

# Beginners Guide in Computer Hardware

## Contents

- 1. The Fetch and Execute Cycle: Machine Language. . . . . 1
  - 1.1 Machine language. . . . . 2
- 2. Asynchronous Events: Polling Loops and Interrupts. . . . . 2

### 1. The Fetch and Execute Cycle: Machine Language

A COMPUTER IS A COMPLEX SYSTEM consisting of many different components. But at the heart - or the brain, if you want - of the computer is a single component that does the actual computing. This is the **Central Processing Unit**, or **CPU**. In a modern desktop computer, the CPU is a single "**chip**" on the order of one square inch in size. The job of the CPU is to execute programs.

A **program** is simply a list of unambiguous instructions meant to be followed mechanically by a computer. A computer is built to carry out instructions that are written in a very simple type of language called machine language. Each type of computer has its own **machine language**, and it can directly execute a program only if it is expressed in that language. (It can execute programs written in other languages if they are first translated into machine language.)

When the CPU executes a program, that program is stored in the computer's main memory (also called the RAM or random access memory). In addition to the program, memory can also hold data that is being used or processed by the program. Main **memory** consists of a sequence of locations. These **locations** are numbered, and the sequence number of a location is called its **address**. An address provides a way of picking out one particular piece of information from among the millions stored in memory. When the CPU needs to access the program instruction or data in a particular location, it sends the address of that information as a signal to the memory; the memory responds by sending back the data contained in the specified location. The CPU can also store information in memory by specifying the information to be stored and the address of the location where it is to be stored.

On the level of machine language, the operation of the CPU is fairly straightforward (although it is very complicated in detail). The CPU executes a program that is stored as a sequence of machine language instructions in main memory. It does this by repeatedly reading, or **fetching**, an instruction from memory and then carrying out, or **executing**, that instruction. This process - **fetch an instruction, execute it, fetch another instruction, execute it, and so on forever** - is called the **fetch-and-execute cycle**. With one exception, which will be covered in the next section, this is all that the CPU ever does.

The details of the fetch-and-execute cycle are not terribly important, but there are a few basic things you should know. The CPU contains a few internal registers, which are small memory units capable of holding a single number or machine language instruction. The CPU uses one of these registers - the **program counter**, or **PC** - to keep track of where it is in the program it is executing. The PC stores the address of the next instruction that the CPU should execute. At the beginning of each fetch-and-execute cycle, the CPU checks the PC to see which instruction it should fetch. During the course of the fetch-and-execute cycle, the number in the PC is updated to indicate the instruction that is to be executed in the next cycle. (Usually, but not always, this is just the instruction that sequentially follows the current instruction in the program.)

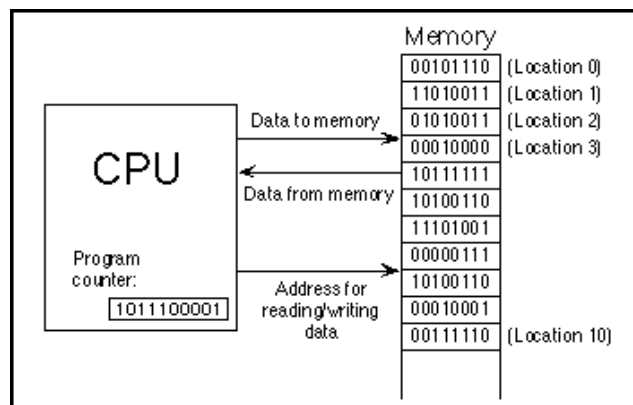
A computer executes machine language programs mechanically - that is without understanding them or thinking about them - simply because of the way it is physically put together. This is not an easy concept. A computer is a machine built of millions of tiny switches called transistors, which have the property that they can be wired together in such a way that an output from one switch can turn another switch on or off. As a computer computes, these switches turn each other on or off in a pattern determined both by the way they are wired together and by the program that the computer is executing.

### 1.1 Machine language

**Machine language** instructions are expressed as binary numbers. A binary number is made up of just two possible digits, zero and one. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. The data that the computer manipulates is also encoded as binary numbers. A computer can work directly with binary numbers because switches can readily represent such numbers: Turn the switch on to represent a one; turn it off to represent a zero. Machine language instructions are stored in memory as patterns of switches turned on or off. When a machine language instruction is loaded into the CPU, all that happens is that certain switches are turned on or off in the pattern that encodes that particular instruction. The CPU is built to respond to this pattern by executing the instruction it encodes; it does this simply because of the way all the other switches in the CPU are wired together. So, you should understand this much about how computers work: Main memory holds machine language programs and data. These are encoded as binary numbers. The CPU fetches machine language instructions from memory one after another and executes them. It does this mechanically, without thinking about or understanding what it does -- and therefore the program it executes must be perfect, complete in all details, and unambiguous because the CPU can do nothing but execute it exactly as written. Here is a schematic view of this first-stage understanding of the computer:

## 2. Asynchronous Events: Polling Loops and Interrupts

THE CPU SPENDS ALMOST ALL ITS TIME fetching instructions from memory and executing them. However,



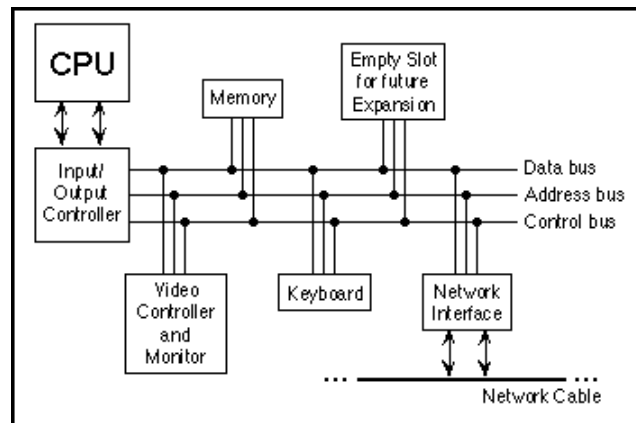
the CPU and main memory are only two out of many components in a real computer system. A complete system contains other devices such as:

- A **hard disk** for storing programs and data files. (Note that main memory holds only a comparatively small amount of information, and holds it only as long as the power is turned on. A hard disk is necessary for permanent storage of larger amounts of information, but programs have to be loaded from disk into main memory before they can actually be executed.)
- A **keyboard** and **mouse** for user input.
- A **monitor** and **printer** which can be used to display the computer's output.
- A **network interface** that allows the computer to communicate with other computers that are connected to it on a network.
- A **scanner** that converts images into coded binary numbers that can be stored and manipulated on the computer.

The list of devices is entirely open ended, and computer systems are built so that they can easily be expanded by adding new devices. Somehow the CPU has to communicate with and control all these devices. The CPU can only do this by executing machine language instructions (which is all it can do, period). So, for each device in a system, there is a device driver, which consists of software that the CPU executes when it has to deal with the device. Installing a new device on a system generally has two steps: plugging the device physically into the computer, and installing the device driver software. Without the device driver, the actual physical device would be useless, since the CPU would not be able to communicate with it.

A computer system consisting of many devices is typically organized by connecting those devices to one or more busses. A bus is a set of wires that carry various sorts of information between the devices connected to those wires.

The wires carry data, addresses, and control signals. An address directs the data to a particular device and perhaps to a particular register or location within that device. Control signals can be used, for example, by one device to alert another that data is available for it on the data bus. A fairly simple computer system might be organized like this:



Now, devices such as keyboard, mouse, and network interface can produce input that needs to be processed by the CPU. How does the CPU know that the data is there? One simple idea, which turns out to be not very satisfactory, is for the CPU to keep checking for incoming data over and over. Whenever it finds data, it processes it. This method is called **polling**, since the CPU polls the input devices continually to see whether they have any input data to report. Unfortunately, although polling is very simple, it is also very inefficient. The CPU can waste an awful lot of time just waiting for input.

To avoid this inefficiency, interrupts are often used instead of polling. An **interrupt** is a signal sent by another device to the CPU. The CPU responds to an interrupt signal by putting aside whatever it is doing in order to respond to the interrupt. Once it has handled the interrupt, it returns to what it was doing before the interrupt occurred. For example, when you press a key on your computer keyboard, a keyboard interrupt is sent to the CPU. The CPU responds to this signal by interrupting what is doing, reading the key that you pressed, processing it, and then returning to the task it was performing before you pressed the key.

Again, you should understand that this is purely mechanical process: A device signals an interrupt simply by turning on a wire. The CPU is built so that when that wire is turned on, it saves enough information about what it is currently doing so that it can return to the same state later. This information consists of the contents of important internal registers such as the program counter. Then the CPU jumps to some predetermined memory location and begins executing the instructions stored there. Those instructions make up an interrupt handler that does the processing necessary to respond to the interrupt. (This interrupt handler is part of the device driver software for the device that signalled the interrupt.) At the end of the interrupt handler is an instruction that tells the CPU to jump back to what it was doing; it does that by restoring its previously saved state.

Interrupts allow the CPU to deal with asynchronous events. In the regular fetch-and-execute cycle, things happen in a predetermined order; everything that happens is "synchronized" with everything else. Interrupts make it possible for the CPU to deal efficiently with events that happen "asynchronously", that is, at unpredictable times.

As another example of how interrupts are used, consider what happens when the CPU needs to access data that is stored on the hard disk. The CPU can only access data directly if it is in main memory. Data on the disk has to be copied into memory before it can be accessed. Unfortunately, on the scale of speed at which the CPU operates, the disk drive is extremely slow. When the CPU needs data from the disk, it sends a signal to the disk drive telling it to locate the data and get it ready. (This signal is sent synchronously, under the control of a regular program.) Then, instead of just waiting the long and unpredictable amount of time the disk drive will take to do this, the CPU goes on with some other task. When the disk drive has the data ready, it sends an interrupt signal to the CPU. The interrupt handler can then read the requested data.

Now, you might have noticed that all this only makes sense if the CPU actually has several tasks to perform. If it has nothing better to do, it might as well spend its time polling for input or waiting for disk drive operations to complete.

All modern computers use multitasking to perform several tasks at once. Some computers can be used by several people at once. Since the CPU is so fast, it can quickly switch its attention from one user to another, devoting a fraction of a second to each user in turn. This application of multitasking is called timesharing. But even modern personal computers with a single user use multitasking. For example, the user might be typing a paper while a clock is continuously displaying the time and a file is being downloaded over the network.

Each of the individual tasks that the CPU is working on is called a thread. (Or a process; there are technical differences between threads and processes, but they are not important here.) At any given time, only one thread can actually be executed by a CPU. (In fact, though, there are computers that have more than one CPU, and such computers can run several threads at the same time - one per CPU.) The CPU will continue running the same thread until one of several things happens:

The thread might voluntarily yield control, to give other threads a chance to run. The thread might have to wait for some asynchronous event to occur. For example, the thread might request some data from the disk drive, or it might wait for the user to press a key. While it is waiting, the thread is said to be blocked, and other threads have a chance to run. When the event occurs, an interrupt will "wake up" the thread so that it can continue running. The thread might use up its allotted slice of time and be suspended to allow other threads to run. Not all computers can "forcibly" suspend a thread in this way; those that can are said to use preemptive multitasking. To do preemptive multitasking, a computer needs a special timer device that generates an interrupt at regular intervals, such as 100 times per second. When a timer interrupt occurs, the CPU has a chance to switch from one thread to another, whether the thread that is currently running likes it or not.

Ordinary users, and indeed ordinary programmers, have no need to deal with interrupts and interrupt handlers. They can concentrate on the different tasks or threads that they want the computer to perform; the details of how the computer manages to get all those tasks done are not relevant to them. In fact, most users, and many programmers, can ignore threads and multitasking altogether. However, threads have become increasingly important as computers have become more powerful and begun to make more use of multitasking.

While programmers don't actually deal with interrupts directly, they do often find themselves writing event handlers, which, like interrupt handlers, are called asynchronously when specified events occur. Such "**event-driven programming**" has a very different feel from the more traditional straight-through, synchronous programming. We will begin with the more traditional type of programming, which is still used for programming individual tasks, but we will return to threads and events.

By the way, the software that does all the interrupt handling and the communication with the user and with hardware devices is called the **operating system**. The operating system is the basic, essential software without which a computer would not be able to function. Other programs, such as word processors and World Wide Web browsers, are dependent upon the operating system. Common operating systems include UNIX, OS/2 Warp, DOS, Windows 95, 98, NT, 2000 and the Macintosh OS.