UNIVERSITY OF LONDON

Introduction to computing and the internet: Volume 1

D. Miller

CO1110

2018

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London by:

D. Miller, formerly of the Department of Computing, Goldsmiths, University of London.

This guide draws on material in the previously published edition of the Volume 1 subject guide by F. Lin.

University of London
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

london.ac.uk

Published by: University of London

Contents

Chapter 1: Introduction to the subject guide

Welcome to **Introduction to computing and the internet**, a Level 4 course in an undergraduate degree in computing. The subject guide is divided into two parts, Volume 1 and Volume 2, which correspond to Part A and Part B in the examination. In this introductory chapter we will look at the overall structure of the subject guide – in the form of a **Route map** – and introduce you to the subject, to the aims and learning outcomes, and to the learning resources available. We will also offer you some examination advice.

We hope you enjoy this subject and we wish you good luck with your studies.

1.1 Route map to the guide

The aim of the full course is to give you an appreciation of some of the basic issues and concepts in computer science and the internet; including hardware, software, the representation and transmission of information as well as security and legal issues.

Volume 1 considers the way modern digital computers work at the level of hardware and the operating system; and how information is represented and processed on individual computers. There are six chapters in total in Volume 1; one introductory chapter, and five main content chapters, which are described below.

Chapter 1 introduces the subject and includes some general information about reading, learning resources, as well as some examination advice.

Chapter 2 gives a brief history of computers ending with the von Neumann computer architecture, and discusses why binary numbers are used in computers.

Chapter 3 looks in some detail at the different forms of computer memory, and at how data is stored and accessed.

Chapter 4 considers the components of the central processing unit (CPU) and their functions; the fetch-execute cycle; and ways of improving computer performance.

Chapter 5 describes the functions and processes of operating systems.

Chapter 6 describes how numbers, characters (text) and other forms of data are represented in a computer.

Appendix 1: Answers to the sample examination question included with each chapter.

Appendix 2: Answers to activities included with the chapters.

Appendix 3: Sample examination paper: Part A.

Appendix 4: Sample examination paper answers: Part A.

Volume 2 of the guide deals with the technologies and standards that enable computers to communicate and share information across a network. The protocols and technologies that support the internet are considered, together with security and legal issues.

Chapter 1 is the introductory chapter.

Chapter 2 explains why standards for internet working were developed and how they underpin the successful development of the internet. The chapter looks at networking models, focusing on the TCP/IP protocol suite.

Chapter 3 looks in detail at the major data structures and processes of the most important components of the TCP/IP protocol suite.

Chapter 4 considers the world wide web and email; briefly describes the operation of web servers and browsers and then discusses web authoring with HTML, XHTML and CSS.

Chapter 5 looks at the legal framework, including patent, copyright and applicable UK law with some reference to US and EU law. Cybercrime is defined and discussed.

Chapter 6 considers computer security from a technological perspective, and discusses the vulnerability of networked systems to malicious exploits and unwanted intrusions. Professional, legal and social issues in computer security are also reviewed.

Appendix 1: Answers to the sample examination question included with each chapter.

Appendix 2: Answers to activities included with the chapters.

Appendix 3: Sample examination paper: Part B.

Appendix 4: Sample examination paper answers: Part B.

These two volumes of the subject guides introduce vocabulary, concepts and skills that you will need to pass the examination at the end of the course. At the end of each chapter, a **Learning outcomes** section lists what you should be able to do.

1.2 Glossary of key terms

Some chapters include a list of new terms at their start, in order to help you to understand the material in the chapter.

1.3 Introduction to the subject area

Computers are interwoven with the fabric of our existence. A basic understanding of their architecture, storage and representation of data is an important foundation of knowledge and understanding for all computing professionals. Leading on from this, how computers work together through networks, particularly the internet, is key to understanding the opportunities and challenges of the 21st century. Computers and the internet need to be seen and understood in technological, legal, social and ethical terms. Professionals also need to understand new developments, so as to understand and anticipate both new opportunities for development, and emerging security threats.

This subject guide covers the first half of the course, as given in the first two course objectives, and the first three course learning outcomes below. The other objectives and learning outcomes will be addressed in Volume 2. Hence this subject guide focuses on **how computers work**. More specifically, it answers the following important questions:

In a computer –

- How is information **stored**?
- How is information **processed**?
- How is information **represented**?
- How are computing operations **coordinated** and **managed**?

In explaining the answers to these questions, various components of the hardware and software of a computer system will be discussed. These components include:

- main memory and other storage media, including fixed and removable media
- the central processing unit and system bus
- number representation: including two's complement and floating-point representation
- text representations: ASCII; Unicode
- operating systems.

1.3.1 Hardware and software

Chapter 3 describes in some detail the hardware implementations of computer memory; that is to say the physical components of electronic memory and secondary storage. Chapter 4 concentrates on the components and operation of the CPU, so again this chapter is mostly concerned with hardware, although Section 4.7 considers how hardware and software can work together to increase the speed of a machine. Chapter 5 considers the operating system, which is system software, and how it interacts with the machine's hardware. Chapter 6 considers data representation in a computer. Number representation is part of the hardware of a computer.

Hardware means those physical parts of the computer that still exist when the machine is switched off; for example, RAM chips, the processor and hard drive. **Software** refers to processes that are, or can be, run on the machine.

1.4 Syllabus

1.4.1 Volume 1

Computer architecture: von Neumann architecture, CPU, memory, I/O devices, data, address and control buses.

Data storage: main memory, storage including disk storage.

Central Processing Unit: the fetch-decode-execute cycle, instructions, clocks, cache memory, pipelining, CISC versus RISC.

Data representation: representing characters, numbers, images, movies, sound.

Operating systems: file management, process management, memory management, I/O management, network management.

Terminology: hardware, software, compiler, Unix, Windows, etc.

1.4.2 Volume 2

How the internet works: addressing and routing – URLs, domain names, IP addresses. Protocols: TCP/IP; HTTP; SMTP; POP; client-server model; web servers and browsers; search engines; electronic mail; file transfer.

Standards and regulation: role of W3C consortium; domain name registration; standards for SGML, HTML, XML.

Webpage design and coding: HTML basics: document structure; links; URLs; fonts; colours; images; lists; tables and frames. Dynamic HTML: Javascript; style sheets. Basic overview of advanced technologies for dynamic and active web documents: CGI scripting; Active Server Pages; PHP; Java applets.

Professional issues: security of networked systems: unauthorised access and malicious code, procedural and technical defences; legal framework. Data protection and privacy: principles of data protection; role of the Information Commissioner; rights of data subjects. Internet usage in the workplace: auditing and monitoring; intellectual property rights. Liabilities of ISPs: defamation; obscenity; secure communications.

1.5 Aims of this course

To give you, the student, an appreciation of some of the basic issues and concepts in computer science and the internet, including hardware, software, the representation and transmission of information and security issues.

1.6 Learning objectives for the course

- Give students the information they need to understand some of the basic computer science terminology.
- Provide an overview of how computers work by looking at issues such as: how computers are composed, how data is stored, how information is represented, how information is processed, and how computing operations are coordinated.
- Provide an introduction to the internet and the world wide web in terms of technologies, protocols, standards and applications.
- Provide an appreciation of professional, legal and social issues relating to networked computing.

1.7 Learning outcomes for students

On completion of this course, you should be able to:

- demonstrate understanding of some common, basic terminology in computer science
- explain the operation of modern digital computers at the level of hardware and the operating system
- demonstrate understanding of some common means of representing and storing information in digital format
- demonstrate a basic understanding of internet/www mechanisms, architecture and protocols and applications such as electronic mail, file transfer and web browsers
- code web pages with some dynamic features, using a text editor or web authoring tool
- describe professional, legal and social issues impacting on provision and use of internet services with particular reference to data protection, privacy, computer misuse and cybercrime.

1.7.1 Prior knowledge required

This subject guide does not assume any prior knowledge except for some mathematics. You will be expected to be able to:

- demonstrate understanding of what raising a number to both a positive and a negative (whole number) power means; for example:
  - that 2^{-2} means $2/2^2$ or 2/4 i.e. 0.5
  - that 2^2 means 2 x 2 and 2^3 means 2 x 2 x 2
  - that any number raised to the power of zero is 1

- demonstrate understanding of the binary number system and of what place value means for binary numbers; namely, that the digits in a binary number have the values:
  $b \times 2^n$, $b \times 2^{n-1}$, $b \times 2^{n-2}$,…, $b \times 2^2$, $b \times 2^1$, $b \times 2^0$ where b can be either 0 or 1
- demonstrate understanding of the fractional values of the binary system; for example, that 0.101_2 means $0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$
- convert between binary numbers and decimal numbers (for example, explain that 10101.11_2 is a binary number, and that its decimal equivalent is $16 + 0 + 4 + 0 +1 + 0.5 + 0.25 = 21.75$)
- convert decimal numbers to binary numbers. You can find good videos on the internet (for example, YouTube) demonstrating how to do the conversion, or you can consult your other study materials. There are online converters, which you can use to check your work, but do not rely on them, as you will not be able to use them in an examination
- demonstrate understanding of scientific notation for decimal numbers; for example, that 5019×10^n is the same number as $5.019 \times 10^{n+3}$ and that 6.036×10^3 is the same number as 6036
- apply scientific notation to binary numbers; for example, show that $1.01011_2 \times 2^5$ is the same number as 101011_2. Note in the example the number could be given as $1.01011_2 \times 100000_2$ but the exponent is usually given in decimal form
- demonstrate a basic understanding of logarithms, and explain how to work out the log base 2 of a decimal number. If you have a calculator (you are allowed to use calculators in the examination) use it to work out $\log_2(x)$ using the natural log function, i.e. $\log_2(x) = \ln(x)/\ln(2)$ or use log base 10; that is, $\log_2(x) = \log_{10}(x)/\log_{10}(2)$
- add binary numbers.

1.8 Overview of learning resources

1.8.1 The subject guide

Each chapter in this guide starts with a list of **Essential reading** and possibly some **Further reading**.

The **Essential reading** section directs you to the textbook sections that you have to read. For each chapter you are given Essential reading from Stallings.

Certain topics in the subject guide are covered in greater depth in the Essential reading given at the start of each chapter. You do not have to read every page of each textbook, but you should pay careful attention to those sections and chapters that are given as Essential reading.

The **Further reading** section gives chapters and sections you are advised to read. None of this suggested reading is compulsory, but it is designed to broaden and deepen your knowledge and understanding of the topics covered in that chapter. You are also encouraged to find other relevant books in libraries, or search for information on the internet.

1.8.2 Essential reading

The following book is recommended to support this course:

- Stallings, W. *Computer organization and architecture: designing for performance*. (Harlow: Pearson Education Ltd, 2016) 10th global edition [ISBN 9781292096858]. See also author's website for updates and useful information: http://williamstallings.com/

Detailed reading references in this subject guide refer to the editions of the set textbook listed above. New editions of the textbooks may have been published by the time you study this course. You can use a more recent edition of the book; use the detailed chapter and section headings and the index to identify relevant readings. Also check the virtual learning environment (VLE) regularly for updated guidance on readings.

1.8.3 Further reading

Please note that as long as you read the Essential reading you are then free to read around the subject area in any text, paper or online resource. You will need to support your learning by reading as widely as possible and by thinking about how these principles apply in the real world. To help you read extensively, you have free access to the VLE and University of London Online Library (see below).

Other useful texts for this course include:

- Harris, S.L. and D.M.H. *Digital design and computer architecture*. (Waltham, MA: Elsevier/Morgan Kaufmann, 2016) ARM edition [ISBN 9780128000564].
- Patterson, D.A. and J.L. Hennessy *Computer organization and design: the hardware/software interface*. (Amsterdam: Elsevier/Morgan Kaufmann, 2017) ARM edition [ISBN 9780128017333].
- Stokes, J. *Inside the machine: an illustrated introduction to microprocessors and computer architecture*. (San Francisco, CA: No Starch Press, 2015) [ISBN 9781593276683]. Chapter 1 'Basic computing concepts' is very useful as an introduction.

1.8.4 Websites

Stallings, W. 'COA10e-student', Books by William Stallings: http://williamstallings.com/ComputerOrganization/styled-6/ – Stallings' own web page for the 10th edition of his book. Includes a link to the most up-to-date errata file, and some links for extra reading for each chapter of his book.

Stokes, J. 'Understanding the microprocessor – part 1: basic computing concepts' arstechnica: http://archive.arstechnica.com/paedia/c/cpu/part-1/cpu1-2.html – some very useful discussions by Jon Stokes on CPU basics, including pipelining.

1.8.5 Online Library and the VLE

In addition to the subject guide and the Essential reading, it is crucial that you take advantage of the study resources that are available online for this course, including the VLE and the Online Library.

You can access the VLE, the Online Library and your University of London email account via the Student Portal at: http://my.londoninternational.ac.uk/

Unless otherwise stated, all websites in this subject guide were accessed in June 2018. We cannot guarantee, however, that they will stay current and you may need to perform an internet search to find the relevant pages.

1.8.6 End of chapter Sample examination questions and Sample answers

To help you practise for the examination, we have included some end of chapter Sample examination questions with their scope limited to the learning outcomes of the chapter they are found in. Each question is similar in style and difficulty to actual examination questions.

You will have three hours for the examination, and will be required to answer four questions. This gives you about 45 minutes per question, including reading and checking time.

Once you have attempted these Sample examination sections under timed conditions, you should check the Appendix at the end of the subject guide for the Sample answers.

1.9 Examination advice

Important: the information and advice given here are based on the examination structure used at the time this guide was written. Please note that subject guides may be used for several years. Because of this we strongly advise you to always check both the current programme regulations for relevant information about the examination, and the VLE where you should be advised of any forthcoming changes. You should also carefully check the rubric/instructions on the paper you actually sit and follow those instructions.

The examination paper for the course **Introduction to computing and the internet** contains six questions in all, three from the material covered in Volume 1 and three from the material covered in Volume 2. The examination lasts for three hours. Therefore you have about 45 minutes on each of the **four** questions you choose to answer (two questions from each section). Each question is worth 25 marks, with 100 marks in total available on the paper. If you answer all three questions from any section, your third answer will not be marked. You will be asked to do some calculation, hence calculators are allowed.

It is good practice to read all of the questions before you start.

When you are asked to show all your working, it is important that you do so; students who only write down the final answer, even when correct, will lose marks.

For example, a question asks students to convert a decimal number into an IEEE 754 number – for seven marks – and to show all their working. Student A writes down the correct answer. Student B writes down the wrong answer. Student A gets one mark, student B gets six marks. The marking scheme specifies one mark for the final answer, and six marks for the various stages needed to find the final answer. Student A wrote down no working, but student B wrote down all of theirs, and it was correct, with a small mistake made when translating all of their working out into a final answer.

When answering a question, you should try to be precise and try to cover all the relevant points. When asked to explain something, imagine that you are explaining it to someone whose knowledge of the subject is very limited. Always take note of the marks awarded for a question as this is likely to reflect both the amount of time the question is expected to take you and the number of elements in the answer.

You will find Sample examination questions at the end of most chapters. Use them as a tool to remember what you have read and in order to measure your progress. You will find sample answers in the appendix that should help you both in testing yourself, and in judging how to answer examination questions successfully.

Many aspects of the examination will recur, with changes, in different years. You can access previous papers and *Examiners' commentaries* on the VLE. Practising with past examination papers is probably the single best way to prepare in the weeks before the examination. You are strongly advised to practise under timed examination conditions.

Remember, it is important to check the VLE for:

- up-to-date information on examination and assessment arrangements for this course
- where available, past examination papers and *Examiners' commentaries* for the course.

1.10 Overview of chapter

In this chapter, we have introduced the course and this volume of the subject guide, listing the aims and objectives of the course and outlining some practical aspects of working through the subject guide, the reading and other resources, before offering some advice on examination preparation.

1.11 Test your knowledge and understanding

1.11.1 A reminder of your learning outcomes

By the end of this course, and having completed the Essential readings and activities in both Volume 1 and Volume 2, you should be able to:

- demonstrate understanding of some common, basic terminology in computer science
- explain the operation of modern digital computers at the level of hardware and the operating system
- demonstate understanding of some common means of representing and storing information in digital format
- demonstrate a basic understanding of internet/www mechanisms, architecture and protocols and applications such as electronic mail, file transfer and web browsers
- code web pages with some dynamic features, using a text editor or web authoring tool
- describe professional, legal and social issues impacting on provision and use of internet services with particular reference to data protection, privacy, computer misuse and cybercrime.

Chapter 2: The development of computers

2.1 Introduction

This chapter considers the historical development of modern computer architecture.

2.1.1 Aims of the chapter

This chapter aims to give you an overview of the history and development of general-purpose computers and their architecture, together with the motivation for the use of binary numbers.

2.1.2 Learning outcomes

By the end of this chapter, and having completed the Essential reading and activities, you should be able to:

- outline the history of computers
- explain the von Neumann architecture
- explain the advantages of using the binary system in computing.

2.1.3 Essential reading

- Stallings, W. *Computer organization and architecture: designing for performance*. (Harlow: Pearson Education Ltd, 2016) 10th global edition [ISBN 9781292096858], Section 1.3 'A brief history of computers'.

2.1.4 Further reading

- Patterson, D.A. and J.L. Hennessy *Computer organization and design: the hardware/software interface*. (Amsterdam: Elsevier/Morgan Kaufmann, 2017) 5th revised edition [ISBN 9780128017333], Section 1.12 in print and online: http://booksite.elsevier.com/9780128017333/historial.php
- Harris, S.L. and D.M. Harris *Digital design and computer architecture*. (Waltham, MA: Elsevier/Morgan Kaufmann, 2016) ARM edition [ISBN 9780128000564], Section 1.3 'The digital abstraction'.
- Stokes, J. *Inside the machine: an illustrated introduction to microprocessors and computer architecture*. (San Francisco, CA: No Starch Press, 2016) [ISBN 9781593271046] Chapter 1 'Basic computing concepts'.
- See also Stallings' website for updates and useful information: http://williamstallings.com/

2.1.5 References cited

- *Dougherty, R.C. 'Claude Shannon', NYU*: www.nyu.edu/pages/linguistics/courses/v610003/shan.html accessed 18 June 2018.
  Rather old now as evidenced by the writer assuming that Shannon is still alive (he died in 2001), but accurate (note: grammatical errors in text).
- WhatIs.com 'WhatIs TechTarget' WhatIs.com htpp://whatis.techtarget.com accessed 14 June 2018

2.2 Glossary of key terms

- A **transistor** is a device that regulates current or voltage flow and acts as a switch or gate for electronic signals. Transistors consist of three layers of a semiconductor material, each capable of carrying a current. The transistor was invented by three scientists at Bell Laboratories in 1947, and it rapidly replaced the vacuum tube as an electronic signal regulator (http://whatis.techtarget.com/)

- Transistors are the basic elements in **integrated circuits (IC)**, which consist of large numbers of transistors interconnected with circuitry and baked into a single silicon microchip. (http://whatis.techtarget.com/). ICs are also called chips or microchips. Since the first generation of ICs, the number of transistors per chip has followed Moore's Law (Stallings 2016, Chapter 1, Section 1.3) in roughly doubling every year, slowing to doubling every 18 months since the 1970s. This leads to the following generations of ICs:
  - **Large scale integration** – thousands of transistors on a single chip.
  - **Very large scale integration** – hundreds of thousands of transistors on a single chip.
  - **Ultra large scale integration** – millions of transistors per chip.
- A **vacuum tube** (also called a VT, electron tube or, in the UK, a **valve**) is a device sometimes used to amplify electronic signals. In most applications, the vacuum tube is obsolete. (http://whatis.techtarget.com/).
- **Relays** are switches, extensively used in early telephone exchanges to route calls. In the 1940s Claude Shannon demonstrated how to use relays to implement Boolean logic.

2.2.1 Abstraction

This is a way of masking complexity in order to highlight the key parts of a system. In terms of computer systems, lower level details are hidden so that higher levels can be viewed more simply, in order to bring out their most important design features and processes. For example, to operate application software such as a spreadsheet program, a user does not need to know how the CPU works. Computers can be viewed as having a hierarchy of levels, starting with the physics of electrons moving through wires. After this there are the transistors, capacitors and diodes used to implement cells and logic gates, followed by the cells and logic gates themselves at the next level. Successive levels include the architecture of the CPU, the operating system OS and application software. Different hierarchies are possible, at different levels of abstraction, but the key point in ordering the hierarchy is that each level must be able to exist without any of the levels above it, but cannot exist without the levels below it.

2.3 What is a computer?

First of all, what is a computer? Originally, the English word computer, meant a person who computes. In fact the Chambers dictionary for the year 2000 defines a computer as:

> An electronic machine for carrying out complex calculations, dealing with numerical data or with stored items of other information, also used for controlling manufacturing processes or coordinating parts of a large organisation; a calculator; **a person who computes**.

2.3.1 The abacus

Some might argue that the earliest computer was the abacus. An abacus is made up of a number of rows (or columns), with each row having a number of beads on it. The abacus has been used for hundreds of years, with its exact origins unknown. It can allow a skilled person to add, subtract, multiply and divide quickly by moving the beads according to certain rules. The configuration of the beads in an abacus represents a decimal number. The rules for moving beads are instructions. They are stored in the memory of the person who uses the abacus, but the abacus itself has no memory. Since it has no memory and no stored instructions the abacus does not process information, but helps its human operator to process data efficiently.

2.3.2 The slide rule

The slide rule was developed in the 17th century to give mathematicians a fast way of performing calculations with recently developed logarithms. The first slide rule, performing multiplication and division was invented by the English clergyman and mathematician William Oughtred, in about 1622. Over time other calculations were added.

So the question is, how do we define a computer?

2.4 Definition of a computer

A computer should have four things

1. a way of inputting information (**input**)
2. a way of storing information (**memory**)
3. a way of processing information (**processor/instructions**)
4. a way to output the processed information (**output**).

We could further divide the processing of information into two parts:

- general controls that see to it that instructions are carried out
- actual instructions specific to a particular task.

This definition would mean that an abacus, which has no way of processing information independently of its human operator, is not a computer. However, a slide rule has these four things and can be considered to be a non-digital computer, although not a general purpose one since it is dedicated to a particular task: calculations. Slide rules were used by scientists, engineers and students of mathematics until the 1970s, when they began to be replaced by hand-held electronic calculators.

2.5 Development of computers

Logarithms and slide rules were what allowed mathematicians and engineers to do quicker calculations until the computing age. Science fiction magazines from the 1920s are illustrated with engineers clutching their trusty slide rules as they climb the sides of grounded space rockets. Mathematicians invented log tables, because arithmetic in itself is not that interesting, but it underpins mathematics. A faster way of performing calculations allows mathematicians to more quickly move on to the interesting mathematics. For this reason the history of the development of computers is littered with mathematicians attempting to find a way to make a machine do the calculations so that they do not have to.

2.5.1 Pascal’s and Leibniz’s computation machines

The first machine that could do some processing, was invented by the French mathematician, inventor and philosopher Blaise Pascal (1623–62), in whose honour the programming language Pascal is named. In 1642 Pascal invented a device to help his father, a tax collector for the French government. It consisted of a number of wheels, each divided into 10 segments and capable of being rotated through any number of these. The wheels were connected with each other by a 'carry-over' lever so that, on 10 being registered on one wheel, a carry-over of one was made to the next. The position of the wheels at any given time represents a number. If a number were to be added to an existing number, then each of the relevant wheels would need to be rotated through a corresponding number of segments. The machine would then make some adjustment itself (for example, doing the carry-over operation) and the resultant state of all the wheels would represent the required sum. The adding operation was done jointly by a person, who turned the wheels, and the machine, which did the carrying-over operation.

Pascal's machine could only do addition and subtraction. In 1694, the German mathematician Gottfried Leibniz (1646–1716) built a machine that could do multiplication and division as well. Leibniz's machine was made up of a number of drums and gear wheels, and also some carry-over mechanisms. Rotating the drums caused the gear wheels to move. Calculation was done by turning the drums in an appropriate way, and the resultant state of the gear wheels showed the result. Thus, the processing was also jointly done by a person and the machine.

2.5.2 Pascal's and Leibniz's machines considered as modern computers

Let us pause a moment to ponder the nature of Pascal's and Leibniz's machines. Let us focus on Pascal's machine (Leibniz's machine was similar) and try to evaluate it using modern computing terms.

- **INPUT – adding two positive numbers**: The user would first input one number, by turning the wheels into certain positions, then input another number, by further turning some wheels.
- **INPUT – adding negative numbers**: The user needed to input the two numbers in a different way since there was no single algorithm of addition.
- **MEMORY**: The machine could remember one number while the second one was entered, and the result of the computation would be kept until the user reset the machine for another calculation.
- **PROCESSOR**: The machine calculated using an ingenious mechanism to carry numbers. The details of the calculation used depended on how the user had input the numbers.
- **OUTPUT**: The machine would display the result of the calculation.

In terms of modern computers:

- neither machine was a general purpose machine since they were both dedicated to addition
- both machines relied on the user entering numbers in a particular way, meaning that the user did some of the processing
- since processing was partly done by the user, and partly by the machine, input and processing were not entirely separate and distinct
- the machines had a limited memory, but no long-term storage
- instructions were not stored and loaded, rather the user had to know how to operate the machine; processing was entirely mechanical hence neither machine was programmable.

Digression 1

One of Pascal's *Pensées*: 'Say nothing unless it improves on silence.'

(Pascal wrote a collection of *Pensées* (or 'thoughts') on theology and philosophy.)

2.5.3 Babbage's difference engine

For the next landmark in the development of computing machines we turn to the English mathematician Charles P. Babbage (1792–1871) of the University of Cambridge. He built a 'difference engine', designed to compute tables of numbers useful for naval navigation. For example, the machine could compute the values of x^2 for any x. The machine could accept a number, compute the value of the number squared, and then punch the result into a copper engraver's plate with a steel die. The difference engine had a clear algorithm. There was a distinction between the input device, output device and the processing unit. But it had only one algorithm, determined by the physical make-up of the machine. Thus the use of the difference engine was limited.

2.5.4 Babbage's analytical engine

Babbage went on to design an 'analytical engine'. This machine used hole-punched cards for input. Different algorithms could be punched onto the input cards, so the machine could perform different computations. The processing unit was made up of thousands of cogs, wheels and gears and it was this complexity that meant that it was not built in Babbage's lifetime, despite his efforts. It was too expensive for Babbage to build without help. However, because of the engineering challenge posed by all those moving parts, those who seriously considered the matter were unsure – if it was built – that it would work as intended.

2.5.5 The analytical engine as the first general purpose computer

Babbage's machine is considered the first general-purpose computer, and looking at the definition of a computer given earlier:

- 'A way of inputting information': **done via hole-punched cards**.
- 'A way of storing information': **the machine could punch its own cards in order to store data**.
- 'A way of processing information': **a unit to perform arithmetic operations**.
- 'A way to output the processed information': **a printer, a plotter and a bell**.

Babbage's analytical engine, in addition to having separate processing capability, memory and input/output devices, also divided the processing of information into two parts: (1) the general controls that see to it that instructions are carried out; and (2) the actual instructions specific to a particular task. In this way it anticipated the basic architecture of modern general-purpose computers.

Activity 2.1

Question: Explain why the difference engine is not considered to be the first general-purpose computer.

Digression 2

The mathematician Ada Lovelace (properly Augusta Ada King, Countess of Lovelace) was a friend of Charles Babbage's. She recognised that the analytical engine would be able to do more than simple arithmetic, and is now credited with writing the first algorithm for a machine to implement. For this reason she is sometimes called 'the first computer programmer' and had a 20th century computer programming language, Ada, named after her.

2.5.6 ENIAC

The processing unit in Babbage's analytical engine was made of cogs, wheels and gears. The positions of the latter represented certain numbers, and the algorithms were implemented in terms of the movement of the cogs, wheels and gears. Such a processing unit was considered too great an engineering challenge to build in the 19th century. The next great idea in the development of computers was to use vacuum tubes for number representation and computation. This idea was seen in the Electronic Numerical Integrator and Computer (ENIAC), built between 1943 and 1946 at the University of Pennsylvania. In the ENIAC each decimal digit was represented by a ring of 10 vacuum tubes. At any time, only one vacuum tube was in the ON state, representing one of the 10 digits. Thus, to represent a 10-digit number, at least 100 vacuum tubes would be needed. The ENIAC had more than 18,000 vacuum tubes for representing and calculating numbers. The calculations were realised by a large number of switches and electrical cables, which caused certain vacuum tubes to be ON or OFF according to the inputs and the algorithms. The ENIAC was the world's first general-purpose electronic digital computer, and it was capable of performing 5,000 additions per second.

The ENIAC was an impressive invention. But it had a major drawback: it had to be programmed manually by setting switches and plugging and unplugging cables. Thus, it was extremely tedious to enter and alter programs for the ENIAC. It would be much easier, and more desirable, if programs could be separate from the computer and simply 'read-in' by the computer, as with Babbage's analytical engine. This led to the von Neumann architecture.

Activity 2.2

If the 100 vacuum tubes needed to represent a 10-digit decimal number in the ENIAC computer were used to represent binary numbers, and assuming that each tube could be OFF or ON, give the largest binary number the tubes could represent. You should write your number in decimal in terms of powers of 2.

Digression 3

Colossus is now considered to be the world's first programmable electronic digital computer. It was developed by British engineers and mathematicians to break German military encryption during the Second World War, and for many years was hidden from history as after the war it was broken up and kept secret, hence it had little or no influence on the development of computers in the 1940s and 1950s. Since Colossus was dedicated to a particular task, it was not a general-purpose computer like ENIAC.

2.5.7 von Neumann machine

John von Neumann was a mathematician, and a consultant on the ENIAC project. He saw clearly ENIAC's defect, and proposed a better architecture for computers, which is illustrated in Figure 2.1 below:

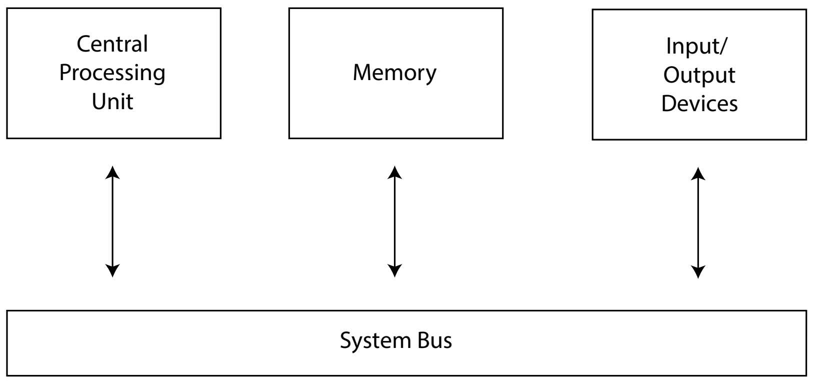


Figure 2.1: von Neumann architecture.

According to von Neumann, a general-purpose computer should have the following components (as shown in Figure 2.1):

- **Central processing unit (CPU)** – for interpreting instructions and doing calculations. The CPU consists of:
  - a **control unit**
  - an **arithmetic and logic unit** (ALU).
- **Memory** – for holding instructions and data used for certain calculations.
- **Input devices** – for inputting data and instructions.
- **Output devices** – for outputting results.

All these components are connected through the system bus, which is a set of wires. Most of today's computers have this same general structure. Modern CPUs have a number of registers, discrete memory locations for data and instructions, which will be discussed in Chapter 4 of the subject guide.

2.5.8 The IAS computer and the binary system

John von Neumann and his colleagues developed a new computer implementing von Neumann's architecture, at the Princeton Institute for Advanced Study between 1943 and 1952. The computer has been referred to as the IAS computer.

The IAS also used vacuum tubes, but it was based on the binary system, rather than the decimal system used in most of the earlier computers. Binary arithmetic and calculus were invented by Leibniz around 1694. Leibniz showed that all decimal numbers can be represented, and all arithmetic calculations can be done, in the binary system. The first computing machine based on the binary system was developed in 1938 by Konrad Zuse. The machine was entirely mechanical, with an arithmetic unit composed of large numbers of mechanical switches, and a memory consisting of layers of metal bars between layers of glass. It could be programmed by means of a punched tape, was dedicated to numerical calculations, did not work well because it was poorly engineered, and was destroyed by an air-raid on Berlin in 1943. Zuse developed two successor machines, also destroyed by Allied bombing, and a final machine he developed during the war was completed and sold in 1950.

The next binary computing machine was developed by John Atanasoff and his student Clifford Berry, at what was then Iowa State College (now University) during the period 1937–42. The machine was called the Atanasoff–Berry Computer, or ABC, a small-scale special-purpose electronic digital machine for the solution of systems of linear algebraic equations. The machine contained approximately 300 vacuum tubes.

IAS was not the first computer to use the binary system, nor the first to use vacuum tubes. But it had the following components:

- a main memory (M) of 1,000 storage locations, which stored both data and instructions
- an arithmetic-logical unit (ALU) capable of operating on binary data
- a control unit, which interpreted the instructions and controlled execution
- input and output (I/O) equipment operated by the control unit.

Therefore, the IAS was the prototype for all subsequent general-purpose computers.

Digression 4

Konrad Zuse, 1910–95, computer pioneer, was the first person to suggest that our universe may be a computer program. To date there is no evidence against this idea.

2.5.9 Why binary?

A major factor contributing to the choice of the binary system was the invention of transistors in 1947 at Bell Telephone Laboratories. A transistor is a small semi-conductor device which has two states: open and closed; and these two states correspond to 0 and 1 in the binary system, hence calculations in binary could be implemented with transistors. This meant faster computing, but also hardware that was easier to build, as binary requires two states rather than the 10 states required to implement decimal numbering. Additionally, a computer that has to differentiate between 10 states would be more error prone than one that only has to detect the difference between two.

The mathematician Claude Shannon, of Bell Telephone Laboratories, in his important 1948 paper 'A mathematical theory of communication', enumerated the advantages of binary numbers in computer science. (Claude Shannon is considered to be the 'founding father' of the age of electronic communication. See: www.nyu.edu/pages/linguistics/courses/v610003/shan.html)

2.5.10 The first three computer generations

Computer generations are defined by the technology used for their basic hardware. The difference between the first and second computer generations is considered to be the replacement of the vacuum tube by the transistor. The difference between the second and third generation is considered to be the introduction of integrated circuits, which not only simplified many technical issues with incorporating thousands of transistors into one computer, but also, according to Stallings (2016, Section 1.3) 'revolutionized electronics and started the era of microelectronics'.

The first generation of computers used vacuum tubes and relays for logic and registers. Vacuum tubes, invented in 1906, consumed a lot of electricity and thus generated a lot of heat, which made them prone to failure. The larger the system, the more often vacuum tubes would fail. The ENIAC took up 1,500 square feet of floor space, and had thousands of vacuum tubes, some of which could be relied upon to fail during every hour of its operation. Each failure had to be traced and replaced. The ENIAC was also the last computer built with vacuum tubes.

The first transistors increased calculating speeds, by being smaller (perhaps 100th the size of a vacuum tube), faster and more reliable. They were also cheaper, used less energy and put out much less heat. Stallings (2016, Section 1.3, p.41) notes that, as well as transistors, 'The second generation saw the introduction of more complex arithmetic and logic units and control units, the use of high level programming languages, and the provision of **system software** with the computer' (emphasis Stallings).

2.5.11 List of computer generations

After the third generation, different authorities are at variance on the division of generations; however, Stallings (2016, Section 1.3) gives the following six generations of general-purpose computers, each of which represents a significant increase in processing power.

- First generation (1946–57): Bulky, temperamental machines built using relays and vacuum tubes.
- Second generation (1957–64): Transistor-based machines with magnetic core memory, programmed with high-level languages such as Fortran and COBOL, and controlled by software called an operating system.
- Third generation (1965–71): Machines built with integrated circuits. Operating systems that allowed shared use of machines.
- Fourth generation (1972–77): Machines built with large-scale integrated circuits.
- Fifth generation (1978–91) Machines built with very large-scale integrated circuits.
- Sixth generation (1991–present): Machines with ultra large-scale integration.

With each generation of computing the typical number of operations per second has increased by at least an order of magnitude, going from 40,000 in the first generation, 200,000 in the second, to greater than 1,000,000,000 today.

2.5.12 Other ways of classifying computers

Computers could be characterised by how many users they can support:

A **personal computer** (PC) is a microcomputer intended to be used by a single user at any one time.

Or by their portability:

A **desktop computer** is a personal computer designed to be used at a fixed location, normally a desk or table, because of its size and power needs.

A **laptop** is a portable personal computer.

Or their purpose:

A **server** is any computer that runs server software, allowing it to be networked and provide services to network users (other computers and people).
While a personal computer could be used as a server, normally industry will choose machines that are designed for the task, with large hard drives (often implementing RAID in order to be able to recover from data loss), a large and fast main memory and fast processor(s) so that they can quickly serve multiple user (client) requests. Usually a server is designed to run all the time, and so will be connected to backup power supplies in case of outage, and will have systems

in place to regularly back up data. A server will also normally have security to allow only authorised users access, and users often have levels of access permissions. Servers can be dedicated to particular functions, such as gaming, databases or email, and often run the Unix operating system, or systems based on Unix such as Linux, because of its good security and reliability (servers running on Unix or Unix-based operating systems rarely fall over).

2.5.13 Types of general-purpose computers

It might be more meaningful to consider the types of general-purpose machines developed since the first generation:

- mainframe
- minicomputer
- microcomputer
- supercomputer.

The first generations of computers were known as mainframes; they were very large, usually demanding their own air-conditioned room. After this the minicomputer was invented in the 1960s, allegedly named 'mini' because of the fashion for short skirts at the time, and to distinguish it from the larger and more expensive mainframes. With the 1970s came the microcomputer, whose development into the home computer was not driven by industry, but by computer enthusiasts who wanted a machine that they would be able to own and run themselves. A microcomputer has a microprocessor as its CPU. Supercomputers, designed to optimise mathematical and logical operations, began to be developed in the 1960s.

We will consider each one in turn.

2.5.14 Mainframe

The first generation of computers were called mainframes after the cabinets ('main frames') that the (very large) central processing unit and main memory were stored in.

Today's mainframes are high-speed multi-user computers. A mainframe typically supports a large database, has high storage, lots of processing power and high reliability, plus high-end I/O hardware designed to handle high volume I/O. A mainframe might be used in a central data processing facility.

2.5.15 Minicomputer

In the 1960s, IBM developed a computer that was much cheaper than the mainframes then on sale. They called it a minicomputer; it was aimed at businesses that could not afford a mainframe, or whose computing requirements would not justify such an expensive outlay. The machine could support multiple users and handle high workloads. Today the term minicomputer is not used, and has been replaced with mid-range computer; mid-range computers are often used as servers.

2.5.16 Microcomputer

With the development of the integrated circuit, it was suddenly possible to conceive of a general-purpose computer that was dedicated to a single user, and cheap enough to be owned and used by individuals. The development of microcomputers began in the 1970s, and although there were companies offering microcomputers for sale to business users, its development was really driven by hobbyists, many of whom were engineers who had long dreamed of owning a computer. A microcomputer was smaller and crucially, cheaper, than the mainframes and minicomputers being sold to businesses and universities when microcomputers started to be developed. A microcomputer has a microprocessor, main memory, storage, and an I/O facility.

The first microcomputers did not involve abstraction, and there were no commercially available applications; the user needed to be familiar with binary numbers and machine language to program them. What came to be called the home computer, was a microcomputer that was easier to use because it operated at a higher level of abstraction with the addition of a screen to allow the user to see their input as text and numbers. In the home computer, machine code was replaced by the BASIC programming language, familiar to many of the engineers involved in the early designs. In 1981, IBM released its own version of the home computer, the IBM PC, with a keyboard and screen. In 1982, *Time* magazine chose the PC as their 'Man of the Year'.

Today the term microcomputer is rarely used; people say 'PC' when they mean a computer based on the original IBM PC design; and 'Mac' for Apple computers. Both are general-purpose microcomputers.

Microcomputers include: desktop computers, laptops, tablets, smart-phones, video game consoles, calculators and embedded systems. Microcomputers dedicated to a particular task, such as video game consoles and calculators, are not general-purpose. Also many embedded systems might not be general-purpose computers, being dedicated to particular tasks such as an anti-lock braking system in a vehicle.

2.5.17 Supercomputer

A supercomputer is an extremely fast, extremely expensive computer. The first supercomputer was developed and sold in the 1960s, designed by Seymour Cray to optimise the speed of mathematical and logical operations. It had one specially designed processor, the next had eight processors, today's can have millions. Most supercomputers run Linux or Linux-based operating systems.

Computational performance records always belong to supercomputers, currently the fastest use millions of processors, and measure their speed in petaflops. A petaflop is 10^{15} floating point operations per second, for comparison there are thought to be in the order of 10^{8} stars in the Milky Way, and it has been estimated that there are 10^{20} stars in the universe. Supercomputers are very expensive because of the processing power and high speed memory packed into them, so expensive that the price of the fastest is only affordable by very large and well-resourced organisations such as corporations and governments.

Supercomputers are used for scientific and engineering purposes such as modelling the weather for forecasting, or analysis of earthquake resistance in structures. They are also used for military and intelligence purposes, for example attempting to break modern encryption. This makes their development political, as evidenced by the USA banning the export of supercomputer chips to China in 2015, which was after China's Tianhe-2 computer, built with more than three million Intel Xeon E5 cores, became the world's fastest publicly known supercomputer, capable of 34 petaflops.

It is impossible to really know how fast the fastest supercomputer is, since organisations such as the National Security Agency (NSA) are not going to tell you what they have. From 2016 until 2018 China's TaihuLight computer, built with Chinese-designed chips because of the US export ban, and capable of 93 petaflops, was officially considered to be the world's fastest. In June 2018 IBM and the USA's Department of Energy announced that they had developed a computer capable of 200 petaflops, thus taking the title of the world's fastest computer from the Chinese.

Exascale computing is a current focus of international research and development. An exascale computer is defined as one that could reliably carry out at least one exaflop, i.e. 10^{18}, of floating point operations per second. Countries including China, Japan, Russia and the USA are researching their development.

2.6 Overview of chapter

In this chapter we gave an overview of the historical development of general-purpose computers, including historical characters from Pascal to von Neumann who played a significant role in their development. We considered how to define a computer, discovered why the basic computer architecture of general-purpose computers is called the von Neumann architecture, discussed the motivation for using the binary system, and finally considered the development of general-purpose computers since the 1940s in terms of computer generations, and in terms of the different types of general-purpose computers.

2.7 Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- outline the history of computers
- explain the von Neumann architecture
- explain the advantages of using the binary system in computing.

2.8 Test your knowledge and understanding

2.8.1 Sample examination question

This question is in three parts, a, b and c.

(a)

(i) What was the name of first general-purpose computer, designed by Charles Babbage? [2 marks]

1. Analytical engine
2. Difference engine
3. Colossus
4. ENIAC

(ii) Which one of the following was the world's first programmable electronic digital computer? [2 marks]

1. Analytical engine
2. Difference engine
3. Colossus
4. ENIAC

(iii) ENIAC stands for: [2 marks]

1. Electronic Numerical Integrator And Computer
2. Electronic Numerical Integrator And Calculator
3. Electronic Numerically Implemented Automatic Computer
4. Electronic Numerically Implemented Automatic Calculator

(b)

(i) List the components of the von Neumann architecture. [5 marks]

(ii) With each generation of computers processing power has increased. What else has increased/decreased significantly with each successive generation? [4 marks]

(c)

(i) Who proposed and formulated a theory and practice for the use of binary logic in computers? [1 mark]

(ii) Name and describe the technology that distinguishes second-generation computers; compare it to the technology that it replaced; and explain why the new technology increased processing speed. [5 marks]

(iii) How did the new second-generation technology favour the use of binary numbers? Give two advantages of using binary numbers rather than decimal. [4 marks]

Notes

Chapter 3: Data storage and processing

3.1 Introduction

Information can be stored in various ways. Books contain a great deal of information/data; as do films and paintings. Even rocks contain a lot of information about geological changes. However, information would not be information if it could not be used. Information in computers must be capable of being processed by computers. In order for computers to process information, it must be represented in appropriate formats and stored in appropriate places.

From the preceding chapter we know that one very important breakthrough in the development of computers was the use of the binary system. In the binary system, there are only two types of values, 1s and 0s. Within computers both information (data) and instructions are represented as binary numbers. This chapter will consider how binary data and instructions are constructed, stored and processed, and the physical media and technology used to do so.

3.1.1 Aims of the chapter

This chapter aims to describe the fundamentals of memory construction, processing and organisation in computers. The chapter further aims to introduce and define important terms for describing memory and its organisation. The chapter also aims to familiarise you with the media used to store the data used by computers, which includes:

- electronic memory
- flash memory
- magnetic memory
- solid state memory
- RAID
- optical memory.

Note: Electronic memory is also called **main memory**; storage media is often referred to as **secondary storage**.

3.1.2 Learning outcomes

By the end of this chapter and having completed the Essential reading and activities, you should be able to:

- outline the difference between electronic memory, magnetic memory, solid state memory and optical memory
- explain how data are stored and accessed in these types of memories
- explain how memory is organised in terms of cells, words and addresses and be aware of the logic gates used to process data
- demonstrate understanding of the main terms concerning memory; for example, memory capacity, access time, transfer rate
- explain how the address decoder works.

You are not required to know the details of multiplexed addressing.

3.1.3 Essential reading

- Stallings, W. *Computer organization and architecture: designing for performance.* (Harlow: Pearson Education Ltd, 2016) 10th global edition [ISBN 9781292096858]: Chapter 5 'Internal memory' and Chapter 6 'External memory'.

3.1.4 Further reading

- Harris, S.L. and D.M. Harris *Digital design and computer architecture.* (Waltham, MA: Elsevier/Morgan Kaufmann, 2016) ARM edition [ISBN 9780128000564]:
  - Section 1.5 (logic gates)
  - Sections 5.5–5.5.6 (memory).
- Patterson, D.A. and J.L. Hennessy *Computer organization and design: the hardware/software interface.* (Amsterdam: Elsevier/Morgan Kaufmann, 2017) ARM edition [ISBN 9780128017333]: Section A.3 (page A-9), *'Combinational Logic* (decoders)'.

3.2 Glossary of key terms

Before we look at data storage, we will define some key terms.

3.2.1 Primary and secondary memory

Main memory holds data and instructions that the CPU is using and is volatile – when the power is switched off the memory is lost. Non-volatile memory, such as CDs, are used to store data and instructions so that main memory can access them again, the next time it is turned on. Main memory is sometimes called primary memory; and non-volatile memory is called secondary memory.

3.2.2 Read-only memory (ROM)

Read-only memory provides a permanent data store that cannot be changed. ROM chips need no power source to keep them alive, and are loaded into main memory. The disadvantage of ROM is the cost to produce – since the data is fixed and cannot be rewritten, a mistake in manufacture has fatal consequences, adding to the cost to produce. Stallings (2016) (Chapter 5, Section 5.1 'Semiconductor main memory') discusses ROM and some technological solutions to the problem of the expense of producing ROM chips. Harris and Harris (2016, Section 5.5.1, under heading 'Memory types') note that ROM memory is so-called for historical reasons that have lost some of their meaning today with changes in technology. You are not required to know the details of ROM memory.

CD-ROMs also implement read-only memory, in that once written to they cannot be overwritten.

3.2.3 Memory capacity

How much information main memory can store depends on the number of cells; each cell holds one bit of information; by common agreement, 8-bits make one byte. Computer memory capacity has been commonly measured in bits and bytes, but this led to some ambiguity, as it could be unclear whether memory capacity was being measured in base 10 or base 2. For example, a kilobyte was sometimes used to mean 10^3, or 1,000 bytes, and sometimes used to mean 1,024, or 2^{10} bytes. As terms got larger, the difference in size between them also increased both numerically and absolutely; for example, a gigabyte could mean 1,000,000,000 or it could mean 1,073,741,824.

The ambiguity came about because while computer memory is measured in powers of 2, kilo, under the International System of Units (SI), means one thousand.

To resolve this ambiguity, the following terms are now preferred:

Decimal name	Abbreviation	Value	Binary name	Abbreviation	Value
kilobyte	KB	10^3	kibibyte	KiB	2^{10}
megabyte	MB	10^6	mebibyte	MiB	2^{20}
gigabyte	GB	10^9	gibibyte	GiB	2^{30}
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}

Table 3.1 Table of computer memory size in decimal and binary.

Kilobyte is still used sometimes to mean 1,000 and sometimes to mean 1,024. Sometimes Kb is used to mean 1,000 and KB 1,024, but to avoid any ambiguity it is best to use KiB when 1,024 is meant.

3.2.4 The system bus

In a computer, main memory is connected to the CPU via three sets of wires (lines):

- address lines
- data lines
- control lines.

These three sets of lines are also referred to as the address bus, the data bus, and the control bus, respectively. These three buses together form the **system bus**.

Increasingly, the bus is being replaced with 'point to point interconnection' (Stallings 2016, Section 3.5). You should be aware of this trend, but this guide will consider the connection of the CPU to other components to be via a **system bus**.

3.2.5 Word

A word is the unit of organisation of memory. The data bus is normally as wide as the word length, allowing a word to be moved between main memory and storage in one operation. Both instructions and numbers are words, since computers store both data and instructions in binary; for this reason a word is typically equal to the number of bits allocated to represent an integer, and to the instruction length. Word size in a particular computer is fixed. Processors today have a word size of 8, 16, 24, 32, or 64 bits, with general-purpose computers normally having 32 or 64-bit word sizes.

3.2.6 Address space

The range of addresses used to reference main memory. Can also refer to the range of addresses used to reference I/O modules.

3.2.7 Addressable memory

The smallest unit that can be referenced by an address is usually the word, although sometimes it is the byte. Hence memory is either word addressable, or byte addressable (note that a computer with a word size of a byte is considered to be word addressable). According to Stallings (2016, Section 4.1), 'The relationship between the length in bits *A* of an address, and the number *N* of addressable units is $2^A = N$.' This allows every addressable memory unit to have a unique address.

Activity 3.1

Suppose that the total size of main memory is 1GiB.

1. What is the length of each address in bits if the memory is byte addressable?
2. What is the length of each address in bits if a word is 32 bits, and the memory is word addressable?
3. Is the memory space smaller or larger when the memory is byte addressable versus word addressable with a 32 bit word?

Activity 3.2

1. A computer's memory is composed of 8K words of 32 bits each. How many total bits in memory?
2. A computer's memory is composed of 4K words of 32 bits each, and the smallest addressable memory unit is 8 bits (one byte). How many bits will be required for the memory address?

3.2.8 Most and least significant bit

In any bit pattern, the bit with the greatest numerical value is the most significant bit (MSB). Similarly, the bit contributing least in numeric terms would be the least significant bit (LSB). In the bit pattern **1**000 001**0**$_2$, the most significant bit contributes 2^7 to the numerical value of the bit pattern. The least significant bit contributes 0.

3.3 Main memory

The 1s and 0s of binary numbers can be physically represented in different ways:

- **1** can correspond to a high voltage or an electrical current in a transistor, a magnetised spot, or a transparent optical area
- **0** can correspond to a low voltage or the absence of an electrical current in a transistor, an unmagnetised spot, or an opaque optical area.

Main memory is used to store programs and data that are currently being used by the CPU. If a program is to be run, it is first loaded into main memory, because main memory is fast. Main memory is highly **volatile**. Its content changes frequently, as different programs are run at different times. When the power is off, all data in main memory is lost. So main memory is not for long-term storage. Main memory is electronic memory, because it is based on electronic principles and implemented with transistors.

3.3.1 DRAM and SRAM

'Each addressable location in memory has a unique, physically wired-in addressing mechanism' (Stallings, 2016, Section 4.1). If the CPU wants to read/write a particular memory location, it puts the address of the location – a series of 1s and 0s – on the address lines; the memory location will then be activated and the read/write operation will be done. The time to access any memory location is constant, and is independent of any previous memory access. This means that any location can be chosen at random, and accessed within the same time as any other location. This method of accessing information is called **random access**. And for this reason, main memory is often referred to as **random-access memory**, or RAM.

Main memory is normally composed of **dynamic random-access memory**, DRAM, called dynamic because the charge it holds will leak away without periodic refreshing (since it is physically implemented using capacitors). **Static random-access memory** (SRAM) does not need refreshing, and is faster and more expensive than DRAM.

3.3.2 The cell

The fundamental building block of main memory is the cell. This is an electronic circuit that stores one bit of binary information – either a 1 or a 0.

We will describe the technology used to build cells in DRAM and SRAM.

3.3.3 Transistors

The smallest unit of an electronic memory is the **transistor**. A transistor is a semiconductor device with three electrodes:

- emitter
- base
- collector.

Figure 3.1 shows a symbolic representation of a transistor. The electrical characteristics of the semiconductors cause a transistor to act as an electrical switch. When the correct voltage is applied to the base, there will be an electric current flowing from the collector to the emitter. In the absence of such a voltage on the base, there will be no such current.

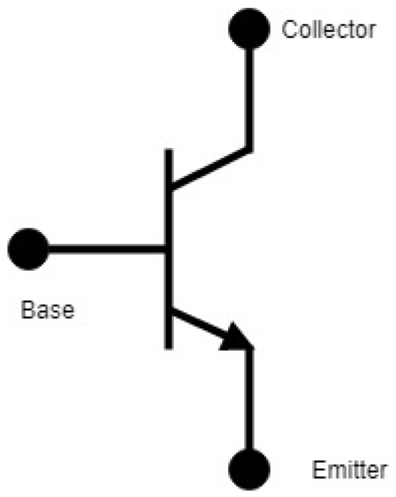


Figure 3.1: Transistor symbol.

3.3.4 Flip-flops

A **flip-flop** is a special type of circuit, Figure 3.2 shows a typical computer flip-flop. This device has two inputs, **control** and **data-in**, and one output, **data-out**. When control is turned on (i.e. when control = 1) we have data-in = data-out. If control is then switched off (i.e. when control = 0) data-out will not change, no matter how often data-in changes. This means that when control is turned off, the device remembers the last value of data-in when control was on (the value can be obtained from data out, which does not change, even after control is switched off). A flip-flop is really a 1-bit memory, because it can store a 1 or a 0.

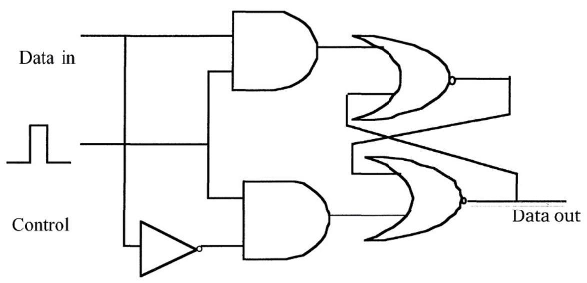


Figure 3.2: A controlled flip-flop.

This means that if we want to write a datum (1 or 0) into a flip-flop, we only need to send the datum to the data-in point, and at the same time switch the control on (by sending a 1 to the control point). The datum (1 or 0) will be stored in the device. If later on we want to write another datum (1 or 0) into the device, we only need to repeat the above process. What datum is stored in the device at a time can always be obtained, or read, from the data-out point.

3.3.5 The physical principles of electronic memory

DRAM cells are implemented with a transistor (acting as a switch) and capacitor to hold the charge. Flip-flops are built from logic gates which themselves are implemented using transistors. Since many transistors may be needed to build a flip-flop they are larger and more expensive than DRAM cells, but are also significantly faster, and are used for the processor's registers, and for SRAM.

Since semiconductor transistors make up main memory, it can also be called semiconductor memory.

3.4 Logic gates

Memory is provided by cells and processed by **logic gates**. Logic gates are electronic circuits built to implement Boolean and other logical operations, and are formed from groups of transistors.

Figure 3.3 shows an AND gate formed by combining two transistors (in a highly idealised manner). In this device, when the appropriate voltage is applied to the bases of both transistors, there will be an electric current flowing from the collector of one transistor to the emitter of the other. This electric current will not exist if the appropriate voltage is absent from the base of either transistor.

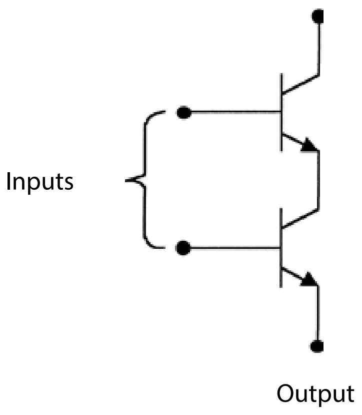
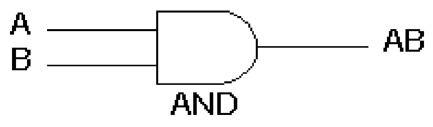


Figure 3.3: One AND gate made from two transistors (highly idealised).

Let us say that when there is (or is not) an appropriate voltage on the base of a transistor, the input is 1 (or 0), and that when there is (or is not) an electric current flowing from a collector to an emitter, the output is 1 (or 0). Then we can describe the function of an AND gate using the table in Figure 3.4, which says that the output is **1** if and only if both inputs are **1**.

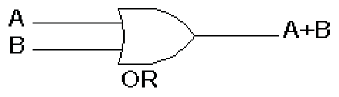


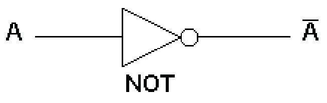
A	B	A x B
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3.4: AND logic gate with function table.

When we talk about inputs/outputs as 1s or 0s, we are abstracting away from voltages and electric currents. On this abstract level, we can speak of an AND gate as a device having two inputs and one output, without mentioning its internal structure. So we can symbolise an AND gate as in Figure 3.4.

Similarly, we can build OR gates and NOT gates using groups of transistors. In an OR gate, the output will be 1 if either input is 1, and 0 if both inputs are 0. In a NOT gate, there is only one input, and the output is the negation of the input.





A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

A	-A
0	1
1	0

Figure 3.5: OR and NOT logic gates with their function tables.

Since AND, OR and NOT are often called logic connectives, AND gates, OR gates and NOT gates are also called logic gates. With these three types of gates, circuits of various degrees of complexity can be built.

More complicated logic gates, such as NAND gates and NOR gates, can also be built:

- A NAND gate is a combination of an AND gate and a NOT gate.
- A NOR gate is formed by joining an OR gate and a NOT gate.

3.5 Data organisation and access in main memory

We have been looking at how information is stored in an electronic memory. Now let us see how the information in an electronic memory is organised and accessed. The CPU handles data in words, and instructions and data are stored in main memory as words in a serial order. The CPU executes instructions one by one top-down. We will look at how the CPU works in detail in the next chapter.

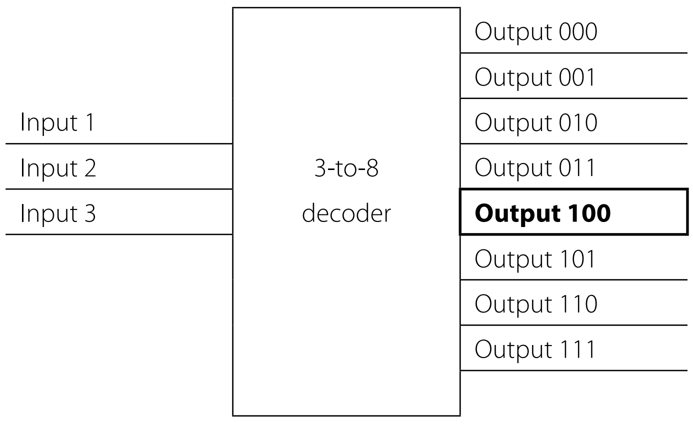
3.5.1 Decoders

Any system with an address bus has need of an address decoder. Recall that main memory is connected to the CPU via the address bus, the data bus, and the control bus. An address bus has a number of lines; addressing means putting a signal on each of the address lines at the same time. The signal is binary, corresponding to a 1 or a 0. Each time a memory location is addressed, all of the address lines (the inputs) are used. Every addressable memory unit must have its own address, and a decoder between the address bus and the CPU reads this address.

In address decoding the most significant bits of the address are used to generate the address of a device, known as 'chip select' signals. The least significant bits of the address are used as addresses to memory locations or registers on the device. A decoder with 2 input lines can have $2^2 = 4$ output lines; a decoder with 3 input lines can have 8 outputs. Three input lines give 8 potential inputs corresponding to the binary numbers 000, 001, 010, 011, 100, 101, 110 and 111. In general a decoder with *n* inputs, can have 2^n outputs; namely, it can address 2^n devices.

3.5.2 RAM chips and address decoders

Suppose that a computer's address bus has 10 lines, and the computer has 8 memory chips. An address decoder sits between the address bus and the chips. The first 3 most significant bits of the address will be sent to a 3-to-8 decoder, which will determine which of the 8 chips to select.



Assume input 1 = 1, input 2 = 0 and input 3 = 0, output line 100 is asserted, all other outputs are false.

Figure 3.6: A basic model of a 3-to-8 decoder.

Suppose the address was **100**1010111. The three most significant bits, 100, would be placed on the lines that connect to the 3-to-8 decoder, which would have three lines in, and 8 lines out. The output line to the chip selected by 100 would be asserted; that is, become true (corresponding to a '1'), and all other output lines would become false (or 0). The least significant bits, 1010111, would be given to the chip as the address of the memory location. Hence 3 address bus lines would be used for chip select, and 7 for the chips themselves to address their cells.

We have been describing main memory as if it were just one single piece of memory. In fact, main memory is made up of a number of memory chips. One very common size for a RAM chip is 256 x 8 bits. That means that the chip can be conceptualised as being 8 bits wide, with 256 of these 8-bit memory locations. Hence the chip has the capacity to store 256 x 8 bits, or 256 bytes.

In address decoding the most significant bits of the address are used to select the chip. The least significant bits give the address of the memory locations on the chip.

Let us consider a memory composed of four 256 x 8 RAM chips. With 4 chips, we would have a memory capacity of 4 x 256 bytes, 1024 bytes, or 2^{10} bytes. Let us say that the memory is byte addressable. With 2^{10} bytes we need 10 bits for the address of each memory location. Each chip has 256 8-bit memory location, or 2^8 bytes, so each chip needs 8 address lines. Therefore in the 10-bit address, the 2 most significant bits would be used to specify the chip, and the final 8 bits the address of the memory locations on the chips.

How would this be different if the memory were word addressable? If the word was 32 bits long, then the number of addressable memory locations (the address space) would be:

$(4 \times 256 \times 8)/32$

$= (2^2 \times 2^8 \times 2^3)/2^5$

$= 2^{13}/2^5$

$= 2^8$.

Recall that in arithmetic with powers to the same base number (in this case, 2) multiplication is achieved by adding the powers together hence:

$2^8 \times 2^3 \times 2^2 = 2^{8+3+2} = 2^{13}$.

Division is accomplished by subtracting the power of the denominator from that of the numerator, hence $2^{13}/2^5 = 2^{13-5} = 2^8$.

How is it possible to address a word consisting of 4 bytes, with only 8 bits in the address, when 8 bits can only specify the memory locations on each chip? That is, if the address can only specify one 8-bit memory location, how can a 32-bit word be addressed? The answer is that the chips are grouped together, and so the processor does not see 4 chips, but 256 32-bit memory locations. Each word has one byte on each chip, and each byte of the word has the same address. So if we wished to access word *W*, with address $0000\ 0000_2$, this would mean that the first byte of the word is in memory location 0000 0000 on the first chip; the second is in memory location 0000 0000 on the second chip; the third byte is in address 0000 0000 on the third chip; and the final byte is on the fourth chip at the same address.

If the memory was larger – for example, if we had eight 256 x 8 RAM chips with a 32-bit word – then the size of our address space would be:

$(8 \times 256 \times 8)/32$

$= (2^3 \times 2^8 \times 2^3)/2^5$

$= 2^{14}/2^5$

$= 2^9$.

This would mean that the most significant address bit would be for chip selection, and the final 8 bits for the memory locations on each chip. In this case, the processor would see each block of 4 chips as one entity. One input, corresponding to the output 0 or 1, would be sufficient to specify either the first or the second block of 4 chips.

With a word size of 16 bits, chips would be paired, with each pair of 2 chips storing 256 words. The first byte of a word *W* would be on the first chip of the pair, and the second byte would be on the second chip, addressed by the same bit pattern as the first.

Whether we view main memory as a single chip or as a collection of chips, memory locations are arranged in a serial order. Each has a unique address, and each address uses all the computer's address lines. If main memory is big, then a large number of address lines will be needed. In order to reduce the number of address lines in a computer, we need a multiplexer, which this course does not cover.

Activity 3.3

Suppose that a 2MiB x 16 bit word main memory is built using 256KiB x 8 RAM chips and memory is word-addressable.

1. How many RAM chips are necessary?
2. How many RAM chips are there per memory word?
3. How many address bits are needed for each RAM chip?
4. How many address bits are needed for all of memory?

3.5.3 Transfer rate

Transfer rate is the amount of information that can be transferred per second between one place and another. In the case of main memory, this refers to the rate of information exchange between main memory and the CPU, and is equal to 1 divided by the cycle time. The transfer rate from main memory to the CPU can be many millions of bits per second. What about the transfer rate in magnetic memories? In the case of magnetic memories, transfer rate refers to the speed of transferring information between a magnetic memory and main memory. Stallings (2016, Section 4.1) gives the following transfer rate for non-RAM memory:

$$T_n = T_A + \frac{n}{R}$$

where:

T_n = average time to read or write n bits

T_A = average access time

n = number of bits

R = transfer rate in bits per second (bps)

First, we must understand how data is transferred from a magnetic memory to main memory. Let us consider hard disks. Suppose that some data is stored in some sectors on a disk.

Transferring data involves at the least the following two activities:

1. The read/write head needs to go to those sectors and read the data.
2. The data must be put on the data lines, and the addresses of certain words in main memory need to be put on the address lines; consequently, the data will be written into those words in main memory.

Activity (2) above mainly involves electronic signals, so it is very fast. By contrast, Activity (1) involves a lot of mechanical movement, so it is much slower. So how fast data can be transferred from a disk to main memory really depends on how fast Activity (1) is. Activity (1) depends on the rotational speed of the disk; the number of sectors per track; the number of bytes per sector; the speed of the head movement mechanism, and so on.

3.5.4 Access time

Access time is the time taken to get ready to read some information from a certain part of memory, or to get ready to write some information to that part. In the case of main memory, access time is the time taken between the moment when the CPU wants to read/write a memory location; and the moment when the location is activated. We know that this time must be very short, because it depends mainly on how fast electronic signals travel in wires (e.g. address lines); and electronic signals travel at nearly the speed of light (3.08×10^8 metres/second). Typical access time in main memory is about 60ns (nanoseconds, 10^{-9} seconds), while for SRAM this can be closer to 10ns.

In general, the access time for disk drives is on the scale of milliseconds (10^{-3} second) and main memory is on the scale of nanoseconds (10^{-9} seconds). It is easily seen that accessing data in main memory is much faster, about one million times faster, than accessing data in disks.

Disk drives provide random access to memory, in that any addressable unit can be found, on average, as quickly as any other. Tape drives are sequential access devices, and hence have the slowest access time, and it is not the case that any memory location can be found as quickly as any other. The reading/writing head has to move to the right place for reading/writing by moving through the tape, backwards or forwards, as necessary; the time taken to do this will vary with the tape position required.

3.5.5 Transfer time for a disk

Transfer time for secondary storage may be complicated by I/O delays, but in general, ignoring I/O waits, it is given by the access time plus the time to read/write the data. Access time for a disk is the time taken to position the reading/writing head in the right place. The head needs first to move to the right track and then to wait for the right sector to come under it. The times taken are called **seek time** and **rotational latency**, respectively. Access time is then the sum of the average seek time and the average rotational latency.

The average rotational latency is calculated by:

(minimum latency + maximum latency)/2.

The minimum latency is 0; this corresponds to the case where the head happens to be above the desired sector. The maximum latency is the time taken for the disk to rotate once; this corresponds to the case where the head just missed the desired sector, and it has to wait for the sector to spin around again. Hence the average rotational latency is the time for half a rotation.

The average seek time is more difficult to find. It depends on the speed of the head movement mechanism, and the number of tracks that need to be crossed.

The average rotational latency = $1/2r$, where r is the rotation speed in revolutions per second. This just means that to find the average rotational latency we first find the rotation speed in seconds, then find the number of rotations (revolutions) in one second by $1/r$, then divide the result by 2 to get the time for half a rotation, giving $1/2r$.

The time to read or write is:

$$\frac{x}{rN}$$

Where x is the number of bytes to be transferred and N is the total number of bytes on a track. Note that instead of bytes we could use a different unit of transfer, bits, or sectors containing many bytes. Provided that both x and N are measuring the same quantity, we should get the same result. That is, if x means sectors, then N means the number of sectors on a track.

Hence if T is the transfer time, and S is the seek time:

$$T = S + 1/2r + x/rN$$

$T =$	transfer time
$S =$	seek time
$r =$	revolutions per second

x = total units to be read or written (e.g. units could be bits, bytes or sectors)

N = total number of units on one track

Activity 3.4

Consider a disk system, which has a track seek time of 10 ms (milliseconds). The disk rotation speed is 12,000 rpm (revolutions per minute), and each track on the disk has 600 sectors. Given that each sector has 512 bytes of data, and that data is sequentially organised, what is the average time it takes to read 1,024 bytes of data? Assume in your answer that the system I/O can process data as it is read with no queuing delays. Give your answer in milliseconds. Keep notes of your working.

Access time can be very different for sequentially stored and non-sequentially stored data. If the data is stored non-sequentially, then each addressable memory unit has to be first accessed, then read. For example, assume that we wish to read 250 sectors from a disk, with a seek time of 5 ms and rotational delay of 2ms. The data is sequentially organised. Suppose that the time to read one sector was 0.005 ms, then the transfer time would be 5 + 2 + (0.005 x 250) = 5 + 2 + 1.25 = 8.25 ms.

Now suppose that each sector is in a different place on the disk. The time to read one sector would be 5 + 2 + 0.005 = 7.005 ms. The time to read 250 sectors would be 250 x 7.005 ms. This is 1751.25 ms, or 1.75125 seconds.

3.6 Flash memory

Main memory is semiconductor memory, as it is comprised of semiconductor chips. Another form of semiconductor memory is flash memory, used as internal and external memory. There are two types of flash memory, NOR and NAND.

NOR flash memory is bit addressable, with cells connected in parallel so that each cell can be individually accessed. NOR memory provides high speed random access and is used for internal memory in embedded systems (that is, systems that reside in products such as cars and washing machines). NAND has higher density and greater write speed, but data must be read in blocks which can be very large. NAND flash memory is used in USB flash drives; memory cards for cameras and other portable devices; and solid state disks.

3.7 Magnetic memory

3.7.1 Physical principles of magnetic memory

In main memory a 1 corresponds to a high voltage, or the presence of an electrical current, and a 0 corresponds to a low voltage or the absence of an electrical current. Information can be represented in other ways within the binary framework, as long as we can distinguish between 1s and 0s.

Magnetic memory is another way of storing information in the binary framework. The basic idea is that a magnetised spot is seen as storing data 1, and an unmagnetised spot is taken to hold data 0. A magnetic memory contains a large number of spots. This is how it stores information. An unmagnetised spot can be magnetised and a magnetised spot can be demagnetised. The job of magnetising or demagnetising a spot is done by a device called a 'read/write head', which also does the job of reading data from a spot. A typical read/write head uses an electromagentic field to magnetise or demagnetise a spot, thus writing a 1 or a 0 into a spot. See Stallings, Section 6.1, 'Magnetic read and write mechanisms' for more details.

We shall now look at: hard disks and magnetic tapes.

3.7.2 Hard disks

A hard disk consists of one or more disk platters, attached to a central spindle. A platter may have a single or a double surface, with a separate read/write head for each surface. Previously systems would have a single read/write head, but Stallings (2016) Section 6.1 Magnetic Disk (under the heading Magnetic Read and Write Mechanisms) notes that 'in many systems there are two heads, a read head and a write head' since '[c]ontemporary rigid disk systems use a different read mechanism, requiring a separate read head'. Hard disks are kept in a sealed metal box allowing for fast spinning, when reading or writing the head stays still while the disk rotates. If there are multiple disks, all disks rotate at once.

A typical hard disk has thousands of tracks on each platter's surface, arranged as concentric circles. A track is divided into sectors, and each sector stores a block of data, and a block has a fixed size, normally either 512 or 4096 bytes. Each track has the same number of sectors, however tracks, and sectors, become shorter as they approach the centre of the disk. Sectors have gaps between bits that decrease in size as the centre of the disk is approached. The innermost track has no gaps between bits. This means that the disk can spin at a constant rate, and will read the same amount of data wherever the track it is reading is located on the disk. This constant spin speed is known as **constant angular velocity (CAV)**.

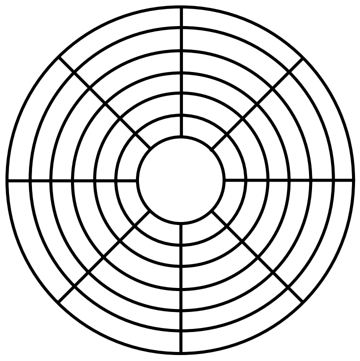


Figure 3.7: A simplified schematic of a CAV hard disk's tracks and sectors. Note that the space between bits decreases as sectors decrease in length, such that all sectors contain the same number of bits. Hence the bit density is greater in shorter segments.

The disadvantage of CAV is that gaps between bits mean that storage space is wasted. Stallings (2016, section 6.1) states that:

> disk storage capacity in a straightforward CAV system is limited by the maximum recording density that can be achieved on the innermost track. To maximize storage capacity, it would be preferable to have the same linear bit density on each track. This would require unacceptably complex circuitry.

Stallings then describes multiple zone recordings (MZR) in modern hard disks, which allow a variable number of sectors per track, once again each sector stores a block of data, with a block having a fixed size. MZR divides tracks into zones, tracks in the same zone have the same number of sectors per track, and each track within a particular zone has the same number of bits. Tracks in zones are divided into sectors such that the outermost zones have more sectors per track that the inner zones. This means that tracks of about the same length can have the same bit density. This increases the bit density of the disk, and hence the storage capacity, 'at the expense of somewhat more complex circuitry' since the time for reading and writing is the same for each sector within a zone, but differs between zones (Stallings 2016, section 6.1, heading Data Organization and Formatting).

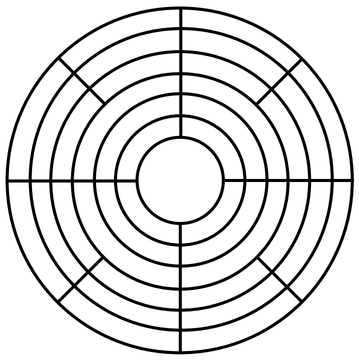


Figure 3.8: A simplified schematic of a MZR hard disk, with tracks divided into two zones. In Figure 3.8, the outer zone has eight sectors per track and the inner zone has four sectors per track.

For more information on the structure and organisation of hard disks, see Stallings (2016), Section 6.1 Magnetic disk and the text and graphics under the heading 'Data organization and formatting'.

3.7.3 Magnetic tapes

A tape system uses the same reading/writing techniques as hard disks. A tape is coated with magnetic oxide. It is divided into a number of parallel tracks, and each track is further divided into a number of squares; each square can store 1 bit of information (1 or 0). Earlier tape systems typically used nine tracks. This made it possible to store data one byte (8 bits) at a time, and the extra bit was used as a parity bit for the purpose of error detection. After this, tapes used 18 or 36 tracks.

Magnetic tapes are still used as a low cost way of backing up data with large storage capacity and excellent reliability.

3.7.4 Memory capacity of magnetic memory

Magnetic memories are used to store a large quantity of data. Unlike main memory, data in a magnetic memory will not be lost if the power is switched off. Hard drives are used for storage in computer systems, and tapes are used for backups.

3.7.5 Accessing magnetic memory

Recall that main memory is a random-access device. Accessing a memory location need not be done in any sequential order, and the time taken to access any location is constant. This is because to access any location, the CPU needs only to put the address (a series of 1s and 0s) on the address bus and the memory location will be activated.

What about disk drives and tape drives? A tape drive is not a random access device; rather, it is a **sequential**-access device. If the tape head is positioned at block 1, then to read block N, it is necessary to read blocks 1 through N–1, one at a time. If the head is currently positioned beyond the desired block, then the tape has to be rewound a certain distance and reading has to start again. So, different blocks will take different times to access.

Now, let us look at disk drives. One might think that they are sequential-access devices, because to access a sector the read/write head has to move to the right track and wait for the desired sector to spin to the position underneath the head. One might therefore think accessing different sectors will take different amounts of time. It is right to say that accessing different sectors takes different amounts of time, but the times do not form a linear order. More importantly, to read a sector on a track, the head does not need to read all the tracks before it, including sectors on other tracks before the track in question. The head can move directly to the desired track. For this reason, disk drives are also called random, or direct, access devices.

3.7.6 Other features of magnetic memory

Hard disks are enclosed in an airtight case to prevent dust and other contaminates from jamming the microscopic space between the read/write heads and the platters on which data is stored. Hard disks are usually fixed inside a computer.

Magnetic memories (disks and tapes) are cheap to produce, and can store mass data.

Activity 3.5

What access method is used to retrieve data from a magnetic tape?

3.8 Solid state memory

Solid state drives (SSDs) are now being used to supplement and – in some cases – replace hard drives. SSDs are made with semiconductors and the newer ones use NAND flash memory. Their advantages over hard disks are that they significantly increase the speed of I/O processing, are more durable, use less power, are quieter, do not have the same need for cooling as hard disks, use less space and have improved access times. In contrast, hard disks have lower cost per bit and greater capacity.

However, solid state memory can slow as a device is used because of the way that memory is accessed as a block, usually of 512KB, giving 128 pages per block. To write a page to memory the entire block must be transferred. When the memory is empty this is not problematic, but as time goes by and memory fills up, pages can no longer be written contiguously, leading to greater and greater fragmentation and hence slower and slower writing to memory.
SSDs share a second issue with flash memory, in that flash memory becomes unstable after a certain number of writes, Stallings (2016, Section 6.3) gives a typical limit as 100,000 writes.

3.9 RAID

RAID stands for Redundant Array of Independent Disks and is an industry agreed standard for organising data across multiple disks to improve reliability and aid data recovery. RAID consists of a number of disk drives (there are different protocols) that the operating system will perceive as a solitary drive. Data is stored in such a way that data recovery is guaranteed if a single disk should randomly fail. Data recovery is partly achieved by striping – storing related data in strips across the different disks. If one disk should fail, gaps in the data can be filled in using error correcting codes or a parity bit to calculate the missing bits.

3.10 Optical memory

3.10.1 Physical principles of optical memory

In main memory, a 1 corresponds to a high voltage or the presence of an electrical current, and a 0 corresponds to a low voltage or the absence of an electrical current. In magnetic memory, a 1 is stored in a magnetised spot (on a disk or a tape), and a 0 is stored in an unmagnetised spot. When data is read from a magnetic memory, the read/write head turns the information about whether spots are magnetised or blank into the information about the presence or absence of electrical currents. This latter information is then put on to the data lines and is consequently written into main memory.

Optical memory is another way of storing binary data. A typical type of optical memory is the CD-ROM (compact disk read-only memory). A CD-ROM is a disk that is formed from a resin, such as polycarbonate, and is coated with a highly reflective surface, usually aluminium. The surface of the disk is divided into a large number of tiny areas. Some of the tiny areas are flat, and they are called **lands**. Other tiny areas are depressed into **pits**. Light that strikes a land is reflected directly back, and the strong reflection will cause a photo sensor to generate a certain small electrical voltage. Light that strikes a pit is scattered, so the reflection will not be strong enough to generate such a voltage in the photo sensor. Thus, lands can be said to store data 1, and pits data 0.

Information is retrieved from a CD-ROM by a low-powered laser housed in an optical-disk driver. The laser shines through the clear protective coating of the spinning disk. The intensity of the reflected light of the laser is different depending on whether lands or pits are encountered. The differences are detected by a photo sensor and are converted into a series of 1s and 0s.

3.10.2 The organisation of a CD-ROM

A CD-ROM has only one long spiral track, divided into sectors of 2,048 bytes. This is because CDs were originally designed to play music, smoothly and continuously; hence gaps – however brief – when the reading head moves to a new track, were not wanted. Each sector has the same length and hence the same bit density.

3.10.3 Accessing data on a CD-ROM

The disk drive varies the rate at which the disk is spinning depending on which part of the disk is being read. Since data has the same bit density wherever it is placed on the CD-ROM, if the spin speed did not vary then data closer to the outer edge of the CD would be read at a faster rate than data closer to the centre of the disk. This is because with a constant spin speed the outer edge moves faster than the centre. To achieve a constant data transfer rate, the spin speed must be varied. Hence the disk rotates slowly for access near the outer edge, and more quickly for access near the centre. This is called **constant linear velocity** (CLV). This is because if the track were to unwound into a straight line, which would be about five kilometres long, then each part of the line would be traversed at the same speed by the reading head.

3.10.4 Features of optical memory

Optical disks have a high storage capacity. A typical CD-ROM can store 650MB. Optical disks are portable, they can be removed from the drive, are reliable (much more reliable than the first widely used portable storage, floppy disks) and have a long life.

Access time for CDs is quite long, up to half a second. This is due to the complexity of locating a particular sector: the head must first move to the general area, the disk drive must adjust the rotation speed of the disk, and some minor adjustments must be made in order to find the specific sector. Optical disks are still random-access devices, because finding a sector does not have to be done in any sequential order.

3.10.5 Types of optical memory

- CD – a disk that stores audio.
- CD-ROM – a disk that stores digital information, up to 650 MB.
- CD-R – the user can write once, read many times.
- CD-RW – the user can write to the disk many times, information can also be deleted. Has the advantage over magnetic disks of greater reliability and longer life.
- DVD – digital versatile disk. Can store video and other digital information, with capacity up to 17 gigabytes. Usually read-only. Greater storage space than a CD because bits are packed more closely; a second layer of pits and lands is applied on top of the first layer, nearly doubling capacity, and the DVD can be double sided, giving another doubling of storage.
- DVD-R – the user can write to the disk once.
- DVD-RW – the user can write to the disk, and can delete information, as often as they wish.
- Blu-ray DVD – for video information. Can store up to 25 gigabytes. The name comes from the colour of the laser used to read the disk. Originally there were two types of high definition optical disks, but Blu-ray has come to dominate the market. Blu-ray disks can have smaller pits and tracks (and thus greater storage) because the laser is closer to the data layer.

3.11 Future forms of memory

Future storage methods may include holograms, which will expand the capacity of PC storage to the scale of terabyte – 1,000 gigabytes.

Data might also be stored in biological forms. But there is still a long way to go before biological memory and biological computing can become a reality.

3.12 Overview of chapter

In this chapter we looked at how data is constructed in main memory, in terms of cells. After considering how main memory is constructed from transistors, we looked at how data in main memory is stored in words and/or bytes, and how data is addressed and accessed, considering the size of the address space and the role of the address decoder. We then looked at transfer rates and transfer times concluding with an overview of the different types of secondary storage.

Information about transistors and the construction of cells, flip-flops and logic gates is provided to aid understanding, but is not examinable.

3.13 Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- outline the difference between electronic memory, magnetic memory, solid state memory and optical memory
- explain how data is stored in these types of memories
- explain how memory is organised in terms of cells, words and addresses and how data is processed using logic gates
- demonstrate understanding of the main terms concerning memory; for example, memory capacity, access time, transfer rate
- explain how the address decoder works.

You are not required at this stage to know the details of multiplexed addressing.

3.14 Test your knowledge and understanding

3.14.1 Sample examination question

This question is in three parts, a, b and c.

(a)

(i) A memory in which any location can be reached in a fixed average amount of time after specifying its address is called: [2 marks]

1. Sequential-access memory
2. Random-access memory
3. Secondary memory
4. Mass storage

(ii) In a 32-bit machine, the length of each word will be: [2 marks]

1. 4 bytes
2. 8 bytes
3. 12 bytes
4. 16 bytes

(iii) RAM is called DRAM (Dynamic RAM) when: [2 marks]

1. It requires periodic refreshing.
2. It is constantly moving around data.
3. It can do several things simultaneously.
4. None of the above.

(b)

(i) How many 128 x 8 RAM chips are needed to provide a memory capacity of 1,024 bytes? [2 marks]

(ii) The memory is byte addressable. How many lines must the address bus have to access 1,024 bytes of memory? [2 marks]

(iii) How many lines of the address bus must be used for chip selection? [2 marks]

(iv) How many of these lines are required to address the memory locations on the chips? [3 marks]

(c) Given a moveable-head system with a constant disk rotation speed of 12,000 revolutions per minute (rpm); an average seek time of 6 milliseconds; and 512 byte sectors with 500 sectors per track, answer the following two questions, **showing all of your working** and giving your final answers in milliseconds (ms). Marks will be deducted if you do not **show all of your working**.

(i) The file is sequentially organised and is stored on 6 complete tracks followed by exactly one half of a track. [4 marks]

(ii) The same file as that in part (i) is now distributed at random across the disk; that is, each sector of the file is randomly placed on the disk. [6 marks]

Chapter 4: Central processing unit

4.1 Introduction

In the preceding chapter we looked at how information is stored in the main memory, flash memory, magnetic memory, solid state memory and optical memory. On an abstract level, we say that information is stored in the computer in terms of 1s and 0s. But on a physical level, 1s and 0s correspond to different things in different memories. In main memory a 1 corresponds to a high voltage or the presence of an electrical current, and a 0 to a low voltage or the absence of an electrical current. In the magnetic memory, 1s and 0s correspond to magnetised and unmagnetised spots, respectively. And in the optical memory, 1s correspond to micro lands, whose surfaces are highly reflective; whereas 0s correspond to micro pits, whose surfaces are much less reflective.

Of course, data stored in one type of memory can be transferred into another type of memory. The transferring of data between different memory devices is regulated by the **operating** system on the computer. We will examine how the operating system works in Chapter 5 of the subject guide. But here we will consider the question: **How is information processed in the computer?**

4.1.1 Aims of the chapter

This chapter aims to introduce you to the CPU (the processor); issues in the design and performance of the CPU; and ways to enhance the CPU's performance. To motivate and inform your understanding of these issues we describe in detail the operation of the CPU and its components, and how they work together to execute a process. Following this, we consider how to measure the performance of the processor, followed by a discussion of the design and implementation of techniques to improve hardware performance.

4.1.2 Learning outcomes

By the end of this chapter and having completed the Essential reading and activities, you should be able to:

- explain the components of the CPU and their functions
- explain the instruction format and the instruction cycle (especially the fetch-execute cycle)
- illustrate how the CPU executes instructions using a concrete example
- explain the details of the execution of instructions; for example, READ and WRITE, in terms of CPU registers and the system bus
- explain the following ways of improving computer performance: clock frequency, cache memory, pipelining, CISC and RISC.

At this stage you are not required to know the details of how the ALU and the control unit work.

4.1.3 Essential reading

- Stallings, W. *Computer organization and architecture: designing for performance*. (Harlow: Pearson Education Ltd, 2016) 10th global edition [ISBN 9781292096858]:
    - Sections 3.1, 3.3 and 3.4 (Computer components, the fetch-execute cycle, the system bus) and Section 3.2 up to but not including the heading 'Interrupts', including Figure 3.3 on p.109.
    - Sections 4.1–4.3 (cache principles, memory hierarchy and cache design)
    - Sections 12.1–12.2 (machine instruction characteristics)
    - Sections 14.1–14.4 (processor organisation, registers, pipelining)
    - Chapter 17 'Parallel processing' and
    - Chapter 20 'Control unit operation', including Figure 20.4 on p.740.

4.1.4 Further reading

- Harris, S.L. and D.M. Harris *Digital design and computer architecture.* (Waltham, MA: Elsevier/Morgan Kaufmann, 2016) ARM edition [ISBN 9780128000564]:
    - Section 3.6 (Parallelism)
    - Section 4.7 (Data hazards: forwarding)
    - Section 7.5 (Pipelining).
    - Chapter 8, including: Sections 8.1–8.34 (memory systems, the memory hierarchy and the cache), including graphic under 8.2 'Memory system performance analysis'. Chapter 8, including: Sections 8.1–8.34 (memory systems, the memory hierarchy and the cache), including graphic under 8.2 'Memory system performance analysis'.
- Patterson, D.A. and J.L. Hennessy *Computer organization and design: the hardware/software interface.* (Amsterdam: Elsevier/Morgan Kaufmann, 2017) ARM edition [ISBN 9780128017333]:
    - Section 1.2, 'Eight great ideas in computer architecture', Section 1.10–1.11, and Section 4.7 (Data hazards: forwarding versus stalling).
- Stallings, W. *Computer organization and architecture: designing for performance.* (Pearson Education Ltd, 2016) 10th global edition [ISBN 9781292096858]:
    - Section 3.2–3.4 (Computer function and interconnection)
    - Sections 15.1–15.5 (RISC computing)
    - Sections 2.1–2.4 (Performance, designing for and measuring).
- Stokes, J. 'Understanding the microprocessor – part 1: basic computing concepts', arstechnica: http://archive.arstechnica.com/paedia/c/cpu/part-1/cpu1-1.html
- Stokes, J. 'RISC vs. CISC: the post-RISC era – a historical approach to the debate', arstechnica: http://archive.arstechnica.com/cpu/4q99/risc-cisc/rvc-1.html
- Stack Exchange 'How long is a typical modern microprocessor pipeline?' Software Engineering Stack Exchange: https://softwareengineering.stackexchange.com/questions/210818/how-long-is-a-typical-modern-microprocessor-pipeline.

4.1.5 References cited

Drepper, U. 'What every programme should know about memory', GitHub: https://github.com/tpn/pdfs/blob/master/What%20Every%20Programmer%20Should%20Know%20About%20Memory%20-%20Ulrich%20Drepper%20(2007).pdf access 18 June 2018.

Pike, R. 'Concurrency is no parallelism', Vimeo (updated September 2012): https://vimeo.com/49718712

Stack Overflow: 'a question and answer about pipeline lengths': https://softwareengineering.stackexchange.com/questions/210818/how-long-is-a-typical-modern-microprocessor-pipeline

4.2 Glossary of terms

4.2.1 Clock pulse

Clock frequency is measured in pulses per second. The pulse comes from the oscillations of a crystal in a quartz-crystal circuit. Pulses per second are given as a measure of processor speed. Pulses per second were formerly given in MHz, but now are more likely to be measured in GHz. A 1-GHz processor means one billion pulses per second (i.e. 1,000,000,000 pulses per second).

4.2.2 Clock cycle

The clock cycle is the time from the start of one clock pulse, to the beginning of the next.

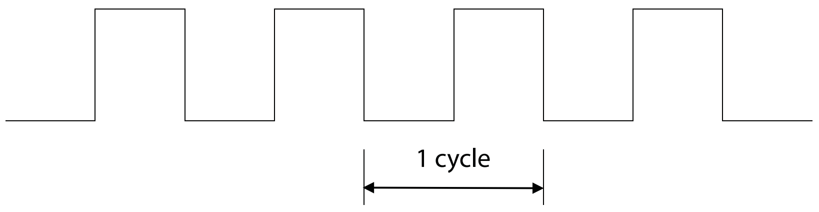


Figure 4.1: Clock signals.

4.2.3 Micro-operation

A micro-operation is the fundamental action of a computer, the building block of all other processes and programs. Stallings (2016, Section 20.2) defines the operation of the processor as 'the performance of a sequence of micro-operations'. The CPU carries out one micro-operation in one clock cycle, directed by its control unit. The CPU may also perform several different micro-operations simultaneously in one clock pulse; for more on this see the section on pipelining later in this chapter.

4.2.4 The instruction set and machine instructions

The **instruction set** is the set of all machine instructions the processor can execute. Different machines have different instruction sets, and the design of instruction sets is an area of debate and research. There are two approaches to instruction set design: **CISC** (Complex Instruction Set Computers) and **RISC** (Reduced Instruction Set Computers). CISC and RISC will be discussed in Section 4.8.

The CPU carries out **machine instructions** or **computer instructions**. We will refer to machine instructions. Machine instructions are carried out via a series of micro-operations. Each machine instruction contains:

- **Opcode** – binary code that specifies the operation to be performed on the data. Opcode is an abbreviation for 'operation code'.

- **Source operand reference** – one or more operands (data) that are the input for the instruction.
- **Result operand reference** – the operation may produce a result and this may need to be stored somewhere.
- **Next instruction reference** – to tell the processor where to fetch the next instruction after completing the current one.

Note that the next instruction reference may be implicit, as the processor may fetch it from a register dedicated to holding the address of the next instruction to be fetched. See later in this chapter for more on registers.

Source and result operands may be in main or virtual memory (see Chapter 5 of the subject guide); in a processor register; contained in the field of the machine instruction being executed; or in an I/O device.

4.2.5 Measures of performance

- **MIPS rate**: Millions of machine-instructions per second.
- **FLOPS**: Binary floating point operations per second.
- **MFLOPS**: Millions of floating-point operations per second; a measure of performance relevant to, for example, measuring the performance of a PC bought for the purpose of playing games; or to one dedicated to scientific work.

4.2.6 Stack memory

Stack memory is for items that need temporary storage. It is implemented as a LIFO structure – last in first out. Stack memory can be compared to a stack of plates. Just as you would naturally add a new plate to the top of the stack; if you wanted a plate you would naturally take the one currently on top of the stack. The processor has access to a stack memory stored in main memory. The processor keeps a register for the purpose of pointing to the top of the stack; that is, the register holds the address of the top of the stack (for more on registers see later in this chapter).

4.2.7 Key terms for pipelining

Pipelining (the instruction pipeline) means that the machine instruction cycle is broken down into stages, allowing the execution of multiple instructions to be overlapped.

Superpipelined processor: In a superpipelined processor the stages of a normal pipeline have been further broken down. The classic RISC pipeline has five stages:

- fetching the instruction
- decoding it
- executing it
- accessing memory
- writing results to registers.

These stages could be broken down, artificially, into smaller and simpler stages. This means that a pipeline can have arbitrarily many stages and that the clock speed can be increased, allowing more instructions to be in the pipeline at the same time. The aim is to improve performance, however a longer pipeline does not automatically translate into better performance, see section 4.10.3.

Conditional branches: A conditional branch is when a process branches to a new instruction only if a certain condition is true. This represents a major obstacle to pipelined processing as the next instruction cannot be taken sequentially from the instruction stream, because the next instruction will not be known until the instruction with the conditional expression has been executed. This prevents the two instructions (and subsequent ones) from being overlapped in the pipeline.

Branch prediction: A mechanism used by the processor to predict the outcome of a program branch before its execution. Used in the implementation of pipelining.

Superpipelined processor: In a superpipelined processor the stages of a normal pipeline have been further broken down. The classic RISC pipeline has 5 stages: fetching the instruction; decoding it; executing it; accessing memory; and writing results to registers. These stages could be broken down, artificially, into smaller and simpler stages. This means that a pipeline can have arbitrarily many stages, and that the clock speed can be increased, allowing more instructions to be in the pipeline at the same time. The aim is to improve performance, however a longer pipeline does not automatically translate into better performance, see Section 4.10.3.

Out of order execution: Executing instructions in a different order to their order in the instruction stream. Used in the implementation of pipelining.

Superscalar processor: A processor design that includes multiple instruction pipelines, so that more than one instruction can be executing in the same pipeline stage simultaneously. A superscalar processor has a single core, but multiple execution units; for example, more than one arithmetic logic unit (see Section 4.4.1).

Interrupt: An interrupt is a signal to the CPU to transfer control to the operating system. There are various reasons for an interrupt that will be discussed in Chapter 5.

4.2.8 Some key terms for the processor

Processor: A physical piece of silicon containing one or more cores. The processor is the computer component that interprets and executes instructions. A processor may contain multiple cores.

Core: Two processors on the same integrated circuit or IC (chip) are referred to as cores; a core contains the logic of a processor, such that each core can read and execute program instructions.

Dual core means a CPU that has two cores on the same IC (chip). Because each core has its own cache, the operating system is able to handle most tasks in parallel. This type of processor can function as efficiently as a single processor and can perform operations up to twice as quickly as a machine with a single processor.

Multicore: An integrated circuit with more than one core.

Process: An instance of a program running on a computer.

4.3 Computer language

As we have seen, the computer uses the language of binary; in simple terms it has a two-character alphabet, consisting of 0 and 1. The first programmers of binary machines wrote instruction in binary, known as 'machine code' but this is very hard for a person to read, hence it is very easy to make a mistake. This motivated the development of assembly language, so-called because something called an 'assembler' was written by the programmers of the first generation digital computers, to translate into machine code simple instructions such as MOV, SUB and ADD.

Assembly language is now considered to be a 'low-level' language, since it is easier to read than machine code, but it still describes operations in terms of the movement of data in the registers. The programmer has to write one instruction for every instruction the computer needs to execute, which is not an intuitive way to describe an algorithm. For example, a programmer might want to say $x = x + y$, but, supposing that the value of *x* is stored in register *P*, and that of *y* in register *Q*, this would break down in machine instructions to:

- LOAD a register with the contents of memory location *P* (x)
- ADD the contents of memory location *Q* (y) to the register
- STORE the contents of the register in memory location *P* (updated value of x).

Today high-level languages, designed to be easy for a human being to read, write and understand, have a **compiler** to translate their code. Some compilers translate directly into machine code, others have a two-stage process; for example, first the compiler translates the program into assembly language, and this is then assembled into machine code.

It will be helpful if we consider a typical example. Suppose that we have a program written in Java, and that the program is stored on the hard disk. We want the computer to run the program. How is this done? The computer cannot deal with the program, or any other high-level program, directly. The Java program is translated into byte code by the Java compiler (aka 'compiled') and put into a **class** file. (Java byte code is the instruction set of the Java Virtual Machine (the JVM). Each instruction has one byte for the opcode, plus more bytes for operands, as necessary.) When the class file is run, the byte code is translated into machine code. This machine-code program is loaded into main memory, and is then executed by the **central processing unit** (CPU).

The machine-code program can be divided into two parts: instructions and data. Instructions ask the CPU to do certain actions (e.g. read data from, or write data to, particular memory locations, add numbers, display data on the screen etc.). Data can be numbers, characters, letters, sounds, colours and so on. When the CPU executes this machine-code program, it fetches an instruction from main memory, executes it, then goes to fetch the next instruction and executes it. It follows this 'fetch–execute' cycle until all the instructions are executed (or when a 'halt' instruction is executed). In the process of executing the instructions, the data also gets accessed and used by the CPU.

4.4 The components of the CPU

Modern machines are often dual core; that is, they have two processors in a single integrated circuit. We will only be considering machines with one processor, and will not be covering the hardware and software necessary to make two processors work together effectively (this particularly relates to the organisation of the cache).

The processor consists of:

- the **ALU**
- **registers**
- internal data paths
- external data paths
- **control unit**.

(Stallings, 2016, Section 20.2)

Stallings (2016, Section 20.2) notes that **internal data paths** are used to move data between registers, and between registers and the ALU. **External data paths** link registers to memory and I/O modules, normally by a system bus, but increasingly, the bus is being replaced with 'point to point interconnection' (Stallings 2016, Section 3.5). You should be aware of this trend, but this subject guide will consider the connection of the CPU to other components to be via a system bus.

We will now consider in detail the ALU, registers and the control unit.

4.4.1 The arithmetic and logic unit (ALU)

The ALU carries out arithmetic and logic and is called by Stallings 'the functional essence of the computer' (Stallings, 2016, Section 20.2). The ALU is composed of logic gates and carries out the computer's data-processing functions. The ALU performs arithmetic operations (for example, add or subtract); Boolean logical operations, such as AND, OR, NOT; and other logical operations (such as comparing two numbers to see if one is greater than the other; comparing two letters to see whether they are the same).

4.4.2 Registers

To keep track of memory addresses, and instructions and data that are currently being operated on, the CPU contains a number of **registers** to store data internally. A register is a small piece of electronic (or semiconductor) memory. Some registers contain status information needed to manage instruction sequencing; others store data that goes to or comes from the ALU, memory and IO modules. Most architectures include an **accumulator**, a register used to store the results of ALU calculations, so that if the next instruction needs the result of the previous one, the processor does not need to retrieve it from main memory.

Some registers are **user visible**, while **control and status registers**, used by the control unit, are not.

4.4.3 Control and status registers

Control and status registers are used by the **control unit** (discussed below) to control the operations of the CPU and by privileged, operation system programs to control the execution of programs. Different processor designs have their own distinct registers and register organisation, depending on the instruction set of the machine. Stallings (2016, Section 14.2) gives the following four control and status registers as 'essential to instruction execution', since they move data between the processor and memory:

- **Program counter** (PC): contains the address of the next instruction to be fetched; thus it keeps track of the current position in a machine-code program in memory.
- **Instruction register** (IR): holds the instruction just fetched from memory.
- **Memory address register** (MAR): holds the address of a word that the CPU is going to access.
- **Memory buffer register** (MBR): contains a word (instruction or data) just read from memory, or a word (data) that is about to be written into memory.

The MAR and the MBR may have different names in different processor designs, but will perform equivalent functions.

4.4.4 The control unit

The control unit has two basic tasks (Stallings, 2016, Section 20.2):

- **Sequencing** – the control unit causes the processor to step through a series of micro-operations in the correct sequence for the program being executed.

- **Execution** – the control unit causes each micro-operation to be performed.

In order to perform its functions the control unit needs to know the state of the system. Hence it receives input so that it knows what control signals to output. See Figure 20.4 (p.740) in Stallings (2016) for a diagram of a control unit with inputs and outputs.

Inputs to the control unit are:

- Pulses from the clock. The control unit can issue one or more control signal(s) in each clock cycle. This will enable the CPU to do one micro-operation in one cycle, or a number of micro-operations simultaneously in one cycle.
- The instruction register. The control unit decodes the opcode of the current instruction. Carrying out an opcode means making several micro-operations, hence with the opcode the control unit knows which micro-operations to tell the processor to make.
- Flags – these are also known as condition codes, bits set by the hardware to record operation results, and stored in one or more registers.
- Control signals from the control bus.

The control unit outputs either control signals to the processor or to the system bus:

- Control signals internal to the processor: either to move data in the registers (activate a data path), or activate ALU functions.
- Control signals via the control bus: either to memory, or to I/O modules.

So to sum up, using all of its inputs the control unit outputs a series of signals causing micro-operations to happen, timed by the clock pulse so that there is a pause between operations for signal levels to stabilise. Stallings (2016, Section 20.2) notes the simple nature of the control unit and yet it is 'the engine that runs the entire computer'.

4.4.5 The relationship between the components of the CPU

The relationships between registers, the ALU and the control unit can be described as follows. Data are presented to the ALU in registers, and the results of operations are stored in registers. The control unit provides signals that control the operation of the ALU and the movement of data into and out of the ALU. Figure 4.2 below depicts the internal structure of the CPU:

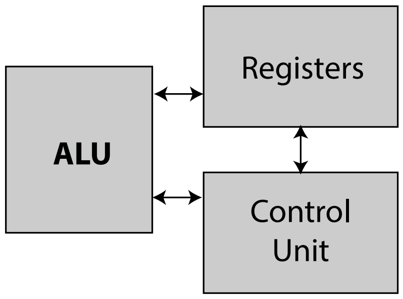


Figure 4.2: The components of the CPU.

The CPU is connected to main memory, secondary memory and I/O devices through the system bus. The data bus is connected to the memory buffer register (MBR) and the address bus to the memory address register (MAR). The control bus is connected to the control unit. This connection is shown in Figure 4.3 below:

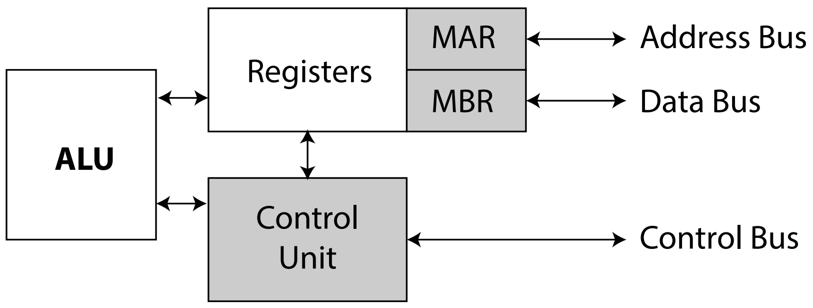


Figure 4.3: CPU and the system bus.

4.5 How the CPU carries out instructions

4.5.1 A more detailed look at micro-operations

Micro-operations do one of the following things:

- transfer data from one register to another
- transfer data from one register to an external interface (for example, the system bus)
- transfer data from an external interface to a register
- perform an arithmetic and logic operation, using registers for input and output.

(Stallings, 2016, Section 20.3)

A micro-operation is so-called because each micro-operation does very little; in fact a micro-operation, as we can see from the above list, does one thing with, or to, a processor register.

Each machine instruction is made up of a number of micro-operations. All the micro-operations needed to perform a machine instruction fall into one of the above categories. This means that all the micro-operations needed to perform every instruction in the machine's instruction set are described in the above list.

4.5.2 The opcode

The CPU carries out **machine instructions**. The set of instructions that the processor can execute is called the processor's **instruction set**. As we have seen, each machine instruction contains:

- **opcode**
- **source operand reference**
- **result operand reference**
- **next instruction reference**.

We will look in more detail at the opcode. Opcodes vary, so the control unit examines the opcode and produces a series of micro-operations based on the opcode. This is known as instruction decoding.

An instruction is a sequence of bits, divided into fields. Since these fields are binary, for readability the instruction's opcode is represented by abbreviations called mnemonics. Common examples:

• ADD	Add
• SUB	Subtract
• MUL	Multiply

• DIV	Divide
• LOAD	Load data from memory
• STORE	Store data to memory

(Stallings, 2016, Section 12.1)

Opcodes vary from machine to machine, but are categorised by Stallings as:

- data transfer
- arithmetic
- logical
- conversion
- I/O
- system control
- transfer of control.

(Stallings, 2016, Section 12.1)

See Table 12.3 from Stallings (2016, Section 12.4: 'Types of operations') for a list of typical instruction types in each of the above categories.

4.5.3 Operands

The op-code indicates what operation is to be performed. An operand specifies the thing that is to be operated on. An operand is an address where some real datum (number, letter, character, colour, sound pitch, etc.) is stored. Operands are also represented symbolically, often together with a symbolic opcode (mnemonic); for example, ADD, R_1, R_2 may mean add the value in register R_2 to the value stored in register R_1, and store the result in R_1.

4.5.4 A close look at machine instructions

Source operands: In any machine instruction two addresses might be needed as the data for an arithmetic operation. This suggests at most four addresses in any machine instruction: two source operands; one destination address; plus the address of the next instruction.

In fact, in most designs the majority of instructions have 1, 2, or 3 operands, and the address of the next instruction is not included since the processor will get it from the program counter. There may also be some special purpose instructions with more operands; Stallings (2016, Section 12.1) notes that there are 17 register operands in a single instruction of the ARM architecture. Note that ARM is a particular specification for computer architecture that has developed from **RISC – reduced instruction set computing**. For more on RISC see later in this chapter. Stallings (2016, Section 12.1) notes that zero operands are possible, since the CPU may take the data for the instruction from the top of the stack.

4.5.5 Interpretation of symbolic machine instructions

Stallings (2016, Section 12.1) gives the following interpretation of symbolic machine instructions, where 'OP' means any opcode:

Number of addresses (operands)	Symbolic representation	Interpretation
0	OP	T ← (T – 1) OP T
1	OP A	AC ← AC OP A
2	OP A, B	A ← A OP B
3	OP A, B, C	A ← B OP C

- T = top of stack
- (T – 1) second element of stack
- AC = accumulator
- A, B, C memory or register locations

Note that the symbol being pointed to in the 'Interpretation' column, is the place where the result of the operation is to be stored. In the 1-address instruction format, the address for the second operand is implicitly the register known as the accumulator – a place to temporarily store the result of arithmetic operations. See Stallings (2016, Section 12.1) for a list of 1, 2 and 3 address instructions, and their meaning.

4.5.6 The instruction cycle

An instruction cycle is the process by which a computer decodes and executes a machine instruction. A machine instruction may take many clock cycles to complete.

A machine-code program, when it is being executed, is kept in main memory. The CPU runs the program by repeatedly performing an instruction cycle. In the most basic cycle, an instruction cycle consists of two steps: (i) the CPU fetches an instruction from main memory, and (ii) the CPU executes the instruction. Because of this, an instruction cycle in this simplest case is also referred to as the fetch–execute cycle. See Figure 3.3 in Stallings (2016), which illustrates this idea.

As Figure 3.3 in Stallings (2016) suggests, the CPU fetches a machine instruction from main memory and then executes it. It then fetches the next instruction and executes it. The CPU repeatedly does this, until it encounters an instruction that halts the computer (or when an unrecoverable error occurs, or the computer is turned off).

The CPU obviously needs to 'know' where to fetch the next instruction in main memory. The program counter (PC) is used for this purpose. The PC contains the address of the instruction to be fetched next. After the next instruction is fetched and executed, the processor increments the PC, so that the PC will contain the address of the next instruction in the sequence.

The fetched instruction is loaded into the instruction register (IR). The instruction contains an opcode that specifies the action the CPU is to take. The CPU interprets the instruction and performs the required action. In general, these actions fall into four categories:

1. **Transfer of data between CPU and memory** (i.e. read from memory, and write to memory).
2. **Transfer of data between CPU and some I/O devices** (e.g. read from a device, write to a device).
3. **Data processing** (i.e. arithmetic and logical operations on data).
4. **Control –** an instruction may specify that the sequence of execution be altered. For example, an instruction I_1 at address 149 may tell the CPU to execute an instruction stored at address 182. When I_1 is executed, the PC will be set to 182. Thus, on the next instruction cycle, the CPU will fetch the instruction at address 182, rather than the instruction at address 150.

Activity 4.1

A computer memory contains 512KiB words. Each word has 64 bits. A machine instruction has four parts: an indirect bit, an operation code, a register code part to specify one of the 32 registers and an address part. A machine instruction is stored in word memory.

1. How many bits are needed for the opcode, the register code and the address part?
2. Draw the instruction in word format indicating the number of bits of each part.
3. How many bits are there in the data and address inputs of the memory?

4.5.7 How the processor performs a machine instruction

At each stage of the instruction cycle the CPU performs one operation, fetch or execute in our most basic description of the instruction cycle. The processes to execute an instruction may vary enormously, but fetching an instruction is usually done the same way every time. Let us now see exactly how the CPU fetches an instruction.

With the help of registers, the control unit and the system bus, we describe the fetch operation as a series of micro-operations and control signals.

4.5.8 Fetch cycle in control signals and micro-operations

When the CPU fetches an instruction from main memory the following actions are taken:

1. The address of the required word is put in the MAR.
2. The MAR is connected to the address bus, so the address – a set of electrical signals, 1s and 0s – is put onto the address bus.
3. The read/write line in the control bus is set to indicate a READ operation.
4. The required word in main memory is activated, and the word – an instruction or datum – is put on the data bus.
5. The MBR is connected to the data bus, so the instruction/datum is stored in the MBR. Incrementing the program counter can be done simultaneously with updating the MBR.
6. The MBR contents are moved into the instruction register.

The control unit regulates all these steps – except Step 4, which is external to the CPU.

- Step 1, the control unit sends a control signal that opens the gates between the PC and the MAR, allowing the content of the PC into the MAR.
- Step 2, a control signal is sent that opens the gates between the MAR and the address bus, allowing the content of the MAR onto the address bus.
- Step 3, a READ signal is put on the control bus.
- In Step 4, the control unit just waits for main memory to react and to put the data on the data bus.
- Step 5, the control unit sends a control signal that opens the gates, allowing the content of the data bus to be stored in the MBR.
- Step 6, the control unit sends another signal to allow the contents of the MBR into the IR.

Let us consider these steps again with their control signals:

1. The address of the required word is put in the MAR.
   Control signal: open the gate between the PC and the MAR

2. The MAR is connected to the address bus, so the address – a set of electrical signals, 1s and 0s – is put onto the address bus.
   Control signal: open the gate between the MAR and the address bus
3. The read/write line in the control bus is set to indicate a READ operation.
   Control signal: memory read control signal on the control bus
4. The required word in main memory is activated, and the word – an instruction or datum – is put on the data bus.
5. The MBR is connected to the data bus, so the instruction/datum is stored in the MBR. Incrementing the program counter can be done simultaneously with updating the MBR.
   Control signals: (1) open the gates allowing the contents of the data bus onto the MBR; and (2) add increment to the PC
6. The MBR contents are moved into the instruction register.
   Control signal: open the gate between the MBR and the IR

The four control signals above in bold are issued by the control unit simultaneously.

The clock emits a regular pulse, the time from one pulse to the next is a time unit in which micro-operations are taken, directed by the control unit. At the first pulse, the contents of the PC are moved to the MAR (one micro-operation). In the next time unit, the contents of the memory location are moved to the MBR, and the PC is incremented (four micro-operations). In the third time unit, data is moved from the MBR to the IR (one micro-operation). Hence, even a simple fetch instruction breaks down to six micro-operations in three clock cycles.

4.6 Processor performance

In a computer, all information processing is done in the CPU. It used to be considered that the faster the CPU speed, the faster the machine, but this simple metric no longer works. Other technology has not kept pace with the increases in processor speed, so machine speed also depends on how well hardware and software work to keep the processor busy. It is now possible that two processors with the same clock do not share the same performance, measured by other metrics such as the performance equation (see later in this section). This means that clock speed has become a more crude measure of performance, and things such as how efficiently the CPU is used, and how well it is supported by the design of the machine (its architecture) have also to be considered.

The processor cannot achieve its (now incredibly fast) potential if it is not kept busy. Some of the techniques used by today's processors to increase throughput of instructions are pipelining, branch prediction, superscalar execution, out of order execution and speculative execution (see the glossary of terms for pipelining in Section 4.2.7 above, or see Stallings, 2016, Section 2.1).

One major drag on processing time is the interface between the processor and main memory, as the speed of main memory has not increased at the same rate as the increase in processor power. Stallings (2016, Section 2.1) lists several options available to the designer for tackling the problem of the slow interface with main memory; this guide covers only one of these options, cache memory.

Stallings (2016, Sections 2.1–2.2) discusses limits on performance. Stallings notes that in the past pipelining was an important strategy to improve processor speed, but suggests that since the 1990s, very little significant increase in performance has come from changes to the pipeline. Pipelining has reached such complexity (including adding multiple pipelines with superscalar execution) that further significant increases in speed are unlikely to be found there. Stallings notes that increasing processor speed is today attempted in three ways:

- **Hardware speed**: Add more logic gates and increase the clock rate. If logic gates are closer together then signals pass between them faster, allowing an increase in the clock rate. If the clock rate is increased then micro-operations happen faster.
- **Caches**: Have more of them, make them bigger and faster, incorporate cache memory onto the chip (possible because of increasing chip density).
- **Architecture and organisation**: Increase the number of machine instructions processed in any time unit, normally by architecture and organisation that implements parallelism such as dual cores (see glossary of terms in Section 4.2.8 above).

Stallings notes that it is now hard to increase chip density and the clock speed because of issues with the heat generated by the extra power used (the issue is hardware will fail if it gets too hot); and because resistance and capacitance increases with greater density (because of shorter and thinner wires), a physical limit on density may be approaching. He considers that since the cache organisation has increased in complexity (cache memory now often has three levels of differing sizes and speeds), it will be hard to squeeze significantly more performance enhancement from better cache organisation. This leaves architecture and organisation, hence the multi-core chip.

Multiple processors on the same chip are referred to as cores. Dual cores, two processors sharing the same cache and acting as one, were introduced first. Dual-core machines allow for an increase in speed without increasing the clock rate, or the complexity, since two simpler cores are used rather than one more complex one. Development of multicore machines has been the focus of much work, and continues rapidly.

If the limits on clock speed due to power issues are solved with multicores, multithreading tackles the limits on performance due to processor complexity and power consumption; it should be noted that multithreading is implemented partly in hardware, but also in software.

4.6.1 Multithreading

Multithreading means that the instruction stream for a process is divided into two or more threads of execution. Each thread has part of the instruction stream, and each thread can be executed in parallel with the other threads. This means either that different threads can be executed in parallel; or, without parallelism, that the processor can switch between threads. Multithreading does not increase power consumption or complexity.

4.6.2 Examinable content

Multithreading and multicore computers are not part of the CO1110 syllabus.

4.6.3 The performance equation

The clock speed was once used as a measure of the performance of a processor. The idea was that a person or organisation should choose the computer with the fastest clock speed they could afford, to guarantee the best performance. As computers approach a limit on clock speed, and with many different ways of supporting the processor with hardware and software, this simple metric is not as reliable as it once was.

For machines dedicated to pursuits that are heavily biased towards floating-point operations, such as gaming or scientific pursuits, MFLOPS is a plausible measure of performance (millions of floating-point operations per second).

For other machines, we can measure processor performance in MIPS – millions of machine instructions per second. Stallings (2016, start of Section 17.4 'Multithreading and chip multiprocessors') considers the rate at which the processor carries out instructions to be the most important performance measure for it, and suggests MIPS as a measure of this. Patterson and Hennessey (2017, Section 1.10 'Fallacies and pitfalls') discuss problems with using MIPS as a way to measure performance. One is that it is not an accurate metric when comparing the performance of machines with different instruction sets, since instruction sets may have more or less complex instructions. More complex instructions means more work is being done per instruction. The second is that MIPS will vary on the same computer with different processes, hence one machine cannot have a single MIPS rate. Finally, suppose that one program is executing its instructions on average faster than a second program. It is using the same instruction set as the second program, but is using fewer of the slower instructions. There could be a significant variation in the MIPS rate of both programs that is not related to the performance of the processor.

Patterson and Hennessey (2017, Section 1.11 'Concluding remarks') go on to promote the performance equation as a reliable indicator of performance. This gives execution time for a process as the product of three things, considered to be independent of each other:

1. Instruction count for a process (IC)
2. Average number of clock cycles per instruction (CPI)
3. Clock time (time for one clock cycle) (CT).

The instruction count is simply the number of machine instructions executed in a complete process; it is likely to be some subset of all the instructions in the complete program. The average number of clock cycles per instruction recognises that some instructions take more clock cycles than others so the time for an instruction cannot be given as a constant value. The clock time is the time for one clock cycle; that is, the time from the start of one pulse to the beginning of the next. In seconds, this is 1/f, where f is the clock frequency; for example, a 1 GHz processor has a clock cycle time of 0.000,000,001 seconds, or 1 nanosecond (ns).

So the processor time, *T,* in seconds, to carry out a process is given by:

$$T = IC \times CPI \times CT$$

Five factors influence the variables in the performance equation:

1. Processor implementation
2. Compiler technology
3. The cache
4. The memory hierarchy
5. The instruction set.

We shall next look at some of the above ways of enhancing performance, starting with the cache and memory hierarchy. After that, we will consider the instruction set and finally pipelining – a method of increasing the number of micro-operations that the processor can carry out in one clock cycle. Compiler technology is beyond the scope of this course.

4.7 The cache and memory hierarchy

For the best performance the processor must not be idle, waiting for instructions or data. So we need very fast memory to keep the processor busy; however, the faster the memory access time, the higher the cost. Mass-market computers cannot be built with only the most expensive memory, so cheaper technology is used for higher capacity memory.

How does the designer give the computer the benefits of the fastest memory, without massively increasing the cost of the machine? This is done through a memory hierarchy, where the smallest, fastest (and most expensive) memory is closest to the processor; and as one moves away from the processor, memory becomes bigger, cheaper and slower. With this hierarchy fast, expensive technology is supported by cheaper but slower memory. The purpose of a memory hierarchy is to fool you that every bit of memory (including secondary storage) is as fast as the processor.

Harris and Harris (2016, Section 8.1 'Introduction') have a very good graphic demonstrating the differences in price and access time at different levels of the hierarchy. (Figure 8.4, 'Memory, hierarchy components with typical characteristics in 2015.'). The graphic shows that in 2015 SRAM memory, used for the cache, cost $5,000 per GB. Main memory cost $7 per GB, a massive price differential.

Note that virtual memory will be discussed in Chapter 5 of the subject guide, but in the Harris and Harris diagram virtual memory refers to secondary storage.

At the very top of the memory hierarchy is volatile semi-conductor memory; non-volatile technologies for secondary storage (hard disks, solid state) is at the bottom.

Electronic semi-conductor memory can be broken down into the following hierarchy, based on distance from the processor:

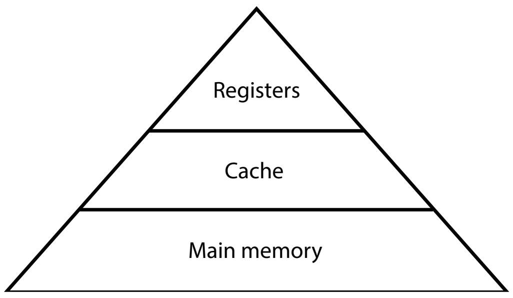


Figure 4.4: Hierarchy of electronic, semi-conductor memory.

Stallings states that for an effective memory hierarchy, the following conditions must hold as one moves down the hierarchy:

a. decreasing cost per bit

b. increasing capacity

c. increasing access time

d. decreasing frequency of access by the processor.

(Stallings, 2016, Section 4.1)

The key to the success of this structure is decreasing frequency of access by the processor. This enables cheaper memories to be effectively used to supplement faster memory, while still giving overall fast performance. The key to condition (d) is the cache, which uses **locality of reference** in its algorithms to control the movement of data between the registers and main memory. **Locality of**

reference describes the idea that over short periods of time the processor tends to access the same memory locations repeatedly. The cache can quickly provide the processor with data and instructions, rather than the processor waiting for data to be retrieved, much more slowly, from main memory.

General-purpose machines now have up to three caches, but we will be discussing cache organisation as if there was just one.

One thing to note is that both Harris and Harris's figure and Figure 4.4 above may give a misleading impression of the size of the cache in relation to main memory. Inhis 2007 article 'What every programme should know about memory', Ulrich Drepper estimates that the cache is typically one thousandth the size of main memory, or one tenth of one per cent of main memory, and gives the figures 4MiB cache versus 4GiB main memory.

Cache size has increased since 2007, but it is likely to always be a very small percentage of main memory, while the huge price differential demonstrated in the Harris and Harris figure remains.

4.7.1 Locality of reference and cache memory

Usually, there is another clock that co-ordinates the events on the system bus. This clock is much slower than the clock used by the CPU, because main memory is slower than the CPU. If the CPU were linked directly to main memory through the system bus, then it would be slowed down by the lower clock rate of the bus. In order to avoid this, a cache memory is put between the CPU and the main memory. Cache memory is small, but can operate at (nearly) the same speed as the CPU.

The cache contains a portion of the main memory. Different portions may reside in the cache at different times. Cache design uses locality of reference, which breaks down into two principles, **temporal locality** and **spatial locality** (see Stallings 2016, Appendix 4a). Temporal locality refers to the tendency for a processor to access memory locations that have been used recently (most program execution time is spent on loops, and loops mean accessing the same set of instructions repeatedly). Spatial locality notes that the processor accesses locations that are clustered in one place, because instructions are normally accessed sequentially. This means that in a particular short period we can keep a copy of a particular portion of the main memory in the cache.

4.7.2 Read and write with the cache

When the CPU needs to read a word from memory, it first checks whether the word is in the cache. If so (called a 'cache hit'), the word is delivered to the CPU. If not (a 'cache miss'), a block of the main memory, consisting of some fixed number of words and containing the desired word, is read into the cache, and then the word is passed to the CPU.

When the CPU writes data, it either writes back to main memory, or writes to the cache and then lets the cache write the data to main memory at its convenience.

A system may have more than one cache, each sitting between main memory and the processor. If there is more than one cache then the cache nearest to the processor will be the smallest and fastest, with each subsequent cache bigger and slower (although still faster than main memory) with the slowest and largest next to main memory (see Stallings 2016, Section 4.2).

Activity 4.2

What is locality of reference and how does the cache exploit locality of reference to increase cache hits?

4.7.3 Cache organisation

Main memory is divided up into blocks, each block contains a certain number of words. The cache is divided up into lines; a block can fit in one line. A line contains, as well as the block, control and tag bits. The size of the line is considered to be the same size as the block; control and tag bits are ignored. Loading main memory in blocks to the cache takes advantage of the locality of reference principle. Each time a word is needed by the processor that is not in the cache, the cache loads a block of memory containing that word and others; it is likely that the processor will soon need other words from the same block, or will need the same word again.

Virtual memory will be discussed further in Chapter 5 of the subject guide. For the moment, when a system implements virtual memory it is with the help of a memory management unit (MMU). The MMU has the responsibility of translating logical (virtual) addresses to physical addresses in main memory. How the MMU is linked to the cache depends on whether the cache is using logical (virtual) or physical addresses. The cache can take logical addresses directly from the processor, or operate on physical addresses delivered by the MMU (see Stallings, 2016, Section 4.3). A **logical cache** uses logical addresses; and a **physical cache** uses physical addresses. You are not required to know the details of logical versus physical caches.

When the CPU wishes to save a word into the cache, where is it put? When the CPU wants a word, how does it know whether or not it is already in the cache? There are three schemes for cache addressing; we will describe each in turn.

4.7.4 Direct mapped caches

With direct mapping the modulo function is used to decide which blocks from the memory map to which lines in the cache. The modulo function simply means taking the remainder of integer division as the output of the function. For example, for modulo 5 we would have:

0 mod 5 = 0	5 mod 5 = 0	10 mod 5 = 0
1 mod 5 = 1	6 mod 5 = 1	11 mod 5 = 1
2 mod 5 = 2	7 mod 5 = 2	12 mod 5 = 2
3 mod 5 = 3	8 mod 5 = 3	13 mod 5 = 3
4 mod 5 = 4	9 mod 5 = 4	14 mod 5 = 4

Table 4.1: Table of values modulo 5.

As you can see from the above table, the results of the modulo function repeat, modulo 5 can only return 5 values 0, 1, 2, 3 and 4. Modulo 6 would give 6 values 0, 1, 2, 3, 4, and 5. The modulo function is also expressed with a straight line; for example, 3 | 5 = 3.

In order to decide which blocks of memory map to which lines in the cache, each block in main memory is numbered, and the result of that number modulo the number of lines in the cache, gives the cache line that the block maps to. Let us consider a (simplified) example.

Suppose that we have a main memory of 256 bytes, and a block size of 4 words, with a word size of 32 bits. Memory is word addressable so the CPU will issue a request for one word at a time from the cache.

How many blocks does main memory have? Each block has 4 words, a word is 4 bytes, so each block has 16 bytes. Main memory has 256 bytes. So the number of blocks is 256/16 or $2^8 / 2^4 = 2^4 = 16$.

Suppose that the cache has 4 lines, meaning that our mapping function is, if *b* is any block number from main memory, *b* modulo 4. This will mean that the first block of the main memory, block 0, maps to cache line 0, the second to 1, the third to 2, the fourth to 3 and by the fifth we are back to 0. Figure 4.6 below illustrates.

Main memory block numbers		
{0, 4, 8, 12}	$\rightarrow$	Cache line 0
{1, 5, 9, 13}	$\rightarrow$	Cache line 1
{2, 6, 10, 14}	$\rightarrow$	Cache line 2
{3, 7, 11, 15}	$\rightarrow$	Cache line 3

Figure 4.5: Direct mapping 16 blocks of main memory to a 4 line cache.

4.7.5 Direct mapped addressing

Each cache line has an address, not included in the size of the line. The address is divided into an offset, a cache block index and a tag. The tag has the most significant bits of the address, and the offset the least significant bits.

The following calculations should be done in addressable units (words or bytes). The following meanings apply:

- b = block size = 2^w (words or bytes)
- P = number of main memory blocks = 2^s
- C = size of cache memory (in words or bytes)
- Q = size of main memory blocks in words or bytes
- m = number of lines in the cache = 2^r
- m = total number of bits in the address
- w = the number of bits for the offset
- r = number of bits for the cache block index
- t = number of bits for the tag

If the size of main memory (in words or bytes) is 2^n, then *n* bits are needed for addressing. This breaks down as follows:

- **Offset** (position of the requested word in the block/line): Log_2 of the block size in words or bytes, if the size of a block is 2^w, then *w* bits are needed to uniquely identify the offset.
- **Cache block index** (which cache line the block could be in): Log_2 of the number of cache lines, given by C/b. Hence, if $C/b = 2^r$ then r bits are needed.
- **Tag** (tells the processor which block from main memory is actually in the cache): Tag bits needed are given by $\log_2 (P/m)$. In other words if $P/m = 2^t$, then *t* bits are needed to uniquely identify the block.

Let us consider the example from the previous section. Recall that in the example we have a main memory of 256 bytes. Memory is word addressable, and each word is 4 bytes, therefore we have 256/4 words in main memory, or $2^8/2^2 = 2^6$. This means that we need **6 bits for addressing**.

The block/line size in words is $4 = 2^2$, so we need **2 bits for the offset**.

We know that there are 4 cache lines, $4 = 2^2$, so we need **2 bits for the cache block index**.

Tag bits stored in the register tell the processor which block from main memory is actually in the cache line. In our example, there are 4 different blocks that could actually be in each cache line; so 2 bits will be enough to number each block uniquely, as follows:

Tag bits →		**00**	**01**	**10**	**11**	
Main memory block numbers	{	0,	4,	8,	12	}
	{	1,	5,	9,	13	}
	{	2,	6,	10,	14	}
	{	3,	7,	11,	15	}

Figure 4.6: Direct mapped memory blocks with their tag numbers.

To put this another way, tag bits = P/m, which is 16/4 = 4 = 2^2, hence we need **2 bits for the tag**.

This gives us the following address division:

Tag (2 bits)	Cache block index (2 bits)	Offset (2 bits)

4.7.6 Advantages and disadvantages of direct mapping

The advantage of direct mapping is that its simplicity makes it cheap to implement and searching is fast; its disadvantage is that every block can only go into one line. This means that if the processor should repeatedly request words from blocks that map to the same line, then the lines will be swapped often (known as 'thrashing'), which is clearly undesirable since it will slow the hit rate, and hence the processor's operations.

4.7.7 Associative mapping

Associative mapping overcomes the disadvantage of direct mapping by allowing any block to map to any cache line.

4.7.8 Associative mapped addressing

Tag (4 bits)	Offset (2 bits)

Because any block can be in any cache line, there is no cache line index in the address. Using the previous example the 6 bits for the address will break down into 2 bits for the offset, leaving 4 bits for the tag. In fact, in all three addressing systems we will discuss, the offset is always the same value when the block size is kept constant, since it is $\log_2$ [block size in words or bytes].

There are 16 blocks in main memory, so we will need 4 bits to address them all in the tag.

In general:

- **offset** (position of the requested word in the block/line): Log_2 of the block size in words or bytes, if the size of a block is 2^w, then *w* bits are needed to uniquely identify the offset.
- **tag** (tells the processor which block from main memory is actually in the cache): Tag bits needed are given by $\log_2$ (P). In other words if P = 2^t, then *t* bits are needed to uniquely identify the block. P is the number of blocks in main memory.

4.7.9 Advantages and disadvantages of associative mapping

Given that any block can go in any line, a block replacement algorithm designed to increase the hit rate can be implemented. A disadvantage is the need to check the tag for every line to determine if the requested word is in the cache.

4.7.10 Set associative mapping

Set associative mapping is designed to utilise the strengths of both direct mapping and associative mapping, and to mitigate their disadvantages. With set associative mapping the lines of the cache are divided up into equal sized sets. Each block from the main memory can only be placed in one of the sets. Within the set the block can be placed in any line.

We have a main memory of 256 bytes, and a block size of 4 words, with a word size of 32 bits. Main memory has 16 blocks, the cache has 4 lines, which is all the same as our previous example. The difference is that now the cache is organised into 2 sets, each of 2 lines.

So the cache set number that a main memory block maps to is the main memory block number modulo the number of sets. With 2 sets the possible results of the modulo function are 0 and 1. Hence even numbered blocks are put into set 0, and odd numbered into set 1.

Main memory block numbers		**Cache sets**
{0, 2, 4, 6, 8, 10, 12, 14}	→	Cache set 0 (2 lines)
{1, 3, 5, 7, 9, 11, 13, 15}	→	Cache set 1 (2 lines)

Figure 4.7: Set associative mapping example.

4.7.11 Set associative addressing

If the size of main memory (in words or bytes) is 2^n, then *n* bits are needed for addressing. This breaks down as follows:

- **offset** (position of the requested word in the block/line): Log_2 of the block size in words or bytes, if the size of a block is 2^w, then *w* bits are needed to uniquely identify the offset.
- **cache set index** (which cache set the block could be in): Log_2 of the number of sets. If there are 2^r sets, then r bits are needed.
- **tag** (tells the processor which block from main memory is actually in the set): Tag bits needed are given by $\log_2$ (P/s). In other words if P/s = 2^t, then *t* bits are needed to uniquely identify the block.
  - b = block size = 2^w (words or bytes)
  - P = number of main memory blocks = 2^s
  - s = number of sets in the cache = 2^r
  - w = the number of bits for the offset
  - r = number of bits for the cache set index
  - t = number of bits for the tag

Continuing with the same example as before, there are 6 bits in the address:

The block/line size in words is $4 = 2^2$, so we need **2 bits for the offset**.

We know that there are 2 cache sets, $2 = 2^1$, so we need **1 bit for the cache set index**.

There are 16 blocks in main memory and 2 sets, $16/2 = 8 = 2^3$ hence we need **3 bits for the tag**.

Tag (3 bits)	Set index (1 bit)	Offset (2 bits)

4.7.12 Advantages of set associative addressing

With set associative mapping a replacement algorithm can be run when bringing in a new block to a set that is full. Because a block can go into any line in the set, an algorithm designed to maximise cache hits can choose the line. It is easier to search for a word within the cache, as only the tags in a particular set must be checked, rather than every tag.

Conflicts are much less likely than in direct mapping.

In general, k-way set associative, means the cache is divided into sets with k-blocks in each. Stallings (2016, Section 4.3), reports that 2 lines in a set is the most usual, and improves the hit rate from direct mapping. Four lines in a set makes another small, incremental gain, but more than 4 lines does not improve the hit rate.

Activity 4.3

Assume a system with the following characteristics:

- a direct mapped cache
- data words are 8 bits long
- data addresses are to the word
- a physical address is 20 bits long
- the tag is 11 bits
- each block holds 16 bytes of data.

Work out the number of blocks in this cache.

4.8 CISC and RISC

4.8.1 The instruction set

The **instruction set** is all of the machine instructions that the processor can execute. The machine instruction set 'provides the functional requirements for the processor' (Stallings 2016, Section 12.1).

Since the instruction set is the programmer's means of controlling the processor, a computer should have an instruction set such that any task set by a high-level language can be expressed in it. This would mean being able to describe, for the processor to execute, the following types of operations:

- **Data processing**: arithmetic and logic.
- **Data storage**: data movement into or out of register locations, and/or register and memory locations.
- **Data movement**: I/O instructions.
- **Control**: test and branch instructions.

(Stallings 2016, Section 12.1)

As the **instruction set** defines many of the functions performed by the CPU it affects such things as the number of registers, and the complexity of the control unit. Design issues relating to instruction sets remain in dispute, Stallings (2016) notes the following are included among issues still in contention:

- **Operation repertoire**: How many and which operations to provide, and how complex operations should be.
- **Data types**: The various types of data upon which operations are performed.

- **Instruction format**: Instruction length (in bits), number of addresses, size of various fields, etc.
- **Registers**: Number of processor registers that can be referenced by instructions, and their use.
- **Addressing**: The mode or modes by which the address of an operand is specified.

(Stallings, 2016, Section 12.1)

CISC and RISC are two different design concepts relating to the first item in the above list – most crucially, how complex instructions should be. Would a machine run faster with only simple instructions, such that complicated tasks had to be built out of simple ones? Would it be better to allow the instruction set to contain some complex instructions that are closer to the sort of operations that a high-level language might want to carry out? While there is general agreement that the length, complexity and number of machine instructions in the instruction set has an important effect on a computer's performance, there is no general agreement as to what that effect is.

4.8.2 History of CISC and RISC

High-level languages (HLLs) such as Pascal and C were introduced in the 1970s. Stallings argues (2016, Section 15.4) that the development of ever more complex instruction sets was motivated by the desire of designers to provide support for HLLs, as developers started to use them; designers were looking for ways to simplify compilers and improve performance, and this motivated CISC. Jon Stokes provides more historical background to this movement. He notes that in the late 1970s/1980s technology presented some serious challenges to the designer and to the programmer. Stokes argues that slow and expensive memory (he describes 'core memory' in the 1970s as expensive and 'agonizingly slow') had a significant impact on design. Stokes notes that memory speed improved with 'the introduction of RAM' but that in 1977 '1MB of DRAM cost about $5,000'. He also notes that secondary storage was expensive and slow, so moving data in and out of storage was a major drag on performance. This meant that program code needed to be compact, so that a program could fit into a small amount of memory.

Stokes also notes that compilers of that generation (for the Pascal and C languages, for example) took a long time, and the output was not optimal. He states that:

> As long as the HLL => assembly translation was correct, that was about the best you could hope for. If you really wanted compact, optimized code, your only choice was to code in assembler. (In fact, some would argue that this is still the case today.)

At the time, some argued that as hardware was getting cheaper, and software costs were increasing, the solution was to shift the complexity into hardware. This would mean implementing common functions and procedures from HLLs (such as squaring a number) in hardware: 'This idea of moving complexity from the software realm to the hardware realm is the driving idea behind CISC, and almost everything that a true CISC machine does is aimed at this end.' In other words, make software cheaper by making it simpler to write programs in HLLs, and do this by making assembly code more like a high level language:

[I]f a complex statement in a HLL were to translate directly into exactly one instruction in assembler, then it follows:

- Compilers would be much easier to write. This would save time and effort for software developers, thereby reducing software development costs.
- Code would be more compact. This would save on RAM, thereby reducing the overall cost of the system hardware.
- Code would be easier to debug. Again, this would save on software development and maintenance costs.

Stokes believes that only after the fact was this design approach labelled 'CISC', and that was to differentiate it from RISC, once David Patterson had coined the term RISC, and with it the famous instruction to 'make the common case fast'. (David Patterson was one of the developers of RAID technology, and at the time of writing, works for Google. He has also written a series of textbooks with John Hennessey, including *Computer organization and design: the hardware/software interface*.)

4.8.3 Make the common case fast

Stokes notes that RISC was developed as a reaction to the complex instruction set design approach. In the 1980s, and driven by problems with an increasingly large instruction set, researchers began to run application code to see what the computer spent most time working on, with the idea of designing the architecture for fast implementation of often-repeated tasks. At this point compilers were improving and memory was getting cheaper, so, Stokes argues, the historical circumstances that motivated complex instruction sets were changing. In addition, researchers discovered that compiler writers did not use the complex instructions at their disposable, finding them too difficult to work with. In fact a small percentage of simple instructions were doing the majority of the work. Removing complex instructions, keeping only small and essential instructions with which to build more complex tasks, motivated the design approach labelled *Reduced Instruction Set Computing* by Patterson.

Researchers proposed moving to one-cycle length instructions in order to simplify the implementation of pipelining (see next section) which would improve performance by reducing the average number of clock cycles per instruction. One-cycle length instructions mean 'one machine instruction per machine cycle' (Stallings 2016, Section 15.4, p.575):

A machine instruction is defined to be the time it takes to fetch two operands from the registers, perform an ALU operation, and store the result in a register.

Such simple instructions could be directly executed; that is a small set of these simple instructions could be wired into the hardware, giving fast execution. RISC architecture also features more registers than CISC, in order to make better use of memory, instructions are register-to-register, and only LOAD and STORE are allowed to access memory.

Jon Stokes notes that:

In a RISC architecture, the compiler's role is much more prominent. [...] Since the hardware was now simpler, this meant that the software had to absorb some of the complexity by aggressively profiling the code and making judicious use of RISC's minimal instruction set and expanded register count.

4.8.4 RISC: Reduced instruction set computers

There is no agreed definition of RISC architecture. Stallings (2016, Section 15.4) describes the RISC approach as one where complex instructions are forbidden, and only simple instructions that occur most frequently in compiled programs are used. In Section 15.1 Stallings notes that different groups have different definitions of RISC, but gives the following definition as containing the 'key elements shared by most designs':

- A large number of general purpose registers, and/or the use of compiler technology to optimise register usage.
- A limited and simple instruction set.
- An emphasis on optimising the instruction pipeline.

(Stallings, 2016, Section 15.1)

In Section 15.4, Stallings (2016) notes that there are different approaches to implementing RISC architecture, but gives the following characteristics common to all implementations:

- one instruction per cycle
- register-to-register operations
- simple addressing modes
- simple instruction formats.

One instruction per cycle, as discussed in the above section, means keeping instructions to a limited time period.

Register to register operations meant that only LOAD and STORE operations can access memory, meaning a simpler instruction set giving a simpler (and hopefully faster) control unit. More registers are needed enabling memory use to be optimised by keeping the most recently used operands in the registers, rather than copying over them as in CISC architecture, when executing the next instruction. This reduces LOAD and STORE operations.

Simple addressing modes simplify the instruction set and the control unit.

Simple instruction formats means that either just one, or a very few formats are allowed and 'instruction length is fixed and aligned on word boundaries', Stallings (2016, Section 15.4). Usually, in a RISC machine the instruction length is fixed at 32 bits. This means that a single instruction cannot extend across a **word boundary** (requiring multiple fetches) or across a memory management boundary (requiring multiple address translations). This helps enhance cache fetching and paging (see next chapter for more on paging), since an instruction cannot go over the boundary of a block, or a page. Instruction fetch is optimised because words are being fetched.

Simplified formats simplify the design of the control unit, which is thought to give a faster control unit. Each distinct part of an instruction is called a **field**. Field locations are fixed, meaning that the hardware immediately knows how to read each instruction, and opcode decoding and register access can be done simultaneously.

4.8.5 CISC and RISC today

In Sections 15.4 to 15.5, Stallings (2016) gives 10 characteristics of RISC machines and reproduces a chart showing which modern machines have CISC characteristics, which have RISC, and which have both (Table 15.7, 'Characteristics of some processors'). He notes that some machines are starting to incorporate features of RISC and CISC, denoting a growing awareness that CISC machines can benefit from some RISC characteristics and vice versa.

Stokes argues that the situation today is 'post RISC', with technologies that improve performance being implemented because they improve performance, and not because they are CISC or RISC. He notes that **superscalar execution** which is 'an essential and common component of all modern CPUs', could be said to be 'in the spirit of RISC' but is implemented because it improves performance, not because it is a RISC technology. He argues that other technologies are definitely not RISC, in particular **out of order execution**:

> OOO is one of the least RISC-like features that modern processors have; it directly contradicts the RISC philosophy of moving complexity from hardware to software. In a nutshell, a CPU with an OOO core uses hardware to profile and aggressively optimize code by rearranging instructions and executing them out of program order. This aggressive, on-the-fly optimization adds immense amounts of complexity to the architecture, increasing both pipeline depth and cycle time, and eating up transistor resources.
>
> Not only does OOO add complexity to the CPU's hardware, but it simplifies the compiler's job. Instead of having the compiler reorder the code and check for dependencies, the CPU does it. This idea of shifting the burden of code optimization from the compiler to the hardware sounds like the exact opposite of an idea we've heard before.

Stokes goes on to say that:

> **Branch prediction** is a feature that adds complexity to the on-chip hardware, but it was included anyway because it has been shown to increase performance. Once again, what matters is performance and not principle.

Stokes also argues that the environment now is post RISC because all of the poor or expensive technology that motivated CISC has improved enormously leading to a possible move of complexity back to software. Finally, a key part of the RISC approach was keeping instructions to one clock cycle as much as possible, this approach has evolved into one called 'post RISC' by Stokes – **FISC**, the **Fast Instruction Set Computer**. This means that:

> Any instructions, no matter how special-purpose and rarely-used, are included if the cycle-time can be kept down. Thus the number of instructions is not reduced in modern, post-RISC machines – only the cycle time.

4.8.6 Performance of CISC and RISC computers

Stallings (2016, Section 15.4) argues that two things motivated the development of CISC machines, driven by the desire to provide support for high-level languages (newly in use in the 1970s when the CISC approach was developed).

- **Simplify compilers**: If machine instructions were more complex then the compiler could be simpler.
- **Improve performance**: Performance was to be improved because programs would contain fewer instructions, so that, in theory, they would be smaller and would run faster.

Stallings goes on to argue there is scant evidence that either objective has been achieved. Complex machine instructions are often hard to exploit because the compiler must find those cases that exactly fit the construct. In fact, when programs compiled on CISC machines have been examined, most of the instructions were relatively simple ones, with the complex instructions hardly used.

Stallings also disputes that a CISC approach actually improves computer performance. He notes that a CISC machine-code program may be shorter (i.e. containing fewer instructions), but it may not occupy a noticeably smaller space in main memory. One reason is because complex instructions are longer (that is, occupy more bits in main memory), another is that the register-to-register operations favoured by RISC need less bits to describe. To accommodate longer machine instructions with CISC, changes to the control unit must be made, these changes increase instruction execution time for simple instructions. If we add to this that programs tend to be composed mostly of the more simple instructions, this is another reason that expected performance gains from CISC may not be met.

Stokes accepts that RISC architecture has improved performance, but puts forward no evidence. In fact, there is a consensus among scientists and engineers that RISC improves performance. Stallings notes a lack of definitive evidence to support the claim that CISC architecture improves performance, but suggests that 'circumstantial evidence' implies that RISC machines do (Stallings, 2016, Section 15.4). Stallings presents the following circumstantial evidence.

- With simple instructions 'more effective optimizing compilers' can be built that, for example, can reorder instructions, or take a function call out of a loop so that it does not have to be done at every iteration.
- It seems likely that a simpler control unit built for simpler instructions can execute faster than one built for a CISC machine.
- The intention is that the fixed time for each instruction to complete simplifies and enhances instruction pipelining. Pipelining is believed to be, anecdotally, more effective in a RISC machine.
- RISC processors are better at dealing with interrupts. CISC processors must restrict interrupts to the start or end of an instruction, or implement a way to flush the registers and restart the instruction after an interrupt.

4.9 Parallelism

In the past most computers were serial computers, with only a single processor. Today there is an increasing use of multiple processors in a single computer, allowing the performance gain of **parallel processing** (that is, more than one processor working simultaneously).

4.9.1 Parallel processing (parallel computing)

Harris and Harris (2016, Section 3.6, 'Parallelism') define parallelism, the ability to do multiple tasks at the same time, to be either spatial or temporal. Spatial means having more hardware to implement parallelism. Temporal means splitting a job up into tasks, so that many tasks can be done at once.

Consider the manufacturing process. Without parallelism one worker might make the output from raw material to finished product. Spatial parallelism could be implemented by buying more equipment so that more workers can each make the product alongside each other. The workers do not share any equipment, but each uses the equipment they have to make the finished product from start to finish. Temporal parallelism can be viewed as a production line, where instead of buying more machinery, the process of making the product is split up into stages, each doing one part of the manufacturing process, and arranged in a logical order on a production line. Each worker is assigned to one of the stages on the production line. A new product is started along the line at stage 1, as soon as the preceding product has entered the second stage. In this way each stage of the line is kept busy, with no idle time.

In computer science spatial parallelism would be implemented by using more than one processor; temporal parallelism, known as **pipelining**, would mean making better use of a single processor. For more on pipelining see the next section. For more on the advantages of parallel computing see Stallings (2016), Chapter 17. The learning outcomes do not cover parallelism, and we introduce the definition here to give background for the later discussions of pipelined processors, scalar and superscalar architecture.

4.9.2 Concurrency

When reading about parallelism you may come across the concept of concurrency. In this talk, https://vimeo.com/49718712 Rob Pike, software engineer and author, discusses how concurrency is a design concept that allows for the implementation of parallelism, but is not parallelism. This is given for interest, and is not examinable.

In the example of a factory, designing the manufacturing process as a production line allows for parallelism. It is possible for the production line to run without parallelism, in that each product could move along the line from beginning to end, with a second product only starting once the first had finished and exited the line. So the design of the production line is concurrency, and allows for parallelism to be implemented, but is not parallelism. Parallelism is when the production line is run so that each stage is kept busy by raw materials entering at stage 1, as soon as the previous process has moved to stage 2.

4.10 Pipelining

Pipelining is a means of introducing parallelism into the essentially sequential nature of a machine-instruction program run on a single processor. Without pipelining, the instructions in a program are dealt with one by one. With pipelining, a number of instructions can be dealt with simultaneously; as a result, a program will take less time to run.

Stallings (2016, Section 2.1, p.71) describes and defines pipelining as follows:

> The execution of an instruction involves multiple stages of operation, including fetching the instruction, decoding the opcode, fetching operands, performing a calculation and so on. Pipelining enables a processor to work simultaneously on multiple instructions by performing a different phase for each of the multiple instructions at the same time. The processor overlaps operations by moving data or instructions into a conceptual pipe with all stages of the pipe processing simultaneously. For example while one instruction is being executed, the computer is decoding the next instruction.

First, let us consider the simple fetch–execute cycle depicted in Stallings (2016), Figure 3.3 (p.109). This is a two-stage instruction cycle. An instruction is fetched and then executed. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one. Suppose that there are three instructions A, B and C. In this simplified case, supposing that fetch and execution takes one time unit each, these three instructions would take 6 time units. But with pipelining, the execution of A, B and C will only take 4 time units. Figure 4.9 explains why.

Time unit	1	2	3	4
Fetch	A	B	C	
Execute		A	B	C

Figure 4.8: Two-stage pipelining.

In general, with two-stage pipelining, *n* instructions will take *n* + 1 time units, instead of *2n* units.

4.10.1 A more detailed instruction cycle

We have assumed that the instruction cycle consists of only two steps, or stages; fetch instruction, and execute instruction. But that assumption is too simplistic, because the handling of instructions actually involves more, finer, steps. The instruction cycle on a computer is divided into several stages; the actual number of stages in an instruction depends on the design of the processor.

Here we look at a case where the instruction cycle breaks down to the following six stages:

1. **Fetch instruction (FI)**: read the next expected instruction into a buffer.
2. **Decode instruction (DI)**: determine the op-code and the operands.
3. **Calculate operand (CO)**: calculate the address of the real operands.
4. **Fetch operand (FO)**: fetch each operand from main memory. Operands in the registers do not need fetching.
5. **Execute instruction (EI)**: perform the indicated operation and store the result in the destination register given by the instruction.
6. **Write operand (WO)**: store the result in main memory.

Suppose now we have 5 instructions: A, B, C, D and E. These instructions would take 30 time units without pipelining (5 x 6 = 30, supposing that each stage takes 1 time unit). With pipelining, they would only take 10 time units. This is explained in Figure 4.9.

For simplicity, we assume:

- One time unit per stage in the instruction cycle (in practice, stages may be of different lengths, so there may be some waiting for a stage to complete. There will be one unvarying sequence of micro-operations for most stages, but for EI there will be a number of micro-operations depending on the opcode)
- That every instruction goes through every stage of the pipeline (not true in practice; for example, LOAD does not use the WO stage)
- That there are no memory conflicts. FI, FO and WO all involve a memory access which will not normally be able to take place simultaneously. The table below is shaded to show these **potential** conflicts, remembering that a stage may not be used, or the value needed may be in the cache.

	Time →									
Stage	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
FI	A	B	C	D	E					
DI		A	B	C	D	E				
CO			A	B	C	D	E			
FO				A	B	C	D	E		
EI					A	B	C	D	E	
WO						A	B	C	D	E

Figure 4.9: 6-stage pipelining.

In general, if there are *n* stages in the instruction cycle, then 1 instruction will take n time units; 2 instructions will take n + **1** units; 3 instructions will take n + 2 units; and *m* instructions will take n + m – 1 units. This can be gathered easily from Figure 4.8. As the figure shows, in the first time unit instruction A is fetched. In the second time unit, A is executed; but instruction B is also fetched. In the third time unit B is executed while C is fetched. In the fourth time unit C is executed.

To appreciate how much time pipelining can save, consider 1,000 instructions with a 6-stage instruction cycle. Suppose that each stage takes 1 time unit. Then 1,000 instructions would take 6,000 time units in the case of no pipelining. They would only take 6 + 1000 – 1 = 1,005 time units with pipelining.

But in practice, pipelining is never quite this good, for the following reasons:

- The mechanics of operating the pipeline can slow the execution time of each individual instruction.
- Data, control and resource hazards may lead to pipeline stalls (see next section). In particular, conditional branches are a major reason for pipeline stalls.
- Some stages may take longer than others to complete, delaying the pipeline.
- The pipeline needs a way of dealing with interrupts; this can slow the pipeline depending on how interrupts are handled. In some systems it may be that the pipeline is flushed and restarted after an interrupt, which can cause significant delay.

4.10.2 Pipeline hazards

Hazards in the pipeline cause the pipeline to stall in order to address some conflict. This is also known as a pipeline bubble. There are three types of hazard: resource, data and control.

Resource hazards – Also known as structural hazards. When hardware units are being used by instructions already in the pipeline, these units will not be available for use by other instructions. This will mean executing instructions serially until the conflict is resolved. One solution is extra hardware, but careful design can reduce these conflicts.

Data hazards – Data hazards can occur when two instructions use data from the same register or memory location. Care must be taken to see that the register is updated by the first instruction before the second instruction accesses it. The pipeline may have to stall to allow this to happen.

Data hazards are divided into three categories:

- **Read after write (RAW)**: The first instruction modifies the contents of a register or memory location, the second instruction reads those contents. **Possible hazard**: The read happens before the write.
- **Write after read (WAR)**: The first instruction reads a register or memory location and the second instruction writes to that location. **Possible hazard**: The write happens before the read.
- **Write after write (WAW)**: The first instruction writes to a register or memory location, the second writes to the same register or memory location. **Possible hazard**: The second instruction writes before the first.

One solution to avoid stalls is to design the hardware such that, if a register is to be read and written to in the same clock cycle, then the write happens before the read (Patterson and Hennessey, 2017, Section 4.7 'Data hazards: forwarding versus stalling'); this would prevent RAW hazards. Another possible solution to data hazards is **operand forwarding**. This means forwarding the result of an operation, before it is available from the register, to the stage that needs it.

Control hazards – In the ideal pipeline, we fetch instructions one after another in order. As long as the location of the next instruction is known, this process can go forward. When a branch instruction is fetched, the next instruction location is not known until the branch instruction finishes execution. Branch prediction is used to prevent stalls, by pre-fetching instructions. If the algorithm predicts the wrong branch these instructions must be discarded, and the pipeline stalls. This is a control hazard. The conditional branch instruction is the major obstacle to maintaining a steady flow through the pipeline.

One way to reduce control hazards and improve performance is **delayed branching**, which means rearranging program instructions so that branches are moved to later in the execution stream (namely, out of order execution). Other approaches to improving performance with conditional branches include **multiple streams** (the speculative execution of both branches); **prefetch branch target**; and having a **loop buffer**. You are not required to know the details of any of these schemes.

4.10.3 Number of pipeline stages

Is it the case that more stages will speed up execution time? There are two reasons this may not be the case (Stallings, 2016, Section 14.4).

- There is a time overhead for implementing a pipeline that can increase execution time for an instruction, and this is particularly significant when there are many branches or memory access dependencies (when one instruction relies on the result of an instruction still in the pipeline).
- Control logic is needed to manage hazards and optimise the pipeline; the complexity of this logic increases significantly with each stage added.

This means that more pipeline stages increases the potential for control and data hazards, and does not necessarily improve performance. The number of pipeline stages varies, but most machines will have less than 20 stages, and ARM processors typically have less than 10. See https://softwareengineering.stackexchange.com/questions/210818/how-long-is-a-typical-modern-microprocessor-pipeline for more on pipeline lengths in modern general-purpose machines.

The classic RISC pipeline has five stages:

1. **Fetch instruction (IF)**: read the next expected instruction into a buffer.
2. **Decode instruction (ID)**: determine the op-code and the operands; read data from registers; decide whether or not to stall.
3. **Execute instruction (EX)**: perform the indicated operation. Results can be forwarded at the end of this stage.
4. **Memory access (MEM)**: access memory, results can be forwarded at this stage.
5. **Write back (WB)**: write results to registers.

4.10.4 The floating-point unit and pipelining

To speed up floating point arithmetic early computers had a **floating-point unit (FPU)**, dedicated to floating-point operations, separate from the processor. Starting in the late 1990s the FPU became physically integrated into the processor's chip. The FPU can be viewed as a specialised ALU for floating point operations. Since calculations with integers use very different algorithms for those with floating-point numbers, separating out integer and floating point arithmetic speeds up the operation of the system. Integer and floating-point calculations will be discussed in detail in Chapter 6 of this subject guide.

With an FPU, the control unit can direct integer arithmetic to the ALU, and floating-point operations to the FPU, pointing towards the implementation of superscalar computing.

4.10.5 Superscalar computing

Separate pipelines are maintained, so that instructions can be run independently through them. This means that more than one instruction can be started at each clock cycle. Each pipeline sends its instruction to a different unit of execution; the most obvious example being one integer instruction and one floating-point instruction. This is not multicore computing, there is a single processor, but there may be, for example, four ALUs and two floating-point units.

4.10.6 Superpipelining

Some micro-operations complete in less than one clock cycle. Superpipelining exploits this by splitting the pipeline into further stages. A doubled internal clock allows two tasks to be completed in one external clock cycle. The advantage is that more instructions can be processed; the disadvantage is that more instructions in the pipeline increase the potential for data dependencies to introduce stalls.

4.11 Other ways of enhancing computer performance

Operating systems can also help to make the computer more efficient. We will study how operating systems work in Chapter 5 of the subject guide.

4.12 Overview of chapter

In this chapter we described the workings of the CPU in terms of its components: registers; control unit; and the ALU. We looked in some detail at how the CPU runs a process, and gave an overview of performance issues, before looking at some important ways of improving the performance of the hardware, including cache memory and pipelining. We also introduced some terms that you should be familiar with to give a complete picture of the state of play – multithreading and multicore – that are not examinable.

4.13 Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- explain the components of the CPU and their functions
- explain the instruction format and the instruction cycle (particularly the fetch–execute cycle)
- illustrate how the CPU executes instructions using a concrete example
- explain the details of the execution of instructions; for example, READ and WRITE, in terms of CPU registers and the system bus
- explain the following ways of improving computer performance: clock frequency, cache memory, pipelining, CISC and RISC.

At this stage you are not required to know the details of how the ALU and the control unit work.

4.14 Test your knowledge and understanding

4.14.1 Sample examination question

This question is in three parts, a, b and c.

(a)

(i) An opcode is: [2 marks]

1. A data address contained in an instruction
2. A micro-operation
3. Binary code that specifies the operation to be performed on data
4. None of the above

(ii) The register which keeps track of the execution of a program and which contains the memory address of the next instruction to be executed is known as: [2 marks]

1. Index register
2. Memory address register
3. Memory data register
4. Program counter

(iii) If an addressable memory unit is found in the cache this is called: [2 marks]

1. Cache hit
2. Cache line
3. Cache memory
4. None of the above

(b) Explain the following three different types of mapping procedures in the organisation of cache memory, with their advantages and disadvantages:

(i) direct mapping [3 marks]

(ii) associative mapping [3 marks]

(iii) set associative mapping [3 marks]

(c) Consider a pipeline that has five stages: instruction fetch (IF); instruction decode (ID); instruction execution (EX); memory access (MEM); write back (WB).

I1 Sub r_1; r_2; r_3 store the result of $r_2 - r_3$ in r_1
I2 Add r_1; r_4; r_1 store the result of $r_2 + r_3$ in r_1
I3 Addi r_5; r_1; 2 store the result of value(r_1) + 2 in r_5

i. Does pipeline execution of the above sequence of instructions generate a hazard? If yes, what type of hazard is it? [3 marks]

ii. Fill in the table below to show how this sequence of instructions executes. Show stalls in the schedule, if any, by writing 'stall' in that square. The first row is written for you. How many clock cycles does it take for the instruction sequence to complete? [2 marks]

Clock cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Sub r_1; r_2; r_3	IF	ID	EX	MEM	WB							
Add r_1; r_4; r_1												
Addi r_5; r_1; 2												

(iii) Forwarding is a technique used to reduce pipeline stalls: The result of the computation of the I₁ instruction at the EX/MEM stage is available to be used directly by the data fetch unit or to the execute unit for the I_2 and I_3 instructions. Fill in the table below to show how this sequence of instructions executes if forwarding is implemented. How many clock cycles does it take for the instruction sequence to complete? [5 marks]

Clock cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Sub r_1; r_2; r_3	IF	ID	EX	MEM	WB					
Add r_1; r_4; r_1										
Addi r_5; r_1; 2										

Chapter 5: Operating systems

5.1 Introduction

In the previous chapters we studied how information is represented in the binary system; how it is stored in various types of memory; and how the CPU processes it. However, we have not yet looked at how operations in the computer are regulated. For example, when a text editor is opened, what is going on in the computer? If the CPU is running a program and suddenly the user opens a web browser, what should the CPU do? The user may do several jobs concurrently; for instance, they may listen to some music, while sending email messages. How should the CPU process all these demands? It is obvious that there must be something in the computer system that coordinates the various types of events and actions: this is the operating system.

An operating system (OS) is an important part of every sophisticated computer system. It is an enormously large and complex piece of software with many responsibilities. It controls every file, every device, every section of the memory and every nanosecond of the CPU's processing time. It can be regarded as the executive manager, the boss.

5.1.1 Aims of the chapter

This chapter aims to introduce you to the responsibilities of the operating system, and to consider in detail the operating system's processes for managing I/O and main memory.

5.1.2 Learning outcomes

By the end of this chapter and having completed the Essential reading and activities, you should be able to:

- describe the functions of the operating system and its major components
- explain where the operating system is usually stored, and how it gets started
- distinguish between multiprogramming and time-sharing operating systems
- explain the concept of process, and describe the process control block associated with each process
- explain the four different process queues: job queue, ready queue, intermediate queue, and I/O queues
- explain how the long-term, short-term and medium-term schedulers operate on processes and the process queues
- explain how the operating system manages memory by using techniques such as swapping, partitioning, paging, demand paging and virtual memory
- explain how the operating system manages I/O devices using techniques such as programmed I/O, interrupt-driven I/O and direct memory access
- explain the concept of interrupts, and describe how interrupts are caused, received and handled.

At this stage you are not required to know about any scheduling algorithms. You should know that an I/O module has a buffer (that is, data registers) and a status register. You should be able to explain how an I/O module facilitates communication of data between the CPU and an I/O device. However, you are not required to know about the control register and the control logic inside an I/O module.

5.1.3 Essential reading

- Stallings, W. *Computer organization and architecture: designing for performance.* (Harlow: Pearson Education Ltd, 2016) 10th global edition [ISBN 9781292096858]
  - Section 3.2, pages on interrupts
  - Sections 7.1–7.5 (I/O modules and I/O management), including Figure 7.3 (p.258) and Figure 7.4 (p.261)
  - Sections 8.1–8.3 (operating system, scheduling and memory management), including Figure 8.7 (p.312), Figure 8.13 (p.319), Figure 8.14 (p.320), Figure 8.15 (p.321) and Figure 8.16 (p.322).

5.1.4 Further reading

- Harris, S.L. and D.M. Harris *Digital design and computer architecture.* (Waltham, MA: Elsevier/Morgan Kaufmann, 2016) ARM edition [ISBN 9780128000564]. Chapter 8 'Memory Systems', includes the cache and virtual memory.
- Patterson, D.A. and J.L. Hennessy *Computer organization and design: the hardware/software interface.* (Amsterdam: Elsevier/Morgan Kaufmann, 2017) ARM edition [ISBN 9780128017333]:
  - Section 1.1 (from an historical overview to predictions for the future of computing) Introduction, up to but not including the heading 'What you can learn in this book'
  - Section 1.3 (high-level view of the operating system). Below your program, up to, but not including the heading 'From a high-level language to the language of hardware'
  - Section 5.7 'Virtual Memory'.

5.1.5 References cited

Stack Overflow 'What is an OS kernel? How does it differ from an operating system?' Stack Overflow: https://stackoverflow.com/questions/2013937/what-is-an-os-kernel-how-does-it-differ-from-an-operating-system

5.2 Glossary of key terms

Before we look at the role of the operating system, we will define some important terms.

5.2.1 System software

Software that is designed to run hardware and application programs. This would include file management systems, compilers and device drivers. The operating system is the best-known example of system software.

5.2.2 Operating system (OS)

The OS is system software that provides an interface for the user and acts as the system's resource manager. The OS is run by the processor, just like any other program, but unlike other software it directs the processor in the use of other system resources and in the timing of program execution. In order for the CPU to do useful work, the OS must give control to the CPU, and wait for control to be returned.

5.2.3 The kernel

You might come across different definitions of this, including a long argument about whether the kernel is or is not the OS on Stack Overflow: see, for example: https://stackoverflow.com/questions/2013937/what-is-an-os-kernel-how-does-it-differ-from-an-operating-system. Stallings (2016, Section 8.1) simply defines the kernel as that part of the OS that is in the main memory at a particular time. That is, some functions of the OS are always in main memory, with other parts currently in use/no longer in use are added/removed as necessary.

5.2.4 Device drivers

System software that interfaces between the I/O signals of the OS, and the particular I/O device; for example, mouse, printer, keyboard. The kernel interfaces with device drivers.

5.2.5 Application software

Software that the user interacts with directly, such as a web browser, word processing or email application. Application software is designed to perform a specific task and can be uninstalled without affecting the operation of the system as a whole.

5.2.6 Multi-programming and multi-tasking

Some authorities claim that both terms mean the same thing, while others give slightly different definitions. Multi-programming is considered to be running more than one program on the same processor. Multi-tasking is considered to be a more general idea, where the tasks could be separate programs, or it could be a program divided up into separate tasks. This subject guide is going to stick with the definition provided by Stallings (2016, Section 8.1) who refers to multi-programming or multi-tasking as the ability of the operating system to run more than one program at a time, so that the processor need never be idle when waiting for I/O, since it can switch to another job. Stallings notes that: 'The term multi-tasking is sometimes reserved to mean multiple tasks within the same program that may be handled concurrently by the OS, in contrast to multiprogramming, which would refer to multiple processes from multiple programs. However it is more common to equate the terms multitasking and multiprogramming as is done in most standard dictionaries'. By standard dictionaries Stallings means technical dictionaries, such as those provided by the Institute of Electrical and Electronics Engineers (IEEE).

5.2.7 Memory management unit (MMU)

This is hardware that supports the OS in implementing virtual memory and paging by translating virtual addresses into physical ones. The MMU sits between the processor and main memory.

5.2.8 Physical address

An actual location in main memory **(or on a memory mapped I/O device)**.

5.2.9 Virtual address

A virtual address is a reference to a physical address.

5.2.10 Virtual memory

Achieving the illusion that main memory is larger than it really is by keeping only the active part of a program in main memory at any one time. By implementing virtual memory the OS makes it possible for a process to be larger than main memory, and still run effectively.

5.2.11 Paging

A memory management process controlled by the OS that will be discussed in this chapter.

5.2.12 Memory management

Memory management is one part of the OS that is always in main memory; namely, it is in the unchanging part of the kernel. In a multiprogramming system, the main memory is divided into two; one part for the OS (the kernel) and another part for applications. The application part of memory is sub-divided to allow for multiple processes, and this sub-division is done by the OS.

5.2.13 Time-sharing

Used by computers with multiple users, to give each user the illusion of having all of the processor's attention. Time-sharing was developed for use with early (third generation) computers, to allow multiple users to access the very expensive machine through different terminals. An operating system controlled rapid switching between users. Now a computer is likely to have one user, but switch processor time rapidly between programs.

5.2.14 Fragmentation

Fragmentation describes a process that can happen in main memory or in certain types of storage, where, as memory is added to and deleted over time, space can be wasted, and it may also be that the system is slowed because the most efficient use is not being made of the memory available. **Internal fragmentation** refers to the organisation of main memory with a fixed partition size. This means that every process gets allocated a fixed amount of memory, whether or not it needs it, which can lead to unused memory that the system cannot recover. **External fragmentation** means that there are only small amounts of contiguous usable memory in between used (or allocated) memory locations. There may be a lot of free memory, but if a system is severely fragmented, then there will not be any large contiguous areas. There is also **data fragmentation**, which happens when data is written into a storage medium that is severely fragmented. The data will need to be broken up and stored in many separate locations, rather than contiguously.

5.2.15 ROM

Non-volatile memory that is permanently in main memory. Used for tasks such as booting the computer. ROM memory used to be read only, or write once, since once written to it could not be changed because it was reserved for critical processes that should not be altered. Today ROM memory can be written to more than once, but this is not straightforward, and is achieved by such things as exposing the hardware to UV light in order to reset it.

5.2.16 Polling

Polling is the periodic checking of other programs or devices by a controlling program or device, to determine their status.

5.3 What is an operating system?

You may be familiar with operating systems from your mobile phone or tablet (e.g. iOS, Android) or from your laptop or desktop system (e.g. Windows, Linux). You are also likely to have other devices with operating systems, such as a digital camera or digital music player; these systems are referred to as **embedded systems**. An embedded system is software within a product – such as a car, fridge, camera, photocopier, toy or security system – rather than on a general-purpose machine.

Embedded systems may well have sensors that allow them to monitor and respond to the external environment without human intervention (such as turning on the lights). With the embedding of mobile transceivers, these products can communicate with each other. The 'Internet of things' (IoT) is the much anticipated interconnection of various smart devices, and seems a step closer to realisation with virtual assistants such as Amazon's Alexa that are themselves operating systems, and liaise with the operating systems of other embedded devices within a home ('Alexa, activate the robot floor cleaner and put some music on to cover the noise').

Up to the 1980s, operating systems had text-based user interaction (Unix, DOS), but after that graphical user interfaces (GUIs) were developed. Alexa's human interface is a voice user interface (VUI) bringing the world of Star Trek that one step closer, for good or ill.

Operating systems of embedded devices are different in various ways from those for general-purpose computers; for example, they may or may not have a human interface and the interfaces they do have can range from a flashing light, to Alexa's VUI. Similarities include the ability to be patched to fix security issues, and to be upgraded to the latest version.

Patterson and Hennessy (2017, Section 1.1, 'Introduction') claim that the world is now entering the 'post PC era'. They state their belief that the world is in transition, from personal computing to the 'personal mobile device (PMD)'. PMDs are mobile, battery operated, wireless connected to the internet, have a touch screen instead of a keyboard and mouse, and may be voice operated. They quote statistics showing that smartphone and tablet sales combined have overtaken PC sales.

Despite the claims of Patterson and Hennessy (2017), the rest of this chapter will be concerned with the responsibilities of a typical operating system in a general-purpose machine, such as a laptop or desktop system.

5.3.1 The operating system in a general-purpose machine

The operating system in a general-purpose computer has two main tasks,

1. providing an interface between the user and the computer
2. managing the computer's resources.

We will take a very brief look at the operating system's user interface, but shall not concern ourselves with the details of the user interface.

Resources that have to be managed by the operating system are: processes; memory; I/O; and networking. This course does not cover networking, hence this chapter will focus on process, memory and I/O management.

5.3.2 The operating system as a user/computer interface

Recall that the instructions processed by the CPU are in machine-code, and that they are minute operations, such as fetching a datum from main memory, adding up two numbers stored in the registers, and so on (see Chapter 4 of the subject guide). However, the instructions that the user is interested in are at a much higher level of abstraction. For example, the user is often interested in the following commands:

load a program	**save** a file
print a file	**run** a program
open a file	**copy** a file

Each of the above commands would correspond to a large sequence of machine-code instructions. It would be very inconvenient if the user had to provide such a sequence of machine-code instructions every time they wanted to open a file, run a program, and so on. Besides, the user might need, for the same command, to provide a different sequence of machine-code instructions when using different computers, because different computers might employ different instruction sets and different designs of CPUs.

The operating system on a computer solves this problem for the user, allowing the user to interact with application software at the highest level of abstraction. Details such as the design of the hardware, or how the CPU works, are hidden from the user who can issue commands such as open and print, and regard them as being primitive.

Commands such as the ones listed above are therefore system commands, as they are provided by the operating system. Stallings (2016, Section 8.1) provides a list of services that the OS typically provides, viewed as a user/computer interface. However, only a brief description of the user/computer interface is given here, as this chapter is mostly concerned with the operating system as a resource manager.

5.3.3 The operating system as a resource manager

The computer has a set of resources for storing, moving, and processing information, such as the memory devices, the system bus, the CPU and I/O devices. The operating system manages these resources and allows them to be used in an efficient manner.

The operating system typically does the following jobs.

- Processes (jobs) management:
  Which program should the processor execute next? How much time should be given to each program?
- Memory management:
  How can the available memory be best used to keep the processor as busy as possible?
- I/O management
  The OS controls access to files, and decides when a program can use an I/O device.
- Network management
  Which computer should be used to execute this program?

We shall examine the first three tasks of the operating system in this chapter. The fourth task, network management, will not be dealt with here.

5.4 A brief history of operating systems

In the first generation of general-purpose machines there was **no operating system**, and hence no abstraction. The programmer interacted directly with system hardware. If an error happened during a program run, the programmer would have to examine the registers and main memory to discern the reason. The first operating systems were simple **batch programming** ones that introduced a level of abstraction, in that the programmer could no longer directly interface with the processor. In batch systems several jobs were input by a computer operator, and a monitor, resident in main memory, would queue the jobs and submit them to the processor. The processor would run each program until completion, or until an error was triggered. The result of each job was sent to a printer.

In early machines, batch operating systems improved the utilisation of the processor, but even so the processor was often idle, because of the time taken by I/O. Another step forward was taken with **multi-programmed batch** systems, allowing more than one program to be admitted to main memory. Hence, when the processor was waiting for I/O, it could switch to another program. To implement this, hardware supporting I/O interrupts was critical. With interrupts, the processor could turn over I/O processing to a device controller and switch to another program. Once the I/O processing had finished, an interrupt signal would be sent, and the OS would decide what process would then get processor time.

Multi-programming could also be implemented for interactive programming. In this case, multiple users interact with the system, each from their own terminal, each user running one process. Instead of the job control instructions being provided with the job (as in programs run by batch OS) the user entered commands at the terminal. Since different users could run different programs, this was **time-sharing** (as well as multi-programming). The OS switched rapidly between users, to give the illusion to each user that they were the only one using the system.

5.4.1 Multi-programming operating systems

Multiprogramming operating systems are described by Stallings as 'the central theme of modern operating systems' (Stallings, 2016, Section 8.1). They make efficient use of resources by dividing main memory up among several processes at once. It is the job of the OS to allocate memory to processes, to make efficient use of the CPU, to prevent any one process from monopolising processor time, and to protect memory, so that processes do not interfere with each other's instructions and data in main memory.

5.4.2 Interrupts

In any multi-programming system, control must pass between the OS and the CPU. The OS must give control to the processor, in order for the processor to do the tasks that the OS has directed it to do, but the OS needs control back again in order to prepare further work for the processor. The passing of control from the processor to the OS is handled with interrupts.

An **interrupt** is a signal to the CPU to transfer control to the OS's **interrupt handler** (more on this later in the chapter). Reasons for an interrupt include:

- **Time**: in a multi-programming system, the operating system might rotate processes by only allowing each process a fixed amount of processor time. The processor might also run a timer for other processes that it needs to undertake regularly.
- **Process issues**: the process should not attempt to alter the memory area of the kernel, or that of other processes in main memory. If it does the processor should detect this and issue an interrupt. Running processes may also generate arithmetic errors, such as divide by zero, or a NaN (not a number) error.
- **Hardware failure**: such as a bad sector on a disk.
- **I/O operations**: the completion of an I/O operation (more on this later) can generate an interrupt. I/O modules also signal error conditions to the processor.

Activity 5.1

Read Stallings (2016), Section 8.1: 'Operating system overview' and Section 8.2: 'Scheduling' and then answer the following questions:

1. Explain two problems of early computers without operating systems.
2. What was a monitor?
3. What does it mean to say that I/O instructions were privileged in early batch operating systems?
4. What motivated the development of multi-programming?
5. You are using a computer from a terminal. A number of other people are at other terminals using the same computer. Why might the response time seem to you to be as good as if the computer were single use?

5.5 How the operating system gets started

When a computer is turned on the kernel of the operating system is loaded into main memory and executed. This involves the following steps:

1. The program counter starts with a particular predetermined physical address (there are no logical addresses when a computer is first turned on).
2. At this address, a program saved in ROM memory called BIOS is stored.
3. The BIOS performs various tests, and looks for the OS on the hard drive, finding and loading code that boots (loads) the kernel, the kernel booting code.
4. This program is executed. It directs the CPU to load the operating system's kernel from the hard disk.
5. The kernel is now in main memory. The CPU starts to execute it. The kernel takes over and begins controlling the computer's activities.

5.6 Process management

Multiprogramming means that the operating system can handle a number of programs at the same time. However, it is more precise here to talk about a process rather than a program. This is because the same program can be associated with several processes or jobs. For example the user may have two documents open, each one in a separate instance of the same text-editing package. In this case, there is only one program (the text editor package), but from the viewpoint of the operating system, there are two processes or jobs to deal with. So, at any time the operating system may be dealing with many processes, which must be managed, and a decision made with each of them as to what to do. For example, a process may be executed, or allowed to wait in main memory, or temporarily swapped out of main memory, and so on.

5.6.1 Scheduling queues

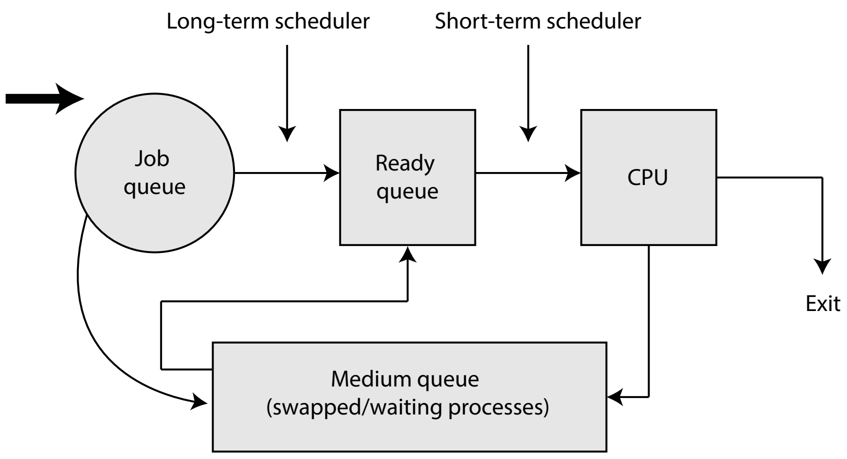


Figure 5.1: Scheduling queues.

Since there may be many processes for the operating system to deal with at any time, the operating system must decide which processes can be executed and in which order they should be executed. This is called **scheduling**. Stallings states that 'the key to multi-programming is scheduling', (Stallings 2016, Section 8.2).

The operating system maintains a number of queues of processes for scheduling purposes:

- a **job queue** (also called a long-term queue)
  As processes enter the computer system, they are put into a job queue, which consists of all jobs waiting to be processed by the system, and usually resides on the hard disk.
- a **short-term queue** (also called a ready queue)
  This queue holds all the processes which are in main memory and which are ready for execution. The short-term queue resides in main memory.
- an **intermediate** (medium-term) **queue**
  This queue holds the processes waiting for an I/O operation and swapped out of main memory, and usually resides on the hard disk. Medium-term scheduling is part of managing the swapping function, described later. In some systems new processes are sent straight to this queue by the long-term scheduler.
- an **I/O queue for each I/O device.**
  Several processes may need to access the same I/O device. If that happens, the processes will be held in an I/O queue on the I/O device.

A process is first added to the job queue. Once chosen by the long-term scheduler, it normally joins either the ready queue, although in some systems it is sent first to the medium queue. A process that requires an I/O operation is put in the I/O queue on the relevant I/O device. When a process is completely executed, it is removed from all the queues and its process control block (see next section) is destroyed. If the CPU is idle, because too many processes are waiting on I/O, a process requiring an I/O operation may be temporarily swapped out of main memory on to the hard disk (joining the medium queue), in order to free some space in main memory for other processes to come in.

Long-term scheduling	The decision as to which processes to admit to the system for processing; top-level, infrequent decisions. Controls the degree of multi-programming in a multi-programming system.
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory, helps to balance the processor's workload. Only exists in systems that implement swapping.
Short-term scheduling	The decision as to which ready process will next be executed by the processor. Fine grained decisions, executes frequently.
I/O scheduling	The decision as to which processes pending I/O request shall be handled by an available I/O device.

Table 5.1: Scheduling processes.

5.6.2 The process control block (PCB)

Each time a job is accepted by the long-term scheduler, the OS creates a **process control block** (PCB), which makes the job into a process in the 'new' state. Once the OS has completed the PCB, the process is in the 'ready' state. A process can be in one of five states. Firstly, it is new; when admitted to the short-term queue it is in the ready state. After that it can be running, terminated, or blocked (waiting for I/O). If suspended because it reached a time limit, a process is returned to the ready queue. For a diagrammatic representation, please see Figure 8.7 in Stallings (2016) on p.312.

If anything new happens to a process, its process control block will be updated by the operating system. For example, if a process is waiting for an I/O operation and is therefore swapped out of main memory (see Section 5.8.1 of the subject guide on 'Swapping'), then at least the program counter of the process must be recorded (if later the process is swapped back, the operating system will know which instruction in the process it should execute first). Other entries in the process control block, such as the state, I/O status and the accounting information, need also to be updated. The PCB is deleted once the process has completed.

A process control block contains the following information:

- a unique identifier
- the state of the process, whether it is New, Ready (to be executed), Running, Waiting (for I/O), or Terminated
- the priority of the process
- program counter, the address of the next instruction to be executed for the process
- memory pointers, which indicate the start and end address of the process in memory
- context data, data that was in the processor's registers when the process was executing
- I/O status information, listing the files and I/O devices associated with the process and outstanding I/O requests.
- Accounting information, which includes the amount of CPU time used, time limits, etc.

5.6.3 Long-term scheduling

The long-term scheduler executes infrequently, making the high-level decision of which jobs to turn into processes and to admit to the system for processing. Since the long-term scheduler controls the number of processes in the system, it controls the degree of multiprogramming. The long-term scheduler has to decide: (1) whether or not the OS can accept any more processes at that point; and (2) what job to turn into a process next. In a multiprogramming system, the scheduler makes decisions based on criteria such as priority, expected executing time, I/O requirements, etc. In a time-sharing system, the scheduler will accept all authorised users until the system is saturated.

In general, most processes are either I/O bound or CPU bound. An I/O-bound process spends a great deal of time doing I/O operations; whereas a CPU-bound process does not require I/O operations frequently. The long-term scheduler should select a good mix of both types of processes so that there is a combination of I/O-bound and CPU-bound processes in the ready queue. The most common problem is that the CPU is idle because all processes are waiting for I/O.

Long-term scheduling is very important in multiprogramming systems, but in time-sharing systems the long-term scheduler may be absent or minimal: new processes can go straight from the job queue to the ready queue (if the system is not saturated). This is because in time-sharing systems, each process is given a slice of the CPU's processing time, and can therefore be executed by the CPU readily.

5.6.4 Short-term scheduling

The short-term scheduler decides what to do with the processes in the ready queue (e.g. which process should be executed next, and for how long). The scheduler executes frequently, resides in main memory, and is fast. A process, which is in main memory, is either in the ready queue (if it is ready to run); or in an I/O queue (if it is waiting for an I/O device).

The short-term scheduler is also known as the **dispatcher**. To understand how the short-term scheduler works, we need to consider the following three questions, which will be dealt with in turn below.

1. Why does the short-term scheduler need to select a new process for the CPU?
2. What happens when switching from an existing process to a new process?
3. How should the short-term scheduler select a new process?

Why does the short-term scheduler need to select a new process for the CPU?

Assume that a process *A* in the ready queue is running. There may be a time when:

- In working through process *A*'s machine instructions, the CPU reaches a request for an I/O operation.
- Process *A* causes an interrupt (e.g. because of an error, or it has used too much of the CPU's time).
- An I/O device has finished an I/O operation and issues an interrupt.

What happens when switching from an existing process to a new process?

In all of these cases the processor saves the context data and program counter for *A* into *A*'s PCB, then transfers control to the OS; process *A* is suspended, but since its PCB has been updated, it can be started again from where it stopped.

The short-term scheduler must adjust the queues in main memory. If process *A* requires I/O, then it is put in the I/O queue of the relevant I/O device. If process *A*'s suspension is caused by the process itself, then *A* is put back into the ready queue. If process *A*'s suspension is due to an I/O device's having finished an I/O operation, then the process containing the I/O operation will be deleted from the device's I/O queue and put back into the ready queue.

If the short-term scheduler chooses process *B* as the next process for the CPU, then the OS tells the processor to restore *B*'s context data, and begin process *B* from where it was suspended, using process *B*'s program counter to get the next instruction.

How should the short-term scheduler select a new process from the ready queue?

The short-term scheduler does this by following a scheduling algorithm. There are several possible scheduling algorithms, such as:

- **First-come, first-served** – the process that requests the CPU first is allocated the CPU first.
- **Shortest job first** – the process with the shortest time length is selected.
- **Priority scheduling** – a priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- **Round-robin scheduling** – each process is given a small unit of time, called a time quantum (10–100 milliseconds), and the ready queue is treated as a circular queue. The scheduler goes around the ready queue and picks the next process in the queue. This algorithm is the one most likely to be used, possibly combined with some priority scheduling.

5.6.5 Medium-term scheduling

Medium-term scheduling is part of the swapping process, which will be described further in Section 5.9 'Memory management'. If all the processes in main memory are waiting for I/O then the processor swaps one of these processes to the intermediate queue on the hard disk. The OS then brings in another process from the intermediate queue, or it accepts a new process request from the long-term queue. Medium-term scheduling helps to balance the system's workload, by managing the degree of multi-programming. Swapping to the hard disk and back to main memory is an I/O process; since I/O is slow there is a potential to slow down processing, rather than speed it up. Stallings reports that usually simple swapping does increase throughput (2016, Section 8.3) but virtual memory systems need no swapping and increase performance more than swapping; that is, OS's that implement virtual memory have no intermediate queue. This will be discussed further in Section 5.9 of the subject guide.

5.6.6 I/O scheduling

All the processes in main memory that require access to an I/O device are put in the device's I/O queue. Scheduling is also needed for an I/O queue in order to improve overall system performance. For example, suppose that the read/write head is near the centre of the hard disk, and that three processes A, B and C need to read data from the disk. Process A requests a block of data near the edge; B requests a block near the centre; and C also requests one near the centre. The operating system can reduce the distance that the read/write head travels by serving the processes in order B, C, A.

5.7 I/O operation and I/O modules

A system would be of limited use without some way of saving processed data, and of inputting both saved and new data. I/O is any transfer of data between a **peripheral** (that is, a device directly attached to the computer) and main memory/the CPU. I/O is managed by the operating system.

5.7.1 Types of I/O devices

Below are some examples of I/O devices (peripherals):

- keyboard
- mouse
- speakers
- monitor
- scanner
- printer
- CD-drive
- magnetic disks
- tape back-up systems.

Note that disks and tape storage are controlled by I/O devices, so are functionally memory, and structurally I/O.

Let us now see how the operating system manages I/O operations and I/O devices with the help of I/O modules.

5.7.2 I/O modules

The operating system manages I/O, but does not normally interface directly with I/O devices; but instead, it communicates via one, or more, I/O modules. I/O modules communicate via the system bus and control one or more peripherals. They are interfaces between the CPU and main memory on the one side, and I/O devices on the other. An I/O module may also be called an I/O interface, I/O processor, I/O channel and other terms, but we will use the term I/O module, since this is the term that Stallings (2016) uses.

There are three main reasons why the OS does not interface directly with I/O devices.

- **Control logic**: Each I/O device has its own device-specific control logic. Without I/O modules, the necessary logic for control of every I/O device the processor may encounter would have to be incorporated into the processor.
- **Data formats**: I/O devices may also use different word lengths than the CPU and main memory. I/O modules hide these details from the processor. (Note: transformation of data to electrical energy from other forms and vice versa is done by a transducer in the I/O device itself. For example, on a CD, 1s and 0s are stored with the aid of tiny grooves known as 'pits'. The CD-Drive will handle the transformation of such data.)
- **Data buffering**: The CPU can normally send data at a much faster rate than a peripheral can receive it. There are a few peripherals that can transfer data faster than the CPU or main memory. In either case, the I/O module buffers the data.

Hence an I/O module contains the logic necessary for communicating with the peripheral and the system bus, buffers data from the CPU/main memory so that a peripheral receives data at a rate it can manage, and buffers data to the CPU/main memory so that the processor is not waiting for slow I/O devices (or alternatively overwhelmed by very fast I/O). An I/O module aids communication between the CPU and the peripheral by managing different data formats and word lengths; many I/O devices send data 8 or 16 bits at a time, a CPU may have a word length of 32 or 64 bits. In addition to these responsibilities, the I/O device must also report any errors to the CPU (e.g. transmission errors, or device-specific errors), and manage the timing of I/O.

Data buffering is an essential task of an I/O module, and means that the module must be able to operate both at the speed of the device and at the speed of main memory. Control and timing is also very important, since without an I/O module to interface, the processor might unpredictably receive I/O communication (such as keyboard input). The intercession of the I/O module allows the processor to manage the sharing of internal resources such as the system bus and main memory without chaotic interference from I/O devices.

To sum up, the responsibilities of an I/O module are:

- processor communication
- device communication
- control and timing
- data buffering
- error detection and reporting.

5.7.3 I/O module and the system bus

In the first place, I/O devices must be connected to the rest of the computer system, especially to the CPU and main memory. In a system with a bus, an I/O device is connected to the system bus via an **I/O module**, as shown in Figure 5.2.

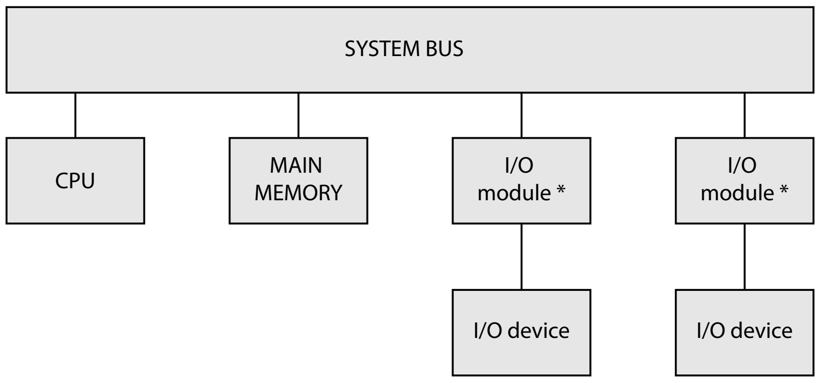


Figure 5.2: I/O modules and the bus connection.

**A single I/O module may control more than one I/O device*

5.7.4 Structure of an I/O module

Data transferred to and from the module is buffered into one or more data registers. The module will also have one or more status registers. It may have a control register or registers, or the status registers may double up as control registers and accept control signals from the processor. Control signals are received from the processor via control lines; the I/O module has communication logic that interprets the control signals. The I/O module is directly connected to its peripheral(s), and for each peripheral has device-specific interface logic. See Figure 7.3 on p.258 of Stallings (2016).

5.7.5 Communication between the I/O module and the I/O device

How does an I/O module communicate between the CPU and an I/O device? It does this through control, data and status signals.

- **Control signals** determine what the I/O device will do, namely:
    - send data to the I/O module
    - receive data from the I/O module
    - report status
    - perform an action particular to the I/O device, such as rewinding a tape.
- **Data signals** are the bits to be sent or received by the device.
- **Status signals** report the device's status, such as ready, or not-ready, or device specific signals such as 'out of paper'.

5.7.6 Communication between the I/O module and the processor

The I/O module communicates with the CPU via the system bus as follows.

- The CPU sends the address of the I/O device over the address bus.
- I/O commands are sent to the I/O module (and responded to) via the control bus.
- Data exchange is done over the data bus.

I/O commands can be **control** or **status** commands. Control commands include READ and WRITE commands, and device specific commands. Some commands may include a parameter sent via the data bus. With status commands the CPU queries the readiness of the device, which can be READY, BUSY, or some ERROR condition.

Control and status commands sent via the control bus are broken down by Stallings further (2016, Section 7.3), as follows:

Control – activate a peripheral and issue device-specific commands, such as to move a reading head, or rewind a tape drive.

Test – test the status of the peripheral; report on the status of an I/O instruction (completed or ongoing) and if any errors happened.

Read – an instruction to get data from the peripheral. When the processor is ready it requests the data by asking the I/O module to place it on the data bus.

Write – an instruction to take data from the data bus and write to the peripheral.

For example, consider the following steps in a data transfer from an I/O device to the CPU/main memory:

- The CPU asks for the status of the I/O device via the control bus.
- The I/O module reports that the device status is READY.
- The CPU tells the I/O module to READ data.
- The I/O module interfaces with the I/O device to get a unit of data from the device, buffers the data and reports success to the CPU.
- The CPU requests that the data is placed on the data bus.
- The data is transferred from the I/O module to the CPU via the data bus.

5.7.7 Advantages of I/O modules

'An I/O module functions to allow the processor to view a wide range of devices in a simple-minded way.' (Stallings 2016, Section 7.3, p.259)

- I/O modules are an example of abstraction. They 'hide the details of timing, formats and electromechanics' of I/O devices from the processor, thus saving space in main memory and allowing the CPU to interface with peripherals by means of simple commands.
- By performing functions such as buffering and timing of I/O interactions, an I/O module allows the CPU to work more efficiently.

5.8 Techniques of I/O operation

There are three ways of managing I/O processes:

- programmed I/O
- interrupt-driven I/O
- direct memory access.

We shall look at each of these in turn.

5.8.1 Programmed I/O

When the processor reaches an I/O instruction, it saves the process it is working on and issues a command to the appropriate I/O module.

In programmed I/O:

- The CPU sends an I/O command to I/O module.
- The I/O module carries out the I/O command and updates its status register (and any other registers as necessary).
- The processor keeps checking the status register of the I/O module, until it finds the command has been completed, or has resulted in an error message. The constant status checking is known as 'polling' which is why programmed I/O is sometimes known as 'polled I/O'.
- While waiting for the I/O device to report 'ready' the processor does no other work.

Programmed I/O slows the performance of the system, because the CPU spends much time waiting on the status of the I/O device, during which time many cycles pass in which the processor could have done useful work. In programmed I/O the I/O module has no responsibility to inform the processor that it has completed the command; the I/O module's responsibilities have ended once it has completed the command and updated its registers as necessary.

Figure 7.4(a) (see Stallings, 2016, Section 7.3) gives an example of the use of programmed I/O to read in a block of data, which consists of many words, from an I/O device (e.g. an audio clip on a flash drive). If each instruction is to read a word and write it to main memory, then reading a block of data corresponds to executing a large number of such instructions sequentially.

When executing a single instruction, the CPU first issues a read command to the I/O module. The I/O module then tries to obtain a word from the I/O device; this takes some time, and when that is done, the word will be in the data register in the I/O module, and the status register of the I/O module will show ready. But during this process the CPU is idle, except for checking the status of the I/O module. When the I/O module status is ready, the CPU requests that the word contained in the I/O module's data register is placed on the data bus for transfer to main memory. This procedure continues until all the instructions are executed – when the whole block of data is transferred into main memory.

As Figure 7.4(a) on p.261 in Stallings (2016) shows, when transferring **any single word** from the I/O device to main memory, the CPU may waste some time on checking the status of the I/O module (that is, on waiting for the completion of an I/O operation).

Thus, a great deal of CPU time is wasted when a large block of data is transferred.

5.8.2 Interrupt-driven I/O

The problem with programmed I/O is that the CPU has to wait a long time for the I/O module concerned to be ready for reception or transmission of data. It is better if the CPU can do other things while the I/O module is performing an I/O operation with the I/O device. The I/O module can then let the CPU know when it has completed an I/O operation – this can be achieved in terms of an **interrupt**.

Figure 7.4(b) in Stallings illustrates the interrupt-driven approach. In Figure 7.4(b), when the CPU wants to read a word from an I/O device, it issues a READ command to the device's I/O module. The CPU then goes to do other things. When the I/O module finishes reading a word from the device, it sends an interrupt signal to the CPU and waits for the data to be requested by the processor. When the CPU makes the request, the I/O module places the data on the data bus, and is ready for the next I/O request. In this way, the CPU is still doing many jobs while the I/O operation is taking place. This approach is therefore more efficient than programmed I/O, but still takes up a lot of the processor's time as every word transferred goes through the CPU.

A number of details are needed to make clear how interrupt-driven I/O works.

- **How does the I/O module send an interrupt to the CPU?**

  The I/O module sends an interrupt to the CPU over a control line. The I/O module is connected to the control bus, as Figure 5.2 shows. When the I/O module has read a word from the I/O device, it puts the word in the data register, and sends an interrupt to the CPU, again over a control line.

- **How does the CPU go to do other jobs while an I/O operation is taking place?**

  Recall from the section on 'Process management' that the operating system maintains a ready queue, which consists of all the processes in main memory that are ready to be executed. When one process is waiting on I/O, the short-term scheduler updates the process control block of the process and saves it (so that the process can be resumed later after the I/O operation is completed). It then selects another process in the ready queue for the CPU. The process waiting on I/O is put on the I/O queue of the relevant device.

- **How does the CPU know whether an interrupt has occurred?**

  The CPU executes instructions in main memory in terms of instruction cycles. To accommodate interrupts, an **interrupt** stage is added to the instruction cycle. At the end of an instruction cycle (namely, in the interrupt stage), the CPU tests for interrupts. If no interrupt has occurred, the CPU carries on executing the next instruction of a process.

- **What happens if the CPU detects an interrupt?**

  If an interrupt has occurred, the CPU acknowledges it, allowing the I/O module to remove the signal from the control line. The CPU saves the data of the current process onto its stack (its own internal memory).

- **How does the CPU process interrupts?**

  When an interrupt has been detected, the CPU loads the program counter with the entry location of the **interrupt-handling program**, and starts executing that program. There may be a single program, or several, depending on the design of the processor and the operating system. If more than one, the processor may have to query the I/O module for the correct program, or the information may have been in the interrupt signal.

- **What happens when the interrupt has been handled?**

  When the interrupt is dealt with, the interrupt handler may ask the CPU to resume the execution of the process that was interrupted; in which case the CPU will restore the process that was interrupted from the stack. If the interrupt handler decides that another process now takes priority, it will update the interrupted process's PCB, and switch the CPU to another process. Hence the interrupt handler decides whether to push data on to the stack in order to resume the paused (interrupted) process, or whether to update the PCB in order to load another process after dealing with the interrupt.

- **What happens if there is more than one interrupt?**

  At any time there may be many interrupts. There are various strategies for handling multiple interrupts, including prioritising the interrupts, which this subject guide will not consider.

With this in mind, let us look again at the example shown in Figure 7.4(b) in Stallings (2016).

- Let us call the process of reading a block of data from an I/O device 'Process α'. When Process α requires the I/O module to read a word from the device, the short-term scheduler puts α on hold, and selects another process, β, for the CPU to execute.
- When the I/O module has obtained a word from the device, it raises an interrupt. The CPU receives the interrupt while executing β.
- The CPU jumps to a particular address in main memory and starts executing the interrupt-handling program.
- The interrupt handler finds out that it is Process α that has raised the interrupt, so it updates the process control block of β, and asks the CPU to resume the execution of α, based on the saved process control block of α.
- When Process α requires the I/O module to read another word from the device, the short-term scheduler puts α on hold again, and selects another process for the CPU to execute. This process could be β, which was temporarily stopped; it could be another process in the ready queue. The operating system continues this procedure until all the processes, α and β included, are completed.

5.8.3 A comparison between programmed I/O and interrupt-driven I/O

Programmed I/O does not make efficient use of the CPU, because the CPU has to be idle when waiting for I/O commands to complete. But when transferring a block of data, the data transfer rate can be quite high, because the CPU is dedicated to that job only. Interrupt-driven I/O makes more efficient use of the CPU, but the data transfer rate can be lower than programmed I/O, because the CPU has to spend additional time handling interrupts.

However, programmed I/O and interrupt-driven I/O suffer from two drawbacks according to Stallings:

- The I/O transfer rate is limited by the speed with which the CPU can test and service a device.
- The processor is tied up in managing I/O transfer; a number of instructions must be executed for each I/O transfer.

(Stallings, 2016, Section 7.5)

In other words, both methods slow down the processor, and limit the speed of data transfer.

5.8.4 Direct memory access

For a device that does large transfers, such as a disk drive, it seems wasteful to use the CPU to spend time dealing with the transfer of every word. Many computers have a special-purpose processor on the system bus, called a **direct memory access** (DMA) module.

The basic idea is that when reading/writing a block of data:

- The CPU tells the DMA whether it is issuing a read or write command, by using the read or write control line as appropriate. It gives the DMA the following information via the data lines:
  - the address of the I/O device
  - the starting address in main memory to read from or write to
  - the number of words to be transferred.
- The DMA then transfers the block of data, one word at a time, without intervention from the CPU.
- When DMA completes the block transfer, it sends an interrupt to the CPU.

The DMA module acts like the CPU, transferring data between main memory and an I/O module using the data bus. It sits between the CPU and the I/O module of a device. The DMA module interacts with the I/O module and main memory, transferring a word between them each time until all the block of data is transferred. However, during this transfer process, the CPU is not involved, so the CPU is free to do other jobs (namely, execute other processes in the ready queue).

Now, there seems to be a problem: how can the CPU and the DMA module work at the same time, since they both need to access the bus and main memory? The answer is that they cannot really work at the same time. When the DMA module uses the bus to transfer data from or to main memory, the CPU cannot also use the bus to transfer data from or to main memory. The DMA must, when transferring a word between an I/O module and main memory, 'use the bus only when the CPU does not need it, or it must force the processor to suspend operation temporarily. This latter technique is more common and is referred to as **cycle stealing,** because the DMA module in effect steals a bus cycle' (Stalling, 2016, Section 7.4). Note that the process under execution may be paused but the CPU does not save any context or other data, as the process is paused for one cycle and resumes immediately.

Cycle stealing slows the processor, but DMA is still more efficient for multiple-word transfers than programmed or interrupt driven I/O. In addition, the number of stolen cycles can be significantly reduced if the DMA is configured so that it does not need to use the system bus to communicate with I/O modules, but can interface with them directly. Stallings (2016, Section 7.5) notes that various I/O and DMA configurations to implement this are possible.

To summarise this section, there are three major ways of managing I/O modules: programmed I/O; interrupt-driven I/O; and direct memory access. Programmed I/O is inefficient, and is rarely used. Interrupt-driven I/O is widely used in computers. Direct memory access is also often used when blocks of data need to be transferred.

5.8.5 DMA and the cache

DMA enhances I/O transfer with peripherals and with network traffic, but does not scale. For a significant increase in I/O transfer above the rates achieved with DMA there is DCA – direct cache access. Since DCA is designed to improve network transfer, it is not covered by this course.

5.9 Memory management

Every process, when being processed by the CPU, must be in main memory. This is because the CPU can access the main memory much more quickly than it can access secondary memory. If a system can only run one process at a time, then the main memory is divided between the operating system and the currently running process. In a multi-programming system, main memory is divided between the operating system and 'live' processes, with the operating system deciding dynamically how to divide the memory among the processes. This is memory management, an important task of the operating system. In this section we shall describe a number of memory management techniques, and conclude with paging, the technique that Stallings states made possible 'truly effective multiprogramming systems', (Stallings, 2016, Section 8.3).

5.9.1 Swapping

In a multiprogramming system, the long-term scheduler chooses some processes from the job queue and puts them in main memory for execution by the CPU. In the case of no swapping, each process is either in the ready queue or in an I/O queue; there is no intermediate queue. If a process is completed, it is moved out of main memory. The short-term scheduler decides for the CPU which process to execute next. If a process, which is running, requires an I/O operation, the short-term scheduler simply allocates the CPU to another process in main memory. If this process then requests an I/O operation, then the CPU will move to yet another process in main memory. Because the CPU runs much faster than any I/O devices, it is common for **all** the processes in main memory to be waiting on I/O, leaving the CPU idle.

To keep the CPU busy, there must be new processes in main memory, which are ready to run. Stallings (2016, Section 8.3), notes that expanding main memory to include more processes is not the ideal solution, partly because main memory is expensive, and partly because larger memory results in developers writing larger programs.

However, main memory may be full already, so how can new processes come in? The solution is that some processes in main memory, which are waiting on I/O, can be temporarily moved out of main memory onto disk; this will leave some free space in main memory, so new processes can be loaded into the free space to be executed by the CPU. This method is called **swapping**.

Swapping is used when all the processes in main memory are waiting on I/O. The medium-term scheduler swaps some of the processes out of main memory onto disk, into the intermediate queue. The intermediate queue holds processes that have been temporarily removed out of main memory (swapped out). The operating system then loads new, ready processes, either by swapping a process from the intermediate queue back into main memory, or by choosing a process from the long-term queue. The CPU can then execute these newly added processes.

Swapping a process from main memory to disk takes quite a lot of time, because it is an I/O operation, and I/O operations are slow. However, Stalling notes that since disk I/O is usually the fastest on the system, simple 'swapping will usually enhance performance' (Stallings, 2016, Section 8.3). Even more improvement in performance is achieved with virtual memory, implemented with the help of swapping, and discussed later in this chapter.

5.9.2 Partitioning: fixed-size memory partitions

Recall that processes must be in main memory before the CPU can execute them. How should processes be arranged in main memory? The simplest method is fixed-sized partitions. The idea is to divide main memory into a number of chunks, or partitions, before processes are loaded. The sizes of the chunks are predetermined, fixed. However, the chunks need not be of equal size. When a process is loaded into main memory, it is placed in the smallest available chunk that will hold it.

For example, consider a memory of 32 MiBs. It could be divided into 4 equal fixed-size partitions of 8 MiB each. It could be divided into 8 equal fixed-size partitions of 4 MiB each. Or it could be divided into fixed, but unequal sized partitions; for example, 1 of 8 MiB, 1 of 6 Mib, 1 of 5 MiB, 1 of 4 MiB, 1 of 3 MiB and 3 each of 2 MiB.

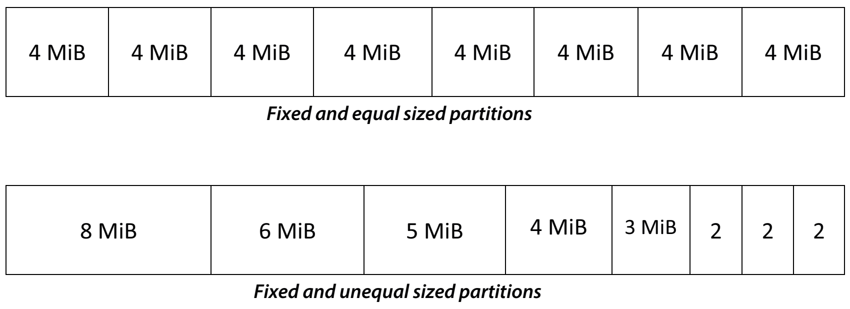


Figure 5.3: Fixed-size partitions of a 32 MiB memory.

However, most processes will not need exactly the amount of memory provided by the partitions. For example, if a process requires 1 MiB of memory, then in the figure above it will either be put in a partition of 4 MiB, or a partition of 2 MiB. Hence at least 1 MiB of memory will be unused, which is a significant amount of memory.

5.9.3 Partitioning: variable-sized memory partitions

A better way of partitioning the main memory is to use **variable-sized partitions**. In this approach, a process is allocated exactly the amount of memory it requires (see Figure 8.14 on p.320 in Stallings, 2016).

In the example shown in Figure 8.14 in Stallings, there are four processes. The size of Process 2 is smaller than that of Process 1, the same as Process 3, and greater than that of Process 4.

Initially there is only the operating system in main memory (see 8.14(a)). Then Processes 1, 2 and 3 are loaded, and each is given the exact amount of memory they require (see (8.14b, 8.14c and 8.14d)). In (e) Process 2 has been swapped out because it needs an I/O operation, and in (f) Process 4 has been brought in. In (g) Process 1 has completed and been removed from main memory, and in (h) Process 2 is swapped back in.

The above example shows that the approach of variable-sized partitioning is good initially, but it gets worse quite quickly because many small holes (i.e. unused memory) will be left in main memory. In time, fragmentation grows, hence memory utilisation shrinks. This will occur in the fixed-sized partitioning approach too. The operating system can address this with compaction – moving processes in main memory such that all free memory is in one block. However, while the algorithm is running the processor can be idle for many cycles, which is not ideal.

It will be clear that as processes are swapped in and out of main memory, their place in main memory changes. If the operating system uses compaction this will also move at least some of the processes in main memory. A logical address is an address relative to the start of the process; a physical address is an actual location in main memory. Processes themselves contain only logical addresses. When the processor is executing a process, it adds the starting point of the process in main memory – its base address – to each logical address.

5.9.4 Paging

With partitioning, whether fixed or variable sized, provided that space exists for the entire process, a process is loaded into main memory as a whole process in one contiguous block. With **paging,** a process is divided up, and the separate parts of the program need not be stored contiguously in main memory.

Paging has the following features:

- a process is divided into many small fixed-sized chunks **(pages)**
- main memory is divided into many small fixed-sized chunks **(frames)**
- a page is stored in one frame
- a process is stored in many frames, and when loaded into main memory the frames need not be contiguous; that is, if there is not enough contiguous space, the program can be loaded into whatever non-contiguous spaces are available
- the page size (and the frame size) is defined by the hardware.

When a program is loaded into main memory, it is stored in available frames. At most, the wasted space in memory for a process is a fraction of the last page. For example, suppose that a process divides into 100 pages: the first 99 pages are all full, only the last page is half-full. When this process is loaded into main memory, the first 99 pages will fully occupy 99 frames; only the last page will waste half of the frame allocated to it. Because frames are much smaller than partitions, much less memory is unused.

Figure 8.15 on p.321 in Stallings (2016) illustrates how paging works. Process A consists of four pages, numbered 0 to 3. Before A is loaded, some frames in main memory are being used by other processes marked as 'IN USE' in the diagram. So the four pages of Process A are loaded into frames 13, 14, 15 and 18. Note that these frames, which now hold Process A, are not contiguous in main memory.

The operating system must know which pages of a process are loaded in which frames, in order to find the pages in main memory for the CPU to execute. A base address (the starting address of the process) will no longer do since the process may not be stored in one contiguous block of memory. The operating system maintains a **page table** for each process. In Figure 5.4, the page table for Process A shows that its pages, pages 0, 1, 2 and 3, are ordered in the page table from the lowest numbered page to the highest. The first entry in the page table, frame 18, corresponds to page 0, the second entry, frame 13, to page 2 and so on. Hence the first entry in the page table now takes the role of the base address for the process.

18
13
14
15

Figure 5.4: Process A page table after memory allocation.

The operating system must know where each instruction of a process is. An instruction in a process has a **logical address**, which is its relative position in the program. In the case of paging, the logical address of an instruction is its relative position in the page containing the instruction, remembering that a page will contain more than one instruction. So given the logical address (page number, relative address within the page) the processor consults the page table to produce a physical address (frame number, relative address within the frame).

For example, in Figure 8.16 on p.322 in Stallings (2016), instruction 1 in Process A is at position 30 in page 1 (page 0 contains data), hence its logical address is: **1:30**. Consulting the page table shows that page 1 has been loaded into Frame 13. So the physical address of Instruction 1 in main memory is **13:30**.

5.9.5 Demand paging

The last section introduced the concept of paging, which may be called **simple paging**.

In simple paging, the operating system:

- loads all the pages of a process into memory in order to run the process
- makes more efficient use of memory than fixed-sized or variable-sized partitioning.

However, not all the pages of a process need to be in main memory in order to be executed. The operating system can still execute the process if only some of the pages are loaded, due to the **locality of reference principle**, which we have seen before when discussing cache memory. This states that over a short period of time the processor is likely to access a small subset of the program's instructions and data. To remind you, there are two parts to the principle: **temporal locality** and **spatial locality**. Temporal locality means that data and instructions that have been used recently are likely to be used again soon; and is exploited by keeping recently used data and instructions in main memory. Spatial locality means that data and instructions that are close together are likely to be used close together in time; and is exploited by fetching not just the current needed data and/or instruction, but nearby data/instructions too.

Thus the CPU only needs to access a small number of pages of a process in any short time period; so only those pages need to be loaded into main memory, particularly as, in a multi-programming system, the process will only run for a short time before being suspended. In the next short period, the CPU may demand another small set of pages of the process, so the next set of pages will be loaded.

Since in demand paging only a small number of pages of a process are in main memory at any time, at some point a page required by the CPU will not be in main memory (the program may branch to an instruction on a page not in main memory, or require data not loaded in main memory), triggering a page fault which tells the OS to bring in the needed page. The OS must remove a page to bring in a page, and uses a page replacement algorithm to do so. If the OS makes a mistake, it may end up fetching back the page that it has just removed, and doing this repeatedly triggers a process called **thrashing**, where the processor spends most of its time replacing pages rather than executing machine instructions. Stallings (2016, Section 8.3) notes that: 'The avoidance of thrashing was a major research area in the 1970s and led to a variety of complex but effective algorithms.'

Demand paging has the following advantages:

- Since not all pages of a process need to be loaded, more processes can be in main memory at any one time
- Loading a process from the hard disk is faster since only a part of the process needs to be loaded. This reduces the I/O overhead.
- There is no swapping. With swapping, entire processes are moved into and out of main memory. Bringing new pages into main memory from secondary storage has much less I/O overhead than swapping, since the pages brought in are normally only a small part of the process.

5.9.6 Virtual memory

Demand paging gives rise to the concept of **virtual memory**. As we have seen, with demand paging, only a small part of a process needs to be in main memory at one time. So, even if a process requires more memory than that given by the main memory, it can still be executed. Stallings (2016, Section 8.3) notes that with demand paging: 'One of the most fundamental restrictions in programming has been lifted', the restriction placed on programs by the size of main memory.

The programmer no longer needs to consider the size of the computer's main memory when developing new software. If a program can be stored on the machine it can be run with demand paging. This gives the programmer the impression that a computer's main memory is big enough to hold all programs; in other words that main memory is as big as secondary memory. This is called virtual memory. In virtual memory systems, main memory is referred to as 'real memory'. Words in main memory have 'real' or 'physical' addresses.

Activity 5.2

A computer system is designed to allow many processes to run at once, by restricting the number of pages a process can have in main memory. What potential problem do you see with this scheme?

5.9.7 Virtual addresses and the TLB

With virtual memory, processes are not swapped out of memory, since they are always in virtual memory; the CPU just needs to establish the physical address of an instruction from a virtual address. The CPU does this with the help of a cache of recently accessed page table entries, usually known as the translation lookaside buffer (TLB). The TLB exploits the locality of reference principle, and cuts down the need for two main memory accesses to load an instruction or data – one to look up the page table entry, and one to fetch the machine instruction.

The following algorithm shows how the CPU and the OS combine to turn virtual addresses into physical ones:

START: CPU issues virtual address

- CPU checks the TLB
- If the page table entry is in the TLB the CPU **generates the physical address**.
- If the page table entry is not in the TLB:
    - CPU accesses the page table
    - Is the page in main memory?
        - Yes, CPU updates the TLB and **generates the physical address**
        - No, start page fault handling routine (control passed to OS):
            - OS tells the CPU to read the page from the disk
            - If the main memory is full OS runs the page replacement algorithm
            - OS updates page tables
            - OS returns control to CPU
        - CPU updates the TLB and **generates the physical address**.

Note: once the physical address has been generated, the cache is consulted to see if the block with the word is in the cache.

There is one page table per process in memory; each page table can be very long; and, if stored in main memory, could take up most or all of its space. For this reason, page tables are stored in virtual memory, with only a portion of the page table stored in real memory at any one time. Hence, page tables themselves are subject to paging. You are not required to know the details of how page tables are organised in real and virtual memory, or the details of the processor's use of the TLB to generate a physical address from a virtual one. These details are given so that you can appreciate the complexity in generating a single memory address.

5.9.8 Memory protection and virtual memory

One of the most important jobs of the operating system is making sure processes do not interfere with each other; or memory protection. For example, one process should not be able to overwrite another process's data. With virtual memory, each process has its own page table, and can only access physical pages mapped to its own page table.

5.9.9 Virtual memory summary

- With virtual memory systems, main memory is known as real memory.
- Main memory is divided up into small, equal-sized frames.
- Every process is divided up into pages, a page is the size of a frame.
- One page is placed in one frame.
- For each process a page table is created, mapping virtual addresses to physical addresses (address translation).
- Address translation is faster with the use of a cache for page table entries, the translation lookaside buffer (TLB).
- At any one time a subset of pages are in real memory.
- At any one time only part of the page table for a process is in real memory.

5.9.10 Advantages of virtual memory

- Virtual memory increases the capacity of the operating system to run more processes, since at any one time only a few pages of any process are in main memory (that is, the entire process does not need to be in main memory in order for the process to run).
- Virtual memory increases the size of processes that can be run by the operating system; processes are no longer restricted to the size of main memory.
- Pages need not be contiguous, making good use of available memory, and overcoming problems caused by memory fragmentation.
- Because each process has its own page table, the integrity of the process's memory is guaranteed (memory protection).
- Virtual memory does not require entire processes to be swapped in and out of main memory, with heavy I/O overhead for each swap. While page table faults may require a page to be brought in from secondary memory with I/O overhead, a page is only a small part of a process, so the I/O overhead is much smaller.

5.10 Overview of chapter

This chapter defined 'operating system', considered the history of operating systems, and then discussed some of the major responsibilities of an operating system considered as a resource manager: process; I/O; and memory management. An operating system's network management responsibilities were not considered as networking is not part of this course.

The section on process management introduced the concept of a process, and a process control block. I/O management discussed how I/O is managed with the help of I/O modules, and the three techniques of I/O management that you are expected to be familiar with. Memory management looked at different memory schemes in turn, culminating with virtual memory. With virtual memory, secondary memory is addressed as if it were part of main memory, allowing for programs larger than main memory to be run. A key feature of modern operating systems, virtual memory is something that you are expected to have a good understanding of.

5.11 Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- describe the functions of the operating system and its major components
- explain where the operating system is usually stored, and how it gets started
- distinguish between multiprogramming and time-sharing operating systems
- explain the concept of process, and describe the process control block associated with each process
- explain the four different process queues: job queue, ready queue, intermediate queue, and I/O queues
- explain how the long-term, short-term and medium-term schedulers operate on processes and the process queues
- explain how the operating system manages memory by using techniques such as swapping, partitioning, paging, demand paging and virtual memory

- explain how the operating system manages I/O devices using techniques such as programmed I/O, interrupt-driven I/O and direct memory access
- explain the concept of interrupts, and describe how interrupts are caused, received and handled.

5.12 Test your knowledge and understanding

5.12.1 Sample examination question

This question is in three parts, a, b and c.

(a)

(i) An operating system is: [2 marks]

1. Application software
2. System software
3. A software protocol
4. None of the above

(ii) With paging, main memory is divided into sets of finite size called:[2 marks]

1. Frames
2. Pages
3. Chunks
4. Vectors

(iii) Which one of the following is the best definition of virtual memory? [2 marks]

1. An extremely large main memory
2. An extremely large secondary memory
3. An illusion of extremely large main memory
4. A type of memory used in super computers

(b)

(i) Explain why swapping a process from the main memory to the hard disk could slow the computer's performance, and why demand paging is preferred. [6 marks]

(ii) Swapped out processes are held in an intermediate queue. Describe the two other queues maintained by the operating system for managing the scheduling of processes; and explain the purpose of each queue and what it holds; and whether it is in main memory or the hard disk (or equivalent). [3 marks]

(c)

(i) Explain the concept of interrupt-driven I/O; and how it differs from programmed I/O. Describe a disadvantage that both I/O techniques have. [10 marks]

Notes

Chapter 6: Data representation

6.1 Introduction

In Chapter 3 we looked at how information (instructions and data) is stored in the main memory, magnetic memory and optical memory. In Chapter 4 we studied how the CPU executes instructions. We showed how the CPU could add numbers. However, numbers are only one type of data. Other types of data include: letters, characters, colours and sound. In this chapter we shall have a close look at how numbers, letters and characters are represented in the computer, because these types of data are the most important for computing. How other types of data are represented will be briefly discussed.

6.1.1 Aims of the chapter

This chapter aims to discuss in some detail the most widely used representation of integers, two's complement, and the universally used IEEE 754 floating point number representation; and to familiarise you with other number formats historically used in computers, such that you will be able to do the mathematical operations needed to convert between decimal and binary numbers and these representations. The chapter also aims – by familiarising you with the way that text is represented in a computer, and giving very brief details of the representation of colours, images, sound and movies – to underline that all data within a computer is represented to the processor as 0s and 1s.

6.1.2 Learning outcomes

By the end of this chapter and having completed the Essential reading and activities, you should be able to:

- explain how integers are represented in computers (e.g. using unsigned notation, signed magnitude notation, excess notation, and two's complement notation)
- explain how fractional numbers are represented in computers; for example, using fixed-point notation and floating-point notation (particularly the IEEE 754 single precision format)
- calculate the decimal value represented by a binary sequence in unsigned notation; signed magnitude notation; excess notation; two's complement notation; and the IEEE 754 single precision floating-point format
- when given a decimal number, determine the binary sequence that represents the number in unsigned notation; signed magnitude notation; excess notation; two's complement notation; and the IEEE 754 single precision floating-point format
- explain how characters are represented in computers; for example, using ASCII and Unicode
- explain how colours, images, sound and movies are represented in computers.

6.1.3 Essential reading

- Stallings, W. *Computer organization and architecture: designing for performance*. (Harlow: Pearson Education Ltd, 2016) 10th global edition [ISBN 9781292096858] Chapter 10 'Computer arithmetic'.

6.1.4 Further reading

- Harris, S.L. and D.M. Harris *Digital design and computer architecture*. (Waltham, MA: Elsevier/Morgan Kaufmann, 2016) ARM edition [ISBN 9780128000564] Section 5.3 'Number Systems' (includes floating-point numbers).
- Patterson, D.A. and J.L. Hennessy *Computer organization and design: the hardware/software interface*. (Amsterdam: Elsevier/Morgan Kaufmann, 2017) ARM edition [ISBN 9780128017333] Sections 3.5 'Floating Point' and 3.10 'Fallacies and Pitfalls'.
- Stallings, W. *Computer organization and architecture: designing for performance*. (Harlow: Pearson Education Ltd, 2016) 10th global edition [ISBN 9781292096858] Sections 9.1–9.5 from Chapter 9 'Number systems' (converting between decimal and binary, hexadecimal notation).

6.1.5 References cited

- Arnold, D.N. 'Some disasters attributable to bad numerical computing', https://cs.fit.edu/~ryan/library/Some_disasters_attributable_to_Numerical_Analysis.pdf

6.2 Key terms

Absolute value: in this chapter **absolute** is used to mean the size of a number, ignoring its sign. For example, 101 and –101 would be considered to be the same number, in absolute terms. Similarly, –6.2 and 6.2 have the same value, absolutely.

Overflow: computers have a limit on the size of numbers they can represent. This means that the result of a calculation could be a number that is too big to be expressed in the number of bits assigned to the representation. This is known as **overflow**.

6.3 Integer representations

Data in a computer is represented in terms of 1s and 0s. This is of course true of integers (whole numbers). However, there are several ways of representing integers, which are discussed in turn.

6.3.1 Unsigned representation

Unsigned representation can only represent positive integers; it is not possible to represent negative integers in unsigned representation. In unsigned representation an integer is considered as a sequence of bits, each bit being a 1 or a 0, and each making a contribution to the value represented. Figure 6.1 shows the unsigned representation of the integer 157:

Position	7	6	5	4	3	2	1	0	
Bit pattern	1	0	0	1	1	1	0	1	
Contribution	2^7			2^4	2^3	2^2		2^0	(=157)

Figure 6.1: Unsigned representation of an integer.

The binary representation of 157 is 10011101. The rightmost bit is said to be at position 0, and second from the right position 1, and so on. A 0 value at any position does not make any contribution to the overall value presented. A 1 at a position *i* makes a contribution of 2^i. So, the overall value represented by 10011101 is:

$$2^7 + 2^4 + 2^3 + 2^2 + 2^0 = 128 + 16 + 8 + 4 + 1 = 157$$

Addition is simple in this scheme: when adding two numbers in unsigned representation, we write the numbers down, one below the other in any order, so that bits with the same weight are in the same column, and then we add column by column, carrying a 1 to the left when necessary. Namely, we do just what we would do to add two decimal numbers, but we remember that in this instance the result of adding 1 + 1 is 10. This would mean placing a zero in the result row, and carrying over the 1 to the next column. Figure 6.2 shows how two 4-bit binary patterns can be added in unsigned representation:

Carried bit →	1

	1	**0**	**0**	**1**
+)	**0**	**1**	**0**	**1**
	1	**1**	**1**	**0**

Figure 6.2: Addition in unsigned representation.

In addition, when two 1s in any column are added 0 is written in the result row and a 1 is carried over to the column to the immediate left. Subtraction can also be done easily; for example, 1001 – 0101.

Borrow 1 from left ↓

	0~~1~~	**1**0	0	1
-)	0	1	0	1
	0	1	1	0

Figure 6.3: Subtraction in unsigned representation.

Note that in the above example, when the result of the subtraction is –1, we borrow a bit from the left, giving 10 – 1 = 1.

We could not subtract a larger number from a smaller; for example, 0001_2 – 1011_2 since the result would be a negative number, and negative numbers are undefined in this representation.

6.3.2 Signed magnitude representation

The unsigned representation cannot distinguish between positive and negative integers. To represent negative integers, we must use other representation schemes. One is **signed magnitude** representation.

In the signed magnitude representation (assuming that *n* bits are used), the leftmost bit (that is, the most significant bit) is used for representing the (positive or negative) sign of the integer, and this bit is called the **sign bit**. 0 indicates the integer is positive and 1 means that the integer is negative. The remaining *n*–1 bits represent the magnitude (namely, the absolute size) of the number.

Take the binary pattern 10011101 again, as an example of an 8-bit signed magnitude number. In signed magnitude representation it does not represent the number 157_{10}. Firstly, 10011101 is not a positive number in this representation, because the sign bit is 1, hence it is negative. The rest of the bits determine the absolute size of the number, their total value is calculated in the same way as for an unsigned integer.

Hence:

1	0011101
(determines sign)	(determines absolute size)

0011101 has the decimal value in absolute terms of:

$$2^4 + 2^3 + 2^2 + 2^0 = 16 + 8 + 4 + 1 = 29$$

So the number represented by 10011101 in signed magnitude is –29. This calculation can be summarised as follows:

Position	7	6	5	4	3	2	1	0	
Bit pattern	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	
Contribution	–			2^4	2^3	2^2		2^0	TOTAL = –29

Figure 6.4: Signed magnitude representation of an integer.

Consider a different 8-bit signed magnitude number: 00010010. The sign bit is 0, so the number represented is positive. The remaining bits are 0010010, which has the (unsigned) value:

$$2^4 + 2^1 = 16 + 2 = 18$$

So 0001 0010 represents +18.

Activity 6.1

What is the representable range in 8-bit signed magnitude representation? Give your answer in base 10.

6.3.3 Problems with signed magnitude representation

There are several drawbacks to signed magnitude representation:

- Addition and subtraction are difficult. This is because these operations require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation.

  For example, if we added 1101 (–5 in decimal) and 0001 (+1 in decimal) using binary addition as described in Figure 6.2, we would get 1110 (–6 in decimal) when clearly the result we expect is –4 (1100 in signed magnitude). To get the correct result, a different method will have to be used.
- There are two representations of 0. For example, if 8-bits are used, then we will have:

  00000000 = +0

  10000000 = –0

  To test whether a number is 0 or not, the CPU will need to see whether it is 00000000 or whether it is 10000000. Since testing for 0 is often performed in a program, having two representations for 0 is inconvenient.
- Detection of integer overflow takes a little more work than two's complement representation (see later on in this chapter).

  Because of these drawbacks, signed magnitude representation is rarely used in implementing the integer portion of the ALU inside the CPU. Instead, the most common scheme is two's complement representation, discussed later in this chapter.

6.3.4 Excess notation

Another way of representing both positive and negative integers is **excess notation**.

Let us consider excess notation with 4 bits. There are 16 possible (unsigned) binary sequences, from 0000 all the way up to 1111. These 4-bit binary patterns in the unsigned scheme represent positive numbers from 0 to 15. To be able to represent both positive and negative numbers using these binary bit patterns, we can draw a line in the middle of the bit patterns (see the shaded line in the table below). If we do so, then all the binary patterns above our imaginary line represent positive numbers and all those below it represent negative numbers. In this way, we can use half of the 4-bit binary patterns to represent positive numbers and the other half to represent negative numbers.

Decimal number in unsigned notation		Decimal number in excess notation
15	**1 1 1 1**	7
14	**1 1 1 0**	6
13	**1 1 0 1**	5
12	**1 1 0 0**	4
11	**1 0 1 1**	3
10	**1 0 1 0**	2
9	**1 0 0 1**	1
8	**1 0 0 0**	0
7	**0 1 1 1**	–1
6	**0 1 0 1**	–2
5	**0 1 0 1**	–3
4	**0 1 0 0**	–4
3	**0 0 1 1**	–5
2	**0 0 1 0**	–6
1	**0 0 0 1**	–7
0	**0 0 0 0**	–8

Figure 6.5: Excess notation with 4 bits.

6.3.5 Converting between unsigned and excess notation

As shown in Figure 6.5, for any binary pattern (of 4 bits) there is a number it represents in binary unsigned notation (the one on its left) and a number it represents in excess notation; the unsigned number is greater than the excess notation number, by 8, or 2^3. This is why the new scheme is called excess notation. From Figure 6.5 you can easily verify that the difference between the decimal values in each row is 8.

We can generalise from the case where 4 bits are used to the case where *n* bits are used. In the case of 5 bits, we will choose 16 to represent zero, hence numbers will be shifted by 16 or 2^4. In the case of 6-bit excess notation, 32, or 2^5, will represent zero. We can see that this pattern will continue, giving that for any *n*-bit binary pattern the following holds:

Decimal value in unsigned notation – 2^{n-1} = decimal value in excess notation.

When dealing with excess notation, it will always be helpful to keep an image like Figure 6.5 in mind: unsigned values are on the left, and excess notation values are on the right; and the difference between them is a constant 2^{n-1}, where *n* is the number of bits in the bit pattern used to represent each number. Once this is clear, it will be easy to deal with excess notation.

Suppose that we have a bit pattern 0100 0010 in excess notation. In unsigned notation this represents the number $2^6 + 2^1 = 56 + 2 = 58$. There are 8 bits in the number. Subtracting 2^{8-1} (= 128) from 58, we get –70, which the bit pattern 0100 0010 represents in excess notation.

Suppose we want to represent 24 in 8-bit excess notation. We want a bit pattern, call it *B*, which will represent 24 in 8-bit excess notation. The unsigned decimal value *B* represents should be 24 + 2^{8-1} = 24 + 128 = 152. If we determine the value of 152 in unsigned notation, which is 10011000, then we have our *B*. Therefore 24 in 8-bit excess notation = 10011000.

Excess notation has a number of advantages.

- It can represent both positive and negative integers.
- There is only one representation for 0. For instance, in the case of 4-bit words, 0 is represented as 1000 (see Figure 6.5).
- It is easy to compare two numbers to see whether they are equal or whether one is greater than the other. As far as comparison is concerned, the bits in excess notation can be treated as unsigned integers, because the relative magnitude of the numbers does not change.

Excess notation is not normally used to represent integers in itself. It is mainly used in floating-point representation (see later on in this chapter) for representing exponents.

Activity 6.2

1. Give a table of numbers like that in Figure 6.5, for 3-bit excess notation.
2. What decimal number does the bit pattern 0001 0100 represent in 8-bit excess notation?
3. What is –3 in 8-bit excess notation?

6.3.6 Problems with excess notation representation

- Zero is represented as 1 followed by *n*–1 0s, where *n* is the length of the bit pattern for the representation. This is not ideal.
- Addition works – simply add two numbers in the normal way – but takes longer than addition in two's complement.
- Subtraction does not work in a straightforward way.
- Detecting overflow is not straightforward.

6.3.7 One's complement notation

Some early computers used one's complement to represent integers. In this system, positive numbers are as they are in unsigned representation; and negative numbers are positive numbers inverted by changing 1s to 0s and 0s to 1. For example 7, in 4-bit one's complement would be 0111 (the same as for unsigned representation), inverting this gives 1000, which is –7 in one's complement.

Binary pattern				Value represented in one's complement notation
0	**1**	**1**	**1**	7
0	**1**	**1**	**0**	6
0	**1**	**0**	**1**	5
0	**1**	**0**	**0**	4
0	**0**	**1**	**1**	3
0	**0**	**1**	**0**	2
0	**0**	**0**	**1**	1
0	**0**	**0**	**0**	0 (positive zero)
1	**1**	**1**	**1**	0 (negative zero)
1	**1**	**1**	**0**	–1
1	**1**	**0**	**1**	–2

1	**1**	**0**	**0**	−3
1	**0**	**1**	**1**	−4
1	**0**	**1**	**0**	−5
1	**0**	**0**	**1**	−6
1	**0**	**0**	**0**	−7

Figure 6.6: One's complement notation with 4 bits.

Note that positive numbers begin with 0, and negative numbers with 1.

One's complement was used as it simplified subtraction, which could be done by inverting the bits of the number to be subtracted, and adding the two numbers. For example: 4 – 2.

$4_{10} = 0100_2$; $2_{10} = 00102$. Inverting the digits of 0010 gives 1101, leading to:

```
      ┌──────────────────────────┐
      ↑                          │
                                 │
Carried bits ->  1  1            │
                                 │
                0   1   0   0    │
           +)   1   1   0   1    │
           ────────────────────  │
Interim result: 0   0   0   1    │
Plus carry bit              1  ←─┘
           ────────────────────
Final result    0   0   1   0
```

Figure 6.7: One's complement addition.

In the above addition, the last carry bit is 'wrapped around' to the beginning and added to make the final value, 0010_2 or 2_{10}.

One's complement is simpler to implement than signed magnitude, but the two representations for zero are problematic. Addition and subtraction are easy to implement, although not as simple as in two's complement.

6.3.8 Two's complement notation

The representation for integers used by most computers today is **two's complement**.

Two's complement has the following advantages.

- Zero has one representation.
- Zero is represented as 0.
- The algorithms for addition and subtraction are fast and easy to implement.
- There is a simple way to detect integer overflow.

The idea is illustrated in Figure 6.8:

Binary pattern				Value represented in two's complement notation
0	**1**	**1**	**1**	7
0	**1**	**1**	**0**	6
0	**1**	**0**	**1**	5
0	**1**	**0**	**0**	4
0	**0**	**1**	**1**	3
0	**0**	**1**	**0**	2
0	**0**	**0**	**1**	1
0	**0**	**0**	**0**	0
1	**1**	**1**	**1**	−1
1	**1**	**1**	**0**	−2
1	**1**	**0**	**1**	−3

1	**1**	**0**	**0**	–4
1	**0**	**1**	**1**	–5
1	**0**	**1**	**0**	–6
1	**0**	**0**	**1**	–7
1	**0**	**0**	**0**	–8

Figure 6.8: Two's complement notation with 4 bits.

In this 4-bit example, the binary patterns from 1111 to 1000, which are usually placed above the patterns from 0111 to 0000, are now moved to below the latter, and are used to represent negative numbers. In this new scheme:

- all positive numbers begin with 0
- all negative numbers begin with 1
- 0 is represented as 0000
- all positive numbers have the bit pattern for unsigned integers, plus an extra bit for the sign. Since the sign bit is zero, positive numbers can be directly converted to decimal using place value, namely: $2^n + 2^{n-1} + \ldots + 2^1 + 2^0$.

There is an interesting, and also important, relationship between +*n* and –*n* (*n* being any integer). We can always turn a number *n* in two's complement notation to –*n* by:

- copying from the right all the bits until and including the first 1; and
- changing the remaining bits to the complementary bits (1 to 0, and 0 to 1).

For example:

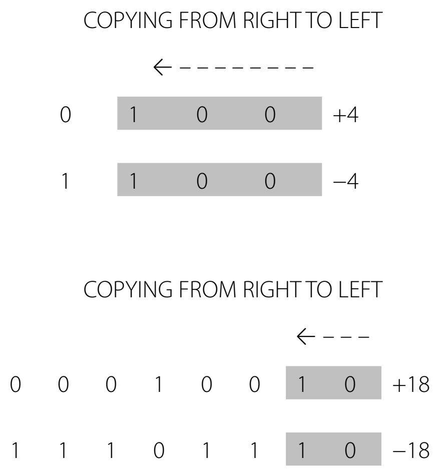


Figure 6.9: Two's complement conversion between positive and negative numbers.

In the above examples the shaded part of the representation of +4/–4 in excess notation stays the same, and only the left-most bit is changed. The shaded part of the representation of +18/–18 in excess notation stays the same, and the rest of the bits are changed to their complement; that is, 0 to 1 and vice versa.

Another way to express this process is to invert the bits, then add one. This process is more complicated; hence, not as fast as the above process. For example, to convert –4 to 4, invert 1100 to get 0011 then add 1:

```
Carried bits →   1    1

         0    0    1    1
  +)                    1
  ----------------------
         0    1    0    0
```

Figure 6.10: Two's complement conversion between +ve and –ve numbers by inverting and adding one.

Note that the process shown in Figure 6.10 above has one problem – the bit pattern 1 followed by zeros, does not convert properly. For example, 1000 in 4-bit represents –8. If we invert the bits we get 0111, adding 1 we have:

$$\begin{array}{r} 0111 \\ \underline{+\ 1} \\ \underline{1000} \end{array}$$

We get our original number back. This holds true for any *n*-bit two's complement. The problem, of course, is that the bit pattern 1 followed by all zeros is the smallest number in the representation, –8 in 4-bit, –128 in 8-bit, and so on. There is no positive representation for the smallest negative number, and so the conversion fails.

The nice thing about two's complement notation is that it is simple to add any two integers, no matter whether they are positive or negative. The method is the same as that for unsigned integers (see Figure 6.2). Figure 6.11 illustrates how addition is done in two's complement notation:

	0	0	0	1	(+1)		1	1	1	1	–1
+)	0	1	0	1	(+5)	+)	0	1	0	1	+5
	0	1	1	0	(+6)	1	0	1	0	0	+4

Result 0110 = 6_{10} Result 0100 = 4_{10}

Figure 6.11: Addition in two's complement notation.

Note that in the second example in Figure 6.11 above there should be a carry-bit, giving a 5-bit binary pattern. But the carry-bit is ignored, because only 4 bits are used to represent integers in this representation. Ignoring the carry-bit gives the correct result of 0100 or 4_{10}.

6.3.9 Integer overflow

Adding two large integers of the same sign can result in a number that is too large (in absolute terms) to be represented in the bits available. This is called overflow. Positive overflow means that the result is a positive number that is too large to be represented in the bits available. Negative overflow means a negative number that, in absolute terms, is too large for the bits available. For example, adding +6 and +7 in 4-bit two's complement:

6 + 7 = 0110 + 0111 =		0	1	1	0
	+)	0	1	1	1
		1	1	0	1

So the result of adding +6 (0110) and +7 (0111) is –3 (1101), which is clearly wrong. However, we can tell that an overflow has occurred in two's complement notation, simply by noting that the sign bit of the result is different from the sign bits of the operands.

For example, adding –3 to –8 in two's complement:

–3 + –8 = 1101 + 1000 =		1	1	0	1
	+)	1	0	0	0
	1	0	1	0	1

In the above addition the shaded digit is discarded. In general, if a two's complement addition results in an extra digit, the extra bit is discarded.

The result of –3 + –8 is 0101 (which is 5_{10}). We know that the result contains an overflow because the sign of the result is 0, and the sign bit of both operands was 1. In general, a change of sign indicates overflow since two positive numbers added together cannot make a negative number, and two negative numbers added together cannot make a positive number.

What happens if we add a positive number to a negative number? In this case, overflow is not possible. You can satisfy yourself of this by looking at Figure 6.8 and trying to find one positive and one negative number to add together such that the result is not in the representable range of numbers for 4-bit two's complement; that is, 7 to –8.

6.3.10 Integer overflow and Ariane 5

Overflow is an issue for developers that must be addressed in safety critical applications, and has in the past caused devastating failures. The European Space Agency spent billions to develop Ariane 5, an unmanned rocket that was intended to be Europe's workhorse for satellite launches. However, on the maiden flight in June 1996 the Ariane 5 exploded; the rocket together with its payload were completely destroyed. An investigation later determined that the cause of the explosion was integer overflow in Ariane 5's software, which caused a cascade of events resulting in the rocket's self-destruct mechanism being triggered.

See here for a list of disasters caused by faulty computer arithmetic, including Ariane 5: http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html

Note that since the 1996 disaster, several variants of the Ariane 5 rocket have been produced, and many successful flights undertaken.

6.3.11 Subtraction in two's complement

Subtraction can also be easily performed in two's complement notation. To subtract *b* from *a*, we only need to negate *b* (using the procedure described in Figure 6.9), and then add the result (–*b*) to *a*.

Multiplication can be performed by repeated addition. Division can be performed using repeated subtraction. There are efficient algorithms for doing multiplication and division, which you are expected to be aware of, but will not be asked to apply (see Stallings, 2016, Chapter 10).

Because of the ease of doing arithmetic operations, two's complement notation is, as Stallings (Section 10.2) notes, 'the most common scheme' for 'implementing the integer portion of the ALU'.

6.3.12 Evaluating numbers in two's complement notation

Given any binary sequence in two's complement notation, we can determine what decimal value it represents:

- If the sign bit is 0, then the number is positive. The value can be determined in the usual way using place value; namely, $2^n + 2^{n-1} + \ldots + 2^1 + 2^0$
- If the sign bit is 1, then the number is negative. Then there are three methods of determining the value (assuming that n bits are used):

  Method 1: Determine the value of the remaining n–1 bits. Then subtract 2^{n-1} from it.

  Method 2: Treat the sign bit as making a contribution of -2^{n-1}, and then add up all the bits' contributions.

  Method 3: Convert the number to its positive counterpart (by applying the procedure shown in Figure 6.9), and then determine the latter's decimal value (see the first bullet point above). If the decimal value of the converted number is *V* then the decimal value of the original two's complement negative number is –*V*.

Example: Determine the value represented by 10101 in two's complement notation.

Method 1

The sign bit is 1, so the number is negative. The unsigned value of the remaining bits is $0101_2 = 5_{10}$. Subtracting 2^{5-1} (= 2^4 = 16) gives 5 – 16 = –11. So the value represented is –11.

Method 2

Place value	4	3	2	1	0
Bit pattern	**1**	**0**	**1**	**0**	**1**
Contribution	-2^4	+0	$+2^2$	+0	$+2^0$

Result $-2^4 + 2^2 + 2^0 = -16 + 4 + 1 = -11$

Note that in this method the left-most bit is regarded as making a negative contribution.

Method 3

The positive number corresponding to 10101 is 01011.

The value of 01011 is $2^3 + 2^1 + 2^0 = 8+ 2+ 1 = 11$.

So the value represented by 10101 is –11.

6.3.13 Sign extension

It may sometimes be necessary to convert a number represented in a fixed number of bits, to representation with more bits. For example, converting the 4-bit unsigned number 1111 to 6 bits just means adding some leading zeros to get 001111. Converting the signed magnitude number 1001 from four bits to six bits, means moving the sign bit to the most significant position in the 6-bit number; namely, 1 _ _ 001. Then the gaps are filled with zeros, giving 100001. This does not work to convert two's complement numbers from a lower range to a higher one. Instead the rule is to move the sign bit to the left-most position, then fill in the gaps with copies of the sign bit. For example, converting the 4-bit two's complement numbers 0111 and 1011 we would have:

$$0\,\_\,\_\,111 \rightarrow 000111$$

$$1\,\_\,\_\,011 \rightarrow 111011$$

For more information on this conversion, called 'sign extension', see Stallings (2016), Section 10.2: 'Integer representation' to Section 10.3 'Integer arithmetic'.

Activity 6.3

The decimal number A = –63 and the decimal number B = –109. Answer the following questions showing all of your working.

1. Give the 8-bit two's complement representations of A and B.
2. Compute A – B using 8-bit two's complement arithmetic.
3. Compute A + B using 8-bit two's complement addition. Does the result contain an overflow? Explain your answer
4. Compute A + B in 16-bit two's complement arithmetic. Does the result contain an overflow? Explain your answer.

6.4 Fraction representations

In the last section we saw some ways of representing integers. However, those representation schemes cannot handle fractional numbers, such as: 0.156, 38604.78, ¾, and so on. In order to represent fractional numbers, other representation schemes are required. The next section looks at the fixed-point representation; the section after that discusses the floating-point representation; and in the last section we will concentrate on a particular kind of floating-point representation, called the IEEE 754 floating-point standard, which is widely used.

6.4.1 Fixed-point representation

In a fractional number, such as 12.345, the dot is called the radix point. The radix point separates the integral part of the number (to the left of the radix) from its fractional part (to the right of the radix). In a binary number, the point separating the integer and fractional parts is called the 'binary' point. In fixed-point representation, the binary point is fixed to a particular location in the binary sequence, so that the bits to the left of the binary point represent an integer, and the bits to the right represent a fraction. Figure 6.12 illustrates this idea.

Unsigned								
Place value	7	6	5	4	3	2	1	0
Actual place value	4	3	2	1	0	–1	–2	–3
Bit pattern	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**
Contribution	2^4			2^1	2^0	2^{-1}		2^{-3}

Figure 6.12: Fixed-point representation of 19.625.

In this example, 8 bits are used to represent fractional numbers. The binary point is assumed to be between the third and fourth bits from the right. The bits 10011 represent the integer, which is 16 + 2 + 1 = 19. The bits 101 represent the fraction, which is $2^{-1} + 2^{-3} = 0.625$. So 10011.101 in this scheme represents the fractional number 19.625. In the example shown in Figure 6.12, 5 bits are used for representing integers, and 3 bits for representing fractions; and this limits the range of both integers and fractions.

The representation of very large numbers and very small fractions may involve some redundancy. This is because we have to give bits to both the integer part and the fraction part. For example the decimal number 0.125 can be represented exactly in the above scheme as 00000.001. The problem is that all of the information in the number is in the final place on the right, but all the places that are zero have to be represented too. What about the number 0.0625_{10}? This would be 00000.0001_2, and hence could not be represented in this notation, since there are only 3 bits for representing fractions, and to represent 0.0625 we need 4.

6.4.2 Floating-point representation

In the decimal system we can get round this problem using scientific notation. For example, 1,245,000,000,000 can be represented as 1.245×10^{12}, and 0.000,000,000,76 can be represented as 7.6×10^{-11}. To represent the first number, we only need to use 6 digits: 4 digits for 1.245, and 2 digits for 12. The representation of the second number only requires 5 digits: 2 digits for 7.6 and 3 digits for –11. This shows that with scientific notation we can represent a large range of numbers using just a few digits.

Binary numbers can be dealt with in the same way. We can represent any number in the form:

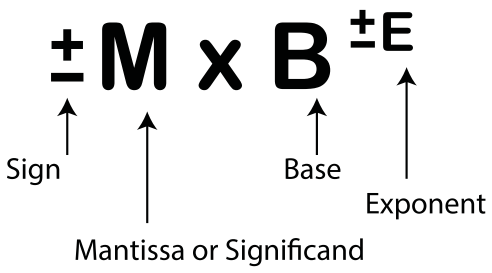


Figure 6.13: Floating-point representation.

In the above, **mantissa** means that part of the number which represents the significant digits of the number.

For example, 5.75 = 101.11, and we can represent the number 5.75 using any of the binary sequences below:

$$0.10111 \times 2^{3}$$
$$1.0111 \times 2^{2}$$
$$10.111 \times 2^{1}$$
$$101.11 \times 2^{0}$$
$$1011.1 \times 2^{-1}$$
$$10111.0 \times 2^{-2}$$
$$101110.0 \times 2^{-3}$$
$$1011100.0 \times 2^{-4}$$

If we pay attention to the binary point and the exponent in all the above binary sequences, we will see that the binary point can float to the left or to the right. This is where the term floating-point representation comes from. If the binary point floats to the left, the exponent increases and if it floats to the right, then the exponent decreases.

To simplify operations on floating point numbers, it is typically required that they be normalised into the form:

$$\text{(N1):} \quad \pm 0.1bbb...b \times 2^{\pm E}$$

So, the normalised floating-point representation for 5.75 is $+ 0.10111 \times 2^{3}$. Similarly, the normalised floating-point representation for -0.3125 is -0.101×2^{-1}.

If we use 16 bits to represent numbers, then we can, for example, use:

- 1 bit for the sign of the number
- 9 bits for the mantissa
- 1 bit for the sign of the exponent
- 5 bits for the mantissa of the exponent.

Then the above two fractional numbers (5.75 and –0.3125) can be stored as follows:

Sign (1bit)	Mantissa (9 bits)	Sign of Exponent (1 bit)	Mantissa of Exponent (5 bits)
0	101110000	0	00011
1	101000000	1	00001

Figure 6.14: Table showing possible 16-bit floating point representation.

6.4.3 IEEE standard 754

In any floating point number scheme there is a trade-off between range and accuracy. Since the word size is fixed, if an extra bit is given to the fractional part, then a bit must be taken away from the exponent. The larger the exponent, the greater the range of numbers that can be expressed. The larger the fractional part, the greater the accuracy of the number, since it will be less likely that rounding will have to be applied.

The most important floating-point representation is defined in IEEE standard 754. It was developed in 1985 to facilitate the portability of programs from one processor to another, and revised in 2008. The standard has been widely adopted, and is used on virtually all contemporary CPUs. IEEE 754-2008 specifies both binary and decimal floating point representations; however, we shall only be concerned with binary representations.

The IEEE standard defines both a 32-bit single and a 64-bit double format. We shall only discuss the former here. The 32-bit single precision format has the following allocation of bits:

- 1 bit for the sign
- 8 bits for the biased exponent
- 23 bits for the normalised mantissa.

Figure 6.15 shows the layout of this format:

1 bit	8 bits	23 bits
Sign	Exponent	Mantissa

Figure 6.15: IEEE single precision format.

6.4.4 IEEE 754: The biased exponent

The exponent here is biased, and this requires some explanation. A biased notation is just an excess notation. It differs from the standard excess notation in that the excess is ($2^{n-1} - 1$), rather than 2^{n-1}, where n is the number of bits used. Recall from Section 6.3.5 that the same binary pattern represents one value in unsigned notation, and a different value in excess notation; the difference between them is 2^{n-1}, where *n* is the number of bits used in the representation. If *n* = 8 (that is, we are considering 8-bit excess notation), then the difference in excess notation is $2^7 = 128$. But in the biased notation used in IEEE single format, the difference is 127, since the difference is $2^7 - 1$. In essence, in 8-bit biased notation, we have:

Decimal value unbiased									Decimal value biased
255	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	128
:	**:**	**:**	**:**	**:**	**:**	**:**	**:**	**:**	:
:	**:**	**:**	**:**	**:**	**:**	**:**	**:**	**:**	:
:	**:**	**:**	**:**	**:**	**:**	**:**	**:**	**:**	:
129	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	2
128	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	1
127	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0
126	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	–1
125	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	–2
:	**:**	**:**	**:**	**:**	**:**	**:**	**:**	**:**	:
:	**:**	**:**	**:**	**:**	**:**	**:**	**:**	**:**	:
:	**:**	**:**	**:**	**:**	**:**	**:**	**:**	**:**	:
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	–127

Figure 6.16: Biased notation (biased by 127).

As Figure 6.16 shows, in the biased notation, numbers ranging from –127 to +128 can be represented using 8 bits. This means that there is no need to have a sign bit for the exponent, as we had in section 6.4.2, where the exponent was unbiased.

In the standard 8-bit excess notation, the numbers that can be represented range from –128 to +127. In a biased 8-bit notation, the numbers that can be represented range from –127 to +128. It is desirable to be able to represent more positive numbers. So in IEEE 754, exponents are biased by 127.

Having made clear what a biased notation is, we now try to see why we choose this representation in order to have both positive and negative exponents. Unsigned notation cannot represent negative numbers. Signed magnitude notation is no good for addition or subtraction. We are left with two choices: excess notation, or two's complement notation. Both can represent positive and negative numbers. However, in calculating with two floating-point numbers a frequent operation is to compare the two exponents. Excess notation makes it easy to compare two numbers whereas the operation would be difficult in two's complement notation. So excess notation is chosen for representing exponents.

6.4.5 IEEE 754: Normalised numbers and the implicit leading '1'

We now come to the 23-bit normalised mantissa. Recall that the normalised format is (repeated here for convenience):

(N1): ±**0.1**bbb...b x $2^{\pm E}$

With this format, the 23 bits will store '1bbb...b' (22 bits after the 1). But we can move the binary point one position to the right, to arrive at the following format:

(N2): ±**1**.bbb...b x $2^{\pm E}$

If we do this, we will use the 23 bits to store 'bbb...b' (23 bits after the 1), and leave the '1' implicit (because we can assume that the left-most bit, the real mantissa, is always 1). The effect of this manoeuvre is that we will be able to use 23 bits to store a 24-bit mantissa, and thus will be able to achieve greater precision. This is what happens in IEEE 754 single precision format.

For example, suppose that we have a number: 0.**1**111 0000 0000 1001 0100 1111 x 2^8.

Using the format (N1), only the first 23 bits after the binary point can be stored in the mantissa field. This effectively means that we can only store 0.1111 0000 0000 1001 0100 111 x 2^8, which means losing one bit of precision. However, if the format (N2) is used, then the stored 23 bits will be 111 0000 0000 1001 0100 1111. Since the '1' in bold is implicitly there, this effectively means that all the 24 bits in 0.1111 0000 0000 1001 0100 1111 x 2^8 will be stored. This will give one extra bit of precision.

To summarise, in IEEE 754 floating-point standard, the mantissa is normalised. This means:

- The binary point is shifted one point further to the right.
- There should be a 1 to the left of the binary point, but this 1 is not stored.

A couple of examples will help us to understand the IEEE 754 floating-point standard.

Example 6.1

Represent 5.75 in IEEE 754 single precision format.

Answer

$$
\begin{aligned}
5.75_{10} &= 101.11_2 \\
&= 101.11 \times 2^0 \\
&= 1.0111 \times 2^2
\end{aligned}
$$

So, the exponent is 2. Biasing it by 127, we get 2 + 127 = 129 = 1000 0001 (that is, in the biased notation 10000001 represents the number 2). The mantissa is 1.0111, but the 1 to the left of the radix point is not stored. So in IEEE 754, 5.75_{10} is represented as:

0	1000 0001	0111 0000 0000 0000 0000 000

Example 6.2

Which number does the following bit pattern in IEEE 754 represent in base 10?

1	1000 0000	0100 0000 0000 0000 0000 000

Answer

- The sign bit is 1, so the number is negative.
- The exponent is 1000 0000 = 128. It is biased by 127, so the true exponent is 128–127 = 1
- The mantissa is 0100 000 0000 0000 0000 000. It is normalised, so the true mantissa is 1.01 = 1.25
- So the number represented in base 10 is: $-1.25 \times 2^1 = -2.5$.

Activity 6.4

1. Represent –1/64 in IEEE 754 32-bit format.
2. Give the decimal number represented by the following IEEE 754 32 bit number:

1	1000 0111	1101 0000 0000 0000 0000 000

6.4.6 Special values in IEEE 754

In the IEEE 754 standard the all zeros biased exponent, and the all 1s biased exponent, are reserved for use by special values. The standard has a representation for positive and negative zero, and for positive and negative infinity. These representations – together with some other special values called NaNs (Not a Number) that signify an error has happened – use the all 1s and all 0s exponent. The following special values are for the 32-bit format:

	Sign	Biased exponent	Mantissa
Positive zero (0)	0	0000 0000	All zero
Negative zero (-0)	1	0000 0000	All zero
Positive infinity (∞)	0	1111 1111	All zero
Negative infinity (-∞)	1	1111 1111	All zero
NaNs	0 or 1	1111 1111	Non-zero

Figure 6.17: Table of special values in IEEE 754 32-bit representation.

Note: there are various NaNs; the only thing that is fixed is the all 1s exponent; and that the fractional part, the mantissa, is non-zero.

6.4.7 Overflow and underflow

While integers can overflow, floating-point numbers can both overflow and underflow. Overflow means a number that is too large, absolutely, to be represented. Underflow means a number that is too small, absolutely, to be represented in the number scheme. In terms of floating point numbers:

- **overflow**: when the positive exponent is too large to fit into the exponent field
- **underflow**: when the negative exponent is too large to fit into the exponent field.

Recall that a negative exponent means that the exponent divides the mantissa, rather than multiplying it. The larger the negative exponent, the smaller the number.

The mantissa should not overflow in any sensible scheme as rounding will be applied. It is the exponent that determines the size of the number, and the exponent cannot be rounded, but must be represented exactly. The larger the positive exponent, the larger the number represented in absolute terms, while a negative exponent that is very large absolutely (namely, ignoring the sign) means a very small number. Recall that $2^{-2} = 0.25$; $2^{-4} = 0.125$; $2^{-8} = 0.00390625$; that is, as the exponent becomes larger absolutely, the number becomes smaller.

This is because the larger the exponent in absolute terms, the larger the divisor, and a larger divisor means a smaller quotient. Therefore underflow means a number that is too small, in absolute terms, to be represented in the number scheme, and this depends on the absolute size of the exponent.

- The smallest value the exponent can have is –126; underflow would happen with an exponent field smaller than this (for example, –127); or, alternatively, a negative exponent that was larger than 126 in absolute terms.
- Overflow in this representation would happen when the exponent was larger than 127.

Positive overflow: the exponent is a positive number greater than 127 and hence too large to fit into the exponent field, and the number represented is a positive number.

Negative overflow: the exponent is a positive number greater than 127 and hence too large to fit into the exponent field, and the number represented is a negative number.

Positive underflow: the exponent is a negative number smaller than –126 and hence too large in absolute terms to fit into the exponent field, and the number represented is a positive number.

Negative underflow: the exponent is a negative number smaller than –126 and hence too large in absolute terms to fit into the exponent field, and the number represented is a negative number.

Underflow and overflow can happen with floating-point arithmetic; it may be that the result of a calculation with two floating-point numbers is too large or too small. Positive and negative overflow may be returned by some systems as positive or negative infinity respectively, using the special values from Section 6.4.6 in the subject guide. Positive and negative underflow may be returned as positive and negative zero, respectively.

6.4.8 Range of IEEE 754 single precision

In this representation some bit patterns are reserved to represent special values. In particular, the all 0s biased exponent, and the all 1s biased exponent, are used for a range of special values. This means that for 32-bit numbers the exponent –127 (the all 0s biased exponent) and the exponent 128 (the all 1s biased exponent) cannot be used to represent normalised numbers, so the range of actual exponent values for normalised numbers is –126 to 127 inclusive. However, this range is extended with subnormal, or denormalised numbers.

When the result of an arithmetic operation produces underflow, the IEEE 754 standard specifies right shifting the mantissa and incrementing the exponent. This process continues until the exponent is in the representable range; namely, –126 for 32-bit numbers. This is called gradual underflow, and can prevent divide by zero errors.

The smallest number in the IEEE 754 32-bit format is the subnormal (or denormalised number):

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001 \times 2^{-126} \text{ or } 1.0 \times 2^{-149}$$

For example, if the result of a calculation was 1.11×2^{-149} then the FPU will right shift the binary point in the mantissa and add one to the exponent repeatedly as follows:

$$0.111 \times 2^{-148}$$

$$0.0111 \times 2^{-147}$$

$$0.00111 \times 2^{-146}$$

.

.

.

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 111 \times 2^{-128}$$

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 011 \times 2^{-127}$$

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001 \times 2^{-126}$$

In the above example, the two least significant bits are lost with the final two shifts of the mantissa, meaning some loss of precision, but this is preferable to underflow.

This process is called 'subnormalising'; the term denormalised numbers is also used, since subnormal numbers have lost the implicit leading 1 that the mantissa of a normalised number has.

6.5 Character representations

In the preceding sections we saw how numbers are represented in the computer. The basic idea is that each number is represented using a sequence of 1s and 0s. Characters can also be represented in the binary system. In this section we will look at some schemes for representing characters in languages such as English, French and Chinese.

6.5.1 ASCII

Most computers use the ASCII (American Standard Code for Information Interchange) code to represent text. In this scheme each character is represented using a 7-bit binary code. There are two types of characters that are represented:

- **printable characters**: alphabetic, numerical and special characters that can be printed on paper or displayed on a screen (e.g. @)
- **control characters**: some of these have to do with controlling the printing and displaying of characters (e.g. end-of-line, tab, new-page); and others concern communication (e.g. beep).

Printable ASCII characters are given in Figure 6.18. Note that upper case and lower case letters differ by 32. The ASCII code was published in 1963, and was adopted for federal computers by President Lyndon B. Johnson in 1968. Presidential intervention was necessary because at that time each computer company had developed their own standard for representing characters, limiting the interoperability of machines.

ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	64	@	96	`
33	!	65	A	97	a
34	“	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39		71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	I
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	\	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w

ASCII value	Character	ASCII value	Character	ASCII value	Character
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	DEL

Figure 6.18: Table of printable ASCII characters.

6.5.2 Unicode

ASCII code is only really suitable for representing the English language alphabet and some printable and control characters. However, there are many different languages in the world, with symbols not included in the standard English alphabet.

The Unicode standard is a character-coding system designed to support the worldwide interchange, processing and display of written texts of the diverse languages and technical disciplines of the modern world. In addition, it supports classical and historical texts of many written languages. Unicode has done this by increasing the word size, allowing many more characters to be represented; it is backwards compatible with ASCII (the first 128 Unicode characters are the same as the ASCII characters, but with an extra leading zero in front of them), and has taken over the internet. Unicode defines several character encodings with UTF-8 being the most universally used on websites around the world.

6.6 Representing other types of information

Apart from numbers and characters, other types of information – such as colours, sound, images and movies – can also be represented in the binary system.

6.6.1 Colours

The RGB model is used to represent colours on any device that emits light, such as computer monitors, digital cameras, televisions, mobile phones, etc. The idea is to represent any colour by adding red, blue and green together in varying proportions. For this reason it is called an additive model. The implementation of the model is device dependent.

Colours in the RGB model can be represented using a number of bits per pixel; where a pixel is the smallest possible programmable colour area on a display screen. The number of bits per pixel is the bit depth, and the higher the bit depth, the greater the number of colours that can be represented. The table below shows some possible bit depths:

Bit depth	Number of colours
1 bit per pixel	2 (2^1)
8 bits per pixel	256 (2^8)
16 bits per pixel	65,536 (2^{16})
24 bits per pixel	16,777,216 (2^{24})

As you can see from the table, a bit depth of 24 gives more than 16 million possible colours. 24-bit colour is often referred to as true colour, as any real-life shade that can be detected with the naked eye must be among the 16 million.

In fact, the 24-bit depth represents more colours than the human eye can distinguish.

In addition to the RGB model, there is also the CMYK model, used for colour printing. In this scheme, dots of cyan, magenta and yellow are printed (sometimes black, although sometimes black is achieved by combining the other colours). The dots are too small to be seen individually by the human eye; they would correspond to pixels on a computer screen; and would be sent from the processor to the printer as words representing the colour of each dot.

6.6.2 Images

An image is composed of pixels, and each pixel has a certain colour. So an image can be represented using a list of words, each word representing the colour of a pixel.

The quality of a digital image depends on:

- the density of the pixels
- the size of the words representing the colours.

The density of pixels in an image is referred to as its resolution. The higher the resolution, the more information the image contains.

6.6.3 Sound

A sound wave is continuous (analogue) and periodic. The frequency of a sound wave, measured in Hertz (Hz), is the number of repeated wave cycles (periods) per second. Audible sound waves have frequencies ranging between 20Hz and 20,000 Hz (20 kHz).

Digital sound, such as on a CD, for example, is produced by sound **sampling** to digitise analogue sound waves. A sample is a measurement of the amplitude at a point in time. A sample is essentially a 'snapshot' of the instantaneous amplitude of a sound.

The quality of digital sound depends on:

- **The sampling rate**: the faster the sample, the better the quality.
- **The size of the word representing a sample**: the more bits used, the greater the resolution.

A sound file contains a large list of words, each word representing a sample; it also contains information about the sampling rate.

It is clear that the choice of sample frequency rate affects the quality of the final sound produced. Analogue signals will normally comprise sound at many different frequencies. If the frequencies are limited to a certain range, and the highest frequency is f_{max}, then the sampling rate must be at least $2 \times f_{max}$, or twice the highest analogue frequency component, in order to be able to reproduce the original signal with no loss of information. This theory is known as the Nyquist-Shannon theory.

6.6.4 Movies

A movie is essentially a sequence of images, called frames. The quality of a digital movie depends on the quality of the images and the sampling rate. The sampling frequency or sampling rate is a measure of the number of snapshots taken from the signal each second. The MPEG-2 standard is widely used by DVDs and digital television transmissions.

MPEG-2 encoding is in two parts.

- A frame is broken down into blocks of 8 pixels by 8 lines, and these blocks are described numerically.
- Since the standard compresses as well as encodes, some inter-frame prediction is used, to fill in missing frames. This is known as 'motion compensated inter-frame prediction'. MPEG-2 defines three picture types: Intra pictures (I-frames) are sampled pictures, predictive pictures (P-frames) are predicted from previous I- or P-pictures, and bidirectionally-predictive pictures (B-frames) are predicted from both previous and next I- or P-pictures.

These two systems are combined to produce the images seen by the viewer.

6.7 Overview of chapter

We examined representation schemes for numbers that have been used in computers historically. We examined in detail the two schemes that are in common use today: two's complement notation for representing integers, and IEEE floating-point representation for representing fractional, or real, numbers. After this we looked at character representation, introducing ASCII codes and the Unicode character-encoding standard. Finally, we looked at how other types of data, sound, colour and images, can be represented in a computer so that the CPU can process them.

6.8 Reminder of learning outcomes

Having completed this chapter, and the Essential reading and activities, you should be able to:

- explain how integers are represented in computers; for example, using unsigned notation; signed magnitude notation; excess notation; and two's complement notation
- explain how fractional numbers are represented in computers; for example, using fixed-point notation and floating-point notation (particularly the IEEE 754 single precision format)
- calculate the decimal value represented by a binary sequence in unsigned notation; signed magnitude notation; excess notation; two's complement notation; and the IEEE 754 single precision floating-point format
- when given a decimal number, determine the binary sequence that represents the number in unsigned notation; signed magnitude notation; excess notation; two's complement notation; and the IEEE 754 single precision floating-point format
- explain how characters are represented in computers; for example, using ASCII and Unicode
- explain how colours, images, sound and movies are represented in computers.

6.9 Test your knowledge and understanding

6.9.1 Sample examination question

This question is in three parts, a, b and c.

(a)

(i) Floating point representation is used to store: [2 marks]

1. Boolean values
2. Integers
3. Whole numbers
4. Real numbers

(ii) Which of the following bit patterns represents the value –6 in 8-bit two's complement notation? [2 marks]

1. 10000110
2. 11111001
3. 00000110
4. 11111010

(iii) Binary numbers can be used to represent: [2 marks]

1. Integers only
2. Fractions only
3. Both fractions and integers
4. None of the above

(b)

(i) Given that the decimal number A = 54 and the decimal number B = –77, give their 8-bit two's complement representation. [2 marks]

(ii) Compute A + B in 8-bit two's complement. [2 marks]

(iii) Compute A – B in 8-bit two's complement. Does the result contain an overflow? Explain your answer. [3 marks]

(iv) Compute A – B in 16-bit two's complement representation. Does the result contain an overflow? Explain your answer. [2 marks]

(c)

Assume we are using the 32-bit IEEE single precision floating-point format. The mantissa has 24 bits including the hidden bit. There is one sign bit and there are eight exponent bits.

(i) What decimal floating point number is represented by the following 32 bits? Show all of your working. [7 marks]

0100 0011 0111 0000 0000 0000 0000 0000

(ii) What special values do the following two numbers represent in the 32-bit IEEE single precision floating point-format? [3 marks]

0111 1111 1000 0000 0000 0000 0000 0000

1111 1111 1000 0000 0000 0000 0000 0000

Notes

Appendix 1: Answers to Sample examination questions

Chapter 2

(a)

(i) 1. Analytical engine.

(ii) 3. Colossus.

(iii) 1. Electronic Numerical Integrator And Computer.

(b)

(i)

1. A control unit
2. An Arithmetic and Logic Unit (ALU)
3. Memory
4. Input devices
5. Output devices.

(ii) **Increased**: memory
Decreased: size and price.

(c)

(i) Claude Shannon proposed the use of binary numbers in computer science.

(ii) The first generation of computers used vacuum tubes. Vacuum tubes used a lot of electricity and thus generated a lot of heat, which made them prone to failure. The larger the system, the more often vacuum tubes would fail. The second generation of computers replaced vacuum tubes with transistors, a small semi-conductor device that has two states: open and closed. Processing speed was increased because transistors are faster than vacuum tubes, and also because their reliability meant less delays due to hardware failure. Since transistors are also much smaller than vacuum tubes (perhaps 100th the size of a vacuum tube) more could be used, giving another increase in processing speed.

(iii) A transistor has two states: open and closed; and these two states correspond to 0 and 1 in the binary system, hence calculations in binary can easily be implemented with transistors.

- Hardware is easier to build, as binary requires two states rather than the 10 states required to implement decimal numbering.
- A computer that has to differentiate between two states rather than 10 is likely to be less error prone.

Chapter 3

(a)

(i) 2. Random-access memory.

(ii) 1. 4 bytes.

(iii) 1. It requires periodic refreshing.

(b)

(i) The number of RAM chips required is 1024/128 = 8.

(ii) 1024 = 2^{10}. Hence, 10 address lines are needed to access 1024 bytes of memory.

(iii) 8 = 2^3; therefore 3 bits are needed to select 8 chips.

(iv) 10–3 = 7; hence 7 lines are required to address each memory location on the chip.

Or, 128 = 2^7; therefore 7 lines are needed to address each memory location on the chips.

(c)

(i) The file occupies 6.5 tracks sequentially.

The time to read the first track is:

Average seek time: 6ms
+ Rotational latency
+ Time to read one track

Rotational latency = half the time for one disk rotation.

12,000/60 = 200; therefore there are 200 rotations each second.

1/200 = 0.005 – there is one rotation every 0.005 seconds; that is, one rotation per 5 ms.

This gives a rotational latency of 5ms/2 = 2.5ms.

The time to read one track is 5ms, the time for one rotation.

Time to read first track is (6 + 2.5 + 5)ms = 13.5ms.

The next 5.5 tracks can be read with no seek time, just rotational latency, followed by reading time for each track. For five tracks we have:

5 x **(2.5 + 5) = 5 x 7.5 = 37.5**.

For the final half a track we have a rotational latency of 2.5ms, and the time to read half a track is the time for half a rotation = 2.5ms, so the total time is **5ms**.

Total time to access and read = (13.5 + 37.5 + 5)ms = 56ms.

(ii) Since there are 500 sectors per track, and since there were 6.5 tracks when the file was stored sequentially, there are 6.5 x 500 sectors; that is: 3,250 sectors.

Since the sectors are distributed randomly, each sector must be accessed and read independently of the others. Therefore the time to access and read the entire file is 3,250 x (time to access and read one sector).

The time to access and read one sector is:

> Seek time = 6ms.
> + Rotational latency = 2.5 ms
> + 1/500 x 5ms = 0.01 ms ([1 sector divided by 500 per track] times the time taken for one rotation)

Total transfer time per sector = 8.51 ms.

Total transfer time for the file

= 3,250 x 8.51 ms
= **27,657.5 ms** (or 27.6575 seconds, or nearly half a minute)

Chapter 4

(a)

(i) 3. Binary code that specifies the operation to be performed on data.

(ii) 4. Program counter.

(iii) 1. Cache hits.

(b)

i. In **direct mapping**, the cache line is determined by taking the block number from main memory, modulo the number of lines in the cache. In this way each block in main memory will map to only one line of the cache. The advantages of direct mapping is that its simplicity makes it easy and cheap to implement; and searching is fast as only one cache line needs to be queried. The disadvantage of direct mapping is that every block can only go into one line. This means that if the processor should repeatedly request words from blocks that map to the same line, then the lines will be swapped often, known as 'thrashing' and which is clearly undesirable since it will slow the hit rate, and hence the processor's operations.

ii. In **associative mapping** any block of main memory can be mapped to any cache line. An advantage is that given that any block can go in any line, a block replacement algorithm designed to increase the hit rate can be implemented. The major disadvantage is that searching is slow, as each cache line must be queried in turn to determine if it contains the required word.

iii. **Set-associative mapping** is a compromise between the above two: the cache lines are grouped in sets, and then blocks of main memory are directly mapped to only one set of the cache, again using the modulo function. A block can be placed in any line of the set, but can only be placed in one set. Because a block can go into any line in the set, an algorithm designed to maximise cache hits can choose the line when bringing a block into a set that is full. It is easier to search for a word within the cache, as only a particular set must be checked, rather than every line. Conflicts are much less likely than in direct mapping. Hence the advantages of direct mapping and associative mapping are combined, and their disadvantages mitigated.

(c)

(i) Data hazards between first ***Sub*** and ***Add*** through register r_1.

Data hazards between ***Add*** and ***Addi*** through register r_1.

(ii) The execution takes 11 clock cycles to finish.

Clock cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Sub r_1; r_2; r_3	IF	ID	EX	MEM	WB							
Add r_1; r_4; r_1		IF	stall	stall	ID	EX	MEM	WB				
Addi r_5; r_1; 2			stall	stall	stall	stall	IF	ID	EX	MEM	WB	

(iii) First forwarding is from the EX stage of ***Sub*** instruction to the EX stage of the ***Add*** instruction.

The second forwarding is from the EX stage of ***Add*** instruction to the EX stage of the ***Addi*** instruction (3 marks).

The execution takes 7 clock cycles to finish (2 marks).

Clock cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Sub r_1; r_2; r_3	IF	ID	EX	MEM	WB					
Add r_1; r_4; r_1		IF	ID	EX	MEM	WB				
Addi r_5; r_1; 2			IF	ID	EX	MEM	WB			

Chapter 5

(a)

(i) 2. System software

(ii) 1. Frames

(iii) 3. An illusion of extremely large main memory

(b)

(i) To avoid the case where the main memory is full of processes not ready to be executed yet, swapping is used to swap some processes out of the main memory to free space for new processes that are ready to be executed. However, swapping processes back to the hard drive is slow. Demand paging is preferred because a process is divided up into small chunks called pages, and not all pages need to be in memory at any one time. This reduces the I/O overhead because loading a process from the hard disk is much faster since only a part of the process needs to be loaded. Demand paging also has the advantages that since not all pages of a process need to be loaded, more processes can be in memory at any one time.

(ii) The following scheduling queues are maintained by the operating system:

- a job queue (also called a long-term queue)
  As processes enter the computer system, they are put into a job queue, which consists of all jobs waiting to be processed by the system, and usually resides on the hard disk.
- a short-term queue (also called a ready queue).
  This queue holds all the processes which are in main memory and which are ready for execution. The short-term queue resides in main memory.

(c)

Interrupt-driven I/O: The CPU issues a read/write command to the device's I/O module then goes to do other things. When the I/O module is ready it uses an interrupt signal to tell the CPU it is ready to read/write from/to the

device. Unlike programmed I/O, the CPU does not waste time in waiting and checking if the state for the I/O module of the device is ready.
(4 marks)

Programmed I/O: The CPU issues a read/write command to the device's I/O module and sits idle. The CPU wastes time in checking if the state for the I/O module of the device is ready. During this time it does no other work.
(4 marks)

In both programmed and interrupt-driven I/O the CPU spends a lot of time transferring data as it is responsible for data transfer, and has to handle the transmission of each word of data. (2 marks)

Chapter 6

(a)

(i) 4. Real numbers

(ii) 4. 11111010

(iii) 3. Both fractions and integers

(b)

(i) A = 54 = 0011 0110 (1 mark)

B =–77 = 1011 0011 (1 mark)

(ii) A + B = 0011 0110 + 10110011 = 1110 1001 (=–23).

(iii) A – B= 0011 0110 – 10110011 = 0011 0110 + 01001101= 11000 0011

Change of sign in the answer indicates that a positive + a positive = a negative, therefore overflow.

(iv) A= 0000 0000 0011 0110 and –B = 0000 0000 0100 1101

A – B= 0000 0000 0011 0110 + 0000 0000 0100 1101

= 0000 0000 1000 0011; no overflow as no change of sign in the answer.

(c)

Assume we are using the 32-bit IEEE single precision floating point format. The mantissa has 24 bits including the hidden bit. There is one sign bit and there are eight exponent bits.

(i) 0100 0011 0111 0000 0000 0000 0000 0000

- positive number, since the sign-bit is zero (1 mark)
- biased exponent = $10000110_2 = 128 + 4 + 2 = 134_{10}$ (1 mark)
- the real exponent = 134 – 127 = 7 (1 mark)
- the normalised mantissa = 111 0000 0000 0000 0000 0000 (1 mark)
- the real mantissa = 1.111 (1 mark)
- the final value represented = $(1.111_2) \times 2^7 = 11110000_2$
  $= (2^7 + 2^6 + 2^5 + 2^4) = 128 + 64 + 32 + 16 = 240$ (2 marks)

(ii) Positive infinity : 0111 1111 1000 0000 0000 0000 0000 0000

Negative infinity : 1111 1111 1000 0000 0000 0000 0000 0000

Notes

Appendix 2: Answers to activities

Chapter 2

Activity 2.1

The difference engine is not considered to be the first general-purpose computer because it was not programmable, and was dedicated to a particular task, so was not general purpose.

Activity 2.2

The largest number would have all 1s, one hundred of them. Using the place value system the first one would represent 2^0, or 1, and the final 1 would represent 2^{99}. This is equivalent to $2^{100} - 1$.

Chapter 3

Activity 3.1

4. 1GiB = 2^{30} bytes. Hence the length of the address in bits is 30.
5. 1GiB = 2^{30} bytes. The length of a word is 32 bits, or 4 bytes. Therefore the number of words in memory is $2^{30}/2^4 = 2^{26}$, so there are 2^{26} addressable units, and 26 bits needed to address them.
6. The address space, the number of addresses needed, is smaller when the memory is word addressable, since it is 2^{26} which is smaller than 2^{30} which is the size of the address space when the memory is byte addressable.

Activity 3.2

1. 8K words = $2^3 \times 2^{10}$ words and each word of 32 bits = 2^5 bits. Hence: the total number of bits in memory is $2^3 \times 2^{10} \times 2^5 = 2^{18}$ = 131,072 bits.
2. $2^2 2^{10}$ words and each word is 32 bits = 4 bytes = 2^2 bytes. In total we have $2^2 \times 2^{10} \times 2^2 = 2^{14}$ bytes. Hence 14 bits are needed to address each byte in memory.

Activity 3.3

1. Main memory is 2^{21} bytes x 2^1 bytes (16 bits = 2 x 8 bits = 2 bytes) =2^{22} bytes.

   Each chip has 256 x 1024 bytes or $2^8 \times 2^{10} = 2^{18}$ bytes per chip.

   Main memory size in bytes divided by chip size in bytes = $2^{22}/2^{18} = 2^{22-18} = 2^4$

   Hence 16 chips are needed.
2. The processor will view a 16-bit word across two RAM chips, hence 2 are needed per word.
3. 16 8-bit chips with a 2 byte memory word, means a word is on 2 chips. The processor will view each pair of chips as one chip, so one address is needed per every 2 chips, hence 8 addresses are needed, $8 = 2^3$, so 3 bits of the address are for the chip.
4. The memory is 2MiB of 16 bit words, and is word addressable. Therefore every word needs an address. Total words = 2MiB = $2^1 \times 2^{20} = 2^{21}$ therefore 21 address bits are needed.

Activity 3.4

Since 1024 bytes = 512 bytes x 2 this means we need to read 2 sectors. Access is sequential, so we can read both sectors consecutively with no extra seek time or rotational delay; namely, we only have to find the place to start reading once.

The time taken is the average access time, plus the time to read. The average access time is the seek time plus the average rotational delay.

So the time taken is the seek time, plus the average rotational delay, plus the time taken to read 2 sectors.

We know that **seek time = 10ms**.

Average rotational delay is agreed to be the time for half a revolution.

12,000 rpm/60 seconds = 200 revolutions per second.

Time for 1 revolution in seconds = 1/200 = 0.005 = 5 ms; time for half a revolution = 2.5 ms

Average rotational delay = 2.5 ms.

Time to read = (number of sectors to read/total number of sectors per track) x time for one revolution

= (2/600) x 5ms = 0.0167 ms (given to 3 decimal places)

Time to read 2 sectors = 0.0167 ms (3 decimal places)

Seek time = 10ms

Rotational delay = 2.5 ms

Time to read (sequentially) = 0.0167 ms

TOTAL = 10

+ 2.5

+ 0.0167

= $\underline{\underline{12.5167}}$

So the total time to read is 12.52 ms (2 decimal places)

Activity 3.5

Magnetic tapes are accessed sequentially.

Chapter 4

Activity 4.1

Part 1: Since there are 32 (25) registers, 5 bits are required to address any particular one of them. There are 512KiB (i.e. 512 KiB of) words, or 512(2^9) x 1,024(2^{10}) words; that is, 19 bits are required to address any individual word. With 1 bit needed for the indirect bit, this means that the opcode accounts for the rest of the 64-bit word length; namely,

opcode bit size = 64 – (5 + 19 + 1) = 64 – 25 = 39.

Summary

Indirect bit: 1 bit

Register: 5 bits (2^5 = 32, because there are 32 registers)

Address: 19 bits (2^{19} = 512KiB, because = 512 x 1024 = 2^9 x 2^{10} = 2^{19})

Opcode = 64 – 5 – 19 – 1 = 39 bits

Part 2

Indirect bit (1)	Opcode (39 bits)	Register (5 bits)	Memory address (19 bits)

Total bits = 64 = (1 + 39 + 5 + 19)

Part 3

As the word is the 'container' for the data, there are 64 bits in the data inputs of the memory.

Since there are 512(2^9) x 1,024(2^{10}) words, there are 19 bits in the address inputs of the memory.

Activity 4.2

Locality of reference refers to spatial and temporal locality.

Spatial locality: instructions in close proximity to a recently executed instruction are likely to be called in the immediate future.

The cache exploits this principle by reading from main memory into the cache in blocks that contain the word needed, and the words surrounding it. Hence if the next word needed is the one that is close to the current word it will be ready in the cache. The spatial locality principle arises because usually program instructions are usually stored and carried out sequentially.

Temporal locality: items that were recently used (referenced) have a high probability of being re-referenced in the immediate future.

The cache exploits temporal locality by keeping, for a short time, recently accessed data and instructions. The temporal locality principle arises because of the time spent iterating through loops in program execution.

Activity 4.3

The physical address is 20 bits long and, since this a direct-mapped cache, these 20 bits comprise three distinct fields:

- a tag (referencing a particular main memory block)
- a cache line index (referencing a particular cache line)
- an offset (referencing a particular byte of the cache line).

Since the tag is given as 11 bits, this means that the cache line index and offset, together, account for the remaining 9 bits.

As a block holds 16 (2^4) bytes of data, and, because the data are, effectively, byte addressable, this means that 4 bits are required for the offset. Hence the remaining portion of the 9 bits – namely, 5 bits – are used for the cache line index. Since 5 bits permit the addressing of 25 items, there are 32 blocks of data in this cache.

Chapter 5

Activity 5.1

1. (1) **Scheduling**: Users signed up to use the machine, so scheduling was done externally to the system. Users signed up for a certain amount of time; if they did not use all their time the machine was idle until the time for the next user.

   Job setup time: the processes to go through to set up a job had to be undertaken individually by each user, each time that a program was to be compiled and run. Since the set up involved mechanical tasks such as mounting tapes, set up took some time.

2. A monitor was an early operating system used in batch processing computers to schedule jobs. A human operator would batch jobs together, and input them in sequence.
3. I/O instructions were privileged so that the monitor kept control of them. If the processor came to an I/O instruction, it returned control to the monitor.
4. Slow I/O processing meant that the processor on an early uniprogrammed computer was often idle, and processor time was very expensive. The development of operating systems established the principle that a computer could have two processes running; the OS and the program it was managing. Giving the OS more programs to manage was a natural next step.
5. Slow human reaction time. In a time-sharing system, the OS switches control between users in very short bursts of time. Given the speed of a human compared to a computer, most users will not notice any degradation in speed of response (assuming the system is well designed).

Activity 5.2

Thrashing. The fewer the number of pages a process has in main memory, the more likely a page fault is to occur. If the number of pages is restricted too much, then page faults will happen so often that the system thrashes; that is, it is spending so much time reading pages into main memory – and sending pages back to storage – that the processor's time to execute instructions is compromised.

Chapter 6

Activity 6.1

–127 to 127

Activity 6.2

(1)

Decimal number in unsigned notation				Decimal number in excess notation
7	**1**	**1**	**1**	3
6	**1**	**1**	**0**	2
5	**1**	**0**	**1**	1
4	**1**	**0**	**0**	0
3	**0**	**1**	**1**	–1
2	**0**	**1**	**0**	–2
1	**0**	**0**	**1**	–3
0	**0**	**0**	**0**	–4

(2) In unsigned value $0001\ 0100_2$ is 4 + 16 = 20. The excess in 8-bit is 2^7.

Therefore $0001\ 0100_2$ in 8-bit excess notation is 20 – 128 = –108

(3) The first step in converting –3 to 8-bit excess notation is adding the excess.

–3 + 128 = 125.

Converting 125 to unsigned binary gives the answer, which is 0111 1101.

Activity 6.3

The decimal number A = –63 and the decimal number B = –109. Answer the following questions showing all of your working.

1. Give the 8-bit two's complement representations of A and B.
2. Compute A – B using 8-bit two's complement arithmetic.
3. Compute A + B using 8-bit two's complement addition. Does the result contain an overflow? Explain your answer
4. Compute A + B in 16-bit two's complement arithmetic. Does the result contain an overflow? Explain your answer.

Question (1)

To convert a negative number to its two's complement representation, first we find its unsigned binary representation, which gives us its positive two's complement representation, then we convert the positive two's complement number to negative two's complement.

A simple way of converting to unsigned binary is repeated integer division by 2, until the result of the division is zero. With integer division the result of a/b is how many times b divides into a exactly, plus remainder. Dividing by 2 means the remainder can only be 0 or 1. The remainders of the division become the digits of the binary number, with the least significant bit found first.

Calculation	**Remainder**
63/2 =31	(LSB) 1
31/2 = 15	1
15/2 = 7	1
7/2 = 3	1
3/2 = 1	1
1/2 = 0	(MSB) 1

So 63 in unsigned binary = 111111, adding leading zeros in 8-bit unsigned binary we have 0011 1111 which gives the two's complement representation of 63. Convert to negative two's complement to get –63.

Convert by writing down from least significant bit until the first one, then flip the rest of the bits.

63= 0011 111**1**, –63= 1100 000**1**

A = 1100 0001

Similarly for B = –109, 109 in unsigned binary is

Calculation	**Remainder**
109/2 =54	(LSB) 1
54/2 = 27	0
27/2 = 13	1
13/2 = 6	1
6/2 = 3	0
3/2 = 1	1
1/2 = 0	(MSB) 1

So 109 in unsigned binary is 1101101, adding a leading zero, 109 in 8-bit two's complement is 0110 1101.

Convert by flipping all the bits then adding 1 gives:

```
  1001 0010
        + 1
-----------
  1001 0011
===========
```

B = 1001 0011

Answer to (1)

A = 1100 0001

B = 1001 0011

Question (2)

A – B = 1100 0001 – 1001 0011

To subtract, convert B to its positive representation, and add it to A giving:

Discard the extra bit, giving 0010 1110

```
   1100 0001
  +0110 1101
------------
 1 0010 1110
============
```

Answer to (2)

0010 1110

Question (3)

A + B =

```
   1100 0001
  +1001 0011
------------
 1 0101 0100
============
```

discard the extra bit, giving 0101 0100

0101 0100 is a positive number, therefore the addition has resulted in a change of sign, hence the result contains an overflow. We would expect this since –63 plus –109 = –172, which is outside the representable range for 8-bit two's complement, which is from –128 to +127.

Answer to (3)

The result of the addition is 0101 0100

This has an overflow because the addition of 2 negative numbers has given a positive number.

Question (4)

To compute A + B we sign extend A = 1100 0001, and sign extend B = 1001 0011 and then add the results.

```
     1111 1111 1100 0001
     1111 1111 1001 0011
------------------------
   1 1111 1111 0101 0100
========================
```

Discard the extra carried bit.

There has been no change of sign, so no overflow.

Converting 1111 1111 0101 0100 to decimal we get –172, which is the result we expect.

Answer to (4)

1111 1111 0101 0100

There has not been an overflow because the addition of 2 negative numbers has given another negative number.

Activity 6.4

1. –1/64 is a negative number, hence the sign bit is 1

   The exponent is–6 and the bias is 127, the real exponent is –6 + 127 = 121
   Hence, the exponent is $0111\ 1001_2$

   The mantissa is 1.0_2 therefore, the normalised mantissa is:

   0000 0000 0000 0000 0000 000

   Finally, the required representation is
   1 01111001 0000 0000 0000 0000 0000 000

2. The sign bit is 1, hence, the number is negative.

   The exponent is $1000111_2 = 135_{10}$

   The bias is 127, therefore the real exponent is 135–127 = 8

   The normalised mantissa (significand) is 1101 0000 0000 0000 0000 000

   Hence, the real mantissa is 1.1101_2

   Therefore the decimal number represented is $-1.1101_2 \times 2^8 = -111010000_2$

   $= -(256 + 128 + 64 + 16)_{10} = -464$

Notes

Appendix 3: Sample examination paper: Part A

Important note: This Sample examination paper reflects the examination and assessment arrangements for this course in the academic year 2018–19. The format and structure of the examination may have changed since the publication of this subject guide. You can find the most recent examination papers on the VLE where all changes to the format of the examination are posted.

Note that the examination covers both volumes of the guide, and is divided into two parts. You are expected to answer **two** questions from Part A covering the material in this Volume 1 subject guide; and **two** questions from Part B which covers the topics in Volume 2. From each section you are asked to choose **two** questions from a choice of **three**.

The following three questions are an example of **Part A** of the examination.

Question 1

(a) Consider the 8-bits binary sequence, 10101011. Find the decimal value it represents in each of the following:

(i) Signed notation [2 marks]

(ii) Excess notation [2 marks]

(iii) Two's complement notation [2 marks]

(b)

(i) Explain why it is easy to compare two numbers expressed in excess notation. [5 marks]

(ii) Why is it easy to perform arithmetic operations in two's complement notation? [4 marks]

(c) Assume we are using the 32-bit IEEE single precision floating-point format. The mantissa has 24 bits including the hidden bit. There is one sign bit and there are eight exponent bits.

(i) What decimal floating point number is represented by the following 32 bits? Show all of your working. [6 marks]

1 10000101 1101 0000 0000 0000 0000 000

(ii) Give the smallest positive normalised number in this representation. [2 marks]

(iii) Give the smallest positive subnormal (denormalised) number. [2 marks]

Question 2

(a)

(i) What is the role of the instruction register? [2 marks]

1. It stores data to write/read to/from the main memory
2. It stores the address of the memory location to be activated
3. It stores the instruction that has just been fetched from the main memory

(ii) Which one the following is part of the operating system [2 marks]

1. Floating point unit (FPU)
2. Memory management unit (MMU)
3. The kernel
4. None of the above

(iii) What is a PCB? [2 marks]

1. Process cancelling block
2. Process control block
3. Process calibrating block
4. None of the above

(b) Suppose a computer memory with 4,000 addressable memory locations is linked to an address decoder with 6 address lines.

(i) What problem might this computer have? [6 marks]

(ii) How could this problem be solved? [3 marks]

(c) An 8-bit processor has instructions that consist of a 3-bit opcode with a 5-bit operand, as described in the following table. The operand 'ddddd' stands for any sequence of 5-bits that can be interpreted as data. The operand 'aaaaa' stands for any sequence of 5-bits that can be interpreted as an address.

Opcode	Operand	Description
001	ddddd	Load the accumulator with the data ddddd
010	aaaaa	Write the content of the accumulator to the address aaaaa
011	aaaaa	Subtract from the accumulator the data at the address aaaaa
100	aaaaa	Add to the accumulator the data at the address aaaaa
101	aaaaa	Make the content of the address aaaaa 11111111
110	aaaaa	Make the content of the address aaaaa 00000000
111	aaaaa	Halt

Given the following program that starts at the address 00000, describe, step by step, what the program does. In your answer you should reference the program counter (PC), accumulator (AC) and the instruction register (IR). [10 marks]

Address	Instruction
00000	001 00000
00001	100 10011
00010	011 10010
00011	010 10100
00100	111 00000

Question 3

(a)

(i) The memory hierarchy: [2 marks]

1. Is a way of organising computer memory to give the impression that all memory is as fast as the processor
2. Is a way of organising computer memory such that moving down the hierarchy means memory gets cheaper, slower and bigger
3. Is a way of giving a computer the benefits of the fastest memory, without massively increasing the cost of the machine
4. All of the above

(ii) Which one the following is a technique for organising the cache? [2 marks]

1. Modulo mapping
2. Indirect mapping
3. Set associative mapping
4. None of the above

(iii) Which one of the following is true? [2 marks]

1. Forwarding is a technique to reduce pipeline stalls.
2. A longer pipeline always means faster execution time.
3. There are four types of pipeline hazard: Resources; Address; Data; and Control.
4. None of the above.

(b) There are three pipeline data hazards, known by their initials as RAW; WAR; and WAW.

(i) What do the initials RAW stand for? Briefly describe the hazard. [3 marks]

(ii) What do the initials WAR stand for? Briefly describe the hazard. [3 marks]

(iii) What do the initials WAW stand for? Briefly describe the hazard. [3 marks]

(c)

(i) Abstraction is a way of reducing complexity by modelling the essential features of a system, and hiding details of its implementation. Explain why I/O modules are a good example of abstraction. [6 marks]

(ii) The operating system manages I/O, but does not normally interface directly with I/O devices (peripherals); instead the operating system communicates via one, or more, I/O modules.
Give two reasons why the operating system interfaces with peripherals via I/O modules. [4 marks]

Notes

Sample examination answers: Part A

Question 1: Answers

(a) Consider the following 8-bits binary sequence, 10101011. Find the decimal value it represents in each of the following:

(i) Signed notation

10101011 in signed notation:

The first bit is 1 hence it is a negative number.

The magnitude of the remaining bits = 0101 011= 32 + 8 + 2 + 1 = 43.

Therefore the number represented is -43.

(ii) Excess notation

= unsigned – 2^7

= 128 + 32 + 8 + 2 + 1 – 128 = **43**.

(iii) Two's complement notation

The first bit is 1, hence it is a negative number.

Treating the sign bit as making a contribution to the final value of –27: 10101011 = – 128 + 32 + 8 + 2 + 1 = – 85.

Alternatively: the decimal value of its corresponding positive number is: 01010101 = 64 + 16 + 4 + 1 = 85. Hence, 10101011 = – 85.

(b)

(i) Excess notation is a representation of a number with an excess constant value that depends on the number of binary bits used. Since for n-bit numbers the same excess value is subtracted from each number [1], numbers can be considered to be shifted along the number line by a fixed amount, while maintaining their values relative to each other [2]. Hence it is enough to compare their unsigned values to determine whether 2 numbers are equal, or whether one is greater (or lesser) than the other [2].

(ii) In two's complement notation it is easy to negate a number by switching all 1s to and 0s and 0s to ones and adding 1 [1]. Hence subtraction can easily be implemented by using addition, as can division and multiplication [1]. In addition there is only one representation of zero, and zero is represented by zero [1]. This makes arithmetic operations easy to implement using two's complement [1].

(c)

(i) The sign bit is 1, hence, a negative number. [1]

The exponent is $1000\ 0101_2 = 133_{10}$. [1]

The bias is 127, therefore the real exponent is 133 – 127 = 6. [1]

The normalised mantissa (significand) is 1101 0000 0000 0000 0000 000. [1]

Hence, the real mantissa is 1.1101_2. [1]

Therefore the decimal number represented is:

$- 1.1101_2 \times 2_6$

$= (-)1110100_2$

$= - (64 + 32 + 16 + 4)_{10} =$ **–116**. [1]

(ii) The minimum positive normalised value is 2^{-126}.

(iii) The minimum positive subnormal value is 2^{-149}.

Question 2: Answers

(a)

(i) 3. It stores the instruction that has just been fetched from the main memory

(ii) 3. the kernel

(iii) 2. process control block

(b)

(i) With 6 address lines, the maximum addresses that can be generated – the address space – is $2^6 = 64$. [2]

This is not enough to give each addressable memory location a unique address. [2] Hence, only 64 memory units can be used and the remaining memory will be wasted. [2]

(ii) With 11 address lines, the maximum number of addresses available is $2^{11} = 2048$, which is not enough to use all 4000 memory locations. 12 address lines will generate $2^{12} = 4096$ addresses [2], which is enough to give each addressable memory location a unique address. [1]

(c)

Initially, the program counter (PC) contains the address 00000. The instruction at this address, which is 001 00000, is fetched and put into the instruction register (IR). This instruction is then executed. As a result, the data 00000 is loaded into the accumulator (AC). [2]

Now the PC is incremented and its content is 00001. The instruction at this address, which is 100 10011, is fetched and stored in the IR. This instruction is then executed. The data at the address 10011 is now added to the accumulator. [2]

The content of the PC becomes 00010. The instruction at this address, which is 011 10010 is fetched and put into the IR. This instruction is then executed. As a result, the content at address 10010, is subtracted from the AC. [2]

Now the PC becomes 00011. The instruction at this address, which is 010 10100 is fetched, and put into the IR. [1] This instruction is then executed. As a result, the content of the AC is loaded into the location 10100. [2]

Now the PC becomes 00100. [1] The instruction at this address, which is 111 00000, is fetched and put into the IR. This instruction is then executed. As a result, the program halts. [2]

Question 3: Answers

(a)

(i) 4. All of the above

(ii) 3. Set associative mapping.

(iii) 1. Forwarding is a technique to reduce pipeline stalls

(b)

(i) **Read after write (RAW) [1]**: In the case where an instruction modifies the contents of a register or memory location, and the next instruction reads those contents, a possible data hazard is that the second instruction reads before the first has written. [2]

(ii) **Write after read (WAR) [1]**: In the case where an instruction reads a register or memory location and the next instruction writes to that location, a possible data hazard is that the second instruction writes before the first has read. [2]

(iii) **Write after write (WAW) [1]**: In the case where an instruction writes to a register or memory location and the next writes to the same register or memory location, a possible data hazard is that the second instruction writes, and then the first instruction overwrites the data written by the first. [2]

(c)

(i) I/O modules are interfaces between the CPU and main memory on the one side, and I/O devices (or peripherals) on the other. Each I/O module controls one or more peripherals. A computer may have any number of peripherals connected to it. Each peripheral may operate at a different speed to the processor (normally much slower) [1], use different data formats [1] and be based on varying hardware principles [1]. In addition each peripheral will have its own device-dependant control logic [1]. Because I/O modules hide details of these things the processor can communicate with any number and type of peripherals via a few simple commands [1]. For example, the processor does not need to have a different read command for each peripheral device, even though each peripheral may store and read data in different ways. [1]

(ii) Full credit for any two of the following:

- **Control logic**: Each I/O device has its own device-specific control logic. Without I/O modules, the necessary logic for control of every I/O device the processor may encounter, would have to be incorporated into the processor.
- **Data formats**: I/O devices may also use different word lengths than the CPU and main memory. I/O modules hide these details from the processor.
- **Data buffering**: The CPU can normally send data at a much faster rate than a peripheral can receive it. There are a few peripherals that can transfer data faster than the CPU or main memory. In either case, the I/O module buffers the data.

Notes