal (signed $\geq$)
`jl l`	`jnge`	$SF \wedge OF$	Less (signed $<$)
`jle l`	`jng`	$(SF \wedge OF) \mid ZF$	Less or equal (signed $\leq$)
`ja l`	`jnbe`	$\sim CF \& \sim ZF$	Above (unsigned $>$)
`jae l`	`jnb jnc`	$\sim CF$	Above or equal (unsigned $\geq$)
`jb l`	`jnae jc`	CF	Below (unsigned $<$)
`jbe l`	`jna`	$CF \mid ZF$	Below or equal (unsigned $\leq$)

Table 4.9 The IA32 `JUMP` instructions.

```
if (a==b)                           cmpl %eax, %bx
   c+=3;                            jne .lb2
else                          .lb1: addl $3, %ecx
   c++;                             jmp .lb3
                              .lb2: addl $1, %ecx
                              .lb3: ...
```

The example assumes that variables `a`, `b`, and `c` are saved in registers eax, ebx, and ecx, respectively. The instruction `jmp .lb3` causes the program to skip over the instruction labeled `.lb2` and instead to continue execution with the instruction labelled by `.lb3`. This in an unconditional jump, as it is always performed and does not depend on the evaluation of a given condition. The `jne .lb2` instruction is a conditional jump that depends on the ZF flag, that was set by the `cmpl` instruction. If the $ZF = 1$, the program jumps to the instruction labelled by `.lb2`. Otherwise (i.e., if $ZF = 0$), the jump does not occur and the program flows as usual, executing the next instruction, which in this case is the one labeled by `.lb1`.

A **label** constitutes a human convenience and is represented by a symbolic name (string) followed by a colon (:). Whenever the assembler encounters a new label, it calculates its memory address and saves that information in an auxiliary data structure. When the object-code is generated, the assembler uses that data and encodes the jump targets (the addresses of the destination instructions) as part of the jump instructions.

Direct jumps are written in assembly code by giving a label as the jump target, for example, the label `.lb1` in the example considered previously. Indirect jumps are specified with a '*' followed by an operand specifier which can be a register or a memory operand with any of the formats listed in Table 4.2. For instance, the instruction

```
jmp *%eax
```

uses the value in register eax as the jump target, and the instruction

```
jmp *4(%eax)
```

reads the jump target from memory, using as the read address the sum of the value in eax with 4.

The typical control flow constructs available in high-level languages, like `FOR`, `WHILE-DO`, `DO-WHILE`, `REPEAT-UNTIL`, and `SWITCH`, can be reproduced. The next example shows a `WHILE-DO` loop in C code and its possible translation into assembly code. Variable `a` is saved in register eax.

```
while (a<50) {                .lb1: cmpl $50, %eax
   ...                              jge .lb2
   a++;                             ...
}                                   incl %eax
                                    jmp .lb1
                              .lb2: ...
```

Assembly code for `DO-WHILE` loops can be made more efficient than for WHILE loops, since they only require one jump instruction (and not two), as next exemplified:

```
do {                          .lb1: ...
   ...                              incl %eax
   a++;                             cmpl $50, %eax
} while (a<50)                      jl .lb1
                              .lb2: ...
```

It is relevant to present how the jumps are coded at the machine level. IA32 uses two major encodings for jumps. In an **absolute jump**, the instruction directly indicates the location in the memory to where the program jumps (i.e., the value to be directly copied to register eip). In IA32, this location occupies four bytes. In a **relative jump**, the value indicated in the instruction is a signed offset that is added to the address of the instruction following the jump instruction to calculate the destination. This offset value can be 8-bit, 16-bit, or 32-bit long. The use of the address of the next instruction (and not the one of the jump instruction) results from the fact that the processor updates the instruction pointer as one of the first steps in executing an instruction. At the machine level, two assembly instructions with the same mnemonics can correspond to different machine level instructions (with different opcodes). This should be not seen as a surprise, since the opcode is used to identify the instruction, but also other elements, namely the number and the type of operands. The next example show the representations of two `jg` instructions at the machine-level (bytes are represented with two hexadecimal digits):

```
jg .lb1     7F 0C
jg .lb2     0F 8F 02 01 00 00
```

The first `jg` instruction (opcode 7F) uses just one byte to indicate the offset, while the second one (opcode 0F) uses four bytes, which are the last ones (00 00 01 02, according to the little endian order). The assembler and the linker select whether a jump uses relative or absolute destinations. The next example exemplify how relative jumps work at the machine level.

```
      movl %edi, %eax          0: 89 C7
      jmp .lb1                 2: EB 01
.lb2: decl %eax                4: 48
```

```
.lb1: testl %eax, 400(%esi)     5: 85 46 90 01 00 00
      jg .lb2                   B: 7F F7
      incl %ebx                 D: 43
```

In each line, one sees an assembly instruction, the relative address (in decimal), and its machine-level representation (in hexadecimal). The `movl` instruction is located in relative address 0 and occupies two bytes (89_{16} and $C7_{16}$). The `jmp` instruction is located in relative address 2 and also occupies two bytes (EB_{16} and 01_{16}). The last byte indicates the offset for the jump, which in this case is +1. It is calculated as the offset that must be added to the relative address of the next instruction (`decl`) to reach the destination (relative address of the instruction in label `.lb1`). For the `jg` instruction the offset is the second byte, whose value is $F7_{16}$=-9. This is in fact the distance in bytes of the instruction following the `jg` instruction (in this case `incl`) to the instruction with label `.lb2`. This negative value indicates that the destination is above in the memory (i.e., an address with a smaller numeric value).

4.2.7 Procedures

A **procedure** (also called subprogram, function, method, subroutine, handler) is a fundamental concept in software and programming. It constitutes an important mechanism to encapsulate code that implements some functionality with a well-defined set of arguments and a return value. Procedures can be invoked from different points in a program, thus being a good mechanism to reuse code. They allow the implementation of some function to be hidden, while providing a clear interface of what arguments are used and what results are generated.

To support the discussion in this subsection, assume that procedure procA (the calling or caller procedure) calls procedure procB (the called or callee procedure), which is executed and returns back to procA. This simple idea implies the following steps:

1. **Control**: The instruction pointer must be set to the starting address of the code for procB upon entry; when procB ends, the instruction pointer must be set to the instruction in procA that follows the call to procB.
2. **Arguments and return value**: procA must be able to provide the arguments to procB, and procB must be able to return a value back to procA.
3. **Local memory**: procB may need to allocate space for local variables when it starts executing and then free it before returning back to procA.

The IA32 support of procedures entails a combination of specific instructions and a set of conventions on how to use the machine resources, such as the registers and the program memory. While procB is executing, procA and its predecessors in the sequence of calls are temporarily in standby. While procB is running, it is able to allocate new memory space for its local variables or to call another procedure (or even itself). When procB finishes (i.e., returns the control to procA), the storage allocated must be freed. This dynamic nature of the temporary data within procedures is naturally supported by a stack. The stack and some registers save the information needed to pass control, the arguments and the return

value, and to allocate memory for local variables. As procA calls procB, control and data information is pushed to the stack. This information is popped when procB returns the control to procA.

The structure that is created in the stack to support the execution of each procedure call is called **stack frame** (or activation record). It includes space for the arguments, the return information, the registers, and the local variables, as shown in Fig. 4.8. The stack frames for most procedures are of fixed size, allocated at the beginning of the procedure. In this book, variable-sized frames are not considered. Some parts may be omitted if not needed, to optimise the program for size or time. So, in many cases, procedures allocate only the parts of stack frames that are strictly needed. For example, when a given procedure has six or fewer arguments, all can be passed in registers.

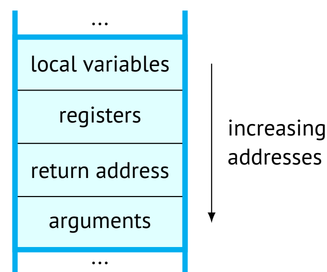


Fig. 4.8 The general structure of a stack frame.

Before procedure procA calls procedure procB, it needs to put the arguments into the stack frame. Generically, this can be accomplished by pushing the values of the arguments into the stack with `PUSH` instructions. In some cases, arguments are passed in registers, a faster and more convenient way to do it, but this approach is only possible when the number of arguments is low (and the chain of calls is relatively short). When all arguments are in the stack, procedure procA can pass the control to procedure procB, simply by setting the instruction pointer (IP) to the starting address of the code for procB (i.e., to its first instruction). This is similar to a `jump` instruction, with the difference that the processor must record the code location where it should resume the execution of procA, when procB returns. This information is recorded by invoking procedure procB with the instruction `call procB`. This instruction pushes an address onto the stack, called the **return address**, and sets the IP to the beginning of procB . The return address is computed as the address of the instruction immediately following the `call` instruction. This mechanism is necessary since a procedure can be called from many different parts of the program, and it must be able to get back to wherever it was called from.

The formats of the `call` instructions are listed in Table 4.10. Like jumps, `call` instructions can be either direct or indirect. The target of a direct `call` is given as a label, whereas the target of an indirect `call` is given by '*' followed by an operand (a register or a memory location).

The procB procedure is then responsible for allocating the space required for saving the values of registers and the local variables. Registers, whose values need to be preserved, are

instruction	effect	description
`call l`	pushl $eip; eip \leftarrow l$	Procedure call
`call *o`	pushl $eip; eip \leftarrow *o$	Procedure call
`leave`	$esp \leftarrow ebp$; popl ebp	Procedure exit
`ret`	popl eip	Return from call

Table 4.10 Instructions that specifically support procedures in IA32.

just pushed onto the stack. In fact, not all registers need to be saved, as there is a convention that all programmers and compilers are expected to follow. A register is classified according to two types:

- A **callee-saved register** is a register whose value must be preserved by the callee procedure;
- A **caller-saved register** is a register whose value must be preserved by the caller procedure.

Registers ebx, esi, edi, and ebp are classified as callee-saved registers. When procA calls procB, the latter must preserve the values of these registers, ensuring that they have the same values when procB returns to procA as they had when procB was called. Procedure procB can preserve a register value by two ways: (1) not changing its value, or (2) pushing the original value on the stack, using it, and then popping the old value from the stack before the control is passed to procA. Register ebp is used as a reference pointer for accessing all the elements in the stack frame. Since ebp is a callee-saved register, all called procedures must always save this register. Registers eax, ecx, and edx are caller-saved registers, which means that they can be freely modified by a calling procedure. Thus, it is the responsibility of the caller procedure to push these registers onto the stack or copy them somewhere else, before a call is made, whenever one wants to make sure that they are not affected by a calling procedure.

Space for local data with no specified initial value can be allocated on the stack by just decrementing the stack pointer by an appropriate amount. The size of the space for local variables must be at least equal to the sum of the sizes of each local variable. Otherwise, the initial values of the local variables can be directly pushed to the stack. Each variable must be associated with a specific part of that space. As happens for arguments, in some cases, local variables are assigned to registers, but again this approach is only possible when the number of local variables is low. When these two allocations (local data and registers) are finished, procedure procB can start its execution.

When procB is finished, the control must be passed to procA. To this end, procB passes the return value, if present, through register eax (recall that this is a caller-saved register). It also needs to "clean" its stack frame, because it is no longer needed. So, the parts in the stack frame are subsequently freed. The space for local variables can be deallocated simply by incrementing the stack pointer. The next elements to be freed are the registers. This implies popping their values from the stack. In this way, the values they had just before procedure procA called procB are recovered. The last register to be popped is the instruction pointer (eip in IA32). This occurs when the instruction `ret` (see Table 4.10) is executed, which pops the return address from the stack and copies it to the eip.

To better explain how a stack frame is built in a concrete context, let us assume the following procedure `sum`.

```
int sum (int n, int v) {
   int i, val=0;
   for (i=1; i<n; i++)
      val = val + v + i;
   return (val);
}
```

It has two arguments (`n` and `v`) and two local variables (`i` and `val`), and returns a value. All these elements are integers (type `int`), which occupy four bytes. Assume also that in the body of the procedure, registers eax and ebx are modified. Before procedure `sum` is called, the state of the registers and the memory is as shown in Fig. 4.9(a). The stack pointer points to cell $211C_{16}$.

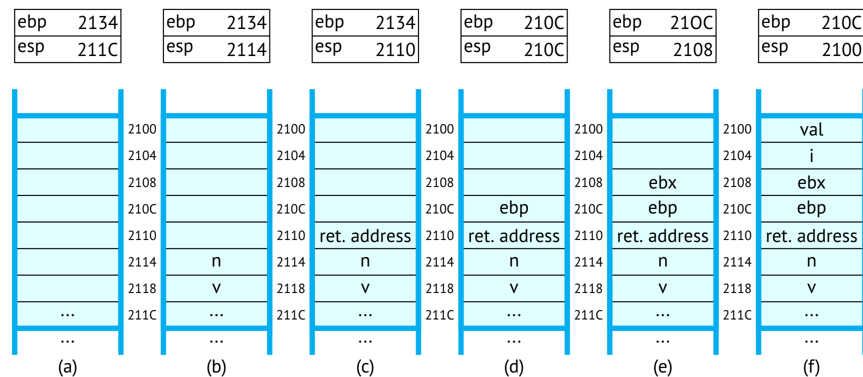


Fig. 4.9 The stepwise construction of the stack frame for procedure `sum`. The values in the registers and to indicate memory addresses are represented in hexadecimal.

The arguments `n` are `v` are the first elements to be pushed to the stack. The responsibility of this operation belongs to the calling procedure. In C programs, the convention is to push arguments from right to left. So, the value of `v` is the first to be pushed to the stack (in position 2118_{16}), followed by the value of `n` (in position 2114_{16}) as illustrated in Fig. 4.9(b).

The next step is to call procedure `sum`, which is obviously executed by the calling procedure. The effect of the call is to write in the top of the stack (position 2110_{16}) the return address and to pass the control to procedure `sum`, as Fig. 4.9(c) depicts.

The first instructions of the procedure `sum` (actually of all procedures) are:

```
pushl %ebp
movl %esp, %ebp
```

The effect of these two instructions is to save register ebp into the stack and to attribute it a new value, as Fig. 4.9(d) shows. This caller-save register is used within the procedure `sum` as the base pointer to access all the elements in the stack frame, by adding or subtracting

appropriate offsets. Its value is fixed in this step, which in this case is $210C_{16}$. To access, for example, argument n, an offset of +8 needs to be added to ebp. Argument v requires an offset of +12. Register ebx is also pushed into the stack frame, since its value is changed during the execution of the procedure. The state of the memory and the registers after this push is represented in Fig. 4.9(e).

The final step is to allocate space for the two local variables (of type int) and to initialise the value of val with 0. An alternative to allocate this space is to use the following two instructions:

```
subl %esp, 8
movl $0, -12(%ebp)
```

The subl instruction moves the stack pointer eight positions above, reserving thus this number of bytes for variables val and i. The order of the variables is totally arbitrary, as long as they do not overlap. The movl instruction initialises the value of val with 0. Note that this variable is 12 positions ($2100_{16} - 210C_{16} = -C_{16} = -12$) above in the memory with respect to the position pointed by ebp.

A second alternative to perform this allocation is the following:

```
subl %esp, 4
pushl $0
```

At this point, the stack frame is complete and the procedure can start its execution. When the procedure sum is ready to pass the control to its calling procedure, the various steps in Fig. 4.9 are executed in the reverse order.

The first operation is to pass the return value, which in IA32 follows the convention of simply using register eax for that purpose. The next step is to deallocate the space occupied by the local variables. This is accomplished by adding +8 to the esp register. The effect is shown in Fig. 4.9(e). Next, register ebx is popped from the stack (Fig. 4.9(d)). Then, the value of ebp is also popped from the stack, restoring the stack frame of the caller. The new value for ebp is in fact the base pointer for the stack frame of the calling procedure. This operation can be achieved with the instruction leave. The situation is now represented in Fig. 4.9(c). The next step is jump to the return address with the ret instruction. This passes the control to the calling procedure and leaves the registers/memory in the state shown in Fig. 4.9(b). Finally, the calling procedure deallocates the space that was used to pass the arguments. In this case, this can be made with the instruction add %esp, 8. The stack returns to the initial state (Fig. 4.9(a)).

The conventions for using the registers and the stack permit a procedure to call itself recursively. In fact, the process followed when a procedure calls itself is roughly the same as when it calls a different procedure. Each procedure call uses its private space on the stack, and so the local variables of a specific call do not interfere with those of a different call. The conventions on using the stack constitute a proper mechanism for allocating local storage whenever a procedure is called and deallocating it whenever returning.

4.2.8 Data structures

A **scalar variable** contains only a value of a primitive datatype, like a character, an integer, or a floating-point number. Since a given scalar variable occupies a reduced number of bytes (1, 2 or 4, depending on its datatype), it can be saved in the memory or in a given register.

In general, global variables are allocated in memory. The address of each memory address is determined by the linker, when all modules are merged into a single executable program. In assembly, the global variables are referred by their symbolic name (i.e., by the respective label). Global variables can also be stored in registers, but this is not an approach that can be generalised, as the number of registers is very reduced.

A **structured variable** is composed of a set of scalar variables and other structured variables. Examples of variables in this category are a `record` in Pascal, a `struct` and a `union` in C, and an `array`, which is a construct available in all high-level languages. The size of these variables does not permit their allocation to registers. They should be stored in the memory.

C uses a simple implementation of arrays, so the translation into machine code is straightforward. In C, one can use pointers to access the elements of the arrays and perform arithmetic with the pointers. Compilers are good at simplifying the address computations used by array indexing. However, these simplifications make less evident the mapping between the C code and its translation into machine code.

Assume the following generic declaration in C, where N is an integer:

```
datatype arr[N];
```

This declaration allocates a contiguous region of $N \times S$ bytes in memory, where S is the size in bytes of the datatype. It also introduces an identifier `arr` that can be used as a pointer to the beginning of the array. The value of this pointer is denoted as b_{arr}. The array elements are accessed using an index that ranges from 0 to N-1. Array element i is stored at address $b_{\text{arr}} + i \times S$.

To exemplify these concepts, consider the following declarations in C:

```
char e[12];
int f[5];
char *g[4];
short int h[5];
```

Array `e` consists of 12 single-byte elements. Array `f` consists of five integers, each requiring four bytes. So, in total, 20 bytes are allocated to this array. Array `g` contains pointers, and hence the array elements are four bytes each. In total, it needs 16 bytes. Elements of the array `h` have the type `short int`, each occupying two bytes. So, this 5-element array occupies 10 bytes. Fig. 4.10 shows for these four arrays the space allocated in the memory and the addresses for their elements.

The addressing modes of IA32 ease the access to arrays. If registers ebx and esi store respectively the base address of array `e` and the index i that is needed, the following instruction initialises `e[i]` with constant value 0.

```
movb $0, (%ebx, %esi, 1)
```

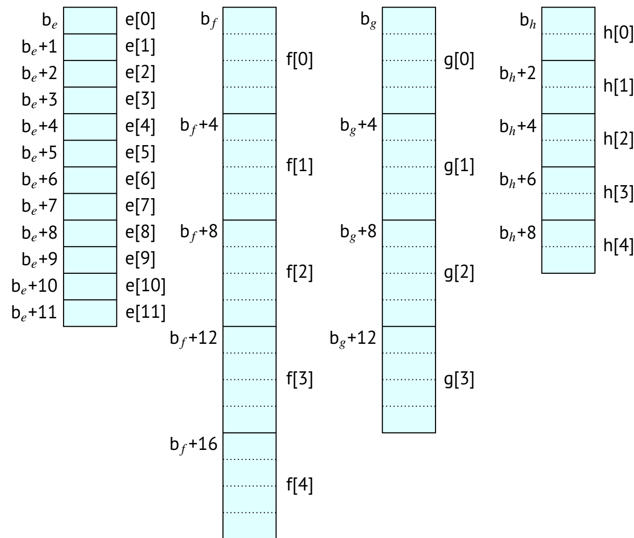


Fig. 4.10 The space reserved in the main memory for different arrays.

If esi is equal to 2, the memory address that is affected is given by $b_e + 2 \times 1$, that is two positions below the base address which is the location for `e[2]`.

Similarly, for array `f`, the following instruction initialises `f[i]` with constant value 10, if registers ebx and esi store the base address of the array `f` and the index i that is needed, respectively.

```
movl $10, (%ebx, %esi, 4)
```

If esi is equal to 4, the memory address that is written is $b_f + 4 \times 4$, that is 16 positions above the base address, which is the location for `f[4]`.

The general mechanism used for unidimensional arrays also applies when one handles multidimensional arrays. Consider a two-dimensional array of integers, with two rows and two columns:

```
int i[2][2];
```

This array has four integers, so it occupies 16 bytes. The array elements are stored in the memory in row-major order, meaning that all the elements of row 0 go first, which are followed by all elements of row 1. Fig. 4.11(a) shows the space allocated in the memory and the relative addresses for the elements of array `i`.

With respect to a `struct` declaration, its implementation is similar to that of arrays in the sense that all its components are stored in a contiguous area of the memory and the base pointer to that structure is the address of its first byte. The compiler uses information about each structure type to store the byte offsets of all fields, which are used as displacements with respect to the base address.

As an example, consider the following structure declaration:

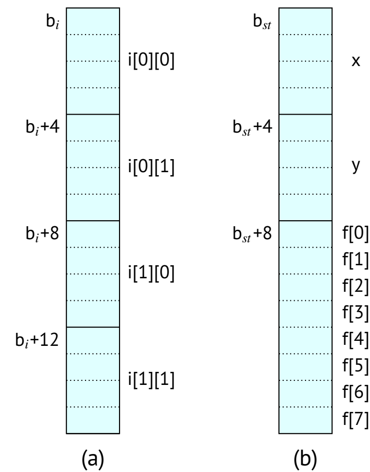


Fig. 4.11 The space reserved in the main memory for (a) two-dimensional array `i` and (b) variable `st` of type `struct rec`.

```
struct rec {
    int x;
    int y;
    char f[8];
}
```

This structure contains three fields: two 4-byte values of type `int`, an eight-element array of type `char`, giving a total of 16 bytes. A possible placement of this structure in memory is illustrated in Fig. 4.11(b). This placement is arbitrary in the sense that any arrangement where the fields are contiguously located is valid. Notice that the `f` array is part of the structure.

The addressing modes of IA32 are also very convenient to access structures. Suppose that variable `st` is of type `struct rec`. If register ebx contains the base address of `st`, then the following instructions access `st.x`, `st.y` and `st.f[5]`. To calculate the offset for `st.f[5]`, one needs to add the offset for the `f` array field (+8) with the offset of the array element with index 5 with respect to the base of that array (+5). So, the total offset is 13. Note also that the offset of field `x` is zero, as it is the first field of the structure in the memory.

```
movl $10, (%ebx)            /* st.x = 10 */
movl $15, 4(%ebx)           /* st.y = 15 */
movl $75, 13(%ebx)          /* st.field[5] = 'K' */
```

Exercises

Exerc. 4.1: After the execution of instruction `movl %ebx, $721`, what are the decimal values for the contents of registers ebh and ebl?

Exerc. 4.2: Consider that the following values are stored at the indicated memory addresses and registers. All values are represented in hexadecimal.

address	value	address	value	register	value
110	FF	118	13	eax	110
111	0	119	0	ebx	A
112	0	11A	0	ecx	1
113	0	11B	0	edx	3
114	AB	11C	55		
115	0	11D	0		
116	0	11E	0		
117	0	11F	0		

(a) Calculate the values for the indicated operands: i) `%eax`, ii) `0x114`, iii) `$0x118`, iv) `(%eax)`, v) `4(%eax)`, vi) `9(%eax,%edx)`, vii) `280(%ecx,%edx)`, viii) `0xFC(,%edx,8)`, ix) `2(%eax,%ebx)`.

(b) For each instruction, indicate the result and where it is stored:

```
addl %eax, %ebx
addl (%eax), %ecx
subl 4(%eax), %edx
andl $43, (%eax,%edx,4)
decl %edx
incl 8(%eax)
imull %eax, %ebx
sall 2, %ebx
```

Exerc. 4.3: Complete the targets of the instructions.

```
40F780: 75 03             jne  ...

8318A1: 0F 85 F1 FE FF FF jne  ...
```

Exerc. 4.4: Indicate the addresses of the instructions.

```
 ...  : 77 20   jne 300834

 ...  : EB E8   jmp 854FA2
```

Exerc. 4.5: Consider the following C program, named `sc1.c`:

```
<#include<stdio.h>
int a, b, c;
int main () {
   scanf("%d", &a);
   b = a*2;
   c = b-a;
   printf("%d %d\n", b, c);
}
```

Generate the assembly code for this program, with the following command line:

```
gcc -m32 -O0 -S -o sc1-0.s sc1.c
```

(a) Identify how each C instruction/constructor was translated into assembly code
(b) Repeat the previous question, but first replace the `scanf` instruction with:

```
a=10;
```

Exerc. 4.6: Consider the following C program, named `sc2.c`:

```
<#include<stdio.h>
int i=10, j, k, l;
int main () {
   scanf("%d", &j);
   if (i<j)
      k = i+j;
   else
      k = i-j;
   l=3*k;
}
```

Generate the assembly code for this program, with the following command lines:

```
gcc -m32 -O0 -S -o sc2-0.s sc2.c
```

(a) Identify how each C instruction/constructor was translated into assembly code.
(b) Identify which modifications occur if the constant 3 in the instruction "l=3*k" is replaced by 4, 7, 9, 24, and 39.

Exerc. 4.7: Consider the following program written in C and analyse the assembly code generated by the `gcc` compiler with `-O0` option.

```
<#include<stdio.h>
int array[100], sum=0;
int main () {
   int i;
   for (i=0; i<100; i++)
      scanf("%d", &array[i]);
   for (i=0; i<100 && array[i]>0; i++)
      sum += array[i];
}
```

Exerc. 4.8: Consider the following C program and compile it into IA-32 assembly code with the `gcc` compiler.

```
1  #include <stdio.h>
2  int main() {
4     int n;
4     scanf("%d", &n);
5     if (n%2!=1)
6        badDec2bin(16);
7     else
8        badDec2bin(44);
9  }
10 int badDec2bin (int n) {
11    int c;
12    for (c=16; c>=0; c-) {
13       if (n»c & 16)
14          printf("1");
15       else
16          printf("0");
17    }
18 }
```

(a) Build with the maximum detail the stack frame for function `badDec2bin`, indicating the size and the position of each element.
(b) Identify how the C instructions in lines 5, 8, 12, and 13 have been translated into assembly code by the `gcc` compiler with `-O0` option.

Exerc. 4.9: Complete the C program based on the respective assembly code.

```
int cmpXY (int x, int y) {
   int val =  ... ;
   if ( ... ) {
      if ( ... )
         val = ... ;
      else
         val = ... ;
   }
   else
      if ( x ... )
         val = ... ;
      return val;
}
```

```
cmpXY: ...
       movl 12(%ebp), %eax
       movl 8(%ebp), %ecx
       movl $0, -4(%ebp)
       movl 8(%ebp), %edx
       cmpl 12(%ebp), %edx
       je LBB5
       jle LBB3
       movl $1, -4(%ebp)
       jmp LBB4
LBB3:  movl $2, -4(%ebp)
LBB4:  jmp LBB8
LBB5:  cmpl $10, 8(%ebp)
       jle LBB8
       movl $3, -4(%ebp)
LBB8:  movl -4(%ebp), %eax
       ...
```

Further reading

Tanenbaum and Austin (2013, Chapter 1) and Null and Lobur (2003, Chapter 6) present throughout discussions of the different levels one can see a digital computer.

A good complementary source is the book by Bryant and O'Hallaron (2011) (2nd edition), because they use Y86, a simpler version of IA32. The 3rd version of their book (Bryant and O'Hallaron, 2016) was upgraded to a 64-bit processor and is obviously also a recommended reference.

Not all aspects related to the IA32 assembly language, namely its instruction set is covered in this chapter. The interested reader is pointed to the official instruction set documents provided by Intel in its website.

Blum (2005, Chapter 3) presents in great detail the set of tools that is need for creating assembly code programs, with a special focus ion the programming development tools provided by the GNU project.

Other computer architectures are addressed in different books: ARM (and also x86) (Stallings, 2019), MIPS (Patterson and Hennessy, 2014), MARIE Null and Lobur (2003), and RISC-V (Hennessy and Patterson, 2017).

Chapter 5
Cache memory

Abstract A CPU cache is a hardware memory used by the CPU of a computer to reduce the average cost (time or energy) to access data or instructions from the main memory. A cache is a smaller, faster memory, located closer to a processor core, which stores copies of the data and instructions from frequently used main memory locations. This chapter discusses the main principles of caches, namely the different mapping functions and replacement algorithms. The concepts and principles discussed here are important, because they can be generalised to other computational contexts. For instance, mobile applications also use cached data, which are files, scripts, images, and other multimedia stored on the mobile device, after opening the application for the first time. This data in the cache is then used to quickly load information about the application every time it is reopen.

5.1 Main principles

CPUs have always been faster than memories. As semiconductor technology progresses, this processor-memory gap continues to increase. It is easier and cheaper to make processors run faster than it is to make main memory operate faster. So, whenever the CPU needs to access the main memory, it must wait for many CPU cycles. The slower the memory, the more cycles the CPU must wait. This undoubtedly degrades the performance of the computer. A typical solution to the processor-memory gap is to design a computer to have **cache memory**. Caches are designed to combine the access time of expensive and fast memory with the large size of cheap but slow memory. The purpose of a cache is to speed up memory accesses by storing recently used data closer to the CPU, instead of storing it in the memory. Although the cache is not as large as the main memory, it is significantly faster.

The basic idea for using a cache is simple: the most frequently requested memory words are stored in the cache. Whenever the CPU needs a given word, it first looks in the cache. It is expected that most of the times, the needed word is stored in the cache. If this is not the case, then the CPU must access the main memory to get it. If a substantial percentage of the words are in the cache, the average access time is greatly reduced.

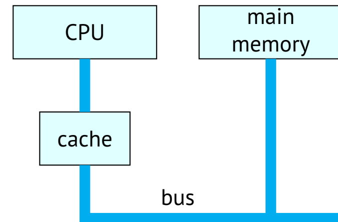


Fig. 5.1 The cache is logically between the CPU and main memory.

The success of the cache in reducing the accesses to the memory greatly depends on what fraction of the words are in the cache. The fact that programs do not access the memory completely in an arbitrary way favours the success of caches. The use of the memory has some degree of predictability. In fact, an important observation related to the execution of programs is the **principle of locality**. Programs tend to reuse data and instructions that were recently used. A well-known rule of thumb is that many programs spend 90% of their execution times in 10% of the code. Thus, one can predict which instructions and pieces of data are to be used in the near future based on the most recent accesses. This principle has two different types:

- **temporal locality** implies that recently accessed data or instructions are likely to be needed in the near future;
- **spatial locality** states that items whose memory addresses are close to each other tend to be referenced close together in time.

These two types of locality are observed, for example, during the execution of a loop that accesses all the elements of an array or matrix, which are contiguously stored in the memory. This principles also applies to code, as instructions, by default, are fetched and executed by the processor from consecutive positions in the memory, except for jumps and procedure calls. Furthermore, most program execution time is spent in loops, in which a reduced number of instructions are repeatedly executed.

Whenever the processor tries to obtain a given word from the cache but fails because the data is not stored there, i.e., whenever there is a **cache miss**, the word must be read from memory. To take into account the two types of locality, instead of just getting that specific word, its neighbours are also brought from the main memory into the cache, because it is very likely that many of them will be needed shortly. This approach is more efficient than fetching individual words, since it is faster to fetch n words all at once than one word n times.

A **cache hit** occurs when the requested data is located in the cache. With the number of hits and misses, it is possible to calculate the **hit rate** and the **miss rate**, which are the percentages of memory accesses found and not found, respectively, in the cache, during the execution of a program or of a part of it. The time required to fetch the requested data in the cache is designated as **hit time**. The **miss penalty** is the time required to process a miss, which includes substituting a block in the cache and the additional time to deliver the requested data to the processor. Typically, the time to process a hit is significantly smaller than the time to process a miss.

To explain how caches operate, consider Fig. 5.2, which depicts the elements of a cache/main-memory system. A **cache block** is the minimum unit of transfer between the cache and the main memory. This term also refers to the physical location in the main memory and in the cache. A **cache line** is the portion of the cache memory that can store one block. A line includes control information, namely a **tag**, which is a group of bits that permits identifying the memory address of the block stored in that cache line. Each cache line also includes a **valid bit** that indicates whether the data in the line is valid or not. When the system is booted, all cache lines are obviously marked as not valid.

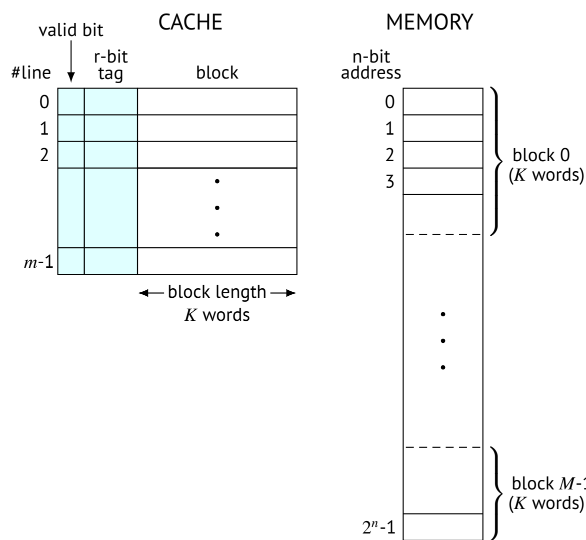


Fig. 5.2 A memory-cache system.

The main memory is composed of 2^n addressable words, with each one having a unique n-bit address. This memory is viewed, for the purpose of transferring data from or to the cache, as consisting of a number of fixed-length blocks of K words each. So, M blocks exist in the main memory ($M = \frac{2^n}{K}$). The cache consists of m lines. Each line contains a block (K words), plus a tag. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since it was loaded into the cache. The **line size** is the length of a line, not including tag and control bits, i.e., it refers to the number of data bytes contained in a line.

The number of lines is much smaller than the number of the main memory blocks ($m \ll M$). In each moment, the cache contains a subset of the blocks of memory. If a word is read from memory , the respective block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a tag that identifies which particular block is currently being stored. The tag is usually the initial part of the address of the main memory.

Whenever reading a word from the memory, the processor requests that word from the cache. If there is a hit, it is delivered to the processor. When a cache miss occurs, the block containing the word must be loaded into the cache, not just the word needed, and the requested word must be delivered to the processor. For example, with a 64-byte line size, a reference to memory address 772 (001100 000100$_2$) brings the line consisting of bytes 768 (001100 000000$_2$) to 831 (001100 111111$_2$) into one cache line.

If a word is read or written w times in a very short interval, the computer needs one reference to the main memory and $w-1$ references to the cache. The larger w is, the better the overall performance. To formalise this calculation, the following values are needed: the cache access time t_c, the main memory access time t_m (which is the same as the miss penalty), and the hit ratio r_h. In the previous example, $r_h = \frac{w-1}{w}$. The miss ratio is given by $r_m = 1 - r_h$. The **mean access time** is calculated by $t_{ma} = t_c + r_m \times t_m$. If $r_h \to 1$ ($r_m \to 0$), all references can be obtained from the cache, and the access time approaches t_c. Contrarily, if $r_h \to 0$ ($r_m \to 1$), a memory reference is needed every time, so the access time approaches $t_c + t_m$, first a time t_c to unsuccessfully check the cache, and then a time t_m to access the memory. Some systems initiate the memory reference simultaneously with the cache search, so that if a cache miss occurs, the memory cycle is already under way. This strategy is however harder to implement.

5.2 Mapping function

Contrarily to what happens with the main memory, caches are not accessed by address, but instead by content. For this reason, cache is often called **content-addressable memory (CAM)**. In the cache, both (part of) the address and the content are stored side by side. When a given memory address is being requested, all cache entries must be searched to check if it is stored there. When the address matches, the corresponding content is fetched from the cache memory.

This need results from the fact that the number of cache lines is smaller than the number of memory blocks. Therefore, an algorithm for mapping memory blocks into cache lines is needed. Additionally, one needs a mechanism for determining which specific main memory block currently occupies a cache line. The selection of the mapping function determines how the cache is logically organised. Three techniques are usually adopted: direct, associative, and set-associative.

To explain the concepts associated with caches, assume the memory-cache system illustrated in Fig. 5.3. This system has a cache with four lines ($m = 4$), each one storing two words ($K = 2$). The memory has 16 positions, which implies 4-bit addresses ($n = 4$). In this example, the address is at the byte level, which is aligned with most contemporary machines. In total, there are eight different blocks in the memory ($M = 8$).

The simplest approach is **direct mapping**, which maps each block of main memory into only one possible cache line. This mapping follows a modular approach. The number of the cache line i is given by formula $i = j \mod m$, where j is the memory block number. The memory block number is given by the memory address but ignoring the $log_2 K$ LSBs. So, for the example, the mapping is expressed as $i = j \mod 4$. Therefore, blocks 0 and 4

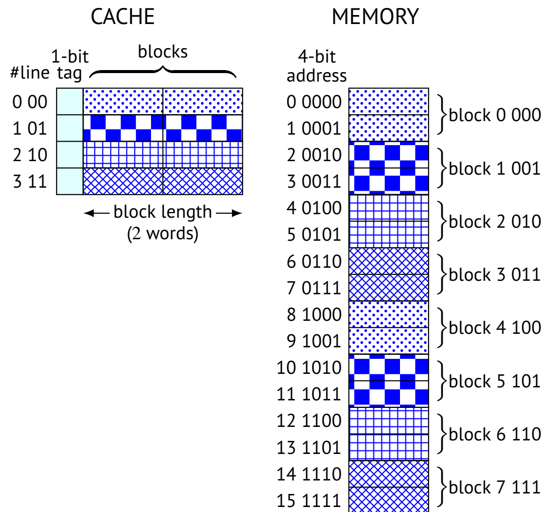


Fig. 5.3 A small memory-cache system with direct mapping. All numbers are indicated in decimal and binary.

($j \in \{0,4\}$) are mapped to line 0 ($i = 0$), blocks 1 and 5 to line 1, and so on (the colours in Fig. 5.3 show which blocks map into which cache lines). So, the two last bits of the block number (i.e., the two central bits of the memory addresses) indicate the respective cache line. The MSB of the block number is stored in the tag of the cache line, to identify which block is stored there. If a '0' is stored in the tag of cache line '10', the block that is stored there is 010 (block 2); if instead the tag stores a '1', the block is 110 (block 6). In general, with this technique, blocks 0, m, $2m$, $\ldots$ all map to cache line number 0. Similarly, blocks 1, $m+1$, $2m+1$, $\ldots$ all map to cache line number 1, and so on.

The direct mapping function can be implemented using the main memory address. For the purpose of cache access, each main memory address can be viewed as consisting of three fields: the o LSBs identify a unique word or byte within a memory block ($o = log_2 K$). The next b bits specify part of the block number ($b = log_2 m$). The remaining t bits are saved in the tag ($t = n - b - o$). Again, for the example, $o = log_2 2 = 1$, $b = log_2 4 = 2$, and $t = 4 - 2 - 1 = 1$, as Fig. 5.4 shows.

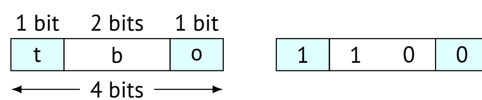


Fig. 5.4 The three fields of the memory addresses for a direct mapped cache. Example with address 1100_2, which is mapped to cache line 10_2 and whose tag is 1.

The direct mapping technique is inexpensive to implement. Its major drawback is that each memory block has a unique cache location. If a program refers words repeatedly from two different blocks that map into the same line, then the blocks need to be repeatedly swapped in the cache. The hit ratio becomes very low and the cache is not able to improve the performance of the program.

The disadvantage of the direct mapping can be overcome by the **fully associative mapping**, which allows each memory block to be loaded into any cache line. Thus, all the words of the block, as usual, and also the block number (i.e., the bit pattern that encodes the block number) must be stored in the cache line.

Fig. 5.5 shows a fully associative mapped cache similar to the one in Fig. 5.3. The major differences are the possibility to map each memory block to any cache line and the bigger size of the tag. In this case, the tag has 3-bits, since the cache line where a block is stored gives no clue about the block number.

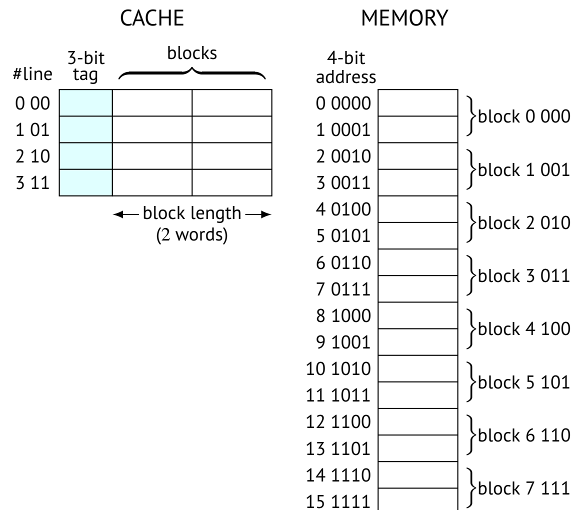


Fig. 5.5 A small memory-cache system with fully associative mapping. All numbers are indicated in decimal and binary.

For the purpose of cache access, each main memory address can be viewed as consisting of two fields: the o LSBs identify a unique word or byte within a memory block ($o = log_2 K$). The remaining t bits are saved in the tag ($t = n - o$). Again, for the example, $o = log_2 2 = 1$ and $t = 4 - 1 = 3$, as Fig. 5.6 shows.

There is a third mapping scheme designated as **set associative mapping**, which is a combination of the direct and the fully associative mappings. It is similar to direct mapping, since the address is used to map the memory block to a certain cache location. The major difference here is that instead of mapping to a single cache block, an address maps to a set of cache blocks. The sets have all the same size. For example, in a 2-way set associative cache ($N = 2$), there are two cache blocks per set, as Fig. 5.7 depicts. In this case, there are

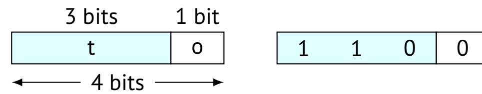


Fig. 5.6 The two fields of the memory addresses for a fully associative mapped cache. Example with address 1100_2, which can be mapped to any cache line and whose tag is 110_2.

only two cache sets ($m = 2$), so the cache is able to store four 2-word memory blocks. Direct mapped cache can be seen as a special case of N-way set associative cache mapping where the set size is one.

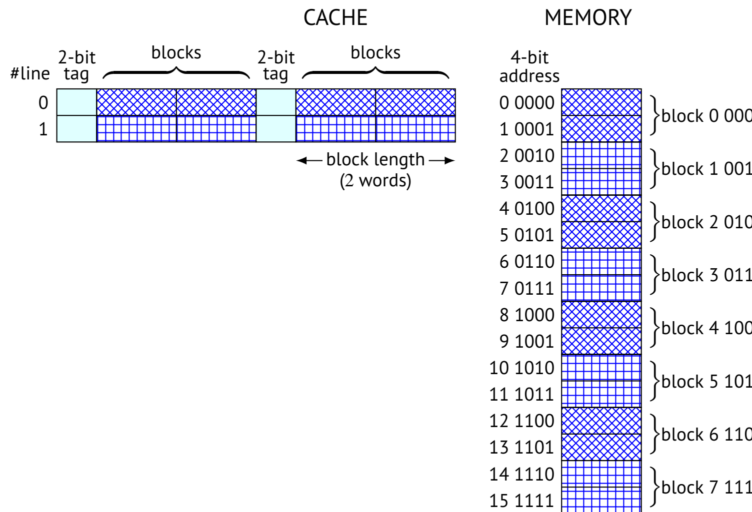


Fig. 5.7 A small memory-cache system with 2-way set associative mapping. All numbers are indicated in decimal and binary.

The 2-way set associative mapping function can be implemented using the main memory address. For the purpose of cache access, each main memory address can be viewed as consisting of three fields: the o LSBs identify a unique word or byte within a memory block ($o = log_2 K$). The next s bits specify the set number ($s = log_2 m$). The remaining t bits are saved in the tag ($t = n - s - o$). Again, for the example, $o = log_2 2 = 1$, $b = log_2 2 = 1$, and $t = 4 - 1 - 1 = 2$, as shown in Fig. 5.8.

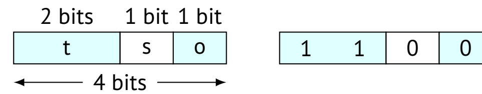


Fig. 5.8 The three fields of the memory addresses for a 2-way set associative mapped cache. Example with address 1100_2, which is mapped to cache line 0 and whose tag is 11_2.

5.3 Replacement algorithm

In a direct-mapped cache, if there is contention for a cache block, there is only one possible action: the existing block is removed from the cache to make room for the new block. This process is called replacement. With direct mapping, there is no need for a sophisticated replacement algorithm, since the location for each new block is fixed. Anyhow, with direct mapping, if a block already occupies the cache location where a new block is about to be placed, the block currently in the cache should be updated in the main memory if it has been modified after being loaded into the cache.

For associative mappings, whenever the cache is full and a new block needs be stored, a **replacement algorithm** is needed. It decides in which cache line to store the new block. Many alternatives exist for this algorithm.

If temporal locality is considered, it is likely that any value that has not been used recently will not be needed again soon. If the last time each cache line was accessed is registered, it is possible to select for replacement the line that has been used least recently. This is the least recently used (LRU) algorithm, which requires the system to keep a history of accesses for every cache line. Another possible approach is the first in first out (FIFO) strategy. The block that has been in cache the longest is selected as the one to be replaced by the new block. Another alternative is to randomly select the cache line to be replaced.

Other possibilities are the not most-recently used (NMRU) that randomly choose among all blocks but the one most recently referenced, and the least frequently used (LFU) that replaces the block that has been referenced the fewest number of times. The algorithm selected by the computer architects often depends on how the system is to be used. This selection is not trivial as no single algorithm is the best for all contexts.

Exercises

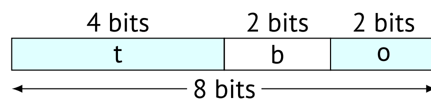
Note: In all exercises in this chapter, assume that a word is four bytes and that the memory is addressed at the byte level.

Exerc. 5.1: Consider four computers with different caches:

- **C1**: direct mapping, 2^{20} words of main memory, cache with 32 blocks, cache block with 16 words.
- **C2**: direct mapping, 2^{32} bytes of main memory, cache with 1024 blocks, cache block with 32 words.

- **C3**: fully associative mapping, 2^{16} words of main memory, cache with 64 blocks, cache block with 32 words.
- **C4**: fully associative mapping, 2^{24} words of main memory, cache with 128 blocks, cache block with 64 words.

(a) Calculate the number of blocks that exist in the main memory.
(b) Draw the format of a memory address as seen by each cache, indicating the sizes of the tag, block (when applicable), and offset fields?
(c) Indicate the cache block to where is mapped the memory reference $3DB63_{16}$ in C1 and $13463FA_{16}$ in C2. Specify in each case the tag value.
(d) Calculate the size in bytes of each cache.
Exerc. 5.2: Consider a computer with a memory with 128Mi words. Blocks are 64 words in length and the cache consists of 32Ki blocks. For a 2-way set associative cache mapping scheme, illustrate the format for a main memory address, including the fields and their sizes.
Exerc. 5.3: A 2-way set associative cache consists of four sets. The main memory contains 2Ki 8-word blocks.
(a) Show the main memory address format (including the fields and their sizes) that allows mapping the addresses from main memory to the cache.
(b) Compute the hit ratio for a program that loops three times from locations 8 to 55 in main memory. Assume that all instructions occupy four bytes.
Exerc. 5.4: A computer, using a set associative cache, has 2^{16} words of main memory and a cache of 32 blocks, and each cache block contains eight words.
(a) What is the format of a memory address as seen by a 2-way set associative cache, i.e., what are the sizes of the tag, set, and word fields?
(b) Repeat the previous question if the cache is 4-way set associative.
Exerc. 5.5: A computer uses a memory address word size of 8 bits. This computer has a 16-byte direct-mapped cache with 4 bytes per block. The format of a memory address as seen by the cache is the following:



While running a program, the computer accesses several memory locations, according to the following sequence: 6D, B9, E3, 16, E3, 4E, 4F, 14, 91, A4, A5, A7, A9, 98, and 99 (in hexadecimal). The memory addresses of the first four accesses have been loaded into the cache blocks as shown below. The contents of the tag are shown in binary and the cache contents are simply the hexadecimal addresses whose contents are stored at each cache location.

block	tag				
0	1110	(E3)	(E2)	(E1)	(E0)
1	0001	(17)	(16)	(15)	(14)
2	1011	(BB)	(BA)	(B9)	(B8)
3	0110	(6F)	(6E)	(6D)	(6C)

(a) What is the hit ratio for the memory reference sequence given above?
(b) What memory blocks are in the cache after the last address has been accessed?
Exerc. 5.6: Consider a byte-addressable computer with 24-bit addresses, a cache capable of storing a total of 64KiB of data, and blocks of 32 bytes. Show the format of a 24-bit memory address for the following mapping functions: **(a)** direct, **(b)** fully associative, and **(c)** 16-way set associative.

Further reading

Caches constitute a topic that is addressed by all books on computer architecture, namely by those written by Tanenbaum and Austin (2013, Chapter 2), Patterson and Hennessy (2014, Chapter 5), Bryant and O'Hallaron (2016, Chapter 6), and Stallings (2019, Chapter 5). Another interesting material about caches is provided by Null and Lobur (2003, Chapter 6). In particular, they present non-computer examples of caching, which are quite enlightening to understand its major principles. A very complete and detail survey about caches authored by Smith (1982) is also worth reading.

Chapter 6
Code optimisations

Abstract The primary objective in writing a computer program is to make it work correctly. But programs should also address quality attributes (also designated non-functional requirements), namely performance, size and maintainability. There are situations where, for example, it is relevant to reduce the time a program takes to execute. This chapter explores how to make programs execute faster through several different types of program optimisation.

6.1 Main principles

Writing an efficient program in a given high-level programming language requires several types of skills. First, one must select the adequate algorithms and data structures. Second, source code must be in such a condition that the compiler can effectively produce code that runs as fast as possible or that occupies the minimum space. Speed and size are the traditional criteria of optimisation addressed by compilers, but energy consumption is gaining importance due to environmental reasons and the popularisation of mobile devices that are not always connected to the electric grid. It is relevant to comprehend the capabilities and limitations of the compilers, when they try to apply some type of **code optimisation** to the programs. The term “optimisation” is somehow abusive as there is no guarantee that the result is the best possible (i.e., the optimum). Often, minor changes in the source program can have a profound impact in the quality of the machine code generated by the compiler. It is important that programmers know these differences, since this knowledge may allow programs to be written in ways that make them more amenable for compilers to generate (more) efficient code.

In approaching program development and optimisation, some aspects must be taken into account, like the way the code is expected to be used and the critical factors affecting it. In general, programmers must establish a trade-off between several criteria, like performance, size, portability, and readability. These criteria are non-functional requirements and each one cannot be achieved in an isolated way. This means that one cannot maximise a given non-functional requirement without sacrificing some other non-functional requirements.

The selected level for the satisfaction of a given non-functional requirement affects, either positively or negatively, the satisfaction of other non-functional requirements. For example, a program optimised for performance can see a reduction in its characteristics associated with maintainability. In a simple way, one can say that the fastest program is probably written in a very cryptic way and is difficult to change (and to maintain). Traditionally, C compilers, namely `gcc`, only handle size and speed. A new criterion, energy consumption, gained importance in the last decades.

At the algorithmic level, there are differences between a normal algorithm and a more efficient one. Usually, the former can be programmed in a matter of minutes, while the latter requires more effort to implement and refine. At the coding level, many low-level optimisations tend to reduce the readability and the modularity of the program, making harder its modification.

For code that is expected to execute repeatedly and whose performance is relevant, applying optimisations is worthwhile. However, it is important to maintain some level of readability in the code. A compiler is able to take a valid program written in the source language code and generate a behaviourally-equivalent machine-level program. Compilers make use of sophisticated mechanisms to generate code that can be optimised according to different metrics.

6.2 Limitations of the compilers

Most compilers employ advanced techniques to determine what values are computed in a program and how they are used. The compilers can exploit opportunities to simplify expressions, to use a single computation in several different places, and to reduce the number of times a given computation must be performed. Most compilers allow programmers to control the level of optimisations. For example, `gcc` can be used with the option `-O` to specify which type of optimisations to apply.

Compilers must be careful to apply only safe optimisations to a program, which means that the resulting program must have the same external behaviour as an unoptimised version for all possible paths the program may take. A compiler is expected to be conservative in applying optimisations in the sense that, whenever in doubt, it does not go ahead with them. The compiler operates in a constrained context. Firstly, it must not modify the behaviour of the program under any possible condition. Often, this restriction prevents the compiler from applying optimisations that would only affect behaviour under illogical, but possible, conditions. Secondly, the compiler has a limited and localised view of the program, so broader optimisations are not applied. Thirdly, the compilation process itself must be fast enough, so (marginal) gains are not appreciated if the compiler needs a large amount of time to achieve them.

Compilers usually have problems in dealing with any **optimisation blocker**, which is an aspect of computer programs that can greatly limit the opportunities for a compiler to generate optimised code. Often, the optimisation blockers are dependent on the execution environment. An example of an optimisation blocker is present in the following two implementations of the same procedure:

```
int pr1(int *x, int *y) {        int pr2(int *x, int *y) {
   *x += *y;                        *x += 2* *y;
   *x += *y;                     }
}
```

It seems safe to transform the first version of the procedure into the second one, as both apparently have identical behaviour. They both add twice the value stored at the location designated by pointer `y` to that designated by pointer `x`. Procedure `pr2` is more efficient, since it requires only three memory references (read `*x`, read `*y`, write `*x`), while `pr1` requires six (read twice `*x`, read twice `*y`, and write twice `*x`). So, it seems that a compiler when handling procedure `pr1` could generate more efficient code based on the computations performed by the equivalent `pr2`. This approach cannot however be applied when `x` and `y` are equal. Then procedure `pr1` performs the following computations:

```
*x += *x;          /* double value at x */
*x += *x;          /* double value at x */
```

The value at `x` is thus increased by a factor of four. However, procedure `pr2` executes the following computation:

```
*x += 2* *x;       /* triple value at x */
```

The result is that the value at `x` is increased by a factor of three. The compiler cannot assume that arguments `x` and `y` are not equal. Therefore, the compiler cannot generate code in the style of `pr2` as an optimised version of `pr1`. The situation where two pointers may designate the same memory location is called as **memory aliasing**. Compilers tend to assume that different pointers may be aliased.

Another optimisation blocker may occur when a function has side effects. A function (or expression) has a **side effect**, if it alters the values of some variables outside its local context. So, the function has an observable effect besides returning a value to the calling procedure. Consider the two following versions of a function, which appear to have the same behaviour:

```
int func1(int x) {             int func2(int x) {
   return (f(x)+f(x));            return ( 2*f(x) );
}                              }
```

However, if the function `f` has the following implementation and `counter` is a global integer variable, then the two versions of the function are not equivalent.

```
int f(int p) {
   return (p+counter++);
}
```

Note, that the expression `p+counter++`, calculates the value `p+counter` and afterwards increments the value of `counter` by one. If `counter` is equal to 0, calling `func1(5)` returns 11 $(5+6)$, while calling `func2(5)` returns 10 (2×5). The value of counter is also different in both cases: 2 after calling `func1` and 1 after `func2`.

Another example is the following procedure `lower2upper` that converts the lowercase letters of a string into uppercase ones. It seems that the expression `strlen(s)` can be

safely calculated outside the loop just once, avoiding thus several calls to the `strlen` function.

```
void lower2upper(char *s) {
   int i;
   for (i=0; i<strlen(s); i++)
      if (s[i]>='a' && s[i]<='z')
         s[i] -= ('a' - 'A');
}
```

```
void lower2upper(char *s) {
   int i, length=strlen(s);
   for (i=0; i< length ; i++)
      if (s[i]>='a' && s[i]<='z')
         s[i] -= ('a' - 'A');
}
```

Compilers tend to keep the call to `strlen` inside the loop, because it can have side effects. Additionally, the loop body may change the string and its size. Compilers usually treat procedures as black boxes that cannot be analysed. Following a conservative approach, compilers assume the worst case and the function call remains intact. In this case, the compiler may actually apply the optimisation, as long as `strlen` is recognised as a built-in function.

These examples, which just cover some dimensions of the issues associated with code optimisation, show that it is often important that programmers know how to write code that can be easily optimised by a compiler. The optimisations can be divided into two major groups:

1. machine-independent optimisations;
2. machine-dependent optimisations;

In the two next sections, some of the approaches and techniques applied in each of these two groups are presented and discussed. The discussion presents fragments of assembly code, which in some cases were generated by the `gcc` compiler. The examples serve to show how to examine assembly code and relate it to the high-level program languages. The reader may also try to obtain fragments of assembly code with a C compiler, but most likely they will be different from the ones presented here. Anyhow, they are behaviourally equivalent. The analysis made here may require some adaptation for the examples obtained by the reader.

6.3 Machine-independent optimisations

Machine-independent optimisations improve the target code without taking into account any properties at the machine level. They include, for example, choosing the best (i.e., the fastest) algorithm for the problem at hand. This issue is not addressed in this book, as it is outside its scope. Other optimisations that fit in this group are:

1. code motion;
2. elimination of unnecessary accesses to memory;
3. loop unrolling;
4. reduction of the number of procedure calls.

The next subsections explain these four optimisations based on illustrative examples.

6.3.1 Code motion

Loops are good candidates for improvements, as they have an iterative nature. The most typical cases of code motion consists of statements in a loop or some of their parts (like, expressions) that can be moved outside its body, without affecting its semantics. The idea here is that moving statements from the inside to the outside of the loop obviously reduces the execution time of the program. In the piece of C code on the left, two optimisations are possible, which results in the code on the right:

```
do {
   x=y+z;
   a[i]=i+x*x;
   i++;
} while (i<n);
```

```
x=y+z;
int const j=x*x;
do {
   a[i]=i+ j ;
   i++;
} while (i<n);
```

First, the statement that calculates the values of `x` does not depend on variables that are modified inside the `DO-WHILE` loop body, so it can be moved to the outside. Additionally, the value of `x*x` just needs to be calculated once (outside the loop) and can be reused in each iteration of the loop. The use of the `const` qualifier indicates that `j` is read-only.

Similar transformations can be applied to `WHILE-DO` loops, but in some cases the result is more complex. A `WHILE-DO` loop starts by testing the condition, while a `DO-WHILE` loop initiates by executing the loop body. The former executes the loop body zero or more times, while the latter executes the loop body at least once. In the piece of C code in the right side, the need of an `IF-THEN` statement prevents the loop to execute if the loop condition fails in the very first iteration. In that case, the loop body is not executed.

```
while (i<n) {
   x=y+z;
   a[i]=i+x*x;
   i++;
}
```

```
if (i < n) {
   x=y+z;
   int const j=x*x;
   do {
      a[i]=i+ j ;
      i++;
   } while (i<n);
}
```

These two examples justify why compilers try to transform `WHILE-DO` (and `FOR`) loops in equivalent `DO-WHILE` loops, to avoid the `IF-THEN` statement. For example, the following two blocks have equivalent behaviour, since it is guaranteed that the `FOR` loop executes at least once (in this case, `i` equals 0 in the beginning and the loop executes while it is smaller than 100):

```
for (i=0; i<100; i++) {
   ...
}
```

```
i=0;
do {
   ...
   i++;
} while (i<100);
```

The assembly code for DO-WHILE loops, in general, can be made much simpler (i.e., faster) than the equivalent versions using WHILE-DO and FOR loops, as already discussed in Subsection 4.2.6.

6.3.2 Elimination of unnecessary accesses to memory

Accesses to main memory, which are significantly slower than those to registers, are also obvious candidates to optimise the performance of a program. Consider the following procedure:

```
int addAll (int *a, int *value) {
   int i;
   *value=0;
   for (i=0; i<100; i++)
      *value += a[i];
}
```

The addition inside the for loop accesses the argument value, which was passed as a pointer. So, in general, this argument is in the memory as occurs in the following piece of assembly code:

```
addAll:
   ...
   movl 12(%ebp), %edx
   movl 8(%ebp), %esi
   movl $0, (%edx)
   xorl %ecx, %ecx
LBB0:
   movl (%esi,%ecx,4), %eax
   addl %eax, (%edx)
   incl %ecx
   cmpl $100, %ecx
   jne LBB0
   ...
```

There is here an opportunity for improvement, since there are two accesses to the memory for each loop iteration: one to read the value of the respective array element (movl (%esi,%ecx,4), %eax) and another one to accumulate it into the value argument (addl %eax, (%edx)). It is possible to just write this value to memory at the end of the loop. As the next C program shows, the optimisation is based on accumulating the sum of all elements of the array arr into a local variable and on assigning it to a register.

```
int addAll (int *arr, int *value) {
   int i;
   int acc=0;
   for (i=0; i<100; i++)
      acc += arr[i];
   *value=acc;
}
```

The assembly code for this new version of the procedure shows that there is only one access to the memory in each loop iteration (`addl (%esi,%ecx,4), %eax`). This is a relevant improvement as memory accesses are much slower than accesses to registers.

```
addAll:
   ...
   movl 12(%ebp), %edx
   movl 8(%ebp), %esi
   movl $0, %eax
   xorl %ecx, %ecx
LBB0:
   addl (%esi,%ecx,4), %eax
   incl %ecx
   cmpl $100, %ecx
   jne LBB0
   movl %eax, (%edx)
   ...
```

6.3.3 Loop unrolling

A well-known code optimisation technique is **loop unrolling**, which aims reducing the number of iterations for a loop, by increasing the number of elements computed on each iteration. The rationale behind this technique is the fact that each loop iteration incurs in some non-effective computations, related to the control of the loop (incrementing the loop variable, comparing it and jumping to the beginning of the loop). Thus, by decreasing the number of the operations that do not contribute directly to the program result, the loop is made faster.

It is straightforward to apply loop unrolling to array processing loops, whenever the number of iterations is known at compile time. The next code fragments show three similar loops that iterate through a 120-element array, with different levels of unrolling.

1

```
i=0;
do {
   arr[i]=0;
   i++;
} while (i<120);
```

2

```
i=0;
do {
   arr[i]=0;
   arr[i+1]=0;
   arr[i+2]=0;
   arr[i+3]=0;
   i+=4;
} while (i<120);
```

3

```
arr[0]=0;
arr[1]=0;
arr[2]=0;
...
arr[119]=0;
```

In fragment 1, all iterations include an increment and a comparison/jump. Fragment 2 uses loop unrolling to improve performance, by handling a block of four elements in each iteration. The new loop has to make only 30 iterations, instead of 120. Consequently, only 25% of the jumps and conditional branches need to be taken, which represents a significant decrease in the loop administration overhead. However, the new loop occupies more space, as it includes more C statements (an also more machine instructions). Fragment 3 is the fastest solution, but also the one with the highest number of C instructions (120). In this

case, there is no loop overhead, as there is literally no loop. As these examples illustrate, loop unrolling constitutes a space–time tradeoff, since it attempts to optimise the execution speed of a given program at the expense of its binary size.

The following assembly code blocks are the possible translations of the first two C code fragments presented above.

1

```
   movl $0, %eax
.lbl:
   movl $0, arr(,%eax,4)
   incl %eax
   cmpl $120, %eax
   jle .lbl
```

2

```
   movl $0, %eax
.lbl:
   movl $0, arr(,%eax,4)
   movl $0, arr+4(,%eax,4)
   movl $0, arr+8(,%eax,4)
   movl $0, arr+12(,%eax,4)
   addl $4, %eax
   cmpl $120, %eax
   jle .lbl
```

It is simple to calculate the number of machine instructions that is executed in both versions of the code (for 120 array positions). In fragments 1 and 2, the number of executed instructions is 481 ($1+120\times4$) and 211 ($1+30\times7$), respectively. If one assumes, for simplification purposes, that all instructions take the same time to execute (which is not the case), the reduction in time is around 56%.

Generalising, if the array has n positions and a block of $k \leq n/2$ positions are manipulated in each iteration, the total number of instructions that are executed within the loop is given by $1+(3+k)\times\lfloor n/k\rfloor+(n \mod k)$. Note that if n is not a multiple of k, additional instructions need to be put at the beginning or the end of the loop. For instance, if $n=21$ and $k=4$, there are five iterations with blocks of four elements, plus one instruction outside the loop to handle the remaining element. Note also that it does not make sense to have $k > n/2$, because in those situations the loop is useless, as it is just executed once. Fig. 6.1 shows the number of instructions that need to be executed considering different values for the number of elements of the 120-element array that are handled in each loop iteration. The performance improves whenever the block has more elements, but the code also includes a higher number of machine instructions (it occupies more space in memory).

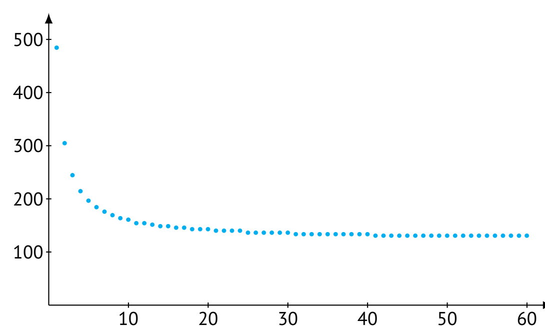


Fig. 6.1 Loop unrolling metrics for a loop that initialises the 120 elements of an array. The XX axis represents the number of elements handled in each loop iteration, while the YY axis represents the total number of instructions needed to run the loop.

6.3.4 Reduction of the number of procedure calls

Procedure calls imply a great overhead (pushing arguments, saving return address, jumping to procedure, managing stack frame, passing result) and also tend to block many possible program optimisations. The idea of replacing a procedure call by the body of the called procedure is called **inline expansion** (or inlining). This optimisation reduces time, at the cost of increasing the space usage. The code expansion due to the replication of the procedure body dominates, except for simple cases.

An inlined procedure run faster than the normal procedure, as the calling overheads are avoided. However, there is a memory penalty, i.e., code gets larger. If a procedure is inlined n times, n copies of the function body are inserted into the code. Maintenance of the procedure gets harder and more error prone, because when the body of the procedure needs to be changed, one must update it in n places. Thus, inlining is adequate for small functions.

Consider the following C function that returns 1 if the argument `num` is even, and 0 otherwise (if it is odd):

```
int isEven (int num) {
   return !(num & 1);
}
```

Assume that this function is invoked in some part of a program:

```
if (isEven (number))
```

A possible optimisation of this code is to replace the call to function `isEven` with the respective body.

```
if (!(number & 1))
```

6.4 Machine-dependent optimisations

Machine-dependent optimisations, as the name suggests, depend on the specific processor that is being considered. Optimisations that work in a given processor are not necessarily effective in a different one.

Optimisations that fit in this category are related to the use of instructions that are faster or occupy less space in memory. When a given high-level statement or construct can be supported by different machine-level alternatives, it is possible to use the best one, according to the preferred criterion. For example, in IA32 to assign the value 0 to a register can be done with any of the two following alternatives:

```
movl $0, %eax                    xorl %eax, %eax
```

The `movl` instruction occupies more space in memory, as it includes a 32-bit immediate value. So, compilers usually select the `xorl` alternative.

Another typical example occurs with multiplications, which in some processors take much longer to execute that additions and shifts. So, whenever compilers encounter a multiplication involving a variable and a constant, they transform the multiplication in a series of additions and shifts. The simple situations occur when the constant is a power of two. Consider the following piece of C code:

```
y = 8*a;
```

Multiplying by 8 is equivalent to shift the bits 3 positions to the left. So, this multiplication can be optimally supported by using the following instructions, which assume variables `a` and `y` to be saved in registers eax and ebx, respectively:

```
movl %eax, %ebx
sall 3, %ebx
```

Things are not that simple when the constant is not a power of two. Consider now the following piece of C code:

```
y = 37*a;
```

A straightforward approach to tackle this C statement is to use the three-operand multiplication in IA32:

```
imull $37, %eax, %ebx
```

It is also possible to use additions and shifts to calculate this multiplication. Since $37 = 32+4+1$, calculating $37 \cdot a$ is equivalent to compute $32 \cdot a + 4 \cdot a + a$. So, this multiplication can be optimally supported by the following IA32 instructions:

```
movl %eax, %ecx     /* ecx = a */
movl %ecx, %ebx     /* ebx = a */
sall 2, %ecx        /* eax = 4xa */
addl %ecx, %ebx     /* ebx = a + 4xa = 5xa */
sall 3, %ecx        /* eax = 8x4a = 32xa */
addl %ecx, %ebx     /* ebx = 5xa + 32xa = 37xa */
```

The efficient computation of addresses is another machine-dependent optimisation. Again, assume the C code discussed in Subsection 6.3.3 that initiates the values of a 120-element array and particularly the execution of the following statement

```
arr[i]=0;
```

If `i` is saved in register ecx and the initial address of the array `arr` is in register ebx, then the two following alternatives for the assembly code are possible. The left-sided solution is the preferred one, as it is both faster and shorter. It benefits from the rich addressing modes available in the IA32 instruction set. The right-sided solution calculates the address outside the `movl` instruction.

```
movl $0, (%ebx,%ecx,4)
```

```
movl %ecx, %edx
sall 2, %edx
addl %edx, %ebx
movl $0, (%ebx)
```

6.5 Cache-oriented optimisations

A program that exhibits good locality is very likely to execute faster than one that does not. Good locality is possible if a program refers data items that are near other recently referenced data items or that were recently referenced themselves. This situation may have a great impact on the performance of the program, because the hit rate of the cache gets higher. Therefore, programmers must understand the principle of locality in order to positively exploit it. In fact, this principle is present in all levels of modern computer systems, from the hardware, to the operating system, to the software application. For example, web browsers explore temporal locality by locally caching recently referenced documents.

Consider a slightly different version of the `addAll` procedure, which accesses and sums all the elements of an array.

```
int addAll (int *arr) {
   int i, acc=0;
   for (i=0; i<100; i++)
      acc += arr[i];
   return (acc);
}
```

This procedure has a good temporal locality with respect to local variables `i` and `acc`, which are accessed in every loop iteration. The elements of array `arr` are accessed one after the other, in the same order as they are in the memory. So, the procedure has good spatial locality with respect to array `arr`. Overall, the `addAll` procedure exhibits good locality. It has a stride-1 reference pattern, as it accesses each element of the array sequentially. The **stride** of an array is the number of locations in memory between successive array elements, measured in bytes or in units of the size of the array's elements. In general, as the stride increases, the spatial locality decreases, since two element arrays consecutively accessed are more distant.

To highlight the importance of writing code that takes into account the existence of cache memories, consider the two following pieces of code that add all the elements of a two-dimensional matrix `arr`, where all variables are of type `int`:

1
```
for (i=0; i<100; i++)
   for (j=0; j<100; j++)
      acc += arr[i][j];
```

2
```
for (i=0; i<100; i++)
   for (j=0; j<100; j++)
      acc += arr [j][i] ;
```

Fig. 6.2 shows the access order of the array elements for both pieces of code. It is observable in Fig. 6.2(a) that for the first piece of code the elements are accessed sequentially. For the second one, the access is not sequential. In fact, the stride is 100 units (or 400 bytes). With respect to the array, the conclusion is that the first piece of code presents a good spatial locality, while the second one does not.

Exercises

Exerc. 6.1: Consider again the two C programs in Exercise 4.5 and the one in Exercise 4.6. Generate new versions of the assembly code for those programs, with the `-O1`, `-O2` `-O3`

(a)		(b)
1	[0][0]	1
2	[0][1]	101
3	[0][2]	201
...		...
99	[0][98]	9801
100	[0][99]	9901
101	[1][0]	2
102	[1][1]	102
...		...
200	[1][99]	9902
201	[2][0]	3
...		...
10 000	[99][99]	10 000

Fig. 6.2 A 100x100 array in memory and the access order according to different programs.

`-Os` compilation flags. For each C program, compare the new versions of the assembly code with the one obtained with the `-O0` compilation flag.

Exerc. 6.2: Consider again the C programs in Exercises 4.7, 4.8, and 4.9 and analyse the assembly code generated with different levels of optimisation.

Exerc. 6.3: For each block of code, calculate the stride associated with the access to array `vec`.

```
#define size 500
typedef struct {
   int a[2];
   int b[2];
} rec;
rec vec[size];
```

1

```
for (i=0; i<size; i++)
  for (j=0; j<2; j++) {
     vec[i].a[j]=0;
     vec[i].b[j]=0;
  }
```

2

```
for (i=0; i<size; i++)
   for (j=0; j<2; j++)
      vec[i].a[j]=0;
   for (j=0; j<2; j++)
      vec[i].b[j]=0;
```

3

```
for (j=0; j<2; j++)
  for (i=0; i<size; i++)
     vec[i].a[j]=0;
  for (i=0; i<size; i++)
     vec[i].b[j]=0;
```

Exerc. 6.4: Change the order of the index variables `i`, `j`, and `k` in the `if` statement, so that the procedure accesses array `arr` with a stride-1 reference pattern.

```
int countNulls(int arr[N][N][N]) {
   int i, j, k, count = 1;
   for (j=N-1; i>=0; i-)
      for (k=N-1; j>=0; j-)
         for (i=N-1; k>=0; k-)
            if (arr[k][i][j]==0)
               count++;
   return count;
}
```

Further reading

The topic of program optimisation is too large, so in this chapter only a reduced number of techniques were touched on. The interested reader is pointed to the landmark book on compilers, written by Aho et al. (1986, chapters 9 and 10), to learn more about code optimisation.

Bryant and O'Hallaron (2016, Chapter 5) present many relevant aspects related to the optimisation of assembly code, obtained by C compilers. Blum (2005, Chapter 15) discusses common methods to optimise assembly code, including calculations, variables, loops, and conditional branches.

Solutions to exercises

1.3: Memory cells 132104 to 132107 are affected. Values 00010010, 00110100, 01010110, and 01111000 are stored in cells 132104, 132105, 132106, and 132107, respectively. **1.4**: 0000 1111 0000 0000 1111 1111 0001 0010. **1.5**: Generically, this accident is not positive. The program was modified and no longer will behave exactly as programmed. Depending on the location of the modified instruction, the program may terminate abruptly or may show a strange behaviour. **1.6**: (a): 120 000 s; (b) 35 085 s.

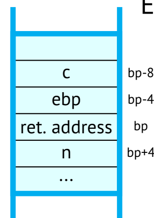
2.1: The encoding is valid; DLXII. **2.2**: "U. Minho". **2.3**: $49,152$ bytes. **2.4**: 1111_2 and 15_{10}. **2.5**: $62^4 > 1.4 \cdot 10^7$. **2.6**: $126^3 > 2 \cdot 10^6$. **2.7**: 6KB; 5.86 KiB. **2.8**: 12 bits. **2.9**: (a) $63,151,360$ B; (b) 21.982 MiB; (c) 1.28 PiB.

3.1: minimum 0.01 EUR; maximum $99,999,999.99$ EUR. **3.2**: (a) 10000011_2; (b) 111111111_2; (c) 1101111000_2; (d) 1000000000000_2. **3.3**: (a) 31; (b) 1023; (c) 262,143; (d) 4,294,967,295. **3.4**: $0(0000_2), 1(0001_2), 2(0010_2), 3(0011_2), 4(0100_2), 5(0101_2), 6(0110_2), 7(0111_2), 8(1000_2),$ $9(1001_2)$, A(1010_2), B(1011_2), and C(1100_2). **3.5**: (a) BED$_{16}$; (b) 276_{16}; (c) 7FF$_{16}$; (d) 51E$_{16}$. **3.6**: (a) 1011111011101111_2; (b) 1000000000000.11111111_2; (c) $101010111100.110111101111_2$; (d) 110110101100.001101_2. **3.7**: (a) 302_5; (b) 1011_5; (c) 4021_5; (d) $13,000_5$. **3.8**: (a) 60; (b) 102; (c) 259; (d) 648. **3.9**: (a) 1000010_2, 123_7, 73_9,; (b) 319.25_{10}, 227.3_{12}; (c) $11020.(1)_3$, 1302.2_4, $222.(3)_7$; (d) 11100110000_2, 12304_6, 3460_8. **3.10**: This value corresponds to 2^{30}, which means that this computer has a memory that is addressed with 30 bits. **3.11**: 9 hours, 25 minutes, and 46 seconds. **3.12**: 3000000_{16}. **3.13**: 16 bits. **3.14**: 17 bits; 16 bits. **3.15**: 950 bits. **3.16**: $3 \cdot 2^{20} > 3 \cdot 10^6$. **3.17**: A 32-bit signed integer variable was used to represent the number of views. The maximum value in this representation is 2,147,483,647. If this value is incremented the result is negative. The simplest solution is to change the datatype to 32-bit unsigned integer. YouTube engineers decided to change the datatype to 64-bit signed integer. **3.18**: (a) 9; (b) 4. **3.19**: (a) 1001000_2; (b) 1528_{12}; (c) 1641_{16}. **3.20**: (a) 8764; (b) 9963; (c) $888\ 878$. **3.21**: (a) 1100100_2; (b) 0011001101_2; (c) 11111111_2; (d) 00011111111_2. **3.22**: (a) 00010010_2 in all; (b) 1111001_2 in all; (c) 10100001_2, 11011110_2, 11011111_2; (d) 11100100_2, 10011011_2, 10011100_2. **3.23**: (a) -199; (b) -312; (c) -313; (d) 200. **3.24**: (a) 1011101001_2; (b) 1100010110_2; (c) 1100010111_2; (d) 0100010110_2. **3.25**: (a) 11111_2; (b) 10110_2; (c) 1010010_2; (d) 1. **3.26**: (a) 100100_2; (b) 1100100_2; (c) 1011010100_2; (d) 11000011_2. **3.27**: (a) overflow;

(b) overflow; (c) 11001010; (d) overflow. **3.28**: (a) $(-1)^s \times (1+f) \times 2^{e-7}$, $(-1)^s \times (1+f) \times 2^{e-3}$; (b) i) $0\ 0000\ 001_2 = \frac{1}{512}$, $0\ 000\ 0001_2 = \frac{1}{64}$, ii) $0\ 0000\ 111_2 = \frac{7}{512}$, $0\ 000\ 1111_2 = \frac{15}{64}$, iii) $0\ 0001\ 000_2 = \frac{1}{64}$, $0\ 001\ 0000_2 = \frac{1}{4}$, iv) 0 0111 000, 0 011 0000, v) $0\ 1110\ 111_2$=240, $0\ 110\ 1111_2 = \frac{15}{2}$; (c) i) $-\frac{11}{16}$, ii) NaN, iii)$-\frac{9}{256}$, iv) $\frac{3}{512}$, v) -2.25; (d) i) 1 1010 101, ii) 0 1110 000, iii) 0 1101 110, iv) 1 1111 000 ($-\infty$), v) 0 0000 101; (e) i) 0 010 0110, ii) 1 111 0000, iii) 0 000 0010, iv) 1 101 1100, v) 1 000 0000.

4.1: ebh=2 and ebl=209. **4.2**: (a) i. 110_{16}, ii. AB_{16}, iii. 118_{16}, iv. FF_{16}, v. AB_{16}, vi. 55_{16}, vii. 55_{16}, viii. AB_{16}, ix. 55_{16}; (b) ebx=$11A_{16}$, ecx=100_{16}, edx=-168, M[$11C_{16}$]=1 (4 bytes), edx=2, M[118_{16}]=14_{16} (4 bytes), ebx=$AA0_{16}$, edx=28_{16}. **4.3**: 40F785; 831798. **4.4**: 300812; 854FB8. **4.8**: see below **4.9**: `0`; `x!=y`; `x>y` or `x>=y`; 1; 2; `x>10`; 3.

4.8: (a) Each element occupies four bytes in memory.



5.1: (a) C1: 2^{16}, C2: 2^{25}, C3: 2^{11}, C4: 2^{18}; (b) C1: tag 11 bits, block 5 bits, offset 6 bits, C2: tag 15 bits, block 10 bits, offset 7 bits, C3: tag 16 bits, offset 2 bits, C4: tag 24 bits, offset 2 bits; (c) C1: block 01101_2=13, tag 00001111011_2, C2: $00\ 1100\ 0111_2$=199, tag 000000010011010_2; (d) C1: 2400, C2: $146,432$, C3: 9216, C4: $35,840$. **5.2**: tag 7 bits, set 14 bits, offset 8 bits. **5.3**: (a) tag 9 bits, set 2 bits, offset 5 bits; (b) $\frac{17}{18}$. **5.4**: (a) tag 9 bits, set 4 bits, offset 5 bits; (b) tag 10 bits, set 3 bits, offset 5 bits. **5.5**: a) $\frac{2}{5}$; (b) see below. **5.6**: (a) tag 8 bits, block 11 bits, offset 5 bits; (b) tag 19 bits, offset 5 bits; (c) tag 15 bits, set 4 bits, offset 5 bits. **5.5**: (b)

1001	(93)	(92)	(91)	(90)
1010	(A7)	(A6)	(A5)	(A4)
1001	(9B)	(9A)	(99)	(98)
0100	(4F)	(4E)	(4D)	(4C)

6.3: 2 units (8 bytes); 1 unit (4 bytes); 4 units (16 bytes); **6.4**: `arr[j][k][i]`.

Glossary

absolute jump
branch instruction whose possible target is given by a complete memory address; the same as long jump. 62

address
location in memory of an operand. 7

arithmetic logic unit (ALU)
combinational digital circuit that performs arithmetic and logical operations in a CPU. 6

ASCII
character encoding standard for electronic communication. 18

assembler
computer program that translates programs written in assembly language into machine language. 3

assembly phase
process undertaken to convert an assembly language into an object program. 49

binary-coded decimal (BCD)
class of binary encodings of decimal numbers in which each decimal digit is represented by a specific binary pattern with a fixed number of bits. 41

bias
integer value that is subtracted to the natural value of a bit pattern to obtain its corresponding value in an excess representation; the same as offset. 33

big endian
order of bytes of a word in computer memory, which stores the least-significant byte at the largest address; see also little endian. 8

binary point
sign used to separate the integer part of a binary number from its fractional part. 26

bit
basic unit of information that represents a logical state with one of two possible values; the same as binary digit. 16

branch instruction
instruction in a program that can cause a computer to begin the execution of a different instruction sequence, thus deviating from its default behaviour of executing instructions in sequential order. 12

bus
communication system that transfers data between components inside a computer or between computers. 5

byte
unit of digital information that consists of eight bits. 17

cache block
minimum unit of transfer between the cache and the main memory. 77

cache hit
successful attempt to read or write a piece of data in the cache. 76

cache line
portion of the cache memory that can store one block. 77

cache memory
high-speed memory used to store frequently accessed or recently accessed data and instructions, so that future requests for that data/instructions can be provided faster. 75

cache miss
request to access data from the cache that is not filled because the data are not present in the cache, which implies an access to the main memory. 76

callee-saved register
processor register whose value must be preserved by the callee procedure. 65

caller-saved register
processor register whose value must be preserved by the caller procedure. 65

cell
smallest addressable unit of the main memory. 7

central processing unit (CPU)
component of a computer that orchestrate the execution of the instructions that belong to a program, by controlling all the other parts; the same as processor. 5

clock
signal that regulates the passage of time within the computer. 6

code optimisation
transformation technique that tries to improve the code of a program according to a given criterion, like speed, size, or energy consumption. 85

colour depth
number of bits used to indicate the colour (or the colour component) of a single pixel. 22

compilation phase
process undertaken to convert a program written in high-level programming language into an equivalent assembly-language program; process that converts a high-level program into an executable program that the computer can directly execute. 49

computer
machine that can be programmed to carry out sequences of operations automatically. 1

condition code
bit that saves a given post-operation condition (e.g., an arithmetic overflow), which can be read to decide subsequent instructions, namely conditional jumps. 59

content-addressable memory (CAM)
computer memory that accesses the stored information based on input data (either completely or in part), rather than on input address. 78

control unit
component of a CPU that directs the operation of the computer. 6

data dependency
situation in a pipelined CPU in which the result of one instruction, not yet available, is also the operand of a subsequent instruction. 12

decimal point
sign used to separate the integer part of a decimal number from its fractional part in a positional numeral system. 26

device driver
software executed by the CPU when it has to deal with the respective device. 10

digit
symbol used alone or in combinations to represent numbers in a positional numeral system. 25

digital abstraction
process by which analog (or continuous) behaviour is ignored and is rather seen as discrete. 16

direct mapping
cache mapping scheme that maps blocks of memory to blocks in cache using a modular approach. 78

dynamic random access memory (DRAM)
type of RAM that stores each bit of data in a memory cell consisting of a tiny capacitor and a transistor, both typically based on metal-oxide-semiconductor (MOS) technology. 7

end carry-around
type of carry required when two integers, represented as a radix-minus-one complement numbers, are summed. 32

excess-b representation
digital integer coding representation, where all-zero corresponds to the minimal value and all-one to the maximal value; the same as offset binary and biased representation. 33

exponent
part of a number expressed in a scientific notation that indicates the integer value that the base should be raised to, in order to obtain the desired value. 37

fetch-decode-execute cycle
set of steps a computer follows to execute an instruction. 6

fixed-point number
number represented in a numeral system with a radix point in a fixed position. 36

floating-point number
non-integer fractional number represented as a product of a mantissa with an integer power of the base; the same as number in scientific notation. 36

fully associative mapping
cache mapping scheme that maps blocks of main memory to any block in the cache. 80

hexadecimal numeral system
base-16 numeral system that uses the digits 0 to F. 27

hit rate
fraction of memory references that are found in the cache during the execution of (a part of) a program. 76

hit time
time to deliver a word stored in the cache to the CPU. 76

IEEE 745 floating-point standard
technical standard for floating-point arithmetic originally established by the Institute of Electrical and Electronics Engineers (IEEE) in 1985; significantly revised by IEEE 754-2008, published in August 2008. 40

image resolution
number of pixels per space unit (traditionally an inch) of an image. 22

inline expansion
code optimisation technique that substitutes a procedure call with the body of the called procedure; the same as inlining. 93

instruction pointer (IP)
special-purpose register of a processor that indicates where a computer is in its program sequence; the same as program counter. 6

instruction register (IR)
special-purpose register of a processor that stores the instruction currently being executed or decoded. 6

interrupt
signal sent by a device to the CPU to request its attention, thus interrupting the normal fetch-decode-execute cycle of execution. 10

I/O device
hardware system used by humans or systems to communicate with a computer. 9

label
sequence of characters (string) that identifies a specific location within the source code. 61

latency
number of clock cycles needed to complete the execution of an instruction. 11

least-significant bit (LSB)
bit of a binary number located at the far right of the string. 27

least-significant digit (LSD)
digit of a number located at the far right of the string. 27

line size
number of data bytes contained in a cache line, excluding the tag and the control bits. 77

linking phase
process of collecting and combining various pieces of code and data into a single file that can be loaded into memory and executed. 49

little endian
order of bytes of a word in computer memory, which stores the least-significant byte at the smallest address; see also big endian. 8

loader
program that loads other programs and libraries into memory and prepares them for execution. 49

loop unrolling
code optimisation technique that expands a loop so that each new iteration contains several of the original iterations, thus performing more effective computations per iteration and reducing the control overhead. 91

machine language
programming language that provides little or no abstraction from the instruction set architecture of a computer; the same as low-level programming language. 2

magnitude
absolute value of a number; the same as modulus. 31

main memory
storage system, typically implemented with RAM memory, used to store data and instructions for immediate use in a computer; the same as memory. 7

mantissa
fractional part of a number expressed in a scientific notation; the same as significand. 38

mean access time
metric to analyse the performance of a cache-memory system, based on the cache access time, the miss penalty, and the hit ratio. 78

memory aliasing
situation in which a data location in memory can be accessed through different symbolic names. 87

microprogram
program used to interpret instructions into machine language. 3

miss penalty
time required to process a cache miss, including the replacement of the block in the cache and the delivery of the requested data to the processor. 76

miss rate
fraction of memory references that are not found in the cache during the execution of (a part of) a program. 76

most-significant bit (MSB)
bit of a binary number located at the far left of the string. 27

most-significant digit (MSD)
digit of a number located at the far left of the string. 27

natural number
number used for counting and ordering; the same as non-negative integer or counting number. 26

nibble
group of four bits; half a byte. 17

normal number
non-zero number in a floating-point representation which is within the supported balanced range; the same as normalised number. 37

not a number (NaN)
member of a numeric data type that can be interpreted as a value that is undefined or unrepresentable. 39

numeral system
a mathematical notation for representing numbers of a given set, using digits or other symbols; the same as number system or system of numeration. 25

octal numeral system
base-8 numeral system that uses the digits 0 to 7. 27

one's complement
complement of a n-bit number with respect to $2^n - 1$. 32

one's complement numeral system
numeral system in which a negative number is represented by the inverse of the binary representation of its corresponding positive number. 31

operating system
system software that manages computer hardware, software resources, and provides common services for computer programs. 3

optimisation blocker
aspect of a computer program that can limit the opportunities for a compiler to generate optimised machine-level code. 86

overflow
situation that occurs when the value of an integer number is outside the range that can be represented with a given fixed-sized numeral system (either higher than the maximum or lower than the minimum representable values). 33

pipelining
technique for implementing instruction-level parallelism within a single processor, by dividing instructions into a series of sequential steps performed by different processor units in parallel. 10

pixel
the smallest addressable element in a raster image. 21

polling
the process of synchronously checking the status of an external device by a program. 10

positional numeral system
system for representing numbers by an ordered set of digits, in which the value of each digit depends on its position. 25

preprocessing phase
pre-compilation step in which the text of a high-level program is substituted according to the indicated directives. 48

principle of locality
tendency of a processor to access the same set of memory locations repetitively over a short period of time, when executing a program. 76

procedure
sequence of program instructions that performs a specific task, packaged as a unit, which can be used in programs wherever that particular task should be performed; the same as subprogram, function, method, subroutine, handler. 63

program
collection of instructions, written in a given programming language, that can be executed by a computer to perform a specific task. 1

radix
number of unique digits, including the digit zero, used to represent numbers in a positional numeral system; the same as base. 25

radix point
sign used to separate the integer part of a number from its fractional part in a positional numeral system. 26

random access memory (RAM)
volatile computer memory that stores data at a unique address and can recall that data upon presentation of the complete unique address. 7

range
difference between the largest and the smallest representable numbers in a numeral system. 26

raster image
dot matrix data structure that represents a generally rectangular grid of pixels; the same as bitmap image. 21

register
type of computer memory used to quickly access data and instructions used by the CPU; the same as processor register. 6

relative jump
branch instruction whose possible target is given by an offset relative to the current value stored in the instruction pointer. 62

replacement algorithm
algorithm that decides in which cache line to store a new memory block in an associative-mapped cache that is full; the same as cache replacement policy. 82

resource conflict
situation in which two instructions in a pipelined CPU require the simultaneous access to the same resource. 12

return address
field of the stack frame of a given procedure that indicates where to resume execution after the procedure is completed. 64

sample
value (or set of values) at a point in time of a given signal. 22

sample resolution
number of bits used to digitally represent the value of each sample. 22

sampling
reduction of a continuous-time signal to a discrete-time signal. 22

sampling rate
number of samples taken per second (measured in Hertz). 23

scalar variable
variable of a scalar type, i.e., one that has no user visible components. 68

self-modifying code
program that modifies its own instructions during its execution. 8

set associative mapping
cache mapping scheme that requires the cache to be divided into sets of associative memory blocks. 80

side effect
modification of the value of some state variable outside the local environment of a procedure. 87

sign
attribute of a number, indicating if it is either zero, positive, or negative. 37

sign bit
bit in a signed number representation that indicates the sign of a number. 30

sign-magnitude
signed number representation that uses a sign bit to indicate the sign of the number and a set of additional bits to indicate its magnitude. 31

spatial locality
tendency of a processor to access the memory locations located near the recently accessed ones over a short period of time, when executing a program. 76

special value
bit pattern in a floating point numeral systems, with the largest or the smallest exponent, that is used to represent either zero, positive or negative infinities, or values that are not representable as real numbers. 39

stack
data structure optimised for processing the data elements according to the last-in first-out (LIFO) principle. 54

stack frame
data structure in the stack that stores the information needed by a single execution of a procedure; the same as activation record. 64

stored program computer
computer that stores the programs (and the data) in its internal memory to execute them; the same as general-purpose computer. 8

stride
number of locations in memory between successive array elements, measured in bytes or in units of the size of the array's elements. 95

structured variable
variable composed of a set of scalar variables and other structured variables. 68

subnormal number
non-zero floating-point number with magnitude smaller than the smallest positive normal number; the same as denormalised number. 38

system bus
internal bus commonly found on a computer that connects the CPU, the main memory, and all other internal parts. 4

tag
group of bits that permits identifying the memory address of the block stored in a given cache line. 77

temporal locality
tendency of a processor to access the same memory location several times over a short period of time, when executing a program. 76

throughput
number of instructions completed per unit time. 11

Turing machine
mathematical model of a simple computer used to analyse the logical foundations of computer systems. 8

two's complement
complement of a n-bit number with respect to 2^n. 32

two's complement numeral system
numeral system in which a negative number is represented by the inverse of the binary representations of its corresponding positive number plus one. 32

underflow
situation that occurs when the value of a number is smaller (that is, closer to zero) than the smallest value representable in the floating point numeral system. 39

Unicode
information technology standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. 18

valid bit
bit in a cache line that indicates whether the data stored in that line is valid or not. 77

vector image
computer image defined in terms of points on a Cartesian plane, connected by lines and curves to form polygons and other shapes. 22

von Neumann architecture
stored-program machine architecture composed of a CPU, an ALU, registers, and main memory. 4

word
basic unit of data handled by a given family of computers. 17

References

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers, principles, techniques, and tools*. Addison-Wesley.

Blum, R. (2005). *Professional assembly language*. Wiley Publishing, Indianapolis, IN, USA.

Borda, M. (2011). *Fundamentals in information theory and coding*. Springer, Berlin, Germany.

Bryant, R. E. and O'Hallaron, D. R. (2011). *Computer systems: A programmer's perspective*. Pearson Education, Boston, MA, USA, 2nd edition.

Bryant, R. E. and O'Hallaron, D. R. (2016). *Computer systems: A programmer's perspective*. Pearson Education, Boston, MA, USA, 3rd edition.

Ceruzzi, P. E. (1998). *A history of modern computing*. MIT Press, Cambridge, MA, USA, 2nd edition.

Fenwick, P. (2015). *Introduction to computer data representation*. Bentham Science Publishers, Sharjah, U.A.E.

Gajski, D. D. (1997). *Principles of digital design*. Prentice-Hall, Upper Saddle River, NJ, USA.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48. DOI 10.1145/103162.103163.

Harel, D. (2000). *Computers Ltd.: What they really can't do*. Oxford University Press, Oxford, United Kingdom.

Hennessy, J. L. and Patterson, D. A. (2007). *Computer architecture: A quantitative approach*. Morgan Kaufmann, San Francisco, CA, USA, 4th edition.

Hennessy, J. L. and Patterson, D. A. (2017). *Computer architecture: A quantitative approach*. Morgan Kaufmann, San Francisco, CA, USA, 6th edition.

Jacob, B., Ng, S. W., and Wang, D. T. (2008). *Memory systems: Cache, DRAM, disk*. Morgan Kaufmann, Burlington, MA, USA.

Katz, R. H. (1994). *Contemporary logic design*. Benjamin/Cummings, Redwood City, CA, USA.

Kneusel, R. T. (2017). *Numbers and computers*. Springer, Cham, Switzerland, 2nd edition. DOI 10.1007/978-3-319-50508-4.

Knuth, D. E. (1997). *The art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley, Boston, MA, USA, 3rd edition.

McEliece, R. (2005). *The theory of information and coding*. Cambridge University Press, 2nd edition.

Null, L. and Lobur, J. (2003). *The essentials of computer organization and architecture*. Jones and Bartlett Publishers, Sudbury, MA, USA.

Patterson, D. A. and Hennessy, J. L. (2014). *Computer organization and design: The hardware software interface*. Morgan Kaufmann, Waltham, MA, USA, 5th edition.

Rojas, R. and Hashagen, U., editors (2000). *The first computers: History and architectures*. MIT Press, Cambridge, MA, USA.

Smith, A. J. (1982). Cache memories. *ACM Computing Surveys*, 14(3):473–530. DOI 10.1145/356887.356892.

Stallings, W. (2019). *Computer organization and architecture: Designing for performance*. Pearson Education, New York, NY, USA, 11th edition.

Tanenbaum, A. S. and Austin, T. (2013). *Structured computer organization*. Prentice-Hall, Upper Saddle River, NJ, USA, 6th edition.

Unicode Consortium (2007). *The Unicode Standard, Version 5.0*. Addison-Wesley, 2nd edition.

Wakerly, J. (2001). *Digital design: Principles and practices*. Prentice-Hall, Upper Saddle River, NJ, USA, 3rd edition.

Index

absolute jump, 62
addition, 34
address, 7, 17, 77
addressing modes, 52
analog system, 15
arithmetic logic unit (ALU), 3, 6
ASCII, 18
assembler, 3
assembly phase, 49

bias, 33
big-endian, 8
binary, 15
binary point, 26
binary-coded decimal (BCD), 41
bit, 16
Boolean algebra, 16, 24
branch instruction, 12
bus, 5, 10
byte, 17

cache, 8, 95
cache block, 77
cache hit, 76
cache line, 77
cache memory, 75
cache miss, 76
calculator, 1, 59
callee-saved register, 65
caller-saved register, 65
cell, 7
central processing unit (CPU), 5
clock, 6
clock cycle, 6
code optimisation, 85
colour depth, 22
compilation phase, 49
computer, 1, 1, 4, 59
condition code, 59
content-addressable memory (CAM), 78
control unit, 6

data dependency, 12
data path, 3
decimal point, 26
device driver, 10
digit, 25
digital abstraction, 16
digital image, 21
digital system, 15
direct mapping, 78, 82
dots per inch (dpi), 22
dynamic random access memory (DRAM), 7

end carry-around, 32
excess-b representation, 33
exponent, 37

fetch-decode-execute cycle, 6
fixed-point-number, 36
flip-flop, 3
floating-point-number, 36
fully-associative mapping, 80

gate, 3

hexadecimal numeral system, 27
high-level language, 2
hit rate, 76
hit time, 76

I/O device, 9
IEEE 754 floating-point standard, 40
image metadata, 21
image resolution, 22
information theory, 24

inline expansion, 93
instruction, 1, 20
instruction pointer (IP), 6, 10, 51, 63, 64
instruction register (IR), 6
instruction set architecture, 3
interrupt, 10

label, 61, 68
latency, 11
least significant bit (LSB), 27
least significant digit (LSD), 27
line size, 77
linking phase, 49
little-endian, 8
loader, 49
loop unrolling, 91

machine language, 2
magnitude, 31
main memory, 7, 8, 17, 75
mantissa, 38
mean access time, 78
memory, 17
memory aliasing, 87
microprogram, 3, 6
miss penalty, 76
miss rate, 76
most significant bit (MSB), 27
most significant digit (MSD), 27

natural number, 26
nibble, 17
non-functional requirement, 85
normal number, 37
not a number (NaN), 39
numeral system, 25

octal numeral system, 27
one's-complement, 32
one's-complement numeral system, 31
opcode, 20
operating system, 3
optimisation blocker, 86
overflow, 33

pipelining, 10
pixel, 21
polling, 10
positional numeral system, 25
preprocessing phase, 48
principle of locality, 76, 95
procedure, 63
program, 1

radix, 25
radix point, 26, 36
random-access memory (RAM), 7
range, 26
raster image, 21
register, 3, 6, 8, 50, 52
relative jump, 62
replacement algorithm, 82
resource conflict, 12
return address, 64
return value, 63, 65

sample, 22
sample resolution, 22
sampling, 22
sampling rate, 23
scalar variable, 68
self-modifying code, 8
set-associative mapping, 80
side effect, 87
sign, 37
sign bit, 30
sign-magnitude, 31
software application, 2
sound, 22
spatial locality, 76, 95
special value, 38, 39
stack, 54, 63
stack frame, 64
status register, 6
stored-program computer, 4, 8
stride, 95
structured variable, 1, 68
subnormal number, 38, 38
system bus, 4

tag, 77, 77
temporal locality, 76, 95
throughput, 11
Turing machine, 8
two's-complement, 32, 33
two's-complement numeral system, 32

underflow, 39
Unicode, 18, 24

valid bit, 77
vector image, 22
von Neumann architecture, 4, 8

word, 17

Computers were originally invented to solve all sort of mathematical problems. Nowadays, computers do much more than that and are present in all human activities. In fact, a computer is a fantastic machine capable of doing the most amazing tasks, if an appropriate program is provided. A computer system contains hardware and systems software that work together to run software applications. Interestingly, the underlying concepts that support the construction of a computer are relatively stable. In fact, (almost) all computer systems have a similar organisation, i.e., their hardware and software components are arranged in hierarchical layers and perform similar functions. This book was written for programmers and software engineers who want to comprehend how the components of a computer work and how they affect the correctness and performance of their programs.

UMinho Editora

Universidade do Minho

ISBN 978-989-8974-60-0