# Topic 6 Fundamentals of Computer Systems

**Hardware and Software**

Hardware is the physical parts of the computer system and includes both internal components such as a motherboard or hard drive, and external components such as a keyboard or webcam. If you can physically touch a part of the system, then it is hardware.

Software is computer code and applications which run to perform specific tasks on the system.

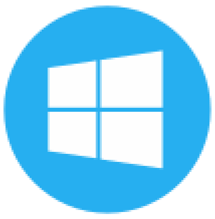
**Software Types**

Application Software

Application software performs a specific task for the person using it, this might be a web browser, presentation software or computer game

System Software

System software controls, maintains and operate the computer system and its components. Its main purpose is to look after the running of the system rather than perform a task for the user. This category includes the operating system, utility programs, library programs and translators.



Operating System

The operating system provides what is known as a virtual machine, which hides the underlying function and complexity of the computer system, allowing the user to easily control and operate it. It also controls and manages the computer's resources through memory management, processor scheduling and interrupt handling.

Utility Programs

Utility programs keep the computer running smoothly and in order, they can be thought of as performing housekeeping tasks. This might include backing up data, defragmenting the drive, and compressing or encrypting data.

Library Programs

These contain small, frequently used functions which programmers can use in their own programs to make their work easier. Programmers have to import the library within their code before they can use it.

Translators

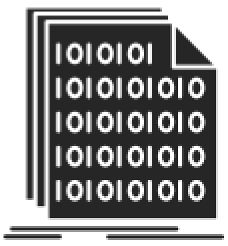
These translate code between different programming languages. The three main types of translators are compilers, assemblers and interpreters.

**The History of Programming Languages**

The first computers were limited in their processing power and memory, meaning programs needed to be as efficient as possible. Low level programming languages are very efficient because they can be executed directly by the processor without manipulation. However, they are complicated to learn and use and very prone to errors.

As processing power increased, high level languages were developed. These are much easier for humans to understand, learn and write in but have to be translated before the processor can understand them. This means they require more processing power and so are less efficient.

## Low Level Languages



Low level languages directly manipulate the processor, and so programs written in these languages are specific to a certain processor.

### Machine Code

Machine code uses binary, meaning each character can be only a 1 or 0, making it difficult for humans to read and understand. Programs written in machine code are long and complicated, making them prone to errors and difficult to debug.

Machine code works directly with the computer's processor making it very powerful and allowing programmers to work unconstrained. Machine code does not need to be translated before being executed, making it efficient and useful for embedded systems and real-time applications where execution speed is important.

```
01010111 01010101 11010100 00101111 00001101 00000001 11110011 00000000
```

### Assembly Language

This was developed to make writing computer programs easier. It uses mnemonics such as ADD and HALT instead of binary making it more compact and less error prone.

Each instruction in assembly language represents one instruction in machine code, giving a 1 to 1 correlation.

AQA have a specific version of assembly language used in exam papers and it is important you are familiar with this.

```
STR R1, #22
ADD R2, R1, 3
MOV R2, R1
HALT
```

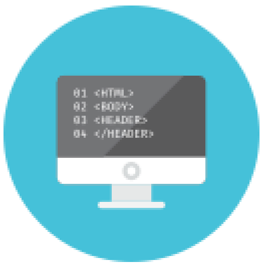
## High Level Languages

High level languages are what most people think of when the term programming language is used and include things like C# and Java. They are not specific to any particular processor, meaning code does not need to be rewritten to run on different processors. However, this also means that the code has to be translated by a compiler or interpreter before it can be executed, making it less efficient.

These languages use English like instructions, which humans can easily understand, making them much easier to work with and debug than low level languages. They also have built in functions to complete common tasks, allowing code to be written more quickly.

Most high level languages have features designed to make them easier to read and understand such as named variables and indentation.

High level languages include imperative high level languages which tell the computer how to perform a task. This is in contrast to declarative programming which tells the computer exactly what to do.



**Translators**

Processors can only execute machine code, and so code written in assembly language, or a high level language must be translated into machine code before they can be executed. Translators are programs which perform this task. Assemblers, compilers and interpreters are all types of translators, each of which operates in a different way.

Assemblers

Assemblers translate assembly language into machine code. Each line in assembly language corresponds to a single line in machine code, this process is quick and straightforward. Assemblers will only work on a single processor type; this is called being processor specific and means that a different assembler is needed for each different processor type.

Compilers

Compilers translate high level programming language code into machine code. They check the high level code for errors then translate the entire program in one go. Source code which contains errors cannot be translated. The compiler produces a compiled program, which can be run on its own without needing any additional software to support it. This also means that the source code cannot be produced from the compiled program, helping to protect the code. Compilers produce machine code, and so are said to be platform specific.

Interpreters

Interpreters translate high level code into machine code one line at a time and have procedures which are used to translate different kinds of program instruction. Interpreters check for errors line by line, meaning code can be partially translated until an error is reached.

Programs translated by interpreters need both the source code and the interpreter to be present in order to run. This also means that the source code is not protected and visible to anyone running the program.

**Compilers and Intermediate Languages**

Some compilers translate source code into an intermediate language (often bytecode), rather than directly to machine code. This allows for platform independence. After translating into the intermediate language, a virtual machine is used to execute the code, and each processor has its own virtual machine.

This allows the source code to be translated one time only and still used with a variety of processor types.

**Source and Object Code**

The code inputted into a translator is called source code. With assemblers, this is assembly language code, whilst for interpreters and compilers this is high level code. The output from the translator is called object code.

**Logic Gates**

Computer processors are made from a huge number of logic gates. These gates apply logical operations to boolean inputs to produce a single boolean output. The processor combines these gates into logic circuits, which perform more complex operations. Each input or output can only be 1 (true) or 0 (false).

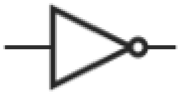
**Truth Tables**

Truth tables show the different combinations of inputs, and the output each would produce when run through a logic gate or circuit. Inputs are labelled alphabetically starting with A and the output is labelled as Q.

**NOT Gate**

The not gate has a single input and a single output. The output is always the opposite of the input.

| A | Q |
|---|---|
| 0 | 1 |
| 1 | 0 |



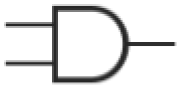
NOT

$$Q = \overline{A}$$

**AND Gate**

This gate has two inputs, it outputs 1 only if both inputs are 1.

| A | B | Q |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



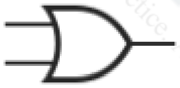
AND

$$Q = A \times B$$

**OR Gate**

The OR gate adds together the inputs, producing the sum of them. It only outputs 0 if both inputs are 0

| A | B | Q |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



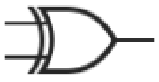
OR

$$Q = A + B$$

**XOR Gate**

XOR is short for exclusively or. This gate outputs 1 only when a single input is 1. If both or neither input is 1 it will output 0.

| A | B | Q |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



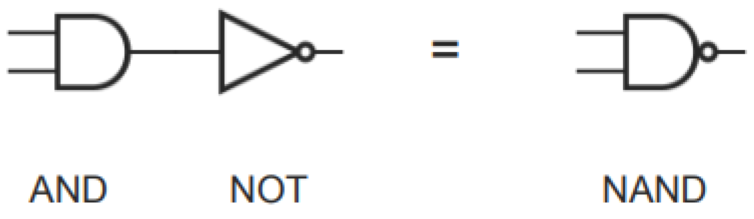
XOR

$$Q = A \oplus B$$

**NAND Gate**

NAND is short for NOT and AND combines the NOT and AND gates into a single gate.

| A | B | Q |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

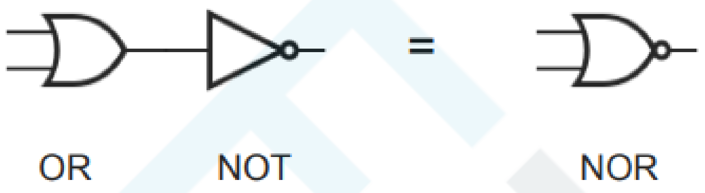


$$Q = \overline{A \times B}$$

**NOR**

The NOR gate combines the NOT and OR gates, and NOR is short for NOT OR.

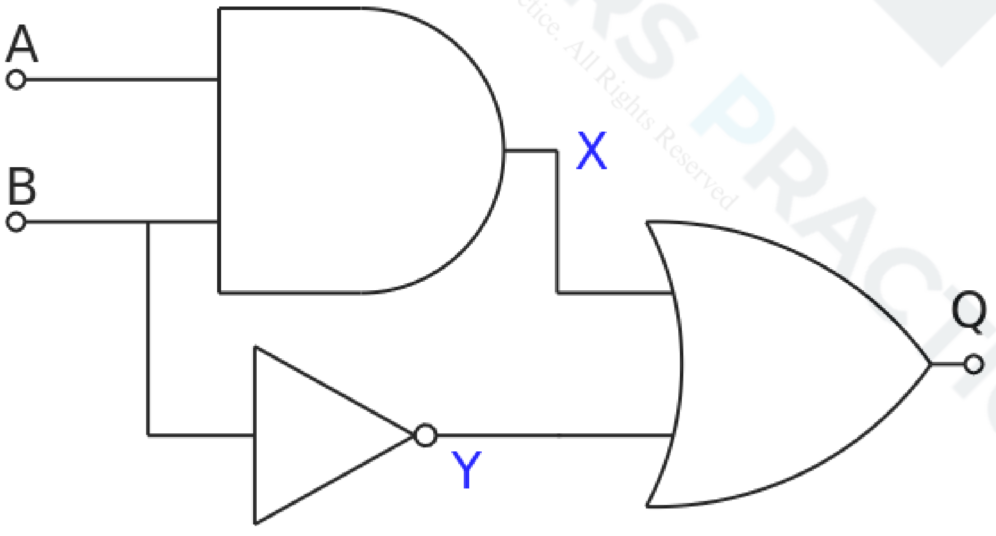
| A | B | Q |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |



$$Q = \overline{A + B}$$

**Logic Circuits**

Logic gates can be joined together to form logic circuits. As with individual gates, we can draw diagrams and produce truth tables for circuits. To make a truth table for a logic circuit, it is usually easiest to work through the circuit and work out the value after each individual gate, as shown below.



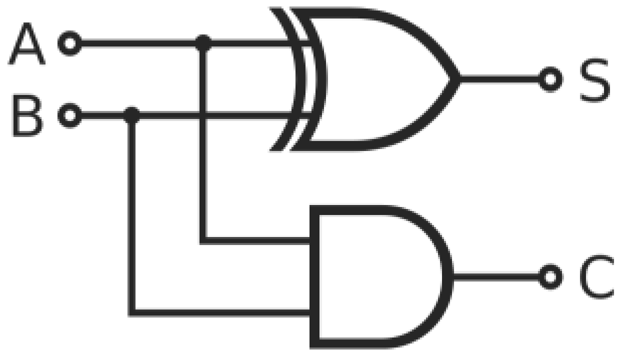
| A | B | X | Y | Q |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |

**Adders**

An adder is a special type of logic circuit which adds two boolean values together.

Half Adders

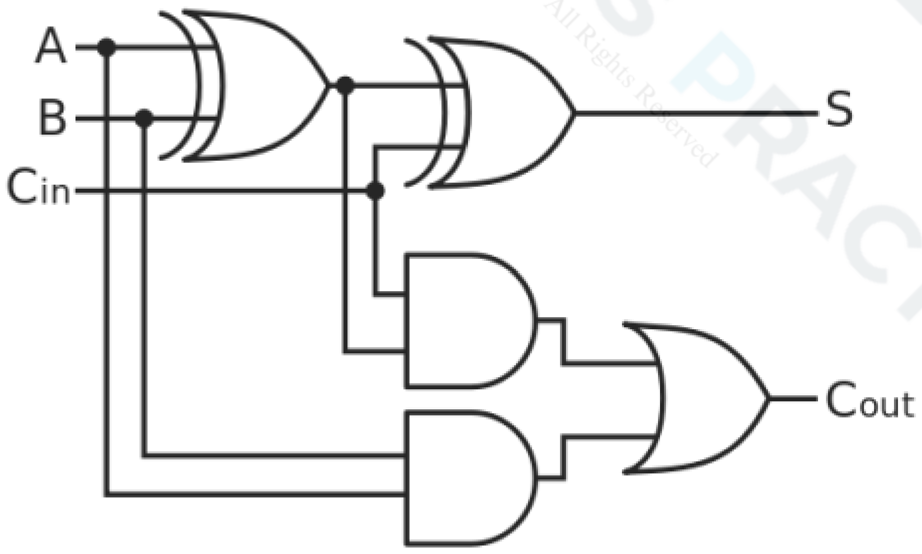
Half adders have two inputs, two logic gates and two outputs. The exam may ask you to draw this circuit.



| A | B | S | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

Full Adders

These circuits have three inputs and two outputs, letting them take two inputs and carry the bit forward from a previous, less significant operation. The exam won't ask you to draw this circuit out, but you do need to be able to recognise it from a drawing.



| A | B | $C_{in}$ | S | $C_{Out}$ |
|---|---|---|---|---|
| 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 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**Edge-Triggered D-Type Flip-Flop**
This is a special type of logic circuit which stores the value of a single bit. It has two inputs, one for data and one for clock. There is a single output which always holds the value of the stored bit.

The clock is generated by the computer, alternating between 0 and 1 whilst the value of the stored bit is set to the value of the data input each time the clock changes. The clock signal can also be used to synchronize several flip flops as a part of a larger system.

**Boolean Algebra**
Much like ordinary maths, boolean algebra represents values using letters, and aims to simplify expressions as much as possible.

| Expression | Meaning |
|---|---|
| A, B, C | A boolean value represented by a letter. |
| $\overline{A}$ | NOT A. The overline is a NOT operation applied to the value beneath the line. |
| A • B<br>$AB$ | A AND B (pronounced A dot B)<br>A AND B |
| A + B | A OR B |

<u>Order of Precedence</u>
Much as with regular mathematics, boolean algebra has an order in which operations should be performed. This order is Brackets, NOT, AND, OR.

**Boolean Identities**

Identities can be used to simplify boolean expressions.

| Expression | Meaning |
|---|---|
| `A • 0 = 0` | Any value AND 0 will always be 0, just as multiplying by 0 is always 0. |
| `B • 1 = B` | Any value AND 1 is always the same value, just as multiplying by 1 is always 1. |
| `C • C = C` | Any value AND itself is the original value. |
| `D + 0 = D` | Any value OR 0 is the original value, since adding 0 makes no change to the value. |
| `E + 1 = 1` | Any value OR 1 is always 1, since it is the equivalent of adding 1 to the value. |
| `F + F = F` | Any value OR itself equals the original value. |
| $\overline{\overline{G}} = G$ | Any boolean value with a double overline has the NOT operation performed on it twice, meaning the value has not changed. |

**De Morgan's Law**

The law can be used to simplify expressions and can be remembered using the phrase "break the bar and change the sign". The bar is the overline representing the NOT operation and the sign means to change between OR and AND.

$$\overline{A + B}$$

Break the Bar:

$$\overline{A} + \overline{B}$$

Change the Sign:

$$\overline{A} \cdot \overline{B}$$

$$\overline{A} \cdot \overline{B} = \overline{A + B}$$

This also works in reverse, by changing the sign and building the bar.

$$\overline{C} + \overline{D}$$

Change the sign:

$$\overline{C} \cdot \overline{D}$$

Build the bar:

$$\overline{C \cdot D}$$

$$\overline{C} + \overline{D} = \overline{C \cdot D}$$

**Distributive Rules**

Distributive rules are used to expand brackets in boolean expressions, for example:

`A • (B + C) = A • B + A • C`