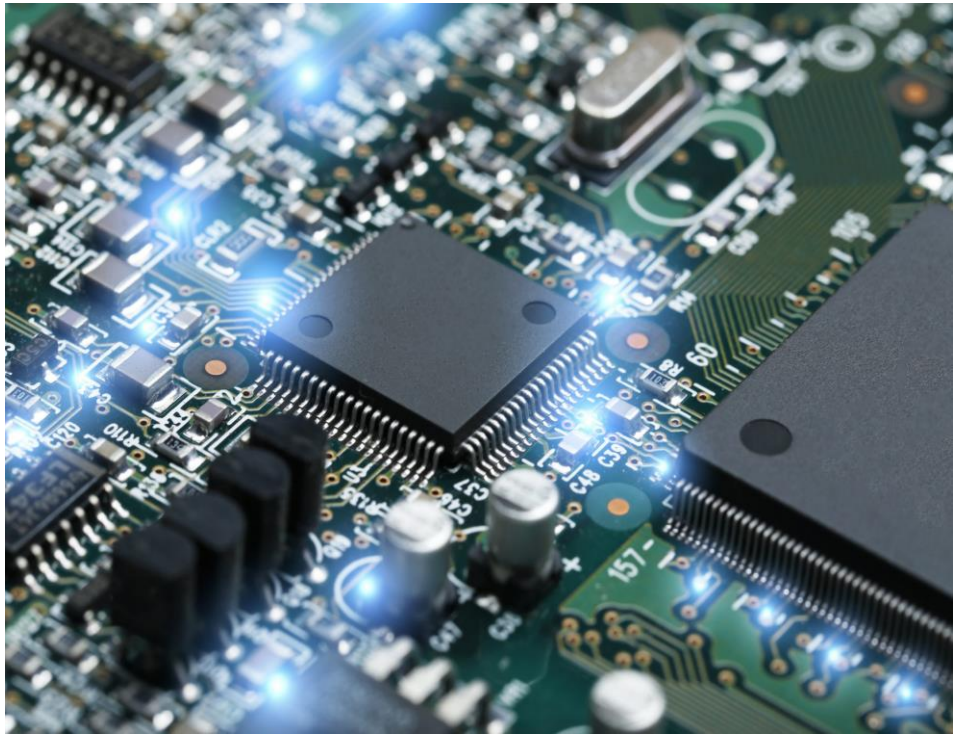
# How do programming languages work ?



## The typical programming work flow

Programs are expressed according to the rules of a given programming language

In most cases, we write code in what is commonly referred to as a *source file*

The task is then to translate this program code to the native language of the computer – machine code

We refer to the process of translation as ***compilation***

Img src: https://railgallery.wongm.com/sunbury-line-disruption-ginifer-2019/F133_6745.jpg.html

# Code Compilation

Files containing programming syntax must be translated to a binary form that the computer can then execute.

Compilation is itself a series of programs performing a a sequence of steps.

The most important components are:

- The Compiler
- The Assembler
- The Linker

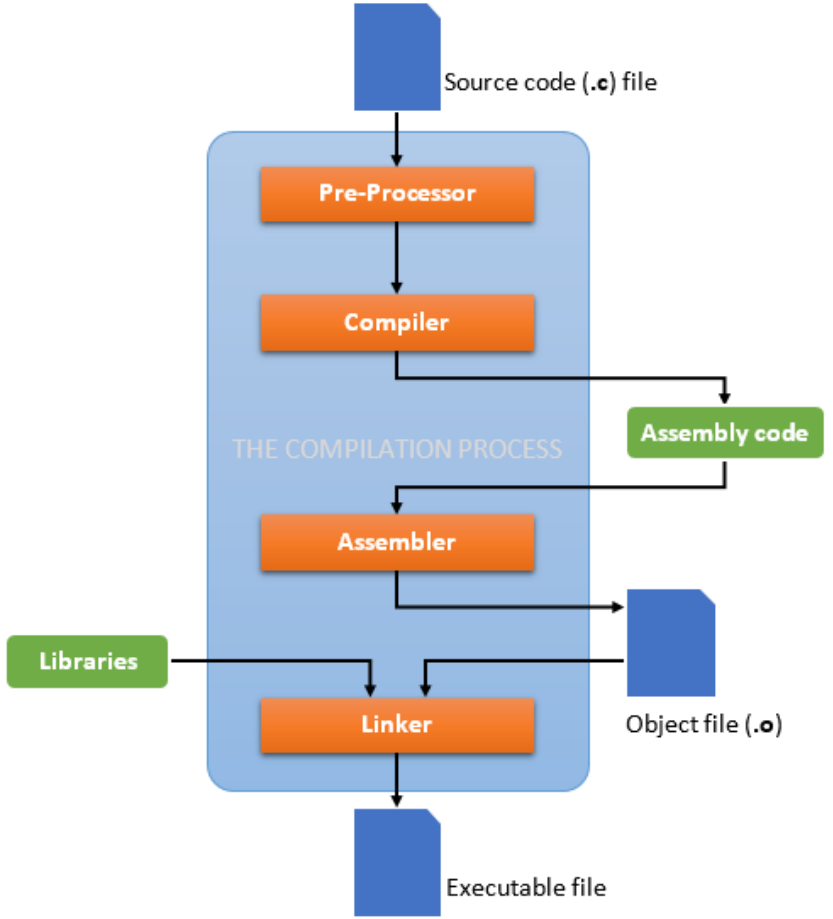



Image src: https://medium.com/@laura.derohan/compiling-c-files-with-gcc-step-by-step-8e78318052

# Code Compilation

## The compiler

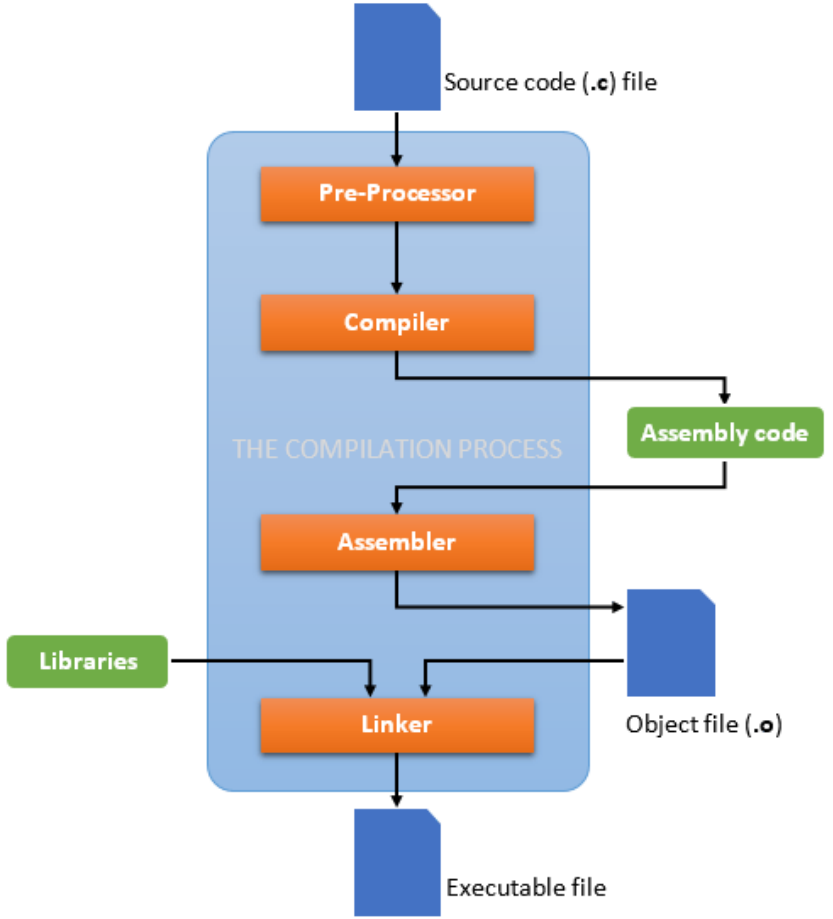
Compilers perform the task of parsing the syntax in the program source file

Program code is tokenised and checked for conformity with the languages rules and grammar

Variables and functions are identified, and symbol tables created to track their existence

Types of different variables and functions (eg., integers, floats, characters, strings etc) are checked for consistency, and validity

And finally, the tokenised program code is translated to a different lower level language, most commonly *Assembly Code*



# Code Compilation

## The assembler

Assembly code is close to machine code, but requires further translation into binary instructions for execution.

This is typically performed by an assembler, the output of which is commonly referred to as object code.

Object code is machine code almost ready to execute, but needs any external code upon which the program relies on to be *linked in*

Most commonly this external code is in the form of software libraries

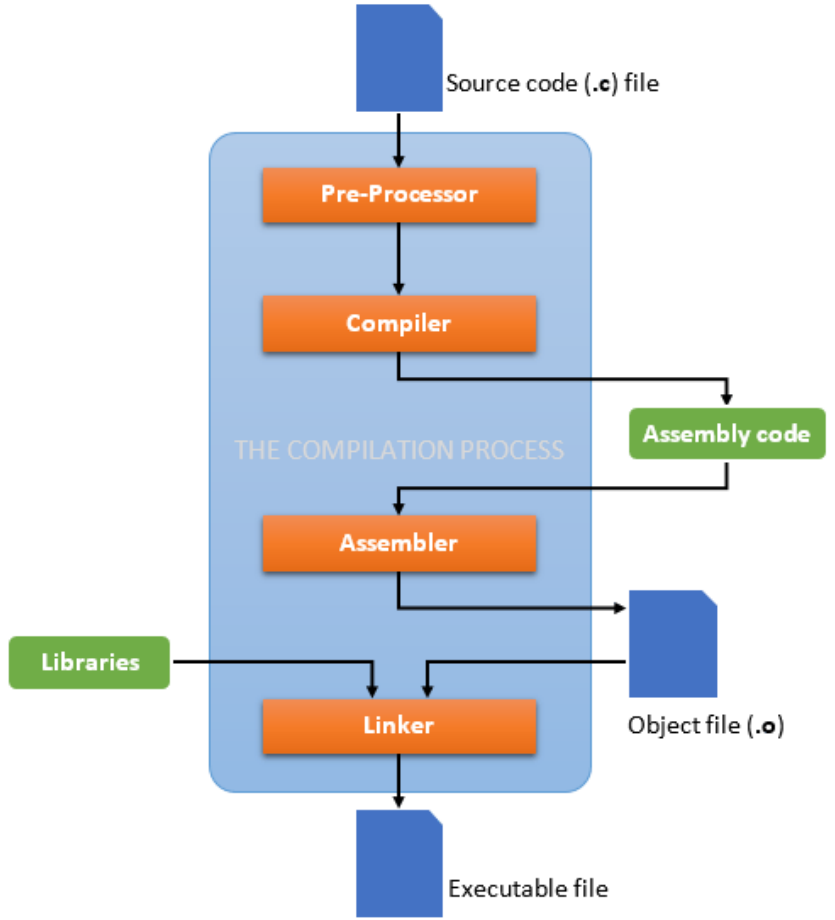
## Software Libraries

Pre-existing code offering specific services, support, or features

Libraries support OS interfacing, screen display, graphics, UI, database management, networking, security, AI, sensor access, mobile dev, … anything

Almost impossible to develop meaningful programs without some library support

Being proficient with a programming language typically entails being familiar with the language *and* the standard libraries that support its use



# Code Compilation

## Software Libraries

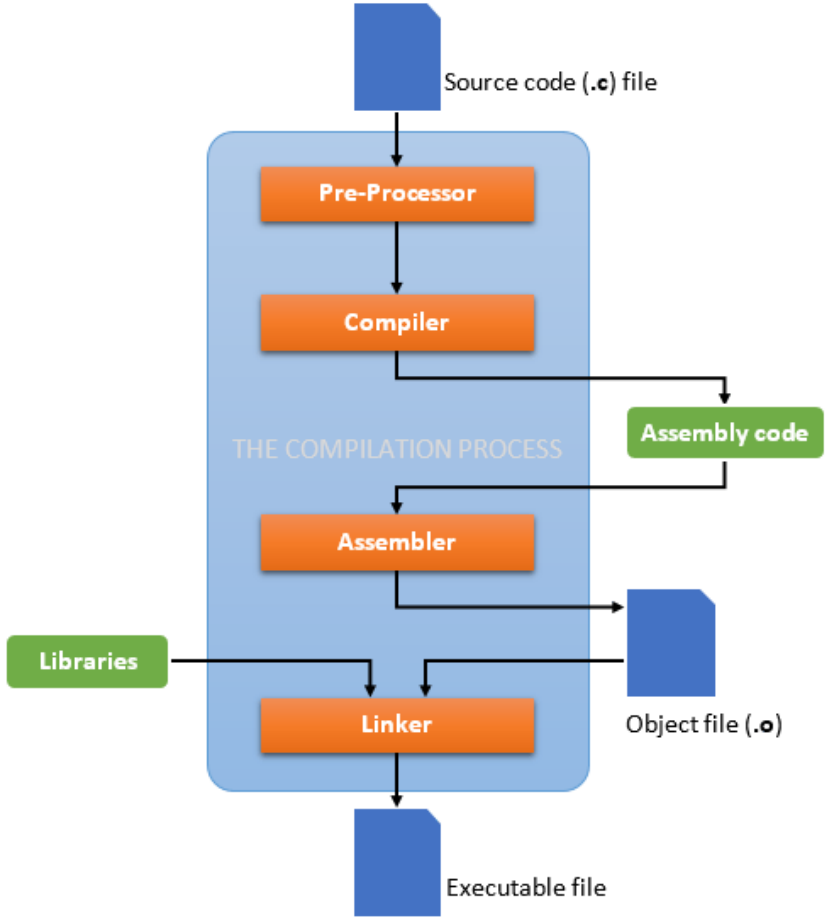
Pre-existing code offering specific services, support, or features.

Abstract away details to offer application programmer simpler, *higher level* interface

Libraries support OS interfacing, screen display, graphics, UI, database management, networking, security, AI, sensor access, mobile dev, … anything

Almost impossible to develop meaningful programs without some library support

Being proficient with a programming language typically entails being familiar with the language ***and*** the standard libraries that support its use



# Code Compilation

## The Linker

The final step in the compilation process

During the compilation process, the compiler identifies references to functions/procedures/processes that exist elsewhere, either in other files you have written, or in libraries.

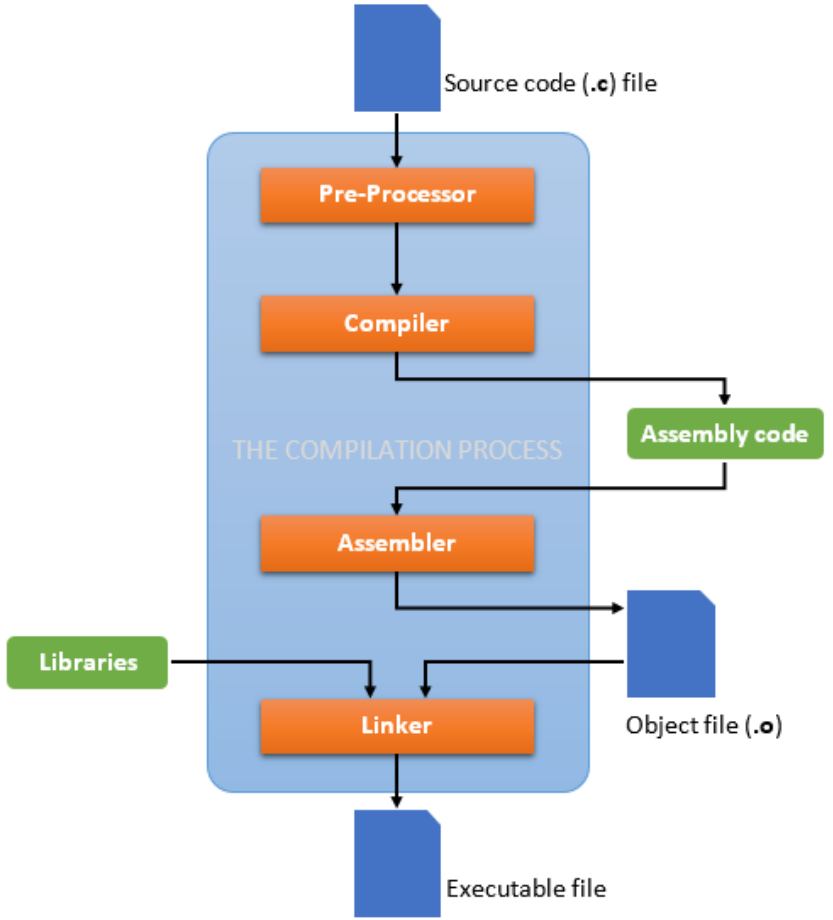
The Linker's task is to lookup this list of unresolved calls and find where the implementation exists in memory

The Linker searches default folders where software libraries are commonly installed, but also looks up any specific configurations provided by the programmer (they may have specified a folder where a library they seek to link in resides).

Some languages support two types of linking: static and dynamic.

*Static linking* ensures all external executable code is linked in before the executable is created. This is generally less likely to cause run time errors, but can result in very large executable files

*Dynamic linking* allows external executable code to be linked in at run time. This can significantly reduce the size of the executable but runs the risk of errors during run time (and a performance hit due to the loading of code from memory during run time)



# Code Compilation

## The executable – Machine Code

Machine code is the binary representation of the executable code

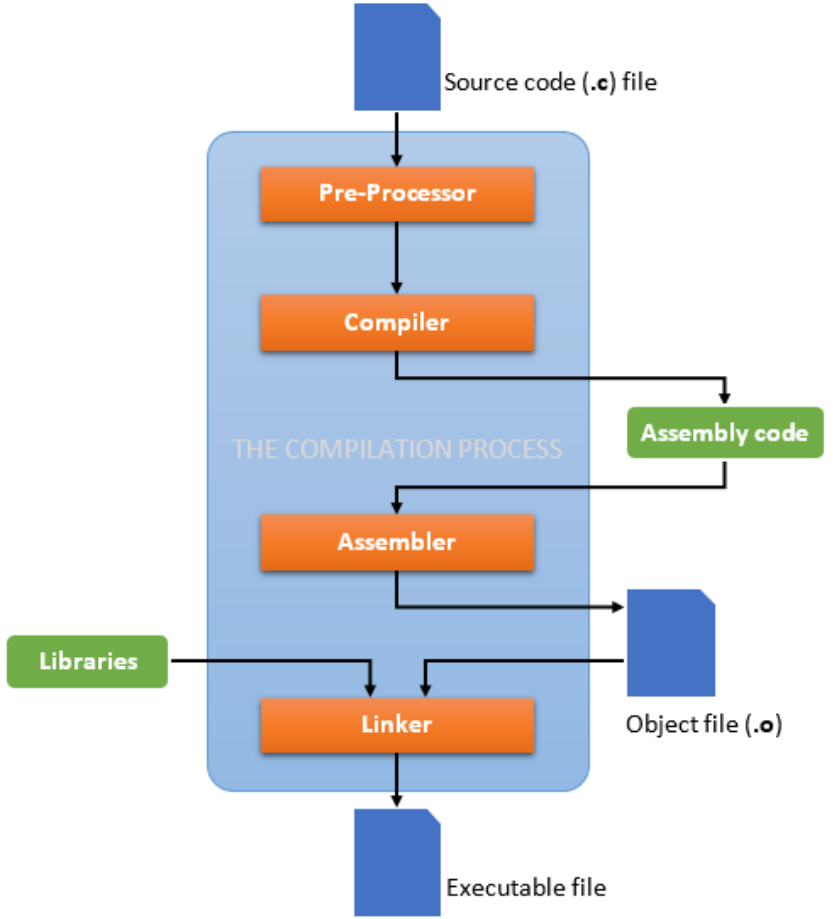
Machine code is defined by the instruction set of the computer's CPU architecture (i.e., the processor)

As such, the translation process must be aware of which CPU architecture it is targeting

Machine code consists of a binary string, most commonly 32 or 64 bits in length, which encodes a specific machine instruction.

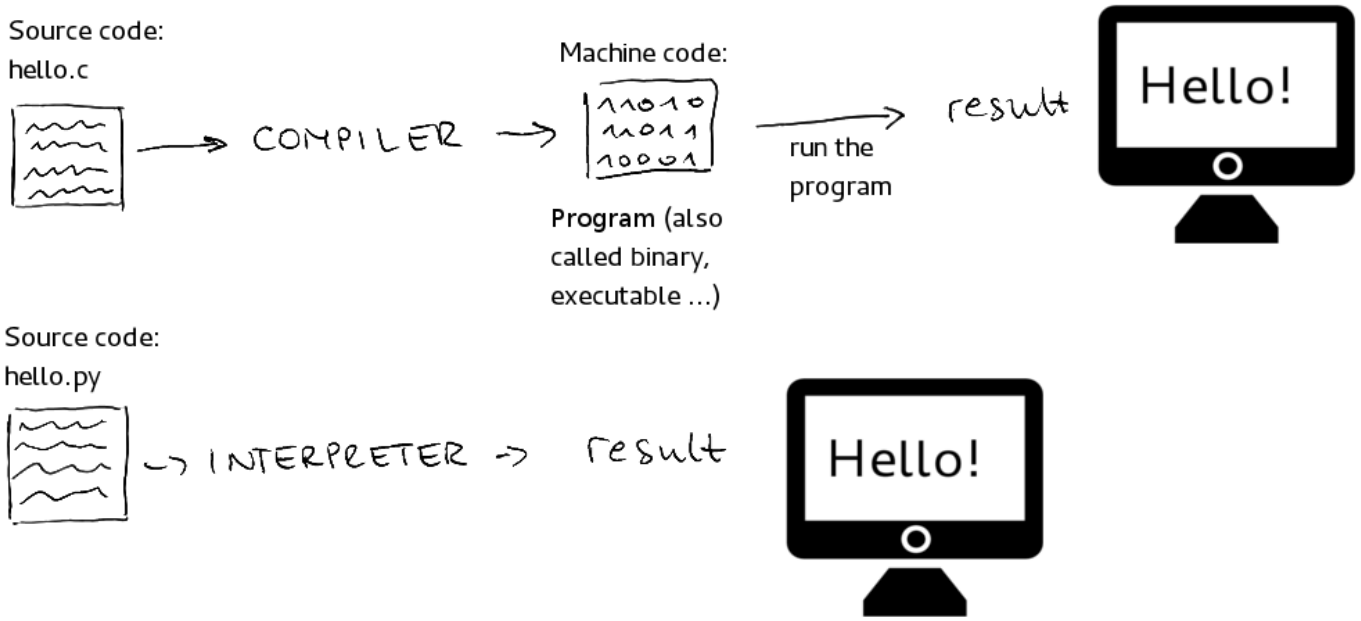
Each machine code instruction is loaded sequentially into the CPU, and operated on using a combination of registers, an instruction decoder, and other ALU components to execute each instruction.

We will discuss this more in the context of ARM assembly programming in the coming weeks



# Code Compilation

**Other pathways to code execution …**




## Interpreted Programs

Programming languages are not all fully compiled to machine executable code at compile time

Some are *interpreted* on the fly during run time

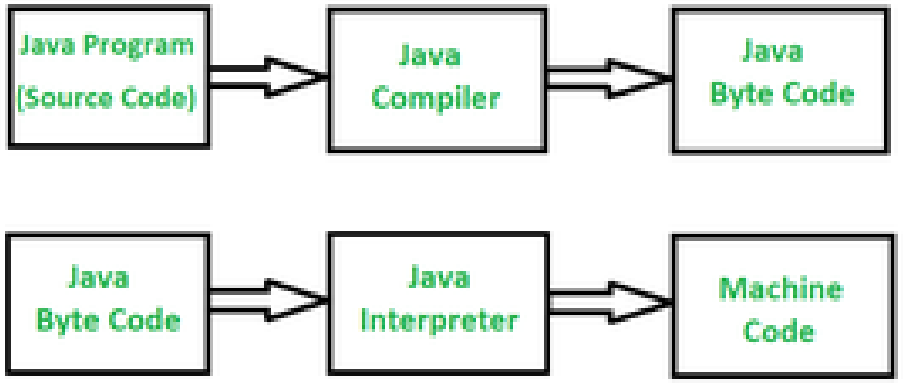
Examples include some high level scripting languages such as Python, Perl or Shell script.

This typically represents a trade-off between:

- the needs of rapid development (often for prototyping or simply performing an operation)
- speed of execution

Img src: https://medium.com/from-the-scratch/stop-it-there-are-no-compiled-and-interpreted-languages-512f84756664

# Code Compilation




## Other pathways to code execution …

## Bytecode and virtual machines

Languages like Java and C# adopt a middle ground, and compile into what is called bytecode

The bytecode is then interpreted within a virtual run time environment (or virtual machine).

- another program designed to translate bytecode into the native machine code of the computer

Bytecode represents a lower level of code and so is more quickly interpreted

The run time environment abstracts away the computer architecture details, allowing the compiled bytecode to be executed on any computer with the run time environment installed (eg., Logisim).

Img src: https://www.geeksforgeeks.org/difference-between-byte-code-and-machine-code/