ASGHINLABS ACADEMY

ASGHINLABS

Computer Science

LOWER SIXTH, COMPLETE LESSON NOTES

By **Mr Ayemelong S. Ghislain**
Contact: **670922913**
www.asghinlabs.com

Academic Year **2026–2027**

Computer Science

Lower Sixth Complete Lesson Notes

Academic Calendar: 2026–2027 **Class:** Lower Sixth (All) **Based on the MINESEC Harmonised Progression Sheet, 122 Lessons**

These notes follow the official progression sheet chapter by chapter. Related "Part 1 / Part 2" lessons and Integration Activities have been combined into single topic sections so that each note is complete and ready to teach or revise from.

By **Mr Ayemelong S. Ghislain**

Contact: **670922913**

Table of Contents

TERM ONE

Chapter 1: Exploring AI Concepts and Emerging Technologies

Chapter Overview

This chapter introduces students to Artificial Intelligence (AI), one of the most important and rapidly developing fields in computer science today. Students will learn what AI is, how it is categorised, how Machine Learning works, how AI systems are developed, and why data is central to everything AI does. The chapter also examines the social, ethical, and practical implications of AI in society.

Learning Objectives

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

1. Define Artificial Intelligence and explain why it has become important in modern society.
2. Distinguish between the categories of AI by capability (Narrow, General, Super) and by function (Reactive, Limited Memory, Theory of Mind, Self-Aware).
3. Identify everyday examples of AI and emerging technologies linked to AI.
4. Define Machine Learning and distinguish it from traditional programming.
5. Explain the three main types of Machine Learning (supervised, unsupervised, reinforcement) with examples.
6. Describe key AI techniques including neural networks, deep learning, NLP, computer vision, and expert systems.
7. Outline the stages in developing an AI system.
8. Explain why data is important in AI, including issues of bias, privacy, and ethics.

9. Describe methods of data collection and the data analysis process.
10. Discuss the benefits and concerns/limitations of AI.

Key Terms

Term	**Definition**
Artificial Intelligence (AI)	Branch of computer science building systems that perform tasks normally requiring human intelligence
Narrow AI	AI designed for one specific task; all AI today is narrow AI
General AI (AGI)	Hypothetical AI with human-level intelligence across any task
Super AI	Hypothetical AI surpassing human intelligence in all respects
Reactive machine	AI that responds to current input only, no memory
Limited memory	AI that uses past data briefly to make decisions
Machine Learning (ML)	Subset of AI where systems learn patterns from data rather than being explicitly programmed

Term	Definition
Supervised learning	ML trained on labelled data (input paired with correct output)
Unsupervised learning	ML that finds patterns in unlabelled data on its own
Reinforcement learning	ML where an agent learns by trial and error, receiving rewards or penalties
Neural network	Layered structure loosely inspired by the brain, used for pattern recognition
Deep learning	Neural networks with many layers, powering image recognition and NLP
Natural Language Processing (NLP)	Enables computers to understand and generate human language
Computer Vision	Enables computers to interpret images and video
Expert System	Rule-based system that mimics a human expert's decision-making in a narrow domain
Training data	Data used to teach a machine learning model

Term	Definition
Test data	Data used to check a model's accuracy on unseen examples
Bias (in AI)	Systematic unfairness learned from biased training data
Big Data	Extremely large datasets that require special tools to analyse
IoT (Internet of Things)	Network of everyday objects with sensors connected to the internet
Primary data	Data collected directly for the purpose at hand
Secondary data	Data already collected by someone else
Data cleaning	Removing errors, duplicates, and missing values from data

1.1 Introduction to Artificial Intelligence

Definition: Artificial Intelligence (AI) is the branch of computer science concerned with building systems (hardware/software) that can perform tasks which normally require human intelligence, reasoning, learning, perception, problem-solving, and decision-making.

In simpler terms: AI is about making machines that can "think" and "learn" in ways that resemble human intelligence.

Why AI Matters Now

AI has existed as a concept since the 1950s, but it has only become practical in everyday life in the last two decades. Three factors made this possible:

1. Cheaper and more powerful computing power, modern processors and GPUs can perform billions of calculations per second at low cost.
2. Huge amounts of digital data ("Big Data"), billions of people generate data every day through phones, websites, sensors, and transactions. AI needs this data to learn.
3. Improved algorithms, especially deep learning, which has become far more effective since around 2012.

These three factors together have made AI practical in everyday devices: phones, cars, search engines, hospitals, farms, and homes.

Categories of AI (by Capability)

Category	**Description**	**Example**	**Exists today?**
Narrow / Weak AI	Designed for one specific task	Chess engine, spam filter, Siri, Google Translate	Yes, all AI in use today is narrow AI
General AI (AGI)	Hypothetical AI with human-level intelligence across any task	A machine that could learn and reason like a human across all domains	No, does not yet exist
Super AI	Hypothetical AI surpassing human	A machine far smarter than the best human in	No, theoretical only

Category	Description	Example	Exists today?
	intelligence in all respects	every field	

Important point: Despite what science fiction suggests, all AI today is narrow AI. Even the most advanced systems (like ChatGPT or self-driving cars) are designed for specific tasks and cannot generalize beyond them.

Categories of AI (by Function)

Category	Description	Example
Reactive machines	Respond to current input only; no memory of past events	IBM's Deep Blue (chess)
Limited memory	Use past data briefly to make decisions	Self-driving cars
Theory of mind	Would understand emotions, beliefs, and intentions of others (research stage)	—
Self-aware	Would have consciousness and a sense of self (theoretical)	—

Everyday Examples of AI

- **Voice assistants:** Siri, Alexa, Google Assistant
- **Recommendation systems:** Netflix, YouTube, TikTok, Amazon
- **Facial recognition:** phone unlock, security cameras, passport control
- **Spam filters:** email classification
- **Chatbots:** customer service on websites
- **GPS route optimization:** Google Maps, Waze
- **Medical diagnosis support:** X-ray and scan analysis
- **Agricultural drones:** crop monitoring, precision farming
- **Fraud detection:** banking and mobile money systems
- **Language translation:** Google Translate, DeepL

Emerging Technologies Linked to AI

Technology	**Description**
Internet of Things (IoT)	**Everyday objects with sensors connected to the internet**
Robotics	**Machines that perform physical tasks, often guided by AI**
Big Data analytics	**Extracting patterns from massive datasets**

Technology	Description
Cloud computing	**On-demand computing power and storage over the internet**
Autonomous vehicles	**Self-driving cars, drones, autonomous ships**
Blockchain	**Decentralised, tamper-resistant record-keeping**
Virtual/Augmented Reality (VR/AR)	**Immersive digital experiences**

How these technologies work together: IoT sensors on a farm collect soil and weather data → cloud computing stores it → AI analyses it → the farmer receives recommendations on when to water or harvest. This is often called the IoT–Cloud–AI pipeline.

Benefits of AI

- **Increased efficiency and speed**: AI can process data far faster than humans.
- **24/7 availability**: AI systems do not get tired or need breaks.
- **Reduced human error** in repetitive tasks.
- **Ability to process huge datasets**: beyond human capacity.
- **Improved accuracy** in prediction and diagnosis.
- **Cost savings** over time in automation.

Concerns / Limitations of AI

- **Job displacement**: automation of routine work (manufacturing, customer service, driving).
- **Bias in training data**: leading to unfair decisions (e.g., biased hiring or lending).
- **Lack of transparency**: the "black box" problem: we can't always explain why AI decided something.
- **Privacy risks**: AI often uses personal data.
- **High development cost**: building and maintaining AI systems is expensive.
- **Dependence on quality data**: poor data → poor AI.
- **Ethical questions**: autonomous decision-making (e.g., self-driving car accidents, autonomous weapons).
- **Security risks**: AI can be used for deepfakes, phishing, and cyberattacks.

1.2 AI Techniques and Machine Learning

What is Machine Learning?

Machine Learning (ML) is a subset of AI in which a system learns patterns from data rather than being explicitly programmed with fixed rules.

In traditional programming, a human writes the rules. In machine learning, the system discovers the rules itself by studying examples.

Traditional Programming vs Machine Learning

Traditional Programming	**Machine Learning**
Programmer writes explicit rules	System learns rules from data

Traditional Programming	Machine Learning
Input + Program → Output	Input + Output (training data) → Program (model)
Poor at handling unseen/complex patterns	Good at recognising complex patterns
Rules are transparent and editable	Rules are learned and often opaque
Works well for well-defined problems	Works well for pattern-recognition problems

Example: To filter spam traditionally, you'd write rules like "if the email contains 'free money', mark as spam." In ML, you'd show the system thousands of spam and non-spam emails, and it would learn the patterns itself.

Types of Machine Learning

1. Supervised Learning

The model is trained on labelled data, input paired with the correct output.

- **Classification**, predicting a category (spam / not spam; cat / dog; pass / fail).
- **Regression,** predicting a continuous value (house price, temperature, salary).

Example: A model trained on 10,000 emails labelled "spam" or "not spam" learns to classify new emails.

2. Unsupervised Learning

The model finds patterns in unlabelled data on its own**.**

- **Clustering:** grouping similar items (customers by buying behaviour).
- **Association:** finding rules (people who buy bread also buy butter).

Example: A supermarket's loyalty-card data is clustered into customer types without any pre-set labels.

3. Reinforcement Learning

An agent learns by trial and error, receiving rewards or penalties for actions.

Example: A robot learning to walk; a game-playing AI (AlphaGo); a self-driving car learning to stay in lane.

Comparison table:

Type	**Data**	**Goal**	**Example**
Supervised	Labelled	Predict output for new input	Spam filter, price prediction
Unsupervised	Unlabeled	Find hidden patterns	Customer segmentation
Reinforcement	Interaction + rewards	Learn optimal actions	Robot walking, game AI

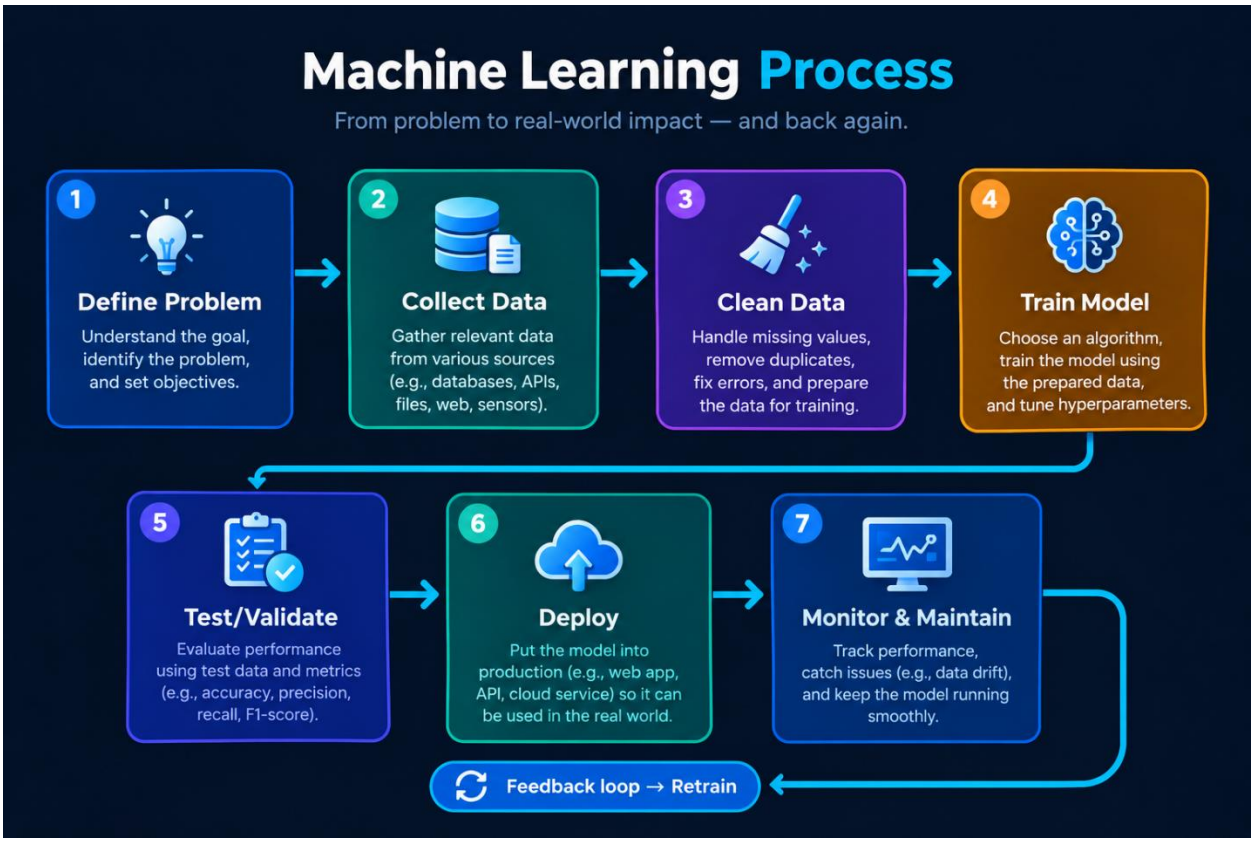
Key AI Techniques

Technique	Description	Example Use
Neural Networks	Layered structures loosely inspired by the brain	Pattern recognition
Deep Learning	Neural networks with many layers	Image recognition, NLP
Natural Language Processing (NLP)	Understand and generate human language	Chatbots, translation
Computer Vision	Interpret images and video	Facial recognition, self-driving cars
Expert Systems	Rule-based systems mimicking a human expert	Medical diagnosis in a narrow field
Genetic Algorithms	Simulate evolution to optimise solutions	Scheduling, design optimisation
Speech Recognition	Convert spoken words into text	Voice assistants, dictation

1.3 Developing AI Systems and the Importance of Data

Stages in Developing an AI System

1. **Define the problem:** what task should the AI perform? What does success look like?
2. **Collect data:** gather relevant, sufficient data.
3. **Prepare / clean data:** remove errors, duplicates, outliers, and irrelevant data.
4. **Choose and train a model:** select an algorithm and let it learn from the training data.
5. **Test / validate the model:** check accuracy using data the model has never seen before (test set).
6. **Deploy**: put the system into real use.
7. **Monitor and maintain:** retrain as new data becomes available; fix errors; watch for drift.



Why Data is Important in AI

Data is the "fuel" of AI, an ML model is only as good as the data used to train it.

Quality	Explanation
Quantity	**More data generally improves accuracy (up to a point).**
Quality	**Accurate, relevant, and representative data avoids bias and errors.**
Variety	**Diverse data helps the model generalise to new/unseen situations.**

The GIGO principle: Garbage In, Garbage Out. If training data is poor, the AI's decisions will be poor.

Bias in AI

If training data reflects existing human biases (e.g., historical hiring data favoring one gender), the AI system will learn and repeat that bias.

Example: A recruitment AI trained on 10 years of hiring data from a company that mostly hired men may learn to reject female applicants, not because it was programmed to, but because it learned from biased history.

Why this matters:

- AI decisions affect real people (jobs, loans, healthcare, policing).
- Bias can be hidden and hard to detect.
- This is why data must be checked and cleaned carefully, and outcomes tested for fairness.

Data Privacy and Ethics

Because AI systems often use personal data, developers must follow data protection principles:

- Collect only what is needed (data minimization).
- Secure it (encryption, access control).
- Get consent where required.
- Allow users to access and delete their data.
- Be transparent about how data is used.
- Do not use data for purposes users did not agree to.

1.4 Data Collection and Data Analysis

Data Collection Methods

Method	Type	Description / Example
Surveys / questionnaires	Primary	Asking people directly
Interviews	Primary	In-depth one-to-one questioning
Sensors (IoT)	Primary	Automatic environmental data (temperature, motion)
Direct observation	Primary	Watching and recording behaviour
Web forms	Primary	Online data entry

Method	Type	Description / Example
Databases	Secondary	Using data already collected by an organisation
Published reports	Secondary	Government, NGO, or research reports
Open government data	Secondary	Public datasets
Web scraping	Secondary	Automatically extracting data from websites

Primary vs Secondary data:

Primary Data	Secondary Data
Collected directly by you	Collected by someone else
Specific to your purpose	May need adapting
Often more expensive/time-consuming	Often cheaper and faster
More control over quality	Quality may be unknown

Qualities of Good Data

- **Accurate**: free from errors.
- **Complete**: no missing values that would skew results.
- **Relevant**: actually, related to the question being asked.
- **Up-to-date**: timely for the purpose.
- **Collected ethically**: with consent/permission where personal data is involved.

Data Analysis Process

1. **Data cleaning**: removing errors, duplicates, and missing values.
2. **Data organization**: sorting/structuring data (spreadsheets, databases).
3. **Data interpretation**: using statistics, charts, and graphs to find patterns and trends.
4. **Drawing conclusions**: using analyzed data to make decisions or predictions.

Common Tools for Data Analysis

Tool	Use
Spreadsheets (Excel, Google Sheets)	Formulas, pivot tables, charts
Database software (MySQL, Access)	Querying structured data with SQL
Python (with Pandas, NumPy, Matplotlib)	Advanced analysis and visualisation
R	Statistical analysis
Tableau / Power BI	Business intelligence dashboards

Review Questions

1. Define Artificial Intelligence and give three reasons why it has become practical in recent years.
2. Distinguish between Narrow AI, General AI, and Super AI. Which exists today?
3. Explain the difference between reactive machines and limited-memory AI, giving an example of each.
4. List five everyday examples of AI and explain what each does.
5. Describe how IoT, cloud computing, and AI work together in a real-world example.
6. State four benefits and four concerns/limitations of AI.
7. Define Machine Learning and explain how it differs from traditional programming.
8. Describe the three types of Machine Learning, giving an example of each.
9. Explain what a neural network is and how deep learning differs from it.
10. List the seven stages in developing an AI system.
11. Explain why data is described as the "fuel" of AI.
12. What is bias in AI? Give an example of how it can occur.
13. State four principles of data privacy and ethics in AI development.
14. Distinguish between primary and secondary data collection, giving two examples of each.
15. List the four steps of the data analysis process.
16. Name three common tools used for data analysis and state what each is used for.

Chapter 2: Description of Computer Systems

Chapter Overview

This chapter introduces students to what a computer system actually is. Rather than seeing a computer as just a box or a screen, students will learn that a computer system is a combination of hardware, software, data, and people working together. The chapter covers the Input–Process–Output–Storage (IPOS) cycle, the major hardware components of a computer, the difference between system software and application software, and basic computer operations such as booting, executing instructions, and multitasking.

Learning Objectives

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

1. Define a computer system and list its four main elements.
2. Describe the Input–Process–Output–Storage (IPOS) cycle with real examples.
3. Identify the major hardware components of a computer and explain the function of each.
4. Distinguish between system software and application software, giving examples of each.
5. Explain what happens during the booting process.
6. Describe how a computer executes instructions.
7. Explain what multitasking is and how the CPU achieves it.
8. Appreciate the relationship between hardware, software, data, and people in a working computer system.

Key Terms

Term	Definition
Computer system	A combination of hardware, software, data, and people working together to process input into output
Hardware	The physical, tangible parts of a computer (e.g., CPU, keyboard, monitor)
Software	Programs and instructions that tell the hardware what to do
Data	Raw, unprocessed facts and figures
Information	Processed data that is meaningful and useful
Input	Data entered into a computer
Processing	Manipulating or transforming data using the CPU
Output	Processed data (information) presented to the user
Storage	Saving data/information for later use

Term	Definition
IPOS cycle	Input → Process → Output → Storage
CPU	Central Processing Unit, the "brain" of the computer
Control Unit (CU)	Part of the CPU that directs operations and decodes instructions
ALU	Arithmetic Logic Unit, performs arithmetic and logic operations
Register	Tiny, very fast storage location inside the CPU
RAM	Random Access Memory, volatile main memory
ROM	Read-Only Memory, non-volatile memory holding startup instructions
Volatile	Loses its contents when power is switched off
Non-volatile	Retains its contents without power
Motherboard	Main circuit board connecting all components

Term	Definition
Bus	Pathway that carries data, addresses, or control signals between components
System software	Software that manages the computer itself (e.g., OS, drivers)
Application software	Software that helps the user perform specific tasks
Operating System (OS)	System software that manages hardware and provides services to programs
Device driver	Software that allows the OS to communicate with a specific hardware device
Utility program	System software that performs maintenance tasks (e.g., antivirus, backup)
Booting	The process of starting up a computer and loading the OS
POST	Power-On Self-Test, a check of hardware during booting
BIOS / UEFI	Firmware that runs at startup to initialise hardware and find the OS

Term	Definition
Multitasking	Running several programs seemingly at once by rapidly switching the CPU between them

2.1 Components of a Computer and Basic Operations

What is a Computer System?

A **computer system** is not just the physical machine. It is made up of **four elements** that work together:

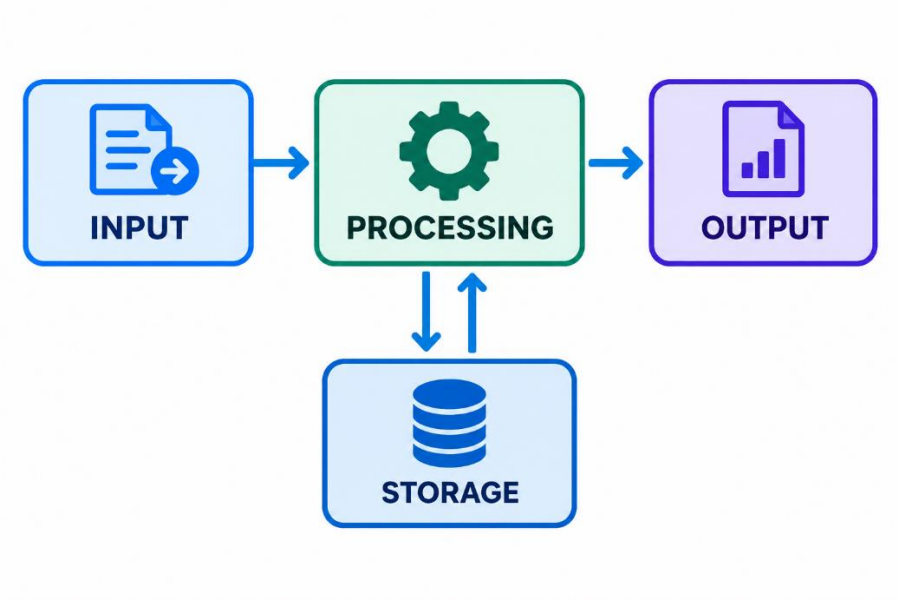
Element	Description	Example
Hardware	The physical, tangible parts	CPU, keyboard, monitor, motherboard
Software	Programs and instructions	Windows, Word, Python, Chrome
Data	Raw facts and figures	Names, numbers, images, sounds
People (Users)	The human beings who use the system	Students, teachers, accountants, programmers

All four are needed. A computer with hardware and software but no data does nothing useful; a computer with data but no people to use it serves no purpose.

The Input–Process–Output–Storage (IPOS) Cycle

Every computer system follows the same basic cycle:

1. **Input**: data is entered into the computer.
2. **Processing**: the CPU manipulates or transforms the data.
3. **Output**: the processed data (information) is presented to the user.
4. **Storage**: data and information are saved for later use.



Example 1: Writing an essay:

- **Input:** Student types the essay on a keyboard.
- **Processing:** The word processor formats, checks spelling, and arranges the text.
- **Output:** The essay appears on the monitor and can be printed.
- **Storage:** The essay is saved to the SSD or cloud.

Example 2: ATM withdrawal:

- **Input:** User inserts card and enters PIN and amount.
- **Processing:** Bank's computer verifies the PIN and checks the balance.

- **Output:** Cash is dispensed and a receipt is printed.
- **Storage:** The transaction is recorded in the bank's database.

Example 3: Taking a photo with a phone:

- **Input:** Light enters the camera lens (sensor captures image).
- **Processing:** The phone's processor compresses and enhances the image.
- **Output:** The photo appears on the screen.
- **Storage:** The photo is saved to internal storage or cloud.

Major Hardware Components

Component	Function
CPU (Central Processing Unit)	The "brain", fetches, decodes, and executes instructions. Contains CU, ALU, and registers.
Main Memory (RAM)	Volatile memory that temporarily holds data and instructions currently in use.
ROM (Read-Only Memory)	Non-volatile memory holding essential startup instructions (BIOS/firmware).
Storage devices	HDD, SSD, USB flash drives, optical discs, for permanent storage.
Input devices	Keyboard, mouse, scanner, webcam, microphone, touchscreen, barcode reader.

Component	Function
Output devices	Monitor, printer, speakers, projector.
Motherboard	The main circuit board connecting all components.
Buses	Pathways (data bus, address bus, control bus) that carry information between components.
Power Supply Unit (PSU)	Converts mains AC to low-voltage DC for internal components.
Cooling system	Fans and heat sinks prevent overheating of CPU and other components.
Expansion cards	Add extra functionality (graphics card, sound card, network card).

Inside the CPU (brief recap, more in Chapter 7):

Sub-component	Function
Control Unit (CU)	Directs operations; decodes instructions; controls timing
ALU	Performs arithmetic (+, −, ×, ÷) and logic (AND, OR, NOT) operations

Sub-component	Function
Registers	Tiny, very fast storage locations (PC, MAR, MDR, CIR, ACC)

Software Categories

Software is divided into two main categories:

1. System Software

System software manages the computer itself and provides a platform for other software to run.

Type	**Description**	**Examples**
Operating System (OS)	Manages hardware, files, memory, and processes; provides a user interface	Windows, Linux, macOS, Android, iOS
Device drivers	Allow the OS to communicate with specific hardware	Printer driver, graphics driver
Utility programs	Perform maintenance and housekeeping tasks	Antivirus, disk cleanup, backup tools, file managers
Firmware	Permanent software stored in ROM	BIOS, UEFI

2. Application Software

Application software helps the user perform specific tasks.

Type	Description	Examples
Word processors	Create and edit text documents	Microsoft Word, Google Docs
Spreadsheets	Calculations and data analysis	Excel, Google Sheets
Presentation software	Create slideshows	PowerPoint, Google Slides
Web browsers	Access the internet	Chrome, Firefox, Edge
Media players	Play audio and video	VLC, Windows Media Player
Games	Entertainment	Various
Graphics software	Create and edit images	Photoshop, GIMP, Canva
Database software	Store and query structured data	MySQL, Access

Comparison table:

System Software	Application Software
Manages the computer itself	Helps the user perform tasks
Runs in the background	Runs when the user opens it

System Software	Application Software
Essential for the computer to work	Optional, installed as needed
Examples: Windows, Linux, drivers	Examples: Word, Chrome, games

Basic Operations of a Computer

1. Booting

Booting is the process of starting up a computer and loading the operating system from storage into RAM.

Types of boot:

- **Cold boot**: starting from a powered-off state.
- **Warm boot**, restarting without powering off (e.g., pressing Ctrl+Alt+Del or choosing "Restart").

Steps in the boot process:

1. **Power on**: the PSU supplies power to components.
2. **POST (Power-On Self-Test)**: the BIOS/UEFI checks that essential hardware (CPU, RAM, keyboard, disk) is working.
3. **BIOS/UEFI runs**: it initialises hardware and looks for a bootable device.
4. **Bootloader loads**: the bootloader finds the OS on the boot device.
5. **OS loads into RAM**: the operating system kernel is loaded into memory.
6. **System services start**: background services and drivers are loaded.
7. **User interface appears**: the login screen or desktop is displayed.

Diagram (text representation):

Power On → POST → BIOS/UEFI → Bootloader → OS loads into RAM

→ System services → User interface

2. Executing Instructions

The CPU executes instructions using the **fetch–decode–execute cycle** (covered in detail in Chapter 7). In brief:

1. **Fetch**: the CPU fetches the next instruction from memory.
2. **Decode**: the Control Unit interprets the instruction.
3. **Execute**: the instruction is carried out (e.g., ALU performs a calculation).

This cycle repeats billions of times per second.

3. Multitasking

Multitasking is running several programs seemingly at once by rapidly switching the CPU between them.

How it works:

- The CPU can only execute one instruction at a time per core.
- The OS switches rapidly between programs (many times per second).
- To the user, it appears that all programs are running simultaneously.
- Modern multi-core CPUs allow true parallel execution of multiple programs.

Example: You can listen to music, browse the web, and have a word processor open, all at the same time. The OS is switching the CPU between these tasks so quickly that you don't notice.

Review Questions

1. Define a computer system and list its four main elements.
2. Explain the IPOS cycle using the example of an ATM withdrawal.
3. State the function of each of the following: CPU, RAM, ROM, motherboard, bus.
4. What is the difference between volatile and non-volatile memory? Give an example of each.
5. Distinguish between system software and application software, giving two examples of each.
6. List the main steps in the booting process.
7. What is POST and why is it important?
8. Describe the fetch–decode–execute cycle in your own words.
9. Explain what multitasking is and how the CPU achieves it.
10. Why is "people" considered part of a computer system?

Chapter 3: Utilization of Digital Resources

Chapter Overview

This chapter introduces students to the digital tools and systems that people use every day to communicate, collaborate, and process information. It covers communication tools (like email and video conferencing), collaboration tools (like cloud document editors and project management software), the rules of good online behaviour (netiquette), and the different types of digital systems: manual, automated, and computerised processing systems. It also explains simulation and modelling, two powerful ways computers help us understand and test real world systems without risk.

Learning Objectives

1. By the end of this chapter, students should be able to:
2. Identify and describe common digital communication tools.
3. Identify and describe common digital collaboration tools.
4. State the advantages and disadvantages of digital communication and collaboration.
5. Explain netiquette and list key rules for respectful online behaviour.
6. Distinguish between manual systems and automated systems.
7. Explain the advantages and disadvantages of automation.
8. Describe the three main types of data processing: batch, real time, and online.
9. Define simulation and modelling, giving examples of each.
10. State the benefits and limitations of simulation and modelling.

Key Terms

Term	Definition
Digital	Exchanging information electronically using digital tools

Term	Definition
communication	
Digital collaboration	Multiple people working together on the same task or document using digital tools
Email	Electronic mail. Formal written communication sent over a network
Instant messaging	Real time text chat between users
Video conferencing	Real time audio and video communication between multiple participants
VoIP	Voice over Internet Protocol. Making voice calls over the internet
Social media	Online platforms for sharing information and interacting with others
Forum	An online discussion board where users post messages on topics
Cloud storage	Storing files on remote servers accessed over the internet
LMS	Learning Management System. Software for delivering and managing

Term	Definition
	educational courses
Netiquette	Etiquette (good manners) for online communication
Manual system	A system where tasks are carried out entirely by people without computer assistance
Automated system	A system where machines or computers perform tasks with little or no human intervention
Data processing	Converting raw data into meaningful information
Batch processing	Processing data collected over a period, all at once, with no immediate feedback
Real time processing	Processing data immediately as it is received, with instant feedback
Online processing	Processing data as it arrives, but responses need not be instantaneous
Simulation	Using a computer program to imitate the behaviour of a real world system over time

Term	Definition
Modelling	Creating a simplified mathematical or logical representation of a real system
Digital divide	The gap between those who have access to technology and those who do not

3.1 Digital Communication Tools

Digital communication tools allow people to exchange information electronically. They have transformed how we work, learn, and socialise.

Tool	Description	Example
Email	Formal written communication with attachments	Gmail, Outlook, Yahoo Mail
Instant messaging	Real time text chat	WhatsApp, Telegram, Teams chat
Video conferencing	Real time audio visual meetings	Zoom, Google Meet, Skype, Teams
Social media	Public or semi public sharing	Facebook, X (Twitter), Instagram,

Tool	Description	Example
	and interaction	LinkedIn
Forums and discussion boards	Asynchronous group discussion on topics	Stack Overflow, Reddit, Quora
VoIP	Voice calls over the internet	WhatsApp calls, Skype, Zoom
Blogs and wikis	Publishing and collaboratively editing content	WordPress, Wikipedia

Choosing the right tool:

- Use **email** for formal, recorded communication with attachments.
- Use **instant messaging** for quick, informal exchanges.
- Use **video conferencing** when you need to see and hear people in real time.
- Use **forums** for detailed discussions that can happen over time.
- Use **social media** for public announcements or community engagement.

Digital Collaboration Tools

Digital collaboration tools allow multiple people to work on the same task or document at the same time, even if they are in different locations.

Tool	Description	Example
Cloud document editors	Multiple users edit the same file simultaneously, with automatic version history	Google Docs, Office 365
File sharing and cloud storage	Store and share files accessible from any device	Google Drive, OneDrive, Dropbox
Project management tools	Organise tasks, deadlines, and team responsibilities	Trello, Asana, Notion
Learning Management Systems (LMS)	Distribute coursework and assessments	Google Classroom, Moodle, Canvas
Collaborative whiteboards	Draw, brainstorm, and plan together in real time	Miro, Jamboard
Version control systems	Track changes to code and documents over time	Git, GitHub

Advantages of Digital Communication and Collaboration

- **Speed:** Messages arrive in seconds, no matter the distance.
- **Low cost:** No postage, no travel expenses.

- **Global reach:** Work with people in different cities, countries, or continents.
- **Easy sharing:** Large files can be sent instantly.
- **Automatic saving and version control:** No lost work, and you can see who changed what.
- **Searchable history:** Conversations and documents can be searched later.
- **Flexibility:** Work from home, school, or anywhere with internet access.

Disadvantages and Risks

- **Dependence on internet connectivity:** No internet means no communication.
- **Miscommunication:** Text lacks tone and body language, so messages can be misunderstood.
- **Security and privacy risks:** Hacking, phishing, and data leaks.
- **Information overload:** Too many messages and notifications.
- **Distraction:** Constant notifications reduce productivity.
- **Digital divide:** Not everyone has equal access to technology.
- **Impersonation and fraud:** Fake accounts and scams.

Netiquette (Etiquette Online)

Netiquette refers to the rules of polite and respectful behaviour when communicating online.

Key rules of netiquette:

1. **Be respectful and polite.** Treat others as you would like to be treated.
2. **Use clear and concise language.** Avoid ambiguity.
3. **Respect privacy.** Do not share someone else's personal information without permission.
4. **Avoid ALL CAPS.** It is interpreted as shouting.

5. **Verify information before sharing.** Do not spread misinformation.
6. **Think before posting.** Your digital footprint is permanent.
7. **Follow the specific rules of each platform or forum.**
8. **Use proper grammar and spelling** in formal contexts.
9. **Do not spam** or send unwanted messages.
10. **Give credit** when using someone else's work.

3.2 Digital Systems: Manual, Automated, Processing, Simulation and Modelling

Manual Systems

A **manual system** is one where tasks are carried out entirely by people without computer assistance.

Examples:

- A handwritten ledger in a small shop.
- A paper based school register.
- A manual filing system with physical folders.
- Taking orders by writing them on paper.

Advantages:

- Simple and cheap to set up.
- No electricity or internet needed.
- No training in software required.

Disadvantages:

- Slow.

- Prone to human error.
- Difficult to search or back up.
- Hard to share with others at a distance.
- Takes up physical space.

Automated Systems

An **automated system** is one where machines or computers perform tasks with little or no human intervention, based on pre set instructions.

Examples:

- Automatic washing machine.
- ATM (Automated Teller Machine).
- Traffic light controller.
- Factory assembly robot.
- Autopilot on an aircraft.
- Automatic irrigation system on a farm.

Advantages of Automation	Disadvantages of Automation
Speed	High initial setup cost
Consistency and accuracy	Job losses
Runs 24/7 without fatigue	Systems can fail if not maintained

Advantages of Automation	Disadvantages of Automation
Reduces human error and labour cost over time	Less flexible to unexpected situations than a human
Can operate in dangerous environments	Requires skilled staff for maintenance
Increases production	Depends on power and technology

Data Processing

Data processing is the conversion of raw data into meaningful information.

There are three main types of data processing:

Type	**Description**	**Example**
Batch processing	Data is collected over a period and processed together in one go, with no immediate feedback	End of month payroll, bank statement generation, exam results processing
Real time processing	Data is processed immediately as it is received, with instant feedback	ATM withdrawal, airline booking system, GPS navigation
Online	Data is processed as it arrives, but	Online order processing, web

Type	Description	Example
processing	responses need not be instantaneous	form submission

Comparison table:

Feature	Batch	Real Time	Online
Timing	Delayed	Immediate	As it arrives
Feedback	None until batch finishes	Instant	Usually quick but not instant
Example	Payroll	ATM	Online shopping
Volume	Large	Small to medium	Medium

Simulation

A **simulation** is a computer program that imitates the behaviour of a real world system or process over time, in order to study it without using or risking the real thing.

Examples:

- Flight simulators for pilot training.
- Weather forecasting models.

- Crash test simulations for cars.
- Epidemic spread models (e.g., for disease control).
- Nuclear reaction simulations.
- Economic models for government planning.

Modelling

A **model** is a simplified mathematical or logical representation of a real system.

A simulation is often run using a model. The model describes the system; the simulation runs it over time.

Examples of models:

- A spreadsheet model of a school's finances.
- A climate model predicting temperature changes.
- A population growth model.
- A business model predicting profit based on sales.

Benefits of Simulation and Modelling

- **Safer:** No real world risk (e.g., no pilot injured in a flight simulator).
- **Cheaper:** Much less expensive than building and testing the real thing repeatedly.
- **Allows "what if" experiments:** You can test different scenarios without real consequences.
- **Faster:** You can simulate years of climate change in minutes.
- **Repeatable:** The same scenario can be run many times.
- **Allows study of dangerous or inaccessible systems:** e.g., nuclear reactions, deep sea exploration.

Limitations of Simulation and Modelling

- **Accuracy depends on the model:** A simulation is only as good as the model and data behind it.
- **Oversimplification:** Models simplify reality, so they may miss important factors.
- **Cost:** Building realistic simulations can be expensive.
- **Skill required:** Requires experts to design and interpret.
- **Cannot capture everything:** Human behavior and rare events are hard to model accurately.

Review Questions

1. List five digital communication tools and state one use of each.
2. List four digital collaboration tools and explain what each is used for.
3. State four advantages and four disadvantages of digital communication.
4. What is netiquette? List five rules of good netiquette.
5. Distinguish between a manual system and an automated system, giving an example of each.
6. State three advantages and three disadvantages of automation.
7. Explain the difference between batch processing, real time processing, and online processing. Give an example of each.
8. Define simulation and modelling, and explain the relationship between them.
9. Give three examples of simulation and three examples of modelling.
10. State four benefits and three limitations of simulation and modelling.

Chapter 4: Application Areas of Computers and Evaluating the Implications of Computers

Chapter Overview

This chapter explores where and how computers are used in society, the tools that make people more productive, and the effects computers have on social and economic life. It also covers the security of computer systems, computer crimes, professional ethics and conduct, and the laws that govern the use of computers and global communication. Students will learn that computers bring great benefits but also serious risks that must be managed carefully.

Learning Objectives

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

1. Identify and describe major application areas of computers.
2. Explain the purpose of common productivity tools.
3. Discuss the positive and negative social implications of computers.
4. Discuss the positive and negative economic implications of computers.
5. Explain the importance of computer security and describe common security threats.
6. Describe measures used to protect computer systems.
7. Define computer crimes and give examples of common cybercrimes.
8. Explain professional ethics in computing and the purpose of codes of conduct.

9. Explain why legislation is needed for computers and global communication.
10. Describe common areas covered by ICT related legislation.
11. Outline guidelines for responsible and ethical use of computers.

Key Terms

Term	**Definition**
Productivity tool	Software designed to help users complete tasks more efficiently
Word processor	Software for creating and editing text documents
Spreadsheet	Software for calculations, data analysis, and budgeting
Presentation software	Software for creating slideshows
DBMS	Database Management System. Software for organising and retrieving structured data
PIM	Personal Information Manager. Software for managing calendars, tasks, and contacts

Term	Definition
Digital divide	The gap between those who have access to technology and those who do not
Computer security	Protecting hardware, software, and data from damage, theft, and unauthorised access
Malware	Malicious software designed to harm or exploit computers
Virus	Malware that attaches itself to files and spreads when they are opened
Worm	Malware that spreads itself across networks without user action
Trojan	Malware disguised as legitimate software
Ransomware	Malware that encrypts files and demands payment to unlock them
Spyware	Malware that secretly collects information about the user
Hacking	Unauthorised access to a computer system

Term	Definition
Phishing	Tricking users into revealing personal or financial information
DoS attack	Denial of Service. Overwhelming a system so it cannot function
DDoS	Distributed Denial of Service. A DoS attack from many computers at once
Firewall	Hardware or software that monitors and controls network traffic
Encryption	Scrambling data so it cannot be read without the correct key
Two factor authentication	Using two different methods to verify identity
Backup	A copy of data stored separately in case the original is lost
Cybercrime	Illegal activity carried out using computers or networks
Software piracy	Illegal copying or distribution of copyrighted software

Term	Definition
Identity theft	Stealing someone's personal information to commit fraud
Cyberstalking	Using technology to harass or threaten someone
Cyberterrorism	Attacks on critical infrastructure for political or ideological aims
Professional ethics	Moral principles guiding the behaviour of ICT professionals
Code of conduct	A set of rules published by a professional body guiding ethical behaviour
Data protection law	Legislation regulating how personal data is collected, stored, and used
Copyright	Legal protection for original creative and technical works
Intellectual property	Creations of the mind protected by law (e.g., software, music, writing)
Freedom of Information	The right of citizens to access certain government held information

Term	Definition
Jurisdiction	The legal authority of a court or government over a particular area

4.1 Computer Applications and Productivity Tools

Major Application Areas of Computers

Computers are used in almost every field of human activity. Below are the main areas.

Area	Examples of Use
Education	E learning platforms, digital libraries, educational software, online research, virtual classrooms
Business and Commerce	Accounting software, point of sale systems, e commerce, inventory management, customer databases
Healthcare	Patient record systems, diagnostic equipment, telemedicine, robotic surgery, health monitoring apps
Government	E government services, census and data management, national ID systems, tax collection

Area	**Examples of Use**
Banking and Finance	ATMs, online banking, mobile money, fraud detection systems, stock trading
Agriculture	Precision farming, weather prediction, drone monitoring of crops, soil analysis
Entertainment	Gaming, streaming media, digital art and music production, animation
Transport	GPS navigation, traffic management, flight booking and control systems, railway scheduling
Communication	Telecommunication networks, social media, video conferencing, messaging apps
Science and Research	Simulation, data analysis, computational modelling, space exploration
Security and Defence	Surveillance, encryption, cybersecurity, drones, military planning
Manufacturing	Computer aided design (CAD), robotics, quality control, supply chain management

Productivity Tools

Productivity tools are software designed to help individuals and organisations complete tasks more efficiently.

Tool	Purpose	Example
Word processors	Creating and editing text documents	Microsoft Word, Google Docs
Spreadsheets	Calculations, data analysis, budgeting	Microsoft Excel, Google Sheets
Presentation software	Creating slideshows for communication	Microsoft PowerPoint, Google Slides
Database Management Systems	Organising and retrieving large sets of structured data	MySQL, Microsoft Access, Oracle
Personal Information Managers	Managing calendars, task lists, and contacts	Microsoft Outlook, Google Calendar
Graphics and design	Creating and editing images	Adobe Photoshop,

Tool	Purpose	Example
tools	and layouts	Canva, GIMP
Collaboration suites	Combined tools for teams to work together	Microsoft 365, Google Workspace
Project management tools	Planning tasks, deadlines, and resources	Trello, Asana, Microsoft Project

Why productivity tools matter:

- They save time on repetitive tasks.
- They reduce errors through automation.
- They enable collaboration across locations.
- They provide templates and formats that meet professional standards.

4.2 Social and Economic Implications of Computers; Security of Computer Systems

Social Implications of Computers

Computers have changed how people live, learn, work, and interact. These changes bring both benefits and problems.

Positive social implications:

- Improved access to information and education (e learning, online libraries).

- Easier communication across distances (email, video calls, social media).
- New forms of social interaction and community (online groups, forums).
- Assistive technology for people with disabilities (screen readers, speech to text).
- Access to healthcare information and telemedicine.
- Entertainment and creativity (streaming, digital art, music production).

Negative social implications:

- Reduced face to face interaction and loneliness.
- Spread of misinformation and fake news.
- Cyberbullying and online harassment.
- Addiction to devices and social media.
- The digital divide: unequal access between rich and poor, urban and rural.
- Privacy concerns and loss of personal data.
- Job displacement and changing skill requirements.
- Exposure to inappropriate content, especially for children.

Economic Implications of Computers

Positive economic implications:

- Creation of new jobs and industries (ICT sector, software development, data science).
- Increased business efficiency and productivity.

- Growth of e commerce and online services.
- Remote work opportunities and flexible employment.
- Economic growth through innovation and entrepreneurship.
- Access to global markets for small businesses.
- Improved financial inclusion through mobile money.

Negative economic implications:

- Job losses due to automation in manufacturing, retail, and administration.
- High cost of technology adoption for small businesses.
- Widening economic inequality where only some can access and benefit from technology.
- Dependence on foreign technology and expertise.
- Cybersecurity costs and losses from cybercrime.
- Environmental impact of e waste and energy consumption.

Security of Computer Systems

Computer security means protecting hardware, software, and data from damage, theft, and unauthorized access.

Common security threats:

Threat	**Description**

Threat	Description
Malware	Malicious software including viruses, worms, trojans, ransomware, and spyware
Hacking	Unauthorized access into a system
Phishing	Tricking users into revealing personal or financial information via fake messages or websites
DoS and DDoS attacks	Overwhelming a system with traffic so it cannot function
Data theft and breaches	Unauthorized copying or leaking of sensitive data
Social engineering	Manipulating people into breaking security procedures
Keylogging	Recording keystrokes to capture passwords and other data
Man in the middle	Secretly intercepting and altering communication between two parties
Ransomware	Encrypting files and demanding payment for their release

Security measures:

Type	Examples
Physical security	Locks, biometric access, security guards, CCTV for server rooms
Software security	Antivirus software, firewalls, encryption, strong passwords, two factor authentication
Policy and procedural security	Regular data backups, staff training, access control policies, regular software updates and patches
Network security	VPNs, secure Wi Fi (WPA3), intrusion detection systems
Human security	Awareness training, verifying identities, reporting suspicious activity

Best practices for users:

- Use strong, unique passwords for each account.
- Enable two factor authentication wherever possible.
- Keep software and operating systems updated.
- Back up important data regularly.

- Be cautious of unexpected emails, links, and attachments.
- Avoid using public Wi Fi for sensitive transactions.

4.3 Computer Crimes, Professional Ethics and Conduct

Computer Crimes (Cybercrimes)

Computer crimes are illegal activities carried out using computers or networks.

Crime	**Description**
Hacking	Unauthorised access to systems or data
Software piracy	Illegal copying or distribution of copyrighted software
Identity theft	Stealing someone's personal information to commit fraud
Cyberstalking and harassment	Using technology to threaten, harass, or bully
Fraud	Online scams, fake investment schemes, phishing
Cyberterrorism	Attacks on critical infrastructure for political or ideological aims
Spreading malware	Distributing viruses, ransomware, and other

Crime	Description
	malicious software
Financial crimes	Card fraud, money laundering online, cryptocurrency scams
Data breaches	Illegally accessing and leaking confidential data
Intellectual property theft	Stealing or illegally using copyrighted material

Effects of cybercrime:

- Financial loss for individuals and organisations.
- Damage to reputation.
- Loss of privacy.
- Psychological harm to victims.
- Disruption of essential services.
- Erosion of trust in technology.

Professional Ethics in Computing

Professional ethics are the moral principles that guide the behaviour of ICT professionals.

Key ethical principles:

- Respect intellectual property and copyright.
- Maintain confidentiality of client and user data.
- Be honest and accurate in reporting work.
- Avoid conflicts of interest.
- Use skills only for legitimate, lawful purposes.
- Give proper credit for others' work and avoid plagiarism.
- Refuse to design or deploy systems that harm people.
- Consider the social and environmental impact of your work.
- Keep your skills and knowledge up to date.
- Report unethical behaviour by colleagues.

Codes of Conduct

Professional bodies publish codes of conduct that guide professional and ethical behaviour for computing practitioners.

Major professional bodies:

- ACM (Association for Computing Machinery)
- BCS (British Computer Society)
- IEEE (Institute of Electrical and Electronics Engineers)

Common themes in codes of conduct:

- Honesty and integrity.
- Competence and professionalism.
- Public interest and safety.
- Confidentiality and privacy.
- Compliance with laws and regulations.
- Respect for colleagues and users.

4.4 Legislation and Global Communication; Responsible and Ethical Use of Computers

Why Legislation is Needed

Laws are needed to:

- Protect individuals' data and privacy.
- Define and punish computer crimes.
- Regulate the use of digital technology and global communication.
- Protect intellectual property and copyright.
- Ensure fair competition and consumer protection.
- Enable international cooperation on cybercrime.

Common Areas Covered by ICT Related Legislation

Area	**What it Covers**
Data Protection laws	How personal data is collected, stored, and used.

Area	What it Covers
	Consent, right to access one's data, security obligations
Copyright and Intellectual Property laws	Protect original creative and technical works (software, music, writing) from unauthorised copying or use
Computer misuse laws	Criminalise hacking, spreading malware, and unauthorised access to systems
Freedom of Information laws	Give citizens the right to access certain government held information
E commerce laws	Regulate online transactions, consumer rights, and electronic signatures
Cybercrime laws	Define and punish online fraud, identity theft, and cyberterrorism
Privacy laws	Protect individuals from unwarranted surveillance and data collection

Global Communication Issues Raised by the Internet

The internet crosses national borders, which raises complex legal questions:

- Jurisdiction: Whose laws apply when a crime crosses borders?
- Differing national laws: Countries have different rules on privacy, censorship, and content.
- Enforcement: It is difficult to investigate and prosecute crimes across multiple countries.
- International cooperation: Organisations like Interpol and agreements like the Budapest Convention help countries work together on cybercrime.

Responsible and Ethical Use of Computers: Guidelines for Users

1. Respect the privacy of others. Do not access accounts or data without permission.
2. Use software legally. Avoid piracy and use licensed or open source software.
3. Verify information before sharing. Avoid spreading misinformation.
4. Practise good digital citizenship. No cyberbullying, respectful communication.
5. Protect personal information. Use strong passwords and be careful what you share.
6. Cite sources. Avoid plagiarism in academic and professional work.
7. Consider the environmental impact. Dispose of old devices responsibly and reduce e waste.

8. Follow the law. Know and obey the laws of your country regarding computer use.
9. Think before you post. Your digital footprint is permanent.
10. Report unethical or illegal behaviour. Help keep the digital world safe.

Review Questions

1. List six major application areas of computers and give one example of use for each.
2. Name four productivity tools and state what each is used for.
3. State four positive and four negative social implications of computers.
4. State four positive and four negative economic implications of computers.
5. What is computer security? Why is it important?
6. List six common security threats and briefly describe each.
7. Describe four types of security measures used to protect computer systems.
8. Define cybercrime and give five examples.
9. What are professional ethics? List five ethical principles for ICT professionals.
10. What is a code of conduct? Name two professional bodies that publish them.
11. Why is legislation needed for computers and global communication?
12. List five areas covered by ICT related legislation.
13. Explain the concept of jurisdiction in the context of global communication.
14. List six guidelines for responsible and ethical use of computers.

Chapter 5: Representing Data in Computer Systems

Chapter Overview

This chapter explains how computers represent, store, and manipulate data. It begins with a recap of computer hardware components and embedded systems, then moves into number systems (binary, octal, denary, and hexadecimal) and how to convert between them. Students will learn how computers perform arithmetic on binary numbers, how signed integers are represented using sign and magnitude and two's complement, and how characters, images, and sound are represented digitally. This is a foundational chapter for understanding everything else in computer science.

Learning Objectives

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

1. Describe the main components of the CPU and the function of each register.
2. Define embedded systems and give examples of their use.
3. Explain the characteristics, advantages, and disadvantages of embedded systems.
4. Convert numbers between binary, octal, denary, and hexadecimal.
5. Perform binary addition, subtraction, multiplication, and division.
6. State the units of computer storage and their relationships.
7. Represent signed integers using sign and magnitude and two's complement.
8. Perform arithmetic on signed integers and detect overflow.
9. Explain how characters are represented using ASCII and Unicode.
10. Explain how images are represented using pixels, resolution, and colour depth.
11. Explain how sound is represented using sampling rate and bit depth.

12. Calculate file sizes for images and sound.

Key Terms

Term	Definition
CPU	Central Processing Unit. The brain of the computer
Control Unit (CU)	Part of the CPU that directs operations and decodes instructions
ALU	Arithmetic Logic Unit. Performs arithmetic and logic operations
Register	Tiny, very fast storage location inside the CPU
Program Counter (PC)	Holds the address of the next instruction to be fetched
Memory Address Register (MAR)	Holds the address of the memory location being accessed
Memory Data Register (MDR)	Holds the data being transferred to or from memory

Term	Definition
Current Instruction Register (CIR)	Holds the instruction currently being decoded or executed
Accumulator (ACC)	Holds the results of calculations performed by the ALU
Embedded system	A computer built into a larger device to perform one or a few specific functions
Binary	Base 2 number system using digits 0 and 1
Octal	Base 8 number system using digits 0 to 7
Denary	Base 10 number system using digits 0 to 9
Hexadecimal	Base 16 number system using digits 0 to 9 and A to F
Bit	Smallest unit of data. A single 0 or 1
Nibble	4 bits
Byte	8 bits

Term	Definition
Kilobyte (KB)	1024 bytes
Megabyte (MB)	1024 KB
Gigabyte (GB)	1024 MB
Terabyte (TB)	1024 GB
Petabyte (PB)	1024 TB
Sign and magnitude	A method of representing signed integers where the leftmost bit is the sign
Two's complement	The method used by computers to represent signed integers
Overflow	When a calculation produces a result too large to fit in the available bits
ASCII	American Standard Code for Information Interchange. 7 or 8 bit character encoding

Term	Definition
Unicode	16 bit or more character encoding covering all world scripts
Pixel	A single dot in a digital image
Resolution	The number of pixels in an image (width times height)
Colour depth	The number of bits used to represent the colour of each pixel
Vector graphic	An image stored as mathematical shapes rather than pixels
Sampling	Measuring the amplitude of a sound wave at regular intervals
Sampling rate	The number of samples taken per second, measured in Hertz
Bit depth (sound)	The number of bits used to represent each sound sample
Analogue	A continuous signal (e.g., a sound wave)
Digital	A discrete signal represented by 0s and 1s

5.1 Computer Hardware Components and Embedded Systems

Recap: Inside the CPU

The CPU is the brain of the computer. It contains three main parts:

Component	**Function**
Control Unit (CU)	Directs operations. Decodes instructions. Controls timing and data flow
ALU (Arithmetic Logic Unit)	Performs arithmetic (+, -, x, /) and logic (AND, OR, NOT) operations
Registers	Tiny, very fast storage locations inside the CPU

Key CPU Registers

Register	**Full Name**	**Purpose**
PC	Program Counter	Holds the address of the next instruction to be fetched
MAR	Memory Address Register	Holds the address of the memory location being accessed

Register	Full Name	Purpose
MDR	Memory Data Register	Holds the data or instruction being transferred to or from memory
CIR	Current Instruction Register	Holds the instruction currently being decoded or executed
ACC	Accumulator	Holds the results of calculations performed by the ALU

Embedded Systems

An **embedded system** is a computer system (a microprocessor or microcontroller with dedicated software) built into a larger device to perform one or a few specific functions, rather than being a general-purpose computer.

Examples of embedded systems:

- Washing machines
- Microwave ovens
- Digital watches
- Car engine management systems
- Medical devices such as pacemakers
- Traffic light controllers
- Home appliances (refrigerators, air conditioners)

- ATMs
- Digital cameras
- Fitness trackers

Characteristics of embedded systems:

- Dedicated to a specific task
- Usually, low power and low cost
- Often operate in real time
- Generally, have no or minimal user interface
- Software is often stored permanently in ROM or firmware
- May be part of a larger device

Advantages of embedded systems:

- Reliable for their single task
- Small size
- Low power consumption
- Fast response (real time)
- Cost effective in mass production
- Can operate in harsh environments

Disadvantages of embedded systems:

- Difficult to upgrade or reprogram
- Limited flexibility
- Higher design cost for initial development
- Hard to debug once deployed

- Limited memory and processing power

5.2 Base Conversions (Number Systems)

Computers store all data in **binary (base 2)**, but humans often work in **denary or decimal (base 10)**. Programmers frequently use **octal (base 8)** and **hexadecimal (base 16)** as shorthand for binary.

Common Number Bases

Base	Name	Digits Used
2	Binary	0, 1
8	Octal	0 to 7
10	Denary (Decimal)	0 to 9
16	Hexadecimal	0 to 9, A to F (A=10, B=11, C=12, D=13, E=14, F=15)

Denary to Binary

Method: Repeated division by 2, reading remainders from bottom to top.

Example: Convert 45 to binary.

- 45 divided by 2 = 22 remainder 1
- 22 divided by 2 = 11 remainder 0

- 11 divided by 2 = 5 remainder 1
- 5 divided by 2 = 2 remainder 1
- 2 divided by 2 = 1 remainder 0
- 1 divided by 2 = 0 remainder 1

Reading remainders bottom to top: **101101**

So 45 in denary = 101101 in binary.

Binary to Denary

Method: Multiply each bit by its place value (powers of 2) and sum.

Example: Convert 101101 to denary.

- 1 x 32 = 32
- 0 x 16 = 0
- 1 x 8 = 8
- 1 x 4 = 4
- 0 x 2 = 0
- 1 x 1 = 1

Total = 32 + 8 + 4 + 1 = **45**

Denary to Hexadecimal

Method: Repeated division by 16.

Example: Convert 45 to hexadecimal.

- 45 divided by 16 = 2 remainder 13 (which is D in hex)
- 2 divided by 16 = 0 remainder 2

Reading remainders bottom to top: **2D**

So 45 in denary = 2D in hexadecimal.

Hexadecimal to Denary

Method: Multiply each digit by its place value (powers of 16) and sum.

Example: Convert 2D to denary.

- 2 x 16 = 32
- D (13) x 1 = 13

Total = 32 + 13 = **45**

Hexadecimal to Binary (Shortcut)

Method: Convert each hex digit to its 4 bit binary equivalent.

Example: Convert 2D to binary.

- 2 = 0010
- D (13) = 1101

So 2D in hex = 00101101 in binary.

Binary to Hexadecimal (Shortcut)

Method: Group binary digits in 4s from the right, then convert each group to hex.

Example: Convert 101101 to hex.

- Group from right: 0010 1101
- 0010 = 2
- 1101 = D

So 101101 in binary = 2D in hex.

Binary to Octal

Method: Group binary digits in 3s from the right, then convert each group.

Example: Convert 101101 to octal.

- Group from right: 101 101
- 101 = 5
- 101 = 5

So 101101 in binary = 55 in octal.

Octal to Binary

Method: Convert each octal digit to its 3 bit binary equivalent.

Example: Convert 55 to binary.

- 5 = 101
- 5 = 101

So 55 in octal = 101101 in binary.

Summary Table of Conversions

From	To	Method
Denary	Binary	Divide by 2 repeatedly
Binary	Denary	Multiply by powers of 2 and sum
Denary	Hex	Divide by 16 repeatedly
Hex	Denary	Multiply by powers of 16 and sum

From	To	Method
Hex	Binary	Each hex digit becomes 4 bits
Binary	Hex	Group into 4s from right
Binary	Octal	Group into 3s from right
Octal	Binary	Each octal digit becomes 3 bits

5.3 Operations on Number Systems and Computer Data Units

Binary Addition

Rules:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0 carry 1
- 1 + 1 + 1 = 1 carry 1

Example: 1011 + 1101

1011

+ 1101

11000

Explanation:

- Rightmost: 1 + 1 = 0 carry 1
- Next: 1 + 0 + 1 (carry) = 0 carry 1
- Next: 0 + 1 + 1 (carry) = 0 carry 1
- Next: 1 + 1 + 1 (carry) = 1 carry 1
- Final carry: 1

Result: 11000

Binary Subtraction

Method 1: Direct subtraction with borrowing.

Example: 1011 minus 0110

1011

- 0110

0101

Method 2: Add the two's complement of the number being subtracted. This is how computers do it.

Binary Multiplication

Follows the same long multiplication logic as denary, using only 0s and 1s.

Example: 101 multiplied by 11

101

x 11

101

101

1111

Result: 1111

Binary Division

Follows the same long division logic as denary.

Example: 1100 divided by 10

1100 divided by 10 = 110

Computer Data Units (Storage Hierarchy)

Unit	**Size**
Bit	Smallest unit (0 or 1)
Nibble	4 bits
Byte	8 bits

Unit	Size
Kilobyte (KB)	1024 bytes (2 to the power 10)
Megabyte (MB)	1024 KB
Gigabyte (GB)	1024 MB
Terabyte (TB)	1024 GB
Petabyte (PB)	1024 TB

Note: Storage manufacturers sometimes use decimal (1000) multiples for marketing. This is why an advertised 1TB drive shows slightly less space in the operating system, which uses 1024 based binary counting.

5.4 Representation of Signed Integers

Computers must represent both positive and negative whole numbers using only bits (0s and 1s).

Sign and Magnitude Representation

In sign and magnitude representation, the leftmost (most significant) bit is the **sign bit**:

- 0 = positive
- 1 = negative

The remaining bits represent the **magnitude** (size) of the number.

Example (8 bit):

- +5 = 00000101
- -5 = 10000101

Weaknesses of sign and magnitude:

- Two representations of zero (+0 and -0)
- Arithmetic circuits are more complex
- Addition and subtraction need different circuits

Two's Complement Representation

Two's complement is the method actually used in real computers.

Steps to represent a negative number in two's complement:

1. Write the positive number in binary.
2. Invert (flip) all the bits. This is called one's complement.
3. Add 1 to the result.

Example (8 bit): Represent -5

- Step 1: 5 = 00000101
- Step 2: Invert = 11111010
- Step 3: Add 1 = 11111011

So -5 in two's complement (8 bit) = 11111011

Advantages of two's complement:

- Only one representation of zero
- Addition and subtraction can be done with the same circuit

- No special subtractor needed

5.5 Signed Integer Arithmetic

Adding Two's Complement Numbers

Add normally in binary. If there is a carry out of the leftmost (sign) bit, it is simply discarded.

Example (8 bit): 5 + (-3)

- 5 = 00000101
- -3 = 11111101
- Sum = 100000010

Discard the overflow carry bit: 00000010 = 2. Correct!

Overflow

Overflow occurs when the result of adding two numbers of the same sign produces a result with the wrong sign. This happens because the result cannot fit in the available number of bits.

Example: Adding two large positive numbers that produce a result interpreted as negative.

Overflow must be detected and flagged by the CPU's arithmetic circuitry.

Range of an n bit two's complement number:

From -2 to the power (n-1) to +2 to the power (n-1) minus 1.

Example: An 8 bit signed number can represent -128 to +127.

Number of bits	**Range**
4	-8 to +7

Number of bits	Range
8	-128 to +127
16	-32768 to +32767
32	-2,147,483,648 to +2,147,483,647

5.6 Representation of Characters, Images and Sound

Characters

Each character is represented by a unique binary code.

Encoding	Bits per Character	Characters Possible	Notes
ASCII	7 or 8	128 to 256	English letters, digits, punctuation, control codes
Unicode	16 or more	Virtually all world scripts	Multilingual text, emojis

Example: The letter 'A' in ASCII is 65, which is 01000001 in binary.

Why Unicode is needed:

- ASCII cannot represent characters from other languages (e.g., Chinese, Arabic, Hindi).
- Unicode allows one document to contain text from many languages.
- Unicode also includes symbols, mathematical characters, and emojis.

Images

A digital image is made up of a grid of tiny coloured dots called **pixels**.

Key concepts:

Concept	**Description**
Resolution	The number of pixels in the image (width times height). Higher resolution = more detail but larger file size
Colour depth (bit depth)	The number of bits used to represent the colour of each pixel
File size	Width times height times colour depth (in bits)

Common colour depths:

Colour depth	**Number of colours**
1 bit	2 colours (black and white)

Colour depth	Number of colours
8 bits	256 colours or greyscale levels
24 bits	About 16.7 million colours (true colour)

24 bit colour: 8 bits each for Red, Green, and Blue (RGB).

Example file size calculation:

An image that is 100 pixels wide and 50 pixels high with a colour depth of 24 bits.

- File size = 100 x 50 x 24 = 120,000 bits
- 120,000 divided by 8 = 15,000 bytes
- 15,000 divided by 1024 = approximately 14.6 KB

Vector graphics:

- Alternative to pixel (bitmap) images.
- Store images as mathematical shapes and paths rather than pixels.
- Scale without losing quality.
- Used for logos and diagrams.
- Smaller file sizes for simple images.

Sound

Sound in the real world is an **analogue (continuous) wave**. Computers can only store **digital (discrete)** data, so sound must be converted using **sampling**.

Key concepts:

Concept	Description
Sampling rate	The number of samples (measurements) of the sound wave taken per second, measured in Hertz (Hz)
Bit depth	The number of bits used to represent each sample's amplitude. More bits = more accurate reproduction
Channels	Mono = 1 channel. Stereo = 2 channels
File size	Sampling rate times bit depth times duration (seconds) times number of channels

Common sampling rates:

- CD quality audio: 44,100 Hz (44.1 kHz)
- Telephone quality: 8,000 Hz (8 kHz)
- High resolution audio: 96,000 Hz or higher

Example file size calculation:

A 3 minute stereo recording at 44,100 Hz with 16 bit depth.

- Duration in seconds = 3 x 60 = 180 seconds
- File size = 44,100 x 16 x 180 x 2

- = 44,100 x 16 = 705,600
- 705,600 x 180 = 127,008,000
- 127,008,000 x 2 = 254,016,000 bits
- 254,016,000 divided by 8 = 31,752,000 bytes
- 31,752,000 divided by 1024 = 31,007 KB
- 31,007 divided by 1024 = approximately 30.3 MB

Higher sampling rate and bit depth = better sound quality but larger file size.

Review Questions

1. Name the three main parts of the CPU and state the function of each.
2. List five CPU registers and state what each holds.
3. Define an embedded system and give four examples.
4. State three advantages and three disadvantages of embedded systems.
5. Convert the following denary numbers to binary: 25, 64, 100, 255.
6. Convert the following binary numbers to denary: 1101, 101010, 11110000.
7. Convert the following denary numbers to hexadecimal: 30, 255, 4096.
8. Convert the following hexadecimal numbers to denary: 1A, FF, 100.
9. Convert 2F to binary.
10. Convert 11010110 to hexadecimal.
11. Perform the following binary additions: 1011 + 0110, 1101 + 1011.
12. Perform the following binary multiplication: 101 x 11.
13. List the units of computer storage from smallest to largest.
14. Explain how sign and magnitude represents negative numbers, and state two weaknesses of this method.
15. Explain how two's complement represents negative numbers. Show the steps to represent -12 in 8 bit two's complement.
16. What is overflow? When does it occur?
17. State the range of an 8 bit two's complement number.
18. What is the difference between ASCII and Unicode?
19. Define pixel, resolution, and colour depth.

20. Calculate the file size of a 200 x 100 pixel image with 24 bit colour depth. Give your answer in KB.

21. Define sampling rate and bit depth.

22. Calculate the file size of a 2 minute mono recording at 44,100 Hz with 16 bit depth. Give your answer in MB.

23. What is the difference between a bitmap image and a vector graphic?

Chapter 6: Designing and Analysing Digital Logic Circuits

Chapter Overview

This chapter introduces students to digital logic, the foundation of all modern computers. It begins with logic gates, the basic building blocks of digital circuits, and their truth tables. Students will learn how to write Boolean expressions, apply Boolean laws to simplify expressions, and design logic circuits from problem statements. The chapter then covers binary arithmetic circuits (half adders and full adders) and sequential logic circuits (flip-flops), which are circuits with memory. By the end of this chapter, students will understand how simple gates combine to perform complex operations inside a computer.

Learning Objectives

1. By the end of this chapter, students should be able to:
2. Define digital logic and explain the two states used in digital circuits.
3. Identify and describe the function of each logic gate.
4. Construct truth tables for logic gates and combinations of gates.
5. Write Boolean expressions from truth tables and circuit diagrams.
6. Apply Boolean laws, including De Morgan's Laws, to simplify expressions.
7. Design a logic circuit from a problem statement using truth tables and Boolean expressions.
8. Explain the difference between combinational and sequential logic circuits.
9. Describe the construction and operation of half adders and full adders.
10. Explain how ripple-carry adders work.
11. Describe the operation of SR, D, JK, and T flip-flops.
12. Explain the purpose of a clock signal in sequential circuits.
13. Give examples of sequential circuit applications (registers, counters, memory).

Key Terms

Term	Definition

Term	Definition
Digital logic	The branch of electronics dealing with circuits that operate using only two states: 0 and 1
Logic gate	A basic building block of digital circuits that performs a logical operation on one or more inputs
Truth table	A table showing all possible input combinations and the corresponding output for a logic circuit
Boolean expression	An algebraic expression using variables and logical operators (AND, OR, NOT)
Boolean algebra	A branch of mathematics used to simplify and analyse logic circuits
De Morgan's Laws	Two rules for simplifying expressions involving NOT and AND or OR
Sum of products	A method of deriving a Boolean expression by ORing together AND terms where output is 1
Karnaugh map (K-map)	A graphical method for simplifying Boolean expressions
Combinational circuit	A circuit whose output depends only on the current inputs
Sequential circuit	A circuit whose output depends on current inputs and previous state (has memory)
Half adder	A circuit that adds two single binary digits, producing a sum and a carry

Term	Definition
Full adder	A circuit that adds two binary digits plus a carry-in, producing a sum and a carry-out
Ripple carry adder	Multiple full adders chained together to add multi-bit binary numbers
Flip flop	The basic memory element of sequential circuits, storing a single bit
SR flip flop	Set Reset flip flop
D flip flop	Data flip flop
JK flip flop	An improved SR flip flop with no invalid state
T flip flop	Toggle flip flop
Clock signal	A regular pulse that synchronises when sequential circuits update their stored value
Register	A set of flip flops storing multiple bits
Counter	A circuit that counts clock pulses

6.1 Digital Logic and Logic Gates

What is Digital Logic?

Digital logic is the branch of electronics and computing dealing with circuits that operate using only two states:

- 0 (off, false, low voltage)

- 1 (on, true, high voltage)

These are the same two states used to represent binary data. All operations inside a computer, from addition to memory storage, are built from digital logic circuits.

Logic Gates

Logic gates are the basic building blocks of digital circuits. Each gate performs one logical operation on one or more binary inputs to produce one binary output.

The Seven Basic Logic Gates

Gate	**Symbol Description**	**Function**	**Truth Table (2-input)**
NOT	Triangle with bubble	Inverts the input	0 gives 1, 1 gives 0
AND	Flat-back D shape	Output 1 only if all inputs are 1	00=0, 01=0, 10=0, 11=1
OR	Curved-back shape	Output 1 if at least one input is 1	00=0, 01=1, 10=1, 11=1
NAND	AND + bubble	Opposite of AND	00=1, 01=1, 10=1, 11=0
NOR	OR + bubble	Opposite of OR	00=1, 01=0, 10=0, 11=0
XOR	OR with extra curve	Output 1 if inputs are different	00=0, 01=1, 10=1, 11=0

Gate	Symbol Description	Function	Truth Table (2-input)
XNOR	XOR + bubble	Output 1 if inputs are the same	00=1, 01=0, 10=0, 11=1

Detailed Explanation of Each Gate

NOT Gate (Inverter)

- Has one input and one output.
- Output is the opposite of the input.
- If input is 0, output is 1. If input is 1, output is 0.

Truth table:

A	NOT A
0	1
1	0

AND Gate

- Has two or more inputs.
- Output is 1 only when all inputs are 1.

- Output is 0 in all other cases.

Truth table (2 inputs):

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

- Has two or more inputs.
- Output is 1 when at least one input is 1.
- Output is 0 only when all inputs are 0.

Truth table (2 inputs):

A	B	A OR B
0	0	0

A	B	A OR B
0	1	1
1	0	1
1	1	1

NAND Gate

- **AND followed by NOT.**
- **Output is 0 only when all inputs are 1.**
- **Output is 1 in all other cases.**

Truth table (2 inputs):

A	B	A NAND B
0	0	1
0	1	1
1	0	1

A	B	A NAND B
1	1	0

NOR Gate

- OR followed by NOT.
- Output is 1 only when all inputs are 0.
- Output is 0 in all other cases.

Truth table (2 inputs):

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

XOR Gate (Exclusive OR)

- Output is 1 when inputs are different.

- Output is 0 when inputs are the same.

Truth table (2 inputs):

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate (Exclusive NOR)

- Output is 1 when inputs are the same.
- Output is 0 when inputs are different.

Truth table (2 inputs):

A	B	A XNOR B
0	0	1

A	B	A XNOR B
0	1	0
1	0	0
1	1	1

Boolean Expressions

Boolean expressions describe logic gate behavior algebraically.

Gate	Boolean Expression
AND	A · B (or A AND B)
OR	A + B (or A OR B)
NOT	A with a bar over it (or A')
NAND	NOT (A · B)
NOR	NOT (A + B)

Gate	Boolean Expression
XOR	A XOR B (or A direct sum B)
XNOR	NOT (A XOR B)

Boolean Laws

Boolean laws are used to simplify expressions and reduce the number of gates needed.

Law	AND Form	OR Form
Commutative	$A \cdot B = B \cdot A$	$A + B = B + A$
Associative	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C = A + (B + C)$
Distributive	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Identity	$A \cdot 1 = A$	$A + 0 = A$
Null	$A \cdot 0 = 0$	$A + 1 = 1$
Idempotent	$A \cdot A = A$	$A + A = A$

Law	AND Form	OR Form
Complement	A · NOT A = 0	A + NOT A = 1
Double Negation	NOT (NOT A) = A	NOT (NOT A) = A
Absorption	A · (A + B) = A	A + (A · B) = A

De Morgan's Laws

De Morgan's Laws are two powerful rules for simplifying expressions involving NOT.

First Law: NOT (A · B) = (NOT A) + (NOT B)

In words: The negation of (A AND B) is equal to (NOT A) OR (NOT B).

Second Law: NOT (A + B) = (NOT A) · (NOT B)

In words: The negation of (A OR B) is equal to (NOT A) AND (NOT B).

Example: Simplify NOT (A · B).

Using De Morgan's First Law: NOT (A · B) = (NOT A) + (NOT B).

So a NAND gate can be replaced by an OR gate with inverted inputs.

6.2 Logic Circuit Design

Designing a Circuit from a Problem Statement

Follow these steps:

1. Identify inputs and the required output(s). What variables affect the output? What is the output?
2. Write a truth table listing every combination of inputs and the correct output.
3. Derive a Boolean expression from the truth table using the sum of products method.
4. Simplify the expression using Boolean laws or a Karnaugh map (K-map) to use the fewest gates possible.
5. Draw the logic circuit diagram using standard gate symbols.

The Sum of Products Method

To derive a Boolean expression from a truth table:

1. Look at every row where the output is 1.
2. For each such row, write an AND term:
    - If an input is 1, use the variable as is.
    - If an input is 0, use the NOT of the variable.
3. OR all the AND terms together.

Example: Design a circuit where output Q = 1 only when inputs A and B differ.

Step 1: Inputs A and B. Output Q.

Step 2: Truth table.

A	**B**	**Q**
0	0	0

A	B	Q
0	1	1
1	0	1
1	1	0

Step 3: Sum of products.

Rows where Q = 1:

- A=0, B=1 gives (NOT A) · B
- A=1, B=0 gives A · (NOT B)

So Q = (NOT A) · B + A · (NOT B)

Step 4: This is already simplified. It is the definition of XOR.

Q = A XOR B

Step 5: The circuit uses one XOR gate (or equivalent AND, OR, NOT gates).

Karnaugh Maps (K-maps)

A Karnaugh map is a graphical method for simplifying Boolean expressions. It arranges truth table values in a grid so that adjacent cells differ by only one variable. Groups of 1s (called groupings) can be combined to eliminate variables.

Basic 2-variable K-map:

A\B	0	1
0		
1		

Basic 3-variable K-map:

A\BC	00	01	11	10
0				
1				

Rules for grouping:

- Groups must contain 1, 2, 4, 8, or 16 cells.
- Groups must be rectangular.
- Groups can wrap around edges.
- Each group eliminates the variable(s) that change within the group.
- Use the fewest, largest groups possible.

Combinational Logic Circuits

Combinational logic circuits are circuits whose output depends only on the current inputs. They have no memory.

Examples:

- Adders (half adder, full adder)
- Decoders
- Multiplexers
- Encoders
- Comparators
-

6.3 Binary Arithmetic Circuits

Half Adder

A half adder adds two single binary digits (A and B) and produces:

- Sum
- Carry

Truth table:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Boolean expressions:

- Sum = A XOR B
- Carry = A AND B

Circuit: One XOR gate (for Sum) and one AND gate (for Carry).

Limitation: A half adder cannot accept a carry-in from a previous column. It can only add the two least significant bits.

Full Adder

A full adder adds two binary digits plus a carry-in (Cin) from the previous column. It produces:

- Sum
- Carry-out (Cout)

Truth table:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0

A	B	Cin	Sum	Cout
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Boolean expressions:

- Sum = A XOR B XOR Cin
- Cout = (A AND B) OR (Cin AND (A XOR B))

Construction: A full adder can be built from two half adders and one OR gate.

Ripple-Carry Adder

A ripple-carry adder is made by chaining multiple full adders together. The carry-out of one full adder feeds the carry-in of the next.

Example: 4-bit ripple-carry adder

- Full adder 0 adds bit 0 of A and B (Cin = 0).
- Full adder 1 adds bit 1 of A and B plus the carry from adder 0.
- Full adder 2 adds bit 2 of A and B plus the carry from adder 1.
- Full adder 3 adds bit 3 of A and B plus the carry from adder 2.
- The final carry-out is the overflow bit.

Limitation: The carry "ripples" through each adder in sequence, which takes time. This limits the speed of addition for large numbers.

6.4 Sequential Logic Circuits

Combinational vs Sequential Circuits

Combinational	**Sequential**
Output depends only on current inputs	Output depends on current inputs and previous state
No memory	Has memory
Examples: adders, decoders, multiplexers	Examples: flip-flops, registers, counters

Flip-Flops

Flip-flops are the basic memory element of sequential circuits. Each flip-flop stores a single bit.

Types of Flip-Flops

SR (Set-Reset) Flip-Flop

S	**R**	**Q (next state)**
0	**0**	**Hold current value**
0	**1**	**Reset (Q = 0)**

S	R	Q (next state)
1	0	Set (Q = 1)
1	1	Invalid state

- Set input = 1 stores a 1.
- Reset input = 1 stores a 0.
- Both 0 = hold current value.
- Both 1 = invalid state (avoided).

D (Data) Flip-Flop

D	Q (next state)
0	0
1	1

- Stores whatever value is on the D input when a clock pulse occurs.
- Avoids the invalid state of the SR flip-flop.
- Used in registers and memory.

JK Flip-Flop

J	K	Q (next state)
0	0	Hold current value
0	1	Reset (Q = 0)
1	0	Set (Q = 1)
1	1	Toggle (flip)

- An improved SR flip-flop with no invalid state.
- When J = K = 1, it toggles (flips) its stored value.

T (Toggle) Flip-Flop

T	Q (next state)
0	Hold current value
1	Toggle (flip)

- Changes (toggles) its stored state on every clock pulse when T = 1.
- Used in counters.

The Clock Signal

A clock signal is a regular pulse that synchronises when sequential circuits update their stored value. It ensures all parts of a system change state together.

Key points:

- The clock produces a continuous series of pulses.
- Flip-flops update their value on a clock edge (rising or falling).
- Clock speed is measured in Hertz (Hz). Modern CPUs run at GHz speeds.

Applications of Sequential Circuits

Application	**Description**
Registers	**A set of flip-flops storing multiple bits (e.g., 8-bit register = 8 flip-flops)**
Counters	**Circuits that count clock pulses. Used in clocks, timers, and frequency dividers**
Memory units	**Large arrays of flip-flops forming RAM and other memory**
Shift registers	**Registers that move data left or right on each clock pulse**
State machines	**Circuits that move through a sequence of states**

Review Questions

1. Define digital logic and explain the two states used.
2. Draw the truth table for a 2-input AND gate, OR gate, and XOR gate.
3. What is the output of a NAND gate when both inputs are 1?
4. What is the output of a NOR gate when both inputs are 0?
5. Write the Boolean expression for: (a) AND, (b) OR, (c) NOT, (d) NAND, (e) XOR.
6. State De Morgan's First Law and Second Law.
7. Use De Morgan's Laws to simplify: NOT(A + B).
8. Explain the difference between a combinational circuit and a sequential circuit.
9. Design a circuit where output Q = 1 only when inputs A and B are both 1. Draw the truth table and write the Boolean expression.
10. Design a circuit where output Q = 1 when inputs A and B are the same. Draw the truth table and write the Boolean expression.
11. What is a half adder? Draw its truth table and write its Boolean expressions.
12. What is a full adder? How does it differ from a half adder?
13. Explain how a ripple-carry adder works.
14. What is a flip-flop? What does it store?
15. Describe the operation of an SR flip-flop. What is its invalid state?
16. How does a D flip-flop differ from an SR flip-flop?
17. What is the special feature of a JK flip-flop when J = K = 1?
18. What is a clock signal and why is it needed in sequential circuits?
19. Give two applications of sequential circuits.

20. How many flip-flops are needed to build an 8-bit register?

Chapter 7: Exploring Computer Organization and Architecture

Chapter Overview

This chapter introduces students to the internal organisation and architecture of computers. It begins by distinguishing between computer architecture (what a computer does) and computer organisation (how it does it). Students will learn about the Von Neumann and Harvard architectures, advanced processor designs including CISC, RISC, multi-core processors, and pipelining. The chapter then covers system communication through buses and the system clock, the components of the processor and the machine cycle, interrupts and polling, parallel processing, and finally assembly instructions and addressing modes. This chapter forms the bridge between digital logic (Chapter 6) and computer memory (Chapter 8).

Learning Objectives

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

1. Distinguish between computer architecture and computer organisation.
2. Describe the Von Neumann architecture and explain the Von Neumann bottleneck.
3. Describe the Harvard architecture and its advantages over Von Neumann.
4. Explain the characteristics of CISC and RISC processors.
5. Describe multi-core processors and pipelining.
6. Explain the role of data, address, and control buses.
7. Explain the purpose of the system clock and how clock speed affects performance.
8. Identify the main CPU registers and their functions.
9. Describe the fetch-decode-execute cycle in detail.

10. Explain interrupts and the interrupt handling process.

11. Compare interrupts with polling.

12. Define parallel processing and give examples.

13. Describe assembly language and common instruction types.

14. Explain the main addressing modes with examples.

Key Terms

Term	**Definition**
Computer architecture	The conceptual design and functional behaviour of a computer system (what it does)
Computer organisation	How the hardware components are arranged and implemented (how it does it)
Von Neumann architecture	Classic model where a single memory stores both data and instructions
Von Neumann bottleneck	The limitation caused by sharing one bus between instructions and data
Harvard architecture	Architecture with separate memories and buses for instructions

Term	Definition
	and data
CISC	Complex Instruction Set Computer
RISC	Reduced Instruction Set Computer
Multi-core processor	A single chip containing two or more independent processing units
Pipelining	Overlapping stages of instruction execution to increase throughput
Bus	A set of parallel wires carrying signals between components
Data bus	Carries data and instructions between components
Address bus	Carries memory addresses from the CPU
Control bus	Carries control signals that coordinate component activity
Bus width	The number of bits a bus can carry at once

Term	Definition
System clock	Generates regular pulses that synchronise operations
Clock speed	The number of cycles per second, measured in Hz (GHz)
Program Counter (PC)	Holds the address of the next instruction to fetch
Memory Address Register (MAR)	Holds the address of the memory location being accessed
Memory Data Register (MDR)	Holds data being transferred to or from memory
Current Instruction Register (CIR)	Holds the instruction currently being decoded or executed
Accumulator (ACC)	Holds results of ALU calculations
Fetch-decode-execute cycle	The machine cycle by which the CPU processes instructions
Interrupt	A signal that temporarily suspends the current program so the CPU can handle a more urgent task

Term	Definition
Interrupt Service Routine (ISR)	The code that handles an interrupt
Polling	The CPU repeatedly checks devices to see if they need attention
Parallel processing	Performing multiple computations simultaneously
Assembly language	A low-level programming language using mnemonics
Assembler	Translates assembly language into machine code
Opcode	The part of an instruction that specifies the operation
Operand	The part of an instruction that specifies the data or address
Addressing mode	The way an instruction specifies where to find its operand
Immediate addressing	The operand is the actual value
Direct addressing	The operand is the memory address of the value

Term	Definition
Indirect addressing	The operand is the address of an address
Indexed addressing	The address is calculated by adding an offset to a base address

7.1 Introduction to Computer Organisation and Architecture

Architecture vs Organisation

These two terms are related but distinct:

Computer Architecture	Computer Organisation
The conceptual design and functional behaviour	How the hardware components are arranged and implemented
What the computer does	How it does it
Includes instruction set, addressing modes, data types	Includes control signals, memory technology, bus structure
Visible to the programmer	Transparent to the programmer

Computer Architecture	Computer Organisation
Example: whether the CPU has a multiply instruction	Example: how the multiplier circuit is built

Analogy: Architecture is like the blueprint of a house (number of rooms, layout). Organisation is like the actual construction (bricks, wiring, plumbing).

Von Neumann Architecture

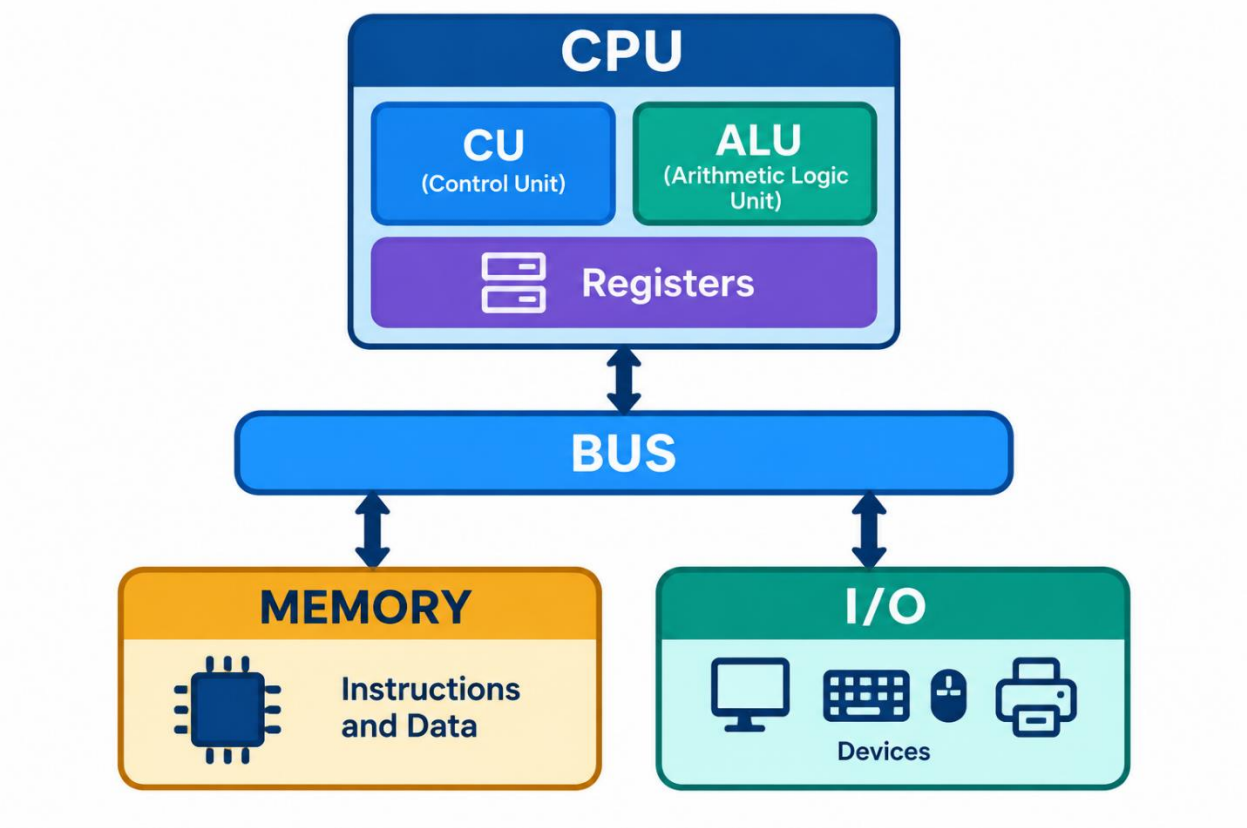
The **Von Neumann architecture** is the classic model used by most modern computers. It was proposed by John von Neumann in 1945.

Key features:

- A single memory stores both data and program instructions together.
- The CPU fetches instructions and data from the same memory, one at a time, via a shared bus.
- Instructions and data are stored in the same address space.
- The CPU processes instructions sequentially (one after another).

Main components:

- CPU (Control Unit, ALU, registers)
- Main Memory (stores both instructions and data)
- Input/Output devices
- System bus connecting them



The Von Neumann Bottleneck

Because the CPU and memory share a single bus, the CPU can be held up waiting for the shared bus to transfer data or instructions. This limits overall speed. This limitation is known as the **Von Neumann bottleneck**.

How it is addressed:

- Cache memory (reduces memory access time)
- Pipelining (overlaps instruction stages)
- Multi-core processors (parallel processing)
- Harvard architecture (separate buses)

Harvard Architecture

The **Harvard architecture** uses separate memories and buses for instructions and data.

Key features:

- Separate instruction memory and data memory.
- Separate buses for instructions and data.
- The CPU can fetch both an instruction and data simultaneously.
- Improves speed compared to Von Neumann.

Advantages:

- Faster execution (simultaneous fetch).
- No bottleneck between instruction and data access.
- Better for real-time and embedded systems.

Disadvantages:

- More complex hardware (two memories, two buses).
- More expensive.
- Less flexible (memory cannot be shared between instructions and data).

Common use: Microcontrollers and embedded systems (e.g., ARM processors, AVR, PIC).

Comparison table:

Feature	**Von Neumann**	**Harvard**
Memory	Single, shared	Separate for instructions and data
Buses	One shared bus	Separate buses

Feature	Von Neumann	Harvard
Speed	Slower (bottleneck)	Faster (simultaneous access)
Complexity	Simpler	More complex
Cost	Lower	Higher
Use	General-purpose computers	Embedded systems, microcontrollers

7.2 Advanced Processor Architectures

CISC (Complex Instruction Set Computer)

Characteristics:

- Large set of complex instructions.
- Some instructions perform multi-step operations in one instruction.
- Fewer lines of code needed per program.
- Each instruction may take multiple clock cycles.
- Emphasis on hardware complexity.

Examples: Intel x86 processors, AMD processors.

Advantages:

- Fewer instructions per program.

- Compiler has less work to do.
- Backward compatible with older software.

Disadvantages:

- Complex hardware.
- Some instructions are rarely used.
- May be slower per instruction.

RISC (Reduced Instruction Set Computer)

Characteristics:

- Small set of simple instructions.
- Each instruction executes in one clock cycle.
- Relies on the compiler to combine simple instructions to perform complex tasks.
- Emphasis on software (compiler) complexity.
- Generally faster and more power-efficient.

Examples: ARM processors (used in smartphones), MIPS, PowerPC.

Advantages:

- Faster execution per instruction.
- Simpler hardware.
- More power-efficient.
- Better for pipelining.

Disadvantages:

- More instructions needed per program.

- Compiler has more work to do.
- May not be backward compatible.

Comparison table:

Feature	**CISC**	**RISC**
Instruction set	Large, complex	Small, simple
Instruction length	Variable	Fixed
Clock cycles per instruction	Multiple	Usually one
Pipelining	Harder	Easier
Hardware complexity	High	Low
Software (compiler) complexity	Low	High
Power consumption	Higher	Lower
Examples	Intel x86, AMD	ARM, MIPS

Multi-Core Processors

A multi-core processor is a single chip containing two or more independent processing units (cores).

Key points:

- Each core can execute its own instruction stream.
- Allows true parallel execution of multiple programs or threads.
- Improves performance for multitasking and parallel workloads.
- Common in modern computers (dual-core, quad-core, octa-core).

Advantages:

- Better multitasking performance.
- Faster for parallel workloads.
- Lower power consumption than multiple separate processors.

Challenges:

- Software must be written to take advantage of multiple cores.
- Communication between cores adds complexity.
- Not all tasks can be parallelised.

Pipelining

Pipelining is a technique where the CPU begins fetching or decoding the next instruction before the current one has finished executing. It overlaps the stages of the fetch-decode-execute cycle for multiple instructions.

Analogy: A factory assembly line. While one worker installs a wheel, another installs the engine on the next car.

Stages in a simple pipeline:

1. Fetch

2. Decode
3. Execute
4. Write back

Benefits:

- Increases instruction throughput (more instructions completed per unit time).
- Does not reduce the time for a single instruction, but increases overall speed.

Challenges:

- Hazards (data hazards, control hazards, structural hazards).
- Branch instructions can disrupt the pipeline.
- Requires careful design.

7.3 System Communication and Timing

Buses

Buses are sets of parallel wires that carry signals between components.

Bus	**Function**	**Direction**
Data bus	Carries actual data and instructions being transferred	Bidirectional
Address bus	Carries the memory address of where data should be read from or written to	Unidirectional (from CPU)

Bus	Function	Direction
Control bus	Carries control signals (read/write, clock signals) that coordinate component activity	Bidirectional

Bus width:

- Determines how much data can be transferred at once.
- A 64-bit data bus transfers 64 bits per cycle.
- Wider buses generally mean faster data transfer.

Address bus width and addressable memory:

- An address bus with n bits can address 2 to the power n unique memory locations.
- Example: A 32-bit address bus can address 2 to the power 32 locations (about 4 GB).
- A 64-bit address bus can address far more.

System Clock

The **system clock** generates regular electrical pulses that synchronise all operations in the computer.

Key points:

- Clock speed is measured in Hertz (Hz). Modern CPUs run at GHz (billions of cycles per second).
- A higher clock speed generally means more instructions can be processed per second.
- Clock speed is not the only factor affecting performance (architecture, cores, cache also matter).

- The clock ensures all components change state together.

Clock cycle:

- One complete pulse.
- Instructions may take one or more clock cycles.

7.4 The Processor, Its Components and the Machine Cycle

Key CPU Registers

Register	Full Name	Purpose
PC	Program Counter	Holds the address of the next instruction to be fetched
MAR	Memory Address Register	Holds the address of the memory location currently being accessed
MDR	Memory Data Register	Holds the data or instruction being transferred to or from memory
CIR	Current Instruction Register	Holds the instruction currently being decoded or executed
ACC	Accumulator	Holds the results of calculations performed by the ALU

The Fetch-Decode-Execute Cycle (Machine Cycle)

The CPU processes instructions in a repeating cycle.

Step 1: Fetch

- The address in the PC is copied to the MAR.
- The instruction at that address in memory is copied to the MDR, then to the CIR.
- The PC is incremented to point to the next instruction.

Step 2: Decode

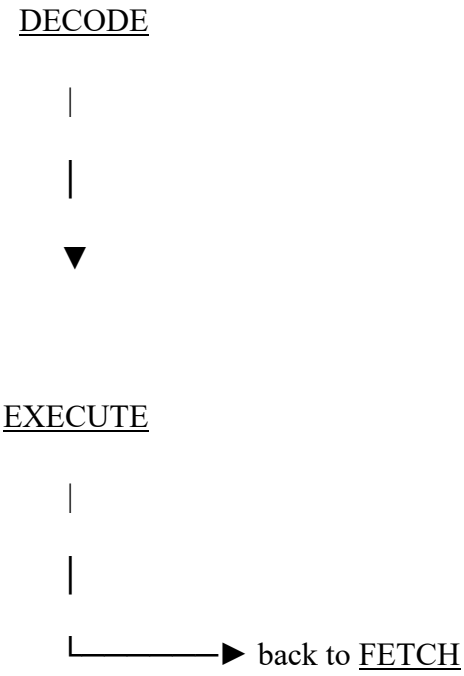
- The Control Unit interprets (decodes) the instruction in the CIR to determine what operation is required.

Step 3: Execute

- The instruction is carried out. For example:
    - The ALU performs a calculation.
    - Data is moved or stored.
    - A jump to another address occurs.

Step 4: Repeat

- The cycle repeats for the next instruction.



Detailed Example

Suppose the PC holds address 100, and the instruction at address 100 is "LOAD 200" (load the value at address 200 into the accumulator).

1. **Fetch:** PC (100) goes to MAR. Memory at address 100 is read into MDR. MDR goes to CIR. PC becomes 101.
2. **Decode:** CU reads "LOAD 200" and determines this is a load operation.
3. **Execute:** Address 200 goes to MAR. Memory at address 200 is read into MDR. MDR goes to ACC.
4. **Repeat:** PC (101) is fetched next.

7.5 Interrupts, Polling and Parallel Processing

Interrupts

An **interrupt** is a signal sent to the CPU (usually by hardware or software) that temporarily suspends the currently running program so the CPU can service a more urgent task. After handling the interrupt, the CPU resumes the original program.

Types of interrupts:

- **Hardware interrupts:** from devices (keyboard press, mouse click, disk ready, timer).
- **Software interrupts:** from programs (system calls, errors, exceptions).

Interrupt handling steps:

1. CPU finishes the current instruction.
2. CPU saves its current state (registers, PC) onto the stack.
3. CPU jumps to the Interrupt Service Routine (ISR) to handle the interrupt.
4. ISR runs.
5. CPU restores the saved state.
6. CPU resumes the original program.

Program running → Interrupt occurs → Save state → Run ISR → Restore state → Resume program

Advantages of interrupts:

- CPU only reacts when needed (efficient).
- Allows real-time response.
- Frees the CPU to do other work while waiting.

Disadvantages:

- More complex to implement.

- Requires careful handling (priority, nesting).

Polling

Polling is an alternative to interrupts. The CPU repeatedly checks (polls) each device in turn to see if it needs attention, rather than waiting to be interrupted.

Advantages of polling:

- Simpler to implement.
- Predictable timing.
- No interrupt overhead.

Disadvantages:

- Wastes CPU time checking devices that do not need service.
- Less efficient, especially with many devices.
- Slower response time.

Comparison table:

Feature	**Interrupts**	**Polling**
How it works	Device signals CPU	CPU checks devices repeatedly
Efficiency	High (CPU only reacts when needed)	Low (wastes time checking)
Complexity	More complex	Simpler

Feature	Interrupts	Polling
Response time	Fast	Depends on polling cycle
Best for	Many devices, real-time systems	Few devices, simple systems

Parallel Processing

Parallel processing is performing multiple computations or instructions simultaneously, rather than one after another (serial processing).

How it is achieved:

- Multiple CPU cores.
- Multiple processors.
- Specialised hardware (GPUs for graphics, DSPs for signals).

Requirements:

- Tasks must be breakable into independent sub-tasks.
- Communication and synchronisation between tasks.

Advantages:

- Faster execution for suitable tasks.
- Better use of hardware.
- Handles large workloads.

Challenges:

- Not all tasks can be parallelised.
- Overhead of coordination.
- Software must be designed for parallelism.

Examples:

- Weather forecasting models.
- Video rendering.
- Scientific simulations.
- Database queries.
- Machine learning training.

7.6 Assembly Instructions and Addressing Modes

Assembly Language

Assembly language is a low-level, human-readable programming language that uses short mnemonics to represent machine code instructions. It is translated into machine code by an **assembler**.

Key points:

- One assembly instruction usually corresponds to one machine code instruction.
- Specific to a particular CPU family.
- Used for embedded systems, device drivers, and performance-critical code.

Typical instruction format:

OPCODE OPERAND

Example:

LDA 45

This means: Load the value at memory address 45 into the Accumulator.

Common Instruction Types

Type	Examples	Description
Data movement	LOAD, STORE, MOV	Move data between memory and registers
Arithmetic	ADD, SUB, MUL, DIV, INC, DEC	Perform arithmetic operations
Logical	AND, OR, NOT, XOR	Perform logical operations
Control flow	JUMP, JMP, JZ, JNZ, CALL, RET	Change the order of execution
Input/Output	IN, OUT	Transfer data to or from devices
Comparison	CMP	Compare two values
Stack operations	PUSH, POP	Manipulate the stack

Addressing Modes

Addressing modes are the different ways an instruction can specify where to find the data (operand) it needs.

Mode	Description	Example	Meaning
Immediate	The operand is the actual value	LDA #5	Load the number 5 directly
Direct	The operand is the memory address where the value is stored	LDA 45	Load the value found at address 45
Indirect	The operand is the address of a memory location which itself contains the address of the actual data	LDA (45)	Load the value at the address stored in address 45
Indexed	The address is calculated by adding an offset (in an index register) to a base address	LDA 45, X	Load the value at address (45 + X)

Examples Explained

Immediate addressing:

LDA #5

The value 5 is loaded directly into the accumulator. The # symbol indicates immediate.

Direct addressing:

LDA 45

The value stored at memory address 45 is loaded into the accumulator.

Indirect addressing:

LDA (45)

Memory address 45 contains another address (say 100). The value at address 100 is loaded into the accumulator. This is useful for pointers.

Indexed addressing:

LDA 45, X

The value at address (45 + X) is loaded into the accumulator, where X is the value in the index register. Useful for accessing elements of an array in sequence.

Why Addressing Modes Matter

- Different modes suit different tasks (constants, variables, arrays, pointers).
- They affect program size and speed.
- Understanding them is essential for low-level programming and compiler design.

Review Questions

1. Distinguish between computer architecture and computer organisation.
2. Describe the Von Neumann architecture. What is the Von Neumann bottleneck?
3. Describe the Harvard architecture. State two advantages over Von Neumann.
4. Compare CISC and RISC processors. Give one example of each.
5. What is a multi-core processor? State two advantages.
6. Explain what pipelining is and how it improves performance.
7. Name the three types of buses and state the function of each.
8. What is the difference between the data bus and the address bus?
9. How does bus width affect performance?
10. What is the purpose of the system clock?
11. Name five CPU registers and state what each holds.
12. Describe the fetch-decode-execute cycle in detail.
13. What is an interrupt? List the steps in interrupt handling.
14. Compare interrupts and polling.
15. Define parallel processing and give two examples of its use.
16. What is assembly language? How is it translated?
17. Give two examples each of data movement, arithmetic, and control flow instructions.
18. Explain the following addressing modes with an example: immediate, direct, indirect, indexed.
19. Why are addressing modes important?

20. How many memory locations can a 16-bit address bus address?

TERM TWO

Chapter 8: Managing Computer Memory

Chapter Overview

This chapter explores how computers manage memory. It begins with primary storage, covering RAM, ROM, and cache memory, and explains the memory hierarchy. Students will then learn about memory addressing concepts, including how addresses work, how address bus width determines addressable memory, word size, memory maps, and virtual memory. The chapter concludes with secondary storage technologies, comparing magnetic, solid-state, optical, flash, and cloud storage. Understanding memory is essential for understanding how programs run and how data is stored permanently.

Learning Objectives

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

1. Define primary storage and explain its role in a computer system.
2. Distinguish between RAM and ROM, and between DRAM and SRAM.
3. Explain the purpose and levels of cache memory.
4. Describe the memory hierarchy and the trade-offs it balances.
5. Define a memory address and explain how addresses are used.
6. Calculate the addressable memory from the address bus width.
7. Explain word size and its effect on performance.
8. Describe memory maps and their purpose.
9. Explain virtual memory and how paging works.
10. Describe secondary storage technologies: magnetic, solid-state, optical, flash, and cloud.

11. Compare storage technologies based on speed, capacity, cost, durability, portability, and power consumption.
12. Choose appropriate storage for a given task.

Key Terms

Term	Definition
Primary storage	Memory directly accessible by the CPU, holding data and instructions currently in use
RAM	Random Access Memory. Volatile main memory
ROM	Read-Only Memory. Non-volatile memory holding startup instructions
Volatile	Loses its contents when power is switched off
Non-volatile	Retains its contents without power
DRAM	Dynamic RAM. Needs constant refreshing. Used as main memory
SRAM	Static RAM. Does not need refreshing. Faster, more expensive. Used for cache

Term	Definition
Cache memory	Small, very fast SRAM storing frequently used data and instructions
L1 cache	Level 1 cache. Smallest, fastest, closest to the CPU core
L2 cache	Level 2 cache. Larger, slightly slower than L1
L3 cache	Level 3 cache. Largest, often shared between cores
Memory hierarchy	The arrangement of storage from fastest/smallest/most expensive to slowest/largest/cheapest
Memory address	A unique numerical label identifying each storage location in memory
Address bus width	The number of bits in the address bus, determining addressable memory
Addressable memory	The number of unique memory locations a CPU can address
Word size	The number of bits a CPU can process or transfer in one operation
Memory map	A conceptual layout showing how address space is divided

Term	Definition
Virtual memory	Using secondary storage as an extension of RAM
Paging	Swapping fixed-size blocks (pages) between RAM and disk
Page fault	Occurs when a program tries to access a page not currently in RAM
Secondary storage	Non-volatile storage for long-term data, not directly accessed by the CPU
HDD	Hard Disk Drive. Magnetic storage with spinning platters
SSD	Solid State Drive. Electronic storage with no moving parts
Optical storage	CDs, DVDs, Blu-ray discs read by laser
Flash storage	USB flash drives and memory cards using solid-state technology
Cloud storage	Data stored on remote servers accessed over the internet

8.1 Primary Storage

Primary storage (also called main memory) is memory directly accessible by the CPU. It holds data and instructions currently being processed.

RAM (Random Access Memory)

RAM is the main working memory of a computer.

Key characteristics:

- **Volatile:** Loses its contents when power is switched off.
- Used to temporarily hold the operating system, running programs, and data currently in use.
- Can be read from and written to.
- Access time is very fast compared to secondary storage.

Types of RAM:

Type	Full Name	Characteristics	Use
DRAM	Dynamic RAM	Needs constant refreshing (thousands of times per second). Cheaper. Slower than SRAM	Main memory
SRAM	Static RAM	Does not need refreshing. Faster. More expensive	CPU cache

Why DRAM needs refreshing: Each bit is stored as a charge in a tiny capacitor. The charge leaks away, so it must be re-written (refreshed) constantly.

Why SRAM is faster: Each bit is stored using flip-flops (see Chapter 6), which hold their state as long as power is supplied. No refreshing needed.

ROM (Read-Only Memory)

ROM is non-volatile memory that holds essential startup instructions.

Key characteristics:

- **Non-volatile:** Retains contents without power.
- Contains permanent instructions the computer needs at startup (BIOS/UEFI firmware, bootloader).
- Cannot normally be rewritten by the user during ordinary use.
- Variants like EEPROM and Flash can be reprogrammed with special procedures.

Uses of ROM:

- BIOS/UEFI firmware
- Bootloader
- Embedded system firmware
- Game cartridges (older systems)
- Calculators and appliances

Cache Memory

Cache memory is a small amount of very fast SRAM located on or near the CPU. It stores copies of frequently used data and instructions from RAM so the CPU does not have to wait for the slower main memory every time.

How cache works:

- When the CPU needs data, it first checks the cache.
- If the data is in the cache (a cache hit), it is retrieved very quickly.
- If not (a cache miss), the data is fetched from RAM and a copy is stored in the cache for next time.

Cache levels:

Level	Size	Speed	Location
L1	Smallest (typically 32-128 KB)	Fastest	Closest to the CPU core
L2	Larger (typically 256 KB to a few MB)	Fast	On the CPU chip
L3	Largest (several MB to tens of MB)	Slower than L1 and L2	Often shared between cores

Why cache improves performance: The CPU can access cache much faster than RAM. By keeping frequently used data close, the CPU spends less time waiting.

The Memory Hierarchy

The memory hierarchy arranges storage from fastest, smallest, and most expensive to slowest, largest, and cheapest.

From fastest to slowest:

1. CPU registers
2. Cache (L1, L2, L3)
3. RAM
4. Secondary storage (SSD, HDD)
5. Off-line/cloud storage

Fastest, Smallest, Most Expensive

Registers
L1 Cache
L2 Cache
L3 Cache
RAM
SSD / HDD
Cloud

Slowest, Largest, Cheapest

The trade-off: Faster memory is more expensive per byte, so computers use a small amount of fast memory (cache) and a large amount of slower memory (RAM and disk) to balance speed, cost, and capacity.

8.2 Memory Addressing Concepts

Memory Address

A **memory address** is a unique numerical label identifying each individual storage location in memory. It allows the CPU to read from or write to a specific location.

Key points:

- Each byte in memory has a unique address.
- Addresses are usually written in hexadecimal (e.g., 0x1A2B3C).
- The CPU uses the address bus to specify which address it wants to access.

Address Bus Width and Addressable Memory

An address bus with n bits can address 2 to the power n unique memory locations.

Formula: Addressable memory = 2^n locations.

Examples:

Address Bus Width	Addressable Locations	Approximate Memory
8 bits	2^8 = 256	256 bytes
16 bits	2^16 = 65,536	64 KB
20 bits	2^20 = 1,048,576	1 MB
32 bits	2^32 = 4,294,967,296	4 GB
64 bits	2^64	Far more than any current system needs

Word Size

Word size is the number of bits a CPU can process or transfer in one operation.

Examples:

- A 32-bit CPU processes 32 bits at a time.
- A 64-bit CPU processes 64 bits at a time.

Effects of word size:

- Larger word size means more data processed per operation.

- Affects the maximum addressable memory.
- Affects the size of registers and data buses.
- Larger word size generally means better performance for large computations.

Memory Maps

A **memory map** is a conceptual layout showing how the address space is divided between different uses.

Typical divisions:

- Ranges reserved for the operating system.
- Ranges for user programs.
- Ranges for I/O devices.
- Ranges for ROM (firmware).
- Ranges for video memory.

Why memory maps matter:

- They prevent programs from accessing memory they shouldn't.
- They allow the OS to manage memory efficiently.
- They help in debugging and system design.

Virtual Memory

Virtual memory is a memory-management technique that allows a computer to use space on secondary storage (such as the hard disk or SSD) as an extension of RAM.

How it works:

- Programs are divided into fixed-size blocks called **pages**.
- Pages are stored on disk when not in use.

- When a program needs a page, it is loaded into RAM (if not already there).
- Pages not currently needed can be swapped out to disk to make room.
- This is called **paging**.

Key terms:

Term	**Definition**
Page	A fixed-size block of memory
Page fault	Occurs when a program tries to access a page not currently in RAM
Swapping	Moving pages between RAM and disk
Page replacement	Choosing which page to remove from RAM when space is needed

Advantages of virtual memory:

- Allows programs larger than physical RAM to run.
- Allows more programs to run simultaneously.
- Provides memory protection between programs.

Disadvantages of virtual memory:

- Slower than physical RAM (disk access is much slower).
- Can cause disk thrashing if too much swapping occurs.
- Requires disk space.

8.3 Secondary Storage Technologies

Secondary storage is non-volatile storage used for the long-term storage of data and programs. It is not directly accessed by the CPU; data must first be loaded into RAM.

Magnetic Storage: Hard Disk Drives (HDD)

How HDDs work:

- Store data magnetically on spinning circular platters.
- Read and written by a moving read/write head.
- The platters spin at high speed (e.g., 5400 or 7200 RPM).

Characteristics:

- Relatively cheap per gigabyte.
- Large capacities (up to several terabytes).
- Slower than SSDs.
- More mechanically fragile (moving parts).
- Sensitive to shock and vibration.

Best for:

- Bulk storage.
- Backup.
- Archival.

- Cost-sensitive applications.

Solid-State Storage: Solid State Drives (SSD)

How SSDs work:

- Store data electronically using flash memory chips.
- No moving parts.

Characteristics:

- Much faster read/write speeds than HDDs.
- More durable and shock-resistant.
- Silent operation.
- Lower power consumption.
- More expensive per gigabyte than HDDs.
- Limited write cycles (though modern SSDs last many years).

Best for:

- Operating system drives.
- High-performance applications.
- Portable devices.
- Environments with vibration or shock.

Optical Storage

How optical storage works:

- CDs, DVDs, and Blu-ray discs store data as microscopic pits and lands.
- Read by a laser beam.

Characteristics:

- Slower and lower capacity than magnetic or solid-state options.
- Inexpensive to produce.
- Good for distributing software and media.
- Archival storage (though not as long-lasting as once thought).
- Susceptible to scratches and degradation.

Types:

Type	**Capacity**
CD	700 MB
DVD	4.7 GB (single layer), 8.5 GB (dual layer)
Blu-ray	25 GB (single layer), 50 GB (dual layer)

Flash and Removable Storage

USB flash drives and memory cards (SD cards) use the same solid-state technology as SSDs, in a small, portable, removable form.

Characteristics:

- Portable.
- Durable (no moving parts).
- Fast enough for most uses.
- Limited capacity compared to HDDs and SSDs.

- Can be lost or stolen easily.

Best for:

- Transferring files between computers.
- Portable storage.
- Cameras, phones, and other devices.

Cloud Storage

Cloud storage stores data on remote servers accessed over the internet.

Examples: Google Drive, iCloud, OneDrive, Dropbox.

Characteristics:

- Accessible from any device with internet.
- Offloads backup and maintenance to the service provider.
- Depends on internet connectivity.
- Trust in the provider's security is essential.
- Ongoing subscription costs may apply.
- Privacy concerns (data stored on someone else's servers).

Best for:

- Backup.
- Collaboration.
- Accessing files from multiple devices.
- Off-site storage.

Comparing Storage Technologies

Technology	Speed	Capacity	Cost per GB	Durability	Portability	Power Use
HDD	Moderate	Very high	Low	Low (moving parts)	Low (internal)	Moderate
SSD	Very high	High	Moderate	High	Moderate	Low
Optical	Low	Low to moderate	Very low	Moderate	High	Low
Flash	High	Moderate	Moderate	High	Very high	Low
Cloud	Depends on internet	Very high	Subscription	High (provider dependent)	Very high (with internet)	Low (local)

Choosing Storage for a Task

Consider the following factors:

- **Speed:** How fast does data need to be accessed?
- **Capacity:** How much data needs to be stored?
- **Cost:** What is the budget?

- **Durability:** Will the storage be moved or exposed to harsh conditions?
- **Portability:** Does it need to be carried around?
- **Power consumption:** Is battery life important?
- **Reliability:** How critical is the data?
- **Connectivity:** Is internet access available?

Examples:

- Operating system: SSD (fast boot, fast program loading).
- Bulk media storage: HDD (large capacity, low cost).
- Transferring files: USB flash drive (portable, fast).
- Backup: Cloud storage or external HDD.
- Distributing a movie: Optical disc or streaming.

Review Questions

1. Define primary storage and give two examples.
2. Distinguish between RAM and ROM.
3. Explain the difference between DRAM and SRAM. Where is each used?
4. What is cache memory and why is it important?
5. Name the three levels of cache and describe each.
6. Draw or describe the memory hierarchy from fastest to slowest.
7. What is a memory address?
8. How many memory locations can a 24-bit address bus address?
9. What is word size and how does it affect performance?
10. What is a memory map? Why is it useful?
11. Define virtual memory. Explain how paging works.
12. What is a page fault?
13. State two advantages and two disadvantages of virtual memory.
14. Describe how a hard disk drive stores data.
15. Describe how a solid-state drive stores data.
16. Compare HDD and SSD in terms of speed, durability, cost, and power consumption.
17. Name three types of optical storage and their typical capacities.
18. What is cloud storage? State two advantages and two disadvantages.
19. Compare USB flash drives and external HDDs for transferring files.
20. You are buying storage for a video editing workstation. Which type of storage would you choose and why?

Chapter 9: Understanding Peripheral Communication and Data Transfer

Chapter Overview

This chapter explains how computers communicate with peripheral devices such as keyboards, printers, and external drives. It begins with peripheral devices and interfaces, covering the different types of peripherals, the ports and interfaces used to connect them, and the role of device drivers. The chapter then covers the I/O controller, explaining how it acts as a middleman between the CPU and slow peripherals, and describes three data transfer methods: programmed I/O, interrupt-driven I/O, and Direct Memory Access (DMA). It also explains buffering and why it is needed. Understanding this chapter helps students appreciate how a computer manages the huge speed differences between its internal components and external devices.

Learning Objectives

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

1. Define peripheral devices and classify them into input, output, storage, and communication peripherals.
2. Describe common interfaces and ports used to connect peripherals.
3. Distinguish between serial and parallel interfaces.
4. Explain the role of device drivers.
5. Explain the purpose of an I/O controller.
6. Describe the functions of an I/O controller.
7. Explain programmed I/O, interrupt-driven I/O, and Direct Memory Access (DMA).
8. Compare the three data transfer methods.
9. Explain buffering and why it is needed.
10. Describe the concept of spooling with an example.

Key Terms

Term	Definition
Peripheral device	Hardware connected to a computer but not part of the core CPU or memory
Input peripheral	Device that sends data into the computer (e.g., keyboard, mouse)
Output peripheral	Device that receives data from the computer (e.g., monitor, printer)
Storage peripheral	Device for storing data externally (e.g., external hard drive, USB flash drive)
Communication peripheral	Device for connecting to networks (e.g., modem, network adapter)
Interface	A connection point and standard for connecting peripherals
Port	A physical socket on a computer for connecting devices
USB	Universal Serial Bus. A common general-purpose interface

Term	Definition
HDMI	High-Definition Multimedia Interface. For digital video and audio
DisplayPort	Interface for digital video and audio
Ethernet	Interface for wired network connections
Bluetooth	Wireless interface for short-range connections
Wi-Fi	Wireless interface for network connections
Serial interface	Sends data one bit at a time over a single wire
Parallel interface	Sends multiple bits simultaneously over multiple wires
Device driver	Software that allows the OS to communicate with a specific hardware device
I/O controller	Hardware that manages communication between the CPU/system bus and a peripheral
Buffer	A small area of memory used to temporarily hold data during transfer

Term	Definition
Programmed I/O	The CPU actively directs and waits for every step of a data transfer
Interrupt-driven I/O	The device signals the CPU via an interrupt when it needs attention
DMA	Direct Memory Access. A controller transfers data directly between a peripheral and memory without CPU involvement
DMA controller	Hardware that manages DMA transfers
Buffering	Temporarily storing data in a buffer to smooth out differences in data rates
Spooling	A type of buffering where data is queued for a device (e.g., a printer)

9.1 Peripheral Devices and Interfaces

What Are Peripheral Devices?

Peripheral devices are hardware components connected to a computer but not part of the core CPU or memory. They extend the computer's functionality.

Classification of Peripherals

Type	Description	Examples
Input peripherals	Send data into the computer	Keyboard, mouse, scanner, microphone, webcam, barcode reader, touchscreen, joystick
Output peripherals	Receive data from the computer	Monitor, printer, speakers, projector, plotter
Storage peripherals	Store data externally	External hard drives, USB flash drives, memory cards, external SSDs
Communication peripherals	Connect to networks or other devices	Modem, network adapter (NIC), Bluetooth adapter

Some devices are both input and output (e.g., touchscreens, multifunction printers, network adapters).

Interfaces and Ports

Interfaces are the physical and logical connection points and standards used to connect peripherals to a computer.

Interface	Description	Typical Use
USB (Universal Serial Bus)	Most common general-purpose interface. Supports many device types and plug-	Keyboards, mice, printers, flash drives, phones

Interface	Description	Typical Use
	and-play	
HDMI	Digital video and audio interface	Monitors, projectors, TVs
DisplayPort	Digital video and audio interface	Monitors, high-resolution displays
Ethernet (RJ45)	Wired network connection	Routers, switches, network cards
Bluetooth	Wireless short-range interface	Headphones, keyboards, mice, phones
Wi-Fi	Wireless network interface	Network connections
Thunderbolt	High-speed interface for data and video	External drives, displays, docking stations
VGA	Older analogue video interface	Older monitors and projectors
Audio jacks	Analogue audio connections	Headphones,

Interface	Description	Typical Use
		microphones, speakers
Serial (RS-232)	Older serial interface	Legacy devices, industrial equipment
Parallel (IEEE 1284)	Older parallel interface	Older printers, scanners

USB in Detail

USB is the most widely used interface today.

Key features:

- Plug-and-play: Devices are detected and configured automatically.
- Hot-swappable: Devices can be connected or removed while the computer is running.
- Power supply: USB can provide power to devices (e.g., charging phones, powering small drives).
- Backward compatible: Newer USB versions support older devices.
- Multiple device support: Up to 127 devices can be connected via hubs.

USB versions:

Version	Speed

Version	Speed
USB 1.1	Up to 12 Mbps
USB 2.0	Up to 480 Mbps
USB 3.0 / 3.1 / 3.2	Up to 5-20 Gbps
USB4	Up to 40 Gbps

Serial vs Parallel Interfaces

Feature	Serial	Parallel
Data transfer	One bit at a time	Multiple bits simultaneously
Wires	Single data wire (plus control)	Multiple data wires
Distance	Better over long distances	Limited over long distances

Feature	Serial	Parallel
Interference	**Less susceptible**	**More susceptible**
Speed	**Historically slower, now very fast**	**Historically faster over short distances**
Modern use	**USB, SATA, PCIe (all serial)**	**Largely obsolete (was used for printers, older drives)**

Note: Modern high-speed interfaces like USB, SATA, and PCIe are all serial. Parallel interfaces are now largely replaced.

Device Drivers

Device drivers are software that allow the operating system to communicate with and control a specific hardware peripheral.

Role of a device driver:

- Translates general OS commands into device-specific instructions.
- Manages data transfer between the OS and the device.
- Handles device-specific features and settings.
- Provides an interface for applications to use the device.

Why drivers are needed:

- Different devices have different command sets and protocols.

- The OS cannot include code for every possible device.
- Drivers allow new devices to be added without changing the OS.

Examples: Printer driver, graphics driver, network driver, sound driver, keyboard driver.

9.2 The I/O Controller

The Problem: Speed Mismatch

Peripherals operate at very different (and generally much slower) speeds than the CPU. For example:

- CPU: billions of cycles per second (GHz).
- Keyboard: a few characters per second.
- Printer: a few pages per minute.
- Hard disk: hundreds of megabytes per second.

An I/O controller manages this speed mismatch and communication so the CPU is not constantly held up waiting for slow devices.

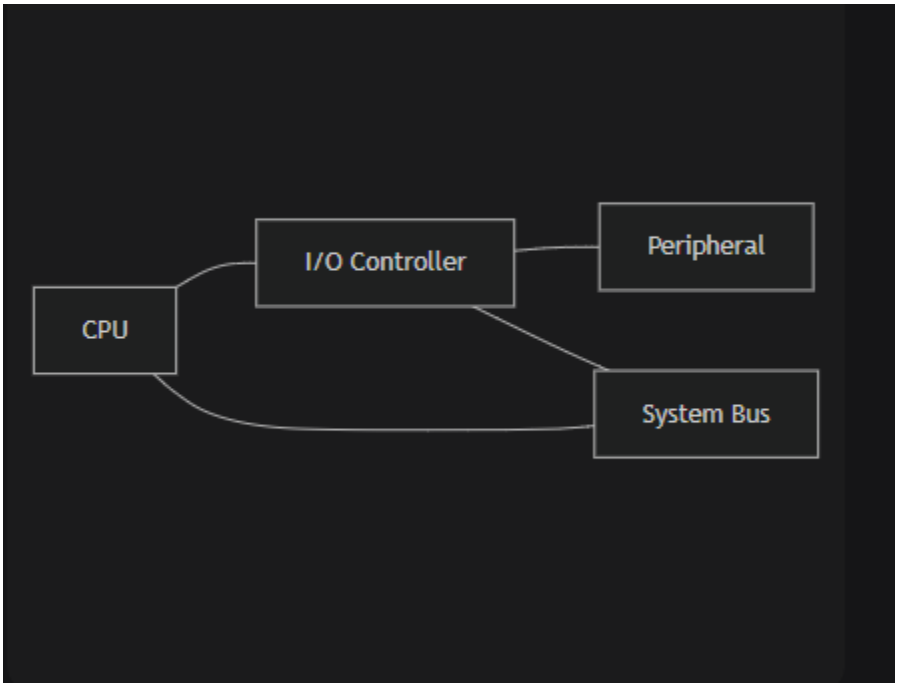
Role of the I/O Controller

The I/O controller (also called the interface) acts as a middleman between the CPU/system bus and a peripheral device.

Functions of an I/O controller:

- Acts as a middleman between the CPU/system bus and a peripheral device.
- Converts data between the format used internally by the computer and the format required by the specific device.
- Contains its own small buffer/registers to temporarily hold data being transferred.

- Signals the CPU (via an interrupt) when a device is ready, or when a data transfer is complete.
- Manages the timing and control of data transfer.
- May perform error checking.



Data Transfer Methods

There are three main methods for transferring data between the CPU and peripherals.

1. Programmed I/O

The CPU actively directs and waits for every step of the data transfer.

How it works:

- The CPU sends a command to the I/O controller.
- The CPU repeatedly checks (polls) the controller to see if the device is ready.
- When ready, the CPU transfers the data.

- The CPU waits for the transfer to complete before doing anything else.

Advantages:

- Simple to implement.
- Predictable.

Disadvantages:

- Wastes CPU time (the CPU is idle while waiting).
- Inefficient for slow devices.
- Related to polling (see Chapter 7).

2. Interrupt-Driven I/O

The device signals the CPU (via an interrupt) only when it needs attention, freeing the CPU to do other work in the meantime.

How it works:

- The CPU sends a command to the I/O controller.
- The CPU continues executing other instructions.
- When the device is ready, the I/O controller sends an interrupt to the CPU.
- The CPU suspends its current program, saves its state, and handles the interrupt (transfers the data).
- The CPU resumes its original program.

Advantages:

- Much more efficient than programmed I/O.
- CPU is not wasted waiting.
- Good for slow devices.

Disadvantages:

- More complex to implement.
- Interrupt overhead (saving and restoring state).
- Can be disrupted by frequent interrupts.

3. Direct Memory Access (DMA)

A specialised DMA controller transfers data directly between a peripheral and main memory without the CPU's continuous involvement.

How it works:

- The CPU sets up the DMA controller with the source, destination, and amount of data.
- The DMA controller takes over the system bus and transfers data directly between the peripheral and memory.
- The CPU is free to do other work during the transfer.
- The CPU is interrupted only once, when the whole transfer is complete.

Advantages:

- Very efficient for large transfers.
- CPU is free to do other work.
- Much faster for bulk data transfer (e.g., loading a large file from disk).

Disadvantages:

- Requires a DMA controller.
- The DMA controller takes over the bus, which can slow the CPU's memory access.
- More complex.

Example: Loading a large file from disk into memory. The CPU sets up the DMA transfer and continues other work. The DMA controller moves the file. When done, the CPU is interrupted once.

Comparison of Data Transfer Methods

Feature	**Programmed I/O**	**Interrupt-Driven I/O**	**DMA**
CPU involvement	Continuous	Only when interrupted	Only at start and end
CPU efficiency	Low	Moderate	High
Speed	Slow	Moderate	Fast
Complexity	Simple	Moderate	Complex
Best for	Simple, slow devices	Slow devices, interactive use	Large, fast transfers
Example	Reading a keyboard	Reading a mouse	Loading a file from disk

Buffering

Buffering is temporarily storing data in a small area of memory (a buffer) to smooth out differences in the rate at which data is produced and consumed.

Why buffering is needed:

- The CPU produces data much faster than a printer can print.
- A network connection may deliver data in bursts, not at a steady rate.
- A keyboard produces data slowly, but the CPU can process it quickly.

How buffering works:

- Data is placed in a buffer by the producer (e.g., CPU).
- The consumer (e.g., printer) takes data from the buffer at its own pace.
- If the buffer fills up, the producer must wait.
- If the buffer empties, the consumer must wait.

Example: A print buffer (spooler) holds documents waiting to be printed. The CPU can send a document to the printer and continue working. The printer prints at its own speed from the buffer.

Spooling

Spooling (Simultaneous Peripheral Operations On-Line) is a type of buffering where data is queued for a device.

How it works:

- Jobs (e.g., print jobs) are placed in a queue (on disk or in memory).
- The device (e.g., printer) processes them one at a time.
- The CPU can continue other work.
- Users can manage the queue (cancel jobs, change order).

Example: Print spooling. When multiple users send documents to a shared printer, the documents are queued and printed one after another. The CPU doesn't wait for each print job to finish.

Benefits of spooling:

- Frees the CPU from waiting for slow devices.
- Allows multiple jobs to be queued.
- Users can manage their print jobs.
- Improves overall system efficiency.
- where jobs are queued for a device (e.g., printer).

Review Questions

1. Define a peripheral device and give two examples of each type: input, output, storage, communication.
2. Name four common interfaces used to connect peripherals.
3. State four features of USB.
4. Distinguish between serial and parallel interfaces. Which is used more today?
5. What is a device driver and why is it needed?
6. What is an I/O controller and what problem does it solve?
7. List four functions of an I/O controller.
8. Describe programmed I/O. State one advantage and one disadvantage.
9. Describe interrupt-driven I/O. State one advantage and one disadvantage.
10. Describe Direct Memory Access (DMA). State one advantage and one disadvantage.
11. Compare the three data transfer methods in a table.
12. What is buffering and why is it needed?
13. Explain what a print buffer (spooler) does.
14. What is spooling? State two benefits.
15. A user is loading a large video file from a hard disk. Which data transfer method would be most efficient and why?

Chapter 10: Designing Algorithms

Chapter Overview

This chapter introduces students to the fundamental skill of algorithm design. It begins with the definition and characteristics of an algorithm, then covers the main approaches to designing algorithms: top-down design, bottom-up design, and structured design. Students will learn how to represent algorithms using pseudocode, flowcharts, structured English, and decision tables. The chapter then covers the three basic control structures, sequence, selection, and repetition, with detailed examples of each. It concludes with subroutines, explaining procedures and functions, and parameter passing, covering pass by value and pass by reference. This chapter is essential because algorithms are the foundation of all computer programs.

Learning Objectives

1. By the end of this chapter, students should be able to:
2. Define an algorithm and state its key characteristics.
3. Describe the main approaches to algorithm design: top-down, bottom-up, and structured.
4. Outline the general problem-solving process.
5. Represent algorithms using pseudocode, flowcharts, structured English, and decision tables.
6. Identify and use the three basic control structures: sequence, selection, and repetition.
7. Write algorithms using IF, IF-ELSE, FOR, WHILE, and REPEAT-UNTIL structures.
8. Choose the appropriate loop structure for a given problem.
9. Define subroutines and distinguish between procedures and functions.
10. Explain the benefits of using subroutines.
11. Distinguish between formal and actual parameters.
12. Explain pass by value and pass by reference with examples.

Key Terms

Term	Definition
Algorithm	**A finite, precise, step-by-step sequence of instructions for solving a problem**
Top-down design	**Breaking a large problem into smaller sub-problems until each is simple enough to solve**
Stepwise refinement	**Another name for top-down design**
Bottom-up design	**Starting with small components and building them up into a complete solution**
Structured design	**Using only sequence, selection, and repetition to keep programs logical**
Pseudocode	**A structured, plain-language description of an algorithm using programming-like keywords**
Flowchart	**A diagram using standard symbols to show the steps and decisions in an algorithm**
Structured English	**A structured text representation of an algorithm**
Decision table	**A tabular representation of an algorithm's logic, useful for many combinations of conditions**
Sequence	**Instructions carried out one after another**
Selection	**Choosing between different paths based on a condition**

Term	Definition
Repetition	**Executing a set of instructions multiple times**
FOR loop	**A count-controlled loop that repeats a fixed number of times**
WHILE loop	**A pre-condition loop that repeats while a condition is true**
REPEAT-UNTIL loop	**A post-condition loop that repeats until a condition becomes true**
Infinite loop	**A loop that never terminates**
Off-by-one error	**Looping one time too many or too few**
Subroutine	**A named, self-contained block of code that performs a specific task**
Procedure	**A subroutine that performs an action but does not return a value**
Function	**A subroutine that returns a single value**
Parameter	**A value passed into a subroutine**
Formal parameter	**The variable name listed in the subroutine's definition**
Actual parameter	**The real value supplied when the subroutine is called**
Pass by value	**A copy of the value is passed; changes do not affect the original**

Term	Definition
Pass by reference	**The memory address is passed; changes affect the original**

10.1 Algorithm Design Approaches

What is an Algorithm?

An **algorithm** is a finite, precise, step-by-step sequence of instructions for solving a problem or carrying out a task.

Key characteristics of a good algorithm:

Characteristic	Meaning
Unambiguous	Each step is clear and has only one meaning
Finite	It must terminate after a limited number of steps
Correct	It must produce the right answer
Effective	Each step must be doable
Efficient	It should use resources wisely (time, memory)

Example of a simple algorithm (making tea):

1. Start
2. Boil water
3. Place tea bag in cup

4. Pour boiled water into cup
5. Wait 3 minutes
6. Remove tea bag
7. Add milk and sugar if desired
8. End

This is an algorithm because it is a finite, precise sequence of steps.

Approaches to Designing Algorithms

1. Top-Down Design (Stepwise Refinement)

Break a large problem into smaller, more manageable sub-problems, and break those down further, until each sub-problem is simple enough to solve directly.

Example: Designing a school report card system.

- Level 1: Generate report cards
- Level 2: Get student data, calculate averages, determine grades, print report
- Level 3: For "calculate averages": get marks, sum marks, divide by number of subjects
- Level 4: For "sum marks": initialise total to 0, loop through marks, add each to total

Advantages:

- Easier to understand and manage.
- Each sub-problem can be solved independently.
- Allows multiple programmers to work on different parts.

2. Bottom-Up Design

Start with the smallest, simplest components or functions and build them up into the complete solution.

Example: Building a calculator.

- Start with basic functions: add, subtract, multiply, divide.

- Combine them into more complex functions: calculate area, calculate interest.
- Combine those into a full calculator program.

Advantages:

- Components are tested early.
- Reusable modules are identified.
- Good when the low-level details are well understood.

3. Structured Design

Using only three control structures (sequence, selection, repetition) to keep programs logical and easy to follow, avoiding uncontrolled jumps (such as GOTO).

Benefits:

- Programs are easier to read and understand.
- Programs are easier to debug and maintain.
- Logic is clear and predictable.

General Problem-Solving Process

1. **Understand the problem.** What is the input? What is the required output? What are the constraints?
2. **Plan a solution.** Design the algorithm using top-down or bottom-up design.
3. **Implement it.** Write the code in a programming language.
4. **Test it.** Check with normal, boundary, and invalid data.
5. **Maintain and improve it.** Fix bugs, add features, optimise.

10.2 Representation of Algorithms

Algorithms can be represented in several ways before being coded.

Pseudocode

Pseudocode is a structured, plain-language description of an algorithm's steps, using programming-like keywords (such as IF, WHILE, FOR, INPUT, OUTPUT) without following the strict syntax of any one programming language.

Advantages:

- Easy to read and understand.
- Easy to translate into any real programming language.
- Focuses on logic, not syntax.

Example (find the largest of three numbers):

```
INPUT a, b, c
IF a > b THEN
   IF a > c THEN
      largest = a
   ELSE
      largest = c
   ENDIF
ELSE
   IF b > c THEN
      largest = b
   ELSE
      largest = c
   ENDIF
ENDIF
OUTPUT largest
```

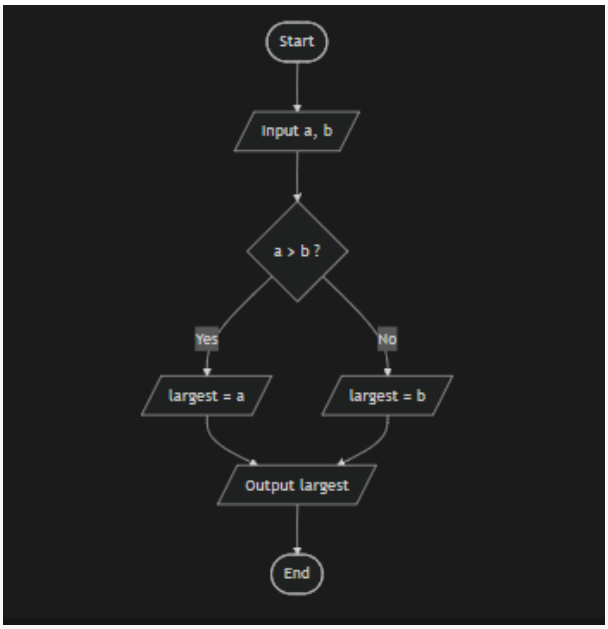
Flowcharts

Flowcharts are diagrams using standard symbols to show the sequence of steps and decisions in an algorithm.

Standard flowchart symbols:

Symbol	Meaning
Oval	Start/End (terminator)
Rectangle	Process/action step
Diamond	Decision (yes/no branch)
Parallelogram	Input/Output
Arrow	Flow direction
Circle	Connector (to another part of the chart)

Example flowchart (find the largest of two numbers):



Structured English

Structured English is a structured text representation of an algorithm, similar to pseudocode but using more natural language.

Example:

1. Get the student's marks for all subjects.

2. Add the marks together to get the total.

3. Divide the total by the number of subjects to get the average.

4. If the average is 50 or more, output "Pass".

5. Otherwise, output "Fail".

Decision Tables

Decision tables are a tabular representation of an algorithm's logic, particularly useful where there are many combinations of conditions.

Example: Deciding what to wear based on weather.

Condition 1: Is it raining?	Yes	Yes	No	No
Condition 2: Is it cold?	Yes	No	Yes	No
Action: Wear raincoat	X	X		
Action: Wear coat	X		X	
Action: Wear light clothes				X

10.3 Algorithm Design Strategies; Sequence and Selection Structures

The Three Basic Control Structures

All algorithms can be built from three basic control structures:

1. **Sequence** - instructions carried out one after another.
2. **Selection** - choosing between different paths based on a condition.
3. **Repetition** - executing instructions multiple times.

Sequence

Instructions are carried out one after another, in the exact order they are written.

Example:

INPUT length

INPUT width

area = length * width

```
OUTPUT area
```

Each step happens in order: input length, input width, calculate area, output area.

Selection

The algorithm chooses between different paths based on a condition.

IF-THEN

Do something only if a condition is true.

```
IF temperature > 30 THEN
   OUTPUT "It is hot"
ENDIF
```

IF-THEN-ELSE

Choose between two alternatives.

```
IF score >= 50 THEN
   OUTPUT "Pass"
ELSE
   OUTPUT "Fail"
ENDIF
```

Nested IF

An IF inside another IF.

```
IF age >= 18 THEN
   IF hasLicense = TRUE THEN
      OUTPUT "You can drive"
   ELSE
```

```
        OUTPUT "You need a license"
    ENDIF
ELSE
    OUTPUT "You are too young to drive"
ENDIF
```

CASE / SWITCH

Selecting among several fixed values.

```
CASE grade OF
    A: OUTPUT "Excellent"
    B: OUTPUT "Good"
    C: OUTPUT "Average"
    DEFAULT: OUTPUT "Try again"
ENDCASE
```

Design Strategies for Selection

- Identify all the necessary conditions and decision points clearly.
- Ensure every possible case/path is handled, including an ELSE/default case.
- Avoid unnecessary duplication of logic.
- Test the algorithm by tracing through it with sample data (a dry run/trace table).

10.4 Repetition Structures

Repetition (looping) structures allow a set of instructions to be executed multiple times.

FOR Loop (Count-Controlled Loop)

Repeats a fixed, known number of times, using a counter variable.

```
text

FOR i = 1 TO 10
   OUTPUT i
ENDFOR
```

This outputs 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

When to use: When the number of repetitions is known in advance.

WHILE Loop (Pre-Condition / Pre-Test Loop)

Repeats while a condition remains true. The condition is checked before each execution, so the loop body may run zero times.

```
count = 0

WHILE count < 10

   count = count + 1

   OUTPUT count

ENDWHILE
```

When to use: When the number of repetitions is unknown and the loop might need to run zero times.

REPEAT-UNTIL Loop (Post-Condition / Post-Test Loop)

Repeats until a condition becomes true. The condition is checked after each execution, so the loop body always runs at least once.

```
REPEAT

   INPUT password

UNTIL password = correctPassword
```

When to use: When the loop must run at least once (e.g., validating user input).

Choosing the Right Loop

Loop	Condition Checked	Minimum Runs	Best For
FOR	Before each iteration, using a counter	0 or more (fixed)	Known number of repetitions
WHILE	Before each iteration	0	Unknown repetitions, may not run
REPEAT-UNTIL	After each iteration	1	Must run at least once

Common Pitfalls

- **Infinite loops:** Forgetting to update the condition variable inside the loop.
- **Off-by-one errors:** Looping one time too many or too few.
- **Example of an infinite loop:**
- text
- count = 0
- WHILE count < 10
- OUTPUT count
- REM count is never incremented
- ENDWHILE
- **Correction:** Add `count = count + 1` inside the loop.

10.5 Subroutines

- A **subroutine** is a named, self-contained block of code that performs a specific task and can be called (invoked) from elsewhere in a program.

- ### Procedures vs Functions

Procedure	**Function**
Performs an action	Returns a value
Does not return a value directly	Returns a single value
Called as a statement	Called within an expression
Example: print a report	Example: calculate the average

- **Example procedure (pseudocode):**

```
PROCEDURE PrintGreeting(name)
  OUTPUT "Hello, " + name
ENDPROCEDURE
```

- **Example function (pseudocode):**

```
FUNCTION Add(a, b)
  RETURN a + b
ENDFUNCTION
```

Benefits of Using Subroutines

- **Avoids repeating code:** Write once, use many times.
- **Easier to maintain:** Fix a bug in one place.
- **Easier to read:** Each subroutine has a clear, single purpose.
- **Allows teamwork:** Different programmers can work on different subroutines.
- **Enables reuse:** Subroutines can be reused in other programs (a library).
- **Easier to test:** Each subroutine can be tested independently.

10.6 Parameter Passing

Parameters (arguments) are values passed into a subroutine so it can work with data supplied by the calling code.

Formal vs Actual Parameters

Formal Parameters	Actual Parameters
Listed in the subroutine's definition	Supplied when the subroutine is called
Variable names	Real values
Example: PROCEDURE Add(a, b)	Example: Add(5, 3)

Methods of Passing Parameters

Pass by Value

A copy of the actual parameter's value is passed to the subroutine. Any changes made to the parameter inside the subroutine do not affect the original variable in the calling program.

Example:

```
PROCEDURE Double(x)
   x = x * 2
   OUTPUT x
ENDPROCEDURE

number = 5
Double(number)
OUTPUT number
```

Output:

```
10
5
```

The procedure doubled its local copy (x), but the original number remains 5.

2. Pass by Reference

The memory address of the actual parameter is passed. The subroutine can directly access and modify the original variable, so changes made inside the subroutine do affect the original value.

Example:

```
PROCEDURE Double(BY REF x)
   x = x * 2
   OUTPUT x
ENDPROCEDURE
```

```
number = 5

Double(number)

OUTPUT number
```

Output:

```
10

10
```

The procedure modified the original variable through its reference.

Comparison Table

Feature	Pass by Value	Pass by Reference
What is passed	A copy of the value	The memory address
Effect on original	Unchanged	Changed
Safety	Safer (no unintended changes)	Less safe
Efficiency	Copying large data is slow	Faster (no copying)
Common use	Default in many languages	Used when changes are needed

When to Use Each

- **Pass by value:** When the subroutine should not modify the original data, or when data is small.
- **Pass by reference:** When the subroutine needs to modify the original data, or when passing large data structures for efficiency.

Review Questions

1. Define an algorithm and list five characteristics of a good algorithm.
2. Explain top-down design with an example.
3. Explain bottom-up design with an example.
4. What is structured design and why is it useful?
5. List the five steps of the general problem-solving process.
6. What is pseudocode? State two advantages.
7. Draw and label five standard flowchart symbols.
8. What is a decision table and when is it useful?
9. Name and describe the three basic control structures.
10. Write pseudocode using IF-THEN-ELSE to check if a number is positive or negative.
11. Write pseudocode using a FOR loop to output the numbers 1 to 20.
12. Write pseudocode using a WHILE loop to keep asking for a password until the correct one is entered.
13. Explain the difference between a WHILE loop and a REPEAT-UNTIL loop.
14. When would you use a FOR loop instead of a WHILE loop?
15. What is an infinite loop and how can it be avoided?
16. Define a subroutine and distinguish between a procedure and a function.
17. State four benefits of using subroutines.
18. Distinguish between formal and actual parameters.
19. Explain pass by value and pass by reference with an example of each.
20. When would you use pass by reference instead of pass by value?

Chapter 11: Designing Modular Algorithms

Chapter Overview

This chapter builds on the algorithm design skills from Chapter 10 and focuses on modularity, the practice of breaking programs into smaller, independent modules (subroutines). It begins by explaining what happens when a subroutine is called, including the role of the call stack and the difference between local and global variables. The chapter then covers recursion in detail: what it is, its essential components (base case and recursive case), how it uses the call stack, and how it compares with iteration. Finally, it examines the limitations and complexity of recursion, including stack overflow, memory overhead, performance overhead, and repeated calculations. By the end of this chapter, students will understand how to design well-structured, modular programs and when to use recursion versus iteration.

Learning Objectives

- By the end of this chapter, students should be able to:
- Explain what happens when a subroutine is called.
- Describe the role of the call stack in subroutine calls.
- Distinguish between local and global variables.
- Explain modularity and its benefits.
- Define recursion and explain its essential components.
- Trace a recursive algorithm using the call stack.
- Compare recursion with iteration.
- Explain the limitations and risks of recursion.
- Identify when to prefer recursion and when to prefer iteration.
- Describe how recursion can be optimised using memoisation.

Key Terms

Term	Definition
Modularity	**Designing a program as a collection of small, independent, well-defined subroutines**
Module	**A self-contained subroutine that handles one part of a task**
Call stack	**A Last-In-First-Out (LIFO) data structure used to track active subroutine calls**
Stack frame	**The block of data pushed onto the call stack for each subroutine call**
Return address	**The address of the instruction to return to after a subroutine finishes**
Local variable	**A variable declared inside a subroutine; only accessible while that subroutine runs**
Global variable	**A variable declared outside any subroutine; accessible from anywhere in the program**
Recursion	**A technique where a subroutine calls itself to solve a problem**
Base case	**The simplest instance of a problem, solved directly without further recursion**
Recursive case	**The part where the subroutine calls itself with a smaller version of the problem**
Stack overflow	**When the call stack runs out of memory due to too many recursive calls**
Memoisation	**Storing already-computed results to avoid recalculating them**

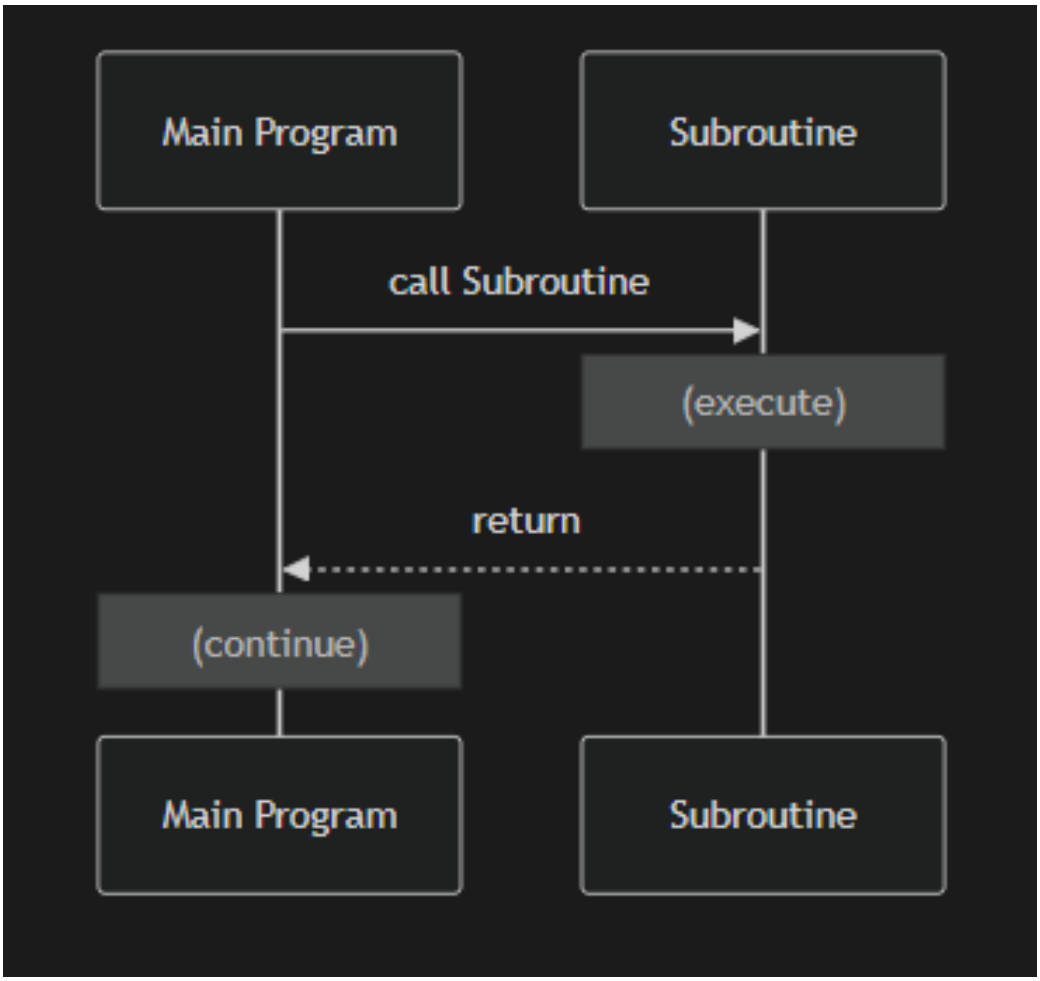
Term	Definition
Iteration	**Repeating a block of code using a loop**
Divide and conquer	**A strategy that breaks a problem into smaller sub-problems, solves them, and combines results**
Fibonacci sequence	**A sequence where each number is the sum of the two previous numbers**
Factorial	**The product of all positive integers up to a given number (e.g., 4! = 4 x 3 x 2 x 1 = 24)**

11.1 Processing Subroutine Calls

What Happens When a Subroutine Is Called

When a subroutine is called, the following steps occur:

1. The current state of the program (including the address of the instruction to return to) is saved, typically onto a stack (a call stack).
2. Control transfers to the first instruction of the subroutine.
3. The subroutine executes using its own local variables and parameters.
4. When the subroutine finishes (reaches a RETURN), the saved return address is retrieved from the stack, and control returns to the instruction immediately after the original call.



The Call Stack

The **call stack** is a Last-In-First-Out (LIFO) data structure used to keep track of active subroutine calls.

Key points:

- Each call adds (pushes) a new **stack frame** onto the stack.
- A stack frame contains:
    - The return address.
    - The parameters passed to the subroutine.

    - The local variables of the subroutine.
- Returning from a subroutine removes (pops) its frame from the stack.
- The top of the stack always represents the currently executing subroutine.

Example:

Main calls Sub1

Sub1 calls Sub2

Sub2 calls Sub3

Call stack (top to bottom):

Sub3 frame

Sub2 frame

Sub1 frame

Main frame

When Sub3 finishes, its frame is popped, and control returns to Sub2.

Local vs Global Variables

Local Variables	**Global Variables**
Declared inside a subroutine	Declared outside any subroutine
Only exist and are accessible while that subroutine executes	Accessible from anywhere in the program

Local Variables	**Global Variables**
Each call creates fresh local variables	Exist for the entire program run
Safer (no unintended side effects)	Can be changed unexpectedly from multiple places
Use less memory (created and destroyed)	Use memory for the whole program
Preferred for most programming	Use sparingly

Example:

```
global total = 0          // global variable


PROCEDURE AddToTotal(x)
   LOCAL temp             // local variable
   temp = x * 2
   total = total + temp   // modifies global
ENDPROCEDURE
```

Why local variables are preferred:

- They reduce the risk of unintended changes.

- They make subroutines independent and reusable.
- They make debugging easier.

Modularity and Modular Programming

Modularity (or modular programming) is designing a program as a collection of small, independent, well-defined subroutines (modules), each handling one part of the overall task.

This is the practical application of top-down design (Chapter 10).

Benefits of modularity:

- **Reusability:** Modules can be used in multiple programs.
- **Maintainability:** Fixing a bug in one module fixes it everywhere.
- **Readability:** Each module has a clear, single purpose.
- **Teamwork:** Different programmers can work on different modules.
- **Testability:** Each module can be tested independently.
- **Abstraction:** Users of a module only need to know what it does, not how.

Example: A school management system might be modularised as:

- Student registration module
- Attendance tracking module
- Grade calculation module
- Report generation module
- Fee management module

Each module can be developed, tested, and maintained independently.

11.2 Recursion

What is Recursion?

Recursion is a technique where a subroutine (function) calls itself, either directly or indirectly, in order to solve a problem by breaking it down into smaller instances of the same problem.

Direct recursion: A function calls itself directly.

```
FUNCTION Factorial(n)
   IF n = 0 THEN
      RETURN 1
   ELSE
      RETURN n * Factorial (n - 1)
   ENDIF
ENDFUNCTION
```

Indirect recursion: Function A calls Function B, which calls Function A.

Essential Components of a Correct Recursive Algorithm

Component	Description
Base case	The simplest instance of the problem, solved directly without further recursive calls. This stops the recursion
Recursive case	The part where the subroutine calls itself with a smaller/simpler version of the original problem, moving it closer to the base case

Without a base case, recursion never stops and causes a stack overflow.

Classic Example: Factorial

The factorial of a non-negative integer n (written n!) is the product of all positive integers up to n.

- 0! = 1 (by definition)
- 1! = 1
- 2! = 2 x 1 = 2
- 3! = 3 x 2 x 1 = 6
- 4! = 4 x 3 x 2 x 1 = 24

Recursive definition:

```
FUNCTION Factorial(n)
   IF n = 0 THEN
      RETURN 1                // base case
   ELSE
      RETURN n * Factorial(n - 1)  // recursive case
   ENDIF
ENDFUNCTION
```

Tracing Factorial(4):

Factorial(4)

= 4 * Factorial(3)

= 4 * (3 * Factorial(2))

= 4 * (3 * (2 * Factorial(1)))

= 4 * (3 * (2 * (1 * Factorial(0))))

= 4 * (3 * (2 * (1 * 1)))

= 4 * (3 * (2 * 1))

= 4 * (3 * 2)

= 4 * 6

= 24

How Recursion Uses the Call Stack

Each recursive call pushes a new stack frame onto the call stack. When the base case is reached, the calls "unwind" one by one, popping frames off the stack and combining results, back up to the original call.

Example: Call stack for Factorial(4):

Call Factorial(4)

Call Factorial(3)

Call Factorial(2)

Call Factorial(1)

Call Factorial(0) -> returns 1

returns 1 * 1 = 1

returns 2 * 1 = 2

returns 3 * 2 = 6

returns 4 * 6 = 24

At the deepest point, the stack contains frames for Factorial(4), Factorial(3), Factorial(2), Factorial(1), and Factorial(0).

Another Classic Example: Fibonacci Sequence

The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Each number is the sum of the two previous numbers.

Recursive definition:

```
FUNCTION Fibonacci(n)
   IF n = 0 THEN
      RETURN 0
   ELSE IF n = 1 THEN
      RETURN 1
   ELSE
      RETURN Fibonacci(n - 1) + Fibonacci(n - 2)
   ENDIF
ENDFUNCTION
```

Note: This naive recursive Fibonacci is very inefficient because it recalculates the same values many times (see 11.3).

Recursion vs Iteration

Feature	Recursion	Iteration
Definition	Function calls itself	Loop repeats a block
Termination	Base case	Loop condition
Memory use	Higher (stack frames)	Lower (no extra frames)
Speed	Often slower	Often faster
Code clarity	Elegant for naturally recursive problems	Clear for simple repetition
Risk	Stack overflow	Infinite loop
Best for	Trees, graphs, divide and conquer	Simple repetition, counting

Key point: Any problem solvable recursively can also be solved iteratively (using a loop) and vice versa. The choice depends on clarity, efficiency, and the nature of the problem.

When Recursion is Naturally Suited

- Tree traversal (e.g., file systems, XML/HTML parsing)
- Graph traversal (e.g., depth-first search)
- Divide-and-conquer algorithms (e.g., merge sort, quicksort)

- Binary search
- Mathematical sequences (factorial, Fibonacci)
- Tower of Hanoi
- Parsing nested structures

11.3 Recursion Limitations and Complexity

Limitations and Risks of Recursion

1. Stack Overflow

If a recursive function has no base case, an incorrect base case, or simply recurses too deeply, the call stack can run out of memory, crashing the program.

Example: A function that calls itself without a base case.

```
FUNCTION InfiniteRecursion(n)
  RETURN InfiniteRecursion(n + 1)   // no base case!
ENDFUNCTION
```

This will cause a stack overflow.

2. Memory Overhead

Each recursive call consumes stack memory for its own stack frame (return address, local variables). Deep recursion can therefore use significantly more memory than an equivalent loop.

Example: Factorial(1000) requires 1000 stack frames.

3. Performance Overhead

The overhead of repeatedly calling a subroutine (saving/restoring state) can make recursive solutions slower than iterative ones for simple tasks.

4. Repeated Calculation

Naive recursive solutions to some problems (e.g., a simple recursive Fibonacci function) can recompute the same values many times, making them highly inefficient unless optimised.

Example: Fibonacci(5) calls Fibonacci(4) and Fibonacci(3). Fibonacci(4) calls Fibonacci(3) and Fibonacci(2). Fibonacci(3) is calculated multiple times.

Number of calls for Fibonacci(n):

n	**Calls**
5	15
10	177
20	21,891
30	2,692,537

This grows exponentially (approximately 2^n), making naive recursive Fibonacci impractical for large n.

Optimising Recursion: Memoisation

Memoisation is storing already-computed results to avoid recalculating them.

Example: Fibonacci with memoisation:

```
CREATE array memo of size n+1, initialised to -1


FUNCTION Fibonacci(n)
    IF memo[n] != -1 THEN
```

```
    RETURN memo[n]          // already computed
  ENDIF
  IF n = 0 THEN
    memo[n] = 0
  ELSE IF n = 1 THEN
    memo[n] = 1
  ELSE
    memo[n] = Fibonacci(n - 1) + Fibonacci(n - 2)
  ENDIF
  RETURN memo[n]
ENDFUNCTION
```

With memorization, each Fibonacci number is calculated only once, reducing time complexity from O(2^n) to O(n).

When to Prefer Recursion

- When a problem is naturally defined in terms of smaller versions of itself.
- When working with tree or graph structures.
- When using divide-and-conquer algorithms.
- When code clarity is prioritised.
- When the expected recursion depth is manageable.

When to Prefer Iteration

- When performance and memory efficiency are critical.
- When recursion depth could be very large or unbounded.
- When the problem is naturally iterative (simple repetition).
- When stack overflow is a risk.

Recursion in Real-World Algorithms

Algorithm	**Recursive?**	**Why**
Merge sort	Yes	Divide and conquer
Quicksort	Yes	Divide and conquer
Binary search	Yes (or iterative)	Halving the search space
Tree traversal	Yes	Naturally recursive structure
Depth-first search	Yes	Exploring branches
Tower of Hanoi	Yes	Naturally recursive
Factorial	Yes (or iterative)	Simple recursion
Fibonacci	Yes (better with memoisation)	Overlapping sub-problems

Review Questions

1. Describe what happens when a subroutine is called.
2. What is the call stack and why is it needed?
3. What is a stack frame? What does it contain?
4. Distinguish between local and global variables, giving two advantages of local variables.
5. Define modularity and state four benefits of modular programming.
6. Define recursion and explain its two essential components.
7. What happens if a recursive function has no base case?
8. Trace the recursive Factorial(5) function, showing all steps.

9. Explain how recursion uses the call stack, using Factorial(3) as an example.

10. Compare recursion and iteration in a table.

11. Give three examples of problems naturally suited to recursion.

12. Explain stack overflow and how it can occur.

13. Explain why naive recursive Fibonacci is inefficient.

14. What is memoisation and how does it improve recursive Fibonacci?

15. When should you prefer recursion? When should you prefer iteration?

16. What is the time complexity of naive recursive Fibonacci? What is it with memoisation?

17. Write a recursive function to calculate the sum of numbers from 1 to n.

18. Write a recursive function to calculate the power of a number (e.g., 2^5).

19. Explain the difference between direct and indirect recursion.

20. What is divide and conquer? Give two examples of algorithms that use it.

TERM THREE

Chapter 12: Understanding Programming Language Concepts

Chapter Overview

This chapter introduces students to the fundamental concepts of programming languages. It begins with the basics: what a programming language is, the different levels of programming languages (low-level and high-level), and the key concepts common to most languages. The chapter then covers programming paradigms, including procedural, object-oriented, functional, and declarative programming. Next, it explains language translators (assemblers, compilers, and interpreters) and compares compilers with interpreters. Finally, it covers program errors and debugging, including syntax errors, logic errors, and run-time errors, and the techniques used to find and fix them. This chapter provides the conceptual foundation for the practical programming work in Chapter 13.

Learning Objectives

- By the end of this chapter, students should be able to:
- Define a programming language and explain syntax and semantics.
- Distinguish between low-level and high-level programming languages.
- Identify key programming concepts common to most languages.
- Define a programming paradigm and describe the main paradigms.
- Explain the characteristics of procedural, object-oriented, functional, and declarative programming.
- Describe the role of assemblers, compilers, and interpreters.
- Compare compilers and interpreters.
- Explain hybrid translation approaches such as bytecode and JIT compilation.
- Identify and distinguish between syntax errors, logic errors, and run-time errors.

- Describe common debugging techniques.

Key Terms

Term	Definition
Programming language	**A formal, structured language with syntax and semantics used to write instructions for a computer**
Syntax	**The grammar rules of a programming language**
Semantics	**The meaning of instructions in a programming language**
Low-level language	**A language close to the hardware (machine code, assembly language)**
High-level language	**A language closer to human language (Python, C, Java)**
Machine code	**Binary instructions the CPU executes directly**
Assembly language	**A low-level language using mnemonics**
Variable	**A named storage location for data**
Data type	**The kind of data a variable can hold (integer, real, string, boolean)**
Operator	**A symbol that performs an operation (+, -, *, /)**
Control structure	**A construct that controls the flow of execution (sequence, selection, repetition)**

Term	Definition
Paradigm	**A fundamental style or approach to programming**
Procedural programming	**A paradigm where a program is a sequence of instructions organised into procedures**
Object-oriented programming (OOP)	**A paradigm organised around objects that bundle data and behaviour**
Class	**A blueprint for creating objects**
Object	**An instance of a class**
Encapsulation	**Hiding internal details of an object**
Inheritance	**A class inheriting properties from another class**
Polymorphism	**Objects of different classes treated through a common interface**
Functional programming	**A paradigm built by composing pure functions**
Pure function	**A function that always gives the same output for the same input and has no side effects**
Declarative programming	**A paradigm where the programmer states what result is wanted, not how**
Translator	**Software that converts source code into machine code**
Assembler	**Translates assembly language into machine code**

Term	Definition
Compiler	**Translates the entire high-level program into machine code before running**
Interpreter	**Translates and executes a high-level program line by line at run time**
Bytecode	**An intermediate form of code between source code and machine code**
JIT	**Just-In-Time compilation. Compiling bytecode to machine code at run time**
Syntax error	**A violation of the language's grammar rules**
Logic error	**The program runs but produces incorrect results**
Run-time error	**An error that occurs while the program is executing**
Debugging	**The process of finding and fixing errors in a program**
Breakpoint	**A point in the code where execution pauses for inspection**
Trace table	**A table used to record variable values during a dry run**

12.1 Programming Language Basics

What is a Programming Language?

A **programming language** is a formal, structured language with strict syntax (grammar rules) and semantics (meaning), used to write instructions (a program) that a computer can execute.

Two key aspects:

- **Syntax:** The grammar rules. What the code looks like.
- **Semantics:** The meaning. What the code does.

A program with correct syntax but incorrect semantics will run but produce wrong results.

Levels of Programming Language

Low-Level Languages

Low-level languages are close to the hardware.

Language	Description
Machine code	Binary instructions the CPU executes directly
Assembly language	Mnemonic representation of machine code (see Chapter 7)

Advantages:

- Fast and efficient.
- Direct control over hardware.
- Small program size.

Disadvantages:

- Hard to write and read.
- Not portable between different CPU types.
- Error-prone.
- Time-consuming to develop.

High-Level Languages

High-level languages are closer to human/natural language.

Examples: Python, C, Java, JavaScript, C++, Ruby.

Advantages:

- Easier to read, write, and debug.
- More portable across different hardware.
- Faster to develop.
- Better abstraction.

Disadvantages:

- Must be translated before execution.
- Less direct control over hardware.
- May be less efficient than low-level code.

Comparison table:

Feature	**Low-Level**	**High-Level**
Closeness to hardware	Close	Far

Feature	Low-Level	High-Level
Readability	Difficult	Easy
Portability	Low	High
Speed	Fast	Slower (after translation)
Development time	Long	Short
Examples	Machine code, assembly	Python, C, Java

Key Programming Concepts Common to Most Languages

Concept	Description
Variables	Named storage for data
Data types	The kind of data (integer, real, string, boolean, character)
Operators	Symbols that perform operations (+, -, *, /, AND, OR, NOT)

Concept	Description
Control structures	Sequence, selection, repetition
Functions/procedures	Reusable blocks of code
Input/output	Reading data in and writing data out
Comments	Notes in the code ignored by the translator
Constants	Values that cannot change during execution

12.2 Programming Paradigms

A **programming paradigm** is a fundamental style or approach to structuring and writing programs.

Main Paradigms

1. Procedural / Imperative Programming

A program is a sequence of instructions/commands, organised into procedures/functions, executed step by step to change the program's state.

Key features:

- Focus on procedures (functions).
- Uses variables and state.

- Sequence, selection, and repetition.
- Top-down design.

Examples: C, Pascal, BASIC, Fortran.

Strengths:

- Simple and straightforward.
- Good for small, linear tasks.
- Efficient.

Weaknesses:

- Hard to manage for large, complex programs.
- Data and functions are separate.
- Difficult to model real-world entities.

2. Object-Oriented Programming (OOP)

A program is organized around objects, which bundle together data (attributes) and behaviour (methods).

Key concepts:

Concept	**Description**
Class	A blueprint for creating objects
Object	An instance of a class
Encapsulation	Hiding internal details; exposing only what is needed

Concept	Description
Inheritance	A class can inherit properties from another class
Polymorphism	Objects of different classes can be treated through a common interface
Abstraction	Focusing on essential features, hiding complexity

Example: A Car class with attributes (color, speed) and methods (accelerate, brake). A Sportscar class can inherit from Car and add its own features.

Examples: Java, Python, C++, C#, Ruby.

Strengths:

- Models real-world entities naturally.
- Reusable and extensible code.
- Easier to manage large, complex systems.
- Promotes modularity.

Weaknesses:

- Steeper learning curve.
- Can be overkill for simple tasks.
- Can be less efficient.

3. Functional Programming

A program is built by composing pure functions, functions that always give the same output for the same input and avoid changing external state.

Key features:

- Emphasises mathematical function evaluation.
- Avoids changing state and mutable data.
- Functions are first-class (can be passed around).
- Recursion is common.

Examples: Haskell, Lisp, Erlang. Many languages like Python support functional-style features.

Strengths:

- Easier to reason about (no side effects).
- Good for parallel and concurrent programming.
- Concise code.

Weaknesses:

- Less intuitive for beginners.
- May be less efficient for some tasks.
- Limited support in many mainstream languages.

4. Declarative Programming

The programmer states what result is wanted, rather than how to achieve it step by step.

Examples:

- SQL for database queries.
- HTML for describing page structure.
- Prolog for logic programming.

Example (SQL):

sql

SELECT name FROM students WHERE grade > 80;

You say what you want (names of students with grade above 80), not how to retrieve it.

Strengths:

- Concise and readable.
- Focus on what, not how.
- Often easier for non-programmers.

Weaknesses:

- Limited to specific domains.
- Less control over execution.
- Not suitable for all problems.

Choosing a Paradigm

The choice depends on the nature of the problem:

- **OOP** suits large, complex systems modelling real-world entities.
- **Procedural** suits small, straightforward, linear tasks.
- **Functional** suits mathematical computations and concurrent systems.
- **Declarative** suits database queries and markup.

Many modern languages (e.g., Python, JavaScript) support multiple paradigms.

12.3 Language Translators

Because computers can only execute machine code directly, high-level (and assembly) programs must be translated.

Types of Translators

1. Assembler

Translates assembly language source code directly into machine code (roughly one-to-one translation).

Example:

Assembly:

LDA 45

ADD 46

STA 47

Becomes machine code (binary) that the CPU can execute.

2. Compiler

Translates the entire high-level source code into machine code (an executable file) before the program is run.

Key features:

- Translation happens once.
- Produces a separate executable file.
- Produces fast-running programs.
- Any error stops the whole compilation.
- The compiled program is specific to the target CPU/OS.

Examples: C, C++.

Process:

Source Code -> [Compiler] -> Object Code -> [Linker] -> Executable

3. Interpreter

Translates and executes a high-level program line by line (or statement by statement) at run time, without producing a separate stand-alone executable file.

Key features:

- Translation happens every time the program runs.
- No separate executable file.
- Easier to test/debug interactively.
- Errors are reported immediately, and execution can proceed up to that point.
- Generally runs more slowly than compiled code.

Examples: Python (in its classic form), Ruby, JavaScript.

Process:

Source Code -> [Interpreter] -> Executes line by line

Comparing Compilers and Interpreters

Feature	**Compiler**	**Interpreter**
Translation	Whole program, once	Line-by-line, every run
Speed of execution	Faster	Slower
Error reporting	All errors listed after full compilation	Stops at first error found
Output	Separate executable file	No separate file; needs interpreter

Feature	Compiler	Interpreter
		each run
Portability of source	Must recompile for each platform	Same source can run wherever interpreter exists
Development	Slower to test	Faster to test interactively
Memory use	More during compilation	Less
Examples	C, C++	Python, Ruby, JavaScript

Hybrid Approaches

Many modern languages use a hybrid approach.

Bytecode and JIT compilation:

1. Source code is first compiled into an intermediate form called **bytecode**.
2. Bytecode is then interpreted or further compiled at run time.

Example (Java):

- Java source code is compiled to bytecode.
- Bytecode runs on the Java Virtual Machine (JVM).
- The JVM may use JIT compilation to convert hot code to native machine code at run time.

Example (Python):

- Python source is compiled to bytecode (.pyc files).
- The CPython interpreter executes the bytecode.

Benefits:

- Portability (bytecode runs on any platform with the right VM).
- Good performance (JIT compilation).
- Faster startup than pure interpretation.

12.4 Program Errors and Debugging

Types of Programming Errors

1. Syntax Errors

Violations of the language's grammar rules.

Examples:

- Missing colon after if in Python.
- Missing semicolon in C.
- Misspelled keyword.
- Unmatched brackets.

Detection: Detected by the compiler/interpreter before/during translation.

Effect: The program will not run until fixed.

Example (Python):

```
if x > 5
   print("Hello")
```

Missing colon after if x > 5. Syntax error.

2. Logic Errors

The program runs without crashing but produces incorrect results, because the algorithm/logic itself is flawed.

Examples:

- Using > instead of >=.
- Using + instead of -.
- Wrong formula.
- Loop running one time too many.

Detection: Hardest to detect, since no error message is given. Requires testing and tracing.

Example:

average = total / count + 1

The + 1 is a logic error if it should not be there.

3. Run-Time Errors

Errors that occur while the program is executing, causing the program to crash or stop unexpectedly.

Examples:

- Division by zero.
- Trying to access a file that does not exist.
- Running out of memory.
- Array index out of bounds.
- Type mismatch.

Detection: Detected at run time. Some languages (like Python) raise exceptions; others (like C) may crash.

Example:

result = 10 / 0

Division by zero. Run-time error.

Comparison Table

Error Type	When Detected	Program Runs?	Example
Syntax	During translation	No	Missing semicolon
Logic	During testing	Yes, but wrong results	Wrong formula
Run-time	During execution	Partially, then crashes	Division by zero

Debugging Techniques

Debugging is the process of finding and fixing errors in a program.

Technique	Description
Reading error messages	For syntax/run-time errors, the message often shows the line and cause
Tracing / dry running	Manually stepping through the code with sample values, recording variable values in a trace table
Using a debugger tool	Software that allows running code step-by-step, setting breakpoints, and inspecting variables

Technique	Description
Print/output statements	Temporarily inserting output statements to display variable values at key points
Testing with a range of data	Including normal, boundary (edge-case), and erroneous/invalid data
Code review	Having another programmer review the code
Rubber duck debugging	Explaining the code line by line to an inanimate object (or person) to spot errors

Using a Debugger

A debugger allows you to:

- Run the program step by step.
- Set **breakpoints** (pause execution at a chosen line).
- Inspect and watch variable values.
- Step into, over, or out of functions.
- View the call stack.

Example workflow:

1. Set a breakpoint at the start of a suspicious loop.
2. Run the program.
3. When the breakpoint is hit, inspect variable values.

4. Step through the loop one iteration at a time.

5. Identify where the values go wrong.

Testing with Different Data

Type of Data	**Description**	**Example**
Normal data	Typical valid input	Age = 25
Boundary data	Values at the edge of valid range	Age = 0, Age = 120
Erroneous/invalid data	Input the program should reject	Age = -5, Age = "abc"

Testing with all three types helps ensure the program handles all situations correctly.

Review Questions

1. Define a programming language and explain syntax and semantics.
2. Distinguish between low-level and high-level languages. Give two examples of each.
3. State three advantages and three disadvantages of high-level languages.
4. List five key programming concepts common to most languages.
5. Define a programming paradigm.
6. Describe procedural programming and give two examples of languages that use it.
7. Explain the four key concepts of OOP: encapsulation, inheritance, polymorphism, and abstraction.
8. Describe functional programming and give one example of a language that uses it.
9. Describe declarative programming and give two examples.
10. What is an assembler? What does it translate?
11. Describe how a compiler works. State two advantages and two disadvantages.
12. Describe how an interpreter works. State two advantages and two disadvantages.
13. Compare compilers and interpreters in a table.
14. Explain the hybrid approach used by Java and Python.
15. Distinguish between syntax errors, logic errors, and run-time errors. Give an example of each.
16. Why are logic errors the hardest to detect?
17. List five debugging techniques.
18. What is a breakpoint and how is it used?

19. Explain what a trace table is and how it helps debugging.

20. Why is it important to test with normal, boundary, and erroneous data?

Chapter 13: Introduction to C & Python Programming

Chapter Overview

This chapter introduces students to two of the most widely used programming languages: Python and C. It begins with an introduction to both languages, their histories, and their basic program structures. Students will learn how to write, run, and debug simple programs in both languages. The chapter covers variables and basic data types, operators and expressions, selection structures (if, else, elif, switch), iterative structures (for, while, do-while), and practical program development. It concludes with a mini-project that brings together algorithm design, language concepts, and Python/C syntax into one complete, working solution.

Learning Objectives

- By the end of this chapter, students should be able to:
- Describe the characteristics of Python and C.
- Write and run a simple "Hello, World!" program in both languages.
- Identify key structural differences between Python and C.
- Declare and use variables and basic data types in both languages.
- Use arithmetic, relational, logical, and assignment operators.
- Write selection structures using if, elif, else, and switch/case.
- Write iterative structures using for and while loops.
- Use do-while in C and its equivalent in Python.
- Write, test, and debug complete programs in Python and C.
- Apply good programming practices (meaningful names, comments, indentation).
- Identify and fix common beginner errors.
- Complete a mini-programming project using the full development process.

Key Terms

Term	Definition
Python	**A high-level, interpreted, general-purpose programming language**
C	**A high-level, compiled, general-purpose programming language**
Interpreted	**Translated and executed line by line at run time**
Compiled	**Translated entirely into machine code before running**
Indentation	**Spaces or tabs used in Python to define blocks of code**
Braces	**Curly brackets { } used in C to define blocks of code**
Semicolon	**Statement terminator used in C**
Dynamically typed	**Variable type inferred at run time (Python)**
Statically typed	**Variable type declared before use (C)**
Integer	**Whole number data type**
Float	**Decimal number data type**
String	**Sequence of characters**
Boolean	**True/false data type**
Character	**Single symbol data type**

Term	Definition
Operator	**Symbol that performs an operation**
Expression	**Combination of values and operators that produces a result**
Operator precedence	**The order in which operators are evaluated**
Selection	**Choosing between different paths based on a condition**
Iteration	**Repeating a block of code**
for loop	**Count-controlled loop**
while loop	**Condition-controlled loop**
do-while loop	**Post-condition loop in C**
Comment	**Text in code ignored by the translator**
Debugging	**Finding and fixing errors**
Breakpoint	**A point where execution pauses for inspection**
Format specifier	**Placeholder in C printf/scanf (e.g., %d, %f, %s)**

13.1 Introduction to C and Python

Python

Python is a high-level, interpreted, general-purpose language known for simple, readable syntax.

Key characteristics:

- Interpreted (executed line by line).
- Dynamically typed (no need to declare variable types).
- Uses indentation to define blocks.
- Simple, English-like syntax.
- Large standard library.
- Cross-platform.

Uses:

- Beginners learning programming.
- Data science and AI.
- Scripting and automation.
- Web development (Django, Flask).
- Scientific computing.

Hello, World! in Python:

```
print("Hello, World!")
```

That is the entire program. No headers, no main function, no semicolons.

C

C is a high-level but relatively low-level-oriented, compiled, general-purpose language, giving the programmer close control over memory and hardware.

Key characteristics:

- Compiled (translated entirely to machine code before running).

- Statically typed (must declare variable types).
- Uses curly braces { } to define blocks.
- Semicolons required at the end of statements.
- Requires a main() function.
- Fast and efficient.
- Close to hardware.

Uses:

- Operating systems (Linux, Windows).
- Embedded systems.
- Performance-critical software.
- System programming.
- Compilers and interpreters.

Hello, World! in C:

```
#include <stdio.h>

int main() {
   printf("Hello, World!\n");
   return 0;
}
```

Key Structural Differences

Feature	**Python**	**C**
Translation	Interpreted (or compiled to bytecode)	Compiled to machine code
Typing	Dynamically typed	Statically typed
Blocks	Defined by indentation	Defined by curly braces { }
Statement end	No semicolon needed	Semicolon required
Entry point	Top of file	main() function
Headers	Not needed	#include needed
Variable declaration	Not needed	Required
Speed	Slower	Faster
Ease of learning	Easier	More difficult
Memory management	Automatic (garbage collected)	Manual (malloc/free)

Variables and Basic Data Types

Concept	Python	C
Integer	x = 5	int x = 5;
Real/float	y = 3.14	float y = 3.14;
Character	Not a separate type	char c = 'h';
String	s = "hi"	char s[] = "hi";
Boolean	b = True	int b = 1; (0 = false, non-zero = true)
Input	name = input("Name: ")	scanf("%d", &x);
Output	print(x)	printf("%d", x);

Python input:

```
name = input("Enter your name: ")

age = int(input("Enter your age: "))  # convert to integer
```

Note: input() always returns a string. You must convert it with int() or float() if you need a number.

C input:

```
int age;

printf("Enter your age: ");

scanf("%d", &age);
```

Note: The & before age is the address-of operator. It tells scanf where to store the value.

Python output:

```
print("Hello", name)

print(f"Age: {age}")  # f-string
```

C output:

```
printf("Hello %s\n", name);

printf("Age: %d\n", age);
```

Common C format specifiers:

Specifier	**Type**
%d	Integer

Specifier	Type
%f	Float
%c	Character
%s	String
%lf	Double

13.2 Operators and Expressions

Arithmetic Operators

Operator	Meaning	Python	C
+	Addition	Yes	Yes
-	Subtraction	Yes	Yes
*	Multiplication	Yes	Yes
/	Division	Yes	Yes

Operator	Meaning	Python	C
%	Modulus (remainder)	Yes	Yes
**	Exponentiation	Yes	No (use pow())
//	Integer/floor division	Yes	No (use integer division)

Relational (Comparison) Operators

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to

Operator	Meaning
<=	Less than or equal to

Logical Operators

Python	C	Meaning
and	&&	AND
or	||	OR
not	!	NOT

Assignment Operators

Operator	Meaning	Example
=	Assign	x = 5
+=	Add and assign	x += 1 (same as x = x + 1)

Operator	Meaning	Example
-=	Subtract and assign	x -= 1
*=	Multiply and assign	x *= 2
/=	Divide and assign	x /= 2

Operator Precedence

Determines the order operations are evaluated in an expression.

Typical order (highest to lowest):

1. Brackets ( )
2. Exponentiation ** (Python)
3. Multiplication, division, modulus * / %
4. Addition, subtraction + -
5. Relational operators == != > < >= <=
6. Logical NOT
7. Logical AND
8. Logical OR
9. Assignment =

Tip: Use brackets to make your intention clear and avoid mistakes.

Example (Python):

result = a + b * 2 - c / 2

This is evaluated as: a + (b * 2) - (c / 2).

Example (C):

int result = a + b * 2 - c / 2;

Same precedence rules apply.

13.3 Selection Structures (Python/C)

Python: if, elif, else

```
age = int(input("Enter age: "))


if age >= 18:
    print("Adult")
elif age >= 13:
    print("Teenager")
else:
    print("Child")
```

Key points:

Colon : after each condition.

Indentation defines the block.

elif means "else if".

else is optional.

C: if, else if, else

```
int age;

printf("Enter age: ");

scanf("%d", &age);


if (age >= 18) {

   printf("Adult\n");

} else if (age >= 13) {

   printf("Teenager\n");

} else {

   printf("Child\n");

}
```

Key points:

- Condition in parentheses.
- Curly braces { } define the block.
- Semicolons at the end of statements.
- else if means "else if".

Multi-Way Selection

Python (3.10+) match/case:

```
grade = input("Enter grade: ")


match grade:
    case "A":
        print("Excellent")
    case "B":
        print("Good")
    case _:
        print("Try again")
```

Python (more commonly) chained elif:

```
if grade == "A":
    print("Excellent")
elif grade == "B":
    print("Good")
else:
    print("Try again")
```

C: switch/case/break/default:

```
char grade;
```

```
printf("Enter grade: ");
scanf(" %c", &grade);

switch (grade) {
    case 'A':
        printf("Excellent\n");
        break;
    case 'B':
        printf("Good\n");
        break;
    default:
        printf("Try again\n");
}
```

Key points:

- break prevents fall-through to the next case.
- default handles all other cases.
- switch works with integers and characters, not strings or floats.

13.4 Iterative Structures (Python/C)

13.5

Count-Controlled Loops (FOR)

Python: for i in range():

```
for i in range(1, 11):
    print(i)
```

Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Note: range(1, 11) includes 1 but excludes 11.

Other range forms:

```
range(5)        # 0, 1, 2, 3, 4
range(2, 8)     # 2, 3, 4, 5, 6, 7
range(1, 10, 2) # 1, 3, 5, 7, 9 (step 2)
range(10, 0, -1) # 10, 9, 8, ..., 1 (countdown)
```

C: for loop:

```
for (int i = 1; i <= 10; i++) {
    printf("%d\n", i);
}
```

Structure:

- Initialisation: int i = 1
- Condition: i <= 10
- Update: i++

Condition-Controlled Loops (WHILE)

Python: while:

```
count = 0
while count < 5:
    count += 1
    print(count)
```

Output: 1, 2, 3, 4, 5

C: while:

```
int count = 0;
while (count < 5) {
   count++;
   printf("%d\n", count);
}
```

Key point: The condition is checked before each iteration. The loop may run zero times.

Post-Test Loops (REPEAT-UNTIL / DO-WHILE)

Python: no built-in do-while. The same effect is achieved with a while True: loop containing a conditional break.

```
while True:
   n = int(input("Enter a positive number: "))
```

```
    if n > 0:
        break
```

C: do-while:

```
int n;
do {
    printf("Enter a positive number: ");
    scanf("%d", &n);
} while (n <= 0);
```

Key point: The body runs at least once, then the condition is checked.

Comparison Table

Loop Type	Python	C	Minimum Runs
Count-controlled	for i in range()	for (init; cond; update)	0
Pre-condition	while cond:	while (cond)	0
Post-condition	while True: + break	do { } while (cond);	1

13.5 Writing and Debugging C and Python Programs

Good Programming Practice

- Use meaningful variable names (e.g., totalScore not ts).

- Add comments to explain non-obvious logic.
- Keep functions/subroutines short and focused.
- Indent consistently for readability.
- Use blank lines to separate logical sections.
- Test with different types of data.

Comments:

Python	**C**
# This is a comment	**// This is a comment**
	/* Multi-line comment */

Sample Complete Program: Sum of Numbers 1 to n

```
# Program to calculate the sum of numbers from 1 to n

n = int(input("Enter n: "))
total = 0

for i in range(1, n + 1):
    total += i

print("Sum =", total)
```

C:

```
#include <stdio.h>


int main() {
   int n, total = 0;


   printf("Enter n: ");
   scanf("%d", &n);


   for (int i = 1; i <= n; i++) {
      total += i;
   }


   printf("Sum = %d\n", total);
   return 0;
}
```

Common Beginner Errors and Fixes

Python errors:

Error	Fix
Forgetting the colon : after if/for/while/def	Add the colon
Mixing tabs and spaces for indentation	Use spaces consistently (4 spaces)
Forgetting to convert input() to int()/float()	Use int(input())
Using = instead of == in conditions	Use == for comparison
Off-by-one errors in range()	Remember range excludes the end value

C errors:

Error	Fix
Forgetting a semicolon ;	Add the semicolon
Using = instead of == in an if	Use == for comparison
Forgetting & before a variable in scanf	Add the &
Mismatched format specifiers (e.g., %d for a float)	Use %f for floats
Not including a header file	Add #include <stdio.h>

Error	Fix
Forgetting return 0; in main	**Add it**

Debugging Workflow

1. Run the program.
2. Read any error message and go to the indicated line.
3. Use print/printf statements to check variable values at key points.
4. Test with several different sets of input data (normal, boundary, invalid).
5. Use a debugger to step through the code and inspect variables.

Test Data Types

Type	Example	Purpose
Normal	**n = 10**	**Check typical behaviour**
Boundary	**n = 0, n = 1**	**Check edge cases**
Erroneous	**n = -5**	**Check error handling**

13.6 Python/C Program Development and Debugging Project

Mini-Project: Grade Calculator

Requirements:

- Accept marks for several subjects.
- Calculate the average.
- Output a grade (A, B, C, D, F) based on the average.

Grade boundaries (example):

- Average >= 80: A
- Average >= 70: B
- Average >= 60: C
- Average >= 50: D
- Below 50: F

Design (pseudocode):

```
FUNCTION CalculateAverage(marks)
    total = 0
    FOR each mark IN marks
        total = total + mark
    ENDFOR
    RETURN total / number of marks
ENDFUNCTION


FUNCTION DetermineGrade(average)
    IF average >= 80 THEN RETURN "A"
    ELSE IF average >= 70 THEN RETURN "B"
```

```
    ELSE IF average >= 60 THEN RETURN "C"
    ELSE IF average >= 50 THEN RETURN "D"
    ELSE RETURN "F"
    ENDIF
ENDFUNCTION

MAIN
    INPUT number of subjects
    FOR each subject
        INPUT mark
    ENDFOR
    average = CalculateAverage(marks)
    grade = DetermineGrade(average)
    OUTPUT average, grade
ENDMAIN
```

Implementation in Python:

```
# Grade Calculator

def calculate_average(marks):
    total = 0
```

```
    for mark in marks:
        total += mark
    return total / len(marks)
def determine_grade(average):
    if average >= 80:
        return "A"
    elif average >= 70:
        return "B"
    elif average >= 60:
        return "C"
    elif average >= 50:
        return "D"
    else:
        return "F"

# Main program
num_subjects = int(input("How many subjects? "))
marks = []

for i in range(num_subjects):
    mark = float(input(f"Enter mark for subject {i + 1}: "))
```

```
    marks.append(mark)
average = calculate_average(marks)
grade = determine_grade(average)
print(f"Average: {average:.2f}")
print(f"Grade: {grade}")
```

Implementation in C:

```
#include <stdio.h>

float calculate_average(int marks[], int count) {
    int total = 0;
    for (int i = 0; i < count; i++) {
        total += marks[i];
    }
    return (float)total / count;
}

char determine_grade(float average) {
    if (average >= 80) return 'A';
    else if (average >= 70) return 'B';
    else if (average >= 60) return 'C';
    else if (average >= 50) return 'D';
```

```
    else return 'F';
}

int main() {
    int num_subjects;
    int marks[100];
    float average;
    char grade;

    printf("How many subjects? ");
    scanf("%d", &num_subjects);

    for (int i = 0; i < num_subjects; i++) {
        printf("Enter mark for subject %d: ", i + 1);
        scanf("%d", &marks[i]);
    }

    average = calculate_average(marks, num_subjects);
    grade = determine_grade(average);

    printf("Average: %.2f\n", average);
    printf("Grade: %c\n", grade);
```

```
    return 0;
}
```

Project Development Steps

1. **Requirements:** Understand what the program should do.
2. **Design:** Write pseudocode or a flowchart.
3. **Implement:** Write the program in Python or C, breaking it into functions.
4. **Test:** Test with normal data, boundary data, and invalid data.
5. **Debug and refine:** Fix any errors and improve structure.
6. **Document:** Add comments explaining what each part does.

Example Test Data

Test Case	Input	Expected Output
Normal	3 subjects: 75, 82, 68	Average: 75.00, Grade: B
Boundary	1 subject: 80	Average: 80.00, Grade: A

Test Case	Input	Expected Output
Boundary	1 subject: 50	Average: 50.00, Grade: D
Invalid	Negative mark	Handle gracefully (or reject)

Review Questions

1. Describe three key differences between Python and C.
2. Write a "Hello, World!" program in both Python and C.
3. Explain the difference between dynamically typed and statically typed languages.
4. How do you declare an integer variable in Python and in C?
5. How do you take user input in Python and in C?
6. What is a format specifier? Give two examples used in C.
7. List the arithmetic operators in Python and C. What is the difference for exponentiation?
8. Explain operator precedence with an example.
9. Write a Python program using if/elif/else to check if a number is positive, negative, or zero.
10. Write the same program in C.
11. Write a Python for loop to print even numbers from 2 to 20.
12. Write the same loop in C.
13. Write a Python while loop that asks for a password until the correct one is entered.
14. Write a C do-while loop that asks for a positive number.
15. Why does Python not have a do-while loop? How can you achieve the same effect?
16. List four good programming practices.

17. Identify the error in each of the following:
18. Python: if x > 5 (missing colon)
19. C: if (x = 5) (should be ==)
20. C: scanf("%d", x); (missing &)
21. Python: age = input("Age: ") then if age > 18: (needs int conversion)
22. Describe a debugging workflow for a program that produces incorrect output.
23. What types of test data should you use? Give an example of each.
24. Write a complete program in Python or C that calculates the average of 5 numbers and outputs the result.

Chapter 14: Selecting and Manipulating Data Structures

Chapter Overview

This chapter introduces students to data structures, the ways computers organise and store data so it can be used efficiently. It begins with primitive data types (integers, reals, characters, booleans, strings), then moves to arrays (one-dimensional and multi-dimensional) and records (grouping different data types together). The chapter explains data types, memory allocation (static and dynamic), and abstract data types (stacks and queues). It then covers linked lists (singly, doubly, and circular), hash tables (including collision resolution), and finally shows how repetition structures are used to process arrays and lists using common patterns (traversal, searching, summing, finding maximum, counting). This chapter is essential for understanding how programs store and manipulate collections of data.

Learning Objectives

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

1. Define primitive data types and give examples of each.
2. Distinguish between statically typed and dynamically typed languages.
3. Define an array and explain its characteristics.
4. Create and use one-dimensional and multi-dimensional arrays.
5. Define a record and explain how it differs from an array.
6. Explain static and dynamic memory allocation.
7. Define Abstract Data Types and describe stacks and queues.
8. Describe linked lists and their advantages and disadvantages over arrays.
9. Distinguish between singly, doubly, and circular linked lists.
10. Explain how hash tables work, including hash functions and collisions.
11. Describe collision resolution techniques (chaining and open addressing).
12. Use repetition structures to process arrays and lists (traversal, search, sum, maximum, count).

Key Terms

Term	**Definition**
Data structure	A way of organising and storing data so it can be used efficiently
Primitive data type	A fundamental, built-in data type provided by a language

Term	Definition
Integer	Whole number data type
Real / Float	Decimal number data type
Character	A single symbol or letter
Boolean	Logical true/false value
String	A sequence of characters
Static typing	Types are checked before the program runs (compile time)
Dynamic typing	Types are checked while the program runs (run time)
Array	A fixed-size, ordered collection of elements of the same type, accessed by index
Index	The position number of an element in an array (usually starts at 0)
One-dimensional array	A simple list of elements
Multi-dimensional array	An array of arrays (e.g., a 2D grid or matrix)
Record	A data structure grouping related fields of possibly different types

Term	Definition
Field	A single data item within a record
Struct	C keyword for defining a record
Dictionary	Python data structure for key-value pairs
Static allocation	Memory reserved at compile time with a fixed size
Dynamic allocation	Memory reserved and released while the program runs
Abstract Data Type (ADT)	A logical description of a data structure and its operations
Stack	Last-In-First-Out (LIFO) data structure
Queue	First-In-First-Out (FIFO) data structure
Push	Add an item to the top of a stack
Pop	Remove an item from the top of a stack
Enqueue	Add an item to the rear of a queue
Dequeue	Remove an item from the front of a queue

Term	Definition
Linked list	A linear data structure of nodes, each pointing to the next
Node	An element of a linked list containing data and a pointer
Pointer / Reference	A link to another node or memory location
Singly linked list	Each node points only to the next node
Doubly linked list	Each node points to both the next and previous nodes
Circular linked list	The last node points back to the first
Hash table	A data structure storing key-value pairs using a hash function
Hash function	Converts a key into an index in an array
Collision	When two different keys hash to the same index
Chaining	Collision resolution using a linked list at each index
Open addressing	Collision resolution by searching for the next available slot
Linear search	Searching an array element by element
Traversal	Visiting every element in a data structure

14.1 Primitive Data Types

Primitive (basic) data types are the fundamental, built-in data types provided directly by a programming language, from which more complex data structures can be built.

Type	Description	Example
Integer	Whole numbers, positive or negative	5, -12, 1000
Real / Float	Numbers with a decimal point	3.14, -0.5
Character	A single symbol or letter	'A', '7', '$'
Boolean	Logical true/false value	True, False
String	A sequence of characters (in many languages, considered a composite/derived type rather than primitive)	"Hello"

Type Checking

Statically Typed	Dynamically Typed
Types checked before the program runs (at compile time)	Types checked while the program runs

Statically Typed	Dynamically Typed
Variables must be declared with a type	Variables do not need type declarations
Errors caught early	Errors caught at run time
Example: C, Java	Example: Python, JavaScript

Example (C, statically typed):

int age = 25;

float price = 9.99;

char grade = 'A';

Example (Python, dynamically typed):

age = 25

price = 9.99

grade = 'A'

In Python, the type is inferred from the value assigned.

14.2 Arrays

An **array** is a fixed-size, ordered collection of elements of the same data type, stored in contiguous memory locations, and accessed using an index (position number), usually starting at 0.

Characteristics of Arrays

Characteristic	Description
Fixed size	Must be declared in advance in many languages
Same type	All elements must be of the same data type
Ordered	Elements have a specific position (index)
Contiguous memory	Elements are stored next to each other in memory
Fast random access	Any element can be accessed directly using its index
Slow insertion/deletion	Inserting or deleting in the middle requires shifting other elements

One-Dimensional Arrays

A 1D array is a simple list.

Example (Python):

```
scores = [67, 82, 45, 90, 78]

print(scores[0])   # 67 (first element)

print(scores[2])   # 45 (third element)

scores[2] = 50     # change third element
```

Example (C):

```
int scores[5] = {67, 82, 45, 90, 78};

printf("%d\n", scores[0]);   // 67
```

```
scores[2] = 50;              // change third element
```

Key points:

- Python lists are dynamic (can grow and shrink).
- C arrays are fixed size (set at declaration).
- Indexing starts at 0 in both languages.

Multi-Dimensional Arrays

A 2D array ("array of arrays") represents data in rows and columns, e.g., a grid, table, or matrix.

Example (Python):

```
grid = [
    [1, 2, 3],
    [4, 5, 6]
]
print(grid[0][0])   # 1
print(grid[1][2])   # 6 (row 1, column 2)
```

Example (C):

```
int grid[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
printf("%d\n", grid[0][0]);   // 1
printf("%d\n", grid[1][2]);   // 6
```

Uses of multi-dimensional arrays:

- Storing tables (e.g., student marks per subject).
- Representing images (pixels in rows and columns).
- Mathematical matrices.
- Game boards (chess, tic-tac-toe).

Array Processing Patterns (Preview)

Common operations on arrays (covered in detail in section 14.7):

- Traversal (visiting every element).
- Searching (linear search).
- Calculating a total/sum.
- Finding the maximum or minimum.
- Counting elements meeting a condition.

14.3 Records

A **record** (in Python often implemented as a dictionary, class, or named tuple; in C, a struct) is a data structure that groups together related fields of possibly different data types under one name.

Example (C struct)

```
struct Student {
    char name[20];
    int age;
    float average;
```

```
};

struct Student s1 = {"Ama", 17, 84.5};

printf("%s is %d years old\n", s1.name, s1.age);
```

Example (Python: using a dictionary)

```
student = {
    "name": "Ama",
    "age": 17,
    "average": 84.5
}

print(student["name"])   # Ama

print(student["age"])    # 17
```

Example (Python: using a class)

```
class Student:
    def __init__(self, name, age, average):
        self.name = name
        self.age = age
        self.average = average


s1 = Student("Ama", 17, 84.5)
```

```
print(s1.name)
```

Arrays vs Records

Array	**Record**
Groups items of the same type	Groups items of different types
Accessed by position (index)	Accessed by field name
All elements are the same type	Fields can be different types
Example: a list of scores	Example: a student's name, age, average

Arrays of records are common: e.g., a whole class list, where each element is one student's record.

Example (C):

```
struct Student classList[30];

classList[0] = s1;
```

Example (Python):

python

```
classList = [
    {"name": "Ama", "age": 17, "average": 84.5},
    {"name": "Kofi", "age": 16, "average": 78.2}
```

]

14.4 Data Types, Memory Allocation and Data Structures

Static vs Dynamic Memory Allocation

Static Allocation	**Dynamic Allocation**
Memory reserved at compile time	Memory reserved and released while the program runs
Fixed, unchanging size	Can grow or shrink as needed
Fast and simple	More flexible but more complex
Wastes space if data is smaller	Uses exactly the space needed
Cannot grow if more space is needed	Can grow on demand
Example: a C array with fixed size	Example: Python lists; C malloc()/free()

Example (C static):

```
int scores[100];   // reserves space for 100 integers
```

Example (C dynamic):

```
int *scores = malloc(100 * sizeof(int));   // allocate at run time

free(scores);                              // release when done
```

Example (Python):

```
scores = []         # empty list (dynamic)

scores.append(67)   # grows as needed
```

Abstract Data Types (ADTs)

An **Abstract Data Type (ADT)** is a logical/conceptual description of a data structure and the operations that can be performed on it, without specifying exactly how it is implemented internally.

Examples:

ADT	Operations	Behaviour
Stack	push, pop, peek, isEmpty	LIFO (Last-In-First-Out)
Queue	enqueue, dequeue, peek, isEmpty	FIFO (First-In-First-Out)
List	insert, remove, get, size	Ordered collection
Map/Dictionary	put, get, remove	Key-value pairs

Stacks

A **stack** is a Last-In-First-Out (LIFO) data structure. Items are added and removed only from the "top".

Operations:

- **push:** Add an item to the top.
- **pop:** Remove the item from the top.
- **peek:** View the top item without removing it.
- **isEmpty:** Check if the stack is empty.

Example (Python):

```
stack = []

stack.append(1)   # push

stack.append(2)   # push

stack.append(3)   # push

print(stack.pop())  # 3 (pop)

print(stack.pop())  # 2 (pop)
```

Uses of stacks:

- The call stack (Chapter 11).
- Undo functions in editors.
- Browser back button.
- Expression evaluation.
- Depth-first search.

Queues

A **queue** is a First-In-First-Out (FIFO) data structure. Items are added at the "rear" and removed from the "front".

Operations:

- **enqueue:** Add an item to the rear.
- **dequeue:** Remove the item from the front.
- **peek:** View the front item without removing it.
- **isEmpty:** Check if the queue is empty.

Example (Python using collections.deque):

```
from collections import deque

queue = deque()

queue.append(1)    # enqueue

queue.append(2)    # enqueue

queue.append(3)    # enqueue

print(queue.popleft())  # 1 (dequeue)

print(queue.popleft())  # 2 (dequeue)
```

Uses of queues:

- Print queues.
- Task scheduling.
- Breadth-first search.
- Buffer management.
- Handling requests in web servers.

Stack vs Queue

Feature	Stack	Queue

Feature	Stack	Queue
Order	LIFO	FIFO
Add	Push (top)	Enqueue (rear)
Remove	Pop (top)	Dequeue (front)
Example	Undo, call stack	Print queue, scheduling

14.5 Linked Lists

A **linked list** is a linear data structure made of individual nodes, where each node contains:

1. Data
2. A pointer/reference to the next node in the sequence

Unlike an array, the nodes are not necessarily stored in contiguous memory.

Diagram concept:

[**Data**| Next] -> [**Data**| Next] -> [**Data**| Next] -> NULL

Advantages Over Arrays

- **Dynamic size:** Can grow/shrink easily at run time by adding/removing nodes.
- **Efficient insertion/deletion:** Just update pointers. No shifting of other elements needed.

Disadvantages Compared to Arrays

- **No direct/random access:** To reach the nth element, you must traverse from the start, following each next pointer in turn (slower search).
- **Extra memory:** Requires extra memory to store the pointer in each node.
- **Less cache-friendly:** Nodes are scattered in memory.

Types of Linked Lists

Type	**Description**
Singly linked	Each node points only to the next node
Doubly linked	Each node points to both the next and previous node, allowing traversal in both directions
Circular linked	The last node points back to the first, forming a loop

Example (Python: simple singly linked list node)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
# Create nodes
n1 = Node(10)
n2 = Node(20)
```

```
n3 = Node(30)

# Link them

n1.next = n2

n2.next = n3
```

Comparison: Array vs Linked List

Feature	Array	Linked List
Size	Fixed (in C)	Dynamic
Access	Random (by index)	Sequential (traverse from start)
Insertion/deletion	Slow (shifting)	Fast (pointer updates)
Memory	Contiguous	Scattered
Extra memory	None	Pointer per node
Cache efficiency	High	Low

14.6 Hash Tables

A **hash table** (or hash map) is a data structure that stores data as key-value pairs, using a hash function to convert each key into an index (a number) in an underlying array, allowing very fast lookup, insertion, and deletion.

How It Works

1. A hash function takes a key (e.g., a name, an ID number) and computes a numeric index (typically hash(key) % array_size).
2. The value is stored at that computed index in the array.
3. To retrieve the value, the same hash function is applied to the key to find the index directly. No searching required.

Example (Conceptual)

Keys: "Ama", "Kofi", "Yaa"

Hash function: sum of ASCII values % 10

"Ama" -> 65+109+97 = 271 -> 271 % 10 = 1

"Kofi" -> 75+111+102+105 = 393 -> 393 % 10 = 3

"Yaa" -> 89+97+97 = 283 -> 283 % 10 = 3 (collision!)

Collisions

A **collision** occurs when two different keys hash to the same index.

Common resolution techniques:

Technique	Description
Chaining	Each array slot holds a small linked list of all items that hashed to that index
Open addressing	If a slot is occupied, the algorithm searches for the next available slot according to a fixed rule (e.g., linear probing checks the next slot in sequence)

Chaining example:

Index 3: ["Kofi" -> value1] -> ["Yaa" -> value2]

Open addressing (linear probing) example:

"Kofi" hashes to 3 -> stored at 3

"Yaa" hashes to 3 -> slot taken -> store at 4

Advantages and Disadvantages

Advantages	Disadvantages
Average-case very fast O(1) lookup/insertion/deletion	Performance can degrade if there are many collisions
Works well for large datasets	Does not maintain any particular order of elements
Simple to implement	Requires a well-designed hash function to spread keys evenly

Real-World Uses

- Implementing dictionaries/associative arrays in programming languages (e.g., Python's dict is a hash table).
- Database indexing.
- Caching.

- Password storage (via hashing).

Example (Python dict, which is a hash table)

python

```
student = {
   "name": "Ama",
   "age": 17,
   "average": 84.5
}
print(student["name"])   # O(1) average lookup
```

14.7 Using Repetition Structures to Process Arrays and Lists

Common patterns for processing arrays/lists using loops.

1. Traversal (Visiting Every Element)

Python:

```
scores = [67, 82, 45, 90, 78]


for score in scores:
   print(score)
```

C:

```
int scores[5] = {67, 82, 45, 90, 78};
for (int i = 0; i < 5; i++) {
```

```
    printf("%d\n", scores[i]);
}
```

2. Searching (Linear Search)

Python:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i   # found - return index
    return -1          # not found
scores = [67, 82, 45, 90, 78]
print(linear_search(scores, 90))   # 3
print(linear_search(scores, 100))  # -1
```

C:

```
int linear_search(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

3. Calculating a Total/Sum

Python:

```
scores = [67, 82, 45, 90, 78]

total = 0

for score in scores:

    total += score

print("Total:", total)
```

C:

```
int scores[5] = {67, 82, 45, 90, 78};

int total = 0;

for (int i = 0; i < 5; i++) {

    total += scores[i];

}

printf("Total: %d\n", total);
```

4. Finding the Maximum Value

Python:

```
scores = [67, 82, 45, 90, 78]

maximum = scores[0]

for score in scores:

    if score > maximum:

        maximum = score
```

```
print("Maximum:", maximum)
```

C:

```
int scores[5] = {67, 82, 45, 90, 78};

int maximum = scores[0];

for (int i = 1; i < 5; i++) {

    if (scores[i] > maximum) {

        maximum = scores[i];

    }

}

printf("Maximum: %d\n", maximum);
```

5. Counting Elements Meeting a Condition

Python:

```
scores = [67, 82, 45, 90, 78]

count = 0

for score in scores:

    if score >= 50:

        count += 1

print("Passes:", count)
```

C:

```
int scores[5] = {67, 82, 45, 90, 78};

int count = 0;
```

```
for (int i = 0; i < 5; i++) {
    if (scores[i] >= 50) {
        count++;
    }
}
printf("Passes: %d\n", count);
```

Summary of Common Patterns

Pattern	Description	Example
Traversal	Visit every element	Print all scores
Search	Find a specific element	Find index of 90
Sum	Add all elements	Total of scores
Maximum/Minimum	Find the largest/smallest	Highest score
Count	Count elements meeting a condition	Number of passes

These patterns are the foundation for almost all algorithms that process collections of data, and lead directly into the efficiency analysis covered in Chapter 15.

Review Questions

1. Define primitive data type and give five examples.
2. Distinguish between statically typed and dynamically typed languages.
3. Define an array and list four characteristics.
4. Write a Python list of five numbers and print the third element.
5. Write a C array of five integers and print the third element.
6. What is a multi-dimensional array? Give one example of its use.
7. Define a record and explain how it differs from an array.
8. Write a C struct for a Book with title, author, and year.
9. Write a Python dictionary for the same Book.
10. Distinguish between static and dynamic memory allocation.
11. Define an Abstract Data Type.
12. Describe a stack. List its operations and give two uses.
13. Describe a queue. List its operations and give two uses.
14. Compare stacks and queues in a table.
15. Define a linked list. State two advantages and two disadvantages over arrays.
16. Distinguish between singly, doubly, and circular linked lists.
17. Explain how a hash table works.

18. What is a collision? Describe two collision resolution techniques.

19. Write a Python function that finds the maximum value in a list.

20. Write a C function that counts how many elements in an array are greater than 50.

21. Write a Python program that traverses a list and prints each element.

22. Write a C program that calculates the sum of an array of 10 integers.

Chapter 15: Evaluating Algorithm Efficiency

Chapter Overview

This chapter introduces students to the concept of algorithmic efficiency, how to measure how well an algorithm performs as the size of its input grows. It begins with the complexity of algorithms, explaining time complexity and space complexity, and the difference between best case, worst case, and average case. The chapter then covers Big-O notation, the standard mathematical way of describing an algorithm's growth rate, and the common complexity classes from best to worst. It also explains computational problem types, including tractable, intractable, decidable, undecidable, and NP-complete problems. Finally, it covers dry runs (tracing) and optimisation techniques, showing how to improve algorithm performance by choosing better algorithms, better data structures, and reducing redundant work. This chapter equips students with the tools to compare and choose algorithms wisely.

Learning Objectives

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

1. Define algorithmic complexity and explain time and space complexity.
2. Distinguish between best case, worst case, and average case performance.
3. Define Big-O notation and explain its purpose.
4. Identify and compare common complexity classes.
5. Estimate the Big-O complexity of simple algorithms from their code.
6. Explain why Big-O matters in practice.
7. Distinguish between tractable and intractable problems.
8. Distinguish between decidable and undecidable problems.
9. Explain what NP-complete problems are.

10. Perform a dry run (trace) of an algorithm using a trace table.

11. Describe optimisation techniques.

12. Explain the trade-off between time and space.

Key Terms

Term	**Definition**
Algorithmic complexity	How the resources an algorithm needs grow as input size grows
Time complexity	How the number of operations grows with input size
Space complexity	How the memory required grows with input size
Best case	The fastest an algorithm can run for a given input size
Worst case	The slowest an algorithm can run for a given input size
Average case	The typical performance across many inputs
Big-O notation	A mathematical way of describing the upper bound (worst-case growth rate) of an algorithm

Term	Definition
O(1)	Constant time
O(log n)	Logarithmic time
O(n)	Linear time
O(n log n)	Linearithmic time
$O(n^2)$	Quadratic time
$O(2^n)$	Exponential time
Tractable problem	A problem solvable in reasonable (polynomial) time
Intractable problem	A problem with no known efficient algorithm
Decidable problem	A problem for which an algorithm exists that always gives a correct yes/no answer
Undecidable problem	A problem for which no algorithm can ever exist for all inputs

Term	Definition
Halting Problem	The problem of determining whether an arbitrary program will eventually stop or run forever
NP-complete	A class of difficult problems with no known polynomial-time solution, but solutions can be quickly verified
Dry run / trace	Manually stepping through an algorithm with sample data, recording variable values
Trace table	A table recording variable values at each step of a dry run
Optimisation	Improving an algorithm's performance
Memoisation	Storing already-computed results to avoid recalculating them
Trade-off	Balancing time and space (improving one may worsen the other)

15.1 Complexity of Algorithms

What is Algorithmic Complexity?

Algorithmic complexity measures how the resources an algorithm needs (mainly time and memory/space) grow as the size of its input (usually called n) grows. This lets us compare algorithms independently of any specific computer's speed.

Time Complexity

Time complexity is how the number of basic operations (comparisons, additions, etc.) grows as n increases.

Example: A linear search through a list of n items may perform up to n comparisons. Doubling the list size doubles the worst-case time.

Space Complexity

Space complexity is how the amount of memory required grows as n increases.

Example: Creating an array of n elements requires O(n) space.

Best Case, Worst Case, Average Case

For a given input size, an algorithm's performance can vary depending on the arrangement of the data.

Example: Linear search for a target in a list of n items.

Case	Description	Comparisons
Best case	Target is the first element	1
Worst case	Target is the last element or not present	n

Case	Description	Comparisons
Average case	On average, target is somewhere in the middle	n/2

Why we focus on the worst case:

- It guarantees an upper bound on how slow the algorithm could be.
- It is often easier to calculate than the average case.
- It helps in safety-critical or time-critical systems.

Example: Sorting algorithms.

- Best case: the data is already sorted (some algorithms run faster).
- Worst case: the data is in reverse order.
- Average case: random order.

15.2 Big-O Notation

What is Big-O Notation?

Big-O notation is a standard mathematical way of describing the upper bound (worst-case growth rate) of an algorithm's time or space complexity, ignoring constant factors and lower-order terms, so we can focus on how it scales for large inputs.

Key idea: Big-O describes the shape of the growth curve, not the exact number of operations.

Example: An algorithm that performs $3n^2 + 5n + 100$ operations is $O(n^2)$, because for large n, the n^2 term dominates.

Common Complexity Classes (from best to worst)

Notation	Name	Example Algorithm
O(1)	Constant	Accessing an array element by index; hash table lookup (average case)
O(log n)	Logarithmic	Binary search
O(n)	Linear	Linear search; traversing a list once
O(n log n)	Linearithmic	Efficient sorting algorithms (merge sort, quicksort, average case)
$O(n^2)$	Quadratic	Simple sorting algorithms (bubble sort, selection sort, insertion sort)
$O(2^n)$	Exponential	Naive recursive Fibonacci; some brute-force algorithms

Visualising the Growth Rates

For n = 10:

Complexity	Approximate Operations

Complexity	Approximate Operations
O(1)	1
O(log n)	3
O(n)	10
O(n log n)	33
$O(n^2)$	100
$O(2^n)$	1,024

For n = 100:

Complexity	**Approximate Operations**
O(1)	1
O(log n)	7

Complexity	Approximate Operations
O(n)	100
O(n log n)	664
$O(n^2)$	10,000
$O(2^n)$	1.27 x 10^30 (impractical)

The difference becomes enormous as n grows. This is why choosing the right algorithm matters.

How to Estimate Big-O from Code

Code Pattern	Complexity	Reason
A fixed number of operations regardless of n (e.g., accessing array[0])	O(1)	No dependence on n
A single loop through n items	O(n)	One pass through the data
A loop inside another loop, both running roughly	$O(n^2)$	n iterations of n

Code Pattern	Complexity	Reason
n times		iterations
Repeatedly halving the search space each step (like binary search)	O(log n)	Each step halves the problem
A loop through n items, and inside each iteration a binary search	O(n log n)	n iterations of a log n operation

Examples

Example 1: Constant time O(1)

python

```
def get_first(arr):
    return arr[0]
```

Regardless of how large the array is, this takes the same time.

Example 2: Linear time O(n)

python

```
def find_max(arr):
    maximum = arr[0]
    for x in arr:
        if x > maximum:
```

```
        maximum = x
    return maximum
```

The loop runs n times. Doubling the array size doubles the time.

Example 3: Quadratic time O(n²)

python

```
def find_duplicates(arr):
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j]:
                return True
    return False
```

Two nested loops, each roughly n times. Doubling the array size quadruples the time.

Example 4: Logarithmic time O(log n)

python

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
```

```
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Each iteration halves the search space. For an array of 1,000,000 items, binary search takes about 20 steps.

Why Big-O Matters

- It allows programmers to predict how an algorithm's performance will scale before running it on very large datasets.
- It helps choose the most appropriate algorithm/data structure for a given task.
- Example: preferring O(log n) binary search over O(n) linear search on a large sorted list.

Practical example: Searching a sorted list of 1,000,000 items.

- Linear search: up to 1,000,000 comparisons.
- Binary search: about 20 comparisons.

The difference is massive.

15.3 Computational Problem Types

Tractable Problems

Tractable problems are problems that can be solved by an algorithm in a "reasonable" amount of time as input size grows (typically polynomial time, e.g., O(n), $O(n^2)$, O(n log n)).

Examples:

- Sorting a list.
- Searching a list.

- Finding the shortest path in a graph (Dijkstra's algorithm).
- Matrix multiplication.

Intractable Problems

Intractable problems are problems for which no known algorithm can solve them efficiently as input size grows large (e.g., requiring exponential time $O(2^n)$). Solutions become practically impossible to compute for even moderately large inputs.

Examples:

- Travelling Salesman Problem (finding the shortest route visiting all cities).
- Boolean satisfiability (SAT).
- Some scheduling and optimisation problems.

Decidable Problems

Decidable problems are problems for which an algorithm exists that will always produce a correct yes/no answer in a finite amount of time.

Examples:

- Is a number prime?
- Does a string contain a specific substring?
- Is a graph connected?

Undecidable Problems

Undecidable problems are problems for which it can be proven that no algorithm can ever exist to solve them for all possible inputs.

Example: The Halting Problem. Determining, for any arbitrary program and input, whether it will eventually stop running or run forever.

Why it matters: It sets a fundamental limit on what computers can do.

NP-Complete Problems (Introductory Concept)

NP-complete problems are a class of particularly difficult problems for which:

- No efficient (polynomial-time) solution is currently known.
- A proposed solution can be quickly checked/verified if given.

Examples:

- Travelling Salesman Problem.
- Boolean satisfiability (SAT).
- Knapsack problem.
- Graph colouring.

Why it matters: If a fast solution is found for one NP-complete problem, it can be adapted to solve all others. This is one of the most important open questions in computer science (the P vs NP problem).

Summary Table

Type	**Description**	**Example**
Tractable	Solvable in polynomial time	Sorting, searching
Intractable	No known efficient algorithm	Travelling Salesman (large instances)
Decidable	Always produces a yes/no answer	Is a number prime?

Type	Description	Example
Undecidable	No algorithm can exist for all inputs	Halting Problem
NP-complete	Hard, but solutions can be verified quickly	SAT, Travelling Salesman

15.4 Algorithm Complexity, Dry Runs and Optimisation

Dry Running (Tracing) an Algorithm

Dry running (tracing) an algorithm means manually stepping through pseudocode/code line by line with specific sample data, recording the value of each variable at every step in a trace table.

Why dry runs are useful:

- Verify the algorithm behaves as intended.
- Help identify logic errors.
- Understand how the algorithm works.
- Estimate complexity (count operations).

Example Trace Table

Algorithm: Sum numbers from 1 to 5

text

```
total = 0
FOR i = 1 TO 5
    total = total + i
```

ENDFOR

OUTPUT total

Trace table:

Step	i	total	Notes
1	-	0	Initial values
2	1	1	First addition
3	2	3	Second addition
4	3	6	Third addition
5	4	10	Fourth addition
6	5	15	Fifth addition
7	-	15	Output total

Result: total = 15.

Another Example: Linear Search Trace

Algorithm: Search for 90 in [67, 82, 45, 90, 78]

```
found = FALSE
FOR i = 0 TO 4
    IF arr[i] = 90 THEN
        found = TRUE
        OUTPUT "Found at index " + i
    ENDIF
ENDFOR
```

Trace table:

Step	i	arr[i]	found	Notes
1	0	67	FALSE	67 != 90
2	1	82	FALSE	82 != 90
3	2	45	FALSE	45 != 90
4	3	90	TRUE	Found!
5	4	78	TRUE	Loop continues (no break)

Note: If we wanted to stop after finding the target, we would add a break or use a WHILE loop with a condition.

Optimisation Techniques

Optimisation means improving an algorithm's performance, either in time (speed) or space (memory).

Technique	Description	Example
Choosing a better algorithm	Replace a slow algorithm with a faster one	Replace $O(n^2)$ bubble sort with O(n log n) merge sort
Choosing a better data structure	Use a structure with faster operations	Use a hash table (O(1) average lookup) instead of a linear search through a list (O(n))
Reducing redundant work	Avoid recalculating the same value repeatedly	Move calculations outside loops where possible
Memoisation	Cache results of expensive function calls	Fibonacci with memoisation
Minimising nested loops	Restructure logic to avoid unnecessary nesting	Use a single loop with a dictionary instead of two nested loops
Early exit	Stop as soon as the result is found	Break out of a linear search when the target is found

Technique	Description	Example
Precomputation	Calculate values once at the start	Precompute a lookup table

Example: Optimising a Search

Before (linear search, O(n)):

python

```
def find_student(students, name):
    for student in students:
        if student["name"] == name:
            return student
    return None
```

After (hash table / dictionary lookup, O(1) average):

python

```
student_dict = {student["name"]: student for student in students}


def find_student(name):
    return student_dict.get(name)
```

The second version is much faster for large lists.

Example: Optimising with Early Exit

Before:

python

```
def contains(arr, target):
    found = False
    for x in arr:
        if x == target:
            found = True
    return found
```

After:

python

```
def contains(arr, target):
    for x in arr:
        if x == target:
            return True   # early exit
    return False
```

The second version stops as soon as the target is found.

Trade-offs: Time vs Space

Optimising for time (speed) can sometimes increase space (memory) usage, and vice versa. A good programmer balances both according to the requirements of the specific task.

Examples of trade-offs:

Trade-off	Description

Trade-off	Description
Caching	Uses more memory to store results, but saves time
Compression	Saves space but takes time to compress/decompress
Precomputation	Uses memory for lookup tables, saves computation time
Using more memory for data structures	Hash tables use more memory than arrays but are faster for lookups

Key point: There is no single "best" solution. The choice depends on the constraints of the problem (available memory, required speed, etc.).

Choosing the Right Algorithm

When choosing an algorithm, consider:

1. **Input size:** How large will n be?
2. **Time constraints:** How fast does it need to run?
3. **Space constraints:** How much memory is available?
4. **Data characteristics:** Is the data sorted? Are there duplicates?
5. **Frequency:** How often will the algorithm run?
6. **Correctness:** Does it produce the right answer?
7. **Simplicity:** Is the code easy to understand and maintain?

Review Questions

1. Define algorithmic complexity and explain the difference between time and space complexity.
2. Distinguish between best case, worst case, and average case. Give an example for linear search.
3. Why do we usually focus on the worst case?
4. Define Big-O notation and explain its purpose.
5. List the common complexity classes from best to worst.
6. What is the Big-O of: (a) accessing an array element by index, (b) linear search, (c) binary search, (d) bubble sort?
7. Estimate the Big-O of a loop inside another loop, both running n times.
8. Estimate the Big-O of an algorithm that halves the search space each step.
9. Explain why $O(n^2)$ is much worse than $O(n \log n)$ for large n.
10. Define tractable and intractable problems, giving an example of each.
11. Define decidable and undecidable problems, giving an example of each.
12. What is the Halting Problem and why is it important?
13. What are NP-complete problems? Give two examples.
14. What is a dry run? Why is it useful?
15. Draw a trace table for an algorithm that calculates the sum of numbers from 1 to 4.
16. Draw a trace table for a linear search looking for 45 in [67, 82, 45, 90, 78].
17. List five optimisation techniques.
18. Explain how memoisation improves the recursive Fibonacci algorithm.

19. Explain the time vs space trade-off with an example.

20. You need to search a large sorted list frequently. Which algorithm and data structure would you choose and why?

Chapter 16: Exploring Internet Technologies

Chapter Overview

This chapter introduces students to computer networks and the internet. It begins with the fundamentals of computer networks, covering network types (LAN, WAN, MAN, PAN), network models (client-server and peer-to-peer), and the benefits and drawbacks of networking. It then covers network connections, architectures, and hardware, including topologies (bus, star, ring, mesh) and key devices (NIC, switch, router, hub, modem, access point). The chapter explains the OSI model in detail, then covers IP addressing (IPv4 and IPv6, public vs private, static vs dynamic, subnet masks). It continues with network configuration and troubleshooting, data communication modes and transmission media, network and data security, network standards and protocols, and finally internet services and safe online use. By the end of this chapter, students will understand how networks work, how data travels across the internet, and how to stay safe online.

Learning Objectives

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

1. Define a computer network and explain its benefits and drawbacks.
2. Distinguish between LAN, WAN, MAN, and PAN.
3. Compare client-server and peer-to-peer network models.
4. Describe common network topologies (bus, star, ring, mesh).
5. Identify key networking hardware and their functions.
6. Describe the OSI model and the function of each layer.
7. Explain IP addressing (IPv4, IPv6, public, private, static, dynamic).
8. Describe subnet masks and their purpose.
9. Outline basic network configuration and troubleshooting steps.

10. Distinguish between simplex, half-duplex, and full-duplex communication.
11. Describe wired and wireless transmission media.
12. Explain common network security threats and measures.
13. Describe key internet protocols (TCP/IP, HTTP/HTTPS, FTP, SMTP, POP3, IMAP, DNS).
14. Outline the development of the internet.
15. Identify common internet services and safe online practices.

Key Terms

Term	**Definition**
Computer network	Two or more computers/devices connected to share resources and communicate
LAN	Local Area Network. Covers a small area (building, school, home)
WAN	Wide Area Network. Covers a large geographic area
MAN	Metropolitan Area Network. Covers a city or large campus
PAN	Personal Area Network. Very short range for personal devices

Term	Definition
Client-server	Network model where dedicated servers provide services to clients
Peer-to-peer (P2P)	Network model where all computers are equal and share directly
Topology	The physical or logical layout of a network
Bus topology	All devices connect to a single shared central cable
Star topology	All devices connect to a central device (switch/hub)
Ring topology	Devices connected in a closed loop
Mesh topology	Every device connects directly to every other device
NIC	Network Interface Card. Hardware for connecting to a network
Switch	Connects devices in a LAN and forwards data only to the intended recipient
Router	Connects different networks and directs data between them

Term	Definition
Hub	Obsolete device that broadcasts data to all connected devices
Modem	Converts digital data to a signal for transmission and back
Access Point	Allows wireless devices to connect to a wired network
OSI model	A 7-layer framework describing how data is transmitted across a network
IP address	A unique numerical label for a device on a network
IPv4	32-bit IP address format (e.g., 192.168.1.10)
IPv6	128-bit IP address format (e.g., 2001:0db8:85a3::8a2e:0370:7334)
Public IP	Unique across the entire internet
Private IP	Unique only within a local network
NAT	Network Address Translation. Allows many private IPs to share one public IP

Term	Definition
Static IP	Manually assigned and does not change
Dynamic IP	Automatically assigned and may change
DHCP	Dynamic Host Configuration Protocol. Assigns IP addresses automatically
Subnet mask	Identifies which portion of an IP address is network and which is host
Default gateway	The router's address, used to reach other networks
DNS	Domain Name System. Translates domain names to IP addresses
Ping	A command that tests if an address is reachable
Simplex	Data flows in one direction only
Half-duplex	Data flows in both directions, but only one at a time
Full-duplex	Data flows in both directions simultaneously

Term	Definition
Bandwidth	The maximum amount of data that can be transmitted in a given time
Firewall	Monitors and controls incoming/outgoing network traffic
Encryption	Scrambling data so it cannot be read without the correct key
VPN	Virtual Private Network. Creates a secure, encrypted connection over a public network
Protocol	An agreed set of rules for how data is formatted, transmitted, and received
TCP/IP	The fundamental protocol suite of the internet
HTTP/HTTPS	Protocols for transferring web pages (HTTPS adds encryption)
FTP	File Transfer Protocol
SMTP	Simple Mail Transfer Protocol. Sends email
POP3	Post Office Protocol 3. Receives email

Term	Definition
IMAP	Internet Message Access Protocol. Receives email
URL	Uniform Resource Locator. The address of a resource on the web
ISP	Internet Service Provider

16.1 Introduction to Computer Networks

What is a Computer Network?

A computer network is two or more computers/devices connected together to share resources (files, printers, internet access) and communicate.

Types of Networks by Geographic Scope

Type	Full Name	Scope	Example
PAN	Personal Area Network	Very short range (a few metres)	Bluetooth connecting a phone to headphones
LAN	Local Area Network	Small area (single building, school, home)	School computer lab
MAN	Metropolitan Area Network	City or large campus	City-wide cable network
WAN	Wide Area Network	Large geographic area (cities, countries, globally)	The Internet

Benefits of Networking

- **Resource sharing:** Printers, files, internet connection.
- **Easier communication:** Email, messaging, video conferencing.
- **Centralised data management/backup**: Data stored and backed up in one place.
- **Cost savings:** Shared hardware and software licences.
- **Collaboration:** Multiple users can work on the same files.
- **Scalability:** Easy to add new users and devices.

Drawbacks of Networking

- **Security risks:** More entry points for attackers.
- **Dependency:** If the network fails, all connected users are affected.
- **Cost and complexity:** Setup and maintenance can be expensive.
- **Performance issues:** Congestion can slow down the network.
- **Management overhead:** Requires skilled staff.

Network Models

Client-Server Model

Dedicated, powerful computers (servers) provide services/resources (files, web pages, printing) to other computers (clients) that request them.

Advantages:

- Centralised management and security.
- Easier to back up and control.
- Scalable for large organisations.

Disadvantages:

- The server is a single point of failure.
- Requires skilled administration.
- Can be expensive.

Example: A school network where a file server stores all student documents.

Peer-to-Peer (P2P) Model

All computers ("peers") have equal status and can share resources directly with each other, with no dedicated central server.

Advantages:

- Simpler and cheaper for small networks.
- No dedicated server needed.
- Easy to set up.

Disadvantages:

- Harder to manage security and backups centrally.
- Less scalable.
- Performance depends on individual computers.

Example: Two computers in a small office sharing files directly.

16.2 Network Connections, Architectures and Hardware

Network Topologies

Topology is the physical or logical layout of a network.

Topology	Description	Advantages	Disadvantages
Bus	All devices connect to a single shared central cable	Simple and cheap	Whole network fails if the main cable breaks; performance degrades with more devices
Star	All devices connect individually to a central device	If one cable/device fails, only that device is affected; easy to add	Central device is a single point of failure; requires more cabling

Topology	Description	Advantages	Disadvantages
	(switch/hub)	devices	
Ring	Devices connected in a closed loop; data travels around the ring	Predictable performance; no collisions	A break in the ring can affect the whole network
Mesh	Every device connects directly to every other device	High redundancy and reliability	Requires a large amount of cabling; expensive

Star topology is the most common in modern LANs.

Key Networking Hardware

Device	Function
NIC (Network Interface Card)	Hardware that allows a device to connect to a network (wired or wireless)
Switch	Connects multiple devices within a LAN and intelligently forwards data only to the intended recipient device (using MAC addresses)
Router	Connects different networks together (e.g., a home LAN to the wider Internet/WAN) and directs data between them based on IP addresses

Device	Function
Hub (largely obsolete)	A simple device that broadcasts incoming data to all connected devices, regardless of the intended recipient (less efficient/secure than a switch)
Modem	Converts digital data from a computer/network into a signal suitable for transmission over a particular medium (e.g., telephone/cable line), and back again
Access Point	Allows wireless (Wi-Fi) devices to connect to a wired network

16.3 The OSI Model

The OSI (Open Systems Interconnection) model is a conceptual, 7-layer framework describing how data is transmitted across a network, standardising how different hardware/software from different vendors can communicate.

The Seven Layers

Layer	Name	Function
7	Application	Interface for the end-user's software (e.g., web browsers, email clients) to access network services
6	Presentation	Formats/translates data (e.g., encryption, compression, character encoding) between the application and the network

Layer	Name	Function
5	Session	Establishes, manages, and terminates communication sessions between devices
4	Transport	Ensures reliable, complete, ordered delivery of data between devices (e.g., TCP breaks data into segments and reassembles it)
3	Network	Handles logical addressing (IP addresses) and routing of data between different networks
2	Data Link	Handles physical (MAC) addressing and transfer of data between devices on the same local network; organises data into "frames"
1	Physical	The actual physical transmission medium (cables, radio waves) and hardware, transmitting raw bits (0s and 1s)

Mnemonic: "All People Seem To Need Data Processing" (top to bottom, Application → Physical).

Purpose of the OSI Model

- Allows each layer to be developed/upgraded somewhat independently.
- Helps in diagnosing exactly where a network problem is occurring.
- Provides a common language for network professionals.
- Standardises network communication across vendors.

Data Encapsulation

As data moves down the layers, each layer adds its own header (and sometimes trailer). This is called encapsulation. When data moves up the layers at the receiving end, the headers are removed. This is called decapsulation.

Application data

+ Transport header -> Segment

+ Network header -> Packet

+ Data Link header/trailer -> Frame

+ Physical -> Bits

16.4 IP Addressing

An IP (Internet Protocol) address is a unique numerical label assigned to every device on a network, allowing it to be identified and located for communication (a Network-layer concept).

IPv4

The traditional format: four numbers (0-255) separated by dots.

Example: 192.168.1.10

Key points:

- 32-bit address.
- Provides about 4.3 billion unique addresses.
- Now largely exhausted given the number of internet-connected devices worldwide.

IPv6

The newer format, using 128-bit addresses written in hexadecimal groups.

Example: 2001:0db8:85a3::8a2e:0370:7334

Key points:

- 128-bit address.
- Provides a vastly larger address space.
- Designed to solve IPv4 exhaustion.
- Also improves routing and security features.

Public vs Private IP Addresses

Public IP	**Private IP**
Unique across the entire internet	Unique only within a local/private network
Assigned by an ISP	Assigned by the router or network administrator
Directly reachable from the internet	Not directly reachable from the internet
Example: 203.0.113.5	Example: 192.168.1.10

NAT (Network Address Translation): Usually done by a router, allows many devices with private IPs to share one public IP address to access the internet.

Static vs Dynamic IP Addressing

Static IP	**Dynamic IP**
Manually assigned	Automatically assigned

Static IP	Dynamic IP
Does not change	May change over time
Used for servers, printers	Used for most client devices
Requires manual configuration	Assigned by DHCP

DHCP (Dynamic Host Configuration Protocol): A server that automatically assigns IP addresses to devices on a network.

Subnet Mask

A subnet mask is used alongside an IP address to identify which portion of the address refers to the network, and which portion refers to the specific host (device) on that network.

Example:

- **IP address:** 192.168.1.10
- **Subnet mask:** 255.255.255.0

This means the first three numbers (192.168.1) identify the network, and the last number (10) identifies the host.

16.5 Network Configuration and Troubleshooting

Basic Network Configuration Settings

Setting	Description
IP address	Static or dynamic
Subnet mask	Identifies network and host portions
Default gateway	The router's address, used to reach other networks
DNS server address	Used to translate domain names into IP addresses

Common Network Problems and Basic Troubleshooting Steps

1. Check physical connections. Cables plugged in correctly, Wi-Fi enabled, hardware powered on.
2. Check the device's IP configuration. Has it obtained a valid IP address?
3. Use diagnostic commands:
    - ping tests if a specific address is reachable and measures response time.
    - ipconfig (Windows) or ifconfig (Linux/macOS) displays a device's current network configuration.
4. Restart devices. Routers, modems, and the affected computer (resolves many temporary glitches).
5. Check for IP address conflicts. Two devices mistakenly assigned the same IP address on the same network.
6. Verify DNS settings. If websites won't load by name but IP addresses work directly, this often points to a DNS problem.

16.6 Data Communication Modes and Transmission

Modes of Data Transmission (Direction)

Mode	Description	Example
Simplex	Data flows in one direction only	Keyboard sending data to a computer, TV broadcast
Half-duplex	Data can flow in both directions, but only one direction at a time	Walkie-talkie
Full-duplex	Data can flow in both directions simultaneously	Telephone call, most modern network connections

Transmission Media

Wired/Guided Media

Medium	Description
Twisted-pair cable	Common for LANs, relatively cheap but limited speed/distance
Coaxial cable	Used in older cable TV/internet
Fibre-optic cable	Transmits data as pulses of light; very high speed and bandwidth, immune to electrical interference, but more expensive to install

Wireless/Unguided Media

Medium	Description
Radio waves	Wi-Fi, mobile data
Microwaves	Satellite/point-to-point links
Infrared	Very short-range line-of-sight communication

Bandwidth

Bandwidth is the maximum amount of data that can be transmitted over a connection in a given time (usually measured in bits per second, e.g., Mbps/Gbps). It is a measure of a connection's capacity, distinct from actual transmission speed achieved in practice.

Serial vs Parallel Transmission

Serial	Parallel
Sends one bit at a time on a single channel	Sends multiple bits simultaneously on multiple channels
Used for most modern long-distance/network communication	Historically used for short distances; now largely obsolete

16.7 Network and Data Security

Common Network Security Threats

Threat	Description
Unauthorised access/hacking	Gaining access to a network or system without permission
Malware	Viruses, worms, trojans, ransomware spreading across a network
Packet sniffing	Intercepting data travelling across a network
DoS/DDoS attacks	Overwhelming a system with traffic so it cannot function
Man-in-the-Middle attacks	An attacker secretly intercepts/alters communication between two parties
Phishing	Tricking users into revealing credentials via fake messages
Social engineering	Manipulating people into breaking security procedures

Network Security Measures

Measure	Description

Measure	Description
Firewalls	Hardware or software that monitors and controls incoming/outgoing network traffic based on predetermined security rules, blocking unauthorised access
Encryption	Scrambling data using an algorithm and a key so that it is unreadable to anyone without the correct decryption key, protecting data both in transit and in storage
Authentication	Verifying the identity of a user/device before granting access (e.g., usernames/passwords, biometrics, two-factor authentication)
VPN (Virtual Private Network)	Creates a secure, encrypted connection ("tunnel") over a public network (like the internet), protecting data privacy, often used for secure remote access
Access control	Restricting what different users are permitted to see or do on a network, based on their role (e.g., a student account vs an administrator account)
Intrusion Detection Systems (IDS)	Monitor network traffic for suspicious activity
Regular updates and patches	Fix known vulnerabilities

16.8 Network Standards, Protocols and Internet Development

What is a Protocol?

A protocol is an agreed set of rules governing how data is formatted, transmitted, and received, allowing different devices/systems to communicate reliably.

Key Internet Protocols

Protocol	Full Name	Function
TCP/IP	Transmission Control Protocol / Internet Protocol	The fundamental protocol suite of the internet. IP handles addressing and routing of data packets; TCP breaks data into packets, ensures they arrive completely, in order, and error-free (re-requesting any lost packets)
HTTP/HTTPS	HyperText Transfer Protocol / Secure	Protocols governing the transfer of web pages between a web server and a browser. HTTPS adds encryption (via SSL/TLS) for secure communication
FTP	File Transfer Protocol	Used for transferring files between computers over a network
SMTP	Simple Mail Transfer Protocol	Used for sending email
POP3	Post Office Protocol 3	Used for receiving email (downloads and often deletes from server)

Protocol	Full Name	Function
IMAP	Internet Message Access Protocol	Used for receiving email (keeps messages on server, syncs across devices)
DNS	Domain Name System	Translates human-readable domain names (e.g., www.example.com) into numerical IP addresses that computers use to locate each other

Why Standards Matter

- They ensure devices and software from different manufacturers can communicate and interoperate reliably.
- They allow the internet to function as one connected global network.
- They enable competition and innovation.

Brief History and Development of the Internet

- 1960s-70s: Originated from ARPANET, a US research network.
- 1980s: Grew through the development of standardized protocols (TCP/IP).
- 1989-1991: The World Wide Web was invented by Tim Berners-Lee.
- 1990s: Web browsers made the internet publicly and commercially significant.
- 2000s-present: Continued growth through broadband, mobile, and now IoT connectivity.

16.9 Internet Services and Safe Online Use

Common Internet Services

Service	**Description**
World Wide Web (WWW)	Accessing web pages via browsers
Email	Sending and receiving electronic messages
Instant messaging / VoIP	Real-time text, voice, and video communication
File sharing / cloud storage	Storing and sharing files online
Streaming media	Watching/listening to audio and video online
Online gaming	Playing games over the internet
E-commerce	Buying and selling goods and services online
Social media	Sharing and interacting on online platforms

Safe and Responsible Online Use

1. Use strong, unique passwords for different accounts, and enable two-factor authentication where available.
2. Be cautious of phishing attempts. Verify the sender and links before clicking or entering personal information.
3. Limit personal information shared publicly on social media. Review privacy settings regularly.

4. Verify sources before believing or sharing information (avoiding misinformation).
5. Keep software and antivirus tools updated to protect against known vulnerabilities.
6. Be aware of cyberbullying and know how to report/block abusive behaviour, and how to seek help.
7. Understand that anything posted online can be difficult to fully remove ("digital footprint"). Think before posting.
8. Use secure connections (HTTPS) when entering sensitive information.
9. Avoid using public Wi-Fi for sensitive transactions, or use a VPN.
10. Back up important data regularly.

Review Questions

1. Define a computer network and list four benefits.
2. Distinguish between LAN, WAN, MAN, and PAN. Give an example of each.
3. Compare client-server and peer-to-peer network models.
4. Describe four network topologies. State one advantage and one disadvantage of each.
5. State the function of each: NIC, switch, router, hub, modem, access point.
6. What is the OSI model? Name the seven layers in order.
7. Describe the function of the Transport layer and the Network layer.
8. What is the difference between IPv4 and IPv6?
9. Distinguish between public and private IP addresses.
10. What is NAT and why is it used?
11. Distinguish between static and dynamic IP addressing.
12. What is a subnet mask and what is its purpose?
13. List four basic network configuration settings.
14. Describe four troubleshooting steps for a network problem.
15. Distinguish between simplex, half-duplex, and full-duplex communication. Give an example of each.
16. Compare twisted-pair, coaxial, and fibre-optic cable.
17. What is bandwidth?
18. List five network security threats and five security measures.
19. Describe the functions of TCP/IP, HTTP/HTTPS, FTP, SMTP, POP3, IMAP, and DNS.

20. List six guidelines for safe and responsible online use.

Chapter 17: Designing Simple Web Pages

Chapter Overview

This chapter introduces students to web technology and web page design. It begins with how the web works, internet connectivity options, online services, and information literacy. Students will learn how to evaluate websites for credibility and avoid misinformation. The chapter then covers HTML (HyperText Markup Language), the standard language for creating the structure and content of web pages. Students will learn the basic structure of an HTML document and common tags for headings, paragraphs, links, images, lists, tables, and divisions. Finally, the chapter covers CSS (Cascading Style Sheets), used to control the presentation and appearance of web pages. Students will learn the three ways to add CSS (inline, internal, and external), basic CSS syntax, common selectors and properties, and the meaning of "cascading." By the end of this chapter, students will be able to create and style a complete, simple web page.

Learning Objectives

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

1. Explain how the web works (browser, server, DNS, HTTP).
2. Define key web terms: URL, web browser, web server, ISP.
3. Describe internet connectivity options.
4. Explain information literacy and how to evaluate a website's credibility.
5. Identify common signs of misinformation.
6. Describe the purpose of HTML.
7. Write a basic HTML document with the correct structure.
8. Use common HTML tags (headings, paragraphs, links, images, lists, tables, divs, line breaks).
9. Explain the purpose of HTML attributes.
10. Describe the purpose of CSS.

11. Add CSS to a web page using inline, internal, and external methods.
12. Write basic CSS rules using common selectors and properties.
13. Explain the meaning of "cascading" in CSS.
14. Combine HTML and CSS to create a complete, styled web page.

Key Terms

Term	Definition
World Wide Web (WWW)	A system of interlinked hypertext documents accessed via the internet
URL	Uniform Resource Locator. The address of a resource on the web
Web browser	Software used to access and display web pages (e.g., Chrome, Firefox, Edge)
Web server	A computer that stores web pages/files and delivers them to browsers on request
ISP	Internet Service Provider. A company that provides access to the internet

Term	Definition
DNS	Domain Name System. Translates domain names to IP addresses
HTTP	HyperText Transfer Protocol. The protocol for transferring web pages
HTTPS	HTTP Secure. HTTP with encryption (SSL/TLS)
Information literacy	The ability to find, evaluate, and use information responsibly
Credibility	The quality of being trusted and believed
Misinformation	False or inaccurate information, often spread unintentionally
HTML	HyperText Markup Language. The standard language for creating the structure and content of web pages
Tag	A markup element enclosed in angle brackets (e.g.,)
Element	A complete tag pair including content (e.g., Hello)

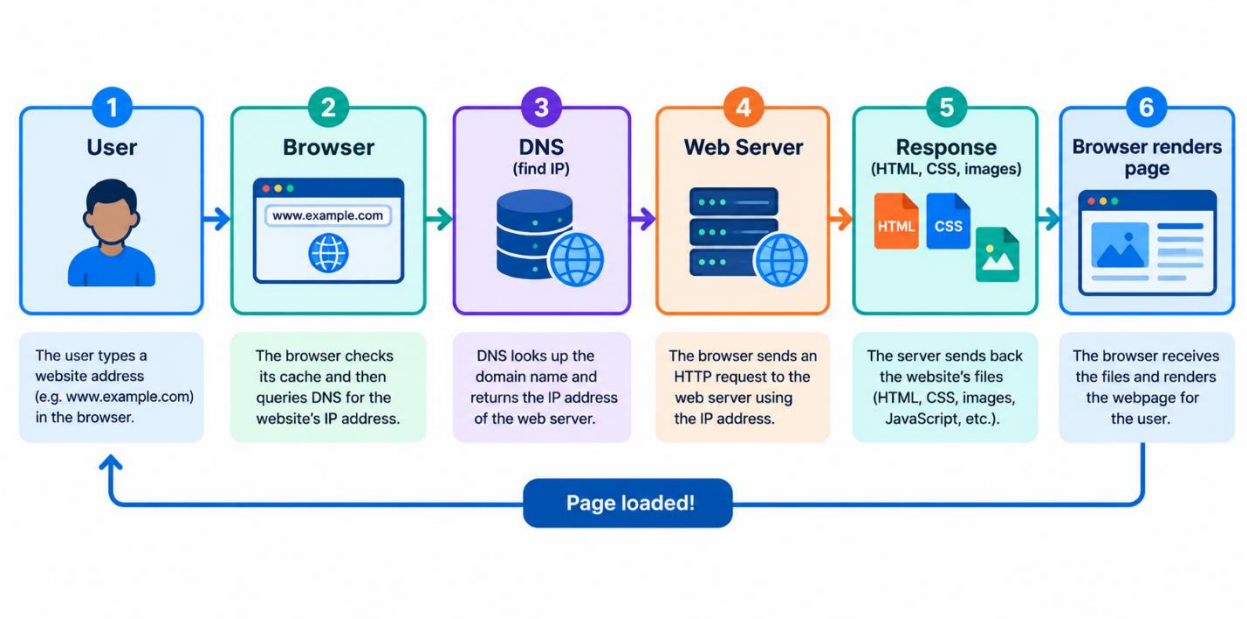
Term	Definition
Attribute	Extra information provided inside an opening tag (e.g., href, src, alt)
Heading	HTML tags <h1> to <h6> for titles and subtitles
Paragraph	<p> tag for text blocks
Hyperlink	<a> tag for linking to another page or resource
Image	<img> tag for embedding images
List	(unordered), (ordered), (list item)
Table	<table>, <tr> (row), <td> (cell)
Div	generic block container for grouping and layout
Line break	 tag for a single line break
CSS	Cascading Style Sheets. Used to control the presentation and appearance of HTML content

Term	Definition
Selector	The part of a CSS rule that targets HTML elements
Property	The aspect of style being set (e.g., color, font-size)
Value	The setting for a property (e.g., blue, 18px)
Inline CSS	CSS applied directly to a single HTML element using the style attribute
Internal CSS	CSS placed inside a <style> element in the document's <head>
External CSS	CSS written in a separate .css file and linked to the HTML document
Cascading	The rules that determine which style applies when several CSS rules could affect the same element
Class selector	Targets elements with a specific class attribute (e.g., .highlight)
ID selector	Targets one unique element with a specific id attribute (e.g., #header)
Element selector	Targets all elements of a given type (e.g., p, h1)

17.1 Introduction to Web Technology, Internet Connectivity, Online Services and Information Literacy

How the Web Works (Simplified)

1. A user types a URL (or clicks a link) in a web browser (the client).
2. The browser uses DNS to find the corresponding IP address of the web server.
3. The browser sends an HTTP/HTTPS request to that server.
4. The server sends back the requested web page (HTML, CSS, images, etc.).
5. The browser renders (displays) the page.



Key Terms

Term	Description
URL (Uniform Resource Locator)	The address of a specific resource on the web, e.g., https://www.example.com/page.html
Web browser	Software used to access and display web pages (e.g., Chrome, Firefox, Edge)
Web server	A computer that stores web pages/files and delivers them to browsers on request
ISP (Internet Service Provider)	A company that provides access to the internet

Anatomy of a URL:

https://www.example.com/page.html

| | |

protocol domain path

Internet Connectivity Options

Option	Description
Broadband (fibre, DSL, cable)	Fixed connections, high speed
Mobile data (3G/4G/5G)	Cellular connectivity, mobile
Satellite internet	For remote areas with no other infrastructure
Wi-Fi	Wireless connection to a local network

Online Services

Covered in Chapter 16 (16.9): email, cloud storage, e-commerce, social media, streaming, etc.

Information Literacy

Information literacy is the ability to effectively find, evaluate, and use information responsibly.

Evaluating a website's credibility:

- Check the author/organisation. Who wrote it? Are they qualified?
- Look for citations/sources. Does the page reference reliable sources?
- Check the publication/update date. Is the information current?
- Consider the domain type. .gov, .edu are often (though not always) more reliable than unknown sites.
- Cross-check facts against multiple independent sources.

- Look for bias. Is the site trying to sell something or push an agenda?

Avoiding misinformation:

- Be sceptical of sensational headlines.
- Verify claims before sharing.
- Recognise that not everything online is accurate or unbiased.
- Use fact-checking websites.
- Check the original source of a claim.

17.2 Introduction to HTML

HTML (HyperText Markup Language) is the standard markup language used to create the structure and content of web pages, using tags (elements) enclosed in angle brackets < >.

Basic HTML Document Structure

```
<!DOCTYPE html>
<html>
<head>
   <title>My First Web Page</title>
</head>
<body>
   <h1>Welcome to My Page</h1>
   <p>This is a paragraph of text.</p>
</body>
</html>
```

Explanation of Structure

Part	Description
<!DOCTYPE html>	Declares the document as HTML5
<html>...</html>	The root element containing the whole page
<head>...</head>	Contains meta-information (not displayed directly), such as the page <title> (shown in the browser tab)
<body>...</body>	Contains all the visible content of the page

Common HTML Tags

Tag	Purpose
<h1> to <h6>	Headings (h1 = largest/most important, h6 = smallest)
<p>	Paragraph of text

Tag	Purpose
<a href="...">	Hyperlink to another page/resource
<img src="..." alt="...">	Embed an image
, ,	Unordered (bulleted) list, ordered (numbered) list, list item
<table>, <tr>, <td>	Table, table row, table cell
	A generic block/container, used for grouping and layout
 	Line break
<hr>	Horizontal rule (line)
,	Bold (strong emphasis), italic (emphasis)
	Inline container for styling part of text
<!-- comment -->	Comment (not displayed)

Example of a Link and Image

Visit Example

<img src="photo.jpg" alt="A description of the photo">

Attributes

Attributes are extra information provided inside an opening tag (e.g., href in <a>, src and alt in <img>), written as attribute="value".

Attribute	Used In	Purpose
href	<a>	The URL the link points to
src	<img>	The path to the image file
alt	<img>	Alternative text if the image cannot be displayed
id	Any element	A unique identifier
class	Any element	A class name for styling
style	Any element	Inline CSS
title	Any element	Tooltip text

Example: A Complete Simple Page

```
<!DOCTYPE html>
<html>
<head>
   <title>My Profile</title>
</head>
<body>
   <h1>My Profile</h1>
   <p>Hello! My name is <strong>Ama</strong>.</p>
   <h2>My Hobbies</h2>
   <ul>
      <li>Reading</li>
      <li>Programming</li>
      <li>Music</li>
   </ul>
   <h2>My Favourite Website</h2>
   <p>Visit <a href="https://www.wikipedia.org">Wikipedia</a>.</p>
   <h2>My Photo</h2>
   <img src="myphoto.jpg" alt="A photo of me">
   <h2>My Timetable</h2>
   <table border="1">
```

```
        <tr>
            <th>Day</th>
            <th>Subject</th>
        </tr>
        <tr>
            <td>Monday</td>
            <td>Computer Science</td>
        </tr>
        <tr>
            <td>Tuesday</td>
            <td>Mathematics</td>
        </tr>
    </table>
</body>
</html>
```

Lists in HTML

Unordered list (bullets):

```
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
</ul>
```

Ordered list (numbers):

```
<ol>
  <li>First</li>
  <li>Second</li>
</ol>
```

Tables in HTML

```
<table border="1">
  <tr>
    <th>Name</th>
    <th>Age</th>
  </tr>
  <tr>
    <td>Ama</td>
    <td>17</td>
  </tr>
  <tr>
    <td>Kofi</td>
    <td>16</td>
  </tr>
</table>
```

Tags:

- <table> starts the table.

- <tr> defines a table row.
- <th> defines a header cell (bold and centred by default).
- <td> defines a data cell.

17.3 Introduction to CSS

CSS (Cascading Style Sheets) is used to control the presentation/appearance of HTML content, such as colours, fonts, spacing, and layout, keeping design separate from structure.

Ways to Add CSS to a Page

1. Inline CSS

Applied directly to a single HTML element using the **style** attribute.

```
<p style="color: blue; font-size: 18px;">Styled text</p>
```

Advantages: Quick for one-off styling.
Disadvantages: Mixes content with presentation; hard to maintain; cannot be reused.

2. Internal CSS

Placed inside a <style> element in the document's <head>, applying to the whole page.

```
<head>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    h1 {
      color: darkgreen;
      text-align: center;
```

```
        }
        p {
            color: navy;
            font-size: 16px;
        }
    </style>
</head>
```

Advantages: Applies to the whole page; better than inline.
Disadvantages: Cannot be shared across multiple pages.

3. External CSS (Best Practice)

Written in a separate .css file and linked to the HTML document, allowing one stylesheet to style multiple pages consistently.

```
<head>
    <link rel="stylesheet" href="style.css">
</head>
```

style.css file:

```
body {
    font-family: Arial, sans-serif;
    background-color: #f0f0f0;
}

h1 {
```

```
    color: darkgreen;
    text-align: center;
}

p {
    color: navy;
    font-size: 16px;
    text-align: justify;
}
```

Advantages: Reusable across many pages; easy to maintain; separates content from presentation.

Disadvantages: Requires an extra file; the page may look unstyled until the CSS loads.

Basic CSS Syntax

```
selector {
    property: value;
    property: value;
}
```

Example:

```
p {
    color: navy;
    font-size: 16px;
    text-align: justify;
```

}

- p is the selector (targets all <p> elements).
- color, font-size, text-align are properties.
- navy, 16px, justify are values.

Common CSS Selectors

Selector	**Example**	**Targets**
Element selector	p	All <p> elements
Class selector	.highlight	Any element with class="highlight"
ID selector	#header	The unique element with id="header"
Universal selector	*	All elements
Group selector	h1, h2, p	All <h1>, <h2>, and <p> elements
Descendant selector	div p	All <p> elements inside a

Class example:

HTML:

```
<p class="highlight">This is highlighted.</p>

<p>This is not.</p>
```

CSS:

```
.highlight {
    background-color: yellow;
    font-weight: bold;
}
```

ID example:

HTML:

```
<div id="header">Welcome</div>
```

CSS:

```
#header {
    background-color: darkblue;
    color: white;
    padding: 20px;
}
```

Common CSS Properties

Property	**Description**	**Example Values**
color	Text colour	red, #ff0000, rgb(255,0,0)

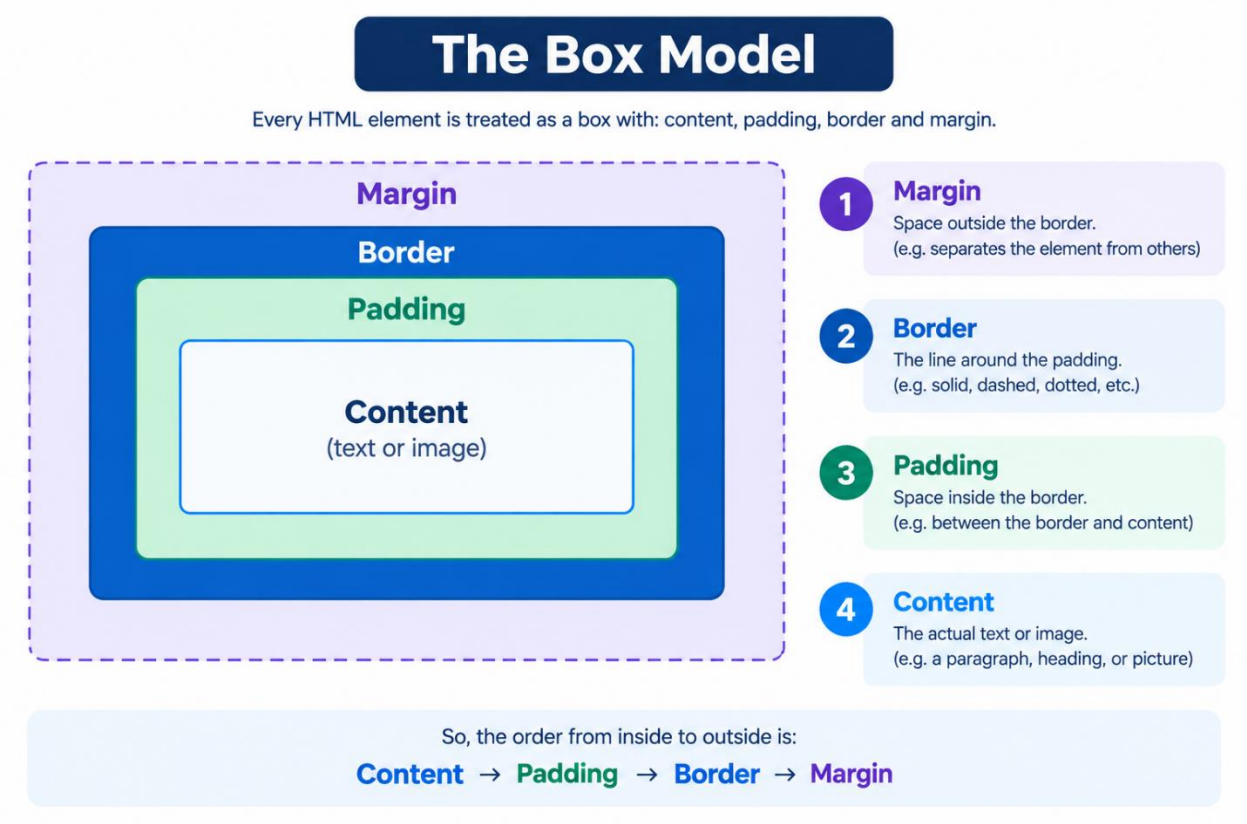
Property	Description	Example Values
background-color	Background colour	white, #f0f0f0
font-family	Typeface	Arial, Times New Roman, sans-serif
font-size	Size of text	16px, 1.2em, 100%
font-weight	Boldness	normal, bold
text-align	Horizontal alignment	left, center, right, justify
margin	Space outside an element	10px, 1em
padding	Space inside an element	10px, 1em
border	Border around an element	1px solid black
width	Width of an element	100px, 50%, auto
height	Height of an element	100px, 50%, auto

Property	Description	Example Values
display	How an element is displayed	block, inline, none

The Box Model

Every HTML element is treated as a box with:

1. Content (the text or image).
2. Padding (space inside the border).
3. Border (around the padding).
4. Margin (space outside the border).



The "Cascading" in CSS

"Cascading" refers to the rules that determine which style applies when several CSS rules could affect the same element.

Priority order (highest to lowest):

1. Inline styles (highest priority).
2. ID selectors.
3. Class selectors.
4. Element selectors.
5. External and internal stylesheets (in order of appearance).
6. Browser default styles (lowest priority).

General rules:

- More specific selectors (ID > class > element) take priority.
- Rules declared later take priority over earlier rules (if specificity is equal).
- !important overrides everything (use sparingly).

Example:

```
p {
   color: blue;
}
.highlight {
   color: red;
}
#special {
```

```
    color: green;
}
```

For <p id="special" class="highlight">, the colour would be green (ID has highest priority among these).

Example: Complete HTML + CSS Page

HTML (index.html):

```
<!DOCTYPE html>
<html>
<head>
    <title>My Styled Page</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <div id="header">
        <h1>Welcome to My Page</h1>
    </div>

    <div class="content">
        <h2>About Me</h2>
        <p>Hello! My name is <strong>Ama</strong>.</p>
        <p class="highlight">I love programming!</p>
```

```
        <h2>My Hobbies</h2>
        <ul>
            <li>Reading</li>
            <li>Programming</li>
            <li>Music</li>
        </ul>
    </div>

    <div id="footer">
        <p>&copy; 2026 Ama</p>
    </div>
</body>
</html>
```

CSS (style.css):

```
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f4;
            margin: 0;
            padding: 0;
        }
```

```
#header {
    background-color: darkblue;
    color: white;
    padding: 20px;
    text-align: center;
}

.content {
    padding: 20px;
    max-width: 800px;
    margin: auto;
    background-color: white;
}

h2 {
    color: darkgreen;
    border-bottom: 1px solid #ccc;
    padding-bottom: 5px;
}

.highlight {
```

```
    background-color: yellow;
    font-weight: bold;
    padding: 5px;
}

ul {
    list-style-type: square;
}

#footer {
    background-color: darkblue;
    color: white;
    text-align: center;
    padding: 10px;
    margin-top: 20px;
}
```

This produces a complete, styled web page with a header, content area, and footer.

Review Questions

1. Describe how the web works, from typing a URL to seeing the page.
2. Define: URL, web browser, web server, ISP.
3. List four internet connectivity options.
4. What is information literacy?
5. List five ways to evaluate a website's credibility.
6. What is HTML and what is it used for?
7. Write the basic structure of an HTML document.
8. Explain the purpose of the <head> and <body> sections.
9. Write HTML code for:
    - A heading "My Page"
    - A paragraph "Hello, world!"
    - A link to https://www.example.com with the text "Visit"
    - An image called "photo.jpg" with alt text "My photo"
10. Write HTML code for an unordered list with three items.
11. Write HTML code for a table with two columns and two rows of data.
12. What is an HTML attribute? Give three examples.
13. What is CSS and what is it used for?
14. List three ways to add CSS to a web page. Which is best practice and why?
15. Write a CSS rule that makes all paragraphs blue and 18px.
16. Write a CSS rule that targets elements with class "highlight" and gives them a yellow background.

17. Write a CSS rule that targets the element with id "header" and gives it a dark blue background and white text.

18. Explain the "cascading" in CSS. What determines which style applies?

19. Draw or describe the CSS box model.

20. Write a complete HTML page with an external CSS file that includes a header, a paragraph, a list, and a footer.