amount of time, we need a precise and often highly stylized definition of what constitutes an algorithm. Turing machines (Section 1.6) are an example of such a definition.

In describing and communicating algorithms we would like a notation more natural and easy to understand than a program for a random access machine, random access stored program machine, or Turing machine. For this reason we shall also introduce a high-level language called Pidgin ALGOL. This is the language we shall use throughout the book to describe algorithms. However, to understand the computational complexity of an algorithm described in Pidgin ALGOL we must relate Pidgin ALGOL to the more formal models. This we do in the last section of this chapter.

1.2 RANDOM ACCESS MACHINES

A random access machine (RAM) models a one-accumulator computer in which instructions are not permitted to modify themselves.

A RAM consists of a read-only input tape, a write-only output tape, a program, and a memory (Fig. 1.3). The input tape is a sequence of squares, each of which holds an integer (possibly negative). Whenever a symbol is read from the input tape, the tape head moves one square to the right. The output is a write-only tape ruled into squares which are initially all blank. When a write instruction is executed, an integer is printed in the square of the

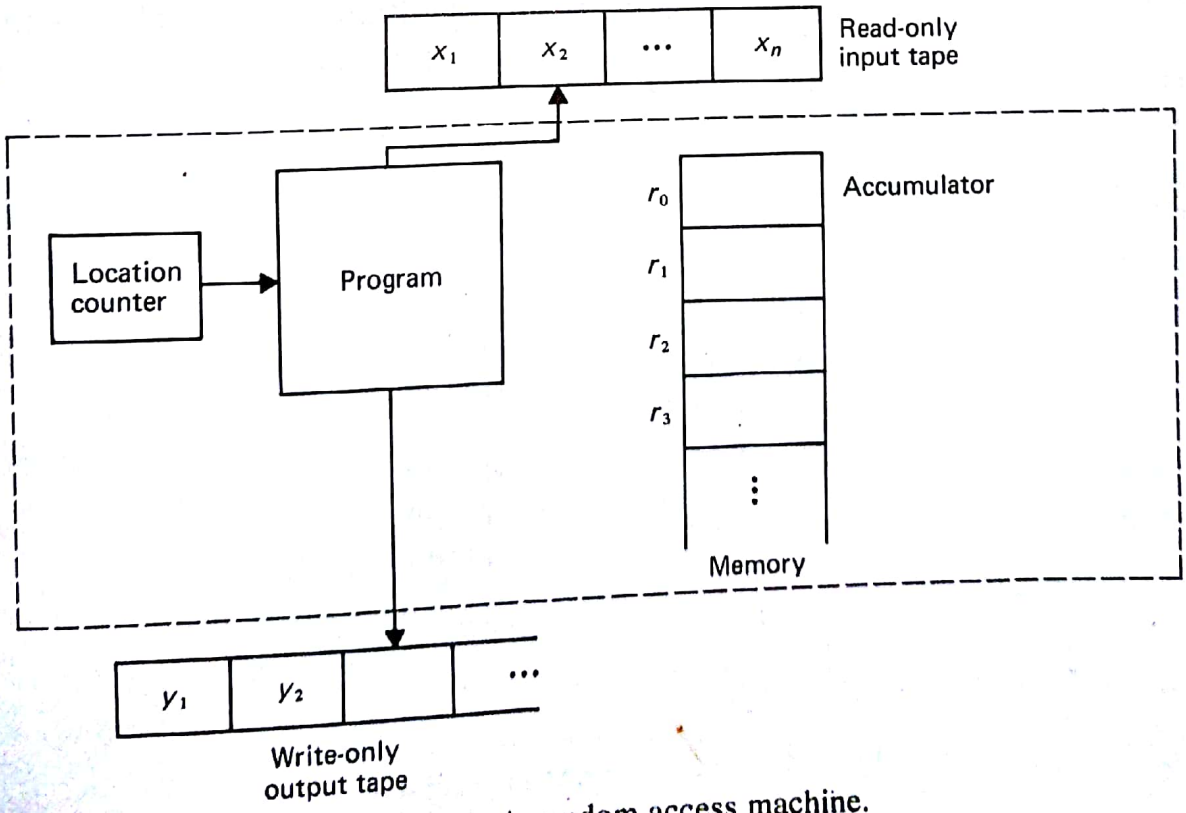

Fig. 1.3 A random access machine.

output tape that is currently under the output tape head, and the tape head is moved one square to the right. Once an output symbol has been written, it cannot be changed.

The memory consists of a sequence of registers, $r_0, r_1, \ldots, r_i, \ldots$, each of which is capable of holding an integer of arbitrary size. We place no upper bound on the number of registers that can be used. This abstraction is valid in cases where:

1. the size of the problem is small enough to fit in the main memory of a computer, and
2. the integers used in the computation are small enough to fit in one computer word.

The program for a RAM is not stored in the memory. Thus we are assuming that the program does not modify itself. The program is merely a sequence of (optionally) labeled instructions. The exact nature of the instructions used in the program is not too important, as long as the instructions resemble those usually found in real computers. We assume there are arithmetic instructions, input-output instructions, indirect addressing (for indexing arrays, e.g.) and branching instructions. All computation takes place in the first register r_0, called the *accumulator,* which like every other memory register can hold an arbitrary integer. A sample set of instructions for the RAM is shown in Fig. 1.4. Each instruction consists of two parts—an *operation code* and an *address.*

In principle, we could augment our set with any other instructions found in real computers, such as logical or character operations, without altering the order-of-magnitude complexity of problems. The reader may imagine the instruction set to be so augmented if it suits him.

Operation code	Address
1. LOAD	operand
2. STORE	operand
3. ADD	operand
4. SUB	operand
5. MULT	operand
6. DIV	operand
7. READ	operand
8. WRITE	operand
9. JUMP	label
10. JGTZ	label
11. JZERO	label
12. HALT	

Fig. 1.4. Table of RAM instructions.

An operand can be one of the following:

1. $=i$, indicating the integer i itself.
2. A nonnegative integer i, indicating the contents of register i.
3. $*i$, indicating indirect addressing. That is, the operand is the contents of register j, where j is the integer found in register i. If $j < 0$, then the machine halts.

These instructions should be quite familiar to anyone who has programmed in assembly language. We can define the meaning of a program P with the help of two quantities, a mapping c from nonnegative integers to integers and a "location counter" which determines the next instruction to execute. The function c is a *memory map;* $c(i)$ is the integer stored in register i (the *contents* of register i).

Initially, $c(i) = 0$ for all $i \geq 0$, the location counter is set to the first instruction in P, and the output tape is all blank. After execution of the kth instruction in P, the location counter is automatically set to $k + 1$ (i.e., the next instruction), unless the kth instruction is JUMP, HALT, JGTZ, or JZERO.

To specify the meaning of an instruction we define $v(a)$, *the value of operand* a, as follows:

$$\begin{aligned} v(=i) &= i, \\ v(i) &= c(i), \\ v(*i) &= c(c(i)). \end{aligned}$$

The table of Fig. 1.5 defines the meaning of each instruction in Fig. 1.4. Instructions not defined, such as STORE $=i$, may be considered equivalent to HALT. Likewise, division by zero halts the machine.

During the execution of each of the first eight instructions the location counter is incremented by one. Thus instructions in the program are executed in sequential order until a JUMP or HALT instruction is encountered, a JGTZ instruction is encountered with the contents of the accumulator greater than zero, or a JZERO instruction is encountered with the contents of the accumulator equal to zero.

In general, a RAM program defines a mapping from input tapes to output tapes. Since the program may not halt on all input tapes, the mapping is a partial mapping (that is, the mapping may be undefined for certain inputs). The mapping can be interpreted in a variety of ways. Two important interpretations are as a function or as a language.

Suppose a program P always reads n integers from the input tape and writes at most one integer on the output tape. If, when $x_1, x_2, \ldots, x_n$ are the integers in the first n squares of the input tape, P writes y on the first square of the output tape and subsequently halts, then we say that P computes the function $f(x_1, x_2, \ldots, x_n) = y$. It is easily shown that a RAM, like any other

Instruction	Meaning
1. LOAD a	$c(0) \leftarrow v(a)$
2. STORE i	$c(i) \leftarrow c(0)$
STORE $*i$	$c(c(i)) \leftarrow c(0)$
3. ADD a	$c(0) \leftarrow c(0) + v(a)$
4. SUB a	$c(0) \leftarrow c(0) - v(a)$
5. MULT a	$c(0) \leftarrow c(0) \times v(a)$
6. DIV a	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$†
7. READ i	$c(i) \leftarrow$ current input symbol.
READ $*i$	$c(c(i)) \leftarrow$ current input symbol. The input tape head moves one square right in either case.
8. WRITE a	$v(a)$ is printed on the square of the output tape currently under the output tape head. Then the tape head is moved one square right.
9. JUMP b	The location counter is set to the instruction labeled b.
10. JGTZ b	The location counter is set to the instruction labeled b if $c(0) > 0$; otherwise, the location counter is set to the next instruction.
11. JZERO b	The location counter is set to the instruction labeled b if $c(0) = 0$; otherwise, the location counter is set to the next instruction.
12. HALT	Execution ceases.

† Throughout this book, $\lceil x \rceil$ (*ceiling* of x) denotes the least integer equal to or greater than x, and $\lfloor x \rfloor$ (*floor*, or *integer part* of x) denotes the greatest integer equal to or less than x.

Fig. 1.5. Meaning of RAM instructions. The operand a is either $=i$, i, or $*i$.

reasonable model of a computer, can compute exactly the *partial recursive functions*. That is, given any partial recursive function f we can define a RAM program that computes f, and given any RAM program we can define an equivalent partial recursive function. (See Davis [1958] or Rogers [1967] for a discussion of recursive functions.)

Another way to interpret a RAM program is as an acceptor of a language. An *alphabet* is a finite set of symbols, and a *language* is a set of strings over some alphabet. The symbols of an alphabet can be represented by the integers $1, 2, \ldots, k$ for some k. A RAM can accept a language in the following manner. We place an input string $s = a_1 a_2 \cdots a_n$ on the input tape, placing the symbol a_1 in the first square, the symbol a_2 in the second square, and so on. We place 0, a symbol we shall use as an endmarker, in the $(n + 1)$st square to mark the end of the input string.