3 THE STRUCTURE OF COMPUTERS

To design programs executable by automatic computer, the programmer must first know his tool. The more precisely he knows his processor, the better he is able to tailor his algorithms into programs utilizing the particular capabilities of that processor. On the other hand, the more an algorithm is tailored and "tuned" to a processor, the larger the effort spent to develop the program. Under normal circumstances, a solution must be found that keeps the program-development effort within reasonable limits while still yielding sufficiently good (i.e., efficient) programs. To find such a solution, the programmer must know the kinds of adaptations that are fairly easy to perform but that, at the same time, yield a relatively large improvement. To this end, it is essential to know the most important, generally valid characteristics of computers while ignoring the idiosyncrasies and peculiarities (called features) of individual machines.

In all modern digital computers, we can distinguish between two main components.

1. The *store* (often called *memory*). The store contains the objects that are manipulated in encoded form. These encoded objects are called *data*. The performance of a store is measured by its capacity (size) and by the speed with which data can be deposited and retrieved. In any case, the store has a *finite* capacity.
2. The *processor* (arithmetic–logical unit). This unit performs additions, multiplications, comparisons, etc. Data are retrieved (read) from the store for processing, and results are deposited (written) into the store.

   At each moment, the processor contains only the data to be processed immediately—that is, very few operands. Its own storage elements are called *registers*. All data that are not immediately needed are handed over to the store, which then plays the role of an "ice box."

Example: Evaluation of an expression

In order to evaluate an arithmetic expression with several operands and intermediate results, we apply again the technique of decomposing a complicated task into a sequence of simpler tasks. This causes individual arithmetic operations to take operands from the processor's registers and to replace them by the results. The evaluation of the expression

$$a * b + c * d \tag{3.1.1}$$

is broken down into simpler *instructions*:

$$\begin{aligned} &\mathrm{R1} := a \\ &\mathrm{R2} := b \\ &\mathrm{R1} := \mathrm{R1} * \mathrm{R2} \\ &z \quad := \mathrm{R1} \\ &\mathrm{R1} := c \\ &\mathrm{R2} := d \\ &\mathrm{R1} := \mathrm{R1} * \mathrm{R2} \\ &\mathrm{R2} := z \\ &\mathrm{R1} := \mathrm{R1} + \mathrm{R2} \end{aligned} \tag{3.1.2}$$

R1 and R2 denote the processor's registers, and z designates the intermediate result temporarily deposited in the store. The final result is made available in register R1.

The evaluation of the expression has thus been transformed into a short program consisting of three kinds of instructions or statements;

(a) instructions fetching operands from the store,
(b) (arithmetic) operations exclusively accessing processor registers, and
(c) instructions depositing results in the store.

This method of decomposing statements into more elementary steps and then temporarily saving intermediate results in the store is the reason that the same computational processes can be executed by relatively simple as well as very sophisticated computers—the former merely require more time. The decomposition method is the very essence of digital computer programming, and it is the basis for the application of relatively simple mechanisms to problems of enormous complexity. A precondition for the success of a computation consisting of billions of single steps (whose operands are always the result of previous steps) is, of course, a processor with a reasonably high speed and an absolute reliability. The realization of such processors is one of the true triumphs of modern technology.

The example of the evaluation of an expression also shows the necessity of a close interconnection between processor and store, since the amount of

information flow between the two units is rather high. The store contains objects that must be accessible through distinct names (e.g., *a*, *b*, *z*, ...). Consequently, there must be an order in the store like that found in a set of post office boxes. The objects are therefore contained in a set of uniquely identifiable *storage cells*, each of which has a unique *address*. Each access to the store must be accompanied by a specification of the address of the cell to be referenced.

Cells in a computer store resemble storage boxes used in daily life insofar as they contain and preserve an object. But this is where the analogy ends. The ability of computer stores to preserve data is not based on the fact that they physically harbor an object, but instead that a certain *encoding* of the object is reflected by the state of the cell. The cell must therefore be capable of assuming a certain number of *discrete states*. It is difficult to realize components capable of assuming and maintaining many clearly distinguishable states over an arbitrarily long time. It is feasible, however, to build such storage elements having only two distinct states; these are called *binary* storage elements. If a group of n binary storage cells is combined, this group can assume 2^n different combinations of states. If the group is considered as an indivisible unit, then it represents a storage cell with 2^n possible states.

Example: *Encoding objects into groups of binary digits*

We choose the positional representation of natural numbers (including 0). A number x is encoded in the following sequence of n *b*inary digi*ts* (called *bits*), that is, zeroes and ones,

$$x : b_{n-1} \ldots b_1 b_0 \tag{3.2.1}$$

where the encoding rule is given by

$$x = b_0 + 2b_1 + \cdots + 2^{n-1} b_{n-1} \tag{3.2.2}$$

This rule is by no means the only one possible, but in many respects, it is the most appropriate. After all, it is the same rule on which the representation of arabic (decimal) numbers is based; that is,

$$x : d_{m-1} \ldots d_1 d_0 \tag{3.3.1}$$

and

$$x = d_0 + 10 * d_1 + 10^2 * d_2 + \cdots + 10^{m-1} * d_{m-1} \tag{3.3.2}$$

Some examples of binary and decimal encodings (representations) of numbers are

Binary	Decimal
1101	13
10101	21
111111	63
1101011	107

(3.4)

The most important lesson to be learned from this example is that finite storage cells—that is, cells able to assume only a finite number of discernible states (no others exist in reality)—are capable of storing numbers only from a *finite range of values*. In a computer the number of binary storage elements grouped into a single addressable storage cell (word) is usually called the *wordlength*. The capabilities of the arithmetic unit are adapted to this measure. Common values of wordlengths n are 8, 16, 24, 32, 48 and 64 with corresponding sets of 2^n distinct values.

The consequence of the basic requirement that the computer must be able to obey a given program is that it must have "easy" access to that program. Where, then, is the most appropriate place to hold a program? It was the brilliant—and nowadays seemingly trivial—idea of *John von Neumann* to put the program into the store. Consequently, the same store is used to hold both the objects and the "recipe" of the computing processes, that is, the data *and* the program.

Obviously, this concept of the *stored-program computer* requires that instructions also be encoded. In our example of an expression evaluation every instruction is representable by an *operation code* (to specify reading, writing, adding, multiplying, etc.) and, in some cases, by an operand. If operands are represented by storage-cell addresses and if these addresses are chosen to be the whole numbers 0, 1, 2, . . . , then the problem of encoding programs is essentially solved; every program can be represented by sequences of numbers (or groups of numbers) and can therefore be deposited in a computer's store.

Another consequence of the stored-program approach is that every program will occupy a certain number of storage cells, that is, a certain amount of *storage space*. The number of occupied cells, which are no longer available to hold data, is proportional to the length of the program text. The programmer must therefore aim to keep his programs as concise as possible.

The following, important capabilities of the modern digital computer are based on the concept of sharing the store between the program and the data.

1. As soon as execution of a certain program P is terminated, a new program Q can be accepted in the store for subsequent execution (flexibility, wide applicability).
2. A computer may generate (according to a certain program) a sequence of numbers that it will subsequently consider and interpret as encoded instructions. Data generated in the first step become the program obeyed in the second step.
3. A computer X may be instructed to consider sequences of numbers actually representing programs as data to be transformed (according to some translation program) into sequences of numbers representing programs encoded for a different computer Y.