Chapter

1 Introduction

1.1 Overview

A *program* is a set of instructions, which are followed by the machine so as to generate a desired output. This means that writing a computer program is giving instructions to a processor, so as to delegate a particular job to the hardware of the computer system. Every instruction is a command given to the computer hardware to perform a specific job. Computer hardware is a digital system (collection of functional switches) and hence every instruction must be converted into the form of `0`'s and `1`'s (where a symbol `0` represents open switch and a symbol 1 represents closed switch). As an example, let us assume that we want the computer system to perform the addition of two numbers say `15` and `25`. The instruction to perform addition of two numbers could be written in the machine language as shown below:

```
10000011  00001111  00011001
```

In this case, the first eight bits represent the code informing the hardware that the addition of the two numbers is to be performed. This is called as an *opcode* (operational code) of the instruction. Different instructions would have different opcodes and their purpose is to convey the meaning of the instruction to the internal hardware circuitry. In this case, we have assumed an arbitrary opcode of `ADD` instruction as `10000011`. Different processors have different decoders and internal designs, hence the length and format of the opcode will certainly differ from processor to processor. Some processors have eight bit opcodes (e.g., intel 8085), some have 16 bit opcodes (e.g., intel 8086). Today's generation processors have 32 bit/64 bit opcodes or even 128 bit opcodes. We need not look into the hardware configurations and designs at this stage, but the key point to understand is that every instruction has an opcode and in this case, we just assume an arbitrary opcode of 8 bits as `10000011`, which represents `ADD` operation. A different combination of 8 bits, say `11001010`, may represent subtraction and so on. In theory, the variety of instructions any processor can offer is indirectly dependent on the length of its opcode. A processor with an opcode length of 8 bits can just offer 2^8 = 256 distinct instructions whereas a processor with an opcode length of 16 bits can offer 2^{16} = 65536 distinct instructions. We cannot just increase the length of opcode arbitrarily, the internal hardware and the instruction decoders must support it too. A processor that has a rich-instruction set certainly has highly effective internal circuitry and decoders to support it. In today's generation, we are working with processors having 32 bit opcodes

or 64 bit opcodes giving us rich- and high-performing instruction set, and this facilitates the execution of even complex programs in an optimized way. The next bits are the binary translations of the data values `15` and `25` over which addition is to be performed. The sample format of the `ADD` instruction is shown in Figure 1.1. We need not go too much in detail about computer hardware, however, the rationale of this discussion was just to make us clear that every processor has a digital circuitry, which only understands the language of `0`'s and `1`'s. This language is called as *machine language*.

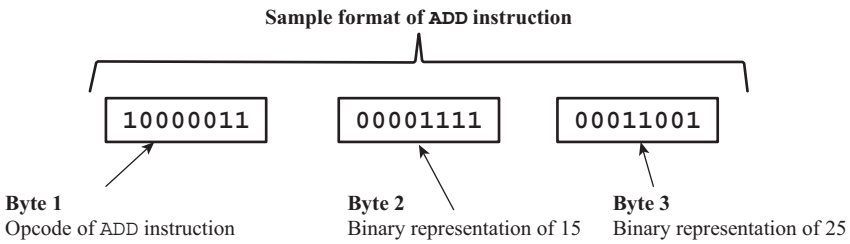


Figure 1.1: Representation of instruction in a machine language

Writing every instruction using *machine language* could be very complex when there are a large number of operations to be performed in the program as it requires us not only to work with `0`'s and `1`'s but also to understand hardware specifications of the processor. In today's generation, computer programs are written to design many complex applications having business challenges in itself, hence, it is practically impossible for a human being to write such programs in machine language.

To make the programmers life easy, an assembly language is designed, which codes every instruction using a *mnemonic*. For example, an instruction to perform the addition of `15` and `25` could be written in the assembly language as

```
ADD 15, 25
```

The symbol `ADD` is called as a mnemonic, which represents addition, whereas the constants `15` and `25` represent the data (also called as operands) over which the 'add' operation is to be performed. It is important to note that mnemonics are English symbols and hence they cannot be directly understood by the machine. Therefore, there is a need for a translator, which can translate the assembly language into a language of `0`'s and `1`'s. This will ensure that the hardware of the computer system can understand the meaning of the instruction, which is actually written in the assembly language by the programmer. The unit that performs the translation of assembly language into the machine language is called as an *assembler*. Therefore, the instruction `ADD 15, 25` will be first translated by the assembler into the machine language as shown in Figure 1.2. After the translation process is completed, the hardware of the computer system can execute the instruction, which will actually perform the addition of constants `15` and `25`.

The assembly language program can still be tedious to create if the program has a large number of operations to be performed. So as to make the programmer's life simple, the *high-level programming* languages are designed. A high-level programming language is an '*English-like*' programming language wherein the programmer can make use of user

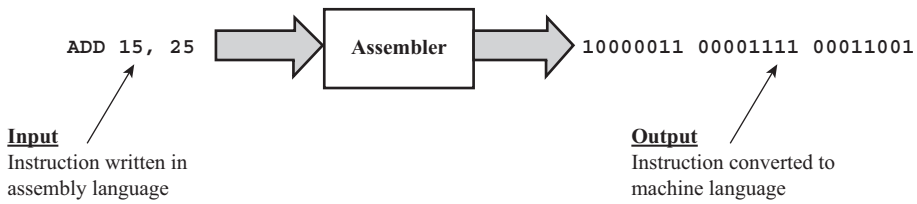


Figure 1.2: Assembler

friendly words and symbols to code an instruction. For example, we can write the following high-level instruction so as to perform the addition of two numbers:

```
Z = 15 + 25
```

Note that we have directly used a user friendly symbol + instead of coding the instruction in a machine language or an assembly language. Hence, the value of `Z` will be evaluated as `40`, which is the addition of `15` and `25`. High-level language is a set of 'English-like' symbols and hence these symbols cannot be directly decoded by the hardware of the system. Therefore, the high-level language is first converted into an assembly language by a unit called as 'compiler'. The assembly code can then be further converted into the machine code using the 'assembler as shown in Figure 1.3. It is important to reinforce on a point that the processor can execute a particular instruction only after it is translated into the machine language.

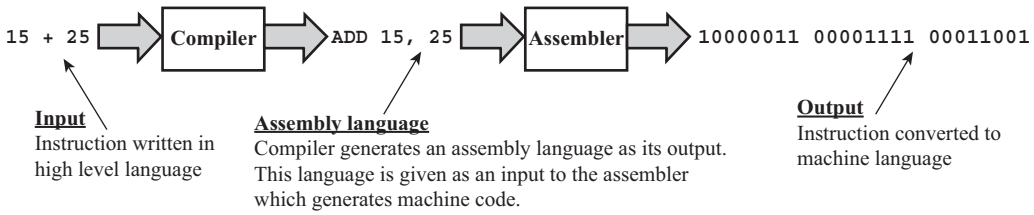


Figure 1.3: Translation of high level language to machine code

As the instructions in high-level programming languages are very user friendly and easy to code, they facilitate the creation of complex programs in a much more readable and maintainable then assembly or machine level languages. Hence, the programs written in high-level languages are easy to create, edit, debug and maintain. There are several high-level programming languages, which are currently being used in the software industries for creating different applications. Some of the high-level programming languages include C, C++, Java, Visual basic, FORTRAN, COBOL, etc. In this text book, we discuss computer programming using '*C and C++*'.

C programming language is designed by Dennis Ritche in 1973 and *C++* is designed by Bjarne Stroustrup in 1980 as an extension of *C*. Both *C* and *C++* are designed in AT&T Bell Laboratories. C++ is an extension of the 'C' language, which means that all the features supported by 'C' are also supported by C++. Furthermore, C++ adds many useful features such as object orientation and template management, which are not supported by 'C'.

What is the difference between Assembly Language and Machine Language? ?

Processor (CPU) is an integrated circuit that responds to a specific set of instructions. Instruction set for any processor is packaged along with its release and can be understood referring to hardware manuals of the processor. The instruction set of any processor is very much coupled and dependent on its internal circuitry. Every CPU has an instruction decoder which decodes the input instruction and passes it to the relevant architectural blocks for activating necessary hardware components to generate required results. Since an instruction needs to be ultimately decoded and executed by hardware, it must have a representation in binary form **(in the form of zeroes and one's).** This language of zeroes and one's which is directly executed by the processor is called as **machine language or Binary language**.

It is technically impossible for Human beings to communicate with processor directly using machine language. Hence any release of processor is also packaged with the mnemonics to each of the instructions it supports. These mnemonics are readable by human beings and is called as the assembly language of a particular processor. Remember, since the mnemonics are English like symbols, they cannot be passed directly to the hardware. It is the responsibility of system programmers to design an assembler which can convert assembly language into machine language. Assembler is a software utility and it is packaged along the system programs required to compile and execute any High Level language.

In practice, both Assembly and machine languages are categorized as Low Level Languages from **Application programmers perspective**.

ROADMAP OF THE BOOK

This text book is divided into two parts

Part 1: Chapters 1 to 9 are the topics, which are common with C and C++. The programs given in these chapters will work with both C as well as C++ compilers unless specified otherwise. Chapter 10 gives an explanation on dynamic memory allocation in C++ style. This feature is also supported by C but this book explains C++ notations.

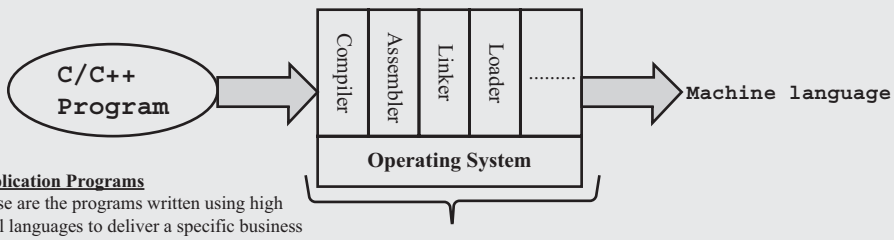
Part 2: Chapters 11 to 17 explain additional features, which are only supported by C++ and not by C. These are object-oriented features, which are not supported by C, hence the programs given in Chapters 11 to 17 will only work with C++ compiler.

It is impossible for a reader to understand the features of object orientation without having a thorough knowledge of structured programming, hence we should be extra careful when reading Chapters 1 to 10 as they become the prerequisite for Chapters 11 to 17.

NOTES

We have been saying that assembly and machine languages are very complex and it is practically impossible for a programmer to use these languages for application programming. Whilst this statement is very true, the *portability of programs* is another big problem in these languages. This is because assembly and machine languages are specific to a particular processor. For example, the assembly/machine language of an intel processor is very different from that of an AMD processor. Hence, even if we manage to write a program in assembly/machine language, our program may only be specific to our own platform and it cannot work if the hardware of the system is changed. It is absolutely not a good practice to design programs, which are very much dependent on hardware, because computer hardware

often gets upgraded with technology improvements. It should not happen that the program runs well on current system and fails after processor or hardware/operating system is upgraded or changed. C/C++ programs do not have a direct dependency on platform (hardware/operating system), once designed they could run on different platforms without major changes. We have used the phrase 'without major changes' because C/C++ is not fully platform independent, there are changes that programmer has to make the program while migrating the program from one platform to another. Note that C/C++ programs are just *platform dependent* and not *hardware dependent*; intermediate system programs such as operating system, compilers, loaders, and linkers make a cohesive architecture so that C/C++ programs can run on any hardware with almost no change in the code as shown in Figure 1.4. Because of the additional layer introduced by operating system, loaders, compilers, linkers (in general called as a *layer of system programs*), we can be sure of the fact that the C/C++ programs we create can run without any changes when just hardware of the system is changed or upgraded. This layered architecture gives hardware independence *to application programs*. Now the next question is, What if there are any changes in the system program? For example, what if we decide to change our operating system from windows to Linux? This is called as a change in platform and in this case, we are not sure that a C/C++ program which runs on a windows platform will also run on Linux. This is because the C/C++ compiler for windows is different than that of Linux. So, in practice, we will also have to change the compiler if we change the platform. Clearly, C/C++ languages have removed the hardware dependency on the programs but not the platform dependency and this is because compiler is different for different platforms. Whilst platform dependency remains one of the challenges with C/C++ programs, it is also a settled situation in software industries that platforms are not changed very often. Hence, C/C++ languages are still used at an extensive scale in the world of application development. In this discussion, we have mentioned names of some system programs such as loaders and linkers and we will debrief about them in section 1.3, for now just understand that these are some of the system programs which help to keep our code independent of the underlying hardware.



Application Programs
These are the programs written using high level languages to deliver a specific business requirement. High level programming languages like C and C++ are extensively used to design application programs.
These applications designed by programmers are used by business users who may not have any technical expertise.
Few examples of Application programs are program for online shopping used to design an online shopping website, a banking program used to perform banking transactions online, online railway reservation system used over the web, web site for booking movie tickets etc. Application programs need not be always **web based**, they can also be **Desktop applications** like Microsoft paint, Microsoft office, desktop games etc. All these applications are designed in some high level programming languages like C/C++. A web based program is a program which can be accessed over internet without a need of any prior installation whereas a desktop application is always accessed locally and needs to be installed on the machine before it can be used. All applications designed in this book are desktop based applications.

System Programs
These are utility programs which are necessary to develop and execute application programs. For example, operating system, compiler, assembler, loader, linker, etc.
Some of programs are tightly coupled with the hardware of the system and hence you must have knowledge about system hardware and configuration before you could design such programs
The scope of this text book is to learn application programming using C/C++. Design of system programs is out of scope of this text book, we may mention about them as and when needed to understand certain application programming concepts though.

Figure 1.4: Layered architecture of application program, system programs, and computer hardware

1.2 Computer System Architecture

Before deep diving into the intricacies of C/C++ programming, we need to first understand the general flow of data and instruction in a computer system. Figure 1.5 shows the basic building blocks of a system, which consist of the following units:

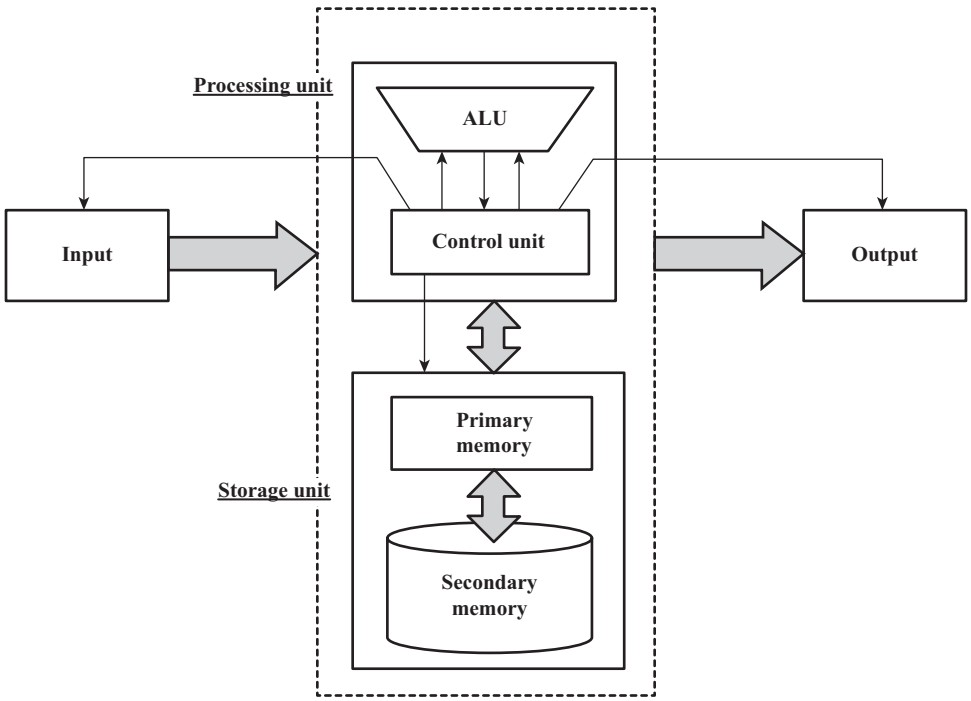


Figure 1.5: Block diagram of a computer system

1. Input unit
2. Processing unit
3. Output unit
4. Storage unit

1.2.1 Input to the system

'Data' and 'instructions' are given as input to the system so as to perform a particular operation. Here, the term 'instruction' represents the operation to be performed, whereas the term 'data' represents the information over which an operation is to be performed. For example, if we want the system to perform the addition of two numbers say x and y then we could write a statement as shown below:

$$x + y$$

The symbol + in this case, will be translated into an instruction `ADD` which will inform the underlying hardware to actually perform the addition of two numbers. The numbers x and y represent the 'data' over which the instruction `ADD` operates to generate the required output. We can give multiple instructions as input to the system, so as to get a consolidated

result. Let us consider that we want the computer system to give us a result of sequence of instructions say `I1,I2,…, In` which operate on a series of data values `d1,d2,…,dn`, respectively as shown in Figure 1.6. These instructions can be stored in the file and can further be processed by the CPU thereby giving the required result as an output.

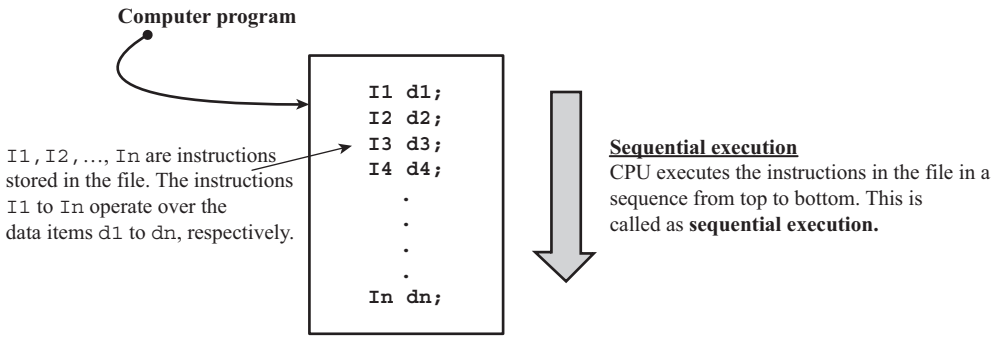


Figure 1.6: Computer program

The sequence of instructions that the system can process so as to generate the required output is called as *program* and the language in which a program can be written is called as a *computer programming language.* Although, we can write multiple instructions in a single file, the CPU can only process one instruction at a time. Hence, in reality, the processor runs just one instruction at a time in a sequence from first to last until all the instructions present in the file are fully executed.

This process of executing instructions in a file one by one is called as *sequential execution.*

1.2.2 Translation

As the hardware of a computer system is ultimately built up of electronic and semiconductor devices, it can only understand a binary language of 0's and 1's. However, it is impossible for us to write programs in machine language as this will involve a detailed study on the circuitry over which the system is built upon. In addition, as the fabrication of the machine changes, every machine will have a different machine language. This is because every machine is built up using different hardware technologies and circuitry. In summary, we do not understand machine language and machine does not understand the language we speak. This means that there is a need of a translator that can translate the 'human language' into the 'machine language'. The language that we can understand is called as high-level language whereas the language that only machines understand is called as a low-level language.

The set of programs that perform a translation of high-level language into a low-level language are called as *system programs.* The system programs take the high-level code (also called as source code) as input and generate an equivalent machine code at the output, as shown in Figure 1.7. The language compilers, assemblers, loader, linkers, etc. are examples of system programs, which are involved in translation of source code to machine code.

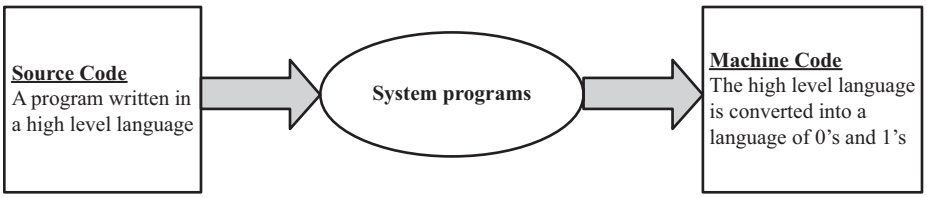


Figure 1.7: Translation of high level language to low level language

So as to ensure that the system programs correctly perform a translation from a source code to a machine code, we must follow certain rules while creating a program in the high-level language. These rules are also called as *'syntax'* of that language. Hence, every programming language has a *syntax,* which describes the set of rules and we must follow while writing a program. Furthermore, every programming language will have a *dedicated 'compiler'*, which converts the high-level language into a low-level language[1]. The compiler also checks if the programmer has preserved the syntax of the language while creating a program before it starts the translation process. The compiler will throw an 'error' message to the programmer if any of the syntax rules are violated, and the translation process is immediately terminated if at least one error is located in the input program. Hence, it becomes easy for the programmer to apply necessary corrections to the code and restart the compilation process after necessary fixes have been applied.

1.2.3 Processing unit

After the high-level language is translated into a language of `0`'s and `1`'s, the machine starts running each instruction one by one, so as to generate required output. The processing stage is called as '*execution stage*' of the program. Of course, the output of the code will be generated at the time when the code is under execution and not at the time when the code is getting translated or compiled. The process of translation or compilation just converts the program from one language to another while the process of execution actually runs the program thereby generating the results on output device. It is ultimately the hardware that performs the execution of the program. The hardware that executes the instructions written in the program one by one is called as '*processing unit*' of the computer system. The processing unit consists of two major blocks in it:

1. Arithmetic and logical unit (ALU)
2. Control unit

The ALU performs all the arithmetic and logical operations on the input data whereas the control unit is a circuitry that manages and controls the overall flow of data and instructions within the computer system. The control unit also consists of a set of decoders that can

[1] The output of compilation process is called as a target code. The target code differs from language to language as every language has its own compiler. The output of C/C++ compiler is called as object code which is a low level language.

understand the input instruction; it also passes the data to the ALU if the execution of the instruction requires any arithmetic or logical operation to be performed. The unit is responsible for reading the data and instructions from the 'input' devices, processing the instructions and sending the final results to the 'output' device so as to generate the output of the program.

1.2.4 Storage unit

The storage unit comprises of the memory devices, which are present in the computer system where instructions and data are stored. The area of the memory where instructions of the program are stored is called as a *'code segment'* whereas the area of the memory where the data required for the program is stored is called as a *'data segment'*. These memory segments are managed by the memory management unit (MMU) of the operating system in the primary memory of the computer system. In general, the memory units of a computer system are categorized as follows:

1. Primary memory (e.g. RAM)
2. Secondary memory (e.g. hard disk)

Hard disk or a secondary memory is generally a magnetic memory that stores the information persistently. The primary memory or RAM is a semiconductor memory that can store the information only for the time when the computer is powered ON. This means that RAM cannot store any information when the system is switched off and hence RAM is a 'volatile memory whereas a hard disk can store the information persistently even if the power is lost, therefore, hard disk as a 'non-volatile' memory.

Processor can directly access the data and information only if it is available in RAM, this is because CPU is also a semiconductor device, which is a very large-scale integrated circuitry (often abbreviated as VLSI). Hence, it is the duty of the MMU (unit part of operating system) to get the necessary data and information from hard disk to RAM when needed for CPU to access it, as shown in Figure 1.8. The transfer of information content from secondary memory to primary memory is also controlled by the MMU. The key point to note is that any program can be executed by the CPU if only the code and data referred by the program are brought into RAM.

When we 'open' any file for read or write operation, the file is actually copied from the hard disk to RAM by the operating system. This facilitates the CPU to directly access the data and instructions stored in the file. Once the MMU brings the file into the primary memory, CPU can then perform read /write operations on the file as shown in Figure 1.9. Therefore, once the file is opened, there are two copies of the file maintained in the system. The first copy of the file is in RAM whereas the second copy of the file is the original file present in the hard disk as shown in Figure 1.9.

When CPU writes any information only the file in RAM will be modified whereas the copy of the file in the hard disk will still represent an older version as shown in Figure 1.10. Hence, this is a state where the data in RAM is modified; however, the file in hard disk is stale. Such a write operation to a file is called as 'uncommitted' (or 'unsaved') write operation, which is performed by the CPU as shown in Figure 1.10.

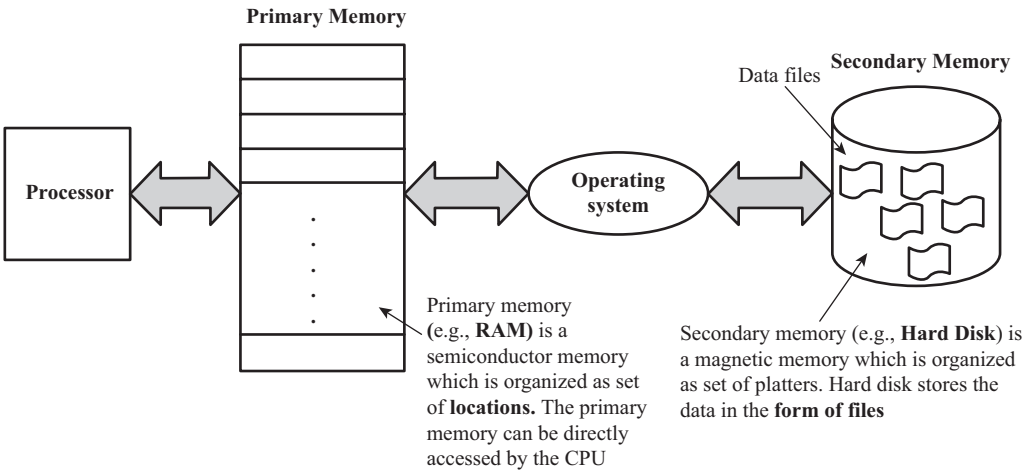


Figure 1.8: CPU accessing RAM

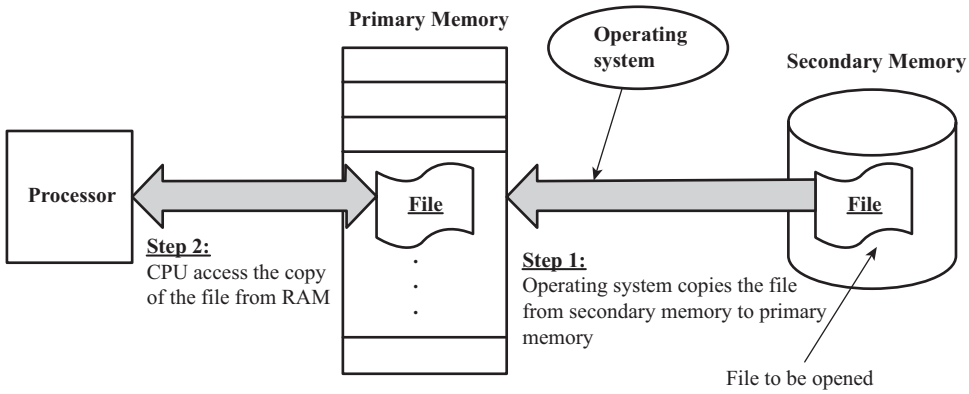


Figure 1.9: CPU accessing the file

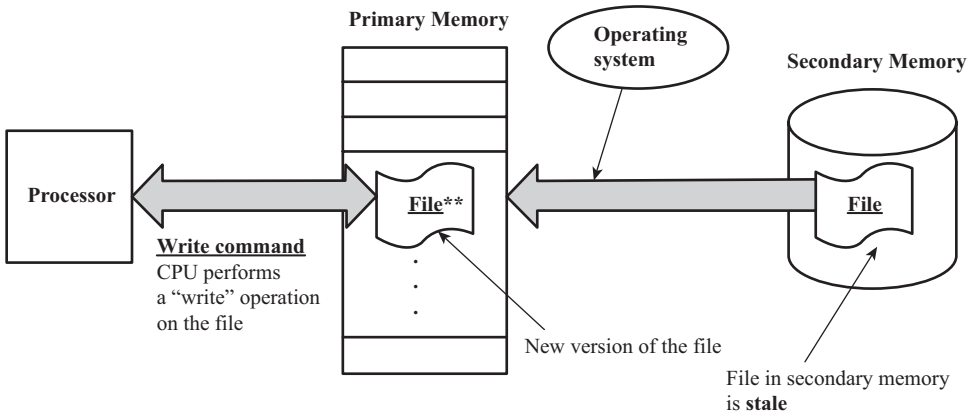


Figure 1.10: Uncommitted writes done to the file