Each type of computer is distinguished by its own repertoire or list of instructions that the control unit is capable of interpreting and executing. The repertoire is generally fixed and is furnished as built-in circuitry by the computer designer or is stored in read-only memory. In some machines there are several hundred different instructions available in the repertoire.

A.6 DESIGN OF A MINIATURE COMPUTER

The principles developed in the foregoing sections can be better understood if we digress here to design a very small, simple computer and see how it might be operated to solve one or two simple problems. By "design" we mean that we will specify the main characteristics of the computer's key components.

The storage unit

First we shall design a small storage unit, as shown in Fig. A.3, which consists of 25 cells numbered from 01 to 25. Each cell will be able to hold a number represented by three decimal digits and a sign.

For simplicity we will assume that our computer will deal with integers that are limited to the range −999 to +999, so that any such number may be contained in a single word of storage. Similarly, we will assume that each instruction is represented by a three-digit code. An instruction can also be stored in a single word. The sign position of an instruction will always be plus.

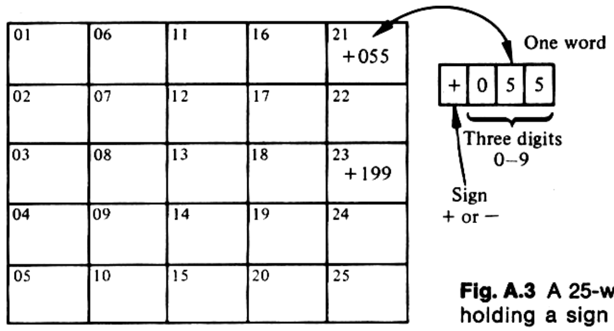

Fig. A.3 A 25-word storage unit with each word holding a sign and three decimal digits.

The order list

If we agree that the computer has an order list of no more than ten instructions, we can assign a one-digit code, 0 through 9, for each instruction. A possible set of simple orders is given in Table A.1. The order list will be better understood when we explain how the processing, control, and input-output units are to function.

Table A.1 A simple order list for a miniature computer

	Action	Symbolic abbreviation	Machine code
1. Arithmetic	Clear and add	CLA	1
	Add	ADD	2
	Subtract	SUB	3
	Store	STO	4
	Multiply	MPY	5
	Divide	DIV	6
2. Input-output	Read	RDS	7
	Print	PRT	8
3. Control	Transfer unconditionally	TRA	9
	Transfer on minus	TMI	0

The processing unit

At the heart of the processing unit is a device for holding the results of individual additions, subtractions, multiplications, or divisions. We call this device the *accumulator*. In our machine, the accumulator, often abbreviated as ACC, holds a sign and six digits. To see how the accumulator might be used, let us imagine that we wish to find the sum of the two numbers found in positions 21 and 23 of storage. The values found in these locations might be +055 and +199, as shown in Fig. A.3. The *clear and add* instruction, whose code is 1, may now be used to make the accumulator zero and then to add to the "cleared" accumulator the number found in any designated word of storage. To designate the contents of a particular word requires two more digits to identify the address of that word uniquely. For example,

```
1 21
```

might be made to mean: *clear* the accumulator to zero *and add* (to the accumulator) the contents of word 21. Then

```
2 23
```

might mean: *add* (to the accumulator) the *contents* of word 23.

In this way we have a concept of what an instruction for our computer should be. In our example, after the 1 21 instruction is executed, the ACC would contain +000055. Then, after the execution of the 2 23 instruction, the ACC would contain +000254. We are now ready to formalize this concept.

Instructions

The instruction code is represented as a three-digit number. The first digit is a code for the particular *operation,* or action, that we wish the computer to take; the last two digits form the *address* of a number that is to be involved in the action.

We continue with our example. Having formed the sum of the two numbers in the accumulator, we now wish to place it in some word of storage, say at address 25. In our example the accumulator now contains +000254. The code

```
4 25
```

means: *store* the sign and the three lowest-order (rightmost) digits of the accumulator† at address 25. If properly interpreted and executed, this instruction would accomplish our objective. In short, the sequence of instructions

```
1 21
2 23
4 25
```

can be thought of as a "program" to form the sum of two numbers found at addresses 21 and 23 and to store this sum at address 25.

Machine codes and a symbolic equivalent

These three-digit instructions are often called *machine codes* because they are the codes understood by the machine we are designing.

Our problem might have been initially stated as: "Form c as the sum $a + b$" or "Let $c = a + b$" or even more simply, "$c \leftarrow a + b$."

Let us imagine that we want the computer to perform this task. As a first step we might choose to describe the desired computer action in an "intermediate" language, say,

```
CLA A
ADD B
STO C
```

† A sum or product that extended beyond the third digit could not be stored in its entirety, using the *store* instruction. However, another type of store instruction might be added to the order list of Table A.1, which when executed would copy and store the three highest-order digits. Without this extension the computer, as we now describe it, is admittedly limited rather severely. Note that, although for practical purposes, no final arithmetic result can exceed three digits in size and be stored, an *intermediate* result of a computation might exceed three digits. Consider the computation $50 \times 50 - 45 \times 55 = 2500 - 2475 = 25$. The intermediate products have four digits, but the final result has only two digits.

These "symbolic" codes can then be transformed to the numerical machine codes, provided we are willing to identify certain storage addresses with particular variables. Thus, if we agree that A, B, and C are to correspond respectively, say, to addresses 21, 23, and 25, we can transform the intermediate symbolic code to machine code in a straightforward one-to-one correspondence. (Recall that there was a corresponding idea in the ideal computer. It involved choosing a container and labeling it.) Figure A.4 displays the process of translation from problem statement to machine code. Professional programmers are often forced to write instruction sequences in a symbolic equivalent of machine code. The translation to machine code is then accomplished automatically by the computer with the aid of specially written programs called *assembly programs.* Of course, our own miniature computer is probably not capable of performing this translation.

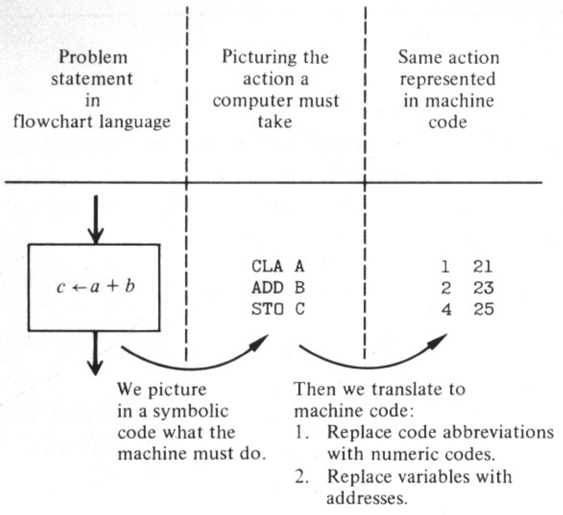
Problem statement in flowchart language
Picturing the action a computer must take
Same action represented in machine code
$c \leftarrow a + b$
CLA A
ADD B
STO C
1 21
2 23
4 25
We picture in a symbolic code what the machine must do.
Then we translate to machine code:
1. Replace code abbreviations with numeric codes.
2. Replace variables with addresses.

Figure A.4

The control unit

The control unit will have two registers, for interpreting and executing the desired sequence of instructions. Registers are much like storage cells in that they will each hold one number. However, they tend to be much more specialized in function and faster in operation as well. One of the control registers, called the instruction register (IR), will have a three-digit capacity to hold the instruction that is being examined and executed. Another register, called the instruction counter (IC), will have a two-digit capacity to hold the address of the next instruction to be brought from storage. When action on one instruction is completed, the control unit will bring the next instruction from the storage address given by the IC and place it in the IR. While this is going on, the number in the IC is increased by 1.

Figure A.5 illustrates this sequence of events for the second step in the problem to compute the sum of $a + b$. The three machine instructions are assumed to be stored at addresses 07, 08, and 09.

In our simple computer it will be possible to manually set a number in the IC while the computer is idle. When we throw a switch, the cyclical action will begin. Action commences by bringing to the IR the instruction found in the cell designated by the initial setting of the IC. At the same time the IC is increased by 1. Now the instruction in the IR is executed. In this way, if the IC is initially set to 07, the computer will automatically execute one instruction after another from sequentially addressed storage positions.

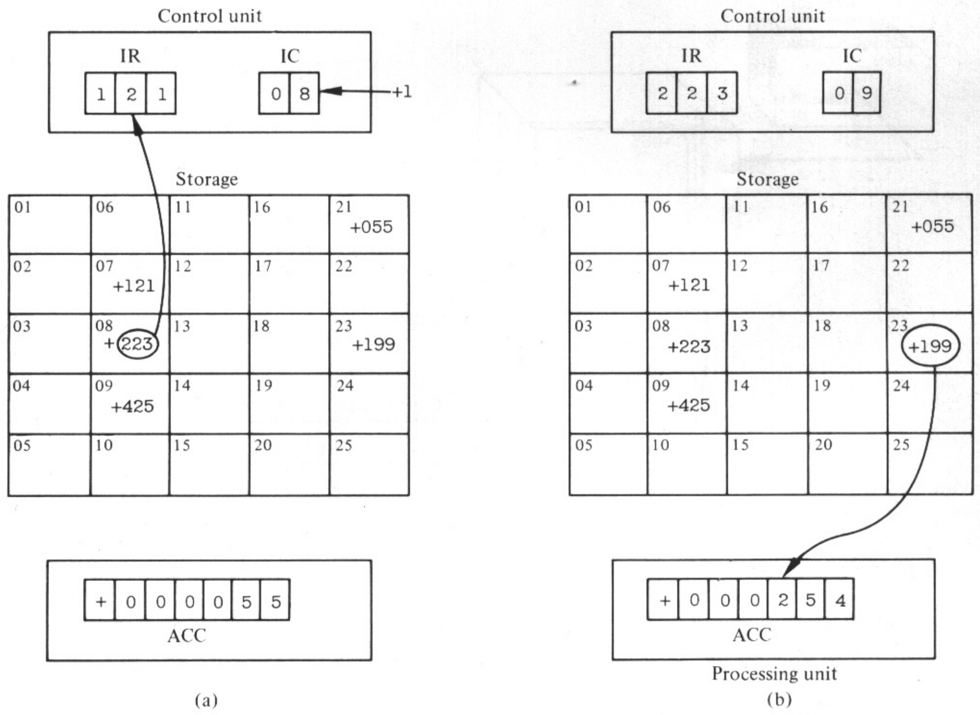

Fig. A.5 (a) The instruction in the IR has just been completed, leaving +000055 in the ACC. The instruction at 08 is being brought from storage to be placed in the IR (destructive read-in). While this is going on, the number in the IC will be incremented by 1. (b) The instruction from 08 is examined and executed, resulting in a new value, +000254, in the ACC.

Input and output units

The input and output devices are designed to function in very simple fashion. We shall imagine that the input unit is capable of reading numbers one at a time from the input device (which might be a punched-card reader). In executing the read instruction,

```
7 21
```

for example, the input unit reads a signed three-digit number, the first one received from the input device. The number so read is transmitted to storage and placed at location 21. Recall that the digit 7 was selected as the code for the read instruction. Figure A.6 schematically represents this input action. Here we can assume that the IC was manually set to 05 to begin this process.

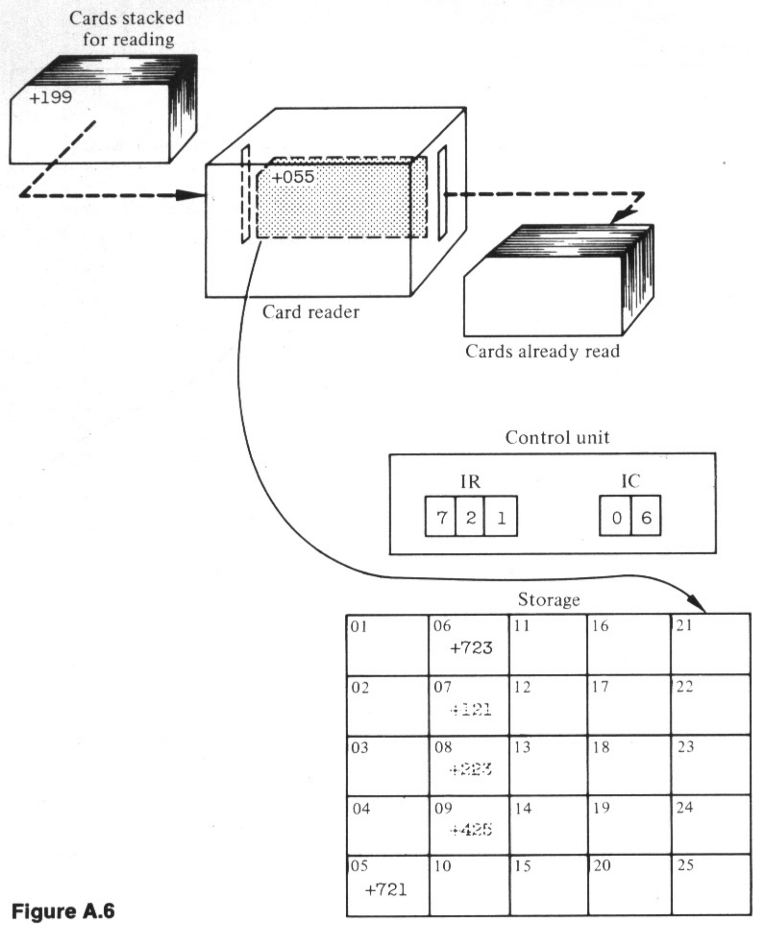

Figure A.6

The next instruction is taken from 06. It might be another input instruction. This time the next number is copied from the input device and placed in storage address 23. In this manner we obtain the values for *a* and *b* so that in the next steps the computer can add them.

If we want to have the computer print the result of the computation, we need an output instruction. Consulting Table A.1, we see that an appropriate instruction would be

```
8 25
```

meaning "display on the printer the contents of storage location 25." Supposing that such an instruction is placed at address 10, we see a full sequence of instructions which, if executed, would read data, compute, and print a result. The output step is illustrated in Fig. A.7.

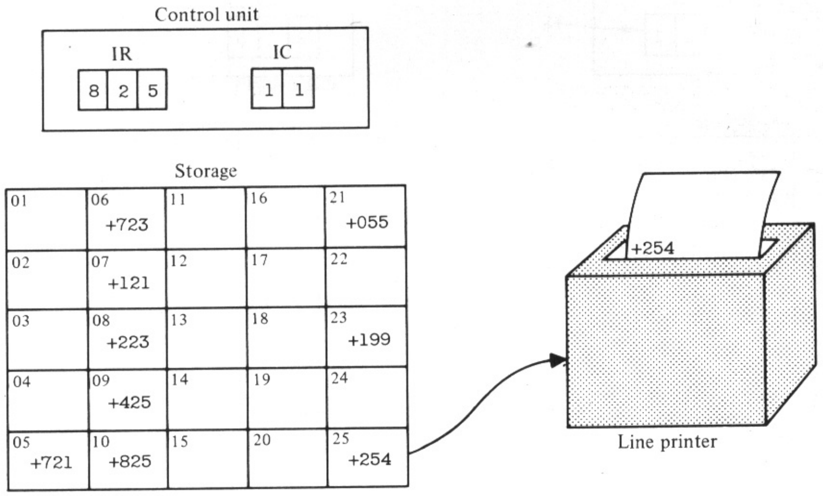

Figure A.7

A closed loop of instructions

A computer's power lies in its ability to repeatedly execute simple or complex processes at high speeds. To repeat the simple process we are considering here, perhaps on many pairs of values for *a* and *b*, we can imagine stacking many sets of data to be read. For each set of data we wish to execute the instructions stored in addresses 05 through 10, so that the computer can read the data, compute, and print the resulting sum. Such repetition can be accomplished by placing a *transfer*

instruction in address 11. The code digit we have chosen for this is 9 (see Table A.1). The instruction we want is then

```
9 05
```

which would mean "just replace the contents of the IC, now 12, with 05 (the address portion of the instruction in the IR)." No other action is taken. Transfer instructions effectively alter the sequence of instructions being executed. In this case the net effect is to form a "closed loop," repeating instructions located at 05 to 11, inclusive. Figure A.8 diagrams the action in the control unit when a transfer instruction is encountered. In Fig. A.8(a) the instruction from address 11 is brought to the IR, and the contents of the IC are increased from `11` to `12`. Execution of a transfer command involves only the control unit. In Fig. A.8(b) the contents of the IC are replaced by the rightmost two digits of the IR.

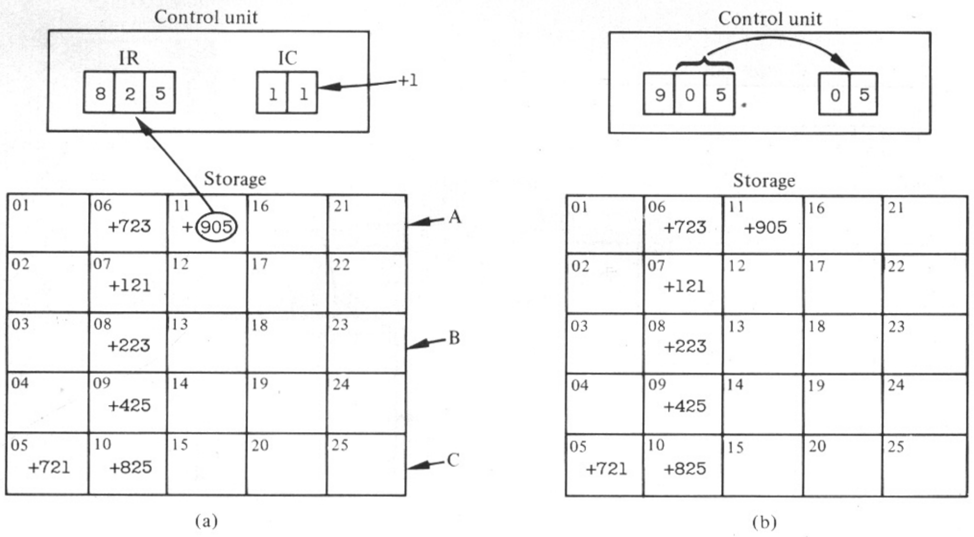

Figure A.8

Figure A.9 summarizes in flowchart representation the algorithm that we have just coded as a machine-language program.

Terminal read process

Under what circumstance will a program loop such as that in Fig. A.9 terminate? When the input stack of cards has been exhausted, the next read instruction cannot be carrried out. We will therefore design our computer so that it will automatically halt when it is asked to execute a read instruction and there are no more cards available to be read.

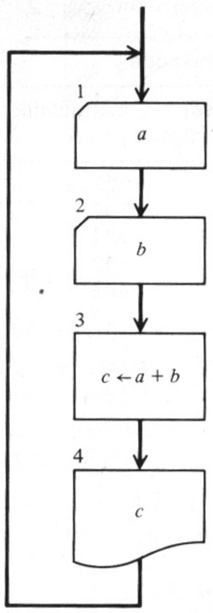

Figure A.9

Exercises

1. Although we have not yet described division in our miniature computer, let us assume that a quotient $p \div q$ (the integral part) may be obtained by the code sequence that corresponds to

```
CLA P
DIV Q
```

The resulting quotient will be found in the three lowest-order positions of the ACC.

Write a sequence of symbolic codes and corresponding machine codes that will compute the value

$$y = \frac{a - b}{a + b}.$$

Draw a picture of the storage cells showing all instructions and locations for original or intermediate results. You must make certain decisions, such as where to keep *a*, *b*, and *y* in storage and at what location to begin the program.

Hint: Before the division can be carried out, both the numerator and the denominator must be computed. At least one of these must be saved, in a place that you will have to designate, while the other is being computed.

2. After completing Exercise 1, write a sequence of instructions that reads a pair of values for *a* and *b* from punch cards, computes *y*, prints it, and returns to read another pair of values.

Table A.2 Code for computing $|a - b|$

Symbolic code		Machine code	
Address name	Instruction	Address†	Instruction
START	RDS A	09	701
	RDS B	10	702
	CLA A	11	101
	SUB B	12	302
	STO C	13	403
	TMI NEG	14	017
AGAIN	PRT C	15	803
	TRA START	16	909
NEG	CLA B	17	102
	SUB A	18	301
	STO C	19	403
	TRA AGAIN	20	915

† We are assuming the following: (1) Instructions begin at 09. (2) *A* at 01, *B* at 02, *C* at 03.

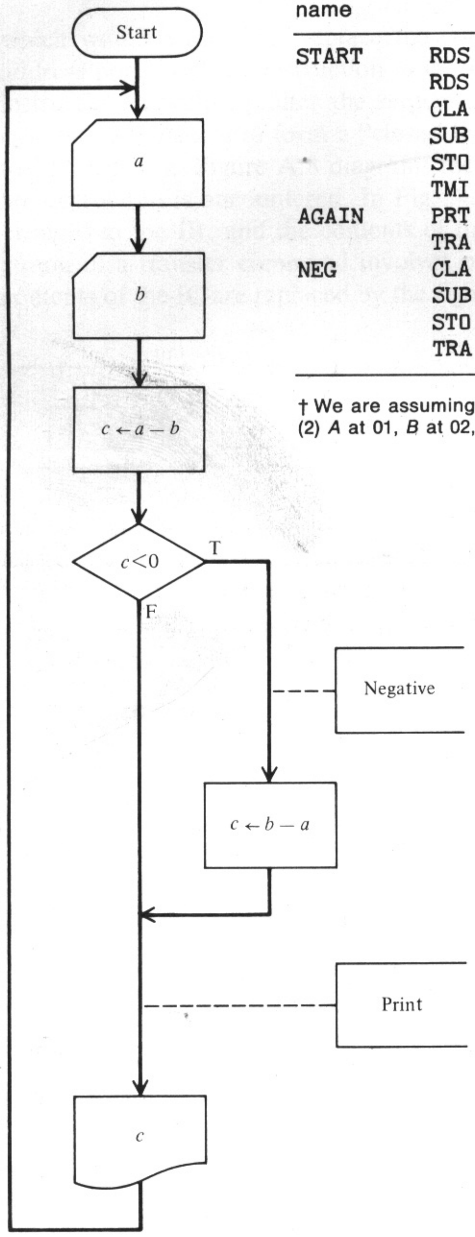

Fig. A.10 Flowchart for computing $|a - b|$.

3. Assume that a product $p \times q$ may be obtained by the code sequence that corresponds to

```
CLA P
MPY Q
```

Write a code sequence to compute the following.

a) $d = \frac{a \times b}{c}$ b) $c = \frac{a}{b \times c}$

Decisions

We have included a conditional transfer instruction in our miniature computer's order list. We have selected the transfer on the minus, or `TMI` instruction, with a code 0. Upon executing a `TMI`, the computer examines the sign of the ACC. If it is negative (−), the instruction is then treated as an *unconditional* transfer, or `TRA` instruction. That is, the last two digits of the instruction replace the current contents of the IC, thus breaking the normal sequence of instructions. If the sign of the ACC is positive (+), no further action is taken. The computer then goes on to get the next instruction in the normal sequence.

Let us see how we might employ the `TMI` instruction to compute and print the absolute value of $a - b$, or $|a - b|$, for many pairs of values, a, b. There would be several alternative ways to code this problem. One approach, not the best necessarily, is diagrammed in Fig. A.10 and coded both symbolically and in machine code in Table A.2. Here is one way you can follow this program step by step and gain an added grasp of the process that goes on inside the computer: (1) Draw the storage unit, arithmetic unit, and control unit, as we have done in earlier figures. (2) Enter the instructions in their proper "squares" of storage. (3) Place 09 in the IC. (4) "Turn the switch on" and begin computing with a pair of values for a and b. Try it.

A.7 REFERENCES ON COMPUTER SYSTEM ORGANIZATION

Bell, C. G., J. C. Mudge, and J. E. McNamara, *Computer Engineering: A DEC View of Hardware Systems Design*. Digital Equipment Corporation, Bedford, Mass., 1978.

Organick, E. I., and J. A. Hinds, *Interpreting Machines: Architecture and Programming of the B1700/B1800 Series*. Elsevier-North Holland, New York, 1977.

Tanenbaum, A. S., *Structured Computer Organization*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

Exercises:

1. Write a program that will read two values, a and b, and will print the quantity $\frac{a+b}{a-b}$, and then loop to read another pair of values.

2. Write a program that will read three values, a, b, and c, and compute $\frac{a}{b*c}$, and $\frac{a*b}{c}$.

3. Write a program that will read a number and print its absolute value.

In all three of the above programs, write the assembly code first, then write the machine code, then place the code into computer memory.