Chapter 3

Introduction to Microcomputers

What is a microcomputer?
A microcomputer is a *fast* and *flexible* machine able to input, store and process data, and output the results as prescribed (dictated) by a group of machine instructions (called a *program*) that are also stored in the microcomputer. Therefore, a microcomputer has to be instructed (or programmed) properly to carry out the above functions; e.g., it may be instructed to read two numbers (call them *data*) from the keypad, add them up, store the sum in the memory and also send it to a 7-segment display. Figure 1 shows a simplified block diagram of a microcomputer:

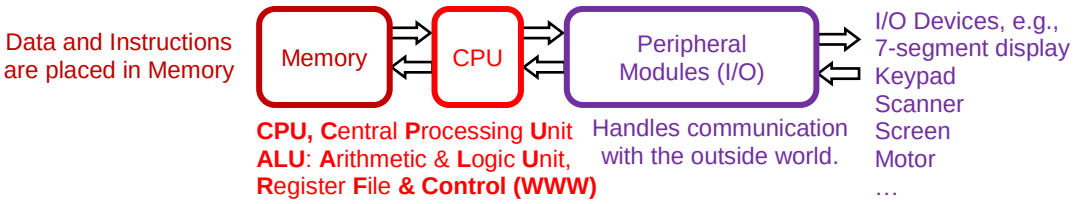

Figure 1. Simplified block diagram of a microcomputer

Let us take a closer look at the three keywords that we just used:

An **instruction** is a *basic*, *indecomposable*, or *atomic* action that a microcomputer can perform. In other words, an instruction is NOT made up of two (or more) smaller instructions. For example, {add 2 numbers} is an instruction.

As an example, the two numbers added by an add instruction are called **data**.

A group of instructions that a programmer puts together to get a job done is called a **program**. For example, you may write a program to calculate your grade average.

As illustrated in Figure 1, there are three main components in a microcomputer:

Memory (storage) holds data and instructions.

Central **Pr**ocessing **U**nit (CPU) reads instructions from memory and executes them.

Peripheral Modules (I/O) facilitate communication between the inside and outside worlds of a microcomputer.

A microcomputer is fast because it is made up of *electronic* devices, and flexible because it is *reprogrammable*. Unlike humans, microcomputers do not get tired, angry, or bored!

Figure 2 shows the block diagram of a microcomputer with more details:

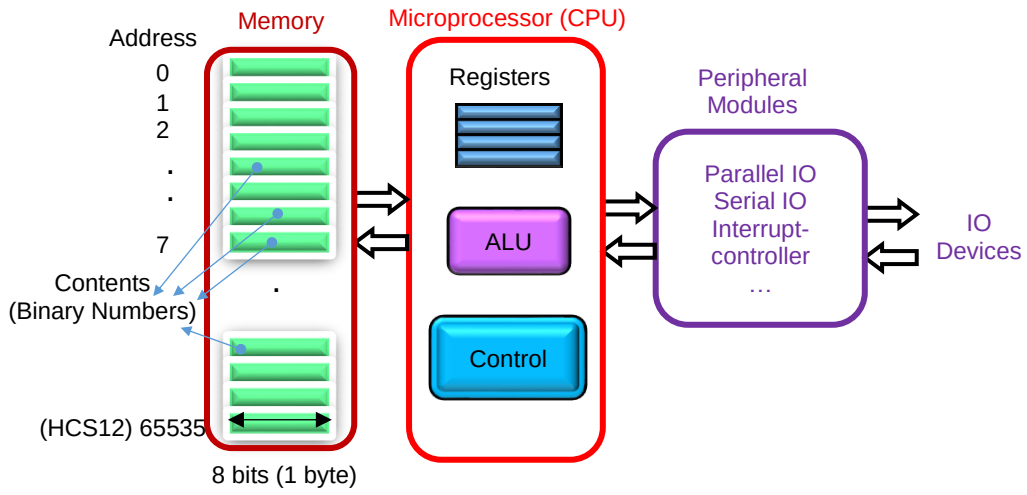

Figure 2. Microcomputer block diagram with more details

A good example of Peripheral Modules is a parallel output port (register) with, say, eight pins exposed to the outside world. This register can be written by the CPU. So, if, for example, an LED is tied to one of the output pins of the port, you can turn the LED ON and OFF under your program's control.

The Central Processing Unit (CPU) consists of three main blocks namely, **A**rithmetic **L**ogic **U**nit, (ALU), **R**egister **F**ile (RF), and **C**ontrol **U**nit (CU) as shown in Figure 2:

- Arithmetic and logic operations, such as addition, subtraction, bitwise AND, and bitwise OR are performed in the **ALU**.
- The **RF** is made up of a group of registers and used as a temporary or scratch-pad memory to hold the operands and the (intermediate) results of operations.
- The **CU** provides the whole system with timing information, in other words, it tells each block what to do at any time.

The memory consists of some groups of 8 bits called *bytes* as shown in Figure 2. Each byte has its unique *address*. To gain a *basic understanding*, you may look at a memory byte as an 8-bit register, which can be written and read as well. Here is a good analogy: the memory bytes are like the rooms in a building, and the address of each byte is the room number. In the HSC12 microcomputers, two back-to-back bytes are called one *word*. Each word has also its unique address, which is the address of the first byte of that word. In larger machines, one word is 4 bytes or even 8 bytes long.

Each memory location (such as a byte or word) has four attributes: *address*, *size*, *content*, and *meaning* of the content:

- **Address**: Each memory location has its specific address, which looks like the room number in a building. No two rooms or two memory locations may have the same number or address. The address of the first memory location is 0. The address of the last memory location depends on the size of the memory. In the HCS12 microcontroller that we will learn about in this textbook, the largest address is $2^{16} - 1 = 65535$. In other words, the maximum number of memory bytes (or the so-called memory space) is 65535. See Figure 2. The address of each memory location stays the same (does not change over time).
- **Size** shows how many bits are there in that memory location. The size stays the same (does not change over time).

- **Content** is the value (number) that each memory location holds at any time. Unlike the address, the content of a memory location may change over time.
- **Meaning**: The content of a memory location may have one of the following meanings:
  - **Data**, such as (part of) your bank account balance, or (part of) the address of another memory location.
  - (Part of) an **instruction**. Remember that instructions tell the CPU what to do.

  The meaning of the content of a memory location may change over time.

Note that a microcomputer is a digital system; therefore, similar to other digital systems its inside world is *binary*, including instructions, data, and addresses.

Back to the original question: *what is a microcomputer*? Let us take a closer look at an instruction: It is a string of 0s and 1s sitting in the memory and telling the CPU to do something. Let us say 0101 … 0001 is an instruction for a hypothetical CPU telling the CPU to take two numbers, A and B, subtract them, and place the difference in location C.

Question: Where is number A or number B located? Where is location C?

Answer: In general, A (as well as B) may be sitting in the memory, a CPU register, or it can be a constant (part of the instruction). Location C, the destination, can be a CPU register or a memory location.

Question: How does the CPU know where the operands (A and B) are, or where the destination (C) is?

Answer: This information and more, provided by the instruction *addressing mode*, is included in the instruction. Additionally, the addressing mode tells the CPU how to calculate the memory address of A, B, or C if they are in the memory. The next chapter covers the addressing modes of the HCS12 microcomputer.

To execute the above hypothetical subtract instruction, the addressing mode provides more information such as:

Take number A from the 8 LSbs of the instruction,

Take number B from CPU register No 5,

Put the difference (result) in the memory location whose address is generated using the following formula:

Destination Address (C) = 12 + Contents of CPU register No 7.

Note that this is just an example; it is not about a specific microprocessor.

The above instruction, which is represented more readably as {C ← A – B} is called a *3-address* instruction as three different locations participate in this subtraction. Now consider instruction {A ← A – B}, which is called a *2-address* instruction because the destination is the same as A, one of the operands, i.e., only two locations participate in the subtraction.

Two-address instructions are shorter than 3-address ones hence taking less memory space; this is an upside of 2-address instructions. In return, one of the operands (A in the above example) is overwritten in 2-address instructions, and therefore, the original value is lost, while in 3-address instructions none of the operands is overwritten; this is the upside of 3-address instructions.

Note: when we say A, we may mean the value of A or the location (address) of A; what we mean should be clear from the context. For example, if A is on the right side of an assignment, by A we mean the content of location A; and if A appears on the left side of an assignment, by A we mean the location of A.

A simplified view of instruction execution

Figure 3 depicts a simplified view of how the three blocks of a microcomputer communicate with each other, and therefore, how instructions execute. In this model, there are three *busses* between the memory and CPU:

- **Data bus** that carries data from the CPU to the memory. We may call it *Data-Out* for clarification purposes if necessary.
- **Address bus**, which carries address from the CPU to the memory.
- **Data bus** that carries data or instruction from the memory to the CPU. We may call it *Data-In* for clarification purposes if necessary.

Definition: A **bus** is a group of wires shared by two or more signal *sources* (*producers* or *transmitters*), or shared by two or more signal *destinations* (*consumers* or *receivers*).

For example, all the memory locations (look at them as receivers or destinations) share/use the same Data-Out lines as the value on Data-Out can be written in any of these memory locations. As another example, all the memory locations (look at them as transmitters or sources) share/use the same Data-In lines as each of these locations may place its content on Data-In when the memory is being read. So, either of these two, Data-Out or Data-In, is a *bus*.

Note that there are also control lines between different components not shown in Figure 3; these details are beyond the scope of this textbook. Interested students may consult a Computer Architecture textbook.

1

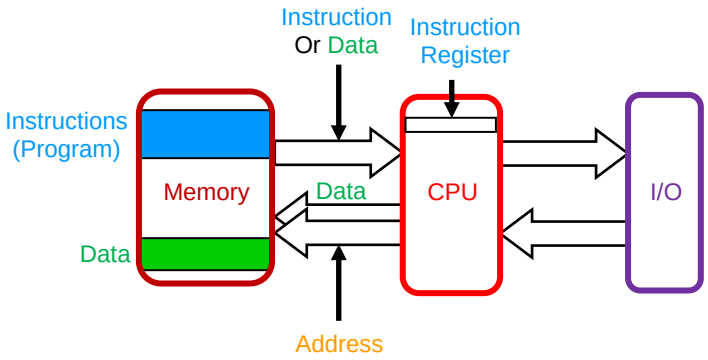

Figure 3. Communication between CPU and memory

To execute an instruction, the CPU takes two steps or cycles namely, the *Fetch Cycle* and the *Execution Cycle*. The combination of these two steps is called the *Instruction Cycle*.

Note: The term "cycle" used here should not be mistaken for the "clock cycle".

Fetch cycle:

In this step, the CPU sends the address of the next instruction to the memory using the address bus, receives the content of that memory location (which is the instruction) through the Data-In bus, and places the instruction in an internal register historically called the IR, **I**nstruction **R**egister, as illustrated in Figure 4:

2

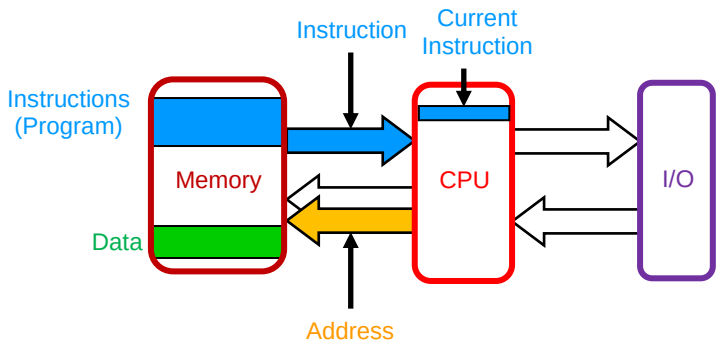

Figure 4. Instruction fetch cycle

Question: How does the CPU know what the address of the next instruction is?

Answer: The CPU has a register (among many other registers) called the PC, **P**rogram **C**ounter, or the IP, **I**nstruction **P**ointer, not shown in Figure 4 The PC is initialized to the address of the first instruction when the microcontroller is turned on. From now on, the PC is updated automatically (without the programmer's intervention) while the current instruction is being executed, so that the PC always shows the address of the next instruction. Technically speaking, we say the PC always *points* to the next instruction; this is why this register is also called the Instruction Pointer.

Definition: When we say a register (such as the PC) points to the memory location at the address, say, 4500, we mean that the content of the register is 4500; we call the register a *pointer*.

Back to the above question, when the CPU wants to fetch the next instruction, it simply checks the PC to obtain the address of the instruction and then places this address on the address bus.

Execution cycle:
Let us assume that the instruction that has just been fetched (and now is sitting in the instruction register) tells the CPU to read data from memory and place it in a CPU register. Figure 5 shows how the CPU executes this instruction: The CPU sends the address of the data to the memory; as a result, the memory sends the data (the content of that location) to the CPU, and then the CPU stores the data in the destination register specified in the instruction.

3

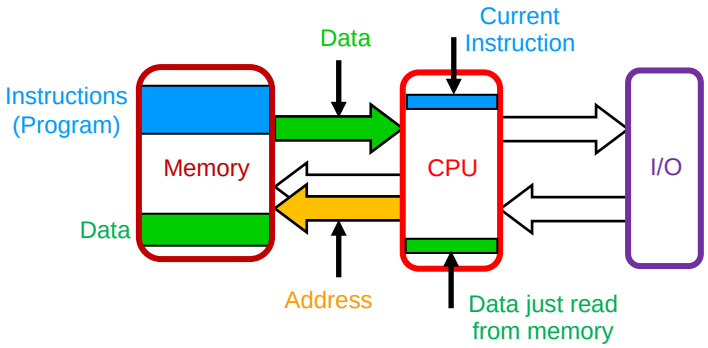

Figure 5. Instruction execution, an example: read data

Question: How does the CPU determine the address of data?

Answer: The addressing mode (included in the instruction) tells the CPU where to take the data from.

***

In our simplified model, once the current instruction executes, the CPU fetches the next instruction. This cycle is shown in Figure 6 again for ease of reference:

4

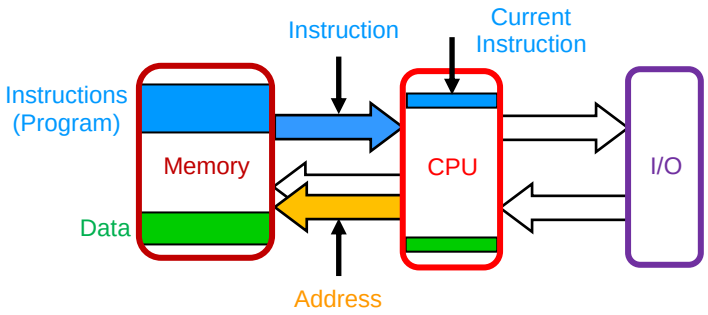

Figure 6. Instruction fetch cycle

Let us say the new instruction tells the CPU to add the constant 5 to the data that was read by the previous instruction and then put the result in a CPU register. As shown in Figure 7, as a result of this instruction execution, a new piece of data, {old data + 5}, is produced and stored in a register. The CPU is done with this instruction. Note that in this execution cycle, the CPU does not communicate with the memory.

5

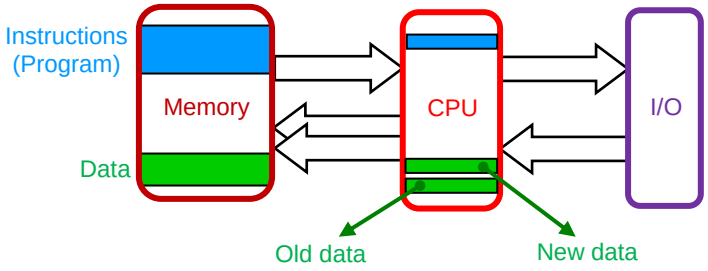

Figure 7. Instruction execution, an example: add 5 to data already in CPU

What is next? The CPU fetches the next instruction. This cycle is shown in Figure 8 again for ease of reference:

6

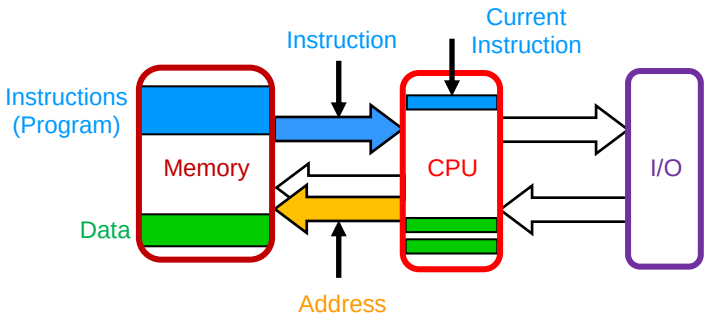

Figure 8. Instruction fetch cycle

Let us assume that the current instruction tells the CPU to store the new data in the memory. Figure 9 shows how the CPU executes this instruction: The CPU sends the data as well as its address to the memory, and then the memory writes the data at that address.

7

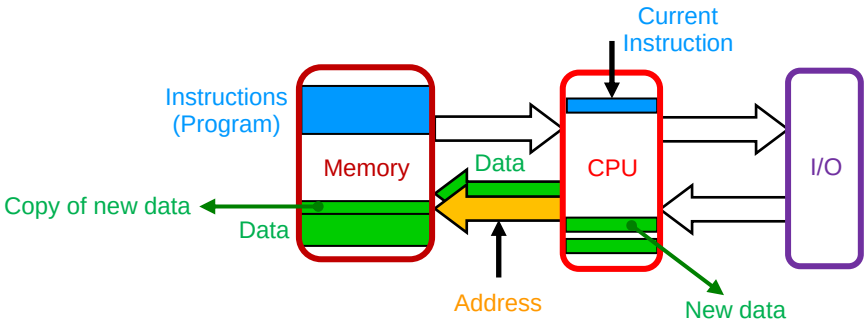

Figure 9. Instruction execution, an example: write data in memory

Using similar fetch/execute cycles, the CPU can, for example, read data from an input module, store it in a register, and then write it in the memory as shown in Figure 10 and Figure 11. Note that only data movements are illustrated in these figures; additionally, the address lines are not shown for the sake of simplicity:

8

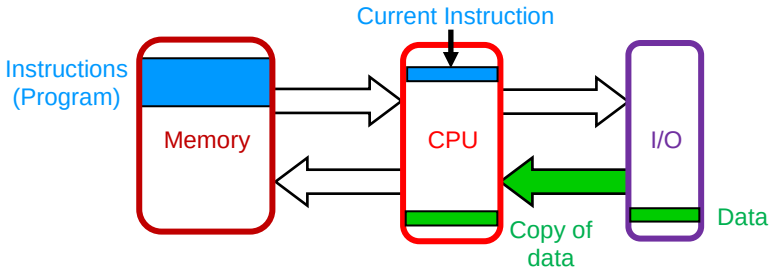

Figure 10. Read data from input

9

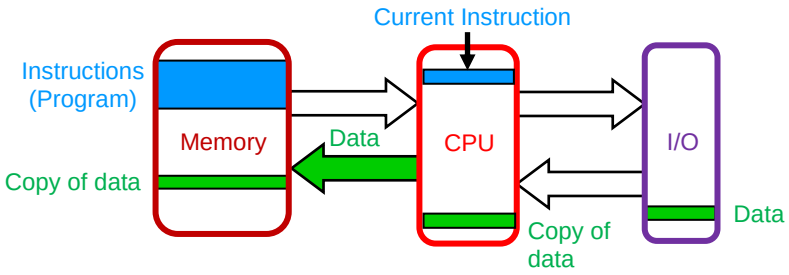

Figure 11. Write data in memory

Also using similar fetch/execute cycles, the CPU can, for example, read data from the memory, store it in a register, and then write it in an output module as shown in Figure 12 and Figure 13. Note that only data movements are illustrated in these figures; additionally, the address lines are not shown for the sake of simplicity:

10

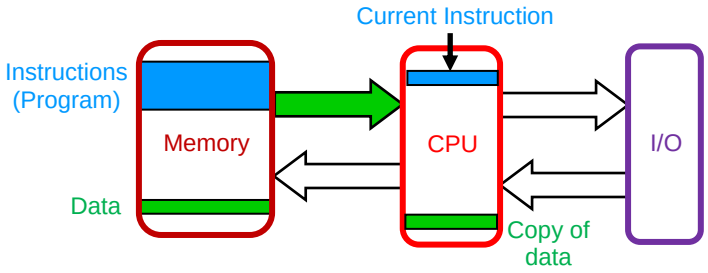

Figure 12. Read data from memory

11

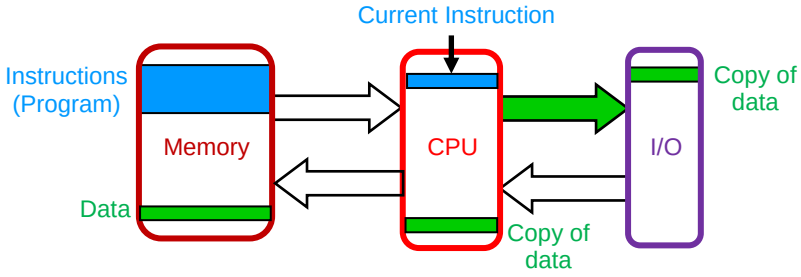

Figure 13. Write data in output

Machine/assembly instruction, language, and program

As you learned above, an instruction (or more specifically, a machine-*understandable* instruction) is a string of 0s and 1s that tells the CPU to do something, such as `{add two numbers}`. Instructions are the *basic*, *indecomposable*, or *atomic* actions that a microcomputer can perform. These instructions are called *machine* instructions.

The set of all the machine instructions of a microcomputer is called the *machine language* (or more specifically, the machine-*understandable* language) of that microcomputer. Every microcomputer has its own machine language.

A *machine program* (or *object code*) is a subset of the machine language chosen by the programmer to get a job done. For example, you may write a machine program to calculate your grade average.

Machine instructions are hard to read and write as they are made up of only 0s and 1s. We may convert these binary numbers to hexadecimal to make them less error-prone, but still not easy to read or write. To make instructions *human readable*, we use *mnemonics*. For example, to add two numbers, instead of the unreadable binary pattern `{1101 ... 0101}` in a hypothetical machine, we may just say `{add Y, A, B}`, which is called an *assembly instruction* (as opposed to a machine instruction). You agree that the assembly instruction is much more readable than its machine equivalent, do you not?

The set of all the assembly instructions of a microcomputer is called the *assembly language* of that microcomputer. Every microcomputer has its own assembly language.

An *assembly program* is a subset of the assembly language chosen by the programmer to get a job done. For example, you may write an assembly program to calculate your grade average.

Remember, however, that to run assembly programs we have to translate them into machine programs, as the machine language is the only one that the hardware understands. Therefore, we need a translator (software) called the *assembler*, which is a reasonable price for such a significant improvement in the readability of programs as well as ease of programming.

There are two different types of assemblers, namely, cross assembler and assembler (with no prefix):

Let us say assembler A translates assembly programs into machine programs for machine A. If assembler A runs on machine A, assembler A is simply called an *assembler* (with no prefix); otherwise, if it runs on a different machine, say B, then assembler A is called a *cross assembler*. See Figure 14:

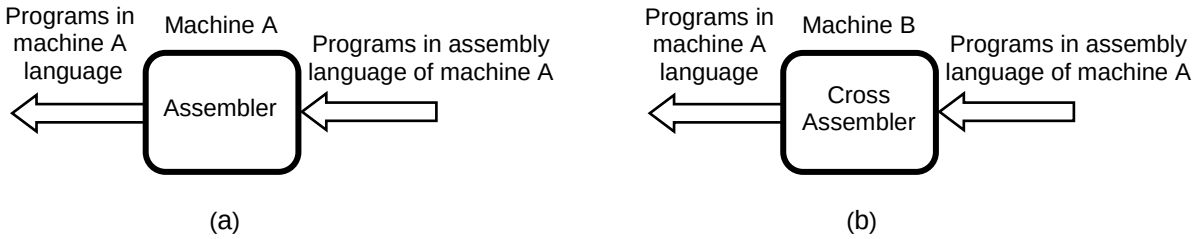

Figure 14. (a) Assembler A, (b) cross assembler A

Machine languages as well as assembly languages are called *low-level programming languages* as they are very close to the hardware, in other words, very far from the natural language. To make programming even easier, and make programs even more readable, we use *high-level programming languages* such as C++ or java instead of low-level programming languages. However, we now need a *compiler* to translate high-level programs into assembly programs; then the assembler will generate machine programs as graphically depicted in Figure 15:

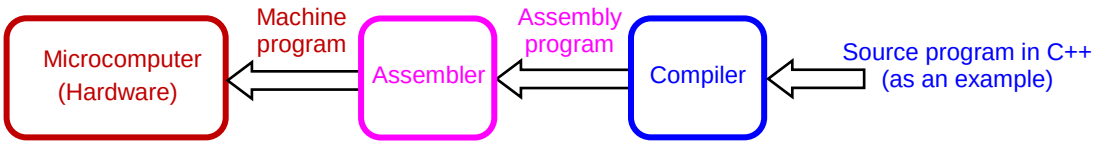

Figure 15. Programming at different levels

Other than the programming convenience offered by high-level languages, there is another major difference between these languages and assembly languages: high-level languages are almost machine-independent, while assembly languages are machine-dependent.