CHAPTER 1: VM PLATFORM/CONVERSIONS/BOOLEAN

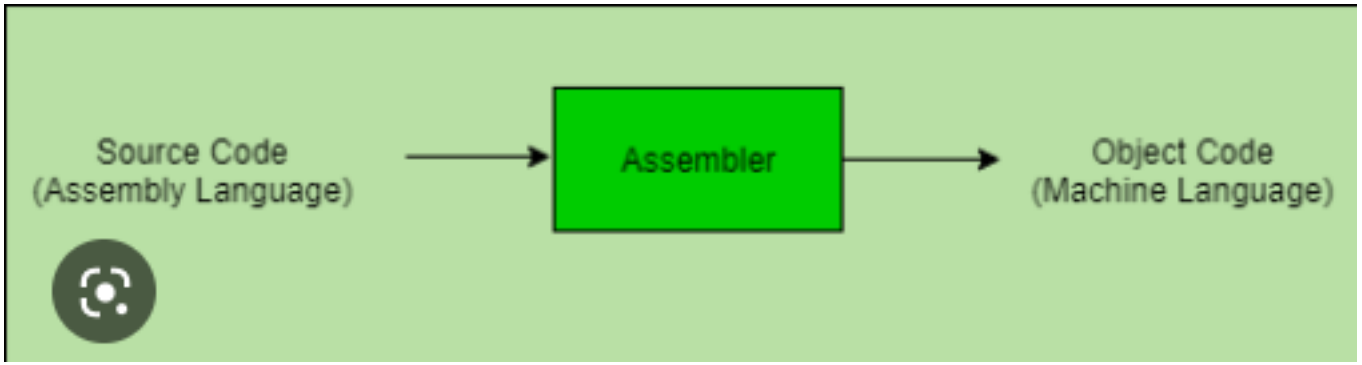
If you are planning to be a C or C++ developer, you need to develop an understanding of how memory, address, and instructions work at a low level.

A lot of programming errors are not eas- ily recognized at the high-level language level.

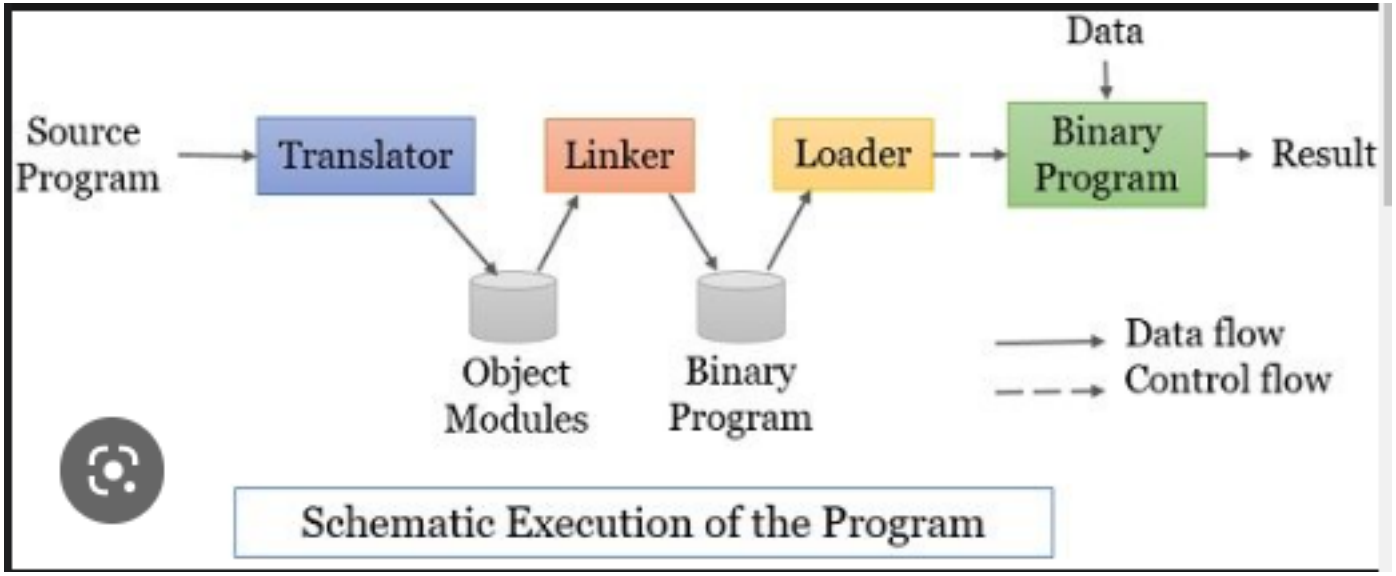
You will often find it necessary to “drill down” into your program’s internals to find out why it isn’t working.

ASSEMBLERS, LINKERS, DEBUGGERS

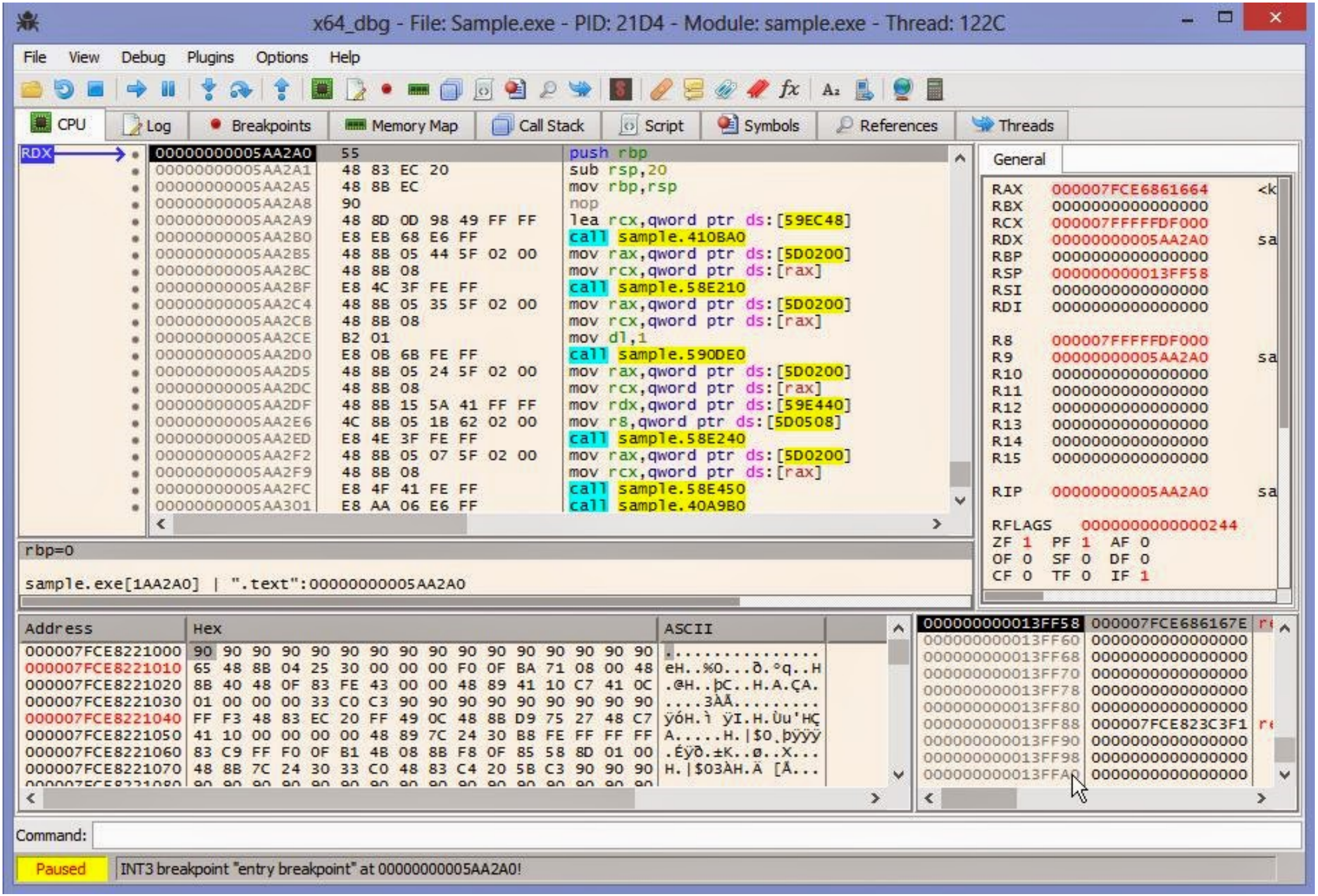
Assembler is a utility program that converts source code programs from assembly language into machine language.



Linkers are programs that combine object code files generated by an assembler or compiler into a single executable program. Linkers resolve any external references between the different modules of the program, such as function calls or variable references, and create a single executable file that can be loaded and run by the computer.



Debuggers, let you to step through a program while it’s running and examine registers and memory. Debuggers are programs that allow developers to find and fix errors in their code. Debuggers provide features such as breakpoints, which allow developers to pause the execution of their program at a specific point and inspect the state of the program's variables and memory. Debuggers also provide tools for examining the call stack, stepping through code one line at a time, and analyzing crash dumps.


x64_dbg - File: Sample.exe - PID: 21D4 - Module: sample.exe - Thread: 122C
File View Debug Plugins Options Help
CPU Log Breakpoints Memory Map Call Stack Script Symbols References Threads
RDX
00000000005AA2A0 55 push rbp
00000000005AA2A1 48 83 EC 20 sub rsp,20
00000000005AA2A5 48 8B EC mov rbp,rsp
00000000005AA2A8 90 nop
00000000005AA2A9 48 8D 0D 98 49 FF FF lea rcx,qword ptr ds:[59EC48]
00000000005AA2B0 E8 EB 68 E6 FF call sample.410BA0
00000000005AA2B5 48 8B 05 44 5F 02 00 mov rax,qword ptr ds:[5D0200]
00000000005AA2BC 48 8B 08 mov rcx,qword ptr ds:[rax]
00000000005AA2BF E8 4C 3F FE FF call sample.58E210
00000000005AA2C4 48 8B 05 35 5F 02 00 mov rax,qword ptr ds:[5D0200]
00000000005AA2CB 48 8B 08 mov rcx,qword ptr ds:[rax]
00000000005AA2CE B2 01 mov dl,1
00000000005AA2D0 E8 0B 6B FE FF call sample.590DE0
00000000005AA2D5 48 8B 05 24 5F 02 00 mov rax,qword ptr ds:[5D0200]
00000000005AA2DC 48 8B 08 mov rcx,qword ptr ds:[rax]
00000000005AA2DF 48 8B 15 5A 41 FF FF mov rdx,qword ptr ds:[59E440]
00000000005AA2E6 4C 8B 05 1B 62 02 00 mov r8,qword ptr ds:[5D0508]
00000000005AA2ED E8 4E 3F FE FF call sample.58E240
00000000005AA2F2 48 8B 05 07 5F 02 00 mov rax,qword ptr ds:[5D0200]
00000000005AA2F9 48 8B 08 mov rcx,qword ptr ds:[rax]
00000000005AA2FC E8 4F 41 FE FF call sample.58E450
00000000005AA301 E8 AA 06 E6 FF call sample.40A9B0
rbp=0
sample.exe[1AA2A0] | ".text":00000000005AA2A0
General
RAX 000007FCE6861664 <k
RBX 0000000000000000
RCX 000007FFFFDF000
RDX 00000000005AA2A0 sa
RBP 0000000000000000
RSP 000000000013FF58
RSI 0000000000000000
RDI 0000000000000000
R8 000007FFFFDF000
R9 00000000005AA2A0 sa
R10 0000000000000000
R11 0000000000000000
R12 0000000000000000
R13 0000000000000000
R14 0000000000000000
R15 0000000000000000
RIP 00000000005AA2A0 sa
RFLAGS 0000000000000244
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1
Address Hex ASCII
000007FCE8221000 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
000007FCE8221010 65 48 8B 04 25 30 00 00 00 F0 0F BA 71 08 00 48 eH..%0...ð.°q..H
000007FCE8221020 8B 40 48 0F 83 FE 43 00 00 48 89 41 10 C7 41 0C .@H..þC..H.A.ÇA.
000007FCE8221030 01 00 00 00 33 C0 C3 90 90 90 90 90 90 90 90 90 ....3ÀÃ.........
000007FCE8221040 FF F3 48 83 EC 20 FF 49 0C 48 8B D9 75 27 48 C7 ÿóH.ì ÿI.H.Ùu'HÇ
000007FCE8221050 41 10 00 00 00 00 48 89 7C 24 30 B8 FE FF FF FF A.....H.|$0¸þÿÿÿ
000007FCE8221060 83 C9 FF F0 0F B1 4B 08 8B F8 0F 85 58 8D 01 00 .Éÿð.±K..ø..X...
000007FCE8221070 48 8B 7C 24 30 33 C0 48 83 C4 20 5B C3 90 90 90 H.|$03ÀH.Ä [Ã...
000000000013FF58 000007FCE686167E
000000000013FF60 0000000000000000
000000000013FF68 0000000000000000
000000000013FF70 0000000000000000
000000000013FF78 0000000000000000
000000000013FF80 0000000000000000
000000000013FF88 000007FCE823C3F1
000000000013FF90 0000000000000000
000000000013FF98 0000000000000000
000000000013FFA0 0000000000000000
Command:
Paused
INT3 breakpoint "entry breakpoint" at 00000000005AA2A0!

Together, assemblers, linkers, and debuggers provide essential tools for creating and maintaining executable code, enabling developers to create efficient, reliable software for a wide range of applications.

FILETYPES CREATED BY MASM

32-Bit Protected Mode: 32-bit protected mode programs run under all 32-bit versions of Microsoft Windows. They are usually easier to write and understand than real-mode programs. From now on, we will simply call this 32-bit mode.

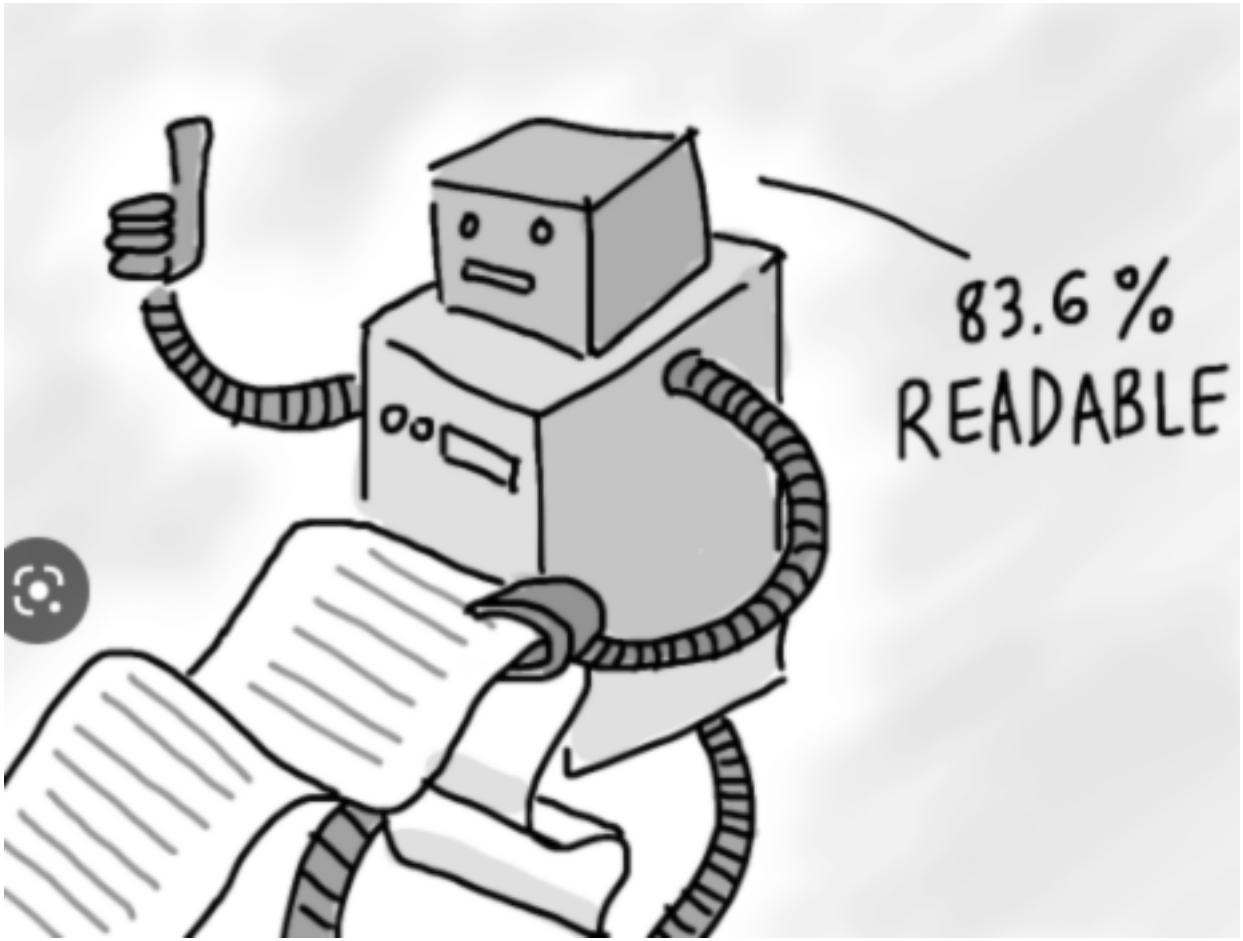
64-Bit Mode: 64-bit programs run under all 64-bit versions of Microsoft Windows.

16-Bit Real-Address Mode: 16-bit programs run under 32-bit versions of Windows and on embedded systems. Because they are not supported by 64-bit Windows, we will restrict discussions of this mode to Chapters 14 through 17.

COMPARE AND CONTRAST ASSEMBLY AND MACHINE LANGUAGE

Assembly language and machine language are both low-level programming languages used to interact directly with computer hardware. However, there are some key differences between the two:

1. Human readability: Assembly language is a symbolic representation of machine language instructions that is easier for humans to read and understand, while machine language is a binary code that is only understandable by the computer.



2. Complexity: Assembly language is a more complex language than machine language since it involves the use of symbols, labels, and mnemonics to represent the various instructions, registers, and memory addresses. Machine language, on the other hand, is a sequence of binary instructions that are executed by the computer's processor directly.

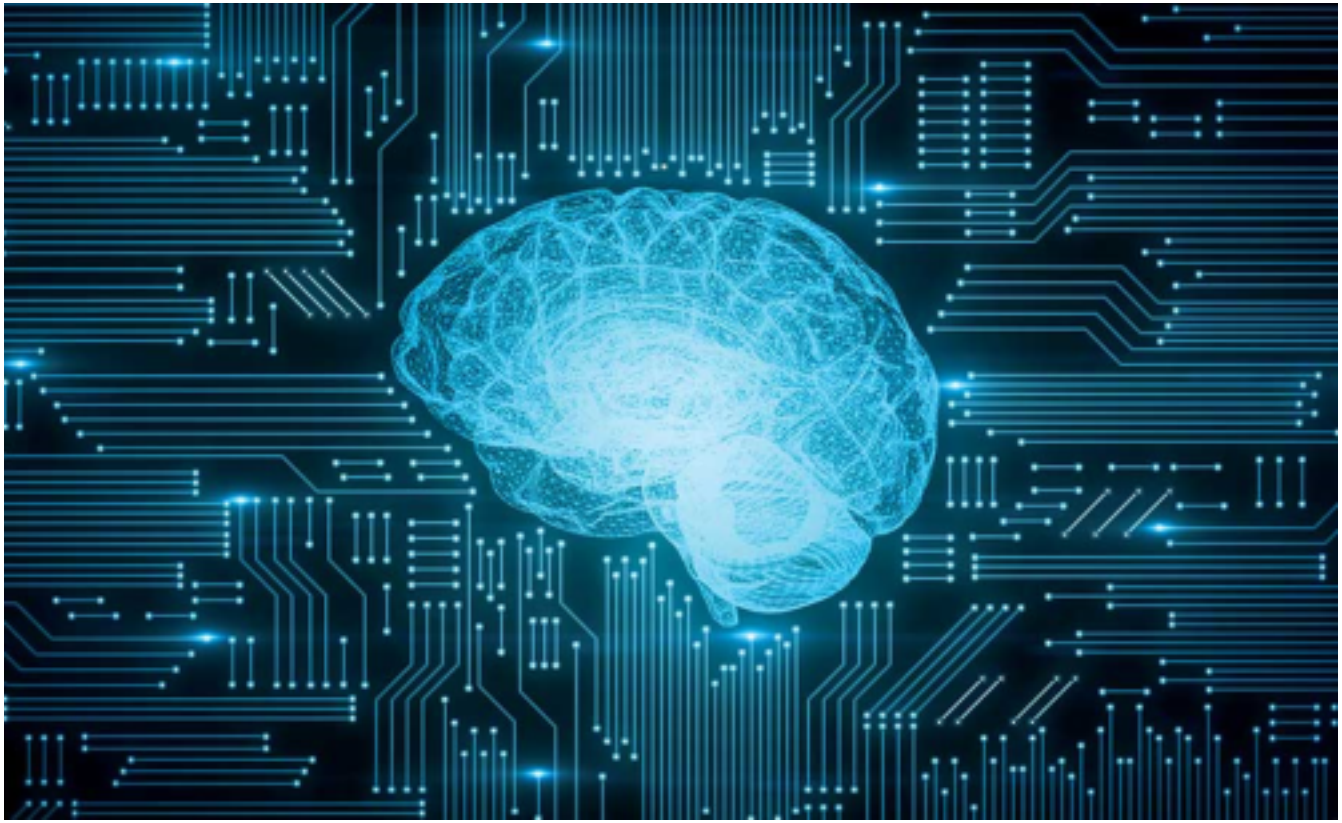
3. Portability: Assembly language is usually specific to a particular processor or architecture, making it less portable than machine language, which can run on any computer that has the same architecture as the code was compiled for.

4. Maintenance: Assembly language code is usually easier to maintain than machine language since it is more human-readable and therefore easier to understand, debug and modify.

5. Efficiency: Machine language is the most efficient code as it is the native code that the computer can execute directly. However, assembly language can still be very efficient, and in some cases, it can be more efficient than high-level programming languages.

6. Assembly language is **time-consuming** and **error-prone** because it requires manual coding of every single instruction.

7. Assembly language is **hardware-dependent**, meaning that code written in one assembly language may not be portable to other systems.



8. **Machine language is the fastest and most efficient** way to communicate with computer hardware because it is the language that the CPU understands.

9. **Machine language programs can be more compact** than assembly language programs because they use binary code, which requires fewer bits than assembly language mnemonics.

In summary, assembly language is a symbolic representation of machine language that is easier to read and write than machine language, but it requires more effort and time than machine language. Machine language is the fastest and most efficient way to communicate with computer hardware, but it

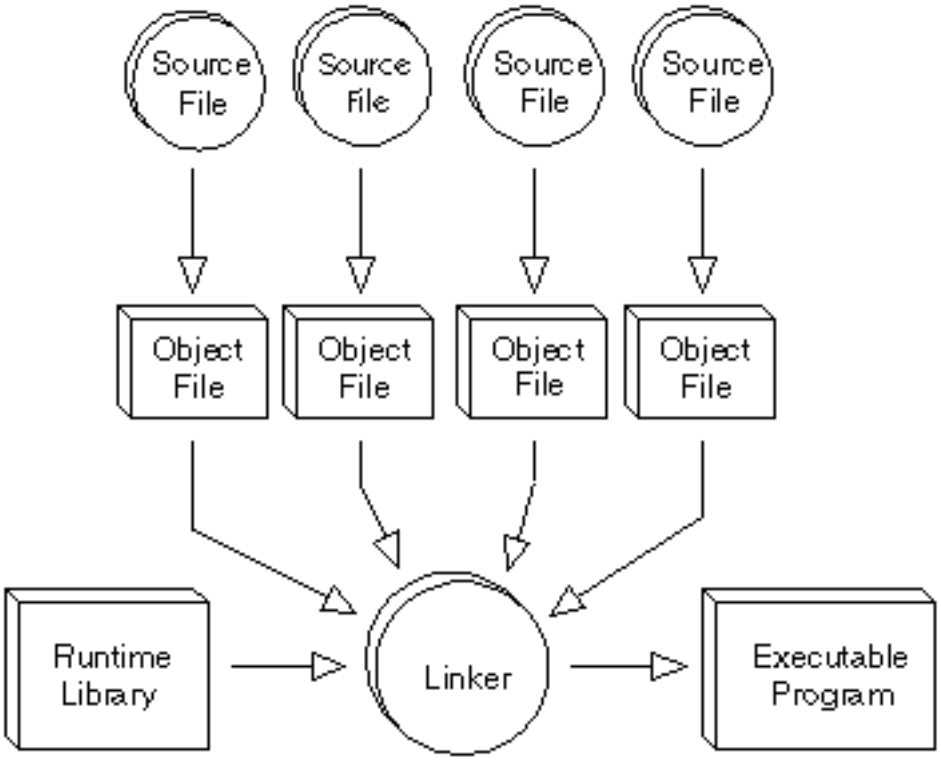
is difficult to read, write, and maintain.

HIGH LEVEL vs LOW LEVEL ASSEMBLY

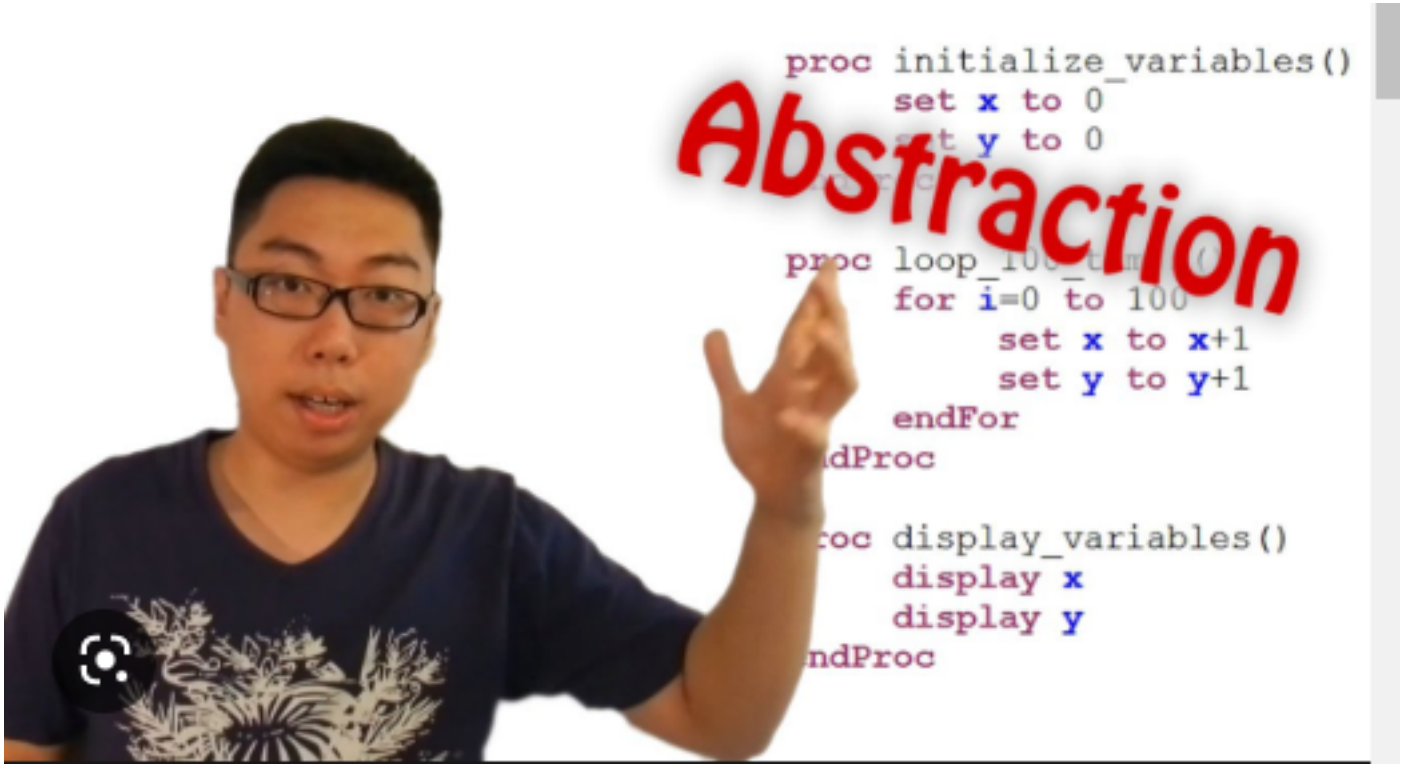
```
;C program vs Assembly program to add and multiply(example)

;                       int Y;
;                       int X = (Y+4)*3;
;
mov eax, Y              ;mov variable Y into EAX register
add eax, 4              ;add 4 to the variable in EAX register
mov ebx, 3              ;mov 3 into EBX register
imul ebx                ;multiply EAX and EBX
mov X, eax              ;mov contents of EAX into variable X
```

Registers are **named** storage locations in the CPU that hold intermediate results of operations.

1. Compilation: C, C++, and Java are all compiled languages, meaning that the code written in these languages is translated into assembly language or machine language instructions by a compiler before being executed on the computer.



2. Abstraction: High-level languages like C, C++, and Java offer a higher level of abstraction than assembly language, allowing programmers to write code that is more readable, maintainable, and easier to understand. These languages provide constructs like variables, functions, and classes that make it easier to manage and manipulate data.



3. Optimization: Although high-level languages are more abstract than assembly language, modern compilers are capable of generating optimized assembly language code that can execute more efficiently than the equivalent assembly language code written by a human.

4. Libraries: High-level languages like C, C++, and Java provide libraries that abstract away many of the low-level details of programming, such as system calls and device drivers. These libraries provide a higher level of abstraction and make it easier to develop complex software applications.

5. Portability: One of the key advantages of high-level languages is their portability. Because high-level languages are compiled into machine language or bytecode, they can be executed on any platform that has a compatible runtime environment. This means that code written in C, C++, or Java can be run on a wide range of hardware and software platforms.

Is Assembly Language Portable? A language whose source programs can be compiled and run on a wide variety of computer systems is said to be portable. A C++ program, for example, will compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system. Assembly language is not portable, because it is designed for a specific processor family.

Assembly language has fewer syntax rules compared to high-level languages like **C++ or Java**, but it also requires a lot of **debugging** due to its low-level data access. Assembly language can **access any memory address**, unlike Java, which restricts specific memory addresses.

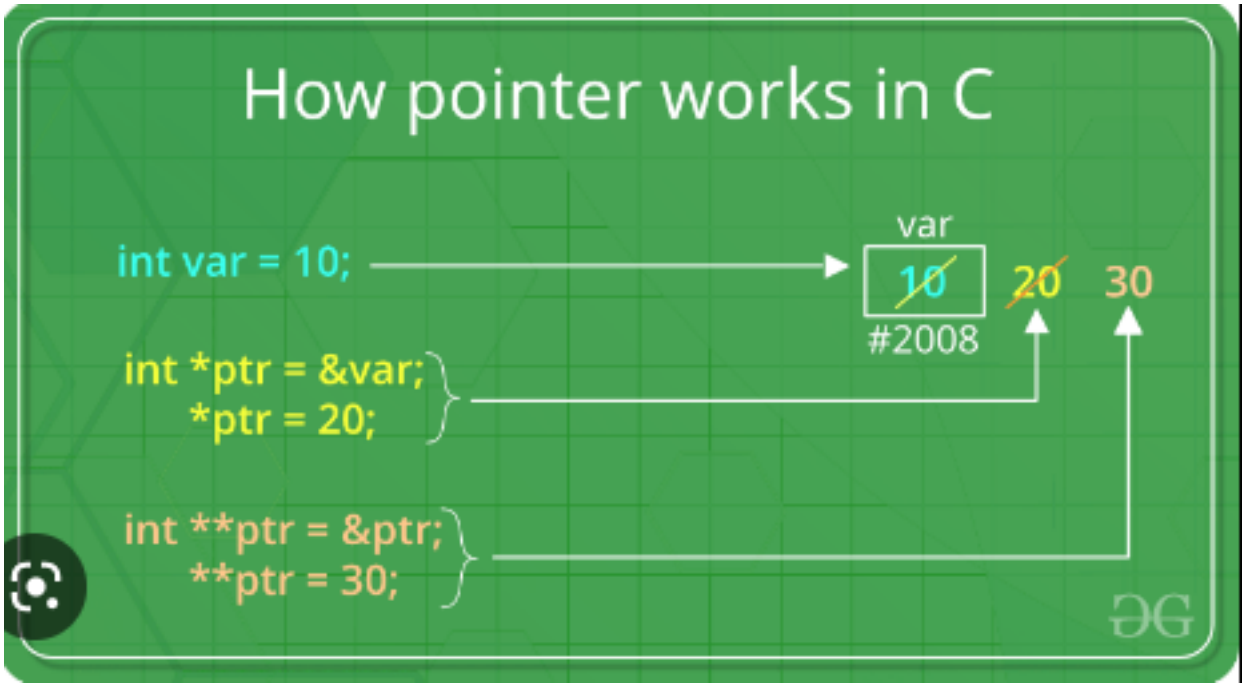
In the early days of programming, **most applications were written partially or entirely in assembly language**, but as programs became more complex, programmers switched to high-level languages. Assembly language is now used to **optimize certain sections of application programs for speed and to access computer hardware.** Overall, assembly language is more adaptable to low-level applications, while high-level languages are better suited for complex programs with a lot of structuring capabilities.

```
01 section .text
02    global_start
03    _start:
04        mov edx, len     ;message length
05        mov ecx, msg     ;message to write
06        mov ebx, 1       ;file descriptor(std_out)
07        mov eax, 4       ;system call number(sys_write)
08        int 0x80         ;call kernel
09
10        mov eax, 1       ;system call number(sys_exit)
11        int 0x80         ;call kernel
12
13 section .data
14
15 msg db 'hello, world!', 0xa ;our dear string
16 len equ $ -msg                ;length of our dear string
```

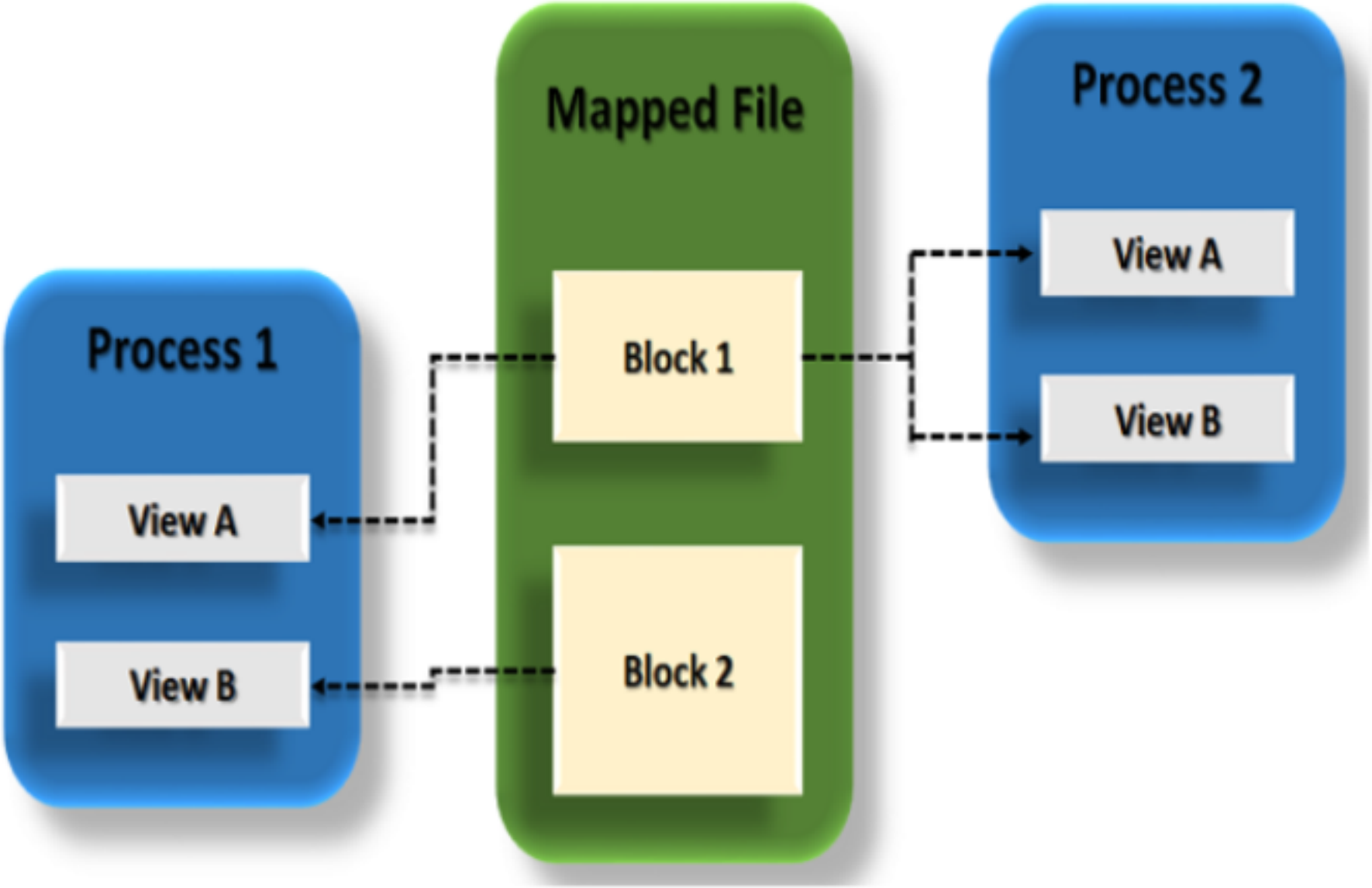
Type of Application	High-Level Languages	Assembly Language
Commercial or scientific application, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	The language may not provide for direct hardware access. Even if it does, awkward coding techniques may be required, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Commercial or scientific application written for multiple platforms (different operating systems).	Usually portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	May produce large executable files that exceed the memory capacity of the device.	Ideal, because the executable code is small and runs quickly.

The C and C++ languages have the unique quality of offering a compromise between high- level structure and low-level details. Direct hardware access is possible but completely non-portable.

In C/C++, direct hardware access is possible through the use of pointers, which allow programmers to directly manipulate memory addresses.

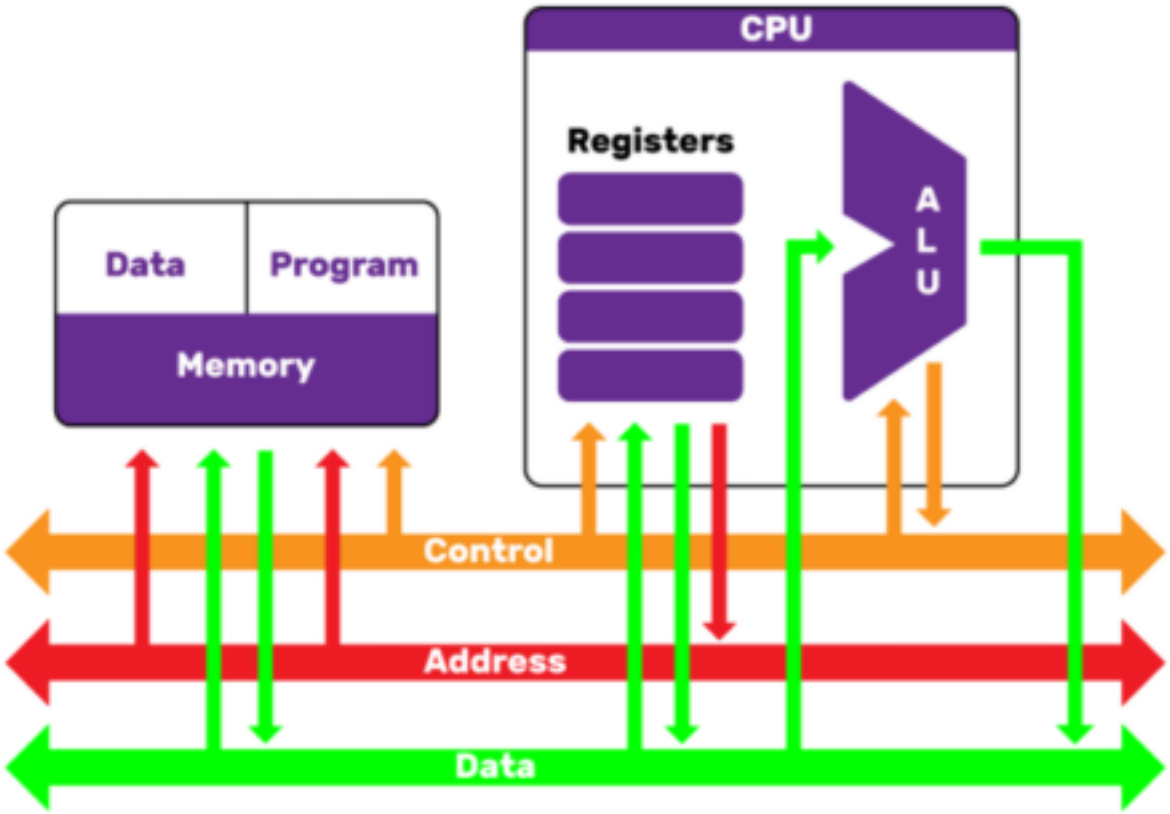


However, this direct hardware access is non-portable because it relies on knowledge of the specific hardware architecture and **memory mapping** of the target system.



Different computer systems have different **memory architectures and arrangements,** and accessing specific memory locations may result in undefined behavior or cause the program to crash on systems

with different memory layouts.

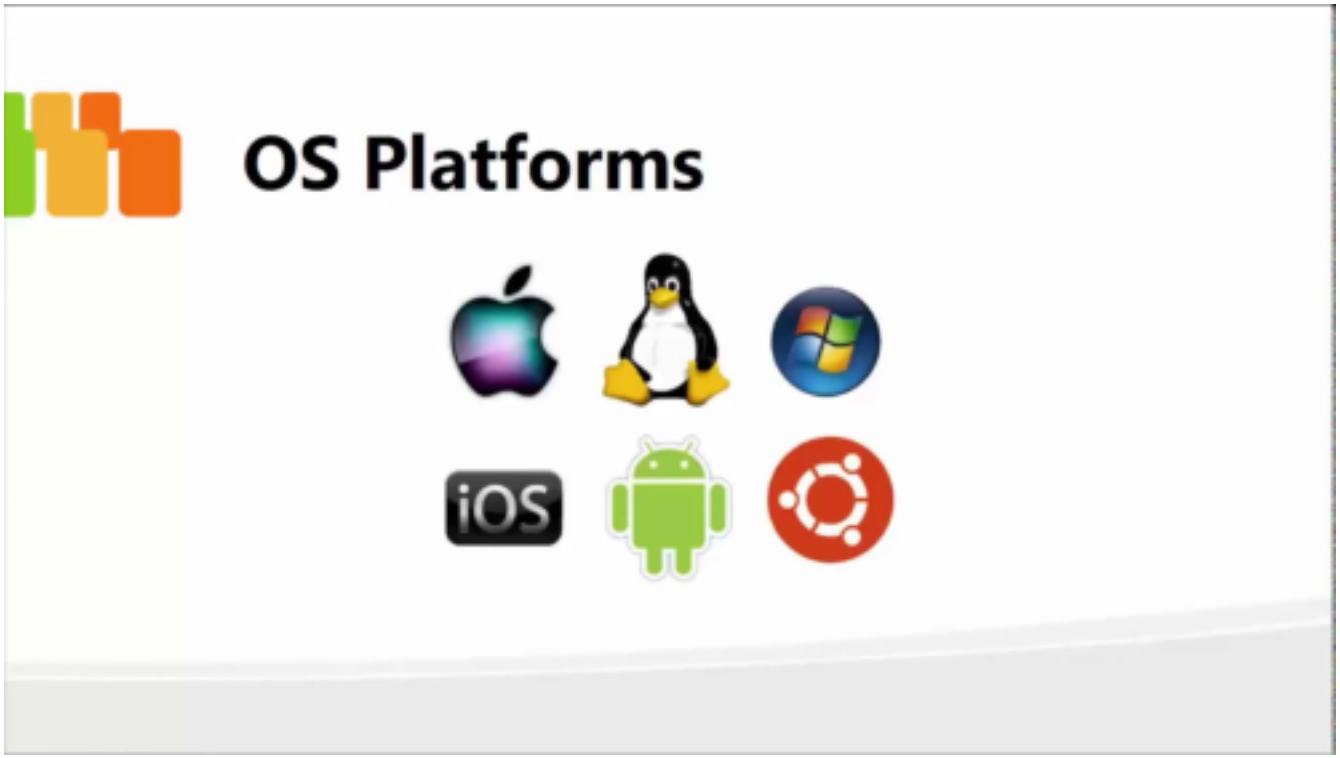


Additionally, direct hardware access in C/C++ may **violate the safety and security of the system**, and it may not be allowed in certain environments or operating systems due to security restrictions.

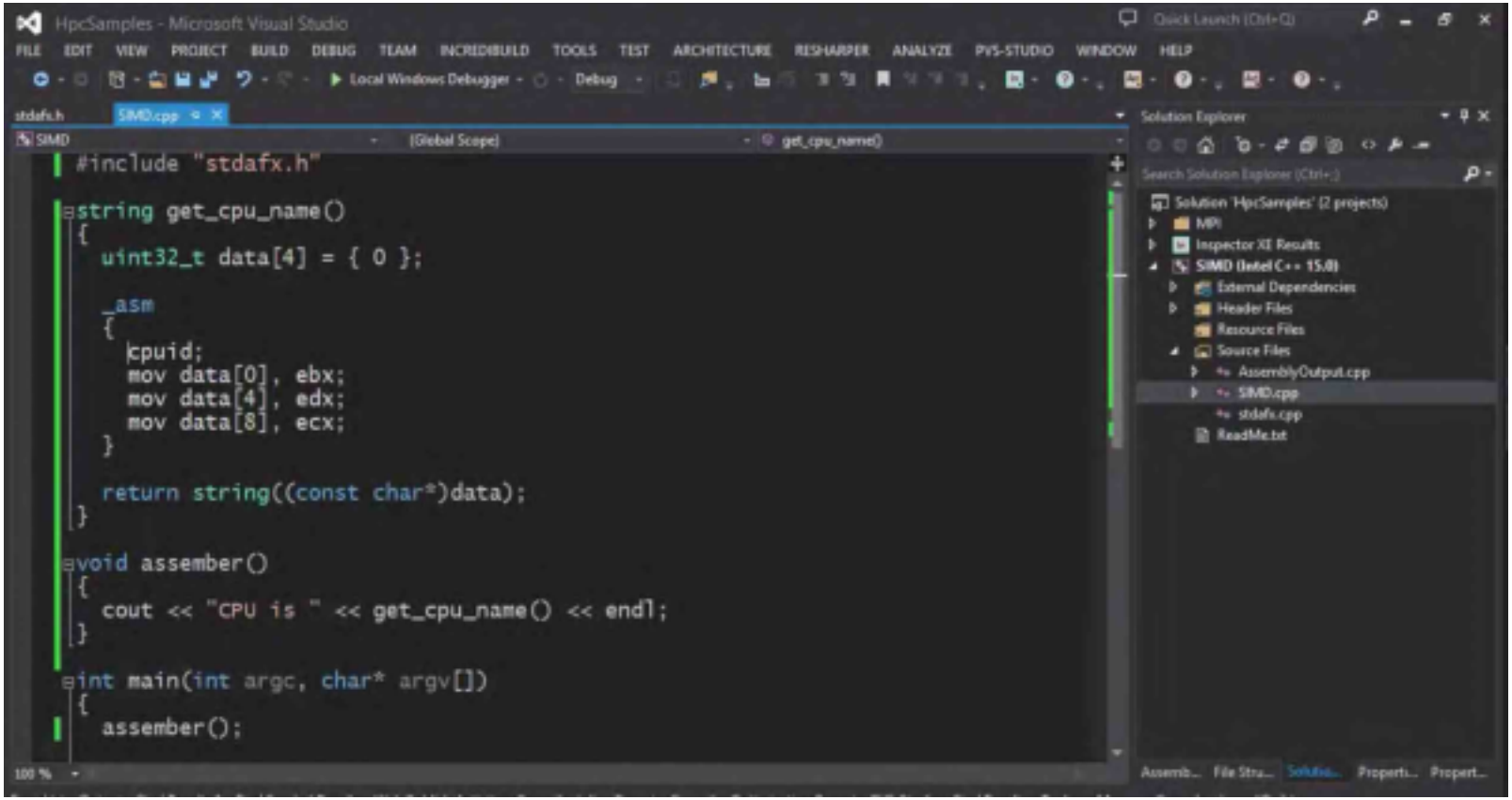
Therefore, to ensure portability and safety, C/C++ provides a set of standard libraries and system calls that **abstract away hardware details** and provide a standardized interface for accessing system resources.

This allows programs to be written in a more **portable and platform-independent way**, without having

to rely on specific hardware or memory layouts.



Most C and C++ compilers allow you to **embed assembly language** statements in their code, providing access to hardware details.



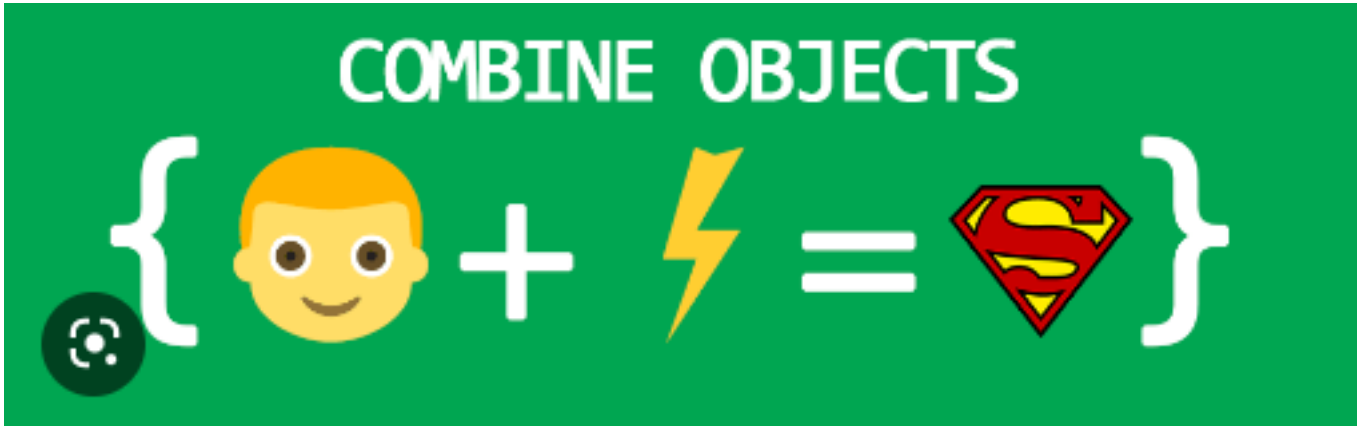
ASSEMBLY AND LINKERS WORKING TOGETHER

Assemblers and linkers work together to create executable programs from assembly language source code. Here's how they work together:

1. Assemblers **convert** assembly language source code into machine language object files that contain executable code and data. During this process, the assembler performs two main tasks: translating the mnemonics and operands of the assembly language instructions into their corresponding machine language opcodes and generating relocation information for the linker.

2. **Relocation information is generated** by the assembler to indicate to the linker where in memory the object code should be loaded and how it should be linked with other object files. This information includes the addresses of external symbols (i.e., symbols that are defined in other object files) that are used in the code, as well as the addresses of any jump or call instructions that reference other parts of the program.

3. The linker takes the object files generated by the assembler and **combines them into a single executable** program. During this process, the linker resolves any external symbol references by finding the addresses of the symbols in other object files and updating the relocation information in the code to reflect the correct addresses.



4. The linker also performs other tasks, such as **merging duplicate sections of code,** resolving references to library functions, and creating the program header and relocation tables.

5. Finally, the linker **produces the executable program**, which can be loaded and run by the operating system.

In summary, assemblers convert assembly language source code into object files, which contain machine language code and relocation information. Linkers then combine these object files into an executable program, resolving any external symbol references and performing other tasks as needed.

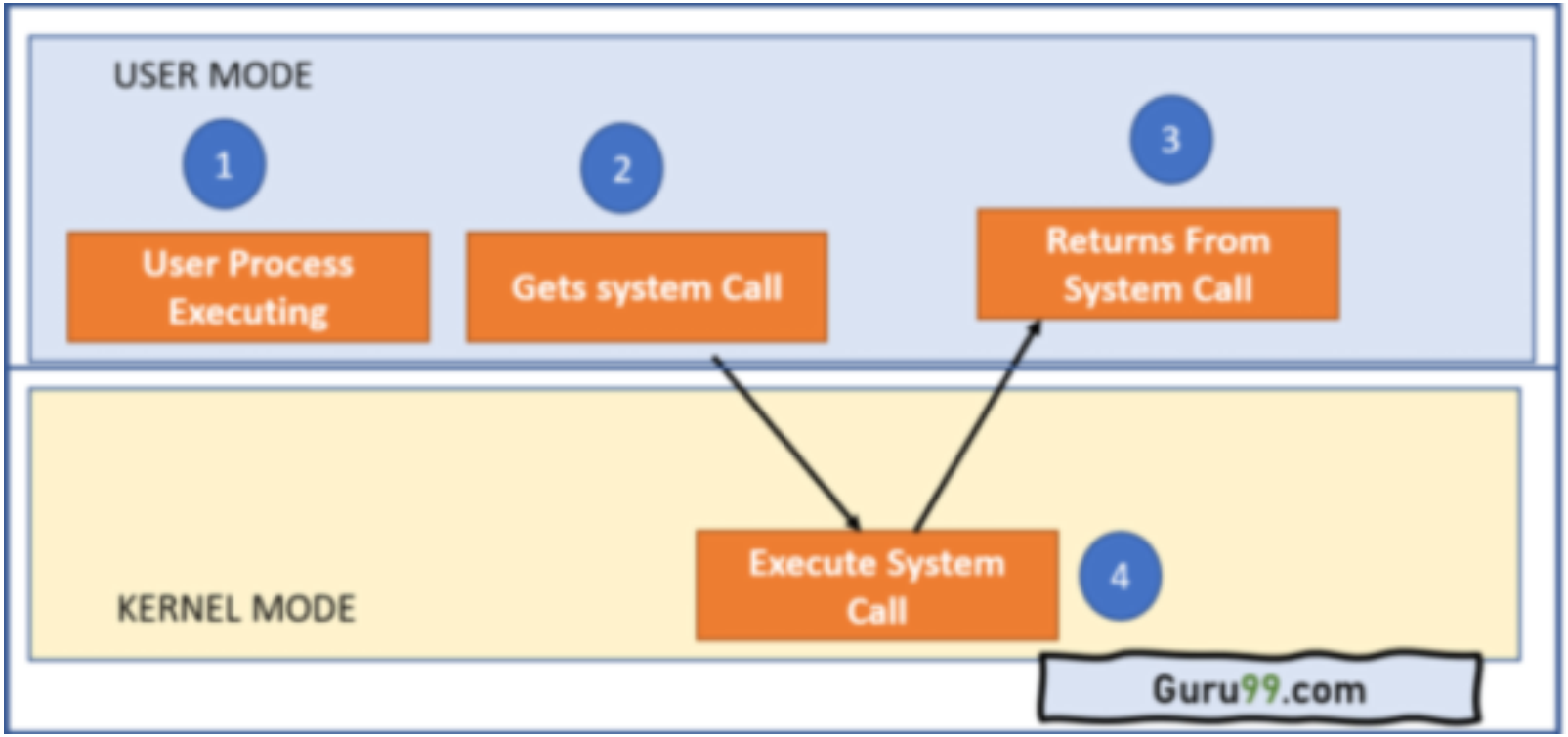
LEARNING ASSEMBLY HELPS IN UNDERSTANDING OPERATING SYSTEMS

Studying assembly language can enhance your understanding of operating systems in several ways:

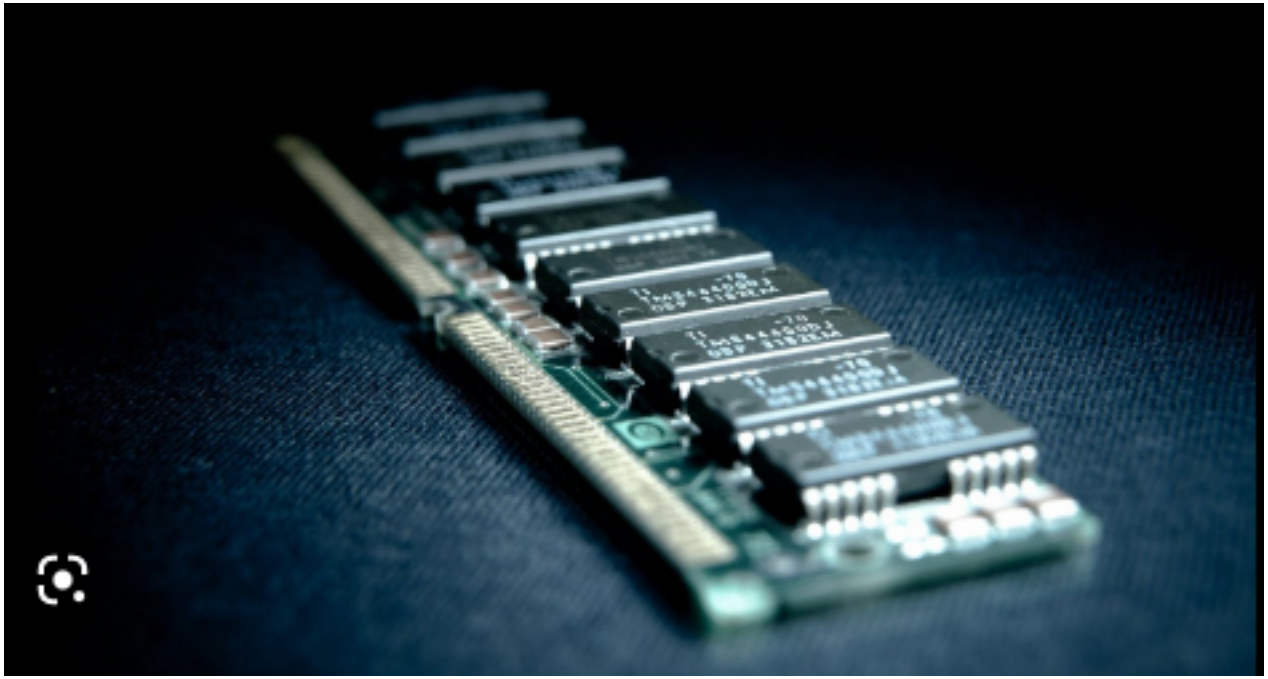
1. Understanding how hardware interacts with software: Assembly language provides a low-level understanding of how software interacts with hardware. Since operating systems are responsible for managing hardware resources, understanding how software interacts with hardware is critical to

understanding operating systems.

2. Understanding system calls: System calls are the primary interface between user-level programs and the operating system. By studying assembly language, you can gain a deeper understanding of how system calls work and how the operating system responds to them.



3. Understanding memory management: Memory management is a critical aspect of operating systems, and assembly language provides a deep understanding of how memory is allocated and managed at the lowest level.



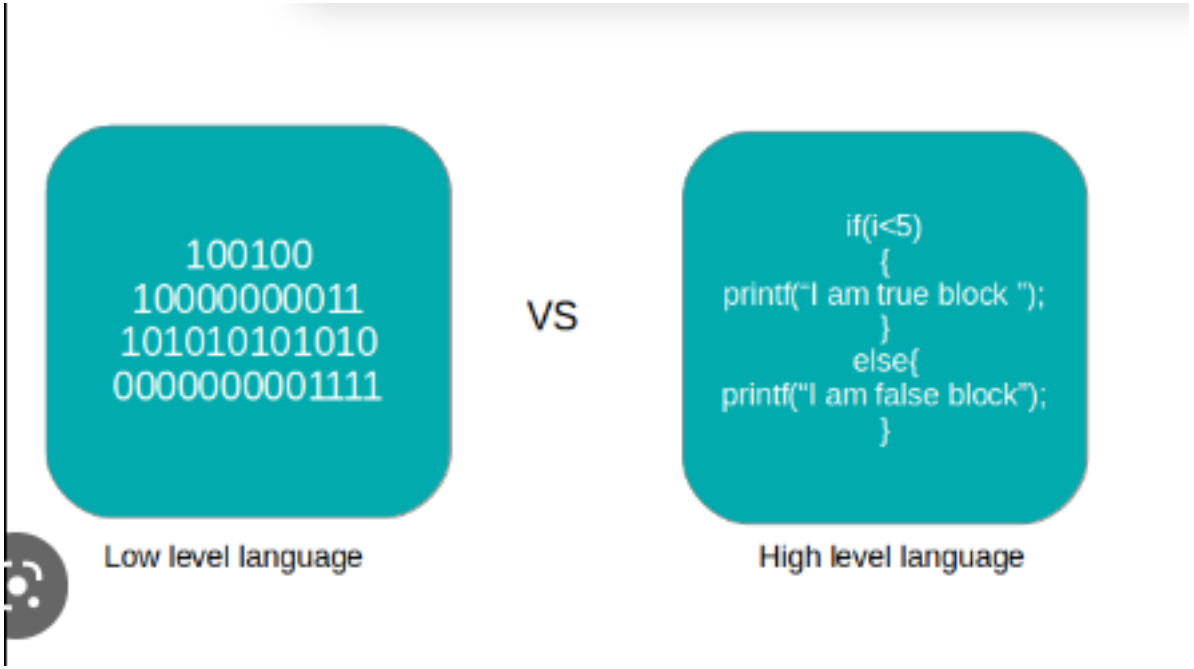
4. Understanding performance optimization: Assembly language is often used to optimize performance-critical parts of an operating system. By studying assembly language, you can gain insight into how to optimize system performance.

5. Understanding security: Understanding assembly language can also help in understanding system vulnerabilities, especially in cases where software exploits hardware or kernel bugs.

In summary, studying assembly language can provide a deeper understanding of how an operating system works at a low level, which can help in various aspects of operating system development, including system calls, memory management, performance optimization, and security.

ONE-TO-MANY RELATIONSHIPS IN HIGH LEVEL VS LOW LEVEL LANGUAGES

In the context of comparing a high-level language to machine language, a one-to-many relationship refers to the fact that a single instruction in a high-level language may be translated into multiple machine language instructions.



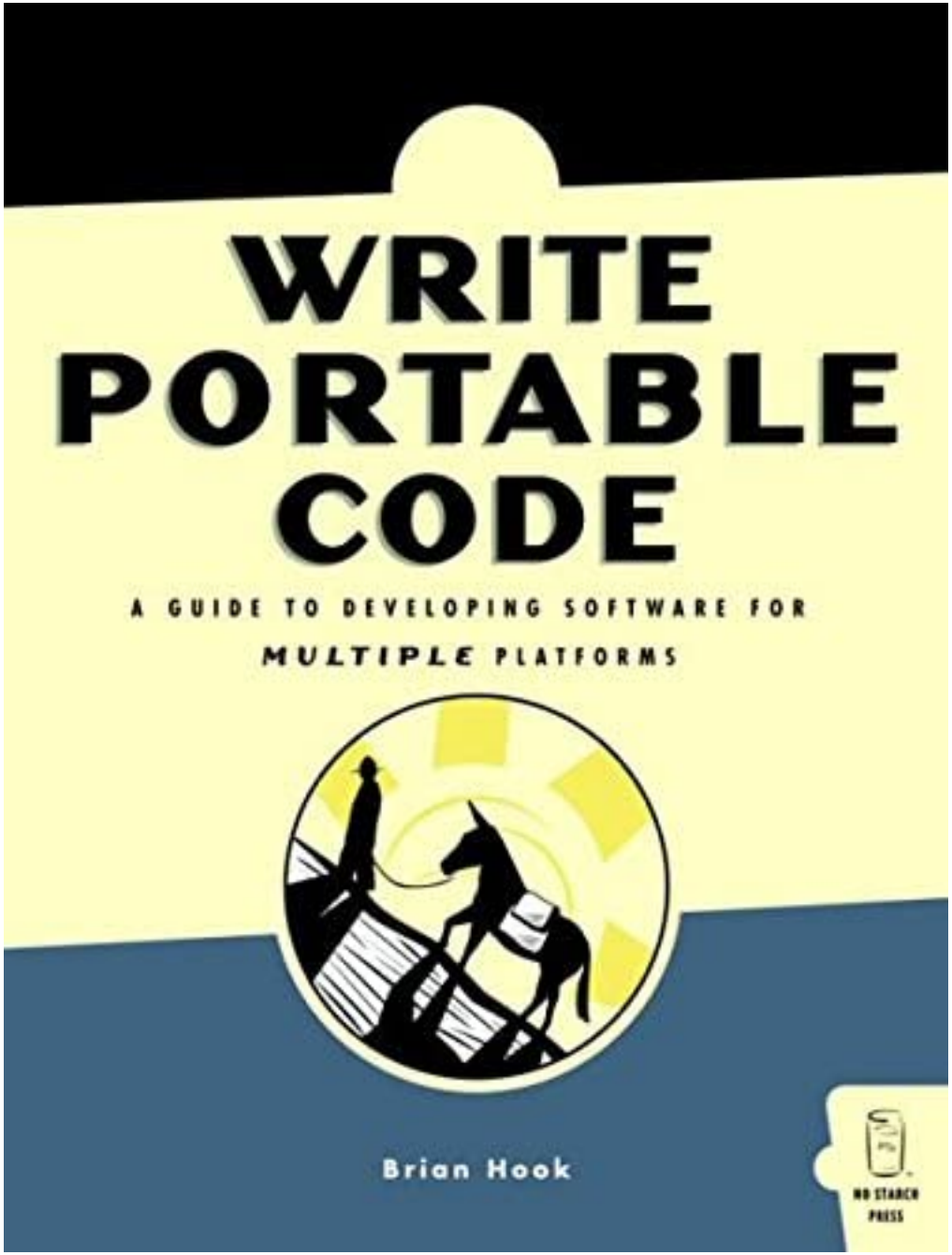
High-level languages, such as C++, Java, or Python, are designed to provide a more abstract and human-readable way of writing code, whereas machine language is the binary code that is executed directly by the computer's processor. When a program written in a high-level language is compiled or interpreted, it is translated into machine language instructions that the computer can execute.

In this translation process, a single instruction in a high-level language may be broken down into multiple machine language instructions that perform the same task. For example, a high-level language statement such as "for (i = 0; i < 10; i++)" may be translated into multiple machine language instructions, including load, compare, increment, and jump instructions, to achieve the same result.

Therefore, a one-to-many relationship in this context refers to the fact that a single high-level language instruction may be translated into multiple machine language instructions, which can make the resulting machine code more complex and difficult to read and understand.

PORTABILITY

Portability is the ability of a program or application to run on different hardware or software platforms without requiring significant modifications. In the context of programming languages, portability refers to the ability of a programming language to write code that can be compiled or interpreted on different platforms with minimal changes.



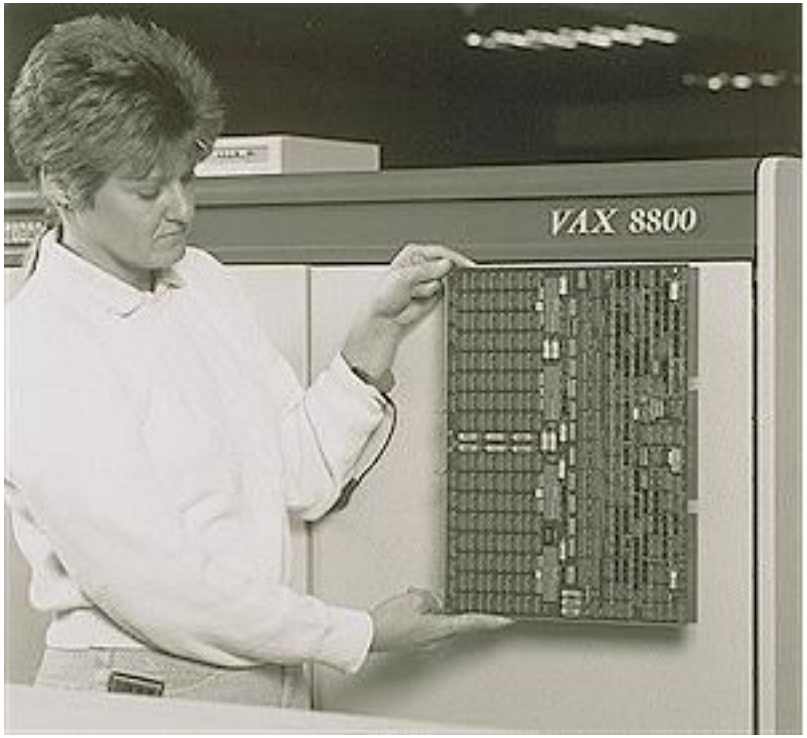
A portable programming language is one that can be used to write code that can run on different

operating systems, hardware architectures, or software environments without requiring significant changes to the code. Portable programming languages achieve this by providing a standardized syntax, data types, and libraries that are common across different platforms. This allows developers to write code that can be compiled or interpreted on different platforms without requiring modifications to the code.

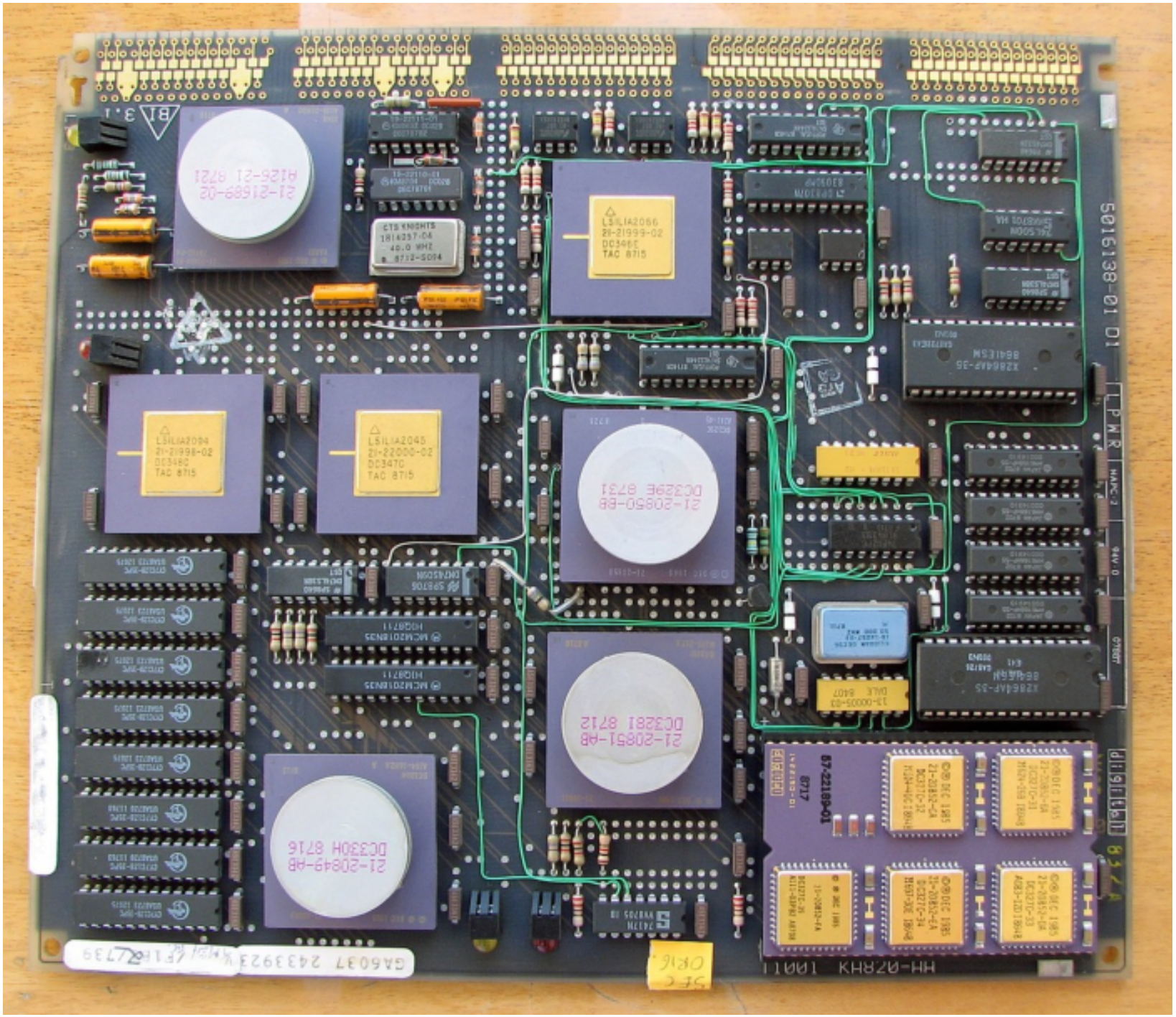
Some examples of portable programming languages include Java, Python, Ruby, and C++. These languages have a large user base and are supported by different hardware and software vendors, which makes them available on different platforms. They also provide a standard set of libraries that can be used across different platforms, which simplifies the development process and reduces the likelihood of errors due to platform-specific dependencies.

In summary, portability is an important consideration when selecting a programming language for a project, as it can reduce development time and cost by allowing code to be used across different platforms with minimal changes. Portable programming languages provide a standard set of syntax, data types, and libraries that can be used across different platforms, which makes them a popular choice for developing cross-platform applications.

Is the assembly language for x86 processors the same as those for computer systems such as the Vax or Motorola 68x00?



No, the assembly language for x86 processors is not the same as those for computer systems such as the VAX or Motorola 68x00. While assembly language is a low-level programming language that is specific to the processor architecture, the instructions and syntax used by different processors can vary significantly.


5016138-01 01
CTS KNIGHTS
18146257-04
40.0 MHz
L5ILIA2086
21-21999-02
DC346E
TAC 8715
L5ILIA2094
21-21998-02
DC348C
TAC 8715
L5ILIA2045
21-22000-02
DC347C
TAC 8715
21-20850-BB
DC329E 8731
21-20851-AB
DC328I 8712
21-20849-AB
DC330H 8716
21-21689-02
A126-21 9721
87-22198-01
8717
HH-020KH 10011
LPWR

The x86 processor family is used by most personal computers and laptops, while VAX processors were used primarily in large mainframe systems, and the Motorola 68x00 processors were used in many embedded systems and personal computers in the 1980s and 1990s. Each of these processor families has its own unique instruction set and assembly language syntax.

Therefore, if you want to write assembly language code for a specific processor, you need to learn the instruction set and syntax for that processor. While there may be similarities between different assembly languages, it is important to be familiar with the specific instruction set and syntax of the processor you are working with in order to write efficient and effective code.

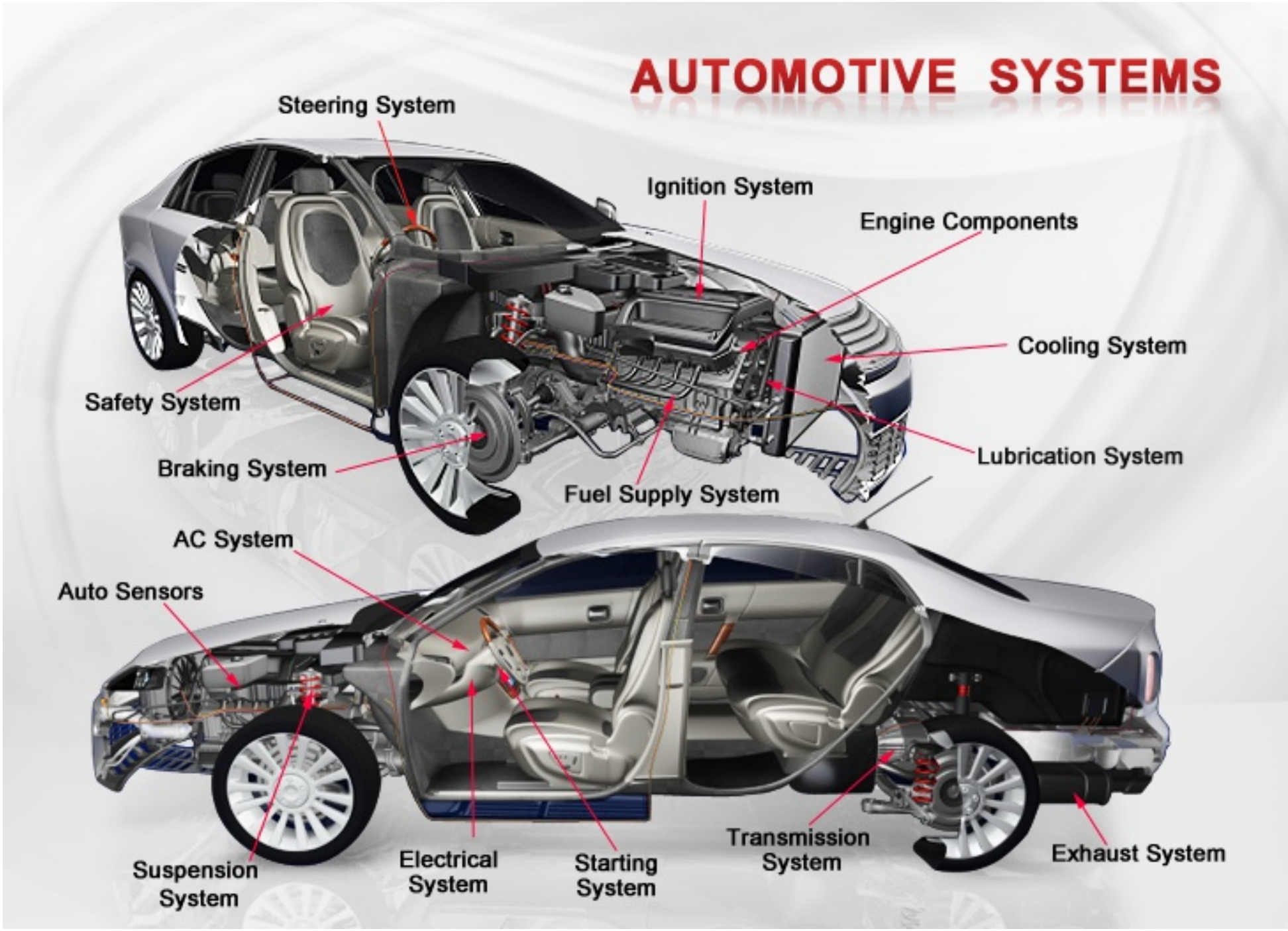
EMBEDDED SYSTEMS APPLICATION

There are many examples of embedded systems applications. Here are a few:

Smart home devices: Many smart home devices, such as thermostats, security systems, and door locks, are powered by embedded systems. These systems enable the devices to communicate with each other and with the internet, and to perform complex functions in real time.

Medical devices: Many medical devices, such as blood glucose monitors, pacemakers, and infusion pumps, are powered by embedded systems. These systems enable the devices to monitor and adjust a patient's vital signs and deliver precise amounts of medication.

Automotive systems: Modern cars contain numerous embedded systems, including those that control the engine, brakes, and entertainment system. These systems enable the car to operate efficiently and safely, and to provide a comfortable and enjoyable driving experience.


AUTOMOTIVE SYSTEMS
Steering System
Ignition System
Engine Components
Cooling System
Safety System
Braking System
Fuel Supply System
Lubrication System
AC System
Auto Sensors
Suspension System
Electrical System
Starting System
Transmission System
Exhaust System

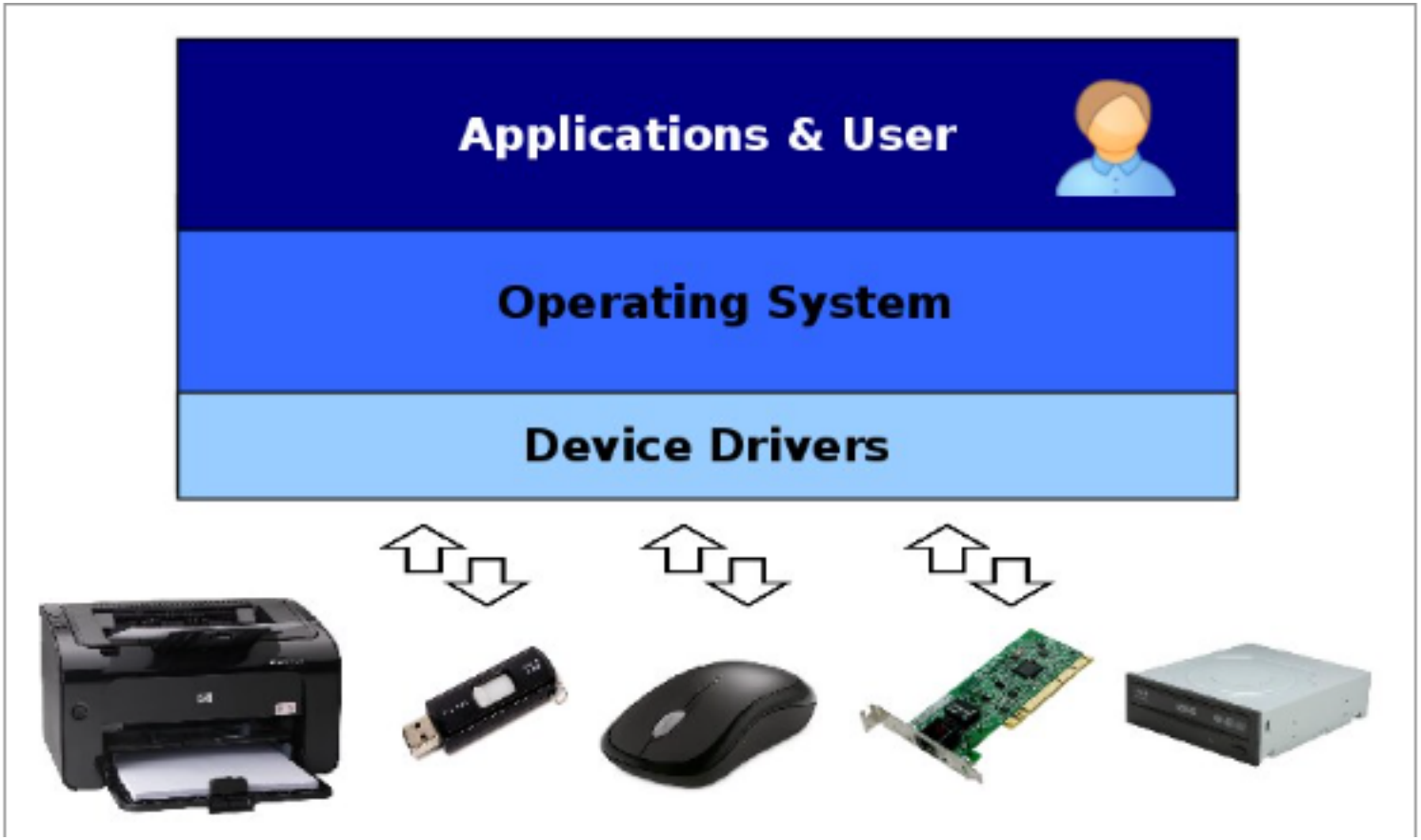
Industrial control systems: Many manufacturing and industrial processes rely on embedded systems to control and automate various tasks, such as monitoring temperature and pressure, controlling motors and actuators, and collecting and analyzing data.

Consumer electronics: Many consumer electronics products, such as digital cameras, smartphones, and gaming consoles, are powered by embedded systems. These systems enable the devices to perform complex functions and provide a seamless user experience.

DEVICE DRIVERS

A **device driver** is a software program that allows an operating system to communicate with a specific hardware device, such as a printer, scanner, or keyboard.

The device driver acts as a translator between the operating system and the hardware device, allowing the operating system to send commands and receive data from the device.



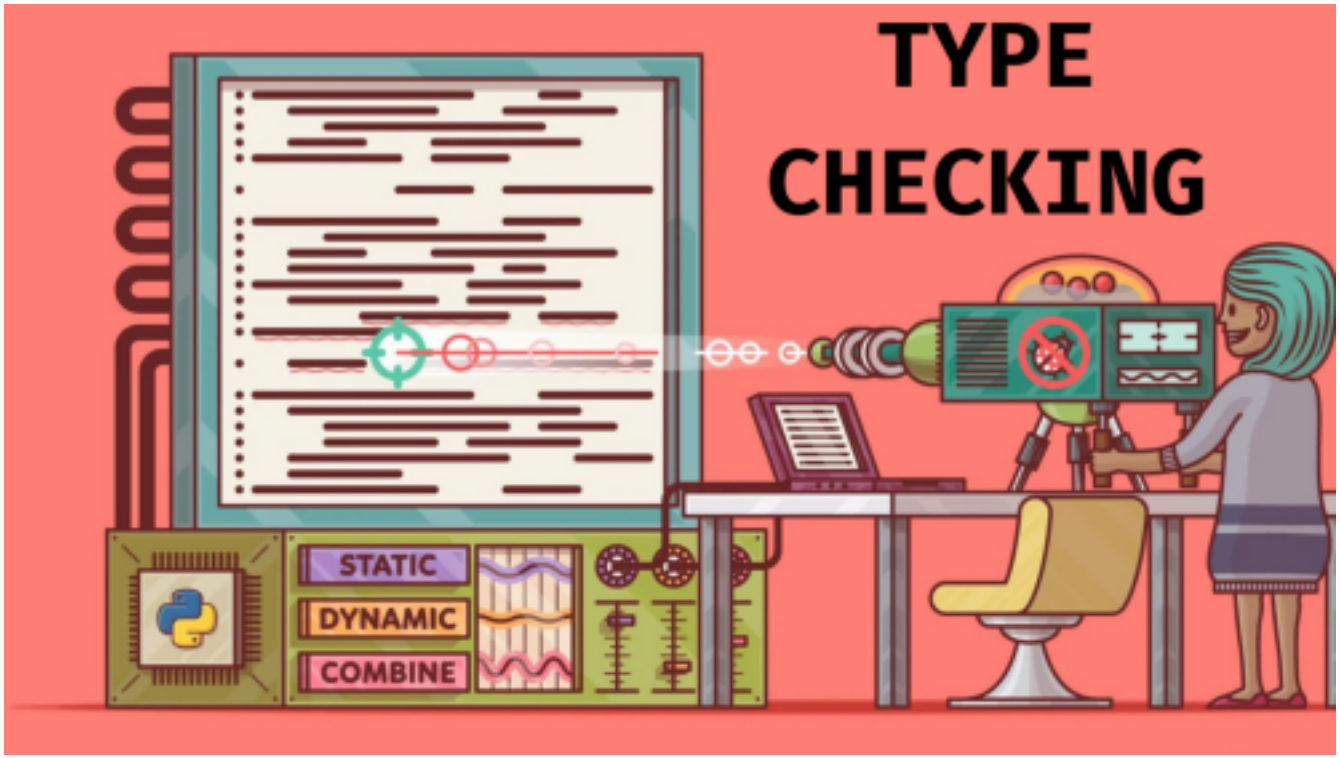
Device drivers are typically written by hardware manufacturers and are specific to the hardware they produce.

The device driver must be compatible with the operating system and version that it is intended to work with.

Without a device driver, the operating system would not be able to communicate with the hardware device, and the device would not function correctly.

Do you suppose type checking on pointer variables is stronger (stricter) in assembly language, or in

C/C++



Type checking on pointer variables is generally stronger (stricter) in C and C++ compared to assembly language.

In C and C++, pointer variables are strongly typed, which means that the compiler enforces strict rules about what type of data can be stored in a pointer variable.

For example, if a pointer variable is declared to point to an integer, then the compiler will prevent the program from assigning a pointer to a different type of data to that variable.

This helps to prevent programming errors and improve program reliability.

In assembly language, pointer variables are typically represented as memory addresses and are not strongly typed.

The assembly language programmer has to manually manage the memory locations where data is stored, and it is up to them to ensure that the correct data types are stored in the correct memory locations.

This can make it easier to introduce programming errors related to type mismatches, such as accessing memory locations using the wrong data type.

Overall, while assembly language provides greater control and flexibility over memory management, it does not have the same level of type checking and safety features that are present in C and C++.

TWO APPLICATIONS BETTER SUITED WITH ASSEMBLY

Assembly language is a low-level programming language that provides direct control over the hardware of a computer. It is generally more difficult to work with than high-level languages such as C and Java, but it can be advantageous in certain situations. Here are two types of applications that might be better suited to assembly language:

1. Operating system components: Certain parts of an operating system, such as device drivers or system utilities, require low-level access to computer hardware. Assembly language can provide the level of control and performance needed for these components, as they need to be efficient and interact directly with the underlying hardware.

2. Real-time systems: Real-time systems, such as those used in robotics or industrial control applications, require very fast and precise responses to inputs. Assembly language can provide the level of control and performance needed for these systems, as they need to respond quickly and deterministically to changes in the environment.

Real-Time Systems

- Controllers in planes, cars, plants, etc. are expected to finish their tasks within reliable time bounds
- It is essential that an upper bound on the execution times of all tasks is known: commonly called the worst-case execution time (WCET), computed at the code level
- WCET of tasks prerequisite for scheduling analysis at system level (e.g. SymTA/S from Symtavision)

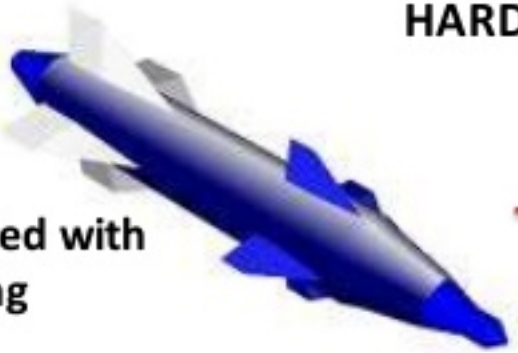
In both of these cases, the benefits of using assembly language come from the level of control and performance it offers, as well as the ability to optimize code for specific hardware platforms. However, it's worth noting that assembly language is generally less productive than high-level languages, and can be more difficult to debug and maintain.

Real Time Embedded Systems

A system in which work has to be done in a specific time period.

HARD REAL TIME

Missile embedded with a tracking system



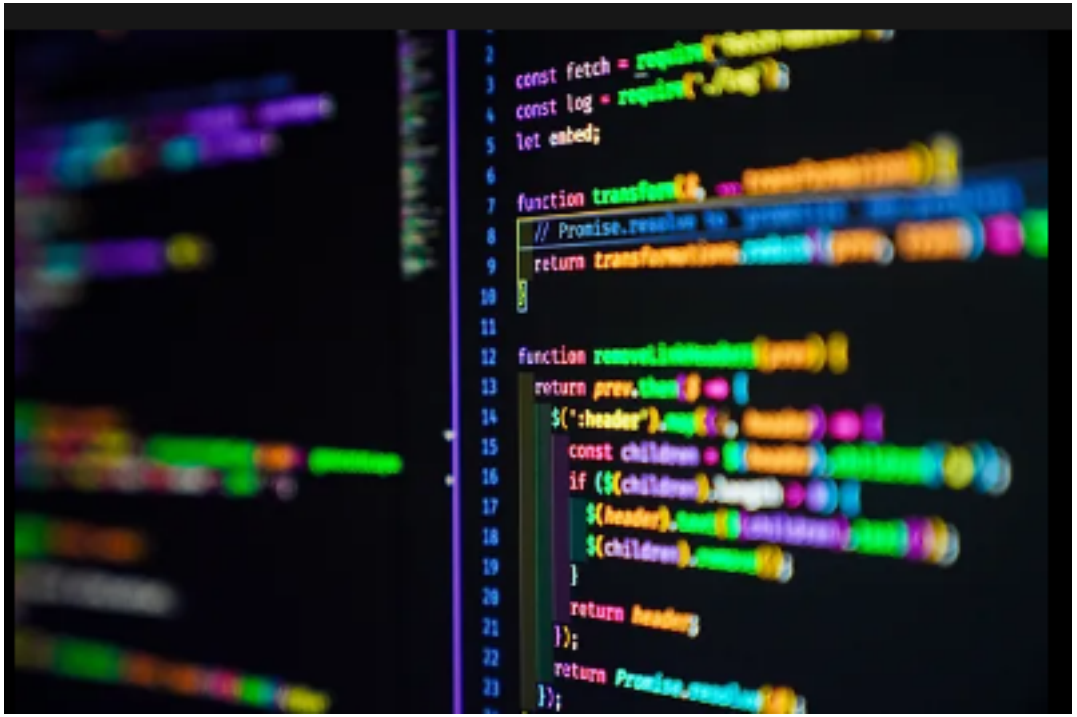
Aircraft

SOFT REAL TIME

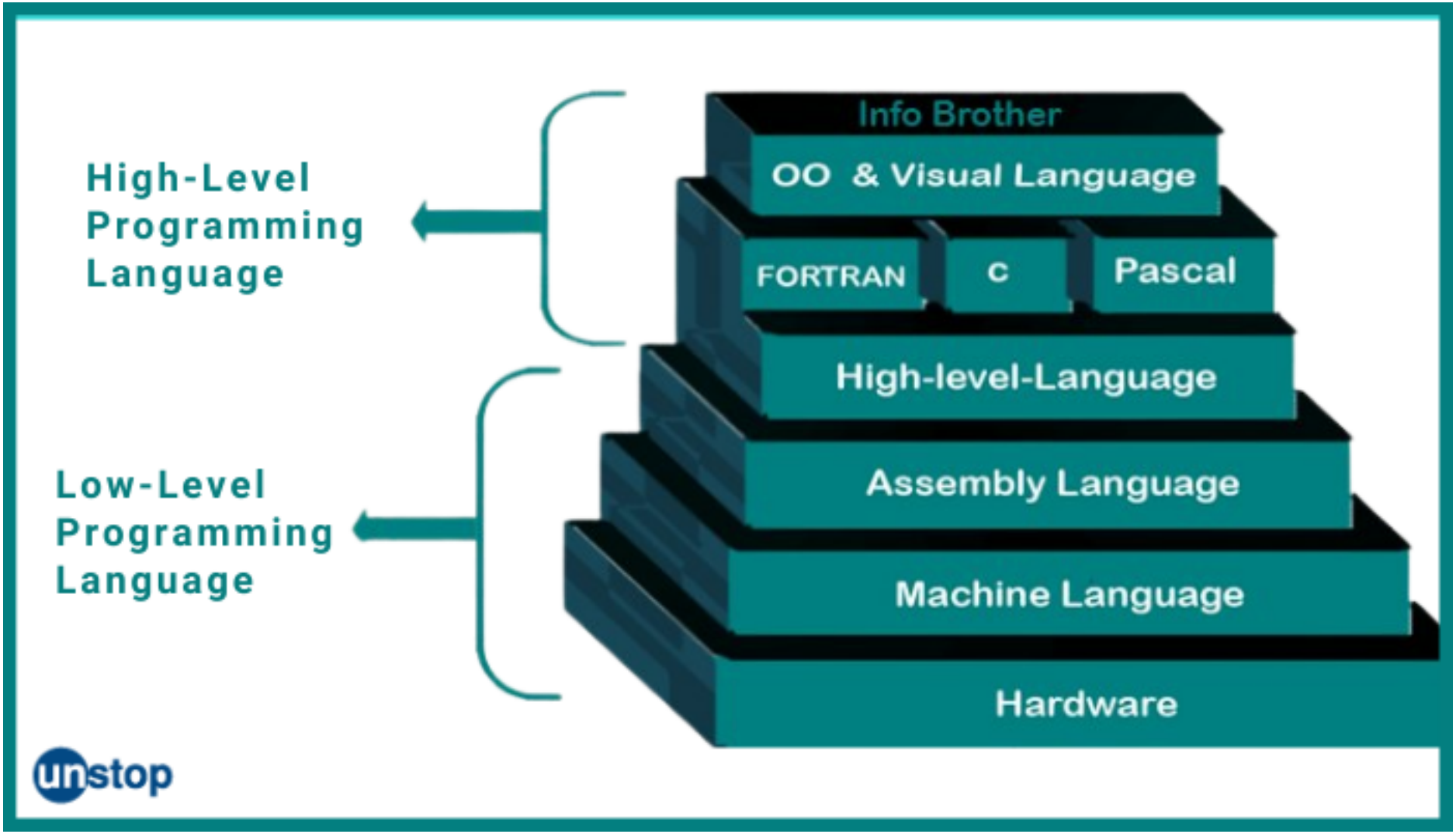
DVD PLAYER

WHY HLL IS NOT SUITED FOR PROGRAM ACCESSING PRINTERS

A high-level language is designed to provide an abstraction layer between the programmer and the hardware, which makes it easier to write portable and maintainable code. However, this abstraction also comes at a cost, as it can **limit the level of control and access** that the programmer has over the hardware.



Directly accessing a **printer port** is an example of a task that **requires low-level access to the hardware**, which is not well-suited to high-level languages. High-level languages such as Java or Python provide a high level of abstraction, which makes it difficult to directly manipulate hardware devices such as printer ports. In contrast, assembly language or a low-level language such as C provide a much lower level of abstraction, allowing for direct access to hardware ports and registers.



Directly accessing a printer port also involves working with specific I/O addresses, interrupts, and other low-level hardware details. These details are generally not exposed in high-level languages, as the language abstracts away these low-level details in favor of providing higher-level abstractions.

In summary, a high-level language is not ideal for writing a program that directly accesses a printer port because it **lacks the low-level access and control required to directly manipulate**

hardware devices. Instead, a low-level language such as assembly or C would be better suited for this task, as it provides the necessary low-level access and control required to work with hardware devices.

LARGE APPLICATIONS DEVELOPMENT AVOIDS ASSEMBLY

Assembly language is a low-level programming language that provides direct control over computer hardware. While it has certain advantages over higher-level languages, it is not typically used when writing large application programs for the following reasons:

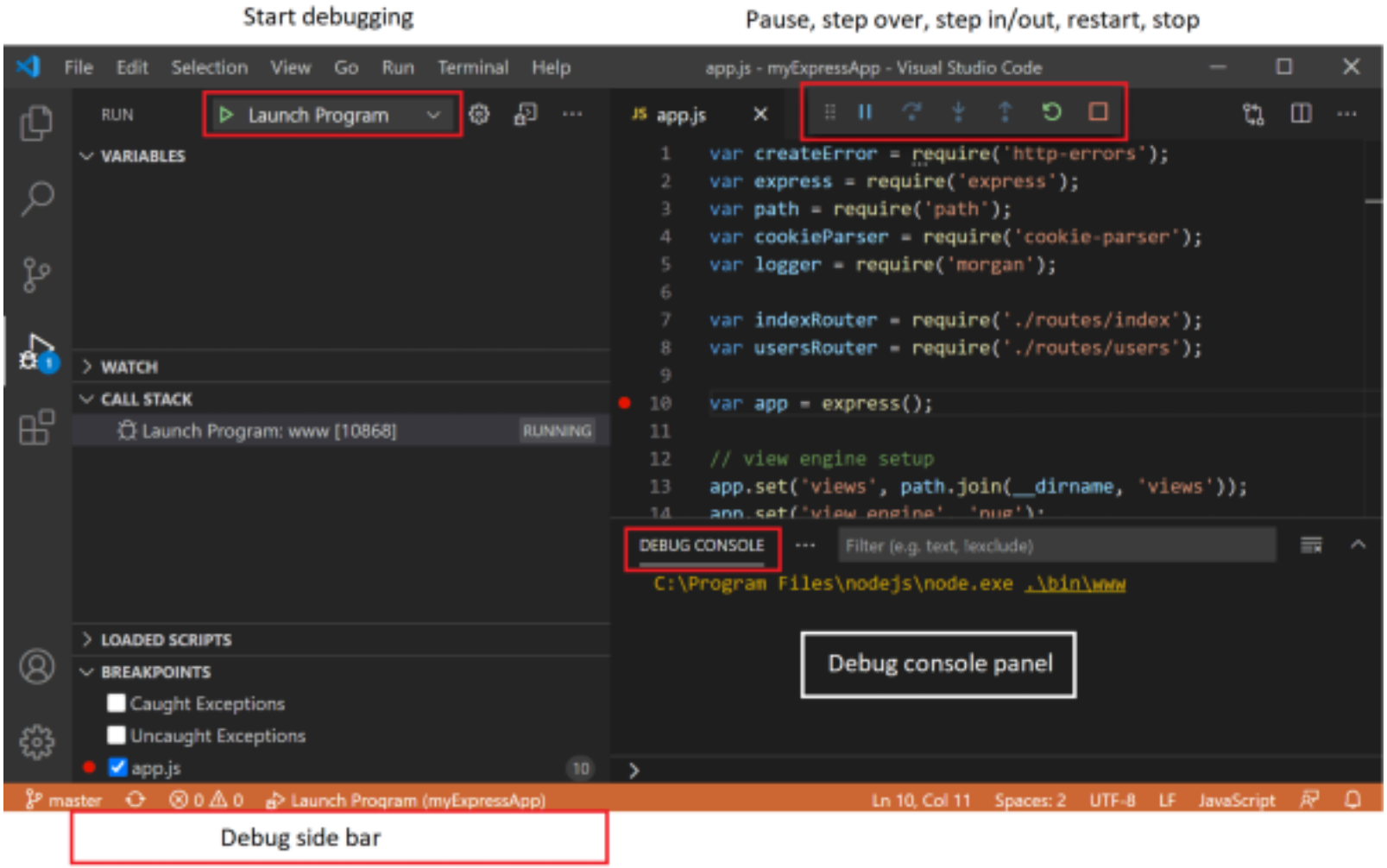
- **Complexity:** Assembly language programming is much more complex and time-consuming than programming in higher-level languages. Writing large applications in assembly language requires a great deal of skill, effort, and attention to detail.

• **Maintainability:** Assembly language code is difficult to read, understand, and maintain. This is because it is written in terms of low-level hardware operations, rather than high-level abstractions that are easier to reason about.

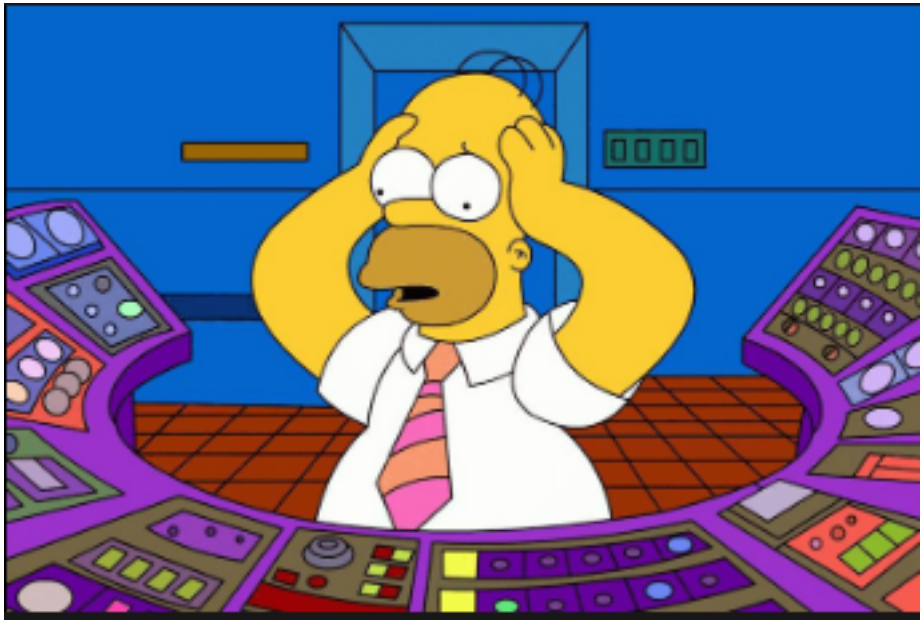
• **Portability:** Assembly language code is not portable across different hardware platforms. This means that code written for one type of hardware may not work on another type of hardware, which makes it more difficult to develop and maintain large applications.

• **Productivity:** Assembly language programming is not as productive as programming in higher-level languages. This is because it requires more code to accomplish the same task, and because it lacks the high-level abstractions and features that make programming in higher-level languages more efficient.

- **Debugging:** Debugging assembly language code can be difficult and time-consuming, especially in large applications. This is because there are no high-level abstractions or debugging tools available to help with the process.

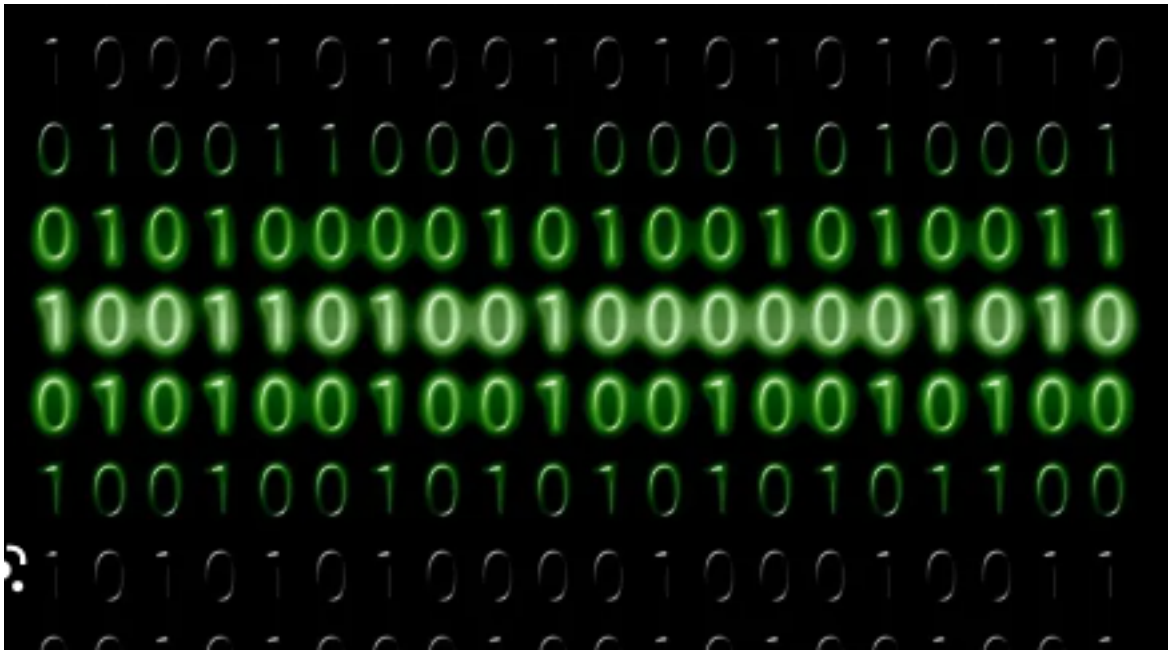


• **Human Error:** Assembly language is more susceptible to human error, as the programmer is responsible for managing memory and performing other low-level operations that are typically handled automatically in higher-level languages. This can lead to bugs and other issues that can be difficult to identify and fix.



VIRTUAL MACHINE CONCEPT

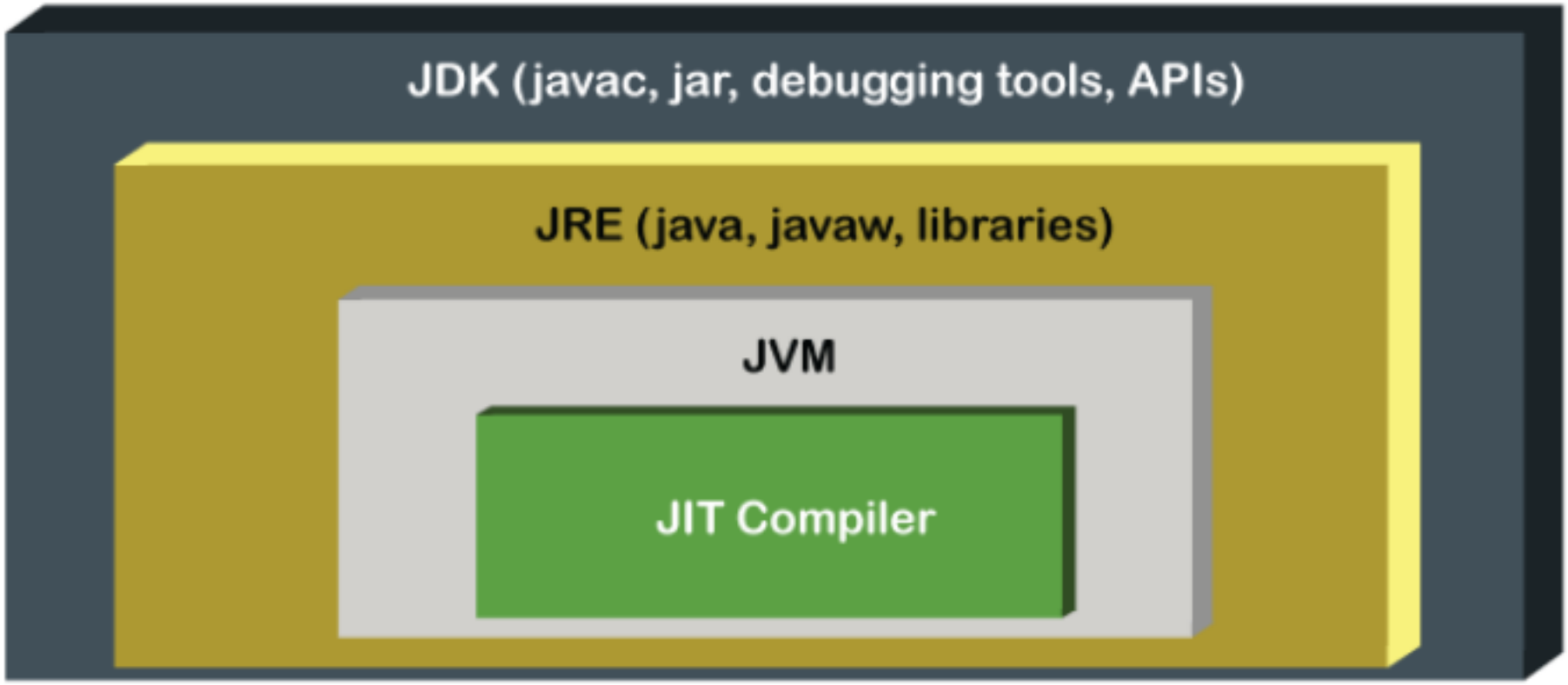
- The **virtual machine concept** explains how a computer's hardware and software are related.
- The most basic function of a computer is executing programs in its **native machine language (L0).**



- L0 is difficult to use, so a **new language, L1,** could be constructed that is easier to use.

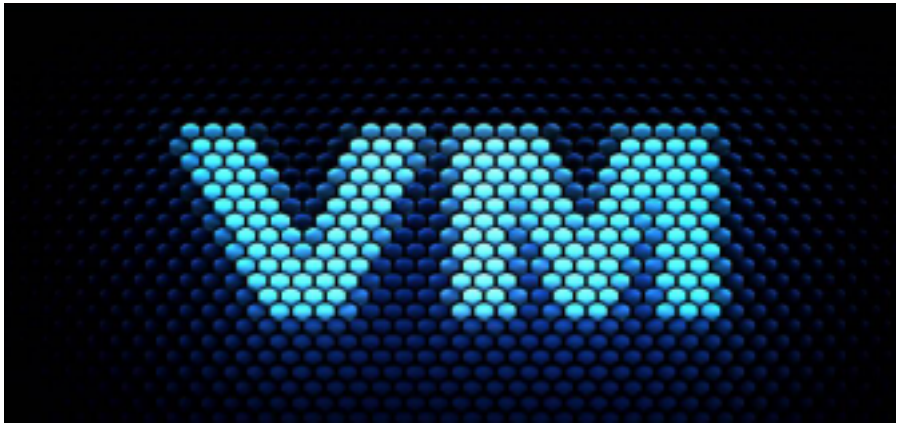
```
 *  0 - deliver
 *  1 - block
 */
static int icmp_filter(const struct sock *sk, const struct sk_b
{
    struct icmphdr _hdr;
    const struct icmphdr *hdr;

    hdr = skb_header_pointer(skb, skb_transport_offset(skb),
                sizeof(_hdr), &_hdr);
    if (!hdr)
        return 1;

    if (hdr->type < 32) {
        __u32 data = raw_sk(sk)->filter.data;

        return ((1U << hdr->type) & data) != 0;
    }

    /* Do not block unknown ICMP types */
    return 0;
}
```

• **L1 can be executed by a virtual machine (VM1)** that emulates the functions of a physical or virtual computer.

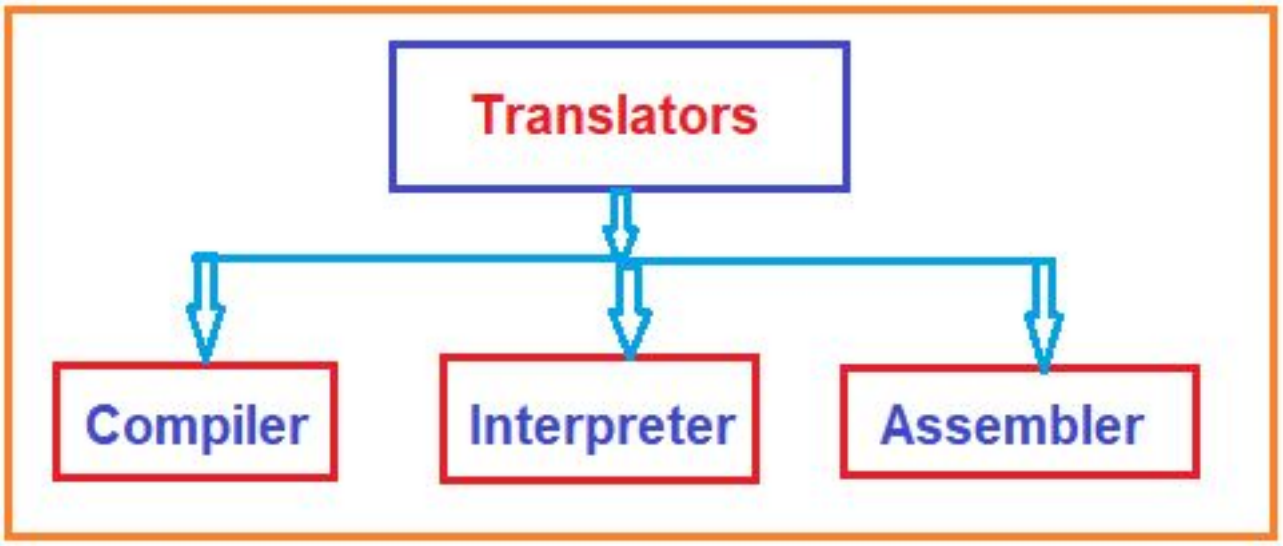


- VM1 can be constructed of either hardware or software, and programs can be written for it.

- If VM1 is practical to implement as an actual computer, programs can be executed directly on the hardware.



• Otherwise, programs written in VM1 can be interpreted/translated and executed on another virtual machine (VM0) that can execute commands written in L0.

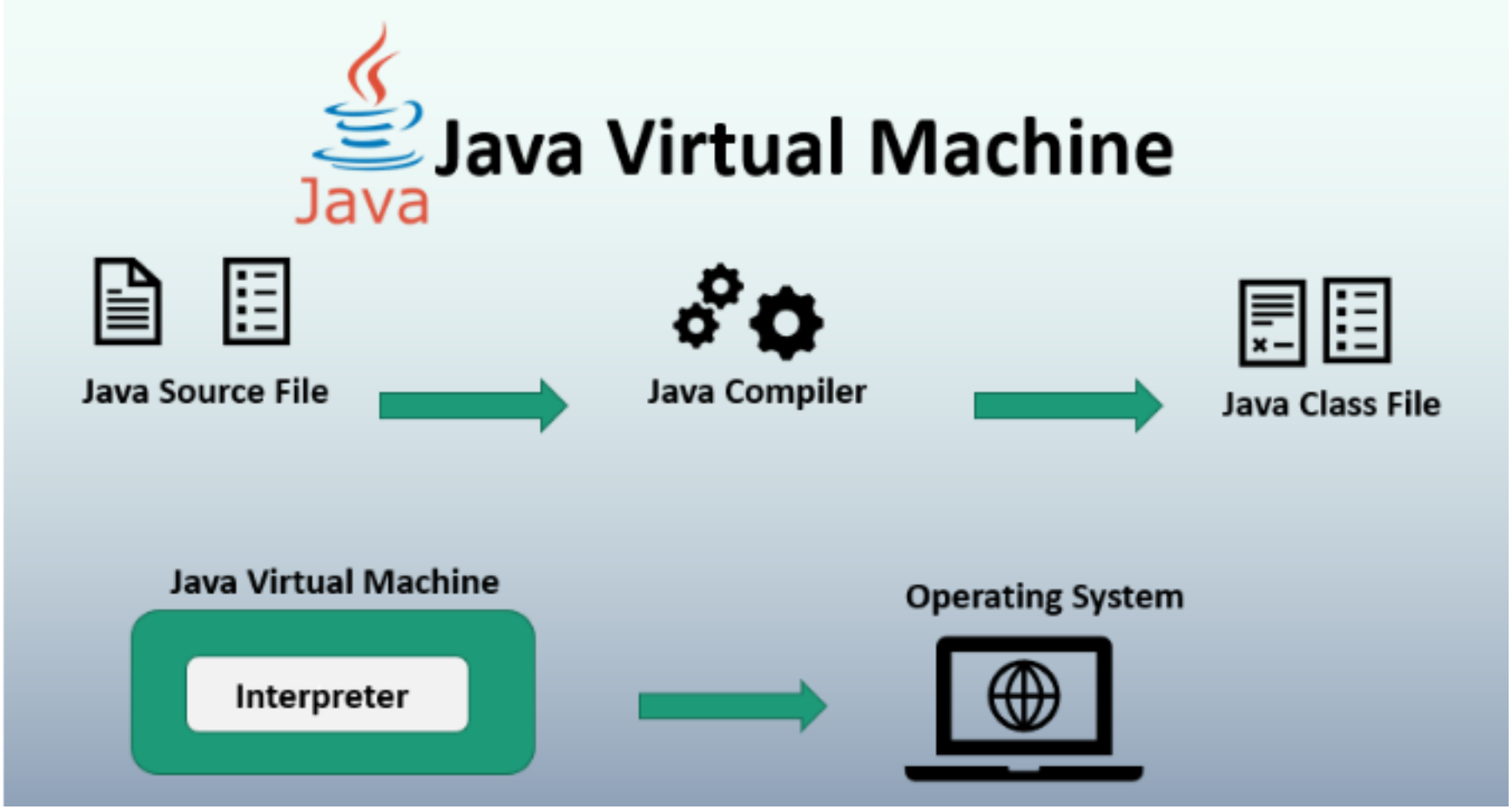


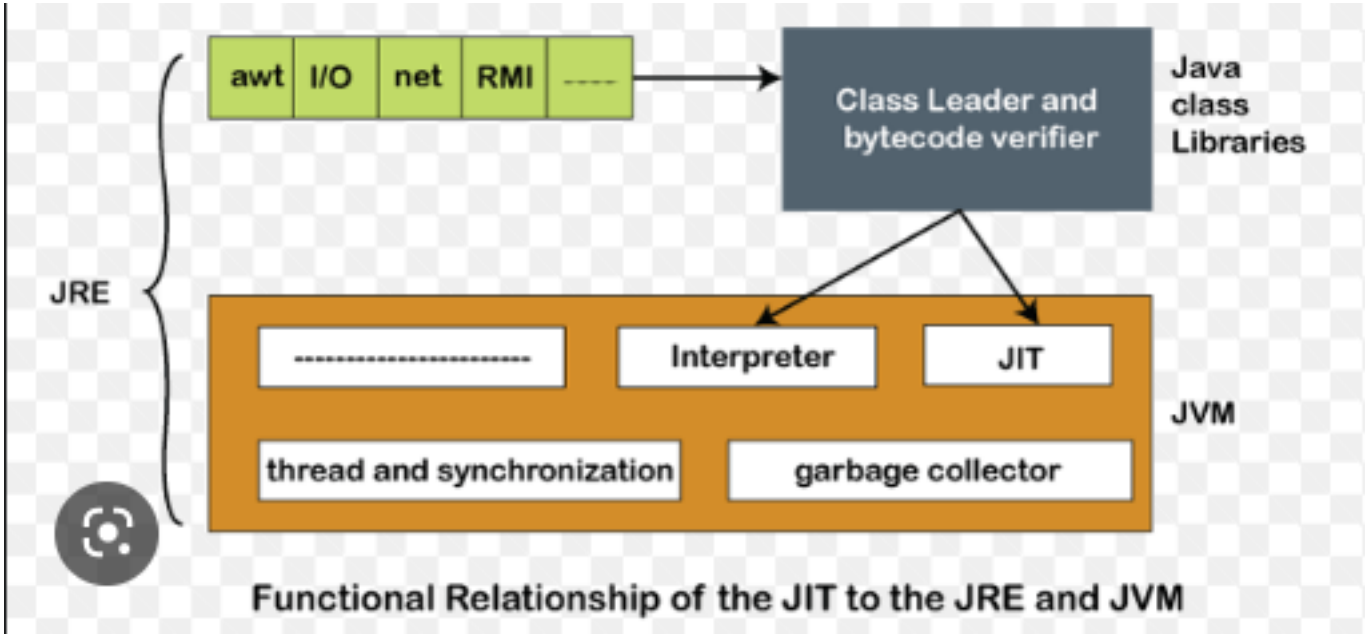
• If the language supported by VM1 is still not easy enough to use, another virtual machine (VM2) can be designed.

• This process can be repeated until a virtual machine (VMn) can be designed to support a powerful,

easy-to-use language.

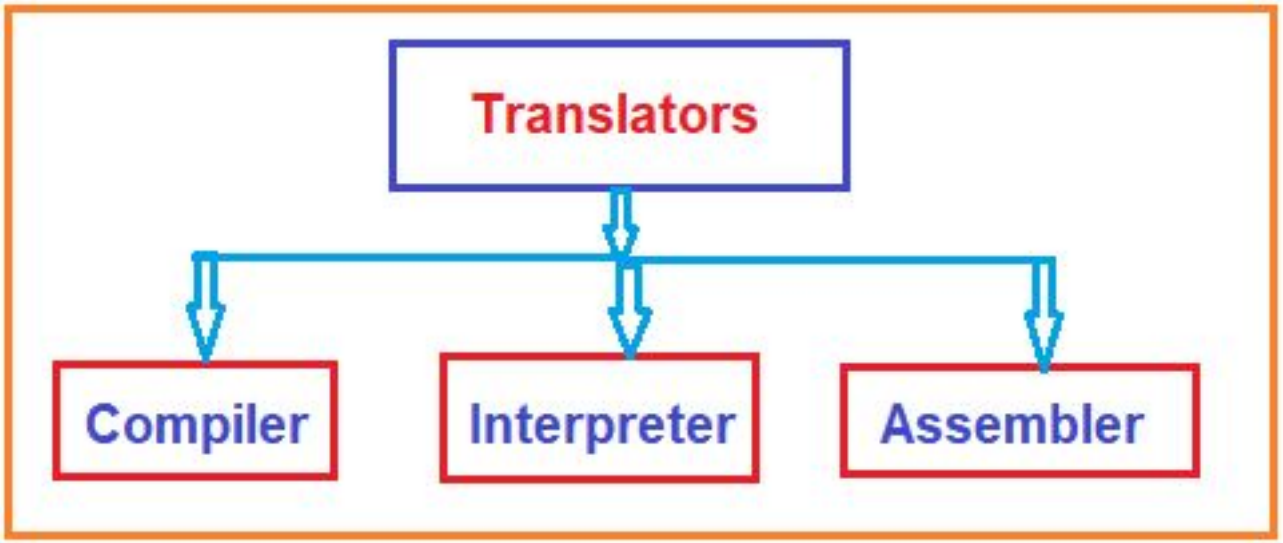
• The **Java** programming language is based on the virtual machine concept, where programs are translated into Java byte code and executed at runtime by a Java virtual machine (JVM).





Functional Relationship of the JIT to the JRE and JVM

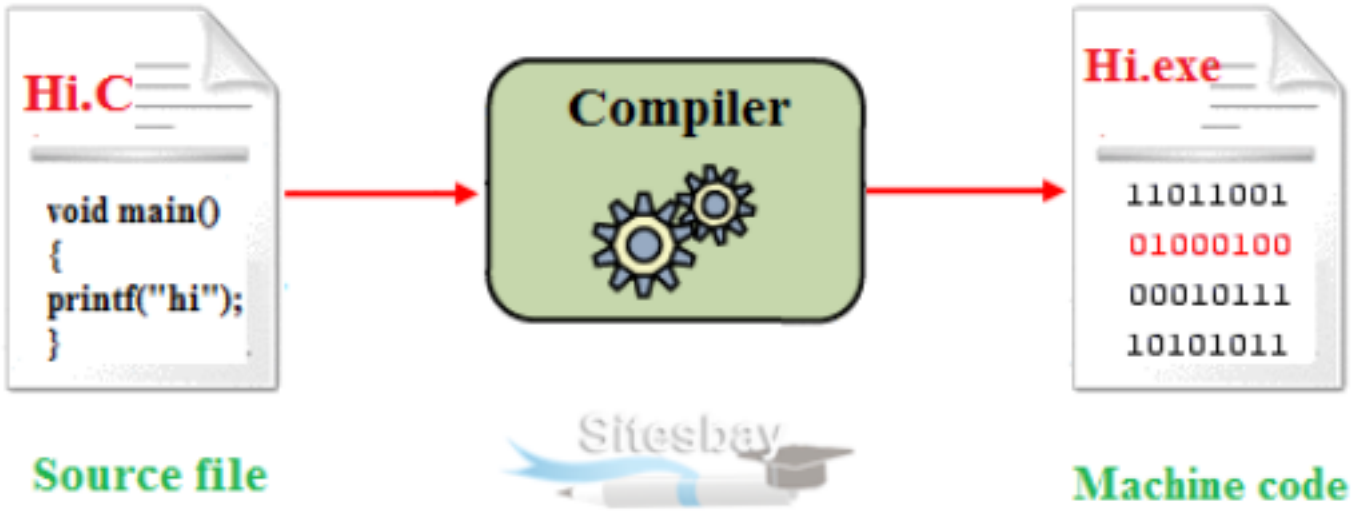
INTERPRETERS, COMPILERS AND TRANSLATORS



Interpreters:
• An interpreter is a program that reads high-level code, such as Python or Ruby, and executes it directly.
• Interpreters do not produce executable files, but instead, they run code line-by-line, translating each line into machine code as they go.
• Interpreters are slower than compilers because they execute code one line at a time.
• Interpreters are useful for quick prototyping, debugging, and scripting.

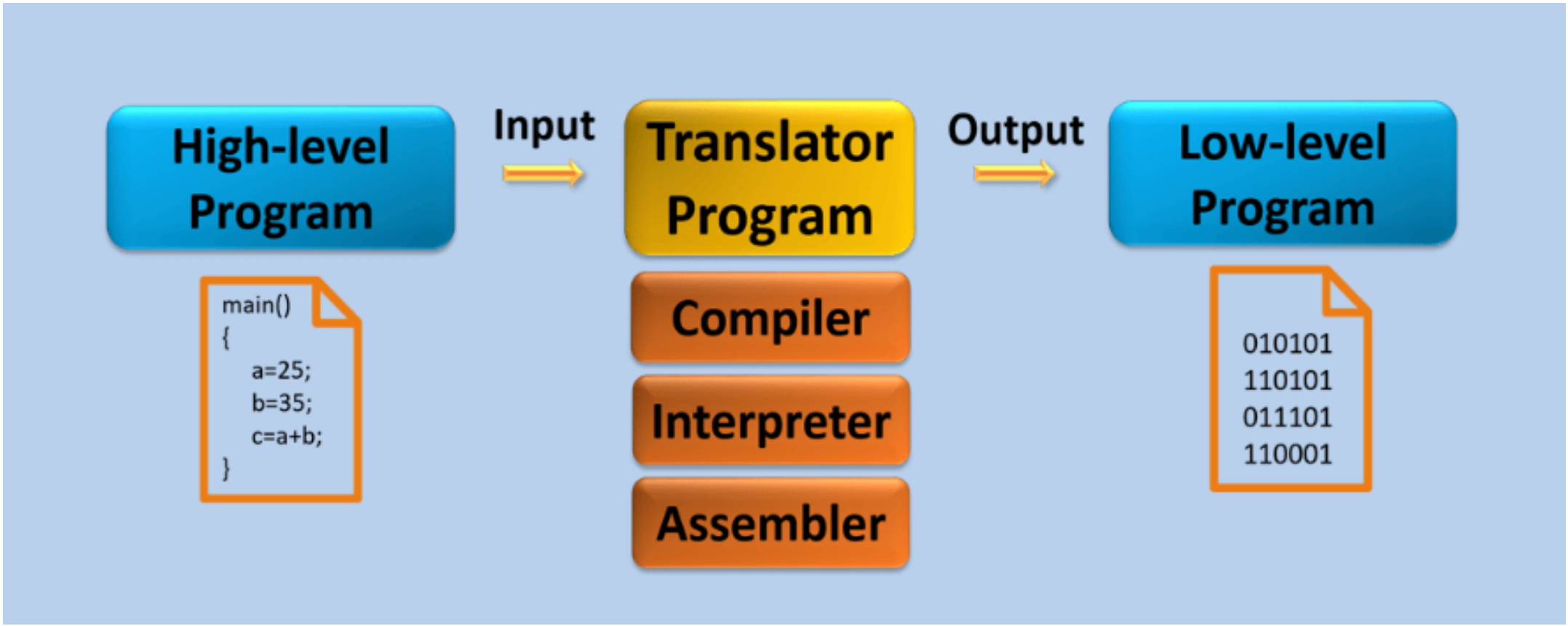
How Interpreter Works

Compilers:
• A compiler is a program that translates high-level code, such as C++ or Java, into machine code.
• Compilers produce executable files that can be run directly on the computer's hardware.
• Compilers analyze the entire code at once and generate optimized machine code that can run faster than interpreted code.
• Compilers are slower than interpreters during the compilation process, but the compiled code runs much faster.
• Compilers are used for developing large-scale applications and software libraries.



Translators:

- A translator is a program that converts code from one high-level language to another.
- Translators can be used to port existing software to a new platform or to convert code from an outdated language to a modern one.
- Translators can be either source-to-source or binary-to-binary.
- Source-to-source translators translate code from one high-level language to another high-level language.
- Binary-to-binary translators translate executable files from one machine architecture to another.



A translator is a program that converts code from one high-level language to another. is it a must that a translator translates high level languages only? isn't an assembler a translator too? but it translates low level assembly?

You are correct that the term "translator" can refer to programs that convert code from one language to another, regardless of whether the source language is high-level or low-level. In the context of programming languages, however, the term "translator" is often used specifically to refer to programs that convert high-level code to another high-level code, rather than converting low-level code to high-level or vice versa.

An assembler is a program that converts assembly language, which is a low-level language, into

machine code, which is the lowest-level code that can be executed by a computer's processor. Assemblers are not typically referred to as translators, as they do not convert high-level code to another high-level code. Instead, they translate assembly code to machine code.

So only compilers and interpreters are the only ones that are called translators. an assembler is not in that list?

The terms "compilers" and "interpreters" are more commonly used to describe programs that translate high-level code, whereas "assemblers" are used to describe programs that translate assembly code to machine code.

However, technically speaking, all of these programs are translators, as they are converting code from one language to another.

So, while it is not as common to refer to an assembler as a translator, it can still be considered a type of translator in the broader sense of the term.

Why do you suppose translated programs often execute more quickly than interpreted ones?

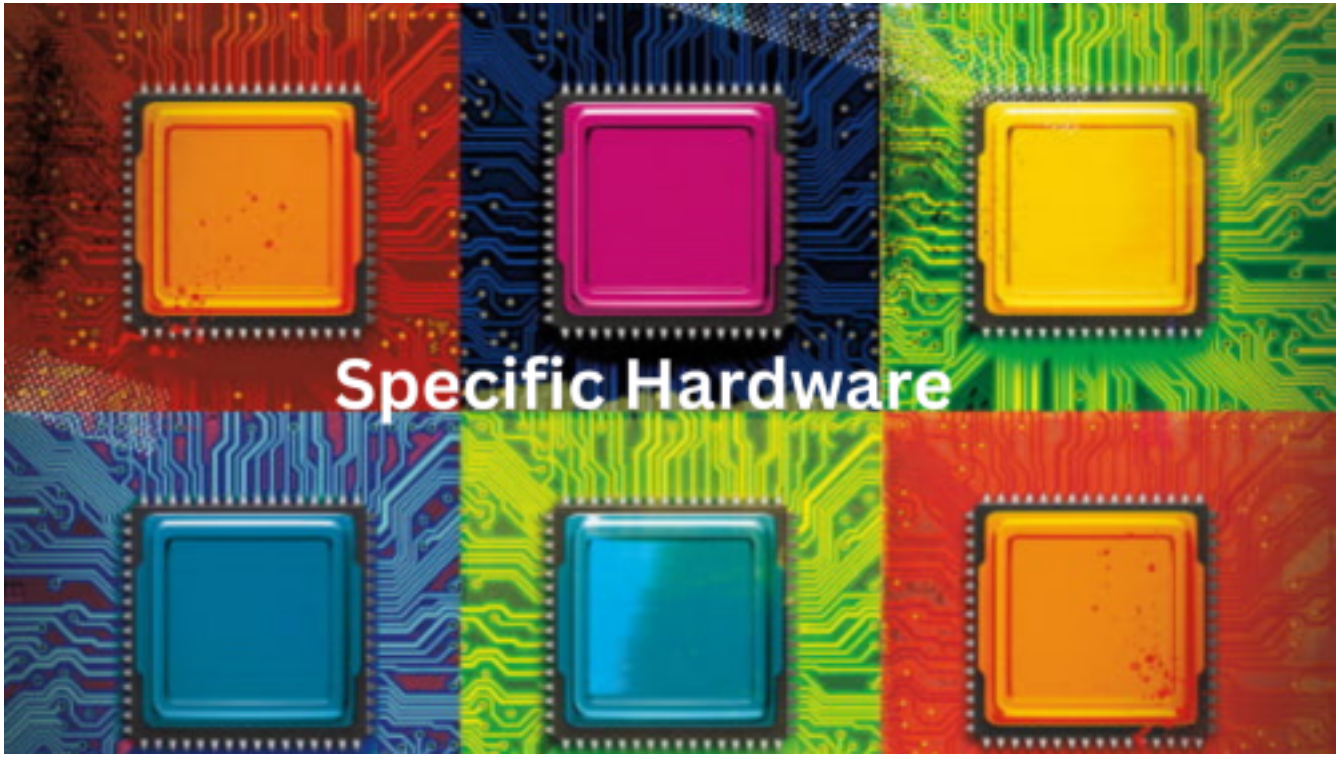
Translated programs often execute more quickly than interpreted ones for several reasons:

1. Compiled code is translated to machine code, which is a low-level language that can be executed directly by the computer's processor. **Interpreted code**, on the other hand, is executed by an interpreter, which must first interpret the code and then execute it. This additional step adds overhead and can slow down the execution of the code.

2. The **translation process for compiled code occurs only once**, during the compilation phase. Once the code is compiled, the resulting machine code can be executed repeatedly without the need for further translation. In contrast, interpreted code must be translated each time it is executed, which can slow down the overall performance.

3. Compiled code can often be optimized by the compiler to improve its performance. For example, the compiler may be able to reorder instructions or eliminate redundant operations. Interpreted code, on the other hand, is typically not optimized in this way.

4. Compiled code can be optimized for the **specific hardware** it will be running on, whereas interpreted code must be interpreted on the fly and cannot be optimized for a specific platform. This can lead to better performance for compiled code on the target hardware.



Overall, the main advantage of compiled code over interpreted code is that it can be executed more efficiently, with less overhead and better optimization. However, interpreted code has the advantage of being more flexible and easier to debug and modify, as there is no need for a separate compilation step.

(True/False): When an interpreted program written in language L1 runs, each of its instructions is decoded and executed by a program written in language L0?

In interpretation, each instruction of the program written in language L1 is decoded and executed by a program written in language L0 as the L1 program is running.

The interpreter is a program written in L0 that reads each instruction of the L1 program, decodes it into L0 instructions, and executes those L0 instructions to perform the task specified by the L1

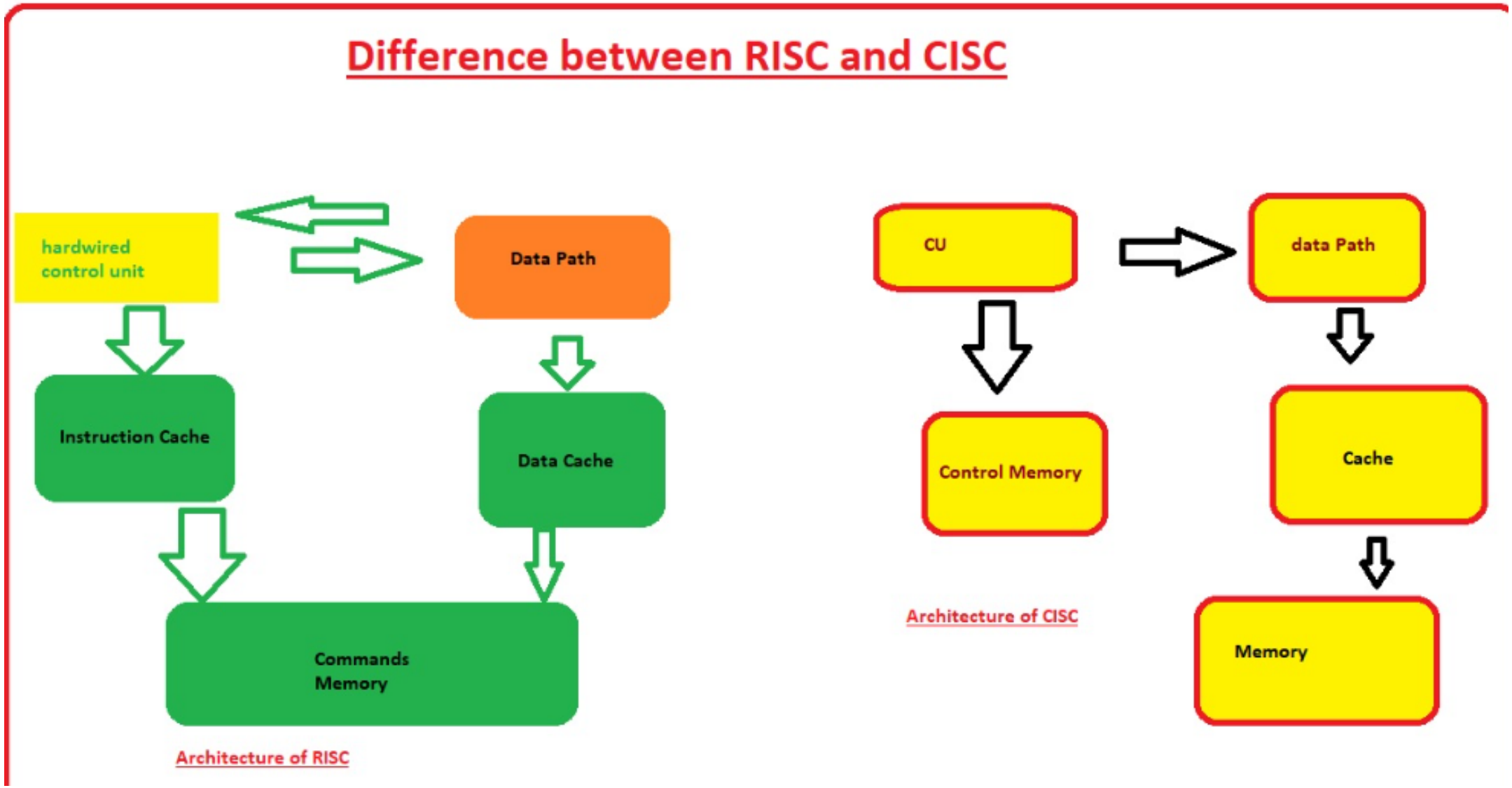
instruction.

This means that the program written in L1 is not directly executed on the computer hardware; rather, it is executed indirectly through the interpreter program, which translates the L1 instructions into L0 instructions that can be executed by the computer hardware.

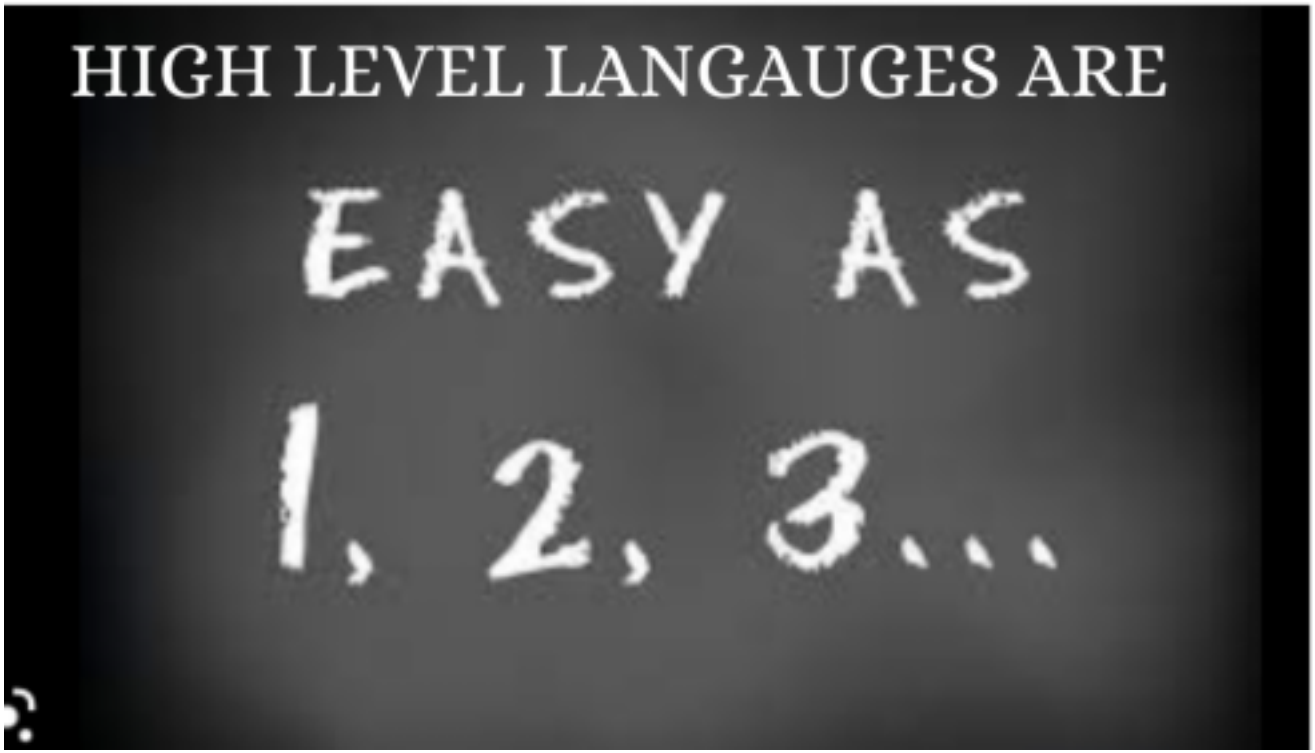
Explain the importance of translation when dealing with languages at different virtual machine levels.

When dealing with languages at different virtual machine levels, translation plays an important role in making it possible for programs written in one language to be executed on a virtual machine designed for a different language. Here are some reasons why translation is important:

1. Different virtual machines have **different instruction sets and formats**, which means that a program written for one virtual machine cannot be directly executed on another virtual machine without modification. Translation allows a program written for one virtual machine to be converted into a format that can be executed on a different virtual machine.



2. Programs written in a **high-level language are easier to read, write, and understand** than programs written in a low-level language. However, high-level languages are generally less efficient than low-level languages in terms of execution speed and memory usage. Translation allows programs written in a high-level language to be converted into a lower-level language, which can improve their performance.



3. Translation allows software developers to **write programs in a language that is suitable for their needs,** even if that language is not natively supported by the virtual machine on which the program will be executed. For example, a developer may choose to write a program in Python, even if the target virtual machine only supports C++.

Overall, translation is important because it allows programs written in one language to be executed on a virtual machine designed for a different language. This makes it possible for software developers to use the languages and tools that are best suited for their needs, without being limited by the capabilities of the underlying virtual machine.

Does assembly ever appear in the JVM?

Assembly language may appear after interpretation, depending on the implementation of the Java Virtual Machine (JVM) and the underlying hardware architecture.

When the JVM interprets Java bytecode, it converts each bytecode instruction into a sequence of

native machine instructions that can be executed directly by the hardware.

These machine instructions are typically represented in assembly language, which is a low-level programming language that is specific to a particular hardware architecture.

However, some JVM implementations may use just-in-time (JIT) compilation to dynamically generate optimized native machine code for frequently executed portions of the bytecode.

In this case, the resulting native machine code may not be represented in assembly language, as it is generated and executed at runtime rather than being precompiled.

Overall, whether assembly language appears after interpretation depends on the specific implementation of the JVM and the execution environment.

Why don’t programmers write applications in machine language?

Programmers do not typically write applications in machine language because it is a low-level language that is **very difficult to read and understand**, and it is **specific to a particular hardware architecture.** Writing applications in machine language requires a **deep understanding of the underlying hardware** and can be a **time-consuming** and **error-prone** process. Instead, programmers typically use higher-level programming languages, such as Java, Python, or C++, that provide more abstraction from the hardware and are easier to read and write.

Statements at the assembly language level of a virtual machine are translated into statements at which other level?

Statements at the assembly language level of a virtual machine are typically translated into statements at the machine language level of the underlying hardware architecture. Assembly language is a low-level language that is specific to a particular hardware architecture, and it provides a more human-readable representation of machine language instructions. The translation process from

assembly language to machine language is typically performed by an assembler, which is a software tool that converts assembly language code into executable machine code that can be run on the hardware.