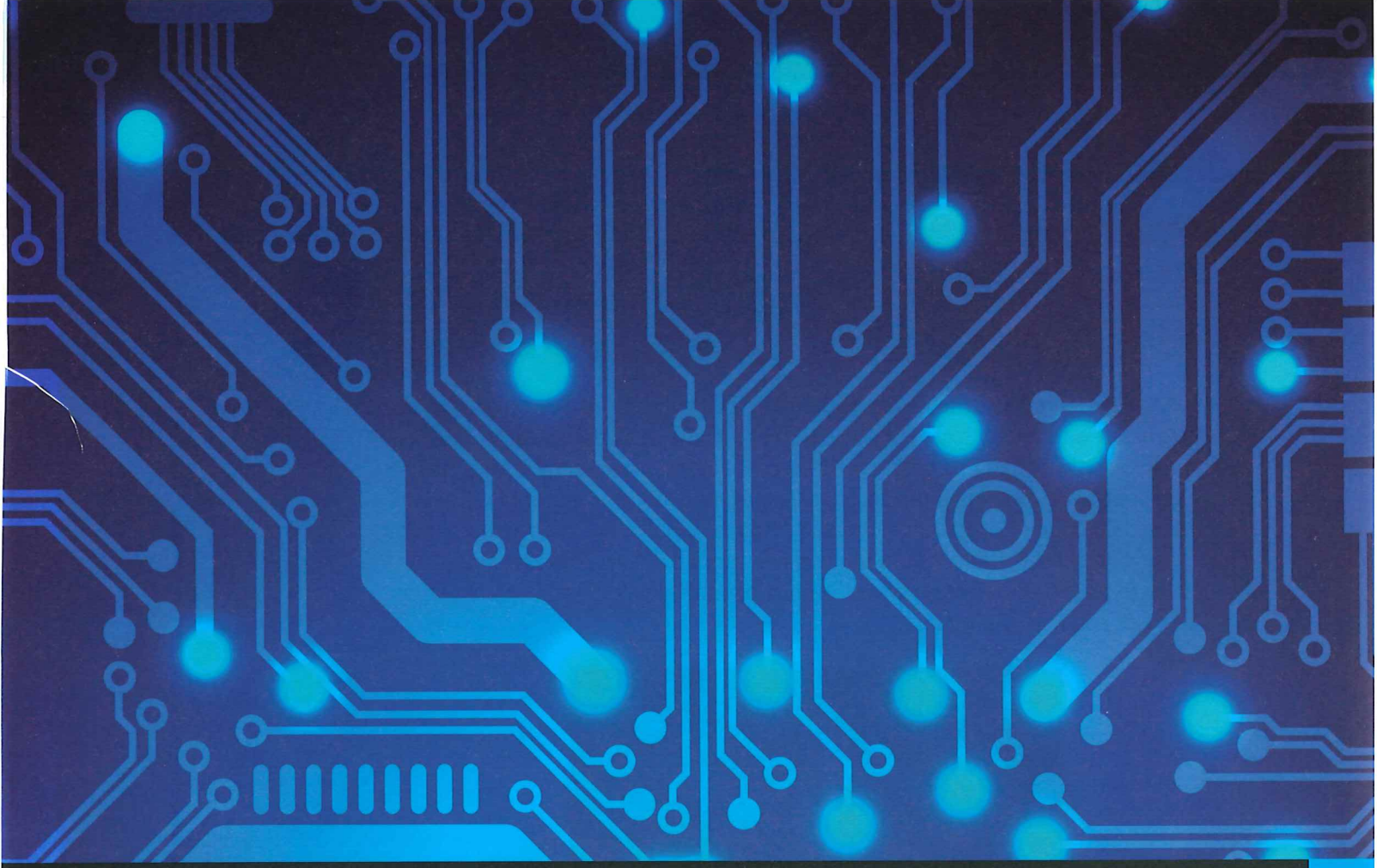
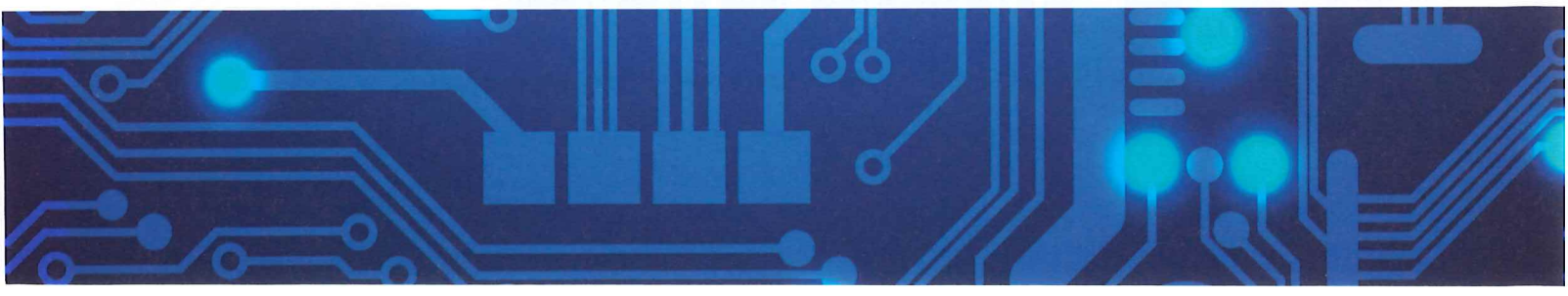


Chapter 3:
Computer Architecture, Languages and Operating Systems

Learning objectives

By the end of this chapter you will:

- understand the basic von Neumann architecture for a computer system
- be able to describe the fetch–execute cycle
- be able to describe the purpose of an operating system
- understand the need for interrupts
- understand what high-level and low-level languages are and why both are needed
- understand the need for compilers, interpreters and assemblers.



3.01 Von Neumann architecture

SYLLABUS CHECK

Show understanding of the basic Von Neumann model for a computer system and the stored program concept (program instructions and data are stored in main memory and instructions are fetched and executed one after another).

In 1945 John von Neumann, a Hungarian mathematician, physicist and inventor, proposed a model to create a computer system. Prior to the von Neumann model computers were usually built to carry out one kind of task. If they were needed to carry out a different task they would need to be completely rebuilt. For example, Alan Turing's Electronic Numeric Integrator and Computer (ENIAC) machine could perform complex calculations, but in order to perform a different set of calculations it would take around three weeks to rewire it. The idea that led to the development of the von Neumann model was to create a machine that would be much easier to re-program.

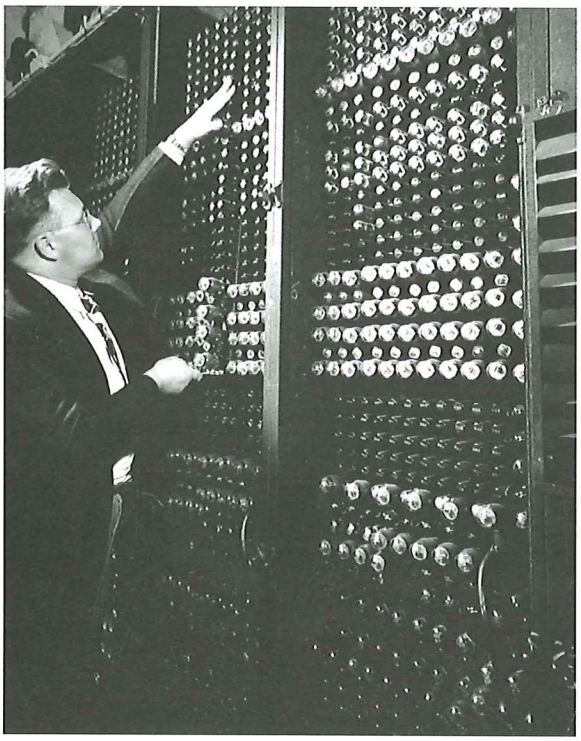


Figure 3.01 The ENIAC Machine

The von Neumann architecture model proposed that not only should the data being processed be stored in memory, but that the instructions being used to process the data should also be stored in the same memory. This would make it much easier to reprogram the system as a different set of instructions could be used to process the data more easily.

The von Neumann architecture, on a simple level, states that a computer system can be thought to consist of three main sections:

- The central processing unit (CPU) – the main processing unit of the system, also known as the processor.

- Storage – Primary memory (also known as main memory) is fast to access and is directly accessible by the processor. RAM and ROM form main memory, along with **cache memory** and **registers**. Secondary storage (also known as backing storage) is slow to access and is not directly accessible by the processor. Instead, communication with secondary memory is through input/output controllers. Hard drives and CD-ROMs are examples of secondary memory.
- Input and output devices – are used to communicate with other devices, for example, a keyboard (input) or printer (output).

To learn more about RAM and ROM, see Chapter 8.

KEY TERM

Cache memory – a portion of memory used for high-speed storage.

Register – an internal memory location within the CPU that temporarily holds data and instructions during processing.

Accumulator – the register that is used for arithmetic and logic calculations.

These three sections can be further separated into several main components:

- **Control Unit (CU)** – This is an internal part of the CPU and it controls the flow of data through the CPU. It also controls the interactions between the different parts of the CPU. It tells the different components, as well as any linked hardware, how to respond to the instructions given.
- **Immediate access store (IAS)** – This is the memory found inside a CPU and is used to hold not only data but also the instructions needed to process that data. It is also more commonly known as the CPU memory. The CPU needs to hold the data and instructions here before processing as it would be much too slow bringing them directly from the main memory to be processed. The data and instructions are firstly loaded into the main memory and then brought into the IAS to be processed.
- **Arithmetic Logic Unit (ALU)** – This is an internal part of the CPU that carries out calculations on data. The arithmetic part uses the usual operators such as multiply, divide, add and subtract. The logic part carries out comparisons such as 'equal to', 'greater than' and 'less than'. Values need to be placed in the **accumulator** for calculations to be carried out.
- **Registers** – These are internal memory locations within the CPU. They temporarily hold data and instructions during processing. Registers are used to move data and instructions into and around the different components of the CPU.
- **Input/Output** – These allow interaction with the computer. Instructions are processed from and to them in the CPU.
- **Buses –** The components of the model need to be connected to one another and this is usually done through buses. A bus is a series of conductors, or pathways, which can be considered a sort of 'highway' for information. Three separate buses are used:
  - The data bus carries the data.
  - The address bus carries the memory address.
  - The control bus carries the instructions.

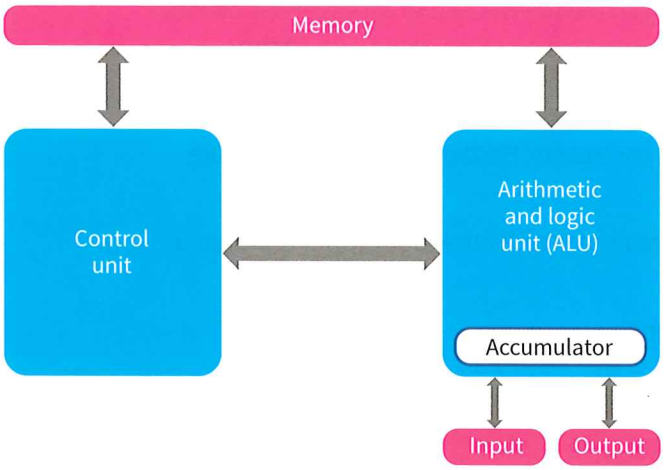


Figure 3.02 The von Neumann architecture

In von Neumann architecture, instructions and data are both stored as binary numbers.

You can learn more about data and instructions stored as binary numbers in Chapter 1.

The stored-program concept

The von Neumann architecture is based on the concept of a stored-program computer. A stored-program computer is a computer that stores programs and instructions in digital memory. This concept is thought to be older than the von Neumann architecture itself and is what inspired its development. A modern stored-program computer is one that keeps its programmed instructions, as well as its data, in read–write, random-access memory (RAM). Before a computer can actually read data, process it and produce information, it must read a set of instructions called a 'program'. The program will state what processing is required. In many computer systems it is possible for more than one program to be stored in the computer at any given time. The only requirement is that sufficient storage locations are available for both the program and the necessary data. This is termed 'multiprogramming'.

The fetch–execute cycle

SYLLABUS CHECK

Describe the stages of the fetch–execute cycle, including the use of registers.

Von Neumann created an architecture with a single processor that follows a linear sequence. This sequence is called the 'fetch–execute cycle':

Step 1 – Fetching the instruction

The CPU fetches the necessary data and instructions and stores them in its own internal memory locations (the IAS).

To fetch the instruction the CPU uses the address bus. The CPU puts the address of the next item that needs to be fetched on to the address bus. Data from this address is then moved from main memory into the IAS in the CPU by travelling along the data bus.

Registers are then used to store and move the data to a special register that will decode the instruction.

Step 2 – Decoding the instruction

The CPU now needs to understand the instruction it has just fetched. To do this it needs to decode the instruction.

Every CPU is designed to understand a specific set of commands. This is called the 'instruction set' and the CPU uses this in order to decode the instruction.

The moving of data and the decoding of instructions is controlled by the CU.

Step 3 – Executing the instruction

Now the CPU understands the instruction, it can execute the instruction.

This is where any processing of the data that is needed for the instruction takes place. The CPU basically carries out the instruction. If any arithmetic calculations are needed in the instruction this will be carried out by the ALU.

Once the CPU has executed the instruction the cycle can begin again for the next instruction.

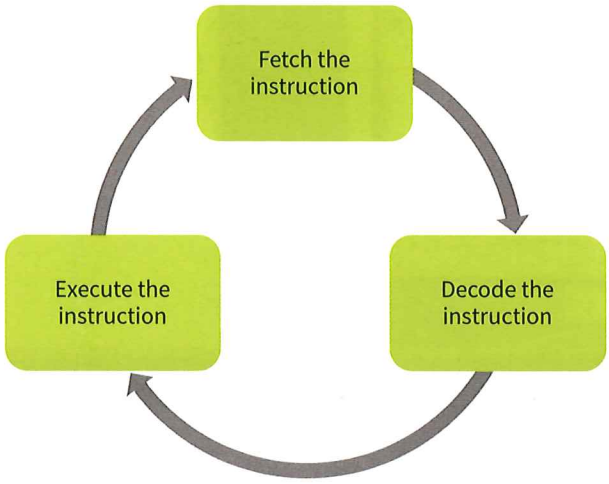


Figure 3.03 The fetch–execute cycle

TEST YOURSELF

1 Describe the function of the arithmetic and logic unit.
2 Explain the stored-program concept.
3 Describe the purpose of buses in the von Neumann architecture.

3.02 Operating Systems

SYLLABUS CHECK

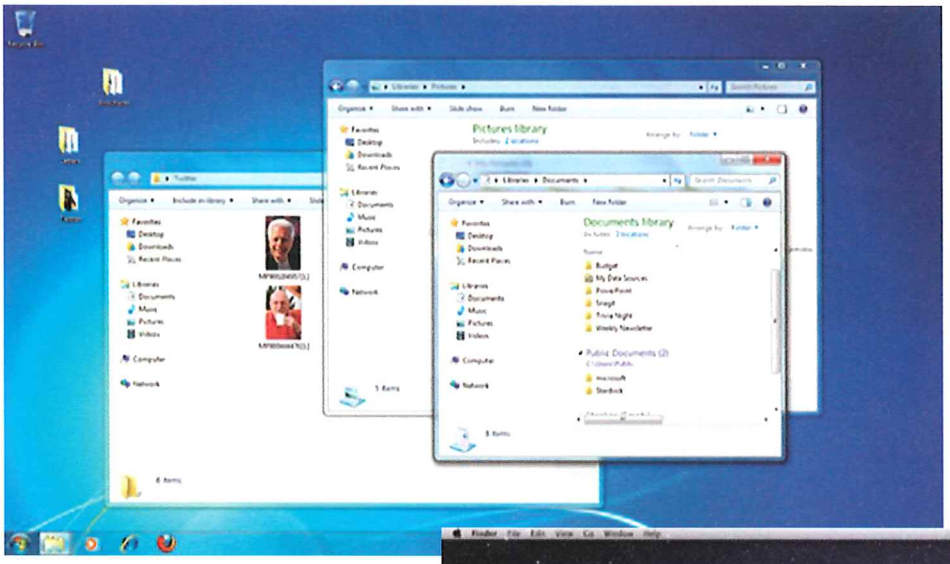
Describe the purpose of an operating system.

Every computer or device needs an operating system (OS) to run other programs. Therefore an operating system is a vital piece of software for a computer. An OS is the framework that allows us to communicate with computer hardware in an interactive way. Without this, we would not be able to tell the computer to do anything and it would not have any instructions to follow.

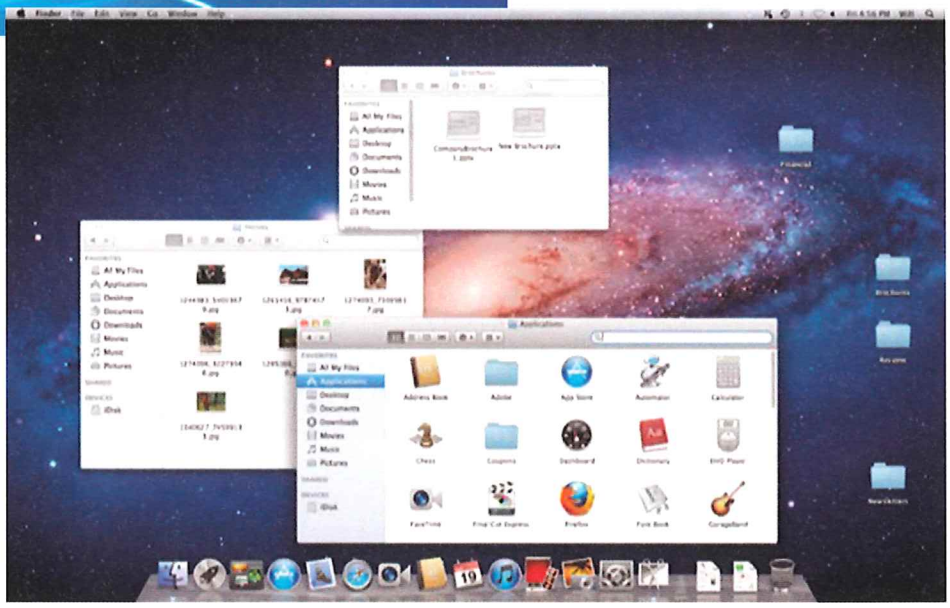
Operating systems perform basic tasks, such as recognising input from the keyboard, sending output to the display screen, keeping track of files and controlling peripheral devices such as printers.

Microsoft Windows is an OS that is available for personal computers and is now also available on some tablet and mobile technologies. Two other examples of OS are MAC OS X and Linux.

A modern OS uses a graphical user interface (GUI) to allow the user to interact with the computer. This kind of interface is made up of icons, buttons and menus that can be clicked to carry out tasks. The layout of these features can differ from company to company with each aiming to create the perfect user layout.



(a)



(b)

Figure 3.04 (a) Windows 7 GUI (b) MAC OS GUI

Before operating systems developed to have a GUI, they had a command line interface (CLI). In this type of interface a user would need to type in the different commands they would like to carry out on the operating system. Some users still use a CLI operating system; an example of this is Linux.

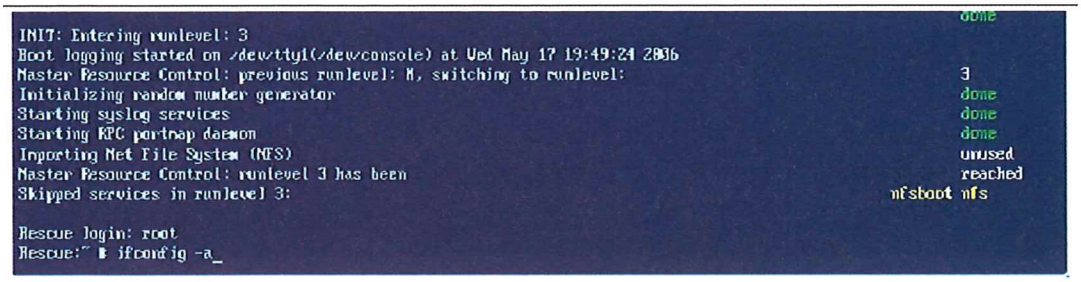


Figure 3.05 CLI operating system Linux

An OS has several main functions:

- It provides the GUI for the user to interact with the computer.
- It manages the hardware and **peripherals** that are connected to the computer.
- It manages the transfer of programs into and out of memory. The OS will manage how the programs will be placed into the available memory.
- It divides the processing time between the different applications that are running. This allows a user to have multiple applications running at a time, such as playing music on the computer whilst completing work in word processing software, as well as finding information on the internet.
- It manages security for the computer, such as anti-virus software and firewalls, and manages access rights (the amount of information a user is allowed access to).
- It manages file handing, allowing users to store, delete and move files on a computer. It keeps track of all actions carried out on files.
- It manages utility software on the computer such as disk defragmentation and disk formatting software.

KEY TERM

Peripheral – a hardware device, used to input, store or output data from a computer, that is not directly part of the computer itself.

Without an operating system a computer would basically be rendered useless as it wouldn't be able to load any software and we would not be able to interact with it.

3.03 Interrupts

SYLLABUS CHECK

Show understanding of the need for interrupts.

An interrupt is a signal. When this signal is received it will inform software that something has happened, an event has occurred. An interrupt could be generated by many sources. It could

be generated by a user pressing keys on a keyboard, it could be from a printer connected to the computer. The signal will come from a device attached to the computer or from a program within the computer.

An interrupt will cause the operating system on a computer to stop and then the operating system will need to work out what to do next. After an interrupt is detected, the operating system will either continue running the program it was currently running or it will begin running a new program.

A computer can only run one program at a time, but because the program it is currently running can be interrupted to run another program, it can perform multiple tasks. It will take it in turns to run the programs so that it appears to be running them at the same time. This is called 'multitasking'. It is the ability to do this that means that we can listen to music on our computer, whilst completing work in a word processing application, whilst surfing the internet for information.

An OS will contain a program called an interrupt handler. The role of the interrupt handler is to prioritise the interrupt signals as it receives them and places them in a queue to be handled.

There are two main categories of interrupts, hardware interrupts and software interrupts. These denote where the interrupt has come from. An example of a hardware interrupt is one that is generated when a peripheral such as a keyboard or a printer produces the signal. An example of a software interrupt is when an application is opened or closed on a system.

TEST YOURSELF

1 Describe the role of an interrupt.
2 Explain why a computer would be useless without an operating system.

3.04 High-level and low-level languages

SYLLABUS CHECK

Show understanding of the need for both high-level and low-level languages.

High-level languages are designed to be used by humans as they are much closer to what humans recognise in terms of language. This means that high-level languages are much easier to read and write programs in than either **low-level languages** or **machine code**.

All computers only process machine code. This is written in binary as a series of 1s and 0s. As humans, we would find it very difficult and time consuming if we had to program using machine code, so programming languages were developed. To learn more about binary numbers, see Chapter 1.

High-level languages

The first high-level languages were developed in the 1950s. One of the very first was FORTRAN (short for 'formula translation'). This was invented in 1954 by John Backus for IBM

to be used as a practical alternative to **assembly language** in programming scientific and engineering projects. FORTRAN was released for sale to the public in 1957. FORTRAN is now more than 60 years old and it is still widely considered an excellent language by scientific programmers. The development of further high level languages followed; the most common include BASIC, C, C++, C#, Pascal, Java and Python.

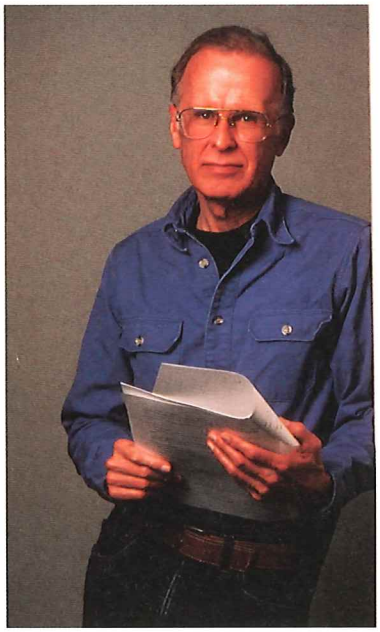


Figure 3.06 John Backus, creator of FORTRAN.

The reason behind the development of high level languages is that they are closer to the language and **syntax** that we as humans understand. This advantages of this are that high-level languages are much easier for humans to read. Also fewer mistakes are made when programming in a high-level language as not as much knowledge is needed for the languages that control the workings of a computer, such as machine code and assembly language.

The features of a high-level language are that it deals with structures such as variables, arrays, loops, conditions, functions and procedures, whereas machine code deals with memory addresses, registers and operation codes. To learn more about variables, arrays, loops, conditions, functions and procedures, see Chapter 11.

KEY TERM

High-level language – a programming language that looks like the language humans generally use.

Low-level language – a programming language that is closer to the native language of computers.

Machine code – a series of binary numbers, made up of 1s and 0s.

Assembly language – a low-level programming language that uses mnemonic codes to create programs.

Syntax – the structure of language in a computer program.

Compiler – a computer program that takes a whole program written in a high-level language and translates it into machine code.

KEY TERM

Interpreter – a computer program that translates a program written in a high-level language line by line into machine code.

Source code – code in its text-based form that is yet to be translated into machine code.

Executable file – a file format that a computer can directly process.

Programmers can create programs and software in a high-level language. This can then be translated into machine code for a computer to process. As well as the advantage of high-level languages being easier to read and write, they are also mostly independent of any hardware they are run on, unlike assembly language which is specific to the hardware it runs on. Therefore a computer programmer can write programs in a high-level language and they can be run on any kind of computer. In high-level languages, one line of code could do several different tasks.

The following code is an example of a program in Python:

```
name = input("What is your name?")
print("Hello " + name)
```

In the first line of code above, the program is creating a variable called 'name'. It is creating a place for the user to input some data. It outputs a question 'What is your name?' for the user. In the variable 'name' it has created, it will store the data that the user inputs. So in one line of code it is doing at least four things.

In the second line of code it is creating an output that will display the text 'Hello'. It is then recalling the data that is stored in the variable 'name' and displaying that also. Therefore in two lines of code this small program is doing at least seven things.

In order to be read by a computer, high-level languages need to be translated into machine code. This translation is done by a **compiler** or an **interpreter**.

A compiler

SYLLABUS CHECK

Show understanding of the need for compilers when translating programs written in a high-level language.

A compiler is a computer program that takes code written in a high-level language and translates it into machine code. The code produced by a programmer is called **source code**. This source code is written generally written in a high-level language and needs to be translated into machine code to be read by the computer. A computer can only process machine code, so unless a high-level language is translated a computer will not understand it.

The act of running the source code through the compiler is called compiling the code. As compiling the code is essential to a computer understanding it, each high-level language has its own compiler.

As a compiler translates the whole of the source code in one go, if there are any errors in the code it will not compile. The compiling process will stop and these errors will need to be removed before the program can be compiled. The machine code from the compiler is output as an **executable file** (.exe). This file will contain the whole of the machine code needed for the CPU to process the program that has been written. The file can then be stored for future use whenever the program is needed.

Error handling is an important feature of a compiler. Most programmers will not write a perfect program on their first attempt. The compiler will run through the program and produce a report of all the errors it detects. A programmer can then use this report to go back to their source code and fix the errors to make their program error free, so that it will compile and run.

An interpreter

SYLLABUS CHECK

Show understanding of the use of interpreters with high-level language programs.

An interpreter is also a computer program that takes code written in a high-level language and translates it into machine code. Unlike a compiler, which takes the whole of the source code and translates it in one go, an interpreter translates the source code into machine code one line at a time.

If there are any errors in the program they will be detected when that line of source code is reached. This is a great advantage of an interpreter; each line of code can be checked as a programmer is writing it, as each line of code is translated in turn. With a compiler, the programmer would need to wait for the whole of the program to be compiled before they can read the error report.

As an interpreter translates each line of code in turn, it takes up less memory than a compiler, which needs to generate an executable file to be stored.

As each line of code is translated immediately, on the surface it seems as if an interpreter would be quicker when translating code. However, as an interpreter has to analyse and translate each line of code to run the program each time, they are mostly slower than compilers, which may initially take time to compile the code, but will then run it much quicker once compiled.

Low-level languages

The raw code that a computer processes is called machine code. In its most raw form this would be a series of binary numbers. Machine code can also be referred to as 'object code'. Machine code is very difficult to understand but it can be represented in a slightly simpler fashion as assembly language.

SYLLABUS CHECK

Show understanding of the need for assemblers when translating programs written in assembly language.

Assembly language assigns **mnemonic codes** to sets of machine code to make them more understandable. Some examples of mnemonics are:

- **ADD** – this would be an instruction to use addition in a calculation
- **SUB** – this would be an instruction to use subtraction in a calculation
- **INP** – this would be an instruction that would require a user to give an input
- **OUT** – this would be an instructions that would output data

Low-level languages are basically a computer's native language. Assembly language and machine code are both examples of a low-level languages. Machine code is directly executable by a computer and assembly language uses software called an **assembler** to be converted into machine code. An assembler is a computer program that will take the basic instructions (mnemonics) used in assembly language and convert them into machine code so they can be processed by the computer. Assembly language was the first progression from machine code to make writing programs simpler. High-level languages then developed at a later stage.

In low-level languages each line of code will perform only one task. The following code is an example of a simple addition program in assembly language with a command and its explanation on the right (after #):

```
INP         #asks the user to give an input e.g. a number
STA ONE     #stores the number input in an address named ONE
INP         #asks the user to give a 2nd input e.g. a number
ADD ONE     #adds the second input to the number stored in ONE
OUT         #outputs the result
HLT         #stops the program
ONE DAT     #assigns the name ONE to the next data location
```

We can see that each line of code performs only one task.

Low-level languages are still used to program certain software. Software applications, such as device **drivers** for hardware, for example a graphics card, need to communicate effectively with the computer they are connected to. This means that when you buy a piece of hardware such as a graphics card or printer you will need to download the correct drivers that enable it to communicate with the computer in which it has been installed. Without the driver, the two cannot communicate. Driver software is written in a low-level language. Low-level languages are mostly still used where very close control of the processing in the CPU is needed.

KEY TERM

Mnemonic codes – instruction codes used in assembly language.

Assembler – converts assembly language into machine code.

Driver – a program that controls a device, for example, a printer or a keyboard.

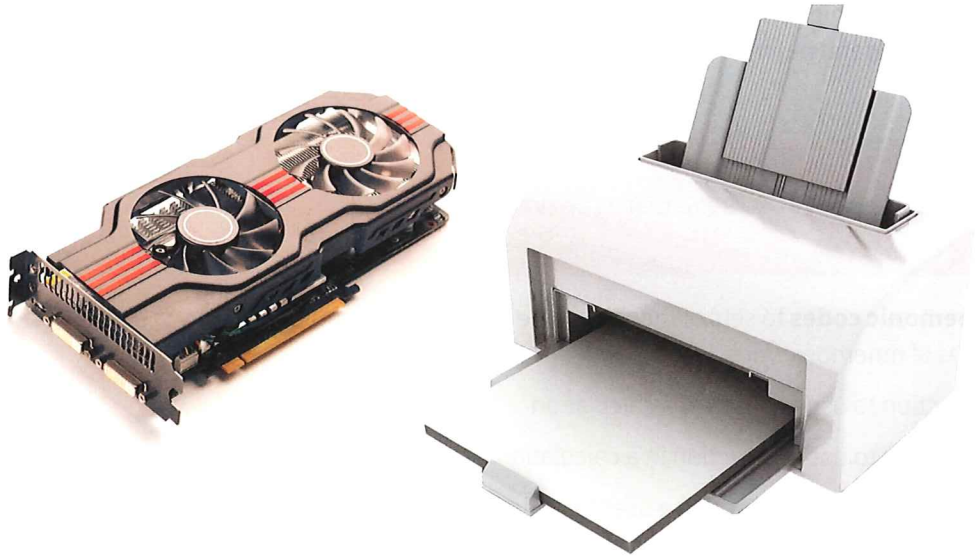


Figure 3.07 A graphics card and a printer need drivers to communicate with a computer; drivers are still mainly written in low-level languages

Table 3.01 compares low-level with high-level languages.

Table 3.01

Low-level language	High-level language
Examples include machine code and assembly language	Examples include C, Java and Python
Difficult for humans to understand but can be easily executed by a computer; a computer's native language	Much easier for humans to understand and much closer to natural language
Needs an assembler to be processed	Needs to be translated by a compiler or an interpreter
One line of code does one thing	One line of code can do several things

Summary

- The von Neumann architecture was designed to be easier to re-program.
- The von Neumann architecture is based on the concept of a stored-program computer, which keeps its programmed instructions as well as its data in read–write, random-access memory (RAM). Before a computer can read data, process it and produce information, it must read a set of instructions called a 'program'. The program will state what processing is required.
- The von Neumann architecture has a single processor that follows a linear sequence, the fetch–execute cycle. The CPU will fetch an instruction from the main memory to its internal memory before decoding and executing the instruction. It does this through the use of registers and buses.
- Every computer or device needs an operating system to run other programs. Operating systems perform basic tasks, such as recognising input from the keyboard, sending output to the display screen, keeping track of files and controlling peripheral devices such as printers.
- An interrupt is a signal that informs software that an event has occurred.
- High-level languages are designed to be used by humans as they are much closer to what humans recognise in terms of language. They need to be run through a compiler or an interpreter to be processed by a computer.
- Low-level languages are basically a computer's native language. Low-level languages can be processed by a computer without being run through a compiler or an interpreter.
- A compiler is a computer program that takes the whole of a program written in a high-level language and translates it into machine code.
- An interpreter is a computer program that translates a program into machine code line by line.
- Assembly language assigns mnemonic codes to sets of machine code to make them more understandable. An assembler converts assembly language back into machine code.

Exam-style questions

1 Describe the fetch–execute cycle carried out in the von Neumann architecture. (3 marks)

2 Define the term 'register'. (2 marks)

3 Draw a line from each term to its correct definition: (4 marks)

Control unit	Connection between components in the CPU along which information travels
Arithmetic and logic unit	Unit that manages the flow of data through the CPU
Accumulator	Register in which values are stored to have calculations carried out on them
Bus	Unit that carries out calculations on data in the CPU

4 Describe three functions of an operating system. (3 marks)

5 Describe two advantages of high-level languages. (2 marks)

6 Explain the difference between a compiler and an interpreter. (2 marks)

7 Describe the role of an assembler. (1 mark)

8 Tick (✓) which statements are true and false about low-level languages: (4 marks)

Statement	True	False
Low-level languages are a computer's native language		
Low-level languages need to be compiled before they can be processed		
Low-level languages are much easier for humans to understand		
In a low-level language one line of code will perform one task only		