CHAPTER 1 The Digital Computer

1.1 The Digital Computer

Overview

We show that a computer is awake when power is turned on because there is a permanently programmed control computer inside managing the computer's affairs. The permanent program in this hidden computer implements the user instruction set available to the using programmer.
Next we view the computer as a hierarchy of levels ranging from the bottom digital logic level to the top level high level language. This leads into a brief discussion defining computer architecture. Then we discuss the levels of instructions used in a computer: user instructions, microinstructions, and control instructions. Finally we touch on the performance issue.

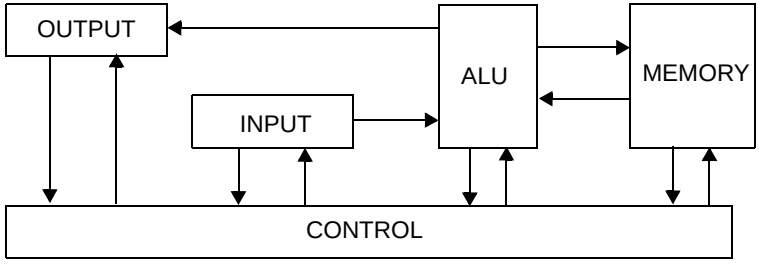
Introduction

The digital computer solves a problem by executing a sequence of instructions a user gives to it. The sequence of instructions is called a computer program. The instructions in the program are drawn from the user set of instructions. We will show that this user instruction set defines the computer. A designer transforms the user instruction set into a computer schematic. The designer may also participate in formulating the user instruction set. We call the transformation process computer design. This process leads to many hardware implementations for the same instruction set because synthesis is not unique, and because many technologies are available.

1.1.1 Computer Elements

The computer block diagram shows an ALU (arithmetic logic unit) interacting with a memory and input/output units in a controlled environment (Figure 1.1 on page 2). The traditional story explaining this block diagram is something like the following. The ALU performs arithmetic and logical operations as required by a user's program. The operations are performed on data stored in the memory. This means the ALU loads data from, and stores data in, the memory. The input block allows a user to enter programs and data into the computer. The computer delivers results to the user via the output block. In the background the control manages the process.

FIGURE 1.1 Traditional Computer Block Diagram

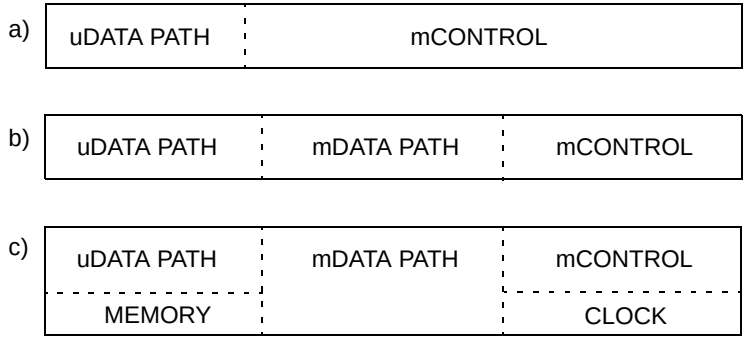


We prefer to present the substance of this matter in another way (Figure 1.2 on page 2) because the story above does not explain why a computer functions even before you have entered a program. If the computer is not awake how can you enter the program? Furthermore, the story does not point the way to a design process.

First know that every computer is managed by an internal computer a user does not have access to. To show this we start by partitioning the computer into two parts as if it were simply another state machine (which it is). The two parts are the uData Path and the mControl (Figure 1.2 on page 2). U means user accessible and m means inaccessible micro. In turn the internal computer mControl partitions into the mData Path and the mCtrl (control) because it also is a state machine (Figure 1.2 on page 2). Finally, we make the memory and clock explicit (Figure 1.2 on page 2).

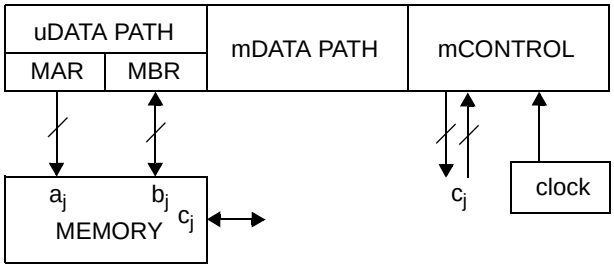
Presumably the next question is: who programs the internal computer? The answer is the designer does by using the computer's microprogramming language. This language may be bits stored permanently in a non volatile memory or simply wiring.

FIGURE 1.2 Evolution of a Computer Block Diagram



A **microprocessor** is a specialized form of computer wherein the memory and clock are literally external to the microprocessor (Figure 1.3 on page 3). Otherwise, nothing is different from the designer's point of view.

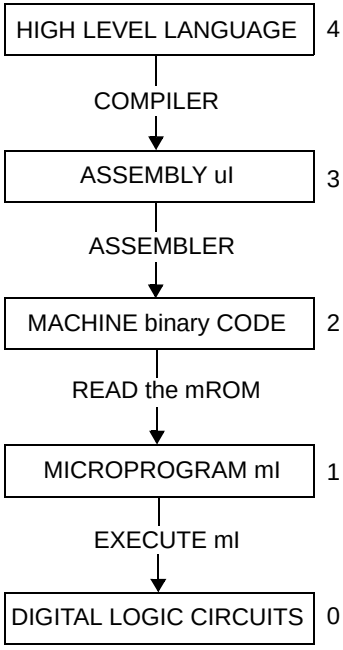
FIGURE 1.3 Microprocessor Configuration



1.1.2 Computer Levels

A computer may be viewed as a hierarchy of levels (Figure 1.4 on page 4).
The bottom level0 consists of digital logic circuits controlled by the next level1. This is the microprogrammed instruction (mI) level.
The mI are stored in what we call the mROM as a set of mI sequences. Any sequence is selected by addressing the first mI in the sequence. This is done by level2 known as the machine binary code level. Note: mI level1 may appear to be missing in some computers (e.g. RISC).
Machine codes are generated by the assembler translating assembly language into binary code. Assembly language level3 is written with user instructions uI. In turn the assembly language may be generated by the compiler translating high level language level4 into assembly language. Now reverse the process. Moving from top to bottom write a program in C (for example) and compile it into an assembly language whose words are taken from the uI defining the computer. The uI are translated by the assembler into binary codes which select sequences of mI. Sequences of mI execute to control digital logic and solve the problem programmed in C in the first place. Each sequence of mI executes one uI.

FIGURE 1.4 Computer Levels



1.1.3 Computer Architecture

Instruction set architecture is the instruction set visible to the programmer. Given this ul set we move on to implementation which is mostly organization and hardware. In simple terms organization is the computer's block diagram and bus structure. Hardware is the digital design and physical implementation. Hardware is the working machine. Finally, we say that to us computer architecture is all of the above.

A more common and circumscribed notion of computer architecture is one which an assembly or machine language programmer understands as he or she writes programs that work. Programs that work independently of the timing of the machine. This notion allows for a family of machines where each member is tailored for a specific market.

1.1.4 Levels of Instructions

In general there are three levels of instructions built into a computer: user instructions ul, microprogram instructions ml, and control instructions cl. We find the distinctive names ul, ml, and cl facilitate the discussion.

ul set: Top down design starts from the ul set. How do you know what ul set you need? You don't know until you analyze the problems you want to solve on the computer. Clearly this is an issue separate from the task of designing a computer. We will pull ul sets out of thin air and proceed (Table 2.2 and Table 3.1). We presume the sets allow us to write any program we please. We will

not digress to discuss the issue concerning whether or not the uI set is complete in the Euclidean sense. From Euclid's axioms you can prove anything (well almost) because the set of Euclidean axioms is complete. We assume our uI sets are complete.
Any computer program is essentially a list of instructions. The instructions used in any program are taken from the uI instruction set defined for your computer. The uI specify what the computer must be able to do. This is the computer designer's point of view. In turn the designer uses the uI set as a basis for computer organization and hardware design. In this way a design starts from the top and works its way down to the minute details.
Analysis of what each uI must do reveals actions common to many uI. For example most uI move data. Later we show that each uI is a small program with its list of "sub instructions". We show the number of different sub instructions is much smaller than the number of uI. The practical consequence from use of these sub instructions is simplified implementation of the uI codes in the computer. In practice the sub instructions are the microprogram instructions, or microinstructions (mI).
mI set: Later we demonstrate heuristically that any uI can be implemented by a microprogram that uses 4 different mI in some combination. They are

TABLE 1.1 mI used in microprograms implementations

mALU	arithmetic and logic
mMOV	copy words
mBR	branch decisions
mNOP	do nothing

The following is not obvious. In most computers uI are not implemented directly by the hardware because they cannot execute in one computer cycle. Each uI is implemented by a list of mI. In turn each mI is implemented directly by the hardware so that each mI executes in one computer cycle. This is why each uI is implemented as a program consisting of a list of microinstructions mI. In some architectures (e.g. RISC) each mI list has only one mI in it.
Reminder: the user does not know about the mI. The designer does.
cI set: Finally there are the very few cI. The mControl section of the computer performs a sequence of fetch and execute cycles. This implies a control program built into the computer hardware. The control program is a list of control instructions we call the cI. The cI are not used in programs by anyone: they are out of sight.
This is why three sets of instructions concern the designer

TABLE 1.2 Basic instructions set

cI	control
mI	micro
uI	user

Note: Later we show the cI are mI used to implement control functions. This is why when you look for cI in texts you will find them disguised as mI.
How is any user program uI list placed in computer memory? First, a computer memory stores binary digits or bits, 0 and 1. More to the point a memory is organized to store bit groups called words. It is organized in the sense each word has a unique memory address so that the word at any

address can be fetched (a read). Conversely a new word can be stored in any addressed memory location (a write).
Next, usually your computer program is written in uI assembly language

```
e.g. ADD r3 r7
```

or in a language such as Pascal (e.g. r7 := r3 + r7). Clearly the text of your computer program must be translated (compiled) into binary words before you can store your program in memory. The binary words describing your program are also a language: the machine language of your computer. Compilers are user tools that create binary word files stored on disks. A loader is another tool that copies your machine language computer program from disk to memory. Tools allow you to place your program in memory and to pass control to the computer's control section. An assembler is a specific type of compiler.

1.1.5 Computer Performance

Performance is not understood in the sense that there is no process you can implement at the end of which you have a predictable increase in performance. Furthermore there is no definitive process by which you can evaluate performance. Nevertheless performance design and evaluation methods and guidelines exist.
The computer that performs a unit of work in the least time is the fastest. Time may be elapsed time; the total time required to do the job. However elapsed time measures more than your job in a multi-tasking system. So matters are more complicated than one might think at first. Part of the elapsed time is cpu time spent on the program and part is cpu time spent on the operating system.
Time spent on a user program can be estimated as follows.

```
seconds        avg # of clock cycles   uI executed        seconds
----------- *  --------------------- * -------------  =  -----------
clock cycle    instruction executed      program          program
```

Example:

$$50 \;\; 10^{-y} \;\; 6 \;\; 10^{5} \;=\; 300 \;\; 10^{-4} \;=\; 30 \frac{milliseconds}{program}$$

Please note that instructions executed by a running program are NOT the number of instructions in the program. Think about loops and branches. Also note the cpu time spent in the user's program depends upon the data. Again think about time spent in loops and branches. Try to calculate the average number of clock cycles and one may reasonably conclude time spent is hard to pin down. As a practical matter you run the same program with the same data on various computers and compare results.
There are alternatives to execution time known as MIPS (million instructions per second) and MFLOPS (million floating point operations per second).
This subject so complex that entire books have been written on it. We refer you to the literature because many aspects of performance are related to specifying the machine.