**Introduction to Embedded C**

**computing system:**

A computing system refers to a combination of hardware, software, and data that work together to perform various computational tasks. It encompasses both the physical components (hardware) and the programs or instructions (software) that enable the system to process information and carry out specific functions.

The key components of a typical computing system:

1. **Hardware:** This includes the physical components of the system, such as the **central processing unit** (CPU), **memory** (RAM), storage devices (hard drives, SSDs), **input devices** (keyboard, mouse), **output devices** (monitor, printer), and other peripheral devices.
2. **Software:** This encompasses the programs and applications that run on the hardware. Software includes operating systems, application software (like word processors, web browsers, and games), system utilities, and other tools that facilitate various computing tasks.
3. **Data:** Data refers to the information processed and stored by the computing system. This could include text, images, videos, databases, and more. Data is manipulated by software and stored in various forms on the system's storage devices.
4. **Networks:** In modern computing systems, connectivity is essential. Many systems are part of larger networks, allowing them to communicate with other systems and access resources such as the internet, shared files, or remote servers.

The interaction between these components allows a computing system to execute tasks, solve problems, and provide various services. Different types of computing systems exist, including personal computers, servers, mainframes, embedded systems, and more, each designed for specific purposes and applications.

**Buses:**

In computing, a bus (short for busbar) refers to a communication system that transfers data between components inside a computer or between different computers. A bus consists of a set of conductors (wires or traces on a circuit board) that carry electrical signals and information between various components of the computer system.

There are different types of buses in a computer, and they serve various purposes:

1. **Address Bus:** This bus carries signals that specify a memory address. It is used by the CPU to indicate where data should be read from or written to in the system's memory.
2. **Data Bus**: The data bus is responsible for carrying the actual data between the CPU and other components like memory or peripherals. It allows the transfer of binary information in both directions.
3. **Control Bus:** This bus carries control signals that coordinate and manage the activities of the various hardware components. Control signals include commands such as read, write, interrupt, and clock signals.

The width of a bus refers to the number of parallel lines or bits it can transfer simultaneously. For example, a 32-bit data bus can transfer 32 bits of data in parallel.

Buses play a crucial role in the communication and coordination of different parts of a computer system, enabling the smooth flow of data and instructions between the processor, memory, and peripheral devices. The design and specifications of buses can vary based on the architecture and purpose of the computer system.

What do the buses look like?

Buses are electrical pathways or sets of conductors (wires or traces on a circuit board) that carry signals and data between various components on the motherboard or within the computer system.

A computer chip and a bus

Description automatically generated with medium confidenceOn a circuit board or motherboard, buses are represented by groups of lines, often in the form of copper traces. These traces connect different components, such as the central processing unit (CPU), memory modules, input/output ports, and other peripherals. The arrangement and layout of these traces depend on the specific design and architecture of the computer.

For example, you might see wide traces for the data bus, which allows the simultaneous transfer of multiple bits of data. Address buses and control buses may also have their own sets of traces. The bus lines are usually labeled to indicate their purpose, such as "ADDR" for the address bus or "DATA" for the data bus.

In more modern computers, especially in laptops and other compact devices, buses might be implemented using layers on a printed circuit board (PCB) rather than visible individual traces. Additionally, buses can be internal (within a single computer) or external (connecting multiple computers or devices).

**How the communication between a microprocessor and memory works:**

**Read Operation**

1. The microprocessor places the memory address it wants to read from on the address bus.

2. Control signals on the control bus indicate that it is a read operation.

3. The memory module at the specified address places the requested data on the data bus.

4. The microprocessor reads the data from the data bus.

**Write Operation**

1. The microprocessor places the memory address it wants to write to on the address bus.

2. The data to be written is placed on the data bus.

3. Control signals on the control bus indicate that it is a write operation.

4. The memory module at the specified address stores the incoming data.

This communication process allows the microprocessor to retrieve instructions and data from memory and store results back in memory during its operation. The efficiency of this communication plays a crucial role in the overall performance of a computer system.

**languages of Microprocessor:**

Microprocessors don't understand high-level programming languages directly; they are designed to execute machine code or assembly language instructions. Machine code is a set of binary instructions that the microprocessor can directly execute. Assembly language is a human-readable representation of machine code, using mnemonics and symbolic instructions.

**Machine language**

Machine language is a low-level programming language consisting of binary code or sets of binary-coded instructions that a computer's central processing unit (CPU) can directly execute. Each instruction in machine language corresponds to a specific operation that the CPU can perform, such as arithmetic calculations, data movement, or control flow.

Key characteristics of machine language include:

**1. Binary Representation:**

Machine language instructions are represented using binary code, which consists of 0s and 1s.

Each instruction corresponds to a specific pattern of 0s and 1s that the CPU recognizes and executes.

**2. Instruction Set:**

The set of instructions that a particular CPU architecture understands is called its instruction set.

The instruction set includes operations such as data manipulation (addition, subtraction, etc.), data movement (loading, storing), control flow (branching, jumping), and more.

Key point about instruction sets:

**The instruction format:**

The instruction format refers to the structure and organization of a machine language instruction. Each instruction in a processor's instruction set follows a specific format, which includes fields representing various components such as the opcode (operation code), operands, and addressing modes. The format is designed to be recognized and interpreted by the processor's control unit.

The exact format varies between different processor architectures, but here is a generalized structure that can be found in many instruction formats:

**Opcode (Operation Code):**

The opcode is a fundamental part of an instruction that specifies the operation to be performed by the processor. It identifies the type of instruction (addition, subtraction, load, store, etc.).

The size of the opcode field depends on the number of different operations the processor can perform.

**Operands:**

Operands are the data on which the operation specified by the opcode is to be performed. These can include registers, memory addresses, immediate values, or a combination of these.

The number of operand fields and their sizes depend on the specific instruction and the addressing modes supported by the processor.

Here's a simplified example of an instruction format:

[Opcode] [Operand 1] [Operand 2] [Operand 3] [Addressing Mode]

The actual format may vary, and different instructions within the same architecture may have different formats depending on their requirements. The structure of the instruction format is a fundamental aspect of the processor's design and determines how the control unit decodes and executes instructions.

**3. Low-Level Nature:**

Machine language is considered a low-level programming language because it directly corresponds to the hardware and architecture of the computer.

It is difficult for humans to read and write machine code directly, as it lacks the symbolic representation and abstraction found in higher-level languages.

**4. Specific to Architecture:**

Machine language is specific to the architecture of a particular CPU. Different CPU architectures have different instruction sets.

For example, the machine language for Intel x86 processors is different from that of ARM processors.

Here's a simple example to illustrate machine language:

In Intel x86 machine language, the binary representation for the "ADD" (addition) operation might look like this:

1000101100100110

This binary pattern instructs the CPU to perform an addition operation.

Programmers generally do not write programs directly in machine language due to its complexity and difficulty. Instead, they use higher-level programming languages or assembly languages, and then use an assembler or compiler to translate the code into machine language that the computer's CPU can execute. The transition from high-level languages to machine language involves a series of steps, including assembly language and compilation or interpretation.

**Assembly language**

Assembly language is a low-level programming language that serves as a human-readable representation of machine code instructions for a specific computer architecture. It provides a symbolic representation of the machine code, using mnemonics and symbols to represent the actual binary instructions understood by the computer's central processing unit (CPU). Programmers use assembly language to write programs that can be translated into machine code by an assembler.

Here are some key characteristics of assembly language:

1. **Mnemonics:**

Assembly language uses mnemonics, which are short, meaningful abbreviations or symbols representing machine code instructions. For example, "MOV" may represent a move operation, and "ADD" may represent an addition operation.

1. **Symbolic Representation:**

Assembly language provides a more human-readable and symbolic representation of machine code instructions. Instead of working with raw binary values, programmers use mnemonics and symbols to write instructions.

1. **Registers and Memory Addresses:**

Assembly language often involves working with registers and memory addresses. Registers are small, fast storage locations within the CPU. Memory addresses represent locations in the computer's memory where data is stored.

1. **Instruction Set Architecture (ISA):**

Assembly language is specific to the Instruction Set Architecture (ISA) of a particular processor or computer architecture. Different ISAs have different instruction sets, and assembly language programs are written accordingly.

1. **Low-Level Programming:**

Assembly language is considered a low-level programming language because it closely reflects the architecture and functionality of the computer's hardware. Programmers have more direct control over the hardware but need to manage details such as memory and registers explicitly.

1. **Assembler:**

Programs written in assembly language are not directly executed by the computer. Instead, they are translated into machine code by an assembler. The assembler converts the symbolic assembly language code into the binary machine code that the CPU can understand and execute.

Here's a simple example of assembly language code (using x86 syntax):

section .data

message db 'Hello, Assembly!',0

section .text

global \_start

\_start:

; write the message to stdout

mov eax, 4 ; syscall number for sys\_write

mov ebx, 1 ; file descriptor for stdout

mov ecx, message ; pointer to the message

mov edx, 17 ; length of the message

int 0x80 ; make syscall

; exit the program

mov eax, 1 ; syscall number for sys\_exit

xor ebx, ebx ; exit code 0

int 0x80 ; make syscall

In this example, the code prints "Hello, Assembly!" to the standard output. Each line corresponds to a machine code instruction, and the assembler translates it into the corresponding binary code. The exact syntax and mnemonics may vary depending on the architecture and assembler used.

Advantages & disadvantages of assembly

Assembly language programming has both advantages and disadvantages, and its suitability depends on the specific context and requirements of a given task. Here are some key advantages and disadvantages of assembly language:

**Advantages:**

1. **Low-Level Control:**
   * Assembly language provides a high degree of control over the hardware, allowing programmers to optimize code for specific tasks or hardware features.
2. **Efficiency:**
   * Programs written in assembly language can be highly efficient in terms of both execution speed and memory usage. This is because programmers have direct control over registers and memory.
3. **Specific to Architecture:**
   * Assembly language is specific to a particular processor architecture. This specificity allows programmers to take full advantage of the features and capabilities of a particular CPU.
4. **Minimal Overhead:**
   * Assembly language instructions directly correspond to machine code, resulting in minimal translation overhead. This can be important in scenarios where performance is critical.
5. **Real-Time Systems:**
   * Assembly language is often used in real-time systems where precise control over the timing and execution of operations is essential.
6. **Embedded Systems:**
   * Assembly language is commonly used in programming embedded systems where resource constraints, performance, and hardware-specific optimizations are crucial.

**Disadvantages:**

1. **Steep Learning Curve:**

Assembly language programming can be challenging for beginners due to its low-level nature and the need to understand the intricacies of the hardware architecture.

1. **Complexity:**

Assembly language code tends to be more complex and harder to read than code written in higher-level languages. This can make maintenance and collaboration more difficult.

1. **Portability Issues:**

Programs written in assembly language are typically not portable across different architectures. Code must be rewritten or adapted for each specific processor architecture.

1. **Time-Consuming:**

Developing software in assembly language is generally more time-consuming than using higher-level languages. It requires detailed attention to memory management and other low-level details.

1. **Limited Abstraction:**

Assembly language lacks the high-level abstractions provided by modern programming languages. Tasks that are straightforward in high-level languages may require more manual effort in assembly.

1. **Error-Prone:**

Due to the lack of built-in error checking and abstraction, assembly language code is more susceptible to human errors, such as typos or incorrect memory addresses.

1. **Decreased Productivity:**

While assembly language allows for fine-grained control, it can also lead to decreased productivity compared to higher-level languages that offer more abstraction and automation.

In summary, assembly language is a powerful tool for certain applications, especially those requiring low-level control and optimization. However, its use comes with trade-offs, and the choice of programming language depends on factors such as performance requirements, development time, and ease of maintenance. In many cases, higher-level languages strike a balance between productivity and performance.

**Assemblers VS compilers**

Assemblers and compilers are both tools used in the process of converting high-level programming languages into machine code or executable files, but they serve different purposes and operate at different levels of abstraction. Let's explore the key differences between assemblers and compilers.

**Assembler:**

1. **Level of Abstraction:**
   * Assemblers operate at a lower level of abstraction compared to compilers.
   * They translate assembly language code into machine code or object code.
2. **Input Language:**
   * Assemblers work with assembly language, which is a low-level programming language using mnemonic codes and symbols that represent specific machine code instructions.
3. **Output:**
   * The output of an assembler is typically an object file or machine code directly executable by the computer's CPU.
4. **Translation Process:**
   * Assemblers perform a one-to-one translation of assembly language instructions into machine code.
   * Each assembly language instruction corresponds to a specific machine code instruction.
5. **Efficiency:**
   * Assembler-generated code is generally more efficient in terms of execution speed and memory usage because it is closer to the machine architecture.
6. **Debugging:**
   * Debugging assembly language code is often more challenging than high-level languages due to the lower-level representation and lack of abstractions.

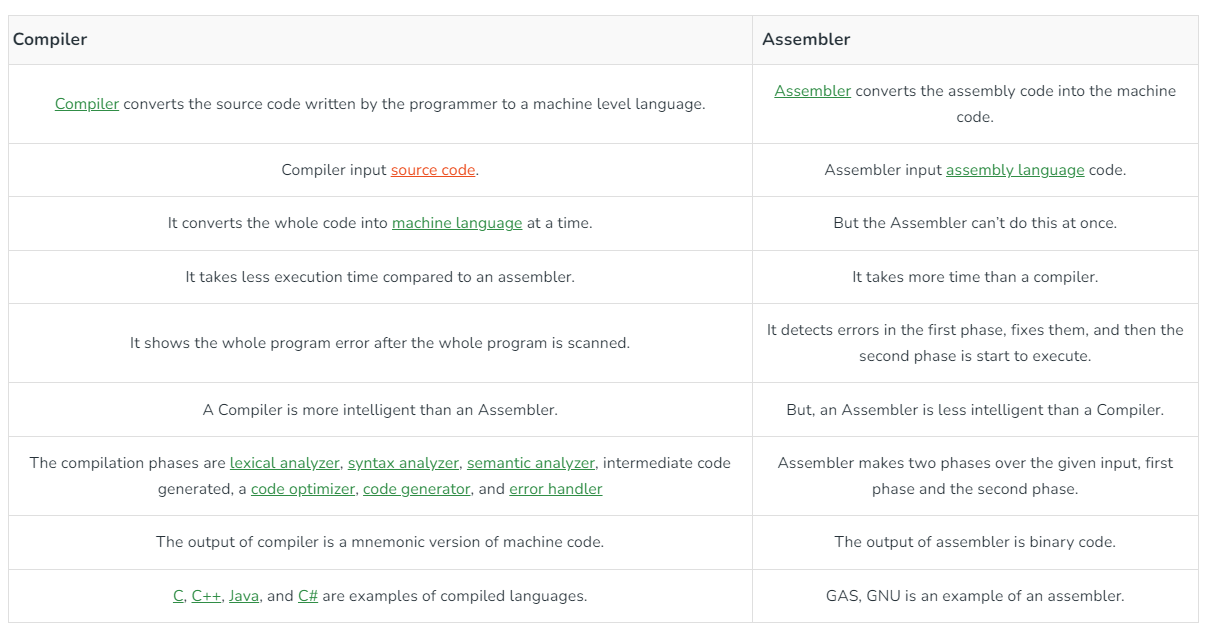
**Compiler:**

1. **Level of Abstraction:**
   * Compilers operate at a higher level of abstraction compared to assemblers.
   * They translate high-level programming languages (e.g., C, C++, Java) into machine code or intermediate code.
2. **Input Language:**
   * Compilers work with high-level programming languages that include more abstract constructs and are closer to human-readable code.
3. **Output:**
   * The output of a compiler can be an executable file, machine code, or intermediate code, depending on the compiler and its target platform.
4. **Translation Process:**
   * Compilers perform a multi-stage process, including lexical analysis, syntax analysis, optimization, and code generation.
   * The compiler translates the entire program into machine code or an intermediate representation before execution.
5. **Efficiency:**
   * Compiled code might be less efficient in terms of execution speed and memory usage compared to assembly language code, but it is more portable across different architectures.
6. **Debugging:**
   * Debugging high-level language code is often easier due to the use of meaningful variable names, abstractions, and higher-level constructs.

**Summary:**

* **Assemblers** are specific to a particular architecture, and they directly translate assembly language code into machine code. They operate at a lower level of abstraction and are closely tied to the machine architecture.
* **Compilers** work with high-level programming languages, providing more abstraction and portability. They perform a comprehensive translation process and generate machine code or intermediate code suitable for a specific target platform.

In practice, modern development often involves a combination of both assemblers and compilers, depending on the requirements of the project and the desired level of control over hardware details.



**why C not assembly?**

Choosing between C and assembly language depends on various factors, including the requirements of the project, the target platform, the level of control needed, development time, and the balance between performance and productivity. Here are some reasons why C is often preferred over assembly language in many scenarios:

**1. Abstraction and Readability:**

* C provides a higher level of abstraction compared to assembly language, making it more readable and easier to understand. This higher level of abstraction allows programmers to focus on the logic of the program rather than the low-level details of the hardware.

**2. Portability:**

* C code is more portable than assembly language code. Programs written in C can be compiled on different platforms with minimal or no modification, while assembly language code is specific to the architecture and often requires rewriting for different platforms.

**3. Development Productivity:**

* C facilitates faster development compared to assembly language. The use of high-level constructs and libraries in C allows for more efficient coding, reducing the development time and effort.

**4. Maintainability:**

* C code is generally easier to maintain than assembly code. The modular and structured nature of C promotes code organization, making it more straightforward for multiple developers to collaborate and maintain the codebase.

**5. Portability:**

* C code is more portable than assembly language code. Programs written in C can be compiled on different platforms with minimal or no modification, while assembly language code is specific to the architecture and often requires rewriting for different platforms.

**6. Easier Debugging:**

* Debugging is more convenient in C due to higher-level abstractions and better tool support. Modern integrated development environments (IDEs) provide advanced debugging features that simplify the identification and resolution of issues.

**7. Optimization by Compiler:**

* Compilers for C are sophisticated and capable of optimizing code for performance. They can generate efficient machine code, often rivaling or surpassing hand-written assembly code in terms of performance.

**8. Balance Between Control and Productivity:**

* C strikes a balance between low-level control and productivity. It allows programmers to have control over system resources and hardware features while providing a higher level of abstraction to simplify programming tasks.

**9. Standard Libraries:**

* C comes with standard libraries that provide a set of commonly used functions and routines. This eliminates the need for developers to reinvent the wheel for routine tasks.

**why C in Embedded systems**

C is commonly used in embedded systems for several reasons, making it a popular choice among developers in this domain. Here are some key reasons why C is preferred for embedded systems:

**Efficiency:**

C allows for low-level programming and manual control over hardware resources, enabling developers to optimize code for size and performance. This efficiency is crucial in embedded systems where resources are often limited.

**Portability:**

C code is generally portable across different embedded platforms with minimal modifications. This portability is valuable in the embedded systems domain, where various hardware architectures are prevalent.

**Access to Hardware Features:**

C provides direct access to hardware features and registers, allowing developers to write code that interacts closely with the embedded system's peripherals and components.

**Close to the Hardware:**

Embedded systems often require precise control over hardware components. C's low-level constructs and ability to manipulate hardware registers make it well-suited for programming at the bare-metal level.

**Memory Management:**

C allows manual memory management, which is important in embedded systems where memory constraints are common. Developers can allocate and deallocate memory with greater control over resource usage.

**Real-Time Capabilities:**

Many embedded systems require real-time responsiveness, and C provides the tools needed to implement real-time features efficiently. Tasks like interrupt service routines (ISRs) and time-sensitive operations can be written in C.

**Extensive Libraries and Tool Support:**

C has a rich set of libraries and tool support, making it easier for developers to work with common tasks and protocols in embedded systems. Standard libraries, like those defined in the C Standard Library, provide functions for various operations.

**Compilers and Optimization:**

C compilers are well-optimized, capable of generating efficient machine code. This is crucial for embedded systems where resource utilization and performance are critical.

**Community and Knowledge Base:**

C has a large and experienced developer community in the embedded systems domain. A wealth of knowledge, tutorials, and examples are available, making it easier for developers to find solutions to common problems.

**Legacy Code and Reusability:**

Many embedded systems have legacy codebases written in C. The ability to reuse existing code and libraries is a significant advantage for maintaining and extending embedded systems.

While C is widely used in embedded systems, it's worth noting that some applications with strict resource constraints might opt for languages like Assembly or even more specialized languages like Ada or MISRA C for safety-critical systems. The choice of programming language depends on the specific requirements of the embedded system, including performance, resource constraints, and the expertise of the development team.

**Building process**

The building process in software development involves transforming source code into an executable or deployable form. The process typically includes several stages, with each stage performing specific tasks. The exact steps can vary depending on the programming language, build tools, and development environment. Here is a generalized overview of the building process:

**1. Source Code:**

* Developers write the source code using a programming language such as C, C++, Java, Python, etc.

**2. Preprocessing:**

* The preprocessor stage handles tasks like macro expansion, inclusion of header files, and conditional compilation directives.
* The output is often referred to as the preprocessed source code.

**- compilation toolchain-**

**3. Compilation:**

* The compiler translates the preprocessed source code into object code (machine code or intermediate code).
* In languages like C and C++, this stage generates object files with a .obj or .o extension.

**4. Assembly (optional):**

* For languages like C and C++, an optional step involves converting assembly code (generated by the compiler) into object code.
* This step produces additional object files.

**5. Linking:**

* The linker combines multiple object files, resolves symbols, and generates the final executable or library.
* External libraries and modules may be linked during this stage.
* The linker produces the final executable file with an extension like .exe, .out, or without an extension, depending on the platform.

Build Tools:

**Makefiles:** A common tool for specifying build instructions and dependencies.

**Build Systems:** Tools like CMake, Gradle, and Maven simplify the build process and handle cross-platform considerations.

**Integrated Development Environments (IDEs):** IDEs often provide build tools and graphical interfaces for managing the building process.

The build process ensures that the software is translated from human-readable source code into a form that can be executed or deployed on a specific platform. It also helps manage dependencies, facilitate code organization, and automate repetitive tasks.