3 The software is in the holes

> Computer programs are the most complicated things that humans have ever created.
> Donald Knuth[1]

Software and hardware

Butler Lampson, recipient of the Turing Award for his contributions to computer science, relates the followi ng anecdote about software:

> There's a story about some people who were writing the software for an early avionics computer. One day they get a visit from the weight control officer, who is responsible for the total weight of the plane.
>
> "You're building software?"
>
> "Yes."
>
> "How much does it weigh?"
>
> "It doesn't weigh anything."
>
> "Come on, you can't fool me. They all say that."
>
> "No, it really doesn't weigh anything."
>
> After half an hour of back and forth he gives it up. But two days later he comes back and says, "I've got you guys pinned to the wall. I came in last night, and the janitor showed me where you keep your software."
>
> He opens a closet door, and there are boxes and boxes of punch cards.
>
> "You can't tell me those don't weigh anything!"
>
> After a short pause, they explain to him, very gently, that the software is in the holes.[2]

Fig. 3.1. Carving holes on a punch card. The software in the early days of computing was represented by these holes but the cards were actually punched by machines.

It is amazing how these holes (Fig. 3.1) have become a multibillion-dollar business and arguably one of the main driving forces of modern civilization.

In the previous chapter we described the idea of separating the physical hardware from the software. Identifying these two entities is one of the key concepts of computer science. This is why we were able to go "down" into the hardware layers and see how the arithmetic and logical operations could be implemented without worrying about the software. Now let's go "up" into the software levels and find out how to tell the computer what to do. Software, also called a *program*, is a sequence of instructions which "give orders" to the hardware to carry out a specific task. In this program we can use only the operations

that the hardware can "understand" and execute. These core operations are called the "instruction set." In terms of implementation the simplest instructions are hard-wired in the processor's circuitry and more complex ones are constructed from the core instructions as we describe later in this chapter. For the program to work the instructions must be arranged in the right order and expressed in an unambiguous way.

A computer's need for such exact directions contrasts with our everyday experience when we give instructions to people. When we ask a person to do something, we usually rely on some unspoken context, such as our previous knowledge of the person or the individual's familiarity with how things work, so that we do not need to specify exactly what we want to happen. We depend on the other person to fill in the gaps and to understand from experience how to deal with any ambiguities in our request. Computers have no such understanding or knowledge of any particular person, how things work, or how to figure out the solution of the problem. Although some computer scientists believe that computers may one day develop such "intelligence," the fact is that today we still need to specify exactly what we want the computer to do in mind-numbing detail.

When we write a program to do some specific task, we can assume that our hardware "understands" how to perform the elementary arithmetic and logic operations. Writing the precise sequence of operations required to complete a complicated task in terms of basic machine instructions at the level of logic gates would make for a very long program. For example, for numerical calculations we know we have an "add" operation in our basic set of instructions. However, if we have not told the machine how to carry out a multiply operation, it will not be able to do a simple calculation such as "multiply 42 by 3." Even though for us it is obvious that we can multiply 42 by 3 by using multiple additions, the computer cannot make this leap of imagination. In this sense, the computer is "stupid" – it is unable to figure out such things for itself. In compensation, however, our stupid computer can add far faster than we can!

If we are trying to solve a problem that involves a numerical calculation requiring several multiplications, it will obviously simplify our program if we introduced a separate "multiply" instruction for the computer. This instruction will just be a block of additions contained in a program. Now when we give the computer the instruction "multiply 42 by 3," the machine can recognize the word *multiply* and can start running this program. This ability to construct compound operations from simple ones and introduce a higher level of abstraction in this way is one of the fundamental principles of computer science. Moving up this ladder of abstraction saves us from having to write our programs using only the most elementary operations.

In the early days of computing, writing programs was not considered difficult. Even von Neumann thought that programming was a relatively simple task. We now know that writing correct programs is hard and error prone. Why did early computer scientists so badly underestimate the difficulty of this task? There are at least two possible explanations. The pioneers of computing were mostly male engineers or mathematicians who rarely spent time on the nitty-gritty details of actually writing programs. Like many men of the time, the early hardware pioneers thought that the actual coding of programs using binary

Fig. 3.2. A filing cabinet is a good analogy for thinking about the way a computer's hard disk stores information in files and folders.

numbers to represent the instructions and setting the appropriate switches on the machines was a secondary task, suitable for young women who needed only appropriate secretarial skills and some aptitude for mathematics. A more charitable explanation is that the difficulties of actually constructing the early computers and keeping them running were so overwhelming that these challenges completely overshadowed the problem of coding. Looking back, it now seems naive for the early computer builders to think that once they managed to get the machine up and running, the coding of a problem for the machine would be a relatively minor task!

There were some early warnings that programming might not be so simple. Maurice Wilkes recalls:

> As soon as we started programming, we found out to our surprise that it was not as easy to get programs right as we had thought. I can remember the exact instant when I realised that a large part of my life from then on was going to be spent in finding mistakes in my own programs.[3]

In a similar vein, one of the pioneers of computing, Edsger Dijkstra of the Netherlands, suggests in his autobiography that programming is even harder than theoretical physics.

> In 1955 I took the decision not to become a theoretical physicist, but to become a programmer instead. I took that decision because I had concluded that of theoretical physics and programming, programming embodied the greater intellectual challenge. You see, in those days I did not suffer from intellectual modesty.[4]

Software now surrounds us like the air that we breathe. It runs the communication networks, the power grid, our PCs, and our smart phones. It is embedded in cars, aircraft, and buildings, and in banks and national defense systems. We use software all the time, even if we are not aware of it. When we drive a car, pay bills, use a phone, or listen to a CD, thousands of lines of code are executed. According to Bjarne Stroustrup, the Danish inventor of the C++ programming language, "Our civilization runs on software."[5] Let's take a closer look at how we write the software that controls such a large part of our lives.

The file clerk model

Computers do much more than just compute. Typically there will be one part of the machine where the computer does all the basic mathematical operations while the rest of the machine is dedicated to moving the digital data around in the form of electrical signals. In many ways, we can think of the operation of a computer as being like the work that file clerks once did in an office. The file clerk was given a job to do and then had to move back and forth to racks of filing cabinets, taking files out and putting them back, scribbling notes on pieces of paper, passing notes around, and so on, until the job was completed (see Fig. 3.2). It will be helpful to use this analogy of the computer as a file clerk as a starting point to explain some of the basic ideas of a computer's structure and organization. This model is a good way for us to understand the essential ideas of how a computer actually does what it does.

Suppose there is a big company that employs many salespeople to sell its products and stores a great deal of information about the sales force in a big filing system consisting of cards in cabinets. Let's say that our file clerk knows how to get information out of the filing system. The data are stored in the filing cabinets on "Sales" cards, which contain the name, location, number of sales, salary, and other information about each salesperson. Suppose we want the answer to a simple question: "What are the total sales in the state of Washington?"

The instructions for the clerk could be:

Take out a "Sales" card.
If "Location" says *Washington*, then add the number of "Sales" to a running count called "Total."
Put the card back.
Take out the next "Sales" card and repeat.

This set of instructions looks fine, but what do we do if our file clerk does not know what is meant by keeping a "running count"? In this case we need to provide the clerk with more detailed instructions on exactly how to do that task. We therefore provide the clerk with a new card called "Total," and our more detailed "program" now reads:

Take out the next "Sales" card.
If "Location" says *Washington*, then take out the "Total" card.
Add the sales number to the number on the card.
Put the "Total" card back.
Put the "Sales" card back.
Take out the next "Sales" card and repeat.

In a modern computer, of course, the data would not be stored on cards and the machine does not physically take out a card. Instead, the computer reads the stored information from a *memory register*, a storage place in its main memory. Similarly, the computer can write from such a register to a "card" without actually physically taking out a card and putting it back.

To go any further with our analogy, we need to specify more precisely how our file clerk carries out the basic set of operations. One of the most elementary operations is that of transferring information from the cards that the clerk reads to some sort of scratch pad or working area where the clerk can do the arithmetic. We can do this by specifying exactly what our "Take" and "Replace" card operations mean:

"Take card X" means that the information on card X should be written on the scratch pad.
"Replace card Y" means that the information on the pad should be written on card Y.

We now need to instruct the clerk exactly how to check if the location on card X was *Washington*. The clerk will need to do this for each card, so he needs to remember *Washington* from one card to the next. One way to do this is to have *Washington* written on another card we shall call C. The instructions are now:

Take card X (from store to pad).
Take card C (from store to pad).
Compare what is on card X with what is on card C.
If the contents match, add the "Sales" to the "Total." If not, replace both cards and take the next one.

It would obviously be more efficient not to keep taking out and putting back card C, the *Washington* card. If there is enough room on the scratch pad, the clerk could store this information on the pad for the duration of the calculation. This is an example of a trade-off in the hardware design: the balance between the clerk having to shuffle cards in and out versus increasing the amount of storage space needed on the pad. We can keep breaking down the clerk's tasks into simpler ones until they correspond directly to basic operations that he knows how to carry out. For example, we need to tell the clerk precisely how to compare the information stored in "Location" and the name *Washington.*

Let us teach the file clerk how to use the scratch pad. The instructions can be divided into two groups. One group is a core set of simple procedures that come with the pad – add, transfer, and so on. In a computer, these instructions, called the *instruction set*, are the ones that have been implemented in the hardware: they do not change when we change the problem. They are like the clerk's intrinsic abilities. Then there is a set of instructions specific to the task at hand, such as calculating the total number of sales for the state of Washington. This set of specialized instructions is the "program." The program's instructions can be broken down into operations from the core set as we have seen. The program represents the detailed instructions about how to use the clerk's intrinsic abilities to do the specific job.

To get the right answer, the clerk must follow exactly the instructions that constitute the "program" in precisely the right order. We can ensure this by designating an area on the scratch pad to keep track of which steps have been completed. In a computer, this area is called a *program counter*. The program counter tells the clerk where he is in the list of instructions that constitute the program. As far as the clerk is concerned, this number is just an "address" that tells him where to look for the card with the instruction about what to do next. The clerk goes and gets this instruction and stores it on the pad. In a computer, this storage area is called the *instruction register*. Before carrying out the instruction, the clerk prepares for the next one by adding one to the number in the program counter. The clerk will also need some temporary storage areas on the pad to do the arithmetic, save intermediate values, and so on. In a computer, these storage areas are called *registers*. Even if you are only adding two numbers, you need to remember the first while you fetch the second. Everything must be done in the correct sequence and the registers allow us to organize things so that we achieve this goal.

Suppose our computer has four registers – A, B, and X plus a special register C that we use to store any number that we need to carry (put into another column) as part of the result of adding two numbers. Let us now look at a possible set of core instructions, the *instruction set*, for the part of a computer corresponding to the scratch pad. These instructions are the basic ones that will be built into the computer hardware. The first kind of instruction concerns the transfer of data from one place to another. For example, suppose we have

a memory location called M on the pad. We need an instruction that transfers the contents of register A or B into M or that moves the contents of M into register A or B. We will also need to be able to manipulate the program counter so we can keep track of the current number in register X. We therefore need an operation that can change this stored number as well as a "clear" instruction so that we can wipe out what was in a register and set it to zero. Then there are the instructions for the basic arithmetic operations, such as "add." These instructions will allow us to add the contents of register B to the contents of register A and update the contents of register A with the sum A + B. We also need the logical operations: the logic gates AND and OR that allow the computer to make decisions depending on the input to these gates. This capability is important because it enables the computer to follow different branches of a program depending on the result of the logical operation. We therefore need to add another class of instructions that enable the computer to "jump" to a specific location. This instruction is called a *conditional jump*, an action in which the computer jumps to the new location in the program only if a certain condition is satisfied. This conditional jump instruction allows the machine to leap from one part of a program to another. And finally we will need a command to tell the computer when to stop, so we should add a "Halt" instruction to our list.

These elementary instructions now enable us to get the computer to do many different types of calculations. The machine is able to perform complex operations by breaking them down into the basic operations it understands. In the example above, our computer had only four registers. In modern computers, although the underlying concepts are exactly the same, a larger set of basic instructions and registers is typically built into the hardware. The lesson we should take from the file clerk analogy is this. As long as our file clerk knows how to move data in and out of registers and can follow a sequence of simple instructions using a scratch pad, he can accomplish many different complex tasks. Similarly, a computer does all of these tasks completely mindlessly, just following these very basic instructions, but the important thing is that it can do the work very quickly. As Richard Feynman (B.3.1) says, "The inside of a computer is as dumb as hell but it goes like mad!"[6] A computer can perform many millions of simple operations a second, and multiply two numbers far faster than any human. However, it is important to remember that, at its heart, a computer is just like a very fast, dumb file clerk. It is only because a computer can do the basic operations so quickly that we do not realize that it is in fact doing things very stupidly.

B.3.1. Richard Feynman (1918–88) lecturing on computing at the workshop on Idiosyncratic Thinking at the Esalen Institute. In this lecture he explained in simple terms how computers work and what they are capable of. According to Feynman calling computers *data handlers* would be more fitting because they actually spend most of the time accessing and moving data around rather than doing calculations.

Maurice Wilkes and the beginning of software development

Let us now look at how the "file clerk model" is actually implemented in real computers. Computers work by allowing small electric charges to flow from the memory to collections of logic gates or to other memory locations. These flows are all regulated by von Neumann's fetch-execute cycle: at each step, the computer reads an instruction from memory and performs the specified action. The section of the computer that does the actual computing is called the *processor* or the *central processing unit* (CPU). The processor has a

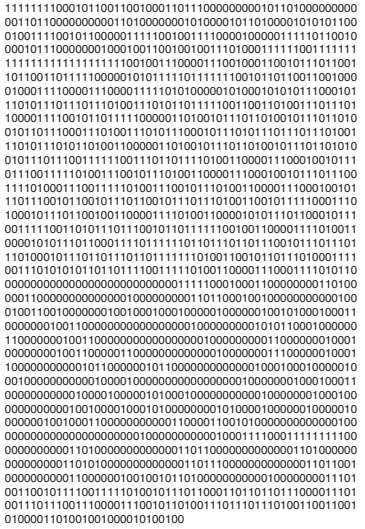


Fig. 3.3. This illustration shows the binary code of the "Hello world" program generated by the GNU CC compiler for a modern Intel microprocessor.

number of holding places for data known as *registers* that may be used to keep an instruction, a storage address, or any other kind of data. As in the case of our file clerk, some instructions tell the computer to copy words from memory to the central processor's registers. Other instructions perform some numerical or logical operation on this data and place the result in one of the processor's registers. The instructions accomplish this by opening control gates that allow the data to flow from location to location within the computer. These control gates are on the paths between the memory storage cells and the logic gates. Between each register and each specific processing element are data paths with gates activated by each instruction. In the architecture of most processors and in their associated instruction sets, this linking of instructions to specific data paths is created using a structure of branching paths called a *decision tree*. Data for arithmetic operations are sent one way, while data for logical operations are sent another way.

In an actual instruction, branch decisions are represented by 1s and 0s in different places in the instruction (see Fig. 3.3). For example, we might choose the first bit to be 1 for an arithmetic operation and 0 for a logical operation. The second bit could be 1 for an ADD in the arithmetic branch or for an AND gate in the logic branch. If we have an eight-bit instruction set, the remaining six bits could be used to specify the three-bit addresses of two registers. Thus the instruction word *11010001* could mean "Use the arithmetic unit to add the number in register 010 to the number in register 001." Modern instruction sets are more complicated but follow similar principles. For the first computers, programmers had to carefully think through how the desired sequence of operations translated into the actual setting of the switches. The use of *machine code* – the representation of instructions that were implemented in the computer hardware as binary numbers – was the first step in liberating the programmer from needing to know all the details of the hardware architecture. Writing programs in machine code is the first step up the software abstraction layers.

A stored-program computer is one that keeps its instructions, as well as its data, in its memory. Because it was Maurice Wilkes and the EDSAC who first proved the feasibility of such an architecture, it is not surprising that many of the early ideas of programming and software development came from the Cambridge team. The EDSAC ran its first program on 6 May 1949. The program, which was the calculation and printing of a table of squares from 0 to 99, ran for two minutes and thirty-five seconds. Within a year, the EDSAC was providing a programming service for the entire university. However, as Wilkes had discovered, programming correctly in machine code was difficult. He therefore introduced a more programmer-friendly notation for the machine instructions. For example, the EDSAC instruction "Add the short number in memory location 25" was stored as the binary string:

11100000000110010

This was abbreviated as:

A 25 S

where A stood for "add," 25 was the decimal address of the memory location, and S indicated that a "short" number was to be used. David Wheeler (B.3.2),

a graduate student in Cambridge, therefore introduced a program he called "Initial Orders" that read these more intuitive shorthand terms for the basic machine instructions, translated them into binary, and loaded them into memory ready for execution. Using these abbreviations made programs much more understandable and made it possible for users who were not computer specialists to begin to develop programs for the machine.

Wheeler's Initial Orders program was really the first practical realization of von Neumann's hardware-software interface. It was also the first example of an *assembly language*, a programming language that used names instead of numbers. Assembly languages constituted the next significant step up the hierarchy of software abstractions. They were easier to use than machine code but still much harder than later high-level languages. Writing an assembly language instruction such as "MOV R1 R2," meaning "Move contents of register 1 to register 2," allowed the user to avoid having to think about the explicit switches that needed to open to direct the flow of charge from register 1 to register 2. Similarly, *memory addresses*, the binary numbers that the computer used to track where data and instructions are stored in its memory, were replaced with more intuitive names like "sum" or "total." Although assembly languages made the writing of programs easier, computers still ran with machine code, the low-level language composed of binary numbers. To run an assembly language program, the assembly language first has to be translated into machine code. The need for such translation required the development of programs called *assemblers* that could perform the translation and produce machine code as output. On the EDSAC, this translation was originally done with Wheeler's Initial Orders program, which we might now call the first assembler.

B.3.2. David John Wheeler (1927–2004) was a British computing pioneer who made a major contribution to the construction and programming of the EDSAC computer. With his colleagues Maurice Wilkes and Stanley Gill, Wheeler invented *subroutines*, the creation of reusable blocks of code. Wheeler also helped develop methods of *encryption*, which puts a message into a form that can be read only by the sender and the intended recipient. Wheeler, Wilkes, and Gill wrote the first programming textbook in 1951, *The Preparation of Programs for an Electronic Digital Computer.*

Despite the great simplification introduced by using assembly language to write programs, the Cambridge team quickly found that a large amount of time was being taken up in finding errors in programs. They therefore introduced the idea of a "library" of tested, debugged portions of programs that could be reused in other programs. These blocks of trusted code are now called *subroutines*. For these blocks of code to be used in different programs, we need to use the *Wheeler jump*. As we have seen in our file clerk discussion, the computer takes its next instruction from the memory location specified in the special memory register called the *program counter*. After each instruction is read, the contents of the program counter are increased by one to point to the next instruction in memory. With a jump instruction, the computer can copy the memory address corresponding to the beginning of the subroutine code into the program counter. The computer is no longer restricted to the next instruction but can jump to the starting address of the subroutine code. The program will then follow the instructions in the subroutine code incrementing the program counter from that entry point. Of course, to know where in the program the subroutine should return after it has completed its execution, the computer also needs to have saved the previous contents of the program counter in another memory location, which was an essential feature of the Wheeler jump.

If we want to call a subroutine from within another subroutine, we will clearly need to save the multiple return addresses. We cannot use just one special memory location because the return address of the first subroutine will be

Fig. 3.4. The information stored in a computer's memory is much like a stack of plates. Just as we can add or take plates only from the top of the stack, the last data added to memory must be the first removed.

overwritten by the return address of the second and so on. To overcome this difficulty, computer scientists introduced the idea of a group of sequential storage locations linked together to operate as a *memory stack* (Fig. 3.4). The memory stack functions much like a stack of dinner plates: plates can only be added or removed at the top of the stack. This is an example of a simple data structure that is useful in many applications. It is called a *LIFO stack*, which stands for Last In, First Out. This data structure is able to handle the storage of the return addresses of nested subroutines.

The conditional jump instruction introduces two powerful new concepts for programmers: *loops* and *branches*. With a conditional jump, the program only performs the jump if a certain condition is met; if not, the program continues down the original path. A loop is where a block of code is executed a specified number of times. This can be done by keeping a tally called a *loop count* that is increased by one on each pass through the code. At the end of each block of code, the loop count is tested to see if the specified number of repetitions has been carried out: if not, the program is sent back to the beginning of the loop. A branch is just what it says: the choice of what section of code to execute is made depending on the result of the condition.

In 1951, Maurice Wilkes, with David Wheeler and Stanley Gill, wrote up their experiences in teaching programming. Their book *The Preparation of Programs for an Electronic Digital Computer* was the first textbook on computer programming. Also in the same year, Wheeler was awarded the first PhD in computer science for his work with the EDSAC.

FORTRAN and COBOL: The story of John Backus and Grace Hopper

Although the computing fraternity was very much male-dominated in the early years, there were a few influential pioneers who were women. Probably the most famous is Grace Hopper, or Rear Admiral Professor Grace Hopper as she later became (B.3.3). Hopper received her PhD in mathematics from Yale University in 1934 and was teaching at Vassar College in Poughkeepsie, New York, when the United States entered World War II. She enlisted in the Naval Reserve in December 1943 and graduated at the top of her class in June 1944.

B.3.3. Grace Hopper (1906–92), an American computer scientist, led the team that developed COBOL, the first programming language for business that allowed programmers to use everyday words.

The Harvard Mark I of Howard Aiken had been commandeered for the war effort, and Aiken was now a Naval Reserve commander (B.3.4). He liked to say that he was the first naval officer in history who commanded a computer. Although Aiken's machine was not very influential on the future development of digital computers, Aiken was one of the first to recognize the importance of programming as a discipline. He persuaded Harvard to start the first master's degree courses in what would now be called computer science. In addition he insisted that the Mark I project be staffed with trained mathematicians. And this is how Lieutenant Grace Hopper of the U.S. Navy found herself being greeted by Aiken in the summer of 1944:

[Howard Aiken] waved his hand and said: "That's a computing machine." I said, "Yes, Sir." What else could I say? He said he would

B.3.4. Howard Aiken (1900–73), an American physicist and computer pioneer, proposed to IBM in 1937 the idea of constructing a large-scale electromechanical computer. By 1944, the computer, known as Mark I, became operational. The machine was fifty-five feet long, eight feet high, and about two feet wide.

> like to have me compute the coefficients of the arc tangent series, for Thursday. Again, what could I say? "Yes, Sir." I did not know what on earth was happening, but that was my meeting with Howard Hathaway Aiken.[7]

Hopper is credited with introducing the word *bug* into computer terminology after her team found a real insect, a moth, stuck in one of the machine's relays (Fig. 3.5). At the end of the war, Hopper chose to remain at Harvard, programming first the Mark I and then the Mark II machines. But in 1949 she defected from Aiken's camp and joined the opposition, the Eckert-Mauchly Computer Corporation in Philadelphia, which Aiken had dismissed as a foolish idea. This company was Eckert and Mauchly's brave effort to commercialize their EDVAC ideas by building the UNIVAC computer. At the time, there was great skepticism as to whether there was a significant market for business computers. Hopper later joked:

> Mauchly and Ekert had chosen their building perfectly. It was between a junk yard and a cemetery, so if it all went wrong … they could throw the UNIVAC out of one window and themselves out of the other.[8]

Following along the same lines as Wilkes and Wheeler in Cambridge, Hopper and her team began to code their programs in UNIVAC assembly language. In doing this, she produced a program that would translate the assembly code into the binary machine code automatically, as Wheeler had done with his Initial Orders program. However, Hopper had much larger ambitions than using just primitive abbreviations for the low-level operations of assembly language. She began to investigate whether it was possible to write a program using expressions that more closely resembled everyday English. Initially focusing on creating a more natural language for scientific computing, Hopper experimented with a language called A-0. She introduced the term *compiler* for the software system she created to translate

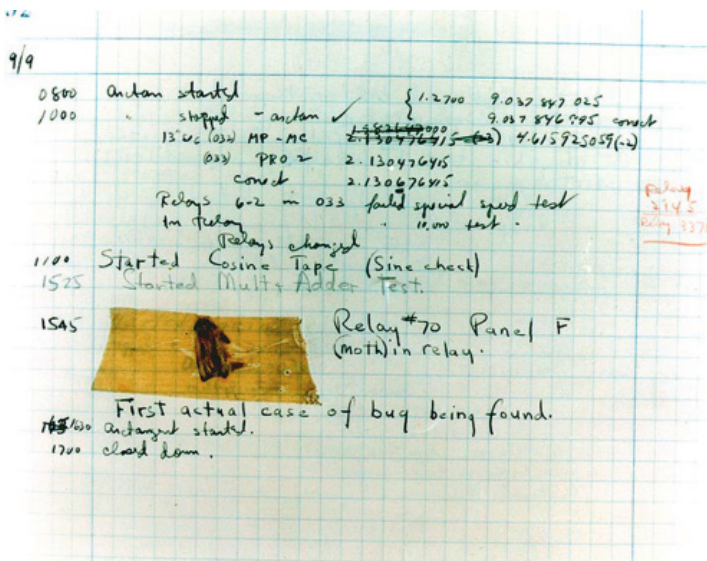


Fig. 3.5. The first recorded appearance of a computer "bug" dates from 1947. Grace Hopper is credited with introducing the word into computer terminology after her team found a real insect, a moth, stuck in one of the relays of the Mark II computer. They removed the insect and taped it into the logbook.

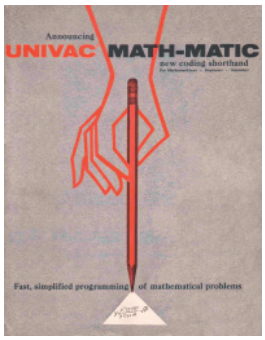


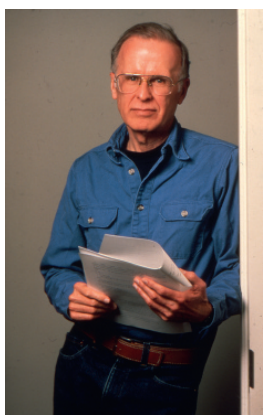
Fig. 3.6. In 1955, Hopper and her team released the MATH-MATIC language for the UNIVAC. MATH-MATIC was one of the first higher-level languages above assembly language to be developed.

programs written in A-0 to UNIVAC machine code. The results, she reported, were mixed:

> The A-0 compiler worked, but the results were far too inefficient to be commercially acceptable. Even quite simple programs would take as long as an hour to translate, and the resulting programs were woefully slow.[9]

However Hopper remained an energetic advocate for *automatic programming*, in which the computer generates the machine code from a program written in a "high-level" programming language. She and her team therefore persevered with the development of the A-0 language and its compiler. In 1955, the company released the MATH-MATIC language for the UNIVAC (Fig. 3.6), and a news release declared, "Automatic programming, tried and tested since 1950, eliminates communication with the computer in special code or language." These attempts in the early 1950s had shown that the outstanding problem in programming technology was to produce a compiler that could generate programs as good as those written by an experienced assembly language or machine code programmer. Enter John Backus and IBM (B.3.5).

IBM introduced its 704 computer for scientific applications in 1954. A major advance in the architecture of the 704 was that the hardware included dedicated circuits to perform the operations needed to handle *floating-point numbers* – numbers containing fractions in which the decimal point is moved to a standard position in order to simplify the hardware required to manipulate such fractional numbers – and not just integers (whole numbers). Now programmers could add, subtract, multiply, and divide real numbers as easily as performing these same operations with integers, without having to call on complex subroutines to do these operations. Backus had been developing an assembly language for another IBM computer, but in late 1953 he sent a proposal to his manager suggesting the development of what he called a "higher level language" and compiler for the IBM 704. It is interesting that Backus made the case for such a language mainly on economic grounds, arguing that "programming and debugging accounted for as much as three-quarters of the cost of operating a computer; and obviously as computers got cheaper, this situation would get worse."[10]



B.3.5. John Backus (1924–2007), a computer scientist at IBM, developed FORTRAN, the first programming language that enabled scientists and engineers to write their own programs.

The project was approved, and the FORTRAN – for FORmula TRANslation – project began in early 1954. Producing code that was nearly as good as that written by an experienced machine code programmer was always the overriding goal of Backus's team:

> We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient [binary] programs. Of course one of our goals was to design a language which would make it possible for engineers and scientists to write programs for the 704. We also wanted to eliminate a lot of the bookkeeping and detailed repetitive planning which hand coding [in assembly language] involved.[11]

In April 1957, the language and the compiler were finished. The compiler consisted of about twenty thousand lines of machine code and had taken a team of about a dozen programmers more than two years to produce.

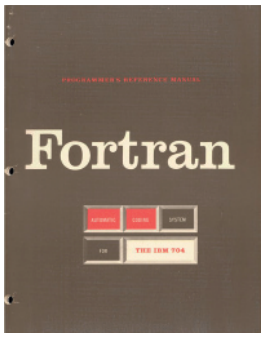


Fig. 3.7. The first FORTRAN book from IBM.

Fig. 3.8. FLOW-MATIC, developed by Grace Hopper in 1956, was the first programming language that allowed users to describe operations in English-like expressions.

Fig. 3.9. COBOL, one of the first programming languages, is still alive and well. Much business transaction software is written in COBOL and there are about 200 billion lines of code in use. Ninety percent of financial transactions are written in COBOL.

A typical statement in FORTRAN looks very like the underlying mathematical equation. Thus

$$y = e^x - \sin x + x^2$$

became

$$y = \mathrm{EXPF}(X) - \mathrm{SINF}(X) + X**2$$

Because of this simplicity and how closely it resembles the language of mathematics, FORTRAN rapidly became the dominant language for scientific computing. Backus's team had come very close to meeting their design goal:

> In practice the FORTRAN system produced programs that were 90% as good as those written by hand, as measured by the memory they occupied or the time they took to run. It was a phenomenal aid to the productivity of a programmer. Programs that had taken days or weeks to write and get working could now be completed in hours or days.[12]

FORTRAN also produced another great benefit – the portability of programs across different machines. Although the first compiler was written for the IBM 704, very soon there were FORTRAN compilers for other IBM computers. Competing computer manufacturers also soon produced FORTRAN compilers for their machines. For the first time there were computers capable of speaking the same language so that programmers did not have to learn a new language for every new computer.

In 1961, Daniel McCracken published the first FORTRAN programming textbook for use in undergraduate courses in universities. In 1966, FORTRAN became the first programming language to be formally standardized by ANSI, the American National Standards Institute, the organization that creates standards for the U.S. computer industry (Fig. 3.7). The FORTRAN language, now written *Fortran* with only one capital letter, has evolved with time to incorporate new structures and technologies from research on programming languages by computer scientists. It is nevertheless surprising that Fortran programs are still much used in scientific computing more than fifty years after the first introduction of the language.

The other major breakthrough in early computer programming was a language for business applications. After her work on MATH-MATIC, Hopper turned to the problem of making business programming – the tasks needed to run a business such as managing accounting and inventory – easier and more intelligible to that community. By the end of 1956, she had produced a compiler for FLOW-MATIC, a language that contained around twenty English-like expressions and allowed the use of long character names (Fig. 3.8). For example, to test whether the value of variable A is greater than that of variable B, in Fortran we would write:

```
IF A.GT. B
```

By contrast, in a language like FLOW-MATIC, one would write a similar comparison test as:

Fig. 3.10. Early computers used punched cards to input programs and data.

IF EMPLOYEE-HOURS IS GREATER THAN MAXIMUM

The language not only made the programs more intelligible to managers but also provided a form of self-documentation that describes what the program is supposed to do.

In May 1959 the U.S. Department of Defense started an initiative to develop a common business language. This led to the COBOL (Fig. 3.9) programming language – for COmmon Business-Oriented Language – which was strongly influenced by Hopper's earlier FLOW-MATIC language. For this reason, Hopper is sometimes known as "the mother of COBOL." What made the language so successful was a declaration by the U.S. government a year later that it would not lease or purchase any new computer without a COBOL compiler. At the end of 1966, Hopper retired from the Navy with the rank of commander. Less than a year later she was recalled to active duty and tasked with the job of rewriting the Navy's payroll system in COBOL. She was promoted to rear admiral in 1985.

For the next twenty years, from about 1960 to about 1980, FORTRAN and COBOL accounted for approximately 90 percent of all applications programs. Backus went on to develop a notation to capture the "grammar" of a programming language – that is, the way in which the special words and concepts of a language can be put together. A Danish computer scientist, Peter Naur, then simplified Backus's notation so that the grammar of any language could be captured in what is now known as Backus-Naur Form or BNF (B.3.6). In the 1970s Bell Labs produced a *compiler-compiler*, a program that could transform a BNF specification into a compiler for that language. There has been much research and experimentation with programming in the fifty years since FORTRAN and COBOL. We will look at some of these developments in the next chapter.

Early operating systems

B.3.6. Peter Naur, a Danish computer scientist, helped develop a successful programming language called Algol 60. In 2005, Naur received the Turing Award for his contributions to computer science.

In using even these early machines, it clearly made no sense for each user to have to figure out independently how to interact with the computer. Originally, users might input their programs and data – send instructions and information to the computer – using a punched card or paper tape reader. Later the input process might involve a keyboard, mouse, or, nowadays, a touch-enabled tablet. Each user could also use disk drives to access and store data, and could read off the results from a printer or some form of screen display. So although the earliest computers had no real *operating system* – that is, no software to control the operation of the entire computer system with anything like the sophistication we see today – it was still useful to collect together all the I/O subroutines – programs for input and output – and have them permanently loaded on the machine.

In the earliest days of computers, users had to book a time slot on the machine, so graduate students were naturally allocated the nighttime slots! Users loaded their programs into the machine using punched cards or paper tape and then waited while the computer ran their program (Fig. 3.10). This personalized system quickly evolved to a more efficient system in which the users were isolated from the machine by "operators." Users now had to give their program deck to the operator, who would load a batch of such programs

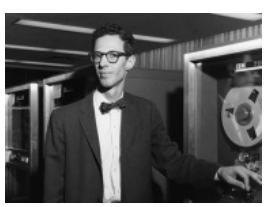
into the machine and then return the output to the users when the jobs were completed. The "operating system" was just the loading routine used by the operator to schedule the jobs on the computer plus the collection of I/O subroutines. As commercial computers began to appear in the early 1950s, such *batch processing* was the norm.

By the mid- to late 1950s, the limitations of batch processing were becoming apparent, and in universities there was a great deal of experimentation with the idea of more interactive computing. In 1955, John McCarthy (B.3.7), one of the pioneers of *artificial intelligence*, spent a summer at IBM's laboratory in Poughkeepsie and got to learn computer programming through batch processing on the IBM 704 computer. He was appalled at having to wait to learn whether or not his program had run correctly. He wanted the ability to debug the program interactively in "real time," before he had lost his train of thought. Because computers were very expensive systems at that time, McCarthy conceived of many users sharing the same computer at one time instead of just being allowed access to the machine sequentially, as in batch processing. For such sharing to be possible, multiple users had to be connected to the machine simultaneously and be assigned their own protected part of memory for their programs and data. Although there was only one CPU, each user would have the illusion that he or she had sole access to it. McCarthy's idea was that because the computer cycles from instruction to instruction very quickly on a human timescale, why not let the CPU switch from one memory area and program to another memory area and program every few cycles? This way the user would have the illusion that they are sole user of the machine. He called his concept *time sharing*:

B.3.7. John McCarthy (1927–2011) contributed many groundbreaking ideas to computing, such as time sharing, the LISP programming language, and artificial intelligence. In recognition of his pioneering work in computer science, he received the Turing Award in 1971.

> Time-sharing to me was one of these ideas that seemed quite inevitable. When I was first learning about computers, I [thought] that even if [time sharing] wasn't the way it was already done, surely it must be what everybody had in mind to do.[13]

Time sharing was not what IBM had in mind. It is perhaps understandable that IBM had little interest in time sharing and interactive computing, despite its longtime involvement in postwar projects with MIT, because all of its business customers were happy with their new batch-mode IBM computers. In order to implement time sharing, McCarthy needed IBM to make a modification to the hardware of the 704. This was needed for an "interrupt" system that would allow the machine to suspend one job and switch to another. Fortunately IBM had created such an interrupt modification for the Boeing Company to connect its 704 computer directly to data from wind-tunnel experiments. IBM allowed MIT to have free use of the package and in 1959 McCarthy was able to demonstrate an IBM 704 computer executing some of his own code between batch jobs. In his live demonstration, the time-sharing software was working fine until his program unexpectedly ran out of memory. The machine then printed out the error message:



B.3.8. Fernando Corbató pioneered the development of operating systems that allowed multitasking and time sharing. He stated a rule of computer science called Corbató's law, according to which "The number of lines of code a programmer can write in a fixed period of time is the same independent of the language used."

> THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS FOLLOWS …[14]

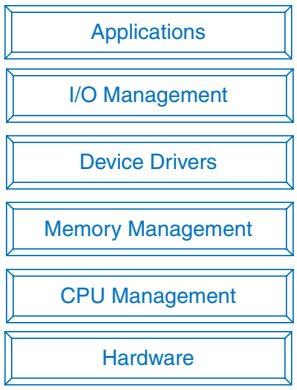


Fig. 3.11. A computer's operating system can be pictured in layers. The bottom layer consists of *hardware*, the mechanical, electronic, and electrical parts that make up the machine. Higher layers represent the main functions of the operating system, including the CPU that does the actual computing, management of the computer's memory, and device drivers that operate devices attached to the computer. Higher still is the I/O layer, which enables users to communicate with the computer. At the top are the applications that perform specific tasks, such as word processing and managing databases.

McCarthy's audience at MIT thought he had been the victim of a practical joke. In fact he was writing his programs in LISP (List Processing), a high-level language he had developed for programming artificial intelligence applications. For this language, he had introduced a *garbage collection* routine to reclaim parts of the memory that were no longer needed by the program. In effect, McCarthy's routine was an early attempt to build an automatic memory management system.

It was not until 1961 that Fernando Corbató (B.3.8) at the MIT Computation Center was able to demonstrate a fully working time-sharing system. This was called the Compatible Time-Sharing System or CTSS. This was the starting point for J. C. R. Licklider's famous Project MAC, a time-sharing system of which the goal was nothing less than what its proponents called "the democratization of computing." The MAC project (MAC could stand for Machine-Aided Cognition or Multiple Access Computer) and the Multics (Multiplexed Information and Computing Service) time-sharing operating system that developed from these beginnings were enormously influential and led to spin-off projects in many different areas. Most modern operating systems use an interrupt system to shift resources when and where they are needed, making multitasking possible.

The many roles of an operating system

Operating systems have progressed a long way from being a simple collection of subroutines and a batch loader to software systems of enormous complexity and power. We end this chapter by listing the major functions that a modern operating system must carry out.

Device drivers and interrupts

One of the earliest roles of the operating system was to allow users to interact with a wide variety of devices, such as keyboards, scanners, printers, disks, and mice without having to write their own code. The key to making this possible with all the multitude of different devices we have today is to hide all the intricate details of a particular device behind a standard piece of software called a *device driver*, a program that operates a particular type of device attached to the computer. The interface of a device driver with the computer needs to be carefully specified because many devices need to access specific memory locations. They also must generate and respond to control signals called *interrupts*, indications that some event happening in the computer needs immediate attention. Handling these interrupts is a key function of the operating system (Fig. 3.11).

Job scheduling

If one program has to wait for some input, another program could start running. The operating system must have the capability of sending a waiting program to "sleep" and then waking it up again with an interrupt when its input has arrived and it is ready to proceed. To do this, the operating system needs to maintain a table of "active processes," the operations that are under way. This list contains all the details for each process – where it is in main memory,

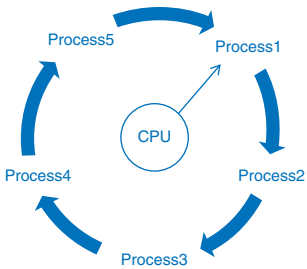


Fig. 3.12. Perhaps the most important task for an operating system is allocating time on the CPU to different processes. Each process is allowed to use the CPU for a limited time.

what the current contents of the CPU registers are, what addresses are in the program counter and the memory stack, and so on. When a process becomes active, the operating system loads all of this information into the CPU and restarts the program from where it left off. The operating system also needs to have some "scheduling policy" to decide which should be the next process to become active. There are many such scheduling policies that attempt to ensure "fair" process selection – but most users would argue that none of them are perfect!

Hardware interrupts

The next problem for the operating system is fundamental. Because the operating system is also a program and the CPU can only run one program at a time, how can the operating system hand the CPU over to one process and then get back control of the CPU so it can schedule another process? This is done by a different type of interrupt, one that switches not between a process and the operating system but between the actual computer hardware and the operating system. It is called a *hardware interrupt*. Such an interrupt happens when some event like a keyboard entry or mouse movement occurs. The hardware interrupt changes the computer's operation from *user mode* to *supervisor mode*. Supervisor mode is a method of operation in which the operating system can access and use all of the computer's hardware and instructions. By contrast, in user mode only a restricted set of hardware and instructions are accessible to the program. At a hardware interrupt, the computer jumps to the *scheduler* program. This software finds out what event has happened and decides which user process should next gain control of the CPU (Fig. 3.12).

System calls

In addition to hiding the complexity of devices through standard interfaces with device drivers (Fig. 3.13) and scheduling user processes, the operating system manages how programs request services from the hardware. When user programs need to access and control devices directly, the role of the operating system is to make sure they do so safely without causing damage to the hardware. The operating system ensures the safety of the entire computer system through a set of special-purpose functions called *system calls*. System calls are the means by which programs request a service from the operating system.

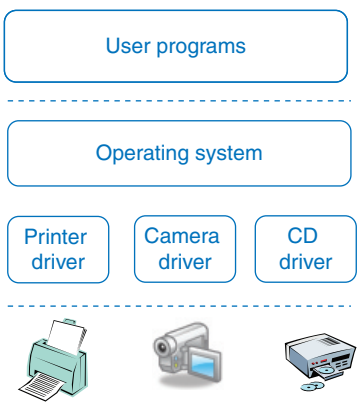


Fig. 3.13. A significant portion of the code of an operating system is made up of device drivers. These are programs that operate various devices attached to the computer, such as a printer, a camcorder, or a CD player.

File management

One special class of system calls has come to symbolize the entire operating system. These are the calls that create and manipulate the file system. Computers store data in a hierarchical arrangement of *files* or *folders* that are accessed through "directories" on a hard disk. The *hard disk*, housed in a unit called a hard drive consists of magnetic plates called *platters* that store information. The computer reads and writes data to the disk. The operating system has to keep track of the file names and be able to map them to their physical location on the disk.

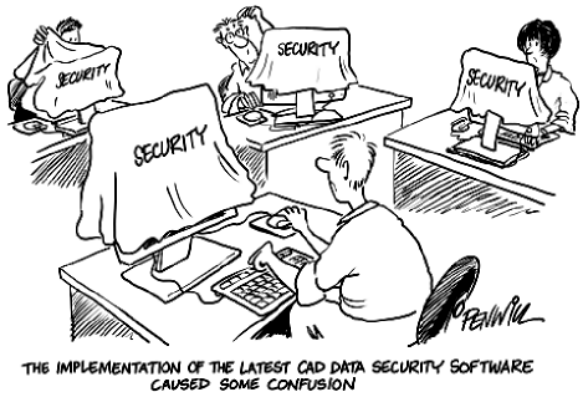


Fig. 3.14. With the widespread use of the Internet, online security is becoming one of the most important aspects of present-day computing.

Virtual memory

Besides managing the file store, where the files are kept on the hard disk, the operating system also manages the computer's main memory. Computer memory is expensive, and typically a computer has much less main memory than its address space can support. A computer's *address space* represents the range of numbers that can be used for addressing memory locations. For example, if the CPU registers are 32 bits wide, they can hold 2^{32} different bit patterns. This is the largest possible address space and is usually referred to as 4 gigabytes, because 2^{32} is 4,294,967,296, or just more than four billion (a thousand million). G stands for *giga*, a prefix that means one billion. Nowadays users can write programs without worrying about the limitations of main memory. Clever *virtual memory* software allows user programs to assume they can employ all of the addressable memory even though the main memory supports far fewer real addresses. The virtual memory creates this illusion by moving blocks of memory called "pages" back and forth between the hard disk and the main memory. This leads to a new type of interrupt called a *page fault*, which occurs when the page that the program needs is not yet in main memory. The operating system then must suspend the program so that the required page can be copied into main memory from the hard disk. To make room for this page in main memory, another page must be swapped out. Memory mismanagement is one of the most common causes of "crashes" and there are many elaborate strategies for deciding which page is best to move out.

Security

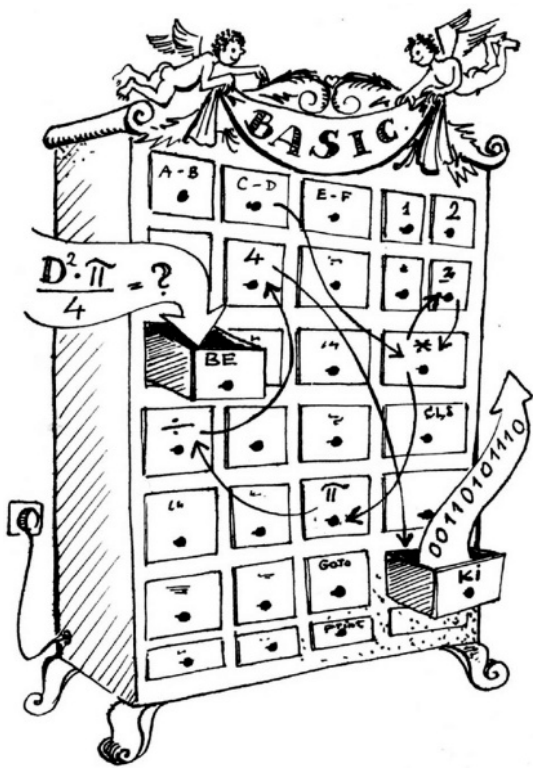
One vital function that an operating system must provide is security (Fig. 3.14). For each user the operating system must maintain the confidentiality and integrity of the information they have stored on the computer. A first step toward this goal is to identify permitted users by a password so that they must use this password to log into the computer before they are allowed access to any of its resources. The operating system must keep track of the users and passwords and ensure that only those files associated with an authorized user can be accessed and manipulated by that user. Alas there is now a thriving subculture of *hackers*, skilled programmers who try to subvert these security

measures. Suppliers of operating systems are locked into an escalating struggle to develop effective countermeasures to foil hackers.

We will return to the problem of hackers later in this book when we come to the creation of the Internet and the personal computer. In the next chapter we look at the continued development of programming languages and of the attempt to turn the business of writing programs into a real engineering discipline.

Key concepts

- Instruction set
- File clerk model of computer
- Machine code and assembly language
- Subroutines, loops, and branches
- FORTRAN and COBOL
- Operating system concepts
  - Batch processing and time sharing
  - Device drivers
  - Interrupts
  - System calls
  - Memory management
  - Security
- Microcode



Microcode

In 1951, at a ceremony to inaugurate the Manchester University computer, Maurice Wilkes argued that "the best way to design an automatic calculating machine"[15] was to build its control section as a stored-program computer of its own. Each control operation, such as the command to add two numbers, is broken down into a series of *micro-operations* that are stored in a *microprogram*. This method had the advantage of allowing the hardware design of the control unit to be simplified while still allowing flexibility in the choice of instruction set.

Fig. 3.15. The IBM System/360 was a family of general-purpose mainframe computers delivered in 1965. It was the first line of computers designed to be compatible with one another.

B.3.9. Thomas Watson Jr. (1914–93), then president of IBM, took an unprecedented gamble by putting huge resources into the IBM 360 project to unify IBM's many different computing systems. The gamble paid off and changed the history of computing. From 1979 to 1981, Watson served as the U.S. ambassador in Moscow.

The microcode approach turned out to be the key insight that enabled IBM to successfully implement its ambitious "360" project (Fig. 3.15). The 360 project was an attempt by IBM in the mid-1960s to make all IBM computers compatible with one another, simplifying what many people saw as the confusing tangle of the company's machines. Thomas Watson Jr. (B.3.9), then president of IBM, set up a group called the SPREAD (Systems Programming, Research Engineering and Development) Committee to investigate how this goal could be achieved. At one stage, two key architects of the System/360 family of mainframe computers, Fred Brooks and Eugene Amdahl, argued that it couldn't be done. However, an English engineer called John Fairclough (B.3.10), who was a member of SPREAD, had studied electrical engineering at Manchester and learned about the advantages of Wilkes's microprogramming and microcode. It was through him that IBM realized that microcode offered a solution to the problem of offering a common instruction set across the System/360 family of computers. Microcode also gave engineers the possibility of offering *backward compatibility*, which would enable a new computer to run the same software as previous versions of the machine. By installing microcode that implemented instructions written for programs developed for earlier machines, the older programs would still be able to run on the new 360 computer. Fairclough later became director of IBM's UK Development Laboratory at Hursley, near Winchester (Fig. 3.16).

Fig. 3.16. IBM Hursley Laboratories near Winchester, U.K. The Lab developed several IBM computers and much important software. The software produced by Hursley includes one of the best-selling software products of all-time, CICS, for transaction processing – the day-to-day transactions of banking, airline ticket systems, and so on.

B.3.10. John Fairclough (1930–2003) played an important role in the British computing industry. He was a member of the IBM 360 team and, in 1974, he became managing director of IBM Hursley Laboratory near Winchester in England. In the 1980s, Fairclough served as chief scientific adviser to the British government. He strongly supported close collaboration between universities and computer designers and manufacturers.