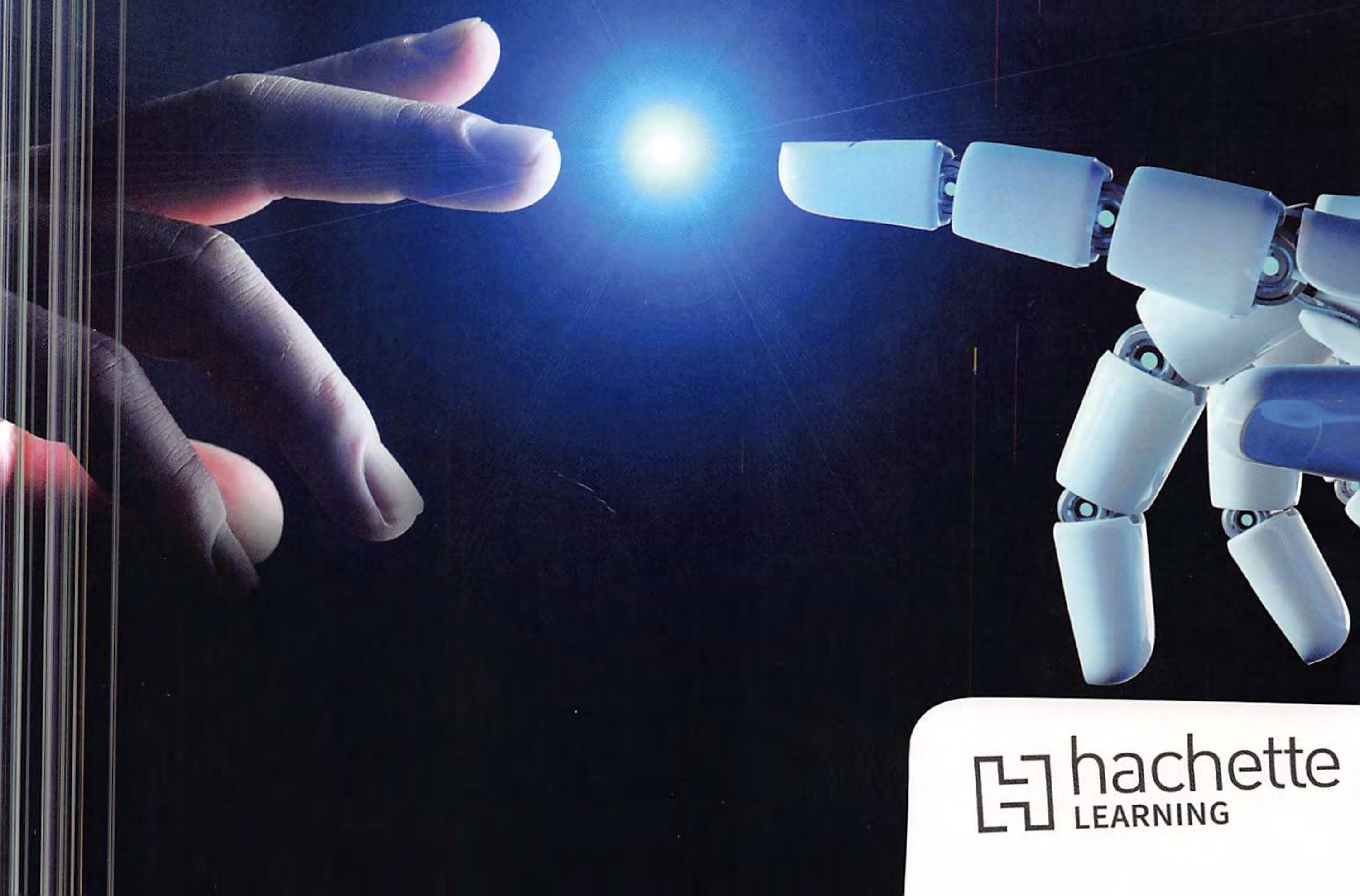
FOR THE
IB DIPLOMA
PROGRAMME

Computer Science

Paul Baumgarten
Ioana Ganea
Carl Turland



hachette
LEARNING

FOR THE
IB DIPLOMA
PROGRAMME

IN COOPERATION WITH ib

Computer Science

Paul Baumgarten
Ioana Ganea
Carl Turland

hachette LEARNING

ISBN: 978 1 0360 0900 7

© Paul Baumgarten, Ioana Ganea, Carl Turland 2025

First published in 2025 by Hachette Learning,
An Hachette UK Company
Carmelite House
50 Victoria Embankment
London EC4Y 0DZ

www.HachetteLearning.com

Impression number 10 9 8 7 6 5 4 3 2 1

Year 2029 2028 2027 2026 2025

Cover photo © sdecoret - stock.adobe.com

Typeset in ITC Berkeley Oldstyle Std 10/14pt by DC Graphic Design Limited, Hextable, Kent

Printed and Bound in Great Britain by Bell & Bain Ltd, Glasgow

A catalogue record for this title is available from the British Library.

Contents

Introduction

Welcome to *Computer Science for the IB Diploma*, written to meet the criteria of the new *International Baccalaureate (IB) Diploma Programme Computer science guide* (published 2025, first exams May 2027). This text addresses the full scope of the syllabus, both the Standard Level and Higher Level components, and caters for both the Python and Java programming language options.

It has been said that computer science is a modern-day superpower, and rightly so. It has a profound impact on society and has driven much of the transformational change we have experienced in recent years. It has advanced fields as diverse as agriculture, finance, manufacturing, health and medicine, transportation, education and global communications. Contemporary living has been forever altered thanks to changes enabled by advances in computing. This impact will only continue to grow exponentially in the years ahead, and the opportunities are limited only by your imagination.

We hope you are excited about the journey ahead and ready to embrace the challenges and opportunities it brings!

The "In collaboration with IB" logo signifies that the content of this book has been reviewed by the IB to ensure it fully aligns with the current IB curriculum and offers high-quality guidance and support for IB teaching and learning.

How to use this book

The following features of this book will help you consolidate and develop your understanding of Computer Science, through concept-based learning:

These are key prompts from the IBDP Computer science guide *that frame each section with the purpose of promoting inquiry.*

SYLLABUS CONTENT

- This coursebook follows the order of the contents of the IB Computer Science Diploma syllabus, with two exceptions.
  - B2.3 is inserted between B2.1 part 1 and B2.1 part 2. This allows us to introduce the programming fundamentals of selection, loops and functions before B2.2, which introduces data structures. B2.2 would have been far more theoretical and abstract if we sought to introduce data structures prior to concepts such as if-statements and loops.
  - B3.1.3 appears after B3.1.5. This allows us to introduce the idea of static methods and properties after learning how to write code that implements objects.

  In both cases, we felt a small reordering created a better flow and provided for a more practical teaching and learning sequence. The alternative would have resulted in attempting to teach programming ideas before introducing the concepts on which they depend.
- At the beginning of each chapter is a list of the content to be covered, with all subsections clearly linked to the content statements, and showing the breadth and depth of understanding required.

Key information

These boxes highlight essential knowledge needed for the examination.

◆ Definitions appear throughout in the margin to provide context and help you understand the language of Computer Science. There is also a glossary of all the key terms at the end of the book.

ACTIVITY

Approaches to learning (ATL), including learning through inquiry, are integral to IB pedagogy. These activities are designed to get you to think about real-world applications of Computer Science.

Common mistake

These boxes detail some common misunderstandings and typical errors made by students, so that you can avoid making the same mistakes yourself.

Top tip!

This feature includes advice relating to the content being discussed and tips to help you retain the knowledge you need. These boxes also include advice on how to approach various common programming scenarios – whether in programming code or in written form, such as in the exams.

TOK

Links to Theory of Knowledge allow you to develop your critical-thinking skills and deepen your understanding of Computer Science by bringing in discussions about the subject beyond the scope of the content of the curriculum.

Linking questions

Each section has a set of linking questions that connect to other parts of the syllabus or TOK. They are designed to facilitate connections and promote conceptual understanding. The list in this coursebook is not exhaustive; you may encounter other connections between concepts, leading you to create your own linking questions.

REVIEW QUESTIONS

Self-assessment questions appear throughout the chapters, phrased to assist comprehension and recall.

PROGRAMMING EXERCISES

Programming exercises appear at the end of chapters. Their purpose is to provide practical, hands-on experience in applying the concepts and principles of Computer Science to a programmed solution. Being able to solve exercises so they work on the computer will be essential to gaining the confidence needed to solve similar problems in exam settings, when you only have paper and pen to work with.

Finally, these programming exercises will help build your expertise for the internal assessment.

Sample answers to the programming exercises in Sections B2 and B3 can be found at www.hachettelearning.com/answers-and-extras

EXAM PRACTICE QUESTIONS

Author-written exam-style questions appear at the end of each section. These simulate scenario-based questions of the breadth and depth that can be anticipated in your examinations. They are intended to serve as a revision and preparation tool to assist you in identifying areas of strength and weakness, as well as to refine your problem-solving skills.

It is recommended that you use these practice questions under exam conditions to make the most of them. Each question has a marks allocation, which also approximates the number of minutes it should take for you to complete. Once you have completed a batch, check the answers while the material is fresh (answers can be found at www.hachettelearning.com/answers-and-extras). Create a log of recurring mistakes for you to review and practise further.

For the programming questions, do make sure you take the time to practise hand-writing your responses. Typing code on the computer is very different from hand-writing it, so you want to have plenty of practice at hand-writing code before your IB examinations. Pay particular attention to consistency of spelling; use of upper and lowercase; and clear lines of indentation.

International mindedness is indicated by this icon. It explores how the exchange of information and ideas across national boundaries has been essential to the progress of Computer Science and illustrates the international aspects of the subject.

The IB learner profile icon indicates material that is particularly useful to help you towards developing the following attributes: to be inquirers, knowledgeable, thinkers, communicators, principled, open-minded, caring, risk-takers, balanced and reflective. When you see the icon, think about what learner profile attribute you might be demonstrating – it could be more than one.

About the authors

Paul Baumgarten

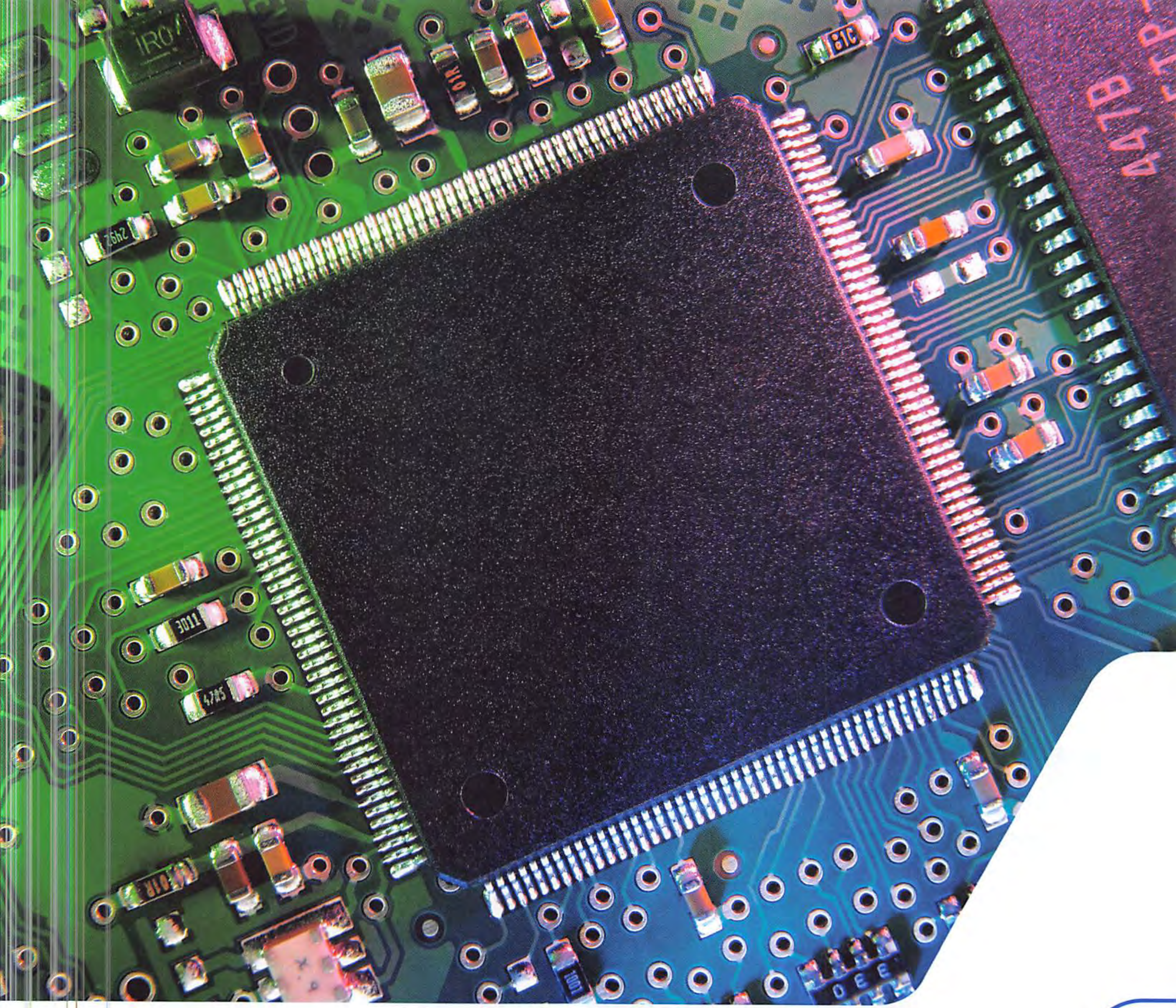
Paul is a Computer Science teacher who has had a life-long fascination with all things geeky. He started tinkering with electronics at age 8 and has been programming since 13, when he taught himself BASIC. Holding a BSc (Computer Science) from Edith Cowan University and a Graduate Diploma in Education from University of Western Australia, he has been teaching Computing since 2006. He moved to Switzerland in 2015, where he began teaching the International Baccalaureate Diploma programme, and is currently teaching in Hong Kong. Passionate about promoting diversity in the tech field, he is committed to increasing representation of women and minorities in Computer Science, believing that societal advances through technology are only truly possible when the contributions and perspectives of everyone are included. Beyond teaching, he is an avid science-fiction reader and enthusiast, particularly relating to space and time travel. He is also the founder of CodingQuest.io, an annual online programming competition for secondary Computer Science students globally.

Ioana Ganea

Ioana Ganea is an experienced educator, having taught Computer Science for over 15 years in different international environments, such as Romania, Germany, the United Kingdom, Egypt and Luxembourg. Her passion for Computer Science started at the age of 11 when her father purchased her very first device and encouraged her to explore both hardware and software concepts without thinking that something can go wrong, as a computer can always be replaced. She graduated from the Academy of Economic Studies in Bucharest, Romania, with a bachelor's degree in Economic Cybernetics, Statistics and Informatics, specializing in Economic Informatics, and she obtained a master's degree in Civil Engineering from the Technical University of Civil Engineering of Bucharest (Computer Assisted Technologies – Department of Teacher Training). She is an experienced examiner, moderator and team leader for various exam boards, and she has collaborated with Oxford Study Courses, offering Computer Science revision courses for IB DP Computer Science, both Standard Level and Higher Level. As an educator, she strives to raise each student's potential and encourage them to believe in themselves. She enjoys teaching students to apply their knowledge, so they can face the challenges of life with confidence, integrity, compassion, creativity and love of peace.

Carl Turland

Originally from Chessington in the United Kingdom, Carl has spent much of his career abroad, teaching in Indonesia, Thailand and Switzerland. He began his professional journey as a programmer for Sky Television in the UK, and now serves as the Head of Design and Computer Science at the International School of Lausanne. Carl holds an HND in Computer Science from Nottingham Trent University, and earned a BA (Hons) in Information Communication Technology with QTS from Brighton University. He was one of the pioneering teachers of the reintroduced Computer Science curriculum in the UK in 2016 and, during that time, helped establish his school as a UK lead in the subject, while contributing to the Compute-IT series (Hodder Education). He later moved abroad to help establish Computer Science programmes at several schools, before joining the International School of Lausanne, where he is now in his sixth year. Carl continues to innovate within the curriculum, expanding into robotics. Outside of the classroom, he is passionate about running, travelling, spending quality time with his wife and young daughter, and cheering on his beloved Crystal Palace football team from the comfort of his sofa.



A1 Computer fundamentals

A1.1 Computer hardware and operation

What principles underpin the operation of a computer, from low-level hardware functionality to operating systems' interaction?

SYLLABUS CONTENT

By the end of this chapter, you should be able to:

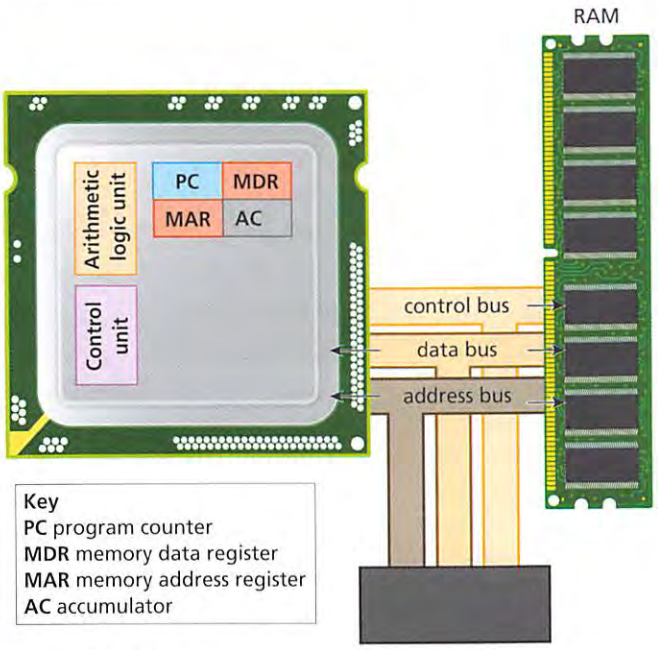
- A1.1.1 Describe the function and interaction of the main central processing unit (CPU) components
- A1.1.2 Describe the role of a graphics processing unit (GPU)
- A1.1.3 Explain the difference between the CPU and the GPU (HL)
- A1.1.4 Explain the purposes of different primary memory types
- A1.1.5 Describe the fetch, decode and execute cycle
- A1.1.6 Describe the process of pipelining in multi-core architectures (HL)
- A1.1.7 Describe the internal and external types of secondary memory storage
- A1.1.8 Describe the concept of compression
- A1.1.9 Describe the different types of services in cloud implementation

A central processing unit (CPU) from above and underneath

A1.1.1 Function and interaction of the main central processing unit components

What is the central processing unit?

The central processing unit (CPU) is often referred to as the "brain" of the computer. It is a critical component that carries out the majority of the processing inside a device.



A model of the CPU

The CPU is made up of two main units: the control unit (CU) and the arithmetic logic unit (ALU).

Control unit (CU)

The control unit directs the operations of the processor. It is responsible for the fetch–decode–execute cycle, managing all three operations and directing the computer's memory, ALU and input/output devices to respond appropriately.

Arithmetic logic lnit (ALU)

This unit is responsible for performing arithmetic and logic operations. These include basic arithmetic operations such as addition, subtraction, multiplication and division, as well as logic operations including AND, OR, XOR and NOT.

What are registers?

Registers are very small amounts of storage that are available directly on the CPU to hold temporary data that the CPU may be working on. The registers are instruction register (IR), program counter (PC), memory address register (MAR), memory data register (MDR) and accumulator (AC).

Instruction register

When an instruction is fetched from memory, it is held in the IR within the CPU. This register holds the instruction that is currently being executed by the CPU.

Program counter

The PC holds the address of the next instruction that is to be fetched from memory. Once the instruction has been fetched, the PC updates to point to the next instruction that will be needed.

Memory address register

The MAR holds the memory address that is currently being fetched. The content from the PC is copied to the MAR, and the MAR provides this address to the memory unit, so that data and instructions can be read from or copied to that location.

Memory data register

This holds the data that has been fetched or is about to be written to the memory address currently in the MAR.

Accumulator (AC)

This stores the intermediate arithmetic or logical results produced by the ALU.

What are buses?

Buses are a critical component of the computer system, as they transfer data between various devices, including the CPU, memory, storage and peripherals. Buses have *widths* that are measured in bits. The bigger the width of the bus, the more data it can transmit at one time. There are three main types of buses: control bus, data bus and address bus.

Control bus

The control bus is used to transmit command and control signals from the CPU to other components of the system, and vice versa. Due to the need for signals to be sent and received, this bus is **bidirectional**. Some of the signals that would be transmitted via the control bus are read / write operations, interrupt requests, clock signals for synchronization and status signals from hardware components.

◆ **Bidirectional bus:** a bus that can transfer data in both directions.

Data bus

The data bus carries the data being processed between the CPU, memory and other peripherals. The width of the data bus is important for determining the amount of data it can transfer at one time. Common data bus widths are 8, 16, 32 and 64 bits. As data needs to be read from and written to memory, data buses are usually bidirectional.

Address bus

The address bus is used to transmit the address that is to be read from or written to in memory. The width of this bus determines the memory capacity of the system. For example, a 32-bit address bus can address 2^{32} memory locations.

What are cores?

CPUs come in a number of different configurations. These include single-core processors, multi-core processors and co-processors.

Single-core processors

This CPU has a single processing unit, meaning it can only handle one task at a time. These are more often found in low-end computers or older machines. They are adequate for simple tasks that do not require heavy multitasking. Single-core processors are able to run more than a single application at a time, but the CPU has to be shared between these applications, which can impact performance.

Multi-core processors

A CPU with multi-core processors has two or more cores that can run multiple instructions simultaneously. These are often referred to as dual-core (two processors), quad-core (four processors), hexa-core (six) or octa-core (eight). Their performance is significantly faster than single-core processors and they are ideal for multitasking, gaming and servers. However, software has to be written to take advantage of these extra cores. Older software that does not do this would likely run at a similar speed as on a single-core processor.

Co-processors

A co-processor is a special type of processor that has a specific job to support the main CPU. These are built with a distinct purpose to achieve optimal performance compared to a general-purpose CPU. Tasks are offloaded by the CPU to the co-processor so they can run in parallel, enhancing the system's performance. Examples of co-processors are graphics processing units (covered in Section A1.1.2), audio processors and digital signal processors (DSPs), which are used in telecommunications and image compression.

Common mistake

A common mistake is thinking that adding more cores to a CPU always makes it faster in a straightforward way – like assuming a dual-core CPU is twice as fast as a single-core, or a quad-core is four times faster. This isn't always true, because the speed increase depends on how well the software can use multiple cores at the same time. Many programs aren't designed to take full advantage of multiple cores, so the extra cores may not make a noticeable difference. Other factors, such as memory speed and how the CPU is designed, also affect how fast it can run. So, just having more cores doesn't automatically mean much faster processing.

REVIEW QUESTIONS

1 What is the primary function of the arithmetic logic unit (ALU) in a computer's CPU?
2 How does the control unit (CU) direct the operations of the CPU?
3 Why is the program counter (PC) important for executing a sequence of instructions?
4 What roles do the data bus and address bus play in the functioning of the CPU?
5 How does the memory address register (MAR) work in conjunction with other CPU components to access memory?
6 How do multi-core processors differ from single-core processors in handling tasks?

A1.1.2 Role of a graphics processing unit

A graphics processing unit (GPU)

A graphics processing unit (GPU) is a specialized electronic circuit designed to accelerate the rendering of images, videos and animations by performing rapid mathematical calculations. Initially developed to handle the demanding graphics workloads of video games and visual applications, GPUs have evolved to play a crucial role in various fields beyond graphics rendering. Their structure, consisting of thousands of small, efficient cores, allows them to process multiple tasks simultaneously, making them exceptionally well-suited for computationally intensive applications. This capability has led to their widespread adoption in scientific research, machine learning, artificial intelligence and cryptocurrency mining. By offloading these intensive tasks from the CPU, GPUs enhance overall system performance, enabling faster and more efficient data processing and visualization.

Graphics processing

Video game graphics

GPUs are designed with a highly parallel structure, enabling them to perform many calculations simultaneously. This makes them exceptionally well-suited for rendering the complex and resource-intensive graphics seen in modern video games and applications. They also handle the application of **shaders and textures** to 3D models, which includes lighting, shading and texture mapping, enhancing the realism of the scene.

◆ **Shaders and textures:** techniques used in 3D rendering to apply effects, lighting and details to models.

◆ **Parallel processing:** the ability of the GPU to perform many calculations simultaneously due to its highly parallel structure.

◆ **Matrix and vector multiplications:** fundamental operations in machine learning and graphics that involve complex mathematical calculations.

Video processing

GPUs assist in the decoding and encoding of video files, making processes such as playback, streaming and editing more efficient and faster. This is particularly helpful for those working with high-resolution video files of 4k or higher.

Artificial intelligence and machine learning

GPUs were originally created for graphical processing; however, in the early 2000s, researchers and engineers began to recognize their potential for handling general-purpose calculations, including those required for machine learning and AI. The shift towards using GPUs for this was largely due to their ability to perform many simple calculations simultaneously, and because many GPUs can be run in **parallel**. Many AI models rely heavily on **matrix and vector multiplications**, and GPUs far outperform a CPU when trying to process these quickly.

◆ **Deep learning:** a subset of machine learning that uses an artificial neural network to imitate the design of the human brain to find generalizations in complex data that can be used for decision-making.

◆ **Proof of work:** a consensus mechanism requiring cryptominers to solve complex problems to add a new block to the blockchain.

This realization gained momentum as machine learning models, especially **deep learning** models, became more complex and required significant computational power for training. By the mid-2000s, GPUs had become essential tools in the field of AI and machine learning, transforming how data scientists and researchers approached problems, significantly reducing the time it took to train complex models.

Blockchain and cryptocurrency mining

The cryptocurrency boom – at its peak in November 2021, the total market capitalization of cryptocurrencies reached approximately $3 trillion

In 2010, the use of GPUs for Bitcoin mining surged as miners discovered that GPUs significantly outperformed CPUs in solving cryptographic puzzles, such as finding the nonce in the hashing algorithm for the **proof-of-work** system. This realization led to a dramatic shift towards GPU mining. The cryptocurrency boom between 2017 and 2021 further escalated the demand for GPUs, resulting in skyrocketing prices and global shortages.

As of 2023, this demand had reduced somewhat and prices of GPUs were becoming more stable. This was for a number of reasons:

- The volatility and reduced profitability of cryptocurrency mining had led to less demand for GPUs, specifically for mining purposes.
- Big manufacturers had increased production to meet the demands.
- Application-Specific Integrated Circuits (ASICs), which are specifically designed for mining, had largely replaced the use of GPUs in many mining operations.

REVIEW QUESTIONS

1 What is the role of a graphics processing unit (GPU) in a computer?
2 How do GPUs enhance the performance of video games and video processing tasks?
3 Why have GPUs become essential in fields such as artificial intelligence, machine learning and cryptocurrency mining?

A1.1.3 Differences between the CPU and the GPU (HL)

The central processing unit (CPU) and the graphical processing unit (GPU) are both core components of modern computers. They are designed differently, which is why they are used for different kinds of tasks. The CPU is great for handling various jobs, but the GPU is better for doing the same job many times on a lot of data at once.

Design philosophies

CPUs are generally called "general-purpose processors" because they can handle many types of tasks. They are designed to run the operating system, process user input and manage programs. CPUs are good at tasks where decisions need to be made quickly, and where different types of work are being done at the same time.

GPUs are specialized processors because they focus on specific types of tasks. They are made for processing large amounts of data in parallel. This means they can work on many calculations at the same time. For example, GPUs are used to process images and videos because they can work on thousands of pixels at once.

Core architecture

The CPU has only a few cores, but these cores are very powerful. Each core can handle many different instructions, but it works best when doing one task at a time. This makes the CPU very good for such tasks as running the operating system, where quick responses are needed. CPUs also have features including branch prediction (where the CPU tries to guess what will happen next) and out-of-order execution (where the CPU can work on tasks that are ready before others).

The GPU has many smaller cores. These cores are not as powerful as the CPU cores, but there are thousands of them, and they all work at the same time. This is why the GPU is very good for tasks such as rendering 3D images, where many similar calculations need to happen at once. The GPU's architecture is designed to work on large sets of data all at the same time.

Memory access and power efficiency

The CPU and GPU access memory differently. The CPU uses a smaller, high-speed memory cache to get data quickly. This is useful when the CPU needs to access small amounts of data many times, such as when running programs or handling user inputs.

The GPU uses its own special memory called VRAM (video RAM). VRAM has a very high bandwidth, meaning it can move large amounts of data at once, such as images and videos. However, the GPU uses more power because it must process a lot of data at the same time, especially when rendering videos or running complex simulations.

Comparison of central processing units (CPUs) and graphics processing units (GPUs)

Processor	Processing	Architecture	Functionality
CPU	It is a **general** purpose processor, capable of handling many different tasks. It executes the instructions of computer programs, involving operations such as arithmetic, logic and controlling input / output (I/O) operations, as directed by the operating system.	CPUs generally have fewer cores. General user devices tend to have between 4 and 8 cores; however, there are some advanced CPUs that now have 64 cores or more. Each core is very versatile, making it capable of handling complex computations that require sequential processing.	Allows the user to switch between multiple tasks and applications. This makes it ideal for running the operating system and general software applications.
GPU	It is a **specialized** processor, with a focus on handling graphics, **rendering** images, video and animations.	Composed of hundreds or thousands of small cores that are well-suited for tasks that can be run in parallel. While each core is not as powerful as a standard CPU core, the high number of cores allows them to perform a large number of calculations simultaneously, making them perfect for graphical processing.	Suited for tasks that require simultaneous processing of large blocks of data, such as rendering images, video processing and deep learning applications.

◆ **Rendering:** the process of generating an image from a model by means of computer programs.

Key information

To summarize, CPUs are better for tasks that require high-speed, complex decision-making and versatility. GPUs are better when the same operation needs to be performed on many data points simultaneously. This means that for tasks such as gaming, video editing and computational research (AI and machine learning), GPUs often significantly outperform CPUs.

How the CPU and GPU work together to increase video-game performance

When playing video games, the CPU and GPU work together to deliver a seamless and immersive experience. The CPU handles the game's core logic, including rules, physical calculations and AI behaviour. It processes the inputs from the player (processing the outcomes of their actions and updating the game state accordingly). The GPU's primary role is to render the game's visuals. It processes **vertex and pixel data** to draw images on to the screen, including 3D objects, textures and effects such as lighting and shadows.

◆ **Vertex and pixel data:** data used by the GPU to render 3D objects and images.

◆ **Frame:** a single image in a sequence of images that makes up a video or animation.

PROGRAMMING EXERCISE

Run benchmark software on your device to see your overall system performance. There are many options out there that you can search for; **https://novabench.com** and **www.userbenchmark.com** have free versions.

Typical scenario

1. **Player input:** The player presses a key to move a character. The CPU processes this input, updates the character's position based on game physics and determines the new game state.
2. **Data preparation:** The CPU prepares the new position and state data and sends it to the GPU.
3. **Rendering:** The GPU updates the **frame** with the character's new position, applies lighting and shading and renders the scene.
4. **Display:** The rendered frame is displayed on the screen, providing immediate feedback to the player.

REVIEW QUESTIONS

1. How do the CPU and GPU work together to enhance video-game performance?
2. Why is a GPU better suited than a CPU for tasks such as video rendering or AI computations?
3. What are shaders and textures, and how do they contribute to the rendering process handled by the GPU?

A1.1.4 Purposes of different primary memory types

Memory types

The primary memory of the computer stores data and instructions that the CPU needs in order to process tasks. Primary memory includes several different types: RAM (random access memory), ROM (read-only memory), caches and registers (covered in Section A1.1.1). These are all types of primary memory, meaning they are used directly by the CPU.

■ Sticks of random access memory (RAM)

RAM

RAM (random access memory) holds instructions and data for programs that are currently running. For example, when you open an app on your phone or computer, it loads into RAM so that is can be accessed quickly by the CPU.

RAM is **volatile**, meaning that it loses its contents when the power to the computer is turned off. This is why, when playing a game, you lose your progress unless you save the game (which is then stored in secondary memory).

One real-world example of using RAM is in smartphones, which use RAM to switch quickly between apps. When you leave an app, it stays in the RAM, so you can return to it quickly without reloading it from scratch.

■ Read-only memory (ROM) attached to a motherboard

ROM

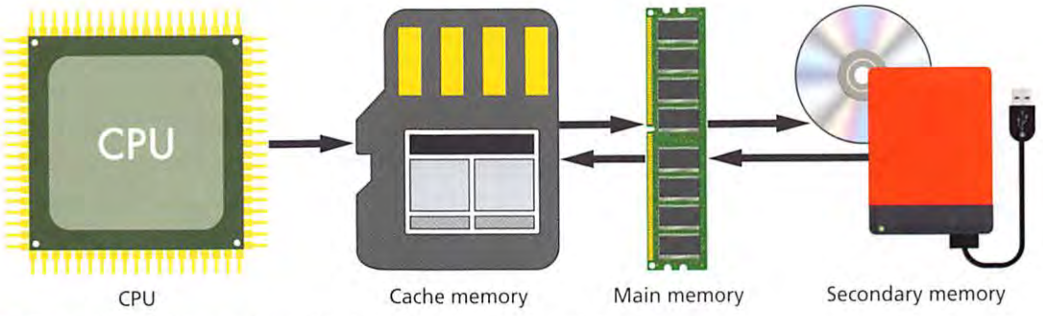
ROM (read-only memory) is used for storing instructions that are very rarely modified. ROM is used for the BIOS (basic input / output system) of the computer, which is located on the motherboard. The BIOS' main role is to initialize and test the system hardware components on startup, and to load the operating system (OS) software from the secondary memory storage into the RAM, ready for the CPU to fetch, decode and execute the instructions.

◆ **Volatile:** a type of memory or storage that loses its data when the power is turned off.

ROM is non-volatile memory, meaning it does not lose its contents when the computer does not have power. While ROM is "read only", meaning it cannot easily change its data, most modern computers use flash memory, which allows for updates and reprogramming. This allows motherboard companies to update their software when required.

A real-world example of using ROM is in smartphones, where ROM stores the operating system and core applications, which do not change unless you perform an update. This ensures that your phone can boot up reliably every time.

Cache (L1, L2 and L3)



■ The order a CPU goes through when trying to retrieve data

Cache memory is small, but provides high-speed access to the CPU compared to the RAM. It acts as a buffer between the CPU and the slower RAM, storing frequently used data and instructions.

There are three types of cache: L1, L2 and L3, each with different sizes and speeds. The closer to the CPU, the faster it is.

- **L1 cache** is located directly on the CPU, making it the fastest type of cache. It can be accessed almost instantly due to its location. However, it is also the smallest, often only a few kilobytes in size (32KB to 128KB per core). Each CPU core usually has its own L1 cache, which is typically split into two sections: L1i to store instructions and L1d to store data.
- **L2 cache** can either be on the CPU, like L1, or situated very close to the CPU. L2 cache is larger than L1 and can be up to several megabytes in size (256KB to 2MB per core), providing more storage for frequently used instructions. It is faster than L3, but slightly slower than L1, though it still significantly speeds up processing by reducing the need to fetch data from the slower RAM.
- **L3 cache** is often located the furthest from the CPU chip. L3 cache may be shared on multiple-core CPUs, whereas L1 and L2 are usually exclusive to a single core. It is the largest of the three, and can be up to tens of megabytes in size (2MB to 64MB shared across all cores). It is the slowest of the three types of caches, but is still significantly faster than RAM.

◆ **Cache hit:** when the CPU requests data and it is found in the cache memory.

◆ **Cache miss:** when the CPU requests data and it is not found in the cache memory, necessitating retrieval from slower main memory or storage.

The terms **cache hit** and **cache miss** are used to describe the efficiency of the CPU's cache memory when retrieving data. A cache hit is the ideal scenario, where the CPU requests data and it is found in the cache memory. A cache miss means it was not found, necessitating retrieval from the slower main memory (RAM) or even slower storage (SSD / HDD).

The percentage of hit rate determines the efficiency and effectiveness of the cache. A low percentage means the system would suffer more from latency, where the data has to be fetched from elsewhere, hindering performance speed. Systems with a larger cache size will generally perform better, as well as systems with more intelligent prefetching techniques that can predict which data will be needed soon and load it into cache ahead of time.

Top tip!

Imagine an onion with its layers representing the levels of cache:

- **L1 cache** is the smallest and fastest, like the very centre of the onion, where everything is tightly packed and closest to the core of the CPU.
- **L2 cache** is slightly larger and slower, like the next layer out – still close to the centre, but not as quick to access as the very core.
- **L3 cache** is the largest and slowest, like the outer layers of the onion. It's still important, but it takes a bit longer to get to, just like how the CPU takes a bit more time to access data in L3 cache compared to L1 and L2.

Optimizing CPU performance with cache

The cache plays a critical role in ensuring that the CPU can access data as quickly as possible. When the CPU finds the searched-for data in the cache (a cache hit), the data can be processed very quickly. However, when there is a cache miss, the CPU has to look for the data in the slower memory, which causes a delay.

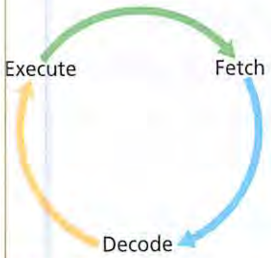
Imagine you are playing a video game on a computer. The CPU frequently checks the L1, L2 and L3 cache to find the data it needs to run the game smoothly. The game's core functions, such as player controls and game logic, might be stored in the L1 cache, while the less frequently accessed data, such as background textures, may be in the L3 cache. The layering system helps to ensure that the game runs smoothly, without interruptions.

A CPU with a larger cache or more advanced prefetching (a technique where the CPU predicts what data it will need and loads it into cache ahead of time) has fewer cache misses and performs better overall.

REVIEW QUESTIONS

1 What is the main purpose of RAM in a computer system, and why is it considered volatile?
2 How does ROM differ from RAM in terms of its function and volatility?
3 Why is cache memory important for CPU performance, and how do the different levels of cache memory (L1, L2, L3) vary in terms of speed and size?
4 What happens during a cache hit and a cache miss, and how do these events impact system performance?

A1.1.5 The fetch–decode–execute cycle



The fetch–decode–execute cycle

The fetch–decode–execute cycle, also known as the "instruction cycle", is the fundamental process that a CPU uses to execute instructions. The cycle consists of three main stages:

1 **Fetch:** The CPU fetches an instruction from the memory.
2 **Decode:** The CPU interprets the instruction and prepares the necessary operations to execute it.
3 **Execute:** The CPU performs the actions required by the instruction.

Little Man Computer

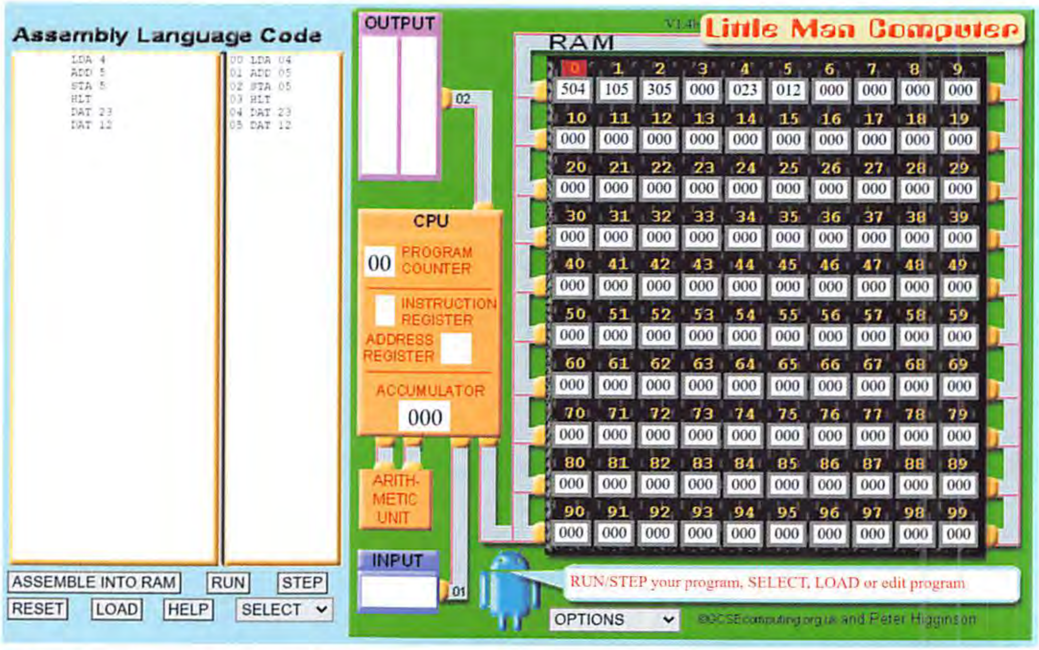
An easier way to see these stages carried out in more detail is to use an educational CPU model known as Little Man Computer, which you can search for online or use the one available here: **https://peterhigginson.co.uk/lmc**. This model uses assembly language – a simple set of instructions, each represented by three letters, which is stored as a three-digit code in the memory. The full set of instructions is:

Instruction	Code	Description
INP	901	Input a value and store it in the accumulator
OUT	902	Output the value from the accumulator
DAT	N/A	Used to define data values directly in memory at the point of declaration, often for constants or variables.
LDA	5XX	Load the value from the specified memory address into the accumulator
STA	3XX	Store the value in the accumulator at the specified memory address
ADD	1XX	Add the value from the specified memory address to the accumulator
SUB	2XX	Subtract the value from the specified memory address from the accumulator
HLT	000	Halt the program
BRA	6XX	Branch (jump) to the specified memory address
BRZ	7XX	Branch to the specified memory address if the accumulator is zero
BRP	8XX	Branch to the specified memory address if the accumulator is positive

Enter the following program into the left-hand column and assemble into RAM. You will see the three-digit representation for each instruction stored at a memory address on the right. For example, LDA 4 has been stored as 504 in memory address 0.

```
LDA 4
ADD 5
STA 5
HLT
DAT 23
DAT 12
```

Your LMC should look like this:



Assembly Language Code

```
LDA 4
ADD 5
STA 5
HLT
DAT 23
DAT 12
```

```
00 LDA 04
01 ADD 05
02 STA 05
03 HLT
04 DAT 23
05 DAT 12
```

RAM

0	1	2	3	4	5	6	7	8	9
504	105	305	000	023	012	000	000	000	000
10	11	12	13	14	15	16	17	18	19
000	000	000	000	000	000	000	000	000	000
20	21	22	23	24	25	26	27	28	29
000	000	000	000	000	000	000	000	000	000
30	31	32	33	34	35	36	37	38	39
000	000	000	000	000	000	000	000	000	000
40	41	42	43	44	45	46	47	48	49
000	000	000	000	000	000	000	000	000	000
50	51	52	53	54	55	56	57	58	59
000	000	000	000	000	000	000	000	000	000
60	61	62	63	64	65	66	67	68	69
000	000	000	000	000	000	000	000	000	000
70	71	72	73	74	75	76	77	78	79
000	000	000	000	000	000	000	000	000	000
80	81	82	83	84	85	86	87	88	89
000	000	000	000	000	000	000	000	000	000
90	91	92	93	94	95	96	97	98	99
000	000	000	000	000	000	000	000	000	000

CPU
PROGRAM COUNTER 00
INSTRUCTION REGISTER
ADDRESS REGISTER
ACCUMULATOR 000

RUN/STEP your program, SELECT, LOAD or edit program

©GCSEcomputing.org.uk and Peter Higginson

■ Peter Higginson's LMC model

First cycle

Click **step**.

1 **Fetch:** The PC (program counter) is currently set to 0, so the instruction at memory location 0 is fetched (504) by opening the 0 address in RAM using the address bus and fetching the instruction on the data bus. The control bus sends a read signal to initiate this process. 5 is stored in the instruction register and 04 in the address register.
While this happens, you will see the PC gets incremented to 1 via the ALU, ready for the next instruction.

2 **Decode:** Once the instruction is fetched, the CPU decodes the instruction. The control unit uses the control bus to co-ordinate this process. The instruction stored in the instruction register is 5, which decodes as "load into the accumulator". The address register 04 indicates the address of the data to load.

3 **Execute:** The command is then carried out. Address 4 is opened on the address bus, and the control bus sends the appropriate signals to retrieve the data (23) from that location on the data bus and store it into the accumulator.

Second cycle

Click **step**.

1 **Fetch:** The CPU now uses the PC to know which instruction to fetch next: 1 is currently stored. Address 1 is opened, and the instruction 105 is fetched. The control bus sends a read signal to initiate this. 1 is stored in the instruction register and 05 in the address register.
The PC is incremented to 2 by the ALU.

2 **Decode:** The instruction 1 is decoded as "add to accumulator"; the address register is the address of the data to add (5). The control unit uses the control bus to co-ordinate this.

3 **Execute:** Address 5 is opened, the data 12 is fetched and both the accumulator (currently 23) and the fetched data (12) are passed to the ALU. The result of 23 + 12 is stored in the accumulator (35).

Third cycle

Click **step**.

1 **Fetch:** The PC is currently 2, so the instruction at memory address 2 is fetched (305). The control bus sends a read signal to initiate this. 3 is stored in the instruction register, and 05 is stored in the address register.
The PC is incremented to 3 via the ALU.
2 **Decode:** The instruction 3 decodes as "store accumulator to address" and the address register gives the location of where to store the data (05). The control unit uses the control bus to co-ordinate this.
3 **Execute:** Memory address 5 is opened via the address bus, and the control bus sends the appropriate signals to send the accumulator contents down the data bus and store them at address 5 (overwriting the current data).

Fourth cycle

Click **step**.

1 **Fetch:** The PC is currently 3, so the instruction at memory address 3 is fetched (000). The control bus sends a read signal to initiate this. 0 is stored in the instruction register, and 00 is stored in the address register.
The PC is incremented to 4 via the ALU.
2 **Decode:** The instruction 0 decodes as "halt". The control unit uses the control bus to signal this operation.
3 **Execute:** The computer halts all operations and ends the program.

Common mistake

A common mistake is assuming that the program counter (PC) gets updated after the execute stage of the fetch–decode–execute cycle. The PC is usually updated during or immediately after the fetch stage, so it points to the next instruction in memory before the current instruction is even decoded or executed. This ensures that the CPU always knows where to find the next instruction in the sequence.

PROGRAMMING EXERCISES

Write an LMC program to:

1 input two numbers, add them, and output the result
2 input a number and output whether it is positive or zero
3 calculate the sum of the first five natural numbers
4 input two numbers and output the larger one
5 input three numbers and output them in ascending order.

REVIEW QUESTIONS

1 What are the main steps in the fetch–decode–execute cycle, and why is this cycle fundamental to CPU operations?
2 How does the CPU use the address, data and control buses during the fetch–decode–execute cycle?
3 Why is the interaction between memory and registers crucial during the fetch phase of the CPU cycle?

A1.1.6 The process of pipelining in multi-core architectures (HL)

◆ **Multi-core architectures:** systems with multiple CPU cores on a single chip, allowing parallel execution of instructions and tasks.

Pipelining is a powerful technique used in **multi-core architectures** to enhance CPU performance by overlapping the execution of multiple instructions. To understand this concept, imagine a carwash service that processes cars through several stages: initial wash, detailed cleaning, rinse and drying. Each stage takes five minutes.

■ A carwash team operating in parallel execution to get the job done faster

In a non-pipelined operation, each car must complete all stages before the next car begins:

Car								
A	initial wash	detailed cleaning	rinse	drying				
B					initial wash	detailed cleaning	rinse	drying

The total time it takes to process two cars is 5 × 8 = 40 minutes. So the time to clean one car is 40 / 2 = **20 minutes**.

The problem with this system is that, once car A has had the initial wash, that stage is then left idle, waiting for car A to complete, before car B enters. This is not efficient and, if we continue with this system, the only way we can improve the operation is to increase the speed of each stage.

It is the same situation with the performance of a CPU, where we are limited by the speed of the hardware, and improving this can be very expensive. Being more efficient with what we have is more beneficial.

In a pipelined solution, as soon as car A finishes a stage, car B enters that stage:

Car					
A	initial wash	detailed cleaning	rinse	drying	
B		initial wash	detailed cleaning	rinse	drying

The total time it takes to process two cars is 5 × 5 = 25 minutes. So the time to clean one car is 25 / 2 = **12.5 minutes**.

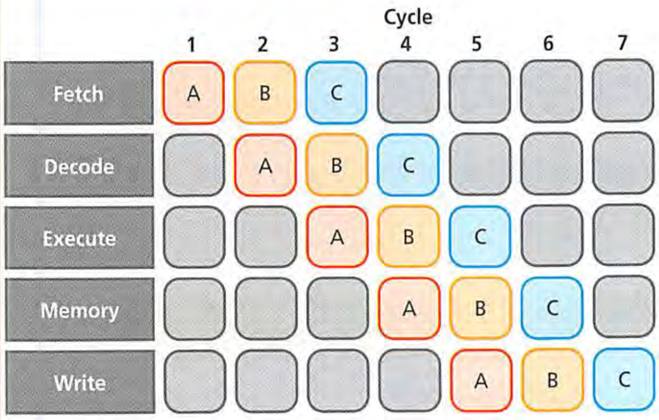
In this pipelined solution, rather than one stage sitting idle until the cycle is complete, the moment it is finished with car A, car B enters that stage.

Design of a basic pipeline

In a pipelined processor, the pipeline consists of multiple stages or segments situated between an input end and an output end. Each stage performs a specific operation, and the output of one stage becomes the input for the next. Intermediate outputs are held in interface registers, also known as "latches" or "buffers". All stages and interface registers are synchronized by a common clock, ensuring co-ordinated operation across the entire pipeline.

In the CPU, the fetch–decode–execute cycle is divided into distinct stages:

1 **Fetch:** The instruction is retrieved from memory.
2 **Decode:** The instruction is interpreted to understand the required operation.
3 **Execute:** The operation is carried out.
4 **Memory access:** Any necessary data is read from or written to memory.
5 **Write back:** The result is written back to the CPU register.



■ Example of a pipeline cycle

Rather than measuring performance in minutes, as in the carwash example, pipeline performance in CPUs is measured in cycles. To manage the five stages mentioned, the CPU is constructed with a five-stage instruction pipeline, ensuring continuous and efficient processing of instructions. A well-optimized pipeline can achieve close to one instruction per cycle, maximizing the CPU's performance by reducing idle times and ensuring continuous instruction processing.

How cores in multi-core processors work independently and in parallel

In multi-core architectures, each core can independently execute its own pipeline of instructions. This is similar to having multiple carwash teams, each capable of processing cars simultaneously but independently. They are also capable of parallel execution when dealing with larger, more complex tasks, where each team completes a part of a larger task to improve execution time. This combination of pipelining and parallelism significantly boosts computational efficiency, enabling modern processors to handle complex and resource-intensive tasks more effectively.

Independent execution

Each core in a multi-core processor has its own set of pipelines, allowing it to fetch, decode, execute and write back instructions independently of the other cores. This independence means that, even if one core is handling a computationally intensive task, other cores can continue to execute their tasks without waiting for the first core to finish. This increases overall efficiency and utilization of the CPU resources.

Consider our carwash with multiple bays:

Team 1 (Core 1): Car A undergoes initial wash – detailed cleaning – rinse – drying

Team 2 (Core 2): Car B undergoes initial wash – detailed cleaning – rinse – drying

While Team 1 is drying car A, Team 2 might be rinsing car B. Both bays operate independently.

Parallel execution

Parallel execution takes the concept further, by allowing multiple cores to work on different parts of a single large task or multiple tasks simultaneously. For instance, in a multi-threaded application, different threads can be scheduled on different cores, with each core processing its thread in parallel. This drastically reduces the time needed to complete complex computations.

Imagine a large car that needs washing, detailing and interior cleaning. Multiple teams (cores) can work on different sections of the car at the same time:

Team 1 (Core 1): Washes the exterior

Team 2 (Core 2): Details the interior

Team 3 (Core 3): Cleans the wheels and undercarriage.

Each team works in parallel on different parts of the same car, drastically reducing the overall time required to complete the job.

Top tip!

Think of pipelining like an assembly line in a factory. Each stage in the pipeline handles a different part of the process and, once a stage finishes its task, it passes the work to the next stage and immediately starts on a new task. This way, multiple instructions are being processed simultaneously, just at different stages. In a multi-core architecture, imagine multiple assembly lines (cores) working in parallel, each running its own pipeline. This set-up greatly increases efficiency because more tasks are completed in less time, and the CPU can handle multiple instructions or even different programs at one time.

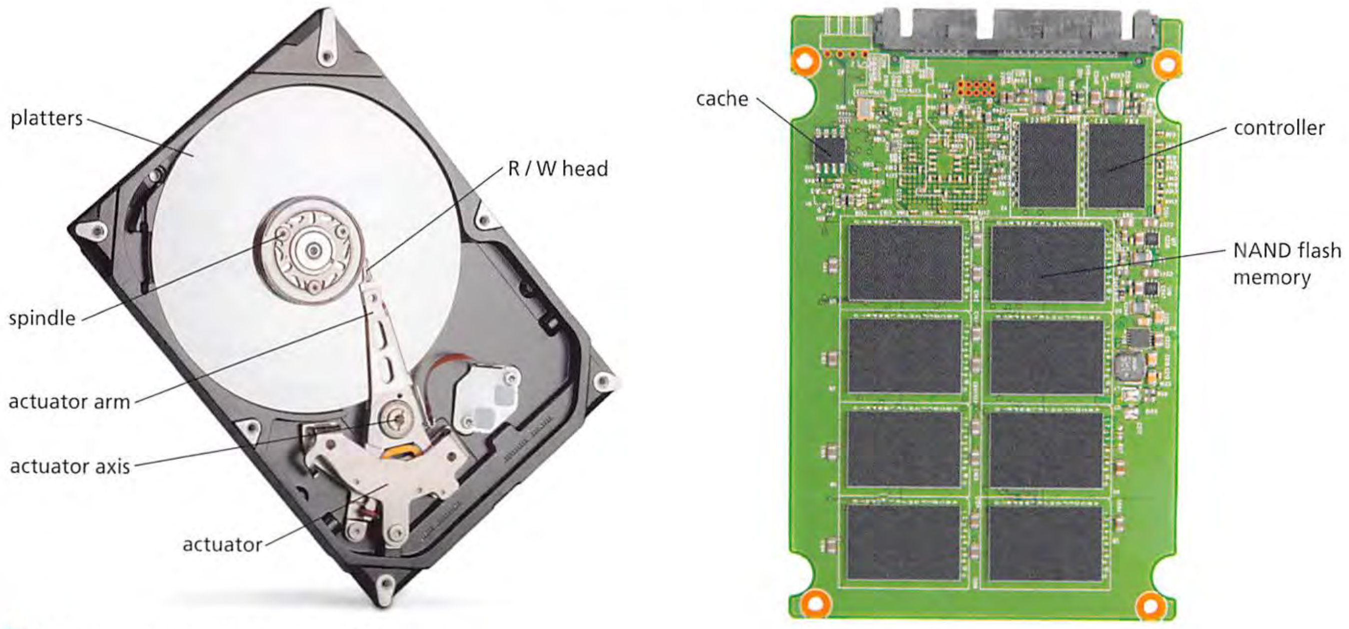
REVIEW QUESTIONS

1. What is pipelining and how does it improve performance?
2. How does a non-pipelined CPU differ from a pipelined CPU in terms of instruction execution?
3. What are the stages of a basic instruction pipeline, and how do they function together in a CPU?
4. How do multi-core processors use pipelining and parallel execution to improve computational efficiency?

A1.1.7 Internal and external types of secondary memory storage

Internal storage

Hard disk drive (HDD) and solid state drive (SSD)



The internals of an HDD and an SSD

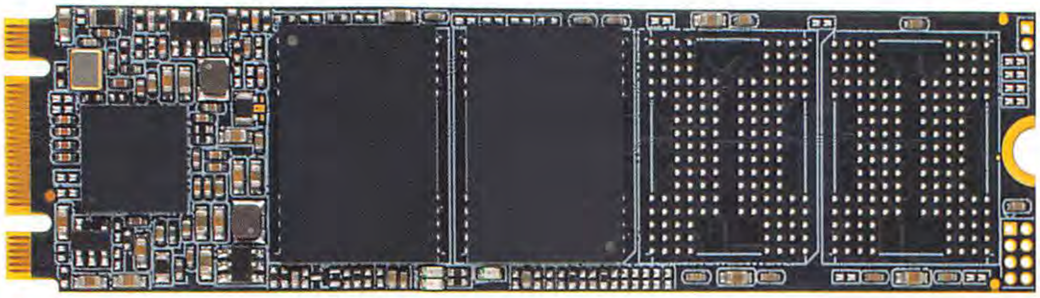
Hard disk drives (HDD) and solid state drives (SSD) are the most typical storage solutions for personal computers. HDDs are older technology but are still often used, especially in non-mobile devices, as they are relatively cheap compared to the amount of storage they offer. HDDs utilize a spinning magnetic disk to read / write data. They are suitable for storing large volumes of data, such as media files, backups and documents, where speed is not so critical.

SSDs have no moving parts. They use flash memory to store data, offering high-speed data access and durability. This makes them very popular in portable devices such as laptops and tablets. They are ideal for operating systems, software applications and games due to their fast read / write speed, which enhances the overall system performance.

■ HDD vs SSD

Feature	HDD (hard disk drive)	SSD (solid state drive)
Storage technology	Magnetic storage with spinning disks and read / write heads	Flash memory with no moving parts
Speed	Slower read / write speeds (generally 50–150 MB/s)	Faster read / write speeds (generally 200–500 MB/s)
Durability	More prone to physical damage due to moving parts	More durable; resistant to physical shock
Noise	Produces noise due to moving parts	Silent operation
Power consumption	Higher power usage due to mechanical parts	Lower power consumption
Cost	Generally cheaper per GB	More expensive per GB
Capacity	Available in larger capacities (up to several TB)	Typically available in smaller capacities (up to several TB, but at a higher cost)
Weight	Heavier due to mechanical components	Lighter
Heat generation	Generates more heat due to moving parts	Generates less heat

There is another form factor for SSDs that is currently popular and offers various advantages. M.2 SSDs look like a stick of chewing gum. They are very small and thin, and take up a lot less space than a standard SSD. M.2 NVMe SSDs are also faster than 2.5" SATA SSDs and are considered easier to install – you just slot them into the motherboard and use a single screw to keep them in place.


■ M.2 SSD

eMMC (Embedded MultiMediaCard)

■ Two eMMCs

In low-cost devices, such as entry-level smartphones and budget laptops, where all the benefits of SSDs are not essential, eMMCs are a popular choice. They are also a type of flash storage that utilizes NAND flash memory. They are soldered directly on to the motherboard of the device. While the capacity and speed do not match a standard SSD, their performance is adequate for basic computing needs and simple applications.

An external SSD

External storage

Hard disk drive (HDD) and solid state drive (SSD)

As external storage solutions, both HDD and SSD are popular choices. Their performance and comparison are identical to the internal versions. Which is used depends on the requirements of the user. If you require quick file transfers, backups and a portable solution that is less likely to be impacted by being carried around, SSDs are the best choice. If you need to do extensive backups, store media files or transport large files, but speed is less critical, you may decide an HDD is the better option.

Optical discs and optical drives

From left to right: CD, DVD and Blu-Ray

Optical drives that read / write optical discs, such as CDs, DVDs or Blu-Rays, are becoming less popular, but are still a consideration for external media storage. The cost of an optical disc is low compared to an HDD or SSD and, while their read / write speeds may be slower, they are sufficient for data archiving and playback. However, the discs are prone to scratches, especially if they are not stored correctly, and they require an optical drive to read and write to them, and these are becoming less common in devices these days.

Memory cards

Memory cards are compact storage devices often used in cameras, smartphones and other portable devices. They are ideal for expanding storage in mobile devices and for storing photos and videos in cameras, using NAND flash memory. They come in multiple sizes, such as SD, microSD and CompactFlash, catering to different devices and space requirements. They are known for their durability – they are resistant to physical shocks, extreme temperatures and water, making them ideal for portable devices. Their read / write times are generally slower than SSDs, but outperform those of optical discs.

◆ **RAID (Redundant Array of Independent Disks):** a data storage technology that combines multiple physical drives into a single logical unit to improve performance, provide redundancy and ensure data protection.

Network Attached Storage (NAS)

NAS is a dedicated file storage connected to a network that allows multiple users to access data. It is often used in homes or businesses for centralizing data storage, file storage and data backup. NAS is usually made up of multiple HDDs or SSDs configured in **RAID (Redundant Array of Independent Disks)** configuration. It is normally connected to the network via Ethernet, and runs a lightweight operating system designed for file storage, and the management and sharing of files. As it uses multiple HDDs or SSDs, its capacity is usually high, and it is possible to expand the system further by adding additional drives.

Memory cards

NAS storage solution

REVIEW QUESTIONS

1 What are the primary differences between an HDD and an SDD in terms of performance and durability?
2 Why might a low-cost device, such as an entry-level smartphone, use eMMC storage instead of an SSD?
3 What advantages do NAS (Network Attached Storage) systems offer for home or business environments?
4 How do memory cards compare to optical discs in terms of durability and data storage capabilities?

A1.1.8 Describe the concept of compression

Compression is the process of encoding information using fewer bits than the original representation. Making file sizes smaller has two main advantages: it takes less room on secondary storage and it is faster to transfer across a network. There are two main types of compression: lossless and lossy.

Lossless vs lossy compression

Lossless compression is when data is compressed to a smaller size, but can be restored back to the original without any loss of information. This is important for files such as text files and databases, where a loss of information would be critical. This technique works by identifying and eliminating **statistical redundancy** within the data, and this process can be reversed when needed.

◆ **Statistical redundancy:** the repetition of information within a data set that does not contribute to its uniqueness.

Lossy compression generally outperforms lossless compression when it comes to file sizes; however, it reduces files by permanently eliminating certain information. This information is redundant or less critical data, resulting in a compressed version that is not identical to the original but is, ideally, indistinguishable from the original to human senses. Lossy compression is commonly used for compressing multimedia files such as images, audio and video, where some loss of quality is acceptable in exchange for significantly reduced file sizes.

This can be seen in the images below. While it may be pretty difficult to visually distinguish the difference in quality, the lossy version uses 50 per cent less data than the original.

Original	Lossless	Saved	Lossy	Saved
1.73 MB	1.58 MB	9%	886 KB	50%

Run-length encoding (RLE)

Run-length encoding is an effective lossless data-compression technique used to reduce the size of files containing many consecutive repeated characters.

For example, take this string:

AAAAABBBCCDAA

RLE looks for "runs" where a character is repeated. In the example above, we have five runs:

AAAAA BBB CC D AA

Once RLE has identified these, it encodes the run by replacing it with a pair: the character that repeats and the number of repetitions. So, the runs above become:

5A 3B 2C 1D 2A

The encoded string is then stored as:

5A3B2C1D2A

If we assume each letter stores 8 bits of information, the initial data is $13 \times 8 = 104$ bits, or 13 bytes.

After compressing with RLE, the data is $10 \times 8 = 80$ bits or 10 bytes: a 23 per cent reduction in size.

RLE is straightforward to implement and it is very effective for data with lots of repetitions, such as simple graphics and certain types of text files. RLE was often used on fax machines, which would send text documents via the telephone line. This was because they contained a lot of white space, which meant RLE could achieve compression ratios of up to 8:1. However, for data that does not contain many repeated characters, like a portrait photograph, RLE may not be very effective and, in some cases, may even increase the file size.

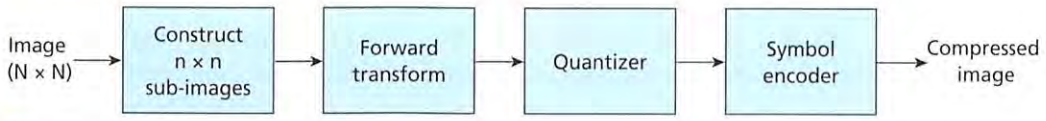
PROGRAMMING EXERCISE

Create an RLE application that has two options: compress or decompress.

The compress option should receive a string and output the encoded version using the RLE algorithm.

The decompress option should do the opposite.

Transform coding



The stages of transform coding

Transform coding is a form of lossy compression often used in JPEG image compression or MP3 audio compression.

Using JPEG compression as an example:

- Transform coding takes an image of N × N size and sections it into smaller sub-images of size n × n.
- Then the *forward transform* is carried out on each of the sub-images. The forward transform can use different algorithms, depending on the type of file compression, but for JPEGs DCT (discrete cosine transform) is used. This takes the image data from the spatial domain (pixel values) to the frequency domain. The output breaks the sub-image down into **low-** and **high-frequency** coefficients.
- These frequency coefficients are then passed to the *quantizer*. This step significantly reduces file size by simplifying the frequency coefficients obtained from the DCT. The purpose of quantization is to reduce the precision of high-frequency components (the fine details) rather than low-frequency components. This is because the human eye is less sensitive to high-frequency data loss compared to low-frequency detail. The extent of the quantization determines the compression level and the quality of the final image.
- The final step of transform coding is the *symbol encoder*. This is where the quantized coefficients are further compressed using entropy coding techniques. This runs through three further algorithms to reduce the file size by efficiently representing the frequency of occurrence of each symbol. The algorithms used at this stage are (in this order):
  1. Zigzag scan
  2. Run-length encoding (RLE)
  3. Huffman coding.

  Once this stage has finished, the final compressed image is complete.

◆ **Low-frequency data:** correspond to slow changes in pixel values, such as broad areas.

◆ **High-frequency data:** correspond to rapid changes in pixel values, representing fine details, edges and textures.

REVIEW QUESTIONS

1. What are the two main advantages of compressing files?
2. Explain the difference between lossless and lossy compression.
3. How does run-length encoding (RLE) work, and in what types of files is it most effective?
4. Describe the process of transform coding in JPEG image compression and explain why it is considered a lossy compression method.

A1.1.9 Types of services in cloud implementation

Cloud computing has revolutionized how organizations manage and deploy IT resources, offering flexible and scalable solutions to meet diverse business needs. There are three primary cloud service models: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Each one provides distinct levels of control, flexibility and management solutions.

Software as a Solution (SaaS)

SaaS delivers software applications over the internet. Users can access these applications through web browsers without needing to install, maintain or update the software locally. SaaS provides a cost-effective and convenient solution for businesses and individuals, offering a wide range of applications from productivity tools to customer-relationship management systems.

SaaS allows users to access their software from anywhere, on any device, as long as they have an internet connection. This eliminates the need for complex software installations. Many SaaS providers charge a subscription fee, which is often less than the cost of purchasing software licences. Additionally, updates and new features are automatically added by the provider, ensuring that users always have the most up-to-date version of the software.

However, SaaS software relies on the user having an internet connection; without it, they cannot run the software, unlike locally installed software. Data security is also a concern, as users rely on the provider's security measures to protect sensitive data.

Example

Google Workspace is an example of SaaS. This suite provides productivity tools, including Gmail, Google Docs and Google Drive, used by businesses and educational institutions for communication, collaboration and storage.

Platform as a Service (PaaS)

PaaS provides a cloud-based platform that allows developers to build, test and deploy applications without managing the underlying infrastructure. PaaS includes tools and services to facilitate application development, such as databases, **middleware** and development frameworks.

◆ Middleware: software that connects different applications, allowing them to communicate and share data. It helps different parts of a computer system work together smoothly.

PaaS accelerates software development by allowing developers to focus on coding rather than infrastructure management. It also makes it easier and cheaper to scale hardware as the user base increases. However, this solution can lead to vendor lock-in, making it difficult to move applications to different platforms and offering less control over the hosting environment.

Example

Microsoft Azure App Service is an example of PaaS. It is a platform for building, deploying and scaling web apps and APIs, used by developers to create scalable and reliable applications without managing the underlying servers.

Infrastructure as a Service (IaaS)

IaaS provides virtualized computing resources over the internet, such as virtual machines, storage and networks. This allows businesses to rent IT infrastructure instead of buying and managing physical servers.

Unlike PaaS, IaaS gives users full control over their virtual machines and networks. This reduces the need for upfront investment in hardware and allows businesses to rent solutions at a lower initial cost using a subscription model. IaaS is also scalable, making it easy to adjust resources as the user base grows. However, IaaS requires more technical knowledge than PaaS, as users must manage their own devices and secure their own data and applications.

Example

Amazon Web Services (AWS) EC2 is an example of IaaS. Businesses use AWS EC2 to create and manage virtual servers, providing the flexibility to run applications without owning physical hardware.

REVIEW QUESTIONS

1 What is Software as a Service (SaaS) and how does it differ from traditional software installation?
2 Explain how Platform as a Service (PaaS) benefits software developers.
3 Why might a business choose Infrastructure as a Service (IaaS) over purchasing physical hardware?

EXAM PRACTICE QUESTIONS

Note: All the exam practice questions are representative of those that will be found on Paper 1 for the International Baccalaureate Diploma in Computer Science.

1 Describe the function of the arithmetic logic unit (ALU). [2]
2 Outline the role of the program counter (PC). [2]
3 Explain the advantages of multi-core processors compared to single-core processors. [3]
4 Describe how the architecture of a GPU differs from a CPU, and why it is better suited for tasks such as video rendering. [3]
5 Compare the processing power of a CPU and a GPU in handling complex computations. [4]
6 Explain the role of L1 cache in a computer system. [2]
7 Describe the fetch–decode–execute cycle that a CPU uses to process instructions. [4]
8 Explain the concept of pipelining in multi-core processors. [3]
9 Describe the differences between solid state drives (SSD) and hard disk drives (HDD). [4]
10 Describe the method of lossy compression and give an example of its use. [3]

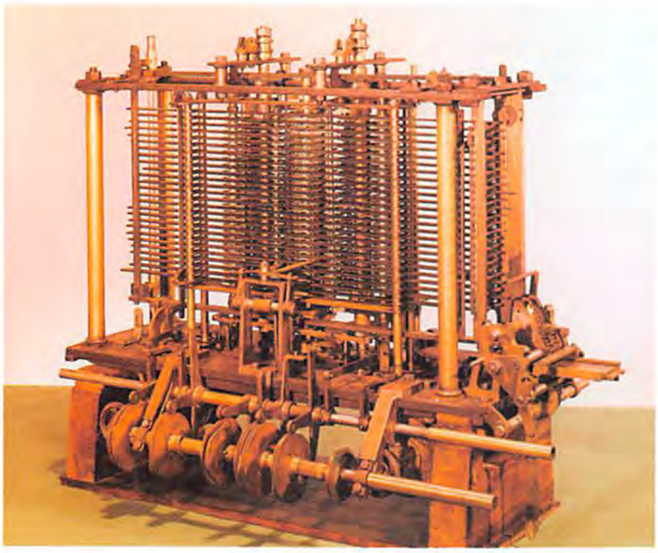
A1.2 Data representation and computer logic

SYLLABUS CONTENT

By the end of this chapter, you should be able to:

- A1.2.1 Describe the principal methods of representing data
- A1.2.2 Explain how binary is used to store data
- A1.2.3 Describe the purpose and use of logic gates
- A1.2.4 Construct and analyse truth tables
- A1.2.5 Construct logic diagrams

A1.2.1 Principal methods of representing data


The Analytical Engine, conceived by Charles Babbage in the 19th century

The Setun computer, developed in 1958

Binary (base-2) is the language for modern-day computers; however, this was not always the case. When developing early computers, several number systems were trialled. Charles Babbage, the inventor of the Analytical Engine, used decimal for his inventions. This seemed a logical choice as people already commonly used base-10.

The ternary system (base-3) was also explored. The Setun computer, developed in the Soviet Union in 1958, used this system. Over 50 of these were produced for educational and scientific institutions to help explore the benefits of ternary logic in computing. Despite its innovative approach, the practical challenges and the widespread adoption of binary logic eventually led to its replacement.

Other scientists and inventors also explored quaternary (base-4) and other number systems. However, practical implementations of these systems were rare due to the increased complexity in hardware design and the limited benefits compared to binary.

Modern computing ultimately settled on binary (base-2) as the primary number system. The base-2 system represents two possible states: 1 or 0. This is in contrast to the number system we are all comfortable with, the decimal system (base-10), which has ten possible states: 0 to 9. The binary system is particularly well-suited to represent the state of electrical switches within a computer system: on (1) and off (0). This simplicity reduces hardware complexity and enhances reliability.

Binary reduces the complexity in hardware design because digital electronics, such as transistors, naturally operate in binary mode. Transistors act as switches that can be turned on or off, aligning perfectly with the binary system's two-state logic. Additionally, Boolean algebra, the mathematical framework for logical circuit design and operation, enables straightforward implementation of complex operations using simple logic gates with binary inputs: 1 (on / true) or 0 (off / false).

The increased reliability of binary systems stems from their use of only two states. Small variations in signal strength do not affect data integrity as much as in systems using larger bases, making binary more robust in **noisy** environments.

◆ **Noise:** unwanted electrical disturbances that can affect the integrity of signals being processed by a computer; this noise is not related to sound, but to variations in voltage or current that can disrupt the accurate transmission and processing of digital data.

◆ **Bit:** binary digit; a single digit, either 1 or 0.

◆ **Byte:** 8 bits.

As all data on a computer system is stored in binary, we need systems to represent numerous types of data, such as integers, strings, characters, images, audio and video, in binary form.

Representation of integers in binary

To represent numbers in binary, it is useful to remember the basics of our decimal system (base-10).

In the decimal system, as we count, we start with a single digit and increment it by 1 until we reach 9. After 9, we introduce a new digit to the front to represent larger numbers. Let's break down the decimal number 1024:

1000s	100s	10s	1s
1	0	2	4

This can be expressed as:

$(1 \times 1000) + (0 \times 100) + (2 \times 10) + (4 \times 1) = 1024$

Each decimal place value increases by a multiple of 10 as we move to the left because we are working in base-10.

Binary, and other base systems, work in a similar way but, instead of 10 possible states per digit, binary has only two (0 and 1). Consequently, each digit increases by a multiple of 2. Let's break down the binary number 0110:

8s	4s	2s	1s
0	1	1	0

This can be expressed as:

$(0 \times 8) + (1 \times 4) + (1 \times 2) + (0 \times 1) = 6$

In this example, we have no 8s, one 4, one 2 and no 1s. Adding 4 + 2 gives us the decimal (base-10) equivalent of the binary (base-2) number 0110. To clearly denote whether we are showing a binary or decimal number, we usually put the base as a subscript, to avoid confusion:

$0110_2 = 6_{10}$

When working with computer systems, we usually deal with **8-bit binary numbers**. A **bit** can be defined as a "binary digit", and 8 bits is equivalent to 1 **byte**. If the number does not require 8 bits to represent it, we usually pad out the extras with 0s. For example, the decimal number 33 would be represented as:

128s	64s	32s	16s	8s	4s	2s	1s
0	0	1	0	0	0	0	1

This means that with 8 bits, we can represent **256 different numbers**, from **0 to 255**. If we want to represent larger numbers, we need more bits to represent this.

Common mistake

When seeing the number 11111111_2, a common mistake is to say this is 256. However, remember that while there are 256 number possibilities, 255 is the largest number we can represent in 1 byte (as 0 can also be represented).

When referring to bits and bytes, a lowercase "b" is used to represent bits and an uppercase "B" is used to represent bytes. We then use a prefix system as the numbers increase.

There are two types of prefixes when referring to bits and bytes: one for base-10 (e.g. kilo, mega, giga) and another for base-2. There was a time when base-10 prefixes were also used for base-2 quantities due to their similarity (e.g. 1024 is close to 1000). However, the confusion this generated led to calls for change. To address this, in 1999 the IEC introduced new prefixes (e.g. kibi, mebi, gibi) specifically for base-2 multiples (1024, 1,048,576, 1,073,741,824).

Kibibyte KiB	**Mebibyte MiB**	**Gibibyte GiB**	**Tebibyte TiB**	**Pebibyte PiB**	**Exbibyte EiB**	**Zebibyte ZiB**
1 KiB = 1024 bytes	1 MiB = 1024 KiB	1 GiB = 1024 MiB	1 TiB = 1024 GiB	1 PiB = 1024 TiB	1EiB = 1024 PiB	1 ZiB = 1024 EiB
Kilobyte KB	**Megabyte MB**	**Gigabyte GB**	**Terabyte TB**	**Petabyte PB**	**Exabyte EB**	**Zettabyte ZB**
1 KB = 1000 bytes	1 MB = 1000 KB	1 GB = 1000 MB	1 TB = 1000 GB	1 PB = 1000 TB	1 EB = 1000 PB	1 ZB = 1000 EB

REVIEW QUESTION

Bits and byte notation are worth knowing when dealing with mobile-phone and internet companies.

Download speeds of up to 100Mb/s!

or

Download speeds of up to 100MB/s!

If the two advertisements above were from two different internet companies, assuming the cost is the same, which one offers faster speeds and by how much?

Converting binary numbers to decimal

There are two main methods for converting a binary number to decimal: the positional notation method and the doubling method.

Positional notation method:

This is possibly the most straightforward method, where you assign the place values and sum.

1 Starting from the right, assign the place values for each binary bit.
2 Sum each of the place values that has a 1 underneath it.

For example, to convert 10111011_2 to decimal:

128	64	32	16	8	4	2	1
1	0	1	1	1	0	1	1

128 + 32 + 16 + 8 + 2 + 1 = 187

Doubling method:

1 Start with the leftmost bit (the most significant bit).
2 Double the current total and add the next bit.
3 Repeat until all bits are processed.

Top tip!

When converting to and from binary, it is always a good idea to write the digit place values down first. Trying to remember these in your head can lead to silly mistakes.

128	64	32
16	8	4
2	1	

For example, to convert 10111011_2 to decimal:

Step	Binary digit	Current total	Calculation
1	1	1	Initial value
2	0	2	$1 \times 2 + 0 = 2$
3	1	5	$2 \times 2 + 1 = 5$
4	1	11	$5 \times 2 + 1 = 11$
5	1	23	$11 \times 2 + 1 = 23$
6	0	46	$23 \times 2 + 0 = 46$
7	1	93	$46 \times 2 + 1 = 93$
8	1	187	$93 \times 2 + 1 = 187$

Common mistake

If you use this method, remember to start with the most significant bit (MSB), not the **least significant bit** (LSB).

◆ **Least significant bit (LSB):** the rightmost bit in a binary number, representing the smallest value position (0 or 1).

◆ **Quotient:** the result obtained when one number is divided by another, e.g. in the division of 15 by 3, the quotient is 5.

REVIEW QUESTIONS

Convert the following binary (base-2) numbers to decimal (base-10):

1 11001010_2

2 01101101_2

3 10110011_2

4 00011110_2

5 11100001_2

Converting decimal numbers to binary

There are two main methods for converting a decimal number to binary: the division method and the subtraction method.

Division method:

1 Divide the decimal number by 2.

2 Write down the **quotient** and the remainder.
The remainder will be either 0 or 1. This represents a digit of the binary number (the LSB on the first division).

3 Update the quotient.

4 Repeat until the quotient is 0.

5 Construct the binary number (this is read from the remainders from the first to the last).

For example, to convert 42_{10} to binary:

Division step	Quotient	Remainder
42 / 2	21	0
21 / 2	10	1
10 / 2	5	0
5 / 2	2	1
2 / 2	1	0
1 / 2	0	1

Construct the binary number from the remainders and pad to 8-bits: 00101010_2

Common mistake

Remember to construct the remainders in the correct order to format your binary number. The first remainder is the least significant bit (LSB).

Subtraction method:

Write down the place values for an 8-bit binary number:

128 64 32 16 8 4 2 1

Starting with the largest place value (128):

1. Try and subtract it from the number you are converting.
   - ☐ If the place value is larger than the number, write a 0 below it.
   - ☐ If it is smaller or equal to it, write a 1 and calculate the remainder of the subtraction, carrying the result to the next place value.
2. Repeat.

For example, to convert 42_{10} to binary:

128_{10} and 64_{10} are larger than 42_{10}, so we write 0 below these.

128	64	32	16	8	4	2	1
0	0						

32_{10} is smaller, so we write a 1 below it and calculate the remainder from the subtraction, the result of which will carry to the next place value:

$42_{10} - 32_{10} = 10_{10}$

128	64	32	16	8	4	2	1
0	0	1	0				

16 is larger than 10, so write a 0

8 is smaller, so write a 1 and calculate the remainder:

$10_{10} - 8_{10} = 2_{10}$

4_{10} is larger than 2_{10}, so write a 0

2_{10} is equal, so calculate the remainder (0) and write a 1

128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	0

REVIEW QUESTIONS

Convert the following decimal (base-10) numbers to binary (base-2):

1. 20_{10}
2. 87_{10}
3. 123_{10}
4. 199_{10}
5. 250_{10}

PROGRAMMING EXERCISE

Write a binary-to-decimal and decimal-to-binary application in either Python or Java.

Representation of integers in hexadecimal

Hexadecimal (often abbreviated as hex) is a base-16 number system that uses 16 distinct symbols to represent values, rather than the 10 of decimal or 2 of binary. The symbols include the digits 0 to 9 and then the letters A to F, where A represents 10, B represents 11, C represents 12, D represents 13, E represents 14 and F represents 15.

◆ **Debugging tools:** software applications or utilities used by developers to identify, analyse and fix bugs or issues within a program by inspecting code, variables and execution flow.

◆ **Memory dump:** a process where the contents of a computer's memory are captured and saved, typically for the purpose of diagnosing and debugging software issues.

◆ **Nibble:** 4 bits.

Hexadecimal is used with computers for several reasons. The ease of conversion between binary and hexadecimal is straightforward because each hex digit maps directly to a 4-bit binary sequence. For example, the binary number 1111 can be represented as F in hex. Another reason is that it provides a more compact way to represent a binary value. This makes it much easier for us to read and communicate large binary numbers. This is why you often see hex used in **debugging tools**, **memory dumps** and assembly language programming.

Converting binary numbers to hexadecimal

Converting binary to hexadecimal is a straightforward calculation.

1 Split the binary byte (8 bits) into two **nibbles** (2 × 4 bits).
2 Calculate the decimal value of these 4 bits.
3 Convert the decimal values into their hexadecimal equivalents and rejoin them.

For example, to convert 01101011_2 to hexadecimal:

1 Split the byte into 2 nibbles:
0110_2 1011_2
2 Calculate the decimal value:
6_{10} 11_{10}
3 Convert both decimal values to their hexadecimal equivalents and rejoin them:
$6B_{16}$

REVIEW QUESTIONS

Convert the following binary (base-2) numbers to hexadecimal (base-16):

1 10101100_2
2 11010110_2
3 11010001_2
4 00111010_2
5 10011101_2

Converting hexadecimal numbers to binary

Moving from hex to binary is just a reverse of the binary-to-hexadecimal process:

1 Split the two hexadecimal digits.
2 Convert each of them to a 4-bit binary number using the same integer-to-binary method.
3 Join the two 4-bit numbers together to form 1 byte.

For example, to convert $F2_{16}$ to binary:

1 Split the two hexadecimal digits:
F_{16} 2_{16}
2 Convert each of them to a 4-bit binary number using the same integer-to-binary method:
1111_2 0010_2
3 Join the two 4-bit numbers together to form 1 byte:
11110010_2

REVIEW QUESTIONS

Convert the following hexadecimal (base-16) numbers to binary (base-2):

1 $3F_{16}$

2 $A9_{16}$

3 10_{16}

4 $7C_{16}$

5 $E2_{16}$

Converting decimal numbers to hexadecimal

To move between decimal and hexadecimal is one of the trickier calculations to perform, as you need to be comfortable with your 16 times table. To convert a decimal number to hex:

1 Divide the decimal number by 16 and record the remainder.

2 Repeat the process with the quotient until the quotient is 0.

3 Form the hex number from the remainders, with the last remainder obtained being the most significant bit (the number on the left).

For example, to convert 254_{10} to hexadecimal:

1 Divide the decimal number by 16 and record the remainder:

$254_{10} / 16_{10} = 15_{10}$ remainder 14_{10}

quotient = 15_{10}

remainder $14_{10} = E_{16}$

2 Repeat the process with the quotient until the quotient is 0:

$15_{10} / 16_{10} = 0_{10}$ remainder 15_{10}

quotient = 0_{10}

remainder $15_{10} = F_{16}$

3 Form the hex number from the remainders, with the last remainder obtained being the most significant bit (the number on the left):

FE_{16}

REVIEW QUESTIONS

Convert the following decimal (base-10) numbers to hexadecimal (base-16):

1 42_{10}

2 157_{10}

3 89_{10}

4 200_{10}

5 123_{10}

Converting hexadecimal numbers to decimal

To convert from hexadecimal to decimal:

1 Convert hex digits to their decimal equivalents.

2 Multiply them by 16 raised to the power of its position index, starting from 0 on the right.

3 Sum the results.

For example, to convert $2F_{16}$ to decimal:

1 Convert hex digits to their decimal equivalents:

$2_{16} = 2_{10}$

$F_{16} = 15_{10}$

2 Multiply them by 16 raised to the power of its position index, starting from 0 on the right:
$2 \times 16^1 = 2 \times 16 = 32_{10}$
$15 \times 16^0 = 15 \times 1 = 15_{10}$

3 Sum the results:
$32 + 15 = 47_{10}$

PROGRAMMING EXERCISE

Add hexadecimal conversion functionality to the binary converter app you created before.

A1.2.2 How b inary is used to store data

The binary system underpins everything from numerical values and textual information to complex multimedia files, ensuring efficient and reliable data processing. In this section, we are going to discover the mechanisms that are used to store such data as characters, strings, images, audio and video in binary form.

Characters and strings

Characters and strings are stored using standardized binary encoding schemes, enabling consistent storage, retrieval and processing across different systems and applications. The most common encoding standards are ASCII (American Standard Code for Information Interchange) and Unicode.

ASCII encoding

The development of ASCII began in 1960 and was officially standardized in 1963. It was developed because there was no standardized way to encode text characters, which led to compatibility issues between devices and systems. Each manufacturer used its own proprietary encoding system, which made it very difficult for devices to communicate with each other. ASCII was designed to provide a common standard for the interchange of text data.

ASCII initially started out as a 7-bit encoding system, which gave it the ability to represent 128 (2^7) different characters, which was considered sufficient for most basic text data (letters, numbers, punctuation and control characters). However, as computing became more global and applications required support for additional characters, an 8-bit extension to ASCII was developed, giving it the ability to represent 256 (2^8) characters. This was referred to as extended ASCII, and the new characters were mainly used for Western European languages.

ASCII uses a simple but clever system to represent characters in binary (as long as we are only considering the Latin (English) alphabet). The first five bits from the right are used to represent the letter by its numerical place in the alphabet.

For example:

01100001_2	=	1_{10}	=	a
01100010_2	=	2_{10}	=	b
01100011_2	=	3_{10}	=	c

Top tip!

If the first five bits from the right are 00000 (five zeros), it is almost certainly a space (00100000).

If the first three bits from the left are not 011 or 010, it is likely to be a punctuation mark.

The first three bits from the left represent whether it is an uppercase or lowercase letter. 011 = lowercase; 010 = uppercase:

01100001_2	=	a
01000001_2	=	A

REVIEW QUESTION

Convert the following binary back into text to reveal the hidden message.

01000110 01101111 01101100 01101100 01101111 01110111 00100000 01110100 01101000 01100101 00100000 01110111 01101000 01101001 01110100 01100101 00100000 01110010 01100001 01100010 01100010 01101001 01110100

PROGRAMMING EXERCISE

Create an application so that you can send secret messages to your friends.

Write an application that accepts either a string of characters or a stream of binary. It should either encode the characters using ASCII and binary or convert the binary back into text.

To make the binary less easy to decode by hand, you could remove all spacing between the 8-bit characters.

Unicode encoding

In the 1960s, the United States and the majority of English-speaking countries had a system in 7-bit ASCII that worked for the English alphabet. Other non-English speaking countries had their own unique encoding systems to work with their own languages. When the ASCII system was increased to 8 bits (extended ASCII), allowing for 256 characters for use in modern computers, countries did not agree on the same standard. Nordic countries started using the extra space to encode characters for their own languages, and Japan used four different systems that were not even compatible with each other. This was not a huge issue as communication between these systems was rare, but then the internet was launched and compatibility became very important as more and more information was being shared between systems in different countries.

In 1991, the Unicode Consortium was created to try and solve this problem. The organization was established to develop, maintain and promote the Unicode Standard, which provides a unique number for every character, regardless of platform, program or language. It needed to create a system that was capable of storing all the characters and punctuation marks from all the languages in the world, but also wanted it to be backwards compatible with ASCII. At the time of writing, the current Unicode Standard version 15.0, released in September 2022, encodes 149,186 different characters. Unicode includes the Latin, Cyrillic, Greek and Arabic alphabets, and Chinese characters, as well as many others, and also includes emojis and mathematical and other technical symbols. In Unicode, each letter or symbol is assigned a unique number, for example:

- A = 65
- 汉 = 27721
- 💩 = 128169

You can find the numerical representation for any character or symbol using the code below:

Python

```python
# Python examples
char_a = 'A'
char_han = '汉'
char_poo = '💩'
# Get Unicode code points as integers
code_point_a = ord(char_a)  # 65
code_point_han = ord(char_han)  # 27721
code_point_poo = ord(char_poo)  # 128169
# Print integer representations
print(code_point_a)  # Output: 65
print(code_point_han)  # Output: 27721
print(code_point_poo)  # Output: 128169
```

Java

```java
public class UnicodeExample {
    public static void main(String[] args) {
        // Define characters
        char charA = 'A';
        char charHan = '汉';
        String charPoo = "💩"; // Note: Java uses UTF-16 and
        // the emoji is usually a surrogate pair
        // Get Unicode code points as integers
        int codePointA = (int) charA; // 65
        int codePointHan = (int) charHan; // 27721
        int codePointPoo = charPoo.codePointAt(0); // 128169
        // Print integer representations
        System.out.println("Unicode code point of 'A': " +
        codePointA); // Output: 65
        System.out.println("Unicode code point of '汉': " +
        codePointHan); // Output: 27721
        System.out.println("Unicode code point of '💩': " +
        codePointPoo); // Output: 128169
    }
}
```

How did they manage this? The story is that it was conceived in a café on the back of a napkin when Joe Becker (Xerox), Lee Collins (Apple) and Mark Davis (Apple and later Google) met and designed the encoding scheme in 1987. There are a few different versions of Unicode: UTF-8, UTF-16 and UTF-32. Each has its own uses:

	UTF-8	UTF-16	UTF-32
Variable length encoding	1–4 bytes per character	2 or 4 bytes per character	4 bytes per character
Note	Compatibility: backward compatible with ASCII	Surrogate pairs: for characters outside the **Basic Multilingual Plane (BMP)**, two 16-bit code units are used	Simplicity: easier to process because each character is exactly 4 bytes
Usage	Most commonly used encoding on the web and in many applications	Often used in Windows and Java environments	Less common due to higher storage requirements

◆ **Basic Multilingual Plane (BMP):** the most commonly used characters and symbols for almost all modern languages.

Let's examine UTF-8, the most commonly used encoding system, and understand its functionality.

Instead of merely expanding the size to accommodate over 100,000 characters, which would have adversely impacted most online content, a more efficient solution was devised. Had all characters been standardized to use 32 bits, each letter in the ASCII system would have quadrupled in size. This would have resulted in significantly larger documents and web pages, leading to increased storage requirements and slower transfer times. The system also needed never to send eight zeros (00000000) in a row, as many older systems would see this as the end of communication and would stop listening.

So the UTF-8 system kept the ASCII system the same. The letter "A" is encoded as:

01000001 = A

However, if the character needed went beyond the standard ASCII system, "é" for example, more than one byte would be required:

11000011 **10**101001 = é

The bits in bold are important. The first three significant bits "110" on the first byte represent that this character is made up of two bytes in total (a 0 is needed at the end to show when this information is finished). The second byte starts "10", which means this is a continuation. If you remove those 5 bits and then put both bytes together:

000 1110 1001 = 233 = é

Another example is:

11110000 **10**011111 **10**011000 **10**000100 = 😁

This emoji requires four bytes using the UTF-8 system. The first byte communicates that this character is made up of four bytes ("11110") and the next three bytes start with "10", showing they are continuation bytes. If we remove that information:

0001 1111 0110 0000 0100 = 128516 = 😁

UTF-8 has been adopted by the internet as the main character encoding system; however, it doesn't come without some issues. Due to the variable length, some characters (especially those from Asian languages or emojis) take more space compared to single-byte encodings. This can lead to larger file sizes in certain contexts. The processing required to handle variable-length encoding also requires more complex processing compared to fixed-length systems such as UTF-32.

◆ **Shift cipher:** a type of substitution cipher, where each letter in the plaintext is shifted a certain number of positions down or up the alphabet.

Despite these issues, UTF-8 has proved to be a versatile and effective encoding standard that meets the needs of the modern internet. Its backward compatibility, efficiency and broad support make it an enduring choice for encoding text. While it does have some challenges, particularly with handling non-ASCII characters and variable-length encoding, these are not significant enough ever to warrant a wholesale replacement. Therefore, it's likely that UTF-8 will continue to be the dominant text encoding standard for the foreseeable future.

PROGRAMMING EXERCISES

The code below uses a Caesar cipher to encrypt the string that is input using a key. A Caesar cipher is a simple **shift cipher**, where each letter is considered to be an integer (a = 1, b = 2, c = 3, and so on) and the key is added to this to find the encrypted letter, for example:

String input: "Hello"

Key input: 1

Output: Ifmmp

Python

```python
def caesar_cipher_encrypt(message, key):
    encrypted_message = ""
    for char in message:
        if char.isalpha():  # Check whether the character is a letter
            shift = ord("A") if char.isupper() else ord("a")  # Determine the
            # ASCII offset
            # Shift the character and wrap around the alphabet if necessary
            encrypted_char = chr((ord(char) - shift + key) % 26 + shift)
            encrypted_message += encrypted_char
        else:
            encrypted_message += char  # Non-letter characters remain unchanged
    return encrypted_message
# User input
message = input("Enter the message to encrypt: ")
key = int(input("Enter the key (an integer): "))
# Encrypt the message
encrypted_message = caesar_cipher_encrypt(message, key)
print(f"Encrypted message: {encrypted_message}")
```

Java

```
import java.util.Scanner;
public class CaesarCipher {
    public static String caesarCipherEncrypt(String message, int key) {
        StringBuilder encryptedMessage = new StringBuilder();
        for (char ch : message.toCharArray()) {
            if (Character.isLetter(ch)) {  // Check whether the character is
            // a letter
                char shift;
                if (Character.isUpperCase(ch)) {
                    shift = 'A';  // Determine the ASCII offset for uppercase
                    // letters
                } else {
                    shift = 'a';  // Determine the ASCII offset for lowercase
                    // letters
                }
                // Shift the character and wrap around the alphabet if
                // necessary
                char encryptedChar = (char) ((ch - shift + key) % 26 + shift);
                encryptedMessage.append(encryptedChar);
            } else {
                encryptedMessage.append(ch);  // Non-letter characters remain
                // unchanged
            }
        }
        return encryptedMessage.toString();
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // User input
        System.out.print("Enter the message to encrypt: ");
        String message = scanner.nextLine();
        System.out.print("Enter the key (an integer): ");
        int key = scanner.nextInt();
        // Encrypt the message
        String encryptedMessage = caesarCipherEncrypt(message, key);
        System.out.println("Encrypted message: " + encryptedMessage);
    }
}
```

1 After studying how this code works, write the decrypt function for someone who receives an encrypted message.

2 Write a function that is able to **brute force** an encrypted message so you can identify the key used.

◆ **Brute force:** a method of breaking a cipher by systematically trying every possible key until the correct one is found.

The first ever digital image: Russel Kirch's son, Walden, in 1957

Images

In 1957, Russel Kirch scanned an **analogue** photo of his son Walden, converting the picture into a digital file. This was the first ever digital image created. It was a significant milestone in the evolution of visual technology, revolutionizing the way we capture, store and manipulate pictures. The development of early digital cameras and scanners, which enabled devices to convert light into digital data, started the trend that has now become commonplace, and the transition from film to digital has transformed numerous industries, from photography and medical imaging to telecommunications and entertainment.

Bitmap images

Bitmap images, also known as "raster" images, are one of the most fundamental forms of digital graphics. They reproduce images by using a grid of **pixels**, with each pixel assigned a specific colour and intensity.

At the bottom of the page is a bitmap image with an **image resolution** dimension of 13×10 (13 pixels wide by 10 pixels high). Each pixel is "described" using 1 bit of data: either 1 or 0. In this case, 1 = black and 0 = white (a monochrome image), and the amount of bits used to describe the colour is known as the "bit depth" or **colour depth**. So, we have a 13×10 image with a 1-bit colour depth in this example.

To calculate the size of this image, the formula is:

image size = width (pixels) × height (pixels) × colour depth (bits per pixel)

13 × 10 × 1 = 130 bits (or 130 / 8 = 16.25 bytes)

However, in reality, this calculation is not completely accurate, as the image would require more data to store **metadata** and other header information. This could include information such as dimensions, colour depth and other attributes that allow the CPU to read the image data accurately so it displays the image correctly to the screen.

◆ **Analogue:** a continuous signal that represents varying physical quantities, such as sound waves, which varies smoothly over a range; digital represents data in discrete binary values (0s and 1s), enabling precise and error-resistant processing.

◆ **Bitmap:** a type of digital image composed of a grid of pixels, each holding a specific colour value, representing the image in a rasterized format.

◆ **Pixel:** short for "picture element"; the smallest unit of a digital image or display, representing a single point in the image with a specific colour and intensity.

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	0	0
0	0	1	1	0	1	1	1	0	1	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	0	1	0	0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0	0	1	0	1	0
0	0	0	0	1	1	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

◆ **Image resolution:** the number of pixels contained within a digital image, typically expressed as the dimensions (width × height) in pixels, and sometimes as the pixel density (PPI / DPI) for print quality.

◆ **Colour depth:** also known as "bit depth"; the number of bits used to represent the colour of each pixel in a digital image, determining the range and precision of colours that can be displayed.

◆ **Metadata:** information that describes other data, providing context and details about the data's content, structure and attributes. In the context of digital images, metadata includes such information as the image's dimensions, colour depth, creation date, author, camera settings and other properties that help with managing, understanding and using the image effectively.

To improve the quality of a bitmap image, we have two options: We can increase the number of pixels (resolution) or we can increase the colour depth.

Resolution – increasing the number of pixels:

Increasing the number of pixels in a bitmap image increases the image quality. A higher resolution allows for greater detail and clarity, and images with lower resolutions can lead to a loss of detail and a pixelated appearance. However, the quantity of pixels is not the only consideration: the size of the screen they are displayed on is also important. Images with a higher PPI (pixels per inch) look clearer than those with a lower PPI. Imagine having an image with a resolution of 1024×768 shown on your phone compared to on a cinema screen. The higher PPI on the phone will give a clearer image due to the increased pixel density. The trade-off for a higher resolution image is larger file size, which can impact storage and transfer efficiency.

■ Common image resolutions

Resolution name	Pixel dimensions	Common usage
VGA	640 × 480	Early computer screens, basic web graphics
SVGA	800 × 600	Standard computer monitors, web graphics
HD (720p)	1280 × 760	HD video, basic HD television
Full HD (1080p)	1920 × 1080	Full HD video, modern monitors and televisions
2K	2048 × 1080	Digital cinema, some monitors
Quad HD (1440p)	2560 × 1440	High-resolution monitors, gaming, professional use
4K (Ultra HD)	3840 × 2160	Ultra HD televisions, high-end monitors, video
8K	7680 × 4320	Cutting-edge televisions, professional video

Colour depth – increasing the amount of colours:

When we increase the colour depth, it allows for a wider range of colours to be represented, resulting in more vibrant and accurate images. If an image's colour depth is low, this can lead to banding, where gradients appear as distinct steps rather than smooth transitions. However, just like image resolution, we must also consider the impact of file size for storage and transfer times. The higher the colour depth, the larger the file size.

To work out the number of colours available, we calculate 2 to the power of the colour depth of the image; for example, for an image with an 8-bit colour depth:

$2^8 = 256$

■ Common colour depths

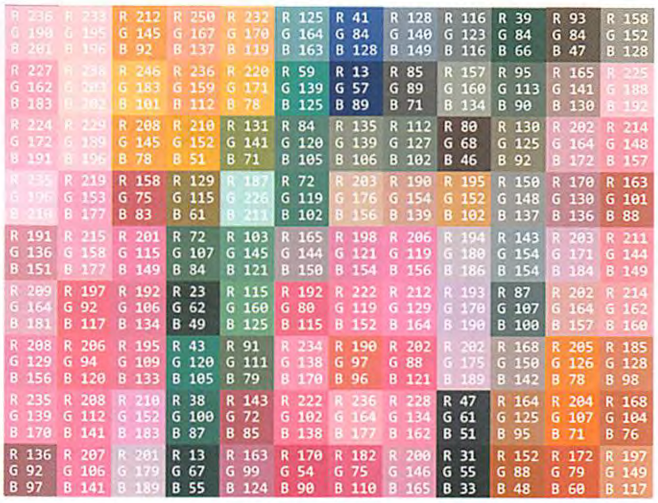
Colour depth (bits per pixel)	Number of colours	Common usage
1 bit	2	Simple graphics, monochrome displays
4 bit	16	Early computer graphics, icons
8 bit	256	GIF images, simple web graphics
16 bit	65,536	High-colour images, some video formats
24 bit (true colour)	16.8 million	Standard for most images and video, digital photography
30 bit (deep colour)	Over 1 billion	Professional photography, high-end monitors and televisions
36 bit	Over 68 billion	Medical imaging, professional graphics
48 bit	Trillions	High-end personal applications, detailed scientific imaging

■ A high-resolution image with a resolution of 2268 × 4032, a 24-bit colour depth and a file size of 1.77 MB

The majority of modern-day screens are 24 bit, allowing for 16.8 million colours. They have three lights per pixel: a red, a green and a blue light, otherwise known as "RGB", and have a value range from 0 to 255 (1 byte per colour channel). This is sufficient for most applications, as most human eyes can only distinguish between around 10 million distinct colours. Monitors that go beyond 24 bit are normally only necessary in professional fields where precision is crucial.

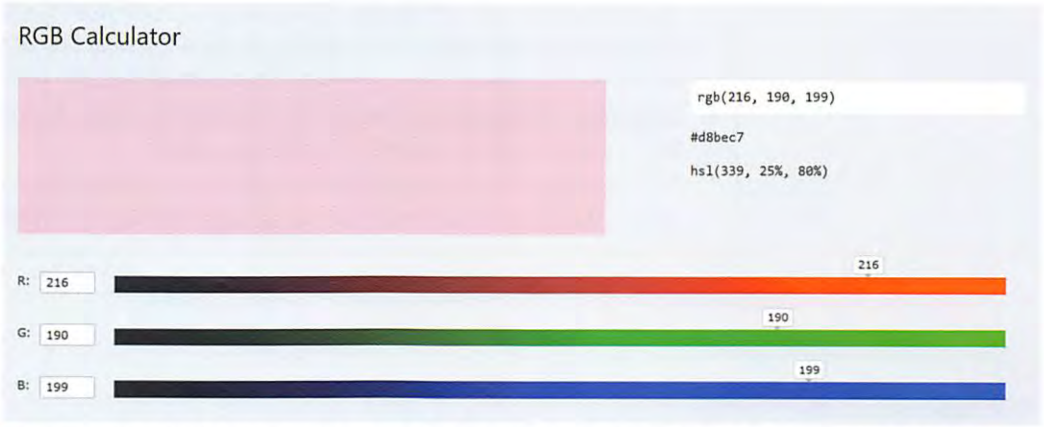
On the left is a high-resolution image. If we zoom in to the dress on this image, we can see the breakdown of the individual pixels and the values of the distinct colour channels. When working with graphics, these values are often shown in hexadecimal. If we take the top left pixel of the dress as an example:

R: 216, G: 190, B: 199 = #d8bec7



R 236 G 190 B 201	R 233 G 195 B 196	R 212 G 145 B 92	R 250 G 167 B 137	R 232 G 170 B 119	R 125 G 164 B 163	R 41 G 84 B 128	R 128 G 140 B 149	R 116 G 123 B 116	R 39 G 84 B 66	R 93 G 84 B 47	R 158 G 152 B 128
R 227 G 162 B 183	R 238 G 203 B 202	R 246 G 183 B 101	R 236 G 159 B 112	R 220 G 171 B 78	R 59 G 139 B 125	R 13 G 57 B 89	R 85 G 89 B 71	R 157 G 160 B 134	R 95 G 113 B 90	R 165 G 141 B 130	R 225 G 188 B 192
R 224 G 172 B 191	R 229 G 189 B 196	R 208 G 145 B 78	R 210 G 152 B 51	R 131 G 141 B 71	R 84 G 120 B 105	R 135 G 139 B 106	R 112 G 127 B 102	R 80 G 68 B 46	R 130 G 125 B 92	R 202 G 164 B 172	R 214 G 148 B 157
R 235 G 196 B 218	R 219 G 153 B 177	R 158 G 75 B 83	R 129 G 115 B 61	R 187 G 226 B 211	R 72 G 119 B 102	R 203 G 176 B 156	R 190 G 154 B 139	R 195 G 152 B 102	R 150 G 148 B 137	R 170 G 130 B 136	R 163 G 101 B 88
R 191 G 136 B 151	R 215 G 158 B 177	R 201 G 115 B 149	R 72 G 107 B 84	R 103 G 145 B 121	R 165 G 144 B 150	R 198 G 121 B 154	R 206 G 119 B 156	R 194 G 180 B 186	R 143 G 154 B 154	R 203 G 171 B 184	R 211 G 144 B 149
R 209 G 164 B 181	R 197 G 92 B 117	R 192 G 106 B 134	R 23 G 62 B 49	R 115 G 160 B 125	R 192 G 80 B 115	R 222 G 119 B 152	R 212 G 129 B 164	R 193 G 170 B 190	R 87 G 107 B 100	R 202 G 164 B 157	R 214 G 162 B 160
R 208 G 129 B 156	R 206 G 94 B 120	R 195 G 109 B 133	R 43 G 120 B 105	R 91 G 111 B 79	R 234 G 138 B 170	R 190 G 97 B 96	R 202 G 88 B 121	R 202 G 175 B 189	R 168 G 150 B 142	R 205 G 126 B 78	R 185 G 128 B 98
R 235 G 139 B 170	R 208 G 112 B 141	R 210 G 152 B 183	R 38 G 100 B 87	R 143 G 72 B 85	R 222 G 102 B 138	R 236 G 164 B 177	R 228 G 134 B 162	R 47 G 61 B 51	R 164 G 125 B 95	R 204 G 107 B 71	R 168 G 104 B 76
R 136 G 92 B 97	R 207 G 106 B 141	R 201 G 179 B 189	R 13 G 67 B 55	R 163 G 99 B 124	R 170 G 54 B 90	R 182 G 75 B 110	R 200 G 146 B 165	R 31 G 55 B 33	R 152 G 88 B 48	R 172 G 79 B 60	R 197 G 149 B 117

■ A zoomed-in area of the image above, showing the value of each pixel – created using www.csfieldguide.org.nz/en/interactives/pixel-viewer



RGB Calculator

rgb(216, 190, 199)

#d8bec7

hsl(339, 25%, 80%)

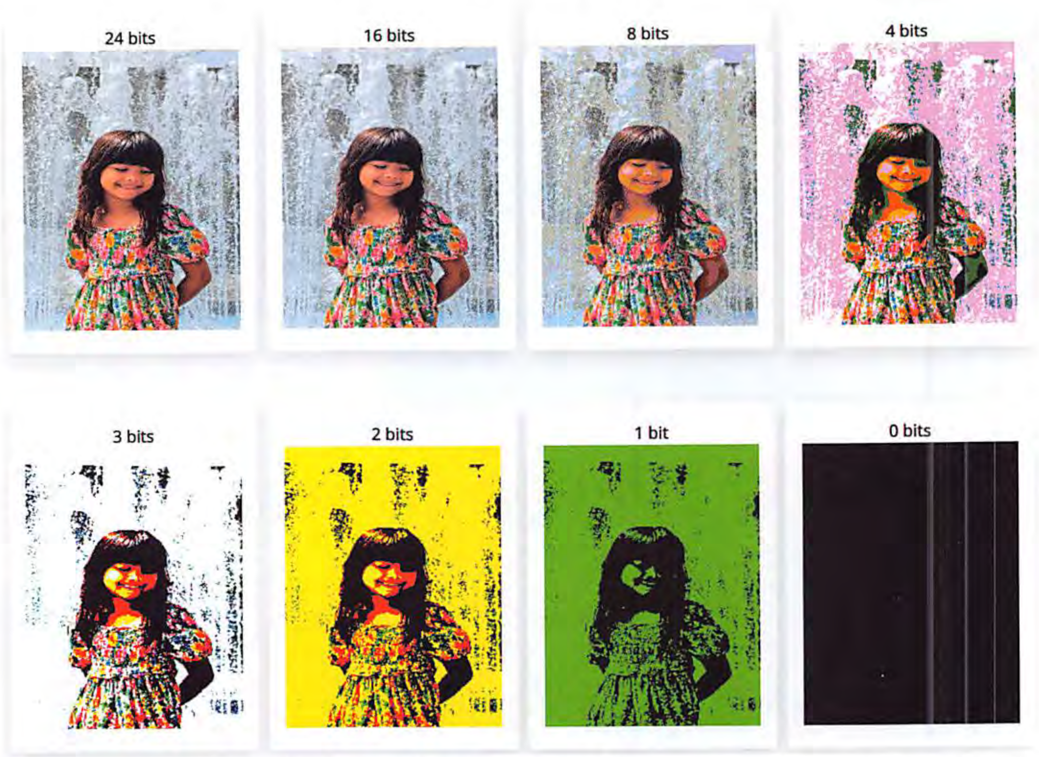
R: 216

G: 190

B: 199

■ The colour values for the top left pixel of the dress in the photo – created using www.w3schools.com/colors/colors_rgb.asp

We can also see the impact of lower colour depths on the same image:



■ The same image using multiple colour depths: 24 bits to 0 bits – created using www.csfieldguide.org.nz/en/interactives/image-bit-comparer

REVIEW QUESTIONS

1 A bitmap image uses a colour depth of 3 bits, allowing for eight distinct colours.

How many bits are needed to represent the colours if the bitmap image uses 32 distinct colours?

2 Raj is creating a bitmap graphic for a game. The image dimensions are 10 pixels wide and 12 pixels tall.

How many pixels are there in total in the image?

3 Alice is organizing her digital artwork collection that she has created over the years.

While transferring her artwork files to a new cloud storage service, she notices that each file is larger than she anticipated. This is because, aside from the actual image data, the file includes extra information necessary for accurate reproduction of the image. What is this additional information, which contains details about the pixel data, called?

4 Determine the storage capacity needed for a bitmap image with dimensions of 800 × 600 pixels that supports 512 different colours.

Then, calculate the file size in kilobytes (kB) if the file metadata occupies an additional 25 per cent of the space. Present your answer as a real number, including the decimal values.

PROGRAMMING EXERCISES

Here are some fun ways to explore images in more depth using Python or Java.

1 Extract and print RGB values.

Java

```java
import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
public class ImageToRGB {
    public static void main(String[] args) {
        try {
            // Load the image
            BufferedImage image = ImageIO.read(new File("sample_image.jpg"));
            // Get image dimensions
            int width = image.getWidth();
            int height = image.getHeight();
            // Loop through each pixel
            for (int y = 0; y < height; y++) {
                for (int x = 0; x < width; x++) {
                    // Get the RGB value of the pixel
                    int pixel = image.getRGB(x, y);
                    Color color = new Color(pixel);
                    // Extract the red, green and blue components
                    int red = color.getRed();
                    int green = color.getGreen();
                    int blue = color.getBlue();
                    // Print the RGB values
                    System.out.println("Pixel at (" + x + ", " + y + "): R=" +
                    red + ", G=" + green + ", B=" + blue);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

For Python, you need to install the Pillow library first. Run this command in your terminal to install the necessary libraries:

```
pip install pillow
```

Python

```
pip install pillow
Use this code to access the RGB values for each pixel
from PIL import Image
# Load the image
image = Image.open("sample_image.jpg")
# Convert the image to RGB mode
image = image.convert("RGB")
# Get the image dimensions
width, height = image.size
# Extract and print RGB values
for y in range(height):
    for x in range(width):
        pixel = image.getpixel((x, y))
        red, green, blue = pixel
        print(f"Pixel at ({x}, {y}): R={red}, G={green}, B={blue}")
```

2 Apply a grayscale filter.

Warning:

This code processes the image pixel by pixel, which means it iterates through every pixel in the image to apply the grayscale filter. For very large images (e.g. high-resolution photos), this process can be computationally intensive and take a significant amount of time to complete. Consider testing this code on smaller images first (e.g. 100x100 pixels) to observe its behaviour before applying it to larger files.

Python

```
from PIL import Image
# Load the image
image = Image.open("sample_image.jpg")
# Convert the image to RGB mode
image = image.convert('RGB')
# Get the image dimensions
width, height = image.size
# Create a new image to store the grayscale result
grayscale_image = Image.new("RGB", (width, height))
# Apply a grayscale filter
for y in range(height):
    for x in range(width):
        pixel = image.getpixel((x, y))
        red, green, blue = pixel
        grayscale = int(0.3 * red + 0.59 * green + 0.11 * blue)
        grayscale_image.putpixel((x, y), (grayscale, grayscale, grayscale))
# Save the grayscale image
grayscale_image.save("grayscale_image.jpg")
```

Java

```
import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
public class ImageToGrayScale {
    public static void main(String[] args) {
        try {
            // Load the image
            BufferedImage image = ImageIO.read(new File("sample_image.jpg"));
            // Get image dimensions
            int width = image.getWidth();
            int height = image.getHeight();
            // Create a new image to store the grayscale result
            BufferedImage grayscaleImage = new BufferedImage(width, height,
            BufferedImage.TYPE_INT_RGB);
            // Apply a grayscale filter
            for (int y = 0; y < height; y++) {
                for (int x = 0; x < width; x++) {
                    // Get the RGB value of the pixel
                    int pixel = image.getRGB(x, y);
                    Color color = new Color(pixel);
                    // Extract the red, green and blue components
                    int red = color.getRed();
                    int green = color.getGreen();
                    int blue = color.getBlue();
                    // Compute the grayscale value
                    int grayscale = (int) (0.3 * red + 0.59 * green + 0.11 *
                    blue);
                    // Create a new Color object with the grayscale value
                    Color grayColor = new Color(grayscale, grayscale, grayscale);
                    // Set the new pixel value in the grayscale image
                    grayscaleImage.setRGB(x, y, grayColor.getRGB());
                }
            }
            // Save the grayscale image
            ImageIO.write(grayscaleImage, "jpg", new File("grayscale_image_
            java.jpg"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3 After studying how the grayscale filter works, are you now able to create your own unique filters?

Audio

Audio in its analogue form is a continuous signal that represents sound waves through variations of air pressure. These sound waves can be captured through input devices, such as microphones, which convert the sound waves into a digital signal, which is stored as binary. This process involves several steps:

◆ **Amplitude:** the magnitude of change in a sound wave, representing the loudness or intensity of the sound.

◆ **Sampling:** the process of converting a continuous analogue signal into a series of discrete digital values by measuring the signal's amplitude at regular intervals.

◆ **kHz (kilohertz):** a unit of frequency equal to 1000 cycles per second, commonly used to measure the sampling rate of audio signals.

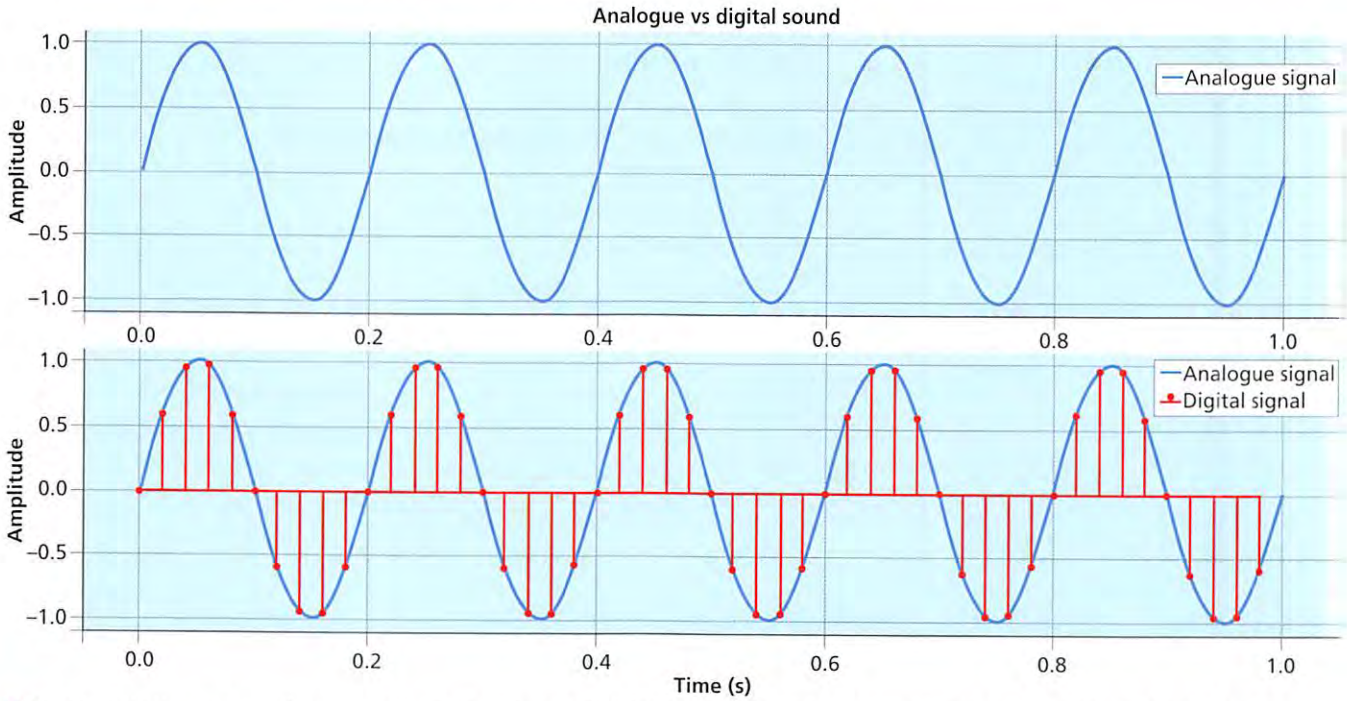
Analogue-to-digital conversion (ADC)

Sound is a continuous analogue signal. An ADC samples the **amplitude** (loudness) of the sound at discrete intervals in a process known as **sampling**. The rate at which this happens is measured in Hertz (Hz) – the higher the Hertz, the more samples are recorded per second. CD-quality audio uses 44.1 **kHz**, but professional quality audio is sampled at 48 kHz.

This sample is then stored and represented as a numerical value in binary. The precision is determined by the bit depth. The larger the bit depth, the more possible values that can be used to describe the sample. For example, the bit depth of CD-quality sound is 16 bit, which gives 2^{16}, or 65,536, values. Professional audio, which uses 24 bit, has 2^{24}, or 16,777,216, values.

A single second of a 44.1 kHz, 16-bit **stereo** (meaning two channels) audio has:

- 44,100 samples per second
- each sample represented by 16 bits
- a total storage need per second of 44,100 samples / second × 16 bits / sample × 2 channels = 1,411,200 bits per second, or 176,400 bytes per second.



■ The blue continuous waveform represents an analogue signal, which is a smooth and continuous representation of sound. The digital signal consists of discrete samples taken at regular intervals (sampling rate), illustrating how the continuous analogue signal is converted into a series of discrete points in digital form.

Storage formats

There are many different types of file formats for storing audio. The most common are WAV, AIFF, MP3 and FLAC. They mainly differ by whether they are compressed or uncompressed. Uncompressed formats store the raw binary data, whereas compressed formats use algorithms to reduce the file size for storage or transmission. Just like with image compression, audio

◆ **Stereo:** a method of sound reproduction that uses two or more audio channels to create the perception of sound coming from different directions, enhancing the sense of spatial depth and dimension.

compression attempts to reduce the file size by removing parts of the audio signal that are less noticeable to human senses, in this case the ears. There are both lossy and lossless types of compression used with audio. Lossless algorithms compress the data without any loss of quality, whereas lossy algorithms permanently remove audio that is less noticeable to the human ear on the recording.

- WAV (Waveform Audio File Format): uncompressed
- AIFF (Audio Interchange File Format): uncompressed
- MP3 (MPEG Audio Layer III): compressed (lossy)
- FLAC (Free Lossless Audio Codec): compressed (lossless)

REVIEW QUESTIONS

1 What is the main difference between an analogue signal and a digital signal in the context of audio?
2 What is the process of converting an analogue audio signal into a digital signal called, and what does it involve?
3 Calculate the storage needed per minute for a 44.1 kHz, 16-bit stereo audio file.
4 Explain the difference between lossy and lossless audio compression and give an example of each type of format.

PROGRAMMING EXERCISE

Explore audio files further using the code below. This will allow you to analyse the amplitude of any MP3 file.
You will need to install the following libraries:

- soundfile
- numpy
- matplotlib
- scipy.

Run this command in your terminal:

```
pip install soundfile numpy matplotlib scipy
```

Python

```python
import soundfile as sf
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft
# Load the audio file
samples, sample_rate = sf.read("name_of_file.mp3")
# If stereo, select one channel
if samples.ndim > 1:
    samples = samples[:, 0]
# Visualize the waveform
plt.figure(figsize=(12, 6))
plt.plot(samples)
plt.title("Audio Waveform")
plt.xlabel("Sample Index")
plt.ylabel("Amplitude")
plt.show()
# Perform FFT
spectrum = fft(samples)
frequencies = np.fft.fftfreq(len(spectrum), 1 / sample_rate)
plt.figure(figsize=(12, 6))
plt.plot(frequencies[:len(frequencies)//2],
np.abs(spectrum[:len(spectrum)//2]))
plt.title("Audio Spectrum")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.show()
```

Video

Videos are made up of various components that are all contained within an encapsulated container format such as MP4, MKV or AVI. The components are:

- frames (visual data)
- audio tracks
- metadata
- subtitles and closed captions.

Audio is stored as described in the Audio section above, and metadata and subtitles are stored as text, so this section will focus only on how the video data is stored.

Video is essentially stored as a sequence of still images, otherwise known as "frames". When played in quick succession (usually 24 to 60 frames per second), these frames create the illusion of motion. This is very similar to the technique you may have used to create a flipbook. The frames are stored and encoded in binary format, utilizing various techniques to optimize space and ensure efficient playback.

Digital video playback is similar to a flipbook: a number of images that are shown quickly, creating the illusion of motion

Frames

In their raw form, frames are stored the same as images, with each pixel having a value that can be represented using a colour model such as RGB. To improve colour efficiency, frames are often converted from the RGB colour model to a different one such as YUV. This helps with compression, as this colour model emphasizes luminance (brightness), which the human eye is more sensitive to than changes in colour detail.

However, we cannot store frames in the same way as we store photos because, in this format, they would be too large. They need to be compressed, and there are two main techniques used for this: spatial (intraframe) and temporal (interframe).

Compression techniques

Spatial compression is particularly effective and commonly used for video that has significant detail variation in each frame. It reduces file size by eliminating redundant information within each frame, such as colour depth or detail levels. This approach is important for videos with a lot of detail that may change significantly between frames, such as animations, nature documentaries and live news broadcasts.

Temporal compression is particularly effective and commonly used for video that has consistent motion across frames. It reduces file size by eliminating redundant information between consecutive frames, capturing only the changes or movements from one frame to the next. As a predictive compression technique, it predicts frame content based on the preceding and sometimes following frames, only storing the differences. This approach is important for videos with a lot of detail that may change significantly between frames, such as animations, nature documentaries and live news broadcasts.

REVIEW QUESTIONS

1 What is the role of frames in a video, and how do they create the illusion of motion?
2 Explain the difference between spatial compression and temporal compression in video storage.
3 Describe how converting video frames from the RGB colour model to the YUV colour model can improve compression efficiency.
4 Calculate the total storage needed for a 10-minute video with a frame rate of 30 frames per second, using 24-bit colour depth and a resolution of 1920×1080 pixels. Assume no compression.

Different binary methods for storing integers

Unsigned binary

This is the system we covered in Section A1.2.1. This system only represents positive integers using straightforward binary digits (0s and 1s).

Signed binary

This includes methods for representing both positive and negative integers.

Two's complement:

Two's complement is a method for representing signed integers in binary, where the most significant bit (MSB) indicates the sign (0 for positive, 1 for negative). To convert a positive binary number to its negative counterpart in two's complement, you first invert all the bits (change 0s to 1s and 1s to 0s) and then add 1 to the least significant bit (LSB).

For example:

00000101 = +5

Invert the bits

11111010

Add 1

11111011 = –5

However, a limitation of two's complement is that, in an 8-bit system, it reduces the range of representable numbers. Instead of being able to represent 0 to 255, as with unsigned binary, two's complement allows for numbers ranging from –128 to +127, effectively halving the number of positive values that can be represented.

One's complement:

One's complement is a binary representation method for signed integers, where the most significant bit (MSB) indicates the sign (0 for positive, 1 for negative). To obtain the one's complement of a positive number, you simply invert all the bits (change 0s to 1s and 1s to 0s).

For example:

00000101 = +5

11111010 = –5

Unlike two's complement, one's complement has two representations of the number zero: positive zero (00000000) and negative zero (11111111). This is one of its main limitations and can cause confusion when using arithmetic and logical operations. This means two's complement is normally the preferred system to use. Similarly to two's complement, one's complement also has a limited range, representing numbers from –127 to +127.

Sign-magnitude:

Sign-magnitude is a binary representation method for signed integers where the most significant bit (MSB) serves as the sign indicator, with 0 representing positive numbers and 1 representing negative numbers. The remaining bits represent the magnitude of the number, like how unsigned binary numbers work.

For example:

00000101 = +5

10000101 = –5

This system also has two representations for zero: positive zero (00000000) and negative zero (10000000). It also has the same range as one's complement, from –127 to +127. It is a simple system but, like one's complement, is less efficient compared to two's complement.

Binary-coded decimal

Binary-coded decimal (BCD) is a method of representing decimal numbers where each digit of the decimal number is encoded separately into its own binary form. Unlike pure binary representation, which converts the entire decimal number into a single binary sequence, BCD assigns a 4-bit binary code to each decimal digit (0–9).

For example:

0100 0101 = 45

as 0100 represents 4, and 0101 represents 5

This system is useful where exact decimal representation is crucial, such as financial applications or digital clocks, as it avoids the rounding errors that can occur in other systems. However, due to using four bits per digit, more bits are required to store numbers, making it less space efficient than pure binary representations. Calculations using BCD are also more complex as they require additional steps to handle carry and overflow, so they are not good choices for general-purpose computing.

Gray code (reflected binary code)

Gray code is a binary system where two successive values are only allowed to differ by one bit. That makes this system particularly useful in situations where data integrity during transitions is important. An example system is a robotic arm where we want to monitor its position. As the arm rotates, the rotary encoder generates a sequence of binary outputs corresponding to the arm's angle. If the encoder used standard binary code, small mechanical vibrations or inaccuracies could cause multiple bits to change simultaneously, leading to incorrect readings. However, by using Gray code, the risk of these transition errors is minimized.

■ Comparison of Gray code to standard binary for the numbers 0–7

Numbers	Standard binary	Gray code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Excess-N (biased representation)

Excess-N is a system where a fixed bias (N) is added to the actual value to form an encoded value, and you subtract this bias to decode it. This is used to make all signed integers appear as non-negative binary numbers to allow for easier comparisons and arithmetic operations.

For example, with Excess-3:

The decimal number 2 would be encoded as:

2 + 3 = 5

0101

The decimal number –2 would be encoded as:

–2 + 3 = 1

0001

In an 8-bit system, Excess-127 is often used, which adds 127 to encode an 8-bit number and subtracts 127 to decode it. If you consider trying to order a set of signed binary numbers, this can be difficult as the negative numbers are larger binary numbers than the positive.

For example, take 127 and –127 (using sign-magnitude):

Pre-encoded numbers:

01111111 = 127

10000001 = –127

When we encode these numbers with Excess-127, the positive numbers now appear larger than the negative numbers, making them easier to put in order:

Encoded numbers (Excess-127):

127 + 127 = 254

11111110

–127 + 127 = 0

00000000

After this process has been completed, we decode the numbers again to return them to their original form:

Decoded numbers (Excess-127):

254 – 127 = 127

11111110

0 – 127 = –127

00000000

Fixed-point representation

Fixed-point representation is a method used to store real numbers (numbers with fractional parts) in binary by fixing the position of the binary point. In a fixed-point system, the binary point is placed at a predetermined position, either between certain bits or at a specific location in the binary sequence. This allows for a straightforward representation of fractional numbers, though with some trade-offs in terms of precision and range.

For example, if we want to represent 5.25 in an 8-bit system where four bits represent the integer and four bits represent the fractional part:

Integer part (four bits): 0101 = 5

Fractional part (four bits): 0100 = 0.25

Combined: 0101.0100

Note: Binary fractions are used for the fractional part, where the first bit to the right represents $\frac{1}{2}$, the second bit represents $\frac{1}{4}$, the third $\frac{1}{8}$ and the fourth $\frac{1}{16}$. So, in the example above, we have 0 $\frac{1}{2}$, 1 $\frac{1}{4}$, 0 $\frac{1}{8}$ and 0 $\frac{1}{16}$.

The number of bits assigned in this system limits the range and precision. In this example, with a 4-bit signed integer, we only have the range of –8 to 7.9375, with the smallest representable value being 0.0625 ($\frac{1}{16}$). This means this system is unable to handle very large or very small numbers effectively. However, it is a simpler and faster system compared to floating-point arithmetic (see below), and does not require any complex operations to adjust the position of the binary point.

Floating-point representation

Floating-point representation is a method used to represent real numbers that can have a very large range or fractional parts. It does this by storing numbers in a format that includes a sign, an exponent and a mantissa (or significand). This format allows computers to efficiently handle very large numbers, very small numbers and numbers with fractional parts, all with a reasonable degree of precision.

Using the IEEE 754 standard for single-precision floating-point numbers (which uses 32 bits):

1. Sign bit (one bit):
   The sign bit determines whether the number is positive or negative.
2. Exponent (eight bits):
   This is used to scale the number by a power of two and is stored using the Excess-127 system in its "biased" form; in other words, 127 is added to the actual exponent value.
3. Mantissa (23 bits):
   This represents the significant digits of the number. The mantissa does not store leading ones (in normalized form).