2.1.2 Instruction Execution

The CPU executes each instruction in a series of small steps. Roughly speaking, the steps are as follows:

1. Fetch the next instruction from memory into the instruction register.
2. Change the program counter to point to the following instruction.
3. Determine the type of instruction just fetched.
4. If the instruction uses a word in memory, determine where it is.
5. Fetch the word, if needed, into a CPU register.
6. Execute the instruction.
7. Go to step 1 to begin executing the following instruction.

This sequence of steps is frequently referred to as the **fetch-decode-execute** cycle. It is central to the operation of all computers.

This description of how a CPU works closely resembles a program written in English. Figure 2-3 shows this informal program rewritten as a Java method (i.e., procedure) called *interpret*. The machine being interpreted has two registers visible to user programs: the program counter (PC), for keeping track of the address of the next instruction to be fetched, and the accumulator (AC), for accumulating arithmetic results. It also has internal registers for holding the current instruction during its execution (instr), the type of the current instruction (instr type), the address of the instruction's operand (data loc), and the current operand itself (data). Instructions are assumed to contain a single memory address. The memory location addressed contains the operand, for example, the data item to add to the accumulator.

The very fact that it is possible to write a program that can imitate the function of a CPU shows that a program need not be executed by a ''hardware'' CPU consisting of a box full of electronics. Instead, a program can be carried out by having another program fetch, examine, and execute its instructions. A program (such as the one in Fig. 2-3) that fetches, examines, and executes the instructions of another program is called an **interpreter**, as mentioned in Chap. 1.

This equivalence between hardware processors and interpreters has important implications for computer organization and the design of computer systems. After having specified the machine language, *L*, for a new computer, the design team can decide whether they want to build a hardware processor to execute programs in *L* directly or whether they want to write an interpreter to interpret programs in *L* instead. If they choose to write an interpreter, they must also provide some hardware machine to run the interpreter. Certain hybrid constructions are also possible, with some hardware execution as well as some software interpretation.

An interpreter breaks the instructions of its target machine into small steps. As a consequence, the machine on which the interpreter runs can be much simpler and less expensive than a hardware processor for the target machine would be.

This saving is especially significant if the target machine has a large number of instructions and they are fairly complicated, with many options. The saving comes essentially from the fact that hardware is being replaced by software (the interpreter) and it costs more to replicate hardware than software.

Early computers had small, simple sets of instructions. But the quest for more powerful computers led, among other things, to more powerful individual instructions. Very early on, it was discovered that more complex instructions often led to faster program execution even though individual instructions might take longer to execute. A floating-point instruction is an example of a more complex instruction. Direct support for accessing array elements is another. Sometimes it was as simple as observing that the same two instructions often occurred consecutively, so a single instruction could accomplish the work of both.

The more complex instructions were better because the execution of individual operations could sometimes be overlapped or otherwise executed in parallel using different hardware. For expensive, high-performance computers, the cost of this extra hardware could be readily justified. Thus expensive, high-performance computers came to have many more instructions than lower-cost ones. However, instruction compatibility requirements and the rising cost of software development created the need to implement complex instructions even on low-end computers where cost was more important than speed.

By the late 1950s, IBM (then the dominant computer company) had recognized that supporting a single family of machines, all of which executed the same instructions, had many advantages, both for IBM and for its customers. IBM introduced the term **architecture** to describe this level of compatibility. A new family of computers would have one architecture but many different implementations that could all execute the same program, differing only in price and speed. But how to build a low-cost computer that could execute all the complicated instructions of high-performance, expensive machines?

The answer lay in interpretation. This technique, first suggested by Maurice Wilkes (1951), permitted the design of simple, lower-cost computers that could nevertheless execute a large number of instructions. The result was the IBM System/360 architecture, a compatible family of computers, spanning nearly two orders of magnitude, in both price and capability. A direct hardware (i.e., not inter-

preted) implementation was used only on the most expensive models. Simple computers with interpreted instructions also some had other benefits. Among the most important were

1. The ability to fix incorrectly implemented instructions in the field, or even make up for design deficiencies in the basic hardware.

2. The opportunity to add new instructions at minimal cost, even after delivery of the machine.

3. Structured design that permitted efficient development, testing, and documenting of complex instructions.