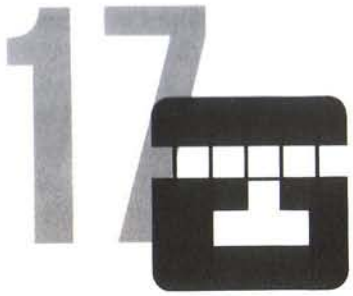
Is it possible for two nearly optimal tours to have an offspring which is far
n optimal?

ferences

rence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley & Sons, New York, 1966.

n H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Mich., 1975.



THE RANDOM ACCESS MACHINE

An Abstract Computer

The term *random access machine (RAM)* tends to have a double use among computer scientists. Sometimes it refers to specific computers with *random access memories,* that is, memories in which the access to one address is just as fast and easy as access to any other address. At other times, it refers to a certain model of computation which is no less "abstract" than, say, a Turing machine, but which is closer in its operation to standard, programmable digital computers.

Not surprisingly, the main difference between Turing machines and (abstract) RAMs lies in the kind of memory employed. On one hand, a Turing machine's memory is all on tape, making it a sort of "serial access" machine. A RAM, on the other hand, has its memory organized into words, with each word having an address. It has, moreover, a number of registers. The model described here has one register called the *accumulator,* (AC for short).

As was the case with Turing machines, RAMs are actually defined in terms of programs, the sort of structure shown in Figure 17.1 being merely a handy vehicle for interpreting such programs. As such, we picture a RAM as being equipped with a control unit able to carry out all the operations specified by a RAM program with which it is "loaded" in some sense. The program dictates the transfer of information between various memory words and the AC. It also specifies certain operations upon the contents of the AC. Finally, it is able to direct which of its own instructions are to be carried out next.

Both the AC and each memory word are assumed capable of holding a single integer, no matter how large. Although the RAM has only one register, it has an infinite number of words in its memory. In the figure on the next page, there is a communication line between the RAM's control unit and each of its memory words. This symbolizes the fact that as soon as an address is specified by the RAM program, the corresponding word of memory is instantly accessible by the control unit.

Although a great variety of RAMs have been defined by various authors, we are content with a fairly simple sort which is programmable in the language shown in the table below. We shall call it the *random access language* (RAL):

Mnemonic	Argument	Meaning
LDA	*X*	Load the AC with the contents of memory address *X*.
LDI	*X*	Load the AC indirectly with the contents of address *X*.
STA	*X*	Store the contents of the AC at memory address *X*.
STI	*X*	Store the contents of the AC indirectly at address *X*.
ADD	*X*	Add the contents of address *X* to the contents of the AC.
SUB	*X*	Subtract the contents of address *X* from the AC.
JMP	*X*	Jump to the instruction labeled *X*.
JMZ	*X*	Jump to the instruction labeled *X* if the AC contains 0.
HLT		Halt.

When we speak of the contents of memory address *X*, we refer, of course, to the integer currently stored in the memory word whose address is *X*. To load the AC *indirectly* with the contents of memory address *X* means not to load the contents of address *X*, but to treat those contents as yet another address (whose

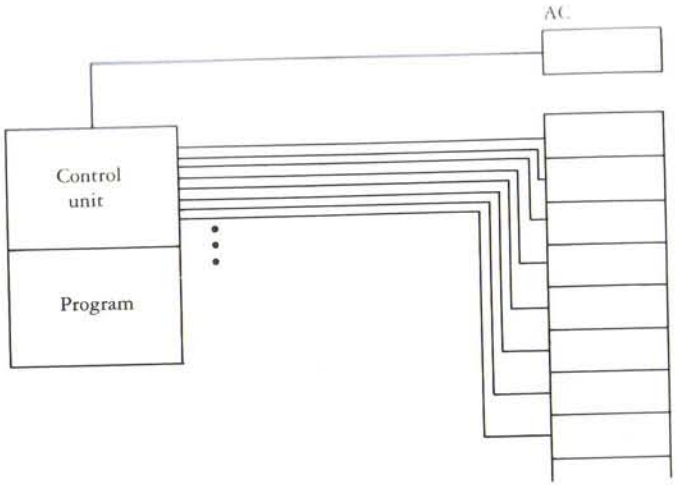


Figure 17.1 A random access machine

contents are to be loaded). The instruction STI uses the same kind of indirection, but in reverse.

When a RAL program is written out (see the example below), we may number its statements so that the jump commands (JMP and JMZ) are clearly understood. Thus, JMP 5 means that the next instruction to be executed is instruction 5. But JMZ 5 means to execute instruction 5 next if the AC contains 0, otherwise, to execute the next instruction in the program.

Indirection is a very useful feature of RAM programs. Both LDI and STI use indirection in the following manner: "LDI *X*" means first to look up the contents of *X*. If the integer stored there is *Y*, then the RAM is next to look up the contents of *Y* and, finally, to load that integer in the AC. This sort of indirect memory reference also occurs in the STI instruction type: Look up the contents of *X*, get the integer stored there, say *Y*, and then store the contents of the AC at *Y*.

As in the case of Turing machines, no attention is paid to input or output in RAMs; since they are not "real" machines, there is no point in trying to communicate with a user. Instead, a certain finite initial set of integers is assumed to exist already in certain memory words, with all the rest containing 0s. At the end of a RAM's computation what remains in memory is considered to be its output.

A RAM halts under one of two conditions. If execution comes to a HLT command, all operations cease and the current computation comes to an end. If the RAM comes to a nonexecutable instruction, execution also ceases. A nonexecutable instruction is one whose arguments make no sense in the context of the RAM pictured above. For example, no memory location has a negative

address, so STA −8 makes no sense. Similarly, JMP 24 makes no sense in a 20-line program. Such nonexecutable instructions are assumed not to exist in RAL programs.

In Chapter 15, we specified an algorithm for detecting duplicates in an input sequence A of positive integers. The integers happen to be stored in array B, each at an index equal to the integer stored.

STOR

for $i \leftarrow 1$ **to** n **do**
  if $B(A(i)) \neq 0$
    then *output* $A(i)$;
      exit
    else $B(A(i)) \leftarrow 1$

This algorithm can be translated to a RAL program as follows:

STOR · RAL

```
 1. LDI  3    /get ith entry from A
 2. ADD  4    /add offset to compute index j
 3. STA  5    /store index j
 4. LDI  5    /get jth entry from B
 5. JMZ  9    /if entry 0, go to 9.
 6. LDA  3    /if entry 1, get index i
 7. STA  2    /and store it at 2.
 8. HLT       /stop execution
 9. LDA  1    /get constant 1
10. STI  5    /and store it in B
11. LDA  3    /get index i
12. SUB  4    /subtract limit
13. JMZ  8    /if i = limit, stop
14. LDA  3    /get index i (again)
15. ADD  1    /increment i
16. STA  3    /store new value of i
17. JMP  1    /go to first instruction
```

The STOR program is more readily understood by referring to Figure 17.2 in which the program's use of both the AC and memory is clearly laid out.

The first five words of memory are devoted to variables and constants used by the program. In this particular case, the next four words contain a set of four integers to be tested for duplicates. These words are referred to collectively as

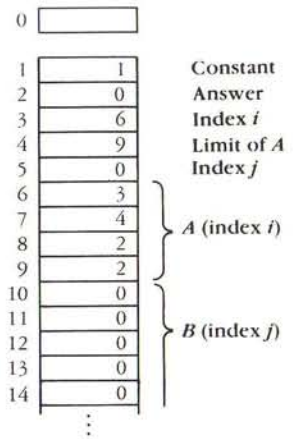


Figure 17.2 Memory layout for the STOR program

A, effectively an array. All the remaining words of memory, from 10 onward, comprise an infinite array B in which STOR will place a 1 each time it processes a new integer in A.

The first memory location contains the constant 1 which is used by STOR to increment the index i. This index, or rather its current value, is stored in location 3. A second index, j, is stored in location 5. It serves to access locations in B.

The first three instructions of STOR are

1. LDI 3
2. ADD 4
3. STA 5

These instructions look up the contents of the location we call i, add the contents of location 4, and then store the result in location 5. Specifically, the LDI instruction looks up the contents of location 3. Initially, the number stored here is 6, the first location in the array A. The number stored in location 6 is a 3 and *this* number is placed in the AC by the LDI instruction. Location 4 contains the constant 9 which delimits the A section of memory. The effect of adding 9 to 3 in the AC is to calculate the third location of B, namely 12. The computer then stores this number in location 5. Thus STOR retrieved 3 as the first integer of A and computed the location of the third member of B.

The next instruction of STOR, namely LDI 5, loads the number stored in the third location of B into the AC. This number happens to be 0 so at instruction 5, execution jumps down to instruction 9. Instructions 9 and 10 retrieve the con-

stant 1 and store it in the third location of B; in reality, this happens to be 12 and that is where STOR places a 1. This indicates that STOR has just encountered a 3. If (given another A sequence) STOR should encounter a 3 later, it would have a 1 in the AC when it reached the JMZ 9 instruction. In such a case, it would immediately load the current value of i in location 3, store it in location 2 and then halt.

Location 2 contains the answer when STOR terminates. If 0, it means that no duplicates were found among the integers of A. Otherwise it will contain the location in A of a repeated integer.

The rest of the program, instructions 11 to 16, retrieves the index i from location 3 and then subtracts 9 (in location 4) to decide whether STOR has reached the end of A. If not at the end of A it increments the index i and stores it again in location 3 before jumping back to the head of the program in order to test the next integer in the A sequence.

At first glance, it would appear that RAMs are more powerful than Turing machines in terms of the functions which they compute, but this is not really true. In Chapter 66, it will be shown that Turing machines can do anything that RAMs can do. In the meantime, it would be useful to check the converse. It may seem obvious, but can RAMs *really* do anything that Turing machines can do? Luckily, it is not hard to prove. Given an arbitrary Turing machine with a semi-infinite tape (see Chapter 31), use the RAM memory to simulate the tape as shown in Figure 17.3.

Given this one-to-one correspondence between words of RAM memory and squares of the Turing machines tape, it is now only a matter of writing a RAM program to mimic the Turing machine's program. Each quintuple of the latter is

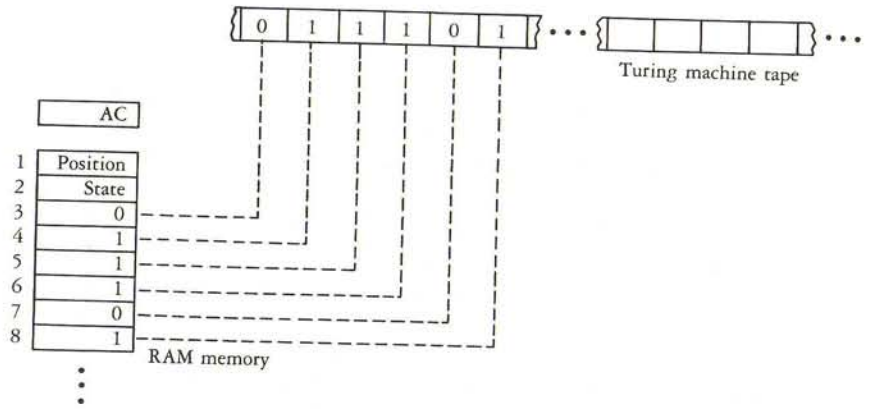


Figure 17.3 A RAM simulating a Turing machine

replaced by a number of RAL instructions which have the same effect on the RAMs memory as the quintuple would on the Turing machine tape. To this end, it is necessary for the RAM program to remember the position of the Turing machine's read/write head as well as its current state.

Because of their equivalence with Turing machines and, indeed, because of their equivalence with all the most powerful known abstract computational schemes, RAMs offer an alternate vehicle in the study of feasible computations. Some questions are more easily asked and answered in the framework of such a model of computation; of all the schemes, it is most like a standard, digital computer. It is also one of the most convenient schemes for expressing actual computations such as the RAL program we have just examined.

Problems

1. Alter the RAL program STOR so that when a computation is finished and the input sequence contained a duplicate integer, we know what integer that was.

2. Complete the proof that a RAM can carry out any Turing machine computation.

3. Our RAM language RAL contains no instruction for multiplication or division. Write a RAL program called MPY which takes two integers X and Y as input in addresses 1 and 2, respectively, and outputs their product in address 4.

References

D. E. Knuth. *The Art of Computer Programming,* vol. 1: *Fundamental Algorithms.* Addison-Wesley, Reading, Mass, 1969.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass. 1974.